HANDOUTS | MATERIAL
HIGHLIGHTED AND PROVIDED BY :

VU TALK

CS301 Final Term Preparation :

https://www.youtube.com/playlist?list=PLgRL5atsKj4p4WaPIiVuPkBUoVF2FNkol

CLICK TO JOIN OUR WHATSAPP GROUP : click me

VU TALK YouTube CHANNEL : click me

For Any Help Contact Me On Whatsapp : click me

Follow VU TALK on Whatsapp Channel : click me

Data Structures

Table of Contents

Data Structures 1
Lecture No. 01 3
Lecture No. 02 12
Lecture No. 03 20
Lecture No. 04 32
Lecture No. 05 46
Lecture No. 06 56
Lecture No. 07 63
Lecture No. 08 71
Lecture No. 09 82
Lecture No. 10 93
Lecture No. 11 105
Lecture No. 12 123
Lecture No. 13 135
Lecture No. 14 146
Lecture No. 15 158
Lecture No. 16 170
Lecture No. 17 189
Lecture No. 18 203
Lecture No. 19 211
Lecture No. 20 220
Lecture No. 21 231
Lecture No. 22 240
Lecture No. 23 257
Lecture No. 24 273
Lecture No. 25 284
Lecture No. 26 296
Lecture No. 27 307
Lecture No. 28 321
Lecture No. 29 333
Lecture No. 30 348
Lecture No. 31 360
Lecture No. 32 370
Lecture No. 33 379
Lecture No. 34 386
Lecture No. 35 393
Lecture No. 36 404
Lecture No. 37 416
Lecture No. 38 424
Lecture No. 39 430
Lecture No. 40 441
Lecture No. 41 449
Lecture No. 42 459
Lecture No. 43 468
Lecture No. 44 474
Lecture No. 45 485

Data Structures

Lecture No. 01

Reading Material

Data Structures and algorithm analysis in C++ Chapter. 3

3.1, 3.2, 3.2.1

Summary

- Introduction to Data Structures
- Selecting a Data Structure
- Data Structure Philosophy
- Goals of this Course
- Array
- List data structure

Welcome to the course of data structure. This is very important subject as the topics covered in it will be encountered by you again and again in the future courses. Due to its great applicability, this is usually called as the foundation course. You have already studied Introduction to programming using C and C++ and used some data structures. The focus of that course was on how to carry out programming with the use of C and C++ languages besides the resolution of different problems. In this course, we will continue problem solving and see that the organization of data in some cases is of immense importance. Therefore, the data will be stored in a special way so that the required result should be calculated as fast as possible.

Following are the goals of this course:

- Prepare the students for (and is a pre-requisite for) the more advanced material students will encounter in later courses.
- Cover well-known data structures such as dynamic arrays, linked lists, stacks, queues, trees and graphs.
- Implement data structures in C++

You have already studied the dynamic arrays in the previous course. We will now discuss linked lists, stacks, queues, trees and graphs and try to resolve the problems with the help of these data structures. These structures will be implemented in C++ language. We will also do programming assignments to see the usage and importance of these structures.

Information about Data Structure subject is available at: “http://www.vu.edu.pk/ds”.

Introduction to Data Structures

Let’s discuss why we need data structures and what sort of problems can be solved with their use. Data structures help us to organize the data in the computer, resulting in more efficient programs. An efficient program executes faster and helps minimize the usage of resources like memory, disk. Computers are getting more powerful with the passage of time with the increase in CPU speed in GHz, availability of faster network and the maximization of disk space. Therefore people have started solving more and more complex problems. As computer applications are becoming complex, so there is need for more resources. This does not mean that we should buy a new computer to make the application execute faster. Our effort should be to ensue that the solution is achieved with the help of programming, data structures and algorithm.

What does organizing the data mean? It means that the data should be arranged in a way that it is easily accessible. The data is inside the computer and we want to see it. We may also perform some calculations on it. Suppose the data contains some numbers and the programmer wants to calculate the average, standard deviation etc. May be we have a list of names and want to search a particular name in it. To solve such problems, data structures and algorithm are used. Sometimes you may realize that the application is too slow and taking more time. There are chances that it may be due to the data structure used, not due to the CPU speed and memory. We will see such examples. In the assignments, you will also check whether the data structure in the program is beneficial or not. You may have two data structures and try to decide which one is more suitable for the resolution of the problem.

As discussed earlier, a solution is said to be efficient if it solves the problem within its resource constraints. What does it mean? In the computer, we have hard disk, memory and other hardware. Secondly we have time. Suppose you have some program that solves the problem but takes two months. It will be of no use. Usually, you don’t have this much time and cannot wait for two months. Suppose the data is too huge to be stored in disk. Here we have also the problem of resources. This means that we have to write programs considering the resources to achieve some solution as soon as possible. There is always cost associated with these resources. We may need a faster and better CPU which can be purchased. Sometimes, we may need to buy memory. As long as data structures and programs are concerned, you have to invest your own time for this. While working in a company, you will be paid for this. All these requirements including computer, your time and computer time will decide that the solution you have provided is suitable or not. If its advantages are not obtained, then either program or computer is not good.

So the purchase of a faster computer, while studying this course, does not necessarily help us in the resolution of the problem. In the course of “Computer Architecture” you will see how the more efficient solutions can be prepared with the hardware. In this course, we will use the software i.e. data structures, algorithms and the recipes through which the computer problems may be resolved with a faster solution.

Selecting a Data Structure

How can we select the data structure needed to solve a problem? You have already studied where to use array and the size of array and when and where to use the pointers etc. First of all, we have to analyze the problem to determine the resource constraints that a solution must meet. Suppose, the data is so huge i.e. in Gega bytes (GBs) while the disc space available with us is just 200 Mega bytes. This problem can not be solved with programming. Rather, we will have to buy a new disk.

Secondly, it is necessary to determine the basic operations that must be supported. Quantify the resource constraints for each operation. What does it mean? Suppose you have to insert the data in the computer or database and have to search some data item. Let's take the example of telephone directory. Suppose there are eight million names in the directory. Now someone asks you about the name of some particular person. You want that this query should be answered as soon as possible. You may add or delete some data. It will be advisable to consider all these operations when you select some data structure.

Finally select the data structure that meets these requirements the maximum. Without, sufficient experience, it will be difficult to determine which one is the best data structure. We can get the help from internet, books or from someone whom you know for already getting the problems solved. We may find a similar example and try to use it. After this course, you will be familiar with the data structures and algorithms that are used to solve the computer problems.

Now you have selected the data structure. Suppose a programmer has inserted some data and wants to insert more data. This data will be inserted in the beginning of the existing data, or in the middle or in the end of the data. Let's talk about the arrays and suppose you have an array of size hundred. Data may be lying in the first fifty locations of this array. Now you have to insert data in the start of this array. What will you do? You have to move the existing data (fifty locations) to the right so that we get space to insert new data. Other way round, there is no space in the start. Suppose you have to insert the data at 25^{th} location. For this purpose, it is better to move the data from 26^{th} to 50^{th} locations; otherwise we will not have space to insert this new data at 25^{th} location.

Now we have to see whether the data can be deleted or not. Suppose you are asked to delete the data at 27^{th} position. How can we do that? What will we do with the space created at 27^{th} position?

Thirdly, is all the data processed in some well-defined order or random access allowed? Again take the example of arrays. We can get the data from 0^{th} position and traverse the array till its 50^{th} position. Suppose we want to get the data, at first from 50^{th} location and then from 13^{th}. It means that there is no order or sequence. We want to access the data randomly. Random access means that we can't say what will be the next position to get the data or insert the data.

Data Structure Philosophy

Let's talk about the philosophy of data structure. Each data structure has costs and benefits. Any data structure used in your program will have some benefits. For this,

you have to pay price. That can be computer resources or the time. Also keep in mind that you are solving this problem for some client. If the program is not efficient, the client will not buy it.

In rare cases, a data structure may be better than another one in all situations. It means that you may think that the array is good enough for all the problems. Yet this is not necessary. In different situations, different data structures will be suitable. Sometimes you will realize that two different data structures are suitable for the problem. In such a case, you have to choose the one that is more appropriate. An important skill this course is going to lend to the students is use the data structure according to the situation. You will learn the programming in a way that it will be possible to replace the one data structure with the other one if it does not prove suitable. We will replace the data structure so that the rest of the program is not affected. You will also have to attain this skill as a good programmer.

There are three basic things associated with data structures. A data structure requires:

- space for each data item it stores
- time to perform each basic operation
- programming effort

Goals of this Course

Reinforce the concept that costs and benefits exist for every data structure. We will learn this with practice.

Learn the commonly used data structures. These form a programmer's basic data structure “toolkit”. In the previous course, you have learned how to form a loop, functions, use of arrays, classes and how to write programs for different problems. In this course, you will make use of data structures and have a feeling that there is bag full of different data structures. In case of some problem, you will get a data structure from the toolkit and use some suitable data structure.

Understand how to measure the cost of a data structure or program. These techniques also allow you to judge the merits of new data structures that you or others might develop. At times, you may have two suitable data structures for some problem. These can be tried one by one to adjudge which one is better one. How can you decide which data structure is better than other. Firstly, a programmer can do it by writing two programs using different data structure while solving the same problem. Now execute both data structures. One gives the result before the other. The data structure that gives results first is better than the other one. But sometimes, the data grows too large in the problem. Suppose we want to solve some problem having names and the data of names grows to10 lakhs (one million). Now when you run both programs, the second program runs faster. What does it mean? Is the data structure used in program one not correct? This is not true. The size of the data, being manipulated in the program can grow or shrink. You will also see that some data structures are good for small data while the others may suit to huge data. But the problem is how can we determine that the data in future will increase or decrease. We should have some way to take decision in this regard. In this course we will do some mathematical analysis and see which data structure is better one.

Arrays

You have already studied about arrays and are well-versed with the techniques to utilize these data structures. Here we will discuss how arrays can be used to solve computer problems. Consider the following program:

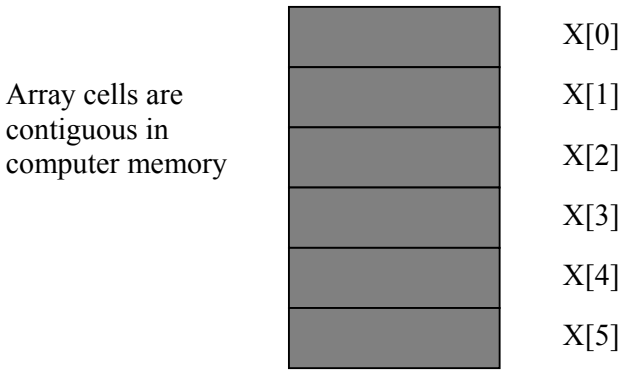
```
main( int argc, char** argv )
{
        int x[6];
        int j;
        for(j = 0; j < 6; j++)
                x[j] = 2 * j;
}
```

We have declared an *int* array of six elements and initialized it in the loop.

Let's revise some of the array concepts. The declaration of array is as *int x[6];* or *float x[6];* or *double x[6];* You have already done these in your programming assignments. An array is collection of cells of the same type. In the above program, we have array *x* of type *int* of six elements. We can only store integers in this array. We cannot put *int* in first location, *float* in second location and *double* in third location. What is *x*? *x* is a name of collection of items. Its individual items are numbered from zero to one less than array size. To access a cell, use the array name and an index as under:

x[0], x[1], x[2], x[3], x[4], x[5]

To manipulate the first element, we will use the index zero as *x[0]* and so on. The arrays look like in the memory as follows:



Array occupies contiguous memory area in the computer. In case of the above example, if some location is assigned to *x[0]*, the next location can not contain data other than *x[1]*. The computer memory can be thought of as an array. It is a very big array. Suppose a computer has memory of 2MB, you can think it as an array of size 2 million and the size of each item is 32 bits. You will study in detail about it in the computer organization, and Assembly language courses. In this array, we will put our programs, data and other things.

In the above program, we have declared an array named *x*. '*x*' is an array's name but there is no variable *x*. '*x*' is not an *lvalue*. If some variable can be written on the left-hand side of an assignment statement, this is *lvalue* variable. It means it has some memory associated with it and some value can be assigned to it. For example, if we have the code *int a, b;* it can be written as *b = 2;* it means that put 2 in the memory location named *b.* We can also write as *a = b;* it means whatever *b* has assign it to *a*, that is a copy operation. If we write as *a = 5;* it means put the number 5 in the memory location which is named as *a*. But we cannot write *2 = a;* that is to put at number 2 what ever the value of a is. Why can't we do that? Number 2 is a constant. If we allow assignment to constants what will happen? Suppose '*a*' has the value number 3. Now we assigned number 2 the number 3 i.e. all the number 2 will become number 3 and the result of *2 + 2* will become 6. Therefore it is not allowed.

'*x*' is a name of array and not an *lvalue*. So it cannot be used on the left hand side in an assignment statement. Consider the following statements

```
int x[6];
int n;
x[0] = 5;   x[1] = 2;
x = 3;                          //not allowed
x = a + b;            // not allowed
x = &n;                         // not allowed
```

In the above code snippet, we have declared an array *x* of *int*. Now we can assign values to the elements of *x* as *x[0] = 5* or *x[1] = 2* and so on. The last three statements are not allowed. What does the statement *x = 3;* mean? As *x* is a name of array and this statement is not clear, what we are trying to do here? Are we trying to assign 3 to each element of the array? This statement is not clear. Resultantly, it can not be allowed. The statement *x = a + b* is also not allowed. There is nothing wrong with *a + b*. But we cannot assign the sum of values of *a* and *b* to *x*. In the statement *x = &n*, we are trying to assign the memory address of *n* to *x* which is not allowed. The reason is the name *x* is not *lvalue* and we cannot assign any value to it. For understanding purposes, consider *x* as a constant. Its name or memory location can not be changed. This is a collective name for six locations. We can access these locations as *x[0], x[1]* up to *x[5]*. This is the way arrays are manipulated.

Sometimes, you would like to use an array data structure but may lack the information about the size of the array at compile time. Take the example of telephone directory. You have to store one lakh (100,000) names in an array. But you never know that the number of entries may get double or decline in future. Similarly, you can not say that the total population of the country is one crore (10 million) and declare an array of one crore names. You can use one lakh locations now and remaining will be used as the need arrives. But this is not a good way of using the computer resources. You have declared a very big array while using a very small chunk of it. Thus the remaining space goes waste which can, otherwise, be used by some other programs. We will see what can be the possible solution of this problem?

Suppose you need an integer array of size *n* after the execution of the program. We have studied that if it is known at the execution of the program that an array of size 20 or 30 is needed, it is allocated dynamically. The programming statement is as follows:

```
int*  y = new int[20];
```

It means we are requesting computer to find twenty memory locations. On finding it, the computer will give the address of first location to the programmer which will be stored in *y*. Arrays locations are contiguous i.e. these are adjacent. These twenty locations will be contiguous, meaning that they will be neighbors to each other. Now *y* has become an array and we can say *y[0] =1 or y[5] = 15*. Here *y* is an *lvalue*. Being a pointer, it is a variable where we can store the address of some variable. When we said *int* y = new int[20];* the new returns the memory address of first of the twenty locations and we store that address into *y*. As *y* is a pointer variable, so it can be used on the left-hand side. We can write it as:

```
y = &x[0];
```

In the above statement, we get the address of the fist location of the array *x* and store it in *y*. As *y* is *lvalue,* so it can be used on left hand side. This means that the above statement is correct.

```
y = x;
```

Similarly, the statement *y = x* is also correct. *x* is an array of six elements that holds the address of the first element. But we cannot change this address. However we can get that address and store it in some other variable. As *y* is a pointer variable and *lvalue* so the above operation is legal. We have dynamically allocated the memory for the array. This memory, after the use, can be released so that other programs can use it. We can use the *delete* keyword to release the memory. The syntax is:

```
delete[ ] y;
```

We are releasing the memory, making it available for use by other programs. We will not do it in case of *x* array, as 'new' was not used for its creation. So it is not our responsibility to delete *x*.

List data structure

This is a new data structure for you. The *List* data structure is among the most generic of data structures. In daily life, we use shopping list, groceries list, list of people to invite to a dinner, list of presents to give etc. In this course, we will see how we use lists in programming.

A list is the collection of items of the same type (grocery items, integers, names). The data in arrays are also of same type. When we say int *x[6];* it means that only the integers can be stored in it. The same is true for list. The data which we store in list should be of same nature. The items, or elements of the list, are stored in some particular order. What does this mean? Suppose in the list, you have the fruit first which are also in some order. You may have names in some alphabetical order i.e. the names which starts with *A* should come first followed by the name starting with *B* and so on. The order will be reserved when you enter data in the list.

It is possible to insert new elements at various positions in the list and remove any

element of the list. You have done the same thing while dealing with arrays. You enter the data in the array, delete data from the array. Sometimes the array size grows and at times, it is reduced. We will do this with the lists too.

List is a set of elements in a linear order. Suppose we have four names *a1, a2, a3, a4* and their order is as *(a3, a1, a2, a4)* i.e. *a3*, is the first element, *a1* is the second element, and so on. We want to maintain that order in the list when data is stored in the list. We don't want to disturb this order. The order is important here; this is not just a random collection of elements but an *ordered* one. Sometimes, this order is due to sorting i.e. the things that start with *A* come first. At occasions, the order may be due to the importance of the data items. We will discuss this in detail while dealing with the examples.

Now we will see what kind of operations a programmer performs with a list data structure. Following long list of operations may help you understand the things in a comprehensive manner.

Operation Name	**Description**
createList()	Create a new list (presumably empty)
copy()	Set one list to be a copy of another
clear();	Clear a list (remove all elements)
insert(X, ?)	Insert element X at a particular position in the list
remove(?)	Remove element at some position in the list
get(?)	Get element at a given position
update(X, ?)	Replace the element at a given position with X
find(X)	Determine if the element X is in the list
length()	Returns the length of the list.

createList() is a function which creates a new list. For example to create an array, we use *int x[6] or int* y = new int[20];* we need similar functionality in lists too. The *copy()* function will create a copy of a list. The function *clear()* will remove all the elements from a list. We want to insert a new element in the list, we also have to tell where to put it in the list. For this purpose *insert(X, position)* function is used. Similarly the function *remove(position)* will remove the element at position. To get an element from the list *get(position)* function is used which will return the element at position. To replace an element in the list at some position the function *update(X, position)* is used. The function *find(X)* will search *X* in the list. The function *length()* tells us about the number of elements in the list.

We need to know what is meant by "particular position" we have used "?" for this in the above table. There are two possibilities:

- Use the actual index of element: i.e. insert it after element 3, get element number 6. This approach is used with arrays
- Use a "current" marker or pointer to refer to a particular position in the list.

The first option is used in the data structures like arrays. When we have to manipulate the arrays, we use index like *x[3], x[6]*. In the second option we do not use first, second etc for position but say wherever is the current pointer. Just think of a pointer in the list that we can move forward or backward. When we say get, insert or update

while using the current pointer, it means that wherever is the current pointer, get data from that position, insert data after that position or update the data at that position. In this case, we need not to use numbers. But it is our responsibility that current pointer is used in a proper way.

If we use the “current” marker, the following four methods would be useful:

Functions	Description
start()	Moves the “current” pointer to the very first element
tail()	Moves the “current” pointer to the very last element
next()	Move the current position forward one element
back()	Move the current position backward one element

In the next lecture, we will discuss the implementation of the list data structure and write the functions discussed today, in C++ language.

Data Structures

Lecture No. 02

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 3

3.1, 3.2, 3.2.1, 3.2.2

Summary

1) List Implementation
   - *add* Method
   - *next* Method
   - *remove* Method
   - *find* Method
   - Other Methods
2) Analysis Of Array List
3) List Using Linked Memory
4) Linked List

Today, we will discuss the concept of list operations. You may have a fair idea of '*start* operation' that sets the current pointer to the first element of the list while the *tail* operation moves the current pointer to the last element of the list. In the previous lecture, we discussed the operation *next* that moves the current pointer one element forward. Similarly, there is the '*back* operation' which moves the current pointer one element backward.

List Implementation

Now we will see what the implementation of the list is and how one can create a list in C++. After designing the interface for the list, it is advisable to know how to implement that interface. Suppose we want to create a list of integers. For this purpose, the methods of the list can be implemented with the use of an array inside. For example, the list of integers (2, 6, 8, 7, 1) can be represented in the following manner where the current position is 3.

Array Size = Total number

Index starting point = 0
index ending point = 1 less than array total size

A	2	6	8	7	1				**current**	**size**
	1	2	3	4	5				**3**	**5**

In this case, we start the index of the array from 1 just for simplification against the usual practice in which the index of an array starts from zero in C++. It is not necessary to always start the indexing from zero. Sometimes, it is required to start the indexing from 1. For this, we leave the zeroth position and start using the array from index 1 that is actually the second position. Suppose we have to store the numbers from 1 to 6 in the array. We take an array of 7 elements and put the numbers from the

index 1. Thus there is a correspondence between index and the numbers stored in it. This is not very useful. So, it does not justify the non-use of zeroth position of the array out-rightly. However for simplification purposes, it is good to use the index from 1.

add Method

Now we will talk about adding an element to the list. Suppose there is a call to add an element in the list i.e. *add(9)*. As we said earlier that the current position is 3, so by adding the element 9 to the list, the new list will be (2, 6, 8, 9, 7, 1).
To add the new element (9) to the list at the current position, at first, we have to make space for this element. For this purpose, we shift every element on the right of 8 (the current position) to one place on the right. Thus after creating the space for new element at position 4, the array can be represented as

Array size starting point = 1 (array size indicates total number of values)
Array index starting point = 0

Maximum index = 1 less than total array size

A	2	6	8		7	1			**current**	**size**
	1	2	3	4	5				**3**	**5**

Now in the second step, we put the element 9 at the empty space i.e. position 4. Thus the array will attain the following shape. The figure shows the elements in the array in the same order as stored in the list.

A	2	6	8	9	7	1			**current**	**size**
	1	2	3	4	5	6			**4**	**6**

We have moved the current position to 4 while increasing the size to 6. The size shows that the elements in the list. Where as the size of the array is different that we have defined already to a fixed length, which may be 100, 200 or even greater.

next Method

Now let's see another method, called '*next'*. We have talked that the next method moves the current position one position forward. In this method, we do not add a new element to the list but simply move the pointer one element ahead. This method is required while employing the list in our program and manipulating it according to the requirement. There is also an array to store the list in it. We also have two variables- *current* and *size* to store the position of current pointer and the number of elements in the list. By looking on the values of these variables, we can find the state of the list i.e. how many elements are in the list and at what position the current pointer is.
The method *next* is used to know about the boundary conditions of the list i.e. the array being used by us to implement the list. To understand the boundary conditions, we can take the example of an array of size 100 to implement the list. Here, 100 elements are added to the array. Let's see what happens when we want to add 101st element to the array? We used to move the current position by *next* method and reached the 100th position. Now, in case of moving the pointer to the next position (i.e. 101st), there will be an error as the size of the array is 100, having no position after this point. Similarly if we move the pointer backward and reach at the first position regardless that the index is 0 or 1. But what will happen if we want to move backward from the first position? These situations are known as boundary conditions and need attention during the process of writing programs when we write the code to use the list. We will take care of these things while implementing the list in C++

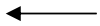
programs.

remove Method

We have seen that the *add* method adds an element in the list. Now we are going to discuss the *remove* method. The *remove* method removes the element residing at the current position. The removal of the element will be carried out as follows. Suppose there are 6 elements (2, 6, 8, 9, 7, 1) in the list. The current pointer is pointing to the position 5 that has the value 7. We remove the element, making the current position empty. The size of the list will become 5. This is represented in the following figure.

A	2	6	8	9		1			**current**	**size**
	1	2	3	4	5	6			**5**	~~**6**~~ **5**

We fill in the blank position left by the removal of 7 by shifting the values on the right of position 5 to the left by one space. This means that we shift the remaining elements on the right hand side of the current position one place to the left so that the element next to the removed element (i.e. 1) takes its place (the fifth position) and becomes the current position element. We do not change the current pointer that is still pointing to the position 5. Thus the current pointer remains pointing to the position 5 despite the fact that there is now element 1 at this place instead of 7. Thus in the *remove* method, when we remove an element, the element next to it on the right hand side comes at its place and the remaining are also shifted one place to the right. This step is represented by the following figure.



A	2	6	8	9	1				**current**	**size**
	1	2	3	4	5				**5**	**5**

find Method

Now lets talk about a function, used to find a specific element in the array. The *find (x)* function is used to find a specific element in the array. We pass the element, which is to be found, as an argument to the *find* function. This function then traverses the array until the specific element is found. If the element is found, this function sets the current position to it and returns 1 i.e. true. On the other hand, if the element is not found, the function returns 0 i.e. false. This indicates that the element was not found. Following is the code of this *find(x)* function in C++.

```
int find (int x)
{
        int j ;
        for (j = 1; j < size + 1; j++ )
                if (A[j] == x )
                        break ;
        if ( j < size + 1)                  // x is found
        {
                current = j ;               //current points to the position where x found
                return 1 ;                  // return true
```

```
        }
        return 0 ;                      //return false, x is not found
}
```

In the above code, we execute a *for* loop to traverse the array. The number of execution of this loop is equal to the size of the list. This *for* loop gets terminated when the value of loop variable (*j*) increases from the size of the list. However we terminate the loop with the *break* statement if the element is found at a position. When the control comes out from the loop, we check the value of *j*. If the value of *j* is less than the size of the array, it means that the loop was terminated by the *break* statement. We use the *break* statement when we find the required element (x) in the list. The execution of *break* statement shows that the required element was found at the position equal to the value of *j*. So the program sets the *current* position to *j* and comes out the function by returning 1 (i.e. true). If the value of *j* is greater than the size of the array, it means that the whole array has traversed and the required element is not found. So we simply return 0 (i.e. false) and come out of the function.

Other Methods

There are some other methods to implement the list using an array. These methods are very simple, which perform their task just in one step (i.e. in one statement). There is a *get()* method , used to get the element from the current position in the array. The syntax of this function is of one line and is as under

```
return A[current] ;
```

This statement returns the element to which the *current* is pointing to (i.e. the current position) in the list A.
Another function is *update(x)*. This method is used to change (set) the value at the current position. A value is passed to this method as an argument. It puts that value at the current position. The following statement in this method carries out this process.

```
A [current] = x ;
```

Then there is a method *length( )*.This method returns the size of the list. The syntax of this method is

```
return size ;
```

You may notice here that we are returning the size of the list and not the size of the array being used internally to implement the list. This *size* is the number of the elements of the list, stored in the array.

The *back()* method decreases the value of variable *current* by 1. In other words, it moves the current position one element backward. This is done by writing the statement.

```
current -- ;
```

The -- is a decrement operator in C++ that decreases the value of the operand by one. The above statement can also be written as

current = current -1 ;

The *start()* method sets the current position to the first element of the list. We know that the index of the array starts from 0 but we use the index 1 for the starting position. We do not use the index zero. So we set the current position to the first element by writing

current = 1 ;

Similarly, the *end()* method sets the current position to the last element of the list i.e. *size*. So we write

current = size ;

Analysis of Array List

Now we analyze the implementation of the list while using an array internally. We analyze different methods used for the implementation of the list. We try to see the level upto which these are efficient in terms of CPU's time consumption. Time is the major factor to see the efficiency of a program.

Add

First of all, we have talked about the *add* method. When we add an element to the list, every element is moved to the right of the current position to make space for the new element. So, if the current position is the start of the list and we want to add an element in the beginning, we have to shift all the elements of the list to the right one place. This is the worst case of adding an element to the list. Suppose if the size of the list is 10000 or 20000, we have to do the shift operation for all of these 10000 or 20000 elements. Normally, it is done by shifting of elements with the use of a *for* loop. This operation takes much time of the CPU and thus it is not a good practice to add an element at the beginning of a list. On the other hand, if we add an element at the end of the list, it can be done by carrying out 'no shift operation'. It is the best case of adding an element to the list. However, normally we may have to move half of the elements. The usage of add method is the matter warranting special care at the time of implementation of the list in our program. To provide the interface of the list, we just define these methods.

Remove

When we remove an element at the current position in the list, its space gets empty. The current pointer remains at the same position. To fill this space, we shift the elements on the right of this empty space one place to the left. If we remove an element from the beginning of the list, then we have to shift the entire remaining elements one place to the left. Suppose there is a large number of elements, say 10000 or 20000, in the list. We remove the first element from the list. Now to fill this space, the remaining elements are shifted (that is a large number). Shifting such a large number of elements is time consuming process. The CPU takes time to execute the *for* loop that performs this shift operation. Thus to remove an element at the beginning of the list is the worst case of *remove* method. However it is very easy to remove an

element at the end of the list. In average cases of the *remove* method we expect to shift half of the elements. This average does not mean that in most of the cases, you will have to shift half the elements. It is just the average. We may have to shift all the elements in one operation (if we remove at the beginning) and in the second operation, we have to shift no element (if we remove at the end). Similarly, in certain operations, we have to shift just 10, 15 elements.

Find

We have discussed that the *find* method takes an element and traverses the list to find that element. The worst case of the find method is that it has to search the entire list from beginning to end. So, it finds the element at the end of the array or the element is not found. On average the find method searches at most half the list.

The other methods get (), length () etc are one-step methods. They carry out their operation in one instruction. There is no need of any loop or other programming structures to perform the task. The *get()* method gets the value from the specified position just in one step. Similarly the *update()* method sets a value at the specific position just in one-step. The length () method returns the value of the size of the list. The other methods *back(), start()* and *end()* also perform their tasks only in one step.

List using Linked Memory

We have seen the implementation of the list with the use of an array. Now we will discuss the implementation of the list while using linked memory. In an array, the memory cells of the array are linked with each other. It means that the memory of the array is contiguous. In an array, it is impossible that one element of the array is located at a memory location while the other element is located somewhere far from it in the memory. It is located in very next location in the memory. It is a property of the array that its elements are placed together with one another in the memory. Moreover, when we have declared the size of the array, it is not possible to increase or decrease it during the execution of the program. If we need more elements to store in the array, there is need of changing its size in the declaration. We have to compile the program again before executing it. Now array will be of the new size. But what happens if we again need to store more elements? We will change the code of our program to change the declaration of the array while recompiling it.

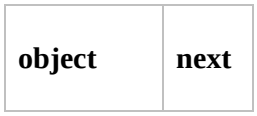
Suppose we have used the dynamic memory allocation and created an array of 100 elements with the use of *new* operator. In case of need of 200 elements, we will release this array and allocate a new array of 200 elements. Before releasing the previous array, it will wise to copy its elements to the new array so that it does not lose any information. Now this new array is in 'ready for use' position. Thus the procedure of creating a new array is not an easy task.

To avoid such problems, usually faced by the programmers while using an array, there is need of using linked memory in which the various cells of memory, are not located continuously. In this process, each cell of the memory not only contains the value of the element but also the information where the next element of the list is residing in the memory. It is not necessary that the next element is at the next location in the memory. It may be anywhere in the memory. We have to keep a track of it. Thus, in this way, the first element must explicitly have the information about the location of the second element. Similarly, the second element must know where the

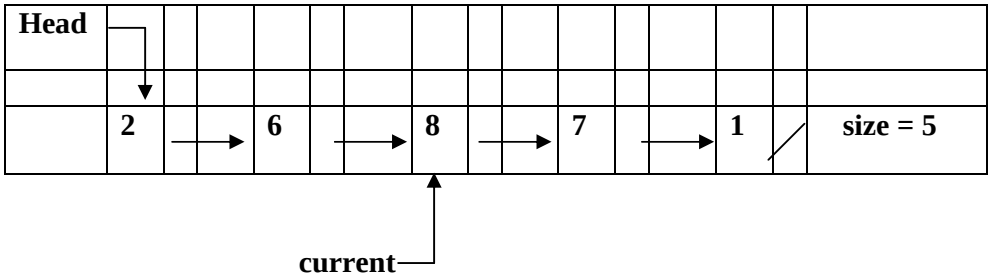
third element is located and the third should know the position of the fourth element and so on. Thus, each cell (space) of the list will provide the value of the element along with the information about where the next element is in the memory. This information of the next element is accomplished by holding the memory address of the next element. The memory address can be understood as the index of the array. As in case of an array, we can access an element in the array by its index. Similarly, we can access a memory location by using its address, normally called memory address.

Linked List

For the utilization of the concept of linked memory, we usually define a structure, called linked list. To form a linked list, at first, we define a node. A node comprises two fields. i.e. the *object* field that holds the actual list element and the *next* that holds the starting location of the next node.



A chain of these nodes forms a linked list. Now let's consider our previous list, used with an array i.e. 2, 6, 8, 7, 1. Following is the figure which represents the list stored as a linked list.



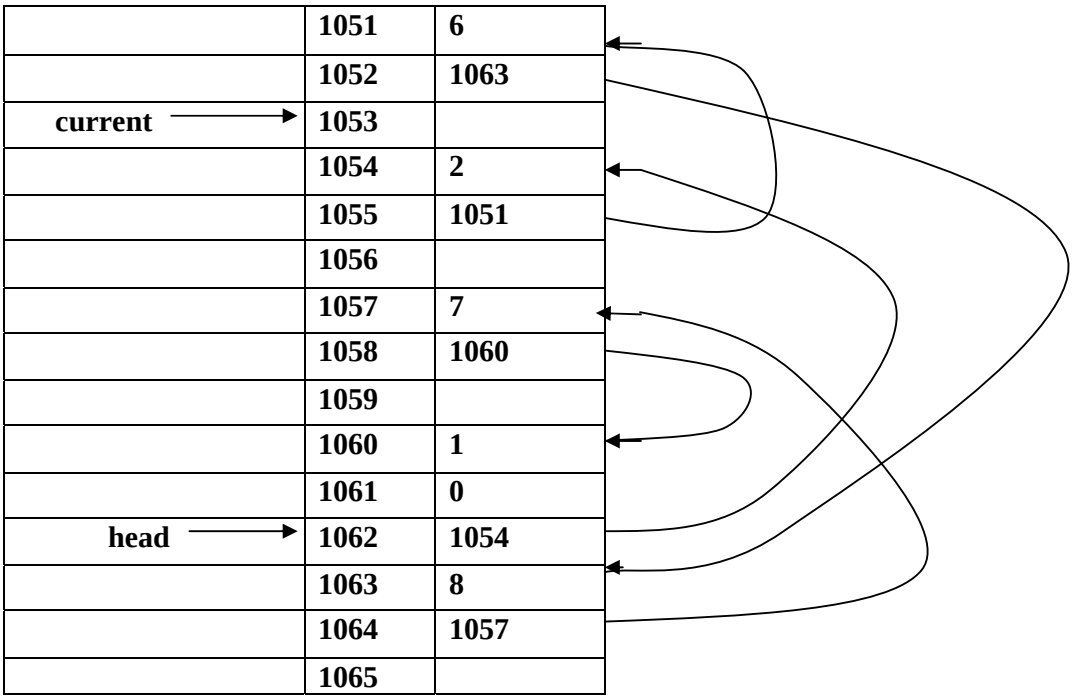
This diagram just represents the linked list. In the memory, different nodes may occur at different locations but the *next* part of each node contains the address of the next node. Thus it forms a chain of nodes which we call a linked list.

While using an array we knew that the array started from index 1that means the first element of the list is at index 1. Similarly in the linked list we need to know the starting point of the list. For this purpose, we have a pointer *head* that points to the first node of the list. If we don't use *head*, it will not be possible to know the starting position of the list. We also have a pointer *current* to point to the current node of the list. We need this pointer to add or remove current node from the list. Here in the linked list, the *current* is a pointer and not an index as we used while using an array. The *next* field of the last node points to nothing .It is the end of the list. We place the memory address NULL in the last node. NULL is an invalid address and is inaccessible.

Now again consider the list 2, 6, 8, 7, 1. The previous figure represents this list as a linked list. In this linked list, the *head* points to 2, 2 points to 6, 6 points to 8, 8 points to 7 and 7 points to 1. Moreover we have the current position at element 8.

This linked list is stored in the memory. The following diagram depicts the process

through which this linked list is stored in the memory.



We can see in the figure that each memory location has an address. Normally in programming, we access the memory locations by some variable names. These variable names are alias for these locations and are like labels that are put to these memory locations. We use *head* and *current* variable names instead of using the memory address in numbers for starting and the current nodes. In the figure, we see that *head* is the name of the memory location 1062 and the name *current* is used for the memory address 1053. The *head* holds the address 1054 and the element 2, the first one in the list, is stored in the location 1054. Similarly *current* holds the address 1063 where the element 8 is stored which is our current position in the list. In the diagram, two memory locations comprise a node. So we see that the location 1054 holds the element 2 while the next location 1055 holds the address of the memory location (1051) where the next element of the list (i.e. 6) is stored. Similarly the *next* part of the node that has value 6 holds the memory address of the location occupied by the next element (i.e. 8) of the list. The other nodes are structured in a similar fashion. Thus, by knowing the address of the next element we can traverse the whole list.

Data Structures

Lecture No. 03

Reading Material

Data Structures and algorithm analysis in C++ Chapter. 3
3.2.2, 3.2.3, 3.2.5

Summary

- Linked List inside Computer Memory
- Linked List Operations
- Linked List Using C++
- Example Program

In the previous lectures, we used an array to construct a list data structure and observed the limitation that array being of fixed size can only store a fixed number of elements. Therefore, no more elements can be stored after the size of the array is reached.

In order to resolve this, we adopted a new data structure called *linked list.* We started discussing, how linked lists are stored in computer memory and how memory chains are formed.

Linked List inside Computer Memory

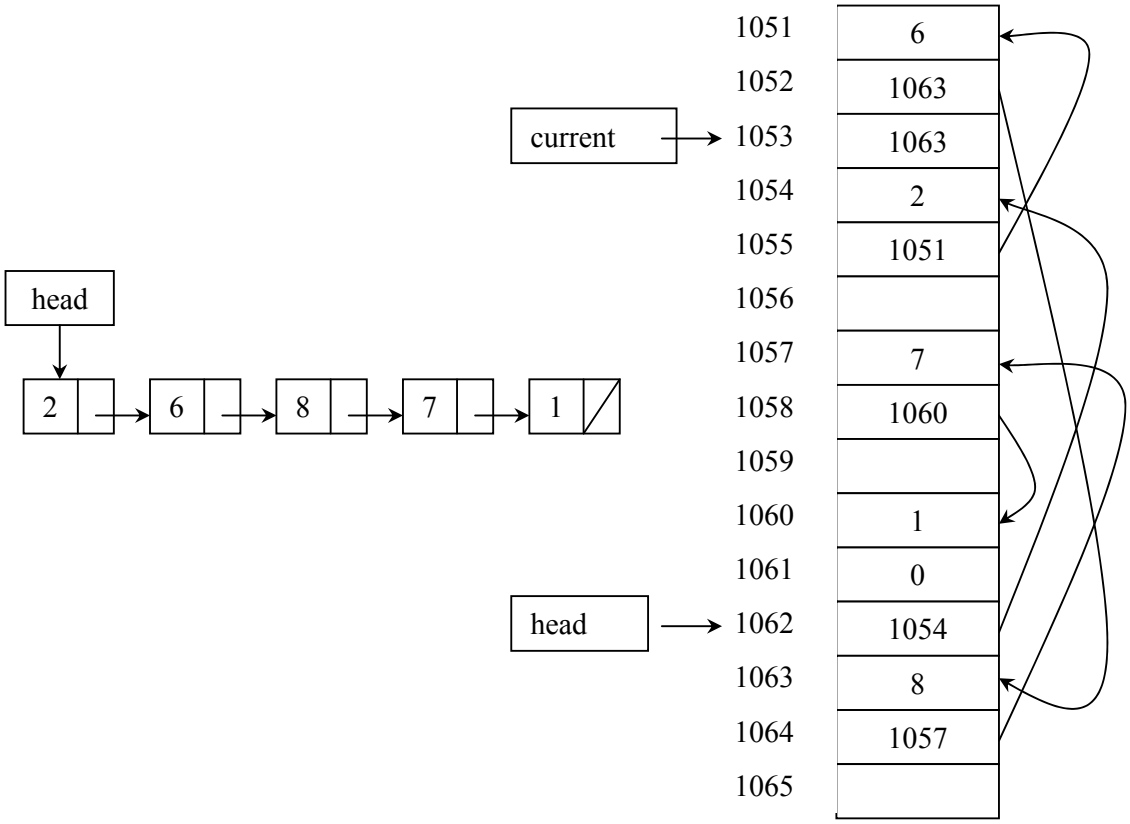


Fig 1. Linked list in memory

There are two parts of this figure. On the left is the linked list chain that is actually the conceptual view of the linked list and on the right is the linked list inside the computer memory. Right part is a snapshot of the computer memory with memory addresses from *1051* to *1065*. The *head* pointer is pointing to the first element in the linked list. Note that *head* itself is present in the memory at address *1062*. *It is* actually a pointer containing the memory address *1054*. Each node in the above linked list has two parts i.e. the data part of the node and the pointer to the next node. The first node of the linked list pointed by the *head* pointer is stored at memory address *1054*. *We* can see the data element *2* present at that address. The second part of the first node contains the memory address *1051*. So the second linked list's node starts at memory address *1051*. We can use its pointer part to reach the third node of the list and in this way, we can traverse the whole list. The last node contains *1* in its data part and *0* in its pointer part. *0* indicates that it is not pointing to any node and it is the last node of the linked list.

Linked List Operations

The linked list data structure provides operations to work on the nodes inside the list. The first operation we are going to discuss here is to create a new node in the memory. The *Add(9)* is used to create a new node in the memory at the current

position to hold '*9*'. You must remember while working with arrays, to add an element at the current position that all the elements after the current position were shifted to the right and then the element was added to the empty slot.
Here, we are talking about the internal representation of the list using linked list. Its *interface* will remain the same as in case of arrays.

We can create a new node in the following manner in the *add()* operation of the linked list with code in C++:

```
Node * newNode = new Node(9);
```

The first part of the statement that is on the left of the assignment is declaring a variable pointer of type *Node*. It may also be written as *Node * newNode*. On the right of this statement, the *new* operator is used to create a new *Node* object as *new Node(9)*. This is one way in C++ to create objects of classes. The name of the class is provided with the *new* operator that causes the constructor of the class to be called. The constructor of a class has the same name as the class and as this a function, parameters can also be passed to it. In this case, the constructor of the *Node* class is called and '*9*' is passed to it as an *int* parameter.

Hence, the whole statement means:
"Call the constructor of the *Node* class and pass it '*9*' as a parameter. After constructing the object in memory, give me the starting memory address of the object. That address will be stored in the pointer variable *newNode*."

To create an object of *int* type in the same manner, we can write as:

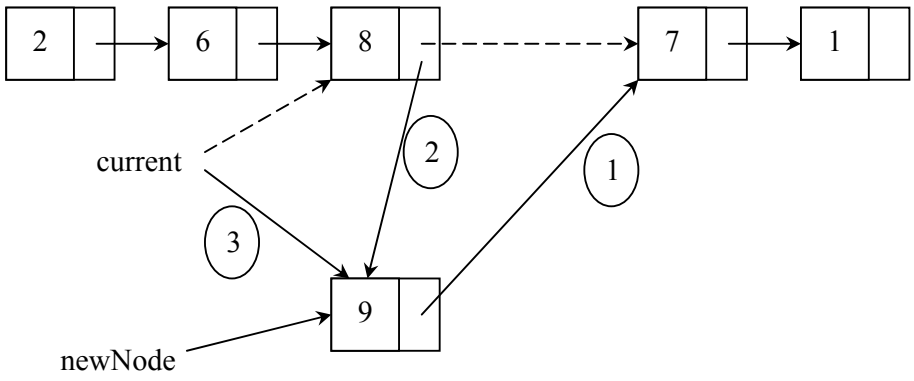
```
int * i = new int ;
```

Previously, we used the same technique to allocate memory for an array of *int*s as:

```
int * i = new int [10] ;
```

Now after the node has been created, how the node is fit into the chain of the linked list.

Fig 2. Insertion of new Node into the linked list



In the above figure, there is a linked list that contains five nodes with data elements as *2*, *6*, *8*, *7* and *1*. The *current* pointer is pointing to the node with element as *8*. We want to insert a new node with data element *9*. This new node will be inserted at the current position (the position where the *current* pointer is pointing to). This insertion

operation is performed in a step by step fashion.

- The first step is to point next pointer of the new node (with data element as *9*) to the node with data element as *7*.
- The second step is to point the next pointer of the node with data element *8* to the node the new node with data element *9*.
- The third step is to change the *current* pointer to point to the new node.

Now, the updated linked list has nodes with data elements as *2*, *6*, *8*, *9*, *7* and *1*. The list size has become 6.

Linked List Using C++

```
/* The Node class */

class   Node
{
public:
        int   get() { return   object; };
        void   set(int   object) { this->object   =   object; };

        Node   * getNext() { return   nextNode; };
        void   setNext(Node   * nextNode) { this->nextNode   =   nextNode; };
private:
        int   object;
        Node   * nextNode;
};
```

Whenever, we write a class, it begins with the word *class* followed by the *class-name* and the body of the class enclosed within curly braces. In the body, we write its *public* variables, methods and then *private* variables and methods, this is normally the sequence.

If there is no code to write inside the constructor function of a class, we need not to declare it ourselves as the compiler automatically creates a default constructor for us. Similarly, if there is nothing to be done by the destructor of the class, it will be better not to write it explicitly. Rather, the compiler writes it automatically. Remember, the default constructor and destructor do nothing as these are the function without any code statements inside.

Let's start with the data members first. These are given at the bottom of the class body with the scope mentioned as *private*. These data members are actually two parts of a linked list's node. First variable is *object* of type *int*, present there to store the data part of the node. The second variable is *nextNode*, which is a pointer to an object of type *Node*. It has the address of the next element of the linked list.

The very first *public* function given at the top is *get()*. We have written its code within the class *Node. It* returns back the value of the variable *object* i.e. of the type of *int.*

When we write class in C++, normally, we make two files (*.h* and *.cpp*) for a class. The *.h* file contains the declarations of *public* and *private* members of that class. The *public* methods are essentially the interface of the class to be employed by the users of this class. The *.cpp* file contains the implementation for the class methods that has the

actual code. This is usually the way that we write two files for one class. But this is not mandatory. In the code given above, we have only one file *.cpp,* instead of separating into two files. As the class methods are very small, so their code is written within the body of the class. This facilitates us in carrying on discussion. Thus instead of talking about two files, we will only refer to one file. On the other hand, compiler takes these functions differently that are called *inline* functions. The compiler replaces the code of these *inline* functions wherever the call to them is made.

The second method in the above-mentioned class is *set() that* accepts a parameter of type *int* while returning back nothing. The accepted parameter is assigned to the internal data member *object*. Notice the use of *this* pointer while assigning the value to the internal data member. It is used whenever an object wants to talk to its own members.
The next method is *getNext() which* returns a pointer to an object of type *Node* lying somewhere in the memory. It returns *nextNode i.e.* a pointer to an object of type *Node*. As discussed above, *nextNode* contains the address of next node in the linked list.
The last method of the class is *setNext() that* accepts a pointer of type *Node,* further assigned to *nextNode* data member of the object. This method is used to connect the next node of the linked list with the current object. It is passed an address of the next node in the linked list.

Let's discuss a little bit about classes. A very good analogy of a class is a factory. Think about a car factory. On the placement of order, it provides us with the number of vehicles we ordered for. Similarly, you can see number of other factories in your daily-life that manufacture the specific products.
Let's take this analogy in C++ language. Suppose, we want to make a factory in C++. By the way, what is our *Node* class? It is actually a factory that creates nodes. When we want to make a new node, a new operator is used. By using *new* operator with the *Node* class, actually, we send an order to *Node* factory, to make as many as nodes for us.

So we have a good analogy, to think about a class as a factory. The products that are made by the factory have their own characteristics. For example, a car made by an automobile factory has an engine, wheels, steering and seats etc. These variables inside a class are called *state variables*. Now the kinds of operations this car can do are the *methods* of its class. A car can be driven, engine can be started, gears can be shifted and an accelerator can be pressed to run it faster.
Similarly, the Node class creates nodes, where every node has two-state variables i.e. *object* and *nextNode*. We have already seen its operations in the above code. We use *new* to create new object or an array of new objects, stored in memory.

Let's see the code below.

```
/* List class */
#include   <stdlib.h>
#include   "Node.cpp"

class   List
{
public:
```

```
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                currentNode = NULL;
                size = 0;
        }
```

We are creating a list factory here employed to create list objects. Remember the list operations; *add*, *remove*, *next*, *back* and *start* etc. Let's see the above class declaration code in detail.
There are two *include statements* at the start. The first line is to include a standard library *stdlib.h while the second line* includes the *Node* class file *Node.cpp*. This *Node* class is used to create nodes that form a *List* object. So this *List* factory will order *Node* class to create new nodes. The List class itself carries out the chain management of these Node objects.

We have written our own constructor of *List* class as the default constructor is not sufficient enough to serve the purpose. The *List* constructor is parameterless. The very first step it is doing internally is that it is asking *Node* class to create a new node and assigning the starting address of the new *Node*'s object to the *headNode* data member. In the second statement, we are calling *setNext()* method of the *Node* class for the object pointed to by the *headNode* pointer. This call is to set the *nextNode* data member to *NULL*, i.e. *Node*'s object pointed to by the *headNode* pointer is not pointing to any further *Node*. The next statement is to set the *currentNode* pointer to *NULL*. So at the moment, we have initialized the *currentNode* pointer to *NULL* that is not pointing to any *Node* object. The next statement is to initialize the *size* data member to *0* indicating that there is no node present in the list. All this processing is done inside the constructor of *List* class, as we want all this done when a list object is created. Considering the analogy of car factory, the constructor function can perform certain tasks: The oil is poured into the engine, the tyres are filled-in with air etc.

Let's see the *add* method of the *List* class:

```
/* add() class method */
void  add (int  addObject)
{
1.      Node *  newNode  =  new  Node();
2.      newNode->set(addObject);
3.      if( currentNode  !=  NULL )
4.      {
5.              newNode->setNext(currentNode->getNext());
6.              currentNode->setNext( newNode );
7.              lastCurrentNode  =  currentNode;
8.              currentNode  =  newNode;
9.      }
10.     else
11.     {
12.             newNode->setNext(NULL);
13.             headNode->setNext(newNode);
```

```
14.             lastCurrentNode = headNode;
15.             currentNode = newNode;
16.     }
17.     size ++;
}
```

The interface or signatures of *add()* method is similar to the one discussed in case of an array. This method takes the object to be added as a parameter. The implementation of this *add()* method is a bit longer as the method is being implemented for linked list. In the first statement, a new *Node* object is created with its address stored in the *newNode* pointer variable. The second statement is to call *set()* method of the *Node* object pointed to by the *newNode* pointer. You can note the way the method is called. A pointer variable is at the left most side then an arrow sign (->), then the name of the method with appropriate arguments within parenthesis. It is followed by the if-statement that checks the *currentNode* is not *NULL* to perform certain operations inside the if-code block. Inside the if-statement, at line 5, the *nextNode* pointer of the new node is being set to the *nextNode* of the object pointed to by the *currentNode* pointer. In order to understand the statements given in this code properly, consider the fig 2 above, where we added a node in the linked list. We have done step 1 at line5. At line 6, we are performing the second step by setting the *newNode* in the *nextNode* pointer of the object pointed to by the *currentNode*. At line 7, we are saving the current position (address) of the *currentNode* pointer in the pointer variable *lastCurrentNode*, which might be useful for backward traversing. Although, the fig 1 (left part) indicates movement in one direction from left to right but the *lastCurrentNode* pointer node can be used by the *back()* member function to traverse one position back from right to left. At line 8, the *currentNode* pointer is assigned the address of the object pointed to by *newNode*. This way, a new node is added in already existent linked list.

Line 10 is start of the *else* part of if-statement. This is executed if the *currentNode* is *NULL*. It means that there is no node present in the list previously and first node is going to be added. At line 12, we are setting the *nextNode* pointer of the object pointed to by *newNode* pointer. The *nextNode* is being set to *NULL* by calling the *setNext()* method. Then at line 13, we point the *head* pointer (*headNode*) to this new node pointed to by *newNode* pointer. Note that *headNode* is pointing to a node that is there despite the fact that the *size* of the linked list is 0. Actually, we have allocated a *Node* object for *headNode* pointer. Although, we don't need a *Node* object here, yet it will be helpful when we perform other operations like *remove()* and *find()*.

At line 14, the *headNode* address is being assigned to *lastCurrentNode.* At line 15, *currentNode* pointer is assigned the address of *newNode*. At the end i.e. at line 17, the *size* of the list is incremented by 1.

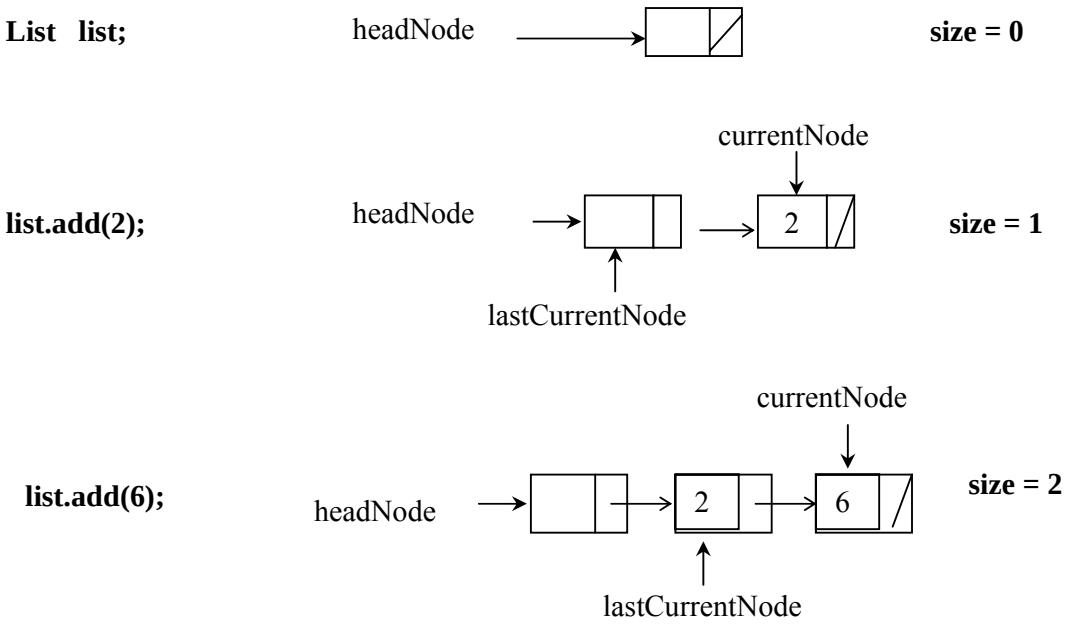


Fig 3. Add operation of linked list

Following is the crux of this *add()* operation :
Firstly, it will make a new node by calling Node class constructor. Insert the value e.g. 2. of the node into the node by calling the set method. Now if the list already exists (has some elements inside or its size is non-zero), it will insert the node after the current position. If the list does not already exist, this node is added as the first element inside the list.

Let's try to add few more elements into the above linked list in the figure. The following are the lines of code to be executed to add nodes with values *8*, *7* and *1* into the linked list.

list.add(8); list.add(7); list.add(1);

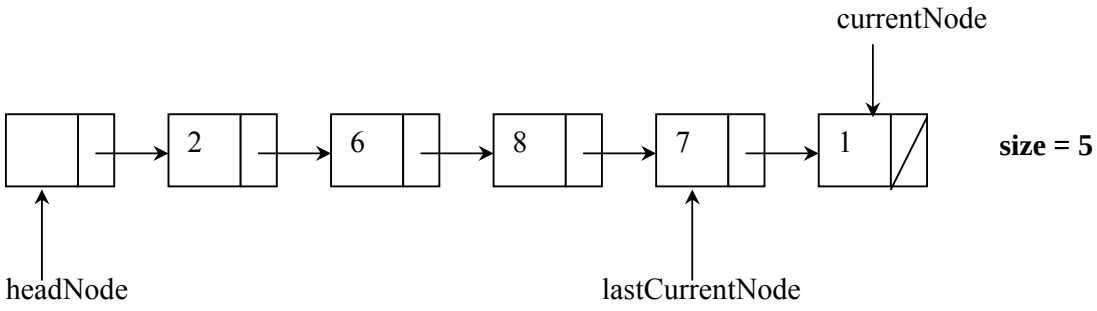


Fig 4. More Nodes added into linked list

Now we will see the remaining methods of the linked list. The *get()* method of the *List* class is given below

```
/* get() class method */
int   get()
{
        if (currentNode  !=  NULL)
                return   currentNode->get();
}
```

This method firstly confirms that the *currentNode* pointer is not *NULL*. If it is not *NULL,* then it must be pointing to some *Node* object as inside the constructor of the *List* class, we have initialized this pointer variable to *NULL*. That indicates that the *currentNode* is *NULL* when there is no element inside the list. However, when a *Node* object is added into it, it starts pointing to it. So, this *get()* returns the address of the node pointed to by the *currentNode* pointer.

Further, we have another method given below:

```
/* next() class method */
bool   next()
{
1.      if (currentNode  ==  NULL)   return  false;
2.
3.      lastCurrentNode  =  currentNode;
4.      currentNode  =  currentNode->getNext();
5.      return   true;
};
```

This is *next*() method, used to advance the *currentNode* pointer to the next node inside the linked list. At line 1, the *currentNode* is being checked to confirm that there are some elements present in the list to advance further. At line 1, the method is returning *false* if there is no element present in the list. At line 3, it is storing the value of the *currentNode* pointer into the *lastCurrentNode*. At line 4, *currentNode* is calling the *getNext()* method to get the address of next node to be stored in the *currentNode* pointer to advance the *currentNode* pointer to the next element. At line 5, it returns *true* indicating the method is successful in moving to the next node.

Example Program

Given below is the full source code of the example program. You can copy, paste and compile it right away. In order to understand the linked list concept fully, it is highly desirable that you understand and practice with the below code.

```
#include   <iostream.h>
#include   <stdlib.h>

/* The Node class */
class   Node
{
   public:
            int     get() { return   object; };
            void   set(int   object) { this->object   =   object; };
```

```
            Node   * getNext() { return   nextNode; };
            void    setNext(Node   * nextNode) { this->nextNode   =   nextNode; };

    private:
      int   object;
      Node   * nextNode;
};

/* The List class */
class   List
{
public:
   List();
   void   add (int   addObject);
   int    get();
   bool   next();
   friend void  traverse(List list);
   friend List   addNodes();

private:
   int     size;
   Node *   headNode;
   Node *   currentNode;
   Node *   lastCurrentNode;

};
/* Constructor */
List::List()
{
         headNode   =   new Node();
         headNode->setNext(NULL);
         currentNode   =   NULL;
         lastCurrentNode   =   NULL;
         size   =   0;
}

/* add() class method */
void   List::add (int   addObject)
{
   Node *   newNode   =   new   Node();
   newNode->set(addObject);
   if( currentNode   !=   NULL )
   {
      newNode->setNext(currentNode->getNext());
      currentNode->setNext( newNode );
      lastCurrentNode   =   currentNode;
      currentNode   =   newNode;
    }
    else
    {
```

```
            newNode->setNext(NULL);
            headNode->setNext(newNode);
            lastCurrentNode  =  headNode;
            currentNode  =  newNode;
        }
        size ++;
}

/* get() class method */
int  List::get()
{
            if (currentNode  !=  NULL)
             return  currentNode->get();
}

/* next() class method */
bool  List::next()
{
   if (currentNode  ==  NULL)  return  false;

   lastCurrentNode  =  currentNode;
   currentNode  =  currentNode->getNext();
   if (currentNode == NULL || size == 0)
            return  false;
   else
            return  true;
}

/* Friend function to traverse linked list */
void traverse(List list)
{
   Node* savedCurrentNode  =  list.currentNode;
   list.currentNode  =  list.headNode;

   for(int i = 1; list.next(); i++)
   {
      cout << "\n Element " << i << "   " << list.get();
   }
   list.currentNode  =  savedCurrentNode;
}

/* Friend function to add Nodes into the list */
List addNodes()
{
   List  list;
   list.add(2);
   list.add(6);
   list.add(8);
   list.add(7);
   list.add(1);
```

```
   cout  <<  "\n List size = "  <<  list.size  <<'\n';
   return  list;
}

main()
{
   List list  =  addNodes();
   traverse(list);
}
```

The output of the example program is as follows:

```
List size = 5

Element  1   2
Element  2   6
Element  3   8
Element  4   7
Element  5   1
```

Data Structures

Lecture No. 04

Reading Material

Data Structures and algorithm analysis in C++ Chapter. 3
3.2.3, 3.2.4, 3.2.5

Summary

- Methods of Linked List
- Example of list usage
- Analysis of Link List
- Doubly-linked List
- Circularly-linked lists
- Josephus Problem

Methods of Linked List

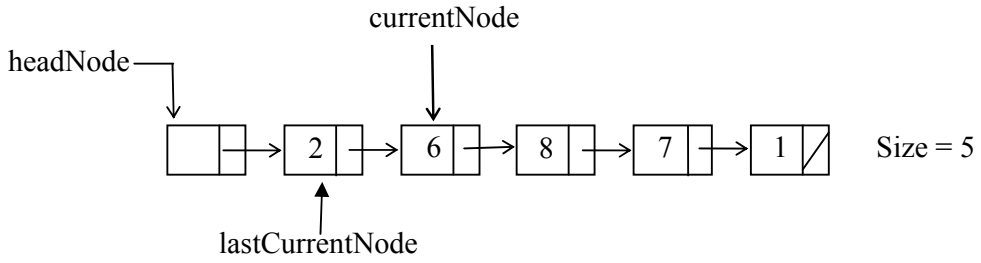
In the previous lecture, we discussed the methods of linked list. These methods form the interface of the link list. For further elucidation of these techniques, we will talk about the *start* method that has the following code.

```
// position currentNode and lastCurrentNode at first element
void start() {
        lastCurrentNode = headNode;
        currentNode = headNode;
};
```

There are two statements in this method. We assign the value of *headNode* to both *lastCurrentNode* and *currentNode*. These two pointers point at different nodes of the list. Here we have pointed both of these pointers at the start of the list. On calling some other method like *next,* these pointers will move forward. As we can move in the singly-linked list in one direction, these pointers cannot go behind *headNode*.

We will now see how a node can be removed from the link list. We use the method *remove* for this purpose.

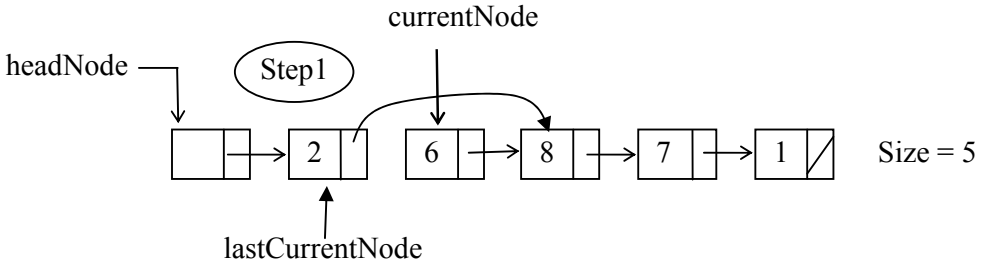
```
        void remove() {
            if( currentNode != NULL && currentNode != headNode) {
(step 1)        lastCurrentNode->setNext(currentNode->getNext());
(step 2)        delete currentNode;
(step 3)        currentNode = lastCurrentNode;
(step 4)        size--;
            }
        };
```



Suppose that the *currentNode* is pointing at the location that contains the value 6. A request for the removal of the node is made. Resultantly, the node pointed by *currentNode* should be removed. For this purpose, at first, the *next* pointer of the node with value 2 (the node pointed by the *lastCurrentNode* pointer), that is before the node with value 6, bypasses the node with value 6. It is, now pointing to the node with value 8. The code of the first step is as:

```
lastCurrentNode->setNext(currentNode->getNext());
```

What does the statement *currentNode->getNext()* do? The *currentNode* is pointing to the node with value 6 while the *next* of this node is pointing to the node with value 8. That is the *next* pointer of node with value 6 contains the address of the node with value 8. The statement *lastCurrentNode->setNext(currentNode->getNext())* will set the *next* pointer of the node pointed by the *lastCurrentNode* to the node with value 8. So the *next* pointer of the node with value 2 is pointing to the node with value 8.

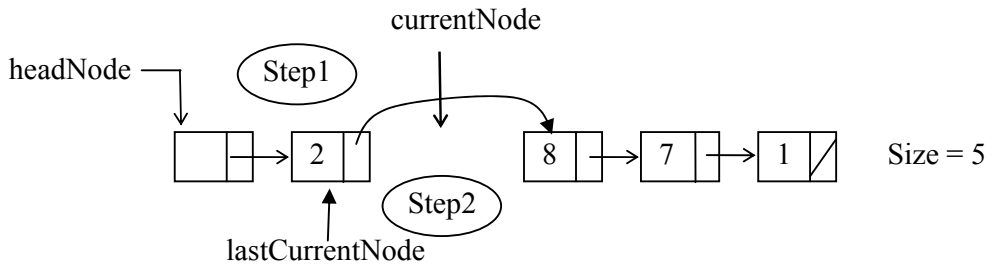


You see that the next pointer of the node having data element 2 contains the address of the node having data element 8. The node with value 6 has been disconnected from the chain while the node with value 2 is connected to the node with the value 8.

The code of the next step is:

```
delete currentNode;
```

You already know, in case of allocation of the memory with the help of the *new* keyword, the *delete* statement releases this memory which returns the memory to the system. Pictorially it can be represented as:

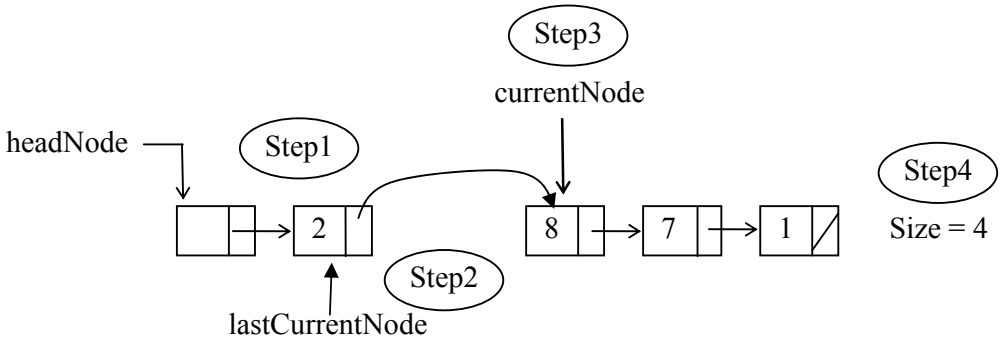


In the next step, we have moved the *currentNode* to point the next node. The code is:

```
currentNode = lastCurrentNode;
```

In the fourth step, the size of the list has been reduced by 1 after the deletion of one node i.e.

```
size--;
```



The next method is *length()* that simply returns the size of the list. The code is as follows:

```
// returns the size of the list
int length()
{
        return size;
};
```

The private data members of the list are:

```
private:
        int size;                 // contains the size of the list
        Node *headNode;           // points to the first node of the list
        Node *currentNode,        // current node
        Node *lastCurrentNode; // last current node
```

The list class completed just now, can be termed as list factory. We have included all

the required methods in it. We may employ more methods if required. A programmer can get the size of the list, add or remove nodes in it besides moving the pointers.

Example of list usage

Now let's see how we use the link list. Here is an example showing the use of list:

```
/* A simple example showing the use of link list */

#include <iostream>
#include <stdlib.h>
#include "List.cpp" // This contains the definition of List class

// main method
int main(int argc, char *argv[])
{
        List list;  // creating a list object

        // adding values to the list
        list.add(5);
        list.add(13);
        list.add(4);
        list.add(8);
        list.add(24);
        list.add(48);
        list.add(12);

        // calling the start method of the list
        list.start();

        // printing all the elements of the list
        while (list.next())
                cout << "List Element: "<< list.get()<<endl;
}
```

The output of the program is:

```
List Element: 5
List Element: 13
List Element: 4
List Element: 8
List Element: 24
List Element: 48
List Element: 12
```

Let's discuss the code of the above program. We have included the standard libraries besides having the "List.cpp" file. Usually we do not include .cpp files. Rather, the .h files are included. Whenever you write a class, two files will be created i.e. .h (header file containing the interface of the class) and .cpp (implementation file). Here for the sake of explanation, we have combined the two files into "List.cpp" file. At the start

of the main method, we have created a list object as:

```
List list;
```

Here the default constructor will be called. If you understand the concept of factory, then it is not difficult to know that we have asked the *List* factory to create a *List* object and named it as *list*. After creating the object, nodes have been added to it. We have added the elements with data values 5, 13, 4, 8, 24, 48 and 12. Later, the *start()* method of list is called that will position the *currentNode* and *lastCurrentNode* at the start of the list. Now there is no need to worry about the implementation of the List. Rather, we will use the interface of the List. So the *start* method will take us to the start of the *list* and internally, it may be array or link list or some other implementation. Then there is a while loop that calls the *next()* method of the *List*. It moves the pointer ahead and returns a boolean value i.e. true or false. When we reach at the end of the list, the *next()* method will return false. In the while loop we have a *cout* statement that prints the value of the list elements, employing the *get()* method. The loop will continue till the *next()* method returns true. When the pointers reach at the end of the list the *next()* will return false. Here the loop will come to an end.

Please keep in mind that list is a very important data structure that will be used in the entire programming courses.

Analysis of Link List

As stated earlier, we will be going to analyze each data structure. We will see whether it is useful or not. We will see its cost and benefit with respect to time and memory. Let us analyze the link list which we have created with the dynamic memory allocation in a chain form.

- **add**
  For the addition purposes, we simply insert the new node after the current node. So 'add' is a one-step operation. We insert a new node after the current node in the chain. For this, we have to change two or three pointers while changing the values of some pointer variables. However, there is no need of traversing too much in the list. In case of an array, if we have to add an element in the centre of the array, the space for it is created at first. For this, all the elements that are after the current pointer in the array, should be shifted one place to the right. Suppose if we have to insert the element in the start of the array, all the elements to the right one spot are shifted. However, for the link list, it is not something relevant. In link lists, we can create a new node very easily where the current pointer is pointing. We have to adjust two or three pointers. Its cost, in terms of CPU time or computing time, is not much as compared to the one with the use of arrays.

- **remove**
  Remove is also a one-step operation. The node before and after the node to be removed is connected to each other. Update the current pointer. Then the node to be removed is deleted. As a result, the node to be removed is deleted. Very little work is needed in this case. If you compare it with arrays, for the deletion of an element from the array, space is created. To fill this space, all the right elements are shifted one spot left. If the array size is two thousand or three thousand, we need to run a loop for all these elements to shift them to left.

- **find**
  The worst-case in find is that we may have to search the entire list. In find, we have to search some particular element say *x*. If found, the *currentNode* pointer is moved at that node. As there is no order in the list, we have to start search from the beginning of the list. We have to check the value of each node and compare it with x (value to be searched). If found, it returns true and points the *currentNode* pointer at that node otherwise returns false. Suppose that *x* is not in the list, in this case, we have to search the list from start to end and return false. This is the worst case scenario. Though time gets wasted, yet we find the answer that *x* is not in the list. If we compare this with array, it will be the same. We don't know whether *x* is in the array or not. So we have to search the complete array. In case of finding it, we will remember that position and will return true. What is the average case? *x* can be found at the first position , in the middle or at the end of the list. So on average, we have to search half of the list.

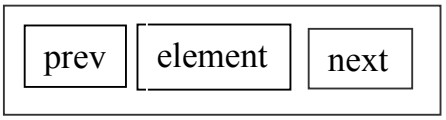
- **back**
  In the back method, we move the *current* pointer one position back. Moving the *current* pointer back, one requires traversing the list from the start until the node whose *next* pointer points to current node. Our link list is singly linked list i.e. we can move in one direction from start towards end. Suppose our *currentNode* pointer and *lastCurrentNode* are somewhere in the middle of the list. Now we want to move one node back. If we have the pointer of *lastCurrentNode,* it will be easy. We will assign the value of *lastCurrentNode* to *currentNode*. But how can we move the *lastCurrentNode* one step back. We don't have the pointer of previous node. So the solution for this is to go at the start of the list and traverse the list till the time you reach the node before the *lastCurrentNode* is pointing. That will be the node whose next pointer contains the value *lastCurrentNode*. If the *currentNode* and the *lastCurrentNode* are at the end of the list, we have to traverse the whole list. Therefore back operation is not a one step operation. We not only need a loop here but also require time.

Doubly-linked List

Doubly linked list is a linear data structure with elements pointing to both the next and previous node

If you look at single link list, the chain is seen formed in a way that every node has a field *next* that point to the next node. This continues till the last node where we set the *next* to NULL i.e. the end of the list. There is a *headNode* pointer that points to the start of the list. We have seen that moving forward is easy in single link list but going back is difficult. For moving backward, we have to go at the start of the list and begin from there. Do you need a list in which one has to move back or forward or at the start or in the end very often? If so, we have to use double link list.

In doubly-link list, a programmer uses two pointers in the node, i.e. one to point to next node and the other to point to the previous node. Now our node factory will create a node with three parts.



First part is *prev* i.e. the pointer pointing to the previous node, second part is *element*,

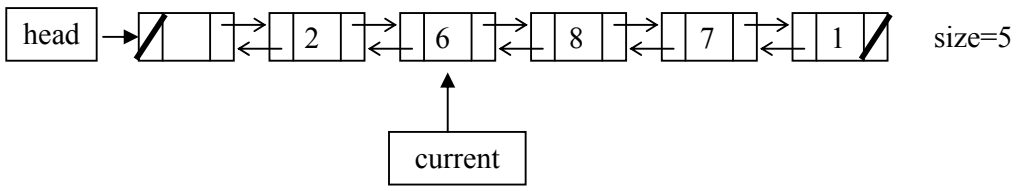
containing the data to be inserted in the list. The third part is *next* pointer that points to the next node of the list. The objective of *prev* is to store the address of the previous node.

Let's discuss the code of the node of the doubly-link list. This node factory will create nodes, each having two pointers. The interface methods are same as used in singly link list. The additional methods are *getPrev* and *setPrev*. The method *getPrev* returns the address of the previous node. Thus its return type is *Node**. The *setPrev* method sets the *prev* pointer. If we have to assign some address to *prev* pointer, we will call this method. Following is the code of the doubly-linked list node.

```
/* this is the doubly-linked list class, it uses the next and prev pointers */

class Node {
public:
        int get() { return object; };      // returns the value of the element
        void set(int object) { this->object = object; };  // set the value of the element

        Node* getNext() { return nextNode; };  // get the address of the next node
        void setNext(Node* nextNode)           // set the address of the next node
                { this->nextNode = nextNode; };

        Node* getPrev() { return prevNode; }; // get the address of the prev node
        void setPrev(Node* prevNode)           // set the address of the prev node
                { this->prevNode = prevNode; };
private:
        int object;                // it stores the actual value of the element
        Node* nextNode;        // this points to the next node
        Node* prevNode;        // this points to the previous node
};
```

Most of the methods are same as those in singly linked list. A new pointer *prevNode* is added and the methods to get and set its value i.e. *getPrev* and *setPrev*. Now we will use this node factory to create nodes.

You have to be very cautious while adding or removing a node in a doubly linked list. The order in which pointers are reorganized is important. Let's have a pictorial view of doubly-link list. The diagram can help us understand where the *prevNode* and *nextNode* are pointing.



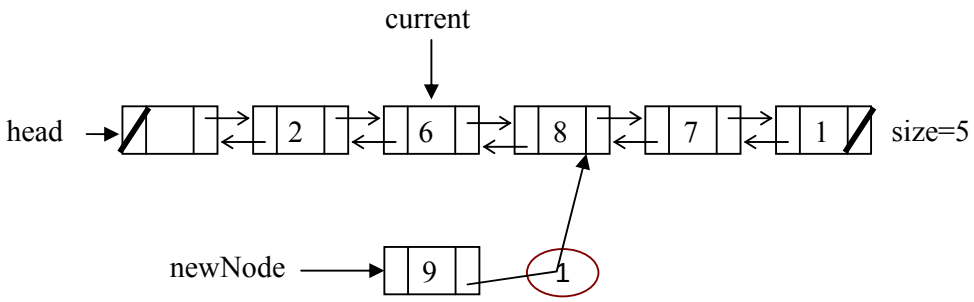
This is a doubly link list. The arrows pointing towards right side are representing *nextNode* while those pointing towards left side are representing *prevNode*. Suppose

we are at the last node i.e. the node with value 1. In case of going back, we usually take the help of *prevNode* pointer. So we can go to the previous node i.e. the node with value 7 and then to the node with value 8 and so on. In this way, we can traverse the list from the end to start. We can move forward or backward in doubly-link list very easily. We have developed this facility for the users to move in the list easily.

Let's discuss other methods of the doubly-linked list. Suppose we have created a new node from the factory with value 9. We will request the node factory to create a new object using new keyword. The newly created node contains three fields i.e. *object*, *prevNode* and *nextNode*. We will store 9 into object and connect this new node in the chain. Let's see how the pointers are manipulated to do that. Consider the above diagram, the current is pointing at the node with value 6. The new node will be inserted between the node with value 6 and the one with value 8.

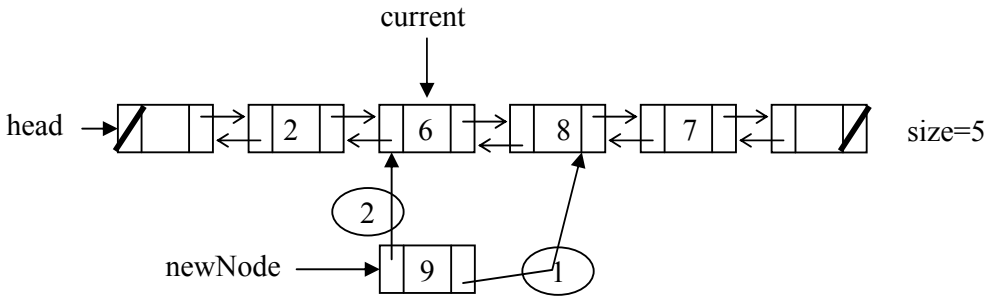
In the first step, we assign the address of the node with value 8 to the *nextNode* of the new node.

```
newNode->setNext( current->getNext() );
```



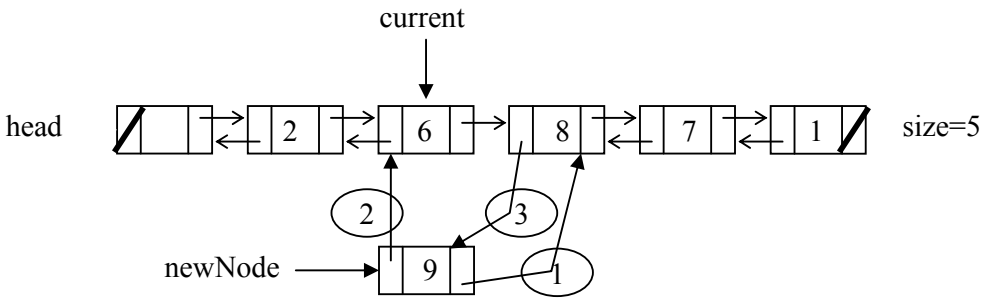
In the next step, a programmer points the *prevNode* of the *newNode* to the node with value 6.

```
newNode->setprev( current );
```



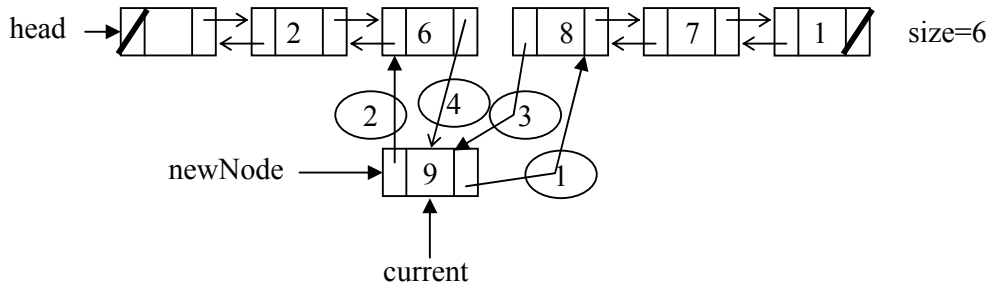
In the third step, we will set the previous node with value 8 to point to the *newNode*.

```
(current->getNext())->setPrev(newNode);
```



Now the *prevNode* of the node with value 8 is pointing to the node with value 9.
In the fourth step, the *nextNode* of the node with value 6 is pointing to the *newNode i.e.* the node with value 9. Point the *current* to the *newNode* and add one to the *size* of the list.

```
current->setNext( newNode );
current = newNode;
size++;
```



Now the newNode has been inserted between node with value 6 and node with value 8.

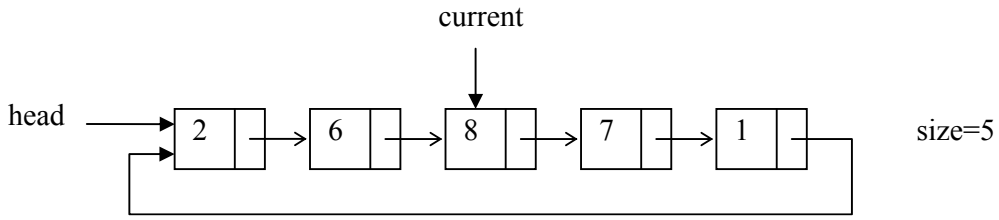
Circular Linked list is a linked list where the last node points back to the first, forming a loop

Circularly-linked lists

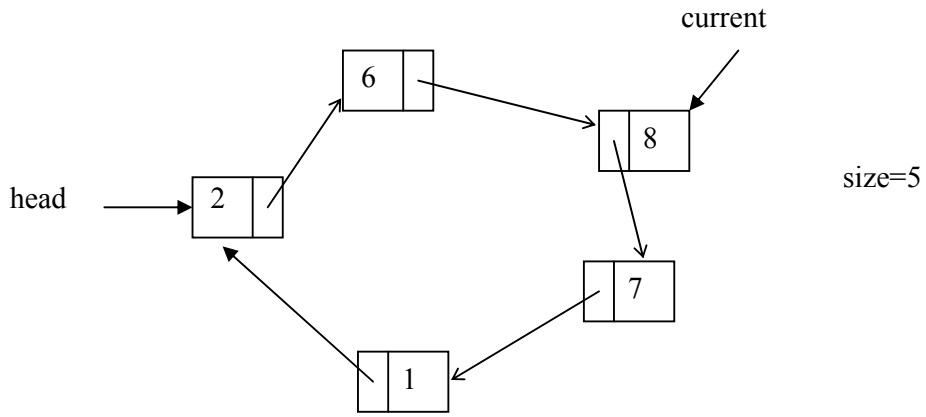
Let's talk about circularly linked list. The next field in the last node in a singly-linked list is set to NULL. The same is the case in the doubly-linked list. Moving along a singly-linked list has to be done in a watchful manner. Doubly-linked lists have two NULL pointers i.e. *prev* in the first node and *next* in the last node. A way around this potential hazard is to link the last node with the first node in the list to create a *circularly-linked list*.

The *next* method in the singly-linked list or doubly-linked list moves the *current* pointer to the next node and every time it checks whether the *next* pointer is NULL or not. Similarly the *back* method in the double-linked list has to be employed carefully if the current is pointing the first node. In this case, the *prev* pointer is pointing to NULL. If we do not take care of this, the current will be pointing to NULL. So if we try to access the NULL pointer, it will result in an error. To avoid this, we can make a circularly linked list.

We have a list with five elements. We have connected the last node with the first node. It means that the *next* of the last node is pointing towards the first node.



The same list has been shown in a circular shape.



You have noticed that there is no such node whose *next* field is NULL. What is the benefit of this? If you use the *next* or *back* methods that move the *current* pointer, it will never point to NULL. It may be the case that you keep on circulating in the list. To avoid this, we get help from the *head* node. If we move the *head* node in the circularly linked list, it will not be certain to say where it was pointing in the start. Its advantages depend on its use. If we do not have to move too much in the list and have no problem checking the NULL, there is little need a circularly-linked list. But this facility is available to us.

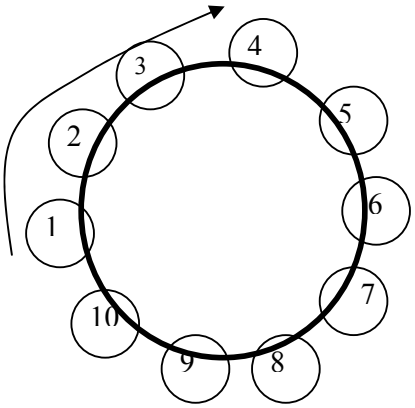
In this example, we made a circular linked list from a singly link list. In a singly link list we move in one direction. We point the *next* pointer of the last node to the first node. We can do the same with the doubly-linked list. The *prev* pointer of the first node will point to the last node and the *next* pointer of the last node will point to the first node. If you arrange all the nodes in a circle, one of the pointers (i.e. next pointer) will move in clockwise direction while the *prev* pointers in anti-clockwise direction. With the help of these pointers, you can move in the clockwise direction or anti-clockwise direction. Head node pointer will remain at its position. You don't need to change it. If there is a need to remove the node pointed by head node than you have to move the head pointer to other node. Now we don't have any NULL pointer in the doubly-linked list. We will not get any exception due to NULL pointers.

Josephus Problem

Now we will see an example where circular link list is very useful. This is Josephus Problem. Consider there are 10 persons. They would like to choose a leader. The way they decide is that all 10 sit in a circle. They start a count with person 1 and go in clockwise direction and skip 3. Person 4 reached is eliminated. The count starts with the fifth and the next person to go is the fourth in count. Eventually, a single person remains.

You might ask why someone has to choose a leader in this way. There are some historical stories attached to it. This problem is also studied in mathematics. Let's see its pictorial view.

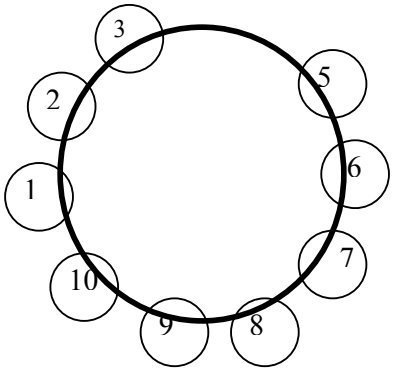
N=10, M=3



We have ten numbers representing the ten persons who are in a circle. The value of *M* shows the count. As the value of *M* is three, the count will be three. *N* represents the number of persons. Now we start counting clockwise. After counting up to three, we have the number four. The number four is eliminated and put in the eliminated column.

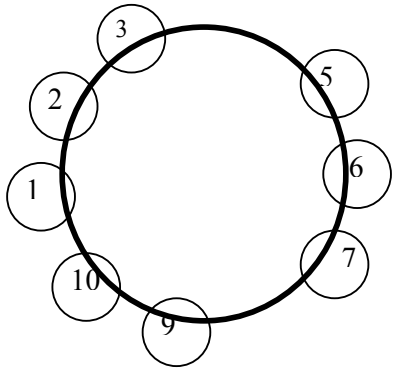
N=10, M=3

Eliminated

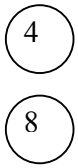


After eliminating the number four, we will start our counting from number five. Counting up to three, we have number eight which is eliminated and so on.

N=10, M=3

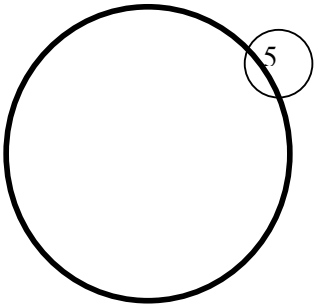


Eliminated

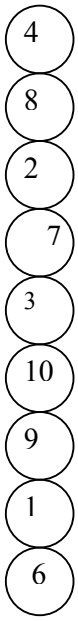


In the end, only number five will remain intact.

N=10, M=3



Eliminated



If we have ten persons (N = 10) in a circle and eliminate after counting up to three (M = 3). If we start our count from one, who will be the leader? We have studied this earlier and know that the person who is sitting at the fifth position will become the leader.

Suppose if the value of *N* is 300 or 400 and the value of *M* is 5 or 10. Now who will be the leader? This is a mathematical problem where we can change the values of *N* and *M*. There is a formula where the values of *N*, *M* are allotted. You can calculate who should become the leader. Here we will not solve it mathematically. Rather, it will be tackled as a computer problem. If you analyze the pictures shown above, it gets clear that this can be solved with the circular link list. We arrange these numbers in a circularly-linked list, point the *head* pointer at the starting number and after calling the *next* method for three times, we will reach the node which is to be removed. We will use the *remove* method to remove the node. Then the *next* method is called thrice from there and the node is removed. We will continue this till we have

only one node.

We are not concerned with the NULL pointers, internal to link list. However, if you want to solve this problem and choose the best data structure, then circular link list is the best option. We can also use the list to solve this.

Let's see the code of the program by which we can solve this problem. The code is as under:

```
/*This program solves the Josephus Problem */

#include <iostream.h>
#include "CList.cpp"  //contains the circularly-linked list definition

// The main method
void main(int argc, char *argv[])
{
        CList list;                 // creating an object of list
        int i, N=10, M=3;
        for(i=1; i <= N; i++ ) list.add(i); // initializing the list with values

        list.start();               // pointing the pointers at the start of the list

        // counting upto M times and removing the element
        while( list.length() > 1 ) {
           for(i=1; i <= M; i++ ) list.next();
                cout << "remove: " << list.get() << endl;
                list.remove();
        }

        cout << "leader is: " << list.get() << endl;     //displaying the remaining node
}
```

We have included the "CList.cpp". It means that we are using the circularly-linked list. In the main method, *CList* factory is called to create a circular link list as *CList list;* After this, we assign the values to *N* and *M*. We have used for loop to add the nodes in the list. When this loop finishes, we have ten nodes in the list having values from 1 to 10. But here a programmer may not pay attention to the internal details of the list. We have created a list and stored ten numbers in it. Then we moved the pointers of the list at the start of the list using the *start* method. It means that the pointers are pointing at the position from where we want to start the counting of the list.

There is a while loop that will continue executing until only one node is left in the list. Inside this loop, we have a for loop. It will execute from 1 to *M*. It has only one statement i.e. *list.next()*. This will move the pointer forward three times (as the value of M is 3). Now the current pointer is at the 4^{th} node. We called the *remove* method. Before removing the node, we display its value on the screen using *cout*. Again we come into the *while* loop, now the length of the list is 9. The 'for loop' will be executed. Now the *list.next()* is not starting from the start. It will start from the

position where the *current* pointer is pointing. The *current* pointer is pointing at the next node to the node deleted. The count will start again. The *list.next()* will be called for three times. The current pointer will point at the 8^{th} node. Again the remove method will be called and the *current* pointer moved to the next node and so on. The nodes will be deleted one by one until the length of the list is greater than one. When the length of the list is one, the while loop will be terminated. Now only one node is left in the list i.e. the leader. We will display its value using the *get* method.

We can change the values of *M* and *N*. Similarly, these values can be read from the file or can use the command line arguments to get values. There are many variations of this problem. One variation is that the value of *M* keeps on changing. Sometimes, it is 3, sometimes 4 or 5 and so on. Due to this, it will become difficult to think that who will become leader. Make a picture in your mind that ten persons are sitting in a circle. Every time the value of *M* is incremented by one. Now try to ascertain which position you should sit to get chosen as a leader. You may like to write a program to solve this or use the mathematical formula.

Data Structures

Lecture No. 05

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 3

3.1, 3.2.5, 3.3.1, 3.3.2
(array implementation)

Summary

5) Benefits of using circular list
6) Abstract Data Type
7) Stacks
8) Stack Implementation using arrays

In the previous lecture, we demonstrated the use of the circular list for the resolution of the Josephus problem. After writing a program with the help of this data structure, a leader among ten persons was selected. You must have noted many things while trying to solve the problem. These things will help us to understand the usage of data structures in C++, thus making the programming easy. The code of the program is given below.

```
#include "CList.cpp"
void main(int argc, char *argv[])
{
   CList list;
   int i, N=10, M=3;
   for(i=1; i <= N; i++ ) list.add(i);

   list.start();

   while( list.length() > 1 ) {
      for(i=1; i <= M; i++ ) list.next();

     cout << "remove: " << list.get() << endl;
     list.remove();
   }
   cout << "leader is: " << list.get() << endl;
}
```

In the program, we include the file of the class *CList and* create its object i.e. *list.* Then we solve the problem by using the *add, start, length, next, remove* and *get* methods of the class *CList*.

In the program, we have included already-defined data structure *CList*. After defining its different methods, we have an interface of *Clist*. There is no need to be worry about the nature of the list i.e. whether it is linked list, doubly linked list or an array. For us, it is only a list to be manipulated according to our requirement. You will see that a programmer may use different methods of the list object to solve the problem. We add elements to the list by a simple call of *add* method and go to the first element of the list by *start* method. Here, the length method is used in the condition of the *while* loop. Then we remove elements from the list and use the *next*, *get* and *remove* methods during this process. We get the current element by using the *get* method, then remove it by calling the *remove* method and then go to the next element by the method *next*. This way, all the elements are removed from the list except one element, called the leader. This one element remains there as we execute the while loop one less than the length of the list.

In singly linked list, the '*next'* returns false when it reaches to the last node due to the fact that the *next* field of the last node is set to NULL. But in a circularly linked list there is no NULL. It will be there only when there is no node in the list.

The whole process, which we carried out to solve the Josephus problem, can also be carried out with functions in C++. While adopting this way (of writing functions), we have to write these functions whenever we write another program that manipulates a list. In this method, we define a class of the data structure list and its different methods for the purpose of manipulation. This way, this class, obviously its methods too, can be used in any program where the manipulation of a list is needed. Thus there is re-usability of the code. In a class, we encapsulate the data and its methods. This shows that we are no longer interested in the internal process of the class. Rather, we simply use it wherever needed. The circular linked list, earlier used for the resolution of the Josephus problem, can also be employed in other problems. We have a class *CList* of this circular linked list through which any number of objects of data type of circular linked list can be created. Thus we can assume the class *CList* as a factory, creating as many objects of list as needed. This class and its objects in any program can be used to solve the problems with the help of its interface. The interface of this class consists of some methods like *add, remove, next, back, get* and some other simple ones. While carrying out programming, we will see that these classes (objects) help us very much to solve different problems.

Benefits of using circular list:

Efficient Traversal: Simplifies traversal, as the last element points back to the first, eliminating the need to check for NULL pointers.

Circular Operations: Useful for applications requiring continuous data processing, like round-robin scheduling or rotating buffers.

Memory Efficiency: Optimizes memory usage by reusing the tail's pointer to connect to the head, reducing overhead compared to doubly linked lists.

Implementation Simplicity: Easier to implement certain algorithms due to the circular nature, often requiring fewer special cases in code.

Benefits of using circular list

While solving the Josephus problem, it was witnessed that the usage of circular linked list helped us make the solution trivial. We had to just write a code of some lines that solved the whole problem. In the program, we included the class *CList* (which is of our data structure i.e. circular linked list) and used all of its methods according to the requirements. There was no problem regarding the working of the methods. We just called these methods and their definition in the class *CList* worked well.

Now we will see what happens if we solve the Josephus problem by using an array instead of the class in our program. In this case, we have to define an array and write code to move back and forth in the array and to remove different elements properly in a particular order. A programmer needs to be very careful while doing this, to reach the solution of the problem. Thus our code becomes very complex and difficult for

someone to understand and modify it. Moreover we cannot use this code in some other problem. Note that here we are talking about the use of an array in the main program, not in the class that defines the *CList* data structure. There is no need to be worried whether an array, singly linked list, doubly linked list is used or circular linked list being employed internally in implementing the list in defining the class of list data type. We only want that it should create objects of list. The usage of the class of a data structure simplifies the code of the program. We can also use this class wherever needed in other programs. This shows that the choice of appropriate data structures can simplify an algorithm. It can make the algorithm much faster and efficient. In this course, we will see that there are different data structures, which makes the algorithms very easy to solve our problems. Later, we will see how some elegant data structures lie at the heart of major algorithms. There is also a course dedicated to study different algorithms and recipes that can be used to solve host of complex problems. Moreover, we will study different data structures in detail and see that with the use of a proper data structure, we can solve a problem efficiently. A properly constructed data structure will always help in the solution of problems.

Abstract Data Type

A data type is a collection of values and a set of operations on those values. That collection and these operations form a mathematical construct that may be implemented with the use of a particular hardware or software data structure. The term abstract data type (ADT) refers to the basic mathematical concept that defines the data type. We have discussed four different implementations of the list data structure. In case of implementation of the list with the use of an array, the size of the array gives difficulty if increased. To avoid this, we allocate memory dynamically for nodes before connecting these nodes with the help of pointers. For this purpose, we made a singly linked list and connected it with the next pointer to make a chain. Moving forward is easy but going back is a difficult task. To overcome this problem, we made a doubly linked list using *prev* and *next* pointers. With the help of these pointers, we can move forward and backward very easily. Now we face another problem that the *prev* pointer of first node and the *next* pointer of the last node are NULL. Therefore, we have to be careful in case of NULL pointers. To remove the NULL pointers, we made the circular link list by connecting the first and last node.

The program employing the list data structure is not concerned with its implementation. We do not care how the list is being implemented whether through an array, singly linked list, doubly linked list or circular linked list. It has been witnessed that in these four implementations of the list, the interface remained the same i.e. it implements the same methods like *add, get, next, start* and *remove* etc. This proves that with this encapsulation attained by making a class, we are not concerned with its internal implementation. The implementation of these abstract data types can be changed anytime. These abstract data types are implemented using classes in C++. If the list is implemented using arrays while not fulfilling the requirements, we can change the list implementation. It can be implemented with the use of singly-link list or doubly link list. As long as the interface is same, a programmer can change the internal implementation of the list and the program using this list will not be affected at all. This is the abstract data type (ADT). What we care about is the methods that are available for use, with the List ADT i.e. *add*, *get*, and *remove* etc methods. We have not studied enough examples to understand all the

benefits of abstract data types. We will follow this theme while developing other ADT. We will publish the interface and keep the freedom to change the implementation of ADT without effecting users of the ADT. The C++ classes provide a programmer an ability to create such ADTs. What benefits can we get with the help of these ADTs and classes? When we develop an ADT or a class or factory then the users of this factory are independent of how this factory works internally. Suppose that we have ordered the car factory (car class) to produce a new car and it replies after a long time. If we ordered the remove method to remove one node and we are waiting and it keeps on working and working. Then we might think that its implementation is not correct. Although, we are not concerned with the internal implementation of this ADT yet it is necessary to see whether this ADT is useful for solving our problem or not. It should not become a bottleneck for us. If the method we are using is too much time consuming or it has some problem in terms of algorithm used. On one side, we only use the interfaces provided by these ADTs, classes, or factories as long as they do what they promise. We are not concerned with the internal details. On the other hand, we have to be careful that these factories or methods should not take too much time so that these will not be useful for the problem.

This distinction will always be there. Sometimes, the source code of classes is not provided. We will be provided libraries, as standard libraries are available with the compiler. These classes are in compiled form i.e. are in object form or in binary form. On opening these files, you will not see the C++ code, rather binary code. When you read the assembly language code, it will give some idea what this binary code is about. You can view the interface methods in the *.h* file. As an application programmer, you have to see that the ADTs being used are written in a better way. The point to be remembered here is that you should not worry about the internal implementation of these ADTs. If we want to change the internal implementation of the ADTs, it can be done without affecting the users of these ADTs. While writing a program, you should check its performance. If at some point, you feel that it is slow, check the ADTs used at that point. If some ADT is not working properly, you can ask the writer of the ADT to change the internal implementation of that ADT to ensure that it works properly.

Stacks

Let's talk about another important data structure. You must have a fair idea of stacks. Some examples of stacks in real life are stack of books, stack of plates etc. We can add new items at the top of the stack or remove them from the top. We can only access the elements of the stack at the top. Following is the definition of stacks.

"Stack is a collection of elements arranged in a linear order"

Let's see an example to understand this. Suppose we have some video cassettes. We took one cassette and put it on the table. We get another cassette and put it on the top of first cassette. Now there are two cassettes on the table- one at the top of other. Now we take the third cassette and stack it on the two. Take the fourth cassette and stack it on the three cassettes.

Now if we want to take the cassette, we can get the fourth cassette which is at the top and remove it from the stack. Now we can remove the third cassette from the stack

and so on. Suppose that we have fifty cassettes stacked on each other and want to access the first cassette that is at the bottom of the stack. What will happen? All the cassettes will fell down. It will not happen exactly the same in the computer. There may be some problem. It does not mean that our data structure is incorrect. As we see in the above example that the top most cassette will be removed first and the new cassette will be stacked at the top. The same example can be repeated with the books. In the daily life, we deal with the stacked goods very carefully.

Now we will discuss how to create a stack data structure or a factory, going to create stack object for us. What will be the attributes of this object? During the discussion on the list, we came to know that a programmer adds values in the list, removes values from the list and moves forward and backward. In case of a stack too, we want to add things and remove things. We will not move forward or backward in the stack. New items can be added or removed at the top only. We can not suggest the removal of the middle element of the stack.

Let's talk about the interface methods of the stacks. Some important methods are:

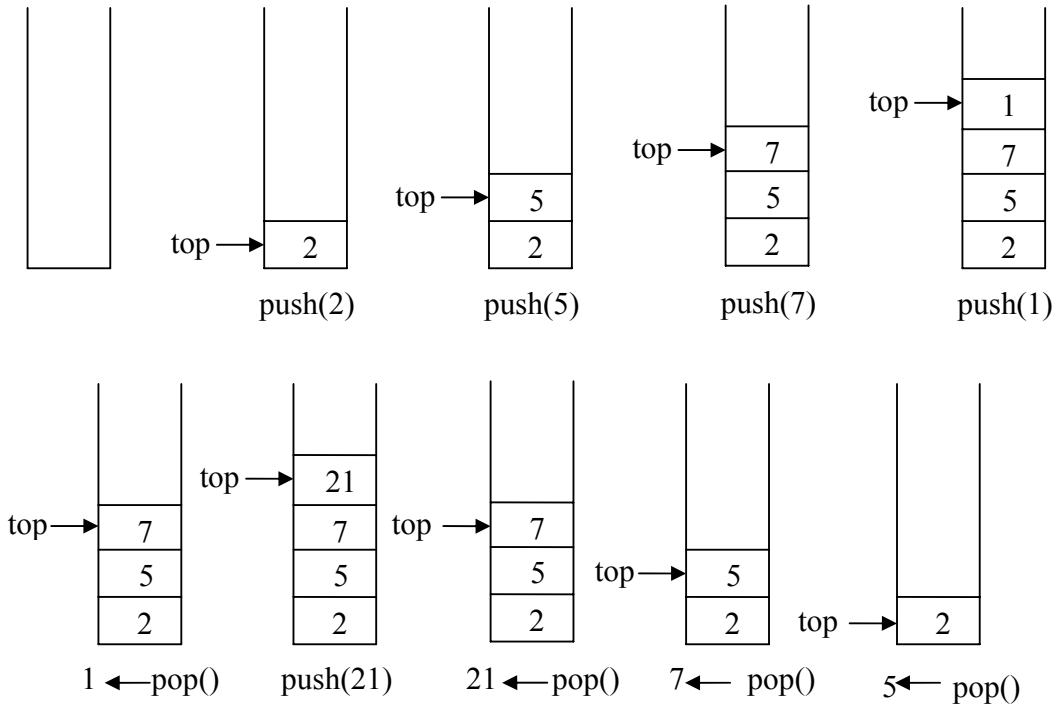
Method Name	Description
push(x)	Insert x as the top element of the stack
pop()	Remove the top element of the stack and return it.
top()	Return the top element without removing it from the stack.

The *push(x)* method will take an element and insert it at the top of the stack. This element will become top element. The *pop()* method will remove the top element of the stock and return it to the calling program. The *top()* method returns the top-most stack element but does not remove it from the stack. The interface method names that we choose has special objective. In case of list, we have used *add*, *remove*, *get*, *set* as the suitable names. However, for stack, we are using *push*, *pop* and *top*. We can depict the activity from the method name like push means that we are placing an element on the top of the stack and pushing the other elements down.

The example of a hotel's kitchen may help understand the concept of stacks in a comprehensive manner. In the kitchen, the plates are stacked in a cylinder having a spring on the bottom. When a waiter picks a plate, the spring moves up the other plates. This is a stack of plates. You will feel that you are pushing the plates in the cylinder and when you take a plate from the cylinder it pops the other plates. The top method is used to get the top- most element without removing it.

When you create classes, interfaces and methods, choose such names which depicts what these method are doing. These names should be suitable for that class or factory.

Let's discuss the working of stack with the help of a diagram.



At the start, the stack is empty. First of all, we push the value 2 in the stack. As a result, the number 2 is placed in the stack. We have a *top* pointer that points at the top element. Then we said *push(5)*. Now see how 2 and 5 are stacked. The number 5 is placed at the top of number 2 and the pointer *top* moves one step upward. Then we pushed the number 7 which is placed on the top and the number 2 and 5 are below. Similarly, we push number 1. The last figure in the first row shows the stacked values of the numbers- 1, 7, 5 and 2.

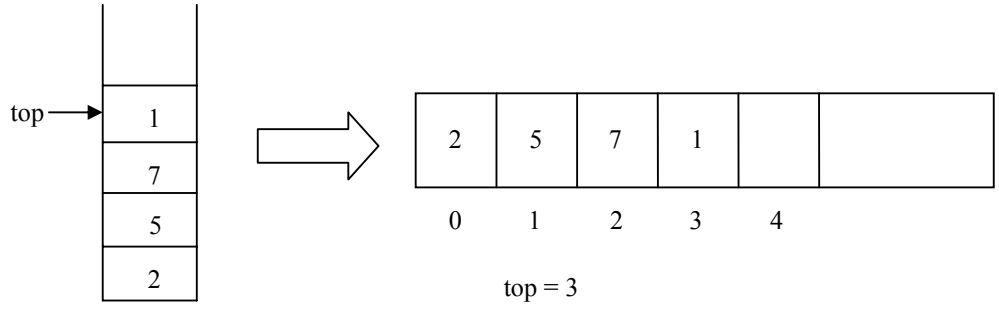
Let's pop the elements from the stack. The first figure of second row shows the pop operation. As a result, the number 1 is popped. Than again we push the number 21 on the stack. The number 7, 5, and 2 are already in the stack and number 21 is pushed at the top. If we pop now, the number 21 is popped. Now number 7 is at the top. If we pop again, the number 7 is popped. Pop again and the number 5 is popped and number 2 remains in the stack. Here with the help of this diagram, we are proving that the values are added at the top and removed at the top in a stack.

The last element to go into the stack is the first to come out. That is why, a stack is known as *LIFO* (Last In First Out) structure. We know that the last element pushed in the stack is at the top which is removed when we call pop. Let's see some other scenarios. What happens if we call *pop()* while there is no element? One possible way-out is that we have *isEmpty()* function that returns true if stack is empty and false otherwise. This is a boolean function that returns false if there is no element in the stack. Otherwise, it will return true. The second option is this that when we call pop on an empty stack, it throws an exception. This is a concept of advanced C++. Exception is also a way to convey that some unusual condition has arisen or something has gone wrong. Suppose, if we have a division method and try to divide some number with zero. This method will throw 'division by zero' exception.

Currently we will not throw an exception but use the *isEmpty()* method. The user who is employing the stack is responsible to call the *isEmpty()* method before calling the *pop*. Call the *pop* method if *isEmpty()* returns false . Otherwise, there will be a problem.

Stack Implementation using array

Let's discuss the implementation of the stack. Suppose we implement the stack using the arrays. The stack shown in the above diagram may be considered as an array. Here the array is shown vertically. We can implement the stack using array. The interface will remain as *push* and *pop* methods. The user of the stack does not need to know that the stack is internally implemented with the help of array. The worst case for insertion and deletion from an array may happen when we insert and delete from the beginning of the array. We have to shift elements to the right for insertion and left for removal of an element. We face the same problem while implementing the list with the use of the array. If we *push* and *pop* the elements from the start of the array for stack implementation, this problem will arise. In case of *push*, we have to shift the stack elements to the right. However, in case of *pop,* after removing the element, we have to shift the elements of stack that are in the array to the left. If we push the element at the end of the array, there is no need to shift any element. Similarly as the *pop* method removes the last element of the stack which is at the end of the array, no element is shifted. To insert and remove elements at the end of the array we need not to shift its elements. Best case for insert and delete is at the end of the array where there is no need to shift any element. We should implement *push()* and *pop()* by inserting and deleting at the end of an array.



In the above diagram, on the left side we have a stack. There are four elements in the stack i.e. 1, 7, 5 and 2. The element 1 is the extreme-most that means that it is inserted in the end whereas 7, 5, and 2 have been added before. As this is a LIFO structure so the element 1 should be popped first. On the right side we have an array with positions 0, 1, 2, 3 and so on. We have inserted the numbers 2, 5, 7 and 1. We have decided that the elements should be inserted at the end of the array. Therefore the most recent element i.e. 1 is at position 3. The *top* is the index representing the position of the most recent element. Now we will discuss the stack implementation in detail using array.

We have to choose a maximum size for the array. It is possible that the array may 'fill-up' if we push enough elements. Now more elements cannot be pushed. Now what should the user of the stack do? Internally, we have implemented the stack using array which can be full. To avoid this, we write *isFull()* method that will return

a boolean value. If this method returns true, it means that the stack (array) is full and no more elements can be inserted. Therefore before calling the *push(x)*, the user should call *isFull()* method. If *isFull()* returns false, it will depict that stack is not full and an element can be inserted. This method has become the part of the stack interface. So we have two more methods in our interface i.e. *isEmpty()* and *isFull()*.

Now we will discuss the actual C++ code of these operations. These methods are part of stack class or stack factory. We have an array named *A* while *current* is its index. The code of *pop()* method is as:

```
int pop()
{
   return A[current--];
}
```

In this method, the recent element is returned to the caller, reducing the size of the array by 1.

The code of *push* method is:

```
void push(int x)
{
   A[++current] = x;
}
```

We know that ++*current* means that add one to the *current* and then use it. That also shows that element *x* should be inserted at *current* plus one position. Here we are not testing that this *current* index has increased from the array size or not. As discussed earlier that before using the *push* method, the user must call *isFull()* method. Similarly it is the responsibility of the user to call the *isEmpty()* method before calling the *pop* method. Therefore there is no *if* statement in the *push* and *pop* method.

The code of the *top()* method is:

```
int top()
{
   return A[current];
}
```

This method returns the element at the current position. We are not changing the value of *current* here. We simply want to return the top element.

```
int isEmpty()
{
   return ( current == -1 );
}
```

This method also tests the value of the *current* whether it is equal to -1 or not. Initially when the stack is created, the value of *current* will be -1. If the user calls the *isEmpty()* method before pushing any element, it will return true.

```
int isFull()
{
   return ( current == size-1);
}
```

This method checks that the stack is full or not. The variable *size* shows the size of the array. If the *current* is equal to the *size* minus one, it means that the stack is full and we cannot insert any element in it.

We have determined the cost and benefit of all the data structures. Now we will see how much time these methods take. A quick examination shows that all the five operations take constant time. In case of list, the *find* method takes too much time as it has to traverse the list. Whereas the *add* and *remove* methods are relatively quick. The methods of stack are very simple. There is no complexity involved. We insert element at one side and also remove from that side not in the middle or some other place. Therefore we need not to carry out a lot of work. During the usage of the array, the stack methods *push*, *pop*, *top*, *isFull* and *isEmpty* all are constant time operations. There is not much difference of time between them.

The complete code of the program is:

```
/* Stack implementation using array */

#include <iostream.h>

/* The Stack class */

class  Stack
{
   public:
      Stack() { size = 10; current = -1;}          //constructor
      int pop(){ return A[current--];}             // The pop function
      void push(int x){A[++current] = x;}          // The push function
      int top(){ return A[current];}               // The top function
      int isEmpty(){return ( current == -1 );}     // Will return true when stack is empty
      int isFull(){ return ( current == size-1);} // Will return true when stack is full

    private:
      int   object;                                // The data element
      int   current;                               // Index of the array
      int   size;                                  // max size of the array
      int   A[10];                                 // Array of 10 elements
};

// The main method
int main()
{
   Stack stack;                                    // creating a stack object
   // pushing the 10 elements to the stack
   for(int i = 0; i < 12; i++)
```

```
    {
        if(!stack.isFull())                    // checking stack is full or not
                stack.push(i);                 // push the element at the top
        else
                cout <<"\n Stack is full, can't insert new element";
    }

    // pop the elements at the stack
    for (int i = 0; i < 12; i++)
    {
        if(!stack.isEmpty())                   // checking stack is empty or not
                cout << "\n The popped element = " << stack.pop();
        else
                cout <<"\n Stack is empty, can't pop";
    }
}
```

The output of the program is:

```
Stack is full, can't insert new element
Stack is full, can't insert new element
The popped element = 9
The popped element = 8
The popped element = 7
The popped element = 6
The popped element = 5
The popped element = 4
The popped element = 3
The popped element = 2
The popped element = 1
The popped element = 0
Stack is empty, can't pop
Stack is empty, can't pop
```

However, a programmer finds the size-related problems in case of an array. What should we do when the array is full? We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements. Further discussion on this issue will be made in the next lecture.

Data Structures

Lecture No. 06

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 3
3.3.2, 3.3.3
(Postfix expressions)

Summary

- Stack From the Previous Lecture
- Stack Using Linked List
- Stack Implementation: Array or Linked List
- Use of Stack
- Precedence of Operators
- Examples of Infix to Postfix

Stack From the Previous Lecture

We started discussing *Stack* data structure and its implementation in the previous lecture. We also implemented stack structure using an array and wrote code for its *push(), pop()* and *top()* operations. We realized that we have to specify the size of the array before using it whether we declare it statically or dynamically. Arrays are of fixed size and when they become full, no more elements can be added to them. In order to get to know that the array has gone full, we wrote the *isFull()* method. It became the responsibility of the user of the stack structure to call *isFull()* method before trying to insert an element using the *push()* method otherwise the whole program could crash.

isEmpty() method is implemented as a *stack* can be empty like a *list* or *set* structures.

It is important to understand that *isFull()* method is there in stack implementation because of limitation of array but *isEmpty()* method is part of the stack characteristics or functionality.

As previously in the implementation of list structure, we used linked list while allocating nodes dynamicallyin order to avoid the fixed sized limitation of array. Now in this case also, again to overcome the limitation of array, we are going to make use of linked list in place of array to implement the stack data structure. Let's see, how we can implement a stack structure using linked list and how the implementation code will look like internally.

Stack Using Linked List

We can avoid the size limitation of a stack implemented with an array, with the help of a linked list to hold the stack elements.

As needed in case of array, we have to decide where to insert elements in the list and where to delete them so that *push* and *pop* will run at the fastest.

Primarily, there are two operations of a stack; *push()* and *pop()*. A stack carries *lifo* behavior i.e. last in, first out.

You know that while implementing stack with an array and to achieve *lifo* behavior, we used *push* and *pop* elements at the end of the array. Instead of pushing and popping elements at the beginning of the array that contains overhead of shifting elements towards right to *push* an element at the start and shifting elements towards left to *pop* an element from the start. To avoid this overhead of shifting left and right, we decided to *push* and *pop* elements at the end of the array.

Now, if we use linked list to implement the stack, where will we *push* the element inside the list and from where will we *pop* the element? There are few facts to consider, before we make any decision:

- For a *singly-linked list*, insert at start or end takes constant time using the *head* and *current* pointers respectively. As far as insertion is concerned, it is workable and equally efficient at the start and end.
- Removing an element at the start is constant time but removal at the end requires traversing the list to the node one before the last. So removing from the start is better approach rather than from the end.

Therefore, it makes sense to place stack elements at the start of the list because insertion and removal take constant time. As we don't need to move back and forth within the list, therefore, there is no requirement of *doubly* or *circular linked list. Singly linked list* can serve the purpose. Hence, the decision is to insert the element at the start in the implementation of *push* operation and remove the element from the start in the *pop* implementation.

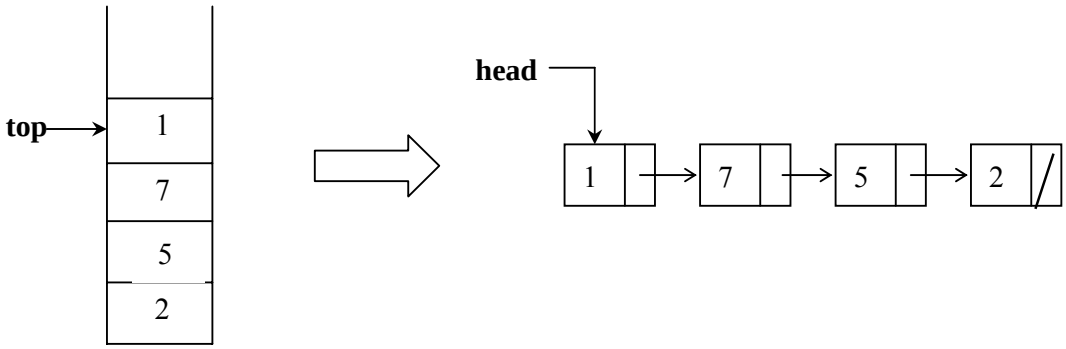


Fig 1. Stack using array (on left side) and linked list (on right side)

There are two parts of above figure.On the left hand, there is the stack implemented using an array. The elements present inside this stack are *1*, *7*, *5* and *2*. The most recent element of the stack is *1*. It may be removed if the *pop()* is called at this point of time. On the right side, there is the stack implemented using a linked list. This

stack has four nodes inside it which are liked in such a fashion that the very first node pointed by the *head* pointer contains the value *1*. This first node with value *1* is pointing to the node with value *7*. The node with value *7* is pointing to the node with value *5* while the node with value 5 is pointing to the last node with value 2. To make a stack data strcuture using a linked list, we have inserted new nodes at the start of the linked list.

Let's see the code below to implement *pop()* method of the stack.

Steps To remember while poping:

1- Get The Value of head and store it in a variable.
2- Make a temporary pointer and equal it to head.
3- point head to the next element .
4- delete the temporary pointer .

```
int  pop()
{
1.  int x = head->get();
2.  Node * p = head;
3.  head = head->getNext();
4.  delete p;
5.  return x;
}
```

At line 1, we have declared *x* as an *int* and retrieved one element from the node of the stack that is pointed by the *head* pointer. Remember, the *Node* class and its get() method that returns the value inside the node.
At line 2, *p* is declared as a pointer of type *Node* and address inside the *head* pointer is being saved inside this *p* pointer.
At line 3, the address of the next node is being retrieved with the help of the *getNext()* method of the *Node* class and being assigned to *head* pointer. After this assignment, the *head* pointer has moved forward and started pointing to the next element in the stack.
At line 4, the node object pointed by the pointer *p* is being deallocated (deleted).
At line 5, the function is returning the value of the node retrieved in step 1.

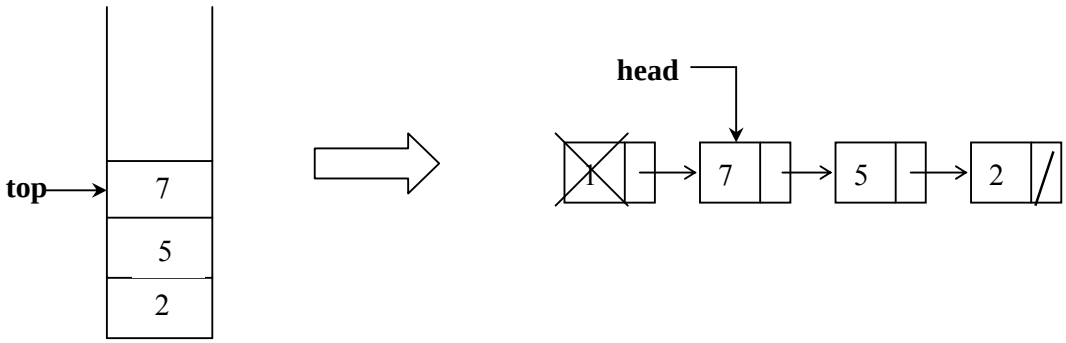


Fig 2. A node removed from the stack after the pop() call

Let's see the code of the *push()* method of the stack:

```
void  push(int x)
{
1.  Node * newNode = new Node();
2.  newNode->set(x);
```

```
3.  newNode->setNext(head);
4.  head  =  newNode;
}
```

In line 1, a new node is created, using the *new Node()* statement and returned pointer is assigned to a pointer *newNode*. So *newNode* starts pointing to the newly created *Node* object.
In line 2, the value *2* is set into the newly created *Node* object.
In line 3, the next node of the newly created node is set to the node pointed to by the *head* pointer using *setNext(head)*.
In line 4, the *head* pointer is made to point to the newly created node.

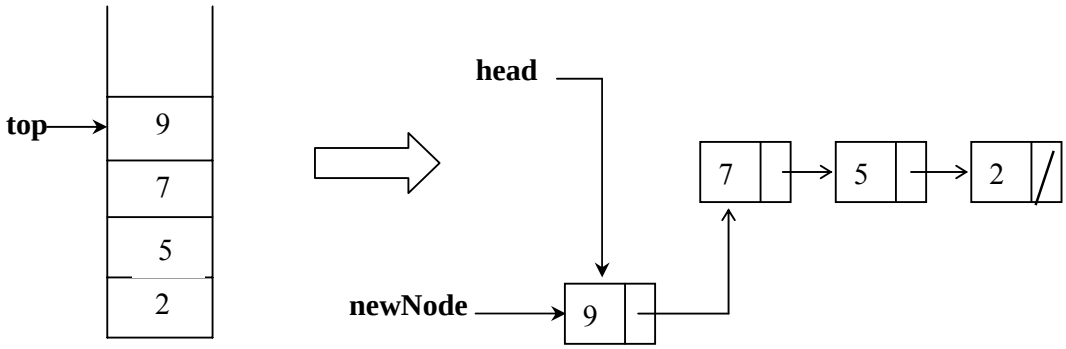


Fig 3. A node added to the stack after the push(9) call

These are two primary methods of a stack. By using the *push()* method, we can keep on pushing elements and using the *pop()* methods. Elements can be removed from the stack till the time, it gets empty. As discussed earlier, *isEmpty()* is the stack characteristic but *isFull()* was implemented because of the size limitation of the array. We are no more using array to implement a stack. Rather, we have used linked list here for stack implementation. Therefore, *isFull()* might not be required here. An interesting question arises here. Can we add infinite elements to the stack now. We should remember that this program of stack will run on computer that definitely has a limited memory. Memory or *Address space* of a computer is the space (physical memory and disk space) that can be addressed by the computer which is limited inlcuding the limited physical memory. Disk space is used as the *virtual memory (* we will not discuss *virtual memory* in detail here). A computer with 32-bit addressing can address upto $2^{32}-1$ memory locations and similarly a computer with 64-bit addressing can address upto $2^{64}-1$ addresses. If this address space becomes full, the stack will definitely be full. However, the stack implementation is not liable for this fullness of address space and it is the limitation of a computer address space. Therefore, we don't need to call *isFull()* before *push*ing the element. Rather, *isEmpty()* is called before *pop*ing an element from the stack.

Let's see the remaining methods of the stack while using linked list to implement it.

```
int  top()
{
```

```
    return   head->get();
}
int   isEmpty()
{
    return   ( head   ==   NULL );
}
```

The above-mentioned methods i.e. *top()* and *isEmpty()* are very simple functions. One statement inside the *top()* is retrieving the top element (pointed to by the *head* pointer) from the stack and returning it back by value. It is important to note that *top()* is not removing the element from the stack, but only retrieving it. The one statement inside *isEmpty()* is a check to see if the *head* pointer is not pointing to any node and it is *NULL*. If the *head* pointer is *NULL* that means the stack is empty, the method returns *true* otherwise it returns *false*.

All four operations *push()*, *pop()*, *top()* and *isEmpty()* take constant time. These are very simple methods and don't contain loops. They are also not CPU hungry operation. Also note that we have not written *isFull()* while implementing stack with the linked list.

Stack Implementation: Array or Linked List

Since both implementations support stack operations in constant time, we will see what are the possible reasons to prefer one implementation to the other.

- Allocating and de-allocating memory for list nodes does take more time than pre-allocated array. Memory allocation and de-allocation has cost in terms of time, especially, when your system is huge and handling a volume of requests. While comparing the stack implementation, using an array versus a linked list, it becomes important to consider this point carefully.
- List uses as much memory as required by the nodes. In contrast, array requires allocation ahead of time. In the previous bullet, the point was the time required for allocation and de-allocation of nodes at runtime as compared to one time allocation of an array. In this bullet, we are of the view that with this runtime allocation and de-allocation of nodes, we are also getting an advantage that list consumes only as much memory as required by the nodes of list. Instead of allocating a whole chunk of memory at one time as in case of array, we only allocate memory that is actually required so that the memory is available for other programs. For example, in case of implementing stack using array, you allocated array for 1000 elements but the stack, on average, are using 50 locations. So, on the average, 950 locations remain vacant. Therefore, in order to resolve this problem, linked list is handy.
- List pointers (*head, next*) require extra memory. Consider the manipulation of array elements. We can set and get the individual elements with the use of the array index; we don't need to have additional elements or pointers to access them. But in case of linked list, within each node of the list, we have one pointer element called *next*, pointing to the next node of the list. Therefore, for 1000 nodes stack implemented using list, there will be 1000 extra pointer variables. Remember that stack is implemented using 'singly-linked' list. Otherwise, for doubly linked list, this overhead is also doubled as two pointer variables are stored within each node in that case.
- Array has an upper limit whereas list is limited by dynamic memory allocation. In

other words, the linked list is only limited by the address space of the machine. We have already discussed this point at reasonable length in this lecture.

Use of Stack

Examples of uses of stack include- traversing and evaluating prefix, *infix* and postfix expressions.

Consider the expression *A+B*: we think of applying the *operator* "+" to the *operands* A and B. We have been writing this kind of expressions right from our primary classes. There are few important things to consider here:

Firstly, + *operator* requires two operators or in other words "+" is a *binary operator*.

Secondly, in the expression *A+B*, the one operand *A* is on left of the operator while the other operand *B* is on the right side. This kind of expressions where the operator is present between two operands called *infix* expressions. We take the meanings of this expression as to add both operands *A* and *B*.

There are two other ways of writing expressions:

– We could write *+AB*, the operator is written before the operands *A* and *B*. These kinds of expressions are called *Prefix* Expressions.
– We can also write it as *AB+*, the operator is written after the operands *A* and *B*. This expression is called *Postfix* expression.

The prefixes *pre* and *post* refer to the position of the operator with respect to the two operands.

Consider another expression in *infix* form: *A + B * C*. It consists of three operands *A, B, C* and two operator *+,** . We know that multiplication () is done before addition (+), therefore, this expression is actually interpreted as: *A + (B * C)*. The interpretation is because of the precedence of multiplication (*) over addition (+). The precedence can be changed in an expression by using the parenthesis. We will discuss it a bit later.

Let's see, how can we convert the infix expression *A + (B * C)* into the postfix form. Firstly, we will convert the multiplication to postfix form as: *A + (B C *)*. Secondly, we will convert addition to postfix as: *A (B C *) +* and finally it will lead to the resultant postfix expression i.e. : *A B C * +*. Let's convert the expression *(A + B) * C* to postfix. You might have noticed that to overcome the precedence of multiplication operator (*) we have used parenthesis around *A + B* because we want to perform addition operation first before multiplication.

*(A + B) * C*	*infix form*
*(A B +) * C*	*convert addition*
*(A B +) C **	*convert multiplication*
*A B + C **	*postfix form*

These expressions may seem to be difficult to understand and evaluate at first. But this is one way of writing and evaluating expressions. As we are normally used to infix form, this postfix form might be little confusing. If a programmer knows the algorithm, there is nothing complicated and even one can evaluate the expression manually.

Precedence of Operators

There are five binary operators, called *addition*, *subtraction*, *multiplication*, *division* and *exponentiation*. We are aware of some other binary operators. For example, all relational operators are binary ones. There are some unary operators as well. These require only one operand e.g. – and +. There are rules or order of execution of operators in Mathematics called precedence. Firstly, the *exponentiation* operation is executed, followed by multiplication/division and at the end addition/subtraction is done. The order of precedence is (highest to lowest):

Exponentiation	↑
Multiplication/division	*, /
Addition/subtraction	+, -

For operators of same precedence, the left-to-right rule applies:

A+B+C means (A+B)+C.

For exponentiation, the right-to-left rule applies:

A ↑ B ↑ C means A ↑ (B ↑ C)

We want to understand these precedence of operators and infix and postfix forms of expressions. A programmer can solve a problem where the program will be aware of the precedence rules and convert the expression from infix to postfix based on the precedence rules.

Examples of Infix to Postfix

Let's consider few examples to elaborate the *infix* and *postfix* forms of expressions based on their precedence order:

Infix	**Postfix**
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

In the next lecture we will see, how to convert *infix* to *postfix* and how to evaluate *postfix* form besides the ways to use stack for these operations.

Data Structures

Lecture No. 07

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 3

3.3.3

Summary

9) Evaluating postfix expressions
10) An example
11) Infix to postfix Conversion

Evaluating postfix expressions

In the previous lecture, we talked about 'infix and postfix expression' and tried to understand how to write the postfix notation of mathematical expressions. A programmer can write the operators either after the operands i.e. postfix notation or before the operands i.e. prefix notation. Some of the examples are as under:

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

The last expression seems a bit confusing but may prove simple by following the rules in letter and spirit. In the postfix form, parentheses are not used. Consider the infix expressions as '4+3*5' and '(4+3)*5'. The parentheses are not needed in the first but are necessary in the second expression. The postfix forms are:

4+3*5 435*+
(4+3)*5 43+5*

In case of not using the parenthesis in the infix form, you have to see the precedence rule before evaluating the expression. In the above example, if we want to add first then we have to use the parenthesis. In the postfix form, we do not need to use parenthesis. The position of operators and operands in the expression makes it clear in which order we have to do the multiplication and addition.

Now we will see how the infix expression can be evaluated. Suppose we have a postfix expression. How can we evaluate it? Each operator in a postfix expression refers to the previous two operands. As the operators are binary (we are not talking about unary operators here), so two operands are needed for each operator. The nature

of these operators is not affected in the postfix form i.e. the plus operator (+) will apply on two operands. Each time we read an operand, we will push it on the stack. We are going to evaluate the postfix expression with the help of stack. After reaching an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack. Now we will see an example to comprehend the working of stack for the evaluation of the postfix form. Here is the algorithm in pseudo code form. After reading this code, you will understand the algorithm.

```
Stack s;                                    // declare a stack
while( not end of input ) {                 // not end of postfix expression
        e = get next element of input
        if( e is an operand )
                s.push( e );
        else {
                op2 = s.pop();
                op1 = s.pop();
                value = result of applying operator 'e' to op1 and op2;
                s.push( value );
        }
}
finalresult = s.pop();
```

We have declared a Stack's'. There is a 'while loop' along with 'not end of input' condition. Here the input is our postfix expression. You can get the expression from the keyboard and use the enter key to finish the expression. In the next statement, we get the next element and store it in '*e*'. This element can be operator or operand. The operand needs not to be single digit. It may be of two digits or even more like 60 or 234 etc. The complete number is stored in the '*e*'. Then we have an 'if statement' to check whether '*e*' is an operand or not. If '*e*' is an operand than we wrote *s.push(e)* i.e. we pushed the '*e*' onto the stack. If '*e*' is not the operand, it may be an operator. Therefore we will pop the two elements and apply that operator. We pop the stack and store the operand in '*op2*'. We pop the stack again and store the element in '*op1*'. Then the operator in 'e' is applied to '*op1*' and '*op2*' before storing the result in *value*. In the end, we push the '*value*' on the stack. After exiting the loop, a programmer may have only one element in the stack. We pop this element which is the final result.

Consider the example of 4+3*2 having a postfix form of 432*+. Here 4, 3, and 2 are operands whereas + and * are operators. We will push the numbers 4, 3 and 2 on the stack before getting the operator *. Two operands will be popped from the stack and * is being applied on these. As stack is a LIFO structure, so we get 2 first and then 3 as a result of pop. So 2 is store in '*op1*' and 3 in '*op2*'. Let's have a look on the program again. On applying * on these, we will push the result (i.e. 6) on the stack. The 'while loop' will be executed again. In case of getting the next input as operand, we will push it on the stack otherwise we will pop the two operands and apply the operator on these. Here the next element is the operator +. So two operands will be popped from the stack i.e. 6 and 4. We will apply the operator plus on these and push the result (i.e. 10) on the stack. The input is finished. Now we will pop the stack to get the final

result i.e. 10.

An Example

In the earlier example, we have used the stack to solve the postfix expression. Let's see another comprehensive example. The postfix expression is:

6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

We want to evaluate this long expression using stack. Let's try to solve it on paper. We have five columns here i.e. input, op1, op2, value and stack. We will run our pseudo code program. In the start, we read the input as a result we get number 6. As 6 is operand, so it will be pushed on the stack. Then we have number 2 which will also be pushed on the stack. Now 2 is the most recent element. The next element is the number 3 that will also be pushed on the stack. Now, there are three elements on the stack i.e. 3, 2 and 6. The number 3 is the most recent. On popping, we will get the number 3 first of all. The next element is '+', an operator. Now the else part of our pseudo code is executed. We will pop two operands from the stack and apply the operator (+) on these. The number 3 will be stored in variable *op2* and number 2 in *op1*. The operator (+) will be applied on these i.e. 2+3 and the result is stored in *value*. Now we will push the *value* (i.e. 5) on the stack. Now we have two numbers on the stack i.e. 5 and 6. The number 5 is the most recent element. The next element is '-'. As it is also an operator, so we will pop the two elements from the stack i.e. 5 and 6. Now we have 5 in op2 and 6 in op1. On applying the operator (-), we will get the result as 1 (6-5). We can't say op2 - op1. The result (1) will be pushed on stack. Now on the stack, we have only one element i.e. 1. Next three elements are operands so we pushed 3, 8 and 2 on the stack. The most recent element is 2. The next input is an operator in the expression i.e. '/', we will pop two elements from the stack. The number 2 will be stored in *op2* while number 8 in *op1*. We apply the operator (/) on the *op1* and *op2* i.e. (op1/op2), the result is 4 (i.e. 8/2). We push the result on the stack. We have, now, three elements i.e. 4, 3, and 1 on the stack. The next element is operator plus (+). We will pop the two elements i.e. 4 and 3 and will apply the operator (+). The result (7) will be pushed on the stack. The next input element is operator multiply (*). We will pop the two elements i.e. 7 and 1 and the result (7*1 = 7) is pushed on the stack. You have noted that whenever we have an operator in the input expression, we have two or more elements on the stack. As the operators we are using are binary and we need two operands for them. It will never be the case that you want to pop two elements from the stack and there is only one or no element on the stack. If this happens than it means there is an error in the program and you have popped more values than required. The next input element is 2 that is pushed on the stack. We have, now, the operator (↑) in the input. So we will pop the two elements, *op2* will hold 2 and *op1* will have the number 7. The operator (↑) will be applied on the operands i.e. (7 ↑ 2) and the result (49) is pushed on the stack. We have, now, the number 3 in the element being pushed on the stack. The last element is the operator plus (+). So we pop the two elements i.e. 49 and 2 and apply the operator on these. The result (49+3 = 52) is pushed on the stack. The input expression is finished, resulting in the final result i.e. 52.

This the tabular form of the evaluation of the postfix expression.

Input	op1	op2	value	stack
6				6
2				2 6
3				3 2 6
+	2	3	5	5 6
-	6	5	1	1
3	6	5	1	3 1
8	6	5	1	8 3 1
2	6	5	1	2 8 3 1
/	8	2	4	4 3 1
+	3	4	7	7 1
*	1	7	7	7
2	1	7	7	2 7
↑	7	2	49	49
3	7	2	49	3 49
+	49	3	52	52

With the help of stack we can easily solve a very big postfix expression. Suppose you want to make a calculator that is a part of some application e.g. some spreadsheet program. This calculator will be used to evaluate expressions. You may want to calculate the value of a cell after evaluating different cells. Evaluation of the infix form programmatically is difficult but it can be done. We will see another data structure which being used to solve the expressions in infix form. Currently, we have to evaluate the values in different cells and put this value in another cell. How can we do that? We will make the postfix form of the expression associated with that cell. Then we can apply the above algorithm to solve the postfix expression and the final result will be placed at that cell. This is one of the usages of the stack.

Infix to postfix Conversion

We have seen how to evaluate the postfix expressions while using the stack. How can we convert the infix expression into postfix form? Consider the example of a spreadsheet. We have to evaluate expressions. The users of this spreadsheet will employ the infix form of expressions. Consider the infix expressions ‘A+B*C’ and ‘(A+B)*C’. The postfix versions are ‘ABC*+’ and ‘AB+C*’ respectively. The order

of operands in postfix is the same as that in the infix. In both the infix expressions, we have the order of operands as A, B and then C. In the postfix expressions too, the order is the same i.e. A, B, followed by C. The order of operands is not changed in postfix form. However, the order of operators may be changed. In the first expression 'A+B*C', the postfix expression is 'ABC*+'. In the postfix form multiplication comes before the plus operator. In scanning from left to right, the operand 'A' can be inserted into postfix expression. First rule of algorithm is that if we find the operand in the infix form, put it in the postfix form. The rules for operators are different. The '+' cannot be inserted in the postfix expression until its second operand has been scanned and inserted. Keep the expression A+B*C in your mind. What is the second operand of the plus? The first operand is A and the second operand is the result of B*C. The '+' has to wait until the '*' has not been performed. You do the same thing while using the calculator. First you will multiply the B*C and then add A into the result. The '+' has to be stored away until its proper position is found. When 'B' is seen, it is immediately inserted into the postfix expression. As 'B' is the operand, we will send the operand to the postfix form. Can the '+' be inserted now? In case of 'A+B*C', we cannot insert '+' because '*' has precedence. To perform multiplication, we need the second operand. The first operand of multiplication is 'B' while the second one is 'C'. So at first, we will perform the multiplication before adding result to 'A'.

In case of '(A+B)*C', the closing parenthesis indicates that '+' must be performed first. After sending the A and B to postfix perform, we can perform the addition due to the presence of the parenthesis. Then C will be sent to the postfix expression. It will be followed by the multiplication of the C and the result of A + B. The postfix form of this expression is AB+C*. Sometimes, we have two operators and need to decide which to apply first like in this case '+' and '*'. In this case, we have to see which operator has higher precedence. Assume that we have a function '*prcd(op1,op2)*' where *op1* and *op2* are two operators. The function '*prcd(op1,op2)*' will return TRUE if *op1* has precedence over *op2*, FASLE otherwise. Suppose we call this function with the arguments '*' and '+' i.e. prcd(*, +), it will return true. It will also return true in case both *op1* and *op2* are '+' e.g. if we have A+B+C, then it does not matter which + we perform first. The call *prcd(+ , *)* will return false as the precedence of * is higher than the + operator. The '+' has to wait until * is performed.

Now we will try to form an algorithm to convert infix form into postfix form. For this purpose, a pseudo code will be written. We will also write the loops and if conditions. The pseudo code is independent of languages. We will be using a stack in this algorithm. Here, the infix expression is in the form of a string. The algorithm is as follows:

```
Stack s;
while( not end of input ) {
   c = next input character;
   if( c is an operand )
      add c to postfix string;
   else {
      while( !s.empty() && prcd(s.top(),c) ){
         op = s.pop();
         add op to the postfix string;
```

```
        }
        s.push( c );
    }
    while( !s.empty() ) {
       op = s.pop();
       add op to postfix string;
    }
```

First we will declare a stack '*s'*. The 'while loop' will continue till the end of input. We read the input character and store it in the '*c*'. Here the input character does not mean one character, but an operand or an operator. Then we have a conditional if statement. If '*c*' is an operand, then we will have to add it to postfix string. Whenever we get an operand in the infix form, it will be added to the postfix form. The order of operands does not change in the conversion. However, in this case, the order of operators may change. If '*c*' is the operator, then we will, at first, check that stack is not empty besides identifying the precedence of the operators between the input operator and the operator that is at the top of the stack. In case of the precedence of the operator that is on the stack is higher, we will pop it from the stack and send to the postfix string. For example if we have * on the stack and the new input operator is +. As the precedence of the + operator is less than the * operator, the operands of the multiplication has already been sent to the postfix expression. Now, we should send the * operator to the postfix form. The plus operator (+) will wait. When the while loop sends all such operators to the postfix string, it will push the new operator to the stack that is in '*c*'. It has to wait till we get the second operand. Then we will again get the input. On the completion of the input, the while loop will be finished. There may be a case that input may be completed even at the time when there are still some elements on the stack. These are operators. To check this, we have another while loop. This loop checks if the stack is not empty, pops the operator and put it in the postfix string. Let's take a look at a comprehensive example to understand it. In case of the infix expression, A + B * C, we have three columns, one each for input symbol, the postfix expression and the stack respectively. Now let's execute the pseudo code. First of all, we get the 'A' as input. It is an operand so we put it on the postfix string. The next input is the plus operator (+) which will be pushed on the stack. As it is an operator and we need two operands for it. On having a look at the expression, you might have figure out that the second operand for the plus operator is B*C. The next input is the operand B being sent to the postfix expression form. The next thing we get is the input element as '*'. We know that the precedence of * is higher than that of the +. Let's see how we can do that according to our pseudo code. The *prcd(s.top(), op)* takes two operands. We will get the top element of the stack i.e. + will be used as first argument. The second argument is the input operator i.e. *. So the function call will be as *prcd(+, *)* while the function returns false because the precedence of the plus operator is not higher than the multiplication operator. So far, we have only one operand for multiplication i.e. B. As multiplication is also a binary operator, it will also have to wait for the second operand. It has to wait and the waiting room is stack. So we will push it on the stack. Now the top element of the stack is *. The next symbol is 'C'. Being an operand, C will be added to the postfix expression. At this point, our input expression has been completed. Our first 'while loop' executes till the end of input. After the end of the input, the loop will be terminated. Now the control goes to the second while loop which says if there is something on the stack, pop it and add it the postfix expression. In this case, we have * and + on the stack. The * is at the

top of the stack. So when we pop, we get * which is at the top of the stack and it will be added to the postfix expression. In the result of second pop, we get the plus operator (+) which is also added to the postfix expression. The stack is empty now. The while loop will be terminated and postfix expression is formed i.e. ABC*+.

Symbol	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	* +
C	ABC	* +
	ABC*	+
	ABC*+	

If we have to convert the infix expression into the postfix form, the job is easily done with the help of stack. The above algorithm can easily be written in C++ or C language, specially, if you already have the stack class. Now you can convert very big infix expressions into postfix expressions. Why we have done this? This can be understood with the help of the example of spreadsheet programming where the value of cell is the evaluation of some expression. The user of the spreadsheets will use the infix expressions as they are used to it.

Sometimes we do need the parenthesis in the infix form. We have to evaluate the lower precedence operator before the higher precedence operator. If we have the expression (A+B) *C, this means that we have to evaluate + before the multiplication. The objective of using parenthesis is to establish precedence. It forces to evaluate the expression first of all. We also have to handle parenthesis while converting the infix expression into postfix one. When an open parenthesis '(' is read, it must be pushed on the stack. This can be done by setting *prcd(op,'(')* to be FALSE. What is the reason to put the parenthesis on the stack? It is due to the fact that as long as the closing parenthesis is not found, the open parenthesis has to wait. It is not a unary or binary operator. Actually, it is a way to show or write precedence. We can handle the parenthesis by adding some extra functionality in our *prcd* function. When we call *prcd(op, '(')*, it will return false for all the operators and be pushed on the stack. Also, *prcd('(',op)* is FALSE which ensures that an operator after '(' is pushed on the stack. When a ')' is read. All operators up to the first '(' must be popped and placed in the postfix string. To achieve this our function *prcd(op,')')* should return true for all the operators. Both the '(' and the')' will not go to the postfix expression. In postfix expression, we do not need parenthesis. The precedence of the operators is established in such a way that there is no need of the parenthesis. To include the handling of parenthesis, we have to change our algorithm. We have to change the line s.push(c) to:

```
if( s.empty()  ||  symb != ')' )
        s.push( c );
else
        s.pop(); // discard the '('
```

If the input symbol is not ‘)’ and the stack is not empty, we will push the operator on the stack. Otherwise, it is advisable to pop the stack and discard the ‘(‘. The following functionality has to be added in the *prcd* function.

prcd(‘(‘, op)	=	FALSE	for any operator
prcd(op, ‘)’)	=	FALSE	for any operator other than ‘)’
prcd(op, ‘)’)	=	TRUE	for any operator other than ‘(‘
prcd(‘)’, op)	=	error	for any operator.

In the next lecture we will see in detail an example regarding the use of parenthesis.

Data Structures

Lecture No. 08

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 3

3.3.3

Summary

- Conversion from infix to postfix
- C++ Templates
- Implementation of Stack
- Function Call Stack

Conversion from infix to postfix

In the previous lecture, we discussed the way to convert an infix notation into a postfix notation. During the process of conversion, we saw that there may be need of parenthesis in the infix especially at the times when we want to give a higher precedence to an operator of lower precedence. For example, if there is a + operator and * operator in an expression and a programmer wants the execution of addition before the multiplication. To achieve this object, it is necessary to put parentheses around the operands of + operator. Suppose, there is the expression *A + B * C* in which we want to give the precedence to the + operator over * operator. This expression will be written as *(A + B) * C*. Now we are going to discuss the conversion of infix expression that includes parentheses to the postfix expression. We have defined the return values for opening '(' and closing ')' parentheses in the precedence function. Let's try to understand this process with the help of an example of converting the infix expression *(A + B) * C* into a postfix expression. We will see how our algorithm, discussed earlier, converts this infix expression into a postfix expression. To carry out the process of conversion we have three columns symbol, postfix and stack. The column symbol has the input symbols from the expression. The postfix column has the postfix string (expression) after each step and the stack is used to put the operators on it. The whole process of converting the infix notation into a postfix is given in the following table. This process of conversion is completed in eight steps. Each of the rows of the table depicts one step.

Step No.	Symbol	Postfix	Stack
1	(		(
2	A	A	(

3	+	A	(+
4	B	AB	(+
5	)	AB+	
6	*	AB+	*
7	C	AB+C	*
8		AB+C*	

First of all, there is the input symbol '('(i.e. opening parenthesis). As this is not an operand, it may be put on the stack. The next input symbol is 'A'. Being an operand it goes to the postfix string and the stack remains unchanged. Then there is + operator of binary type. Moreover, there is one operand in the postfix string. We push this + operator on the stack and it has to wait for its second operand. Now in the input symbol, there is an operand 'B'. We put his operand in the postfix string. Then after this, there is the closing parenthesis ')' in the input symbol. We know that the presence of a closing parenthesis in the input means that an expression (within the parentheses) has been completed. All of its operands and operators are present with in the parentheses. As studied in the algorithm, we discard a closing parenthesis when it comes in the input. Then the operators from the stack are popped up and put in the postfix string. We also pop the opening parenthesis and discard it as we have no need of opening as well as closing parenthesis in the postfix notation of an expression. This process is carried out in the 5^{th} row of the table. The + operator is put in the postfix string. We also discard the opening parenthesis, as it is not needed in the postfix.
Now the next input symbol is *. We put this operator on the stack. There is one operand for the * operator i.e. AB+. The * operator being a binary operator, has to wait for the second operand. 'C' is the Next input symbol that is an operand. We put it in the postfix string. After this, the input string (expression) ends so we come out of the loop. We check if there is any thing on the stack now? There is * operator in the stack. We pop the operator and put it into the postfix string. This way, we get the postfix form of the given infix expression that becomes *AB+C**. In this postfix expression, the + operator is before the * operator. So addition operation is done before the multiplication. This is mainly due to the fact that in the infix expression, we have put parentheses to give + operator the precedence higher than the * operator. Note that there are no parentheses in the postfix form of the given infix expression.

Now we apply the evaluation algorithm on this postfix expression (i.e. AB+C*). The two operands A and B, will go to the stack. Then operator + will pop these operands from the stack, will add them and push the result back on the stack. This result becomes an operand. Next 'C' will go to the stack and after this * operator will pop these two operands (result of addition and C). Their multiplication will lead to the final result. The postfix notation is simple to evaluate as compared to the infix one. In postfix, we need not to worry about what operation will be carried first. The operators in this notation are in the order of evaluation. However, in the infix notation, we have to force the precedence according to our requirement by putting parentheses in the expression. With the help of a stack data structure, we can do the conversion and evaluation of expressions easily.

C++ Templates

We can use C++ templates for stack and other data structures. We have seen that stack is used to store the operands while evaluating an expression. These operands may be

integers, floating points and even variables. We push and pop the operands to and from the stack. In the conversion of an expression, a programmer uses the stack for storing the operators like +, *, -, and / etc which are single characters. In both cases, the functionality is the same. We push and pop things on and from the stack. At times, we check if the stack is empty or not. Thus identical methods are employed while using stack in evaluating and converting the expressions. However, there may be a difference in the type of the elements (data) used in these cases. We may define a stack, which can store different types of data but for the time being we are restricting ourselves to the stack that can store elements of only one type. In C++ programming, we will have to create two classes *FloatStack* and *CharStack* for operands and operators respectively. These classes of stack have the same implementation. Thus, we write the same code twice only with the difference of data type. Is there any method to write the code for the stack once and then use it for different types of data? This means is there any way that we can make a stack for storing integers, floating points, characters or even objects with the same code written once. The language C++ provides us the facility of writing templates. A template can be understood with the example of a factory that bakes biscuits. The factory may use flour, corn or starch as ingredients of the product. But the process of baking biscuits is the same whatever ingredients it uses. There is no difference in the machinery for producing biscuits with different ingredients. So we call the factory as the template for the biscuits. Similarly in C++ language, a template is a function or class that is written with a generic data type. When a programmer uses this function or class, the generic data type is replaced with the data type, needed to be used in the template function or in the template class. We only give the data type of our choice while calling a template function or creating an object of the template class. The compiler automatically creates a version of that function or class with that specified data type. Thus if we write a template class for stack, then later on we can use it for creating a stack for integers, floating points or characters etc. So instead of writing code for different stacks of different data types, we write one code as a template and reuse it for creating different stacks. We declare the template class in a separate file in addition to the main program file. This file can be used in our program by including it in that file. Following is the code of the template class for stack. This is written in the file **Stack.h**.

```
template <class T>
class Stack
{
        public:
        Stack();
        int empty(void);        // 1=true, 0=false
    int push(T &);              // 1=successful,0=stack overflow
    T pop(void);
        T peek(void);
        ~Stack();
  private:
  int top;
  T* nodes;
};
```

In the above code the line

```
template <class T>
```

shows that we are going to write a template. Here *T* is a variable name for generic data type. We can use any other name but generally *T* is used (T evolves from template). A data type will replace this T whenever template is used in a program. Then we declare member functions of the class. To begin with, there is a constructor of the class with the same name as that of the class i.e. Stack. It is followed by the *empty ()* function and then function *push* that is declared as follows

```
int push(T &) ;
```

We have been using the data (element) type *int* or some other data type in the *push()* function. Now there is *T* written as the data type in the *push()* function. This means that the function takes an argument of type *T*, here *T* is a generic data type and we will use a proper data type while calling this function in our program. This data type will replace *T*. There are also *pop* and *peek* functions that take no arguments but return the value which is of type *T*. The *peek* function is similar to the *top* function that returns (shows) the element from the top but does not remove it from the stack.
In the private section of the class, there are two variables. The variable *top* is used to point to the top of the stack. The pointer *nodes* is used to point to nodes of type *T*. We allocate dynamic memory using this *nodes* pointer.
In the definition of this whole class, *T* is a substitution parameter. While using the stack class in the program, we will replace this *T* with a proper data type.

Implementation

Now we will see how to implement this stack class in our program. Following is the code of the program. We save this code in the file named **Stack.cpp**.

```
#include <iostream.h>
#include <stdlib.h>
#include "Stack.h"
#define MAXSTACKSIZE 50

template <class T>
Stack<T>::Stack()
{
        top = -1;
        nodes = new T[MAXSTACKSIZE];
}

template <class T>
Stack<T>::~Stack()
{
        delete nodes;
}

template <class T>
int Stack<T>::empty(void)
{
        if( top < 0 ) return 1;
```

```
        return 0;
}
template <class T>
int Stack<T>::push(T& x)
{
        if( top < MAXSTACKSIZE )
        {
                nodes[++top] = x;
                return 1;
        }
        cout << "stack overflow in push.\n";
        return 0;
}
template <class T>
T Stack<T>::pop(void)
{
        T x;
        if( !empty() )
        {
                x = nodes[top--];
                return x;
        }
        cout << "stack underflow in pop.\n";
        return x;
}
```

In this code, we include different files in which one is Stack.h , written earlier to declare the template class *Stack*. We have defined a constant size of the stack to 50 by writing the line

```
#define MAXSTACKSIZE 50
```

Next is the definition of the constructor. Here before the signature of the constructor function, we have written

```
template <class T>
Stack<T> :: Stack() ;
```

This means that we are writing a template function that uses *T* wherever a data type is written. As we have declared the Stack class as a template class, <T> will be written with the class name before the access specifier (i.e. ::) while defining a method of the class. Then there is the implementation of the constructor in which we assign the value –1 to *top* and allocate memory by using the *new* operator for stack of size MAXSTACKSIZE of type *T* and put its starting address in the pointer *nodes*. It is pertinent to note that we create an array of type *T* (i.e. a generic type).
Similarly we define the destructor ~Stack, which frees the memory by deleting the nodes. Then there are the different methods of the stack i.e. *empty()*, *push()*, and *pop()*. We define all the methods in the same way as done in case of the constructor. It means that while writing *template <class T>* at the start, we use *T* wherever a data type can be used. The function *empty()* returns a Boolean parameter. It returns 1 that

means TRUE if the stack is empty i.e. if top is less than 0. Otherwise it returns zero i.e. FALSE if the stack is not empty.
We define the *push* function as under

```
int Stack<T>::push(T& x)
{
        if( top < MAXSTACKSIZE )
        {
        nodes[++top] = x;
        return 1;
        }
        cout << "stack overflow in push.\n";
        return 0;
}
```

This function takes an argument *x* by reference. It checks whether there is space in the stack by checking the value of the *top*. If *top* is less than the MAXSTACKSIZE, it means there is space available in the stack. Then it puts the element *x* on the stack and returns 1, indicating that push operation has succeeded. Otherwise, it displays a message of stack overflow and returns zero which indicates that the element was not put on the stack.
Next comes the *pop* method. Here we see that the value returned by this method is of type *T*. In the body of the function, we define a local variable of type *T* and check if the stack is empty. If it is not empty, we pop the value from the top in variable x and return it.
Now let's see the use of this template stack. Here we write the *main* program in a separate file including the stack.cpp (we have written before shortly) to use the stack. Following is the program written in the file **main.cpp**. This program demonstrates the implementation of the stack.

```
#include "Stack.cpp"
int main(int argc, char *argv[])
{
        Stack<int> intstack;
        Stack<char> charstack;
        int x=10, y=20;
        char c='C', d='D';

        intstack.push(x);   intstack.push(y);
        cout << "intstack: " << intstack.pop() << ", " << intstack.pop() << "\n";
        charstack.push(c); charstack.push(d);
        cout << "charstack: " << charstack.pop() << ", " << charstack.pop() << "\n";
}
```

In the above code, consider the line

```
Stack <int> intstack ;
```

This line means that while creating an object *intstack* of Stack, the generic data type *T* should be replaced by the type *int*, In other words, it will be a stack for integers. The compiler will replace *T* with *int* wherever it exists in the code, providing a version of code with data type *int*. The compiler does this automatically.

Similarly the next line

```
Stack <char> charstack ;
```

creates an object of Stack that has name *charstack* and replaces the type *T* with *char*. It shows that it will be a stack of characters. Here *T* is replaced with *char* and a version of code is provided by the compiler, used for the *char* data type. Thus we create two objects of two types (i.e. int and char) of Stack by using the same code of the template class.
To demonstrate the implementation of Stack, we declare two variables of type *int* and two of type *char*. Then we push and pop these variables on and from the proper stack. We push the *int* values on the *intstack* (which we have created for *int* data type). The values of type other than *int* on the stack *intstack* can not be pushed as we have created it to store the *int* data type. And then we poop these values from the stack and show on the screen. Similarly we push the *char* values on the *charstack* (a stack to store *char* values) before displaying these values on the screen with the help of the *pop* method to get these values from the stack.
Now we have the three files of our code i.e. *stack.h, stack.cpp* and *main.cpp*. Having these files in the same directory, we compile the main file (main.cpp) and execute it. Following is the output of the above program.

```
intstack: 10, 20
charstack: C, D
```

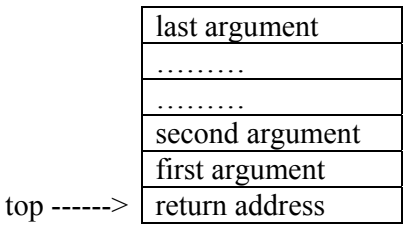
In the above example, we create two objects of Stack to use for two different data types. The compiler automatically provides us two versions of the template code, one for *int* and the other for *char* using the same code, written as template. So it is only due to the use of the template utility, provided by the C++ language only. No other language including C provides this utility of templates. If we write this program in C, the code of the functions of Stack has to be written repeatedly for each data type. Similarly, we will have to write different versions for using different data types. But in templates, a programmer writes the code once and the compiler automatically produces the version of the code with the needed data type. We have implemented the stack by using array. It can also be implemented with the linked list. The use of array or linked list does not matter here. The implementation of the stack remains the same. The templates are so important that C++ provides a library in which a large number of common use functions are provided as templates. This library is a part of the official standard of C++. It is called STL i.e. Standard Template Library. As a library, it is a tested code base. We can use these templates and implement different concepts for our own data types. STL is an important code, pre-developed for us. It is available as a library. Different data structures like stack, queue etc is also there in STL. We can write programs by using them. But here in this course, our goal is to know what the data structures are, what is functioning and how can they be written? So we are writing and discussing the stack templates. You can use data structures from STL in the programming courses or in the professional life. You need not to write the stack or queue from the scratch you can simply use them in your programs from STL.

Function Call Stack

Let's talk about another example of the stack. We know the functionality of the

function calls. Whenever a programmer calls a function, he or she passes some arguments or parameters to the function. The function does work on these arguments and returns a value to the calling function or program. This value is known as the return value of the function. We declare some variables inside the function which are local variables of the function. These variables are demolished when the execution of the function ends. If there are variables in the function that need to be preserved, we have to take care of them. For this purpose, we use global variables or return a pointer to that variable. Now let's see how a stack is used in function calls.

We are using devC++ compiler that actually uses GCC (glue compiler), a public domain compiler. Whenever we call a function, the compiler makes a stack that it uses to fulfill this function call. The compiler puts the entries on the stack in the way that first of all i.e. on the top (i.e. first entry in the stack) is the return address of the function where the control will go back after executing the function. After it, the next entries on the stack are the arguments of the function. The compiler pushes the last argument of the call list on the stack. Thus the last argument of the call list goes to the bottom of the stack after the return address. This is followed by the second last argument of the call list to be pushed on the stack. In this way, the first argument of the call list becomes the first element on the stack. This is shown in the following figure.



In the calling function, after the execution of the function called, the program continues its execution form the next line after the function call. The control comes back here because when the execution of the function ends the compiler pops the address from the stack which it has pushed when the function call was made. Thus the control goes at that point in the program and the execution continues in the calling function or program.

Consider the following code of a function that takes two integer arguments *a, b* and returns the average of these numbers.

```
int i_avg (int a, int b)
{
        return (a + b) / 2;
}
```

To understand the use of stack, look at the assembly language code of the above function that is written as under.

```
globl _i_avg
_i_avg:
        movl 4(%esp), %eax
        addl 8(%esp), %eax          # Add the args
```

```
sarl $1, %eax          # Divide by 2
ret                    # Return value is in %eax
```

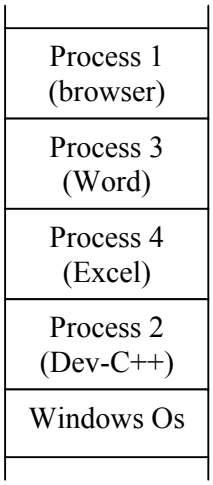
The first statement is *globl_i_avg* which shows that it's a global function that can be called by other functions or programs. After it, there is a label, written as *_i_avg:*
The next statement is *movl 4(%esp), %eax*. Here in this statement, there is the use of stack. Here *esp* is a register in assembly language that is now a stack pointer for us (i.e. top). The *movl* (move long) takes offset 4 from top (4 is number of bytes, we use 4 bytes as in C++ an integer is of 4 bytes.) that means after 4 bytes from the top in the stack it gets the value and put it in the *eax* register. We know that the compiler pushes the arguments of the function in reverse order on the stack. And pushes return address at the end. Thus the order of stack will be that on the top will be the return address and immediately after it will be the first argument. Here in the assembly code generated by the compiler, the compiler pops first argument from offset 4 and puts it in *eax* register. The next statement is

```
addl 8(%esp), %eax
```

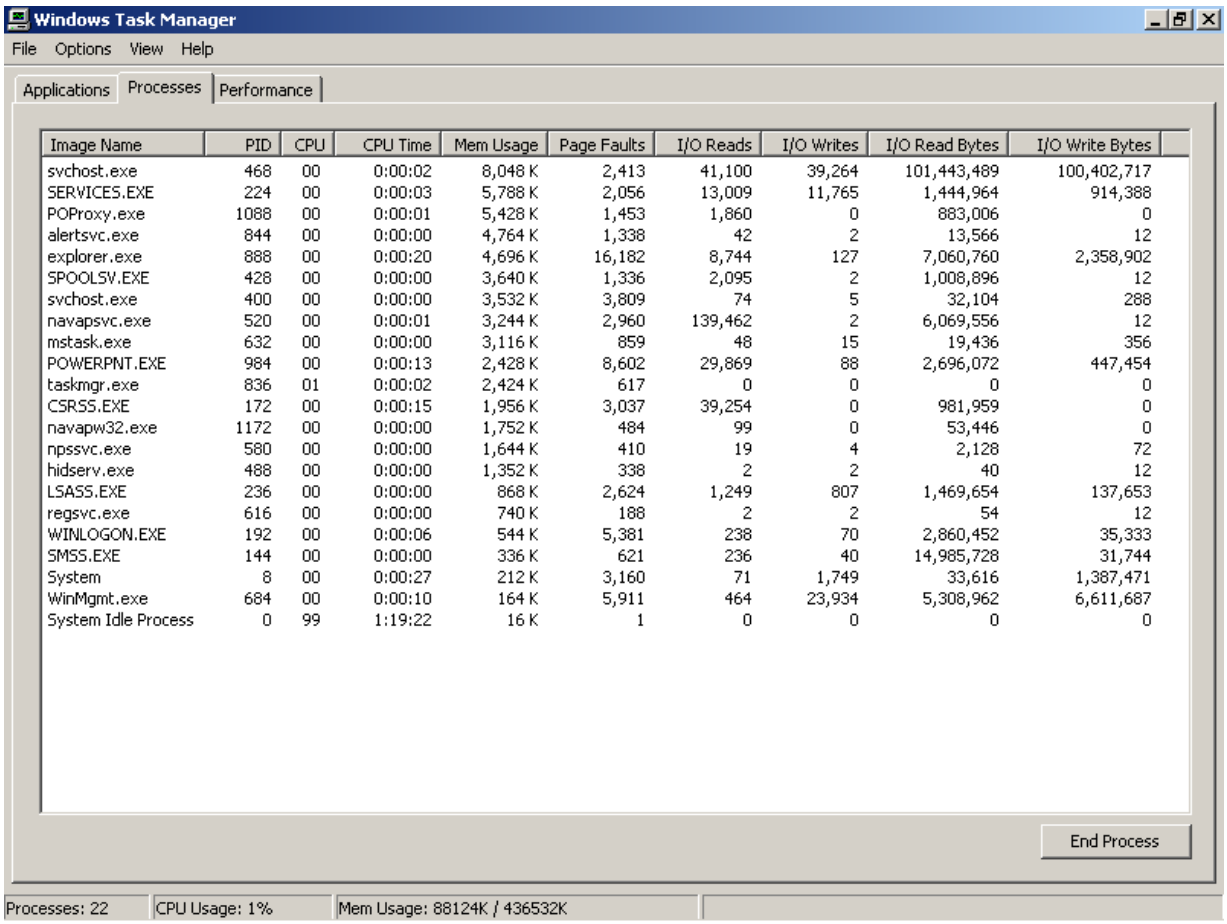
The *addl* takes offset 8 from the stack pointer that is second argument and adds it to *eax*. Thus in the previous two statements, the compiler got the first argument i.e. *a* from the stack and the second argument *b* from the stack, added them before putting the result in *eax*. The next statement

```
sarl $1, %eax
```

is the division statement of assembly language. This statement divides the value in *eax* by 2 and thus *eax* has the resultant value. The last statement i.e. *ret*, returns the value on the top of the stack to the caller function.
So we have seen the use of stack in the execution of a function and how the arguments are passed to the function, how the functions return its return value and finally how the control goes back to the caller function .All this process is executed by using a stack. All the things about the functionality of the function calls are necessary to understand as these will be needed while going to write our own compilers. We will read this in the compilers course. The whole process we have discussed about the use of stack in function calling is known as run time environment.
Different data structures are also used in run time environment of the computer. We know that an executable program while in run, is loaded in the memory and becomes a process. This process is given a block of memory which it uses during its execution. Even the operating system, in which we are working, itself takes memory. Suppose we are running many programs simultaneously, which for example include browser, MS Word, Excel and dev-C++. We can also run programs written by us. Every program which we run takes a block of memory and becomes a process. The following figure shows a part of memory in which different programs occupy a block of memory for their execution.



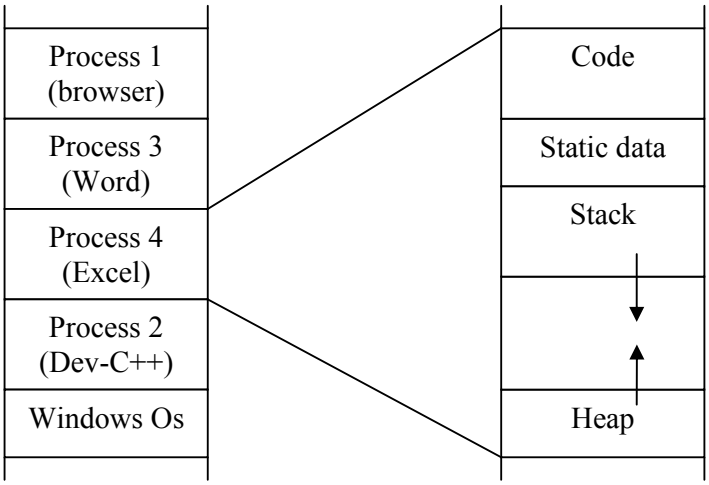
We can also see the details of all the programs running at a specific time. If we press the key combination *Ctrl-Alt-Del*, there appears a window *task manager* on the screen. Following is the figure of the task manager. In the figure, there are many columns i.e. PID (process ID), CPU, CPU time, Memory usage, page faults, I/O Reads, I/O Writes, I/O Read Bytes. These all things we will read in the course of Operating Systems in detail.



Here the thing of our interest is the first, second and fifth column. These columns are

Image Name, PID and Mem Usage (i.e. memory usage). Now look at the row where *explorer.exe* is written in the first column. The process ID (PID) of it is 888 and memory usage is 4696K. This means that the process size of explorer.exe in the memory is 4696 Kilo Bytes (KB). All the processes in the first column of the task manager are present in the memory of the computer at that time. The column Image name has the names of the processes being executed. These have extension *.exe* but there may be other executable programs that have extension other than *.exe*.

The following figure shows the internal memory organization of a process.



This shows that the first part of the memory of the process is for the code. This is the code generated by the compiler of C++, JAVA or VB etc with respect to the language in which the actual code was written. Then the static data of the program occupies the memory. This holds the global variables and different variables of objects. Then in the memory of the process, there is stack. After it there is heap. The stack and heap are used in function calls. We have discussed the use of stack in function calls. When we allocate memory dynamically in our programs, it is allocated from the heap. The use of heap is a topic related to some programming course or to the operating system course.

Data Structures

Lecture No. 09

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 3
3.3.3, 3.4.1, 3.4.2

Summary

- Memory Organization
- Stack Layout During a Function Call
- Queues
- Queue Operations
- Implementing Queue
- Queue using Array
- Use of Queues

Memory Organization

By the end of last lecture, we discussed the uses of stack to develop a process from an executable file and then in function calls. When you run an executable, the operating system makes a process inside memory and constructs the followings for that purpose.

- A code section that contains the binary version of the actual code of the program written in some language like C/C++
- A section for static data including global variables
- A stack and
- Finally, a heap

Stack is used in function calling while heap area is utilized at the time of memory allocation in dynamic manner.

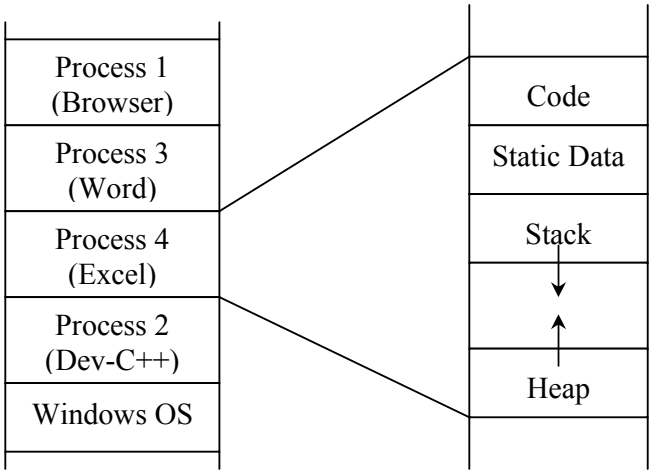


Fig 1. Memory Organization

Stack Layout during a Function Call

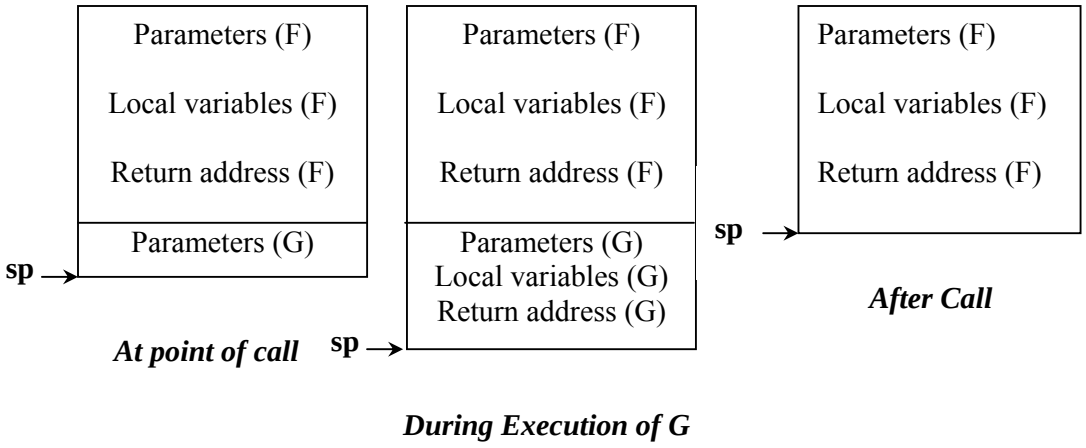


Fig 2: Stack Layout; When function F calls function G

The above diagrams depict the layout of the stack when a function *F* calls a function *G*. Here **sp** stands for stack pointer. At the very left, you will find the layout of the stack just before function *F* calls function *G*. The parameters passed to function *F* are firstly inserted inside the stack. These are followed by the local variables of the function *F and finally* the memory address to return back after the function *F* finishes. Just before function is made to the function *G*, the parameters being passed to the function *G,* are inserted into the stack.

In the next diagram, there is layout of the stack on the right side after the call to the function *G*. Clearly, the local variables of the function G are inserted into the stack after its parameters and the return address. If there are no local variables for a function, the return address is inserted (pushed) on to the stack.

The layout of the stack, when the function *G* finishes execution is shown on the right. You can see that the local variables of function G are no more in the stack. They have been removed permanently along with the parameters passed to the function G. Now, it is clear that when a function call is made, all local variables of the called function and the parameters passed to it, are pushed on to the stack and are destroyed, soon after the the completion of the called function's execution.

In C/C++ language, the variables declared as *static* are not pushed on the stack. Rather, these are stored in another separate section allocated for *static* data of a program. This section for *global* or *static* data can be seen in the fig 1 of this lecture. It is not destroyed till the end of the process's execution. If a variable, say *x* is declared as *static* inside function *G*, *x* will be stored in the *static* data section in the process's memory. Whereas, its value is preserved across *G* function calls. The visibility of *x* is restricted to the function *G* only. But a *static* variable declared as a *class* data is available to *all member functions* of the *class* and a *static* variable declared at *globa*l scope (outside of any class or function body) is available to all functions of the program.

Now, let's move on to another data structure called *queue*.

Queues

A queue is a linear data structure into which items can only be inserted at one end and removed from the other. In contrast to the stack, which is a LIFO (Last In First Out) structure, a queue is a FIFO (First In First Out) structure.

The usage of queue in daily life is pretty common. For example, we queue up while depositing a utility bill or purchasing a ticket. The objective of that queue is to serve persons in their arrival order; the first coming person is served first. The person, who comes first, stands at the start followed by the person coming after him and so on. At the serving side, the person who has joined the queue first is served first. If the requirement is to serve the people in some sort of priority order, there is a separate data structure that supports priorities. The normal queue data structure, presently under discussion, only supports FIFO behavior.

Now, let's see what are the operations supported by the queue.

Queue Operations

The queue data structure supports the following operations:

Operation	Description
enqueue(X)	Place X at the *rear* of the queue.
dequeue()	Remove the *front* element and return it.
front()	Return *front* element without removing it.
isEmpty()	Return TRUE if queue is empty, FALSE otherwise

Implementing Queue

There are certain points related to the implementation of the queue. Suppose we are implementing queue with the help of the linked -list structure. Following are the key points associated with the linked list implementations:

- Insert works in constant time for either end of a linked list.
- Remove works in constant time only.
- Seems best that head of the linked list be the front of the queue so that all removes will be from the front.
- Inserts will be at the end of the list.

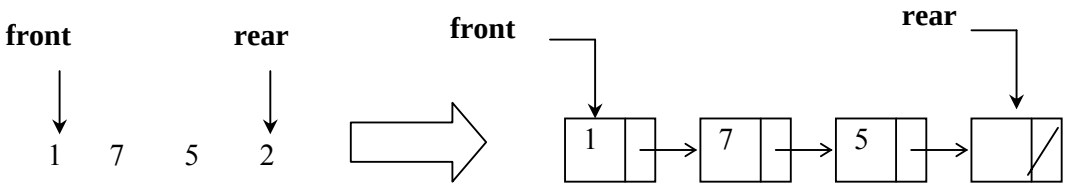


Fig 3. Queue implementation using linked list

The above figure shows *queue* elements on the left with two pointers *front* and *rear*. This is an abstract view of the queue, independent of its implementation method of array or linked list. On the right side is the same *queue* ,using linked list and pointers of *front* and *rear*. When *dequeue()* function is called once, the *front* element *1* is removed. The picture of the *queue* showing one element removal is also depicted below. Note that *front* pointer has been moved to the next element *7* in the list afer removing the *front* element *1*.

After dequeue() is called once

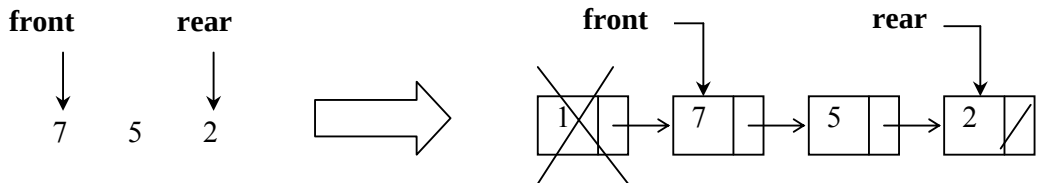


Fig 4. Removal of one element from queue using dequeue()

Now at this stage of the *queue*, we will call enqueue (9) to insert an element *9* in it. . The following figure shows that the new element is inserted at the *rear* end and *rear* pointer starts pointing this new node with element *9*.

At this point of time, the code of these functions of *dequeue()* and *enqueue()* should not be an issue.

Queue after enqueue(9) call

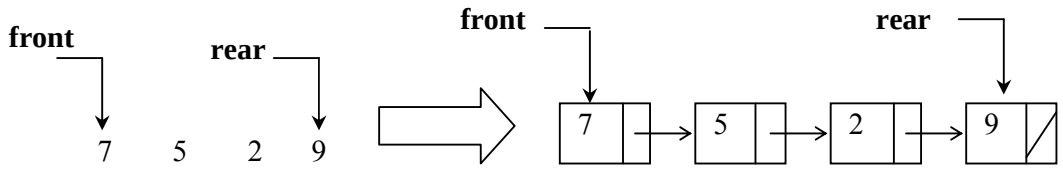


Fig 5. Insertion of one element using enqueue(9)

Note that in this *queue* data structure, the new elements are inserted at *rear* end and removed from the *front*. This is in contrast to *stack* structure where the elements are inserted and removed from the same end.

Let's see the code for queue operations:

```
/* Remove element from the front */
1.  int dequeue()
2.  {
3.     int x = front->get();
4.     Node* p = front;
5.     front = front->getNext();
6.     delete p;
7.     return x;
8.  }
/* Insert an element in the rear */
9.  void enqueue(int x)
10. {
11.     Node* newNode = new Node();
12.     newNode->set(x);
13.      newNode->setNext(NULL);
14.     rear->setNext(newNode);
15.     rear = newNode;
16. }
```

In dequeue() operation, at line 3, the front element is retrieved from the queue and assigned to the *int* variable *x*.
In line 4, the *front* pointer is saved in *Node* pointer variable *p*.
In line 5, the *front* pointer is moved forward by retrieving the address of the next node by using *front->getNext()* and assigning it to the *fron*t pointer.
In line 6, the node pointed to by the *front* pointer is deleted by using *delete front* statement.
At the end of *dequeue()* implementation, the value of deleted node that was saved in the *int* variable *x,* is returned back.

The *enqueue(int)* is used to add an element in the *queue*. It inserts the element in the *rear* of the *queue*. At line 11, a new *Node* object is created using the *new Node()* statement and the returned starting address of the created object is assigned to the *newNode* pointer variable.
In line 12, the value of the passed in parameter *x,* is set in the newly created node

object using the *set()* method.
In line 13, the *next* pointer in the newly created node is set to *NULL*.
In line 14, the newly created node is set as the next node of the node currently pointed by the *rear* pointer.
Ine line 15, the *rear* pointer is set to point to the newly created node.

The code of two smaller functions is as under:

```
/* To retrieve the front element */
int  front()
{
        return  front->get();
}

/* To check if the queue is empty */
int  isEmpty()
{
        return ( front  ==  NULL );
}
```

The *front()* method is used to retrieve the front element. This is the oldest element inserted in the queue. It uses the *get()* method of the *Node* class.
The *isEmpty()* method is used to check whether the queue is empty or not. It checks the address inside the *front* pointer, if it is *NULL. It will* return true indicating that the queue is empty or vice versa.
While studying stack data structure, we implemented it by using both array and linked list. For queue, until now we have been discussing about implementing queue using linked list. Now, let's discuss implementing queue with the help of an array.

Queue using Array

A programmer keeps few important considerations into view account before implementing a queue with the help of an array:
If we use an array to hold the queue elements, both insertions and removal at the front (start) of the array are expensive. This is due to the fact that we may have to shift up to "n" elements.
For the stack, we needed only one end but for a queue, both are required. To get around this, we will not shift upon removal of an element.

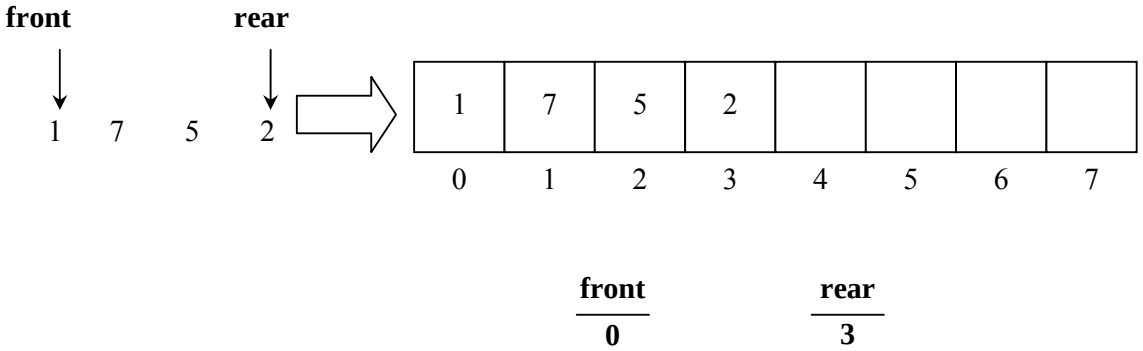


Fig 6. Queue implemented using an array

In the above figure, queue implementation using array is shown. As the array size is *8*, therefore, the index of the array will be from *0* to *7*. The number of elements inside array are *1*, *7*, *5* and *2,* placed at start of the array. The *front* and *rear* in this implementation are not pointers but just indexes of arrays. *front* contains the starting index i.e. *0* while *rear* comprises *3*.
Let's see, how the *enqueue()* works:

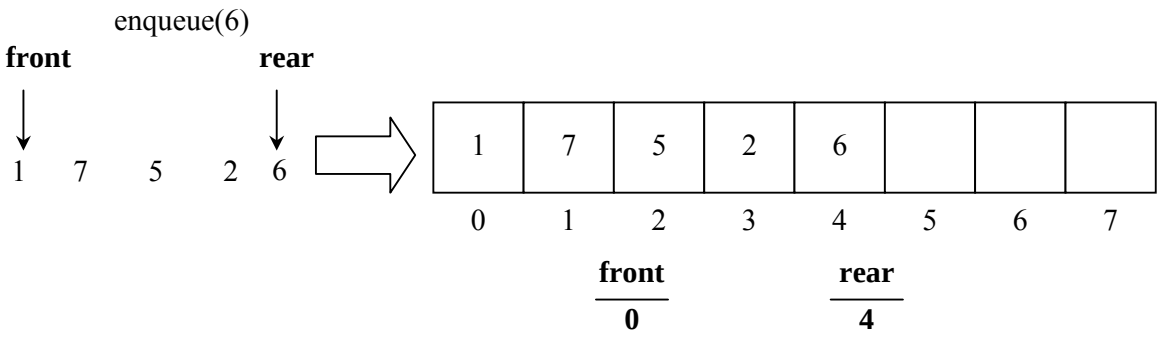


Fig 7. Insertion of one element 6

As shown in the above diagram, an element i.e. *6* has been inserted in the *queue*. Now, the *rear* index is containing *4* while the *front* has the same *0* index. Let's see the figure of the array when another element *8* is inserted in the queue.

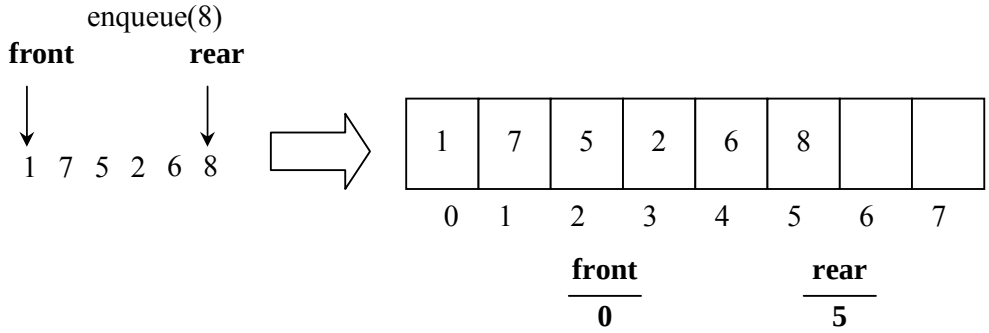


Fig 8. Insertion of another element 8

When an element is removed from the queue. It is removed from the *front* index.

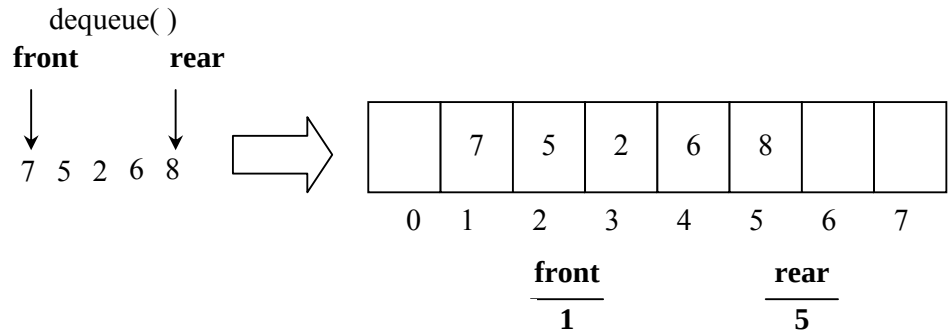


Fig 9. Removal of an element from front

After another call of *dequeue()* function:

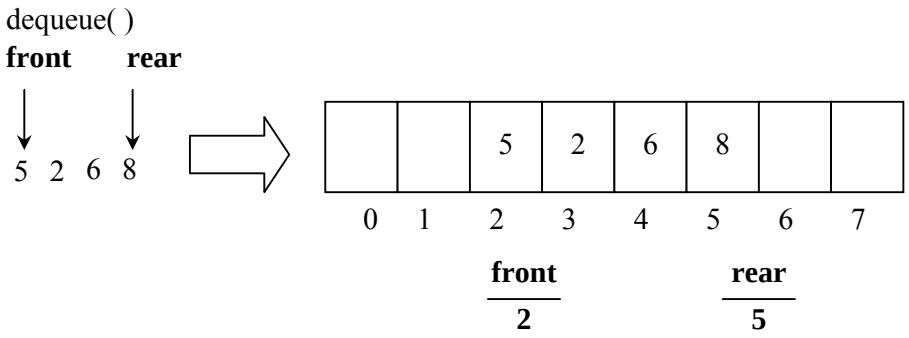


Fig 10. Removal of another element from front

With the removal of element from the queue, we are not shifting the array elements. The shifting of elements might be an expensive exercise to perform and the cost is increased with the increase in number of elements in the array. Therefore, we will leave them as it is.

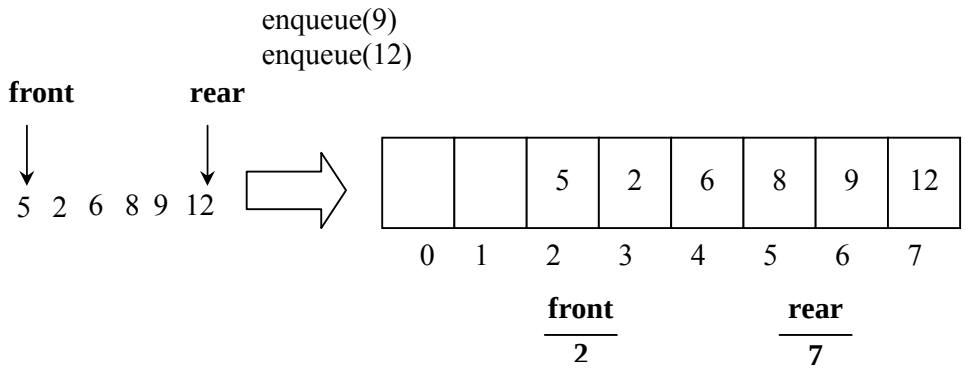


Fig 11. Insertion of elements in the queue

After insertion of two elements in the queue, the array that was used to implement it, has reached its limit as the last location of the array is in use now. We know that there is some problem with the array after it attained the size limit. We observed the similar problem while implementing a stack with the help of an array.

We can also see that two locations at the start of the array are vacant. Therefore, we should can consider how to use those locations appropriately in to insert more elements in the array.

Although, we have insert and removal operations running in constantly, yet we created a new problem that we cannot insert new elements even though there are two places available at the start of the array. The solution to this problem lies in allowing the queue to *wrap around*.

How can we *wrap around*? We can use circular array to implement the queue. We know how to make a linked list circular using pointers. Now we will see how can we make a circular array.

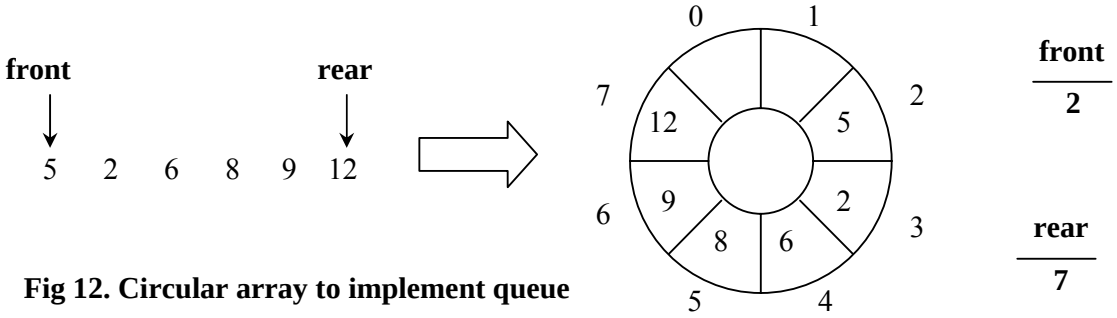


Fig 12. Circular array to implement queue

The number of locations in the above circular array are also eight, starting from index *0* to index *7*. The index numbers are written outside the circle incremented in the clock-wise direction. To insert an element *21* in the array , we insert this element in the location, which is next to index *7*.

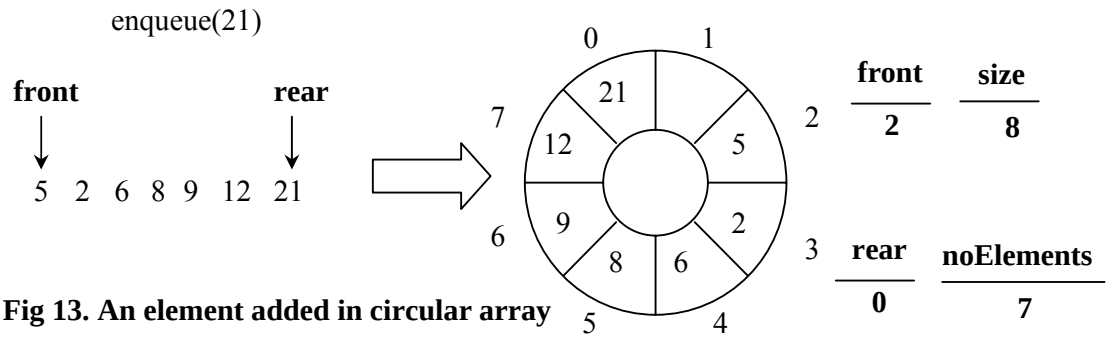


Fig 13. An element added in circular array

Now, we will have to maintain four variables. *front* has the same index *2 while the*, *size* is *8. ' rear'* has moved to index *0* and *noElements* is *7*. Now, we can see that *rear* index has decreased instread of increasing. It has moved from index *7* to *0*. *front* is containing index *2 i.e.* higher than the index in *rear*. Let' see, how do we implement the *enqueue()* method.

```
void  enqueue( int  x)
{
1.   rear  =  (rear + 1) % size;
2.   array[rear]  =  x;
3.   noElements  =  noElements + 1;
}
```

In line 1 of the code, *1* is added in *rear* and the mod operator (that results in remainder of the two operands) is applied with *size* variable. This expression on the right of assignment in line *1* can result from *0* to *7* as size is containing value *8*. This operator ensures that value of this expression will always be from *0* to *7* and increase or decrease from this. This resultant is assigned to the *rear* variable.
In line 2, the *x* (the value passed to *enqueue()* method to insert in the queue) is inserted in the array at the *rear* index position. Therefore, in the above case, the new element *21* is inserted at index *0* in the array.
In line 3, *noElements* is added to accumulate another element in the queue.

Let's add another element in the queue.

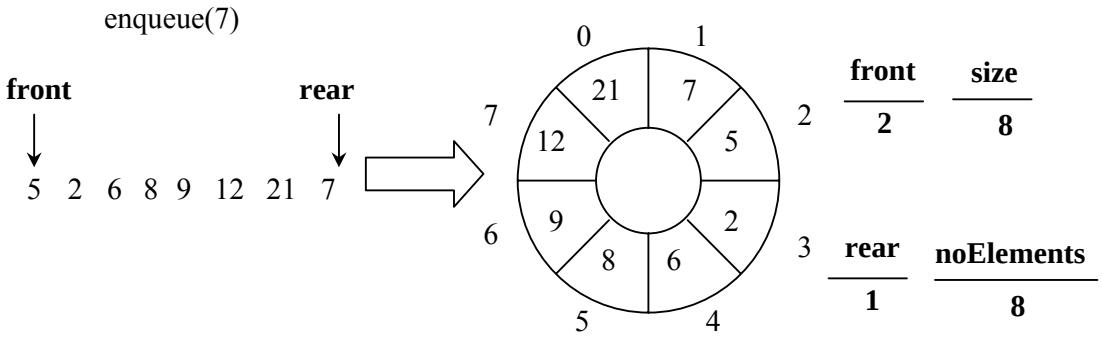


Fig 14. Another element added in circular array

Now, the queue, rather the array has become full. It is important to understand, that queue does not have such characteristic to become full. Only its implementation array has become full. To resolve this problem, we can use linked list to implement a queue. For the moment, while working with array, we will write the method *isFull()*, to determine the fullness of the array.

```
int isFull()
{
   return noElements == size;
}

int isEmpty()
{
        return noElements == 0;
}
```

isFull() returns true if the number of elements (*noElements*) in the array is equal to the *size* of the array. Otherwise, it returns false. It is the responsibility of the caller of the queue structure to call *isFull()* function to confirm that there is some space left in the queue to *enqueue()* more elements.
Similarly *isEmpty()* looks at the number of elements (*noElements*) in the queue. If there is no element, it returns true or vice versa..

Let's see the *dequeue()* method.

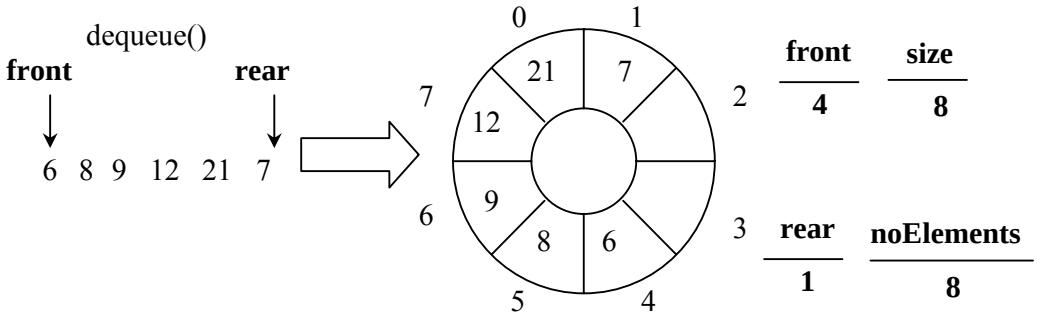


Fig 15. Element removed from the circular array

```
int  dequeue()
{
   int   x  =  array[front];
           front  =  (front + 1) % size;
           noElements  =  noElements - 1;
           return  x;
}
```

In the first line, we take out an element from the array at *front* index position and store it in a variable *x*. In the second line, *front* is incremented by *1* but as the array is circular, the index is looped from *0* to *7*. That is why the *mod (%)* is being used. In the third line, number of elements (*noElements*) is reduced by *1* and finally the saved array element is returned.

Use of Queues

We saw the uses of stack structure in *infix*, *prefix* and *postfix* expressions. Let's see the usage of queue now.

Out of the numerous uses of the queues, one of the most useful is *simulation*. A simulation program attempts to model a real-world phenomenon. Many popular video games are simulations, e.g., SimCity, Flight Simulator etc. Each object and action in the simulation has a counterpart in the real world. Computer simulation is very powerful tool and it is used in different high tech industries, especially in engineering projects. For example, it is used in aero plane manufacturing. Actually Computer Simulation is full-fledged subject of Computer Science and contains very complex Mathematics, sometimes. For example, simulation of computer networks, traffic networks etc.

If the simulation is accurate, the result of the program should mirror the results of the real-world event. Thus it is possible to understand what occurs in the real-world without actually observing its occurrence.

Let us look at an example. Suppose there is a bank with four tellers.

A customer enters the bank at a specific time (t_1) desiring to conduct a transaction.

Any one of the four tellers can attend to the customer. The transaction (withdraws, deposit) will take a certain period of time (t_2). If a teller is free, the teller can process the customer's transaction immediately and the customer leaves the bank at t_1+t_2. It is possible that none of the four tellers is free in which case there is a line of customers at each teller. An arriving customer proceeds to the back of the shortest line and waits for his turn. The customer leaves the bank at t_2 time units after reaching the front of the line.

The time spent at the bank is t_2 plus time waiting in line.

So what we want to simulate is the working environment of the bank that there are specific number of queues of customers in the bank in front of the tellers. The tellers are serving customers one by one. A customer has to wait for a certain period of time before he gets served and by using simulation tool, we want to know the average waiting time of a bank customer. We will talk about this simulation in the next lecture and will do coding also in order to understand it well.

Data Structures

Lecture No. 10

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 3, 6

3.4.3, 6.1

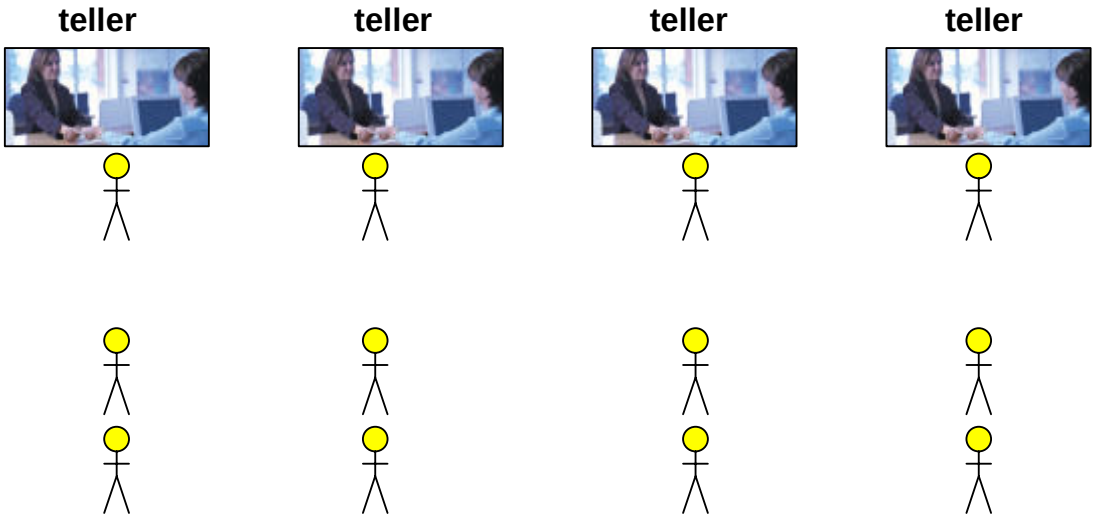
Summary

12) Queues
13) Simulation Models
14) Priority Queue
15) Code of the Bank simulation

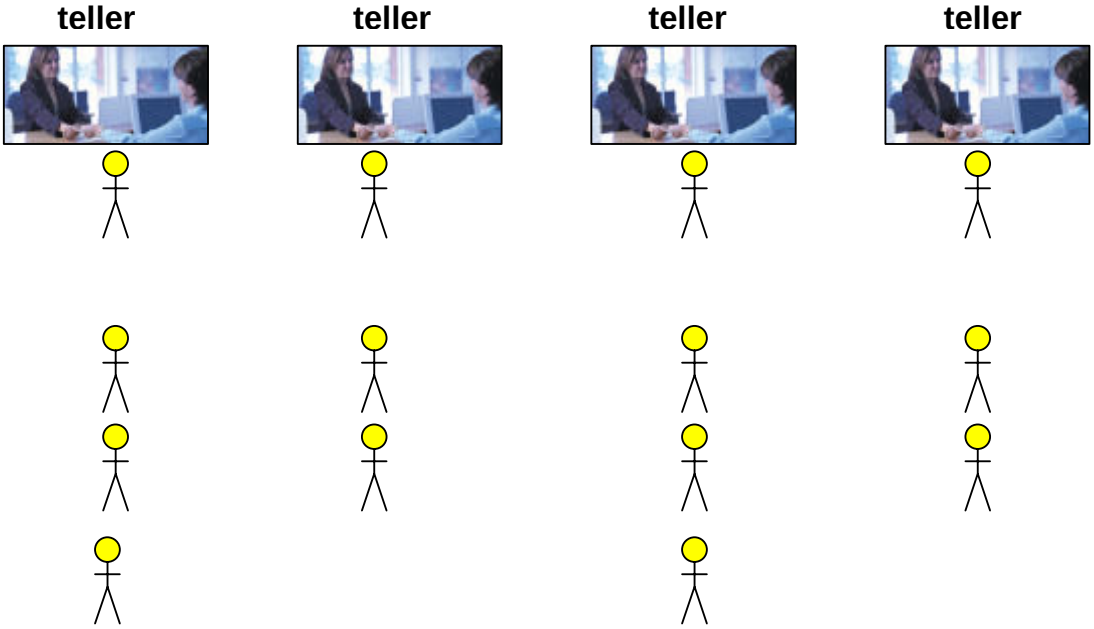
Queues

In the previous lecture, we discussed the queue data structure and demonstrated its implementation by using array and link list. We also witnessed the usefulness of queue as data structure in the simulation example. This concept can be further elaborated with a daily life example relating to banking sector. Suppose, customers want to deposit or withdraw money from a bank, having four cashiers or tellers. The teller helps you in depositing the money or withdrawing the money from the same window. This window is known as teller window. The customer needs some service from the bank like depositing the money, bill etc. This transaction needs some time that may be few minutes. A person enters the bank and goes to the teller who is free and requests him to do the job. After the completion of the transaction, the person goes out of the bank. Now we will discuss a scenario when there is a lot of rush of customers. The tellers are just four. Now the tellers are busy and the new customers will form a queue. In this example, we need a queue in front of each of the tellers. A new customer enters the bank and analyzes the four queues and wants to join the shortest queue. This person has to wait for the persons in front of him to be served. Another person may come behind him. In this simulation, we will restrict the person from changing the queue. A person comes into the bank at 10 O clock. His transaction time is 5 minutes. He has to wait for another fifteen minutes in the queue. After this, the teller serves him in 5 min. This person comes at 10 am and waits for fifteen minutes. As the transaction time is 5 minutes, so he will leave the bank at 1020. Now this is the situation of simulation and we have to write a program for this. We can go to some bank and analyze this situation and calculate the time. At the end of the day, we can calculate the average time for each of the customer. This time can be 30 minutes. Here we will simulate this situation with the help of a computer program. This is the real life example. Let's see the picture of simulations to understand what is happening in the bank.

In the picture below, we have four tellers and four queues, one for each of the tellers. Each teller is serving a customer. When the transaction of a customer is completed, he will leave the bank.



A person enters the bank. He sees that all the four tellers are busy and in each queue there are two persons waiting for their turn. This person chooses the queue no. 3. Another person enters the bank. He analyzed all the queues. The queue no 3 is the biggest and all other are having 2 persons in the queue. He chooses the queue no 1.



Now we have three persons waiting in queue no 1 and 3 and two persons waiting in queue no 2 and 4. The person in queue no.1 completes his transaction and leaves the bank. So the person in the front of the queue no. 1 goes to the teller and starts his transaction. Similarly the person at queue No. 3 finishes his transaction and leaves the premises. The person in front of queue number 3 goes to the teller.

Another person enters the bank and goes to the queue No. 1. This activity goes on. The queues become bigger and shorter. The persons coming in the bank have to wait. If the queues are shorter, people have to wait for less time. However, if the queues are

longer, people have to wait for more time. The transactions can also take much more time, keeping the people waiting. Suppose four persons come with big amount and their transaction takes too much time. These are all parameters which we can incorporate in our simulation making it real. For this, we have carry out more programming. With the introduction of these parameters in the simulation, it will be more close to the real life situation. Simulation, being a very powerful technique, can yield the results, very close to some real life phenomenon.

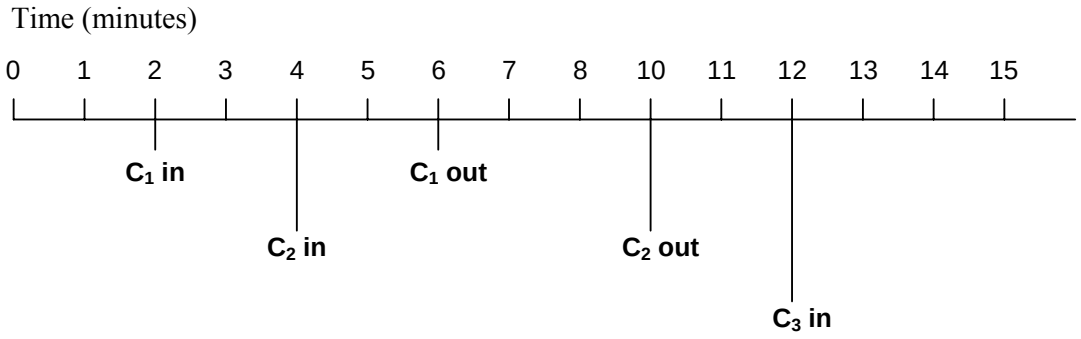
Simulation Models

Let's discuss little bit about the simulation models. Two common models of simulation are time-based simulation and event-based simulation. In time-based simulation, we maintain a timeline or a clock. The clock ticks and things happen when the time reaches the moment of an event.

Suppose we have a clock in the computer. The minute hand moves after every minute. We know the time of the customer's entry into the bank and are aware that his transaction takes 5 minutes. The clock is ticking and after 5 minutes, we will ask the customer to leave the bank. In the program, we will represent the person with some object. As the clock continues ticking, we will treat all the customers in this way. Note that when the customer goes to some teller, he will take 5 minutes for his transaction. During this time, the clock keeps on ticking. The program will do nothing during this time period. Although some other customer can enter the bank. In this model, the clock will be ticking during the transaction time and no other activity will take place during this time. If the program is in some loop, it will do nothing in that loop until the completion of the transaction time.

Now consider the bank example. All tellers are free. Customer C1 comes in 2 minutes after the opening of the bank. Suppose that bank opens at 9:00 am and the customer arrives at 9:02 am. His transaction (withdraw money) will require 4 minutes. Customer C2 arrives 4 minutes after the bank opens (9:04 am). He needs 6 minutes for transaction. Customer C3 arrives 12 minutes after the bank opens and needs 10 minutes for his transaction.

We have a time line and marks for every min.



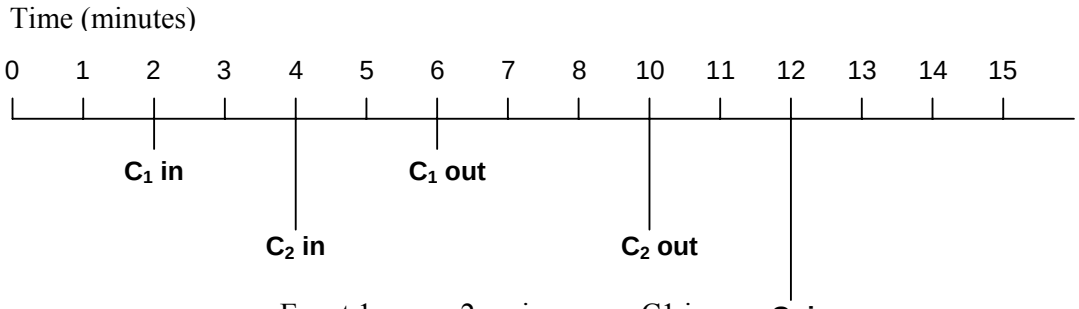
C1 comes at 2 min, C2 enters at 4 min. As C1 needs 4 min for his transaction, so he leaves at 6 min. C2 requires 6 min for the processing so C2 leaves at 10 min. Then C3 enters at 12 min and so on. This way, the activity goes on. Therefore, we can write a

routine of the clock. We take a variable *clock* representing the clock. The clock will run for 24 hrs. Banks are not open for 24 hrs but we will run our loop for 24 hrs. The pseudo code is as under:

```
clock = 0;
while ( clock <= 24*60 ) { // one day
   read new customer;

   if customer.arrivaltime == clock
      insert into shortest queue;

   check the customer at head of all four queues.

   if transaction is over
    remove from queue.

   clock = clock + 1;
}
```

The variable *clock* is initialized to zero. The while loop runs for 24 hrs. Then we read a new customer. This information may be coming from some file. The *if* statement is checking the arrival time of the customer. He comes 10 minutes after the opening of the bank. So when this time is equal to the clock, we insert the customer in the shortest queue. Then we will check the transaction time of all the four customers at each teller. If the transaction time of any customer ends, we will remove it from the queue and he will leave the bank. In the end, we increment the clock with one minute. As seen in the above statements, some activity takes place when the clock reaches at the event time (that is the time to enter the bank or leave the bank arrives). If the customer's arrival time has not come, the first *if* statement becomes false and we do nothing. Similarly if the transaction of customer is not finished, the second *if* statement becomes false. Then this while loop will simply add one min to the clock. This is the clock- based (time- based) simulation.

Let's discuss the other type of simulation i.e. the event-based simulation. Don't wait for the clock to tick until the next event. Compute the time of next event and maintain a list of events in increasing order of time. Remove an event from the list in a loop and process it. Let's see the time line again.



Event 1:	2 mins	C1 in
Event 2:	4 mins	C2 in
Event 3:	6 mins	C1 out
Event 4:	10 mins	C2 out
Event 5:	12 mins	C3 in

The customer C1 comes at 2 min and leaves at 6 min. Customer C2 comes at 4 min and leaves at 10 min and so on. We have written the events list in the above figure. Do not see the clock but see the events on time. Event 1 occurs at 2 min that is the customer C1 enters the bank 2 minutes after its opening. Event 2 is that C2 enters at 4 min. Event 3 is that the customer C1 leaves the bank at 6 min. Event 4 is that the C2 leaves the bank at 10 min and event 5 is that C3 enters the bank at 12 min. Here we have a list of events. How can we process them? We will make a queue of these events. Remove the event with the earliest time from the queue and process it. Insert the newly created events in the queue. A queue where the de-queue operation depends not on FIFO, is called a priority queue.

Priority Queue

As stated earlier, the queue is a FIFO (First in first out) structure. In daily life, you have also seen that it is not true that a person, who comes first, leaves first from the queue. Let's take the example of traffic. Traffic is stopped at the signal. The vehicles are in a queue. When the signal turns green, vehicles starts moving. The vehicles which are at the front of the queue will cross the crossing first. Suppose an ambulance comes from behind. Here ambulance should be given priority. It will bypass the queue and cross the intersection. Sometimes, we have queues that are not FIFO i.e. the person who comes first may not leave first. We can develop such queues in which the condition for leaving the queue is not to enter first. There may be some priority. Here we will also see the events of future like the customer is coming at what time and leaving at what time. We will arrange all these events and insert them in a priority queue. We will develop the queue in such a way that we will get the event which is going to happen first of all in the future. This data structure is known as priority queue. In a sense, FIFO is a special case of priority queue in which priority is given to the time of arrival. That means the person who comes first has the higher priority while the one who comes later, has the low priority. You will see the priority queue being used at many places especially in the operating systems. In operating systems, we have queue of different processes. If some process comes with higher priority, it

will be processed first. Here we have seen a variation of queue. We will use the priority queue in the simulation. The events will be inserted in the queue and the event going to occur first in future, will be popped.

What are the requirements to develop this simulation? We need the C++ code for the simulation. There will be a need of the queue data structure and obviously, the priority queue. Information about the arrival of the customers will be placed in an input file. Each line of the file contains the items (arrival time, transaction duration).

Here are a few lines from the input file.

```
00 30 10  <- customer 1
00 35 05  <- customer 2
00 40 08
00 45 02
00 50 05
00 55 12
01 00 13
01 01 09
```

The first line shows the customer 1. "00 30 10" means Customer 1 arrives 30 minutes after the opening of the bank. He will need 10 minutes for his transaction. The last entry "01 01 09" means customer arrives one hour and one minute after the bank opened and his transaction will take 9 minutes and so on. The file contains similar information about the other customers. We will collect the events now. The first event to occur is the arrival of the first customer. This event is placed in the priority queue. Initially, the four teller queues are empty. The simulation proceeds as follows: when an arrival event is removed from the priority queue, a node representing the customer is placed on the shortest teller queue. Here we are trying to develop an algorithm while maintaining the events queue.

After the opening of the bank, the arrival of the first customer is the first event. When he enters the bank all the four tellers are free. Suppose he goes to the first teller and starts his transaction. After the conclusion of his transaction, he leaves the bank. With respect to events, we have only two events, one is at what time he enters the bank and other is at what time he leaves the bank. When other customers arrive, we have to maintain their events.

If the customer is the only one on a teller queue, an event for his departure is placed on the priority queue. At the same time, the next input line is read and an arrival event is placed in the priority queue. When a departure event is removed from the event priority queue, the customer node is removed from the teller queue. Here we are dealing with the events, not with the clock. When we come to know that a person is coming at say 9:20am, we make an event object and place it in the priority queue. Similarly if we know the time of leaving of the customer from the bank, we will make an event and insert it into the priority queue. When the next customer in the queue is served by the teller, a departure event is placed on the event priority queue. When the other customer arrives, we make an event object and insert it into the priority queue. Now the events are generated and inserted when the customer arrives. But the de-queue is not in the same fashion. When we de-queue, we will get the event which is

going to occur first.

When a customer leaves the bank, the total time is computed. The total time spent by the customer is the time spent in the queue waiting and the time taken for the transaction. This time is added to the total time spent by all customers. At the end of the simulation, this total time divided by the total customers served will be average time consumed by customers. Suppose that 300 customers were served, then we will divide the total time by 300 to get the average time. So with the help of simulation technique, we will get the result that *x* customers came today and spent *y* time in the bank and the average time spent by a customer is *z*.

Code of the Bank Simulation

Let's have a look on the C+ code of this simulation.

```
#include <iostream>
#include <string>
#include <strstream.h>

#include "Customer.cpp"
#include "Queue.h"
#include "PriorityQueue.cpp"
#include "Event.cpp"

Queue q[4];       // teller queues
PriorityQueue pq; //eventList;
int totalTime;
int count = 0;
int customerNo = 0;

main (int argc, char *argv[])
{
  Customer* c;
  Event* nextEvent;

  // open customer arrival file
  ifstream data("customer.dat", ios::in);

  // initialize with the first arriving customer.
  ReadNewCustomer(data);

  While( pq.length() > 0 )
  {
                nextEvent = pq.remove();
                c = nextEvent->getCustomer();
                if( c->getStatus() == -1 ){  // arrival event
                        int arrTime = nextEvent->getEventTime();
                        int duration = c->getTransactionDuration();
                        int customerNo = c->getCustomerNumber();
                        processArrival(data, customerNo,
```

```
                                        arrTime, duration , nextEvent);
    }
            else {  // departure event
                            int qindex = c->getStatus();
                            int departTime = nextEvent->getEventTime();
                            processDeparture(qindex, departTime, nextEvent);
    }
}
```

We have included lot of files in the program. Other than the standard libraries, we have *Customer.cpp*, *Queue.h*, *PriorityQueue.cpp* and *Event.cpp*. With the help of these four files, we will create *Customer* object, *Queue* object, *PriorityQueue* object and *Event* object. You may think that these are four factories, creating objects for us.

As there are four tellers, so we will create equal number of queues (*Queue q[4]*). Then we create a priority queue object *pq* from the *PriorityQueue* factory. We declare *totalTime*, *count* and *customerNo* as int. These are global variables.

In the main method, we declare some local variables of customer and event. Afterwards, the customer.dat file for the input data is opened as:

```
ifstream data("customer.dat", ios::in);
```

We read the first customers data from this file as:

```
readNewCustomer(data);
```

Here *data* is the input file stream associated to *customer.dat*. We will read the arrival time and time of transaction from the file of the first customer. After reading it, we will process this information.

Now there is the *while loop* i.e. the main driver loop. It will run the simulation. First thing to note is that it is not clock-based which is that the loop will execute for 24 hours. Here we have the condition of priority queue's length. The variable *pq* represents the event queue. If there are some events to be processed, the queue *pq* will not be empty. Its length will not be zero. We get the next event from the priority queue, not from the queue. The method *pq.remove()* (de-queue method) will give us the event which is going to happen first in future. The priority of events is according the time. In the event object we have the *customerNo*. In the *if* statement, we check the status of the customer. If the status is –1, it will reflect that this is the new customer arrival event.

We know that when a new customer enters the bank, he will look at the four tellers and go to the teller where the queue is smallest. Therefore in the program, we will check which is the smallest queue and insert the customer in that queue. If the event is about the new customer, the *if* statement returns true. We will get its arrival time, duration and customer number and assign it to the variables *arrTime, duration and customerNo* respectively. We will call the method *processArrival()* and pass it the above information.

If the status of the customer is not equal to –1, it means that the customer is in one of the four queues. The control will go to else part. We will get the status of the customer which can be 0, 1, 2 and 3. Assign this value to *qindex*. Later on, we will see how these values are assigned to the status. We will get the departure time of the customer and call the *processDeparture()* method.

In the main driver loop, we will get the next event from the event queue. In this case, events can be of two types i.e. arrival event and the departure event. When the person enters the bank, it is the arrival event. If any queue is empty, he will go to the teller. Otherwise, he will wait in the queue. After the completion of the transaction, the customer will leave the bank. It is the departure event.

Let's discuss the function *readNewCustomer()*. This function is used to read the data from the file.

```
void readNewCustomer(ifstream& data)
{
        int hour,min,duration;
        if (data >> hour >> min >> duration) {
                customerNo++;
                Customer* c = new Customer(customerNo,
                                hour*60+min, duration);
                c->setStatus( -1 ); // new arrival
                Event* e = new Event(c, hour*60+min );
                pq.insert( e ); // insert the arrival event
        }
        else {
                data.close(); // close customer file
        }
}
```

Here, we have used the >> to read the *hour*, *minute* and *duration* from the file. Then we create a customer object *c* from the customer factory with the *new* keyword. We pass the *customerNo*, *arrival time* and transaction *duration* to the constructor of the customer object. After the object creation, it is time to set its status to –1. This means that it is an arriving customer. Then we create an event object *e* passing it the customer *c* and the *arrival time*. We insert this event into the priority queue *pq*. If there is no more data to read, we go into the else part and close the data file.

Let's see the function *processArrival()*. We have decided that when the customer arrives and no teller is available, he will go to the shortest queue.

```
int processArrival(ifstream &data, int customerNo, int arrTime, int duration,
                                Event* event)
{
        int i, small, j = 0;
        // find smallest teller queue
        small = q[0].length();
        for(i=1; i < 4; i++ )
                if( q[i].length() < small ){
```

```
                        small = q[i].length(); j = i;
                }

        // put arriving customer in smallest queue
        Customer* c = new Customer(customerNo, arrTime, duration );
        c->setStatus(j);  // remember which queue the customer goes in
        q[j].enqueue(c);

        // check if this is the only customer in the.
        // queue. If so, the customer must be marked for
        // departure by placing him on the event queue.

        if( q[j].length() == 1 ) {
                c->setDepartureTime( arrTime+duration);
                Event* e = new Event(c, arrTime+duration );
                pq.insert(e);
        }

        // get another customer from the input
        readNewCustomer(data);
}
```

First of all, we will search for the smallest queue. For this purpose, there is a *for* loop in the method. We will check the length of all the four queues and get the smallest one. We store the index of the smallest queue in the variable *j*. Then we create a customer object. We set its status to *j*, which is the queue no. Then we insert the customer in the smallest queue of the four. The customer may be alone in the queue. In this case, he does not need to wait and goes directly to the teller. This is the real life scenario. When we go to bank, we also do the same. In the banks, there are queues and everyone has to enter in the queue. If the queue is empty, the customers go straight to the teller. Here we are trying to simulate the real life scenario. Therefore if the length of the queue is one, it will mean that the customer is alone in the queue and he can go to the teller. We calculate his departure time by adding the arrival time and transaction time. At this time, the person can leave the bank. We create a departure event and insert it into the priority queue. In the end, we read a new customer. This is the way; a programmer handles the new customers. Whenever a new person enters the bank, we create an event and insert it into the smallest queue. If he is alone in the queue, we create a departure event and insert it into the priority queue. In the main while loop, when we remove the event, in case of first future event, it will be processed. After the completion of the transaction, the person leaves the bank.

We may encounter another case. There may be a case that before leaving the bank, more persons arrive and they have to wait in the queue for their turn. We handle this scenario in the departure routine. The code is:

```
int processDeparture(  int qindex, int departTime, Event* event)
{
        Customer* cinq = q[qindex].dequeue();
```

```
        int waitTime = departTime - cinq->getArrivalTime();
        totalTime = totalTime + waitTime;
        count = count + 1;

        // if there are any more customers on the queue, mark the
        // next customer at the head of the queue for departure
        // and place him on the eventList.
        if( q[qindex].length() > 0 ) {
                cinq = q[qindex].front();
                int etime = departTime + cinq->getTransactionDuration();
                Event* e = new Event( cinq, etime);
                pq.insert( e );
        }
}
```

In this method, we get the information about the *qindex*, *departTime* and *event* from the main method. We get the customer by using the *qindex*. Then we calculate the wait time of the customer. The wait time is the difference of departure time and the arrival time. The total time holds the time of all the customers. We added the wait time to the total time. We incremented the variable *count* by one. After the departure of this customer, next customer is ready for his transaction. The *if* statement is doing this. We check the length of the queue, in case of presence of any customer in the queue, we will check the customer with the *front()* method. We set its departure time (*etime*) by adding the depart time of the previous customer and his transaction time. Then we create an event and insert it in the priority queue.

In the end, we calculate the average time in the main loop and print it on the screen. Average time is calculated by dividing the total time to total customer.

```
// print the final average wait time.

        double avgWait = (totalTime*1.0) / count;
        cout << "Total time:    " << totalTime << endl;
        cout << "Customer:      " << count << endl;
        cout << "Average wait: " << avgWait << endl;
```

You may be thinking that the complete picture of simulation is not visible. How will we run this simulation? Another important tool in the simulation is animation. You have seen the animation of traffic. Cars are moving and stopping on the signals. Signals are turning into red, green and yellow. You can easily understand from the animation. If the animation is combined with the simulation, it is easily understood.

We have an animated tool here that shows the animation of the events. A programmer can see the animation of the bank simulation. With the help of this animation, you can better understand the simulation.

In this animation, you can see the Entrance of the customers, four tellers, priority queue and the Exit. The customers enter the queue and as the tellers are free. They go to the teller straight. Customer C1<30, 10> enters the bank. The customer C1 enters after 30 mins and he needs 10 mins for the transaction. He goes to the teller 1. Then

customer C2 enters the bank and goes to teller 2. When the transaction ends, the customer leaves the bank. When tellers are not free, customer will wait in the queue. In the event priority queue, we have different events. The entries in the priority queue are like *arr, 76* (arrival event at 76 min) or *q1, 80* (event in q1 at 80 min) etc. Let's see the statistics when a customer leaves the bank. At exit, you see the customer leaving the bank as C15<68, 3><77, 3>, it means that the customer C15 enters the bank at 68 mins and requires 3 mins for his transaction. He goes to the teller 4 but the teller is not free, so the customer has to wait in the queue. He leaves the bank at 77 mins.

This course is not about the animation or simulation. We will solve the problems, using different data structures. Although with the help of simulation and animation, you can have a real sketch of the problem.

Data Structures

Lecture No. 11

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4

4.1.1, 4.2.1

Summary

- Implementation of Priority Queue
- Tree
- Binary Tree
- Terminologies of a Binary Tree
- Strictly Binary Tree
- Level
- Complete Binary Tree
- Level of a Complete Binary Tree
- Tips

In the previous lecture, we demonstrated the technique of data structure priority queue with the help of the example from the banking system. Data structure priority queue is a variation of queue data structure to store the events. We have witnessed this simulation by animation. The simulation and animation help a programmer to understand the functionality of a program. As these are not the part of the current course, so we will study simulation and animation in detail in the coming courses.
A queue is a FIFO structure (i.e. first in first out). But there are its variations including the priority queue, discussed in the previous lecture. In priority queue, we were adding data to a queue but did not get the data from the queue by First In, First Out (FIFO) rule. We put events in the queue and got these from the queue with respect to the time of occurrence of the event. Thus we get the items from a priority queue in a particular order with respect to a priority given to the items (events). The priority queue is also an important data structure. Let's see its implementation.

Implementation of Priority Queue

In the priority queue, we put the elements in the queue to get them from the queue with a priority of the elements. Following is the C++ code of the priority queue.

```
#include "Event.cpp"
#define PQMAX 30

class PriorityQueue
{
```

```
    public:
            PriorityQueue()
        {
                size = 0; rear = -1;
            };
            ~PriorityQueue() {};
            int full(void)
            {
                return ( size == PQMAX ) ? 1 : 0;
            };

        Event* remove()
            {
                if( size > 0 )
            {
                    Event* e = nodes[0];
                    for(int j=0; j < size-2; j++ )
                    nodes[j] = nodes[j+1];
                    size = size-1; rear=rear-1;
                    if( size == 0 ) rear = -1;
                    return e;
                }
                return (Event*)NULL;
                cout << "remove - queue is empty." << endl;
            };

        int insert(Event* e)
        {
                if( !full() )
            {
                    rear = rear+1;
                    nodes[rear] = e;
                size = size + 1;
                    sortElements(); // in ascending order
                return 1;
                }
                cout << "insert queue is full." << endl;
                return 0;
            };

            int length() { return size; };
};
```

In this code, the file Events.cpp has been included. Here we use events to store in the queue. To cater to the need of storing other data types too, we can write the PriorityQueue class as a template class.

In the above code, we declare the class PriorityQueue. Then there is the public part of the class. In the public part, at first a programmer encounters the constructor of the class. In the constructor, we assign the value 0 to *size* and –1 to *rear* variables. A destructor, whose body is empty, follows this. Later, we employ the method *full()*

which checks the *size* equal to the PQMAX to see whether the queue is full. If the *size* is equal to PQMAX, the function returns 1 i.e. TRUE. Otherwise, it returns 0 i.e. FALSE. We are going to implement the priority queue with an array. We can also use linked list to implement the priority queue. However, in the example of simulation studied in the previous lecture, we implemented the queue by using an array. We have seen in the simulation example that there may be a maximum of five events. These events include one arrival event and four departure events. That means four queues from where the customers go to the teller and one to go out of the bank after the completion of the transaction. As we know that there is a need of only five queues, so it was decided to use the array instead of dynamic memory allocation and manipulating the pointers.

In the *remove()* method, there are some things which are the property of the priority queue. We don't have these in the queue. In this method, first of all we check the size of the priority queue to see whether there is something in the queue or not. If size is greater than 0 i.e. there are items in the queue then we get the event pointer (pointer to the object Event) *e* from the first position (i.e. at index 0) of the array, which we are using internally to implement the queue. At the end of the method, we return this event object *e*. This means that we are removing the first object from the internal array. We already know that the removal of an item from the start of an array is very time consuming as we have to shift all the remaining items one position to the left. Thus the *remove()* method of the queue will execute slowly. We solved this problem by removing the item from the position where the *front* pointer is pointing. As the *front* and *rear* went ahead and we got empty spaces in the beginning, the circular array was used. Here, the *remove()* method is not removing the element from the *front*. Rather, it is removing element from the first position (i.e. index 0). Then we execute a *for* loop. This *for* loop starts from 0 and executes to *size-2*. We can notice that in this *for* loop, we are shifting the elements of the array one position left to fill the space that has been created by removing the element from the first position. Thus the element of index 1 becomes at index 0 and element of index 2 becomes at index 1 and so on. Afterwards, we decrease *size* and *rear* by 1. By decreasing the size 1 if it becomes zero, we will set *rear* to –1. Now by the statement

```
return e ;
```

We return the element (object e), got from the array. The outer part of the *if* block

```
return (Event*)NULL;
        cout << "remove - queue is empty." << endl;
```

is executed if there is nothing in the queue i.e. *size* is less than 0. Here we return NULL pointer and display a message on the screen to show that the queue is empty.

Now let's look at the *insert()* method. In this method, first of all we check whether the array (we are using internally) is full or not. In case, it is not full, we increase the value of *rear* by 1. Then we insert the object *e* in the nodes array at the position *rear*. Then the *size* is increased by 1 as we have inserted (added) one element to the queue. Now we call a method *sortElements()* that sorts the elements of the array in an order. We will read different algorithms of sorting later in this course.

We have said that when we remove an element from a priority queue, it is not according to the FIFO rule. We will remove elements by some other rule. In the simulation, we had decided to remove the element from the priority queue with

respect to the time of occurrence of an event (arrival or departure). We will remove the element (event) whose time of occurrence is before other events. This can be understood from the example of the traffic over a bridge or crossing. We will give higher priority to an ambulance as compared to a bus. The cars will have the lower priority. Thus when a vehicle has gone across then after it we will see if there is any ambulance in the queue. If it is there, we will remove it from the queue and let go across the bridge. Afterwards, we will allow a bus to go and then the cars. In our simulation example, we put a number i.e. time of occurrence, with the object when we add it to the queue. Then after each insertion, we sort the queue with these numbers in ascending order. Thus the objects in the nodes array get into an order with respect to the time of their occurrence. After sorting, the first element in the array is the event, going to be occurring earliest in the future. Now after sorting the queue we return 1 that shows the insert operation has been successful. If the queue is full, we display a message to show that the queue is full and return 0, indicating that the insert operation had failed.

Then there comes the *length()* method, having a single statement i.e.

```
return size ;
```

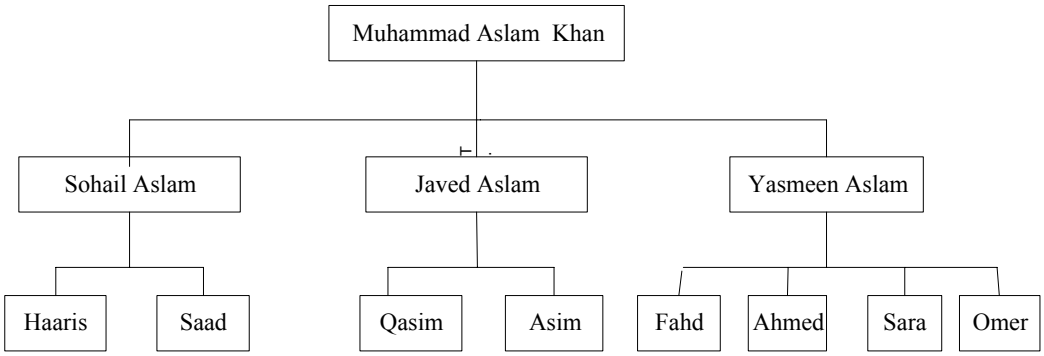
This method returns the size of the queue, reflecting the number of elements in the queue. It is not the size of the array used internally to store the elements of the queue.

We have seen the implementation of the priority queue by using an array. We will use the priority queue in some other algorithms later. Now, we will see another implementation of the priority queue using some thing other than array, which is much better than using an array. This will be more efficient. Its *remove* and *insert* methods will be faster than the ones in array technique. Here in the simulation, we were making only five queues for the events. Suppose, if these go to hundreds or thousands, a lot of time will be spent to remove an element from the queue. Similarly, when an element is added, after adding the element, to sort the whole array will be a time consuming process. Thus the application, with the use of the priority queue, will not be more efficient with respect to the time.

Tree

Now let's talk about a data structure called tree. This is an important data structure. This data structure is used in many algorithms. We will use it in most of our assignments. The data structures that we have discussed in previous lectures are linear data structures. The linked list and stack are linear data structures. In these structures, the elements are in a line. We put and get elements in and from a stack in linear order. Queue is also a linear data structure as a line is developed in it. There are a number of applications where linear data structures are not appropriate. In such cases, there is need of some non-linear data structure. Some examples will show us that why non-linear data structures are important. Tree is one of the non-linear data structures.

Look at the following figure. This figure (11.1) is showing a genealogy tree of a family.



In this genealogy tree, the node at the top of the tree is *Muhammad Aslam Khan* i.e. the head of the family. There are three nodes under this one. These are *Sohail Aslam*, *Javed Aslam* and *Yasmeen Aslam.* Then there are nodes under these three nodes i.e. the sons of these three family members. You may have seen the tree like this of some other family. You can make the tree of your family too. The thing to be noted in this genealogical tree is that it is not a linear structure like linked list, in which we have to tell that who is the father and who is the son. We make it like a tree. We develop the tree top-down, in which the father of the family is on the top with their children downward. We can see the similar tree structure of a company. In the tree of a company, the CEO of the company is on the top, followed downwardly by the managers and assistant managers. This process goes downward according to the administrative hierarchy of the company. Our tree structure is different from the actual tree in the way that the actual tree grows from down to up. It has root downside and the leaves upside. The data structure tree is opposite to the real tree as it goes upside down. This top-down structure in computer science makes it easy to understand the tree data structure.

There may be situations where the data, in our programs or applications, is not in the linear order. There is a relationship between the data that cannot be captured by a linked list or other linear data structure. Here we need a data structure like tree.

In some applications, the searching in linear data structures is very tedious. Suppose we want to search a name in a telephone directory having 100000 entries. If this directory is in a linked list manner, we will have to traverse the list from the starting position. We have to traverse on average half of the directory if we find a name. We may not find the required name in the entire directory despite traversing the whole list. Thus it would be better to use a data structure so that the search operation does not take a lot of time. Taking into account such applications, we will now talk about a special tree data structure, known as binary tree.

Binary Tree

The mathematical definition of a binary tree is

"A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees".

Each element of a binary tree is called a node of the tree.

Following figure shows a binary tree.

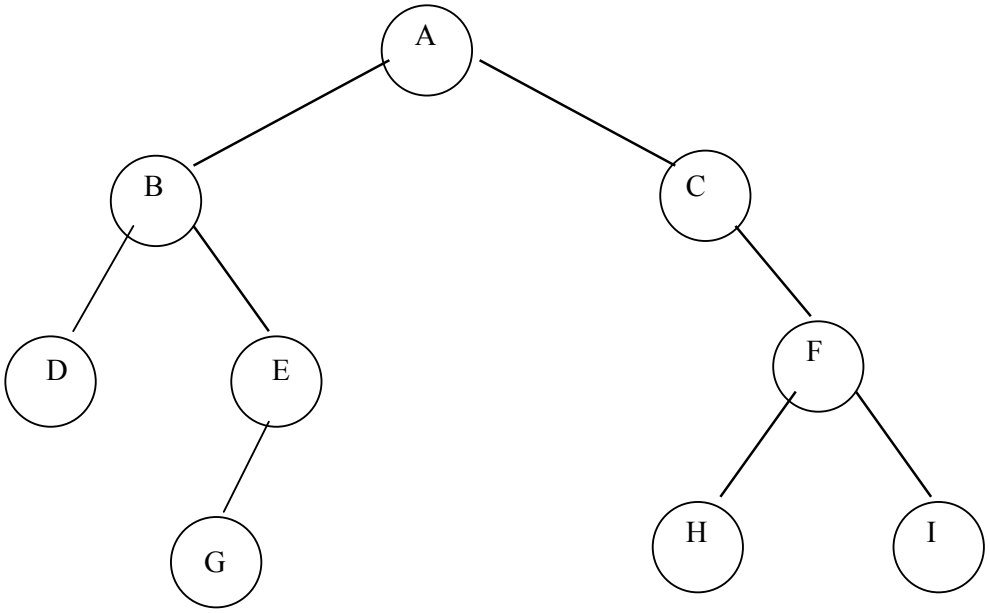


Fig 11.2: A Binary Tree

There are nine nodes in the above figure of the binary tree. The nine nodes are A, B, C, D, E, F, G, H, and I. The node A is at the top of the tree. There are two lines from the node A to left and right sides towards node B and C respectively. Let's have a look at the left side of the node A. There are also two lines from node B to left and right, leading to the nodes D and E. Then from node, E there is only one line that is to the left of the node and leads to node G. Similarly there is a line from node C towards the node F on right side of the node C. Then there are two lines from node F that leads to the nodes H and I on left and right side respectively.

Now we analyze this tree according to the mathematical definition of the binary tree. The node A is the root of the tree. And tree structure (i.e. the nodes B, D, E, G and their relation) on the left side of the node A is a sub-tree, called the *Left subtree*. Similarly the nodes (C, F, H and I) on the right side of the node A comprise the sub-tree termed as *Right subtree*. Thus we made three parts of the tree. One part contains only one node i.e. A while the second part has all the nodes on left side of A, called as *Left subtree*. Similarly the third part is the *Right subtree*, containing the nodes on right side of A. The following figure depicts this scenario.

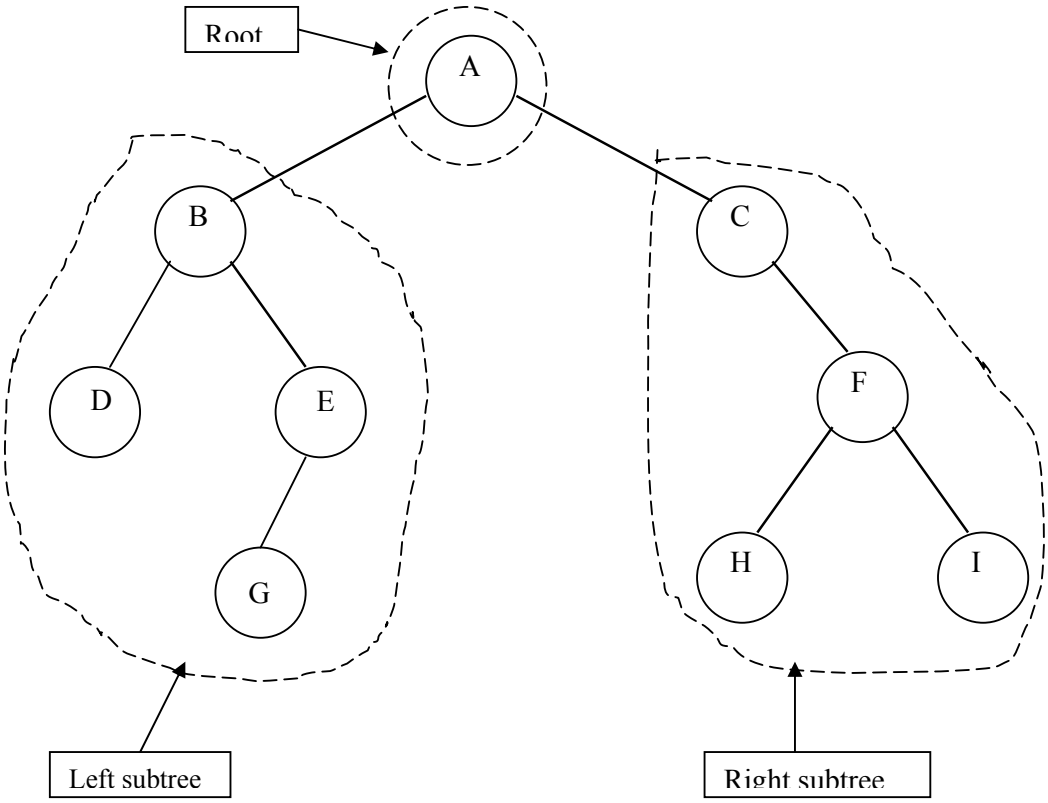


Fig 11.3: Analysis of a binary tree

The same process of sub classing the tree can be done at the second level. That means the left and right subtrees can also be divided into three parts similarly. Now consider the left subtree of node A. This left subtree is also a tree. The node B is the root of this tree and its left subtree is the node D. There is only one node in this left subtree. The right subtree of the node B consists of two nodes i.e. E and G. The following figure shows the analysis of the tree with root B.

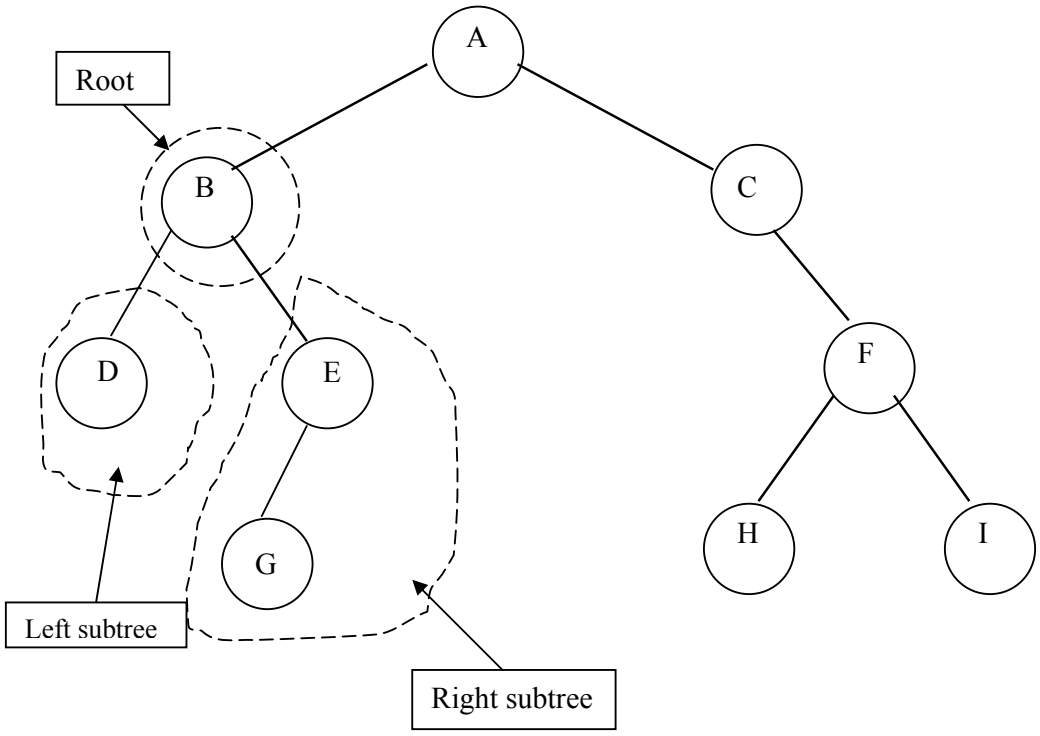


Fig 11.4: Analysis of a binary tree

On going one step down and considering right subtree of node B, we see that E is the root and its left subtree is the single node G. There is no right subtree of node E or we can say that its right subtree is empty. This is shown in the following figure.

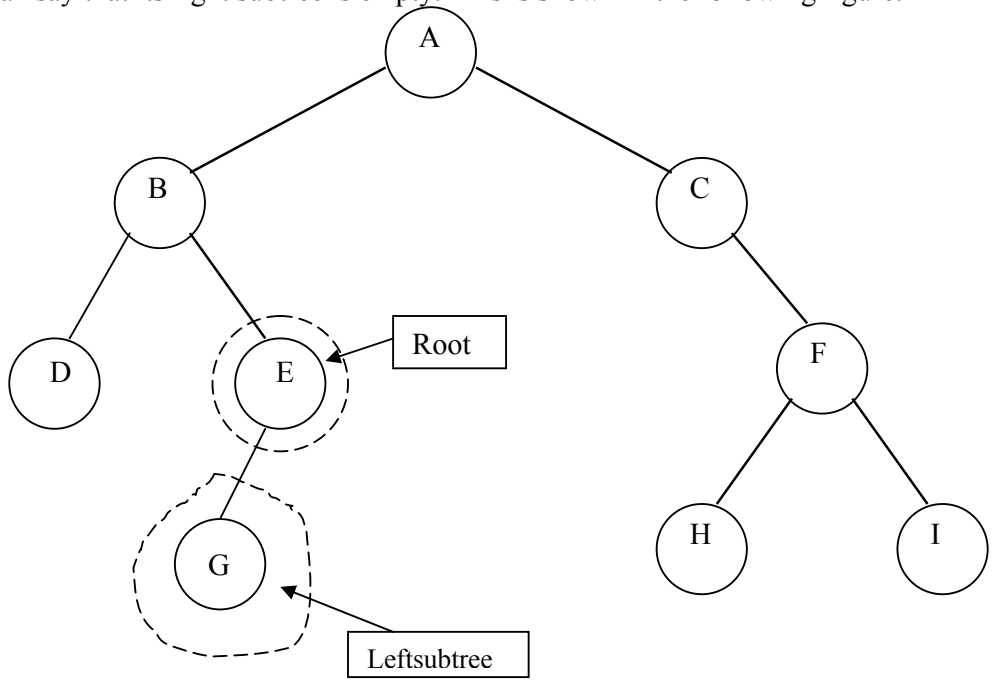


Fig 11.5: Analysis of a binary tree

Now the left sub tree of E is the single node G. This node G can be considered as the root node with empty left and right sub trees.

The definition of tree is of recursive nature. This is due to the fact that we have seen that which definition has been applied to the tree having node A as root, is applied to its subtrees one level downward. Similarly as long as we go down ward in the tree the same definition is used at each level of the tree. And we see three parts i.e. root, left subtree and right subtree at each level.
Now if we carry out the same process on the right subtree of root node A, the node C will become the root of this right subtree of A. The left subtree of this root node C is empty. The right subtree of C is made up of three nodes F, H and I. This is shown in the figure below.

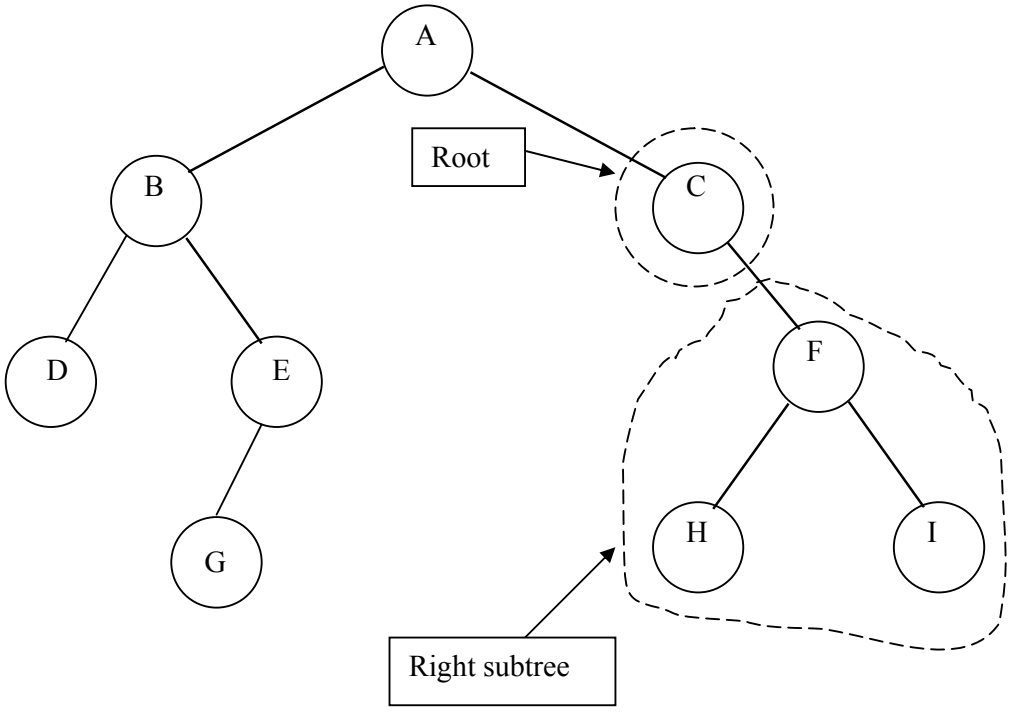


Fig 11.6: Analysis of a binary tree

Now we apply the same recursive definition on the level below the node C. Now the right subtree of the node C will be considered as a tree. The root of this tree is the node F. The left and right subtrees of this root F are the nodes H and I respectively.
The following figure depicts this.

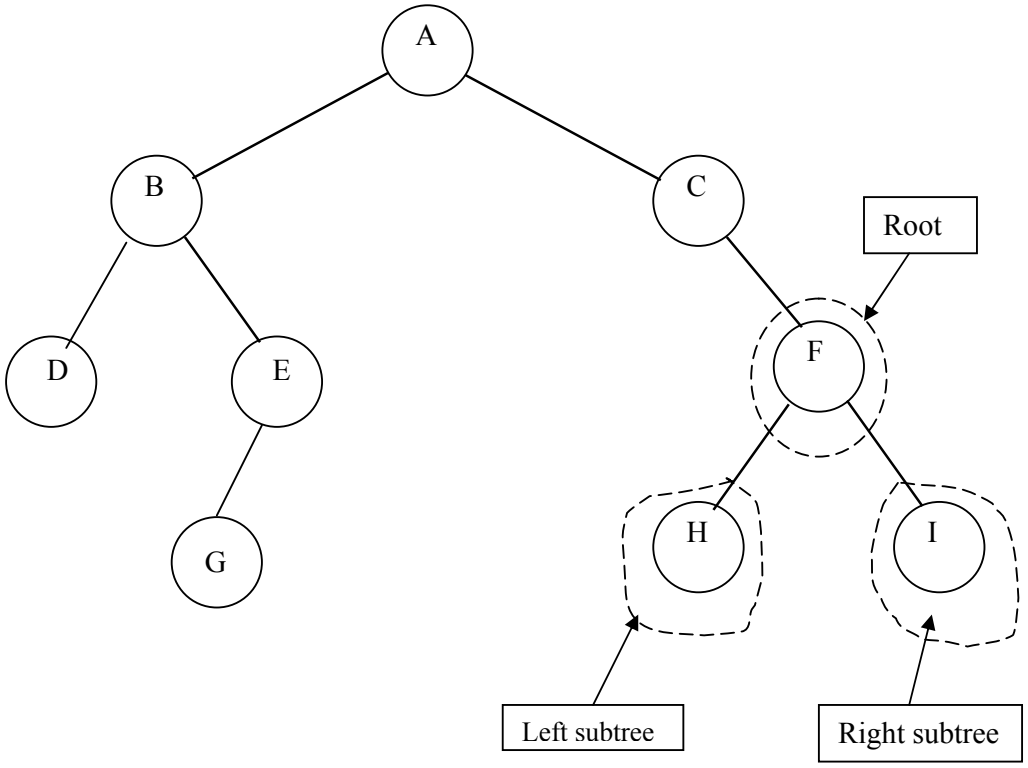


Fig 11.7: Analysis of a binary tree

We make (draw) a tree by joining different nodes with lines. But we cannot join any nodes whichever we want to each other. Consider the following figure.

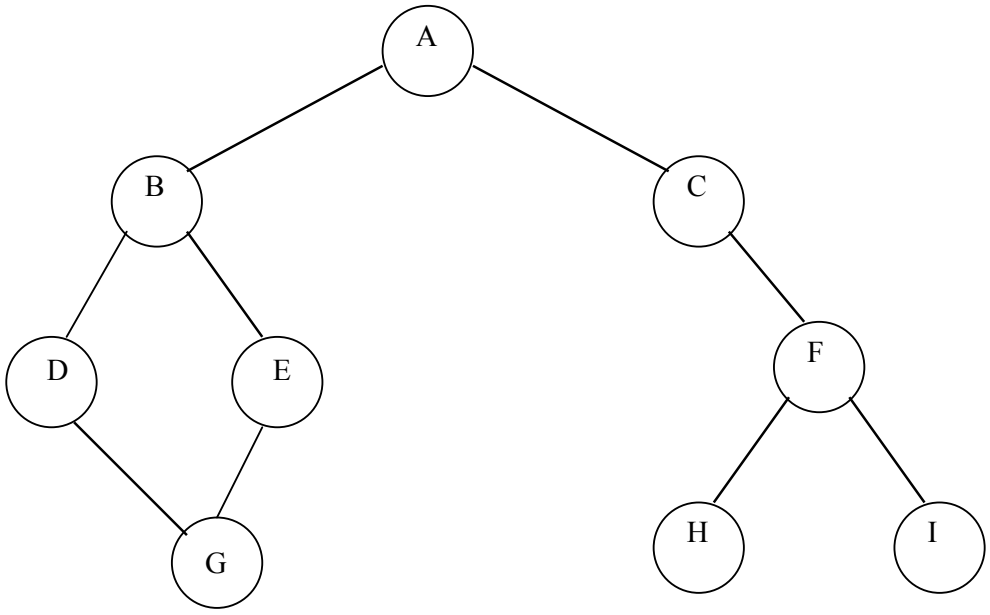


Fig 11.8: A non-tree structure

It is the same tree, made earlier with a little modification. In this figure we join the node G with D. Now this is not a tree. It is due to the reason that in a tree there is always one path to go (from root) to a node. But in this figure, there are two paths (tracks) to go to node G. One path is from node A to B, then B to D and D to G. That means the path is A-B-D-G. We can also reach to node G by the other path that is the path A-B-E-G. If we have such a figure, then it is not a tree. Rather, it may be a graph. We will discuss about graphs at the end of this course.
Similarly if we put an extra link between the nodes A and B, as in the figure below, then it is also no more a tree. This is a multiple graph as there are multiple (more than 1) links between node A and B.

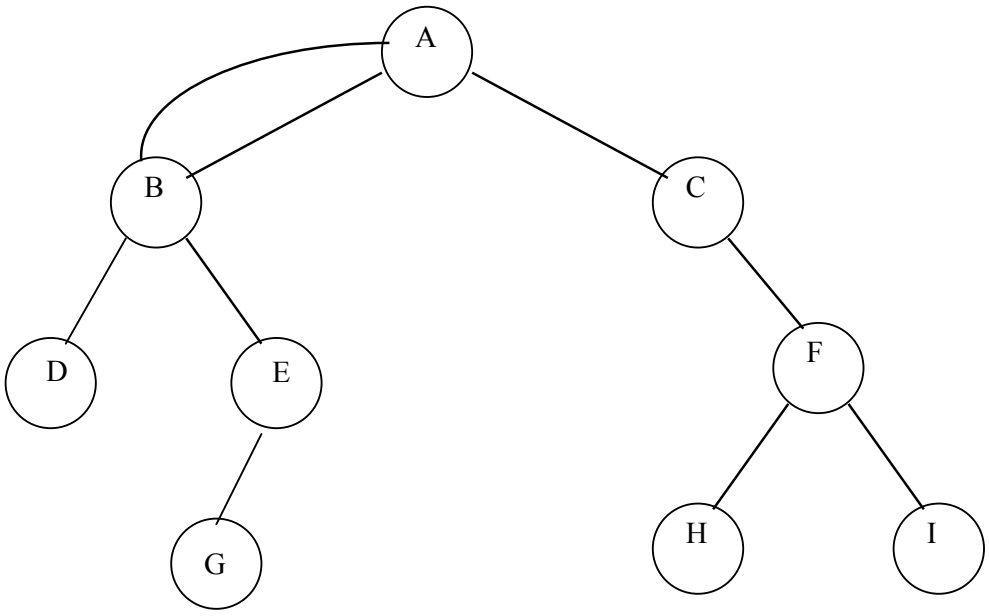


Fig 11.9: A non-tree structure

Similarly if we put other links between different nodes, then the structure thus developed will not be a tree. The following figure is also an example of a structure that is not a tree as there are multiple links between the different nodes.

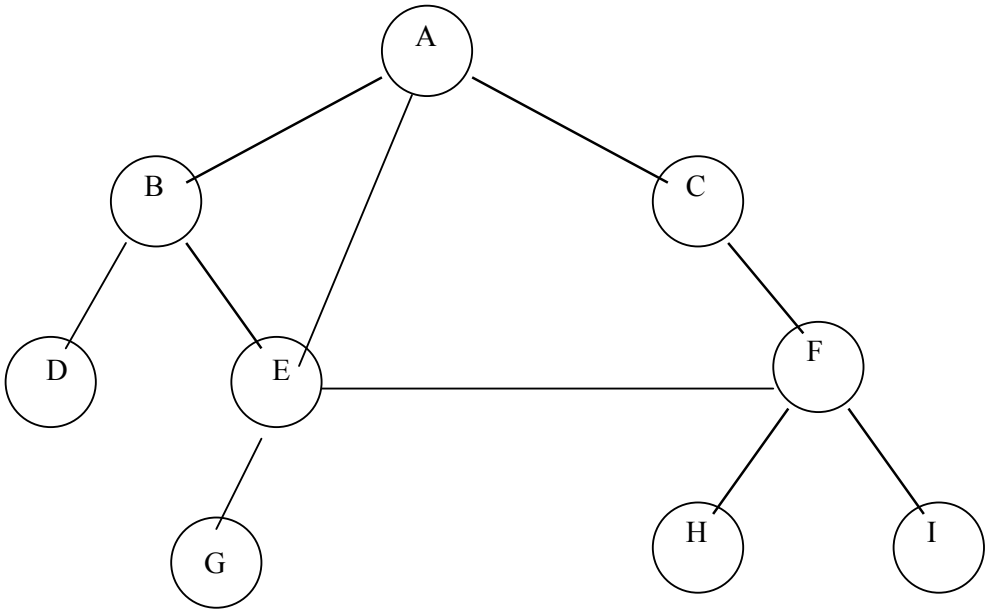


Fig 11.10: A non-tree structure

Terminologies of a binary tree

Now let's discuss different terminologies of the binary tree. We will use these terminologies in our different algorithms. Consider the following figure.

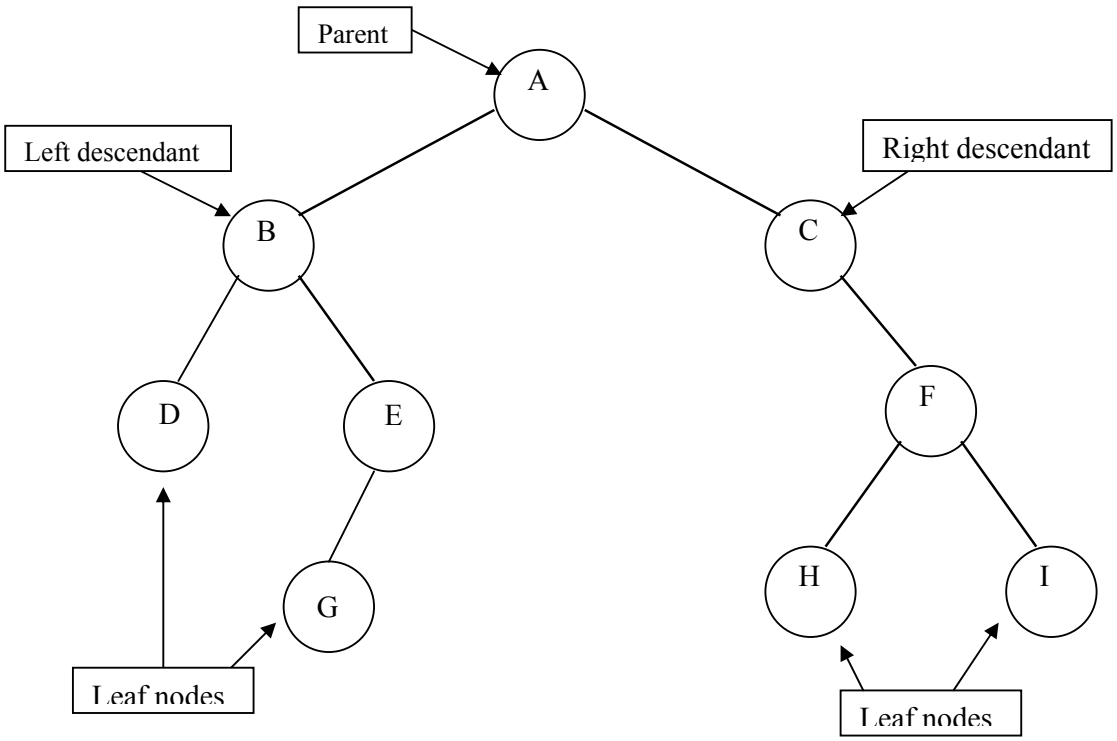


Fig 11.11: Terminologies used in a binary tree

We have discussed that the node on the top is said root of the tree. If we consider the relationship of these nodes, A is the parent node with B and C as the left and right descendants respectively. C is the right descendant of A. Afterwards, we look at the node B where the node D is its left descendant and E is its right descendant. We can use the words descendant and child interchangeably. Now look at the nodes D, G, H and I. These nodes are said leaf nodes as there is no descendant of these nodes. We are just introducing the terminology here. In the algorithms, we can use the words root or parent, child or descendant. So the names never matter.

Strictly Binary Tree

There is a version of the binary tree, called strictly binary tree. A binary tree is said to be a strictly binary tree if every non-leaf node in a binary tree has non-empty left and right subtrees.
Now consider the previous figure 11.11. We know that a leaf node has no left or right child. Thus nodes D, G, H and I are the leaf nodes. The non-leaf nodes in this tree are A, B, C, E and F. Now according to the definition of strictly binary tree, these non-leaf nodes (i.e. A, B, C, E and F) must have left and right subtrees (Childs). The node A has left child B and right child C. The node B also has its left and right children that are D and E respectively. The non-leaf node C has right child F but not a left one. Now we add a left child of C that is node J. Here C also has its left and right children. The node F also has its left and right descendants, H and I respectively. Now the last non-leaf node E has its left child, G but no right one. We add a node K as the right child of the node E. Thus the tree becomes as shown below in the figure 11.12.

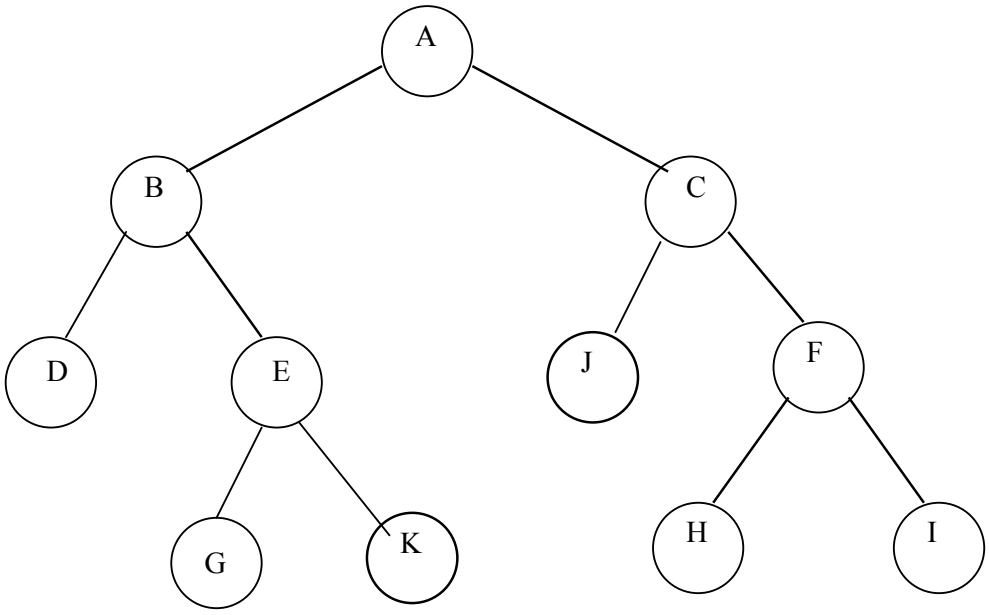


Fig 11.12: A Strictly binary tree

Now all the non-leaf nodes (A, B, C, E and F) have their left and right children so according to the definition, it is a strictly binary tree.

Level

The level of a node in a binary tree is defined as follows:

- Root has level 0,
- Level of any other node is one more than the level its parent (father).
- The *depth* of a binary tree is the maximum level of any leaf in the tree.

To understand level of a node, consider the following figure 11.13. This figure shows the same tree of figure 11.2, discussed so far.

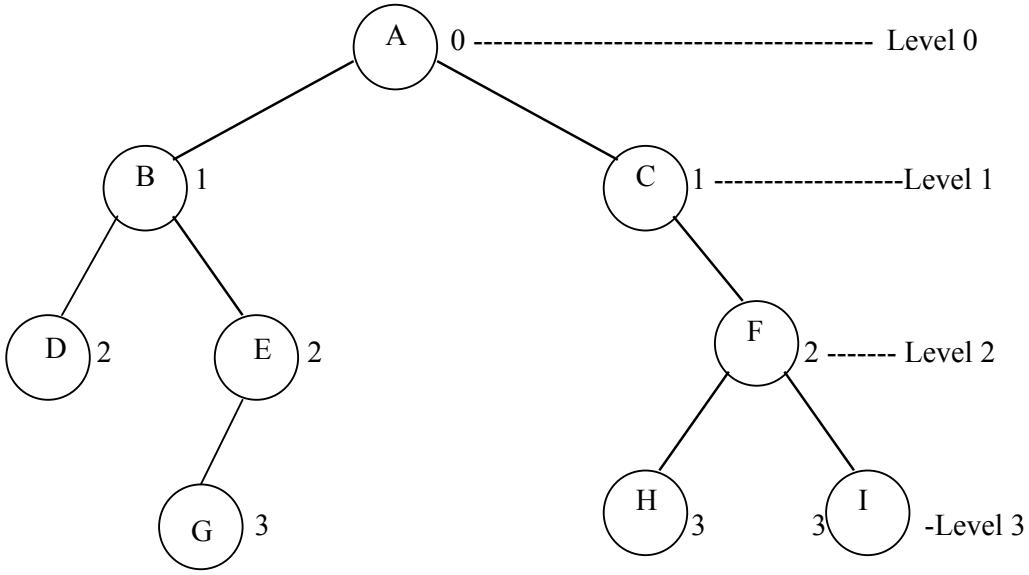


Fig 11.13: Level of nodes of a tree

In the above figure, we have mentioned the level of each node. The figure also shows that the root node has level 0. We have talked in the definition of the level that the level of a node, other than root, is one more than the level of its parent. Now in the tree, the parent of nodes B and C is node A with level 0. Now by definition, the level of B and C will be 1 (i.e. level of A + 1). Now if go downward in the tree and see the nodes D, E and F, the parent node of D and E is B the level of which is 1. So the level of D and E is 2 (i.e. 1 + 1). Similarly the level of F is 2 as the level of its parent (i..e. C) is 1. In the same way, we can easily understand that the level of G, H and I is 3 as the parent nodes of these (E and F) have level 2. Thus we see that tree has multi levels in the downward direction. The levels of the tree increase, as the tree grows downward more. The number of nodes also increases.

Now let's talk why we call it a binary tree. We have seen that each node has a maximum of two subtrees. There might be two, one or no subtree of a node. But it cannot have more than two. So, we call such a tree a binary one due to the fact that a node of it can have a maximum of two subtrees. There are trees whose nodes can have more than two subtrees. These are not binary trees. We will talk about these trees later.

By seeing the level of a tree, we can tell the depth of the tree. If we put level with each node of the binary tree, the depth of a binary tree is the maximum level. In the

figure 11.13, the deepest node is at level 3 i.e. the maximum level is 3. So the depth of this binary tree is 3.

Complete Binary Tree

Now, if there are left and right subtrees for each node in the binary tree as shown below in figure 11.14, it becomes a complete binary tree.

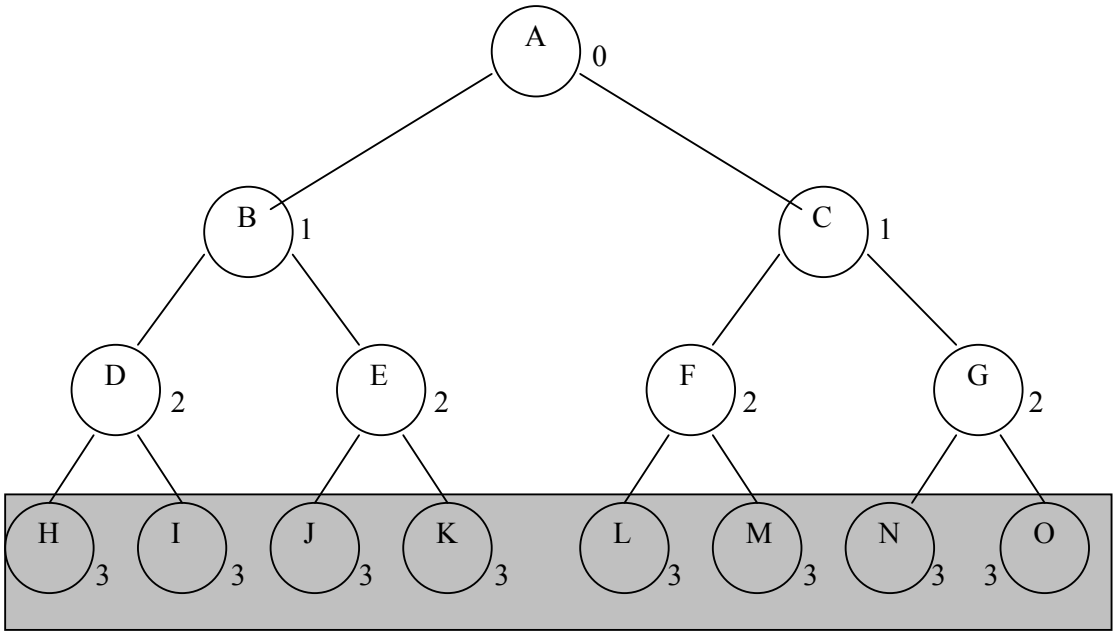


Fig 11.14: A Complete Binary Tree

The definition of the complete binary tree is
"A complete binary tree of depth d is the strictly binary tree all of whose leaves are at level d".
Now look at the tree, the leaf nodes of the tree are at level 3 and are H, I, J, K, L, M, N and O. There is no such a leaf node that is at some level other than the depth level *d* i.e. 3. All the leaf nodes of this tree are at level 3 (which is the depth of the tree i.e. d). So this is a complete binary tree. In the figure, all the nodes at level 3 are highlighted.
We know that the property of the binary tree is that its node can have a maximum of two subtrees, called as left and right subtrees. The nodes increase at each level, as the tree grows downward. In a complete binary tree, the nodes increase in a particular order. If we reconsider the previous tree (figure 11.14), we note that the number of node at level 0 is 1. We can say it 2^0, as 2^0 is equal to 1. Down to this, the node is the level 1 where the number of nodes is 2, which we can say 2^1, equal to 2. Now at the next level (i.e. level 2), the number of nodes is 4 that mean 2^2. Finally the number of nodes at level 3 is 8, which can be called as 2^3. This process is shown pictorially in the following figure 11.15.

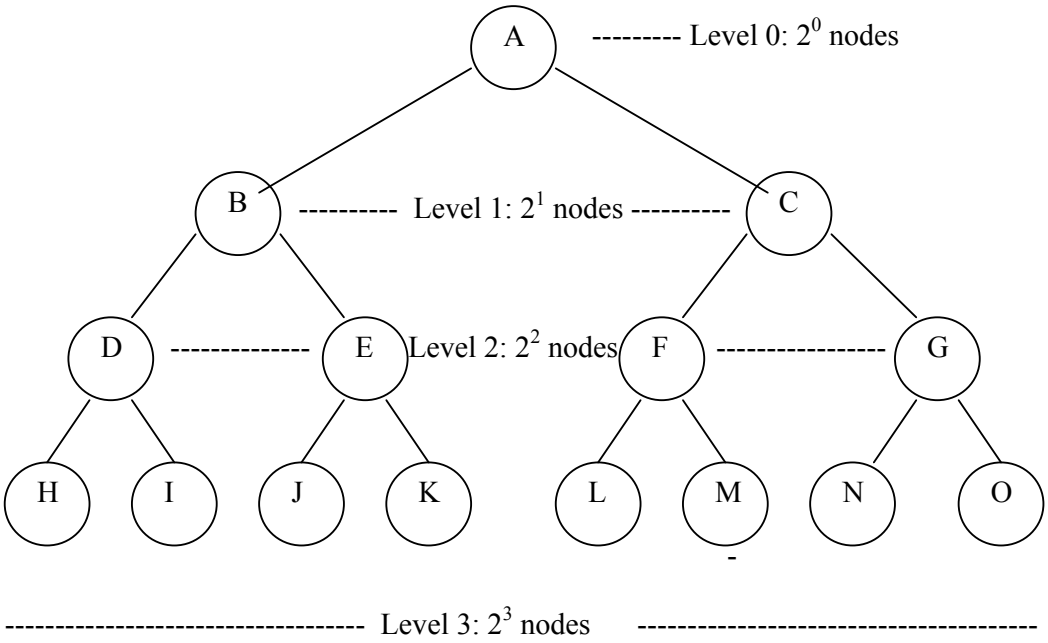


Fig 11.15: Number of nodes at each level in a complete binary tree

By observing the number of nodes at a particular level, we come to the conclusion that the number of nodes at a level is equal to the level number raising to the power of two. Thus we can say that in a complete binary tree at a particular level *k,* the number of nodes is equal to 2^k. Note that this formula for the number of nodes is for a complete binary tree only. It is not necessary that every binary tree fulfill this criterion. Applying this formula to each level while going to the depth of the tree (i.e. d), we can calculate the total number of nodes in a complete binary tree of depth *d* by adding the number of nodes at each level. This can be written as the following summation.

$$2^0 + 2^1 + 2^2 + \ldots\ldots + 2^d = \Sigma^d_{j=0} 2^j = 2^{d+1} - 1$$

Thus according to this summation, the total number of nodes in a complete binary tree of depth *d* will be $2^{d+1} - 1$. Thus if there is a complete binary tree of depth 4, the total number of nodes in it will be calculated by putting the value of *d* equal to 4. It will be calculated as under.

$$2^{4+1} - 1 = 2^5 - 1 = 32 - 1 = 31$$

Thus the total number of nodes in the complete binary tree of depth 4 is 31.

We know that the total number of nodes (leaf and non-leaf) of a complete binary tree of depth d is equal to $2^{d+1} - 1$. In a complete binary tree, all the leaf nodes are at the depth level d. So the number of nodes at level d will be 2^d . These are the leaf nodes. Thus the difference of total number of nodes and number of leaf nodes will give us the number of non-leaf (inner) nodes. It will be $(2^{d+1} - 1) - 2^d$ i.e. $2^d - 1$. Thus we conclude that in a complete binary tree, there are 2^d leaf nodes and $2^d - 1$ non-leaf (inner) nodes.

Level of a Complete Binary Tree

We can conclude some more results from the above mathematical expression. We can find the depth of a complete binary tree if we know the total number of nodes. If we have a complete binary tree with n total nodes, then by the equation of the total number of nodes we can write

$$\text{Total number of nodes} = 2^{d+1} - 1 = n$$

To find the value of d, we solve the above equation as under

$$2^{d+1} - 1 = n$$
$$2^{d+1} = n + 1$$
$$d + 1 = \log_2 (n + 1)$$
$$d = \log_2 (n + 1) - 1$$

After having n total nodes, we can find the depth d of the tree by the above equation. Suppose we have 100,000 nodes. It means that the value of n is 100,000, reflecting a depth i.e. d of the tree will be $\log_2 (100000 + 1) - 1$, which evaluates to 20. So the depth of the tree will be 20. In other words, the tree will be 20 levels deep. The significance of all this discussion about the properties of complete binary tree will become evident later.

Operations on Binary Tree

We can define different operations on binary trees.
If p is pointing to a node in an existing tree, then

- left(p) returns pointer to the left subtree
- right(p) returns pointer to right subtree
- parent(p) returns the father of p
- brother(p) returns brother of p.
- info(p) returns content of the node.

We will discus these methods in detail in next lecture.

Tips

- A priority queue is a variation of queue that does not follow FIFO rule.
- Tree is a non-linear data structure.

- The maximum level of any leaf in a binary tree is called the depth of the tree.
- Other than the root node, the level of each node is one more than the level of its parent node.
- A complete binary tree is necessarily a strictly binary tree but not vice versa.
- At any level k, there are 2^k nodes at that level in a complete binary tree.
- The total number of nodes in a complete binary tree of depth d is $2^{d+1} - 1$.
- In a complete binary tree there are 2^d leaf nodes and $2^d - 1$ non-leaf nodes.

Data Structures

Lecture No. 12

Reading Material

Data Structures And Algorithm analysis in C++ Chapter 4
4.2, 4.3(4.3.2, 4.3.4)

Summary

- Operations on Binary Tree
- Applications of Binary Tree
- Searching for Duplicates
- C++ Implementation of Binary Tree
- Trace of insert

Operations on Binary Tree

In the last lecture, we talked about the uses of binary tree, which is an abstract data type. We discussed about the functions to find the information inside a node, the parent, the siblings(brothers), the left and right children of a node. In this lecture, we will talk about the algorithms and implementation of those functions.
When we discuss about an abstract data type, firstly we focus *what* it can do for us and don't bother about the *how* part. The how part or implementation is thought out later.

While implementing these abstract data types, the implementation is hidden in a class so it is abstract to the user. The user only needs to be aware of the interface. But there can be situations when the users may like to know about the implementation detail, for example, when a data type is performing quite slower than promised.

For now, we start our discussion from the methods of tree data type. Consider a tree has been formed already, following methods will be used to perform different operations on a node of this tree:

Operation	Description
left(p)	Returns a pointer to the left sub-tree
right(p)	Returns a pointer to the right sub-tree
parent(p)	Returns the father node of p
brother(p)	Returns the brother node of p
info(p)	Returns the contents of node p

These methods have already been discussed at the end of the previous lecture, however, few more methods are required to construct a binary tree:

Operation	Description
setLeft(p, x)	Creates the left child node of *p* and set the value *x* into it.
setRight(p, x)	Creates the right child node of *p*, the child node contains the info *x*.

All these methods are required to build and to retrieve values from a tree.

Applications of Binary Tree

Let's take few examples to understand how the tree data type is used and what are its benefits. We will also develop some algorithms that may by useful in future while working with this data type.

Binary tree is useful structure when two-way decisions are made at each point.
Suppose we want to find all duplicates in a list of the following numbers:
14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

This list may comprise numbers of any nature. For example, roll numbers, telephone numbers or voter's list. In addition to the presence of duplicate number, we may also require the frequency of numbers in the list. As it is a small list, so only a cursory view may reveal that there are some duplicate numbers present in this list. Practically, this list can be of very huge size ranging to thousands or millions.

Searching for Duplicates

One way of finding duplicates is to compare each number with all those that precede it. Let's see it in detail.

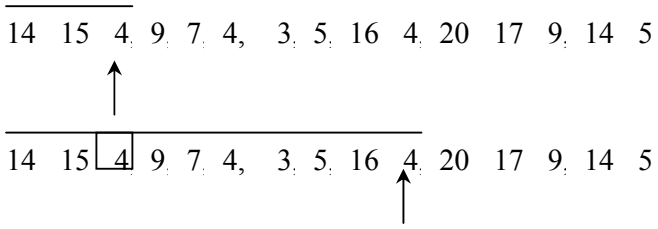


Fig 12.1: Search for Duplicates

Suppose, we are looking for duplicates for the number *4*, we will *start* scanning from the first number *14*. While scanning, whenever we find the number 4 inside, we remember the position and increment its frequency counter by 1. This comparison will go on till the end of the list to get the duplicates or fequence of the number *4*. You might have understood already that we will have to perform this whole scanning of list every time for each number to find duplicates. This is a long and time consuming process.
So this procedure involves a large number of comparisons, if the list of numbers is large and is growing.
A linked list can handle the growth. But it can be used where a programmer has no idea about the size of the data before hand. The number of comparisons may still be large. The comparisons are not reduced after using linked list as it is a linear data

structure. To search a number in a linked list, we have to begin from the start of the list to the end in the linear fashion, traversing each node in it. For optimizing search operation in a list, there is no real benefit of using linked list. On the contrary, the search operation becomes slower even while searching in an array because the linked list is not contiguous like an array and traversing is done in the linked list with the help of pointers.

So, the solution lies in reducing the number of comparisons. The number of comparisons can be drastically reduced with the help of a binary tree. The benefits of linked list are there, also the tree grows dynamically like the linked list.
The binary tree is built in a special way. The first number in the list is placed in a node, designated as the *root* of the binary tree. Initially, both left and right sub-trees of the *root* are empty. We take the next number and compare it with the number placed in the *root*. If it is the same, this means the presence of a duplicate. Otherwise, we create a new tree node and put the new number in it. The new node is turned into the left child of the *root* node if the second number is less than the one in the *root*. The new node is turned into the right child if the number is greater than the one in the *root*.

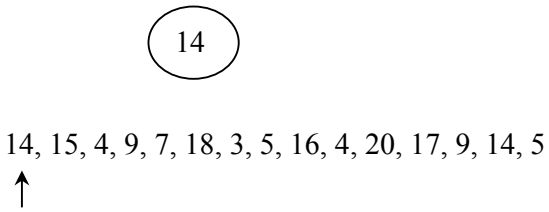


Fig 12.2: First number in the list became the *root*

In the above figure, the first number in the list *14* is placed in a node , making it the *root* of the binary tree. You can see that it is not pointing to any further node. Therefore, its left and right pointers are *NULL* at this point of tree construction time. Now, let's see, how do we insert the next element into it.

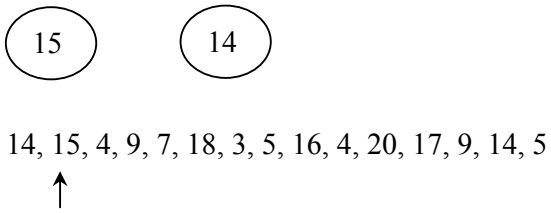
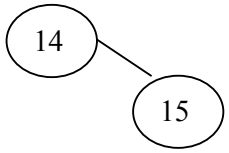


Fig 12.3: A new node is created to insert it in the binary tree

As a first step to add a new node in the tree, we take the next number *15* in the list and compare it with *14*, the number in the *root* node. As they are different, so a new node is created and the number *15* is set into it.

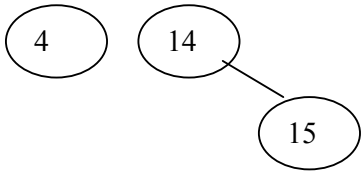


14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Fig 12.4: The second node is added into the tree

The next step is to add this node in the tree. We compare *15*, the number in the new node with *14*, the number in the *root* node. As number *15* is greater than number *14*, therefore, it is placed as right child of the *root* node.

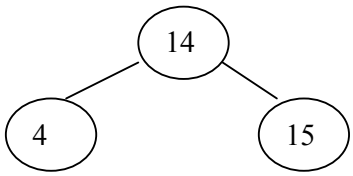
The next number in the list i.e. *4*, is compared with *14*, the number in the *root* node. As the number *4* is less than number *14*, so we see if there is a left child of the *root* node to compare the number *4* with that. At this point of time in the tree, there is no further left child of the *root* node. Therefore, a new node is created and the number *4* is put into it.
The below figure shows the newly created node.



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Fig 12.5: A new node is created and number 4 put into it

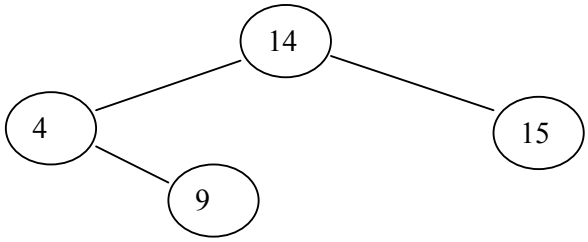
Next, the newly created node is added as the left child of the *root* node. It is shown in the figure below.



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Fig 12.6: The node is added as the left child of the *root* node

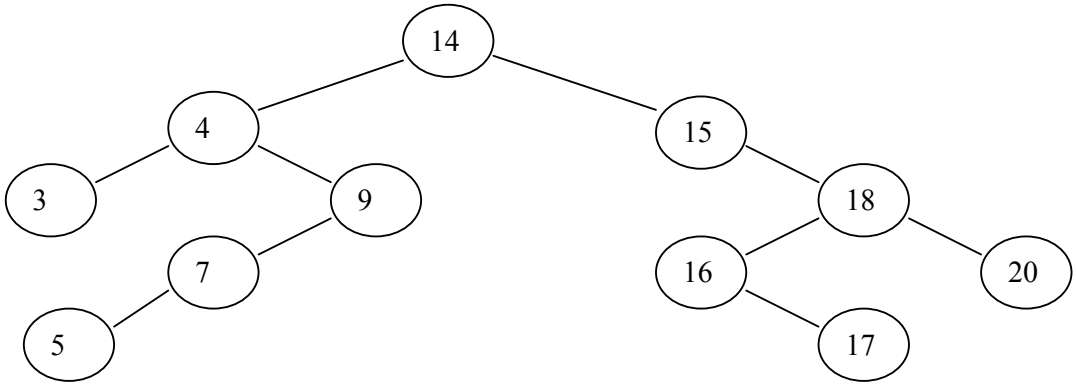
The next number in the list is *9*. To add this number in the tree, we will follow the already defined and experimented procedure. We compare this number first with the number in the *root* node of the tree. This number is found to be smaller than the number in the *root* node. Therefore, left sub-tree of the *root* node is sought. The left child of the *root* node is the one with number *4*. On comparison, number *9* is found greater than the number *4. Therefore*, we go to the right child of the node with number *4*. At the moment, there is no further node to the right of it, necessitating the need of creating a new node. The number *9* is put into it and the new node is added as the right child of the node with number *4*. The same is shown in the figure given below.



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Fig 12.7: A new node is added in the tree

We keep on adding new nodes in the tree in line with the above practiced rule and eventually, we have the tree shown in the below figure.



14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

Fig 12.8: Binary tree of whole list of numbers

It is pertinent to note that this is a binary tree with two sub-nodes or children of each node. We have not seen the advantage of binary tree, the one we were earlier talking about i.e. it will reduce the number of comparisons. Previously, we found that search operation becomes troublesome and slower as the size of list grows. We will see the benefit of using binary tree over linked list later. Firstly, we will see how the tree is

implemented.

C++ Implementation of Binary Tree

See the code below for the file *treenode.cpp*.

```
/* This file contains the TreeNode class declaration. TreeNode contains the
functionality for a binary tree node */
1. #include   <stdlib.h>
2.
3. template   <class  Object>
4.
5. class   TreeNode
6. {
7.      public:
8.        // constructors
9.        TreeNode()
10.      {
11.              this->object   =   NULL;
12.              this->left   =   this->right   =   NULL;
13.      };
14.
15.      TreeNode( Object *   object )
16.      {
17.              this->object   =   object;
18.              this->left   =   this->right   =   NULL;
19.      };
20.
21.      Object *   getInfo()
22.      {
23.              return   this->object;
24.      };
25.
26.      void   setInfo(Object *   object)
27.      {
28.              this->object   =   object;
29.      };
30.
31.      TreeNode *   getLeft()
32.      {
33.              return   left;
34.      };
35.
36.      void   setLeft(TreeNode *   left)
37.      {
38.              this->left   =   left;
39.      };
40.
41       TreeNode *   getRight()
42.      {
43.              return   right;
```

```
44.     };
45.
46.     void   setRight(TreeNode *   right)
47.     {
48.             this->right   =   right;
49.     };
50.
51.     int   isLeaf( )
52.     {
53.             if( this->left  ==  NULL   &&   this->right  ==  NULL )
54.                     return   1;
55.             return   0;
56.     };
57.
58.     private:
59.             Object *   object;
60.             TreeNode *   left;
61.             TreeNode *   right;
62. }; // end class TreeNode
```

For implementation, we normally write a class that becomes a factory for objects of that type. In this case too, we have created a class *TreeNode* to be used to create nodes of the binary tree. As we want to use this class for different data types, therefore, we will make it a *template class*, the line *2* is doing the same thing. Inside the class body, we start with the *private* data members given at the bottom of the class declaration. At line *59*, the *object* is a *private* data element of *Object *,* used to store the tree element (value) inside the node of the tree. *left* is a *private* data member of type *TreeNode**, it is used to store a pointer to the left sub-tree. *right* is a *private* data member of type *TreeNode*,* employed to store a pointer to the right sub-tree.

Now, we go to the top of the class declaration to see the *public* functions. At line *9*, a public parameter-less constructor is declared. The data members have been initialized in the constructor. At line *11*, the *object* data member is initialized to *NULL*. Similarly *left* and *right* data members are initialized to *NULL* at line *12*.

There is another constructor declared at line *15*- that takes *object* value as a parameter to construct a *TreeNode* object with that object value. While the pointers for *right* and *left* sub-trees are pointing to *NULL*.

At line 21, there is method *getInfo()*, which returns the object i.e. the element of the *TreeNode* object.

At line 26, the method *setInfo(Object *)* sets the value of the *object* data member to the value passed to it as the argument.

The method *getLeft()* returns the pointer to the left sub-tree. Similarly, the *getRight()* returns the right sub-tree. Note that both of these methods return a pointer to the object of type *TreeNode*.

The *setLeft(TreeNode *)* method is used to set the pointer *left* to left sub-tree. Similarly, *setRight(TreeNode *)* is used to set the pointer *right* to right sub-tree. Both of these methods accept a pointer of type *TreeNode*.

The *isLeaf()* method at line *51*, is to see whether the current node is a *leaf* node or not. The method returns *1* if it is leaf node. Otherwise, it returns *0*.

Using the above *TreeNode*, nodes for the binary tree can be created and linked up together to form a binary tree. We can write a separate method or a class to carry out the node creation and insertion into tree.
Let's use this class by writing couple of functions. Below is the code of main program file containing the *main()* and *insert()* functions.

```
1. #include   <iostream>
2. #include   <stdlib.h>
3. #include   "TreeNode.cpp"
4.
5. int   main(int   argc,  char *   argv[])
6. {
7.       int  x[]   =   {14,15,4,9,7,18,3,5,16,4,20,17,9,14,5,-1};
8.       TreeNode <int> *   root   =   new   TreeNode<int>();
9.       root->setInfo( &x[0] );
10.      for(int   i = 1; x[i]  >  0;  i++ )
11.      {
12.               insert( root,  &x[i] );
13.      }
14. }
15.
16. void   insert (TreeNode <int> *   root,  int *   info)
17. {
18.      TreeNode <int> *   node   =   new   TreeNode <int> (info);
19.      TreeNode <int>   * p,   * q;
20.      p   =   q   =   root;
21.      while( *info  !=  *(p->getInfo())   &&   q  !=  NULL )
22.      {
23.               p   =   q;
24.               if( *info   <   *(p->getInfo()) )
25.                        q   =   p->getLeft();
26.               else
27.                        q   =   p->getRight();
28.      }
29.
30.      if( *info  ==  *( p->getInfo() )  )
31.      {
32.               cout << "attempt to insert duplicate: " << *info << endl;
33.               delete node;
34.      }
35.      else if( *info < *(p->getInfo()) )
36.               p->setLeft( node );
37.      else
38.      p->setRight( node );
39. } // end of insert
```

We have used the same list of numbers for discussion in this lecture. It is given at line *7* in the code. It is the same, only the last number is *–1*. This is used as delimiter or marker to indicate that the list has finished.
At line *8*, we are creating a new *TreeNode* object i.e. a *root* node as the name implies.

This node will contain an *int* type element as evident from the syntax.
At line *9*, the first number in the list is set into the *root* node. At line *10*, the *for* loop is started, which is inserting all the elements of the list one by one in the tree with the use of the *insert()* function. Most of the time, this loop will be reading the input numbers from the users interactively or from a file. It it is Windows application then a form can by used to take input from the user. In this implementation, our objective is to insert numbers in the tree and see the duplicates, so hard coding the numbers within our program will serve the purpose.
The *insert()* method starts from line *16*. It is accepting two parameters. The first parameter is pointer to a *TreeNode* object, containing a value of *int* type while second is the *info* that is an *int **.
In the first line of the function, line *18*, a new node has been by calling the parameterized constructor of the *TreeNode* class.
Then at line *19*, two pointer variables *p* and *q* are declared.
In line *20*, we are initializing variables *p* and *q* to the *root* node.
In line *21*, the *while* loop is being started, inside the while loop we are checking the equality of the number inside the node (being pointed to by pointer *p)* with the number being passed. The control is entered into the loop, if both the numbers are not equal and *q* is not pointing to *NULL*. This *while* loop will be terminated if the numbers in the two nodes are equal or end of a sub-tree is reached.
At line *23* inside the loop, *p* is assigned the value of *q*.
At line 24 inside the loop, if the number inside the node is smaller than the number in the node pointed to by the pointer *p*. Then at line *25*, we are getting the left sub-tree address (*left*) and assigning it to the pointer *q*.
Otherwise, if this not the case, means the number in the node to be inserted is not smaller than the number in the node pointed to by the pointer *p*. Then at line *27*, the right sub-tree address is retrieved and assigned to pointer *q*.
At line *30*, the comparison is made to see if the both the values are equal (number in the node to be inserted and the number inside the node pointed to by the pointer *p*). In case they are equal, it displays a message with the number informing that a duplicate number has been found. This also means that the upper *while* loop was terminated because of equality of the two numbers. In the line *33*, the newly created node is deleted afterwards.
At line *35*, we are checking if the number in the newly constructed node is less than the number in the node at the end of a sub-tree. If this is so then the newly constructed node is inserted to the left of the tree node. The insertion code is at line *36*.
If the number in the newly constructed node is greater than the number in the tree node, the newly constructed node will be inserted to the right of the tree node as shown on line *38*. To make it clearer, let's see the same thing, the *insert()* method pictorially.

Trace of insert

We will take the tree and the figure, we constructed above. At this time, we want to insert some new numbers in the tree as given in the figure below:

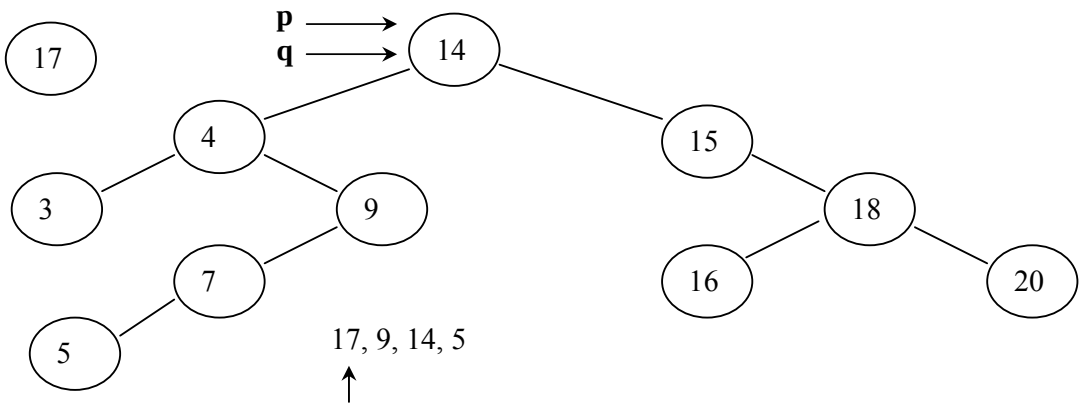


Fig 12.9: Start of insertion of new node in the tree

Initially the pointers *p* and *q* are pointing to the start (*root*) of the tree. We want to insert the number *17* in the tree. We compare the number in the root node (*14*) with number *17*. Because number *17* is greater than the number *14*, so as we did in the while loop in the function above, we will move toward the right sub-tree. In the next picture below, we can see that the pointer q has moved to the right sub-tree of the *root*.

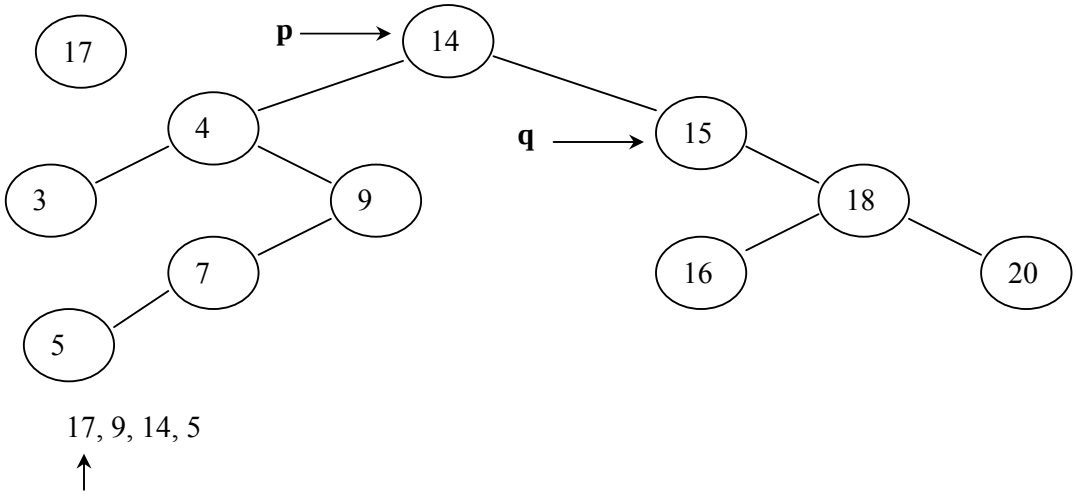


Fig 12.10: Insertion of a new node in progress

After moving the pointer *q* forward, we make the pointer p point to the same node. We do this operation as the first step inside the while loop. It can be seen at line 23 above. So following will be the latest position of pointers in the tree.

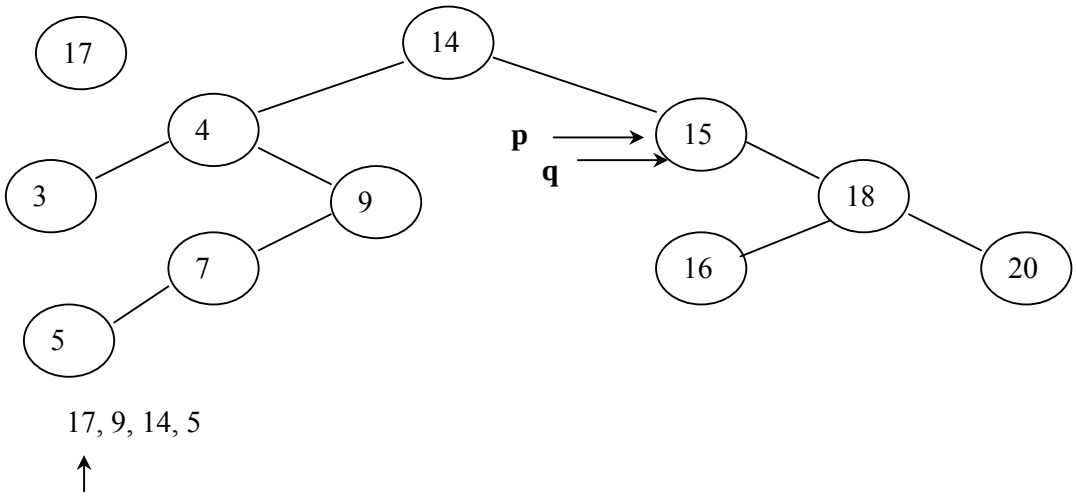


Fig 12.11: Insertion of a new node in progress

Now, the number *15* in the tree node is compared with new number *17*. Therefore, the pointer q is again moved forward to the right child node i.e. the node with number *18*. In the next step, we move forward the *p* pointer also. The following figure depicts the current picture of the tree:

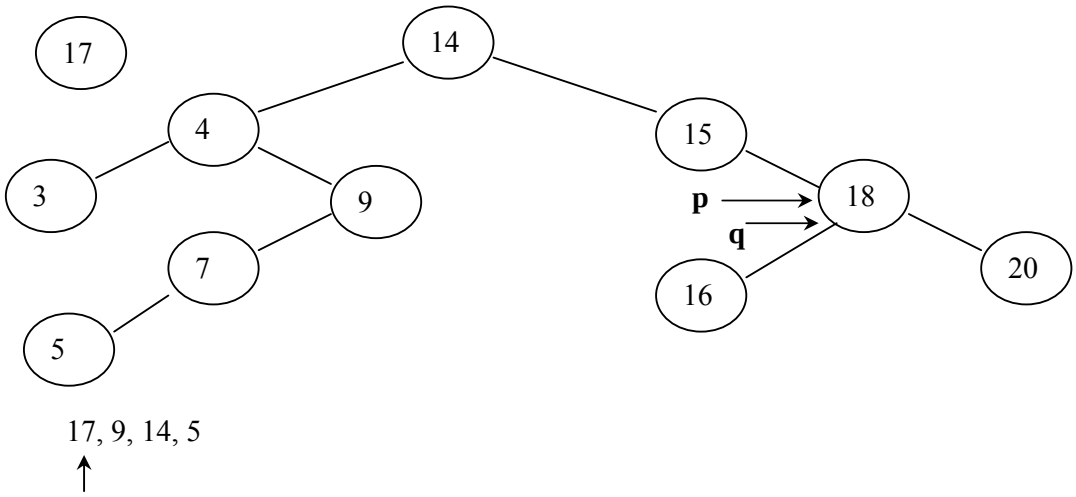


Fig 12.12: Insertion of a new node in progress

The previous process is repeated again (*while* loop is the repeating construct) that the number *17* is compared with the number *18*. This time the left child node is traversed and *q* pointer starts pointing the node with number *16* inside. In the next step, *p* is moved forward to point to the same node as *q*.

The same comparison process starts again, number *17* is found to be greater than the number *16*, therefore, the right child is seek. But we see that the current tree node does not have right child node, so it returns NULL. Following figure depicts it.

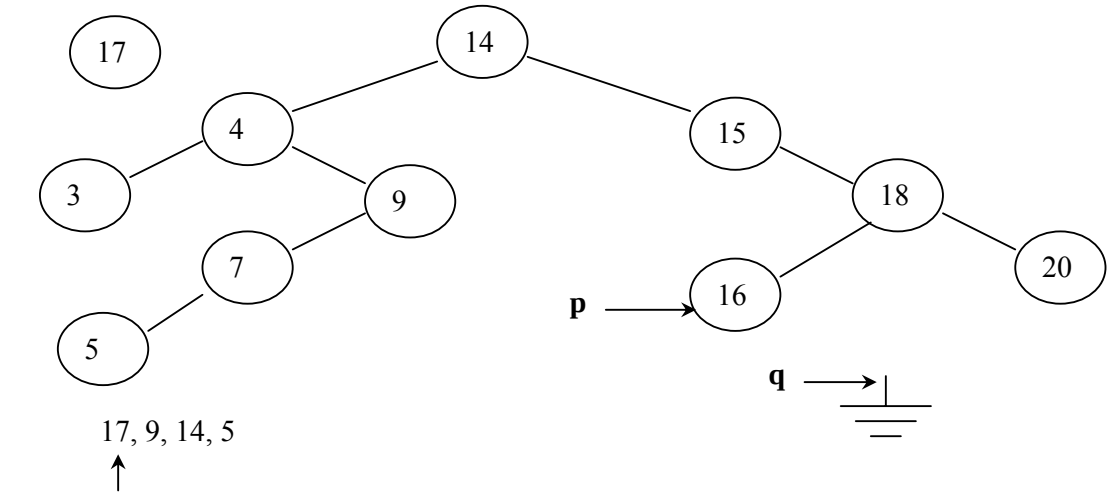


Fig 12.13: Insertion of a new node in progress

Above shown (*q* is pointing to *NULL*) is the condition that causes the *while* loop in the code above to terminate. Later we insert the new node as the right child of the current node.

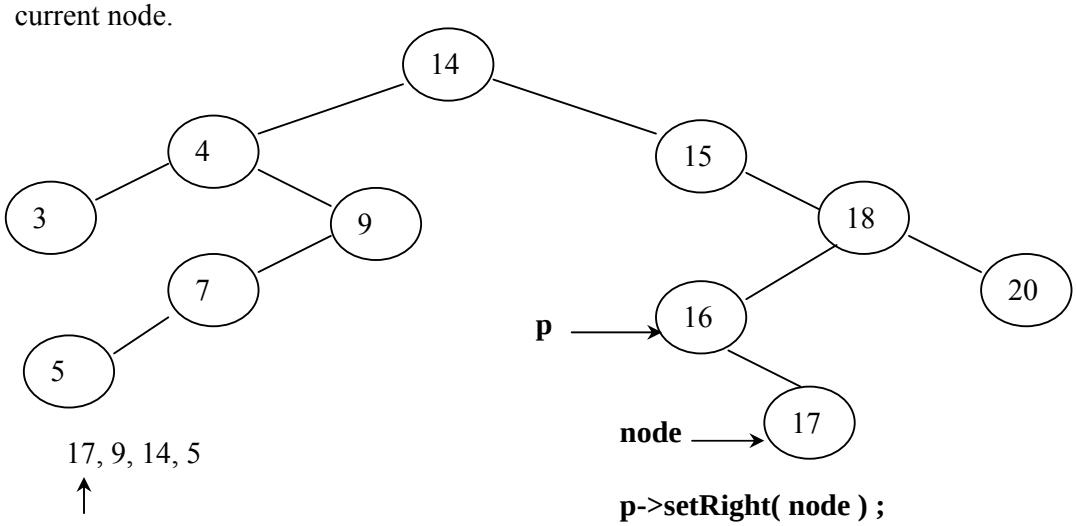


Fig 12.14: A new node is inserted in the tree

It is recommended to execute this algorithm manually to insert the remaining numbers in the tree. This will make the understanding more clear. This tree algorithm is going to be used rigorously in the future and complete understanding of it will make the complex future implementations easier to comprehend. The specific advantages of using binary tree over linked list will be discussed in more detail in the next lecture.

Data Structures

Lecture No. 13

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4

4.3, 4.6

Summary

- Cost of Search
- Binary Search Tree (BST)
- Traversing a Binary Tree
- C++ code
- Example
- Exercise

In the previous lecture, we had written and demonstrated the use of C++ code of *insert* routine for a tree through an example. We also saw how a new node is inserted into a binary tree. If the to-be-inserted number (node) is already in the tree i.e. it matches a number already present in the tree, we display a message that the number is already in the tree. In the last lecture, the advantages of the tree data structure vis-à-vis linked list data structure were also discussed. In a linked list, a programmer has to search the whole list to find out a duplicate of a number to be inserted. It is very tedious job as the number of stored items in a linked list is very large. But in case of tree data structure, we get a dynamic structure in which any number of items as long as memory is available, can be stored. By using tree data structure, the search operation can be carried out very fast. Now we will see how the use of binary tree can help in searching the duplicate number in a very fast manner.

Cost of Search

Consider the previous example where we inserted the number 17 in the tree. We executed a *while* loop in the *insert* method and carried out a comparison in *while* loop. If the comparison is true, it will reflect that in this case, the number in the node where the pointer *p* is pointing is not equal to 17 and also *q* is not NULL. Then we move *p* actually *q* to the left or right side. This means that if the condition of the *while* loop is true then we go one level down in the tree. Thus we can understand it easily that if there is a tree of 6 levels, the while loop will execute maximum 6 times. We conclude from it that in a given binary tree of depth *d,* the maximum number of executions of the *while* loop will be equal to *d*. The code after the *while* loop will do the process depending upon the result of the *while* loop. It will insert the new number or display a message if the number was already there in the tree.

Now suppose we have another method *find*. This method does not insert a new number in the tree. Rather, it traverses a tree to find if a given number is already present in the tree or not. The tree which the *find* method traverses was made in such an order that all the numbers less than the number at the root are in the left sub-tree of the root while the right sub-tree of the root contains the numbers greater than the number at the root. Now the *find* method takes a number *x* and searches out its duplicate in the given tree. The *find* method will return true if *x* is present in the tree. Otherwise, it will return false. This *find* method does the same process that a part of the *insert* method performs. The difference is that the *insert* method checks for a duplicate number before putting a number in the tree whereas the *find* method only finds a number in the tree. Here in the *find* method, the *while* loop is also executed at maximum equal to the number of levels of the tree. We do a comparison at each level of the tree until either *x* is found or *q* becomes NULL. The loop terminates in case, the number is found or it executes to its maximum number, i.e. equal to the number of levels of the tree.

In the discussion on binary tree, we talked about the level and number of nodes of a binary tree. It was witnessed that if we have a complete binary tree with *n* numbers of nodes, the depth *d* of the tree can be found by the following equation:

$$d = \log_2 (n + 1) - 1$$

Suppose we have a complete binary tree in which there are 100,000 nodes, then its depth *d* will be calculated in the following fashion.

$$d = \log_2 (100000 + 1) - 1 = \log_2 (100001) - 1 = 20$$

The statement shows that if there are 100,000 unique numbers (nodes) stored in a complete binary tree, the tree will have 20 levels. Now if we want to find a number *x* in this tree (in other words, the number is not in the tree), we have to make maximum 20 comparisons i.e. one comparison at each level. Now we can understand the benefit of tree as compared to the linked list. If we have a linked list of 100,000 numbers, then there may be 100,000 comparisons (supposing the number is not there) to find a number *x* in the list.

Thus in a tree, the search is very fast as compared to the linked list. If the tree is complete binary or near-to-complete, searching through 100,000 numbers will require a maximum of 20 comparisons or in general, approximately $log_2(n)$. Whereas in a linked list, the comparisons required could be a maximum of *n*.

Tree with the linked structure, is not a difficult data structure. We have used it to allocate memory, link and set pointers to it. It is not much difficult process. In a tree, we link the nodes in such a way that it does not remain a linear structure. If instead of 100,000, we have 1 million or 10 million or say, 1 billion numbers and want to build a complete binary tree of these numbers, the depth (number of levels) of the tree will be log_2 of these numbers. The log_2 of these numbers will be a small number, suppose 25, 30 or 40. Thus we see that the number of level does not increase in such a ratio as the number of nodes increase. So we can search a number *x* in a complete binary tree of 1 billion nodes only in 30-40 comparisons. As the linked list of such a large number grows large, the search of a number in such a case will also get time consuming

process. The usage of memory space does not cause any effect in the linked list and tree data structures. We use the memory dynamically in both structures. However, time is a major factor. Suppose one comparison takes one micro second, then one billion seconds are required to find a number from a linked list (we are supposing the worst case of search where traversing of the whole list may be needed). This time will be in hours. On the other hand, in case of building a complete binary tree of these one billion numbers, we have to make 30-40 comparisons (as the levels of the tree will be 30-40), taking only 30-40 microseconds. We can clearly see the difference between hours and microseconds. Thus it is better to prefer the process of building a tree of the data to storing it in a linked list to make the search process faster.

Binary Search Tree

While discussing the search procedure, the tree for search was built in a specific order. The order was such that on the addition of a number in the tree, we compare it with a node. If it is less than this, it can be added to the left sub-tree of the node. Otherwise, it will be added on the right sub-tree. This way, the tree built by us has numbers less than the root in the left sub-tree and the numbers greater than the root in the right sub-tree. A binary tree with such a property that items in the left sub-tree are smaller than the root and items in the right sub-tree are larger than the root is called a *binary search tree* (BST). The searching and sorting operations are very common in computer science. We will be discussing them many times during this course. In most of the cases, we sort the data before a search operation. The building process of a binary search tree is actually a process of storing the data in a sorted form. The BST has many variations, which will be discussed later. The BST and its variations play an important role in searching algorithms. As data in a BST is in an order, it may also be termed as ordered tree.

Traversing a Binary Tree

Now let's discuss the ways to print the numbers present in a BST. In a linked list, the printing of stored values is easy. It is due to the fact that we know wherefrom, a programmer needs to start and where the next element is. Equally is true about printing of the elements in an array. We execute a *for* loop starting from the first element (i.e. index 0) to the last element of the array. Now let's see how we can traverse a tree to print (display) the numbers (or any data items) of the tree.

We can explain this process with the help of the following example in which we traverse a binary search tree. Suppose there are three nodes tree with three numbers stored in it as shown below.

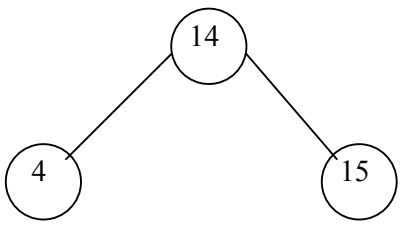


Fig 13.1: A three node binary tree

Here we see that the root node has the number 14. The left sub-tree has only one node i.e. number 4. Similarly the right sub-tree consists of a single node with the number 15. If we apply the permutations combinations on these three nodes to print them, there may be the following six possibilities.

1: (4, 14, 15)
2: (14, 4, 15)
3: (15, 4, 14)
4: (4, 15, 14)
5: (14, 15, 4)
6: (15, 14, 4)

Look at these six combinations of printing the nodes of the tree. In the first combination, the order of printing the nodes is 4-14-15. It means that left subtree-root-right subtree. In the second combination the order is root-left subtree-right subtree. In the third combination, the order of printing the nodes is right subtree-root-left subtree. The fourth combination has left subtree-right subtree-root. The fifth combination has the order root-rigth subtree- left subtree. Finally the sixth combination has the order of printing the nodes right subtree-root-left subtree. These six possibilities are for the three nodes only. If we have a tree having a large number of nodes, then there may increase number of permutations for printing the nodes.

Let's see the general procedure of traversing a binary tree. We know by definition that a binary tree consists of three sets i.e. root, left subtree and right subtree. The following figure depicts a general binary tree.

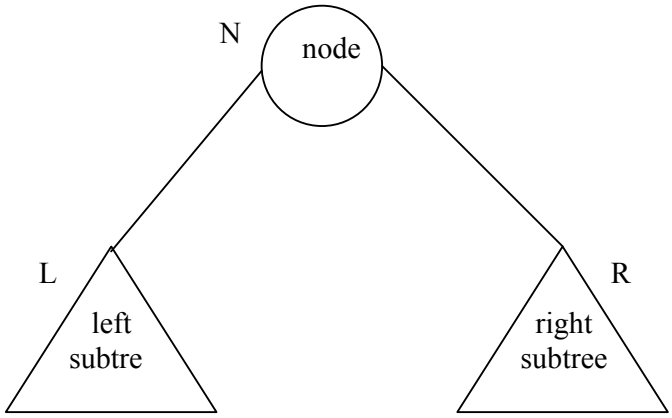


Fig 13.2: A generic binary tree

In this figure, we label the root node with *N*. The left subtree in the figure is in a triangle labeled as *L*. This left subtree may consist of any number of nodes. Similarly the right subtree is enclosed in a triangle having the label *R*. This triangle of right subtree may contain any number of nodes. Following are the six permutations, which we have made for the three nodes previously. To generalize these permutations, we use *N*, *L* and *R* as abbreviations for root, left subtree and right subtree respectively.

1: (L, N, R)
2: (N, L, R)
3: (R, L, N)
4: (L, R, N)
5: (N, R, L)
6: (R, N, L)

In these permutations, the left and right subtrees are not single nodes. These may consist of several nodes. Thus where we see *L* in the permutations, it means traversing the left subtree. Similarly *R* means traversing the right subtree. In the previous tree of three nodes, these left and right subtrees are of single nodes. However, they can consist of any number of nodes. We select the following three permutations from the above six. The first of these three is (N, L, R), also called as preorder traversal. The second permutation is (L, N, R) which we called inorder traversal. Finally the third permutation, also termed as postorder traversal is (L, R, N). Now we will discuss these preorder, inorder and postorder traversal in detail besides having a look on their working. We will also see the order in which the numbers in the tree are displayed by these traversing methods.

C++ code

Let's write the C++ code for it. Following is the code of the *preorder* method.

```
void preorder(TreeNode<int>* treeNode)
{
   if( treeNode != NULL )
   {
      cout << *(treeNode->getInfo())<<" ";
      preorder(treeNode->getLeft());
      preorder(treeNode->getRight());
   }
}
```

In the arguments, there is a pointer to a *TreeNode*. We may start from any node and the pointer of the node will be provided as argument to the *preorder* method. In this method, first of all we check whether the pointer provided is *NULL* or not. If it is not *NULL,* we print the information stored in that node with the help of the *getInfo()* method. Then we call the *getLeft()* method that returns a pointer of left node, which may be a complete subtree. With the help of this method, we get the root of that subtree. We call the *preorder* method again passing that pointer. When we return from that, the *preorder* method is called for the right node. Let's see what is happening in this method. We are calling the *preorder* method within the *preorder* method. This is actually a recursive call. Recursion is supported in C++ and other languages. Recursion means that a function can call itself. We may want to know why we are doing this recursive call. We will see some more examples in this regard and understand the benefits of recursive calls. For the time being, just think that we are provided with a tree with a root pointer. In the preorder, we have to print the value of the root node. Don't think that you are in the preorder method. Rather keep in mind

that you have a preorder function. Suppose you want to print out the left subtree in the preorder way. For this purpose, we will call the preorder function. When we come back, the right subtree will be printed. In the right subtree, we will again call the preorder function that will print the information. Then call the preorder function for the left subtree and after that its right subtree. It will be difficult if you try to do this incursion in the mind. Write the code and execute it. You must be knowing that the definition of the binary tree is recursive. We have a node and left and right subtrees. What is left subtree? It is also a node with a left subtree and right subtree. We have shown you how the left subtree and right subtree are combined to become a tree. The definition of tree is itself recursive. You have already seen the recursive functions. You have seen the factorial example. What is factorial of N? It is N multiplied by N-1 factorial. What is N-1 factorial? N-1 factorial is N-1 multiplied by N-2 factorial. This is the recursive definition. For having an answer, it is good to calculate the factorial of one less number till the time you reach at the number 2. You will see these recursions or recursive relations here and also in mathematic courses. In the course of discrete mathematics, recursion method is used. Here we are talking about the recursive calls. We will now see an example to understand how this recursive call works and how can we traverse the tree using the recursion. One of the benefits of recursion that it prints out the information of all the nodes without caring for the size of the tree. If the tree has one lakh nodes, this simple four lines routine will print all the nodes. When compared with the array printing, we have a simple loop there. In the link list also, we have a small loop that executes till the time we have a next pointer as NULL. For tree, we use recursive calls to print it.

Here is the code of the *inorder* function.

```
void inorder(TreeNode<int>* treeNode)
{
   if( treeNode != NULL )
   {
      inorder(treeNode->getLeft());
      cout << *(treeNode->getInfo())<<" ";
      inorder(treeNode->getRight());
   }
}
```

The argument is the same as in the preorder i.e. a pointer to the *TreeNode*. If this node is not NULL, we call *getLeft()* to get the left node and call the *inorder* function. We did not print the node first here. In the inorder method, there is no need to print the root tree first of all. We start with the left tree. After completely traversing the complete left tree, we print the value of the node. In the end, we traverse the right subtree in recursion.

Hopefully, you have now a fair knowledge about the postorder mechanism too. Here is the code of the postorder method.

```
void postorder(TreeNode<int>* treeNode)
{
   if( treeNode != NULL )
   {
      postorder(treeNode->getLeft());
      postorder(treeNode->getRight());
      cout << *(treeNode->getInfo())<<" ";
   }
}
```

In the postorder, the input argument is a pointer to the *TreeNode*. If the node is not NULL, we traverse the left tree first. After that we will traverse the right tree and print the node value from where we started.

As all of these above routines are function so we will call them as:

```
cout << "inorder: ";
preorder( root);

cout << "inorder: ";
inorder( root );

cout << "postorder: ";
postorder( root );
```

Here the root represents the root node of the tree. The size of the tree does not matter as the complete tree will be printed in *preorder*, *inorder* and *postorder*. Let's discuss an example to see the working of these routines.

Example

Let's have a look on the following tree.

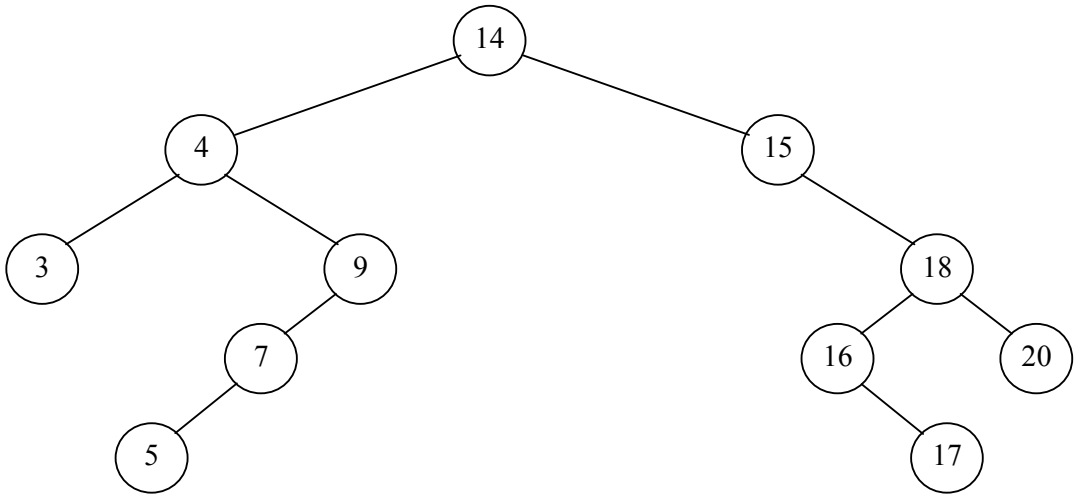


Fig 13.3: Preorder: 14 4 3 9 7 5 15 18 16 17 20

This is the same tree we have been using previously. Here we want to traverse the tree. In the bottom of the figure, the numbers are printed with the help of preorder method. These numbers are as 14 4 3 9 7 5 15 18 16 17 20. Now take these numbers and traverse the tree. In the preorder method, we print the root, followed by traversing of the left subtree and the right subtree respectively. As the value of the root node is 14, so it will be printed first of all. After printing the value of the root node, we call the preorder for the left node which is 4. Forget the node 14 as the root is 4 now. So the value 4 is printed and we call the preorder again with the left sub tree i.e. the node with value 3. Now the root is the node with value 3. We will print its value before going to its left. The left side of node with value 3 is NULL. Preorder will be called if condition is false. In this case, no action will be taken. Now the preorder of the left subtree is finished. As the right subtree of this node is also NULL, so there is no need of any action. Now the left subtree of the node with value 4 is complete. The method preorder will be called for the right subtree of the node with value 4. So we call the preorder with the right node of the node with value 4. Here, the root is the node with value 9 that is printed. We will call its left subtree where the node value is 7. It will be followed by its left subtree i.e. node 5 which will be printed.

In the preorder method, we take the root i.e. 14 in this case. Its value is printed, followed by its left subtree and so on. This activity takes us to the extreme left node. Then we back track and print the right subtrees.

Let's try to understand the inorder method from the following statement.

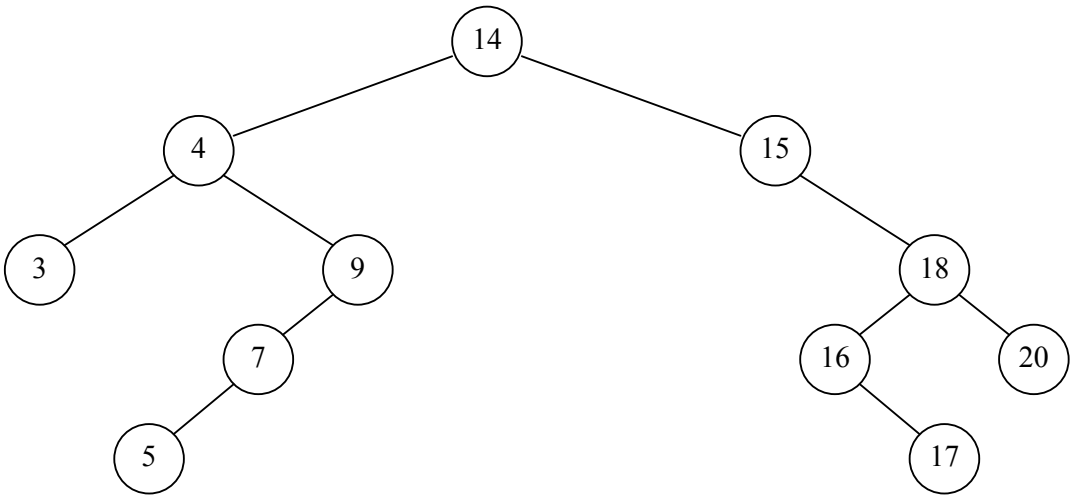


Fig 13.4: Inorder: 3 4 5 7 9 14 15 16 17 18 20

When we call the inorder function, the numbers will be printed as 3 4 5 7 9 14 15 16 17 18 20. You might have noted that these numbers are in sorted order. When we build the tree, there was no need of sorting the numbers. But here we have the sorted numbers in the inorder method. While inserting a new node, we compare it with the existing ones and place the new node on the left or right side. So during the process of running the inorder on the binary tree, a programmer gets the numbers sorted. Therefore we have a sorting mechanism. If we are given some number, it will not be difficult to get them sorted. This is a new sorting algorithm for you. It is very simple. Just make BST with these numbers and run the inorder traversal. The numbers obtained from the inorder traversal are sorted.

In the inorder method, we do not print the value of node first. We go to traverse its left subtree. When we come back from the left subtree and print the value of this node. Afterwards, we go to the right subtree. Now consider the above tree of **figure 13.4**. If we start from number 14, it will not be printed. Then we will go to the left subtree. The left subtree is itself a tree. The number 4 is its root. Now being at the node 4, we again look if there is any left subtree of this node. If it has a left subtree, we will not print 4 and call the inorder method on its left subtree. As there is a left subtree of 4 that consists a single node i.e. 3, we go to that node. Now we call the inorder of node 3 to see if there is a subtree of it. As there is no left subtree of 3, the *if* statement that checks if the node is not NULL will become false. Here the recursive calls will not be executed. We will come back from the call and print the number 3. Thus the first number that is printed in the inorder traversal is 3. After printing 3, we go to its right subtree that is also NULL. So we come back to node 3. Now as we have traversed the left and right subtrees of 3 which itself is a left subtree of 4, thus we have traversed the left subtree of 4. Now we print the value 4 and go to the right subtree of 4. We come at node 9. Before printing 9 we go to its left subtree that leads us to node 7. In this subtree with root 7, we will not print 7. Now we will go to its left subtree which is node 5. We look for the left subtree of 5 which is NULL. Now we print the value 5. Thus, we have so far printed the numbers 3, 4 and 5. Now we come back to 7. After traversing its left subtree, we print the number 7. The right subtree of

7 is NULL. Thus finally, we have traversed the whole tree whose root is 7. It is a left subtree of 9. Now, we will come back to 9 and print it (as its left subtree has been traversed). The right subtree of 9 is NULL. So there is no need to traverse it. Thus the whole tree having 9 as its root has been traversed. So we come back to the root 4, the right subtree of which is the tree with root 9. Resultantly, the left and right subtree of 4 have been traversed now. Thus we have printed the numbers 3, 4, 5, 7 and 9. Now from here (i.e. node 4) we go to the node 14 whose left subtree has been traversed. Now we print 14 and go to its right subtree. The right subtree of 14 has 15 as the root. From here, we repeat the same process earlier carried out in case of the left subtree of 14. As a result of this process, the numbers 15, 16, 17, 18 and 20 are printed. Thus we get the numbers stored in the tree in a sorted order. And we get the numbers printed as 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20. This sorted order is only in the inorder traversal.

When we apply the post order traversal on the same tree, the numbers we get are not in sorted order. The numbers got through postorder traversal are in the following order:

3 5 7 9 4 17 16 20 18 15 14

We see that these are not in a sorted order.

We know that every function has a signature. There is return type, function name and its parameter list in the signature of a function. Then there comes the body of the function. In the body of a function, there may be local variables and calls to some other functions. Now if a statement in the body of a function is calling the same function in which it exists, it will be termed as a recursive call. For example if a statement in function *A* is calling the function *A*, it is a recursive call and the process is called recursion.

Now let's see how recursion is carried out. We know that the function calls are implemented through stacks at run time. Let's see what happens when there is a recursive call. In a recursive call that means calling a function itself makes no difference as far as the call stack is concerned. It is the same as a function *F* calls a function *G*, only with the difference that now function *F* is calling to function *F* instead of *G*. The following figure shows the stack layout when function *F* calls function *F* recursively.

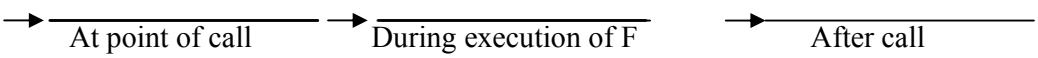


Fig 13.5

When *F* is called first time, the parameters, local variables and its return address are put in a stack, as some function has called it. Now when *F* calls *F,* the stack will increase the parameters, local variables and return address of *F* comes in the stack again. It is the second call to *F*. After coming back from this second call, we go to the state of first call of *F*. The stack becomes as it was at first call. In the next lecture, we will see the functionality of recursive calls by an example.

Exercise

Please find out the preorder, inorder and postorder traversal of the tree given below:

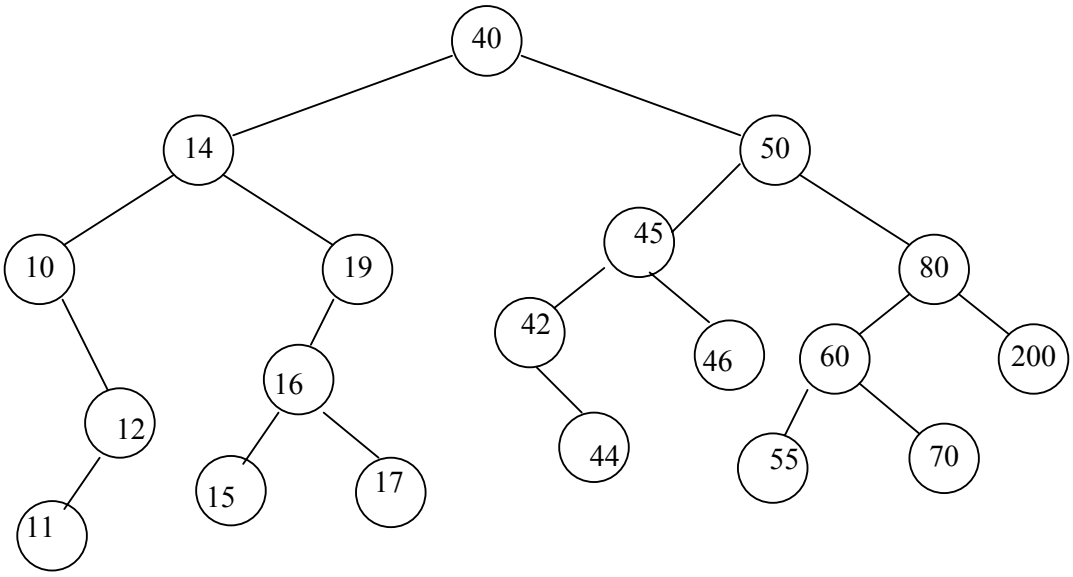


Fig 13.6

Data Structures

Lecture No. 14

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4

4.3, 4.6

Summary

- Recursive Calls
- Preorder Recursion
- Inorder Recursion
- Non-Recursive Traversal
- Traversal Trace

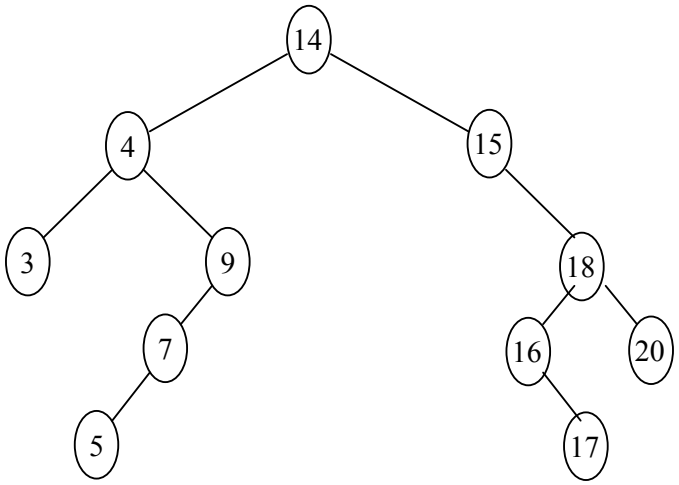
We discussed the methods of traversal of the binary tree in the previous lecture. These methods are- preorder, inorder and postorder. It was witnessed that in the C++ code that the coding of these methods becomes very short with the use of recursive calls. The whole tree can be traversed with a few lines of code. We also demonstrated the benefits of the methods with the help of an example in which a tree was traversed by preorder, inorder and postorder methods. Implementation of the recursion also came under discussion in the previous lecture.

Recursive Calls

We know that function calls are made with the help of stacks. When a function calls some other function, the parameters and return address of the function is put in a stack. The local variables of the function are also located at the stack and the control is passed to the called function. When the called function starts execution, it performs its job by using these parameters and local variables. When there in the function there comes a return statement or when the function ends then the return address, already stored in the stack, is used and control goes back to the next statement after the calling function. Similarly when a function is calling to itself, there is no problem regarding the stack. We know, in recursion a function calls to itself instead of calling some other function. Thus the recursion is implemented in the way as other function calls are implemented.

Preorder Recursion

Now let's see the preorder traversal with respect to the recursion. We will see what changes happen when preorder function calls itself recursively. Consider the following binary search tree used in our previous example.




```
preorder(14)
14
..preorder(4)
4
....preorder(3)
3
......preorder(NULL)
......preorder(NULL)
....preorder(9)
9
......preorder(7)
7
........preorder(5)
5
..........preorder(NULL)
..........preorder(NULL)
........preorder(NULL)
......preorder(NULL)
```

Fig 14.1 (a) : preorder recursion

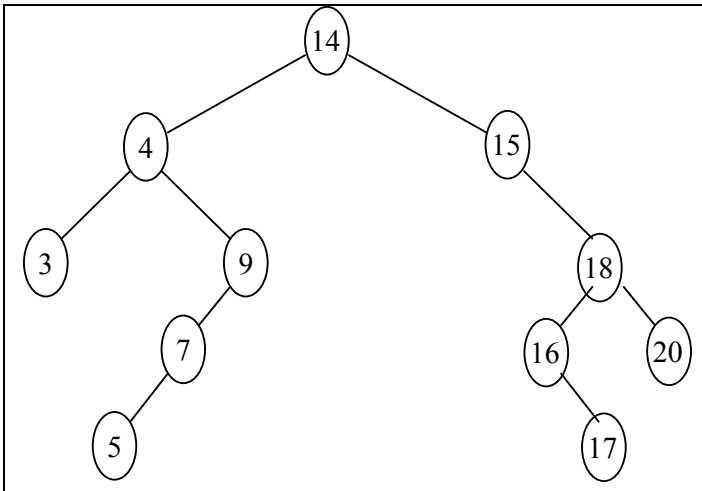
For the preorder traversal, we call the preorder function and pass it the root node of the tree (i.e. 14) as a parameter. We have discussed in the preorder traversing that preorder method first prints the value of the node that is passed to it. So number 14 is printed. After it, the preorder method calls itself recursively. Thus we will call the preorder and give it the left subtree of 14 (actually the root node of left subtree) as an argument. So this call will be *preorder (4)*. As it is preorder traversing, the value of the root node i.e. 4 will be printed. Now the preorder method looks if there is a left subtree of the node (i.e.4). If it has a left subtree, the preorder function will call itself again. The root node of this left subtree will be passed as an argument to the new function call. Here it is 3, so the number 3 is printed. Thus by the previous three calls (i.e. preorder(14), preorder(4) and preorder(3)) we have printed the values 14, 4 and 3. Now the left subtree of 3 is NULL. We know that in preorder method, first of all we check whether the node is NULL or not. If it is not NULL, the recursion process continues. It means that we print the value of the node and call the preorder function again. If the node is NULL, the process ends.

There is always a terminating condition in recursive call or recursive procedure. In the previous lecture, while trying to end the factorial recursion, we defined the condition that the factorial of zero is 1. When the factorial of *n* starts, the factorial is called again and again by decreasing the value of *n* by 1. So when we reach at 0, there is no further recursive call for the factorial as we have set the condition for it i.e. the factorial of 0 is equal to 1. Thus the recursion ends. Equally is true about the preorder process. When a node is NULL, then 'if statement' becomes false and we exit the function (see code of preorder function in previous lecture).

In the above figure, the preorder process that started from node 14, came to an end at node 3 as the left subtree of 3 is NULL. Now this preorder process will make no further recursive call. We will come back from here by using the return address of the

call *preorder (NULL)* from the stack. If we see the stack, this fourth call will come back to the third call i.e. *preorder (3)*. The left subtree of 3 has ended. Now we have to traverse the right subtree of 3. Here the right subtree of 3 is also NULL. So the traversing of tree with node 3 as root has been completed. Now we return from the call *preorder (3)* and reach one level backward at the node 4. The left subtree of node 4 has been traversed. Now we go to traverse the right subtree of 4. The preorder call for the right subtree has started. This started from the call *preorder (9)*.Note in the figure that this call has made at the same level as was for node 3. It is because we are traversing the right subtree of the same tree that has node 4 as its root and 3 was its left subtree. After printing the number 9 we go to call the preorder process for the left subtree of 9. That is *preorder (7)*. When we go to 7, we print it and call the preorder process for its left subtree i.e. *preorder (5)*. Now after printing the number 5, the preorder is applied to its left subtree i.e. NULL. So the preorder traversal of left subtree of 5 ends this way. Now the preorder is applied to the right subtree of 5 which is also NULL. Thus the next two preorder calls (one for left subtree of 5 and one for right subtree of 5) after the call *preorder (5)* are *preorder (NULL)*. Here the returning address takes the control back to the call *preorder (7)* through the use of stack. The left subtree of 7 has been traversed and now we go to the right subtree of it i.e. NULL. Thus there is a call *preorder (NULL)*. So we come back to node 9 completing the traversal of its left subtree. Now we traverse the right subtree of 9 which again happens to be NULL. So there is one more call of preorder with NULL. The number of dots with the preorder calls in the figure describes the state of the stack. When we go ahead to traverse a node the dots increase in figure as the call made is put on the stack. When we come back to a node, the returning address is popped from the stack and we mention it by decreasing a dot in the figure.

After these four calls of preorder with NULL as argument, we return to the node 4. We have completed the tree (left and right subtree) with node 4 as root. As seen in the figure, this tree (having node 4 as root) is a left subtree of node 14. The numbers that we got printed upto now are 14, 4, 3, 9, 7 and 5. So after traversing the left subtree of 14, we come back to node 14 and start to traverse its right subtree. We traverse the right subtree by applying the recursive calls of preorder in the way as done while traversing the left subtree of 14. The recursive calls involved in it are shown in the following figure.



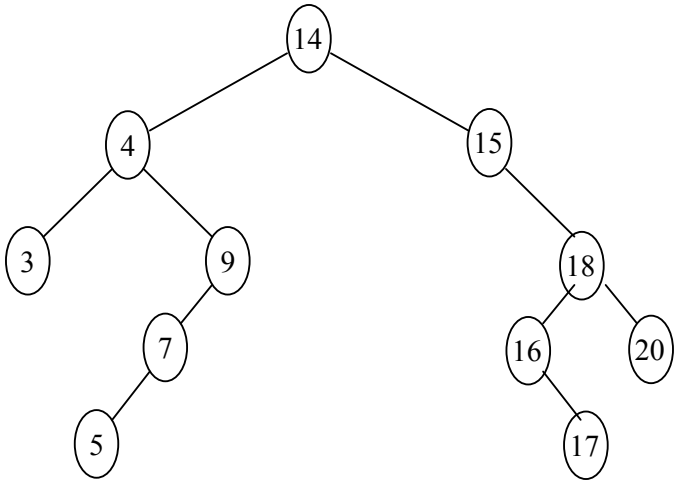
....preorder(15)
15
....preorder(NULL)
....preorder(18)
18
.....preorder(16)
16
.....preorder(NULL)
.....preorder(17)
17
.....preorder(NULL)
.....preoder(NULL)
.....preorder(20)
20
.....preorder(NULL)
.....preorder(NULL)

Fig 14.1 (b): preorder recursion

Now we are at node 14 and know that in preorder, the value of the node is printed first before going to its left subtree. And after traversing the left subtree, a programmer comes back and goes to traverse the right subtree. As we have traversed the left subtree of 14 and have come back to 14, now we will traverse the right subtree of 14. In the right subtree, we reach at node 15. We print the value 15 and look at its left subtree. The left subtree is NULL so the call to the preorder method is *preorder (NULL)*. This call makes the condition of '*if* statement' false and the recursive calls ends on this side. Afterwards, we go to the right subtree of 15. The call *preorder (18)* is made, followed by printing of the value 18. Then we go to the left subtree of it that is node 16. After calling *preorder(16)* and printing it, we go to the left subtree of 16 which is NULL. Now we will move to right subtree of 16 and the call *preorder (17)* prints the number 17. After it, the two next calls are *preorder (NULL)* as the left and right subtree of 17 are NULL. Thus after traversing the left and right subtree of 16 (which itself is a left subtree of 18), we return to node 18. Then we go to the right subtree of 18 and reach at node 20. So *preorder (20)* prints the value 20. There are again two *preorder (NULL)* calls as the left and right subtree of 20 are NULL. The preorder recursive call ends here. Now we go back and reach at node 14 whose left and right subtrees have been traversed. That means the whole tree having 14 as the root has traversed.

Inorder Recursion

Now let's see the inorder recursive calls for traversing a tree. It is not different from the preorder. The pattern of recursive calls will be changed as in the inorder we traverse the left subtree first before printing the root. Afterwards, the right subtree is traversed. The following figures (Fig 14.2(a) and 14.2(b)) explains this phenomenon of inorder recursion by showing the recursive calls.



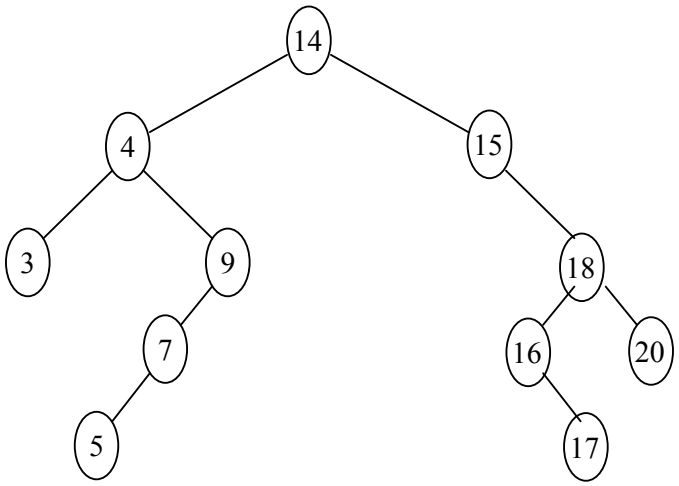

```
inorder(14)
..inorder(4)
....inorder(3)
......inorder(null)
3
......inorder(null)
4
....inorder(9)
......inorder(7)
........inorder(5)
..........inorder(null)
5
..........inorder(null)
7
........inorder(null)
9
......inorder(null)
14
```

Fig 14.2 (a): inorder recursion

We start the inorder with the call inorder (14) as the root of the tree is 14. Now in the inorder traversing, a programmer has to traverse the left subtree before printing the value of the root node and then going to the right subtree. So the call to the left subtree of 14 i.e. *inorder (4)* will lead to the node 4. At the node 4, the same process of calling its left subtree, before printing its value, will be applied and we reach at node 3 that is the left subtree of 4. From the node 3, we go to its left subtree. This left subtree is NULL. Here in the inorder method, we have the same terminating condition as that seen in the preorder i.e. we will not make further recursive calls if there becomes a NULL node in the method call. We end the recursion at that node and come back. Now when we come back from the NULL left subtree of 3 this shows that we have visited the left subtree of 3. So the value of the node i.e. 3 is printed. Now we go to the right subtree of 3. Here the inorder call is *inorder (NULL)*. This call will make the *if* statement, in the inorder method, false, leading to the end of the recursion. Thus we have visited the whole tree whose root is node 3. So we come back to node 4. As the left subtree of node 4 has been visited, we print the value of node i.e. 4. Thus we have printed the numbers 3 and 4. Now we go to the right subtree of 4. The right subtree of 4 is the node 9. Now we send 9 as an argument to inorder method. So there is a call *inorder (9)*. As it is inorder traversing, we go to traverse its left subtree before printing the number 9. We reach the node 7 and make a call *inorder (7)*. Later, we go to the left subtree of 7 i.e. the node 5. Now we visit the left subtree of 5 which is NULL. So here the inorder recursion ends and we come back to node 5, print it and go to the right subtree of 5. The right subtree of 5 is also NULL. So ending recursion here, we have finally traversed the tree with 5 as its root. After traversing it, we come back to the node 7. After visiting the left subtree of 7, we print 7 and go to the right subtree of 7 which is NULL.

After the call *inorder (NULL)* we have traversed the tree having 7 as the root and it is a left subtree of 9. Thus we have visited the left subtree of 9. We print the number 9 before going to traverse its right subtree. The right subtree of 9 is NULL so the recursion ends here and we come back to the node 4. The tree having node 4 as root has been traversed now. This tree is a left subtree of node 14. Thus after traversing it we come back to node 14 and print it (as we have traversed its left subtree i.e. tree with 4 as root). It is evident here that before going to traverse the right subtree of 14, we have printed the numbers 3, 4, 5, 7, 9 and 14.

Now after printing the number 14, we go to traverse the right subtree of 14. The traversal of this right subtree with 15 as a root, is being carried out in the way, we traversed the left subtree. In the right subtree, we reach at node 15 and look at its left subtree before printing it. This left subtree is NULL. The call *inorder (NULL)* makes the condition of *if* statement false and we come back to the root node i.e. 15. We print this number and go to the right subtree of 15. The right subtree of 15 is also a tree the root node of which is 18. From this root node i.e. 18, we go to its left subtree and reach at node 16. NULL being the left subtree of 16, makes us to return to 16 and print it (as its left subtree has been traversed, which is NULL). After printing 16, we go to the right subtree of 16. This leads us to node 17. As we are traversing the tree by inorder process, so before printing the value 17, it will be necessary to go to traverse its left subtree. Its left subtree is NULL. So after the call *inorder (NULL)* we return to 17 and print it. Now we look at the right subtree of 17 that is also NULL. So again there is a call *inorder (NULL)* which ends the recursive calls for this tree. Thus we have traversed the whole tree having node 17 as root and it is a right subtree of 16. We have already traversed the left subtree of 16.




```
..inorder(15)
....inorder(null)
15
....inorder(18)
......inorder(16)
........inorder(null)
16
........inorder(17)
..........inorder(null)
17
..........inorder(null)
18
......inorder(20)
........inorder(null)
20
........inorder(null)
```

Fig 14.2 (b): inorder recursion

So we have traversed the tree with node 16 as its root. This tree is a left subtree of node 18. After traversing the left subtree of 18, we print the number 18 and go to its right subtree and reach at node 20. Now we go to the left subtree of 20 i.e. NULL and come back to 20 and print it. Then go to its right subtree, which is also NULL that ends the recursion. Thus we have traversed the tree having node 20 as its root. This tree (having 20 as root) is a right subtree of 18 which itself is a right subtree of 15. We have already traversed the left subtree of 15 and gone through the whole tree having 15 as root. This tree is a right subtree of the main tree with 14 as the root. We have already traversed its left subtree. So we have traversed the whole tree by the inorder process. From the figure 14.3 we see that by the inorder traversal we have printed the numbers in the order as below

3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20

We have successfully demonstrated the preorder and inorder traversal of a tree by using recursion method. The postorder traversal has been left for you as an exercise. You can do it with the same above tree or build a tree on your own and traverse it.

The tree data structure by nature is a recursive data structure. In the coming lecture, we will see that most of the methods we write for tree operations are recursive. From the programming point of view, the recursion is implemented internally by using call stack. This stack is maintained by the run time environment. The recursion is there in the machine code generated by the compiler. It is the programmer's responsibility to provide a terminating condition to stop the recursion. Otherwise, it will become an infinite recursion. If we do not put a terminating condition, the recursive calls will be continued and the call stack will go on increasing. We know that when a program executes, it becomes a process and some memory is allocated to it by the system for its call stack. This memory has a limit. Thus when the recursive calls do not end, there is no memory left to be used for increasing call stack after a certain time. This will crash the program or the program will halt. So we should be very careful while using the recursive calls and ensure the provision of a terminating condition in the recursive calls.

Non Recursive Traversal

We can also carry out these traversing with the help of non-recursive calls. Think for a few moments that how can we keep track of going left or going right or coming back to the node with out using the recursion.. In the preorder, inorder and postorder traversal, we visit nodes at different level and then return. We know the way to comeback or go further deep. Now if we are not using the recursion, how these things can be managed? We have to manage all this in our methods. Let's see how we can write the non-recursive tree traversal methods. We can implement non-recursive versions of the preorder, inorder and postorder traversal by using an explicit stack. We cannot avoid stack. The stack will be used to store the tree nodes in the appropriate order. Let's try to write methods without using recursion. For this purpose, we have to create stack.

Here, for example, is the routine for inorder traversal that uses a stack.

```
void inorder(TreeNode<int>* root)
{
```

```
   Stack<TreeNode<int>* > stack;
   TreeNode<int>* p;
   p = root;
   do
   {
      while( p != NULL )
      {
         stack.push( p );
         p = p->getLeft();
      }
      // at this point, left tree is empty
         if( !stack.empty() )
         {
         p = stack.pop();
         cout << *(p->getInfo()) << " ";
                  // go back & traverse right subtree
         p = p->getRight();
         }
  } while ( !stack.empty() || p != NULL );
}
```

The signature is same. The name of the function that is *inorder*, its return type is *void* and the argument is pointer to *TreeNode*. Then there is stack abstract data type. We can store int, float or other data type in the stack. Here we want to store the TreeNode in the stack and then in the nodes we want to store integers. The statement is:

```
Stack<TreeNode<int>* > stack;
```

We send a message to the *Stack* factory that creates a stack to store *TreeNode. The integers will be stored in the TreeNode*. We have almost two levels of template. Then we declare a TreeNode pointer as *p* to traverse the tree up and down. We point this pointer *p* to the *root* node. Then we have a do-while loop. The loop checks the value of *p* is not NULL or *stack* is not empty. Inside the outer loop, we have an inner 'while loop' which checks that p is not NULL. If p is not NULL, we push p on the stack i.e. we push the root on the stack. Then we replace the p with p->getLeft() . This shows that p is now pointing to the left subtree node. In case of having a left subtree of p, we come back in the while loop as p is not NULL. Again we push p on the stack. Now p is pointing to the left subtree. You have presumed that this while loop will take us to the left subtree of the root. On the exit from the while loop, we have pushed the node in the stack whose left subtree is NULL. Here, p is pointing to NULL. Now we check that stack is empty or not. If we have pushed some nodes in the stack, it will not be empty. We pop a value from the stack and assign it to p before printing the info stored in that node and assign p the right subtree of that node. We comeback to the outer do-while loop. It will continue executing till stack is empty or p is NULL.

To understand this code, we will trace the code on the same tree, earlier used in the recursive inorder function.

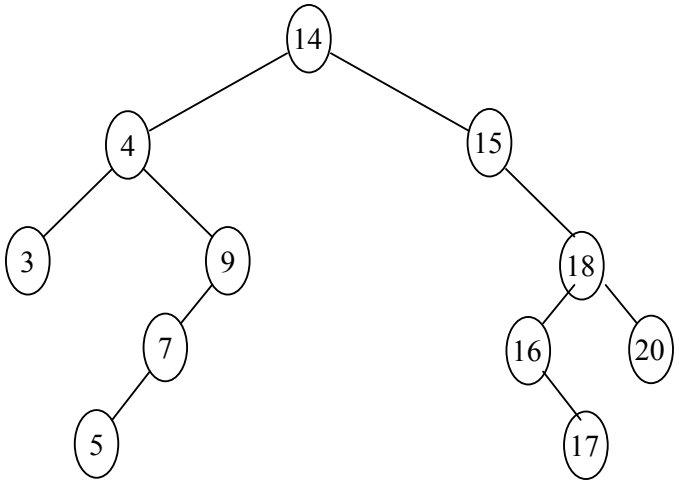


Fig 14.3: Non-Recursive inorder

```
push(14)
..push(4)
....push(3)
3
4
..push(9)
....push(7)
......push(5)
5
7
9
14
push(15)
15
push(18)
..push(16)
16
..push(17)
17
18
push(20)
20
```

This is the same tree earlier discussed with reference to node as 14. We create a stack and assign the root to the pointer p and have an inner while loop. In the while loop, we pushed the root node in the stack i.e. push(14). After this, we assign the pointer p to the left subtree of the root and return to while loop again. Now p is pointing to the node with value 4, so we push 4 and then 3 on the stack. As the left subtree if 3 is null so we exit from the inner loop. Afterwards, we pop the stack in the inner loop and print it. As we know that stack is a LIFO structure (last in, first out). Finally, we have pushed the node 3. So this node will be popped and the number 3 will be printed. Now we will assign the right subtree of the node 3 to the p. As this is NULL, the inner loop will not be executed and again come to the if statement. We will pop the stack in the if statement and as a result, the node 4 will be popped and printed. Then we will assign p to the right subtree of node 4 i.e. node 9. The control comes to the inner loop in which we will push the node 9 and all of its left nodes one by one i.e. push(9), push(7) and push(5). As the left subtree of the node 5 is NULL, the inner while loop will be terminated. We will pop the stack and print 5. As the right subtree of node 5 is NULL, the node 7 will be popped and printed. Here the right subtree of node 7 is NULL. The stack is again popped and the number 9 will be printed. Here, the right subtree of node 9 is also NULL. The stack will be popped resulting in popping and printing of the node 14. Then we go the right subtree of the node 14 which is the node 15. The same rules of left subtree are applied here. You can understand it from the above table.

Traversal Trace

Let's compare the recursive inorder and non-recursive inorder and see whether there is any difference between them. We find no difference between them. Function call takes place with the help of stack. Explicit stack is also a stack and its behavior is also

the same. Let's compare these. In the left column, we have recursive inorder and on the right column there is non-recursive inorder.

recursive inorder	**nonrecursive inorder**
inorder(14)	push(14)
..inorder(4)	..push(4)
....inorder(3)	....push(3)
3	3
4	4
..inorder(9)	..push(9)
....inorder(7)	....push(7)
......inorder(5)	......push(5)
5	5
7	7
9	9
14	14
inorder(15)	push(15)
15	15
inorder(18)	push(18)
..inorder(16)	..push(16)
16	16
..inorder(17)	..push(17)
17	17
18	18
inorder(20)	push(20)
20	20

In recursive inorder ,the function name is *inorder()*. We are printing the values. In case of non-recursive, push() method is employed. You can see the order in which tree nodes are being pushed explicitly on the stack. Then we print the values and use the *pop()* method in a special order to see the values of the nodes.

In the recursive inorder we have *inorder(14) while* in the non-recursive inorder, there is *push(14)*. In the recursive inorder, we have *inorder(4)*, in the non-recursive inorder we have *push(4)*. In the recursive inorder we have *inorder(3. Whereas* in the non-recursive inorder, you may have *push(3)*. It is pertinent to note that the only difference in these two cases is that *inorder() is in the left column while push() will be found in the right column.* You should not be surprised over this as stack is being used in both recursive calls and non-recursive calls.

We can use recursive calls in which stack is used internally. However, in case of non-recursive calls, the stack is used explicitly. Now the question arises which one of these methods is better. Let's see some other aspects of recursion. When should we use the recursion with respect to efficiency and implementation? How many statements were there in the inorder recursive function. The first statement was an if statement which is the terminating condition. After this, we have only three statements, two calls and one cout statement. The order in the inorder is inorder(left), cout and then inorder(right). That's the complete function. Now see the non recursive function. There are a lot of statements in this function so that there should be more code in the non-recursive function. The second thing is the readability. The readability and understanding is better in the recursive function. One can easily understand what is happening. To understand the non-recursive code, one has to read the code with

care. He has to go through every statement to see what you are doing. It will also help ascertain if you are doing it right or not.

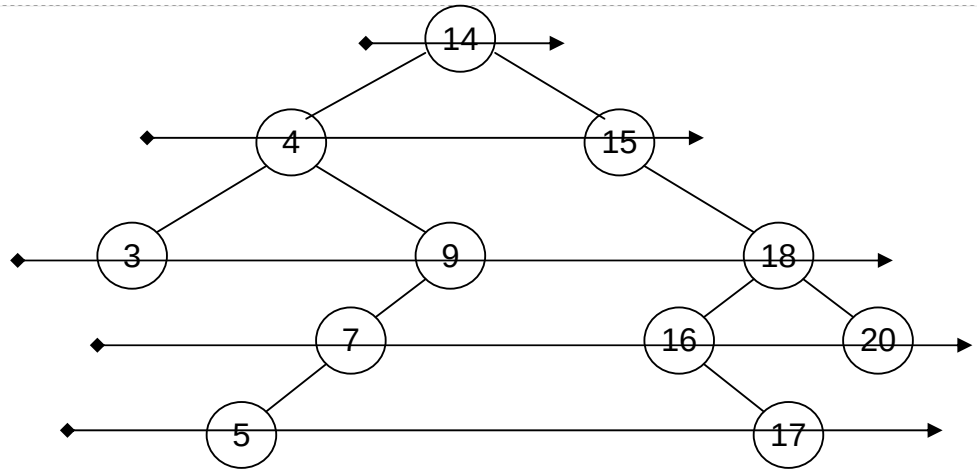
This is also an important aspect of programming. Program readability is also an issue. Suppose you have written some program. Will you understand if after some months that why you have written this and how? The first thing may be that what you are doing in the program and how you do it? After going through the program, you will remember it and recall that you have used this data structure for some purpose. Always comment the code. Comment the data structure, logic and algorithm of the program. Recursive procedure is an elegant procedure. Both the data structure and procedure are recursive. We have traversed a tree with only three four statements. No matter whatever is the size of the tree.

When the recursion happens with the help of function calls and stack.. There are some other values also included. It has return address, local variables and parameters. When a function calls another function irrespective of recursive or non recursive like function F is calling function G. It will take time to put the values in the stack. If you create your own stack, it takes time for creation and then push and pop will also consume time. Now think that which one of these will take more time. The situation is this that function calls always takes place using stack irrespective of the language. The implementation of using stack is implemented very efficiently in the Assembly language. In the computer architecture or Assembly language program, you will study that the manipulation of stack calls that is push and pop and other methods are very efficiently coded. Now you may think that there is a lot of work needed for the recursive calls and non- recursive calls are faster. If you think that the non-recursive function will work fast, it is wrong. The experience shows that if recursive calls are applied on recursive data structures, it will be more efficient in comparison with the non-recursive calls written by you. We will see more data structures which are inherently recursive. If the data structures are recursive, then try to write recursive methods. We will further discuss binary tree, binary search tree and see more examples. Whenever we write a method dealing with these data structures and use recursion. With the use of recursion, our program will be small, efficient and less error prone. While doing programming, we should avoid errors. We don't want to see there are errors while writing a program and executing it.

Try to do these traversals by hands on trees. These traversals are very important in trees. We will write more methods while dealing with tree, used internally in the inorder or postorder traversal.

There is yet another way of traversing a binary tree that is not related to recursive traversal procedures discussed previously. In level-order traversal, we visit the nodes at each level before proceeding to the next level. At each level, we visit the nodes in a left-to-right order. The traversal methods discussed earlier are inorder, preorder and post order. We print the node or to the left subtree or to the right subtree. In the preorder, we traverse the complete left subtree before coming to the right subtree. These procedures are in one way , depth-oriented. Now we may like to traverse the tree in a way that we go at one level and print all the nodes at that level. There are some algorithm in which we need level order traversal.

In the following figure, the level order traversal is represented using the arrows.



Level-order: 14 4 15 3 9 18 7 16 20 5 17

Fig 14.4: level order traversal

We started from the root with 14 as a root node. Later, we printed its value. Then we move to the next level. This is the binary tree so that we may have two elements at next level i.e. 4 and 15. So we printed 4 and 15 while moving from left to right. Now we are not moving to left or right subtree of any node. Rather, we move to the next level. At next level, a programmer will have three nodes that are from left to right as 3, 9 and 18. We printed these values. If we take root node as zero level, it will be at level 2. Now we move to the level 3. Here at level 3, we have 7, 16 and 20. At next level, we have 5 and 17. So the numbers we are having are- 14 4 15 3 9 18 7 16 20 5 17. How can we programmatically traverse the tree in level order? We will discuss it in the next lecture.

Data Structures

Lecture No. 15

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.3, 4.3.5, 4.6

Summary

- Level-order Traversal of a Binary Tree
- Storing Other Types of Data in Binary Tree
- Binary Search Tree (BST) with Strings
- Deleting a Node From BST

Level-order Traversal of a Binary Tree

In the last lecture, we implemented the tree traversal in preorder, postorder and inorder using recursive and non-recursive methods. We left one thing to explore further that how can we print a binary tree level by level by employing recursive or non-recursive method.

See the figure below:

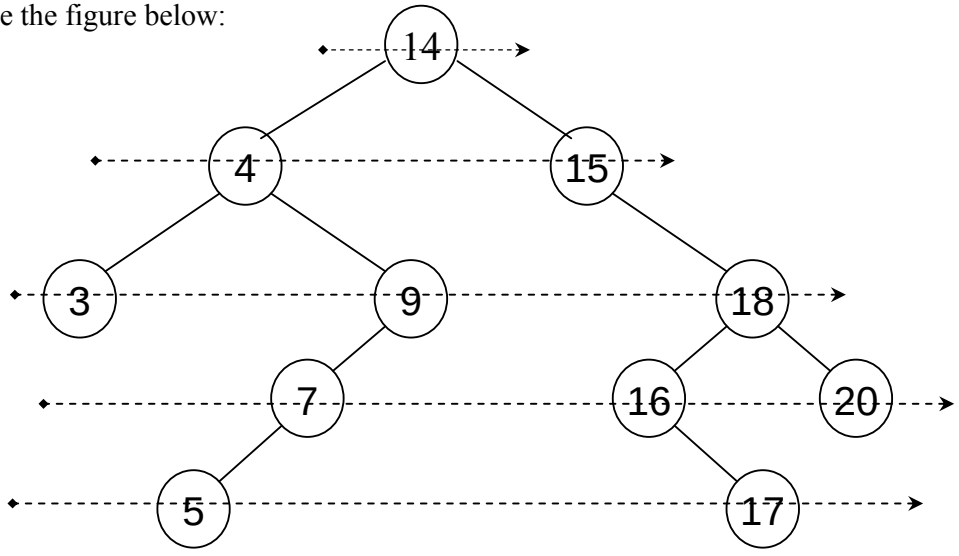


Fig 15.1: Level-order Traversal

In the above figure, levels have been shown using arrows:
At the first level, there is only one number *14*.
At second level, the numbers are *4* and *15*.
At third level, *3*, *9* and *18* numbers are present.
At fourth level the numbers are *7*, *16* and *20*.
While on fifth and final level of this tree, we have numbers *5* and *17*.

This will also be the order of the elements in the output if the level-order traversal is done for this tree. Surprisingly, its implementation is simple using non-recursive method and by employing queue instead of stack. A queue is a FIFO structure, which can make the level-order traversal easier.

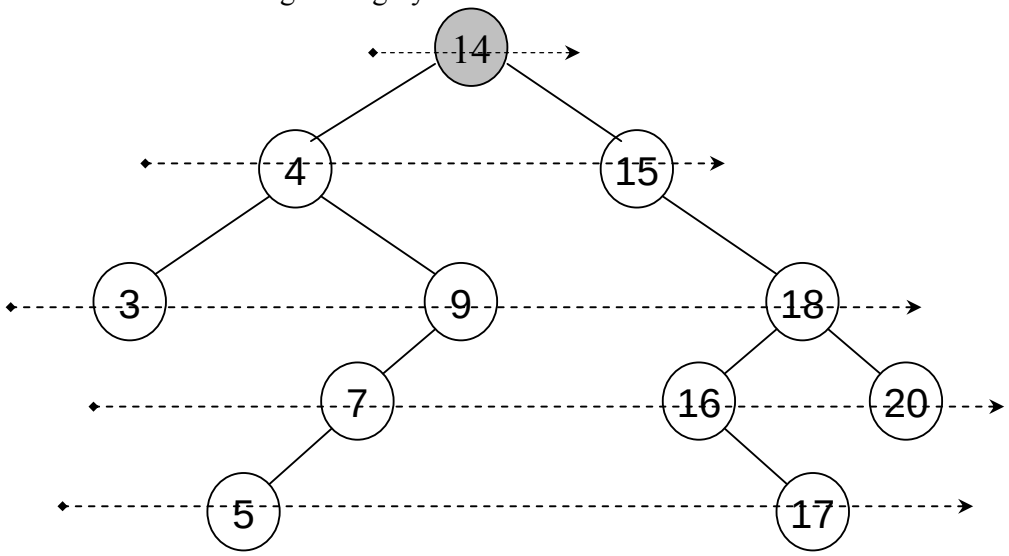
The code for the level-order traversal is given below:

```
1.      void   levelorder( TreeNode <int> *   treeNode )
2.      {
3.         Queue <TreeNode<int> *>   q;
4.
5.         if( treeNode  ==  NULL )   return;
6.         q.enqueue(treeNode);
7.         while( !q.empty() )
8.         {
9.            treeNode   =   q.dequeue();
10.           cout   <<   *(treeNode->getInfo())   <<   " ";
11.           if(treeNode->getLeft()   !=   NULL )
12.                              q.enqueue( treeNode->getLeft());
13.           if(treeNode->getRight()   !=   NULL )
14.                              q.enqueue( treeNode->getRight());
15.        }
16.        cout   <<   endl;
17.     }
```

The name of the method is *levelorder* and it is accepting a pointer of type *TreeNode <int>*. This method will start traversing tree from the node being pointed to by this pointer. The first line (line 3) in this method is creating a queue *q* by using the *Queue* class factory interface. This queue will be containing the *TreeNode<int>* * type of objects. Which means the queue will be containing nodes of the tree and within each node the element is of type *int*.
The line *5* is checking to see, if the *treeNode* pointer passed to the function is pointing to NULL. In case it is NULL, there is no node to traverse and the method will return immediately.
Otherwise at line *6*, the very first node (the node pointed to by the *treeNode* pointer) is added in the queue *q*.
Next is the *while* loop (at line *7*), which runs until the queue *q* does not become empty. As we have recently added one element (at line *6*), so this loop is entered.
At line *9*, *dequeue()* method is called to remove one node from the queue, the element at *front* is taken out. The return value will be a pointer to *TreeNode*. In the next line (line *10*), the *int* value inside this node is taken out and printed. At line *11*, we check to see if the left subtree of the tree node (we've taken out in at line *9*) is present. In case the left subtree exists, it is inserted into the queue in the next statement at line *12*. Next, we see if the right subtree of the node is there, it is inserted into the queue also. Next statement at line *15* closes the *while* loop. The control goes back to the line *7*, where it checks to see if there is some element left in the queue. If it is not empty, the loop is entered again until it becomes empty.

Let's execute this method *levelorder(TreeNode <int> *)* by hand on a tree shown in

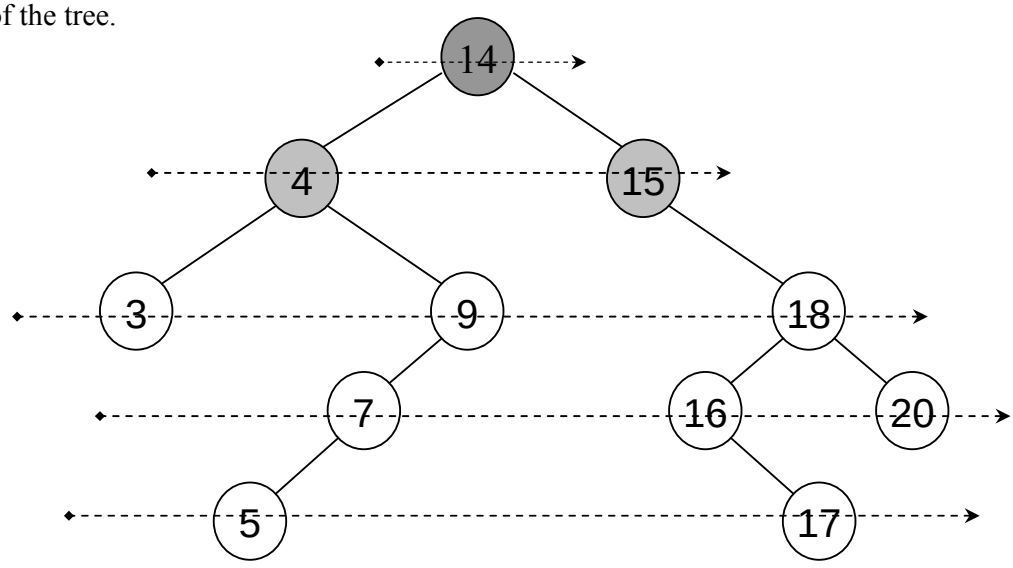
the *fig 15.1* to understand the level order traversal better. We can see that the *root* node of the tree is containing element *14*. This node is inserted in the queue first, it is shown in the below figure in gray shade.



Queue: 14
Output:

Fig 15.2: First node is inserted in the queue

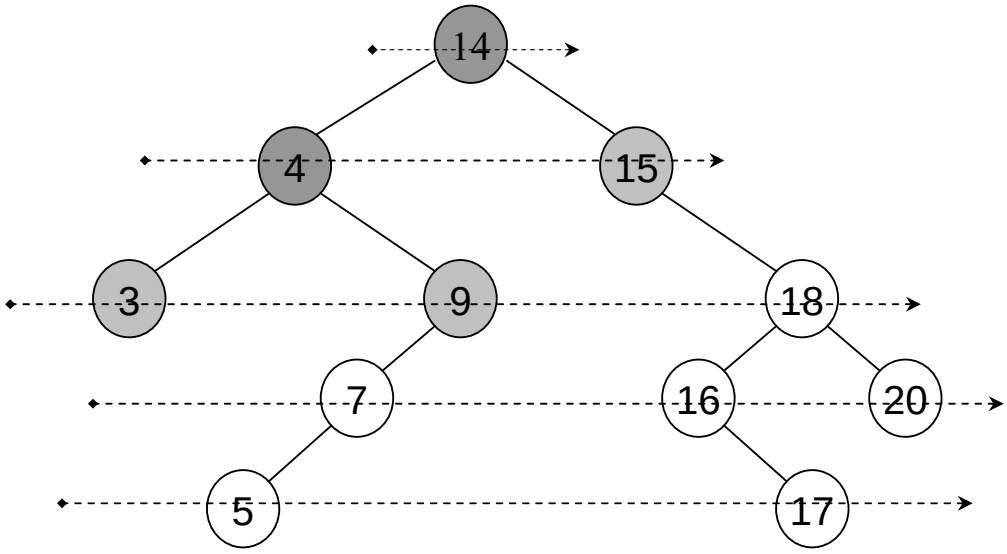
After insertion of node containing number 14, we reach the *while* loop statement, where this is taken out from the queue and element *14* is printed. Next the left and right subtrees are inserted in the queue. The following figure represents the current stage of the tree.



Queue: 4 15
Output: 14

Fig 15.3: 14 is printed and two more elements are added in the queue

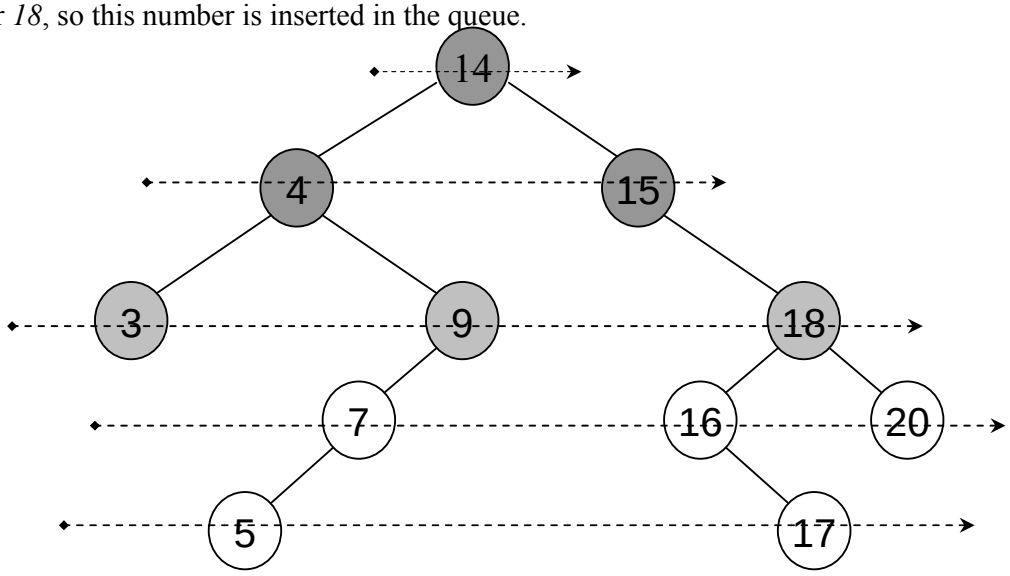
The element that has been printed on the output is shown with dark gray shade while the elements that have been inserted are shown in the gray shade. After this the control is transferred again to the start of the *while* loop. This time, the node containing number *14* is taken out (because it was inserted first and then the right node) and printed. Further, the left and right nodes (*3* and *9*) are inserted in the queue. The following figure depicts the latest picture of the queue.



Queue: 15 3 9
Output: 14 4

Fig 15.4

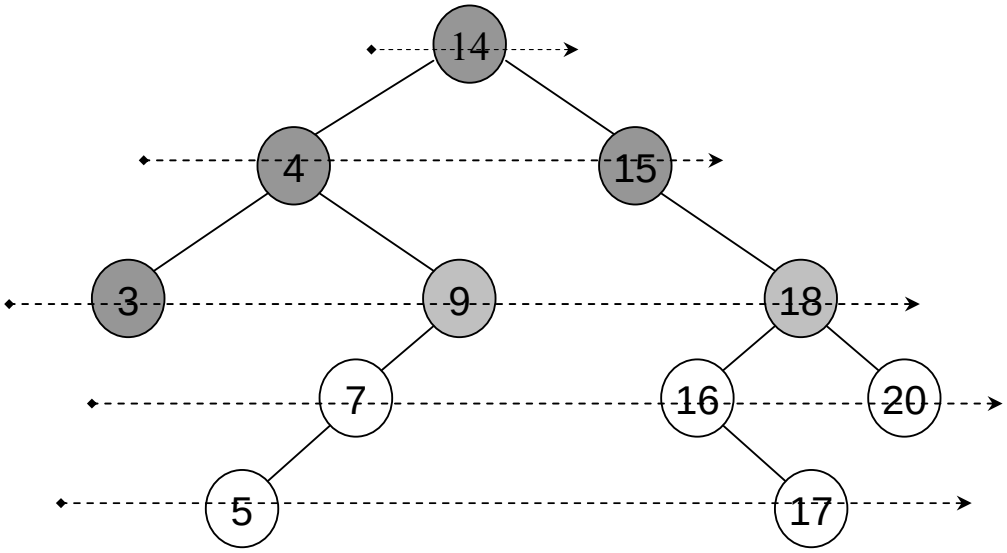
In the above *Queue*, numbers *15*, *3* and *9* have been inserted in the queue until now. We came back to the *while* loop starting point again and dequeued a node from the queue. This time a node containing number *15* came out. Firstly, the number *15* is printed and then we checked for its left and right child. There is no left child of the node (with number *15)* but the right child is there. Right child is a node containing number *18*, so this number is inserted in the queue.



Queue: 3 9 18
Output: 14 4 15

Fig 15.5

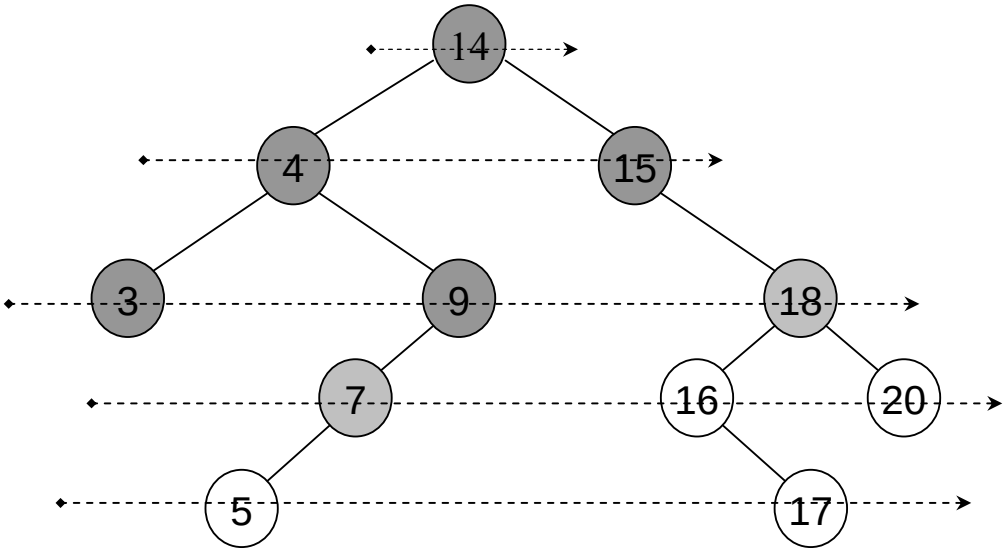
This time the node that is taken out is containing number *3*. It is printed on the output. Now, node containing number *3* does not have any *left* and *right* children, therefore, no new nodes is inserted in the queue in this iteration.



Queue: 9 18
Output: 14 4 15 3

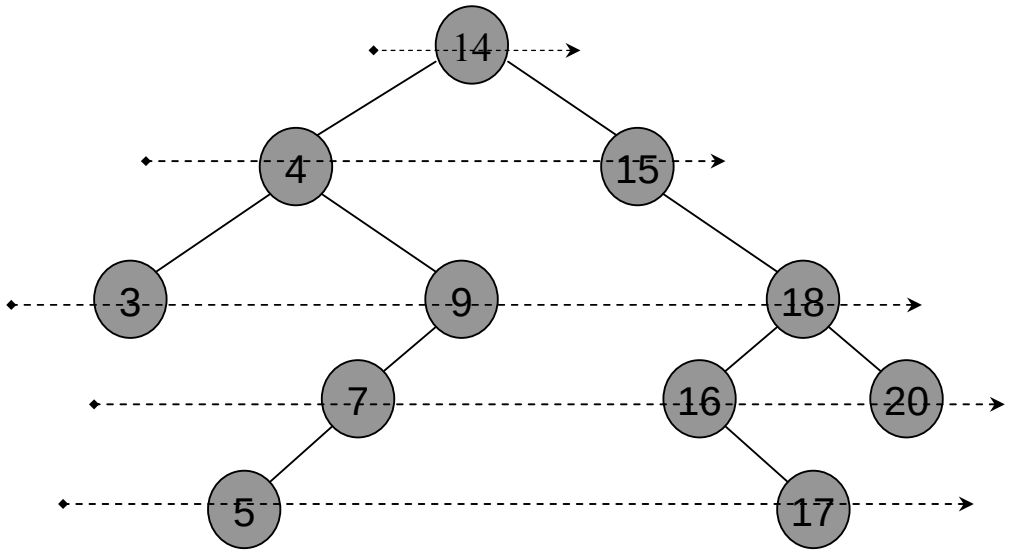
Fig 15.6

In the next iteration, we take out the node containing number *9*. Print it on the output. Look for its left subtree, it is there, hence inserted in the queue. There is no right subtree of *9*.



Queue: 18 7
Output: 14 4 15 3 9
Fig. 15.7

Similarly, if we keep on executing this function. The nodes are inserted in the queue and printed. At the end, we have the following picture of the tree:



Queue:
Output: 14 4 15 3 9 18 7 16 20 5 17

Fig 15.8

As shown in the figure above, the *Queue* has gone empty and in the output we can see that we have printed the tree node elements in the level order.

In the above algorithm, we have used *queue* data structure. We selected to use queue data structure after we analyzed the problem and sought most suitable data structure. The selection of appropriate data structure is done before we think about the programming constructs of *if* and *while* loops. The selection of appropriate data structure is very critical for a successful solution. For example, without using the queue data structure our problem of *levelorder* tree traversal would not have been so easier.

What is the reason of choosing queue data structure for this problem. We see that for this levelorder (level by level) tree traversal, the levels come turn by turn. So this turn by turn guides us of thinking of using queue data structure.

Always remember that we don't start writing code after collecting requirements, before that we select the appropriate data structures to be used as part of the design phase. It is very important that as we are getting along on our course of data structures, you should take care why, how and when these structures are employed.

Storing Other Types of Data in Binary Tree

Until now, we have used to place *int* numbers in the tree nodes. We were using *int* numbers because they were easier to understand for sorting or comparison problems. We can put any data type in tree nodes depending upon the problem where tree is employed. For example, if we want to enter the names of the people in the telephone

directory in the binary tree, we build a binary tree of strings.

Binary Search Tree (BST) with Strings

Let's write C++ code to insert non-integer data in a binary search tree.

```
void wordTree()
{
        TreeNode<char> * root = new TreeNode<char>();
        static char * word[] = "babble", "fable", "jacket",
        "backup", "eagle","daily","gain","bandit","abandon",
        "abash","accuse","economy","adhere","advise","cease",
        "debunk","feeder","genius","fetch","chain", NULL};
        root->setInfo( word[0] );

        for(i = 1; word[i]; i++ )
                insert(root, word[i] );
        inorder( root );
        cout << endl;
}
```

The function name is *wordTree()*. In the first line, a *root* tree node is constructed, the node will be containing *char* type of data. After that is a *static* character array *word*, which is containing words like *babble, fable* etc. The last word or string of the array is *NULL*. Next, we are putting the first word (*babble*) of the *word* array in the *root* node. Further is a *for* loop, which keeps on executing and inserting words in the tree using *insert (TreeNode<char> *, char *)* method until there is a word in the *word* array (*word* is not *NULL*). You might have noticed that we worked in the same manner when we were inserting *int*s in the tree. Although, in the *int* array, we used –1 as the ending number of the array and here for words we are using *NULL*. After inserting the whole array, we use the *inorder( )* method to print the tree node elements.
Now, we see the code for the *insert* method.

```
void insert(TreeNode<char> * root, char * info)
{
        TreeNode<char> * node = new TreeNode<char>(info);
        TreeNode<char> *p, *q;
        p = q = root;
        while( strcmp(info, p->getInfo()) != 0 && q != NULL )
        {
                p = q;
                if( strcmp(info, p->getInfo()) < 0 )
                        q = p->getLeft();
                else
                        q = p->getRight();
        }
        if( strcmp(info, p->getInfo()) == 0 )
        {
                cout << "attempt to insert duplicate: " << * info << endl;
                delete node;
        }
```

```
        else if( strcmp(info, p->getInfo()) < 0 )
                p->setLeft( node );
        else
                p->setRight( node );
}
```

The *insert(TreeNode<char> * root, char * info)* is accepting two parameters. First parameter *root* is a pointer to a *TreeNode*, where the node is containing element *char* type. The second parameter *info* is pointer to *char*.

In the first line of the *insert* method, we are creating a new *TreeNode* containing the *char* value passed in the *info* parameter. In the next statement, two pointers *p* and *q* of type *TreeNode<char>* are declared. Further, both pointers *p* and *q* start pointing to the tree node passed in the first parameter *root*.

You must remember while constructing binary tree of numbers, we were incrementing its count instead of inserting the new node again if the same number is present in the node. One the other hand, if the same number was not there but the new number was less than the number in the node, we used to traverse to the left subtree of the node. In case, the new number was greater than the number in the node, we used to seek its right subtree.

Similarly, in case of strings (words) here, we will increment the counter if it is already present, and will seek the left and right subtrees accordingly, if required.

In case of *int*'s we could easily compare them and see which one is greater but what will happen in case of strings. You must remember, every character of a string has an associated ASCII value. We can make a lexicographic order of characters based on their ASCII values. For example, the ASCII value of *B (66)* is greater than *A* (65), therefore, character *B* is greater than character *A*. Similarly if a word is starting with the letter A, it is smaller than the words starting from B or any other character up to *Z*. This is called lexicographic order.

C++ provides us overloading facility to define the same operators (<, >, <=, >=, == etc) that were used for *int*s to be used for other data types for example strings. We can also write functions instead of these operators if we desire. *strcmp* is similar kind of function, part of the standard C library, which compares two strings.

In the code above inside the *while* loop, the *strcmp* function is used. It is comparing the parameter *info* with the value inside the node pointed to by the pointer *p*. As *info* is the first parameter of *strcmp*, it will return a negative number if *info* is smaller, *0* if both are equal and a positive number if *info* is greater. The *while* loop will be terminated if the same numbers are found. There is another condition, which can cause the termination of loop that pointer *q* is pointing to NULL.

First statement inside the loop is the assignment of pointer *q* to *p*. In the second insider statement, the same comparison is done again by using *strcmp*. If the new word pointed to by *info* is smaller, we seek the left subtree otherwise we go to the right subtree of the node.

Next, we check inside the *if-statement*, if the reason of termination of loop is duplication. If it is, a message is displayed on the output and the newly constructed node (that was to be inserted) is deleted (deallocated).

If the reason of termination is not duplication, which means we have reached to the node where insertion of the new node is made. We check if the *info* is smaller than the word in the current tree node. If this is the case, the newly constructed node is inserted to the left of the current tree node. Otherwise, it is inserted to the right.

This *insert()* method was called from inside the *for* loop in the *wordTree()* method. That loop is teminated when the NULL is reached at the end of the array *word*. At the end, we printed the inserted elements of the tree using the *inorder()* method.

Following is the output of *inorder()*:

Output:
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket

Notice that the words have been printed in the sorted order. Sorting is in increasing order when the tree is traversed in *inorder* manner. This should not come as a surprise if you consider how we built the binary search tree. For a given node, values less than the *info* in the node were all in the left subtree and values greater or equal were in the right. *Inorder* prints the left subtree first, then the node itself and at the end the right subtree.
Building a binary search tree and doing an *inorder* traversal leads to a sorting algorithm.

We have found one way of sorting data. We build a binary tree of the data, traverse the tree in *inorder* fashion and have the output sorted in increasing order. Although, this is one way of sorting, it may not be the efficient one. When we will study sorting algorithms, will prove Mathematically that which method is the fastest.

Deleting a Node From BST

Until now, we have been discussing about adding data elements in a binary tree but we may also require to delete some data (nodes) from a binary tree. Consider the case where we used binary tree to implement the telephone directory, when a person leaves a city, its telephone number from the directory is deleted.
It is common with many data structures that the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.
For case 1, If the node is a *leaf*, it can be deleted quite easily.

See the tree figure below.

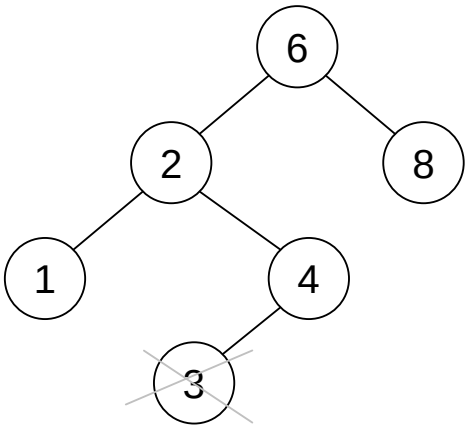


Fig 15.9: BST

Suppose we want to delete the node containing number *3*, as it is a leaf node, it is pretty straight forward. We delete the leaf node containing value *3* and point the right subtree pointer to NULL.

For case 2, if the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to *inorder* successor.

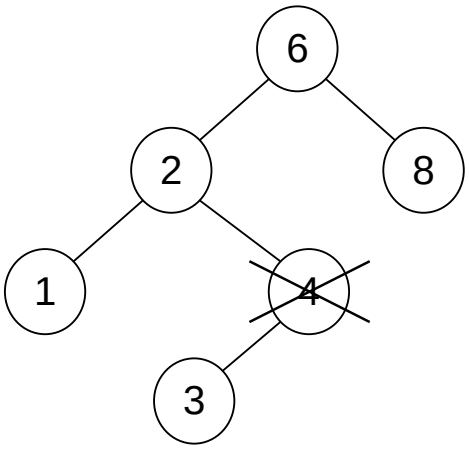


Fig 15.10: Deleting a Node From BST

If we want to delete the node containing number *4* then we have to adjust the right subtree pointer in the node containing value *2* to the *inorder* successor of *4*. The important point is that the *inorder* traversal order has to be maintained after the delete.

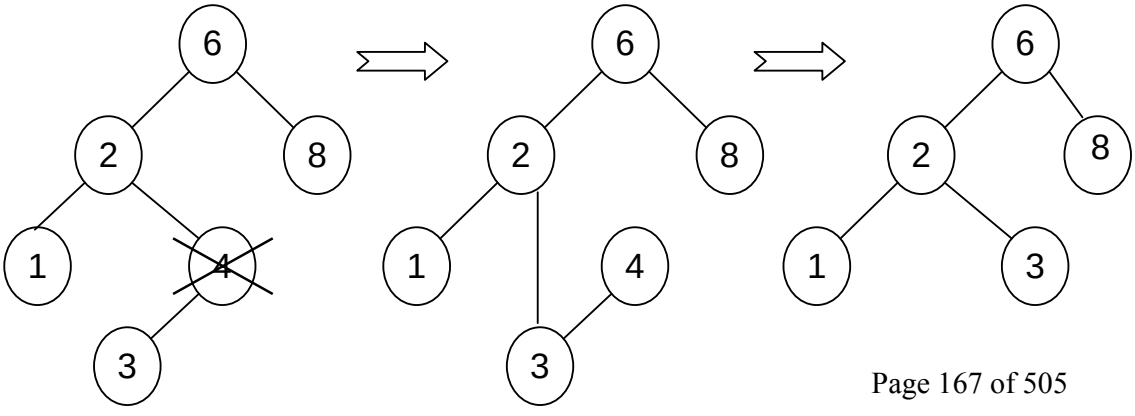
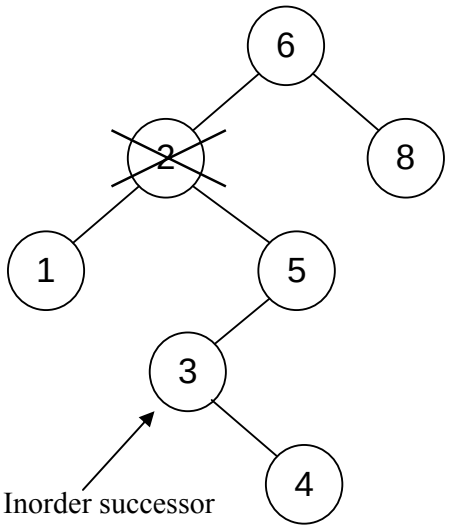


Fig 15.11: Deletion in steps

The case 3 is bit complicated, when the node to be deleted has both left and right subtrees.
The strategy is to replace the data of this node containing the smallest data of the right subtree and recursively delete that node.

Let's see this strategy in action. See the tree below:



In this tree, we want to delete the node containing number 2. Let's do *inorder* traversal of this tree first. *The inorder* traversal give us the numbers: *1*, *2*, *3*, *4*, *5*, *6* and *8*.
In order to delete the node containing number *2*, firstly we will see its right subtree and find the left most node of it.
The left most node in the right subtree is the node containing number *3*. Pay attention to the nodes of the tree where these numbers are present. You will notice that node containing number *3* is not right child node of the node containing number *2* instead it is left child node of the right child node of number *2*. Also the left child pointer of node containing number *3* is NULL.
After we have found the left most node in the right subtree of the node containing number *2*, we copy the contents of this left most node i.e. *3* to the node to be deleted with number *2*.

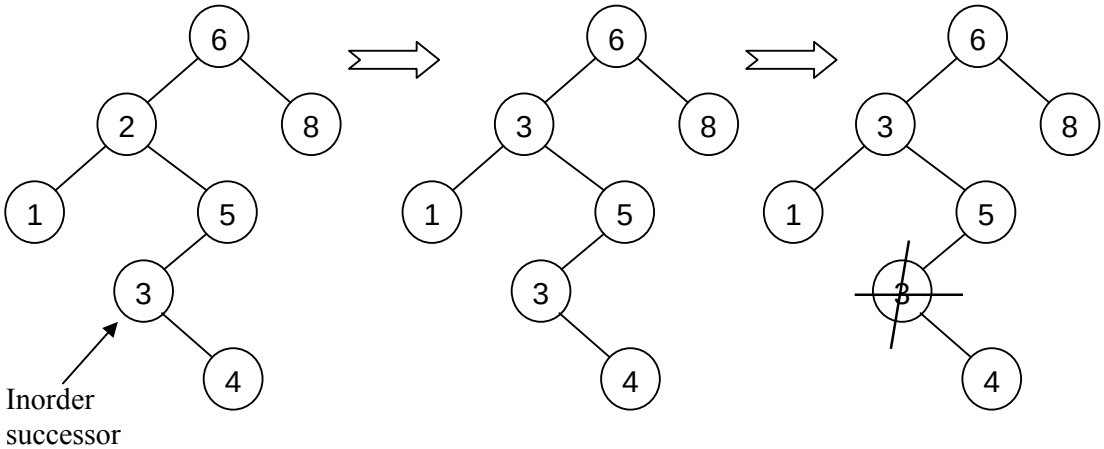


Fig 15.13: delete (2) - remove the inorder successor

Next step is to delete the left most node containing value *3*. Now being the left most node, there will be no left subtree of it, it might have a right subtree. Therefore, the deletion of this node is the case 2 deletion and the delete operation can be called recursively to delete the node.

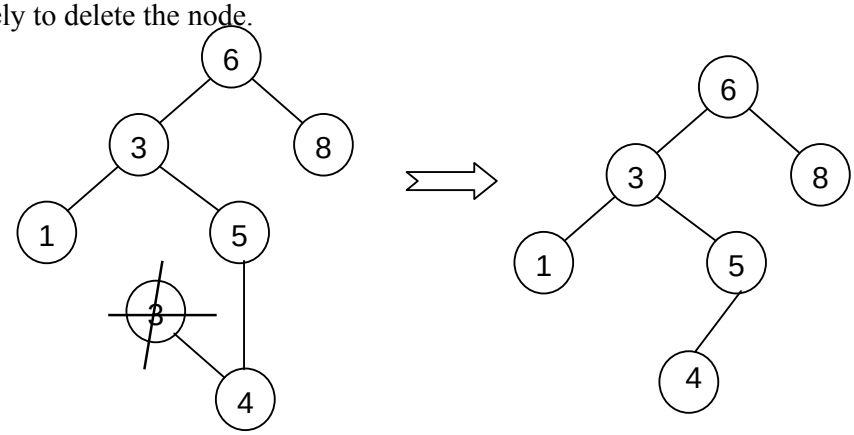


Fig 15.14: delete (2)

Now if we traverse the tree in *inorder*, we get the numbers as: *1*, *3*, *4*, *5*, *6* and *8*. Notice that these numbers are still sorted. In the next lecture, we will also see the C++ implementation code for this deletion operation.

Data Structures

Lecture No. 16

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4

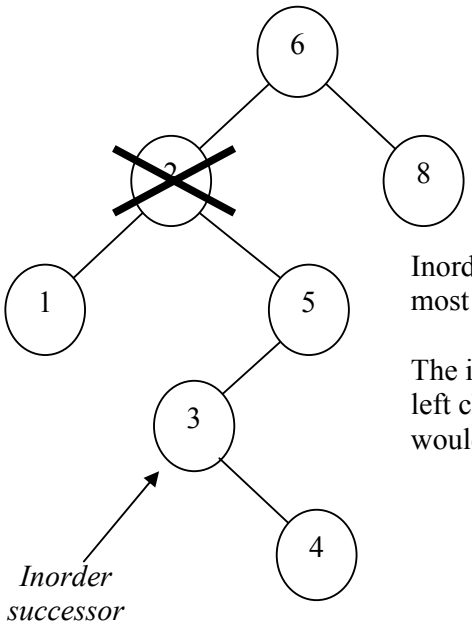
4.3 (all sub sections)

Summary

- Deleting a node in BST
- C++ code for remove
- Binary Search Tree Class (BST)

Deleting a node in BST

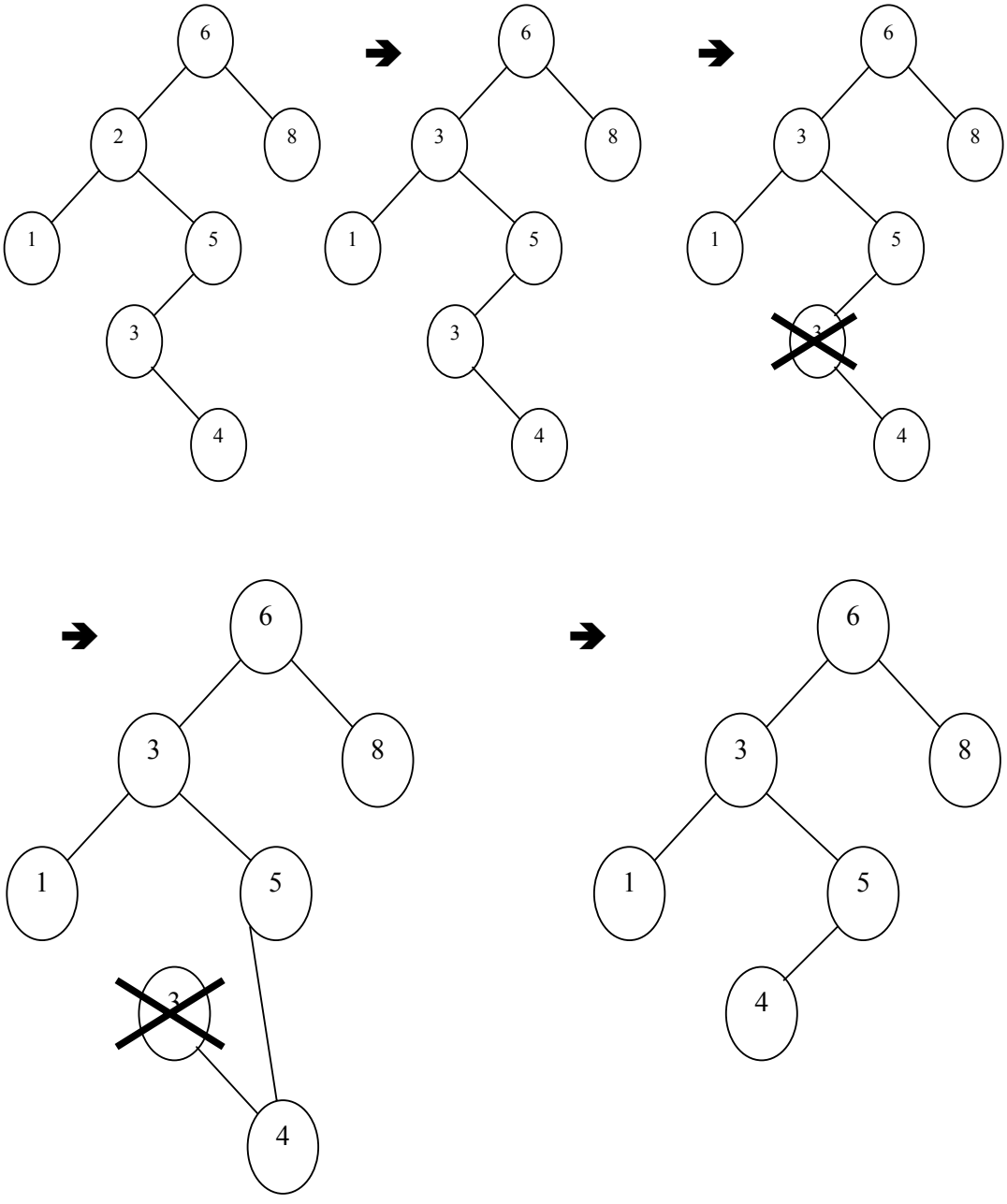
In the previous lecture, we talked about deleting a node. Now we will discuss the ways to delete nodes of a BST (*Binary Search Tree*). Suppose you want to delete one of the nodes of BST. There are three cases for deleting a node from the BST. Case I: The node to be deleted is the leaf node i.e. it has no right or left child. It is very simple to delete such node. We make the pointer in the parent node pointing to this node as NULL. If the memory for the node has been dynamically allocated, we will release it. Case II: The node to be deleted has either left child (subtree) or right child (subtree). Case III: The node to be deleted has both the left and right children (subtree). This is the most difficult case. Let's recap this case. The following figure shows the tree and the node to be deleted.



Inorder successor will be the left-most node in the right subtree of 2.

The inorder successor will not have a left child because if it did, that child would be the left-most node.

In the above example, the node to be deleted is 2. It has both right and left subtrees. As this is the BST, so while traversing the tree in inorder, we will have sorted data. The strategy is to find the inorder successor of the node to be deleted. To search the inorder successor, we have to go to its right subtree and find the smallest element. Afterwards we will copy this value into the node to be deleted. In the above example, the number 3 is the smallest in the right subtree of the node 2. A view of data of inorder traversal shows that number 3 is the inorder successor of number 2. We copy the number 3 in the node to be deleted. Now in this transition tree, the number 3 is at two places. Now we have to delete the inorder successor node i.e. node 3. The node 3 has only right child. So this is the Case II of deletion.



We will get the inorder successor of node 3 i.e. number 4. Therefore we will connect

the node 5 with node 4 that is the left pointer of 5 which was earlier pointing to node 3. But now it is pointing to node 4. We delete the node 3. Remember that number 3 has been copied in the node 2. This way, we will delete the node having right and left both subtrees.

C++ code for remove

Now we will see the C++ code to carry out this deletion. In the above figures, we have named the method as *delete*, but *delete* is a keyword of C++. So we cannot write a method named *delete*. We have to name it something else, say *remove*. Keep in mind all the three cases of delete and the figures.

Here is the code of the *remove* method.

```
/* This method is used to remove a node from the BST */

TreeNode<int>*    remove(TreeNode<int>* tree, int info)
{
   TreeNode<int>* t;
   int cmp = info - *(tree->getInfo());

   if( cmp < 0 ){                        // node to delete is in left subtree
         t = remove(tree->getLeft(), info);
         tree->setLeft( t );
   }
   else if( cmp > 0 ){                 // node to delete is in right subtree
         t = remove(tree->getRight(), info);
         tree->setRight( t );
   }
      //two children, replace with inorder successor
   else if(tree->getLeft() != NULL && tree->getRight() != NULL ){
         TreeNode<int>* minNode;
         MinNode = findMin(tree->getRight());
         tree->setInfo( minNode->getInfo() );
         t = remove(tree->getRight(), *(minNode->getInfo()));
         tree->setRight( t );
   }
else {  // case 1
         TreeNode<int>* nodeToDelete = tree;
         if( tree->getLeft() == NULL ) //will handle 0 children
             tree = tree->getRight();
         else if( tree->getRight() == NULL )
             tree = tree->getLeft();
         else tree = NULL;

         delete nodeToDelete;  // release the memory
   }
   return tree;
}
```

The return type of the *remove* method is a pointer to *TreeNode*. The argument list contains two arguments, the pointer to *TreeNode* pointing the root of the tree and an integer *info* containing the value of the node to be deleted. A thorough analysis of the *delete* method reveals that you need these two things.

Here we are dealing with the templates and the nodes containing the *int* values. Inside the function, we create a temporary variable *t* of type *TreeNode* pointer. Then we have int type variable *cmp*. The value of the *TreeNode* provided in the argument is subtracted from the *info* (this is the value of the node to be deleted) and assigned to the *cmp*. Then we have conditional *if* statement which checks if the value of *cmp* is less than 0, we call *remove* method recursively. As the *cmp* is less than zero, it means that the *info* is not the node provided in the argument. The value of *cmp* is less than zero, depicting that the value of the current node is greater than *info*. Therefore if the node with value *info* exists, it should be in the left subtree. So we call the *remove* method on the left subtree. When we return from this call, the left subtree pointer of *tree* is assigned to our temporary variable *t*.

If the first *if* statement evaluates to true, it means that our required node is in the left subtree of the current node. We call the *remove* method recursively and remove that node. On coming back, we assign *t* to the left subtree pointer of the *tree*. Suppose that the node to be deleted is the immediate left child of the node. It will be deleted by the *remove* method. When the node is deleted, some other node will come to its place, necessitating the readjustment of the left pointer of the parent node. We can do it by pointing the left subtree pointer to *t*.

If the value of *cmp* is greater than zero, the node with *info* value should be in the right subtree of the current node. Therefore we call the *remove* method recursively, providing it the pointer to the right subtree and the *info*. When we return from that call, the right subtree is readjusted by assigning it the *t*.

In the third case, we are at the node, which is to be deleted. Here we have to deal with the three delete cases- 1) the node to be deleted is the leaf node 2) node to be deleted has either right or left child and 3) the node to be deleted has both left and right subtree. We will begin with the third case. We will check in the *else-if* statement that the left and right subtrees are not NULL. It means that the node has both left and right subtrees. We create a temporary variable *minNode* of type *TreeNode* pointer. This variable *minNode* is local to this *if-else* block. We assign it value by calling the *findMin* method. We call the *findMin* method as:

```
findMin(tree->getRight())
```

In this delete case, we need to find the inorder successor of the node to be deleted which is the minimum value in the right subtree. Then we will put the value of this node to the node to be deleted. This node is saved in the variable *minNode*. The node to be deleted is now pointed by *tree*. Then we get the value of the inorder successor, stored in the *minNode* and assign it to the node to be deleted. Now we have to delete the *minNode*. For this purpose, we call the *remove* method recursively providing it the right subtree and the value of the *minNode*. We have also changed the value of the current node and can also send this value to the *remove* method. For clarity purposes, we provide it the *minNode* value. The node returned by the *remove* method, is stored

in the temporary variable *t*. Then we assign *t* to the right subtree of *tree*. We do it due to the fact that the tree has to re-adjusted after the deletion of a node in the right subtree.

Now we will talk about other two cases i.e. the node to be deleted is the leaf node or has left or right child. We will deal with these two cases together. The *else* statement is being employed in these cases. We create a temporary variable *nodeToDelete* and assign it to the *tree*. Now we have to check whether this is the leaf node or has either left or right child. First we check that the left subtree of the tree is NULL before assigning the right subtree to the *tree*. Then we check that the right subtree of the tree is NULL. If it is NULL, the *tree* will be replaced by the left subtree. In the end, we make the tree NULL and delete the *nodeToDelete.* Here the memory is released.

While dealing with this case of delete, we find the inorder successor from the right subtree i.e. the minimum value in the right subtree. That will also be the left most node of the right subtree. We use the *findMin* method for this purpose. Let's have a look on the code of this method.

```
/* This method is used to find the minimum node in a tree
*/

TreeNode<int>* findMin(TreeNode<int>* tree)
{
   if( tree == NULL )
      return NULL;
   if( tree->getLeft() == NULL )
      return tree; // this is it.
   return findMin( tree->getLeft() );
}
```

The return type is a pointer to *TreeNode* which will be the minimum node. The input argument is also a pointer to *TreeNode*. This input argument is a root of some subtree in which we have to find the minimum node. Inside the function, we have a couple of *if* statements. If the *tree* is NULL, it will result in the return of the NULL. If the left child of the tree is NULL, the *tree* will be returned. In the end, we are calling *findMin* recursively providing it the left subtree as argument. When we were talking about the inorder successor or the left most node, it was stated that this node will not contain a left child because if it has a left child then it will not be a left most child. The left-most node is the node which either does not have a left child or its left pointer is NULL. The above method works the same way. It looks for the left node having left pointer NULL. This is the node with the minimum value. You must have noted that in this function, we are making a recursive call in the end of the function. If a function has recursive call as the last statement, it is known as tail recursion. We can replace the tail recursion with the loop. How can we do that? This is an exercise for you. In case of tail recursion, you have either to use the tail recursion or loop. While opting for loop, you will get rid of the recursion stack.

You can trace this function on the sample tree we have been using in the previous lecture. It can also be hand-traced. Provide the root of the BST to the method and try to find the *minNode* while employing the above method. You will see that the value

returned by this function is the minimum node. Also try to write the *FindMin* in a non-recursive way i.e. use of the loop. You will get the same result.

Binary Search Tree Class (BST)

Let's see the code of the binary search tree (BST). We put the interface in the .h file. We also have the state variable in addition to the public and private methods of the class in it. In the public methods, the user of the class calls with their objects. Let's create a .h file for binary search tree and the objects of binary search tree obtained from the factory will use the attributes defined in the .h file. Here is the .h file.

```
/* binarysearchtree.h file contains the interface of binary search tree (BST) */

#ifndef _BINARY_SEARCH_TREE_H_
#define _BINARY_SEARCH_TREE_H_

#include <iostream.h>       // For NULL

// Binary node and forward declaration
template <class EType>
class BinarySearchTree;

template <class EType>
class BinaryNode
{
   EType element;
   BinaryNode *left;
   BinaryNode *right;

public:
   // constructor
   BinaryNode( const EType & theElement, BinaryNode *lt, BinaryNode *rt )
                                   : element( theElement ), left( lt ), right( rt ) { }

   friend class BinarySearchTree<EType>;
};
// continued
```

At the start of the file, we have conditional statement using the *ifndef* that checks a constant _BINARY_SEARCH_TREE_H_. If this constant is not defined, we define it with the help of *define* keyword. This is a programming trick. Sometimes, we include the same .h file more than once. If we do not use this trick, the compiler will give error on the inclusion of the file for the second time. Here, we have a statement:

```
class BinarySearchTree;
```

This is the forward declaration of the class *BinarySearchTree*. We have not defined this class so far. First we define the class *BinaryNode* and then by combining these nodes, a binary tree is created. This class is a template class. The nodes can contain data other than integers. Then we define the *BinaryNode* class and define three state

variables. The data of the node is stored in the *element* which is of type *EType*. Being a binary node, it may have a left and right child. We have declared two pointers for this node. Then we have its constructor that takes three arguments, used to initialize the *element*, *left* and *right* variables. We wrote the colon(:) and the initialization list. There are three state variables in the *BinaryNode*. These are *element*, *left* and *right*. When we create an object of *BinaryNode,* all of these variables are created. The *EType* can also be some class so its object will be created and a call to its constructor is made. Whenever an object is created, its constructor will be called. We have to provide the arguments. If we do not call the constructor explicitly, the default constructor will be called. In the constructor of *BinaryNode*, after the colon, we have written *element(theElement)*. (Here *theElement* is the argument in the constructor of *BinaryNode*). It seems to be a function call but actually it is a call to the constructor for *element*. If element was of type *String,* constructor of *String* will be called and *theElement* is sent as argument. Through this way of writing the variables after the colon, we are actually calling the explicit constructor for the state variables. While calling the constructor, we are not using the name of the class but the variable names. There should be no ambiguity. Then we have *left(lt)* and *right(rt)*, as *left* and *right* are of type *BinaryNode* so the constructor for *BinaryNode* will be called. The body of the constructor of the *BinaryNode* is empty.

You might not be familiar with this type of coding but it is commonly used in C++ coding. You can get further information on this from the C++ book or from the internet. In some cases, a programmer is forced to use this syntax for having no other option. The last statement of the class definition has a friend class:

```
friend class BinarySearchTree<EType>;
```

You are familiar with the *friend* keyword. The class *BinaryNode* has defined the *BinarySearchTree* class as its friend class. It means that the *BinarySearchTree* class can access the state variables of *BinaryNode* class. We can also declare friend method of some class which can access the state variable of that class. We have not defined the *BinarySearchTree* class yet. We have just declared the class with the name *BinarySearchTree* as the forward declaration. This forward declaration is for compilers. When the compiler reads the file from top to bottom and encounters the line defining the *BinarySearchTree* as friend, it gives an error. The compiler does not know about the *BinarySearchTree.* To overcome this error we use forward declaration. With this forward declaration, the compiler is communicated that we want to use the *BinarySearchTree* as a class. Therefore, when the compiler reads the line, which defines the *BinarySearchTree* as a friend of *BinaryNode*, it knows that it is a class, which used *EType* template and will be defined later. Now there is no ambiguity for the compiler. We must have to define this class later on.

Let's see the code of *BinarySearchTree.* We are giving it *EType* template parameter. We want to make our *BinarySearchTree* generic and it can be used with integers, strings, characters or with some other data type. Therefore, we have defined it as a template class. The user of this class will create a *BinarySearchTree* of its specific data type. The code is as follows:

```
/* binarysearchtree.h file also contains the definition of the
BinarySearchTree */

template <class EType>
class BinarySearchTree
{
public:
   BinarySearchTree( const EType& notFound );
   BinarySearchTree( const BinarySearchTree& rhs );
   ~BinarySearchTree( );

   const EType& findMin( ) const;
   const EType& findMax( ) const;
   const EType& find( const EType & x ) const;
   bool isEmpty( ) const;
   void printInorder( ) const;
   void insert( const EType& x );
   void remove( const EType& x );

   const BinarySearchTree & operator = ( const BinarySearchTree & rhs
);

private:
   BinaryNode<EType>* root;
   // ITEM_NOT_FOUND object used to signal failed finds
   const EType ITEM_NOT_FOUND;

   const EType& elementAt( BinaryNode<EType>* t );
   void insert(const EType& x, BinaryNode<EType>* & t);
   void remove(const EType& x, BinaryNode<EType>* & t);
   BinaryNode<EType>* findMin(BinaryNode<EType>* t);
   BinaryNode<EType>* findMax(BinaryNode<EType>* t);
   BinaryNode<EType>* find(const EType& x, BinaryNode<EType>* t
);
   void makeEmpty(BinaryNode<EType>* & t);
   void printInorder(BinaryNode<EType>* t);
};
#endif
```

We start the definition of our class with the public interface. The user of this class is interested in the public interface of the class. These are the methods, which the user can employ. We have two constructors for this class. One constructor takes *EType* reference parameter while other takes *BinarySearchTree* as a parameter. The types of constructors depend on the usage. We can have more constructors if needed. Then we have a destructor of *BinarySearchTree.* Besides, we have interface methods for *BinarySearchTree.* There are also *findMin* and *findMax* methods, which will return the minimum and maximum value of *EType* in the tree respectively. We have used the *const* keyword with these functions. The reference variables in functions are also being used. Next thing we are going to discuss is the *find* method. Its signature is as

under:

```
const EType& find( const EType & x ) const;
```

This method will takes an argument *x* of *EType*. It will search the tree whether *x* exists in the tree or not. Then we have *isEmpty* method that will ascertain if the tree is empty or not. There is also the *printInorder* method, which will print the tree in inorder traversal. If the tree has integers, we have sorted integers as a result of inorder traversal. Next thing we have is the *insert (const EType& x)* method, which inserts the *x* as a new node in the tree. After this, there is the *remove* method i.e.delete method. We have renamed *delete* as *remove* because *delete* is a keyword of C++. We can also name it as *deleteNode* or some other name which you think is suitable. The interface of the class is almost complete. Now think as a user of *BinarySearchTree* and decide if you need more methods. With the help of methods defined above, we can do a lot of work on *BinarySearchTree*. If you feel to add more methods due to the usage need, these can be added later.

We also need some private variables and methods for this class. At First, we have defined a pointer to *BinaryNode* as root before defining an *EType* variable as *ITEM_NOT_FOUND*. Then there are *elementAt*, *insert* and *remove* methods. But the method signatures of these methods are different than public methods. The signature of insert method is:

```
void insert(const EType& x, BinaryNode<EType>* & t);
```

In the public *insert* method, we have only one argument of type *EType*. Similarly there are *findMin*, *findMax* and *find* methods with different signatures. Then we have *makeEmpty* and *printInorder* methods. Why we are talking about these methods especially when these are the private ones. In C++, we have to do like this. In *.h* file we define the class and in .cpp file, the implementation of the class is given. The user of this class is advised not to look at the private part of the class. He should be interested only in the public part of the class. We are not the users of this class, but only the developers. In the *BinarySeachTree.h* file, we have defined two classes i.e. *BinaryNode* and *BinarySearchTree*. Are these classes private or public? Can the user of *BinarySearchTree* benefit from the *BinaryNode* class? Is it possible or not? This is an exercise for you. Study about the reference data types before coming to next lecture.

Sample Program

Here is the code of the program. *BinarySearchTree.h* file.

```
/* This file contains the declaration of binary node and the binary search tree */

    #ifndef BINARY_SEARCH_TREE_H_
    #define BINARY_SEARCH_TREE_H_

    #include <iostream.h>       // For NULL

      // Binary node and forward declaration
```

```
template <class EType>
class BinarySearchTree;

template <class EType>
class BinaryNode
{
   EType element;
   BinaryNode *left;
   BinaryNode *right;

   BinaryNode( const EType & theElement, BinaryNode *lt, BinaryNode *rt )
     : element( theElement ), left( lt ), right( rt ) { }
   friend class BinarySearchTree<EType>;
};

// BinarySearchTree class
//
// CONSTRUCTION: with ITEM_NOT_FOUND object used to signal failed finds
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// EType find( x )   --> Return item that matches x
// EType findMin( )  --> Return smallest item
// EType findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

template <class EType>
class BinarySearchTree
{
 public:
   BinarySearchTree( const EType & notFound );
   BinarySearchTree( const BinarySearchTree & rhs );
   ~BinarySearchTree( );

   const EType & findMin( ) const;
   const EType & findMax( ) const;
   const EType & find( const EType & x ) const;
   bool isEmpty( ) const;
   void printTree( ) const;

   void makeEmpty( );
   void insert( const EType & x );
   void remove( const EType & x );

   const BinarySearchTree & operator=( const BinarySearchTree & rhs );
```

```
        private:
          BinaryNode<EType> *root;
          const EType ITEM_NOT_FOUND;

          const EType & elementAt( BinaryNode<EType> *t ) const;

          void insert( const EType & x, BinaryNode<EType> * & t ) const;
          void remove( const EType & x, BinaryNode<EType> * & t ) const;
          BinaryNode<EType> * findMin( BinaryNode<EType> *t ) const;
          BinaryNode<EType> * findMax( BinaryNode<EType> *t ) const;
          BinaryNode<EType> * find( const EType & x, BinaryNode<EType> *t )
const;
          void makeEmpty( BinaryNode<EType> * & t ) const;
          void printTree( BinaryNode<EType> *t ) const;
          BinaryNode<EType> * clone( BinaryNode<EType> *t ) const;
      };
      #include "BinarySearchTree.cpp"
      #endif
```

BinarySearchTree.cpp file.

```
/* This file contains the implementation of the binary search tree */

      #include <iostream.h>
      #include "BinarySearchTree.h"

      /**
       * Construct the tree.
       */
      template <class EType>
      BinarySearchTree<EType>::BinarySearchTree( const EType & notFound ) :
        ITEM_NOT_FOUND( notFound ), root( NULL )
      {
      }

      /**
       * Copy constructor.
       */

      template <class EType>
      BinarySearchTree<EType>::
      BinarySearchTree( const BinarySearchTree<EType> & rhs ) :
        root( NULL ), ITEM_NOT_FOUND( rhs.ITEM_NOT_FOUND )
      {
         *this = rhs;
      }

      /**
```

```
 * Destructor for the tree.
 */
template <class EType>
BinarySearchTree<EType>::~BinarySearchTree( )
{
   makeEmpty( );
}

/**
 * Insert x into the tree; duplicates are ignored.
 */
template <class EType>
void BinarySearchTree<EType>::insert( const EType & x )
{
   insert( x, root );
}

/**
 * Remove x from the tree. Nothing is done if x is not found.
 */
template <class EType>
void BinarySearchTree<EType>::remove( const EType & x )
{
   remove( x, root );
}


/**
 * Find the smallest item in the tree.
 * Return smallest item or ITEM_NOT_FOUND if empty.
 */
template <class EType>
const EType & BinarySearchTree<EType>::findMin( ) const
{
   return elementAt( findMin( root ) );
}

/**
 * Find the largest item in the tree.
 * Return the largest item of ITEM_NOT_FOUND if empty.
 */
template <class EType>
const EType & BinarySearchTree<EType>::findMax( ) const
{
   return elementAt( findMax( root ) );
}

/**
 * Find item x in the tree.
 * Return the matching item or ITEM_NOT_FOUND if not found.
```

```
 */
template <class EType>
const EType & BinarySearchTree<EType>::
                    find( const EType & x ) const
{
   return elementAt( find( x, root ) );
}

/**
 * Make the tree logically empty.
 */
template <class EType>
void BinarySearchTree<EType>::makeEmpty( )
{
   makeEmpty( root );
}

/**
 * Test if the tree is logically empty.
 * Return true if empty, false otherwise.
 */
template <class EType>
bool BinarySearchTree<EType>::isEmpty( ) const
{
   return root == NULL;
}

/**
 * Print the tree contents in sorted order.
 */
template <class EType>
void BinarySearchTree<EType>::printTree( ) const
{
   if( isEmpty( ) )
      cout << "Empty tree" << endl;
   else
      printTree( root );
}

/**
 * Deep copy.
 */
template <class EType>
const BinarySearchTree<EType> &
BinarySearchTree<EType>::
operator=( const BinarySearchTree<EType> & rhs )
{
   if( this != &rhs )
   {
      makeEmpty( );
```

```
        root = clone( rhs.root );
    }
    return *this;
}

/**
 * Internal method to get element field in node t.
 * Return the element field or ITEM_NOT_FOUND if t is NULL.
 */
template <class EType>
const EType & BinarySearchTree<EType>::
elementAt( BinaryNode<EType> *t ) const
{
    if( t == NULL )
        return ITEM_NOT_FOUND;
    else
        return t->element;
}

/**
 * Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <class EType>
void BinarySearchTree<EType>::
insert( const EType & x, BinaryNode<EType> * & t ) const
{
    if( t == NULL )
        t = new BinaryNode<EType>( x, NULL, NULL );
    else if( x < t->element )
        insert( x, t->left );
    else if( t->element < x )
        insert( x, t->right );
    else
        ;  // Duplicate; do nothing
}

/**
 * Internal method to remove from a subtree.
 * x is the item to remove.
 * t is the node that roots the tree.
 * Set the new root.
 */
template <class EType>
void BinarySearchTree<EType>::
remove( const EType & x, BinaryNode<EType> * & t ) const
{
    if( t == NULL )
```

```
            return;   // Item not found; do nothing
        if( x < t->element )
            remove( x, t->left );
        else if( t->element < x )
            remove( x, t->right );
        else if( t->left != NULL && t->right != NULL ) // Two children
        {
            t->element = findMin( t->right )->element;
            remove( t->element, t->right );
        }
        else
        {
            BinaryNode<EType> *nodeToDelete = t;
            t = ( t->left != NULL ) ? t->left : t->right;
            delete nodeToDelete;
        }
    }

    /**
     * Internal method to find the smallest item in a subtree t.
     * Return node containing the smallest item.
     */
    template <class EType>
    BinaryNode<EType> *
    BinarySearchTree<EType>::findMin( BinaryNode<EType> *t ) const
    {
        if( t == NULL )
            return NULL;
        if( t->left == NULL )
            return t;
        return findMin( t->left );
    }

    /**
     * Internal method to find the largest item in a subtree t.
     * Return node containing the largest item.
     */
    template <class EType>
    BinaryNode<EType> *
    BinarySearchTree<EType>::findMax( BinaryNode<EType> *t ) const
    {
        if( t != NULL )
            while( t->right != NULL )
                t = t->right;
        return t;
    }

    /**
     * Internal method to find an item in a subtree.
     * x is item to search for.
```

```
     * t is the node that roots the tree.
     * Return node containing the matched item.
     */
    template <class EType>
    BinaryNode<EType> *
    BinarySearchTree<EType>::
    find( const EType & x, BinaryNode<EType> *t ) const
    {
      if( t == NULL )
        return NULL;
      else if( x < t->element )
        return find( x, t->left );
      else if( t->element < x )
        return find( x, t->right );
      else
        return t;    // Match
    }
/****** NONRECURSIVE VERSION*************************
    template <class EType>
    BinaryNode<EType> *
    BinarySearchTree<EType>::
    find( const EType & x, BinaryNode<EType> *t ) const
    {
      while( t != NULL )
        if( x < t->element )
          t = t->left;
        else if( t->element < x )
          t = t->right;
        else
          return t;    // Match

      return NULL;   // No match
    }
*****************************************************/

    /**
     * Internal method to make subtree empty.
     */
    template <class EType>
    void BinarySearchTree<EType>::
    makeEmpty( BinaryNode<EType> * & t ) const
    {
      if( t != NULL )
      {
        makeEmpty( t->left );
        makeEmpty( t->right );
        delete t;
      }
      t = NULL;
    }
```

```
/**
 * Internal method to print a subtree rooted at t in sorted order.
 */
template <class EType>
void BinarySearchTree<EType>::printTree( BinaryNode<EType> *t ) const
{
   if( t != NULL )
   {
      printTree( t->left );
      cout << t->element << endl;
      printTree( t->right );
   }
}

/**
 * Internal method to clone subtree.
 */
template <class EType>
BinaryNode<EType> *
BinarySearchTree<EType>::clone( BinaryNode<EType> * t ) const
{
   if( t == NULL )
      return NULL;
   else
      return new BinaryNode<EType>( t->element, clone( t->left ), clone( t->right ) );
}
```

TestBinarySearchTree.cpp file. This file contains the main program.

```
/* This file contains the test program for the binary search tree */

#include <iostream.h>
#include "BinarySearchTree.h"

   // Test program
int main( )
{
   const int ITEM_NOT_FOUND = -9999;
   BinarySearchTree<int> t( ITEM_NOT_FOUND );
   int NUMS = 30;
   int i;

   cout << "Inserting elements (1 to 30) in the tree .....)" << endl;
   for( i = 0; i <= NUMS; i++ )
      t.insert( i );

   cout << "Printing the values of the nodes in tree .....)" << endl;
   t.printTree();
```

```
        cout << "Removing the even number elements in the tree .....)" << endl;
        for( i = 0; i <= NUMS; i+= 2 )
           t.remove( i );

        cout << "Printing the values of the nodes in tree .....)" << endl;
        t.printTree();

        int abc;
        cin >> i;
        return 0;
    }
```

When we run the TestBinarySearch program the following output is obtained.

```
Inserting elements (1 to 30) in the tree .....)
Printing the values of the nodes in tree .....)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Removing the even number elements in the tree .....)
Printing the values of the nodes in tree .....)
1
```

3
5
7
9
11
13
15
17
19
21
23
25
27
29

Data Structures

Lecture No. 17

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4

Summary

- Reference Variables
- Sample Program

After demonstrating the use of const and reference variables in the interface class for a binary search tree in the previous lectures, we will now see what these reference variables are and how does internal process go on during the use of reference variables.

Reference Variables

Before proceeding ahead, there is need to know why the reference variables are used; and what benefits a programmer can get while employing them. We will discuss these all things in detail with the help of examples.

The symbol &, used for reference variable has a few different purposes with respect to its occurrence in the code. In C++ programming, we have seen that when the ampersand sign i.e. & appears in front of a variable name, it is the address operator. It returns the address of the variable in front of which it is written. Thus for example, if *x* is a variable name, then *&x ;* will return the address of the variable *x*. In general we can say that

&variablename ;

will return the address of the variable. We know that an address can be stored in a pointer. To further understand this concept, let's suppose that there are following lines in our code.

```
int x ;
int* ptr = &x;
```

The first line declares a variable *x* of type *int* while the second one declares a pointer to *int* and assigns it the address of *x*. This address of variable *x* is returned by the & sign written in front of the variable *x*. Thus a pointer variable is initialized by assigning it the address of a variable. This address of the variable is gotten by using the & sign with the variable name.

The other place where & sign can appear is the signature of the function where it

appears after the type of the parameter. Consider the insert and remove methods from BinarySearchTree class that were declared as the following.

```
void insert( const EType& x );
        void remove( const EType& x );
```

Notice that the & sign is after the type. Here we define the class and functions as templates. Whenever, we use these methods, the type EType will be replaced with a proper data type.
Suppose that we have designed the BinarySearchTree class to hold the integers only. This means that there are no templates and the class can be defined only for integers. So the insert and remove methods of this class will be as follows.

```
void insert( const int& x );
        void remove( const int& x );
```

Here, in the function signature, & sign after the type of the parameter indicates that the parameter is a reference variable. We can use this & sign with any data type i.e. built- in or user defined, of an argument to show that this is a reference variable. This is the syntax for the reference variable. Now let's see why and how we use it and what is its advantage? Let's look at an example that will explain the reference variable. Suppose we have the following three different functions.
The name of the first function is *intMinus1*, written as under.

```
//Function 1

int intMinus1( int oldVal)
{
        oldVal = oldVal – 1;
        return oldVal;
}
```

This function takes an integer as an argument. The local name of this argument is *oldVal*. The first statement in the body of the function decreases the value of *oldVal* by 1 then the next statement returns this value to the calling function.

The second function is *intMinus2*. This function takes a pointer to integer as an argument. In the statement

```
*oldVal = *oldVal – 2 ;
```

**oldVal* means the value at the location where *oldVal* is pointing to. To get the value this way is called dereferencing. After experiencing a decrease by 2, this value remains at the same location i.e. where *oldVal* is pointing to. Then the pointer to integer is returned. This function is defined as follows.

```
// Function 2

int intMinus2( int* oldVal)
{
```

```
                *oldVal = *oldVal – 2;
                return *oldVal;
}
```

The third function *intMinus3* takes a reference variable as an argument. Its definition is as under.

```
// Function 3

int intMinus3( int& oldVal)
{
                oldVal = oldVal – 3;
                return oldVal;
}
```

The & sign after the type in the signature (declaration line) indicates that this argument is a reference variable. Notice that & sign is used only in the function declaration, leaving no need for its use in the body of the function. The statements in the body of the function decrease the vale of *oldVal* by 3 and return it to the calling function.

We can see that these three functions have the same return type and their body performs the task of decreasing the passed value. However it is clear from the declaration of these functions that the way of passing argument to these functions is different. We will now see what changes occur while passing the argument in different ways. We will write calling functions those call the above three functions one by one. We will see how these calls are made and what differences happen during the process of passing same value (in different ways) to these functions. The use of call stack will help in ascertaining what happens to these local variables and parameters.
Following is the function that calls the first function i.e. *intMinus1*.

```
void caller()
{
        int myInt = 31;
        int retVal;
        retVal = intMinus1( myInt );
        cout << myInt << retVal;
}
```

In this function (caller), we declare an integer variable *myInt* and assign it a value 31. We also declare a variable *retVal* of type *int*. In the statement

```
retVal = intMinus1( myInt );
```

We call the function *intMinus1* by passing it the variable *myInt* (the value of which is 31) and assign the return value of this function call to *retVal*. In the next statement, we print the value of these two variables.

While talking about runtime environment, we noticed that an executable program

while in run, is loaded in the memory. It later becomes a process. This process is given a block of memory, which it uses during its execution. Suppose we are running many programs simultaneously i.e. browser, MS Word, Excel and dev-C++. We can also run programs written by us. The figure 17.1 shows that process4 is a program written by us. Every program, which we run takes a block of memory and becomes a process. The memory occupied by a process is further partitioned into different parts. The first part of the memory is for the code of the program. This code is in binary form i.e. it is the compiled code. Afterwards, there is some area of memory for static data that holds static and global variables. Then in the memory, there becomes the stack. This stack is used in function calls. At the end, there is some area, called heap. When we allocate memory dynamically by using *new* operator, this memory is allocated from the heap. The following figure shows the internal memory organization of a program (ourtest.exe).

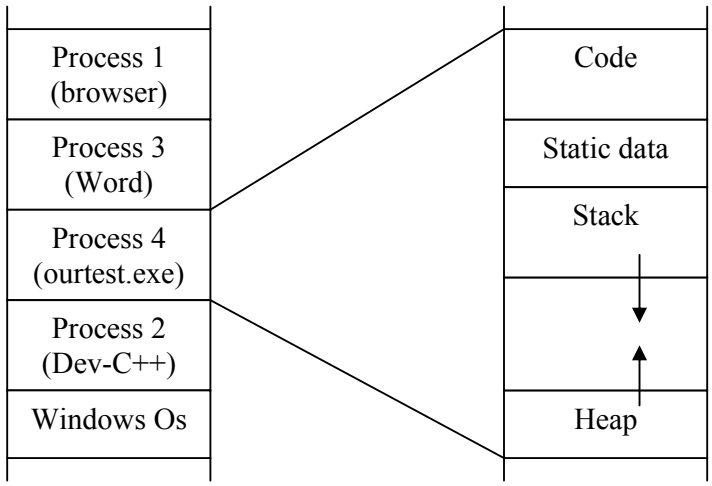


Fig 17.1: Internal memory organization of a process (ourtest.exe)

We have discussed in detail the call stack that is used in function calls. We know that when a function call is made, it's all the arguments go on the stack. The return address of the function also goes to the stack. The local variables are placed on the stack. When a function calls some other function, an activation record is made. It has much information including all these things (parameters, return address etc). The detail of activation record relates to the compiler construction course. Here we are concerned with only the call stack. Following figure shows the call stack layout.

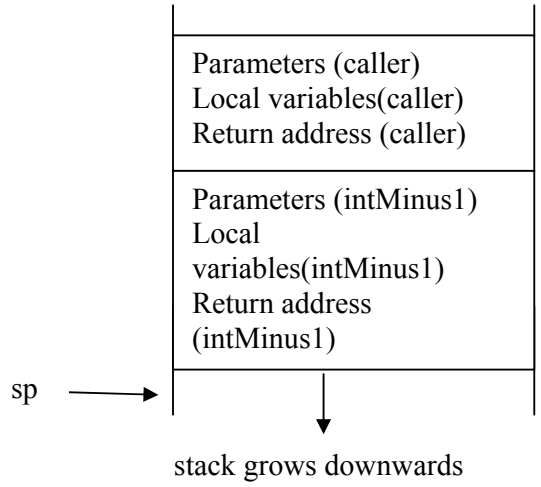


Fig 17.2: call stack layout

In the stack layout figure, there are the entries of the caller in the upper portion. This is due to the fact that the caller function itself was called (executed) by some one else. The lower portion of the stack describes the stack status when the function *intMinus1* is called. The stack contains parameters, local variables and return address of the function *intMinus1*. There is a stack pointer *sp* that points to the top of the stack (the figure shows the stack downward; it should not be confused with the top). As the caller has called *intMinus1*, this function is on the top of the stack.
The following figure shows the contents of the stack when the function *intMinus1* is called.

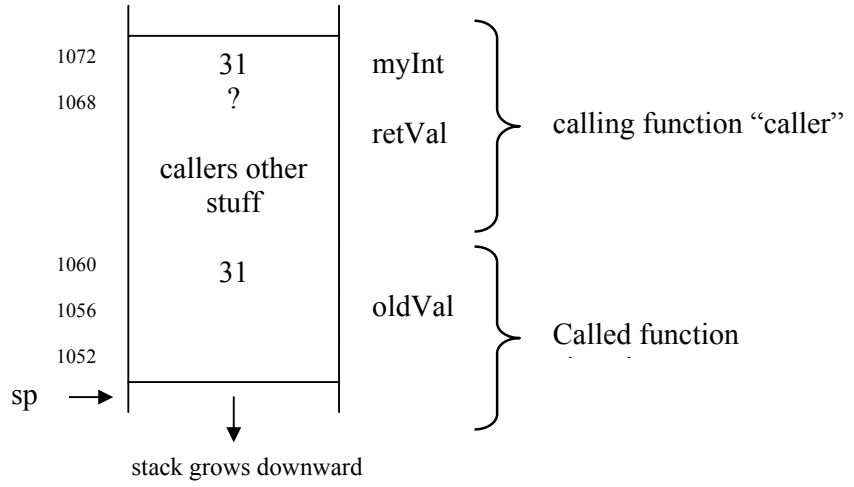


Fig 17.3: call stack layout when intMinus1 is called

The figure shows the stack in two parts. The first part that is in the upper curly bracket, shows the contents of the calling function i.e. caller. The lower part shows the contents of the called function i.e. *intMinus1*. Recall that we declared two

variables in the caller function. These variables were *myInt* and *retVal*. We also assigned a value 31 to *myInt*. This value (31) is in the memory location of *myInt* in the stack. When we declared the variable *retVal,* there was no value assigned to it. So its memory location contains nothing yet. After it, there is the other stuff of the caller function. We have also shown the memory addresses of the locations on the left hand side. We can see that the memory address of *myInt* is 1072 while that of *retVal* is 1068. Similarly, we see the addresses 1060, 1056 and 1052 further down. We note that the addresses are with a difference of four. This is because of the use of the integer variable, as an integer requires four bytes in the memory for storing. Similarly the pointer variables are also of four bytes. The memory addresses are in 32 bits that is four bytes (eight bits make a byte). Generally we start addressing the memory from down. The lowest byte is addressed 0, followed by 1, 2, 3 and so on to upward direction. If we have larger variables, for example, objects created by our defined classes, these will require more memory. In that case, the memory chunks will not be of four bytes. Rather, it will be equal to the size of the object. In C++, we have an operator *size* or *sizeof* by which the size of a type or variable, acquired in the memory, can be found. There are some machines that do not use byte addressing. These machines use word addressing. Every word is of 48 bits. All these things (memory addressing etc) relate to computer architecture course.

Now in the stack, there are the entries of the called function. There is the parameter *oldVal* of the called function in the stack. It has the value 31. This is the same value as that of *myInt* in the caller function. As studied earlier, in call by value phenomenon, when arguments (whatever type they have) are sent to a function, a copy of these arguments is made before sending. That copy is sent to the called function. The called function uses this copy of the argument. Thus a copy of *myInt* is passed to the called function *intMinus1*. Thus its parameter *oldVal* has value 31. Now *intMinus1* function uses this value (i.e. oldVal) to do its functionality. It makes alternations to this copy. The original value that is *myInt* does not change. This phenomenon is shown in the following figure. Here the function *intMinus1* decreases the value of *oldVal* by 1 and becomes 30. The original value (*myInt*) remains unchanged.

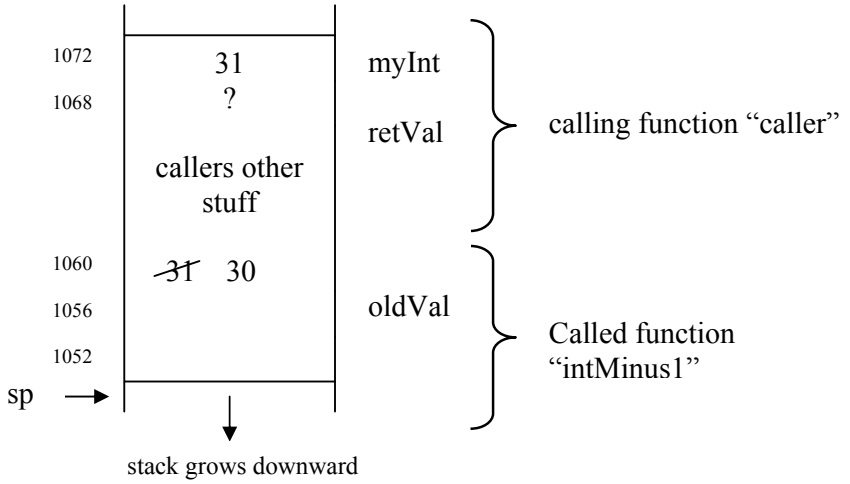


Fig 17.4: call stack layout after subtraction in intMinus1

When the return call of *intMinus1* is executed, the control comes back to the calling function. The stuff of the called function *intMinus1* is popped from the stack and the

pointer *sp* moves up accordingly. In other words, the activation record of the function *intMinus1* has been removed from the stack. The *oldval* (value of which is 30 now) is returned to the caller function. We have written in the code that this value is assigned to *retVal* variable. So in the call stack layout, the memory location of *retVal* has the value 30. The figure 17.5 shows the stack layout after return from the called function i.e. intMinus1.

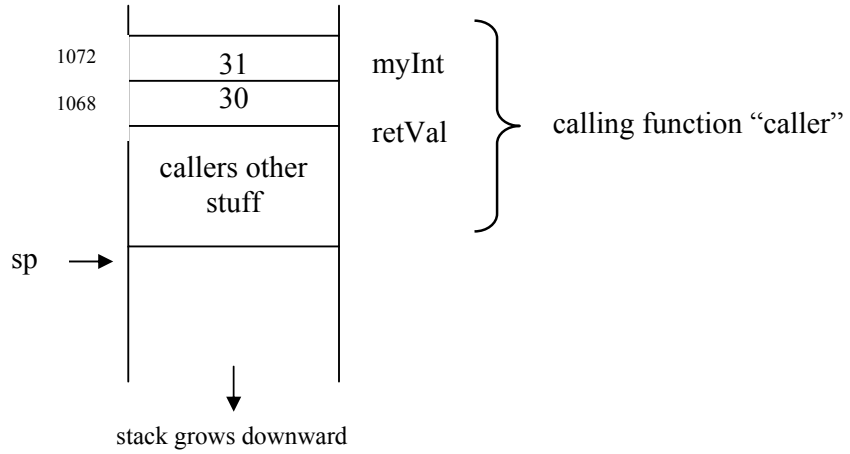


Fig 17.5 call stack layout after return from intMinus1

In the call by value phenomenon of function calling, we can pass a literal or even an expression. Thus in the previous function *caller*, we could have called the function *intMinus1* as follows.

```
void caller()
{
        int retVal;
        retVal = intMinus1( 31 ); // a literal is passed
        cout << retVal;
}
```

Here we did not put the value 31 in an integer variable. Rather, it was directly passed as an argument in the function call i.e. *intMinus1 (31)*. We are not passing a variable as it was in the previous code of the function *caller*. The calling statement also be written as under:

```
retVal = intMinus1( 30 + 1 );
```

In this statement, first the expression 30 + 1 is evaluated and then the copy of the result goes to the call stack as a value of *oldVal* of called function.

In the previous calling function (i.e. caller), it was witnessed that the value of the passed variable (myInt) remained same when we passed the variable by value. Now there may be situations where we want to actually change the value of a variable of the caller function from within the called function. In such a situation, we send the pointer to the variable (i.e. the address of the variable) as an argument. Look at the following code of the function *caller*. In this code, we call the function *intMinus2* by

passing it the address of the variable *myInt*.

```
void caller()
{
        int retVal;
        int myInt = 31;
        retVal = intMinus2( &myInt );
        cout << myInt << retVal;
}
```

Following is the function calling statement

retVal = intMinus2(&myInt);

The '&' sign before the name of the variable *myInt* means that the address of the variable is being passed. The called function will take it as a pointer. Remember that we have declared the argument of the function *intMinus2* as *int* oldVal* that means this function takes a pointer as an argument.
Now, keeping in mind that a pointer is being sent to the function, let's see what is happening in the call stack. The stack portion which has the stuff of the calling function i.e. *caller* is the same as seen in the call by value process. The variable *myInt* has value 31 and value of *retVal* will be written after returning from the called function. Its stack layout is shown in the figure 17.6 below.

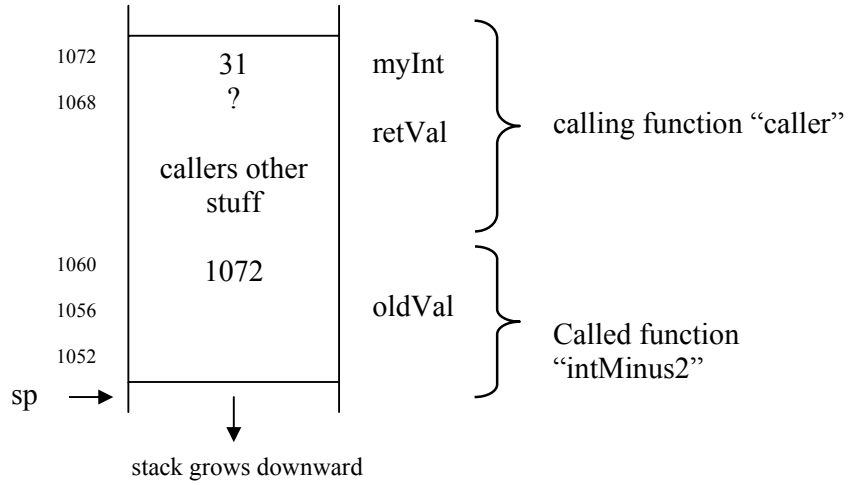


Fig 17.6 call stack layout when intMinus2 is called

In the stuff of the called function i.e. *intMinus2,* the memory location of the *oldVal* holds the value 1072. We can see on the left side of the stack in the figure above that this 1072 is the address of the variable *myInt*. Look at the statement

*oldVal = *oldVal – 2 ;

of *intMinus2*. Here **oldVal* can be expressed as 'the value at the memory address that

is in *oldVal*'. In the stack layout, we see that the memory address in *oldVal* is 1072. The value stored at the address 1072 is 31. The following statement

*oldVal = *oldVal – 2 ;

decreases this value 2 and thus brings the value at the memory address 1072 down to 29. As 1072 is the address of *myInt,* thus actually the value of *myInt* is changed. The following figure of stack layout depicts this process.

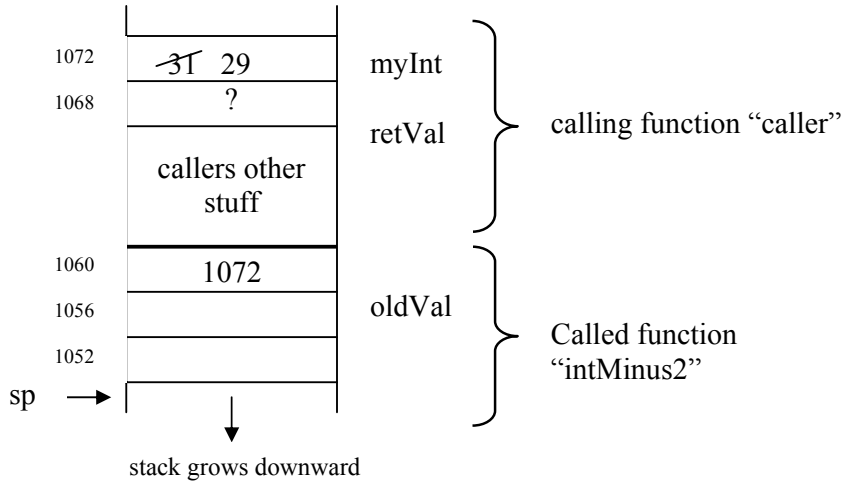


Fig 17.7: call stack layout after *oldVal = *oldVal – 2;

Now when the execution of the called function ends after returning **oldVal*, the activation record of this function is removed from the stack. The pointer of the stack comes up. The value 29 is put in the variable *retVal* of the calling function. Following is the stack layout after return from the function *intMinus2*.

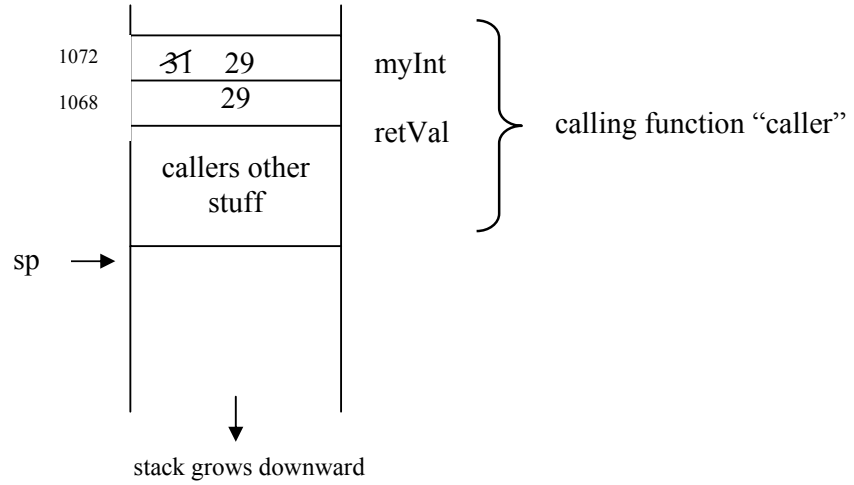


Fig 17.8: call stack after return from intMinus2

Notice that the value of *myInt* of the caller function has also been changed.

We have seen that in call by value a copy of the argument is passed and used by the called function. In this method, the original value of the argument in the calling function remains unchanged. In the other method, we saw that when we pass the address of the variable, the pointer is used to manipulate the variable data. In this case, the original value of the variable in the calling function is changed. Suppose that we want a function to change an object (variable/argument) but don't want to send the copy of the object to the function. The reason of not sending a copy is that the object may be large (as we have seen the object *Customer* in bank simulation example) and making a copy of it costs time and memory space. Say, the object *Customer* is of 500 bytes. Now when a calling function will call a function using call by value method, a copy of this object will be made on the call stack same as in our previous example a copy of *myInt* was made. The copy constructor will make a copy of this whole object that costs time. Moreover, the copy of this object will take 500 bytes on the stack that is a lot of memory consuming. There will be as many copies as the number of calls. In case of a large number of calls, there may be inefficient memory as call stack has a limited memory. So we do not use call by value methodology. Furthermore, we want to avoid the massive syntax of pointers. For this purpose, we do not use pointers in function calling. Now the question arises is there any way through which we can fulfill our requirement. (i.e. we don't want to make a copy and want to change the objective without using the pointers). The use of reference variables may be a suitable answer to this very ticklish situation. The phenomenon of function calls using reference variables is termed as call by reference. Following is the code of the caller function that involves call by reference.

```
Void caller()
{
        int retVal;
        int myInt = 31;
        retVal = intMinus3( myInt );
        cout << myInt << retVal;
}
```

Note that this is almost the same function, we wrote at first to call *intMinus1*. The only difference is that here we are calling the function *intMinus3* instead of *intMinus1*. We did not use & sign with *myInt* as used in the call for *intMinus2*. This is only due to the fact that we are not sending an address. It is pertinent to note that in the definition of the function *intMinus3*, we have used & sign with the argument variable. We have written it as

```
int intMinus3( int& oldVal)
```

This statement means that the function *intMinus3* will take a reference as an argument. In other words, a reference of the argument variable will be passed to this function. Thus the idea of reference variable is that an object will be used exactly as it exists in the caller function. The called function simply accesses it through a different name i.e. other than the name in the caller function. Here in our example, the called function *intMinus3* accesses (and uses) the passed argument i.e. *myInt* with the name *oldVal*. The function *intMinus3* cannot use the name *myInt* as it is in the caller's scope. Now the both variable names (myInt and oldVal) refer to the same object. It

means that the same memory cell where the object lies. We have read the scope of variables that the local variables can be used only by the function in which they are declared. The other functions cannot use the local variables of a function. But in reference variables, we can access the memory location of a local variable in another function by using another name. Thus in our example, we access the local variable *myInt* (actually the memory location) of the function *caller* in the other function *intMinus3* with the name *oldVal*. The following figure explains this phenomenon of reference variables with the help of call stack layout.

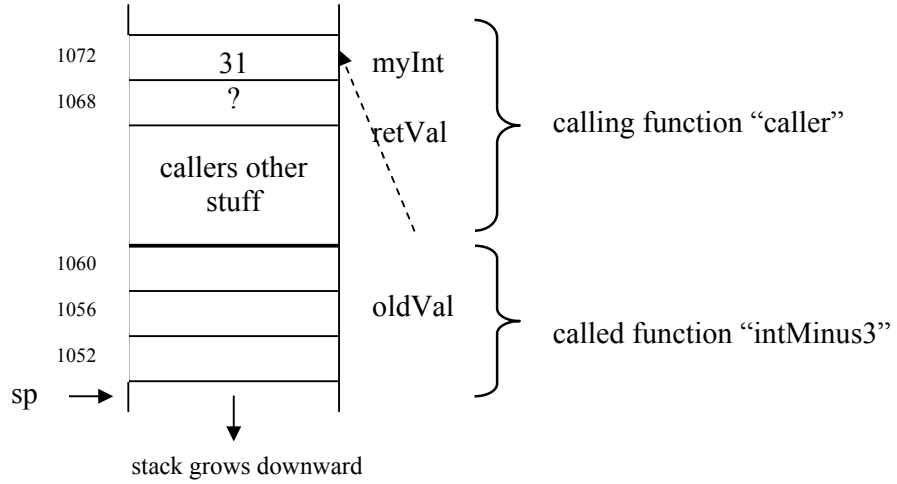


Fig 17.9: call stack when intMinus3 is called

The caller function part of the stack contains *myInt*, *retVal* and other stuff of the caller function. In the stack part *intMinus3*, the name *oldVal* shows that it has nothing in this portion but it is the other name of the memory location 1072, named as *myInt* in the caller function. The dotted arrow shows this.

In the following figure 17.10, we show the *oldVal* along with the *myInt* at same memory location. Here we want to show that the *oldVal* of called function and *myInt* of caller function are the two names of the same memory location.

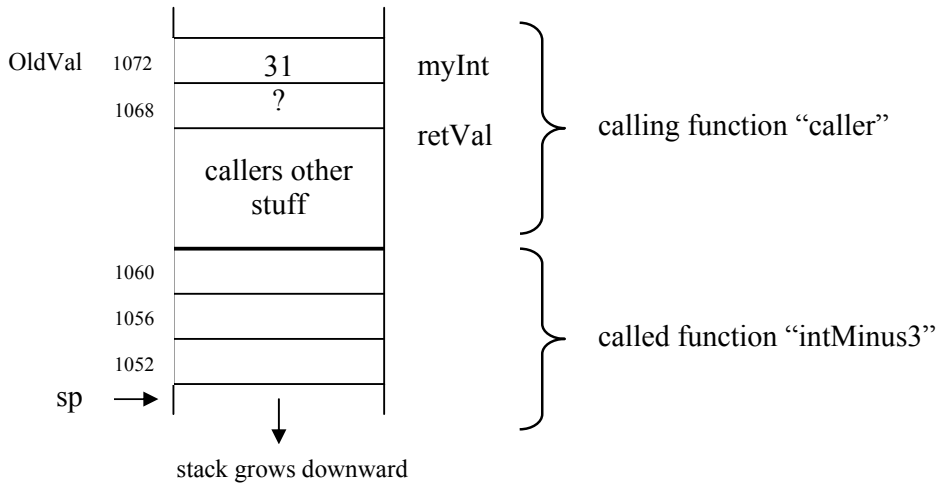


Fig 17.10: call stack when intMinus3 is clled

Now when the body of the *intMinus3* executes the statement i.e.

oldVal = oldVal – 3 ;

It changes the value at the memory location in the caller function as it is referring to that memory location. This value now becomes 28. Actually, the value of *myInt* is changed as it also refers to the same memory location. The following figure explains the stack lay out after the execution of the above statement.

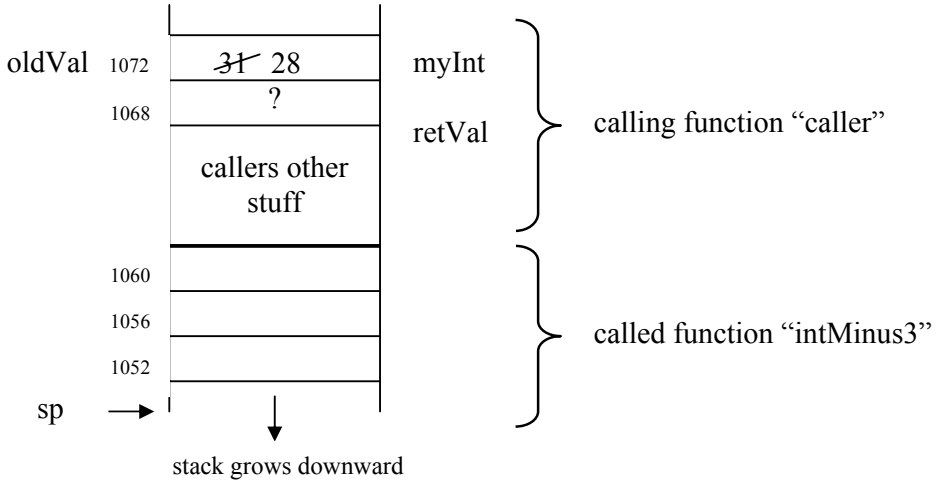


Fig 17.11: call stack layout after oldVal = oldVal - 3 ;

Now when the function intMinus3 returns, the returned value (that is 28) is written in *retVal* and the stuff (activation record) of *intMinus3* is removed. The stack layout becomes as shown in the figure below.

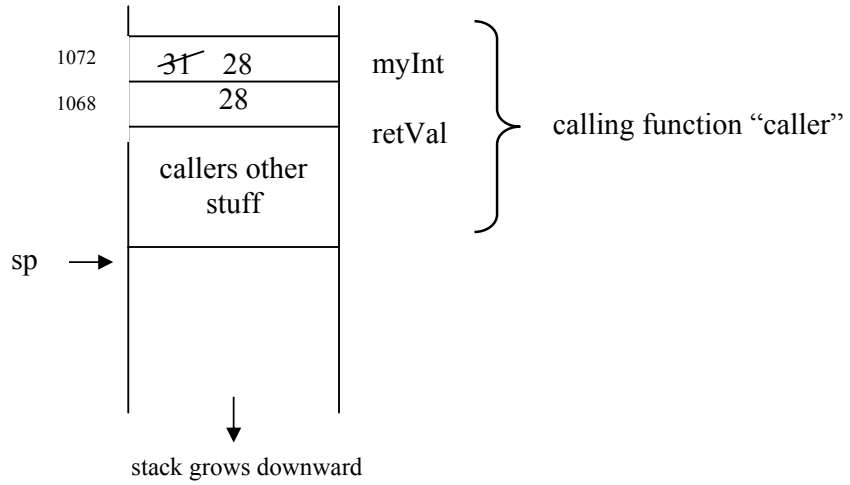


Fig 17.12: call stack layout after return from intMinus3

This phenomenon of call by reference is actually implemented by the compiler by using pointers. The obtaining of address and de-referencing will be done behind the scene. We have no concern to do this. For a programmer, it is simply a renaming abstraction, in which we rename the argument variable of the caller function and use it in the called function.

Sample Program

Following is the program, which demonstrate the above three function calls that we discussed in the previous example. We define the three functions intMinus1, intMinus2 and intMinus3. These functions accept the argument as value, pointer and reference variable respectively. The *endl* puts a new line and is used in the program for the output clarity. Here is the code of the program followed by the output of the program.

```
/*This program demonstrate tha how the value in a caller function is effected when it
is passed to a function by using call by value, by using pointers and by using call by
reference methods.
*/
#include <iostream.h>

//Function 1, call by value

int intMinus1( int oldVal)
{
                oldVal = oldVal – 1;
                return oldVal;
}


// Function 2, call by using pointers

int intMinus2( int* oldVal)
{
                *oldVal = *oldVal – 2;
                return *oldVal;
}

// Function 3, call by reference

int intMinus3( int& oldVal)
{
                oldVal = oldVal – 3;
                return oldVal;
}

void main ()
{
        int myInt = 31;
```

```
        int retVal;
        retVal = intMinus1( myInt );  //call by value
        cout << "After returning from the called function intMinus1" << endl ;
        cout << "The value returned by the called function (retVal) is   :  " << retVal ;
        cout << endl ;
        cout << "The value of the calling function's variable (myInt) is :  " << myInt ;
        cout << endl << endl;

        // now pass the argument by using pointer, also initialize the value of myInt

        myInt = 31 ;
        retVal = intMinus2( &myInt );          //call by passing a pointer
        cout << "After returning from the called function intMinus2" << endl;
        cout << "The value returned by the called function (retVal) is   :  " << retVal ;
        cout << endl;
        cout << "The value of the calling function's variable (myInt) is :  " << myInt ;
        cout << endl << endl;

        // now pass the argument by as reference, also initialize the value of myInt

        myInt = 31 ;
        retVal = intMinus3( myInt );  //call by passing a reference
        cout << "After returning from the called function intMinus3" << endl;
        cout << "The value returned by the called function (retVal) is   :  " << retVal ;
        cout << endl;
        cout << "The value of the calling function's variable (myInt) is :  " << myInt ;
}
```

Following is the output of the program.

```
After returning from the called function intMinus1
The value returned by the called function (retVal) is   :  30
The value of the calling function's variable (myInt) is :  31

After returning from the called function intMinus2
The value returned by the called function (retVal) is   :  29
The value of the calling function's variable (myInt) is :  29

After returning from the called function intMinus3
The value returned by the called function (retVal) is   :  28
The value of the calling function's variable (myInt) is :  28
```

We can see from the output of the program how the passed variable of the caller function is affected by these different ways of function calling. Note that the values of the variables used are the same as we have discussed with the help of call stack layout.

Data Structures

Lecture No. 18

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.3.3

Summary

- Reference Variables
- *const* keyword
- Tips

Reference Variables

In the last lecture we were discussing about reference variables, we saw three examples; call by value, call by reference and call by pointer. We saw the use of stack when a function is called by value, by reference or by pointer. The arguments passed to the function and local variables are pushed on to the stack.

There is one important point to note that in this course, we are using C/C++ but the usage of stack is similar in most of the computer languages like FORTRAN and Java . The syntax we are using here is C++ specific, like we are sending a parameter by pointer using *&* sign. In Java, the native data types like *int*, *float* are passed by value and the objects are passed by reference. In FORTRAN, every parameter is passed by reference. In PASCAL, you can pass a parameter by value or by reference like C++. You might have heard of ALGOL, this language had provided another way of passing parameter called *call by name*. These kinds of topics are covered in subjects like *Study of Computer Languages* or *Compiler's Theory*.

It is recommended while you are doing your degree, you study other computer languages and compare them from different aspects. Java is quite popular now a day, quite similar in syntax to C++. May be as a next language, you can study that and compare its different aspects with C/C++. The concepts like how a program is loaded into memory to become a process, how the functions are called and the role of stack etc are similar in all major languages.

we have discussed when the variables are passed by reference then behind the scene what goes on inside the stack. There are few important things to take care of while using reference variables:

One should be careful about transient objects that are stored by reference in data structures.

We know that the local variables of a function are created on call stack. Those variables are created inside the function, remains in memory until the control is inside the function and destroyed when the function exits. Activation record comprise of function call parameters, return address and local variables. The activation record remains inside stack

until the function is executing and it is destroyed once the control is returned from the function.
Let's see the following code that stores and retrieves objects in a queue:

```
void   loadCustomer( Queue &   q)
{
        Customer   c1("irfan");
        Customer   c2("sohail");
        q.enqueue( c1 );
        q.enqueue( c2 );
}
```

Above given is a small function *loadCustomer(Queue &)*, which accepts a parameter of type Queue by reference. Inside the function body, firstly, we are creating *c1* and *c2 Customer* objects. *c1* and *c2* both are initialized to string values *irfan* and *sohail* respectively. Then we queue up these objects *c1* and *c2* in the queue *q* using the *enqueue()* method and finally the function returns.
Now, the objects created inside the above function are *c1* and *c2*. As local variables are created on stack, therefore, objects are also created on stack, no matter how big is the size of the data members of the object. In the *Bank* example, in previous lecture, we saw that for each customer we have the name (*32* characters maximum), arrival time (*int* type, *4* bytes), transaction time (*int* type)and departure time (*int* type) of the customer. So the size of the *Customer* object is *44* bytes. Our *c1* and *c2* objects are created on stack and have *44* bytes occupied. It is important to mention here that we are referring each *44* bytes of allocation with the name of the object. The allocated *44* bytes are bound with the name of the object *c1* or *c2*. Another significant point here is that the function *enqueue()* accepts the object *Customer* by reference. See the code below of *serviceCustomer()* method, which is executed after the *loadCustomer()*.

```
void   serviceCustomer( Queue &   q)
{
        Customer   c   =   q.dequeue();
        cout   <<   c.getName()   <<   endl;
}
```

The *serviceCustomer(Queue &)* also accepts one parameter of type *Queue* by reference. In the first statement, it is taking out one element from the queue and assigning to newly created object *c*. Before assignment of address of *c1* object (c1 because it was inserted first), the object *c* is constructed by calling the default (parameter less) constructor. In the next statement, *c.getName()* function call is to get the name of the customer and then to print it. What do you think about it? Will this name be printed or not? Do you see any problem in its execution? In short, this statement will not work.
To see the problem in this statement, we have to understand the mechanism; where the object was created, what was pushed on the stack, when the function *loadCustomer()* returned and what had happened to the objects pushed on to the stack.
The objects *c1* and *c2*, which were created locally in *loadCustomer()* function, therefore,

they were created on stack. After creating the objects, we had added their addresses, not the objects themselves in the queue *q*. When the function *loadCustomer()* returned, the local objects *c1* and *c2* were destroyed but their addresses were there in the queue *q*. After some time, the *serviceCustomer()* is called. The address of the object is retrieved from the queue and assigned to another newly created local object *c* but when we wanted to call a method *getName()* of *c1* object using its retrieved address, we encountered the problem.

This shows that this is true that use of reference alleviate the burden of copying of object but storing of references of transient objects can create problems because the transient object (object created on stack) is destroyed when the function execution finishes.

The question arises, what can we do, if we do not want the objects created in a function to be destroyed. The answer to this is dynamic memory allocation. All the variables or objects created in a function that we want to access later are created on memory heap (sometimes called free store) using the dynamic memory allocation functions or operators like *new*. Heap is an area in computer memory that is allocated dynamically. You should remember that all the objects created using *new* operator have to be explicitly destroyed using the *delete* operator.
Let's see the modified code of *loadCustomer()* function, where the objects created in a function are not transient, means they are created on heap to be used later in the program outside the body of the function *loadCustomer()*.

```
void   loadCustomer( Queue &   q)
{
        Customer *   c1   =   new   Customer("irfan");
        Customer *   c2   =   new   Customer("sohail");
        q.enqueue( c1 ); // enqueue takes pointers
        q.enqueue( c2 );
}
```

This time, we are creating the same two objects using the *new* operator and assigning the starting addresses of those objects to *c1* and *c2* pointers. Nameless objects (objects accessed by pointers) are called *anonymous* objects.
Here *c1* and *c2* are pointers to the objects not the actual objects themselves, as it was previously. These starting addresses *c1* and *c2* of the objects are then queued using the *enqueue()* method. As the objects lie on the heap, so there will not be any problem and the objects will be accessible after the function *loadCustomer()* returns.
There is a bit tricky point to understand here. Although, the objects are created on heap but the pointer variables *c1* and *c2* are created on stack and they will be definitely destroyed after the *loadCustomer()* activation record is destroyed. Importantly, you should understand the difference between the pointer variables and the actual objects created on heap. The pointer variables *c1* and *c2* were just used to store the starting addresses of the objects inside the function *loadCustomer()*, once the function is returned the pointer variables will not be there. But as the starting addresses of the objects are put in the queue, they will be available to use later after retrieving them from queue using the

dequeue() operation. These dynamic objects will live in memory (one heap) unless explicitly deleted.
By the way, there is another heap, heap data structure that we are going to cover later in this course.
At the moment, see the layout of computer memory and heap as we previously saw in this course. Heap is an area in memory given to a process from operating system when the process does dynamic memory allocation.

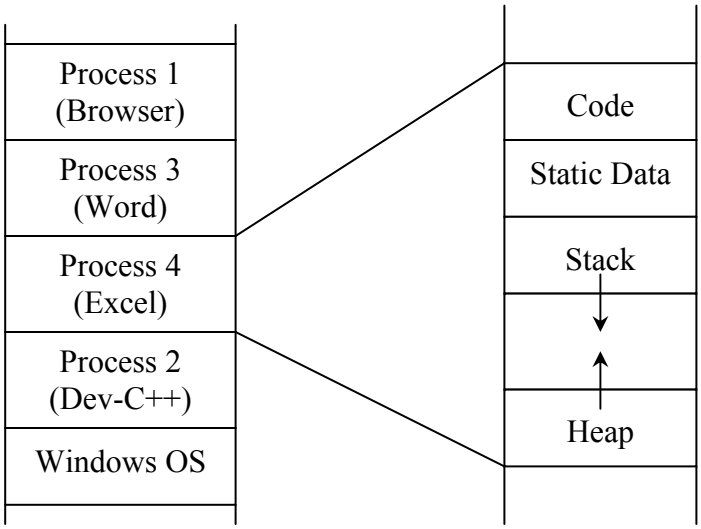


Fig 18.1: Memory Organization

One the left of the picture, we can see different processes in the computer memory. When we zoomed into the one of the processes, we saw the picture on the right. That firstly, there is a section for *code*, then for *static data* and for *stack*. Stack grows in the downward section. You can see the heap section given at the end, which grows upward. An interesting question arises here is that why the stack grows downward and heap in the upward direction. Think about an endless recursive call of a function to itself. For every invocation, there will be an *activation record* on stack. So the stack keeps on growing and growing even it overwrites the heap section. One the other hand, if your program is performing dynamic memory allocation endlessly, the heap grows in the upward direction such that it overwrites the stack section and destroys it.
You might have already understood the idea that if a process has some destructive code then it will not harm any other process, only its own destruction is caused. By the way, lot of viruses exploit the stack overflow to change the memory contents and cause further destruction to the system.
Consider that we allocate an array of *100* elements of *Customer* objects dynamically. As each object is *44* bytes, therefore, the size of memory allocated on heap will be *4400* bytes (*44 * 100*). To explain the allocation mechanism on stack and heap, let's see the figure below where the objects are created dynamically.

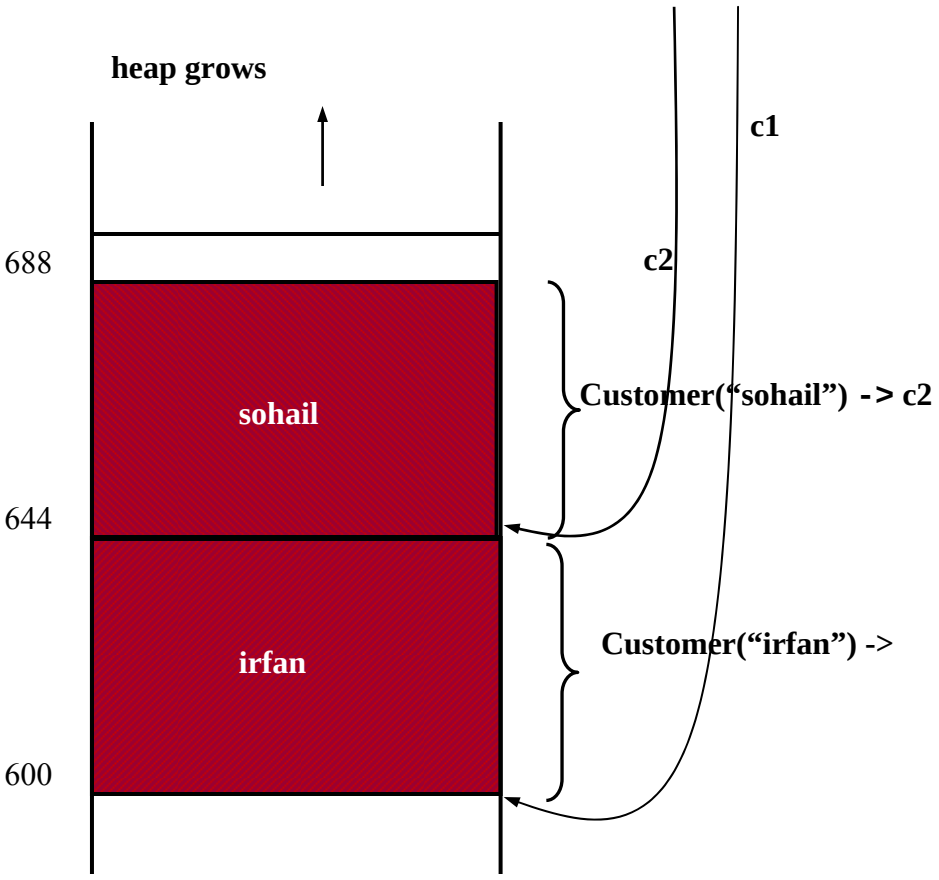


Fig 18.2: Heap layout during call to *loadCustomer*

The objects are shown in this figure by using the names of the customers inside them. Actually, there are three more *int* type variables inside each object. You can see that the object with string *irfan* is from memory address *600* to *643* and object with name customer name as *sohail* is from address *644* to *687*. Now when these objects are inserted in the queue, only their starting addresses are inserted as shown in the below figure.

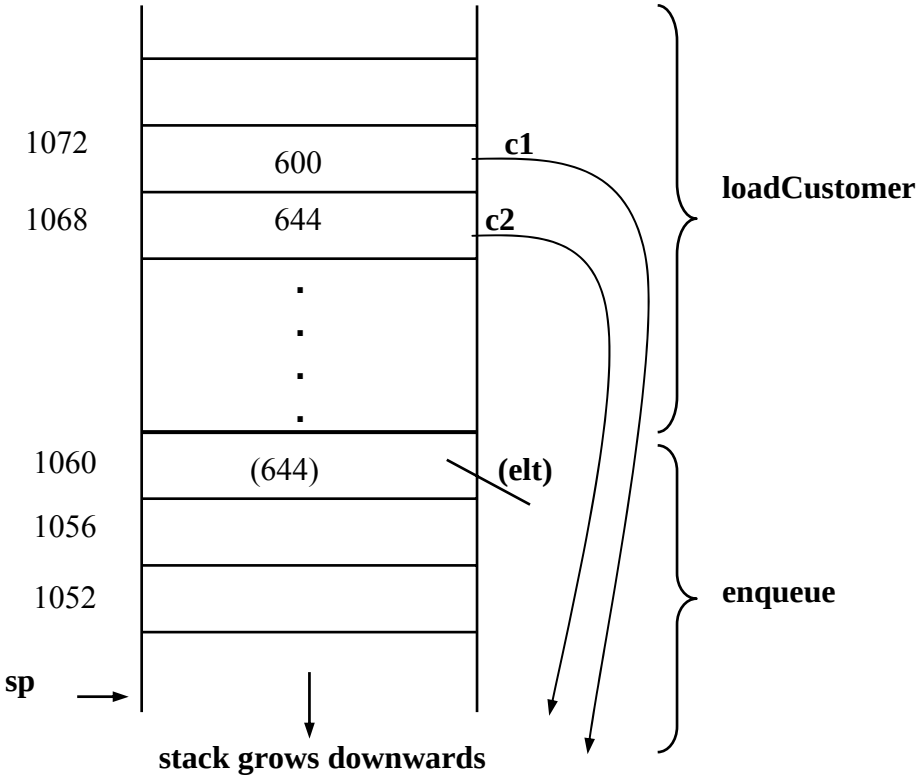


Fig 18.3: Stack layout when *q.enqueue(2)* called from *loadCustomer*

The *loadCustomer()* is being executed. It is containing two pointers *c1* and *c2* containing the addresses *600* and *643* respectively. *enqueue(elt)* method is called and the parameter values (which actually are addresses) *600* and *643* are inserted in the queue.
Because the objects have been allocated on heap, therefore, there will no issue with them. The pointer variables *c1* and *c2*, which we used to store addresses, are destroyed. But the queue *q*, which is passed by reference to *loadCustomer* will be there and it is containing the starting addresses of the *Customer* objects. Those are valid addresses of valid objects, so they can used in the program later to access the customer objects. See the function below:

```
void   serviceCustomer( Queue &   q)
{
        Customer* c   =   q.dequeue();
        cout   <<  c->getName()   <<   endl;
        delete   c; // the object in heap dies
}
```

You can see that we are taking one pointer out of the queue and in the second line calling the method of the *Customer* object *getName()* with *c->*. We are using -> operator because we are taking out pointer from the queue.
Now, we should be sure that this method will be executed successfully because the object

was created dynamically inside the *loadCustomer()* method. The last statement inside the method is *delete*, which has been used to deallocate the object.
So now, we understand that we cannot pass references to transient objects. If we want to use the objects later we create them on heap and keep the address. There is another point to mention here that in case, the object has already been deallocated and we are accessing it (calling any of its member), it may the cause the program to crash. The pointer of the object (when object has already been deallocated or released) is called *dangling* pointer.

The *const* Keyword

The *const* keyword is used for something to be constant. The actual meanings depend on where it occurs but it generally means something is to held constant. There can be constant functions, constant variables or parameters etc.
The references are pointers internally, actually they are constant pointers. You cannot perform any kind of arithmetic manipulation with references that you normally do with pointers. You must be remembering when we wrote header file for binary tree class, we had used *const* keyword many times. The *const* keyword is often used in function signatures. The function signature is also called the function prototype where we mention the function name, its parameters and return type etc.

Here are some common uses of *const* keyword.

1. The const keyword appears before a function parameter. E.g., in a chess program:

```
int movePiece(const Piece & currentPiece)
```

The function *movePiece()* above is passed one parameter, which is passed by reference. By writing *const*, we are saying that parameter must remain constant for the life of the function. If we try to change value, for example, the parameter appears on the left side of the assignment, the compiler will generate an error. This also means that if the parameter is passed to another function, that function must not change it either.

Use of *const* with reference parameters is very common. This is puzzling; why are we passing something by reference and then make it constant, i.e., don't change it? Doesn't passing by reference mean we want to change it?

Think about it, consult your C++ book and from the internet. We will discuss about the answer in the next lecture.

Tips

- The arithmetic operations we perform on pointers, cannot be performed on references
- Reference variables must be declared and initialized in one statement.
- To avoid dangling reference, don't return the reference of a local variable (transient) from a function.
- In functions that return reference, return global, static or dynamically allocated variables.
- The reference data types are used as ordinary variables without any dereference operator. We normally use arrow operator (->) with pointers.
- *const* objects cannot be assigned any other value.
- If an object is declared as *const* in a function then any further functions called from this function cannot change the value of the *const* object.

Data Structures

Lecture No. 19

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.4

Summary

- Usage of const keyword
- Degenerate Binary Search Tree
- AVL tree

Usage of const keyword

In the previous lecture, we dealt with a puzzle of constant keyword. We send a parameter to a function by using call by reference and put *const* with it. With the help of the reference variable, a function can change the value of the variable. But at the same time, we have used the *const* keyword so that it does not effect this change. With the reference parameter, we need not to make the copy of the object to send it to the calling function. In case of call by value, a copy of object is made and placed at the time of function calling in the activation record. Here the copy constructor is used to make a copy of the object. If we don't want the function to change the parameter without going for the use of time, memory creating and storing an entire copy of, it is advisable to use the reference parameter as *const*. By using the references, we are not making the copy. Moreover, with the *const* keyword, the function cannot change the object. The calling function has read only access to this object. It can use this object in the computation but can not change it. As we have marked it as constant, the function cannot alter it, even by mistake. The language is supportive in averting the mistakes.

There is another use of keyword *const*. The *const* keyword appears at the end of class member's function signature as:

```
EType& findMin( ) const;
```

This method is used to find the minimum data value in the binary tree. As you have noted in the method signature, we had written *const* at the end. Such a function cannot change or write to member variables of that class. Member variables are those which appear in the *public* or *private* part of the class. For example in the *BinaryTree,* we have *root* as a member variable. Also the *item* variable in the node class is the member variable. These are also called state variables of the class. When we create an object from the factory, it has these member variables and the methods of this class which manipulate the member variables. You will also use *set* and *get* methods, generally employed to set and get the values of the member variables. The member function can access and change the *public* and *private* member variables of a class. Suppose, we want that a member function can access the member variable but cannot change it. It means that we want to make the variables read only for that member

function. To impose that constraint on the member function, a programmer can put the keyword *const* in the end of the function. This is the way in the C++ language. In other languages, there may be alternative methods to carry out it. These features are also available in other object oriented languages. This type of usage often appears in functions that are supposed to read and return member variables. In the *Customer* example, we have used a method *getName* that returns the name of the customer. This member function just returns the value of member variable *name* which is a *private* data member. This function does not need to change the value of the variable. Now we have written a class and its functions. Why we are imposing such restrictions on it? This is the question of discipline. As a programmer when we write programs, sometimes there are unintentional mistakes. On viewing the code, it seems unbelievable that we have written like this. If these codes contain mistakes, the user will get errors. At that time, it was thought that we have imposed restrictions on the function and can avoid such mistakes at compile time or runtime. The discipline in programming is a must practice in the software engineering. We should not think that our programs are error-free. Therefore, the programming languages help in averting the common errors. One of the examples of such support is the use of *const* keyword.

There is another use of *const*. The *const* keyword appears at the beginning of the return type in function signature:

```
const EType& findMin( ) const;
```

The return type of the *findMin()* function is *ETyper&* that means a reference is returned. At the start of the return type, we have *const* keyword. How is this implemented internally? There are two ways to achieve this. Firstly, the function puts the value in a register that is taken by the caller. Secondly, the function puts the value in the stack that is a part of activation record and the caller functions gets the value at that point from the stack and use it. In the above example, we have return value as a reference as *EType&*. Can a function return a reference of its local variable? When the function ends, the local variables are destroyed. So returning the reference of local variable is a programming mistake. Therefore, a function returns the reference of some member variable of the class. By not writing the & with the return type, we are actually returning the value of the variable. In this case, a copy of the returning variable is made and returned. The copy constructor is also used here to create the copy of the object. When we are returning by value, a copy is created to ascertain whether it is a local variable or member variable. To avoid this, we use return by reference. Now we want that the variable being returned, does not get changed by the calling function especially if it is the member variable.

When we create an object from the factory, the member variable has some values. We do not want that the user of this object has direct access to these member variables. So *get* and *set* methods are used to obtain and change the value of these member variables. This is a programming practice that the values of the object should be changed while using these methods. This way, we have a clean interface. These methods are in a way sending messages to the object like give me the name of the customer or change the name of the customer. The presence of a queue object can help us send a message to it that gets an object and returns it. In these function-calling mechanisms, there are chances that we start copying the objects that is a time consuming process. If you want that the function returns the reference of the member

variable without changing the value of the member variable using this reference, a construct is put at the start of the function. It makes the reference as a *const* reference. Now the value of this member variable cannot be changed while using this reference. The compiler will give error or at the runtime, you will get the error. When we return an object from some function, a copy is created and returned. If the object is very big, it will take time. To avoid this, we return this through the reference. At this point, a programmer has to be very careful. If you do not use the *const* with the reference, your object is not safe and the caller can change the values in it.

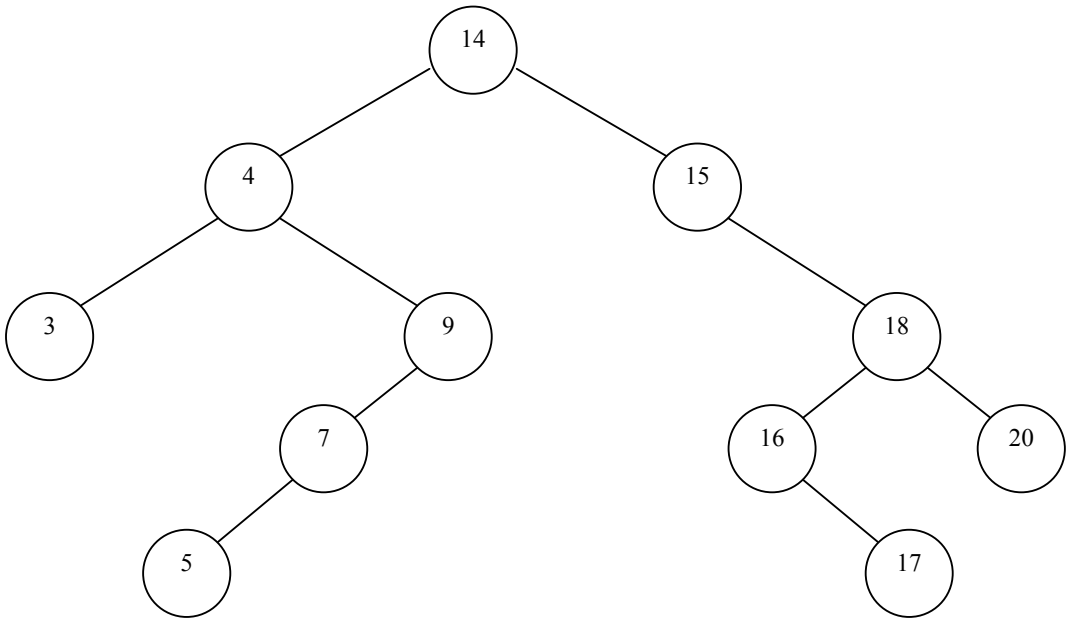
These are the common usage of *const*. It is mostly used with the member function. It is just due to the fact that we avoid creating copy of the object and secondly we get our programming disciplined. When we send a reference to some function or get a reference from some function, in both cases while using the *const*, we guard our objects. Now these objects cannot be changed. If the user of these objects needs to change the object, he should use the *set* methods of the object.

We have used such methods in the *BinarySearchTree.h* file. However, the implementation of this class has not been discussed so far. We advise you to try to write its code yourself and experiment with it.

Degenerate Binary Search Tree

Consider the tree as shown below:

BST for 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17

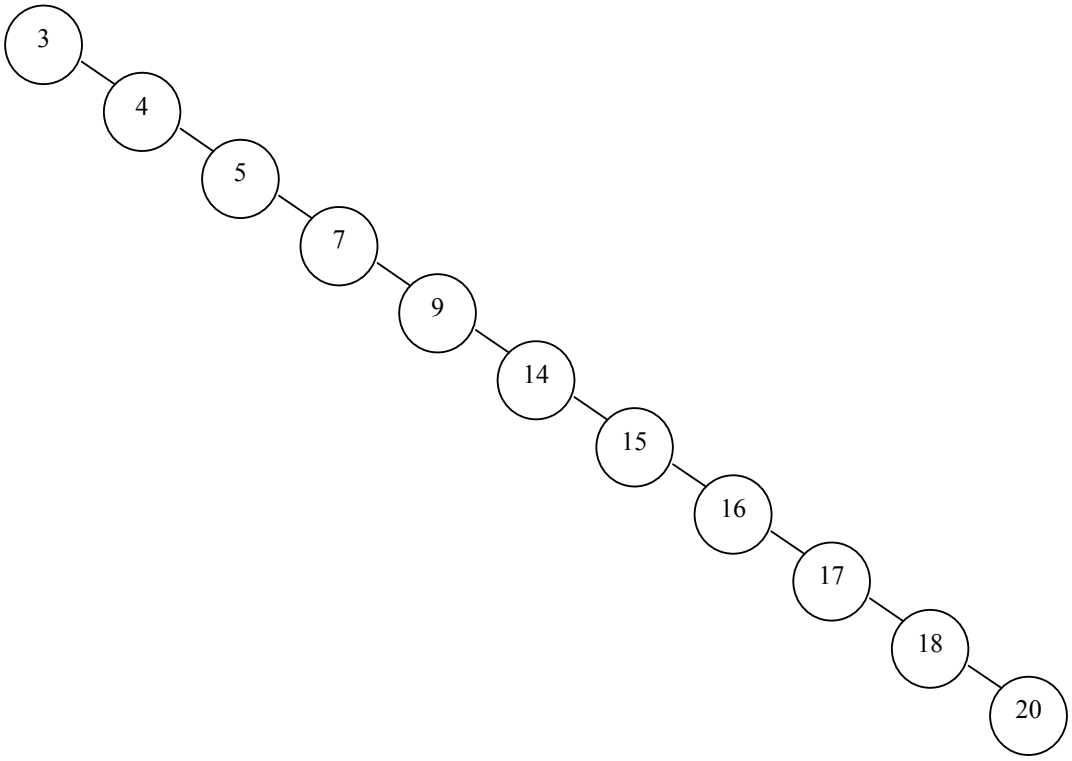


The above tree contains nodes with values as 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17 respectively. The root node is 14. The right subtree contains the numbers greater than 14 and the left subtree contains the numbers smaller than 14. This is the property of the binary search tree that at any node, the left subtree contains the numbers smaller

than this node and the right subtree contains the numbers greater than this node.

Now suppose that we are given data as 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20 to create a tree containing these numbers. Now if our *insert* method takes the data in the order as given above, what will be the shape of our tree? Try to draw a sketch of the tree with some initial numbers in your mind. The tree will look like:

BST for 3, 4, 5, 7, 9, 14, 15, 16, 17, 18, 20



It does not seem to be a binary tree. Rather, it gives a look of a linked list, as there is a link from 3 to 4, a link from 4 to 5, a link from 5 to 7. Similarly while traversing the right link of the nodes, we reached at the node 20. There is no left child of any node. That's why, it looks like a link list. What is the characteristic of the link list? In link list, every node has a pointer that points to the next node. While following this pointer, we can go to the next node. The root of this tree is 3. Now we have to find the node with value 20 in this tree. Remember that it is a tree, not a link list. We will use *find* method to search the number 20 in this tree. Now we will start from the root. As 20 is greater than 3, the recursive call to the method find will be made and we come to the next node i.e. 4. As 20 is greater than 4, so again a recursive call is generated. Similarly we will come to 5, then 7, 9 and so on. In the end, we will reach at 20 and the recursion will stop here.

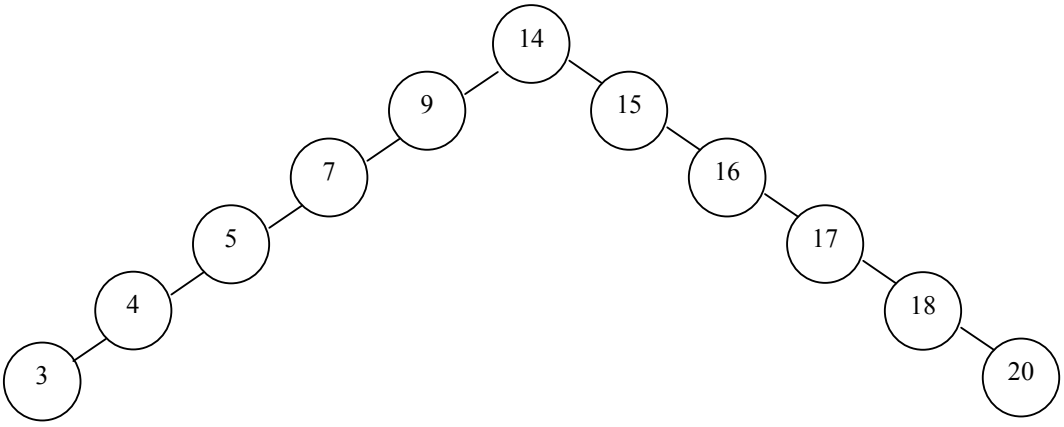
Now if we search the above tree through the method in which we started from 3, then 4, 5 and so on, this will be the same technique as adopted in the link list. How much time it will take to find the number? We have seen in the link list that if the number to be searched is at the last node, a programmer will have to traverse all the nodes. This means that in case of nodes having strength of n, the loop will execute n times.

Similarly as shown in the above tree, our find method will be called recursively equal to number of nodes in the tree. We have designed binary search tree in such a fashion that the search process is very short. You must be remembering the example of previous lecture that if we have one lakh numbers, it is possible to find the desired number in 20 iterations. If we have link list for one lakh elements, the required results can be obtained only after executing the loop for one lakh times if the element to be searched is the last element. However, in case of BST, there are only 20 steps. The BST technique, as witnessed earlier, is quite different as compared to this tree. They have both left and right subtrees. What happened with this tree? The benefit we have due to BST is not applicable here. It seems that it is a link list. This is only due to the fact that the data of the tree was given in the sorted order.

If you want to create a tree out of a sorted data with the *insert* method, it will look like the above tree. It means that you do not want to have sorted data. But it is not easy, as you might not have control over this process. Consider the example of polling. It is not possible that all the voters come to the polling station in some specific order. But in another example, if you are given a list of sorted data and asked to create a BST with this data. If you create a BST with data that is in an ascending order, it will look like a link list. In the link list, the search takes a lot of time. You have created a BST but the operations on it are working as it is a singly link list. How can we avoid that?

We know that the BST is very beneficial. One way to avoid this is that some how we get the sorted data unsorted. How this can be done. It is not possible, as data is not always provided as a complete set. Data is provided in chunks most of the times. Now what should we do? We will apply a technique here so that we can get the benefits of the BST. We should keep the tree balanced. In the above tree, nodes have left child and no right child. So this tree is not balanced. One way to achieve it is that both the left and right subtrees have the same height. While talking about the binary search tree, we discussed the height, depth and level of BST. Every node has some level. As we go down to the tree from the root, the levels of the tree increased and also the number of nodes, if all the left and right subtrees are present. You have earlier seen different examples of tree. The complete binary tree is such a tree that has all the left and right subtrees and all the leaf nodes in the end. In the complete binary tree, we can say that the number of nodes in the left subtree and right subtree are equal. If we weigh that tree on the balance, from the root, both of its sides will be equal as the number of nodes in the right subtree and left subtree are equal. If you have such a balanced binary search tree with one lakh nodes, there will need of only 20 comparisons to find a number. The levels of this tree are 20. We have also used the formula $log_2 (100,000)$. The property of such a tree is that the search comparison can be computed with the help of *log* because subtrees are switched at every comparison.

Now let's see the above tree which is like a singly link list. We will try to convert it into a balanced tree. Have a look on the following figure.



This tree seems to be a balanced tree. We have made 14 as the root. The nodes at the left side occur at the left of all the nodes i.e. left subtree of 14 is 9, the left subtree of 9 is 7, the left subtree of 7 is 5 and so on. Similarly the right subtree contains the nodes 15, 16, 17, 18, 20. This tree seems to be a balanced tree. Let's see its level. The node 14 i.e. the root is at level zero. Then at level one, we have 9 and 15. At level two, there are 7 and 16. Then 5 and 17, followed by 4 and 18. In the end, we have 3 and 20. It seems that we have twisted the tree in the middle, taking 14 as a root node. If we take other nodes like 9 or 7, these have only left subtree. Similarly if we take 15 or 16, these have right subtrees only. These nodes do not have both right and left subtree. In the earlier example, we have seen that the nodes have right and left subtrees. In that example, the data was not sorted. Here the tree is not shallow. Still we can not get the required BST. What should we do? With the sorted data, the tree can not become complete binary search tree and the search is not optimized. We want the data in unsorted form that may not be available.

We want to make a balanced tree, keeping in mind that it should not be shallow one. We could insist that every node must have left and right subtrees of same height. But this requires that the tree be a complete binary tree. To achieve it, there must be $(2^{d+1} - 1)$ data items, where d is the depth of the tree. Here we are not pleading to have unsorted data. Rather, we need as much data which could help make a balanced binary tree. If we have a tree of depth d, there will be need of $(2^{d+1} - 1)$ data items i.e. we will have left and right subtrees of every node with the same height. Now think yourself that is it possible that whenever you build a tree or someone uses your BST class can fulfill this condition. This is not possible that whenever we are going to create a tree, there will be $(2^{d+1} - 1)$ data items for a tree of depth d. The reason is that most of the time you do not have control over the data. Therefore this is too rigid condition. So this is also not a practical solution.

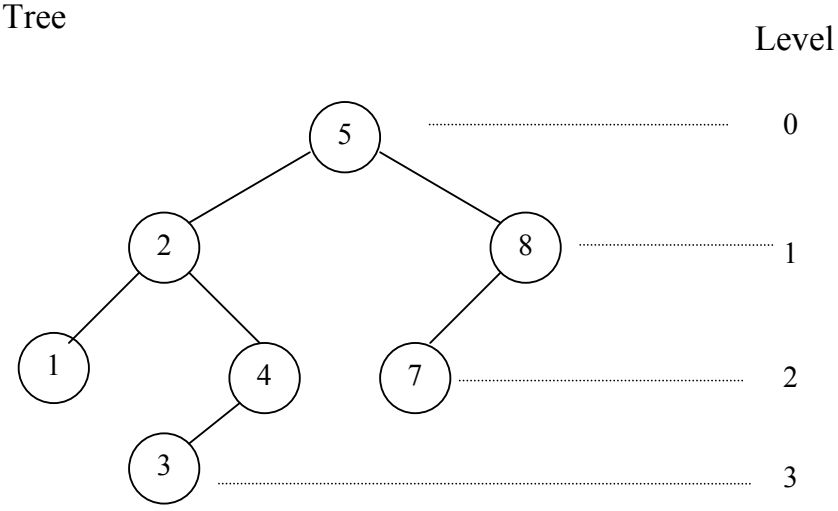
AVL Tree

AVL tree has been named after two persons Adelson-Velskii and Landis. These two had devised a technique to make the tree balanced. According to them, an AVL tree is identical to a BST, barring the following possible differences:

- Height of the left and right subtrees may differ by at most 1.
- Height of an empty tree is defined to be (–1).

We can calculate the height of a subtree by counting its levels from the bottom. At some node, we calculate the height of its left subtree and right subtree and get the difference between them. Let's understand this with the help of following fig.

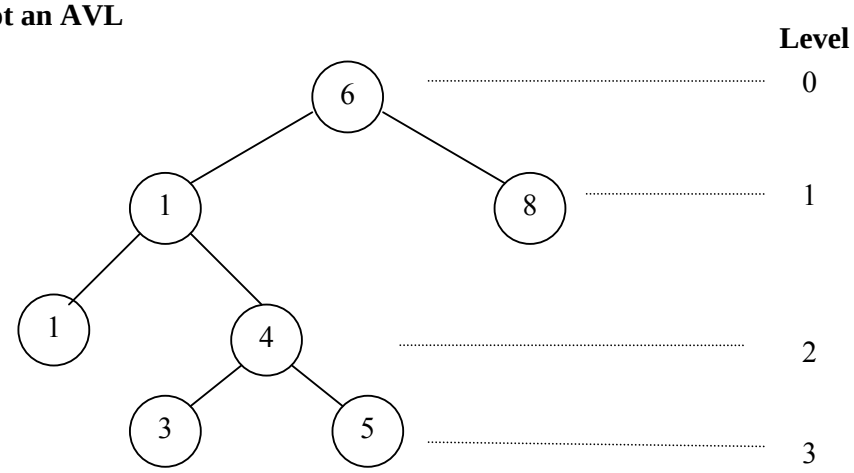
An AVL Tree



This is an AVL tree. The root of the tree is 5. At next level, we have 2 and 8, followed by 1, 4 and 7 at next level where 1, 4 are left and right subtrees of node 2 and 7 is the left subtree of node 8. At the level three, we have 3. We have shown the levels in the figure at the right side. The root is at level 0, followed by the levels 1, 2 and 3. Now see the height of the left subtree of 5. It is 3. Similarly the height of the right subtree is 2. Now we have to calculate the difference of the height of left subtree and right subtree of 5. The height of left subtree of 5 is 3 and height of right subtree of 5 is 2. So the difference is 1. Similarly, we can have a tree in which right subtree is deeper than left subtree. The condition in the AVL tree is that at any node the height of left subtree can be one more or one less than the height of right subtree. These heights, of course, can be equal. The difference of heights can not be more than 1. This difference can be -1 if we subtract the height of left subtree from right subtree where the height of left subtree is one less than the height of right subtree. Remember that this condition is not at the root. It should satisfy at any level at any node. Let's analyze the height of left subtree and right subtree of node 2. This should be -1, 0 or 1. The height of left subtree of node 2 is 1 while that of right subtree of the node 2 is 2. Therefore the absolute difference between them is 1. Similarly at node 8, the height of left subtree is 1 and right subtree does not exist so its height is zero. Therefore the difference is 1. At leaves, the height is zero, as there is no left or right subtree. In the above figure, the balanced condition is satisfactory at every level and node. Such trees have a special structure.

Let's see another example. Here is the diagram of the tree.

- **Not an AVL**



The height of the left subtree of node 6 is three whereas the height of the right subtree is one. Therefore the difference is 2. The balanced condition is not satisfactory. Therefore, it is not an AVL tree.

Let's give this condition a formal shape that will become a guiding principle for us while creating a tree. We will try to satisfy this condition during the insertion of a node in the tree or a deletion of a node from the tree. We will also see later how we can enforce this condition satisfactorily on our tree. As a result, we will get a tree whose structure will not be like a singly linked list.

The definition of height of a tree is:

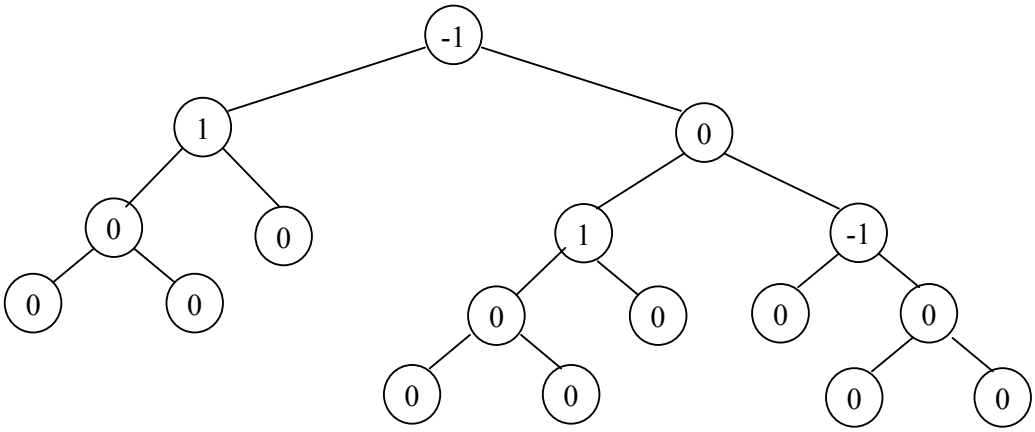
- The *height* of a binary tree is the maximum level of its leaves (also called the depth).

The height of a tree is the longest path from the root to the leaf. This can also be calculated as the maximum level of the tree. If we have to calculate the height of some node, we should start counting the levels from that node.

The balance of a node is defined as:

- The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.

Here, for example, is a balanced tree whose each node has an indicated balance of 1, 0, or –1.



In this example, we have shown the balance of each node instead of the data item. In the root node, there is the value -1. With this information, you know that the height of the right subtree at this node is one greater than that of the left subtree. In the left subtree of the root, we have node with value 1. You can understand from this example that the height of the right subtree at this node is one less than the height of the left subtree. In this tree, some nodes have balance -1, 0 or 1. You have been thinking that we have to calculate the balance of each node. How can we do that? When we create a tree, there will be a need of some information on the balance factor of each node. With the help of this information, we will try to balance the tree. So after getting this balance factor for each node, we will be able to create a balance tree even with the sorted data. There are other cases, which we will discuss, in the next lecture. In short, a balance tree with any kind of data facilitates the search process.

Data Structures

Lecture No. 20

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4

4.4

Summary

- AVL Tree
- Insertion in AVL Tree
- Example (AVL Tree Building)

We will continue the discussion on AVL tree in this lecture. Before going ahead, it will be better to recap things talked about in the previous lecture. We built a balanced search tree (BST) with sorted data. The numbers put in that tree were in increasing sorted order. The tree built in this way was like a linked list. It was witnessed that the use of the tree data structure can help make the process of searches faster. We have seen that in linked list or array, the searches are very time consuming. A loop is executed from start of the list up to the end. Due to this fact, we started using tree data structure. It was evident that in case, both the left and right sub-trees of a tree are almost equal, a tree of n nodes will have $log_2\ n$ levels. If we want to search an item in this tree, the required result can be achieved, whether the item is found or not, at the maximum in the $log\ n$ comparisons. Suppose we have 100,000 items (number or names) and have built a balanced search tree of these items. In 20 (i.e. log 100000) comparisons, it will be possible to tell whether an item is there or not in these 100,000 items.

AVL Tree

In the year 1962, two Russian scientists, Adelson-Velskii and Landis, proposed the criteria to save the binary search tree (BST) from its degenerate form. This was an effort to propose the development of a balanced search tree by considering the height as a standard. This tree is known as AVL tree. The name AVL is an acronym of the names of these two scientists.

An AVL tree is identical to a BST, barring one difference i.e. the height of the left and right sub-trees can differ by at most 1. Moreover, the height of an empty tree is defined to be (–1).

Keeping in mind the idea of the level of a tree, we can understand that if the root of a tree is at level zero, its two children (subtrees) i.e. nodes will be at level 1. At level 2, there will be 4 nodes in case of a complete binary tree. Similarly at level 3, the number of nodes will be 8 and so on. As discussed earlier, in a complete binary tree, the number of nodes at any level k will be 2^k. We have also seen the level order

traversal of a tree. The term height is identical to the level of a tree. Following is the figure of a tree in which level/height of nodes is shown.

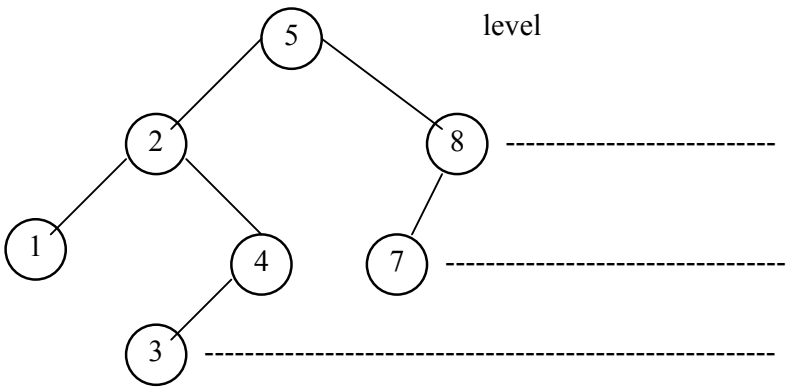


Fig 20.1: levels of nodes in a tree

Here in the figure, the root node i.e. 5 is at the height zero. The next two nodes 2 and 8 are at height (or level) 1. Then the nodes 1, 4 and 7 are at height 2 i.e. two levels below the root. At the last, the single node 3 is at level (height) 3. Looking at the figure, we can say that the maximum height of the tree is 3. AVL states that a tree should be formed in such a form that the difference of the heights (maximum no of levels i.e. depth) of left and right sub-trees of a node should not be greater than 1. The difference between the height of left subtree and height of right subtree is called the balance of the node. In an AVL tree, the balance (also called balance factor) of a node will be 1,0 or –1 depending on whether the height of its left subtree is greater than, equal to or less than the height of its right subtree.

Now consider the tree in the figure 20.1. Its root node is 5. Now go to its left subtree and find the deepest node in this subtree. We see that node 3 is at the deepest level. The level of this deepest node is 3, which means the height of this left subtree is 3. Now from node 5, go to its right subtree and find the deepest level of a node. The node 7 is the deepest node in this right subtree and its level is 2. This means that the height of right subtree is 2. Thus the difference of height of left subtree (i.e. 3) and height of right subtree (i.e. 2) is 1. So according to the AVL definition, this tree is balanced one. But we know that the AVL definition does not apply only to the root node of the tree. Every node (non-leaf or leaf) should fulfill this definition. This means that the balance of every node should be 1, 0 or –1. Otherwise, it will not be an AVL tree.

Now consider the node 2 and apply the definition on it. Let's see the result. The left subtree of node 2 has the node 1 at deepest level i.e. level 2. The node 2, itself, is at level 1, so the height of the left subtree of node 2 is 2-1 i.e. 1. Now look at the right subtree of node 2. The deepest level of this right subtree is 3 where the node 3 exists. The height of this right subtree of node 2 will be 3 –1 = 2 as the level of node 2 is 1. Now the difference of the height of left subtree (i.e. 1) and height of the right subtree (i.e. 2) is –1. We subtract the height of left subtree from the height of the right subtree and see that node 2 also fulfills the AVL definition. Similarly we can see that all other nodes of the tree (figure 20.1) fulfill the AVL definition. This means that the balance

of each node is 1, 0 or –1. Thus it is an AVL tree, also called the balanced tree. The following figure shows the tree with the balance of each node.

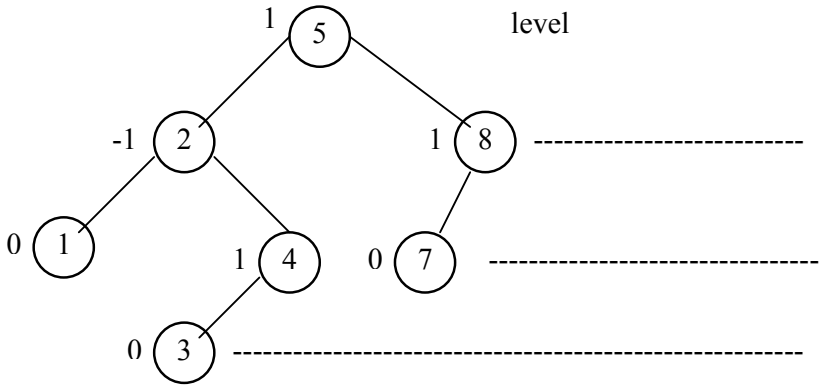


Fig 20.2: balance of nodes in an AVL

Let's consider a tree where the condition of an AVL tree is not being fulfilled. The following figure shows such a tree in which the balance of a node (that is root node 6) is greater than 1. In this case, we see that the left subtree of node 6 has height 3 as its deepest nodes 3 and 5 are at level 3. Whereas the height of its right subtree is 1 as the deepest node of right subtree is 8 i.e. level 1. Thus the difference of heights (i.e. balance) is 2. But according to AVL definition, the balance should be1, 0 or –1. As shown in the figure, this node 6 is only the node that violates the AVL definition (as its balance is other than 1, 0 and -1). The other nodes fulfill the AVL definition. We know that to be an AVL tree, each node of the tree should fulfill the definition. Here in this tree, the node 6 violates this definition so this is not an AVL tree.

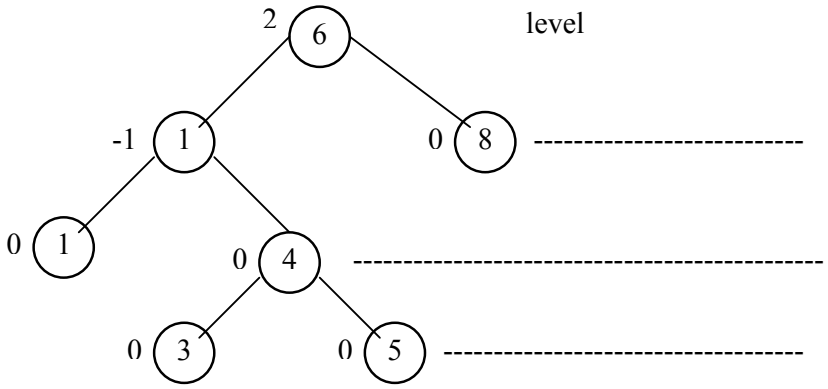


Fig 20.3: not an AVL tree

From the above discussion, we encounter two terms i.e. height and balance which can be defined as under.

Height

The height of a binary tree is the maximum level of its leaves. This is the same definition as of *depth* of a tree.

Balance

The balance of a node in a binary search tree is defined as the height of its left subtree minus height of its right subtree. In other words, at a particular node, the difference in heights of its left and right subtree gives the balance of the node.

The following figure shows a balanced tree. In this figure the balance of each node is shown along with. We can see that each node has a balance 1, 0 or –1.

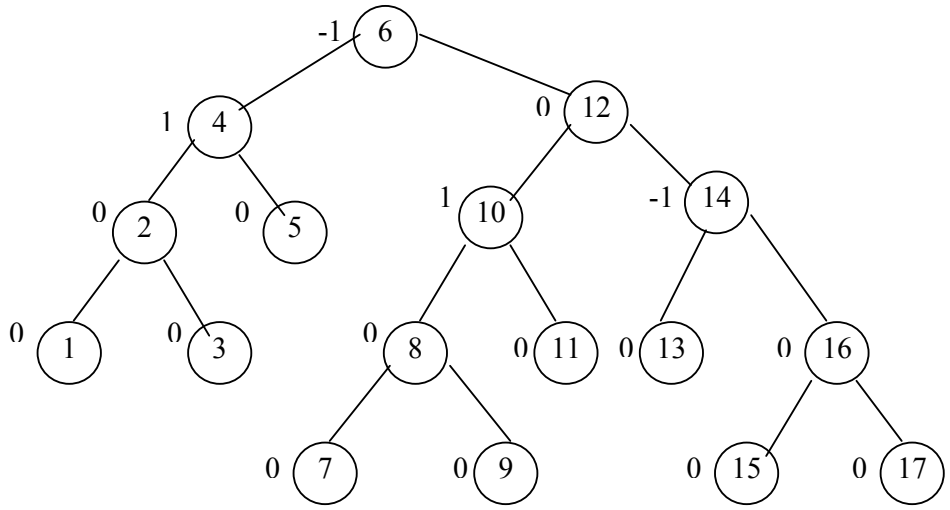


Fig 20.4: A balanced binary tree

Here in the figure, we see that the balance of the root (i.e. node 6) is –1. We can find out this balance. The deepest level of the left subtree is 3 where the nodes 1 and 3 are located. Thus the height of left subtree is 3. In the right subtree, we see some leaf nodes at level 3 while some are found at level 4. But we know that the height of the tree is the maximum level. So 4 is the height of the right subtree. Now we know that the balance of the root node will be the result of height of left subtree minus the height of right subtree. Thus the balance of the root node is 3 – 4 = -1. Similarly we can confirm the balance of other nodes. The confirmation of balance of the other nodes of the tree can be done. You should do it as an exercise. The process of height computation should be understood as it is used for the insertion and deletion of nodes in an AVL tree. We may come across a situation, when the tree does not remain balanced due to insertion or deletion. For making it a balanced one, we have to carry out the height computations.

While dealing with AVL trees, we have to keep the information of balance factor of the nodes along with the data of nodes. Similarly, a programmer has to have additional information (i.e. balance) of the nodes while writing code for AVL tree.

Insertion of Node in an AVL Tree

Now let's see the process of insertion in an AVL tree. We have to take care that the tree should remain AVL tree after the insertion of new node(s) in it. We will now see how an AVL tree is affected by the insertion of nodes.

We have discussed the process of inserting a new node in a binary search tree in previous lectures. To insert a node in a BST, we compare its data with the root node. If the new data item is less than the root node item in a particular order, this data item will hold its place in the left subtree of the root. Now we compare the new data item with the root of this left subtree and decide its place. Thus at last, the new data item becomes a leaf node at a proper place. After inserting the new data item, if we traverse the tree with the inorder traversal, then that data item will become at its appropriate position in the data items. To further understand the insertion process, let's consider the tree of figure 20.4. The following figure (Fig 20.5) shows the same tree with the difference that each node shows the balance along with the data item. We know that a new node will be inserted as a leaf node. This will be inserted where the facility of adding a node is available. In the figure, we have indicated the positions where a new node can be added. We have used two labels *B* and *U* for different positions where a node can be added. The label B indicates that if we add a node at this position, the tree will remain balanced tree. On the other hand, the addition of a node at the position labeled as U1, U2 ….U12, the tree will become unbalanced. That means that at some node the difference of heights of left and right subtree will become greater than 1.

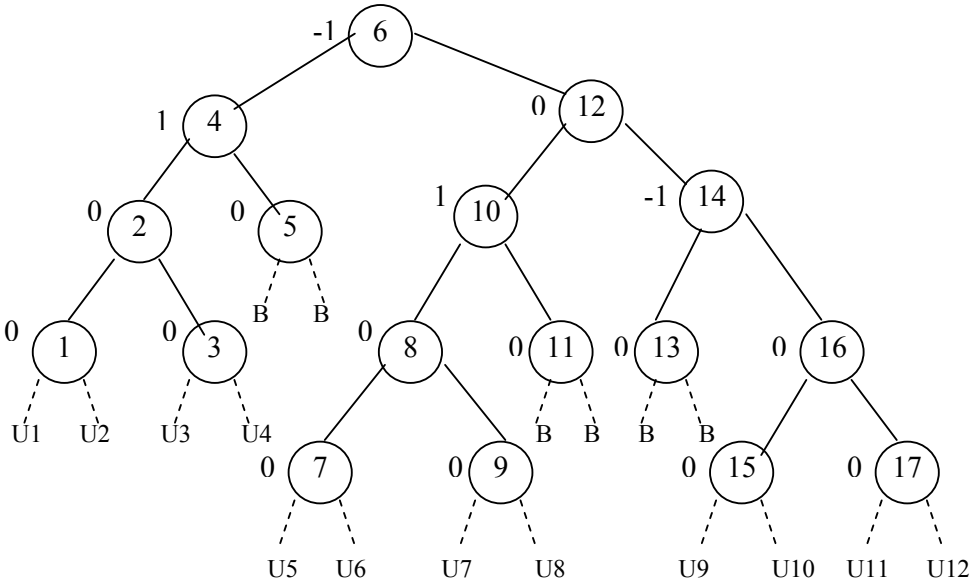


Fig 20.5: Insertions and effect in a balanced tree

By looking at the labels B, U1, U2 …….U12, we conclude some conditions that will be implemented while writing the code for insert method of a balanced tree.
We may conclude that the tree becomes unbalanced only if the newly inserted node

- Is a left descendent of a node that previously had a balance of 1
  (in the figure 20.5 these positions are U1, U2 …..U8)
- Or is a descendent of a node that previously had a balance of –1
  (in the tree in fig 20.5 these positions are U9, U10, U11 and U12)

The above conditions are obvious. The balance 1 of a node indicates that the height of its left subtree is 1 more than the height of its right subtree. Now if we add a node to this left subtree, it will increase the level of the tree by 1. Thus the difference of heights will become 2. It violates the AVL rule, making the tree unbalanced.
Similarly the balance –1 of a node indicates that the right subtree of this node is one level deep than the left subtree of the node. Now if the new node is added in the right subtree, this right subtree will become deeper. Its depth/height will increase as a new node is added at a new level that will increase the level of the tree and the height. Thus the balance of the node, that previously has a balance –1, will become –2.
The following figure (Fig 20.6) depicts this rule. In this figure, we have associated the new positions with their grand parent. The figure shows that U1, U2, U3 and U4 are the left descendents of the node that has a balance 1. So according to the condition, the insertion of new node at these positions will unbalance the tree. Similarly the positions U5, U6, U7 and U8 are the left descendents of the node that has a balance 1. Moreover we see that the positions U9, U10, U11 and U12 are the right descendents of the node that has balance –1. So according to the second condition as stated earlier, the insertion of a new node at these positions would unbalance the tree.

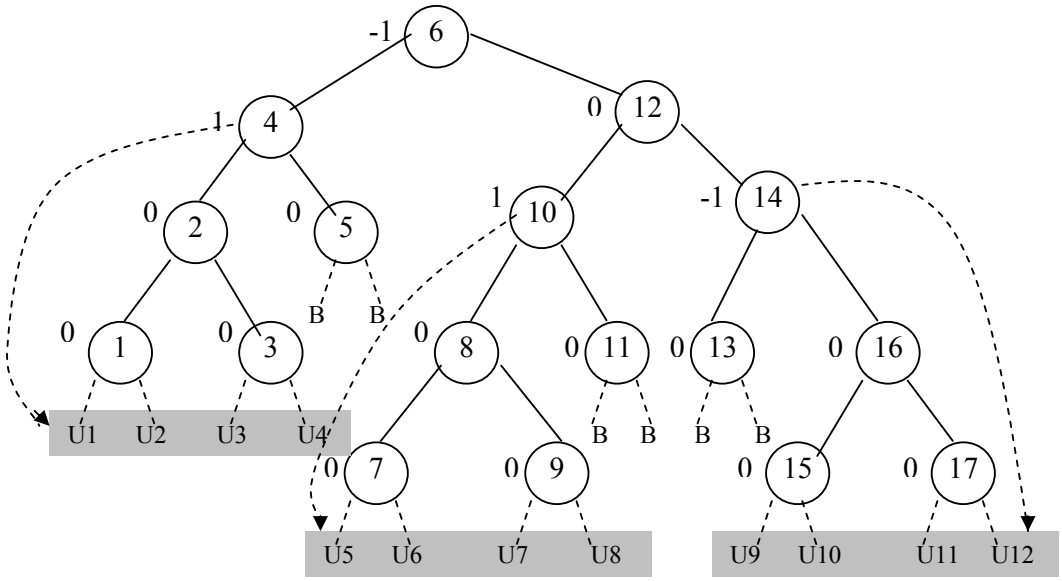


Fig 20.6: Insertions and effect in a balanced tree

Now let's discuss what should we do when the insertion of a node makes the tree unbalanced. For this purpose, consider the node that has a balance 1 in the previous tree. This is the root of the left subtree of the previous tree. This tree is shown as shaded in the following figure.

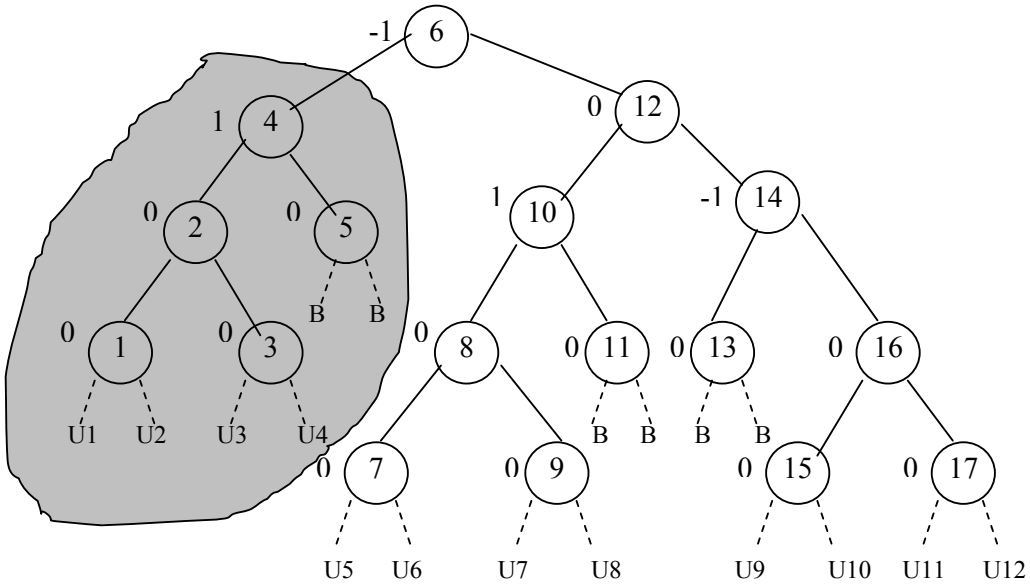
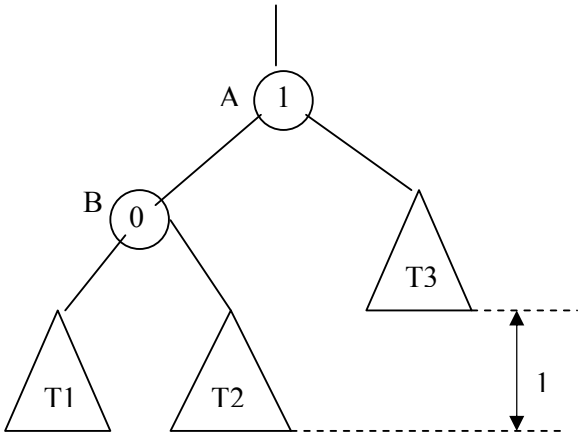


Fig 20.7: The node that has balance 1 under consideration

We will now focus our discussion on this left subtree of node having balance 1 before applying it to other nodes. Look at the following figure (Fig 20.8). Here we are talking about the tree that has a node with balance 1 as the root. We did not mention the other part of the tree. We indicate the root node of this left subtree with label A. It has balance 1. The label B mentions the first node of its left subtree. Here we did not mention other nodes individually. Rather, we show a triangle that depicts all the nodes in subtrees. The triangle T3 encloses the right subtree of the node A. We are not concerned about the number of nodes in it. The triangles T1 and T2 mention the left and right subtree of the B node respectively. The balance of node B is 0 that describes that its left and right subtrees are at same height. This is also shown in the figure. Similarly we see that the balance of node A is 1 i.e. its left subtree is one level deep than its right subtree. The dotted lines in the figure show that the difference of depth/height of left and right subtree of node A is 1 and that is the balance of node A.



Now considering the notations of figure 20.8, let's insert a new node in this tree and observe the effect of this insertion in the tree. The new node can be inserted in the tree T1, T2 or T3. We suppose that the new node goes to the tree T1. We know that this new node will not replace any node in the tree. Rather, it will be added as a leaf node at the next level in this tree (T1). The following figure (fig 20.9) shows this phenomenon.

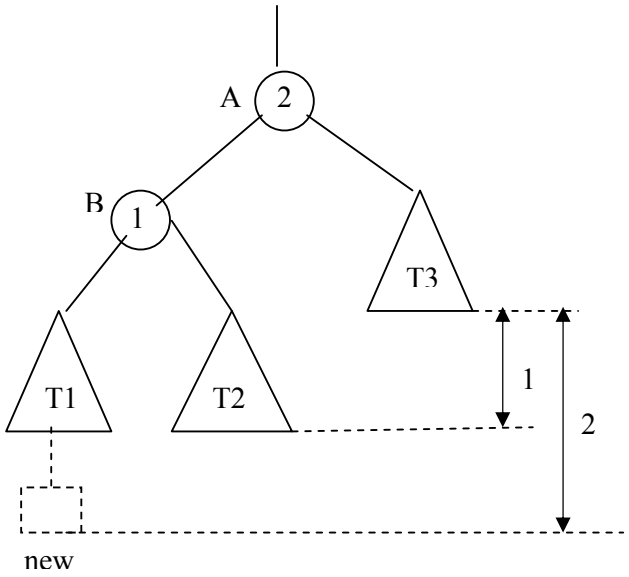


Fig 20.9: Inserting new node in AVL tree

Due to the increase of level in T1, its difference with the right subtree of node A (i.e. T3) will become 2. This is shown with the help of dotted line in the above figure. This difference will affect the balances of node A and B. Now the balance of node A becomes 2 while balance of node B becomes 1. These new balances are also shown in the figure. Now due to the balance of node A (that is 2), the AVL condition has been violated. This condition states that in an AVL tree the balance of a node cannot be other than 1, 0 or –1. Thus the tree in fig 20.9 is not a balanced (AVL) tree.

Now the question arises what a programmer should do in case of violation of AVL condition .In case of a binary search tree, we insert the data in a particular order. So that at any time if we traverse the tree with inorder traversal, only sorted data could be obtained. The order of the data depends on its nature. For example, if the data is numbers, these may be in ascending order. If we are storing letters, then A is less than B and B is less than C. Thus the letters are generally in the order A, B, C ……. This order of letters is called lexographic order. Our dictionaries and lists of names follow this order.

If we want that the inorder traversal of the tree should give us the sorted data, it will not be necessary that the nodes of these data items in the tree should be at particular positions. While building a tree, two things should be kept in mind. Firstly, the tree should be a binary tree. Secondly, its inorder traversal should give the data in a sorted order. Adelson-Velskii and Landis considered these two points. They said that if we see that after insertion, the tree is going to be unbalanced. Then the things should be reorganized in such a way that the balance of nodes should fulfill the AVL condition. But the inorder traversal should remain the same.

Now let's see the example of tree in figure 20.9 and look what we should do to balance the tree in such a way that the inorder traversal of the tree remains the same. We have seen in figure 20.9 that the new node is inserted in the tree T1 as a new leaf node. Thus T1has been modified and its level is increased by 1. Now due to this, the difference of T1 and T3 is 2. This difference is the balance of node A as T1 and T3 are its left and right subtrees respectively. The inorder traversal of this tree gives us the result as given below.

T1 B T2 A T3

Now we rearrange the tree and it is shown in the following figure i.e. Fig 20.10.

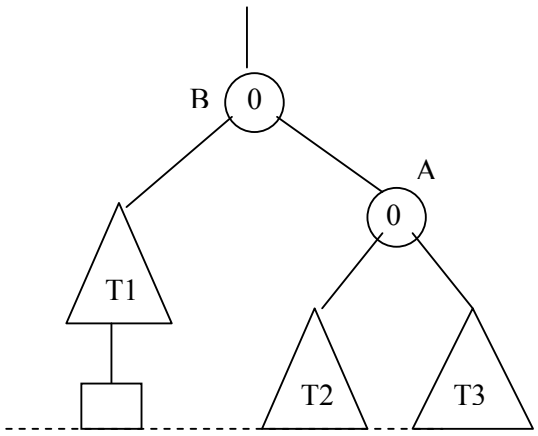


Fig 20.10: Rearranged tree after inserting a new

By observing the tree in the above figure we notice at first that node A is no longer the root of the tree. Now Node B is the root. Secondly, we see that the tree T2 that was the right subtree of B has become the left subtree of A. However, tree T3 is still the right subtree of A. The node A has become the right subtree of B. This tree is balanced with respect to node A and B. The balance of A is 0 as T2 and T3 are at the same level. The level of T1 has increased due to the insertion of new node. It is now at the same level as that of T2 and T3. Thus the balance of B is also 0. The important thing in this modified tree is that the inorder traversal of it is the same as in the previous tree (fig 10.9) and is

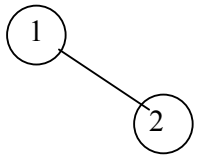
T1 B T2 A T3

We see that the above two trees give us data items in the same order by inorder traversal. So it is not necessary that data items in a tree should be in a particular node at a particular position. This process of tree modification is called rotation.

Example (AVL Tree Building)

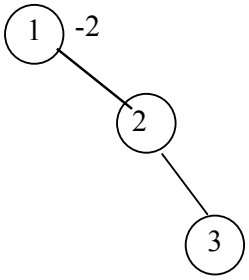
Let's build an AVL tree as an example. We will insert the numbers and take care of the balance of nodes after each insertion. While inserting a node, if the balance of a node becomes greater than 1 (that means tree becomes unbalance), we will rearrange the tree so that it should become balanced again. Let's see this process.

Assume that we have *insert* routine (we will write its code later) that takes a data item as an argument and inserts it as a new node in the tree. Now for the first node, let's say we call *insert (1)*. So there is one node in the tree i.e. 1. Next, we call *insert (2)*. We know that while inserting a new data item in a binary search tree, if the new data item is greater than the existing node, it will go to the right subtree. Otherwise, it will go to the left subtree. In the call to insert method, we are passing 2 to it. This data item i.e. 2 is greater than 1. So it will become the right subtree of 1 as shown below.



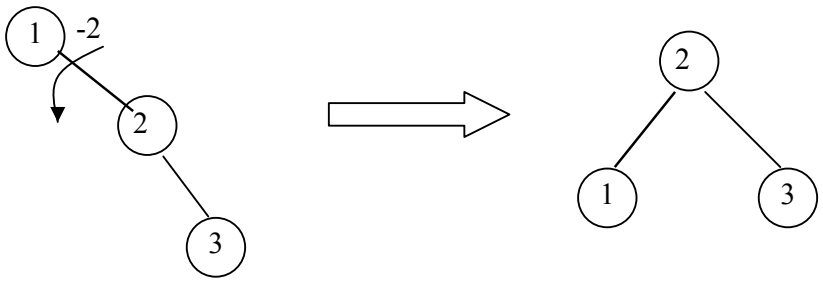
As there are only two nodes in the tree, there is no problem of balance yet.

Now insert the number 3 in the tree by calling *insert (3)*. We compare the number 3 with the root i.e.1. This comparison results that 3 will go to the right subtree of 1. In the right subtree of 1 there becomes 2. The comparison of 3 with it results that 3 will go to the right subtree of 2. There is no subtree of 2, so 3 will become the right subtree of 2. This is shown in the following figure.



Let's see the balance of nodes at this stage. We see that node 1 is at level 0 (as it is the root node). The nodes 2 and 3 are at level 1 and 2 respectively. So with respect to the node 1, the deepest level (height) of its right subtree is 2. As there is no left subtree of node 1 the level of left subtree of 1 is 0. The difference of the heights of left and right subtree of 1 is –2 and that is its balance. So here at node 1, the AVL condition has been violated. We will not insert a new node at this time. First we will do the rotation to make the tree (up to this step) balanced. In the process of inserting nodes, we will do the rotation before inserting next node at the points where the AVL condition is being violated. We have to identify some things for doing rotation. We have to see that on what nodes this rotation will be applied. That means what nodes will be

rearranged. Some times, it is obvious that at what nodes the rotation should be done. But there may situations, when the things will not be that clear. We will see these things with the help of some examples.
In the example under consideration, we apply the rotation at nodes1 and 2. We rotate these nodes to the left and thus the node 1 (along with any tree if were associated with it) becomes down and node 2 gets up. The node 3 (and trees associated with it, here is no tree as it is leaf node) goes one level upward. Now 2 is the root node of the tree and 1 and 3 are its left and right subtrees respectively as shown in the following figure.



Non AVL Tree AVL tree after applying rotation

We see that after the rotation, the tree has become balanced. The figure reflects that the balance of node 1, 2 and 3 is 0. We see that the inorder traversal of the above tree before rotation (tree on left hand side) is 1 2 3. Now if we traverse the tree after rotation (tree on right hand side) by inorder traversal, it is also 1 2 3. With respect to the inorder traversal, both the traversals are same. So we observe that the position of nodes in a tree does not matter as long as the inorder traversal remains the same. We have seen this in the above figure where two different trees give the same inorder traversal. In the same way we can insert more nodes to the tree. After inserting a node we will check the balance of nodes whether it violates the AVL condition. If the tree, after inserting a node, becomes unbalance then we will apply rotation to make it balance. In this way we can build a tree of any number of nodes.

Data Structures

Lecture No. 21

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.4, 4.4.1

Summary

- AVL Tree Building Example
- Cases for Rotation

AVL Tree Building Example

This lecture is a sequel of the previous one in which we had briefly discussed about building an AVL tree. We had inserted three elements in the tree before coming to the end of the lecture. The discussion on the same example will continue in this lecture. Let's see the tree's figures below:

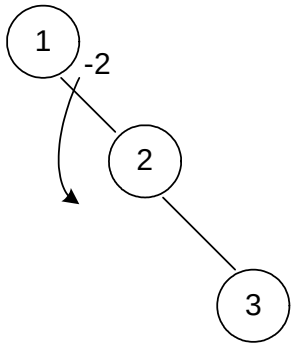


Fig 21.1: insert(3) single left rotation

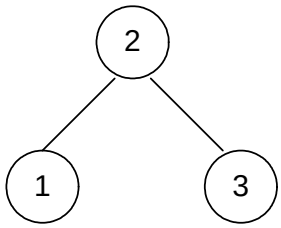


Fig 21.2: insert(3)

Node containing number *2* became the root node after the rotation of the node having number *1*. Note the direction of rotation here.

Let's insert few more nodes in the tree. We will build an AVL tree and rotate the node when required to fulfill the conditions of an AVL tree.

To insert a node containing number *4,*we will, at first, compare the number inside the

root node. The current *root* node is containing number *2*. As *4* is greater than *2*, it will take the right side of the root. In the right subtree of the *root, there* is the node containing number *3*. As 4 is also greater than *3*, it will become the right child of the node containing number *3*.

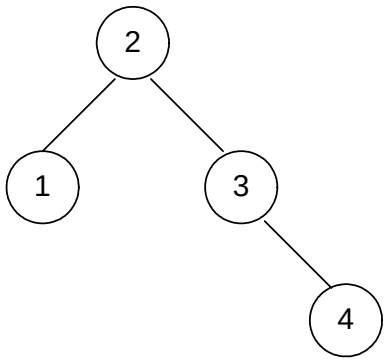


Fig 21.3: insert(4)

Once we insert a node in the tree, it is necessary to check its balance to see whether it is within AVL defined balance. If it is not so, then we have to rotate a node. The balance factor of the node containing number *4* is zero due to the absence of any left or right subtrees. Now, we see the *balance factor* of the node containing number *3*. As it has no left child, but only right subtree, the *balance factor* is *–1*. The *balance factor* of the node containing number *1* is *0*. For the node containing number *2*, the height of the left subtree is *1* while that of the right subtree is *2.* Therefore, the *balance factor* of the node containing number *2* is *1 – 2 = -1*. So every node in the tree in *fig. 21.3* has *balance factor* either *1* or less than that. You must be remembering that the condition for a tree to be an AVL tree, every node's balance needs not to be zero necessarily. Rather, the tree will be called AVL tree, if the *balance factor* of each node in a tree is *0*, *1* or *–1*. By the way, if the *balance factor* of each node inside the tree is *0,* it will be a perfectly balanced tree.

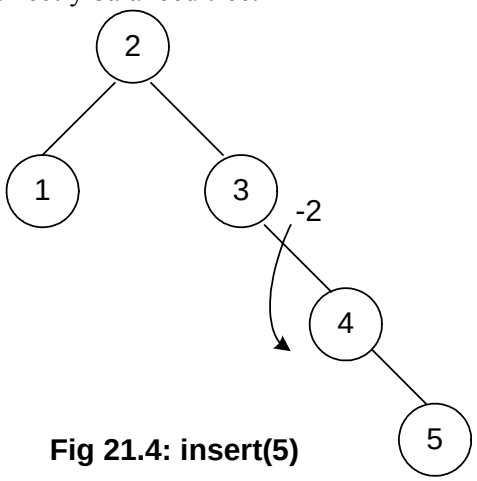


Fig 21.4: insert(5)

Next, we insert a node containing number *5* and see the *balance factor* of each node. The *balance factor* for the node containing *5* is *0*. The *balance factor* for node containing *4* is *–1* and for the node containing *3* is *-2*. The condition for AVL is not satisfied here for the node containing number *3,* as its *balance factor* is *–2*. The rotation operation will be performed here as with the help of an arrow as shown in the above *Fig 21.4*. After rotating the node *3*, the new tree will be as under:

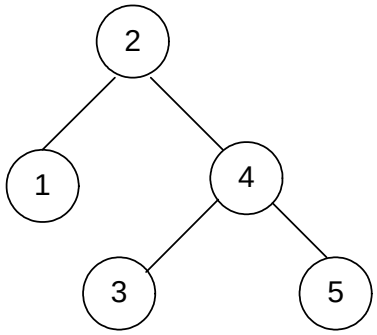


Fig 21.5: insert(5)

You see in the above figure that the node containing number *4* has become the right child of the node containing number *2*. The node with number *3* has been rotated. It has become the left child of the node containing number *4*. Now, the *balance factor* for different nodes containing numbers *5*, *3* and *4* is *0*. To get the *balance factor* for the node containing number *2*, we see that the height of the left subtree containing number *2* is *1* while height of the right subtree is *2*. So the *balance factor* of the node containing number *2* is *–1*. We saw that all the nodes in the tree above in *Fig 21.5* fulfill the AVL tree condition.

If we traverse the tree *Fig 21.5*, in inorder tree traversal, we get:

1 2 3 4 5

Similarly, if we traverse the tree in inorder given in *Fig 21.4* (the tree before we had rotated the node containing number *3*), following will be the output.

1 2 3 4 5

In both the cases above, before and after rotation, we saw that the inorder traversal of trees gives the same result. Also the *root* (node containing number *2*) remained the same.

See the *Fig 21.4* above. Considering the inorder traversal, we could arrange the tree in such a manner that node *3* becomes the *root* of the tree, node *2* as the left child of node *3* and node *1* as the left child of the node *2*. The output after traversing the changed tree in inorder will still produce the same result:

1 2 3 4 5

While building an AVL tree, we rotate a node immediately after finding that that the node is going out of balance. This ensures that tree does not become shallow and remains within the defined limit for an AVL tree.

Let's insert another element *6* in the tree. The figure of the tree becomes:

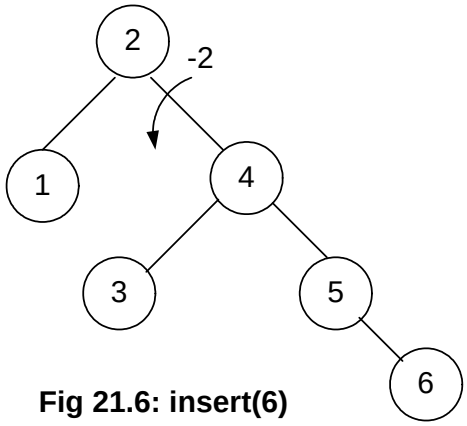


Fig 21.6: insert(6)

The newly inserted node *6* becomes the right child of the node *5*. Usually, after the insertion of a node, we will find out the node factor for each node and rotate it immediately. This is carried out after finding the difference out of limit. The *balance factor* for the node *6* is *0*, for node *5* is *–1* and 0 for node *3*. Node *4* has *–1 balance factor* and node *1* has *0*. Finally, we check the *balance factor* of the *root* node, node *2*, the left subtree's height is *1* and the right subtree's height is *3*. Therefore, the *balance factor* for node *2* is *–2,* which necessitates the rotation of the root node *2*. Have a look on the following figure to see how we have rotated the node *2*.

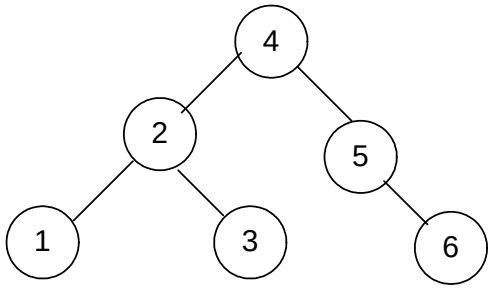


Fig 21.7: insert(6)

Now the node *4* has become the *root* of the tree. Node *2*, which was the *root* node, has become the left child of node *4*. Nodes *5* and *6* are still on their earlier places while remaining the right child and sub-child of node *4* respectively. However, the node *3*, which was left child of node *4*, has become the right child of node *2*.
Now, let's see the inorder traversal of this tree:

1 2 3 4 5 6

You are required to practice this inorder traversal. It is very important and the basic point of performing the rotation operation is to preserve the inorder traversal of the tree. There is another point to note here that in Binary Search Tree (BST), the *root* node remains the same (the node that is inserted first). But in an AVL tree, the *root* node keeps on changing.

In *Fig 21.6*: we had to traverse three links (node *2* to node *4* and then node *5*) to reach the node *6*. While after rotation, (in *Fig 21.7)*, we have to traverse the two links (node

4 and *5*) to reach the node *6*. You can prove it mathematically that inside an AVL tree built of n items; you can search up to $1.44\log_2 n$ levels to find a node inside. After this maximum number of links traversal, a programmer will have success or failure, as $1.44\log_2 n$ is the maximum height of the AVL tree. Consider the BST case, where we had constructed a linked list. If we want to build a BST of these six numbers, a linked list structure is formed. In order to reach the node *6*, we will have to traverse five links. In case of AVL tree, we had to traverse two links only.

Let's add few more items in the AVL tree and see the rotations performed to maintain AVL characteristics of the tree.

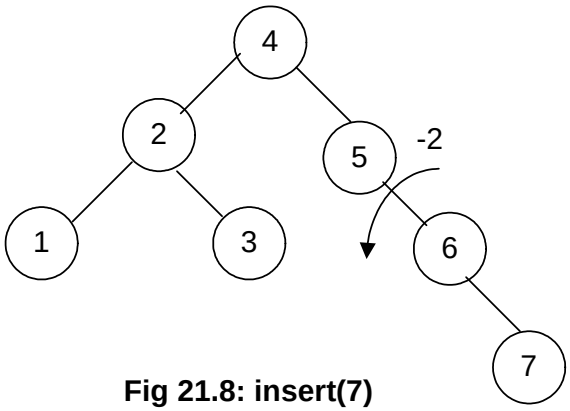


Fig 21.8: insert(7)

Node *7* is inserted as the right child of node *6*. We start to see the balance factors of the nodes. The balance factors for node *7*, *6* are *0* and *–1* respectively. As the balance factor for node *5* is *–2*, the rotation will be performed on this node. After rotation, we get the tree as shown in the following figure.

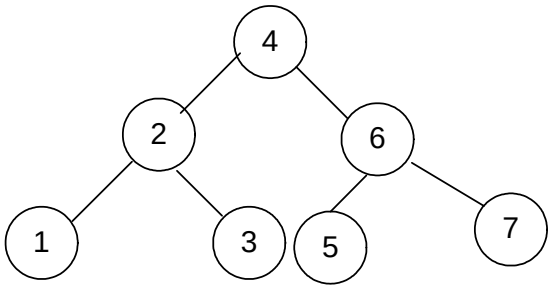


Fig 21.9: insert(7)

After the rotation, node *5* has become the left child of node *6*. We can see in the *Fig 21.9* that the tree has become the perfect binary tree. While writing our program, we will have to compute the balance factors of each node to know that the tree is a perfectly balanced binary tree. We find that balance factor for all nodes *7*, *5*, *3*, *1*, *6*, *2* and *4* is *0. Therefore,* we know that the tree is a perfect balanced tree. Let' see the inorder traversal output here:

1 2 3 4 5 6 7

It is still in the same sequence and the number 7 has been added at the end.

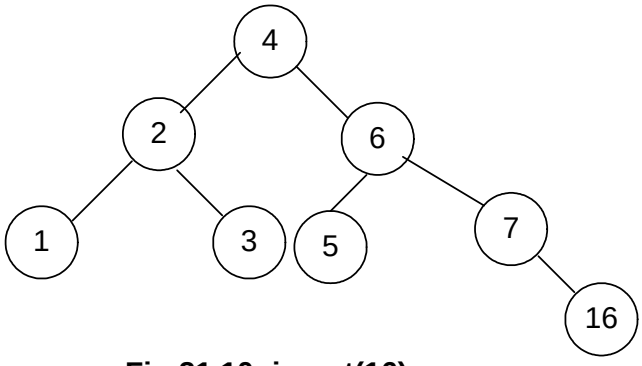


Fig 21.10: insert(16)

We have inserted a new node *16* in the tree as shown in the above *Fig 21.10*. This node has been added as the right child of the node *7*. Now, let's compute the balance factors for the nodes. The balance factor for nodes *16*, *7, 5, 3, 1, 6, 2 and 4* is either *0* or *–1*. So this fulfills the condition of a tree to be an AVL. Let's insert another node containing number *15* in this tree. The tree becomes as given in the figure below:

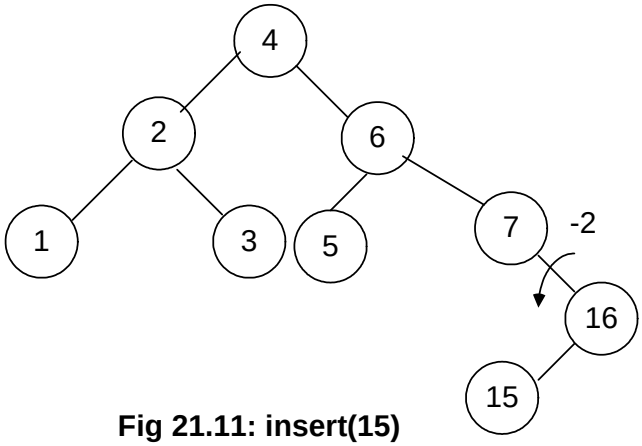


Fig 21.11: insert(15)

Next step is to find out the balance factor of each node. The factors for nodes *5* and *16* are *0* and *1* respectively. This is within limits of an AVL tree but the balance factor for node *7* is *–2*. As this is out of the limits of AVL, we will perform the rotation operation here. In the above diagram, you see the direction of rotation. After rotation, we have the following tree:

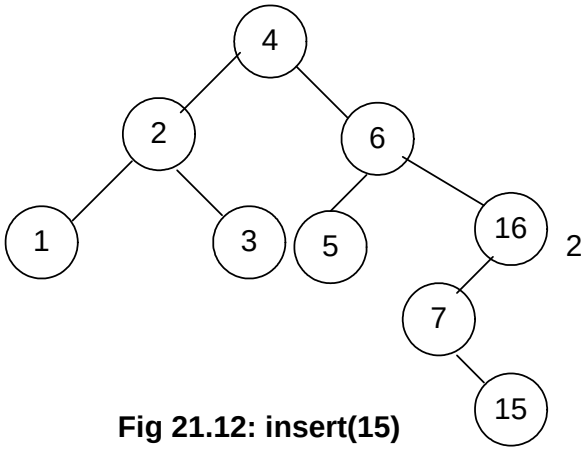


Fig 21.12: insert(15)

Node *7* has become the left child of node *16* while node *15* has attained the form of the right child of node *7*. Now the balance factors for node *15*, *7* and *16* are *0*, *-1* and *2* respectively. Note that the single rotation above when we rotated node 7 is not enough as our tree is still not an AVL one. This is a complex case that we had not encountered before in this example.

Cases of Rotation

The single rotation does not seem to restore the balance. We will re-visit the tree and rotations to identify the problem area. We will call the node that is to be rotated as α (node requires to be re-balanced). Since any node has at the most two children, and a height imbalance requires that α's two sub-trees differ by two (or –2), the violation will occur in four cases:

1. An insertion into left subtree of the left child of α.
2. An insertion into right subtree of the left child of α.
3. An insertion into left subtree of the right child of α.
4. An insertion into right subtree of the right child of α.

The insertion occurs on the *outside* (i.e., left-left or right-right) in *cases 1* and *4*. Single rotation can fix the balance in *cases 1* and *4*.

Insertion occurs on the *inside* in *cases 2* and *3* which a single rotation cannot fix.

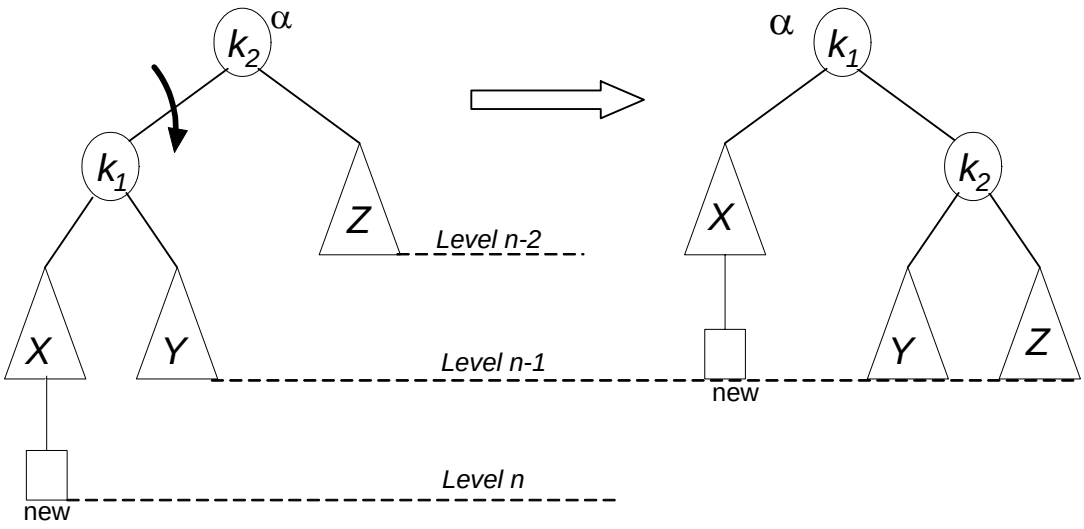


Fig 21.13: Single right rotation to fix case 1

We have shown, single right notation to fix case 1. Two nodes *k2* and *k1* are shown in the figure, here *k2* is the *root* node (and also the α node, *k1* is its left child and *Z* shown in the triangle is its right child. The nodes *X* and *Y* are the left and right subtrees of the node *k1*. A new node is also shown below to the triangle of the node *X*, the exact position (whether this node will be right or left child of the node *X*) is not mentioned here. As the new node is inserted as a child of *X* that is why it is called an *outside* insertion, the insertion is called *inside* if the new node is inserted as a child of the node *Y*. This insertion falls in *case 1* mentioned above, so by our definition above, single rotation should fix the balance. The *k2* node has been rotated single time towards right to become the right child of *k1* and *Y* has become the left child of *k2*. If we traverse the tree in inorder fashion, we will see that the output is same:

X k1 Y k2 Z

Consider the the figure below:

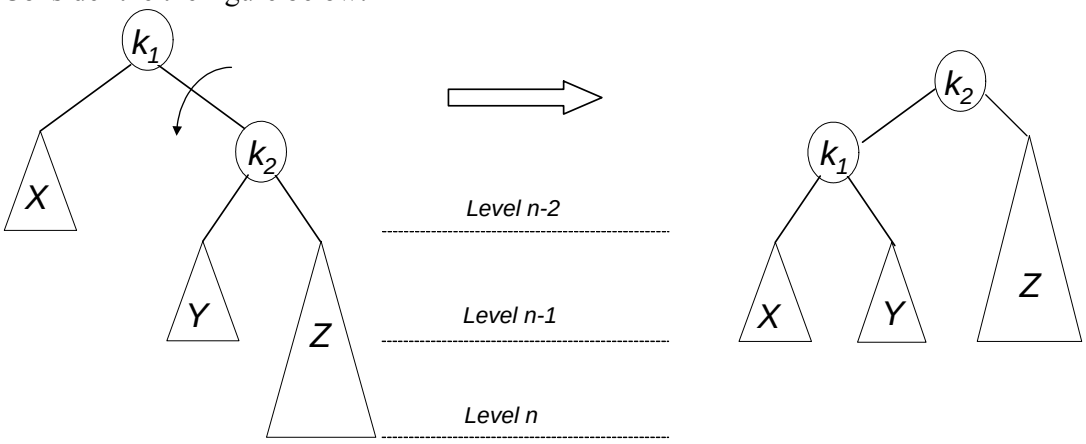


Fig 21.14: Single left rotation to fix case 4

In this figure (*Fig 21.14*), the new node has been inserted as a child node of Z, that is why it is shown in bigger size covering the next level. Now this is an example of *case 4* because the new node is inserted below the right subtree of the right child of the

root node (α). One rotation towards should make it balanced within limits of AVL tree. The figure on the right is after rotation the node *k1* one time towards left. This time node *Y* has become the right child node of the node *k1*.
In our function of insertion in our code, we will do insertion, will compute the balance factors for nodes and make rotations.
Now, we will see the *cases 2 and 3*, these are not resolved by a single rotation.

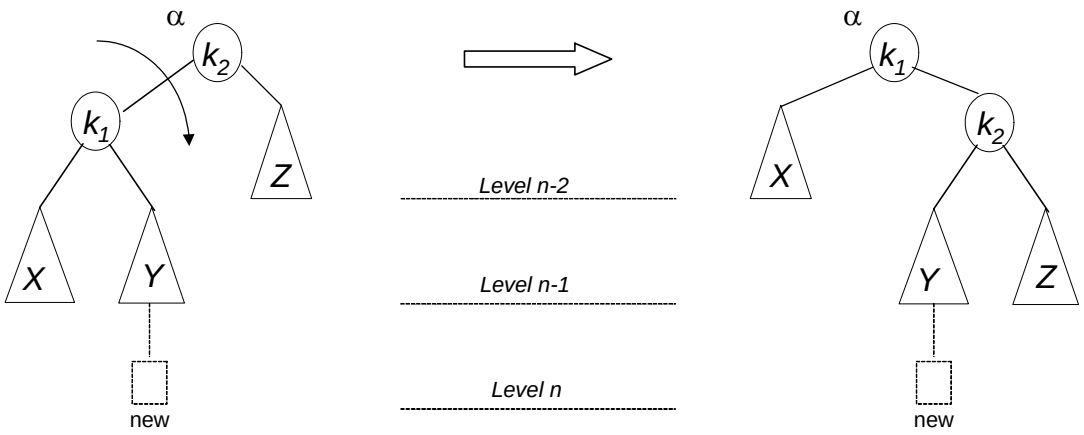


Fig 21.15: Single right rotation fails to fix case 2

We see here that the new node is inserted below the node *Y*. This is an *inside* insertion. The balance factor for the node *k2* became *2*. We make single rotation by making right rotation on the node *k2* as shown in the figure on the right. We compute the balance factor for *k1*, which is –2. So the tree is still not within the limits of AVL tree. Primarily the reason for this failure is the node *Y* subtree, which is unchanged even after making one rotation. It changes its parent node but its subtree remains intact. We will cover the double rotation in the next lecture.

It is very important that you study the examples given in your text book and try to practice the concepts rigorously.

Data Structures

Lecture No. 22

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.4.2

Summary

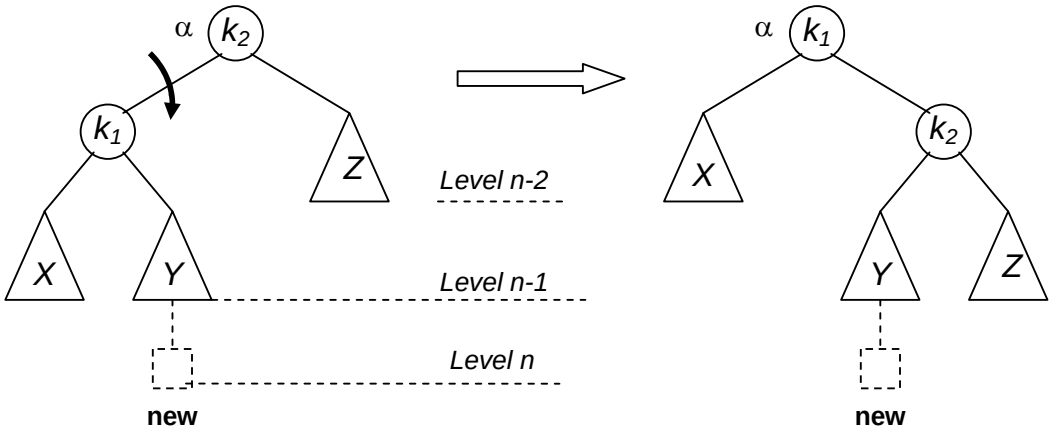
- Cases of rotations
- Left-right double rotation to fix case 2
- Right-left double rotation to fix case 3
- C++ Code for avlInsert method

Cases of rotations

In the previous lecture, we discussed how to make insertions in the AVL tree. It was seen that due to the insertion of a node, the tree has become unbalanced. Resultantly, it was difficult to fix it with the single rotation. We have analyzed the insertion method again and talked about the α node. The new node will be inserted at the left or right subtree of the α's left child or at the left or right subtree of the α's right child. Now the question arises whether the single rotation help us in balancing the tree or not. If the new node is inserted in the left subtree of the α's left child or in the right subtree of α's right child, the balance will be restored through single rotation. However, if the new node goes inside the tree, the single rotation is not going to be successful in balancing the tree.

We face four scenarios in this case. We said that in the case-1 and case-4, single rotation is successful while in the case-2 and case-3 single rotation does not work. Let's see the tree in the diagram given below.

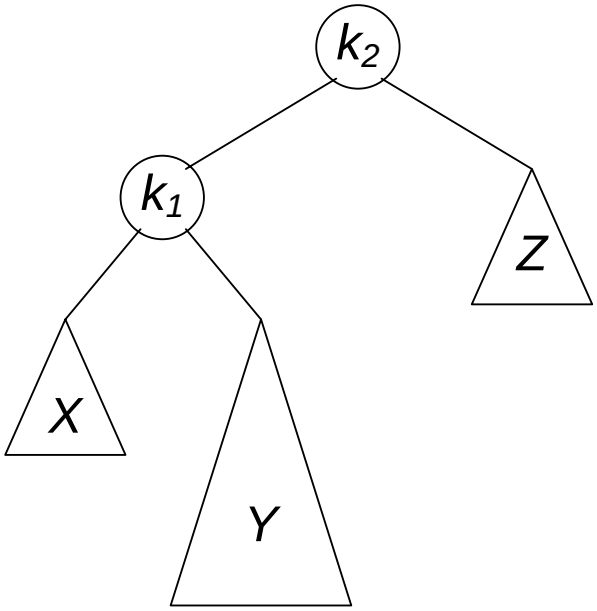
Single right rotation fails to fix case 2.



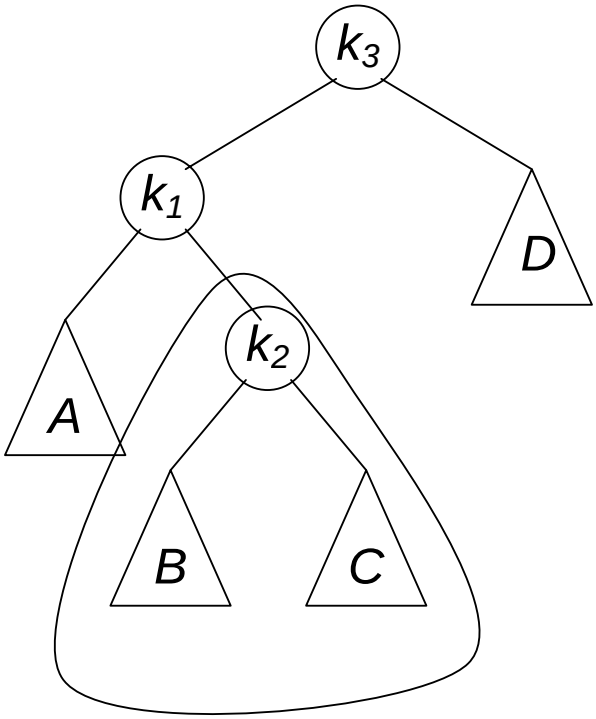
In the above tree, we have α node as *k2*, which has a left child as *k1*. Whereas *X* and *Y* are its left and right children. The node *k2* has a right child *Z*. Here the newly inserted

node works as the left or right child of node *Y*. Due to this insertion, one level is increased in the tree. We have applied single rotation on the link of *k1* and *k2*. The right side tree in the figure is the post-rotation tree. The node *k1* is now at the top while *k2* comes down and node *Y* changes its position. Now if you see the levels of the node, these are seen same. Have a look on the level of the α node i.e. *k1* which reflects that the difference between the left and right side levels is still 2. So the single rotation does not work here.

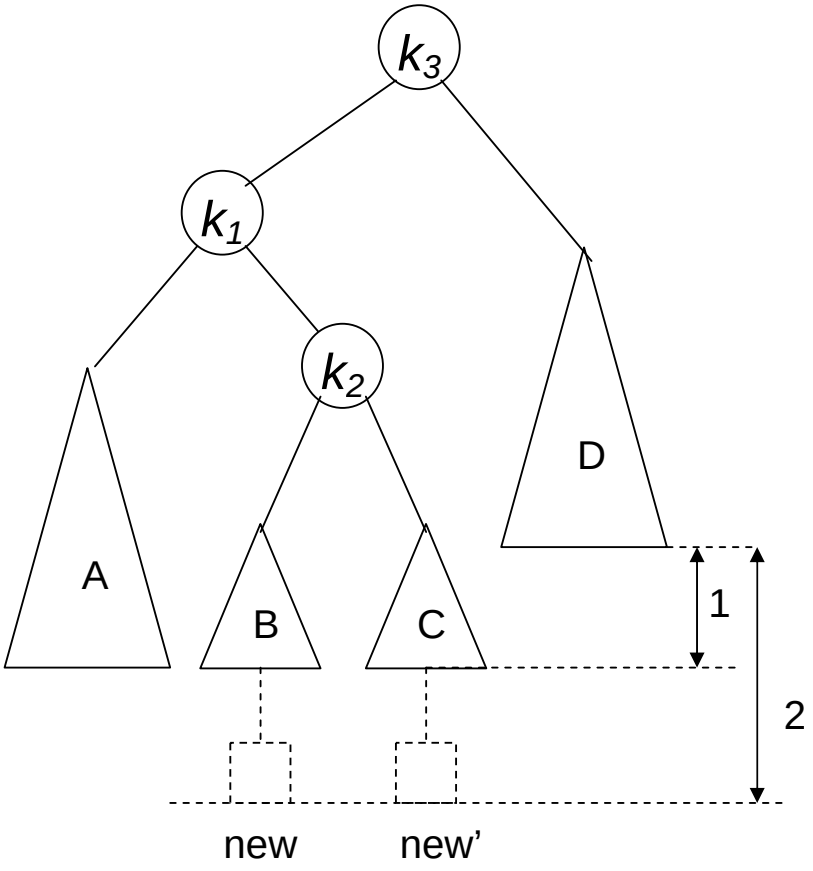
Let's see how we can fix that problem. A fresh look on the following diagram will help us understand the problem.



Here *k2* is the root node while *k1* and *Z* are the right and left children respectively. The new node is inserted under *Y* so we have shown *Y* in a big triangle. The new node is inserted in the right subtree of *k1*, increasing its level by 1. *Y* is not empty as the new node was inserted in it. If *Y* is empty, the new node will be inserted under *k1*. It means that *Y* has a shape of a tree having a root and possibly left and right subtrees. Now view the entire tree with four subtrees connected with 3 nodes. See the diagram below.



We have expanded the *Y* and shown the root of *Y* as *K2, B* and *C* are its left and right subtrees. We have also changed the notations of other nodes. Here, we have *A*, *B*, *C* and *D* as subtrees and *k1*, *k2* and *k3* as the nodes. Let's see where the new node is inserted in this expanded tree and how can we restore its balance. Either tree *B* or *C* is two levels deeper than *D*. But we are not sure which one is deeper. The value of new node will be compared with the data in *k2* that will decide that this new node should be inserted in the right subtree or left subtree of the *k2*. If the value in the new node is greater than *k2*, it will be inserted in the right subtree i.e. *C*. If the value in the new node is smaller than *k2*, it will be inserted in the left subtree i.e. *B*. See the diagram given below:



New node inserted at either of the two spots

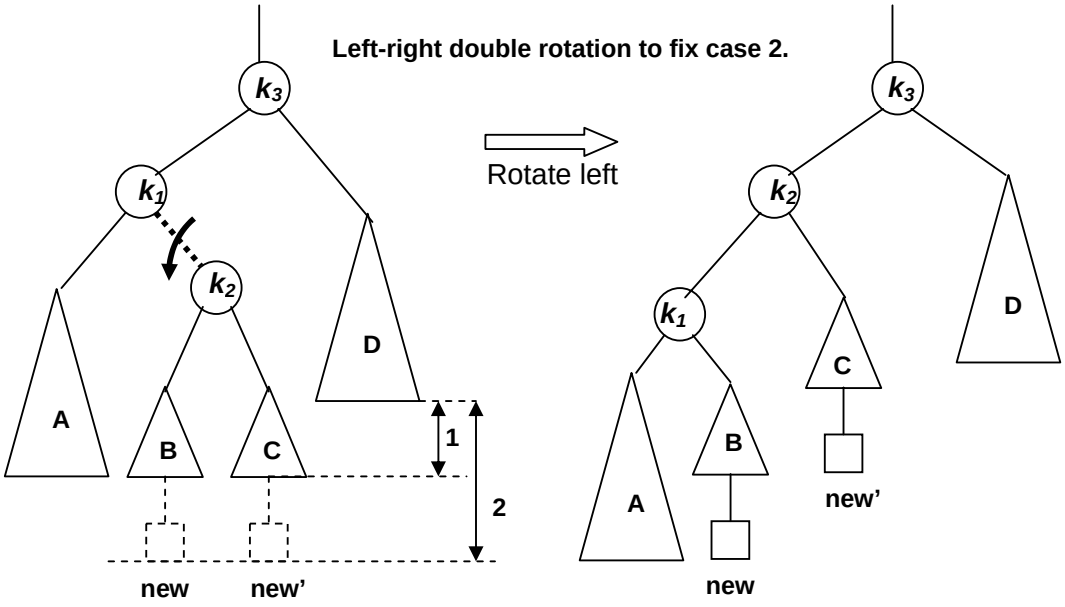
We have seen the both possible locations of the new node in the above diagram. Let's see the difference of levels of the right and left subtrees of the *k3*. The difference of *B* or *C* from *D* is 2. Therefore the expansion of either of *B* or *C*, due to the insertion of the new node, will lead to a difference of 2. Therefore, it does not matter whether the new node is inserted in *B* or *C*. In both of the cases, the difference becomes 2. Then we try to balance the tree with the help of single rotation. Here the single rotation does not work and the tree remains unbalanced. To re-balance it, *k3* cannot be left as the root. Now the question arises if *k3* cannot become root, then which node will become root? In the single rotation, *k1* and *k3* were involved. So either *k3* or *k1* will come down. We have two options i.e. left rotation or right rotation. If we turn *k1* into a root, the tree will be still unbalanced. The only alternative is to place *k2* as the new root. So we have to make *k2* as root to balance the tree. How can we do that?

If we make *k2* the root, it forces *k1* to be *k2*'s left child and *k3* to be its right child. When we carry out these changes, the condition is followed by the inorder traversal. Let's see the above tree in the diagram. In that diagram, the *k3* is the root and *k1* is its left child while *k2* is the right child of *k1*. Here, we have *A, B, C* and *D* as subtrees. You should know the inorder traversal of this tree. It will be *A, k1, B, k2, C, k3* and *D* where *A, B, C* and *D* means the complete inorder traversal of these subtrees. You should memorize this tree traversal.

Now we have to take *k2* as the root of this tree and rearrange the subtrees *A, B, C* and *D*. *k1* will be the left child of *k2* while *k3* is going to be its right child. Finally if we traverse this new tree, it will be the same as seen above. If it is not same, it will mean that there is something wrong in the rotation. We have to find some other solution. Now let's see how we can rotate our tree so that we get the desired results.

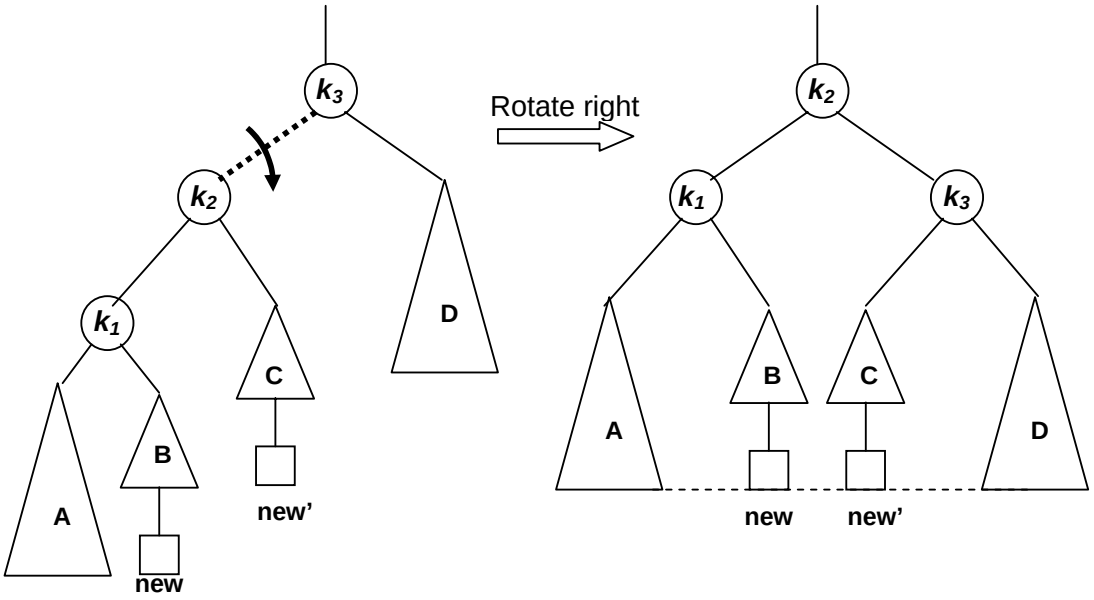
Left-right double rotation to fix case 2

We have to perform a double rotation to achieve our desired results. Let's see the diagram below:



On the left side, we have the same tree with *k3* as its root. We have also shown the new nodes as *new* and *new'* i.e. the new node will be attached to *B* or *C*. At first, we will carry out the left rotation between *k1* and *k2*. During the process of left rotation, the root *k1* comes down and *k2* goes up. Afterwards, *k1* will become the left child of *k2* and the left subtree of *k2* i.e. *B*, will become the right subtree of *k1*. This is the single rotation. You can see the new rotated tree in the above figure. It also shows that the *B* has become the right child of the *k1*. Moreover, the new node is seen with the *B*. Now perform the inorder traversal of this new rotated tree. It is *A, k1, B, k2, C, k3* and *D*. It is same as witnessed in case of the inorder traversal of original tree. With this single rotation, the *k2* has gone one step up while *k1* has come down. Now *k2* has become the left child of *k3*. We are trying to make the *k2* the root of this tree. Now what rotation should we perform to achieve this?

Now we will perform right rotation to make the *k2* the root of the tree. As a result, *k1* and *k2* have become its left and right children respectively. The new node can be inserted with *B* or *C*. The new tree is shown in the figure below:



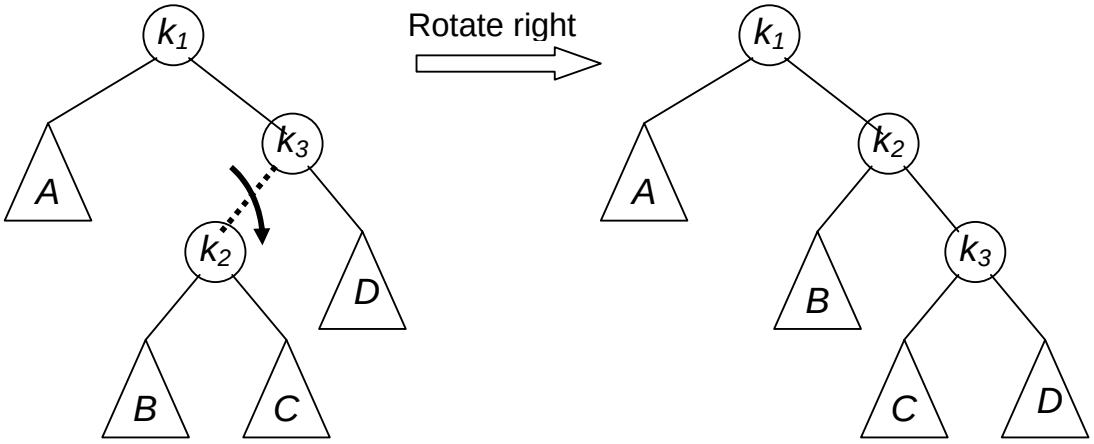
Now let's see the levels of *new* and *new'*. Of these, one is the new node. Here you can see that the levels of *new*, *new'* i.e. *A* and *D* are the same. The new tree is now a balanced one. Let's check the inorder traversal of this tree. It should be the same as that of the original tree. The inorder traversal of new tree is *A, k1, B, k2, C, k3* and *D*, which is same as that of the original tree.

This is known as double rotation. In double rotation, we perform two single rotations. As a result, the balance is restored and the AVL condition is again fulfilled. Now we will see in which order, the double rotation is performed? We performed a left rotation between *k1* and *k2* link, followed by a right rotation.

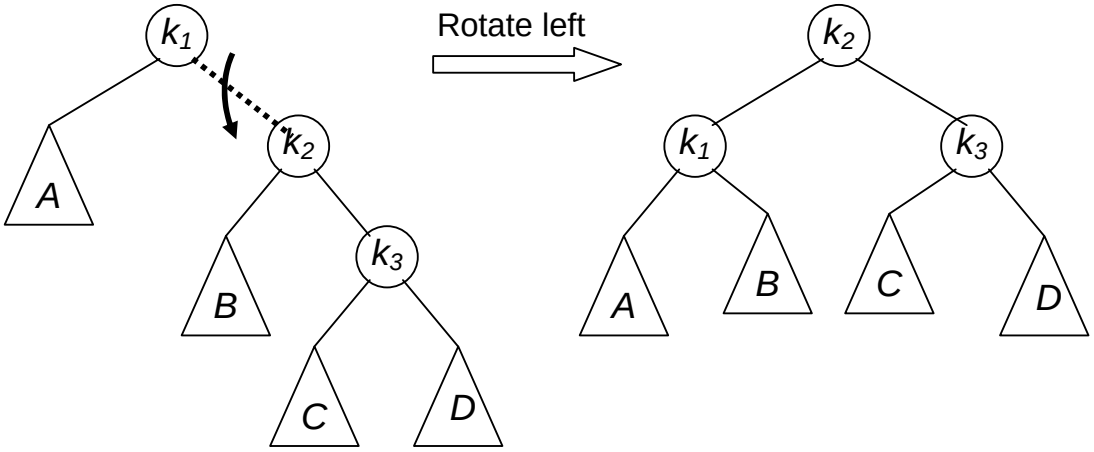
Right-left double rotation to fix case 3

In case, the node is inserted in left subtree of the right child, we encounter the same situation as discussed above. But here, we will perform right rotation at first before going for a left rotation. Let's discuss this symmetric case and see how we can apply double rotation here. First we perform the right rotation.

Right-left double rotation to fix case 3.



Here *k1* is the root of the tree while *k3* is the right child of the *k1*. *k2* is the inner child. It is the *Y* tree expanded again here and the new node will be inserted in the *k2*'s right subtree *C* or left subtree *B*. As we have to transform the *k2* into the root of the tree, so the right rotation between the link *k2* and *k3* will be carried out. As a result of this rotation, *k2* will come up and *k3* will go down. The subtree *B* has gone up with the *k2* while subtree *C* is now attached with the *k3*. To make the *k2* root of the tree, we will perform the left rotation between then *k1* and *k2*. Let's see this rotation in the figure below:

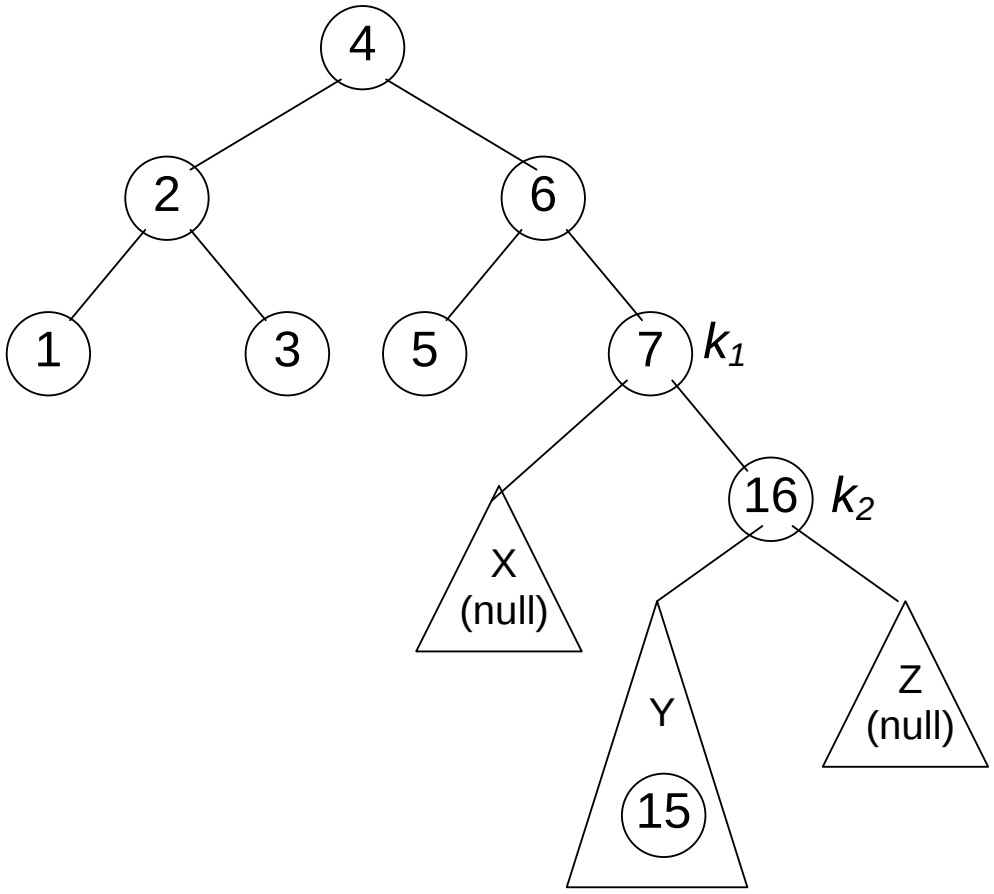


In the above figure at the right side, we have the final shape of the tree. You can see that *k2* has become the root of the tree. *k1* and *k3* are its left and right children respectively. While performing the inorder traversal, you will see that we have preserved our inorder traversal.

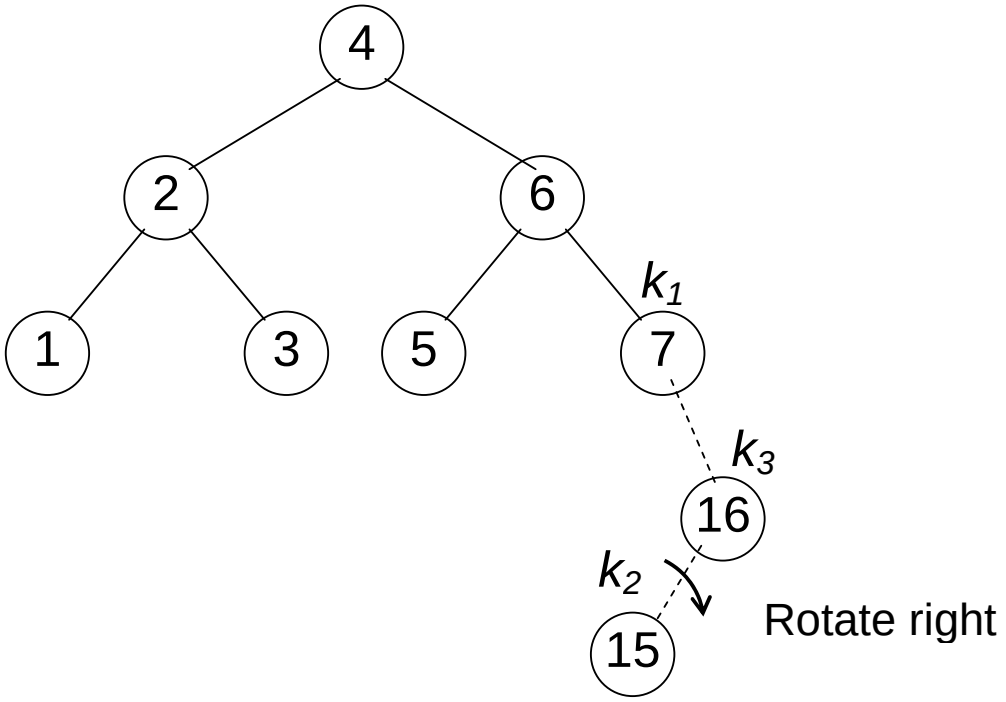
We have started this activity while building an example tree. We inserted numbers in it. When the balance factor becomes more than one, rotation is performed. During this process, we came at a point when single rotation failed to balance the tree. Now there is need to perform double rotation to balance the tree that is actually two single rotations. Do not take double rotation as some complex function, it is simply two single rotations in a special order. This order depends on the final position of the new

node. Either the new node is inserted at the right subtree of the left child of α node or at the left subtree of the right child of α node. In first case, we have to perform left-right rotation while in the second case, the right-left rotation will be carried out.

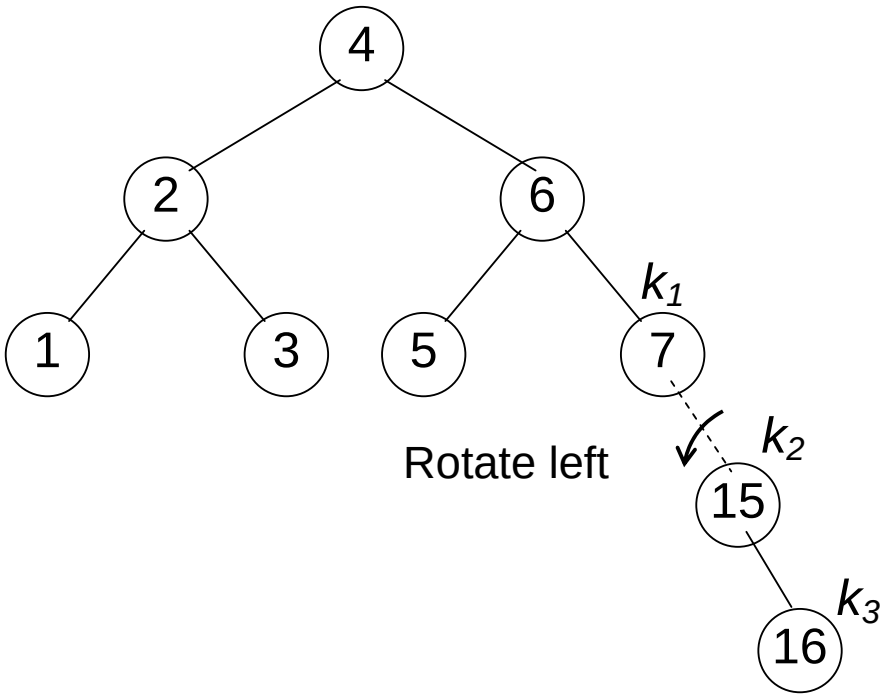
Let's go back to our example and try to complete it. So far, we have 1, 2, 3, 4, 5, 6, 7 and 16 in the tree and inserted 15 which becomes the left child of the node 16. See the figure below:



Here we have shown *X*, *Y* and *Z* in case of the double rotation. We have shown *Y* expanded and 15 is inside it. Here we will perform the double rotation, beginning with the right rotation first.

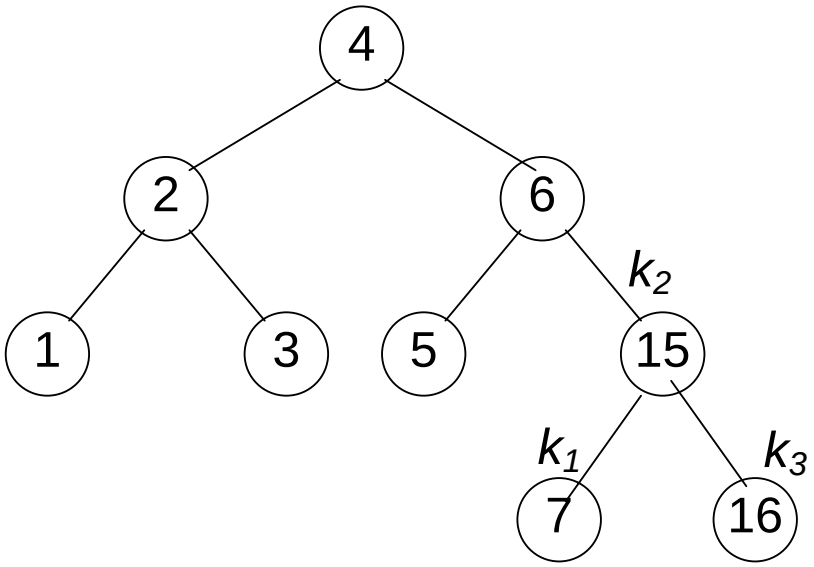


We have identified the *k1, k2* and *k3* nodes. This is the case where we have to perform right-left double rotation. Here we want to promote *k2* upwards. For this purpose, the right rotation on the link of *k2* and *k3* i.e. 15 and 16 will be carried out.



The node 15 now comes up while node 16 has gone down. We have to promote *k2* to the top and *k3* and *k1* will become its right and left children respectively. Now we will

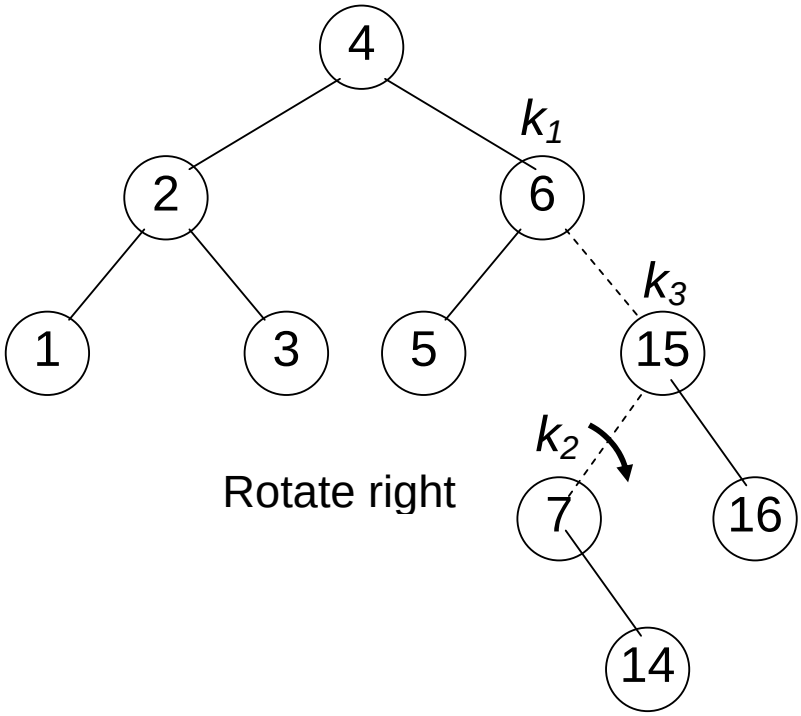
perform left rotation on the link of *k1* and *k2* i.e. 7 and 15. With this left rotation, 15 goes up and 7 and 16 become its left and right children respectively.



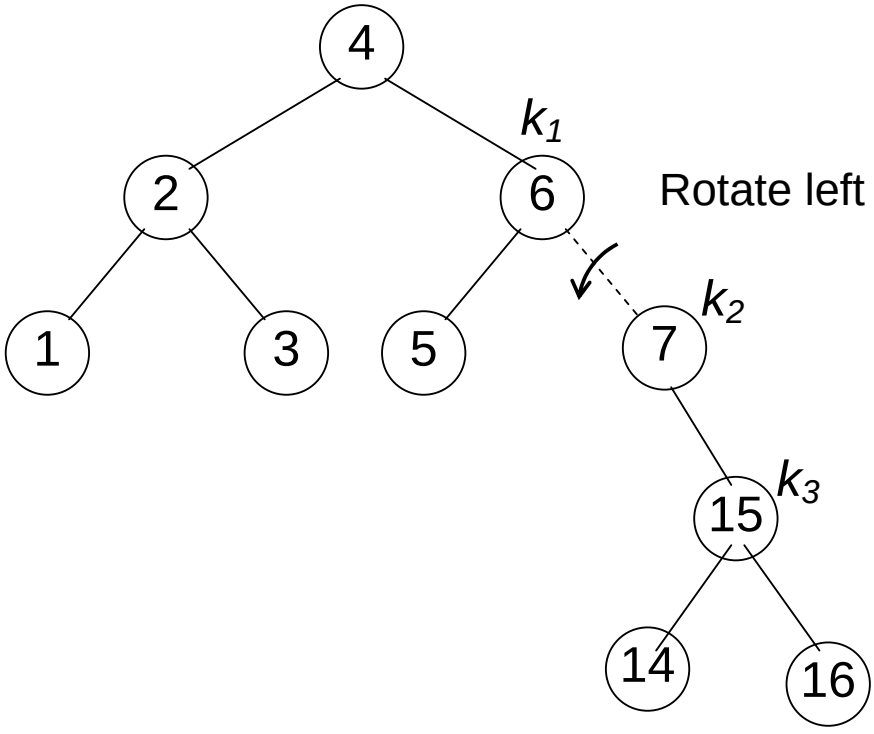
Here we have to check two things. At first, the tree is balanced or not i.e. the AVL condition is fulfilled or not. Secondly we will confirm that the inorder traversal is preserved or not. The inorder traversal should be the same as that of the inorder traversal of original tree. Let's check these two conditions. The depth of the left subtree of node 4 is 2 while the depth of the right subtree of node 4 is three. Therefore, the difference of the levels at node 4 is one. So the AVL condition is fulfilled at node 4. At node 6, we have one level on it left side while at the right side of node 6, there are two levels. As the difference of levels is one, therefore node 6 is also balanced according to the AVL condition. Similarly other nodes are also fulfilling the AVL condition. If you see the figure above, it is clear that the tree is balanced.

We are doing all this to avoid the link list structure. Whenever we perform rotation on the tree, it becomes clear from the figure that it is balanced. If the tree is balanced, in case of searching, we will not have to go very deep in the tree. After going through the mathematical analysis, you will see that in the worst case scenario, the height of the tree is $1.44 \log_2 n$. This means that the searching in AVL is logarithmic. Therefore if there are ten million nodes in an AVL tree, its levels will be roughly as $\log_2(10 \text{ million})$ which is very few. So the traversal in an AVL tree is very simple.

Let's insert some more nodes in our example tree. We will perform single and double rotations, needed to make the tree balanced. The next number to be inserted is 14. The position of node 14, according to the inorder traversal, is the right child of 7. Let's see this in the diagram as:

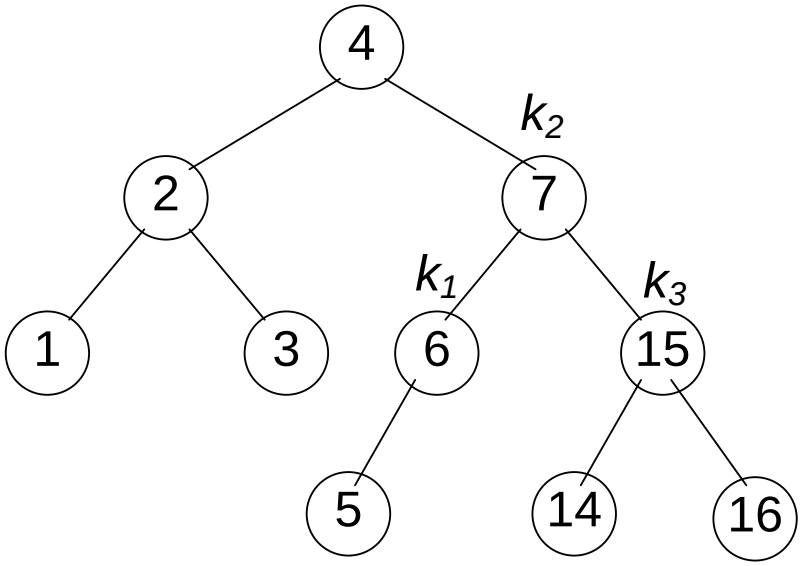


The new node 14 is inserted as the right child of 7 that is the inner subtree of 15. Here we have to perform double rotation again. We have identified the *k1*, *k2* and *k3*. *k2* has to become the root of this subtree. The nodes *k1* and *k3* will come down with their subtrees while *k2* is going to become the root of this subtree. After the right rotation the tree will be as:



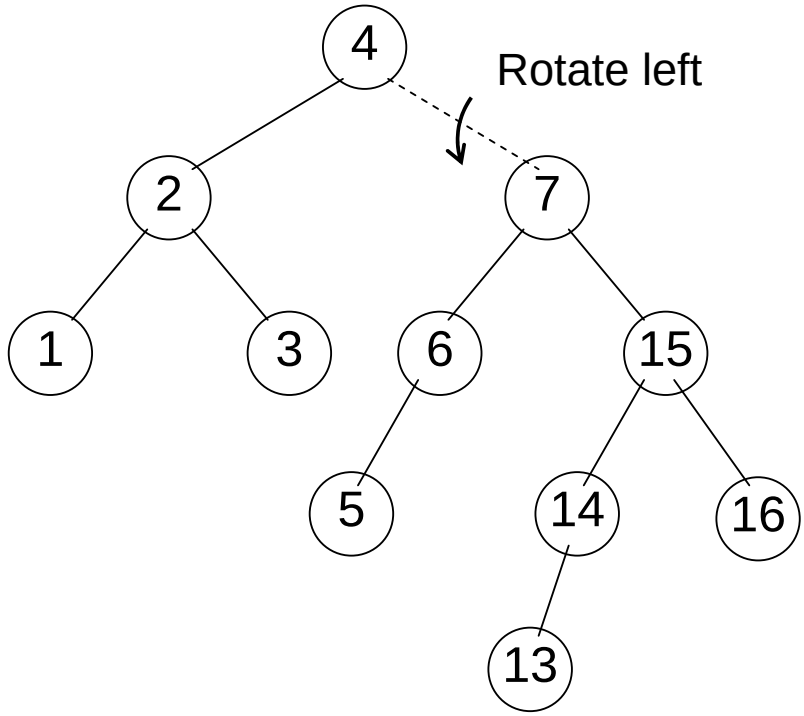
With the right rotation, *k2* has come one step up while *k3* has been turned into the right child of *k2* but *k1* is still up. Now we will perform a left rotation on the link of *k1* and *k2* to make the *k2* root of this subtree. Now think that after this rotation and rearrangement of node what will be the shape of the tree.

After the double rotation, the final shape of the tree will be as:

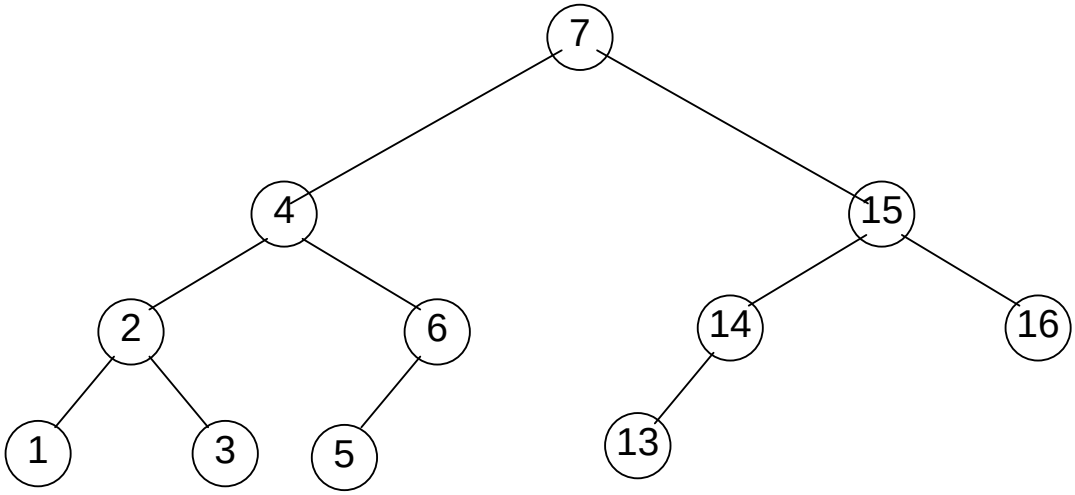


k2 has become the root of the subtree. *k1* has attained the role of the left child of *k2* and *k3* has become the right child of the *k2*. The other nodes 5, 14 and 16 have been rearranged according to the inorder traversal. The node 7 has come up where as node 6 and 15 have become its left and right child. Now just by viewing the above figure, it is clear that the tree is balanced according to the AVL condition. Also if we find out its inorder traversal, it should be the same as the inorder traversal of original tree. The inorder traversal of the above tree is 1, 2, 3, 4, 5, 6, 7, 14, 15, and 16. This is in sorted order so with the rotations the inorder traversal is preserved.

Let's insert some more numbers in our tree. The next number to be inserted is 13.



We have to perform single rotation here and rearrange the tree. It will look like as:



The node 7 has become the root of the tree. The nodes 4, 2, 1, 3, 6, 5 have gone to its left side while the nodes 15, 14, 13, 16 are on its right side. Now try to memorize the tree which we build with these sorted numbers. If you remember that it looks like a link list. The root of that tree was 1. After that we have its right child as 2, the right child of 2 as 3, then its right child 4 and so on up to 16. The shape of that tree looks exactly like a linked list. Compare that with this tree. This tree is a balanced one. Now if we have to traverse this tree for search purposes, we have to go at the most three levels.

Now you must be clear why we need to balance the trees especially if we have to use the balanced search trees. While dealing with this AVL condition, it does not matter

whether the data, provided to a programmer is sorted or unsorted. The data may be sorted in ascending order or descending order. It may contain alphabets or names in any order. We will keep our tree balanced during the insertion and tree will be balanced at each point. Our tree will not be balanced in the end, it will be balanced with each insertion. It will not be completely balanced. At the maximum, if we pick any node, the difference in the levels of its right subtree and left subtree will not be more than 1.

Now if we have 9 or 10 nodes in the tree and take *log* of this, it will be near 3. Therefore our tree has three levels after that there are its leaf nodes. Please keep this in mind that originally we have thought BST as an abstract data type. We have defined operations on it like *insert*, *remove* and the major method was *find*. The *find* method takes a data item and searches the tree. It will also show that this data item exists or not in the tree. We also right *findMin* and *findMax* methods. In case of a BST, we can find the minimum and maximum value in it. The minimum will be the left most node of the tree while the right most node of BST will give the maximum value. You can confirm it by applying this on the above tree.

C++ Code for avlInsert method

Now let's see the C++ code of *avlinsert* method. Now we have to include this balancing procedure in the *insert* method. We have already written this *insert* method which takes some value and adds a node in the tree. That procedure does not perform balancing. Now we will include this balancing feature in our *insert* method so that the newly created tree fulfills the AVL condition.

Here is the code of the function.

```
/* This is the function used to insert nodes satisfying the AVL condition.*/

TreeNode<int>*   avlInsert(TreeNode<int>* root, int info)
{
   if( info < root->getInfo() ){
      root->setLeft(avlInsert(root->getLeft(), info));
      int htdiff = height(root->getLeft()) – height(root->getRight());
      if( htdiff == 2 )
         if( info < root->getLeft()->getInfo() ) // outside insertion case
           root = singleRightRotation( root );
         else  // inside insertion case
           root = doubleLeftRightRotation( root );
  }
  else if(info > root->getInfo() ) {
      root->setRight(avlInsert(root->getRight(),info));
      int htdiff = height(root->getRight()) – height(root->getLeft());
      if( htdiff == 2 )
         if( info > root->getRight()->getInfo() )
            root = singleLeftRotation( root );
         else
            root = doubleRightLeftRotation( root );
   }
```

```
    // else a node with info is already in the tree. In
   // case, reset the height of this root node.
    int ht = Max(height(root->getLeft()), height(root->getRight()));
    root->setHeight( ht + 1 ); // new height for root.
    return root;
}
```

We have named the function as *avlInsert*. The input arguments are *root* node and the *info* is of the type *int*. In our example, we are having a tree of *int* data type. But of course, we can have any other data type. The return type of *avlInsert* method is *TreeNode<int>* *. This *info* will be inserted either in the right subtree or left subtree of *root*. The first *if* statement is making this decision by comparing the value of *info* and the value of root. If the new *info* value is less than the info value of *root*, the new node will be inserted in the left subtree of root. In the next statement, we have a recursive call as seen in the following statement.

```
avlInsert(root->getLeft(), info)
```

Here we are calling *avlInsert* method again. We are passing it the first node of the left subtree and the *info* value. We are trying to insert this *info* in the left subtree of root if it does not already exist in it. In the same statement, we are setting the left of root as the return of the *avlInsert* method. Why we are doing *setLeft*? As we know that this is an AVL tree and the new data is going to be inserted in the left subtree of root. We have to balance this tree after the insertion of the node. After insertion, left subtree may be rearranged to balance the tree and its root can be changed. You have seen the previous example in which node 1 was the root node in the start and in the end node 7 was its root node. During the process of creation of that tree, the root nodes have been changing. After the return from the recursive call in our method, the left node may be different than the one that was before insertion. The complete call is as:

```
root->setLeft(avlInsert(root->getLeft(), info));
```

Here we are inserting the new node and also getting the root of the left subtree after the insertion of the new node. If the root of the left subtree has been changed after the insertion, we will update our tree by assigning this new left-node to the left of the root.

Due to the insertion of the new node if we have rearranged the tree, then the balance factor of the root node can be changed. So after this, we check the balance factor of the node. We call a function *height* to get the height of left and right subtree of the root. The *height* function takes a node as a parameter and calculates the height from that node. It returns an integer. After getting the heights of left and right subtrees, we take its difference as:

```
int htdiff = height(root->getLeft()) – height(root->getRight());
```

Now if the difference of the heights of left and right subtrees is greater than 1, we have to rebalance the tree. We are trying to balance the tree during the insertion of new node in the tree. If this difference is 2, we will perform single rotation or double

rotation to balance the tree. We have to deal with one of the four cases as discussed earlier. This rotation will be carried out with only one node where the balance factor is 2. As a result, the tree will be balanced. We do not need to rotate each node to balance the tree. This is characteristic of an AVL tree.

We have seen these rotations in the diagrams. Now we have to perform these ones in the code. We have four cases. The new node is inserted in the left or right subtree of the left child of α node or the new node is inserted in the left or right subtree of the right child of α node. How can we identify this in the code? We will identify this with the help of the info value of the new node.

We can check whether the difference (*htdiff*) is equal to 2 or not. We have *if* condition that checks that the *info* is less than the value of left node of the current node (α node). If this condition is true, it shows that it is the case of outside insertion. The node will be inserted as the left-most child. We have to perform single rotation. If the else part is executed, it means that this is the case of inside insertion. So we have to perform double rotation. We are passing α node to the rotation function. When a programmer performs these rotations, he gets another node that is assigned to root. This is due to the fact that the root may change during the rotations.

We encounter this case when the new node is inserted in the left subtree of the root. The new node can be inserted in the right subtree of the root. With respect to symmetry, that code is also similar. You can easily understand that. In the code, we have if-else condition that checks whether the *info* is greater than the value of root. Therefore, it will be inserted in the right subtree. Here again, we made a recursive call to *avlInsert* by passing it the first node of the right subtree of the root. When this recursive call is completed, we may have to perform rotations. We set the right node of the root as the node returned by the *avlInsert*. After this, we check the balance factor. We calculate the difference as:

```
int htdiff = height(root->getRight()) – height(root->getLeft());
```

In the previous case, we have subtracted the height of right tree from left tree. Here we have subtraction the height of left tree from the right tree. This is due to the fact that we want to avoid the -ve value. But it does not matter whether we get the -ve value. We can take its absolute value and test it. If the value of *htdiff* is 2 , we have to identify the case. We check that the new node is to be inserted in the right subtree of the right child of α node or in the left subtree of the right child of α node. If the *info* is greater than the value of the right child of root, it is the case 4. Here we will restore the balance of the tree by performing single rotation. If the *else* part is executed, we have to perform the double rotation. These rotation routines will return the root of the rearranged tree that will be set as a root.

In the end, we have the third case. This is the case when the *info* is equal to the root. It means that the data is already in the tree. Here we will readjust the height if it is needed. We will take the maximum height between the left and right subtree and set this height plus one as the height of the root. We have added one to the height after adding one node to the tree. In the end, we return the root. We have included the balancing functionality in our tree and it will remain balanced.

We have not seen the code of rotation routines so far. The code of these methods is also very simple. We have to move some of pointers. You can easily understand that code. We will discuss this in the next lecture. Please see the code of single and double rotation in the book and try to understand it.

Data Structures

Lecture No. 23

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.4.1, 4.4.2

Summary

- Single Right Rotation
- Single Left Rotation
- Double Right-Left Rotation
- Double Left- Right Rotation
- Deletion in AVL Tree
- Cases of Deletion in AVL Tree

We demonstrated how to write the insert procedure for an AVL tree in the previous lecture. The single and double rotation call methods were also discussed at length. While inserting a node if we see that the tree is going to be unbalanced, it means that the AVL condition is being violated. We also carried out the balancing of the tree by calling single or double rotation. Now let's see the code of these routines i.e. single and double rotation.

Single Right Rotation

At first, we will discuss the code of SingleRightRotation function. Its argument is TreeNode, which is given the name k2. The reason of using k2 is that it alongwtih k1 etc was used in the earlier discussion. After the rotation, this function will return a pointer, as tree will be re-organized. Resultantly, its root will be changed. Following is the code of this SingleRightRotation function.

```
TreeNode<int>*  singleRightRotation(TreeNode<int>* k2)
{
   if( k2 == NULL ) return NULL;
   // k1 (first node in k2's left subtree) will be the new root
   TreeNode<int>* k1 = k2->getLeft();

   // Y moves from k1's right to k2's left
   k2->setLeft( k1->getRight() );
   k1->setRight(k2);

   // reassign heights. First k2
   int h = Max(height(k2->getLeft()), height(k2->getRight()));
```

```
    k2->setHeight( h+1 );
    // k2 is now k1's right subtree
    h = Max( height(k1->getLeft()), k2->getHeight());
    k1->setHeight( h+1 );
    return k1;
}
```

In the function, at first, we check whether k2 is NULL. If it is NULL, the function is exited by returning NULL. If it is not NULL, the rotation process starts. The figure below depicts the single right rotation process. In this diagram, we see that k2 has been shown as the root node to this function. We see that k1 is the left subtree of k2.

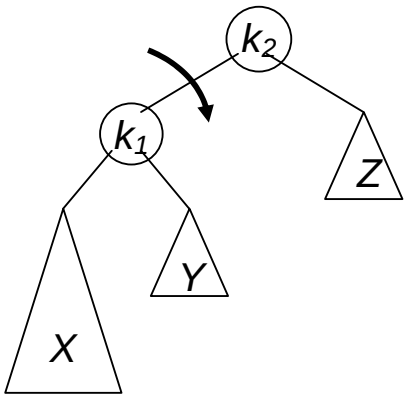


Fig 23.1: Single Right Rotation

We are going to apply single right rotation on the link between k1 and k2. The node k1 will be the new root node after rotation. So we get its value by the following statement

```
TreeNode <int>* k1 = k2 -> getLeft() ;
```

Due to the single right rotation, k2 has come down, resulting in the upward movement of k1. The tree Y has to find its new place. The place of the trees X and Z remains intact. The change of place of Y is written in the code as below.

```
k2->setLeft( k1->getRight() );
```

In the above statement, we get the right child of k1 (i.e. k1 -> getRight()) i.e. Y and pass it to setLeft function of k2. Thus it becomes the left child of k2. By the statement

```
k1 -> setRight (k2) ;
```

We set the node k2 as right child of k1. Now after these three statements, the tree has been transformed into the following figure.

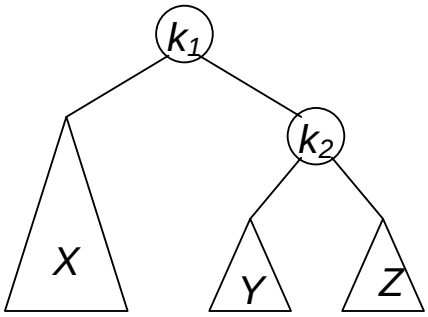


Fig 23.2: Tree after rotation

From the above figure, it is reflected that k1 is now the root of the tree while k2 is its right child. The Y, earlier the right subtree of k1 (see fig 23.1), has now become the right subtree of k2. We see that the inorder traversal of this tree is X k1 Y k2 Z. It is the same as of the tree before rotation in fig 23.1 i.e. X k1 Y k2 Z.
Now we set the heights of k1 and k2 in this re-arranged tree. In the code, to set the height of k2, we have written

```
int h = Max(height(k2->getLeft()) , height(k2->getRight()));
k2->setHeight( h+1 );
```

Here we take an integer h and assign it a value. This value is the maximum height among the two subtrees i.e. height of left subtree of k2 and height of right subtree of k2. Then we add 1 to this value and set this value as the height of k2. We add 1 to h as the height of root is one more than its child.
Similarly we set the height of k1 and get the heights of its left and right subtrees before finding the higher value of these two by the Max function. Afterwards, we add 1 to this value and set this value as the height of k1. The following two statements perform this task.

```
h = Max( height(k1->getLeft()), k2->getHeight());
k1->setHeight( h+1 );
```

In the end, we return the root node i.e. k1. Thus the right single rotation is completed. In this routine, we saw that this routine takes root node of a tree as an argument and then rearranges it, resets the heights and returns the new root node.

Height Function

In the above SingleRightRotation, we used the *height* routine. This routine calls the getHeight function for the argument passed to it and returns the value got from that function. If there is an empty tree, then by definition its height will be -1. So in this routine, we return the value -1 if the argument passed to it is NULL. Following is the code of this routine.

```
int height( TreeNode<int>* node )
{
   if( node != NULL ) return node->getHeight();
   return –1;
}
```

Single Left Rotation

This rotation is almost similar to the single right rotation. We start from k1. k1 is the root while k2 is its right child. Due to the left rotation, the right child k2 of the root k1 will become the root. The k1 will go down to the left child of k2. The following figure shows that we have to change the positions of some pointers. The condition of the tree before and after single left rotation is also seen.

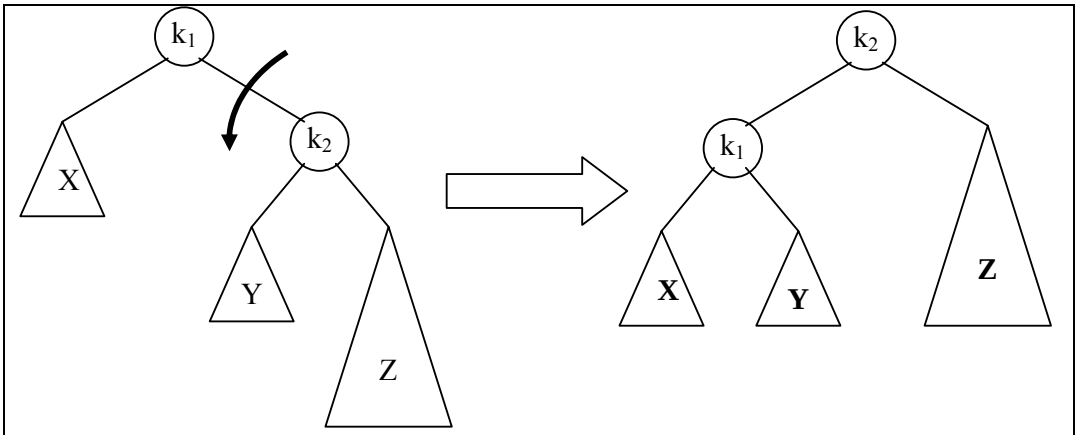


Fig 23.3: Single Left Rotation

Following is the code of the function performing the single left rotation.

```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
   if( k1 == NULL ) return NULL;

   // k2 is now the new root
   TreeNode<int>* k2 = k1->getRight();
   k1->setRight( k2->getLeft() ); // Y
   k2->setLeft( k1 );

   // reassign heights. First k1 (demoted)
   int h = Max(height(k1->getLeft()), height(k1->getRight()));
   k1->setHeight( h+1 );

   // k1 is now k2's left subtree
   h = Max( height(k2->getRight()), k1->getHeight());
   k2->setHeight( h+1 );
   return k2;
```

```
}
```

In the above code, it is evident that if the node passed to the function is not NULL (that means k1 is not NULL), we assign to k2 the value of k1 -> getRight (). Now in the rotation, Y tree i.e. the left subtree of k2, becomes the right subtree of k1. We write it in the next statement as.

```
k1->setRight( k2->getLeft() );
```

In this statement, we get the left child of k2 i.e. Y and set it as the right child of k1. Afterwards, k1 is set as the left child of k2. The following statement reflects this process.

```
k2->setLeft( k1 );
```

After single left rotation, the k2 becomes the root node after going up. However, k1 goes down as the left child of k2. The tree Y, earlier the left child of k2 has, now become the right child of k1.X and Z continue to remain at the previous positions. The tree has been transformed with changes into the heights of k1 and k2. We re-assign the heights to these nodes in line with the process adopted in the right single rotation. At first, we adjust the height of k1. We get the heights of left and right subtrees of k1. The greater of these is taken and assigned to an int h. Then we add 1 to this value and set it as the height of k1. The following two lines of the code execute this task.

```
int h = Max(height(k1->getLeft()), height(k1->getRight()));
k1->setHeight( h+1 );
```

Similarly, we adjust the height of k2 by getting the greater one among the heights of its right subtree and its left subtree. Taking k1 as its left subtree, we get the height of k1. We add 1 and set it as the height of k2. Following are the statements that perform this task.

```
h = Max( height(k2->getRight()), k1->getHeight());
k2->setHeight( h+1 );
```

Finally, k2 is returned as the root of the tree.

Double Right-Left Rotation

As obvious from the nomenclature, in the double rotation, we at first carry out the right rotation before going ahead with the left rotation. Let's say that we pass it the node k1. Look at the following figure to see what are the nodes k1, k2 and k3. The figure 23.4 shows the first step (A) of double right-left rotation that is the single right rotation. It shows the rearranged tree on the right side in the figure.

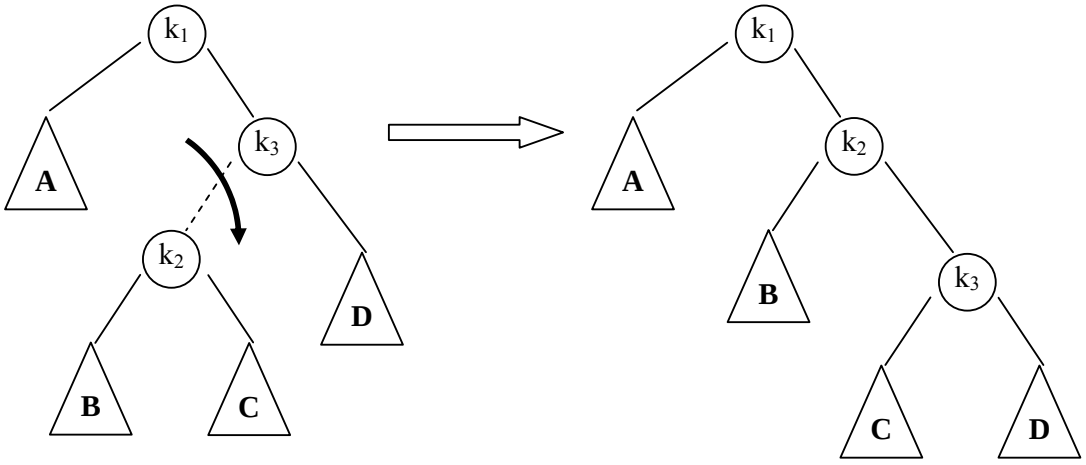


Fig 23. 4: Double Right-Left Rotation (A)

In the left portion of the above figure, we see that k1 is the root of the tree. k3 is the right child of k1 while k2 is the left child of k3. A, B, C and D are trees. We carry out the right rotation between the link of k3 and k2. In the code, it is shown that if k1 is not NULL, we go ahead and perform the rotation. The code of this double rotation is given below.

```
TreeNode<int>* doubleRightLeftRotation(TreeNode<int>* k1)
{
   if( k1 == NULL ) return NULL;

   // single right rotate with k3 (k1's right child)
   k1->setRight( singleRightRotation(k1->getRight()));

   // now single left rotate with k1 as the root
   return singleLeftRotation(k1);
}
```

We perform the right rotation with the help of k3. Now think about the single right rotation. We are at k3 and its left child k2 will be involved in the rotation. In the code, for the single right rotation, we have written

```
k1->setRight( singleRightRotation(k1->getRight()));
```

Here, we are passing the right child of k1 (i.e. k3) to the single right rotation. The function singleRightRotation itself will find the left child of k3 to perform the single right rotation. The singleRightRotation will take k3 downward and bring up its left child i.e. k2. The left subtree of k2 i.e. C, now becomes the right subtree of k3. Due to this single right rotation, the tree will be transformed into the one, shown in the right

part of the figure 23.4.
After this we apply the single left rotation on this tree that is got from the single right rotation. This single left rotation is done by passing it k1. In this rotation, k1 goes down and k2 i.e. the right child of k1, becomes the root. We return the result of single left rotation. The following statement in the above code performs this task.

return singleLeftRotation(k1);

Thus the tree gets its final form i.e. after the double right-left rotation. The following figure shows the single left rotation and the final form of the tree.

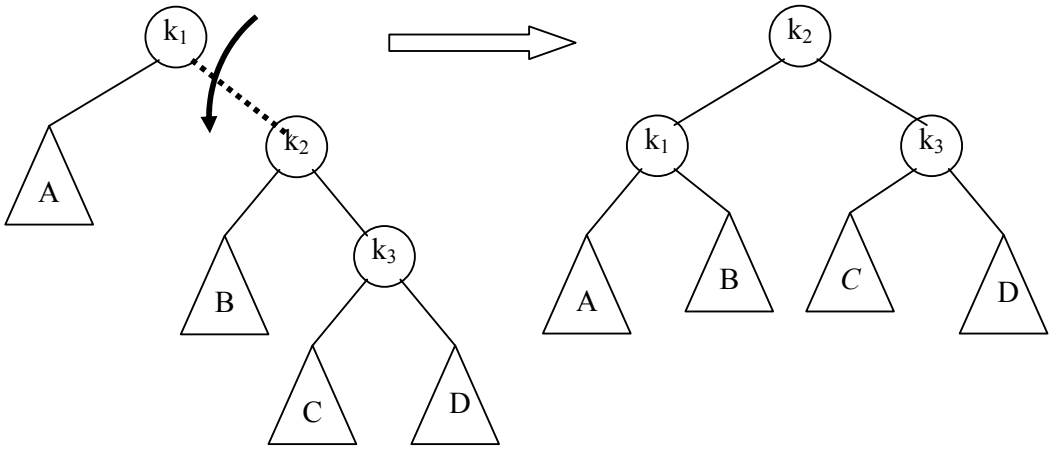


Fig 23.5: Double right-left rotation (B)

In the above figure, it was witnessed that the double rotation consists of few statements. That's why, we have had written the routines for single right and left rotations. In the double right-left rotation, we just carried out a single right rotation and the single left rotation to complete the double right-left rotation.

Double Left-Right Rotation

While performing the double left-right rotation, we simply carry out the single left rotation at first. It is followed by the single right rotation. Following is the code of the double Left-Right rotation.

```
TreeNode<int>* doubleLeftRightRotation(TreeNode<int>* k3)
{
   if( k3 == NULL ) return NULL;

   // single left rotate with k1 (k3's left child)
   k3->setLeft( singleLeftRotation(k3->getLeft()));

   // now single right rotate with k3 as the root
   return singleRightRotation(k3);
```

```
}
```

In the above code, we receive the argument node as k3. As we have to do the left rotation first, it is applied to the left child of k3. From the figure below (Fig 23.6), we see that the left child of k3 is k1. However, when we send k1 to the singleLeftRotation it will take the right child of k1 i.e. k2. This process will rotate the link between k1and k2. The tree formed as a result of this single left rotation, is shown at the right side in the figure 23.6.

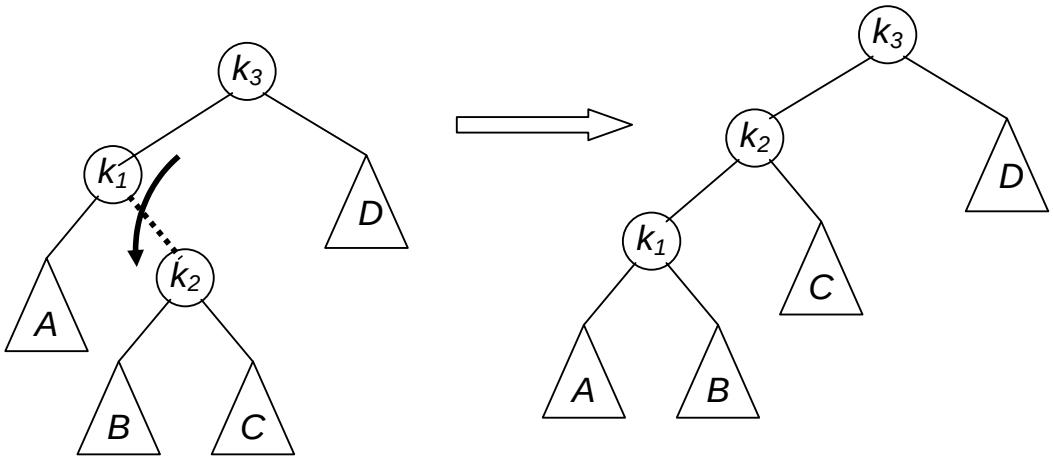


Fig 23.6: Double left-right rotation (A)

In this new rearranged tree, k2 comes up while k1 goes down, becoming the left subtree of k2. The subtree B, earlier left subtree of k2, now becomes the right subtree of k1.

The second step in the double left-right rotation is to apply the single right rotation on this new tree. We will carry out the single right rotation on k3. The pictorial representation of this single right rotation on k3 is given in the figure below.

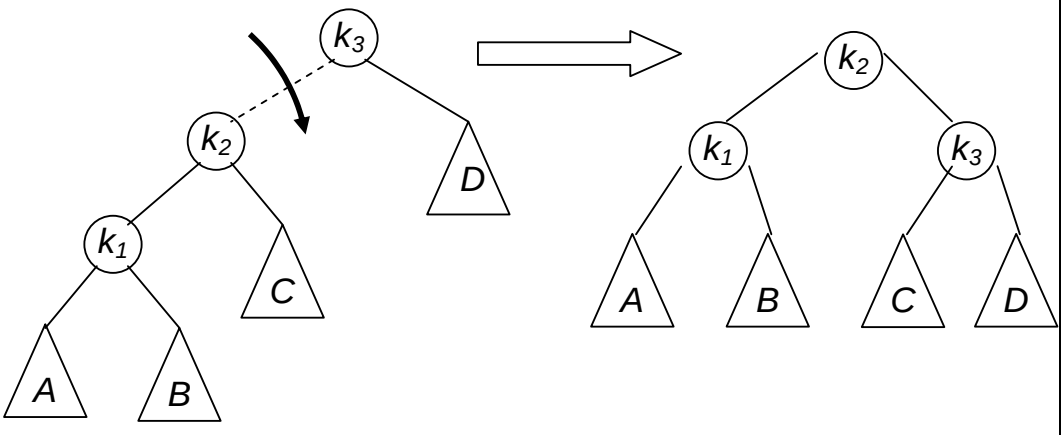


Fig 23.7: Double left-right rotation (B)

In the code, we wrote the statement for this as below.

```
return singleRightRotation(k3);
```

We pass k3 to the singleRightRotation. It is, internally applied to the left child of k3 ie. k2. Now due to the right rotation, k2 will come up and k3 will become its right child. The node k1 will remain the left child of k2. The subtree C that was the right subtree of k2, will now be the left subtree of k3.

By now, the discussion on the insert routine has been completed. We have written the insert routine so that data item could be passed to it and inserted at a proper position in the tree. After inserting a node, the routine checks the balance or height factors and does the left or right rotation if needed and re-balances the tree.

Note that a property of the AVL tree is that while inserting a node in an AVL tree if we have to balance the tree then we have to do only one single right or left rotation or one double rotation to balance the tree. There is no such situation that we have to do a number of rotations. We do the one single or one double rotation at the node whose balance has become 2 after inserting a node.

Deletion in AVL Tree

The deletion of a data item from a data structure has always been difficult whatever data structure we use. The deletion is the inverse of insertion. In deletion there is a given value x and an AVL tree T. We delete the node containing the value x and rebalance the tree if it becomes unbalance after deleting the node. We will see that the deletion of a node is considerably more complex than the insertion of a node. It is complex in the sense that we may have to do more than one rotations to rebalance the tree after deleting a node. We will discuss the deletion case by case and will see that about what points we have to take care in the deletion process.

We are not going to write the code for deletion here. If we have to use AVL tree routines or class, the deletion and insertion routines of AVL tree are available in standard library. We can use these routines in our program. We have no need to write these routines. But here we discuss these to understand their functionality.

We know that insertion in a height-balanced tree requires at most one single rotation or one double rotation. We do this rotation at the node whose balance violates the AVL condition. We can use rotations to restore the balance when we do a deletion. If the tree becomes unbalance after deleting a node then we use rotations to rebalance it. We may have to do a rotation at every level of the tree. Thus in the worst case of deletion we have to do $\log_2 N$ rotations. As $\log_2 N$ is the number of levels of a tree of N nodes.

Let's consider an example of deleting a node from a tree. In this example, we will discuss the worst case of deletion that is we have to do rotation at each level after deleting a node. Look at the following figure i.e. Fig 23.8. In this figure the root of the tree is node N and we have expanded only the left subtree of this node. The right subtree of it is indicated by a triangle. We focus on the left subtree of N. The balance of each non-leaf node of the left subtree of N is –1. This means that at every non-leaf node the depth/height of left subtree is one shorter than the height of right subtree. For example look at the node C. The left subtree of C is one level deep where as it's right subtree is two levels deep. So balance of it is 1 – 2 = -1. If we look at node I its left

subtree has height 2 as there are two levels where nodes G and H exists. The right subtree of I has number of levels (i.e. height) 3 where exists the nodes K, L and M respectively. Thus the balance of I is 2 – 3 = -1. Similarly we can see that other nodes also have the balance –1. This tree is shown in the following figure.

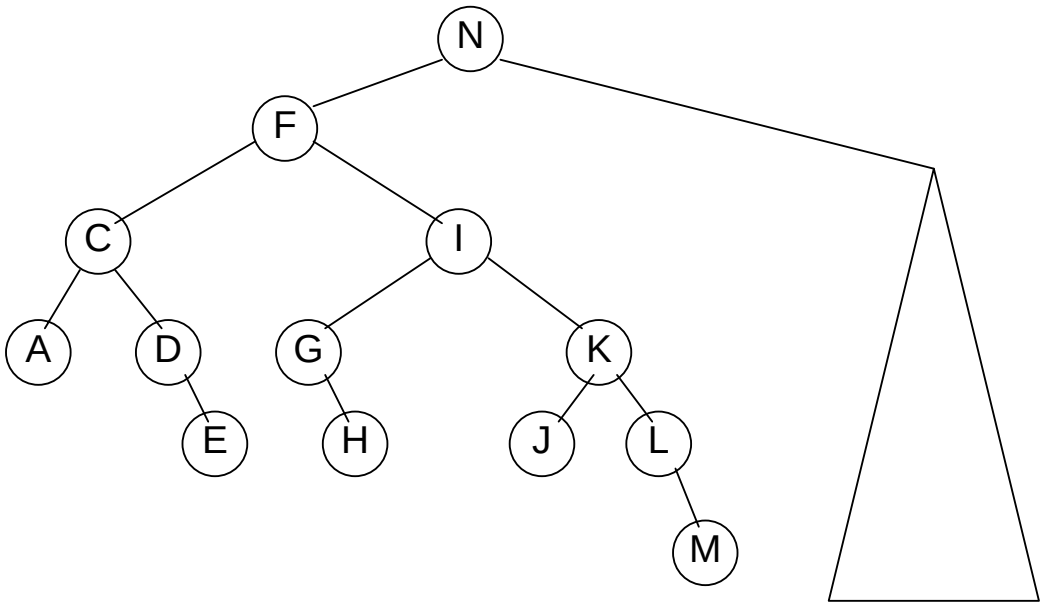


Fig 23.8: Tree under consideration

Here in this tree, the deletion of node A from the left subtree causes the worst case of deletion in the sense that we have to do a large number of rotations. Now we delete the node A from the tree. The effect of this deletion is that if we consider the node C the height of its left subtree is zero now. The height of the right subtree of C is 2. Thus the balance of C is 2 now. This makes the tree unbalance. Now we will do a rotation to make the tree balanced. We rotate the right subtree of C that means the link between C and D so that D comes up and C goes down. This is mentioned in the figure below.

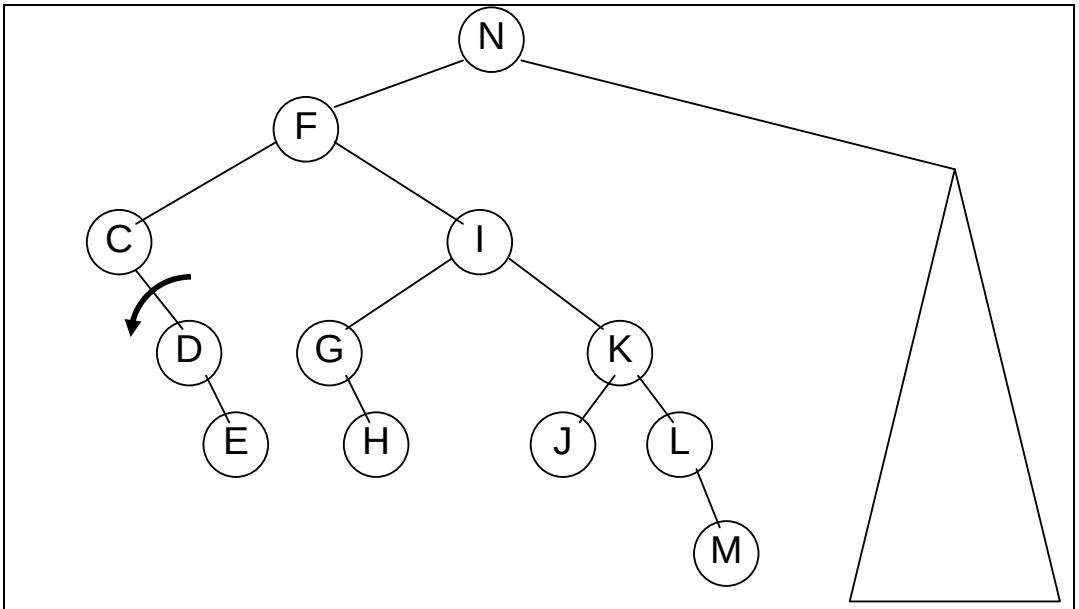


Fig 23.9: Single left rotation after deleting node A

After this rotation the tree has transformed into the following figure. Now D becomes the left child of F and C becomes the left child of D.

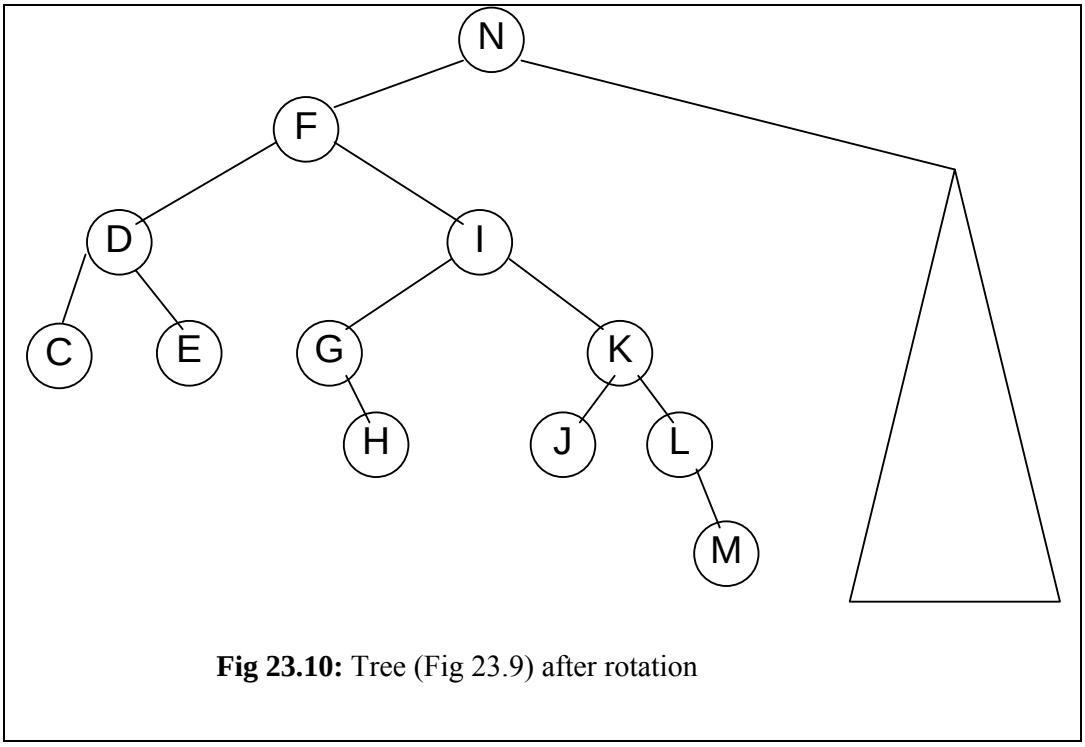


Fig 23.10: Tree (Fig 23.9) after rotation

By looking at the inorder traversal of the tree, we notice that it is preserved. The inorder traversal of the tree before rotation (i.e. fig 23.8) is C D E F G H I J K L M N.

Now if we traverse the tree after rotation (i.e. fig 23.9) by inorder traversal we get C D E F G H I J K L M N, which is the same as it was before rotation.
After this rotation we see that the tree having D as root is balanced. But if we see the node F we notice that the height of its left subtree is 2 and height of its right subtree is 4. Thus the balance of F is –2 (or 2 if we take the absolute value) now. Thus the tree becomes unbalance. So we have to do rotation again to balance the tree. The whole left subtree of F is shorter so we do the left rotation on the link of F and I (in the tree in fig 23.9) to bring F down and I up so that the difference of height could be less. After rotation the tree gets the new form that is shown in the figure below.

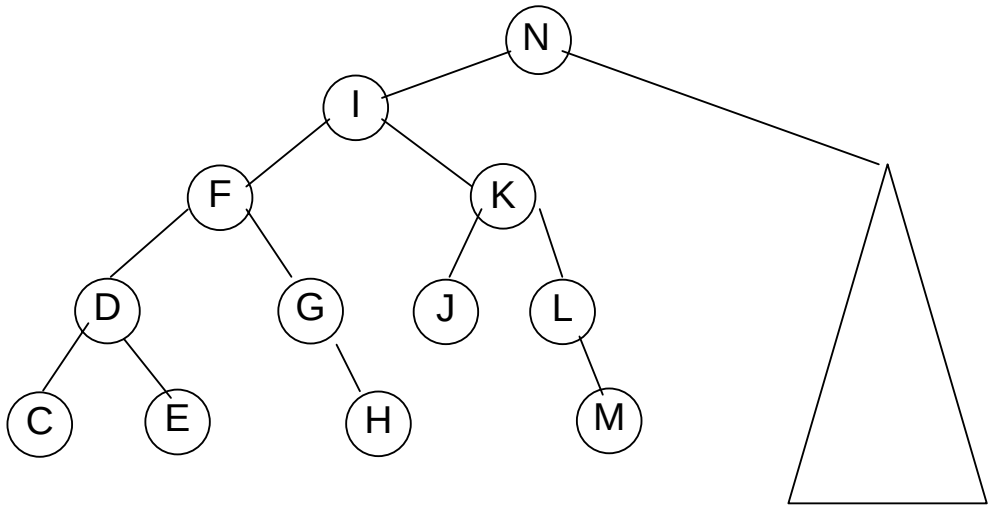


Fig 23.11: Tree (fig 23.10) after rotation

Here we see that the nodes G and H, which were in the left subtree of I, now become the right subtree of F. We see that the tree with I as root is balanced now. Now we consider the node N. We have not expanded the right subtree of N. Although we have not shown but there may be nodes in the right subtree of N. If the difference of heights of left and right subtree of N is greater than 1 then we have to do rotation on N node to balance the tree.
Thus we see that there may be such a tree that if we delete a node from it we have to do rotation at each level of the tree. We notice that we have to do more rotations in deletion as compared to insertion. In deletion when we delete a node we have to check the balance at each level up to the root. We do rotation if any node at any level violates the AVL condition. If nodes at a level do not violate AVL condition then we do not stop here we check the nodes at each level and go up to the root. We know that a binary tree has $\log_2 N$ levels (where N is total number of nodes) thus we have to do $\log_2 N$ rotations. We have to identify the required rotations that mean we have to identify that which one rotation out of the four rotations (i.e. single left rotation, single right rotation, double right-left rotation and double left-right rotation) we have to do. We have to identify this at each level.

We can summarize the deletion procedure as under.

Delete the node as in binary search tree. We have seen in the discussion of deleting a node from a BST that we have three cases, which we discussed as follows

Case I: The node to be deleted is the leaf node i.e. it has no right or left child. It is very simple to delete such node. We make the pointer in the parent node pointing to this node as NULL. If the memory for the node has been dynamically allocated, we will release it.

Case II: The node to be deleted has either left child (subtree) or right child (subtree).
In this case we bypass the node in such a way that we find the inorder successor of this node and then link the parent of the node to be deleted to this successor node. Thus the node was deleted.

Case III: The node to be deleted has both the left and right children (subtree). This is the most difficult case. In this case we find the inorder successor of the node. The left most node of the right subtree will be the inorder successor of it. We put the value of that inorder successor node into the node to be deleted. After it we delete the inorder successor node recursively.

In deletion in AVL tree, we delete the node as we delete it in a BST. In third case of deletion in BST we note that the node deleted will be either a leaf or have just one subtree (that will be the right subtree as node deleted is the left most subtree so it cannot have a left subtree). Now we are talking about deletion in an AVL tree. Since this is an AVL tree so if the deleted node has one subtree that subtree contains only one node. Why it is so? Think about its reason and answer.
After deleting the node we traverse up the tree from the deleted node checking the balance of each node at each level up to the root. Thus the deletion in AVL tree is like the deletion in BST except that here in AVL tree we have to rebalance the tree using rotations.

Cases of Deletion in AVL Tree

Now let's consider the cases of deletion that will help to identify what rotation will be applied at what point.
There are 5 cases to consider. Let's go through the cases graphically and determine what actions are needed to be taken. We will not develop the C++ code for deleteNode in AVL tree. This is left as an exercise.

Case 1a:
The first case is that the parent of the deleted node had a balance of 0 and the node was deleted in the parent's *left* subtree.
In the following figure (Fig 23.12) the left portion shows the parent node with a horizontal line in it. This horizontal line indicates that the left and right subtrees of this node have the same heights and thus the balance of this node is 0. When we delete a node from its left subtree then height of its right subtree becomes larger than the left subtree. The right portion in the figure shows this. We are showing a symbol instead of balance of node inside the node. The notation (symbol) in the node

indicates that the height of left subtree is shorter than the height of right subtree of the node.

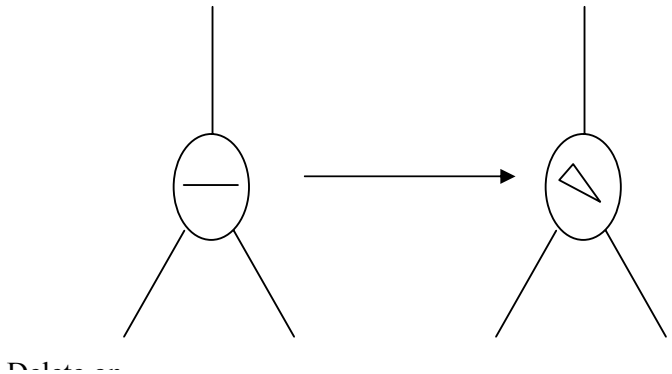


Fig 23.12: Deletion Case 1a

Now the action that will be taken in this case is that, change the balance of the parent node and stop. No further effect on balance of any higher node. There is no need of rotation in this case. Thus it is the easiest case of deletion.

Let's consider an example to demonstrate the above case.
Consider the tree shown in the following figure i.e. Fig 23.13. This is a perfectly balanced tree. The root of this tree is 4 and nodes 1, 2 and 3 are in its left subtree. The nodes 5, 6 and 7 are in the right subtree.

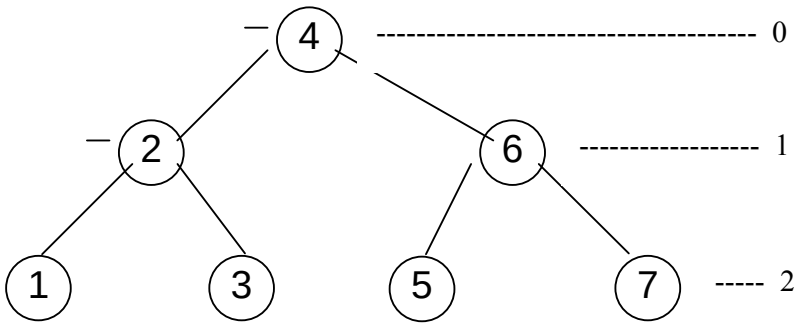


Fig 23.13: Tree under consideration

Consider the node 2. We have shown the balance of this node with a horizontal line, which indicates that the height of its left subtree is equal to that of its right subtree. Similarly we have shown the balance of node 4.
Now we remove the node 1 which is the left subtree of node 2. After removing the left child (subtree) of 2 the height of left subtree of 2 is 0. The height of right subtree of 2 is 1 so the balance of node 2 becomes –1. This is shown in the figure by placing a sign that is down ward to right side, which indicates that height of right subtree is greater

than left subtree. Now we check the node 4. Here the height of left subtree of it is still 2. The height of its right subtree is also 2. So balance of the node 4 is 0 that is indicated by the small horizontal line (minus sign) in the figure below. Here we don't need to change the balance of 4.

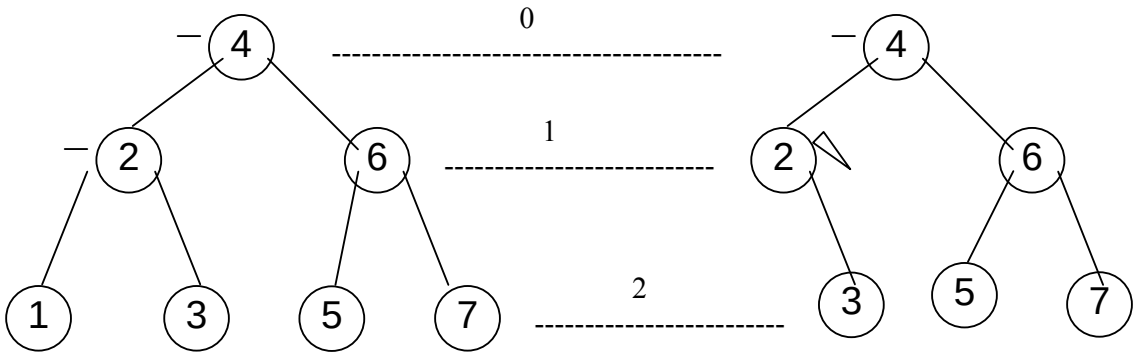


Fig 23.14: Tree before and after deleting node 1

Case 1b:
This case is symmetric to case 1a. In this case, the parent of the deleted node had a balance of 0 and the node was deleted in the parent's *right* subtree. The following figure shows that the balance of the node was zero as left and right subtree of it have the same heights.

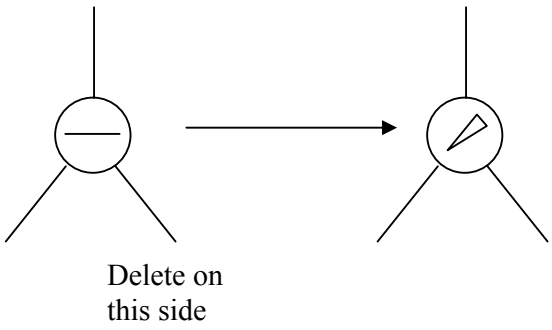


Fig23.15: Deletion Case 1b

After removing the right child the balance of it changes and it becomes 1, as the height of its left subtree is 1 greater than the height of right subtree. The action performed in this case is the same as that in case 1a. That is change the balance of the parent node and stop. No further effect on balance of any higher node. The previous example can be done for this case only with the change that remove the right child of node 2 i.e. 3.

Case 2a:

This is the case where the parent of the deleted node had a balance of 1 and the node was deleted in the parent's *left* subtree. This means that the height of left subtree of the parent node is 1 greater than the height of its right subtree. It is shown in the left portion of the figure below.

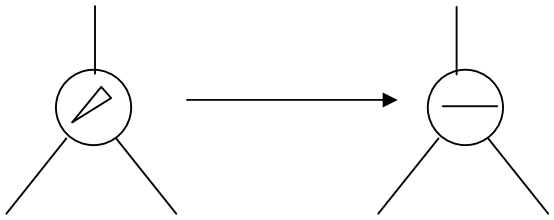


Fig 23.16: Deletion Case 2a

Now if we remove a node from the left subtree of this node then the height of left subtree will decrease by 1 and get equal to the height of right subtree. Thus the balance of this node will be zero. In this case, the action that we will perform to balance the tree is that change the balance of the parent node. This deletion of node may have caused imbalance in higher nodes. So it is advised to continue up to the root of the tree. We will do rotation wherever the balance of the node violates the AVL condition.

Data Structures

Lecture No. 24

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.4

Summary

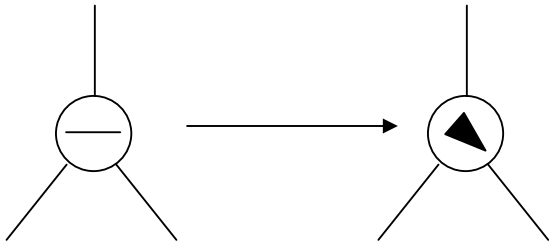
- Deletion in AVL Tree
- Other Uses of Binary Trees

Deletion in AVL Tree

At the end of last lecture, we were discussing about deleting a node from an AVL tree. There are five cases to consider while deleting a node of an AVL tree. When a node is deleted, the tree can become unbalanced. We calculate the *balance factor* of each node and perform rotation for unbalanced nodes. But this rotation can prolong to the *root* node. In case of insertion, only one node's balance was adjusted as we saw in previous lectures but in case of deletion, this process of rotation may expand to the *root* node. However, there may also be cases when we delete a node and perform no or one rotation only.

Now, we will see the five cases of deletion. A side note is that we are not going to implement these cases in C++ in this lecture, you can do it yourself as an exercise with the help of the code given inside your text book. In this lecture, the emphasis will be on the deletion process and what necessary actions we take when a node is required to be deleted from an AVL tree. Actually, there are two kinds of actions taken here, one is deletion and the other one is the rotation of the nodes.

Case 1a: The parent of the deleted node had a balance of 0 and a node was deleted in the parent's left subtree.



Delete on this side

Fig 24.1

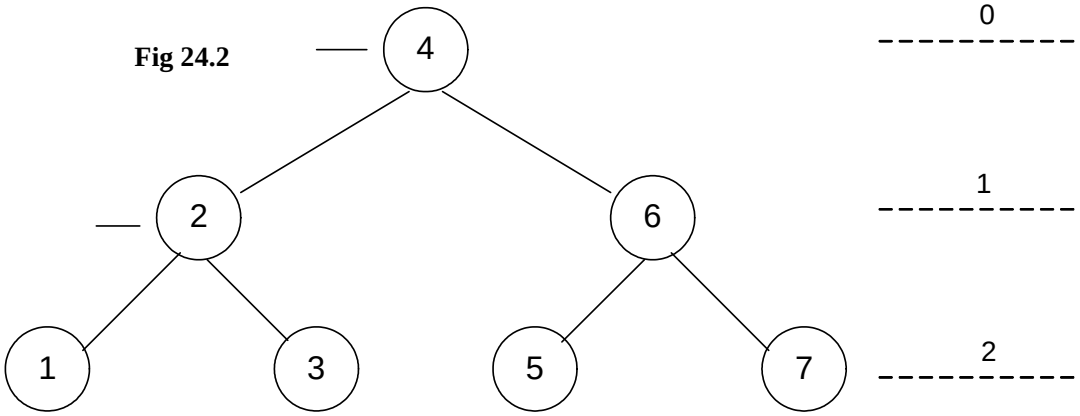
In the left tree in the *Fig 24.1*, the horizontal line inside the tree node indicates that the balance is 0, the *right* and *left* subtrees of the node are of equal levels. Now, when a node is deleted from *the left* subtree of this node, this may reduce by one level and cause the balance of the *right* subtree of the node to increase by 1 relatively. The

balance of the node in favor of the *right* subtree is shown by a triangular knob tilted towards right. Now, the action required in this case to make the tree balanced again is:

Change the balance of the parent node and stop. There is no further effect on balance of any higher node.

In this case, the balance of the tree is changed from 0 to –1, which is within the defined limits of AVL tree, therefore, no rotation is performed in this case.
Below is a tree in which the height of the *left* subtree does not change after deleting one node from it.

Fig 24.2



The node 4 is the root node, nodes 2 and 6 are on level 1 and nodes 1, 3, 5, 7 are shown on level 2. Now, if we delete the node 1, the balance of the node 2 is tilted towards right, it is –1. The balance of the *root* node 4 is unchanged as there is no change in the number of levels within *right* and *left* subtrees of it. Similarly, there is no change in the balances of other nodes. So we don't need to perform any rotation operation in this case.
Let's see the second case.
Case 1b: the parent of the deleted node had a balance of 0 and the node was deleted in the parent's right subtree.

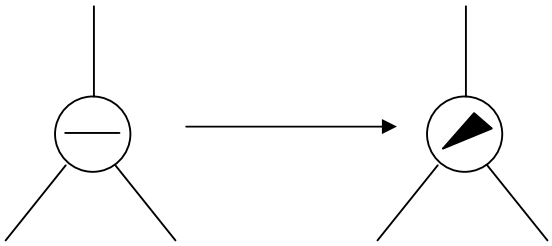


Fig 24.3

On the left of *Fig 24.3*, the tree after a node is deleted from the *right* subtree of it. The balance of the tree is tilted towards left as shown in the right tree show in the *Fig 24.3*. Now, we see what action will be required to make the tree balanced again.

Change the balance of the parent node and stop. No further effect on balance of any higher node (same as 1a).
So in this case also, we don't need to perform rotation as the tree is still an AVL (as

we saw in the *Case 1a*). It is important to note that in both of the cases above, the balance of the parent node was 0. Now, we will see the cases when the balance of the parent node is not 0 previously.

Case 2a: The parent of the deleted node had a balance of 1 and the node was deleted in the parent's left subtree.

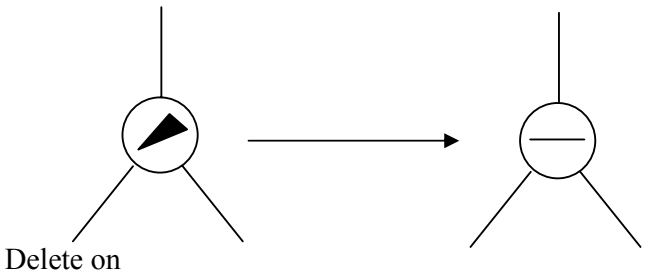


Fig 24.4

In the *Fig 24.4* ılance factor as 1, which means that the *left* subtree of the parent node is one level more than the number of levels in the *right* subtree of it. When we delete one node from the *left* subtree of the node, the height of the *left* subtree is changed and the balance becomes 0 as shown in the right side tree of *Fig 24.4*. But it is very important understand that this change of levels may cause the change of balance of higher nodes in the tree i.e.

Change the balance of the parent node. May have caused imbalance in higher nodes so continue up the tree.
So in order to ensure that the upper nodes are balanced, we calculate their *balance factor*s for all nodes in higher levels and rotate them when required.

Case 2b*: The parent of the deleted node had a balance of -1 and the node was deleted in the parent's right subtree.*
Similar to the *Case 2a*, we will do the following action:

Change the balance of the parent node. May have caused imbalance in higher nodes so continue up the tree.

Now, we see another case.

Case 3a*:The parent had balance of -1 and the node was deleted in the parent's left subtree, right subtree was balanced.*

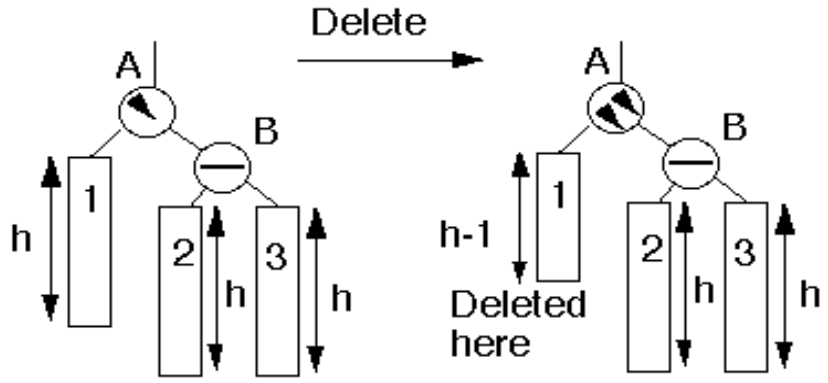
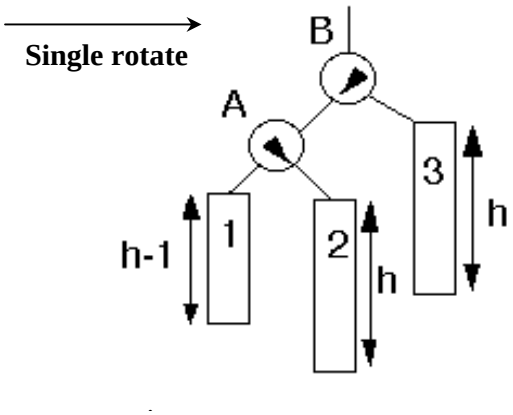


Fig 24.5

As shown in the *left* tree in *Fig 24.5*, the node A is tilted towards right but the *right* subtree of A (node B above) is balanced. The deleted node lies in the *left* subtree of the node A. After deletion, the height of the *left* subtree is changed to *h-1* as depicted in the right tree of above figure. In this situation, we will do the following action:

Perform single rotation, adjust balance. No effect on balance of higher nodes so stop here.



Node A has become the *left* subtree of node B and node 2 *left* subtree of node B has become the *right* subtree of node A. The balance of node B is tiled towards left and balance of node A is tilted towards right but somehow, both are within AVL limits. Hence, after a single rotation, the balance of the tree is restored in this case.

Case 4a*: Parent had balance of -1 and the node was deleted in the parent's left subtree, right subtree was unbalanced.*

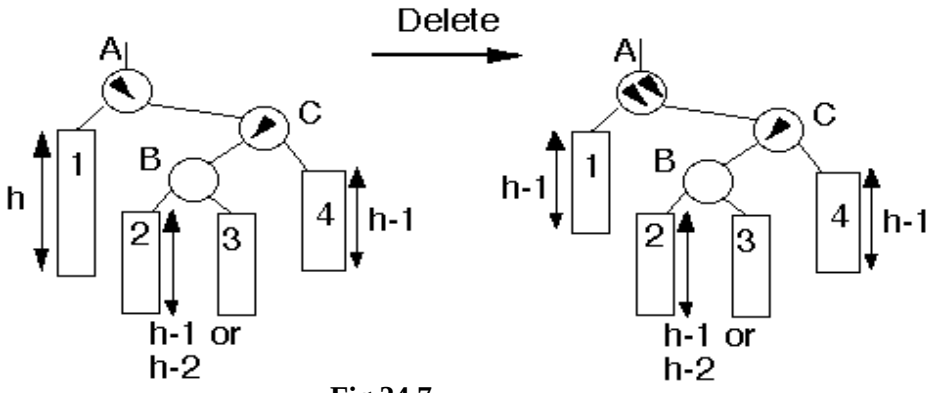


Fig 24.7

In the last *case 3a*, the *right* subtree of node A was balanced. But in this case, as shown in the figure above, the node C is tilted towards left. The node to be deleted lies in the *left* subtree of node A. After deleting the node the height of the *left* subtree of node A has become *h-1*. The balance of the node A is shown tilted towards right by showing two triangular knobs inside node A. So what is the action here.

Double rotation at B. May have affected the balance of higher nodes, so continue up the tree.

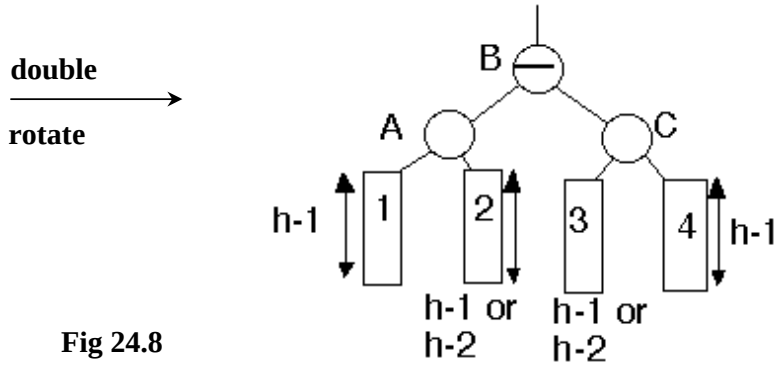


Fig 24.8

Node A, which was the *root* node previously, has become the *left* child of the new *root* node B. Node C, which was the *right* child of the *roo*t node C has now become the *right* child of the new *root* node B.

Case 5a*: The parent had balance of -1 and the node was deleted in the parent's left subtree, right subtree was unbalanced.*

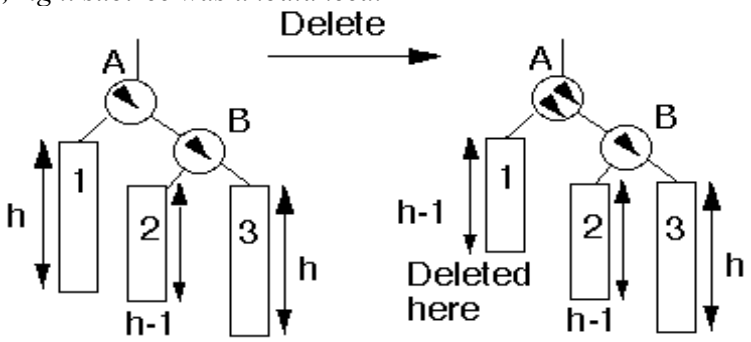


Fig 24.9

In the figure above, the *right* tree of the node B has a height of *h-1* while the *right* subtree is of height *h*. When we remove a node from the *left* subtree of node A, the new tree is shown on the right side of *Fig 24.9*. The subtree 1 has height *h-1* now, while subtrees 2 and 3 have the same heights. So the action we will do in this case is:

Single rotation at B. May have effected the balance of higher nodes, so continue up the tree

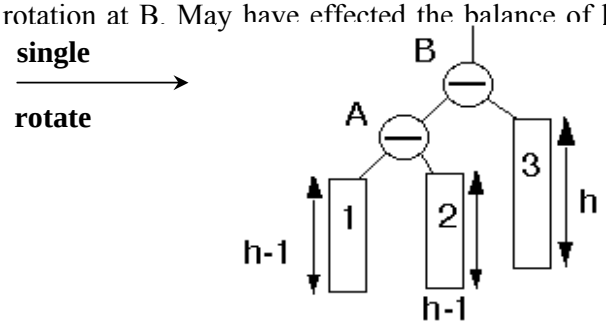


Fig 24.10

These were the five cases of deletion of a node from an AVL tree. Until now, we are trying to understand the concept using the figures. You might have noticed the phrase *'continue up the tree'* in the actions above. How will we do it? One way is to maintain the pointer of the parent node inside each node. But often the easiest method when we go in downward direction and then upward is *recursion*. In recursion, the work to be done later is pushed on to the stack and we keep on moving forward until at a certain point we back track and do remaining work present in the stack. We delete a node when we reach at the desired location and then while traversing back, do the rotation operation on our way to the *root* node.

Symmetrical to case 2b, we may also have cases 3b, 4b and 5b. This should not be a problem in doing it yourself.

Other Uses of Binary Trees

A characteristic of binary trees is that the values inside nodes on the left of a node are smaller than the value in the node. And the values inside the nodes on the right of a node are greater than the value in the node. This is the way a binary tree is constructed.

Whatever is the size of the tree, the search is performed after traversing upto $log_2 n$ levels maximum.

We have observed that the binary tree becomes a linked list and it can become shallow. The AVL conditions came into picture to control the height balance of a binary tree. While searching in an AVL tree, in the worst case scenario we have to search $1.44\ log_2 n$ levels. For searches, binary and AVL trees are the most efficient but we do have some other kinds of trees that will be studied later.

Lets see what could be some other uses of binary trees, we start our discussion with *Expression Trees*.

Expression Trees

Expression trees, the more general parse trees and abstract syntax trees are significant components of compilers.

We already know about compilers that whenever we write our program or code in some computer language like C++ or Java. These programs are compiled into assembly language or byte code (in case of Java). This assembly language code is translated converted into machine language and an executable file is made by another program called the assembler.

By the way, if you have seen your syllabus, you might have seen that there is a dedicated subject on compilers. We study in detail about compilers in that course. For this course, we will see expression or parse trees.

We will take some examples of expression trees and we will not delve into much depth of it rather that would be an introduction to expression trees.

(a+b*c)+((d*e+f)*g)

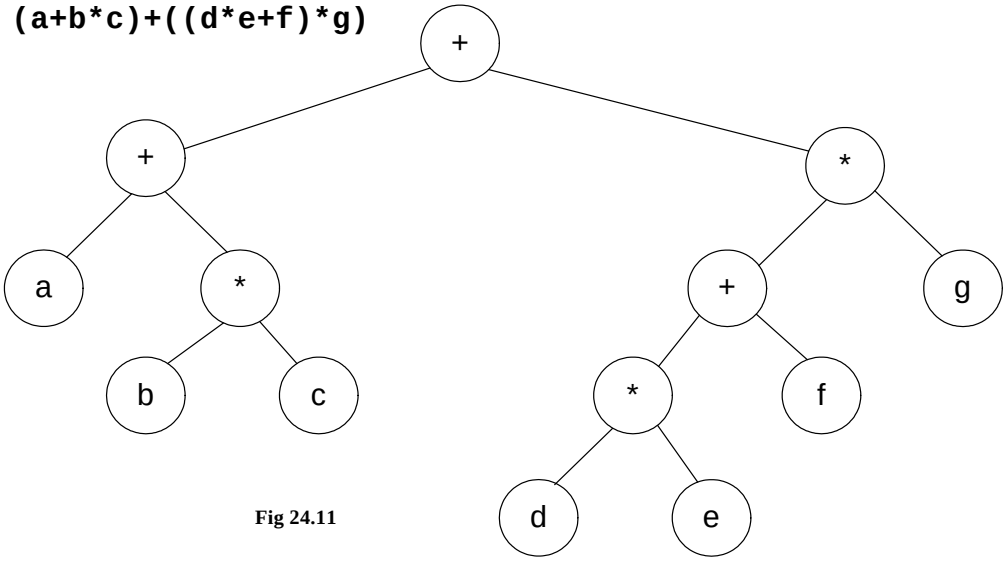
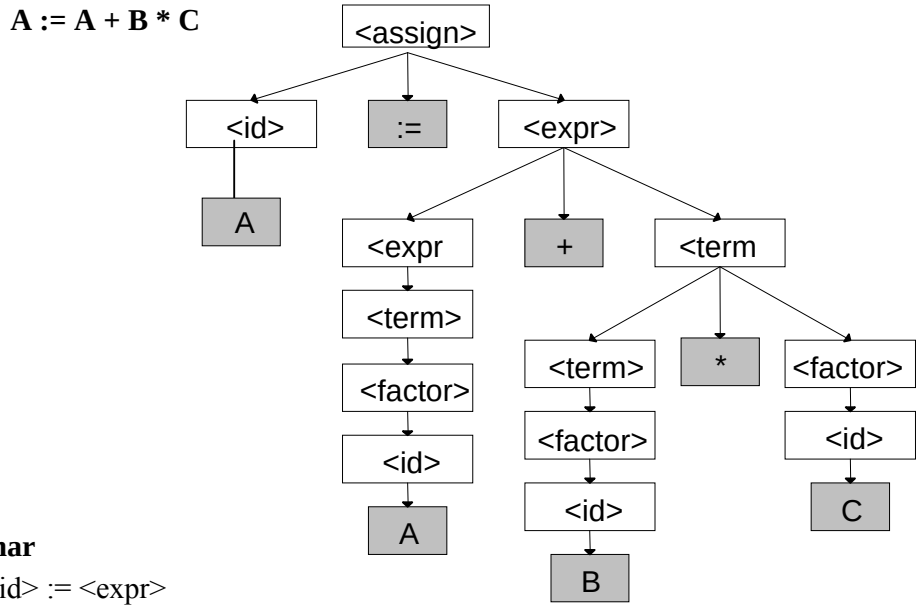


Fig 24.11

You can see the infix expression above (a + b * c) + ((d * e + f) * g), it is represented in the tree form also.
You can see from bottom of the tree that the nodes b and c in the nodes are present at the same level and their parent node is multiplication (*) symbol. From the expression also, we see that the b and c are being multiplied. The parent node of a is + and *right* subtree of + is *b*c***.** You might have understood already that this subtree is depicting *a+b*c*. On the right side, node d and e are connected to the parent *. Symbol + is the parent of * and node f. The expression of the subtree at node + is *d*e+f*. The parent of node + is * node, its *right* subtree is g. So expression of the subtree at this node * is *(d*e+f)*g)*. The root node of the tree is +.
These expression trees are useful in compilers and in spreadsheets also, they are sometimes called parse trees.

Parse Tree in Compilers
See the expression tree of expression A := A + B * C below. We are adding B and C, adding the resultant in A and then finally assigning the resultant to A.



Expression grammar

<assign> → <id> := <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>

Fig 24.12

The *root* node in the parse tree shown above is <assign>.
The assignment statement (<assign>) has three parts. On the left of it, there is always an identifier (single or an array reference) called *l-value*. The *l-value* shown in the tree above is <id> and the identifier is A in the expression above. The second part of assignment statement is assignment operator (= or :=). One the right of assignment operator lies the third part of assignment statement, which is an expression. In the expression A := A + B * C above , the expression after assignment operator is *A + B * C*. In the tree, it is represented by the node <*expr*>. The node <*expr*> has three subnodes: <*expr*>, + and <*term*>. <expr>'s further *left* subtree is <expr>, <term>, <factor>, <id> and then finally is B. The *right* subchild <term> has further subnodes as <term>, * and <factor>. <factor> has <id> as subchild and <id> has C.

Note the nodes in gray shade in the tree above form A = A + B * C.

Compiler creates these parse trees. We will see how to make these trees, when we will parse any language tree like C++. Parsing mean to read and then extract the required structure. A compiler parses a computer language like C++ to form parse trees. Similarly, when we do speech recognition. Each sentence is broken down into a certain structure in form of a tree. Hand writing recognition also involves this. The tablet PCs these days has lot of implementation of parse trees.

Parse Tree for an SQL Query

Let's see another example of parse trees inside databases. The parse trees are used in query processing. The queries are questions to the databases to see a particular kind of data. Consider you have a database for a video store that contains data of movies and artists etc. You are querying that database to find the titles of movies with stars born in 1960. The language used to query from the database is called SQL (Structured Query Language), this language will be dealt in depth in the databases course. The tables lying in this movies database are:

StarsIn(title, year, starName)
MovieStar(name, address, gender, birthdate)

The following SQL query is used to retrieve information from this database:

```
SELECT title
FROM   StarsIn, MovieStar
WHERE starName = name AND birthdate LIKE '%1960' ;
```

This query is asking to retrieve the titles of those movies from *StarsIn* and *MovieStar* tables where the birthdate of an actor is 1960. We see in query processing a tree is formed as shown in the figure below:

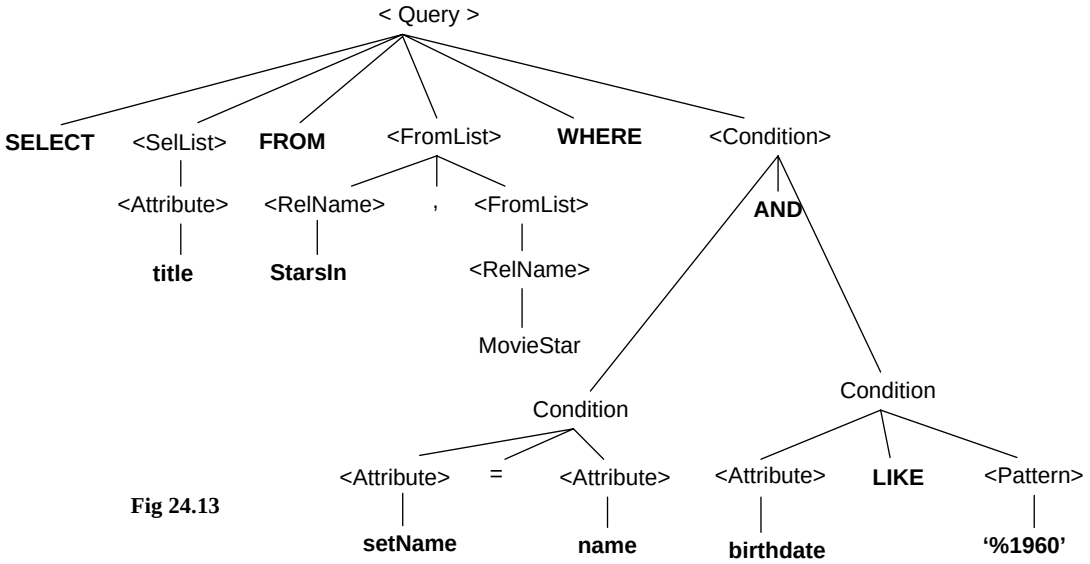


Fig 24.13

The *root* node is Query. This node is further divided into SELECT, <SelList>, FROM, <FromList>, WHERE and <Condition> subnodes. <SelList> will be an *Attribute* and finally a *title* is reached. Observe the tree figure above, how the tree is expanded when we go in the downward direction. When the database engine does the query process, it makes these trees. The database engine also performs query

optimization using these trees.

Compiler Optmization

Let's see another expression tree here:

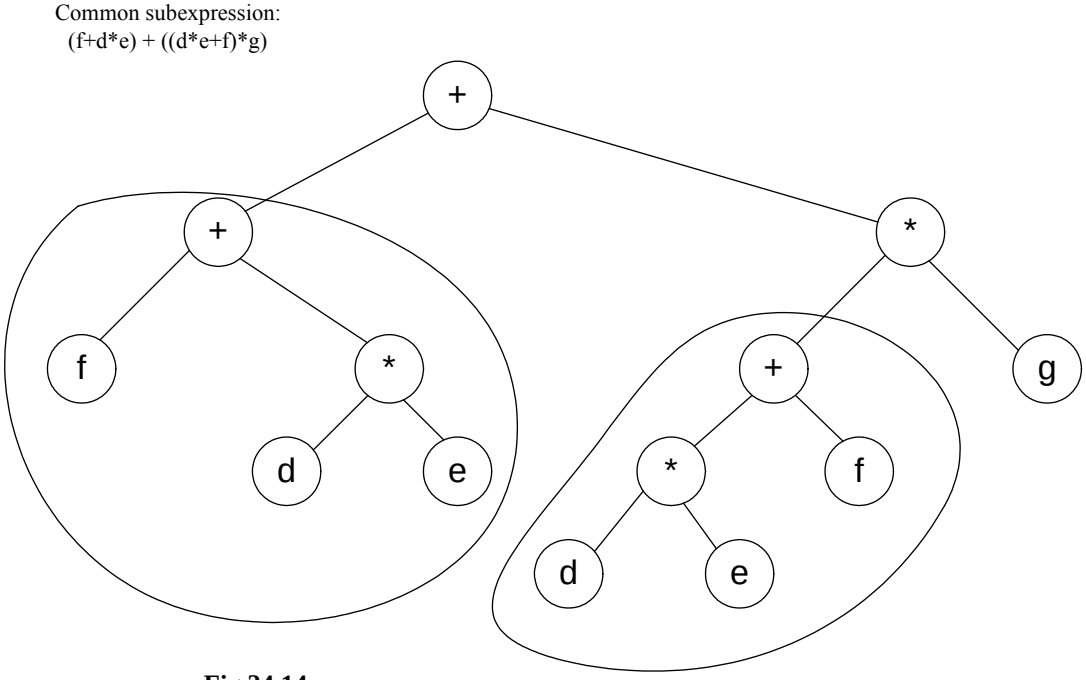


Fig 24.14

The *root* node is +, *left* subtree is capturing the *f+d*e* expression while the right subtree is capturing *(d*e+f)*g*.

Normally compilers has intelligence to look at the common parts inside parse trees. For example in the tree above, the expressions *f+d*e and d*e+f* are same basically. These common subtrees are called *common subexpressions*. To gain efficiency, instead of calculating the *common subexpressions* again, the compilers calculates them once and use them at other places. The part of the compiler that is responsible to do this kind of optimization is called *optimizer*.

See the figure below, the optimizer (part of compiler) will create the following graph while performing the optimization. Because both subtrees were equivalent, it has taken out one subtree and connected the link from node * to node +.

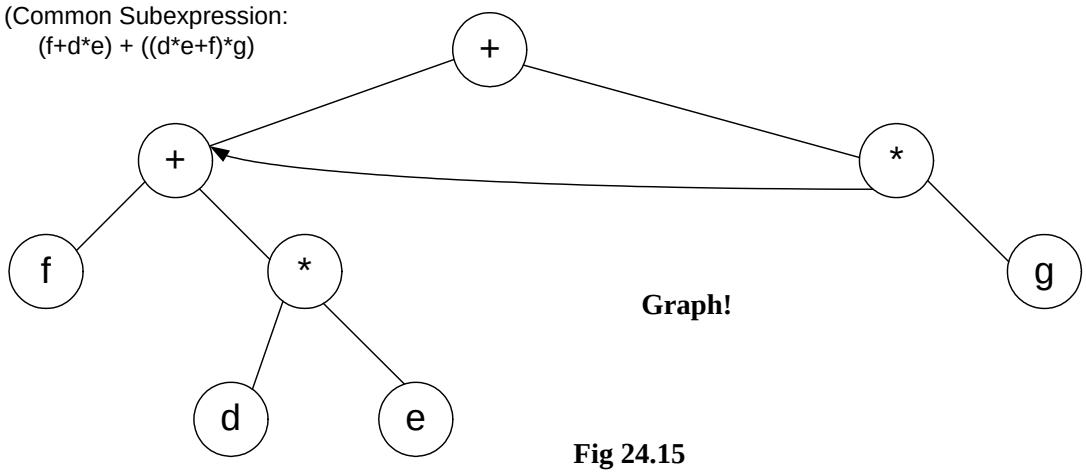


Fig 24.15

This figure is not a tree now because it has two or more different paths to reach a node. Therefore, this has become a graph. The new connection is containing a directed edge, which is there in graphs.

Optimizer uses the expressions trees and converts them to graphs for efficiency purposes. You read out from your book, how the expression trees are formed, what are the different methods of creating them.

Data Structures

Lecture No. 25

Reading Material

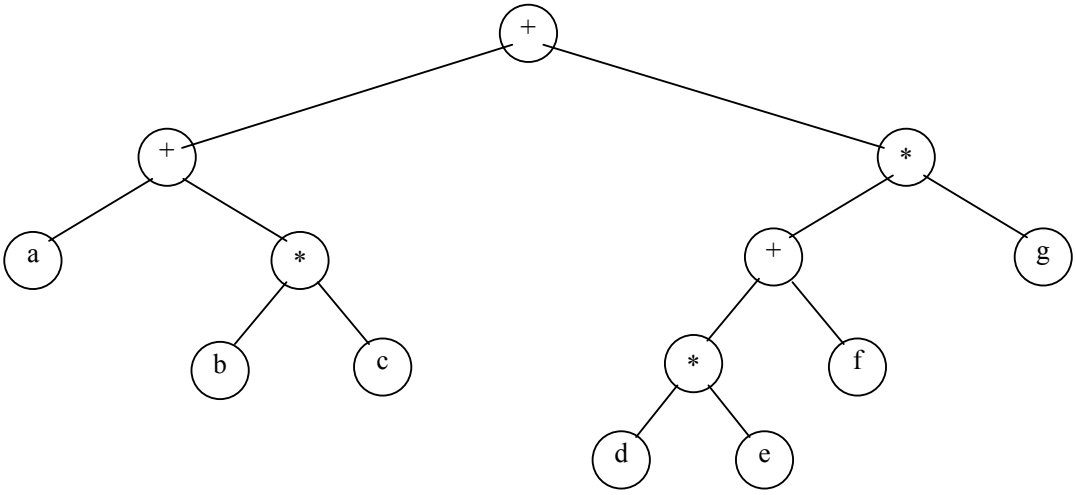
Data Structures and Algorithm Analysis in C++ Chapter. 4, 10
4.2.2, 10.1.2

Summary

- Expression tree
- Huffman Encoding

Expression tree

We discussed the concept of expression trees in detail in the previous lecture. Trees are used in many other ways in the computer science. Compilers and database are two major examples in this regard. In case of compilers, when the languages are translated into machine language, tree-like structures are used. We have also seen an example of expression tree comprising the mathematical expression. Let's have more discussion on the expression trees. We will see what are the benefits of expression trees and how can we build an expression tree. Following is the figure of an expression tree.



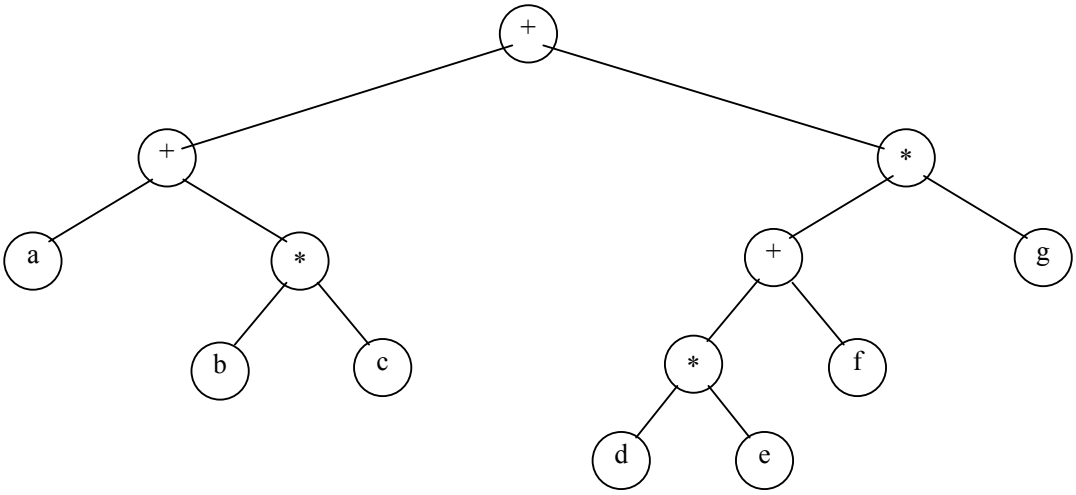
In the above tree, the expression on the left side is *a + b * c* while on the right side, we have *d * e + f * g*. If you look at the figure, it becomes evident that the inner nodes contain operators while leaf nodes have operands. We know that there are two types of nodes in the tree i.e. inner nodes and leaf nodes. The leaf nodes are such nodes which have left and right subtrees as null. You will find these at the bottom level of the tree. The leaf nodes are connected with the inner nodes. So in trees, we have some inner nodes and some leaf nodes.

In the above diagram, all the inner nodes (the nodes which have either left or right

child or both) have operators. In this case, we have + or * as operators. Whereas leaf nodes contain operands only i.e. *a, b, c, d, e, f, g*. This tree is binary as the operators are binary. We have discussed the evaluation of postfix and infix expressions and have seen that the binary operators need two operands. In the infix expressions, one operand is on the left side of the operator and the other is on the right side. Suppose, if we have + operator, it will be written as *2 + 4*. However, in case of multiplication, we will write as *5*6*. We may have unary operators like negation (-) or in Boolean expression we have NOT. In this example, there are all the binary operators. Therefore, this tree is a binary tree. This is not the Binary Search Tree. In BST, the values on the left side of the nodes are smaller and the values on the right side are greater than the node. Therefore, this is not a BST. Here we have an expression tree with no sorting process involved.

This is not necessary that expression tree is always binary tree. Suppose we have a unary operator like negation. In this case, we have a node which has (-) in it and there is only one leaf node under it. It means just negate that operand.

Let's talk about the traversal of the expression tree. The inorder traversal may be executed here.



Inorder traversal yields: a+b*c+d*e+f*g

We use the inorder routine and give the root of this tree. The inorder traversal will be *a+b*c+d*e+f*g*. You might have noted that there is no parenthesis. In such expressions when there is addition and multiplication together, we have to decide which operation should be performed first. At that time, we have talked about the operator precedence. While converting infix expression into postfix expression, we have written a routine, which tells about the precedence of the operators like multiplication has higher precedence than the addition. In case of the expression *2 + 3 * 4*, we first evaluate *3 * 4* before adding *2* to the result. We are used to solve such expressions and know how to evaluate such expressions. But in the computers, we have to set the precedence in our functions.

We have an expression tree and perform inorder traversal on it. We get the infix form of the expression with the inorder traversal but without parenthesis. To have the

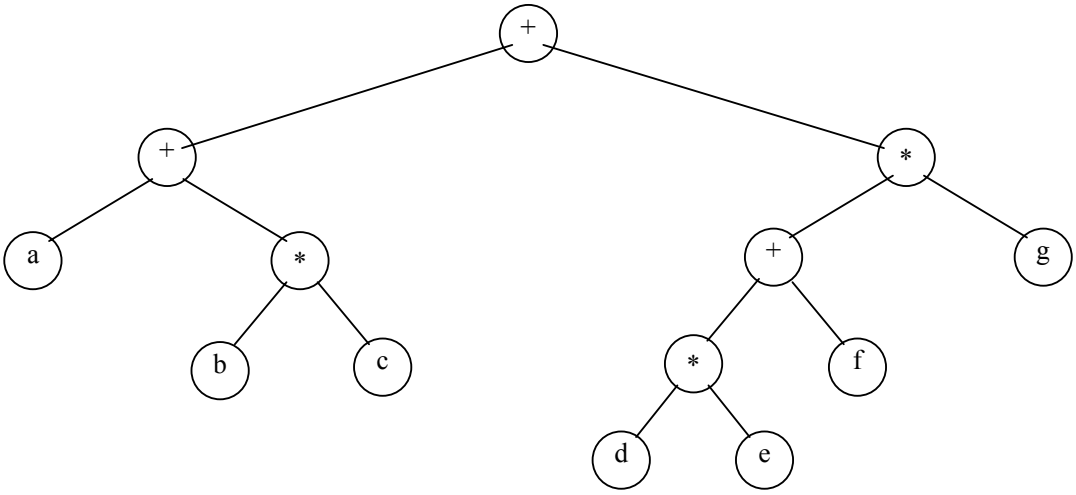
parenthesis also in the expressions, we will have to do some extra work.

Here is the code of the inorder routine which puts parenthesis in the expression.

```
/* inorder traversal routine using the parenthesis */

void inorder(TreeNode<int>* treeNode)
{
   if( treeNode != NULL ){
                cout << "(";
                inorder(treeNode->getLeft());
                cout << ")";
                cout << *(treeNode->getInfo());
                cout << "(";
                inorder(treeNode->getRight());
                cout << ")";
   }
}
```

This is the same inorder routine used by us earlier. It takes the root of the tree to be traversed. First of all, we check that the root node is not null. In the previous routine after the check, we have a recursive call to inorder passing it the left node, print the *info* and then call the inorder for the right node. Here we have included parenthesis using the *cout* statements. We print out the opening parenthesis '(' before the recursive call to inorder. After this, we close the parenthesis. Then we print the *info* of the node and again have opening parenthesis and recursive call to inorder with the right node before having closing parenthesis in the end. You must have understood that we are using the parenthesis in a special order. We want to put the opening parenthesis before the start of the expression or sub expression of the left node. Then we close the parenthesis. So inside the parenthesis, there is a sub expression.

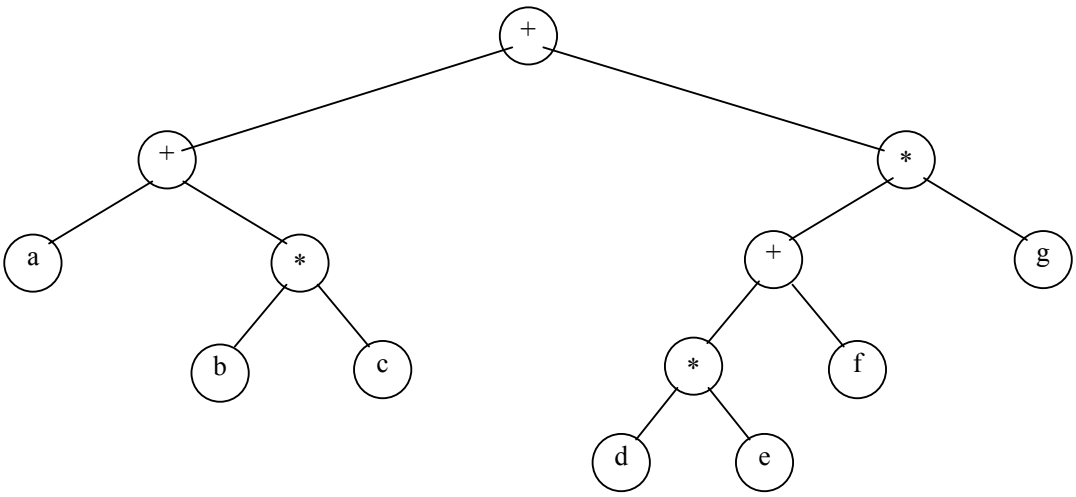
On executing this inorder routine, we have the expression as *(a + (b * c)) + (((d * e) + f) * g)*.



Inorder: *(a + (b * c)) + (((d * e) + f) * g)*

We put an opening parenthesis and start from the root and reach at the node '*a*'. After reaching at plus (+), we have a recursive call for the subtree *. Before this recursive call, there is an opening parenthesis. After the call, we have a closing parenthesis. Therefore the expression becomes as *(a + (b * c))*. Similarly we recursively call the right node of the tree. Whenever, we have a recursive call, there is an opening parenthesis. When the call ends, we have a closing parenthesis. As a result, we have an expression with parenthesis, which saves a programmer from any problem of precedence now.

Here we have used the inorder traversal. If we traverse this tree using the postorder mode, then what expression we will have? As a result of postorder traversal, there will be postorder expression.



Postorder traversal: a b c * + d e * f + g * +

This is the same tree as seen by us earlier. Here we are performing postorder traversal. In the postorder, we print left, right and then the parent. At first, we will print *a*.

Instead of printing +, we will go for *b* and print it. This way, we will get the postorder traversal of this tree and the postfix expression of the left side is *a b c* * + while on the right side, the postfix expression is *d e* * *f* + *g* * +. The complete postfix expression is *a b c* * + *d e* **f* + *g* * +. The expression undergoes an alteration with the change in the traversal order. If we have some expression tree, there may be the infix, prefix and postfix expression just by traversing the same tree. Also note that in the postfix form, we do not need parenthesis.

Let's see how we can build this tree. We have some mathematical expressions while having binary operators. We want to develop an algorithm to convert postfix expression into an expression tree. This means that the expression should be in the postfix form. In the start of this course, we have seen how to covert an infix expression into postfix expression. Suppose someone is using a spreadsheet program and typed a mathematical expression in the infix form. We will first convert the infix expression into the postfix expression before building expression tree with this postfix expression.

We already have an expression to convert an infix expression into postfix. So we get the postfix expression. In the next step, we will read a symbol from the postfix expression. If the symbol is an operand, put it in a tree node and push it on the stack. In the postfix expression, we have either operators or operands. We will start reading the expression from left to right. If the symbol is operand, make a tree node and push it on the stack. How can we make a tree node? Try to memorize the *treeNode* class. We pass it some data and it returns a *treeNode* object. We insert it into the tree. A programmer can also use the *insert* routine to create a tree node and put it in the tree.

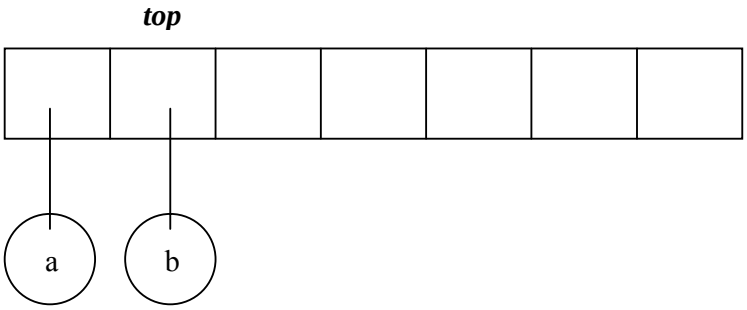
Here we will create a tree node of the operand and push it on the stack. We have been using templates in the stack examples. We have used different data types for stacks like numbers, characters etc. Now we are pushing *treeNode* on the stack. With the help of templates, any kind of data can be pushed on the stack. Here the data type of the stack will be *treeNode*. We will push and pop elements of type *treeNode* on the stack. We will use the same stack routines.

If symbol is an operator, pop two trees from the stack, form a new tree with operator as the root and *T1* and *T2* as left and right subtrees and push this tree on the stack. We are pushing operands on the stacks. After getting an operator, we will pop two operands from the stack. As our operators are binary, so it will be advisable to pop two operands. Now we will link these two nodes with a parent node. Thus, we have the binary operator in the parent node.

Let's see an example to understand it. We have a postfix expression as *a b* + *c d e* + * *. If you are asked to evaluate it, it can be done with the help of old routine. Here we want to build an expression tree with this expression. Suppose that we have an empty stack. We are not concerned with the internal implementation of stack. It may be an array or link list.

First of all, we have the symbol *a* which *is* an operand. We made a tree node and push it on the stack. The next symbol is *b.* We made a tree node and pushed it on the stack. In the below diagram, stack is shown.

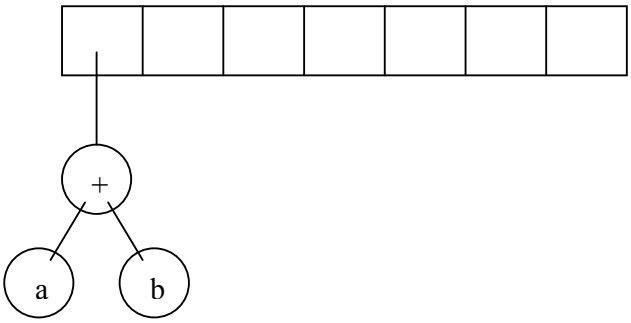
a b + c d e + * *



If symbol is an operand, put it in a one node tree and push it on a stack.

Our stack is growing from left to right. The top is moving towards right. Now we have two nodes in the stack. Go back and read the expression, the next symbol is + which is an operator. When we have an operator, then according to the algorithm, two operands are popped. Therefore we pop two operands from the stack i.e. *a* and *b*. We made a tree node of +. Please note that we are making tree nodes of operands as well as operators. We made the + node parent of the nodes *a* and *b*. The left link of the node + is pointing to *a* while right link is pointing to *b*. We push the + node in the stack.

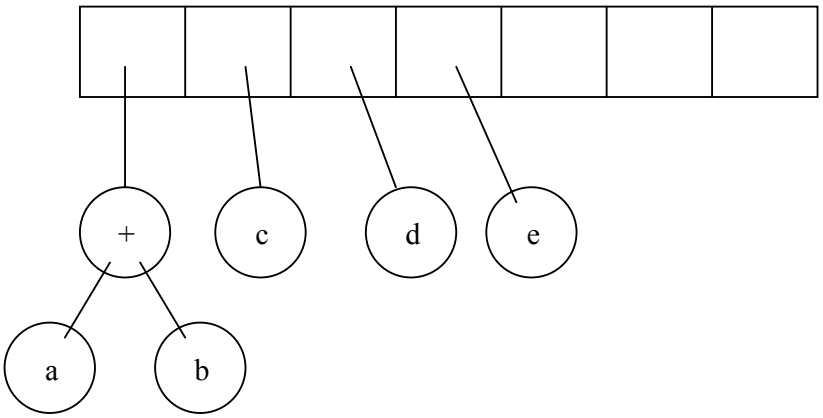
a b + c d e + * *



If symbol is an operator, pop two trees from the stack, form a new tree with operator as the root and T_1 and T_2 as left and right subtrees and push this tree on the stack.

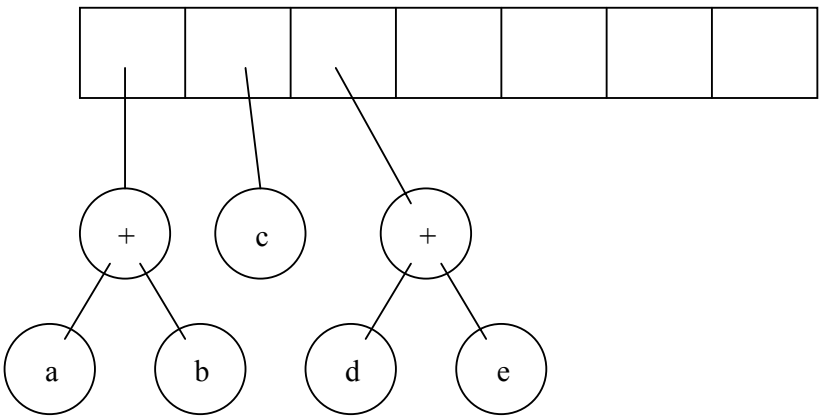
Actually, we push this subtree in the stack. Next three symbols are *c, d,* and *e*. We made three nodes of these and push these on the stack.

a b + c d e + * *



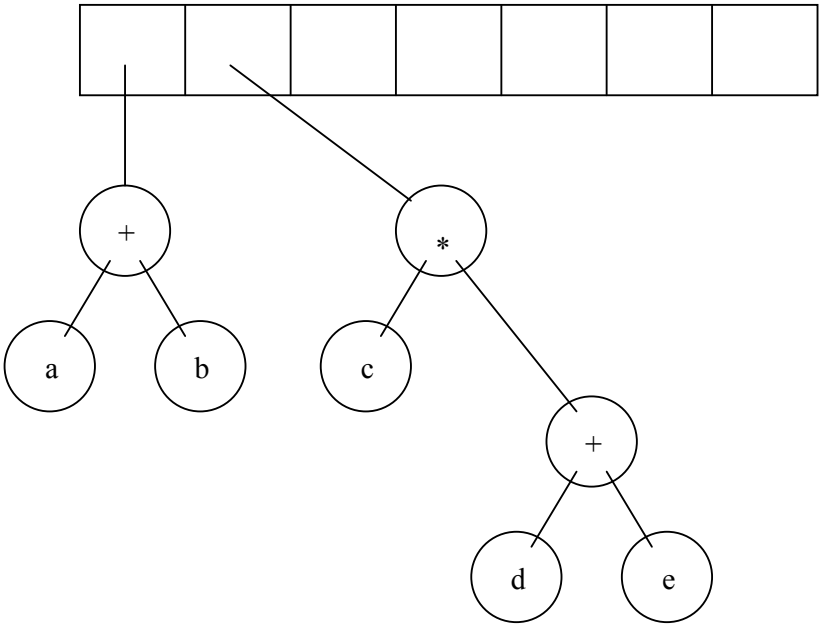
Next we have an operator symbol as +. We popped two elements i.e. *d* and *e* and linked the + node with *d* and *e* before pushing it on the stack. Now we have three nodes in the stack, first + node under which there are *a* and *b*. The second node is *c* while the third node is again + node with *d* and *e* as left and right nodes.

**a b + c d e + * *



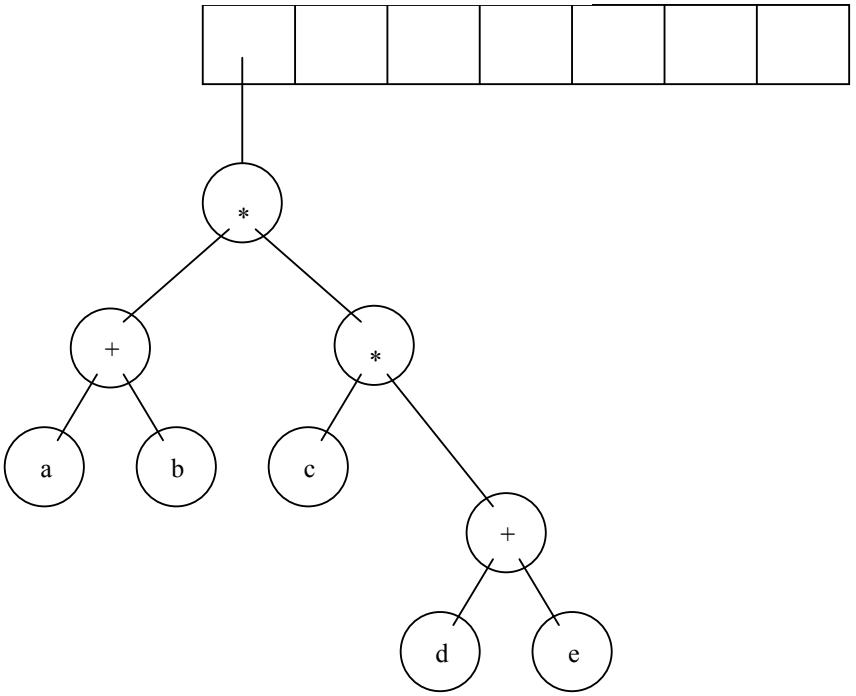
The next symbol is * which is multiplication operator. We popped two nodes i.e. a subtree of + node having *d* and *e* as child nodes and the *c* node. We made a node of * and linked it with the two popped nodes. The node *c* is on the left side of the * node and the node + with subtree is on the right side.

a b + c d e + * *



The last symbol is the * which is an operator. The final shape of the stack is as under:

a b + c d e + * *



In the above figure, there is a complete expression tree. Now try to traverse this tree in the inorder. We will get the infix form which is *a + b * c * d + e*. We don't have parenthesis here but can put these as discussed earlier.

This is the way to build an expression tree. We have used the algorithm to convert the infix form into postfix form. We have also used stack data structure. With the help of templates, we can insert any type of data in the stack. We have used the expression tree algorithm and very easily built the expression tree.

In the computer science, trees like structures are used very often especially in compilers and processing of different languages.

Huffman Encoding

There are other uses of binary trees. One of these is in the compression of data. This is known as Huffman Encoding. Data compression plays a significant role in computer networks. To transmit data to its destination faster, it is necessary to either increase the data rate of the transmission media or simply send less data. The data compression is used in computer networks. To make the computer networks faster, we have two options i.e. one is to somehow increase the data rate of transmission or somehow send the less data. But it does not mean that less information should be sent or transmitted. Information must be complete at any cost.

Suppose you want to send some data to some other computer. We usually compress the file (using winzip) before sending. The receiver of the file decompresses the data before making its use. The other way is to increase the bandwidth. We may want to use the fiber cables or replace the slow modem with a faster one to increase the network speed. This way, we try to change the media of transmission to make the network faster. Now changing the media is the field of electrical or communication engineers. Nowadays, fiber optics is used to increase the transmission rate of data.

With the help of compression utilities, we can compress up to 70% of the data. How can we compress our file without losing the information? Suppose our file is of size 1 Mb and after compression the size will be just 300Kb. If it takes ten minutes to transmit the 1 Mb data, the compressed file will take 3 minutes for its transmission. You have also used the gif images, jpeg images and mpeg movie files. All of these standards are used to compress the data to reduce their transmission time. Compression methods are used for text, images, voice and other types of data.

We will not cover all of these compression algorithms here. You will study about algorithms and compression in the course of algorithm. Here, we will discuss a special compression algorithm that uses binary tree for this purpose. This is very simple and efficient method. This is also used in jpg standard. We use modems to connect to the internet. The modems perform the live data compression. When data comes to the modem, it compresses it and sends to other modem. At different points, compression is automatically performed. Let’s discuss Huffman Encoding algorithm.

Huffman code is method for the compression of standard text documents. It makes use of a binary tree to develop codes of varying lengths for the letters used in the original message. Huffman code is also part of the JPEG image compression scheme. The algorithm was introduced by David Huffman in 1952 as part of a course assignment at MIT.

Now we will see how Huffman Encoding make use of the binary tree. We will take a

simple example to understand it. The example is to encode the 33-character phrase:

"*traversing threaded binary trees*"

In the phrase we have four words including spaces. There is a new line character in the end. The total characters in the phrase are 33 including the space. We have to send this phrase to some other computer. You know that binary codes are used for alphabets and other language characters. For English alphabets, we use ASCII codes. It normally consists of eight bits. We can represent lower case alphabets, upper case alphabets, 0,1,…9 and special symbols like $, ! etc while using ASCII codes. Internally, it consists of eight bits. If you have eight bits, how many different patterns you can be have? You can have 256 different patterns. In English, you have 26 lower case and 26 upper case alphabets. If you have seen the ASCII table, there are some printable characters and some unprintable characters in it. There are some graphic characters also in the ASCII table.

In our example, we have 33 characters. Of these, 29 characters are alphabets, 3 spaces and one new line character. The ASCII code for space is 32 and ASCII code for new line character is 10. The ASCII value for 'a' is 95 and the value of A is 65. How many bits we need to send 33 characters? As every character is of 8 bits, therefore for 33 characters, we need *33 * 8 = 264*. But the use of Huffman algorithm can help send the same message with only 116 bits. So we can save around 40% using the Huffman algorithm.

Let's discuss how the Huffman algorithm works. The algorithm is as:

- List all the letters used, including the "space" character, along with the frequency with which they occur in the message.
- Consider each of these (character, frequency) pairs to be nodes; they are actually leaf nodes, as we will see.
- Pick the two nodes with the lowest frequency. If there is a tie, pick randomly amongst those with equal frequencies.
- Make a new node out of these two, and turn two nodes into its children.
- This new node is assigned the sum of the frequencies of its children.
- Continue the process of combining the two nodes of lowest frequency till the time only one node, the root is left.

Let's apply this algorithm on our sample phrase. In the table below, we have all the characters used in the phrase and their frequencies.

Original text:
traversing threaded binary trees

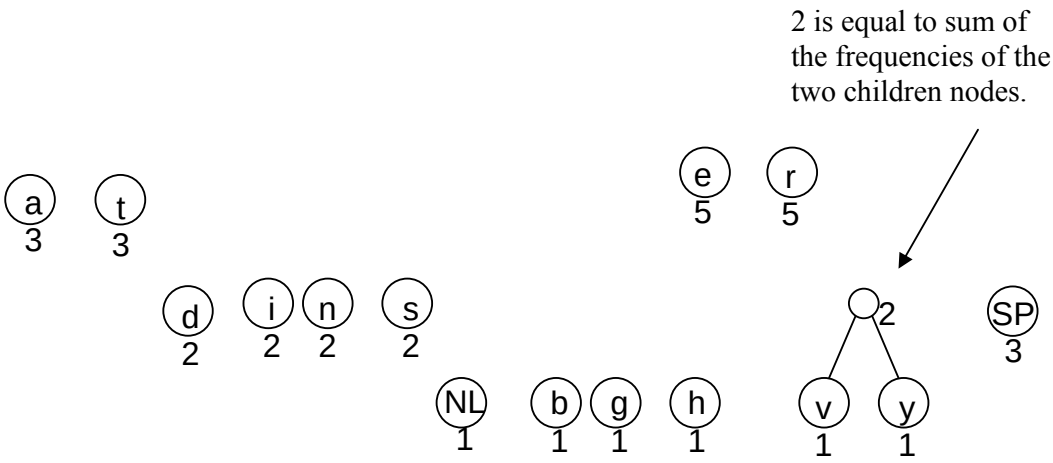
size: 33 characters (space and newline)

Letters	**:**	**Frequency**
NL	:	1
SP	:	3
a	:	3

b	:	1
d	:	2
e	:	5
g	:	1
h	:	1

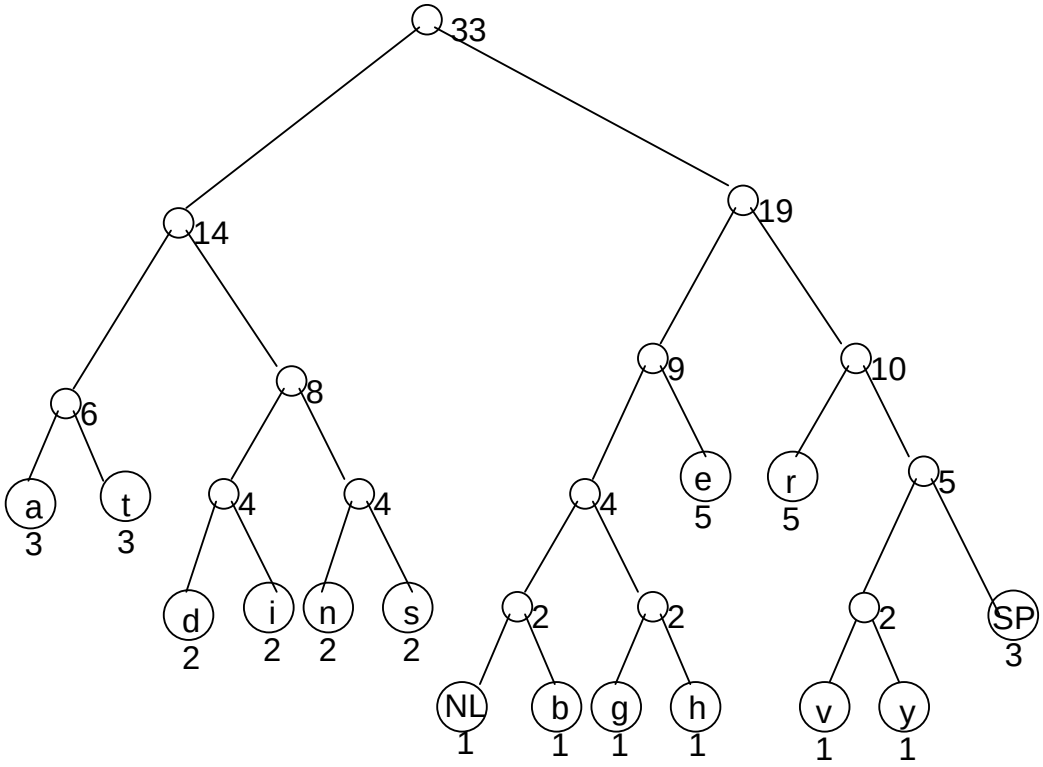
The new line occurs only once and is represented in the above table as NL. Then white space occurs three times. We counted the alphabets that occur in different words. The letter *a* occurs three times, letter *b* occurs just once and so on.

Now we will make tree with these alphabets and their frequencies.



We have created nodes of all the characters and written their frequencies along with the nodes. The letters with less frequency are written little below. We are doing this for the sake of understandings and need not to do this in the programming. Now we will make binary tree with these nodes. We have combined letter *v* and letter *y* nodes with a parent node. The frequency of the parent node is 2. This frequency is calculated by the addition of the frequencies of both the children.

In the next step, we will combine the nodes *g* and *h*. Then the nodes *NL* and *b* are combined. Then the nodes *d* and *i* are combined and the frequency of their parent is the combined frequency of *d* and *i* i.e. 4. Later, *n* and *s* are combined with a parent node. Then we combine nodes *a* and *t* and the frequency of their parent is 6. We continue with the combining of the subtree with nodes *v* and *y* to the node *SP*. This way, different nodes are combined together and their combined frequency becomes the frequency of their parent. With the help of this, subtrees are getting connected to each other. Finally, we get a tree with a root node of frequency 33.



We get a binary tree with character nodes. There are different numbers with these nodes that represent the frequency of the character. So far, we have learnt how to make a tree with the letters and their frequency. In the next lecture, we will discuss the ways to compress the data with the help of Huffman Encoding algorithm.

Data Structures

Lecture No. 26

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4

4.4.2

Summary

- Hoffman Encoding
- Mathematical Properties of Binary Trees

Huffman Encoding

We will continue our discussion on the Huffman encoding in this lecture. In the previous lecture, we talked about the situation where the data structure binary tree was built. Huffman encoding is used in data compression. Compression technique is employed while transferring the data. Suppose there is a word-document (text file) that we want to send on the network. If the file is, say, of one MB, there will be a lot of time required to send this file. However, in case of reduction of size by half through compression, the network transmission time also get halved. After this example, it will be quite easy to understand the Hoffman encoding to compress a text file.

We know that Huffman code is a method for the compression of standard text documents. It makes use of a binary tree to develop codes of varying lengths for the letters used in the original message. Huffman code is also a part of the JPEG image compression scheme. David Huffman introduced this algorithm in the year 1952 as part of a course assignment at MIT.

In the previous lecture, we had started discussing a simple example to understand Huffman encoding. In that example, we were encoding the 32-character phrase: "*traversing threaded binary trees*". If this phrase were sent as a message in a network using standard 8-bit ASCII codes, we would have to send 8*32= 256 bits. However, the Huffman algorithm can help cut down the size of the message to 116 bits.

In the Huffman encoding, following steps are involved:

1. List all the letters used, including the "space" character, along with the frequency with which they occur in the message.
2. Consider each of these (character, frequency) pairs as nodes; these are actually leaf nodes, as we will see later.
3. Pick two nodes with the lowest frequency. If there is a tie, pick randomly amongst those with equal frequencies
4. Make a new node out of these two and develop two nodes as its children.
5. This new node is assigned the sum of the frequencies of its children.
6. Continue the process of combining the two nodes of lowest frequency till the time, only one node, the root, is left.

In the first step, we make a list of all letters (characters) including space and end line character and find out the number of occurrences of each letter/character. For example we ascertain how many times the letter ‘a’ is found in the file and how many times ‘b’ occurs and so on. Thus we find the number of occurrences (i.e. frequency) of each letter in the text file.
In the step 2, we consider the pair (i.e. letter and its frequency) as a node. We will consider these as leaf nodes. Afterwards, we pick two nodes with the lowest frequency in the list. If there are more than one pairs of same frequency, we will choose a pair randomly amongst those with equal frequencies.
Suppose, in a file, the letter ‘a’ occurs 50 times and ‘b’ and ‘c’ five times each. Here, ‘b’ and ‘c’ have the lowest frequency. We will take these two letters as leaf nodes and build the tree from these ones. As fourth step states, we make a new node as the parent of these two nodes. The ‘b’ and ‘c’ are its children. In the fifth step, we assign to this new node the frequency equal to the sum of the frequencies of its children. Thus a three-node tree comes into existence. This is shown in the following figure.

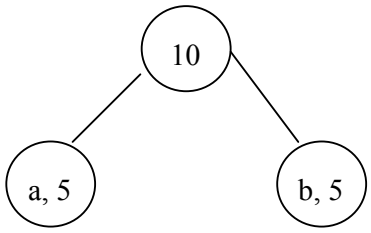


Fig 26.1:

We continue this process of combining the two nodes of lowest frequency till the time, only one node i.e. the root is left.

Now we come back to our example. In this example, there is a text string as written below.

traversing threaded binary trees

The size of this character string is 33 (it includes 3 space characters and one new line character). In the first step, we perform the counting of different characters in the string manually. We do not assign a fake or zero frequency to a letter that is not present in the string. A programmer may be concerned only with the characters/letters that are present in the text. We see that the letters and their frequencies in the above text is as given below.

Character	frequency	character	frequency
NL	1	I	2
SP	3	n	2
A	3	r	5
B	1	s	2
D	2	t	3
E	5	v	3
G	1	y	1
H	1		

Table 1: Frequency table

In the second step, we make nodes of these pairs of letters and frequencies. The following figure (fig 26.2) depicts the letters as nodes. We have written the frequency of each letter with the node. The nodes have been categorized with respect to the frequencies for simplicity. We are going to build the tree from downside i.e. from the lowest frequency.

Now according to third step, two nodes of lowest frequency are picked up. We see that nodes NL, b, g, h, v and y have the frequency 1. We randomly choose the nodes v and y. now, as the fourth step, we make a new node and join the leaf nodes v and y to it as its children. We assign the frequency to this new (parent) node equal to the sum of the frequencies of its children i.e. v and y. Thus in the fifth step; the frequency of this new node is 2. We have written no letter in this node as shown in the figure below.

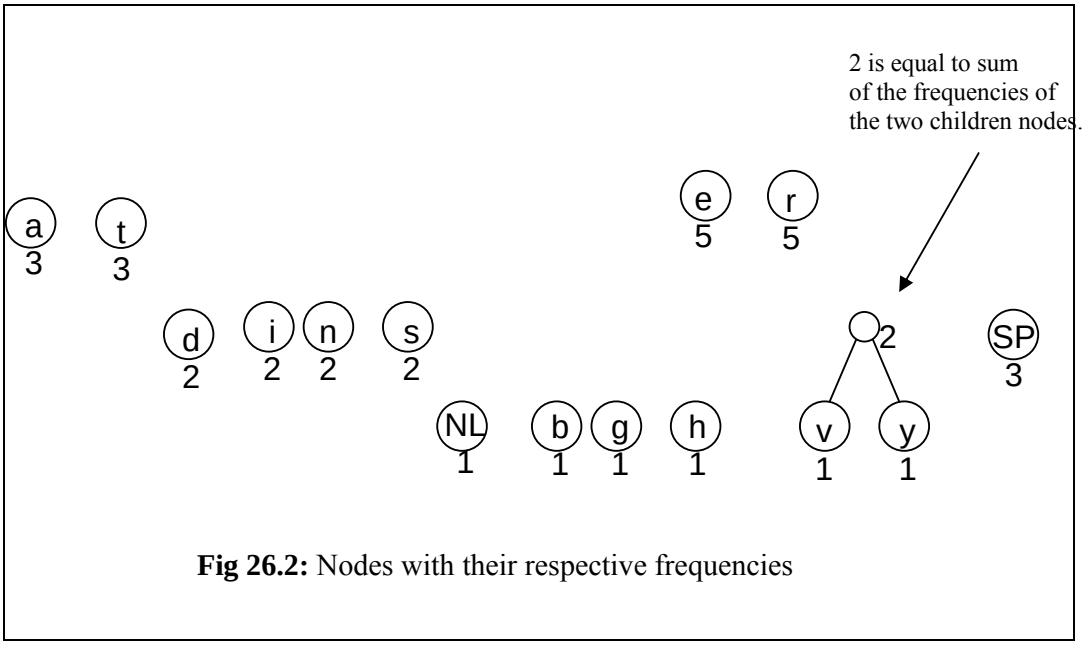


Fig 26.2: Nodes with their respective frequencies

Now we continue this process with other nodes. Now we join the nodes g and h as children of a new node. The frequency of this node is 2 i.e. the sum of frequencies of g and h. After this, we join the nodes NL and b. This also makes a new node of frequency 2. Thus the nodes having frequency 1 have joined to the respective parent nodes. This process is shown in the following figure (Fig 26.3).

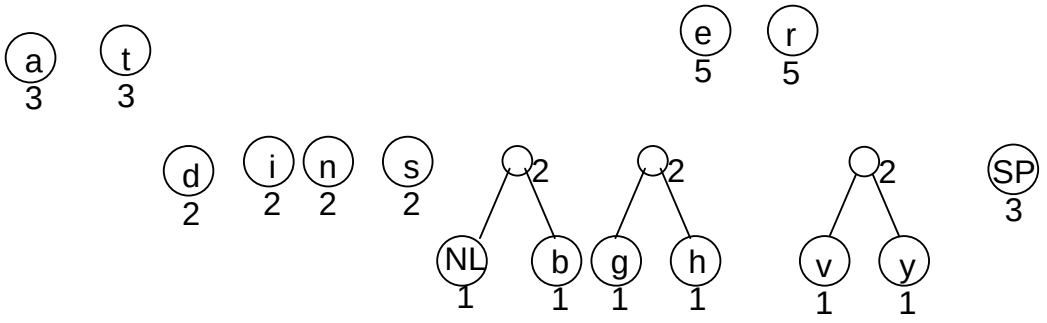


Fig 26.3: joining leaf nodes to create new nodes

Now we come to the nodes with a frequency 2. Here we join the pair of nodes d and i and also the pair n and s. Resultantly, the two nodes coming into being after joining have the frequency 4. This frequency is the sum of the frequencies of their children. Now, we will bring the nodes, a and t together. The parent node of a and t has the frequency 6 i.e. the sum of a and t. These new nodes are shown in the following figure.

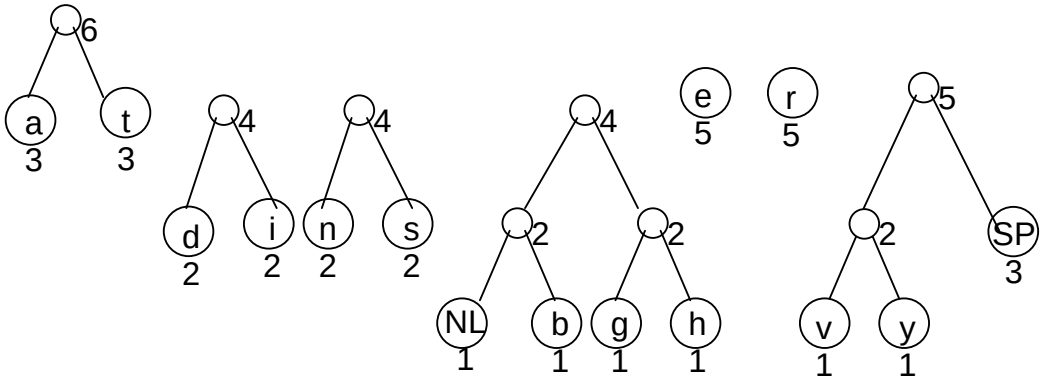


Fig 26.4: joining different nodes to create new nodes

Now we consider these new nodes as inner nodes and build the tree upward towards the root. Now we take the node SP and join it with the node that is the parent of v and y. The resultant node has frequency 5 as the frequencies of its children are 2 and 5 respectively. Now we join these nodes of higher frequencies. In other words, the node r is joined with the newly created node of frequency 5 that is on the left side of node r in the figure 26.5. Thus a new node of frequency 10 is created. We join the node e and the node of frequency 4 on its right side. Resultantly, a new node of frequency 9 comes into existence. Then we join the nodes having frequency 4 and create a new

node of frequency 8. The following figure shows the nodes created so far.

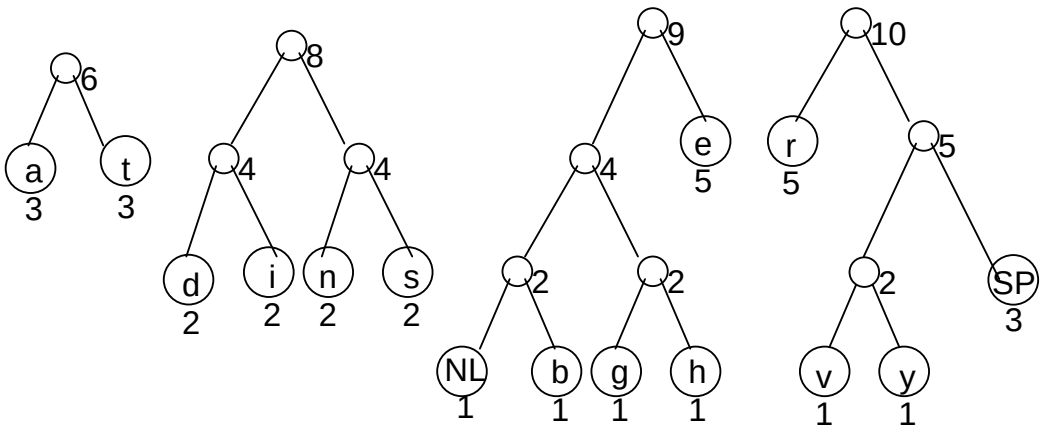


Fig 26.5: joining of nodes to build a tree

Now we will join the nodes of frequency 6 and 8 to create the node of frequency 14 and join the nodes of frequency of 9 and 10 to develop a new node of frequency of 19. At the end, we make the root node with the frequency 33 and it comprises nodes of frequency 14 and 19. Thus the tree is completed and shown in the following figure.

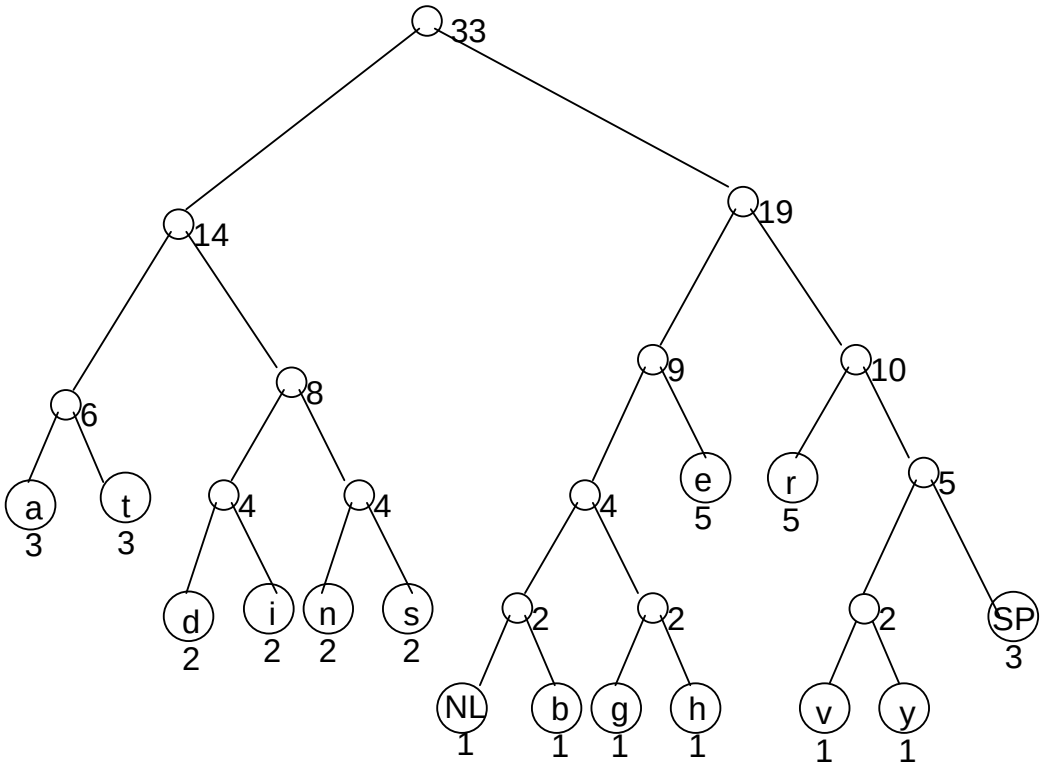


Fig 26.6: Hoffman encoding tree

Now we will perform other steps of Hoffman encoding and develop character-encoding scheme needed for data compression.

To go ahead, we have to do the following steps.

- Start at the root. Assign 0 to left branch and 1 to the right branch.
- Repeat the process down the left and right subtrees.
- To get the code for a character, traverse the tree from the root to the character leaf node and read off the 0 and 1 along the path.

We start from the root node of the tree and assign the value 0 to the left branch and 1 to the right branch. Afterwards, we repeat this value assigning at nodes in left and right subtrees. Thus all the paths have a value 0 or 1 as shown in the following figure. We will develop the code with the help of these path values.

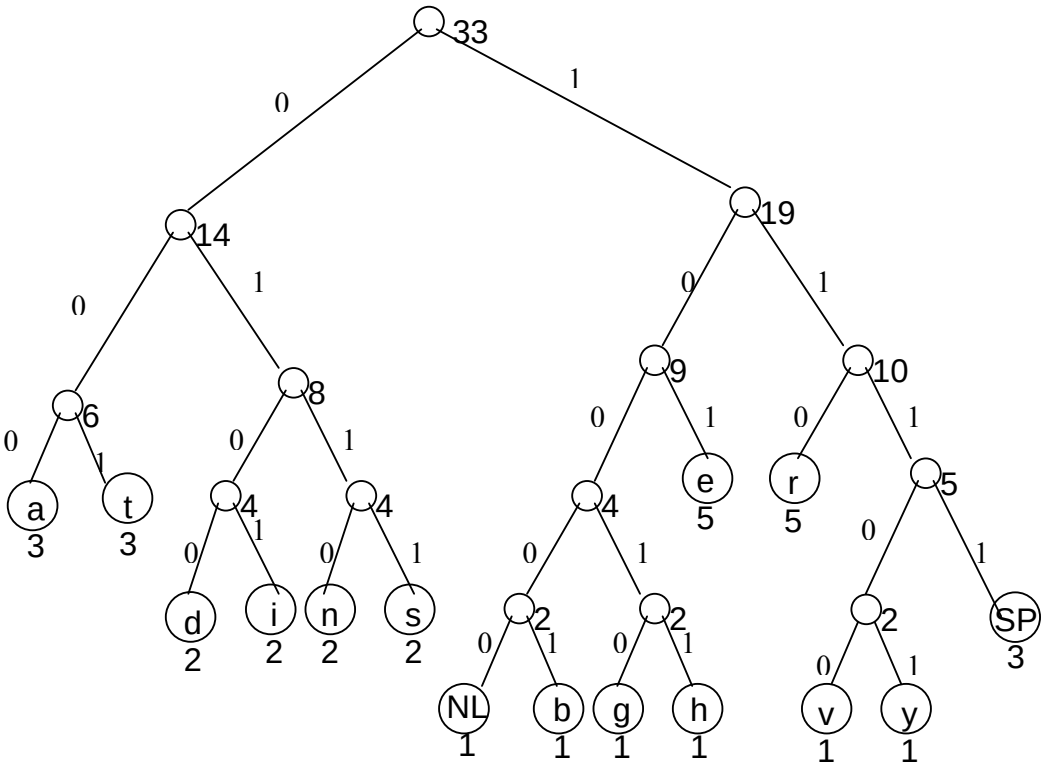


Fig 26.7: Hoffman encoding tree with path values

In the last step, we get the code for the characters. To get the code for a character, there is need of traversing the tree from the root to the character leaf node and read off the 0 and 1 along the path. We start from the root and go to the letter of leaf node following the edges. 0 and 1 are written down in the order in which they come in this traversal to leaf node. For example, we want the code of letter d. To reach d from the root; we have to go through the nodes of frequency 14. This path has value 0. Here, 0 will be the first digit in the code of d. From 14, we go to node of frequency 8. This link (from 14 to 8) has value 1. Thus the second digit in code is 1. From node of frequency 8, we go to node of frequency 4 to its left side. This side has value 0, meaning that 0 is the third digit of code. From 4, we finally go to d and in this link we get the value 0. Thus we see that to reach at d from the root, we have gone through the branches 0, 1, 0 and 0. Thus, the code of letter d is 0100. Similarly the code of i is 0101. The same way, we find the code for each letter in the tree. The following table shows the letters and their correspondent codes.

character	code	character	code
NL	10000	i	0101
SP	1111	n	0110
a	000	r	110

b	10001	s	0111
d	0100	t	001
e	101	v	11100
g	10010	y	11101
h	10011		

Table 2: Hoffman code table

We know that every character is stored in the computer in binary format. Each character has a code, called ASCII code. The ASCII code of a character consists of ones and zeros i.e. it is in binary form. ASCII code is of eight bits. Normally, we remember the decimal value of the characters. For example, letter 'A' has decimal value 65. We can easily convert the decimal value into binary one in bit pattern. We need not to remember these values. ASCII value of any character can be found from the ASCII table. The ASCII code of each character is represented in eight bits with different bit patterns.

Here in the example, we assign a code of our own (i.e. Hoffman code) to the letters that are in the message text. This code also consists of ones and zeros. The above table shows the characters and their Hoffman code. Now we come back to the message text from which we developed the tree and assigned codes to the letters in the text.

Look at the table (Table 2) shown above. Here we notice that the code of letters is of variable length. We see that letters with higher frequency have shorter code. There are some codes with a length five that are the codes of NL, b, g, h, v and y. Similarly we see that the letters SP, d, i, n and s have codes of length four. The codes of the remaining letters have length three. If we look at the frequency table (Table 1) of these letters, we see that the letters with some higher frequency like a, e, r, and t, have the code of shorter length, whereas the letters of lower frequency (like NL, b, g, h, v and y) have codes of larger length.

We see in the table of the Hoffman codes of the letters that there will be need of 5, 4 or in some codes only 3 bits to represent a character, whereas in ASCII code, we need 8 bits for each character. Now we replace the letters in the text message with these codes. In the code format i.e. in the form of ones and zeros, the message becomes as under.

Our original message was

traversing threaded binary trees

The encoded form of the message is as under

```
 t    r    a    v    e    r    s    i    n    g          t
001110000111001011100111010101101001011110010011111010100001001010100111110001010101100001101110111110011101011010111110000
```

We see that there are only ones and zeros in the code of message. Some letters have been shown with their corresponding code. The first three bits 001 are for letter 't'. The next three bits 110 are for letter 'r'. After this the letter 'a' has three bits i.e. 000.

Next to it is the letter ‘v’ that has five bits. These bits are 11100 (shaded in the figure). Similarly we have replaced all the letters of the message with their corresponding Hoffman code. The encoded message is shown above.

Let’s compare this Hoffman encoded message with the encoding of message with ASCII code. The ASCII code encoding means that if we have encoded this message with 8 bits per character, the total length of message would be 264. As there are 33 characters, so the total length is 33 x 8 = 264. But in the Hoffman encoded message (written in above table), there are only 120 bits. This number of bits is 54% less than the number of bits in ASCII encoding form. Thus we can send the message with almost half of the original length to a receiver.

Here the Hoffman encoding process comes to an end. In this process, we took a message, did the frequency count of its characters and built a tree from these frequencies. From this tree, we made codes of letters consisting of ones and zeros only. Then with the help of these codes, we did the data compression. In this process we saw that less data is required for the same message.

Now an important thing to be noted is regarding the tree building. We have built the tree with our choices of nodes with same and different frequencies. Now if we have chosen the nodes to join in different way, the tree built would be in a different form. This results in the different Hoffman code for the letters. These will be in ones and zeros but with different pattern. Thus the encoded message would have different codes for the letters. Now the question arises how does the receiver come to know that what code is used for what letter? The answer is very simple that the sender has to tell the receiver that he is using such codes for letters. One way to tackle this problem is that the sender sends the tree built through the use of frequencies, to the receiver to decode the message. The receiver keeps a copy of this tree. Afterwards, when the sender sends a message, the receiver will match it with respect to bit pattern with the tree to see that what letter the sender has sent. The receiver will find the letter for the first 2, 3, 4 or what numbers of bits match to a letter that it has in the tree as the receiver also has a copy of the tree. We know that a tree has a root, inner nodes and leaf node(s). The leaf node is a node whose left and right links is NULL. An inner node has a left or right or both children. Now consider that the receiver has the same tree data structure that the sender used to construct the codes. The sender has sent the message that we have discussed above. The sender sends the 122 bits encoded message to the receiver. The receiver will take the first bit of message and being on the root of the tree, it will decide that on what side this bit will go. If the bit is zero, it will go to the left of the root. However, if the bit is one, it will go to the right side of the root before reaching to the child node. The next bit will go to the next level of the tree to the left or right side depending on zero or one. Thus the traversal will go one level down on receiving a bit. If we (the receiver) are on a path of the tree where the leaf node is at level 6, it cannot reach the leaf node unless the receiver receives 6 bits. We consider the case of letter ‘e’ whose code was of three bits. In this case, we go to the first level by first bit and the second bit takes us to the third level. Finally on reaching the third level with the third bit, we get the leaf node. We know that this node is letter ‘e’ as the sender has sent us (as receiver) the whole tree with the characters in the nodes. Thus after traversing the three bits, the receiver has confirmed that it has received the letter ‘e’. Now on receiving the fourth bit, the receiver will go back to the root and continue to choose the path i.e. on zero it will go to the left and on one it will go to right. This way, it will reach the leaf node. The character at that node will be the next character of the message. The bit pattern that was comprised of

following these links in the tree is extracted from the message. On the next bit, the receiver again goes to the root node and the previous procedure is repeated to find the next character. This way, it decodes the whole message.

The compression is very useful technique especially for communication purposes. Suppose that we have to send a file of one Mb. Here each line in the file can be compressed to 50 or 60 percent. Thus the file of one MB will be compressed to half MB and can be sent more easily.

There is one more thing about this tree. When we started to build the tree from leaf nodes i.e. bottom-up build of tree, we saw that there were choices for us to choose any two leaf nodes to join. In our example, we chose them at random. The other way to do it is the priority queue. We have seen the example of bank simulation. There we used a priority queue for the events. We know that in a priority queue, the elements do not follow the FIFO (first in first out) rule. But the elements have their position in the queue with respect to a priority. In the example of bank simulation, in the Event Queue, we remove the element from the queue that was going to occur first in the future. We can use priority queue here in such a way that we put the letters in the queue with respect to their frequencies. We will put and remove letters from the queue with respect to their frequency. In this priority queue, the character with lowest frequency is at the start of the queue. If two characters have the same frequency, these will be one after the other in the queue. The character with the larger frequency will be in the last of the queue. Now we take two frequencies from the queue and join them to make a new node. Suppose that the nodes that we joined have frequency 1 and 2 respectively. So the frequency of the new node will be 3. We put this new node in the queue. It takes its position in the queue with respect to its frequency as we are using the priority queue. It is evident in procedure that we proceed to take two nodes form the queue. These are removed and their parent node goes back to the queue with a new frequency. This procedure goes on till the time the queue is empty. The last node that becomes in the queue in the result of this procedure is the root node. This root node has frequency 33 if we apply this procedure to our previous example.

Let's talk about some general things related to Hoffman encoding. We use modems to connect to Internet. These modems do the compression. The modem has a compression feature. The modem has a chip that performs the task of compression. When we give a sentence of say 80 characters to the modem to send, the modem makes a Hoffman encoded tree of these characters. Then it will make codes from this tree. We know that there is also a modem on the other end that will decode this message. The sender modem will send the tree structure to the receiver. Now the sender modem will send the data in compressed form. The receiving modem will decode this compressed data by using the Hoffman encoded tree. Now the question arises, will these codes be useful for other messages? In our example message the letter 'y' has lower frequency and code of five bits. It may happen that in some messages 'y' has higher frequency, needing code of less number of bits. To solve this problem, the modems (sender and receiver) revise the codes after a certain time period or after a particular number of bytes. In this revision, they build a new Hoffman tree and exchange it to use it for communication for the next time period or for the next fixed number of bytes.

There is some other compression techniques/algorithms for compression. The zip routines like winzip and the jpeg, mpeg and other image formatting routines use different algorithms for compression. We will read the compression algorithm in detail in the course of Algorithms.

Mathematical Properties of Binary Trees

There are some mathematical properties of binary trees, which are actually theorems. We will not prove these here. Most of these properties will be studied in some other courses where we will prove them as theorem. Here we are going to talk about some properties, much needed in the next topic about trees.
The first property is that a binary tree of *N* internal nodes has *N+1* external nodes. We are familiar with the term binary tree and internal node. The term external node is a new one. To understand the external nodes, look at the following figure.

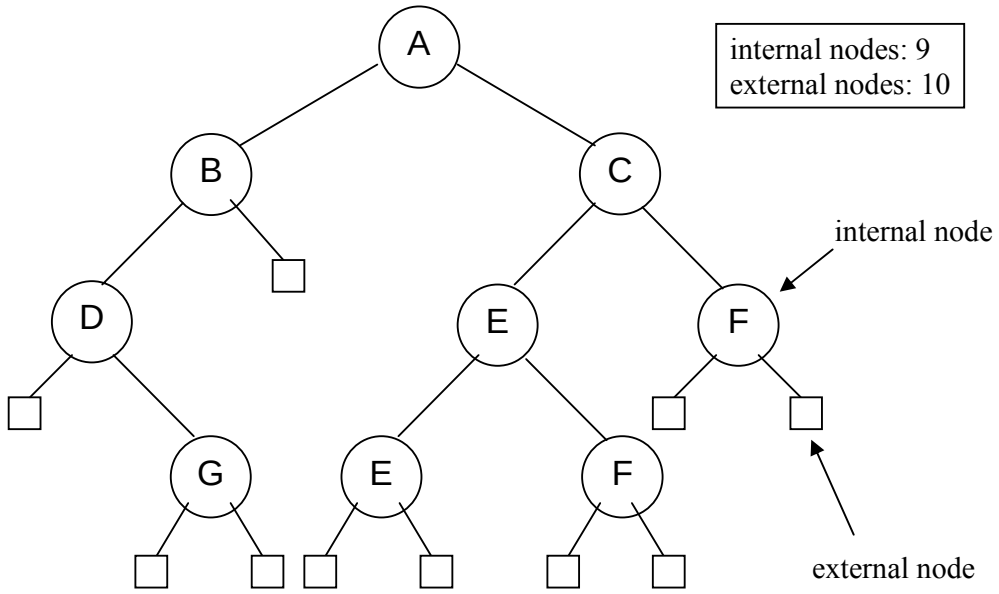


Fig 26.8: Tree with internal and external nodes

In this figure, the nodes with value i.e. A, B, C, D, E, F and G are the internal nodes. Note that the leaf nodes are also included in internal nodes. In the figure, we see that the right pointer of B is NULL. Similarly, the left pointer of D is NULL. The square nodes in the figure are the NULL nodes. There is no data in these nodes as these are NULL pointers. However these are the positions where the nodes can exist. These square nodes are the external nodes. Now we see in the figure that the internal nodes (leaf nodes are also included) are 9 and the external nodes (NULL nodes indicated by squares) are 10 (i.e. 9 + 1). Hence it is the property that we have stated. We will see the usage of this property in the upcoming lectures.

Data Structures

Lecture No. 27

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4
4.3

Summary

- Properties of Binary Tree
- Threaded Binary Trees
- Adding Threads During Insert
- Where is Inorder Successor?
- Inorder Traversal

Properties of Binary Tree

By the end of the last lecture, we were having a discussion about the properties of the binary trees. Let us recall, that I told you about a property of the binary trees regarding relationship between internal nodes and external nodes i.e. If the number of internal nodes is N, the number of external nodes will be N+1. Today I am going to discuss another property of the binary trees, which together with the previous lecture, will give us a start into a new topic. Let me have your attention to the second property of the binary trees.

Property

A binary tree with N internal nodes has 2N links, N-1 links to internal nodes and N+1 links to external nodes.

Please recall that the first property dealt with the relationship between internal and external nodes. This property is dealing with the relationship of links to the internal nodes.

Now, what is a link? As you might already have understood, a link is that line, which we draw between two nodes in a tree. Internally we use pointers in C++ to realize links. In pictorial sketches, however, we use a line to show a link between the two nodes. The property defines, that if you have a binary tree with Nodes, how many links, it will have between the internal nodes (remember, it includes the leaf nodes), and how many links it will have between the external nodes. We have not been showing any links between the external nodes in the diagrams. These are, in fact, null pointers. That means, these are the links, which we will show with the help of the square nodes. Let us see a binary tree, on the basis of which, we will further explore this property. In the following figure, the binary tree is shown again, which, in addition to the normal links between the internal nodes, also contains external nodes as squares and the external links as lines going to those squares.

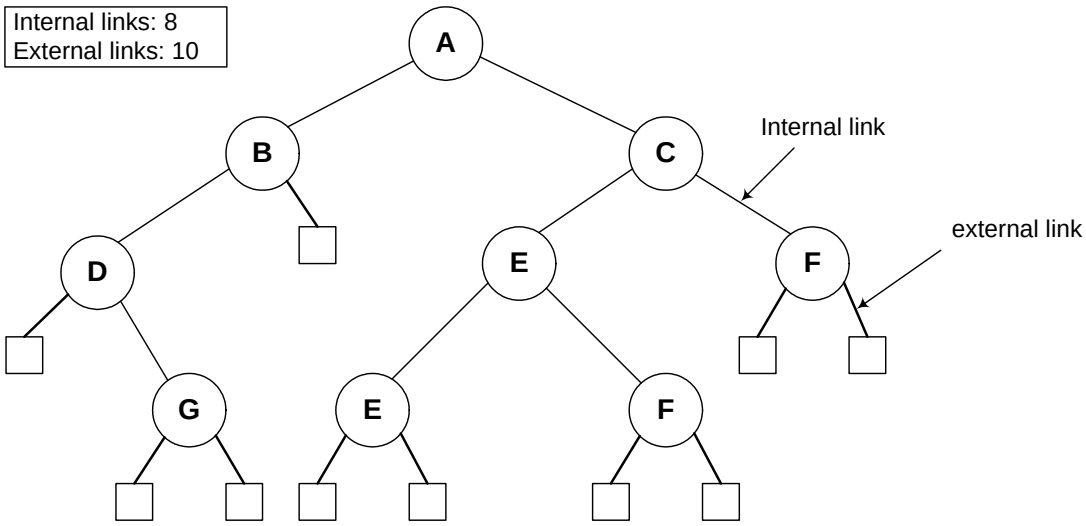


Fig 27.1

Now if you count the total number of links in the diagram between internal and external nodes, it will be 2N. Remember, we are talking about links and not nodes. In this tree, we have 9 nodes marked with capital letters, 8 internal links and 10 external links. Adding the both kinds of links, we get 18, which is exactly 2 x 9.

As discussed already that these properties are mathematical theorems and can therefore be proven mathematically. Let us now prove this property as to how do we get 2N links in a binary tree with N internal nodes.

Property

A binary tree with N internal nodes has 2N links, N-1 links to internal nodes and N+1 links to external nodes.

- In every rooted tree, each node, except the root, has a unique parent.
- *Every link connects a node to its parents, so there are N-1 links connecting internal nodes.*
- *Similarly each of the N+1 external nodes has one link to its parents.*
- *Thus N-1+N+1=2N links.*

In the previous lectures, I told you about the important property of the trees, that they contain only one link between the two nodes. I had also shown you some structures, which did not follow this property and I told you, that those were graphs.

Threaded Binary Trees

- *In many applications binary tree traversals are carried out repeatedly.*
- *The overhead of stack operations during recursive calls can be costly.*
- *The same would true if we use a non-recursive but stack-driven traversal procedure*
- *It would be useful to modify the tree data structure which represents the binary*

tree so as to speed up, say, the inorder traversal process: make it "stack-free".

You must be remembering that there were four traversing methods of binary trees: *preorder*, *inorder*, *postorder* and *levelorder*. First three *preorder*, *inorder* and *postorder* were implemented using recursion. Those recursive routines were very small, 3 to 4 lines of code and they could be employed to traverse a tree of any size.
We also traversed BST in inorder to retrieve the information in sorted order. We employed stacks in recursive implementations. Although, recursive routines are of few lines but when recursion is in action, recursive stack is formed that contains the function calls. We also explicitly used stack for inorder non-recursive traversal. When the calling pattern of recursive and non-recursive stack based routines were compared, the calling pattern of both of the routines were similar.

Suppose that we have a BST that is traversed again and again for some operations of find or print. Due to lot of recursive operations, the stack size keeps on growing. As a result, the performance is affected. To overcome this performance bottleneck, we can use non-recursive method but stack-driven traversal will again be an issue. The *push* and *pop* operations of stack for insertion and retrieval will again take time. So is there a way to do traversal without using a stack neither of implicit function call stack nor explicit. The same idea is presented in the last bullets above that leads to threaded binary trees:

– *It would be useful to modify the tree data structure which represents the binary tree so as to speed up, say, the inorder traversal process: make it "stack-free".*

The idea in the above statement is to modify the tree data structure to speed up and make it stack-free. Now, we see what kind of modification is required in the binary trees.

– *Oddly, most of the pointer fields in our representation of binary trees are NULL!*
– *Since every node (except the root) is pointed to, there are only N-1 non-NULL pointers out of a possible 2N (for an N node tree), so that N+1 pointers are NULL.*

We know that all the leaf node pointers are NULL. Each node of the tree contains the data part, two pointer variables for left and right nodes links. But these pointer variables are used when the node has further child nodes. We know that in a binary tree the total number of links are 2N including both internal and external and the number of NULL pointers is N+1.

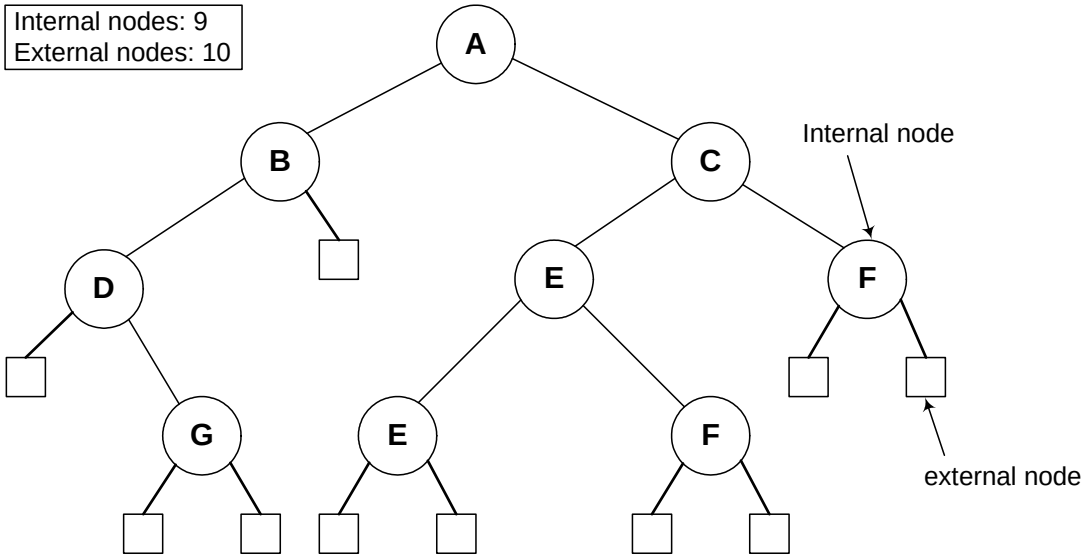


Fig 27.2

In the figure above, the tree is the same as shown in Fig 27.1. The square nodes shown in this figure are external nodes. Thinking in terms of pointers all the pointers of these nodes are NULL or in other words they are available to be used later. We recognize these nodes as *leaf* nodes. Besides that, what can we achieve using them is going to be covered in Threaded Binary Trees.

– *The threaded tree data structure will replace these NULL pointers with pointers to the inorder successor (predecessor) of a node as appropriate.*

We are creating a new data structure inside the tree and when the tree will be constructed, it will be called a threaded binary tree. The NULL pointers are replaced by the inorder successor or predecessor. That means while visiting a node, we can tell which nodes will be printed before and after that node.

– *We'll need to know whenever formerly NULL pointers have been replaced by non NULL pointers to successor/predecessor nodes, since otherwise there's no way to distinguish those pointers from the customary pointers to children.*

This is an important point as we need to modify our previous logic of identifying leaf nodes. Previously the node with left and right nodes as NULL was considered as the leaf node but after this change the leaf node will contain pointers to predecessor and successor. So in order to identify that the pointers has been modified to point to their inorder successor and predecessor, two flags will be required in the node. One flag will be used for successor and other for predecessor. If both the pointers were NULL, left pointer variable will be used to point inorder predecessor, the flag for this will be turned on and the right pointer variable will be used to keep inorder successor and the flag will be turned on once the successor address is assigned.

Adding Threads During Insert

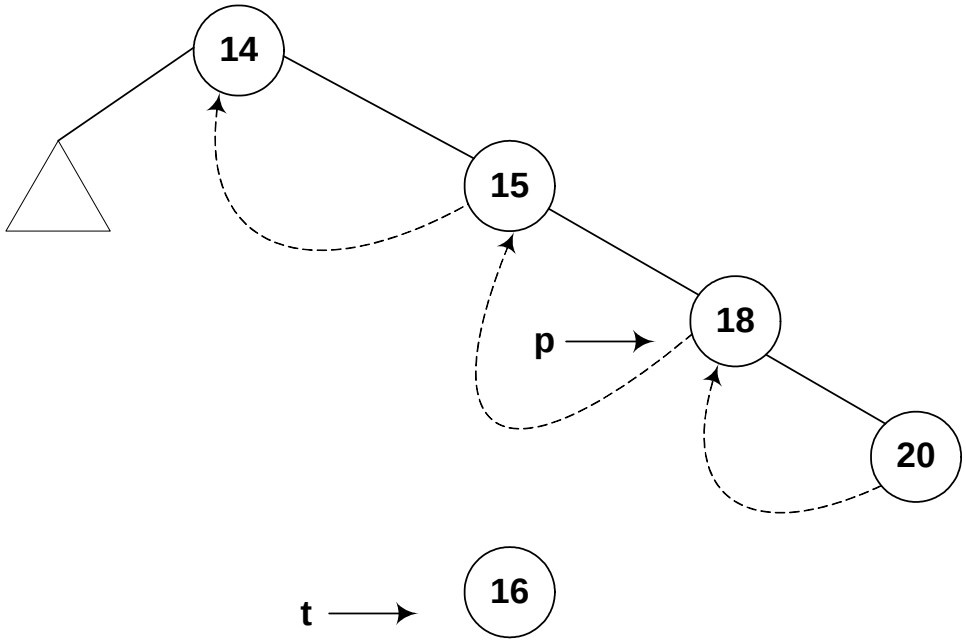


Fig 27.3

If we print the above tree in inorder we will get the following output:
14 15 18 20
In the above figure, the node 14 contains both left and right links. The left pointer is pointing to a subtree while the right subtree is pointing to the node 15. The node 15's right link is towards 18 but the left link is NULL but we have indicated it with a rounded dotted line towards 14. This indicates that the left pointer points to the predecessor of the node.
Below is the code snippet for this logic.

```
t->L = p->L; // copy the thread
t->LTH = thread;
t->R = p;  // *p is successor of *t
t->RTH = thread; p->L = t; // attach the new leaf
p->LTH = child;
```

Let's insert a new node in the tree shown in the above figure. The Fig 27.4 indicates this new insertion.

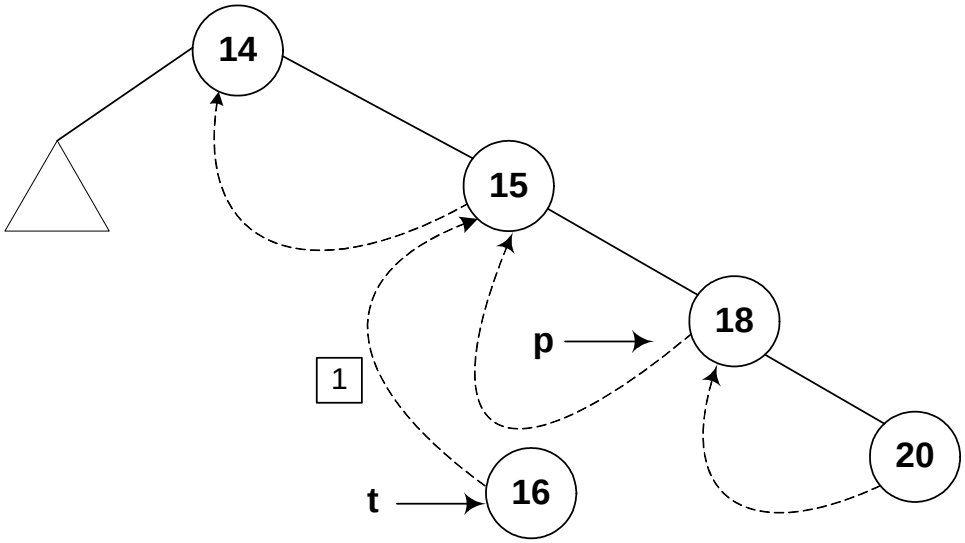


Fig 27.4

The new node 16 is shown in the tree. The left and right pointers of this new node are NULL. As node 16 has been created, it should be pointed to by some variable. The name of that variable is t. Next, we see the location in the tree where this new node with number 16 can be inserted. Clearly this will be after the node 15 but before node 18. As a first step to insert this node in the tree as the left child of the node 18, we did the following:

1. t->L = p->L; // copy the thread
2. t->LTH = thread;
3. t->R = p; // *p is successor of *t
4. t->RTH = thread;
5. p->L = t; // attach the new leaf
6. p->LTH = child;

As the current predecessor of node 18 is 15. After node 16 will be inserted in the tree, it will become the inorder predecessor of 18, therefore, in the first line of the code *t->L = p->L*, left pointer of node 18 (pointed to by pointer p) is assigned to the left pointer of node 16 (pointer to by pointer t).

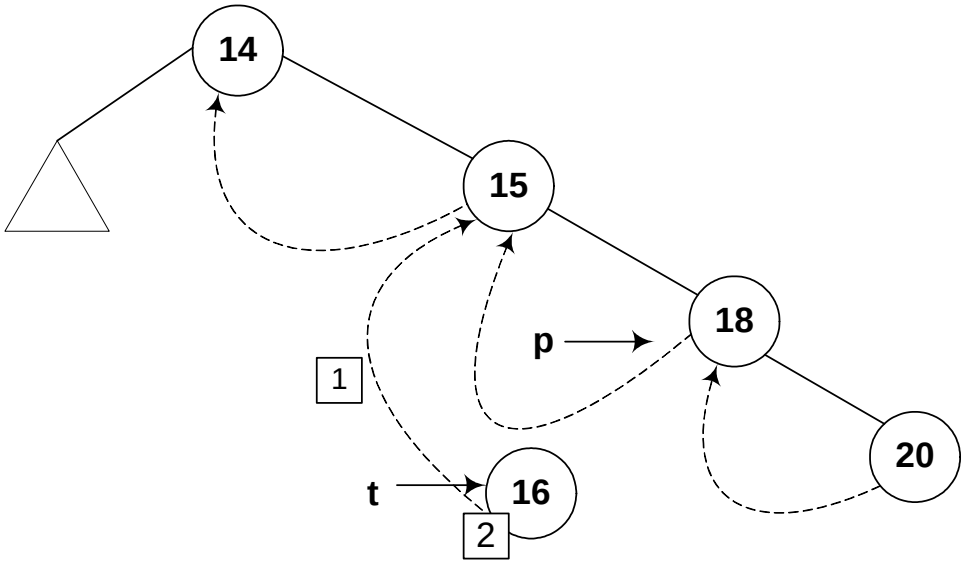


Fig 27.5

In the next line of code *t->LTH = thread*, the left flag is assigned a variable *thread* that is used to indicate that it is on.

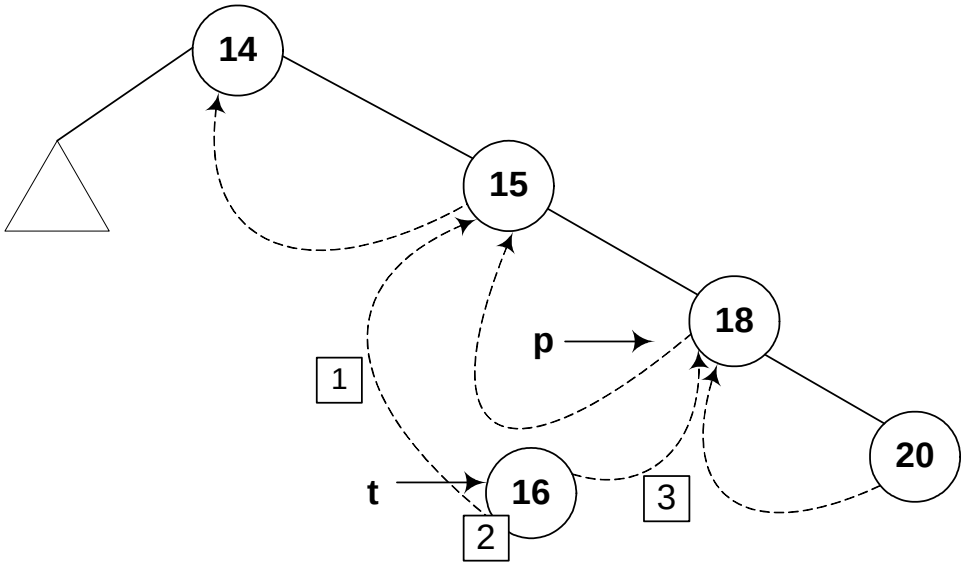


Fig 27.6

In the third line of code, *t->R = p*, 18 being the successor of node 18, its pointer p is assigned to the right pointer (t->R) of node 16.

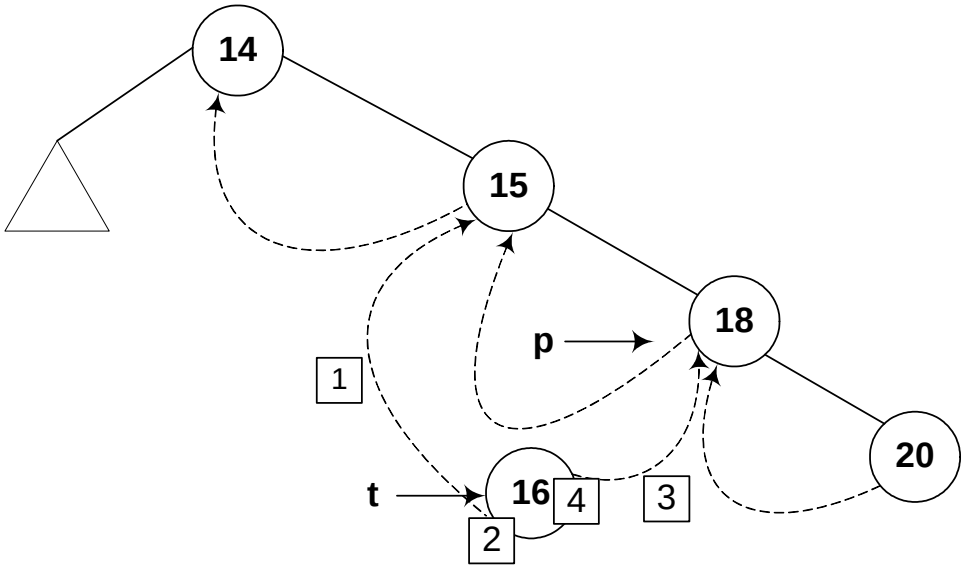


Fig 27.7

Next line, *t->RTH = thread* contains flag turning on code.

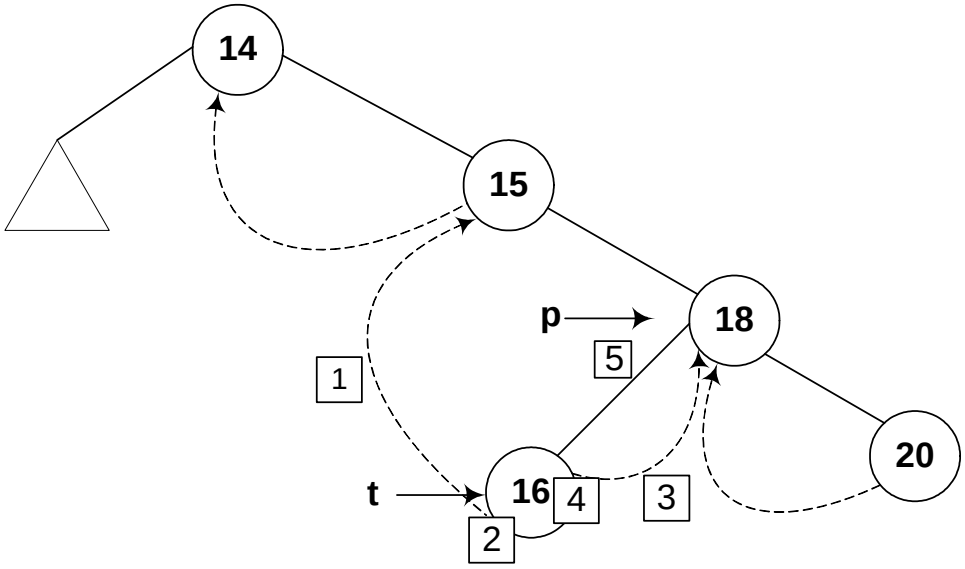


Fig 27.8

In the next line *p->L = t*, the node 16 is attached as the left child of the node 18.

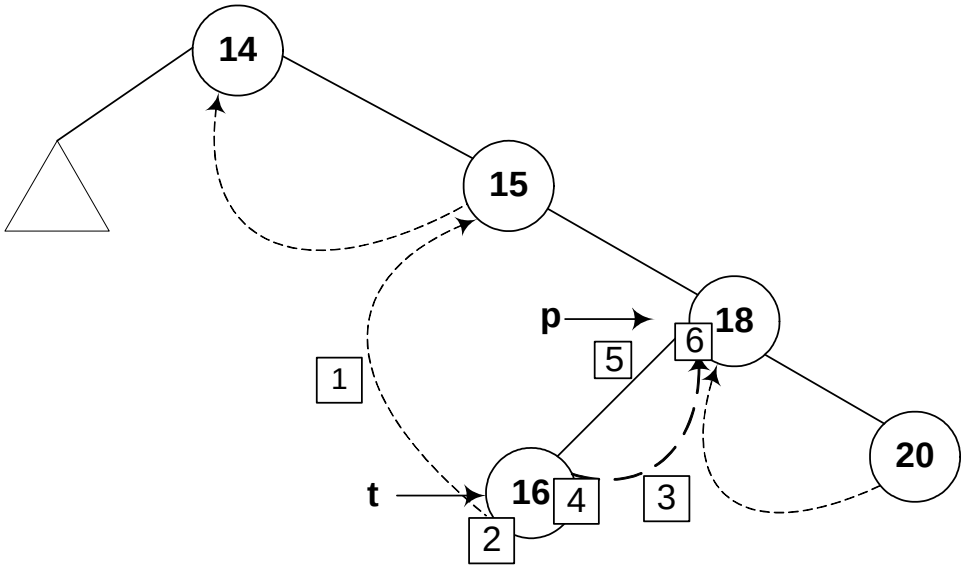


Fig 27.9

The flag is truned on in the last line, *p->LTH = child*.

If we insert few more nodes in the tree, we have the tree as given below:

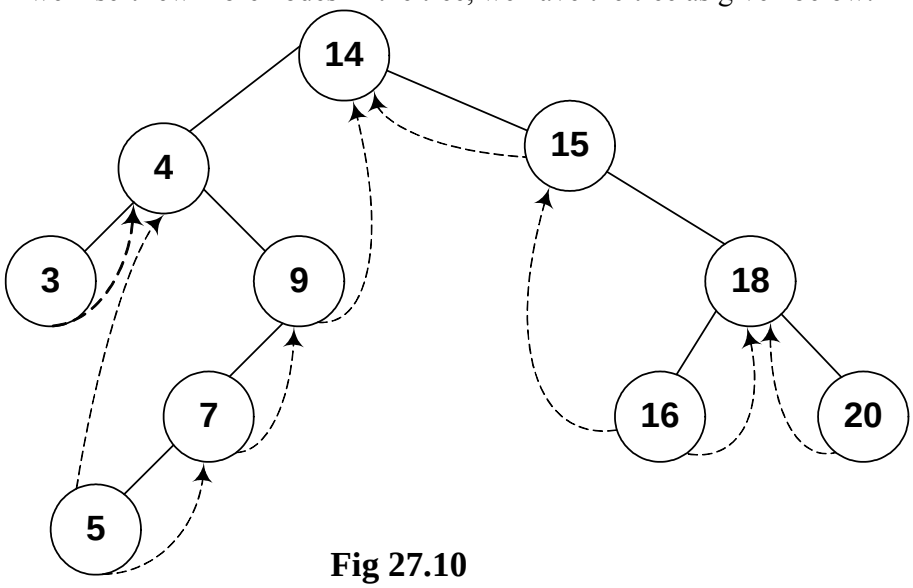


Fig 27.10

Above given is a BST and you have seen many BSTs before, which are not thread binary trees. Without the threads, it is clear from the figure that there are number of links present in the tree that are NULL. We have converted the NULLs to threads in this tree.

Let's do inorder non-recursive traversal of the tree. We started at 14 then following the left link came to 4 and after it to 3. If we use recursion then after the call for node 3 is finished (after printing 3), it returns to node 4 call and then 4 is printed using the recursive call stack. Here we will print 3 but will not stop. As we have used threads, we see the right pointer of node 3 that is not NULL and pointing to its successor node

4, we go to 4 and print it. Now which node is inorder successor of node 4. It is node 5. From node 4, we traversed to right child of it node 9. From node 9, we went to node 7 and then finally node 5. Now, this node 5 is a leaf node. Previously, without using threads, we could identify leaf nodes, whose both pointers left and right were NULL. In this case, using threads, as discussed above, we set the pointers and turn the flags on when a pointer left or right is set to its predecessor or successor. After printing node 5, we traverse its right thread and go to node 7. In this fashion, whole of the tree can be traversed without recursion.

Now, let's see some code:

```
TreeNode*  nextInorder(TreeNode*  p)
{
        if(p->RTH  ==  thread)
                return(p->R);
        else {
                p  =  p->R;
                while(p->LTH  ==  child)
                        p  =  p->L;
                return  p;
        }
}
```

Above given is a routine *nextInorder*, which gives the inorder successor of a node passed in parameter pointer *p*. Now what it does is:
If the *RTH* flag of the *p* node (the node passed in the parameter) is *thread* then it will return the node being pointed by the right thread (*p->R*), which would be its inorder successor. Otherwise, it does the following.
It goes to the right node of *p* and starts pointing it using the same *p* pointer. From there it keeps on moving (in a loop fashion) towards the left of the node as long as the statement *p->LTH == child* is true. After this loop is terminated, the node p is returned.

Next, we see this pictorially.

Where is Inorder Successor?

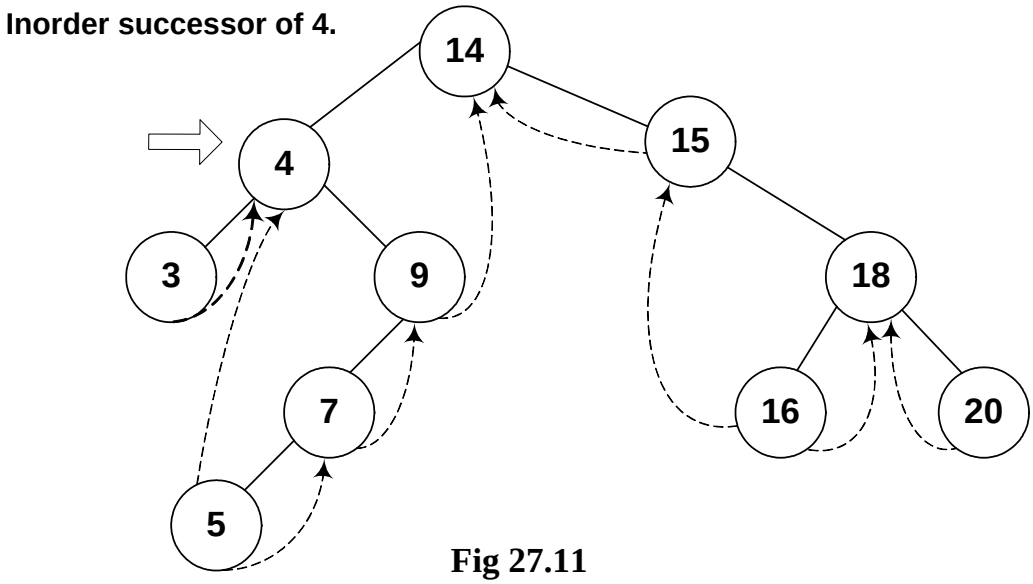


Fig 27.11

We are at node 4 and want to find its inorder successor. If you remember the *delete* operation discussed in the previous lecture, where we searched for the inorder successor and found it to be the *left-most node in the right subtree* of the node.

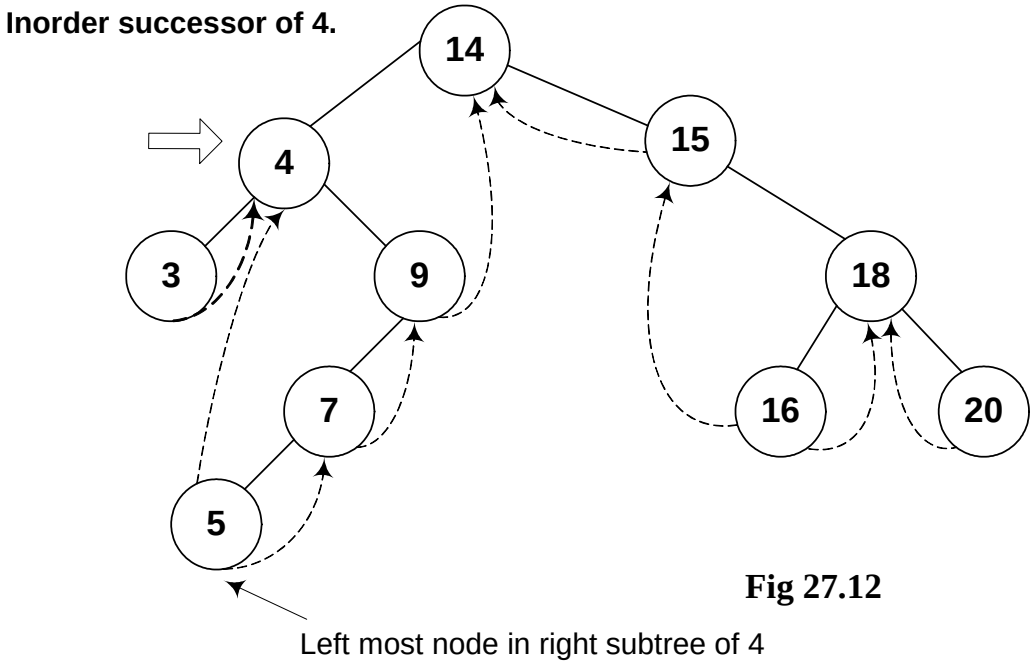


Fig 27.12

In this figure, the right subtree of 4 is starting from node 9 and ending at node 5. Node 5 is the left most node of it and this is also the inorder successor of node 4. We cannot go to node 5 directly from node 4, we go to node 9 first then node 7 and finally to node 5.

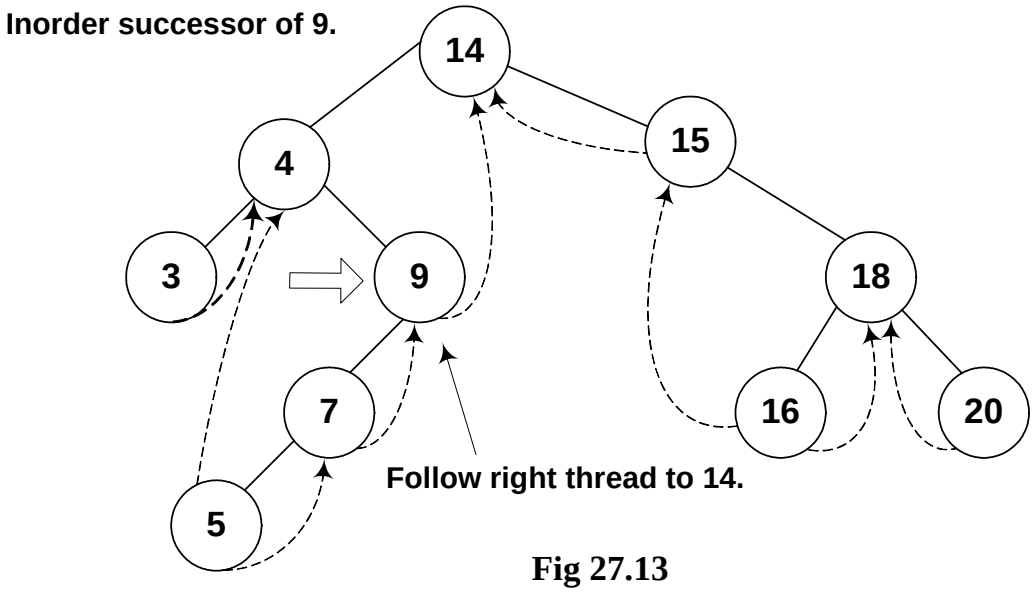


Fig 27.13

We move from node 9 to node 5 following the normal tree link and not thread. As long as the normal left tree link is there of a node, we have set the *LTH* flag to *child*. When we reach at node 5, the left link is a thread and it is indicated with a flag. See the while loop given in the above routine again:

```
while(p->LTH == child)
        p = p->L;
return p;
```

Inorder Traversal

Now by using this routine, we try to make our inorder traversal procedure that is non-recursive and totally stack free.

– *If we can get things started correctly, we can simply call nextInorder repeatedly (in a simple loop) and move rapidly around the tree inorder printing node labels (say) - without a stack.*

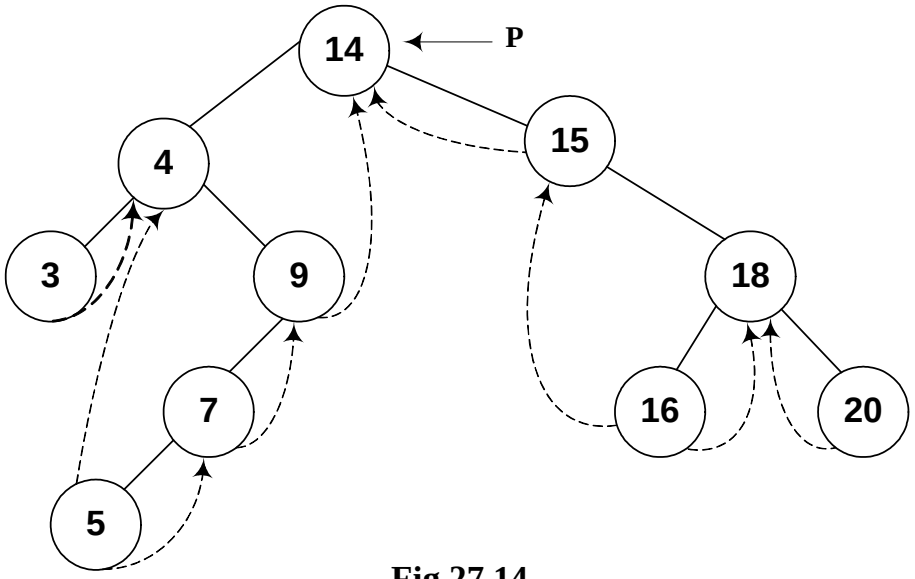


Fig 27.14

The pointer *p* is pointing to the *root* node of the tree. If we start traversing from this node and pass this *root* node pointer to our routine *nexInorder* above, it will create a problem.

We see the routine again to see the problem area clearly:

```
TreeNode*  nextInorder(TreeNode*  p)
{
        if(p->RTH  ==  thread)
                return(p->R);
        else {
                p  =  p->R;
                while(p->LTH  ==  child)
                        p  =  p->L;
                return  p;
        }
}
```

In the first part, it is checking for the RTH flag to be set to *thread*, which is not the case for the root node. The control will be passed to the else part of the routine. In else part, in the very first step, we are moving towards right of root that is to node 15.

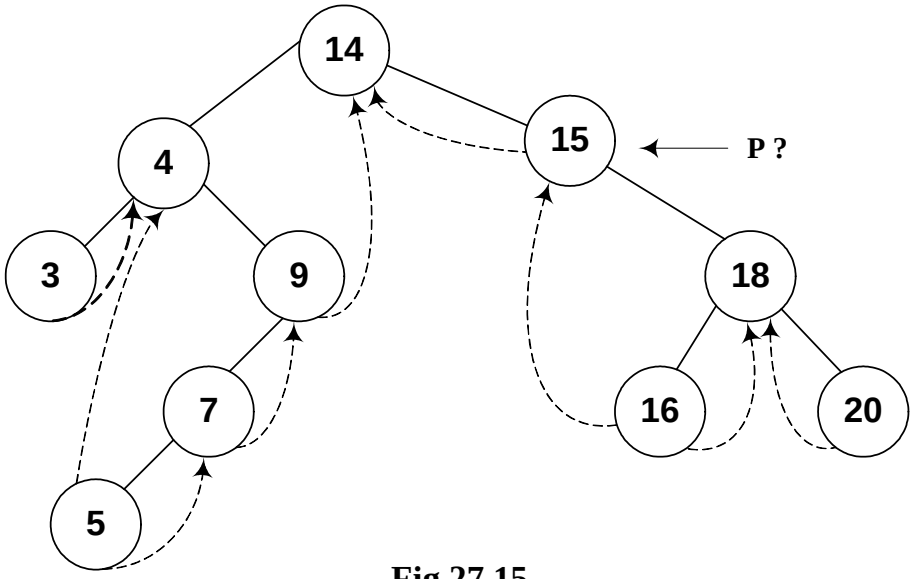


Fig 27.15

– *If we call nextInorder with the root of the binary tree, we're going to have some difficulty. The code won't work at all the way we want.*

Note that in this tree inorder traversal, the first number we should print is 3 but now we have reached to node 15 and we don't know how can we reach node 3. This has created a problem. In the lecture, we will make a small change in this routine to cater to this situation i.e. when a *root* node pointer is passed to it as a parameter. After that change the routine will work properly in case of *root* node also and it will be non-recursive, stack free routine to traverse the tree.

Data Structures

Lecture No. 28

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 6
6.3.1

Summary

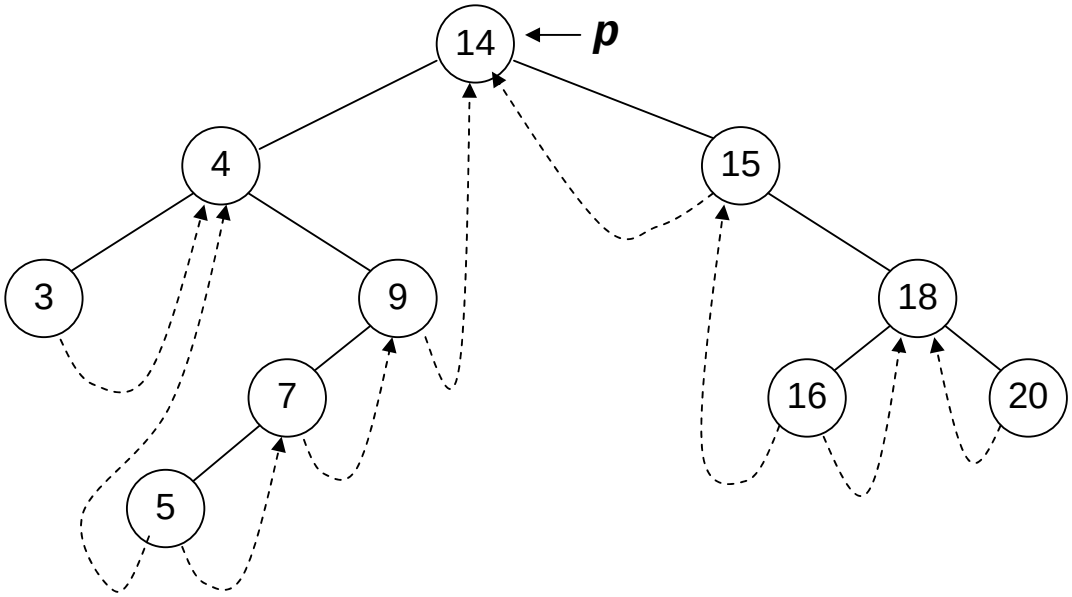
- Inorder traversal in threaded trees
- Complete Binary Tree

Inorder traversal in threaded trees

Discussion on the inroder traversal of the threaded binary tree will continue in this lecture. We have introduced the threads in the tree and have written the *nextInorder* routine. It is sure that the provision of the root can help this routine perform the inorder routine properly. It will go to the left most node before following the threads to find the inorder successors. The code of the routine is given below:

```
/*   The inorder routine for threaded binary tree   */

TreeNode* nextInorder(TreeNode* p){

  if(p->RTH == thread) return(p->R);
  else {
       p = p->R;
       while(p->LTH == child)
           p = p->L;
      return p;
  }
}
```

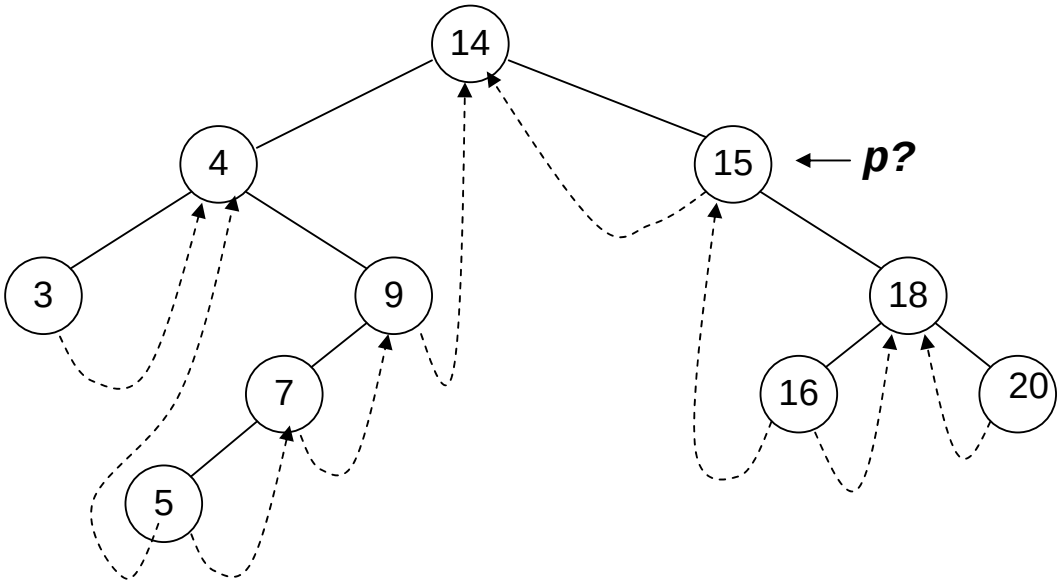
When we apply this routine on the sample tree, it does not work properly because the pointer that points to the node goes in the wrong direction. How can we fix this problem? Let's review the threaded binary tree again:



In the above figure, we have a binary search tree. Threads are also seen in it. These threads points to the successor and predecessor.

Our *nextInoder* routine, first of all checks that the right pointer of the node is thread. It means that it does not point to any tree node. In this case, we will return the right pointer of the node as it is pointing to the inorder successor of that node. Otherwise, we will go to some other part. Here we will change the value of pointer *p* to its right before running a while loops as long as the left pointer is the node. That means the left child is not a thread. We move to the left of the pointer *p* and keep on doing so till the time the left pointer becomes a thread.

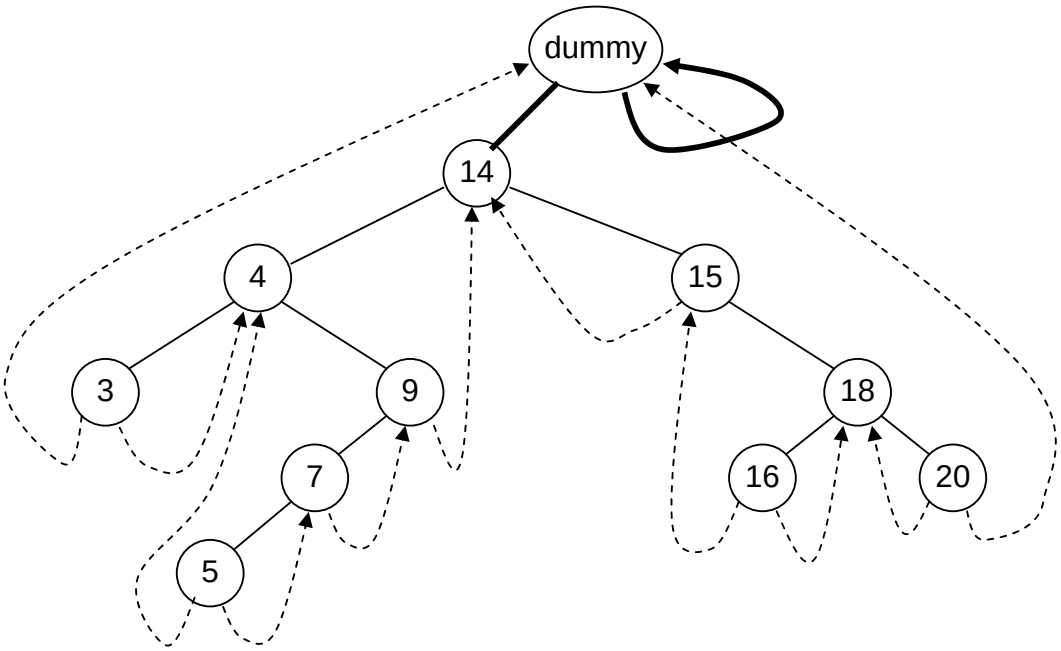
We will pass the root of the tree to the *nextInorder* routine. The pointer *p* is pointing to the node 14 i.e. the root node. As the right pointer of the node 14 is not a thread, so the pointer *p* will move to the node 15 as shown below:



Here we want the inorder traversal. It is obvious from the above figure that 15 is not the first value. The first value should be 3. This means that we have moved in the wrong direction. How this problem can be overcome? We may want to implement some logic that in case of the root node, it is better not to go towards the right side. Rather, the left side movement will be appropriate. If this is not the root node, do as usual. It may lend complexities to our code. Is there any other way to fix it? Here we will use a programming trick to fix it.

We will make this routine as a private member function of the class so other classes do not have access to it. Now what is the trick? We will insert a new node in the tree. With the help of this node, it will be easy to find out whether we are on the root node or not. This way, the pointer *p* will move in the correct direction.

Let's see this trick. We will insert an extra node in the binary tree and call it as a *dummy* node. This is well reflected in the diagram of the tree with the *dummy* node. We will see where that dummy node has been inserted.



This dummy node has either no value or some dummy value. The left pointer of this node is pointing to the root node of the tree while the right pointer is seen pointing itself i.e. to *dummy* node. There is no problem in doing all these things. We have put the address of *dummy* node in its right pointer and pointed the left thread of the left most node towards the *dummy* node. Similarly the right thread of the right-most node is pointing to the *dummy* node. Now we have some extra pointers whose help will make the *nextInorder* routine function properly.

Following is a routine *fastInorder* that can be in the public interface of the class.

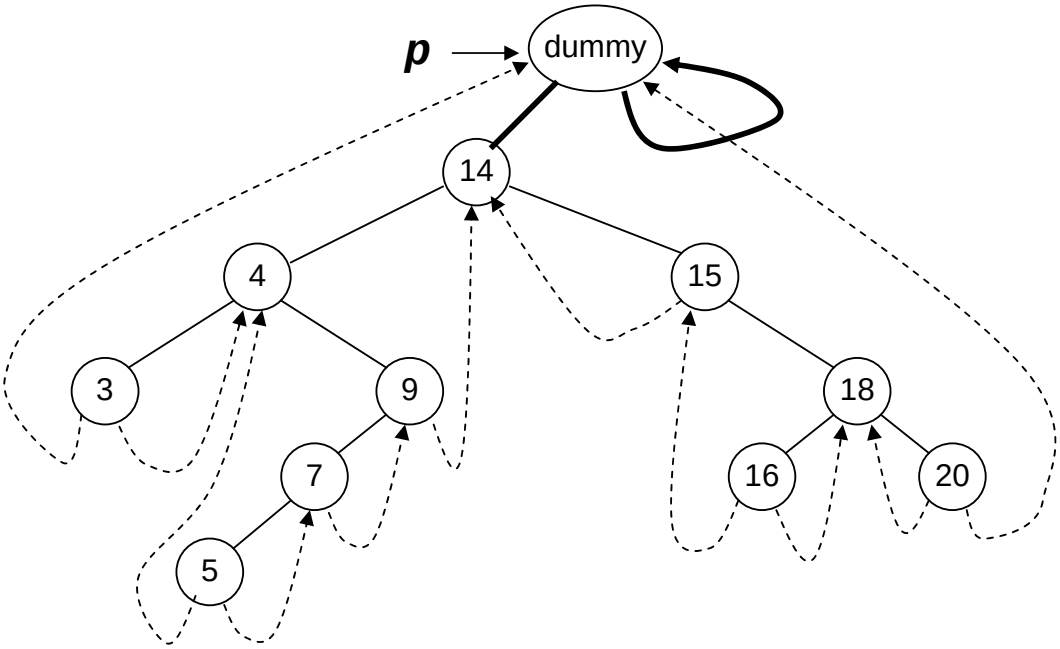
```
/* This routine will traverse the binary search tree  */

void fastInorder(TreeNode* p)
{
        while((p=nexInorder(p)) != dummy)  cout << p->getInfo();
}
```

This routine takes a *TreeNode* as an argument that make it pass through the root of the tree. In the while loop, we are calling the *nextInorder* routine and pass it *p*. The pointer returned from this routine is then assigned to *p*. This is a programming style of C. We are performing two tasks in a single statement i.e. we call the *nextInorder* by passing it *p* and the value returned by this routine is saved in *p*. Then we check that the value returned by the *nextInorder* routine that is now actually saved in *p*, is not a *dummy* node. Then we print the *info* of the node. This function is called as:

```
fastInorder(dummy);
```

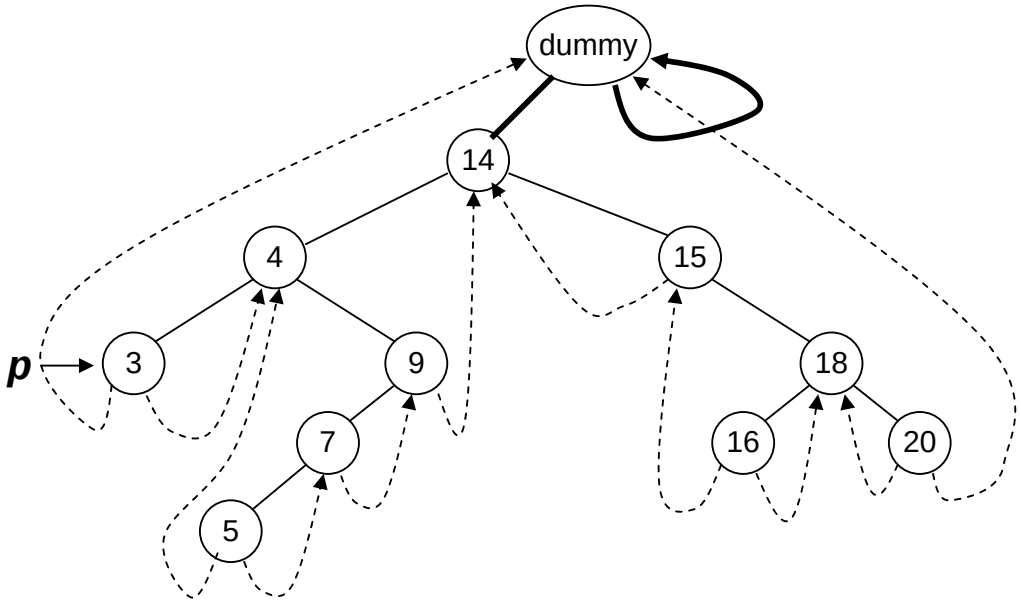
We are not passing it the root of the tree but the *dummy* node. Now we will get the correct values and see in the diagrams below that *p* is now moving in the right direction. Let's try to understand this with the help of diagrams.

First of all we call the *nextInorder* routine passing it the dummy node.



The pointer *p* is pointing to the *dummy* node. Now we will check whether the right pointer of this node is not thread. If so, then it is advisable to move the pointer towards the right pointer of the node. Now we will go to the while loop and start moving on the left of the node till the time we get a node with the left pointer as thread. The pointer *p* will move from dummy to node 14. As the left pointer of node 14 is not thread so *p* will move to node 4. Again the *p* will move to node 3. As the left pointer of *p* is thread, the while loop will finish here. This value will be returned that is pointing to node 3. The node 3 should be printed first of all regarding the inorder traversal. So with the help of our trick, we get the right information.

Now the while loop in the *fastInorder* will again call the *nextInorder* routine. We have updated the value of *p* in the *fastInorder* that is now pointing to the node 3. This is shown in the figure below:



According to the code, we have to follow the right thread of the node 3 that is pointing to the node 4. Therefore *p* is now pointing to the node 4. Here 4 is inorder successor of 3. So the pointer *p* has moved to the correct node for inorder traversal.

As the right pointer of the node 4 is a link, *p* will move to node 9. Later, we will go on the left of nodes and reach at the node 5. Looking at the tree, we know that the inorder successor of the node 4 is node 5. In the next step, we will get the node 7 and so on. With the help of threads and links, we are successful in getting the correct inorder traversal. No recursive call has been made so far. Therefore stack is not used. This inorder traversal will be faster than the recursive inorder traversal. When other classes use this routine, it will be faster. We have not used any additional memory for this routine. We are using the null links and putting the values of thread in it. This routine is very simple to understand. In the recursive routines, we have to stop the recursion at some condition. Otherwise, it will keep on executing and lead to the aborting of our program.

Complete Binary Tree

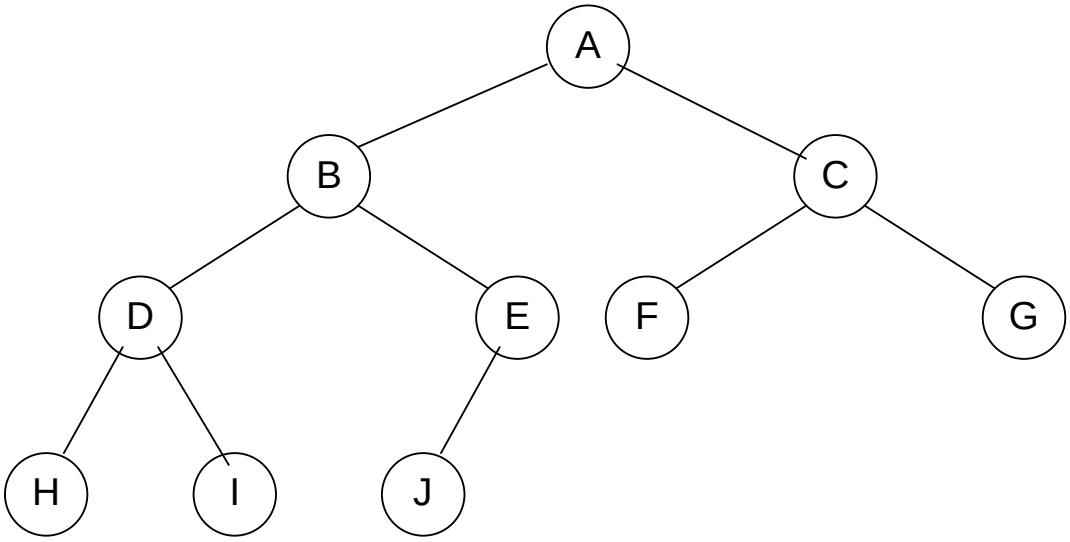
We have earlier discussed the properties of the binary trees besides talking about the internal and external nodes' theorem. Now we will discuss another property of binary trees that is related to its storage before dilating upon the complete binary tree and the heap abstract data type.

Here is the definition of a complete binary tree:

- A complete binary tree is a tree that is completely filled, with the possible exception of the bottom level.
- The bottom level is filled from left to right.

You may find the definition of complete binary tree in the books little bit different from this. A perfectly complete binary tree has all the leaf nodes. In the complete binary tree, all the nodes have left and right child nodes except the bottom level. At the bottom level, you will find the nodes from left to right. The bottom level may not

be completely filled, depicting that the tree is not a perfectly complete one. Let's see a complete binary tree in the figure below:



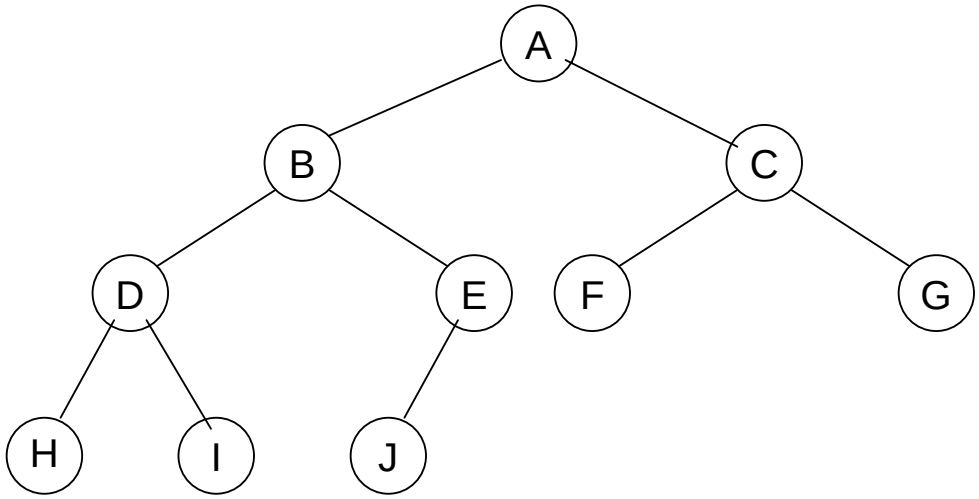
In the above tree, we have nodes as A, B, C, D, E, F, G, H, I, J. The node D has two children at the lowest level whereas node E has only left child at the lowest level that is J. The right child of the node E is missing. Similarly node F and G also lack child nodes. This is a complete binary tree according to the definition given above. At the lowest level, leaf nodes are present from left to right but all the inner nodes have both children. Let's recap some of the properties of complete binary tree.

- A complete binary tree of height h has between 2^h to $2^{h+1} - 1$ nodes.
- The height of such a tree is $\lfloor \log_2 N \rfloor$ where N is the number of nodes in the tree.
- Because the tree is so regular, it can be stored in an *array*. No pointers are necessary.

We have taken the floor of the $\log_2 N$. If the answer is not an integer, we will take the next smaller integer. So far, we have been using the pointers for the implementation of trees. The *treeNode* class has left and right pointers. We have pointers in the balance tree also. In the threaded trees, these pointers were used in a different way. But now we can say that an array can be stored in a complete binary tree without needing the help of any pointer.

Now we will try to remember the characteristics of the tree. 1) The data element can be numbers, strings, name or some other data type. The information is stored in the node. We may retrieve, change or delete it. 2) We link these nodes in a special way i.e. a node can have left or right subtree or both. Now we will see why the pointers are being used. We just started using these. If we have some other structure in which trees can be stored and information may be searched, then these may be used. There should be reason for choosing that structure or pointer for the manipulation of the trees. If we have a complete binary tree, it can be stored in an array easily.

The following example can help understand this process. Consider the above tree again.



	A	B	C	D	E	F	G	H	I	J				
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

We have seen an array of size 15 in which the data elements A, B, C, D, E, F, G, H, I, J have been stored, starting from position 1. The question arises why we have stored the data element this way and what is justification of storing the element at the 1st position of the array instead of 0th position? You will get the answers of these very shortly.

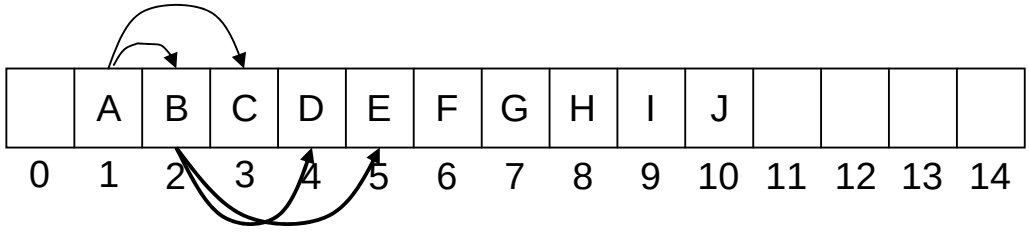
The root node of this tree is *A* and the left and right children of *A* are *B* and *C*. Now look at the array. While storing elements in the array, we follow a rule given below:

- For any array element at position *i*, the left child is at 2*i*, the right child is at (2*i* +1) and the parent is at floor(i/*2)*.

In the tree, we have links between the parent node and the children nodes. In case of having a node with left and right children, stored at position *i* in the array, the left child will be at position *2i* and the right child will be at *2i+1* position. If the value of *i* is 2, the parent will be at position 2 and the left child will be at position *2i* i.e. 4 .The right child will be at position *2i+1* i.e. 5. You must be aware that we have not started from the 0th position. It is simply due to the fact if the position is 0, *2i* will also become 0. So we will start from the 1st position, ignoring the 0th.

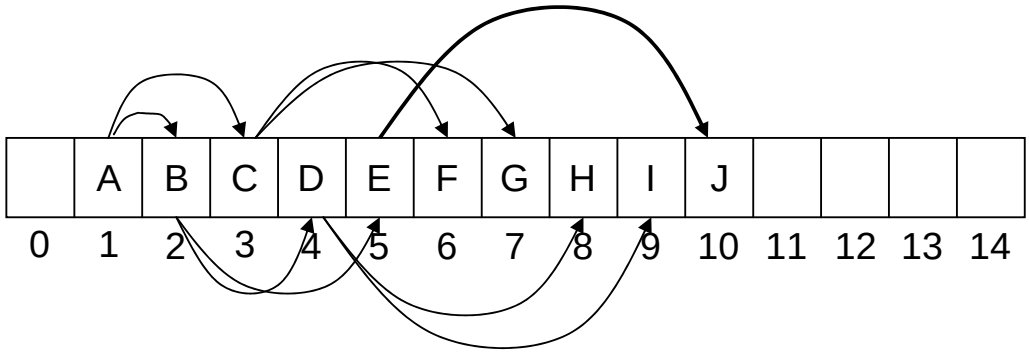
Lets see this formula on the above array. We have *A* at the first position and it has two children *B* and *C*. According to the formula the *B* will be at the *2i* i.e. 2nd position and C will be at *2i+1* i.e. 3rd position. Take the 2nd element i.e. *B*, it has two children *D* and *E*. The position of *B* is 2 i.e. the value of *i* is 2. Its left child *D* will be at positon *2i* i.e. 4th position and its right child *E* will be at position *2i+1* i.e. 5. This is shown in the

figure below:

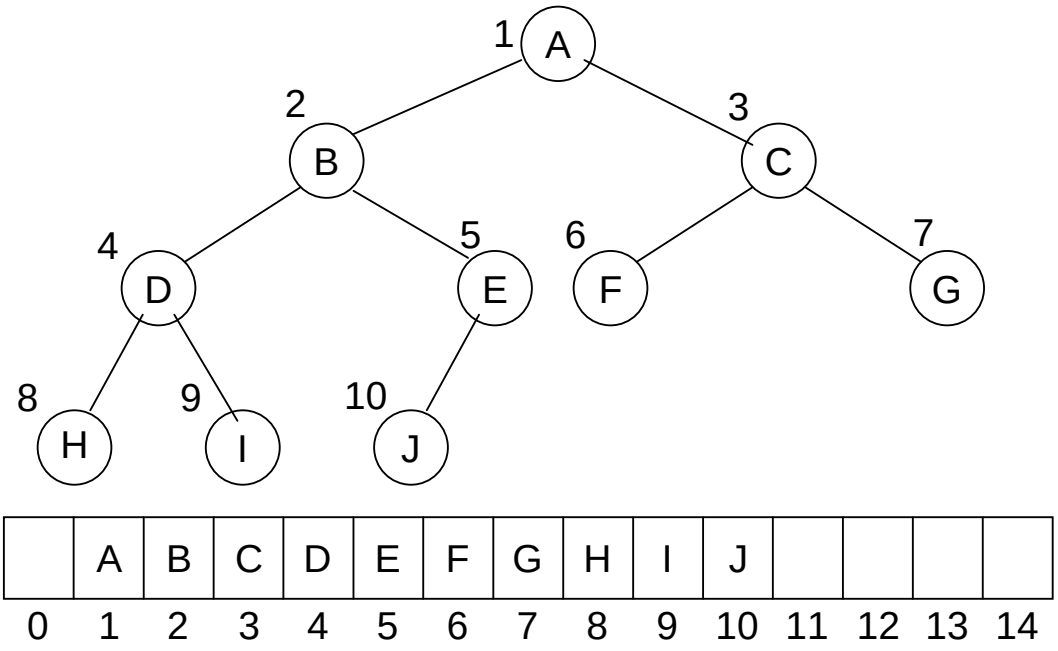


If we want to keep the tree's data in the array, the children of *B* should be at the position 4 and 5. This is true. We can apply this formula on the remaining nodes also. Now you have understood how to store tree's data in an array. In one respect, we are using pointers here. These are not C++ pointers. In other words, we have implicit pointers in the array. These pointers are hidden. With the help of the formula, we can obtain the left and right children of the nodes i.e. if the node is at the *ith* position, its children will be at *2i* and *2i+1* position. Let's see the position of other nodes in the array.

As the node *C* is at position 3, its children should be at 2*3 i.e. 6^{th} position and 2*3+1 i.e. 7^{th} position. The children of *C* are *F* and *G* which should be at 6^{th} and 7^{th} position. Look at the node *D*. It is at position 4. Its children should be at position 8 and 9. *E* is at position 5 so its children should be at 10 and 11 positions. All the nodes have been stored in the array. As the node *E* does not have a right child, the position 11 is empty in the array.



You can see that there is only one array going out of *E*. There is a link between the parent node and the child node. In this array, we can find the children of a node with the help of the formula i.e. if the parent node is at *ith* position, its children will be at *2i* and *2i+1* position. Similarly there is a link between the child and the parent. A child can get its parent with the help of formula i.e. if a node is at *ith* position, its parent will be at *floor(i/2)* position. Let's check this fact in our sample tree. See the diagram below:



Level Order Numbers & Array index

Consider the node *J* at position is 10. According to the formula, its parent should be at *floor(10/2)* i.e. 5 which is true. As the node *I* is at position 9, its parent should be at *floor(9/2)* i.e. 4. The result of 9/2 is 4.5. But due to the floor, we will round it down and the result will be 4. We can see that the parent of *I* is *D* which is at position 4. Similarly the parent of *H* will be at *floor(8/2)*. It means that it will be at 4. Thus we see that *D* is its parent. The links shown in the figure depict that *D* has two children *H* and *I*. We can easily prove this formula for the other nodes.

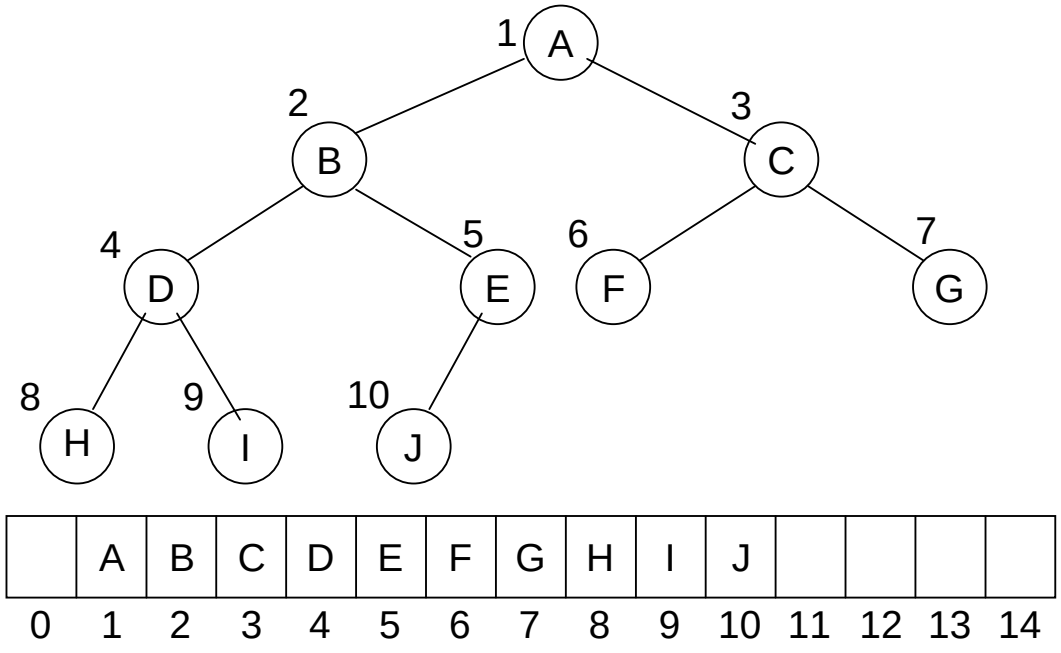
From the above discussion we note three things. 1) We have a complete binary tree, which stores some information. It may or may not be a binary search tree. This tree can be stored in an array. We use *2i* and *2i+1* indexing scheme to put the nodes in the array. Now we can apply the algorithms of tree structure on this array structure, if needed.

Now let's talk about the usage of pointers and array. We have read that while implementing data structures, the use of array makes it easy and fast to add and remove data from arrays. In an array, we can directly locate a required position with the help of a single index, where we want to add or remove data. Array is so important that it is a part of the language. Whereas the data structures like tree, stack and queue are not the part of C or C++ language as a language construct. However we can write our classes for these data structures. As these data structures are not a part of the language, a programmer can not declare them directly. We can not declare a tree or a stack in a program. Whereas we can declare an array directly as *int x [];* The array data type is so efficient and is of so common use that most of the languages support it. The compiler of a language handles the array and the programmer has to do nothing for declaring and using an array.

We have built the binary trees with pointers. The use of pointers in the memory requires some time. In compilers or operating system course, we will read that when a program is stored in the memory and becomes a process, the executable code does not come in the memory. There is a term paging or virtual memory. When a program executes, some part of it comes in the memory. If we are using pointers to go to different parts of the program, some part of the code of program will be coming (loading) to memory while some other may be removed (unloading) from the memory. This loading and unloading of program code is executed by a mechanism, called paging. In Windows operating system, for this virtual memory (paging mechanism), a file is used, called page file. With the use of pointers, this process of paging may increase. Due to this, the program may execute slowly. In the course of Operating System and Compilers, you will read in detail that the usage of pointers can cause many problems.

So we should use arrays where ever it can fulfill our requirements. The array is a very fast and efficient data structure and is supported by the compiler. There are some situations where the use of pointers is beneficial. The balancing of AVL tree is an example in this regard. Here pointers are more efficient. If we are using array, there will be need of moving a large data here and there to balance the tree.

From the discussion on use of pointers and array, we conclude that the use of array should be made whenever it is required. Now it is clear that binary tree is an important data structure. Now we see that whether we can store it in an array or not. We can surely use the array. The functions of tree are possible with help of array. Now consider the previous example of binary tree. In this tree, the order of the nodes that we maintained was for the indexing purpose of the array. Moreover we know the level-order traversal of the tree. We used queue for the level-order of a tree. If we do level-order traversal of the tree, the order of nodes visited is shown with numbers in the following figure.



In the above figure, we see that the number of node *A* is 1. The node *B* is on number 2 and *C* is on number 3. At the next level, the number of nodes D, E, F and G are 4, 5, 6 and 7 respectively. At the lowest level, the numbers 8, 9 and 10 are written with nodes H, I and J respectively. This is the level-order traversal. You must remember that in the example where we did the preorder, inorder and post order traversal with recursion by using stack. We can do the level-order traversal by using a queue. Now after the level-order traversal, let's look at the array shown in the lower portion of the above figure. In this array, we see that the numbers of A, B, C and other nodes are the same as in the level-order traversal. Thus, if we use the numbers of level-order traversal as index, the values are precisely stored at that numbers in the array. It is easy for us to store a given tree in an array. We simply traverse the tree by level-order and use the order number of nodes as index to store the values of nodes in the array. A programmer can do the level-order traversal with queue as we had carried out in an example before. We preserve the number of nodes in the queue before traversing the queue for nodes and putting the nodes in the array. We do not carry out this process, as it is unnecessarily long and very time consuming. However, we see that the level-order traversal directly gives us the index of the array depending upon which data can be stored.

Data Structures

Lecture No. 29

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 6
6.3

Summary

- Complete Binary Tree
- Heap
- Max Heap
- Insertion in a Heap

Complete Binary Tree

In the previous lecture, we talked about the ways to store a complete binary tree in an array. The 2i and 2i+1 scheme were employed to store the link of the parent to children and link of children to parent. Through this link, a programmer can go to the children of a node. We know that array is a very efficient data structure. In a number of languages, it is found as a built-in data type. Now the question arises if we can store the binary tree in an array, why there should be the use of pointers? It is very simple that an array is used when the tree is a complete binary tree. Array can also be used for the trees that are not complete binary trees. But there will be a problem in this case. The size of the array will be with respect to the deepest level of the tree according to the traversal order. Due to incomplete binary tree, there will be holes in the array that means that there will be some positions in the array with no value of data. We can understand it with a simple example. Look at the following figure where we store a complete binary tree in an array by using 2i and 2i+1 scheme. Here we stored the nodes from A to J in the array at index 1 to 10 respectively.

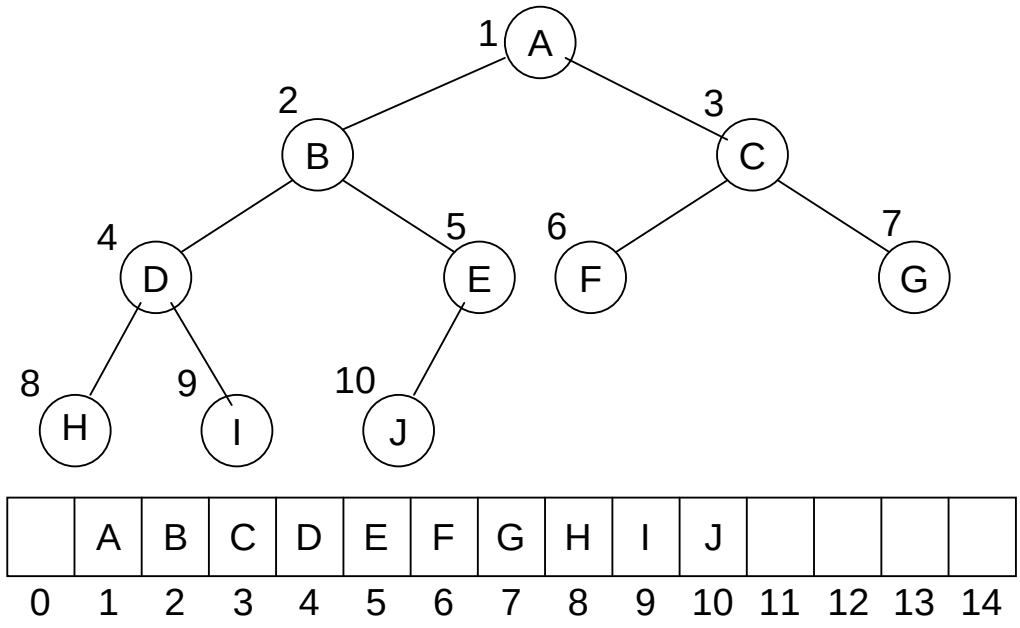


Figure 29.1: Complete Binary Tree

Suppose that this tree is not complete. In other words, B has no right subtree that means E and J are not there. Similarly we suppose that there is no right subtree of A. Now the tree will be in the form as shown in the following figure (29.2).

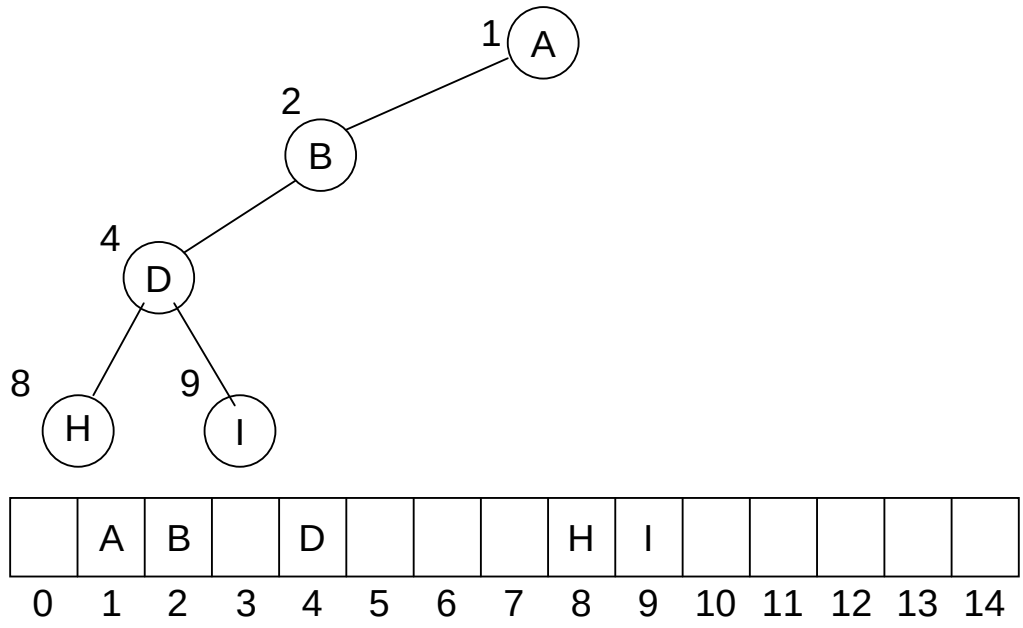
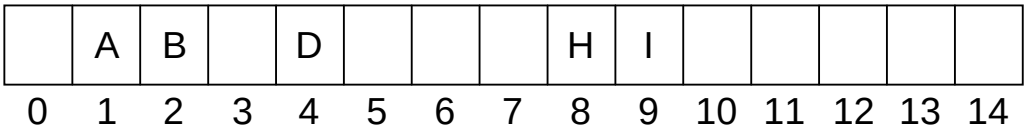


Figure 29.2: Not a complete binary tree

In this case, the effort to store this tree in the array will be of no use as the 2i and 2i+1 scheme cannot be applied to it. To store this tree, it may be supposed that there are nodes at the positions of C, F, G, E and J (that were there in previous figure). Thus we transform it into a complete binary tree. Now we store the tree in the array by using 2i and 2i +1 scheme. Afterwards, the data is removed from the array at the positions of the imaginary nodes (in this example, the nodes are C, F, G, E and J). Thus we notice that the nodes A, B and H etc are at the positions, depicting the presence of a complete binary tree. The locations of C, f, G, E and J in the array are empty as shown in the following figure.



Now imagine that an incomplete binary tree is very deep. We can store this tree in the array that needs to be of large size. There will be holes in the array. This is the wastage of memory. Due to this reason, it is thought that if a tree is not completely binary, it is not good to store it into an array. Rather, a programmer will prefer to use pointers for the storage.

Remember that two things are kept into view while constructing a data structure that is memory and time. There should such a data structure that could ensure the running of the programs in a fast manner. Secondly, a data structure should not use a lot of memory so that a large part of memory occupied by it does not go waste. To manage the memory in an efficient way, we use dynamic memory with the help of pointers. With the use of pointers only the required amount of memory is occupied.

We also use pointers for complex operations with data structure as witnessed in the deletion operation in AVL tree. One of the problems with arrays is that the memory becomes useless in case of too many empty positions in the array. We cannot free it and use in other programs as the memory of an array is contiguous. It is difficult to free the memory of locations from 50 to 100 in an array of 200 locations. To manage the memory in a better way, we have to use pointers.

Now we come to a new data structure, called ‘heap’.

Heap

Heap is a data structure of big use and benefit. It is used in priority queue. Recall the example of bank simulation. In that case, we used event-based queues. We put the events that were going to happen in a special queue i.e. priority queue. This priority queue does not follow the FIFO rule. We put the elements in the queue at the end but later got the element with respect to its priority. We get the element that is going to occur first in the future. In that example, we implemented the priority queue with arrays. It was seen that when we insert an element in the queue, the internally used data was sorted in the array. Thus the event with minimum time of occurrence becomes at first position in the sorted array. We get the event with minimum time first. After the removal of the element, a programmer shifts the array elements to left. When we insert a new element, the array is sorted again. As there, in the bank example, were five or six events, the use of array fulfilled our requirement. We need not to have other data type that makes the queue efficient. The array is efficient but sorting is an expensive procedure. It may be complex and time-consuming. Secondly, we have to shift elements while adding or removing them from the array. Thus the

implementation of priority queue with an array is not efficient in terms of time. There is an alternate that is called heap. The use of priority queue with the help of heap is a major application. The priority queue is itself a major data structure, much-used in operating systems. Similarly priority queue data structure is used in network devices especially in routers. Heap data structure is also employed in sorting algorithms. There are also other uses of heap.
Let's discuss the heap data structure with special reference to its definition,

"The definition of heap is that it is a complete binary tree that conforms to the heap order".

In this definition there is a term 'heap order'. The heap order is a property that states that in a (min) heap for every node X, the key in the parent is smaller than (or equal to) the key in X. In other words, the parent node has key smaller than or equal to both of its children nodes. This means that the value in the parent node is less than the value on its children nodes. It is evident from the definition that we implement the heap by complete binary tree. It can also be implemented by some other method. But normally, we implement heap with complete binary tree. We know that in a binary search tree, the value of a node is greater than the values in its left subtree and less than the values in its right subtree. The heap has a variation to it. In heap, the value of a node is less than the values of its left and right children. The values in left and right children may be more or less with each other but these will be greater than the value in their parent node.
Consider the tree shown in the following figure.

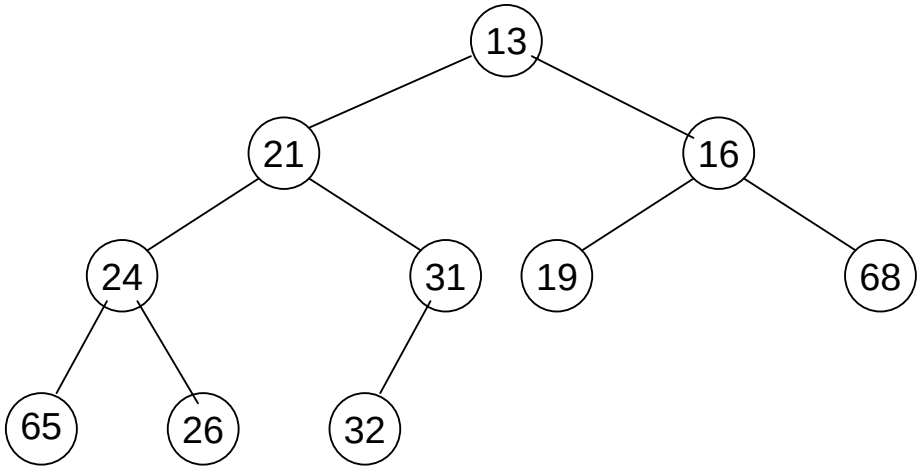


Figure 29.3: A min heap

This is a complete binary tree. Now consider the values (numbers) in the nodes. This is not a binary search tree. If we carry out the inorder traversal, the result will be

65, 24, 26, 21, 32, 31, 13, 19, 16, 68.

We can see that it is not in a sorted order as got while traversing a binary search tree. So it is not a binary search tree. It's simply a complete binary tree.
Now we see the heap order in this tree. We start from node having value 13. Its left

child has value 21 while the value possessed by the right child is 16. So this is in line with the heap order property. Now we come to node 21. The values in its left and right child are 24 and 31 respectively. These values are greater than 21. Thus, it also fulfills the heap order property. Now we consider the node having value 16. The value in left child node of it is 19 and value in right child node is 68. So the value of parent node (i.e. 16) is less than the values of its children nodes. Similarly we can see other nodes. The node 24 is less than its children that are 65 and 26 respectively. And the node 31 is less than its child i.e. 32. Now for this tree, three things have been proved. First, this is a binary tree. Secondly it is a complete binary tree. Thirdly it fulfills the heap order property i.e. the value of the parent node is less than that of its left and right child nodes. This property is valid for every node of the tree. The leaf nodes have no child so there is no question of heap property. The heap shown in the figure above is a min heap. In the min heap, the value of parent node is less than the values in its child nodes. Thus in a min heap, the value of the root node is the smallest value in the tree.

Now consider the tree in the following figure. This is a complete binary tree. This is neither a search tree, nor a heap.

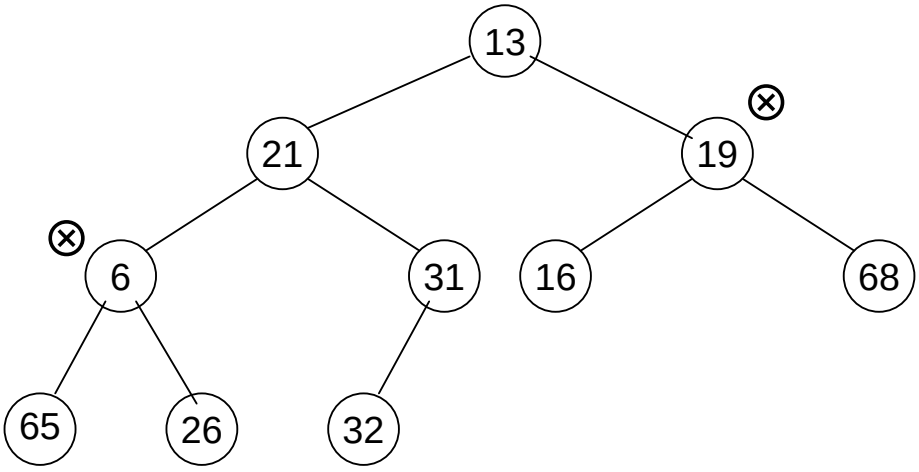


Figure 29.4: Not a heap

Look at the node having value 19. The values in its left and right child nodes are 16 and 68 respectively. Thus the value of left child (i.e. 16) is less than that of the parent. So it is not a heap. If it were a heap or min heap, the value 16 should have been parent and the value 19 should have its child. Due to this violation, (the value of child is less than that of the parent) it is not a heap (min heap).

Max Heap

We can also make a max heap. In max heap, each node has a value greater than the value of its left and right child nodes. Moreover, in this case, the value of the root node will be largest and will become lesser at downward levels. The following figure shows a max heap.

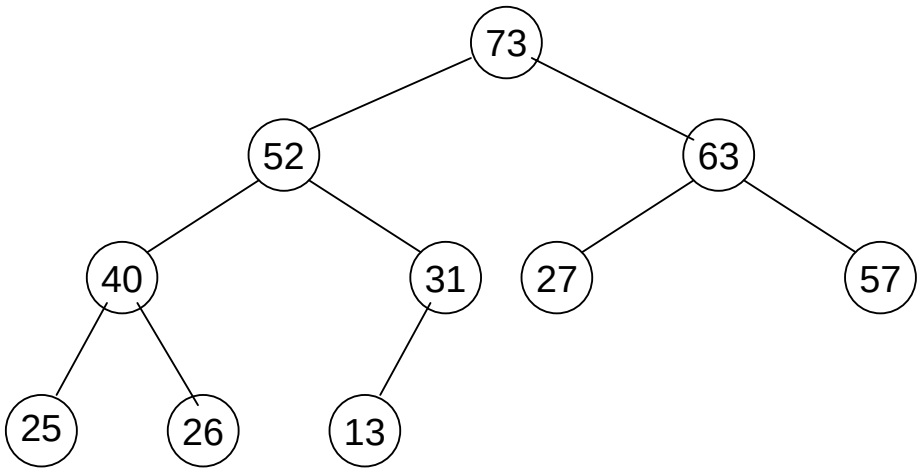


Figure 29.5: A max heap

Consider the min heap again. By removing the root from the min heap, we get the smallest value. Now if the remaining values adjust themselves to again form a heap, the minimum value among these remaining values will get on the top. And if we remove this value, there will be the second minimum value of the heap. The remaining values again form a heap and there will be the smallest number among these on the top. Thus we will get the third minimum number. If we continue this process of getting the root node and forming a heap of remaining values, the numbers will get in ascending order. Thus we get the sorted data. By putting data in heap and getting it in a particular way, we achieve a sorting procedure. This way, a programmer gets sorted data.

Now suppose that the numbers in heap are priorities or events. If we build the max heap, the number with greater priority will get on the top. Now if we get data from the heap, the data on top will be gotten first. As it is max heap, the data on top will be the largest value.

Insertion in a Heap

Now let's discuss the insertion of a value in a min or max heap. We insert a value in a min heap and rearrange it in such a way that the smallest value gets on the top. Similarly, if we are inserting a value in a max heap, the largest value goes on the top. To discuss insertion in min heap, consider the following existing heap. This is the same heap discussed earlier. As it is a complete binary tree, we keep it in an array. In the figure, the level order of numbers of the nodes is also seen that describes the position of nodes (index number) in the array.

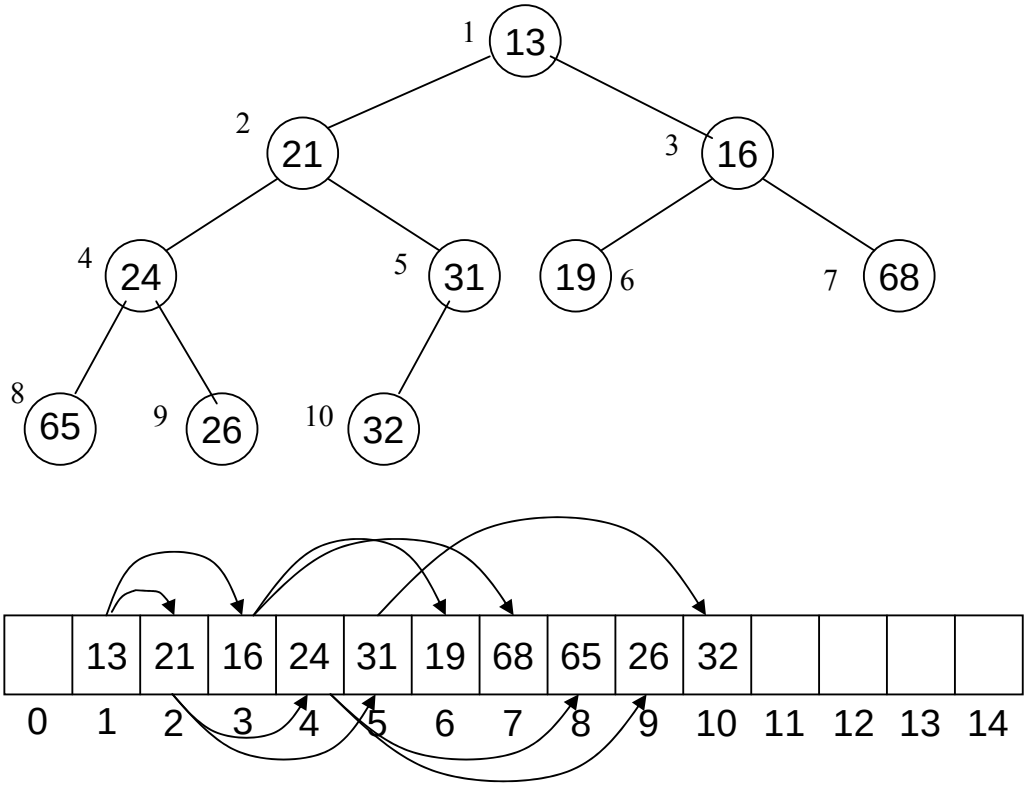


Figure 29.6: An existing heap

Now we add a new node to this existing heap. Suppose the node that we want to add has a value 14. We will add this node at a position in such a way that the tree should remain complete binary one. Following this guideline, this node will be added on right side of node 31. In the array, the node 31 is presently at position 5. Now according to the formula of 2i and 2i+1, the left child of 31 will be at positions 10 that already exists. The right child of 31 (that is 14 now) will be at position 11 in the array. This addition of node in the tree and value in the array is shown in the following figure. We have shown the node link and position in the array with dotted lines. By this, it is shown that the node of value 14 may become here. We did not actually put it there. We have to put it at such position so that the heap should remain intact.

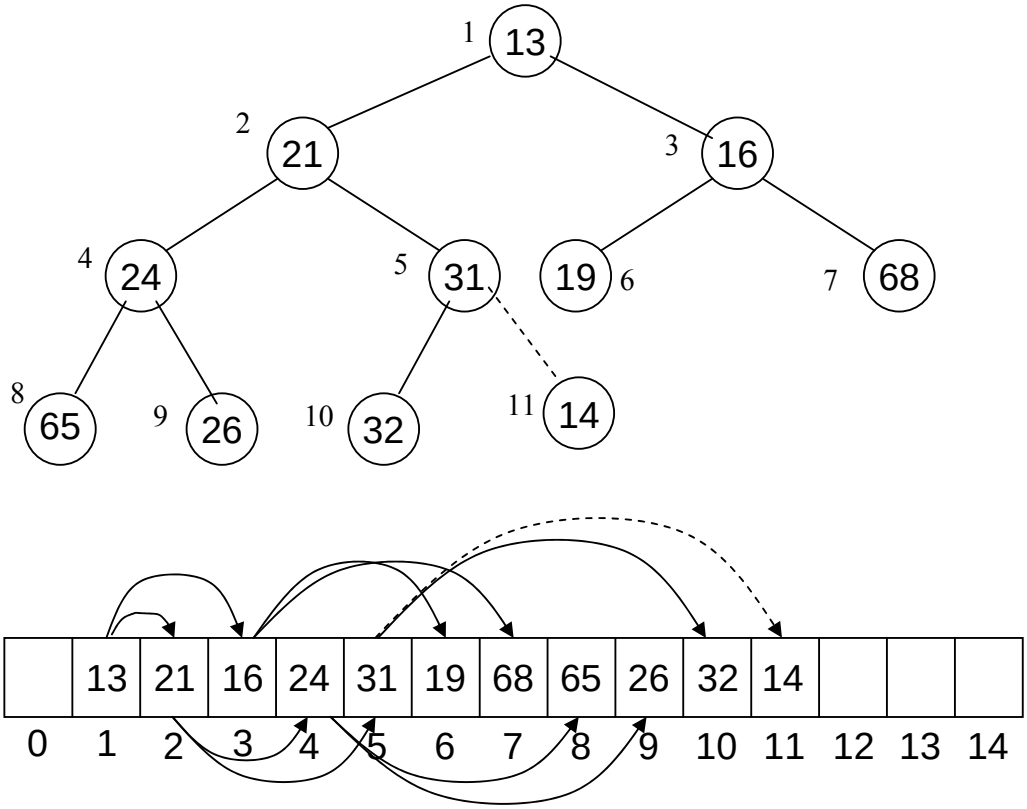


Figure 29.7: inserting new value in a heap

With the addition of new node, we see that this is still a complete binary tree and can be stored in an array. There are no holes in the array, leading to no wastage of the positions.

Now preservation of the heap order is needed. We are discussing the min heap in this example. In the min heap, the smallest element is at the top (root). And at each level, every node has a smaller value than that of its children nodes. Now we have to put the node 14 at its proper position with respect to heap order. Thus heap is being implemented with complete binary tree with the help of an array. We have to find the final; position of node 14 in the tree and array preserving the heap order. A look on the tree in the previous figure shows that if we put node 14 (shown with dotted line) at that position, the heap order will not be preserved. By considering the node 31, we see that the value of its right child (i.e. 14) is less than it. Thus heap property is violated. To preserve heap property, the node 14 should be up and node 31 should be down. If node 14 is at position of 31 and it is compared with its parent that is 21, we come to know that node 21 should also be at down position. How do we take node 14 to its proper position? To achieve this objective, we have to take node 21 and 31 down and move the node 14 up. Now let's see how we can do this. In the following figure, the position is seen where a new node can be added with an empty circle (hole). The value of this node is not shown. However, the new inserted value may go to that position. Similarly in the array we show that there will be some value at position 11.

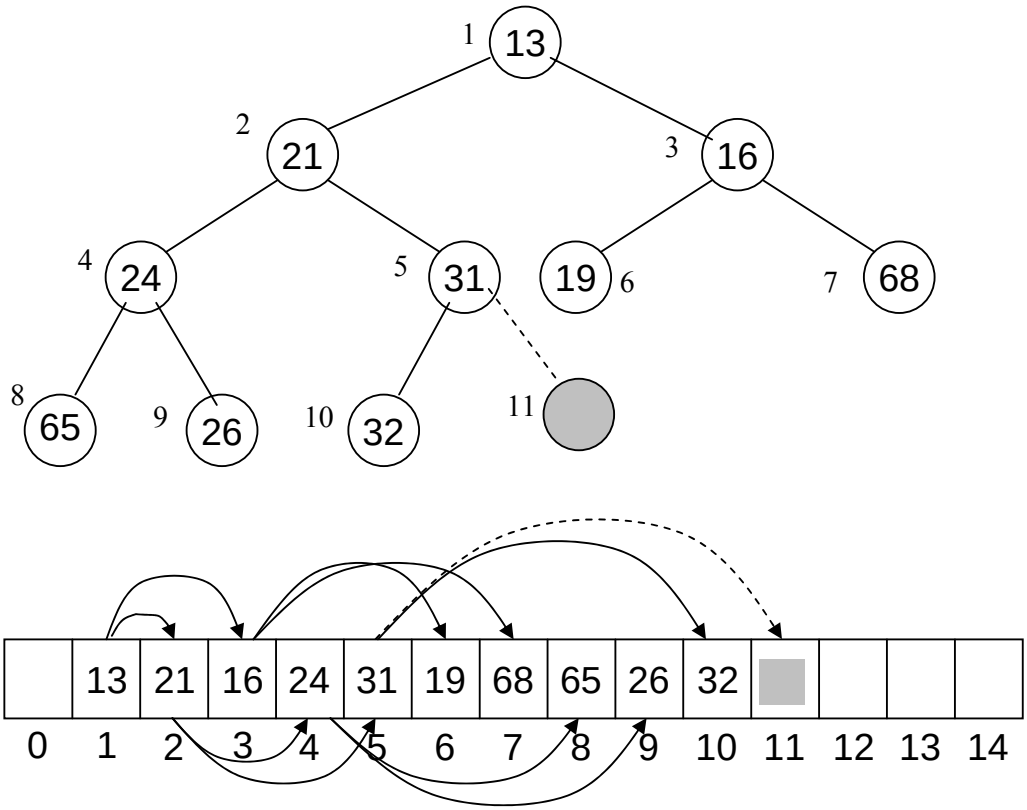
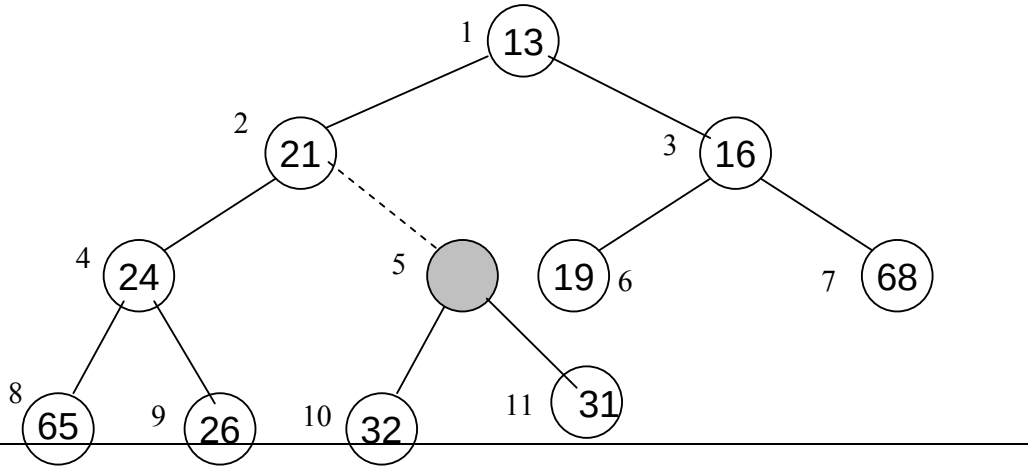


Figure 29.8: inserting new value in a heap

Now we compare the new value to be inserted (that is 14) at position 11 with its parent node. The parent node of newly inserted node is 31 that is greater than the new value. This is against the (min) heap property. According to the heap property, the node 14 may be the parent of 31 but not its child. So we move the node 31 to the position 11 and the new node to the position 5 that was earlier the position of 31. This technique is also employed in the array. Thus we get the array and tree in the form as shown in the following figure.



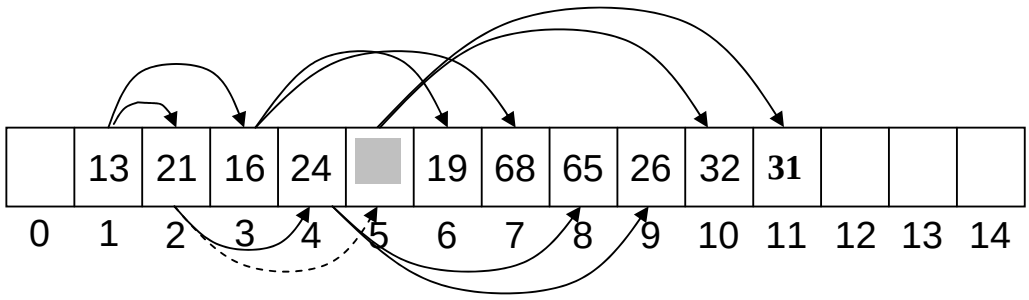


Figure 29.9: inserting new value in a heap

Now if the new node comes at the position 5, its parent (that is node 21 at position 2) is again greater than it. This again violates the heap property because the parent (i.e. 21) is greater than the child whose value is 14. So we bring the node 21 down and take the new node up. Thus the node 21 goes to the position 5 and the new node attains the position 2 in the tree and array as shown in the following figure.

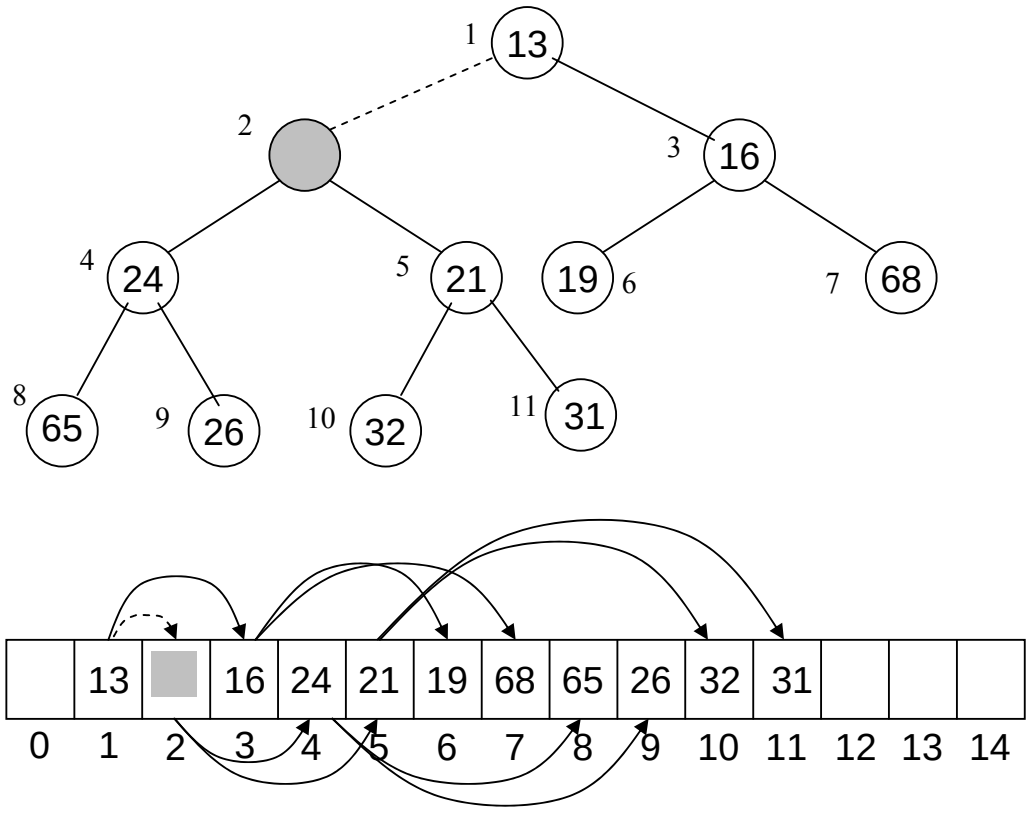


Figure 29.10: inserting new value in a heap

To interchange the positions of values in the array, only these values are swapped which is very easy process. We can do this easily in arrays. Now if the new value 14 comes at the position 2, the heap property will be preserved due to the fact that the

parent of this node i.e. the node of value 13 is less than its child (the new node that is 14). Thus the final position of new node 14 is determined and we put it here i.e. at position 2. This is shown in the following figure.

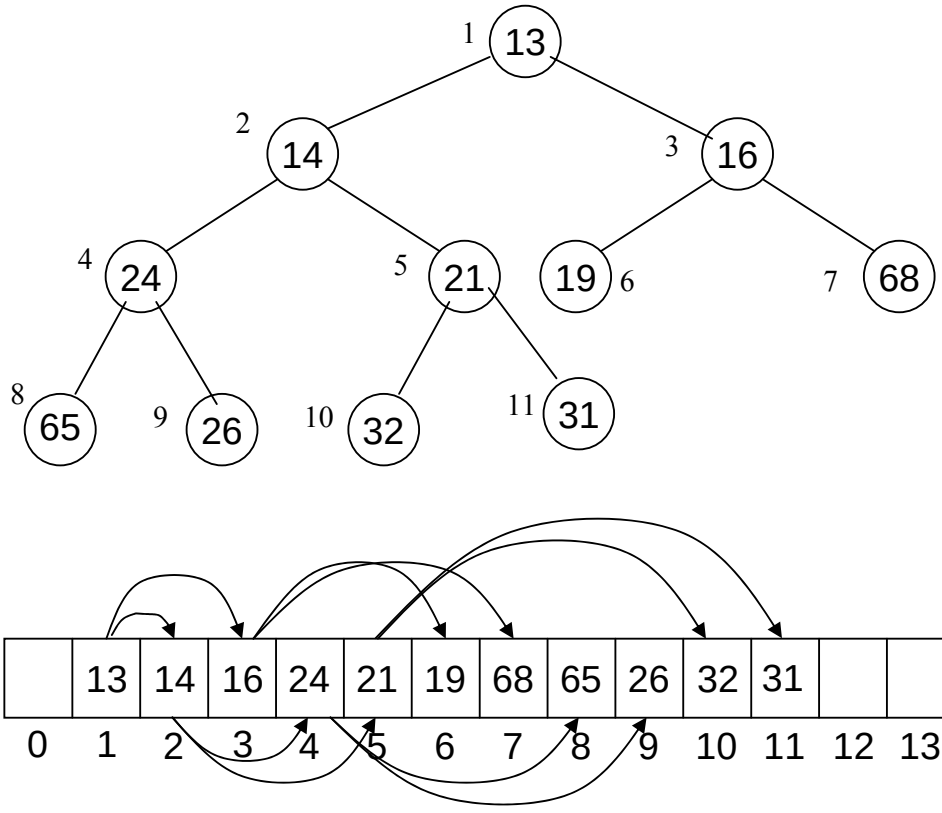


Figure 29.11: inserting new value in a heap

It is clear by now that the tree after insertion of a new value follows the heap property. We see that there is an algorithm for inserting a new value in the heap. If we have a heap stored in an array as a complete binary tree, a new value is put in the array at a position so that it can hold preserving the property of complete binary tree. In our example, it is the position 11 where new value may be put. We know that the parent of a node is at position *floor (i / 2)*. Thus we find the position of parent node of the new node and compare this new value with that. If the parent node has a value greater than this new value (i.e. its child), the heap property is violated. To maintain this property, we exchange these values. Now at the new position we find the parent of that position and compare the value with the value at that position. Thus the array is traversed by level-order. After a comparison, we go one level up in the tree. And we go to upward by looking at the parent node of the newly inserted node level by level. We stop at the position where the value of the parent node is less than the value of its child i.e. the node to be inserted. To insert a node, it is necessary to find the position of the node before putting the value there. This is efficient and fast way as actual values are not exchange in this case. We simply move the data. Under the second method, it can also be done with exchanges. Let's do this for our previous heap. Look at the following figure. Here we put the new node 14 at its position. It is the right child of node 31. We have already seen that this is the position where a new node can

be added so that the tree remains complete binary tree. In the array, it is at position 11.

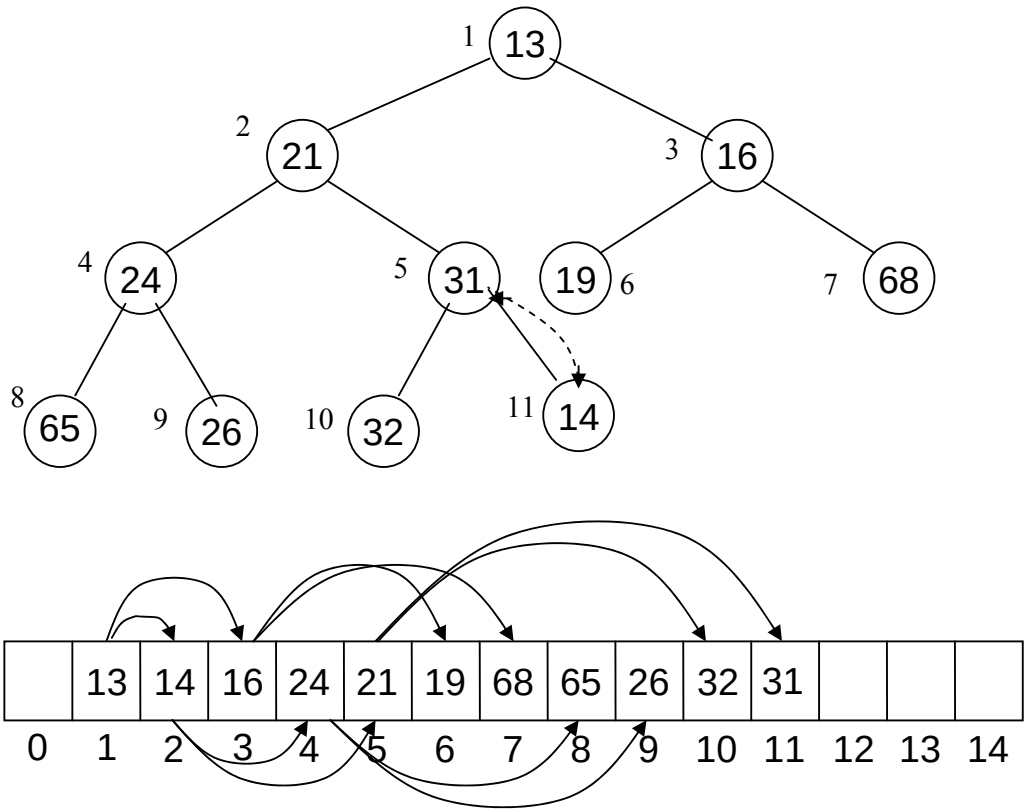
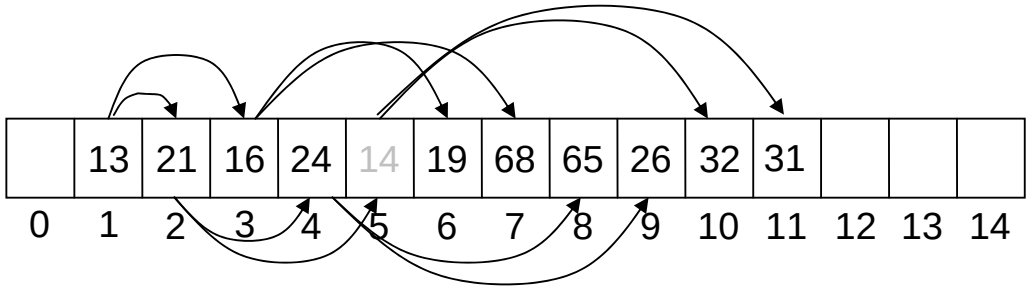


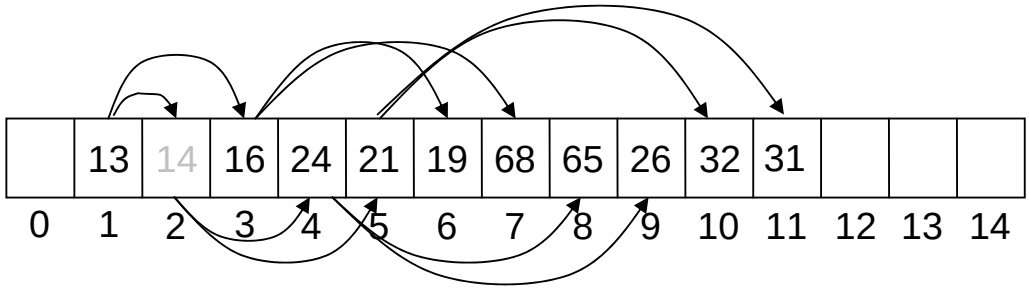
Figure 29.12: inserting new value in a heap

Now we check the heap order and compare node 14 with its parent that is node 31. It is seen that this child node i.e. 14 is less than its parent i.e. 31. Thus the heap order property is violated. We exchange the node 14 and 31due to which the node 14 becomes the parent and node 31 turns into its child. That is why, the node 31 is now at position 11 and node 14 is at position 5. After this exchange, the tree remains complete binary one as we only have exchanged the values of nodes. The following array representation shows this.



After this we compare the node 14 with its new parent. The new parent of node 14 can be found by the formula of *floor (i / 2)*. The parent node of 14 will be at position *floor (5/2)* that is position 2. We can see that the node at position 2 is 21. Thus 21 is greater than its child i.e. 14, violating the heap property. So we again exchange these values. The node 14 now becomes the parent of 21 and 21 gets its child. In the array, the

nodes 14 and 21 are at positions 2 and 5 respectively. The array representation of it is as below.



Now we compare this node 14 with its new parent i.e. 13. Here the heap property stands preserved due to the fact that the parent node i.e. 13 is less than its child node i.e. 14. So this tree is a heap now. The following figure shows this heap and array representation of it.

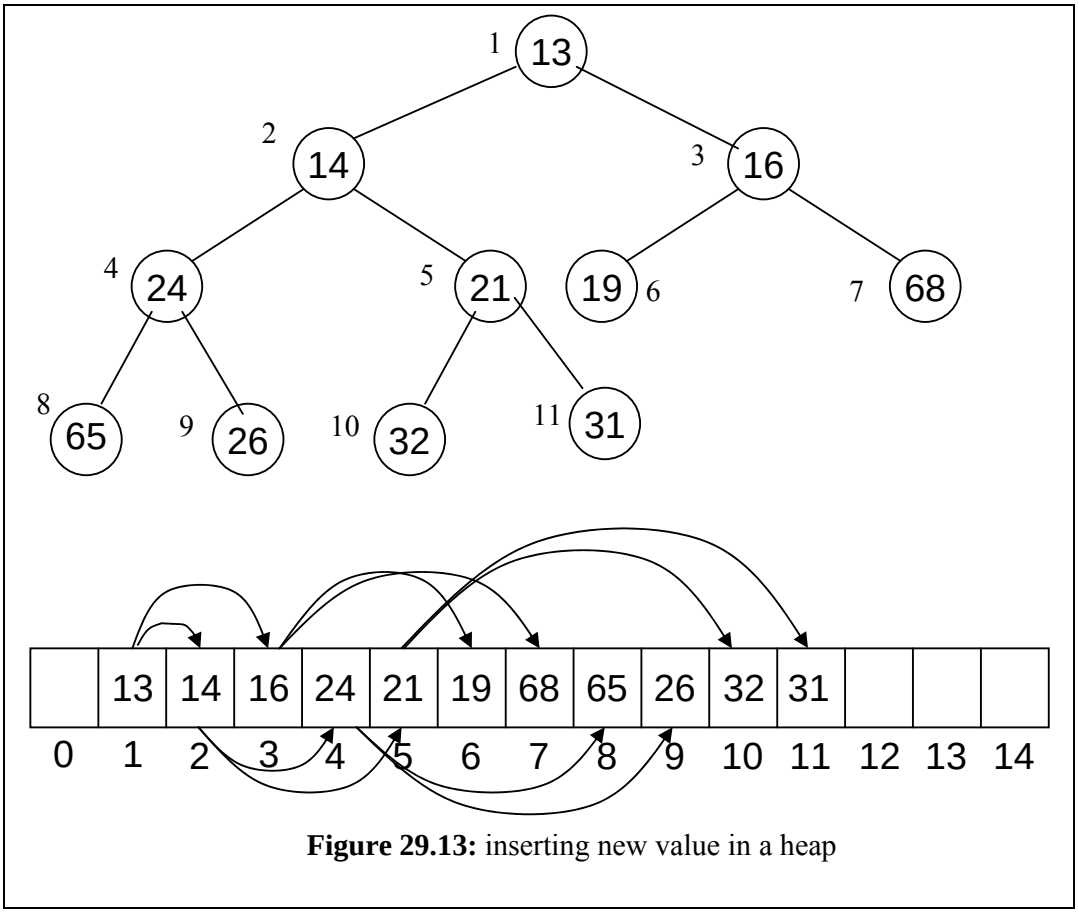


Figure 29.13: inserting new value in a heap

Suppose we want to add another node to this heap. This new node has value 15. As the heap is a complete binary tree, so this new node will be added as left child of node 19. In the array, we see that the position of 19 is 6 so the position of its left child will be (by formula of 2i) 6 x 2 that is 12. This is shown in the following figure.

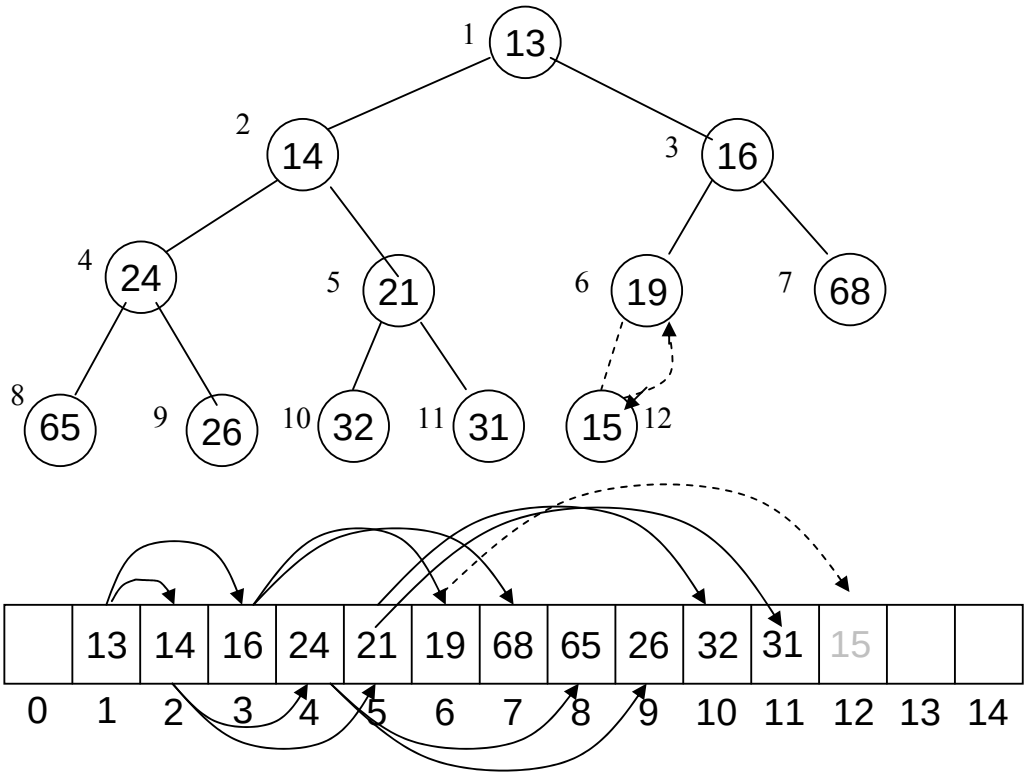
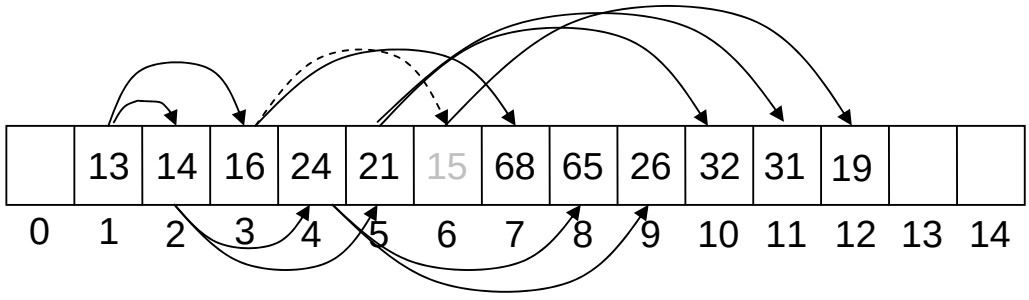


Figure 29.14: inserting new value in a heap

Here the new node 15 is less than its parent node (i.e. 19). To preserve the heap property, we have to exchange these values. Thus the value 15 goes to the position 6 and value 19 attains the position 12 as shown in the array in figure below.



Now we compare the value 15 with its new parent i.e.16. Here again the parent (16) is greater than its child (15). So to preserve the heap property we need to exchange these values. After the exchange, the value 15 is now at position 3 and value 16 is seen position 6. The following figure shows this step.

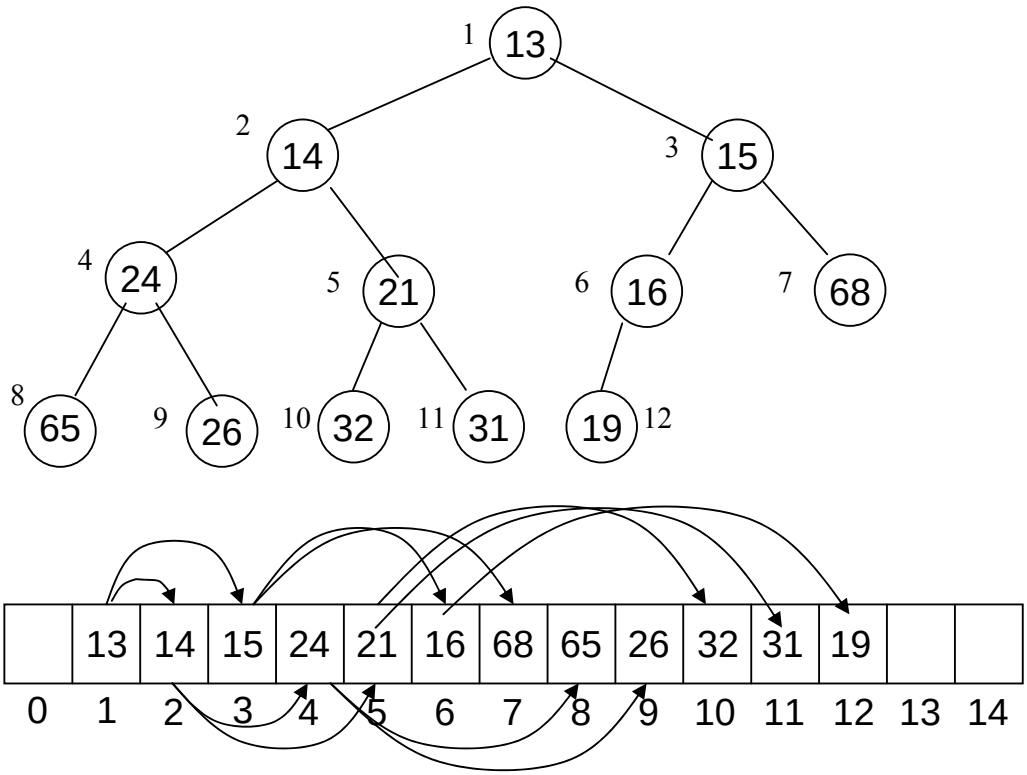


Figure 29.15: inserting new value in a heap

The figure also shows that the node 15 is greater than its new parent i.e. 13. In other words, the parent node is less than its child. So the heap property has been preserved, warranting no exchange of values. The node 15 has gone to its proper position .All the nodes in this tree now follow the heap order. Thus it is a heap in which a new node has been inserted.

Data Structures

Lecture No. 30

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 6
6.3

Summary

- Inserting into a Min-Heap
- Deleting from a Min-Heap
- Building a Heap

Inserting into a Min-Heap

In the previous lecture, we discussed about the heap data structure besides touching upon the concept of storage of complete binary tree in an array. For the study of parent-child relationship, the '2i and 2i+1 scheme' was used. Then we changed our focus to heap and found it best for the priority queues. In the previous lecture, we did not really discuss the uses of heap. Rather, most of the discussion remained focused on insertion method in the binary tree employed at the time of implementation with the help of an array. After inserting a new element in the array and by moving few elements, a programmer can have minimum or maximum heap. In case of minimum heap, the minimum value in the tree is always in the *root* of the tree. However, in case of maximum heap, the maximum value in the tree lies in the *root* node.
When we insert a new element in the tree implemented with the help of an array, we insert element at the last position (of the array). Due to this insertion at the end, the heap order may be violated. Therefore, we start moving this element upwards. While moving upward, this element may reach at the *root* of the tree. It is important to note that only one branch of the tree is affected because of this movement in the upward direction. This process, being of localized nature, will not disturb the entire tree.

To recap, see the figure Fig 30.1, where we are inserting an element 15.

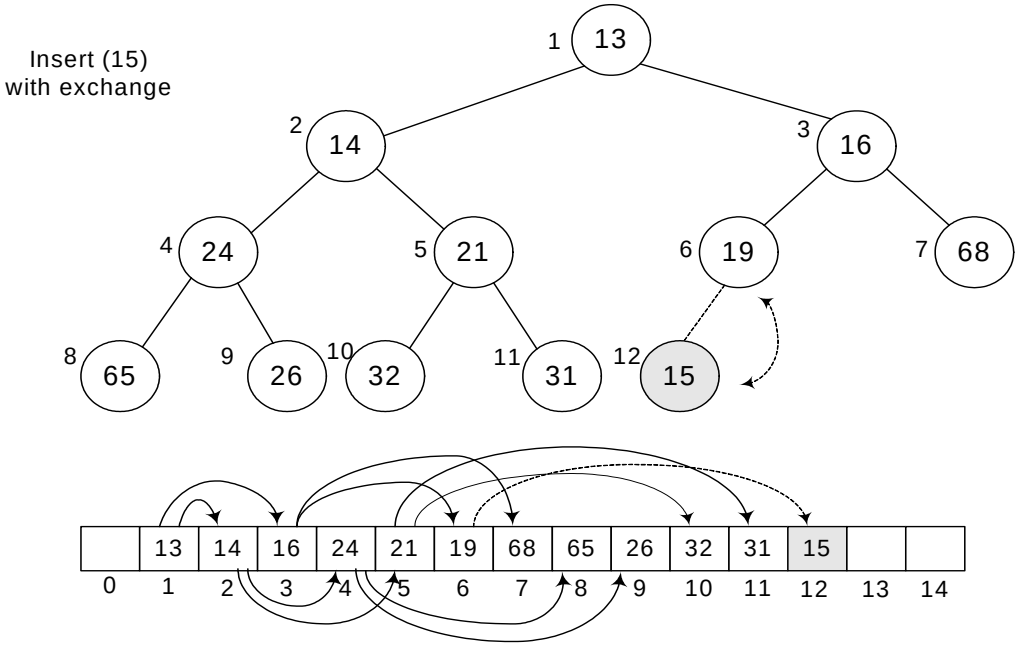


Fig 30.1

The new element 15 is inserted at the last array position 12. Being a complete binary tree, the next new node will be the left child of node 19. As node 19 is at array position 6 (or level order traversal), its left child will be $6 * 2 = 12^{th}$ position node or the value at 12^{th} position in the array. Now, we see where we will have to carry out the exchange operation. As the parent of 15, the number 19 is greater than it, the first exchange will be among 19 and 15 as shown in the above figure. After exchange, the new figure is shown in Fig 30.2.

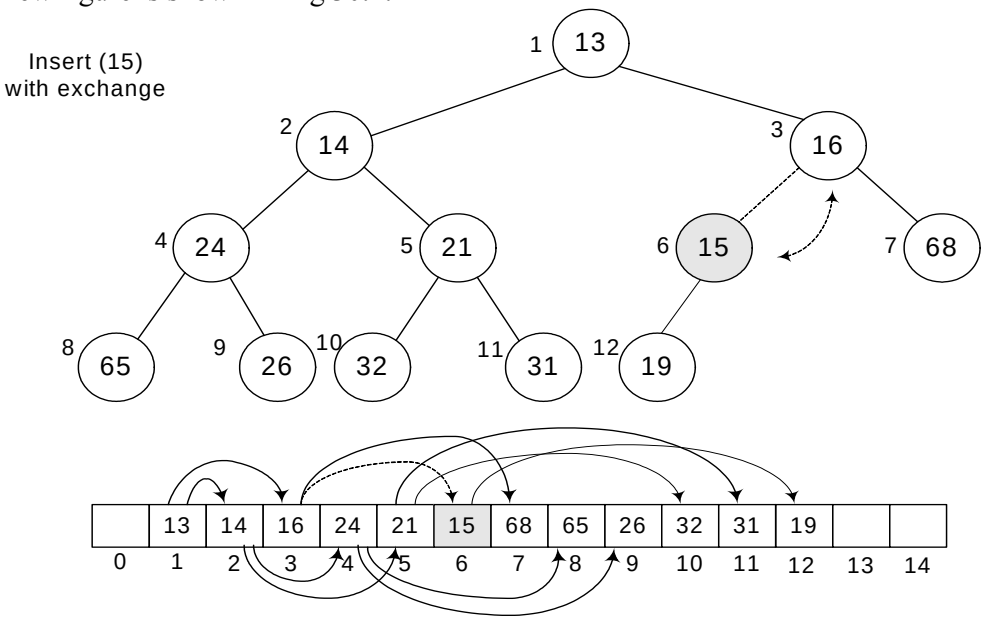


Fig 30.2

You can see that both the elements have exchanged positions i.e. 19 has come down and 15 gone up. But number 15 is still less than its parent 16, so we will have another exchange operation.

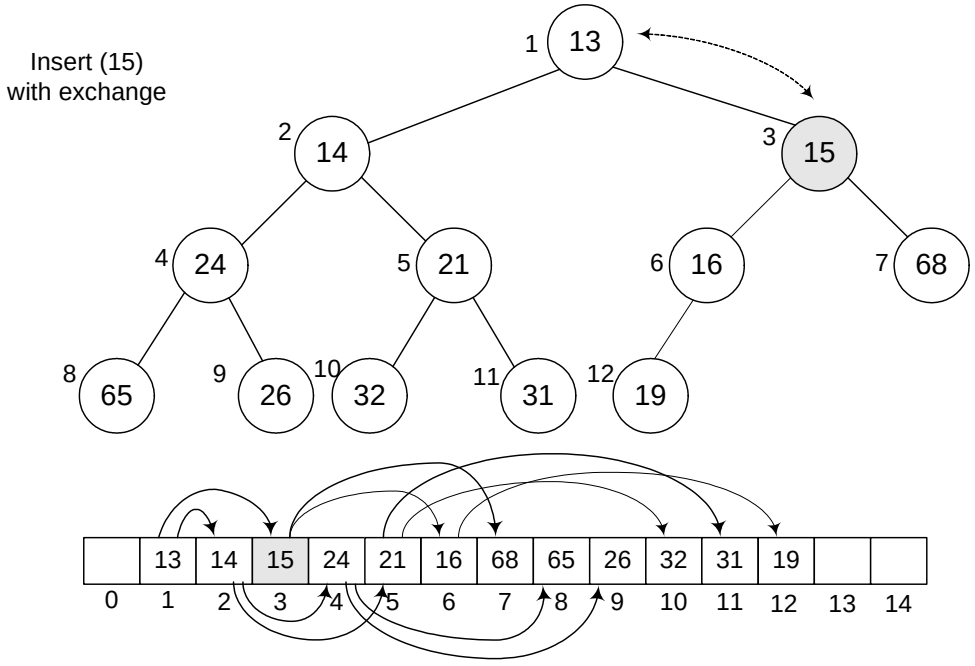


Fig 30.3

Now the new parent of 15 is 13 which is less than it. Therefore, the exchange operation will stop here as 15 has found its destination in the tree. This new tree is not violating any condition of heap order as witnessed before insertion of 15. It has become min-heap again while maintaining its status as a complete binary tree.

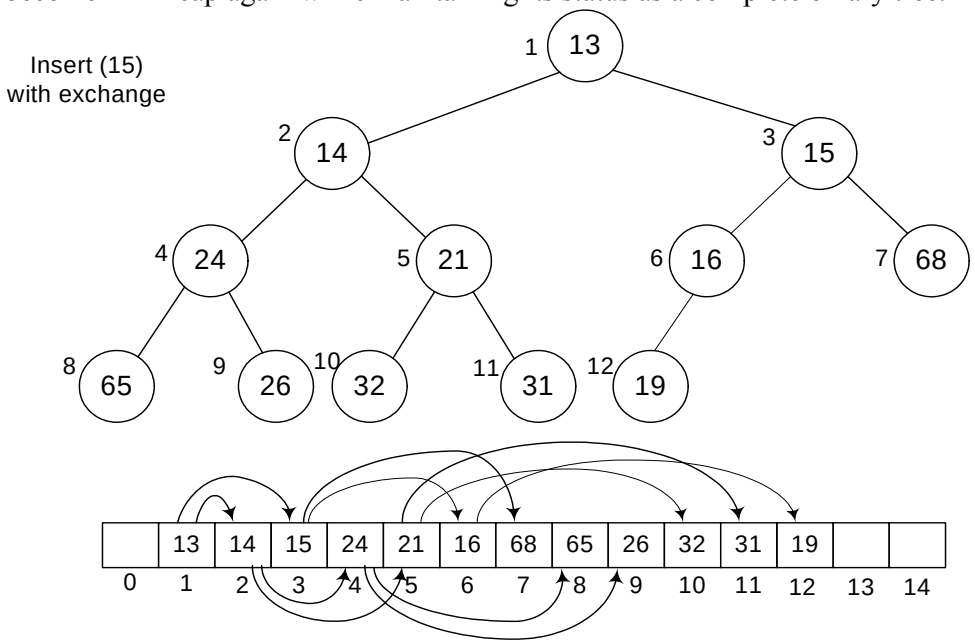


Fig 30.4

A view of the path shows that the number 15 has gone through to find its final destination in the tree. It has only affected the right sub-tree's left branch that was containing 16, 19 and 15. It has not affected any other branch. This is not a binary search tree. Here we are only fulfilling one condition that the parent value should be

less than that of its two-children. These exchanges will not take long as we did mathematically that a tree containing N nodes; can go to $\log_2 N$ level maximum. You know, when we built binary tree for balancing, it had turned into a linked list. Here the case is different. the number of levels in complete binary tree will be around $\log_2 N$ while building a complete binary tree.
Now, we should be clear that a new element can go up to what maximum level in the tree. If the number is greater than all parent nodes, it will be already at its destination place, needing no exchange. If we insert a number smaller than all parent nodes, the exchange operation reaches to the *root* of the tree. But this exchange operation's intensity does not increase from the one we saw in our previous case of linked lists.

Deleting from a Min-Heap (deleteMin)

Now, we will see how the delete operation is executed in the heap. We do a lot of insert and delete (removal from the data structure) operations especially when heap is used in the priority queue. Normally the delete operation is a bit complex. However, in this case, it is quite easy-to-do.
We want to write a function *deleteMin(),* which will find and delete the minimum number from the tree.

– *Finding the minimum is easy; it is at the top of the heap.*

See the heap below. It is a min-heap. Now the question arises where the minimum number will be lying. It is clear from the definition of min-heap that it should be in the *root* of the tree. So finding the minimum is very easy.

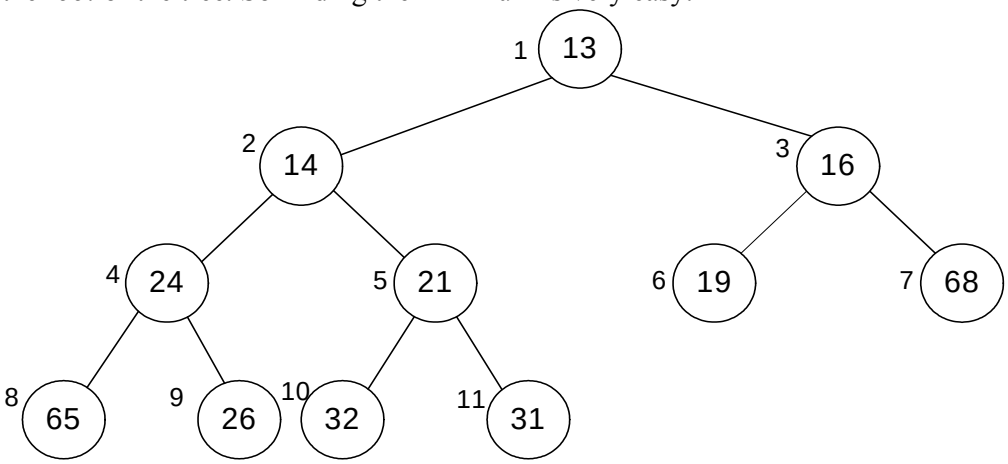


Fig 30.5

To understand it better, consider the tree form first, instead of thinking about the array. Remember, we have implemented the complete binary tree with the help of an array for the sake of efficiency. It is not necessary to implement it this way. We can also implement it with pointers. However, heap is normally implemented through array.
Coming back to the deletion, it is necessary to understand that

– *Deletion (or removal) causes a hole which needs to be filled.*

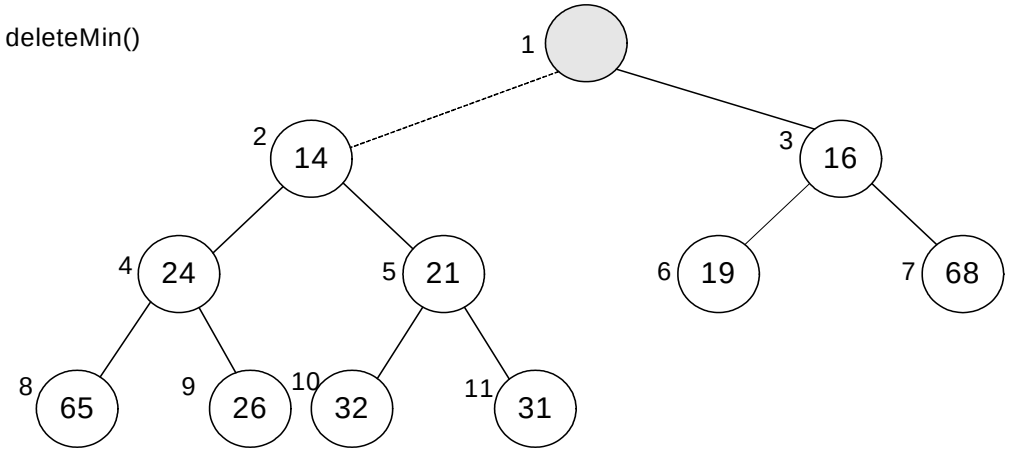


Fig 30.6

In the above figure, we have deleted the *root* node, resulting in a hole at the *root* position. To maintain it as a complete binary tree, we have to fill this hole. Now, we will identify the appropriate candidates to fill this hole. As the parent is the minimum as compared to its right and left children, the appropriate candidates are both right and left children of the *root* node. Both children will be compared and the smaller one will take the vacant place. In the above figure, the children 14 and 16 are candidates for the vacant position. Being the minimum, 14 will be put in the hole.

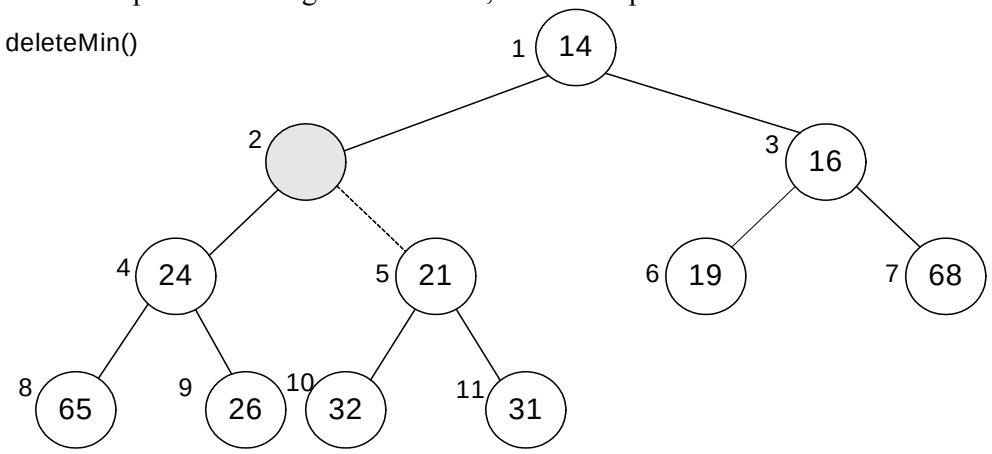


Fig 30.7

In the Fig 30.7, we can see that the 14 has been placed at the *root* position. However, the vacuum at the previous position of 14 has created a hole. To fill this hole, the same logic is applied i.e. we compare both right and left children of the hole and let the minimum take over the place. Here 24 and 21 are compared. Being the smaller one, 21 is placed at the vacant position. This way we get the latest figure of the tree as seen in the following figure. (Fig 30.8.)

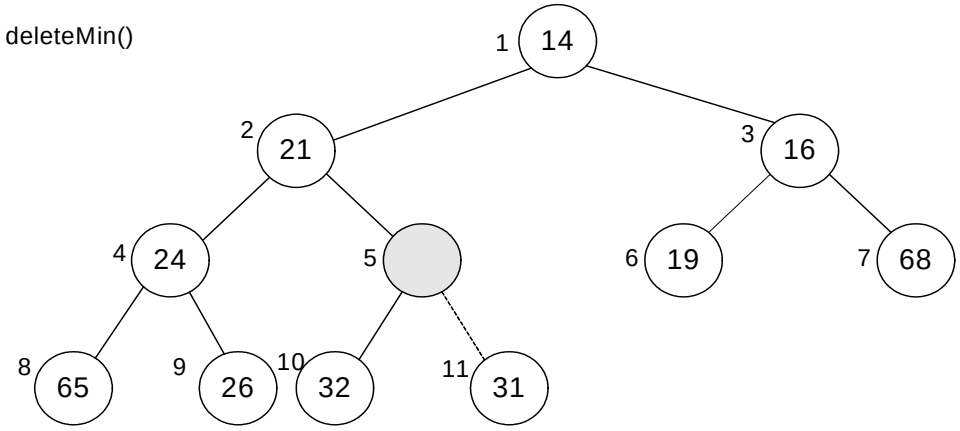


Fig 30.8

The hole is now created at previous position of 21. This time, the children- 32 and 31 are compared and being smaller, 31 takes over the place and the tree becomes as shown in the figure Fig 30.9.

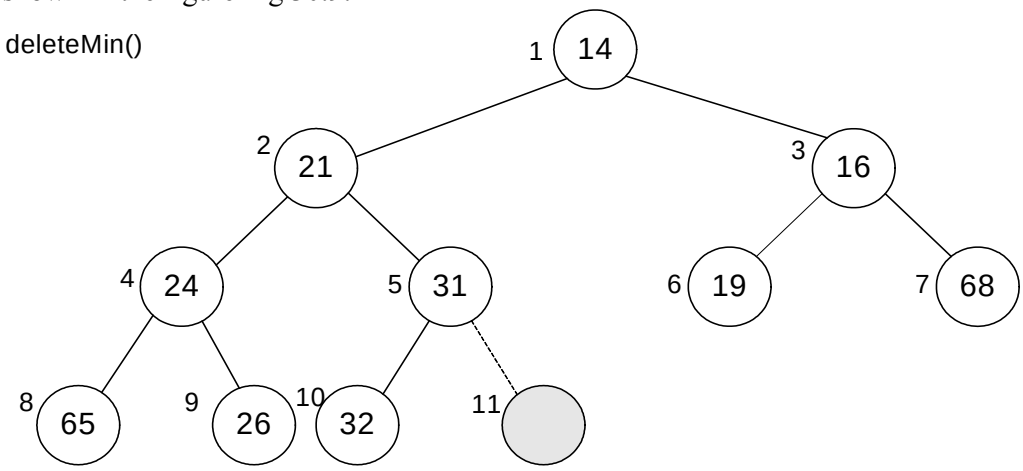


Fig 30.9

The hole has been transferred from top to the bottom. It has reached at such a position where it can be deleted.

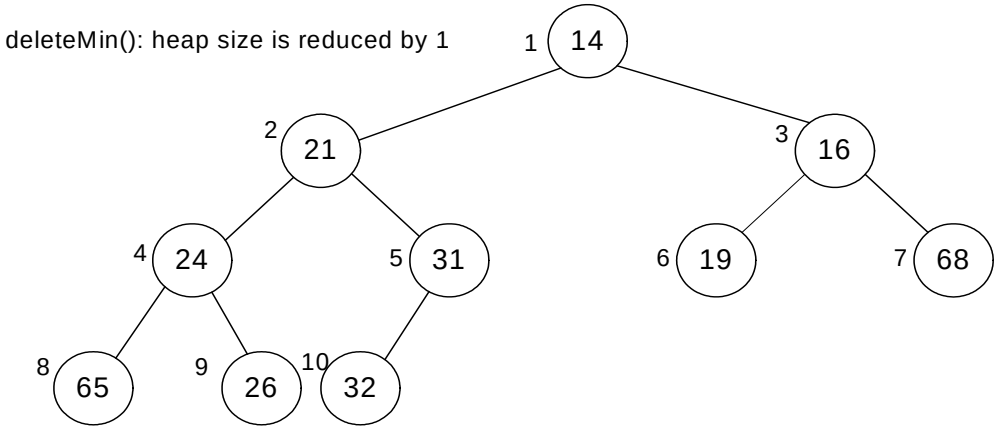


Fig 30.10

While using array to store complete binary tree, we can free array element at the end. From the above figure Fig 30.10, the last array index 11 is no more there and the node has been deleted.

We saw that the data element was inserted at the bottom of the tree and moved upwards while comparing it with the parents (following the i/2 scheme) until its destination was found. Similarly, when we deleted a node, the hole was produced. While following the definition of min-heap, the hole kept on moving from top to the bottom (following the 2i or 2i+1 scheme).

Building a Heap (buildHeap)

Now, we will see how a heap can be made and in which peculiar conditions, it should be built. We may have the data (for example, numbers) on hand, needed to used to construct the tree .These data elements may be acquired one by one to construct the tree. A programmer may face either the situations. If we consider priority queue, it is normally empty initially. Events are inserted in the queue as these are received. You can also consider priority queue in another application where data can be inserted into the queue or taken out at one time.

Let's say we have a large unsorted list of names and want to sort it. This list can be sorted with the help of the heap sort algorithm. When we l construct the heap or complete binary tree of the list, the smallest name (considering alphabet '*a'* as the smallest) in the list will take the *root* node place in the tree. The remaining names will take their places in the nodes below the *root* node according to their order. Remember, we are not constructing binary search tree but a min-heap to sort the data. After the construction of the min-heap from all the names in the list, we start taking the elements out (deleting) from the heap in order to retrieve the sorted data. The first element that is taken out would be the smallest name in the heap. The remaining heap after taking out the deleted node will be reduced by one element in size and the next name will be at the root position. It is taken out to further reduce the tree size by one. This way, the continuation of the process of taking out the elements from the heap will ultimately lead to a situation when there is no more node left in the tree.

While keeping in view the time consumption factor, can we find a better way than this? We may change the data structure or algorithm for that. There are data structures which are more efficient than heap. But at the moment, we are focusing the heap data structure. We will see, what we can do algorithmically to improve the performance, having all the data simultaneously to construct the tree.

Following are some of the important facts about building a heap.

- *Suppose we are given as input N keys (or items) to build a heap of the keys.*
- *Obviously, this can be done with N successive inserts.*
- *Each call to insert will either take unit time (leaf node) or log_2N (if new key percolates all the way up to the root).*
- *The worst time for building a heap of N keys could be $Nlog_2N$.*
- *It turns out that we can build a heap in linear time.*
- Suppose we have a method *percolateDown*(*p*) that moves down the key in node *p* downwards.
- This is what happens in *deleteMin*.We have used the word *keys* in the first point.

This is the same concept as that of the database keys, used to identify information uniquely. Using the example of telephone directory (also discussed in a number of previous lectures), we see that it contains the person name (first, middle and last), address (street address, district and postal code) and telephone number. All these information together form a data record. This set of information fields (record) will always be together. The key item here can be the unique name of the person or the telephone number.

Consider that we have N number of items i.e. names. One way is to call insert for every element one by one. For N elements, it will be N number of calls obviously. Being an iterative process, this insertion technique can be executed in a loop fashion.

Let's talk about the time taken to insert a node in the tree. It depends on the value to be inserted and the values already present in the tree. The number of exchanges can be variable. Therefore, the insertion time is also variable. Each call will be a unit time (in

case of leaf node) or $\log_2 N$ (if new key percolates all the way up to the root node). For the whole tree, the worst time for N keys would be $N\log_2 N$.

Next point tells about the possibility of reducing the time from $N\log_2 N$ and it can turn out to be a linear time. This is our target and we want to achieve it. Let's talk about the ways to achieve the linear time.

The word *percolate* above means something is going upward from downward. This terminology is coming from coffee maker, where the hot water moves upwards. This phenomenon happens at the time of insertion. On the contrary, in delete operation, the hole moves towards the bottom of the tree. You must be remembering that we discussed about insertion in the heap without making exchanges and by finding the destination position first before making any change. Similarly in case of deletion, the hole moves downward while the data moves upwards. Let's write a method for the delete operation *percolateDown*(*p*). This method will find which child of the hole will be moved upwards and the movement of hole to the bottom of the tree.

We will see, using this method, how can we built a heap that will be take lesser time than $N\log_2 N$.

Initial data (N = 15)

	65	31	32	26	21	19	68	13	24	15	14	16	5	70	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fig 30.11

As shown in the figure Fig 30.11, we are given a list of 15 elements and asked to construct a heap of them. The numbers are stored in an array, starting from index (position) 1. This start of array from position 1 is to make the tree construction easier. It will be clear in a moment. Contrary to the previous situation when array was used to store heap or complete binary tree, we should now think what would be the picture of complete binary tree out of it. It may seem complex. But actually it is quite easy. You start from the very first element in the array i.e. the *root* of the tree. The children of root are present at 2i and 2i+1, which in this case, are at positions 2(1) = 2 and 2(1)+1=3. The children nodes for the node at position 2 are at positions 2(2)=4 and 2(2)+1=5. Similarly, the children nodes for the node at position 3 are at positions 6 and 7. Apply this logic to the whole of this array and try to construct a tree yourself. You should build a tree as given in the figure Fig 30.12.

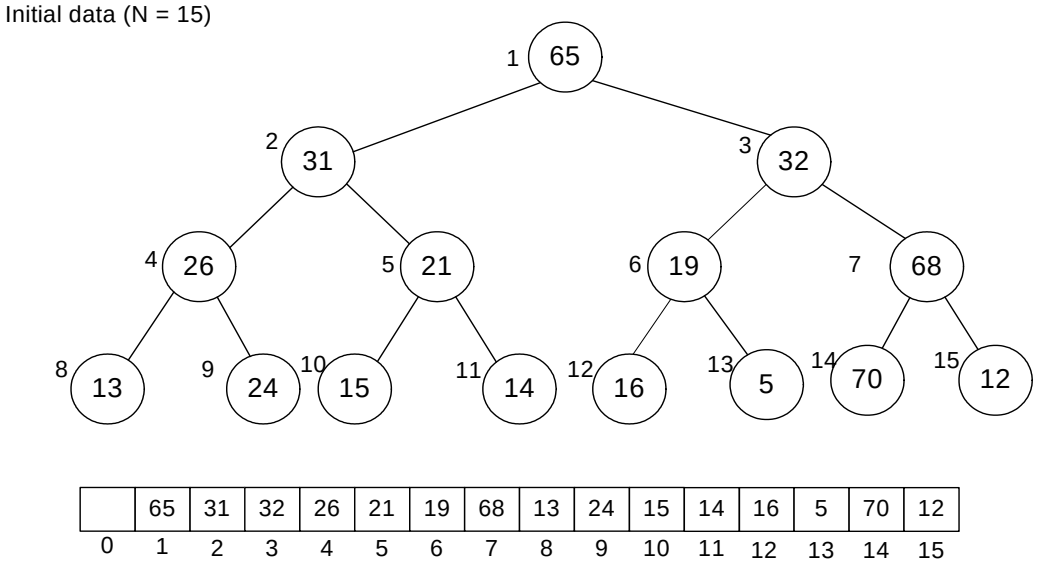


Fig 30.12

Is this tree binary one? Yes, every node has only two left and right children.
Is it a complete binary tree? It surely is as there is no node that has missing left or right child.
The next question is; is it a min-heap. By looking at the root node and its children, as the root node is containing 65 and the children nodes are containing 31 and 32, you can abruptly say that no, it is not a min-heap.
How can we make min-heap out of it? We may look for pe*rcolateDown*(*p*).method to convert it into a min-heap. As discussed above, the *percolateDown*(*p*).moves the node with value *p* down to its destination in the min-heap tree orders. The destination of a node, can be its next level or maximum the bottom of the tree (the leaf node). The node will come down to its true position (destination) as per min-heap order and other nodes (the other data) will be automatically moving in the upward direction.

- *The general algorithm is to place the N keys in an array and consider it to be an unordered binary tree.*
- *The following algorithm will build a heap out of N keys.*

```
for( i = N/2; i > 0; i-- )
    percolateDown(i);
```

A close look on the above loop shows that it is starts from N/2 position and goes down to 0. The loop will be terminated, once the position 0 is reached. Inside, this loop is a single function call *percolateDown(i)*. We will try to understand this loop using figures.

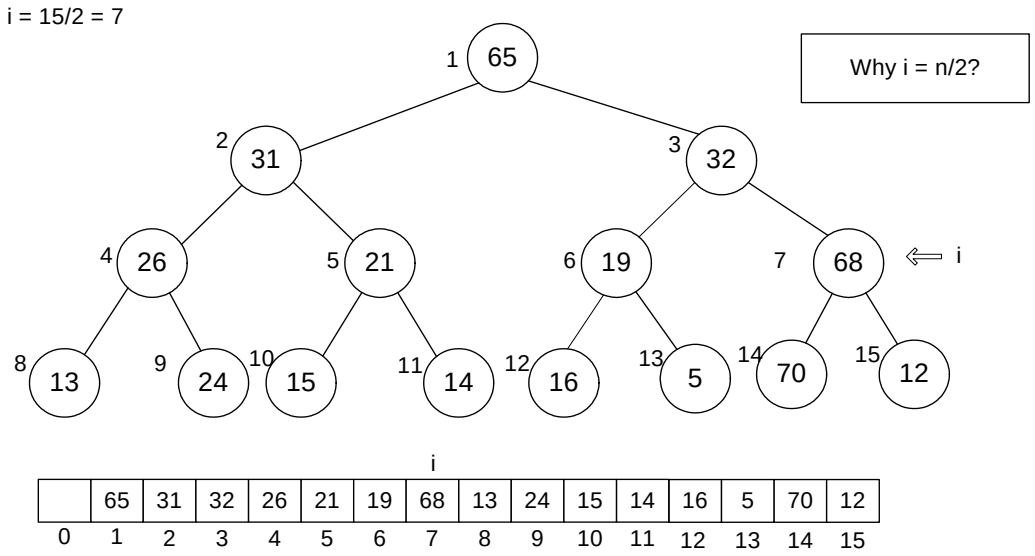


Fig 30.13

You can see the top left of the figure Fig 30.13. It is given i = 15/2 = 7. This is the initial value of i .The value in the array at position 7 is 68. In our discussion, we will interchangeably discuss array and binary tree. The facts arising in the discussion would be applicable to both. Considering this position 7, we will try to build min-heap below that. We know that for position 7, we have children at positions 2(7)=14 and 2(7)+1=15. So as children of the number 68 (which is the value at position 7), we have 70 and 12. After applying the *percolateDown(i)*, we get the tree as shown in Fig 30.14.

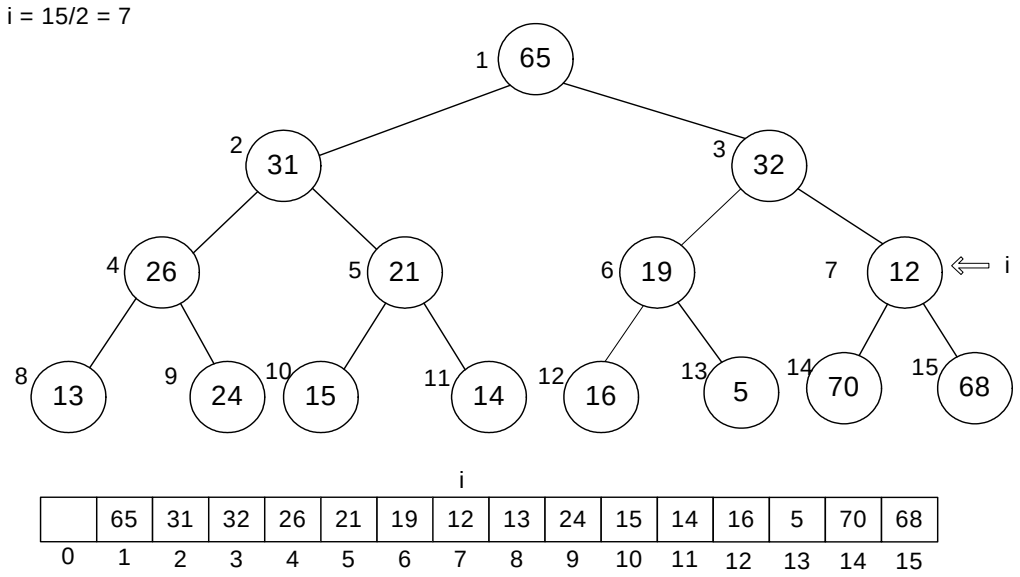


Fig 30.14

You can see in the figure that 12 has moved upward and 68 has gone down. Now, what about this little tree starting from 12 and below it is min-heap. Surely, it is.
Next, we go for second iteration in the loop, the value of i is decremented and it becomes 6.

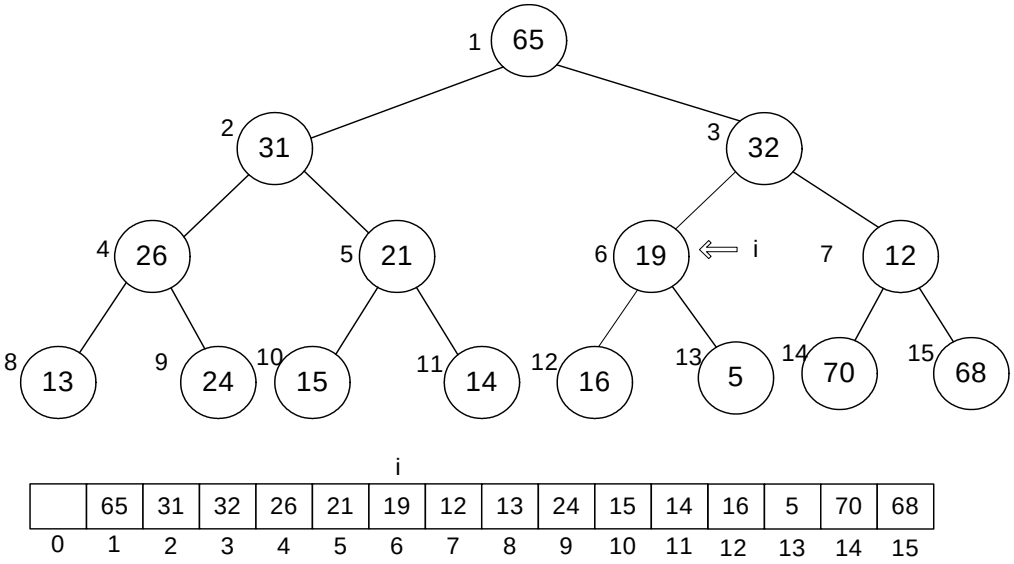


Fig 30.15

The node at position 6 is 19. Here the method *percolateDown(i)* is applied and we get the latest tree as shown in Fig 30.16.

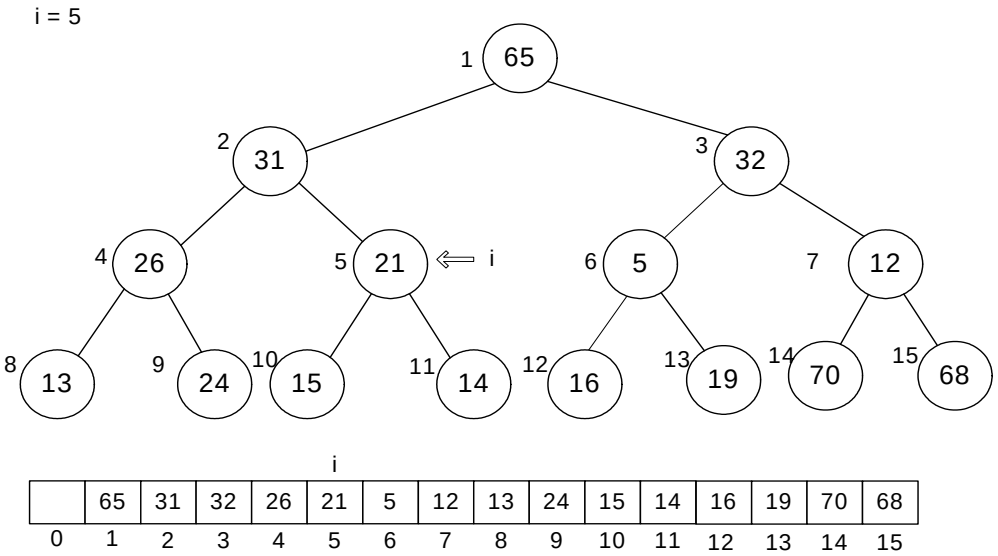


Fig 30.16

The node 5 has come upward while the node 19 has moved downward. Its value of has decremented. It is now ready for next iteration. If we see the positions of i in the tree, we can see that it is traveling in one level of the tree, which is the second last level.

The question might have already arisen in your minds that why did we start i by N/2 instead of N. You think about it and try to find the answer. The answer will be given in the next lecture. We will continue with this example in the next lecture. You are strongly encouraged to complete it yourself before the next lecture.

Data Structures

Lecture No. 31

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 6
6.3.3, 6.3.4

Summary

- BuildHeap
- Other Heap methods
- C++ Code

BuildHeap

In the previous lecture, we discussed about the *BuildHeap* method of heap abstract data structure. In this handout, you are going to know why a programmer adopts this method. Suppose we have some data that can be numbers or characters or in some other form and want to build a *min-Heap* or *max-Heap* out of it. One way is to use the *insert()* method to build the heap by inserting elements one by one. In this method, the heap property will be maintained. However, the analysis shows that it is *NlogN* algorithm i.e. the time required for this will be proportional to *NlogN*. Secondly, if we have all the data ready, then it will be better to build a heap at once as it is a better option than *NlogN*.

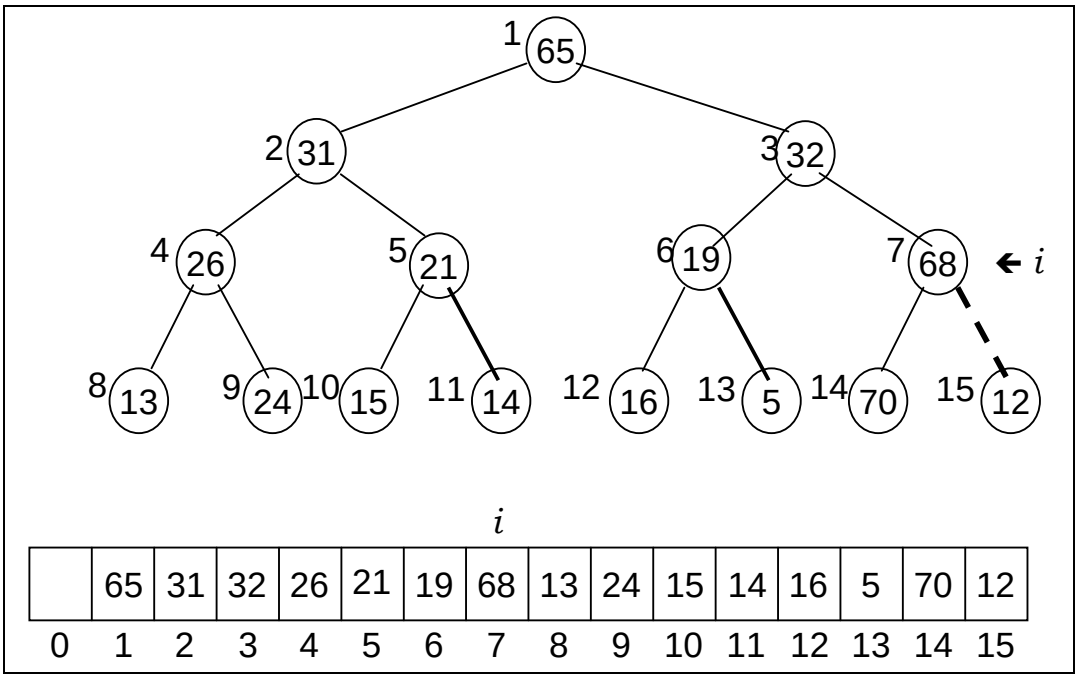
In the delete procedure, when we delete the root, a new element takes the place of root. In case of *min-heap*, the minimum value will come up and the larger elements will move downwards. In this regard, percolate procedures may be of great help. The *percolateDown* procedure will move the smaller value up and bigger value down. This way, we will create the heap at once, without calling the *insert()* method internally. Let's revisit it again:

- The general algorithm is to place the N keys in an array and consider it to be an unordered binary tree.
- The following algorithm will build a heap out of N keys.

```
for ( i = N/2; i > 0; i-- )
        percolateDown(i);
```

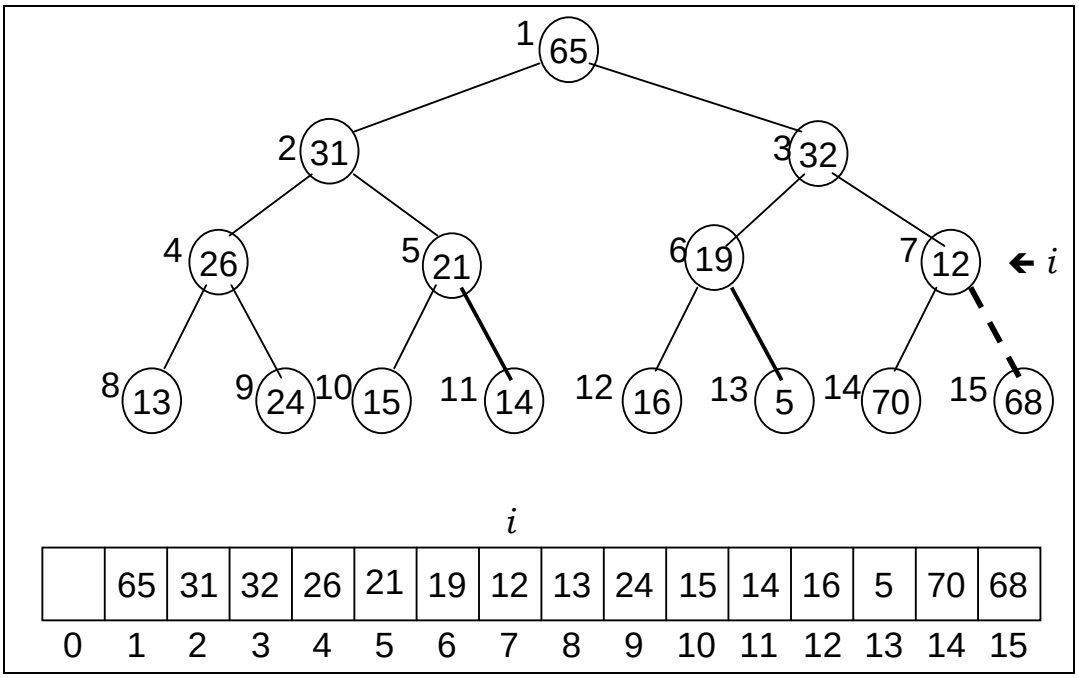
For min heap we'll use percolateUp(i);

Suppose, there are N data elements (also called as N Keys). We will put this data in an array and call it as a binary tree. As discussed earlier, a complete binary tree can be stored in an array. We have a tree in an array but it is not a heap yet. It is not necessary that the heap property is satisfied. In the next step, we apply the algorithm. Why did we start the *i* from *N/2*? Let's apply this algorithm on the actual data. We will use the diagram to understand the *BuildHeap*. Consider the diagram given below:

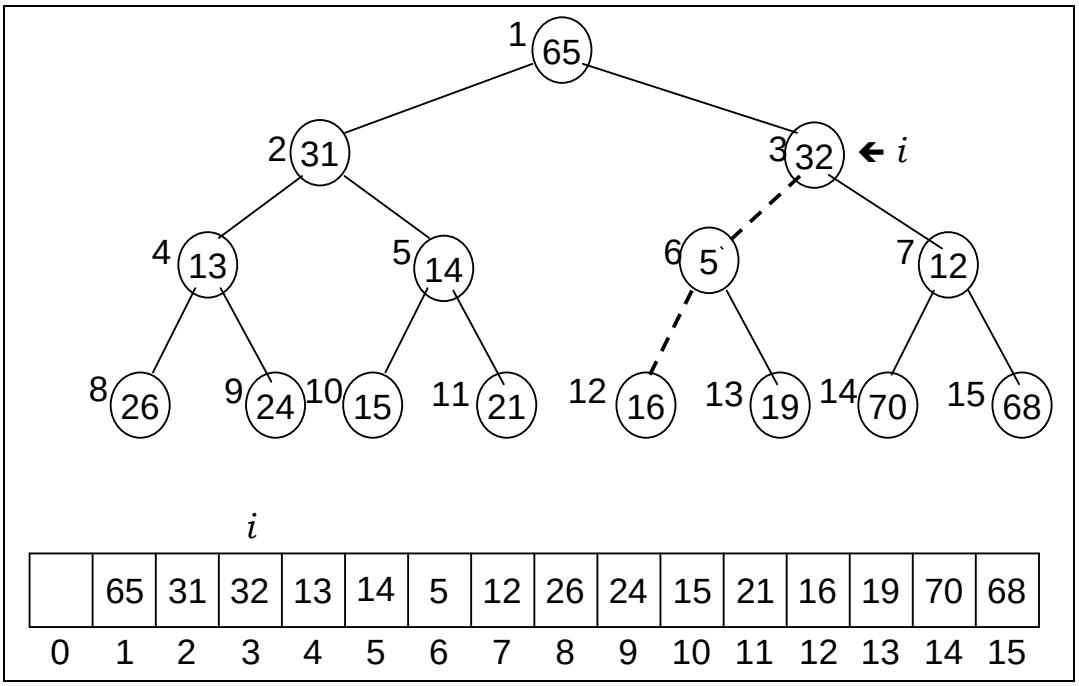


In the above diagram, there is an array containing the elements as 65, 31, 32, 26, 21, 19, 68, 13, 24, 15, 14, 16, 5, 70, 12. We have all the data in this array. The zeroth element of the array is empty. We kept it empty intentionally to apply the *2i*, *2i+1* scheme. We will take this array as a complete binary tree. Let's take the first element that is 65, applying the *2i* and *2i+1* formula. This element has 31 and 32 as its children. Take the second element i.e. 32 and use the formula *2i* and *2i+1*. Its children are 26 and 21. We can take the remaining elements one by one to build the tree. The tree is shown in the above figure. This tree is not a *min-heap* as it has 65 as root element while there are smaller elements like 13, 15, 5 as its children. So, this being a binary tree has been stored in an array.

Let's think about the above formula, written for the heap building. What will be the initial value of *i*. As the value of N is 15, so the value of i *(i=N/2)* will be 7 (integer division). The first call to *percolateDown* will be made with the help of the value of *i* as 7. The element at 7th position is 68. It will take this node as a root of subtree and make this subtree a minimum heap. In this case, the left child of node 68 is 70 while the right child of node 68 is 12. The element 68 will come down while the element 12 moves up.



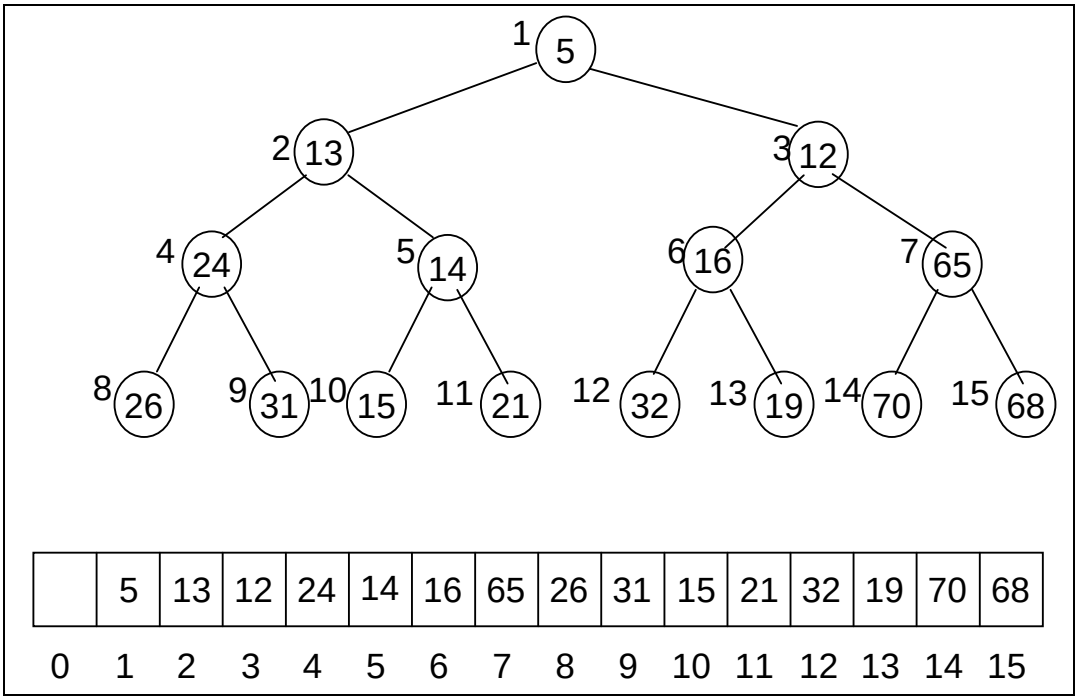
Look at this subtree with 12 as the root node. Its left and right children are smaller than root. This subtree is a minimum heap. Now in the loop, the value of *i* is decreased by 1, coming to the mark of 6. The element at 6th position is 19 that has been passed to *percolateDown* method. It will convert this small tree into a minimum heap. The element 19 is greater than both of its children i.e.16 and 5. One of these will move up. Which node should move up? The node 5 will move up while node 19 will move down. In the next repetition, the value of *i* becomes 5, i.e. the element 21. We apply the *percolateDown()* method again and convert it into minimum heap. Now the subtree with node 26 will be converted into a minimum heap.



Here the value of *i* is 3. The element at 3rd position is 32. You can see that the level of the node has also been changed. We have moved one level up. Now the *percolateDown* method will take this node as the root of the tree. It needs to move the minimum value at this position. The minimum value is 5 so the values 32 and 5 are going to be exchanged with each other. Now 32 becomes the parent of node 16 and node 19 which are smaller values. Therefore, it will further move down to node 16. The dotted line in the above figure shows the path of nodes that will be replaced. This subtree has been converted into the *min-heap*.

Now, the value of *i* is 2 while the element is 31. It will change its position with 13 first and then with 24. So the node 31 will move down and become a leaf node. You have seen that some nodes are moving up and some moving downwards. This phenomenon is known as percolate. Therefore we have named this method as *percolateDown* as it makes big values move down and smaller values upward.

Finally the value of *i* is 1. The element at first position is 65. As it is not the smallest value, so it will move down and the smallest value will move up. It will move to the position 7 as it is greater than 68 and 70. The node 5 will move to the top of the tree. The final figure is shown below:



Is this a minimum heap? Does the heap property satisfy or not? Let's analyze this tree. Start from the root of the tree. Is the value of the root is smaller than that of its left and right children? The node 5 is smaller than the node 13 and node 12. Check this on the next level. You will see that the values at each node are smaller than its children. Therefore it satisfies the definition of the *min-heap*.

Now we have to understand why we have started the 'for loop' with *i = N/2*. We can start *i* from N. As *i* moves from level to level in the tree. If we start *i* from N, it will

start from the right most leaf node. As long as *i* is l less than N/2 it will remain on leaf nodes level. Is the leaf node alone is *min-heap* or not? Yes it is. As it does not have any left or right child to compare that it is smaller or not. All the leaf nodes satisfy the *min-heap* definition. Therefore we do not need to start with the leaf nodes. We can do that but there is no reason to do that. For efficiency purposes we start *i* from N/2. This is the one level up to the leaf node level. We applied the *percolateDown* method on each node at this level. Due to this some nodes move down and some nodes move up. We apply this on each level and reaches at the root. In the end we have a minimum heap.

If we have data available, the use of *BuildHeap* method may be more appropriate as compared to *insert* method. Moreover, *BuildHeap* method takes less time. How can we prove that the *BuildHeap* method takes less time? Some mathematical analysis can help prove it. As *insert* method is *NlogN* and *BuildHeap* method should be better than that.

Other Heap Methods

Let's have a look on some more methods of heap and see the C++ codes.

decreaseKey(p, delta)

This method lowers the value of the key at position 'p' by the amount 'delta'. Since this might violate the heap order, so it (the heap) must be reorganized with percolate up (in *min-heap*) or down (in *max-heap*).

This method takes a pointer to the node that may be the array position as we are implementing it as an array internally. The user wants to decrease the value of this node by delta. Suppose we have a node with value 17 and want to decrease it by 10. Its new value will be 10. By decreasing the value of a node, the heap order can be violated. If heap order is disturbed, then we will have to restore it. We may not need to build the whole tree. The node value may become smaller than that of the parents, so it is advisable to exchange these nodes. We use *percolateUp* and *percolateDown* methods to maintain the heap order. Here the question arises why we want to decrease the value of some node? The major use of heap is in priority queues. Priority queues are not FIFO or LIFO. The elements are taken out on some key value, also known as priority value. Suppose we have some value in the priority queue and want to decrease its priority. If we are using heap for the priority queue, the priority of the some elements can be decreased so that it could be taken out later and some other element that now has higher priority will be taken out first. You can find many real life examples to understand why we need to increase or decrease the priority of elements.

One of these examples is priorities of processes. You will read about this in the operating system course. You can see the priorities of the processes using the task manager. Task manager can be activated by pressing Ctrl+Alt+Del. Under the process tab, we may see the priority of each process. Normally, there is one processor in a computer and there is lot of processes running in it. How can we decide which process should be given the CPU time and how much time should be given to this process? We will cover all of these topics in the operating system course. Suppose we have 25

processes with different priorities. How can we decide which process should be given the CPU time? We gave the CPU to the highest priority process for one second. We can schedule this by using the priority queue. For some reason, we decrease the priority of some process. Now its turn will come later. We may increase the priority of some process so that it should be given the CPU time. Therefore, we adopt the increase and decrease methods. If we are implementing the priority queue by using the heap, the increase or decrease method of priority queue will use the increase and decrease method of heap internally. Using this method, the turn of the element will be according to the new priority.

increaseKey(p, delta)

This method is the opposite of *decreaseKey*. It will increase the value of the element by delta. These methods are useful while implementing the priority queues using heap.

remove(p)

This method removes the node at position p from the heap. This is done first by *decreaseKey(p, ə)* and then performing *deleteMin()*. First of all, we will decrease the value of the node by ə and call the method *deleteMin*. The *deleteMin* method deletes the root. If we have a *min-heap*, the root node contains the smallest value of the tree. After deleting the node, we will use the *percolateDown* method to restore the order of the heap.

The user can delete any node from the tree. We can write a special procedure for this purpose. Here we will use the methods which are already available. At first, the *decreaseKey* method will be called and value of node decreased by ə. It will result in making the value of this node smallest of all the nodes. If the value is in integers, this node will have the smallest integer. Now this node has the minimum value, so it will become the root of the heap. Now we will call the *deleteMin()* and the root will be deleted which is the required node. The value in this node is not useful for us. The ə is a mathematical notation. It is not available in the C++. Actually we want to make the minimum possible value of this node supported by the computer.

C++ Code

Now we will look at the C++ code of the *Heap* class. The objects of *Heap* may be got from this factory class. This class will contain the methods including those discussed earlier and some new ones. Heap is used both in priority queues and sorting.

We have some public and private methods in the class. Let's have a look on the code.

```
/* The heap class. This is heap.h file */

template <class eType>

class Heap
{
  public:
    Heap( int capacity = 100 );
```

```
    Void insert( const eType & x );
    Void deleteMin( eType & minItem );
    Const eType & getMin( );
    Bool isEmpty( );
    Bool isFull( );
    int Heap<eType>::getSize( );

private:
    int  currentSize; // Number of elements in heap
    eType* array;     // The heap array
    int    capacity;

    void percolateDown( int hole );
};
```

We may like to store different type of data in the heap like integers, strings, floating-point numbers or some other data type etc. For this purpose, template is used. With the help of template, we can store any type of object in the heap. Therefore first of all we have:

```
template <class eType>
```

Here *eType* will be used as a type parameter. We can use any meaningful name. Then we declare the *Heap* class. In the public part of the class, there is a constructor as given below.

```
Heap( int capacity = 100 );
```

We have a parameter *capacity* in the constructor. Its default value is 100. If we call it without providing the parameter, the capacity will be set to 100. If we call the constructor by providing it some value like 200, the capacity in the *Heap* object will be 200.

Next we have an insert method as:

```
void insert( const eType & x );
```

Here we have a reference element of *eType* which is of constant nature. In this method, we will have the reference of the element provided by the caller. The copy of element is not provided through this method. We will store this element in the *Heap*.

Similarly we have a delete method as:

```
void deleteMin( eType & minItem );
```

This method is used to delete the element from the heap.

If we want to know the minimum value of the heap, the *getMin* method can be useful.

The signatures are as:

```
const eType & getMin( );
```

This method will return the reference of the minimum element in the *Heap*. We have some other methods in the *Heap* class to check whether it is empty or full. Similarly, there is a method to check its size. The signatures are as:

```
bool isEmpty( );
bool isFull( );
int Heap<eType>::getSize( );
```

When will *Heap* become full? As we are implementing the *Heap* with the help of an array, the fixed data type, so the array may get full at some time. This is the responsibility of the caller not to insert more elements when the heap is full. To facilitate the caller, we can provide the methods to check whether the heap is full or not. We can call the *getSize()* method to ascertain the size of the *Heap*.

In the private part, we have some data elements and methods. At first, we have *currentSize* element. It will have the number of elements in the heap. Then we have an array of *eType*. This array will be dynamically allocated and its size depends on the *capacity* of the *Heap*. We have one private method as *percolateDown*.

This is our *.h* file, now we will see the implementation of these methods that is in our *.cpp* file. The code is as under:

```
/* heap.cpp file */

#include "Heap.h“

template <class eType>
Heap<eType>::Heap( int capacity )
{
   array = new eType[capacity + 1];
   currentSize=0;
}

/* Insert item x into the heap, maintaining heap order. Duplicates
are allowed. */

template <class eType>
bool Heap<eType>::insert( const eType & x )
{
   if( isFull( ) ) {
      cout << "insert - Heap is full." << endl;
      return 0;
   }
   // Percolate up
   int hole = ++currentSize;
   for(; hole > 1 && x < array[hole/2 ]; hole /= 2)
```

```
        array[ hole ] = array[ hole / 2 ];
    array[hole] = x;
}
template <class eType>
void Heap<eType>::deleteMin( eType & minItem )
{
    if( isEmpty( ) ) {
        cout << "heap is empty.“ << endl;
                return;
    }

    minItem = array[ 1 ];
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );
}
```

We include the *heap.h* file before having the constructor of the heap. If we do not provide the value of *capacity*, it will be set to 100 that is default. In the constructor, we are dynamically creating our array. We have added 1 to the *capacity* of the array as the first position of the array is not in use. We also initialize the *currentSize* to zero because initially *Heap* is empty.

Next we have *insert* method. This method helps insert item *x* into the heap, while maintaining heap order. Duplicates are allowed. Inside the *insert* method, we will call the *isFull()* method. If the heap is full, we will display a message and return 0. If the heap is not full, we will insert the element in the heap and take a variable *hole* and assign it *currentSize* plus one value. Then we will have a ‘for loop’ which will be executed as long as *hole* is greater than 1. This is due to the fact that at position one, we have root of the array. Secondly, the element which we want to insert in the array is smaller than the *array[hole/2]*. In the loop, we will assign the *array[hole/2]* to *array[hole]*. Then we divide the hole by 2 and again check the loop condition. After exiting from the loop, we assign *x* to the *array[hole]*. To understand this insert process, you can get help from the pictorial diagrams of the *insert*, discussed earlier. We have a complete binary tree stored in an array and placed this new element at the next available position in the array and with respect to tree it was left most leaf node. This new node may have smaller value so it has to change its position. If we are building the *min-heap*, this value should be moved upward. We will compare this with the parent. If the parent is bigger, it will move down and child will move up. Using the array notation, the parent of a child is at *i/2* if child is at *i* position. We will first try to find the final position of this new node and exchange the values. You might want to remember the example of insert discussed in some earlier lecture. We have shown a hole going up. Now the swapping function generally contains three statements and is not an expensive operation. But if we perform swapping again and again, it may cost some time. Therefore we will first find the final position of the new node and then insert it at that position. In the ‘for loop’ we are finding the final position of the node and the hole is moving upwards. In the statement *array[hole] = array[hole/2];* we are moving the parent down till the time final position of the new node is achieved. Then we insert the node at its final position. It is advisable to execute this code only for actual data and see how it works.

The next method is *deletMin*. First of all, it calls the *isEmpty()* method. If heap is empty, it will display a message and return. If the heap is not empty, it will delete the node. As the minimum value lies at the first position of the array, we save this value in a variable and store the *currentSize* at the first position of the array. At the same time, we reduce the *currentSize* by one as one element is being deleted. Then we call the *percolateDown* method, providing it the new root node. It will readjust the tree. We put the last element of the array which is the right most leaf node of the tree as the root element will be deleted now. With this operation, the heap order can be disturbed. Therefore we will call the *percolateDown* method which has the ability to readjust the tree. This method will make the tree as *min-heap* again. There are some more methods which we will cover in the next lecture.

Data Structures

Lecture No. 32

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 6
6.3

Summary

- perculateDown Method
- getMin Method
- buildHeap Method
- buildHeap in Linear Time
- Theorem

In the previous lecture, we defined a heap class and discussed methods for its implementation. Besides, the insert and delete procedures were talked about. Here, we will continue with discussion on other methods of the heap class. C++ language code will be written for these procedures so that we could complete the heap class and use it to make heap objects.
We are well aware of the *deleteMin* method. In this method, *array[1]* element (a root element) is put in the *minItem* variable. In the min heap case, this root element is the smallest among the heap elements. Afterwards, the last element of the array is put at the first position of the array. The following two code lines perform this task.

```
minItem = array[ 1 ];
array[ 1 ] = array[ currentSize-- ];
```

perculateDown Method

Then we call the *perculateDown* method. This is the same method earlier employed in the build heap process. We passed it the node, say i, form where it starts its function to restore the heap order. Let's look at this *perculateDown* method. It takes the array index as an argument and starts its functionality from that index. In the code, we give the name *hole* to this array index. Following is the code of this method.

```
// hole is the index at which the percolate begins.
template <class eType>
void Heap<eType>::percolateDown( int hole )
{
   int child;
   eType tmp = array[ hole ];
   for( ; hole * 2 <= currentSize; hole = child )
   {
      child = hole * 2;
```

```
            if( child != currentSize && array[child+1] < array[ child ] )
               child++;  // right child is smaller
            if( array[ child ] < tmp )
               array[ hole ] = array[ child ];
            else break;
         }
         array[ hole ] = tmp;
      }
```

In the code of this function, it declares an integer variable named *child* before putting the value at the index *hole* of the array in *tmp* variable. It is followed by a 'for loop', the core of this method. The termination condition of this *for* loop is *hole * 2 <= currentSize;* We know the 2i (or i * 2) formula that gives us index position of the left child of i. In the code given above, we are using *hole* instead of i. This variable i.e. *hole* is the loop variable. In other words, the termination condition means that the loop will continue as long as the left child (hole * 2) is less than or equal to the current size (currentSize) of the heap. Before looking at the third condition in which we change the value of loop variable, let's have a view of the iteration of *for* loop.

In the body of the '*for* loop', we assign the value *hole * 2* to the *child* variable. The left child of an index *i* is located at the index *2i*. So it is obvious that the *child* has the index number of the left child of *hole*. Then there is an *if* statement. In this statement, we check two conditions combined with && (logical AND) operator. This *if* statement returns TRUE only if both the conditions are TRUE. The first condition in this *if* statement checks that the *child* (index) should not be equal to the *currentSize while in* the second condition, we compare the value at index *child + 1* with the value at index *child*. We check whether the value at *child + 1* is less than the value at *child*. The value at index *child* is the left child of its parent as we have found it by the formula of 2i (here we used *hole * 2*). Thus the value at *child + 1* will be the right child of the parent (i.e. hole). Actually, we are comparing the left and right children of *hol*e. If both the conditions are TRUE, it means that right child is less than the left child. Resultantly, we increment the *child* index by 1 to set it to the right child. Now again in an *if* statement, we compare the value at index child (which is left or right child depending upon our previous check in *if* statement) with the *tmp* value. If value at *child* is less than the *tmp* value, we will put the value of index *child* at the index *hole*. Otherwise, if *tmp* value is less than the child, we exit *for* loop by using the *break* statement. Thus the value of *child* comes to the position of the parent (i.e. hole). The *hole* gets downward and *child* goes upward. The 'for loop' continues, bringing the *hole* downward and setting it to its proper position. When the *hole* reaches its position, the 'for loop' exits (it may exit by meeting the break statement) and we put the tmp value in the array at the index *hole*. In this method, at first, the final position of hole is determined before putting value in it. Here the *perculateDown* procedure ends.

Now the question arises why we are bringing the *hole* (index) downward? Why don't we exchange the values? We can execute the process of exchanging values. We will use swap procedure to do this exchange. The swap process is carried out in three statements. We have to do this swapping in the *for* loop. It is time-consuming process. Contrast to it, in the *perculateDown* method, we execute one statement in *for* loop instead of three swap statements. So it is better to use one statement instead of three statements. This increases the efficiency with respect to time.

getMin Method

We discussed this method in the previous lectures while talking about the interface of heap. Under this method, the minimum value in the heap is determined. We just try that element and do not remove it. It is like the top method of stack that gives us the element on the top of the stack but do not remove it from the stack. Similarly, the getMin method gives us the minimum value in the heap, which is not deleted from the heap. This method is written in the following manner.

```
template <class eType>
const eType& Heap<eType>::getMin( )
{
   if( !isEmpty( ) )
      return array[ 1 ];
}
```

Now we will discuss the buildHeap method.

buildHeap Method

This method takes an array along with its size as an argument and builds a heap out of it. Following is the code of this method.

```
template <class eType>
void Heap<eType>::buildHeap(eType* anArray, int n )
{
   for(int i = 1; i <= n; i++)
     array[i] = anArray[i-1];
   currentSize = n;
   for( int i = currentSize / 2; i > 0; i-- )
      percolateDown( i );
}
```

In the body of this method, at first, there is a 'for loop' that copies the argument array (i.e. anArray) into our internal array, called '*array'*. We do this copy as the array that this method gets as an argument starts from index zero. But we want that the array should start from index 1 so that the formula of 2i and 2i +1 could be used easily. In this for loop, we start putting values in the internal array from index 1. Afterwards, we set the currentSize variable equal to the number of the elements got as an argument. Next is the *for* loop that we have already seen while discussing the build of heap. In the previous lecture, we used 'N' for number of elements but here, it will be appropriate to use the currentSize, which is also the number of elements. We start this loop from the value i = currentSize / 2 and decrement the value of i by 1 for each iteration and execute this loop till the value of i is greater than 0. In this for loop, we call the method perculateDown to find the proper position of the value given as an argument to this method. Thus we find the position for each element and get a heap.
Then there are three small methods. The isEmpty method is used to check whether the heap is empty. Similarly the isFull method is used to check if the heap is full. The getSize method is used to get the size of the heap. These three methods i.e. isEmpty, isFull ans getSize are written below.

```
//isEmpty method
template <class eType>
bool Heap<eType>::isEmpty( )
{
   return currentSize == 0;
}

//isFull method
template <class eType>
bool Heap<eType>::isFull( )
{
   return currentSize == capacity;
}

//getSize method
template <class eType>
int Heap<eType>::getSize( )
{
   return currentSize;
}
```

buildHeap in Linear Time

We have seen that *buildHeap* takes an array to make a heap out of it. This process of making heap (buildHeap algorithm) works better than N $\log_2$N. We will prove it mathematically. Although our course is Data Structures, yet we have been discussing the efficiency of the data structures from the day one of this course. We talked about how efficiently a data structure uses memory storage and how much it is efficient with respect to time. In different cases, mathematics is employed to check or compare the efficiency of different data structures.

Here, we will show that the buildHeap is a linear time algorithm and is better than N $\log_2$N algorithm which is not of linear nature. Linear algorithm is such a thing that if we draw its graph, it will be a straight line with some slope. Whereas the graph of a non-linear algorithm will be like a curve. We will discuss it in detail later. To prove the superiority of the buildHeap over the N $\log_2$N, we need to show that the sum of heights is a linear function of N (number of nodes). Now we will a mathematical proof that proves that the buildHeap algorithm is better than N $\log_2$N. Now consider the following theorem.

Theorem

According to this theorem, "For a perfect binary tree of height h containing $2^{h+1} - 1$ nodes, the sum of the heights of nodes is $2^{h+1} - 1 - (h+1)$, or $N-h-1$".

This theorem is about heights of tree. Let's try to understand the concept of height. If we start from the root node and go to the deepest leaf node of the tree, the number of links we pass to go to the deepest leaf node is called the height of the tree. We have been using the term depth and level for it also. Height can also be measured with reference to the deepest leaf node at height zero and going upward to the root node. Normally we start from the root and go to the deepest level to determine the height. A perfect binary tree of height h has total number of nodes N equal to $2^{h+1} - 1$. Now according to the theorem, the sum of heights of nodes will be equal to $N - h - 1$. Let's

prove it mathematically.
In a perfect binary tree, there are 2^0 nodes at level 0, 2^1 nodes at level 1, 2^2 nodes at level 2 and so on. Generally, level *n* has 2^n nodes. Now in terms of height, in a perfect binary tree, if *h* is the height of the root node, there are 2^0 nodes at height *h*, 2^1 nodes at height *h-1*, 2^2 nodes at height *h-2* and so on. In general, there are 2^i nodes at height *h*-i. Following figure shows a perfect binary tree with the number of nodes at its heights.

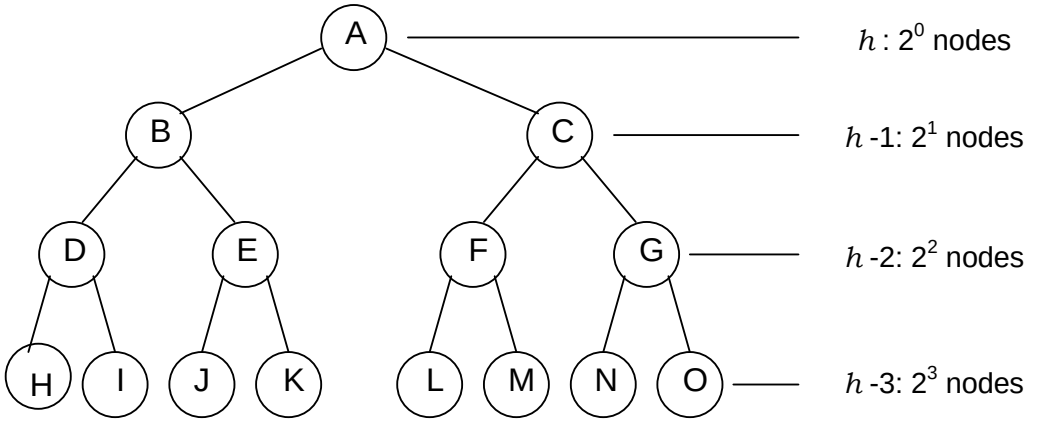


Figure 32.1: Perfect Binary Tree

In the above figure, we can see that there are 2^1 nodes at height *h-1*and 2^3 nodes at height *h-3*. The same is true about the number of nodes at other heights.
By making use of this property, we find the sum of heights of all the nodes mathematically in the following manner. Suppose *S* is the sum of the heights of all the nodes. Then

$$S = \sum 2^i (h - i), \text{ for } i = 0 \text{ to } h - 1$$
$$S = h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots\ldots + 2^{h-1} \quad (1)$$

Now multiplying by 2 the both sides of equation

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots\ldots + 2^h \quad (2)$$

Now subtracting the equation 2 from equation 1

$$-S = h - 2 - 4 - 8 - 16 - \ldots\ldots - 2^{h-1} - 2^h$$
$$S = -h + 2 + 4 + 8 + 16 + \ldots\ldots + 2^{h-1} + 2^h$$
$$S = (2^{h+1} - 1) - (h+1)$$

Here *S* is the sum of heights. Thus it proves the theorem.
As stated earlier, in a perfect binary tree of height *h,* the total number of nodes *N* is $(2^{h+1} - 1)$. *So* by replacing it in the above equation, we get $S = N - (h + 1)$.
Since a binary complete tree has nodes between 2^h and 2^{h+1}, the equation

$$S = (2^{h+1} - 1) - (h+1)$$

Can also be written as

$$S \simeq N - Log_2 (N+1)$$

In this equation, N stands for $(2^{h+1} - 1)$ while $Log_2 (N+1)$ has taken the place of $(h+1)$. This equation is in term of N (number of nodes) and there is no h i.e. height in this equation. Moreover, it also shows that with N getting larger, the $\log_2(N+1)$ term becomes insignificant and S becomes a function of N.
Suppose if N has a value 1000000, the value of log_2 (1000000) will be approximately 20. We can see that it is insignificant with the value of N i.e. 1000000. Thus we can say that S is approximately equal to N. This insignificance of $log\ N$ shows that when we build a heap from N elements, the values move up and down to maintain heap order. Thus in buildHeap case, the values traverse the tree up and down. Now the question arises what will be the maximum number of these traversals? This means that if every node has to go up and down, what will be the maximum number of level or height. We have proved that the sum of heights (i.e. S) is approximately equal to the total number of nodes (i.e. N) as N becomes larger. Now in the buildHeap, the maximum number of up down movement of values is actually the sum of heights i.e. S. This S is the upper limit of number of movements in buildHeap. As this S is equal to N, so this upper limit is N and there is not any log term involved in it. Thus buildHeap is a linear time application.

Now let's prove the previous theorem with a non-mathematical method. We know that the *height* of a node in the tree is equal to the number of edges on the longest downward path to a leaf. So if we want to find the height of a node, it will be advisable to go to the deepest leaf node in the tree by following the links (edges) and this path (the number of links traveled) to the leaf node is the height of that node. The height of a tree is the height of its root node. After these definitions, consider the following two points that prove the theorem.

- For any node in the tree that has some height h, darken h tree edges
  –Go down the tree by traversing left edge then only right edges.
- There are $N - 1$ tree edges, and h edges on right path, so number of darkened edges is $N - 1 - h$, which proves the theorem.

We will explain these two points with the help of figures.
Consider the perfect binary tree, shown in the figure below.

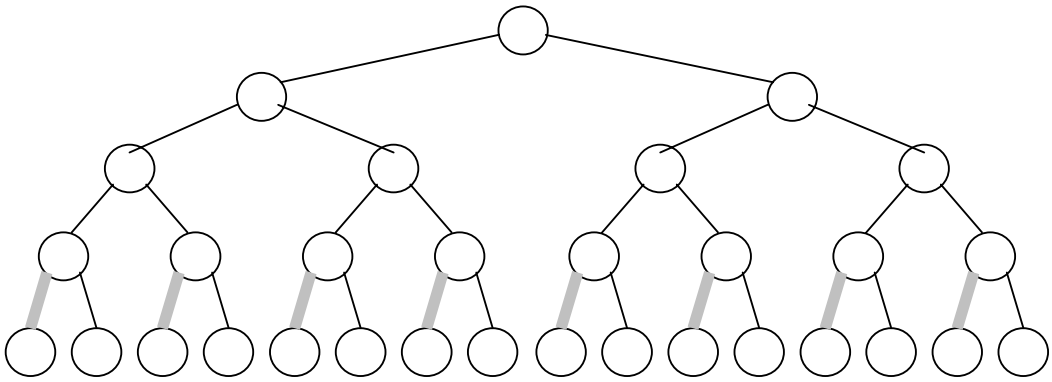


Figure 32.2: Marking the left edges for height 1 nodes

There are no values shown in the nodes. We are concerned with the heights of nodes while the values do not matter. We mark the left edges of the nodes having height 1. The leaf nodes are at height 0, so the one level above nodes has height 1, the left edges of which are marked (shown light gray in above figure).
Now we mark the first left edge and the subsequent right edge up to the leaf node for each node at height 2. These marked edges are shown in the figure below with gray color.

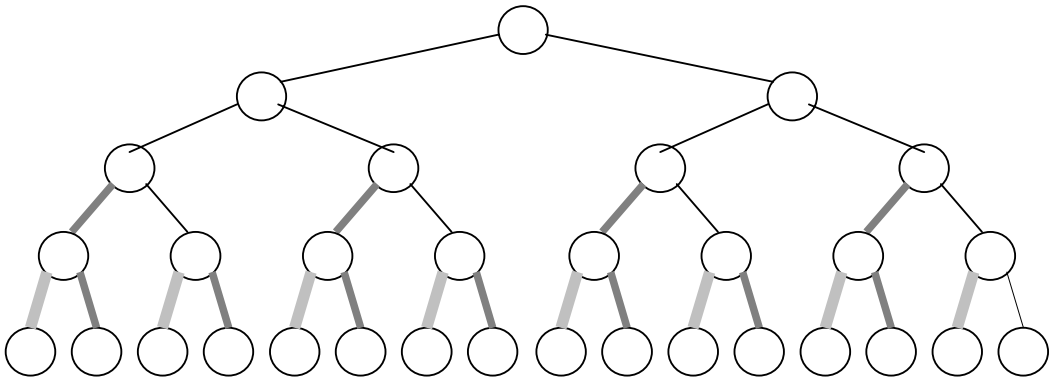


Figure 32.3: Marking the first left edge and the subsequent right edge for height 2 nodes

Similarly we go to the nodes at height 3, mark the first left edge and the subsequent right edge up to the leaf node. Thus we mark one left and then two right edges for the nodes at height 3. This is shown in the following figure. The edges marked in this step are seen as the dark lines.

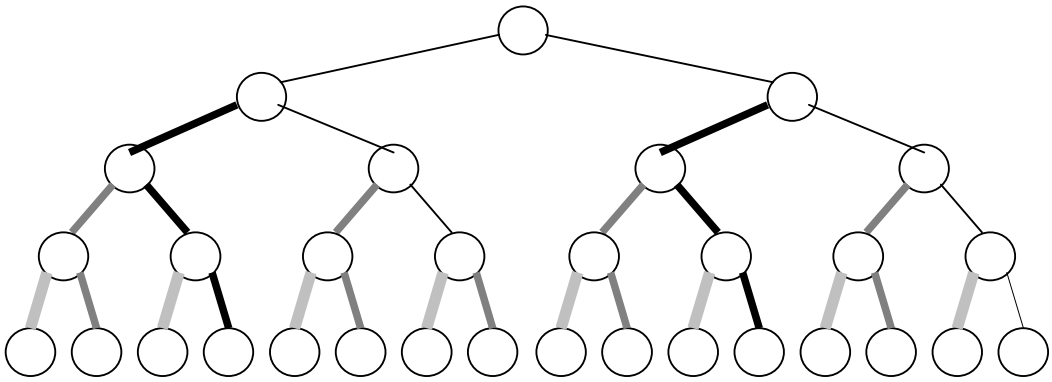


Figure 32.4: Marking the first left edge and the subsequent two right edges for height 3 nodes

Now we reach at the root whose height is 4. We mark the first left edge and the subsequent three right edges for this node. This is shown in the figure below.

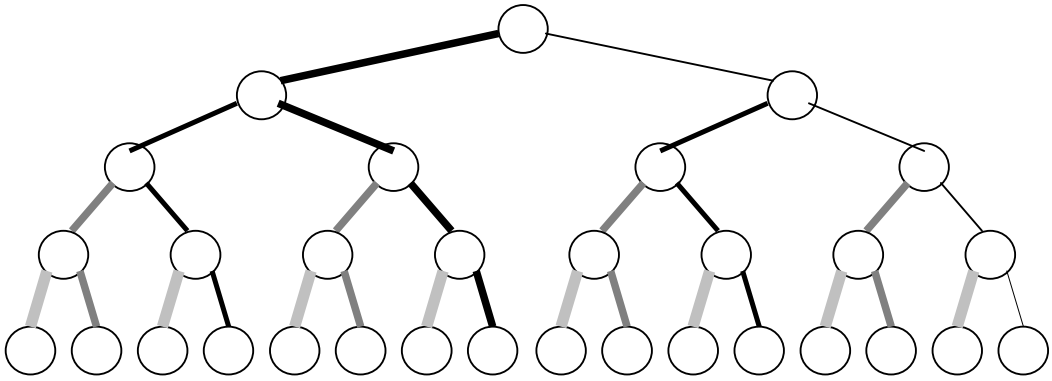


Figure 32.5: Marking the first left edge and the subsequent three right edges for height 4 nodes

Now consider the following figure. In this figure, we show all the marked edges (that we have marked in the previous steps) in gray and the non-marked edges are shown with dotted lines.

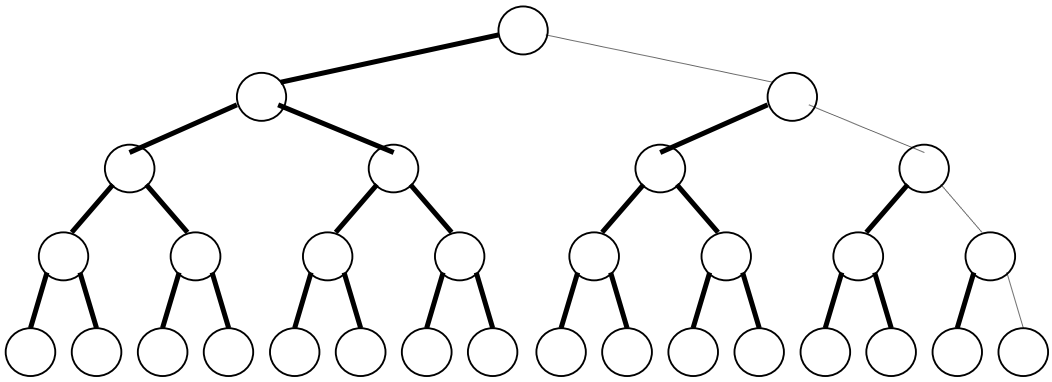


Figure 32.6: Marked and unmarked edges

The marked links are the ones through which the data can move up and down in the tree. We can move the data in any node at a height to a node at other height following the marked links. However, for the movement of the data in the root node to right side of it, we can have opposite of the above figure. The opposite of the above figure can be drawn by symmetry of the above steps. That means we first mark the right edge and then the subsequent left edges. This will give us the figure of marked links in which we can move to the right subtree of root.

Now let's sort out different aspects of the tree shown in above figure. We know that in case of tree of *N* nodes, there will be *N – 1*. Now in the tree above there are 31 nodes that means n = 31, so the number of edges is 31 - 1 = 30. The height of the tree is 4. Height is represented by the letter H. so H = 4. The number of dotted edges (that were not marked) in the tree is 4 that is the same as the height of the tree. Now we put these values in the formula for the sum of the heights of the nodes. We know the formula i.e.

$$S = N – H – 1$$

By putting values in this formula, we get

$$S = 31 – 4 – 1 = 26$$

If we count the darkened edges (marled links) in the above tree, it is also 26 that is equal to the sum of heights. Thus with the help of these figures, the theorem earlier proved mathematically, is established by a non-mathematical way.

Data Structures

Lecture No. 33

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 6, 8
6.3, 8.1

Summary

- Priority Queue Using Heap
- The Selection Problem
- Heap Sort
- Disjoint Set ADT
- Equivalence Relations

Priority Queue Using Heap

As discussed in the previous lecture, we generally prefer to employ the buildHeap to construct heap instead of using insert if we have all the required data. *buildHeap* is optimized as compared to *insert* method as it takes lesser time than $Nlog_2N$. In the previous discussion, we had proved a theorem that if the number of links in the tree are counted, the sum of all is *N-h-1*. We have been iterating a lot of times that the best use of heap is priority queue's implementation. Let's try to develop a class of priority queue with the help of a heap. You are very familiar with the concept of bank simulation. While explaining the bank simulation, we had used a priority queue that was implemented by using an array. Now, we will implement the same queue with the help of a heap. For this purpose, the code will be modified so that heap is used in place of array. The interface (.h file) will remain the same. However, the implementation (.cpp file) will be changed. Let's see the following cpp code, which also shows the change in the .cpp file:

```
1.    #include  "Event.cpp"
2.    #include  "Heap.cpp"
3.    #define  PQMAX  30
4.    class  PriorityQueue
5.    {
6.          public:
7.                PriorityQueue()
8.                {
9.                      heap  =  new  Heap <Event> ( PQMAX );
10.               };
11.
12.               ~PriorityQueue()
13.               {
14.                     delete  heap;
15.               };
16.
```

```
17.                     Event *   remove()
18.                     {
19.                             if( !heap->isEmpty() )
20.                             {
21.                                     Event *   e;
22.                                     heap->deleteMin( e );
23.                                     return   e;
24.                             }
25.                             cout  <<  "remove - queue is empty."  <<  endl;
26.                             return  (Event *)  NULL;
27.                     };
28.
29.                     int   insert(Event *   e)
30.                     {
31.                             if( !heap->isFull() )
32.                     {
33.                                     heap->insert( e );
34.                                     return   1;
35.                             }
36.                             cout  <<  "insert queue is full."  <<  endl;
37.                             return  0;
38.                     };
39.
40.                     int  full(void)
41.                     {
42.                             return  heap->isFull();
43.                     };
44.
45.                     int  length()
46.                     {
47.                             return  heap->getSize();
48.                     };
49.     };
```

The first line has a file *Event.cpp that contains* all the events used for simulation. We are including .cpp files here as done in case of templates of C++. In the second line, there is heap.cpp *while* a constant PQMAX has been defined in third line, declaring the maximum size of the priority queue to be 30. In line 4, class *PriorityQueue* is declared. *public* keyword is given at line 6 indicating that all class members below will be of public scope. Line 7 starts the class constructor's definition while in the line 9; a new heap object is being created. This object is a collection of *Event* type objects and the number of elements in the collection is *PQMAX*. The address (pointer) of the newly created heap is stored in the *heap* object pointer. The *heap* is a private pointer variable of *PriorityQueue* class. Now there is the destructor of the class, where we are deleting (deallocating the The first line is including a file *Event.cpp*, this is containing all the events used for simulation. At the second line, we have included *heap.cpp*. In the third line, we are defining a constant PQMAX, declaring the maximum size of the priority queue to be 30. In line 4, class *PriorityQueue* is

declared. *public* keyword is given at line 6 indicating that all class members below will be of public scope. Line 7 is starting the class constructor's definition. At line 9, a new heap object is being created, this object is a collection of *Event* type objects and the number of elements in the collection is *PQMAX*. The address (pointer) of the newly created heap is stored in the *heap* object pointer. The *heap* is a private pointer variable of *PriorityQueue* class. Next comes the destructor of the class, where we are deleting (deallocating the allocated resources) the pointer variable *heap*. Next is the *remove* method that was sometimes, named as *dequeue* in our previous lectures. *remove* method is returning an *Event* pointer. Inside *remove*, the first statement at line 19 is an if-condition; which is checking whether *heap* is not empty. If the condition is true (i.e. the *heap* is not empty), a local variable (local to the block) *Event** variable *e* is declared. In the next line at line 22, we are calling deleteMin method to delete (remove) an element from the *heap*, the removed element is assigned to the passed in parameter pointer variable *e*. You might have noticed already that in this version of *deleteMin*, the parameter is being passed by pointer, in the previous implementation we used reference. In the next line, the retrieved value is returned, the function returns by returning the pointer variable *e*. But if the heap was empty when checked at line 19, the control is transferred to the line 25. At line 25, a message is displayed to show that the heap is empty and the next line returns a NULL pointer.

It is important to understand one thing that previously when we wrote the array based *remove* method. We used to take an element out from the start of the array and shifted the remaining elements to left. As in this implementation, we are using heap, therefore, all the responsibilities of maintaining the heap elements lie with the heap. When the deleteMin() is called, it returns the minimum number.

We now observe the *insert* method line by line. It is accepting a pointer type parameter of type *Event*. As the heap may become full, if we keep on inserting elements into it, so the first line inside the insert function is checking whether the heap has gone full. If the heap is not full, the if-block is entered. At line 33 inside the if-block, an element that is passed as a parameter to *insert* method of heap is inserted into the queue by calling the *insert* method of heap. This insert call will internally perform percolate up and down operations to place the new element at its correct position. The returned value 1 indicates the successful insertion in the queue. If the heap has gone full, a message is displayed i.e. '*insert queue is full*'. Note that we have used the word queue, not heap in that message. It needs to be done this way. Otherwise, the users of the PriorityQueue class are unaware of the internal representation of the queue (whether it is implemented using a heap or an array). Next line is returning 0 while indicating that the *insert* operation was not successful. Below to *insert*, we have *full()* method. This method returns an *int* value. Internally, it is calling is*Full()* method of heap. The *full()* method returns whatever is returned by the *isFull()* method of heap. Next is *length()* method as the size of heap is also that of the queue. Therefore, *length()* is internally calling the *getSize()* method of heap.

In this new implementation, the code is better readable than the PriorityQueue's implementation with array. While implementing the PriorityQueue with an array, we had to sort the internal array every time at each insertion in the array. This new implementation is more efficient as the heap can readjust itself in $\log_2 N$ times. Gain in performance is the major benefit of implementing PriorityQueue with heap as compared to implementation with array.

There are other significant benefits of the heap that will be covered in this course time to time. At the moment, we will have some common example usages of heap to make you clear about other uses of heap. The heap data structure will be covered in the Algorithms course also.

The Selection Problem

– Given a list of *N* elements (numbers, names etc.) which can be totally ordered and an integer k, find the k^{th} smallest (or largest) element.

Suppose, we have list of N names (names of students or names of motor vehicles or a list of numbers of motor vehicles or list of roll numbers of students or id card numbers of the students, whatever). However, we are confronting the problem of finding out the k^{th} smallest element. Suppose we have a list of 1000 numbers and want to find the 10^{th} smallest number in it. The sorting is applied to make the elements ordered. After sorting out list of numbers, it will be very easy to find out any desired smallest number.

– One way is to put these *N* elements in an array and sort it. The k^{th} smallest of these is at the k^{th} position.

It will take $N\log_2 N$ time ,in case we use array data structure. Now, we want to see if it is possible to reduce the time from $N\log_2 N$ by using some other data structure or by improving the algorithm? Yes, we can apply heap data structure to make this operation more efficient.

– A faster way is to put the *N* elements into an array and apply the *buildHeap* algorithm on this array.
– Finally, we perform *k deleteMin* operations. The last element extracted from the heap is our answer.

The *buildHeap* algorithm is used to construct a heap of given N elements. If we construct ‘min-heap, the minimum of the N elements, will be positioned in the root node of the heap. If we take out (*deleteMin*) k elements from the heap, we can get the Kth smallest element. *BuildHeap* works in linear time to make a min or a max-heap.

– The interesting case is $k = \lceil N/2 \rceil$ as it is also is known as the *median*.

In Statistics, we take the average of numbers to find the minimum, maximum and median. Median is defined as a number in the sequence where the half of the numbers are greater than this number while the remaining half are smaller ones. Now, we can come up with the mechanism to find median from a given N numbers. Suppose, we want to compute the median of final marks of students of our class while the maximum aggregate marks for a student are 100. We use the *buildHeap* to construct a heap for N number of students. By calling *deleteMin* for N/2 times, the minimum marks of the half number students will be taken out. The $N/2^{th}$ marks would be the median of the marks of our class. The alternate methods are there to calculate median. However, we are discussing the possible uses of heap. Let’s see another use of heap.

Heap Sort

To take the 100^{th} minimum element out from the min-heap, we will call *deleteMin* to

take out 1st element, 2nd element, 3rd element. Eventually we will call *deleteMin* 100th time to take out our required 100th minimum element. Suppose, if the size of the heap is 100 elements, we have taken out all the elements out from the heap. Interestingly the elements are sorted in ascending order. If somehow, we can store these numbers, let's say in an array, all the elements sorted (in ascending order in this min-heap case) can be had.
Hence,

- If $k = N$, and we record the *deleteMin* elements as they come off the heap. We will have essentially sorted the N elements.

- Later in the course, we will fine-tune this idea to obtain a fast sorting algorithm called *heapsort*.

We conclude our discussion on the heap here. However, it will be discussed in the forthcoming courses. At the moment, let' see another Abstract Data Type.

Disjoint Set ADT

Before actually moving to an Abstract Data Type (ADT), we need to see what that ADT is, how it works and in which situations it can be helpful. We are going to cover Disjoint Set ADT. Firstly, we will have its introduction, examples and later the ways of its implementation.

- Suppose we have a database of people.
- We want to figure out who is related to whom.Initially, we only have a list of people, and information about relations is obtained by updating the form "Haaris is related to Saad".Let's say we have a list of names of all people in a locality but are not aware of their relationships to each other. After having the list of all people, we start getting some information about their relationships gradually. For example, "Ali Abbas is the son of Abbas".

The situation becomes interesting when we have relationships like "Ali Abbas is first cousin of Ayesha Ali (i.e. fathers of both are brothers) but Ayesha Ali has other immediate cousins also from her mother's side. Therefore, other immediate cousins of Ayesha Ali also get related to Ali Abbas despite the fact that they are not immediate to him".
So as we keep on getting relationship details of the people, the direct and indirect relationships can be established.

- Key property: If Haaris is related to Saad and Saad is related to Ahmad, then Haaris is related to Ahmad.

See the key property line's first part above "Harris is related to Saad and Saad is related to Ahmad". Suppose we have a program to handle this list of people and their relationships. After providing all the names "Harris, Saad and Ahmad" to that program and their relationships, the program might be able to determine the remaining part "Harris related to Ahmad".

The same problem (the intelligence required in the program) is described in the sentence below:

- Once we have relationships information, it will be easy for us to answer queries like "Is Haaris related to Ahmad?"

To answer this kind of queries and have that intelligence in our programs, *Disjoint Set ADT* is used. Before going for more information about *Disjoint Set ADT* , we see another application of this ADT in image analysis. This problem is known as *Blob Coloring*.

Blob Coloring

A well-known low-level computer vision problem for black and white images is the following:

Put together all the picture elements (pixels) that belong to the same "blobs", and give each pixel in each different blob an identical label.

You must have heard of robots that perform certain tasks automatically. How do they know from the images provided to them that in which direction should they move? They can catch things and carry them. They do different things the way human beings do. Obviously, there is a software working internally in robots' body, which is doing all this controlling part and vision to the robot. This is very complex problem and broad area of *Computer Science* and *Electrical Engineering* called *Robotics* (*Computer Vision* in particular).

Consider the image below:

``

Fig 33.1

This image is black and white, consisting of five non-overlapping black colored regions of different shapes, called blobs. These blobs are two elliptics- n and u shaped (two blobs) and one arc at the bottom. We can see these five blobs. How can robot identify them? So the problem is:

– We want to *partition* the pixels into *disjoint sets*, one set per blob.

If we make one set per blob, there will be five sets for the above image. To understand the concept of disjoint sets, we can take an analogy with two sets- A and B (as in Mathematics) where none of the elements inside set A is present in set B. The sets A and B are called disjoint sets.

Another problem related to the Computer Vision is the *image segmentation problem*. See the image below on the left of an old ship in gray scales.

Fig 33.2

We want to find regions of different colors in this picture e.g. all regions in the picture of color black and all regions in the image of color gray. The image on the right represents the resultant image.

Different scanning processes carried out in hospitals through MRI (Magnetic Resonance Imaging), CAT *Scan* or CT *Scan views* the inner parts of the whole body of human beings. These scanned images in gray scales represent organs of the human body. All these are applications of *Disjoint Set ADT*.

Equivalence Relations

Let's discuss some Mathematics about sets. You might have realized that Mathematics is handy whenever we perform analysis of some data structure.

– A binary relation R over a set S is called an *equivalence relation* if it has following propertiesReflexivity: for all element x ∈ S, x R x
  2. Symmetry: for all elements x and y, x R y if and only if y R x
  3. Transitivity: for all elements x, y and z, if x R y and y R z then x R z
– The relation "is related to" is an equivalence relation over the set of people.

You are advised to read about equivalence relations yourself from your text books or from the internet.

Data Structures

Lecture No. 34

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8
8.1, 8.2, 8.3

Summary

- Equivalence Relations
- Disjoint Sets
- Dynamic Equivalence Problem

Equivalence Relations

We will continue discussion on the abstract data structure, 'disjointSets' in this lecture with special emphasis on the mathematical concept of Equivalence Relations. You are aware of the rules and examples in this regard. Let's discuss it further and define the concept of Equivalence Relations.

'A binary relation R over a set S is called an *equivalence relation* if it has following properties':

1. Reflexivity: for all element x ξ S, x R x
2. Symmetry: for all elements x and y, x R y if and only if y R x
3. Transitivity: for all elements x, y and z, if x R y and y R z then x R z

The relation "is related to" is an equivalence relation over the set of people. This is an example of Equivalence Relations. Now let's see how the relations among people satisfy the conditions of Equivalence Relation. Consider the example of Haris, Saad and Ahmed. Haris and Saad are related to each other as brother. Saad and Ahmed are related to each other as cousin. Here Haris "is related to" Saad and Saad "is related to" Ahmed. Let's see whether this binary relation is Equivalence Relation or not. This can be ascertained by applying the above mentioned three rules.

First rule is reflexive i.e. for all element *x ξ S, x R x*. Suppose that *x* is Haris so *Haris R Haris*. This is true because everyone is related to each other. Second is Symmetry: for all elements *x* and *y, x R y* if and only if *y R x*. Suppose that *y* is Saad. According to the rule, *Haris R Saad* if and only if *Saad R Haris*. If two persons are related, the relationship is symmetric i.e. if I am cousin of someone so is he. Therefore if Haris is brother of Saad, then Saad is certainly the brother of Haris. The family relationship is symmetric. This is not the symmetric in terms of respect but in terms of relationship. The transitivity is: 'for all elements *x, y* and *z*. If *x R y* and *y R z*, then *x R z*'. Suppose *x* is Haris, *y* is Saad and *z* is Ahmed. If Haris "is related to" Saad, Saad "is related to" Ahmed. We can deduce that Haris "is related to" Ahmed. This is also true in relationships. If you are cousin of someone, the cousin of that person is also related to you. He may not be your first cousin but is related to you.

Now we will see an example of binary relationship that is not based on equivalence relationship. The ≤ relationship is *not* an equivalence relation. We will prove this by applying the three rules. The first rule is reflexive. It is reflexive, since $x \leq x$, as x is not less than x but surely is equal to x. Let's check the transitive condition. Since $x \leq y$ and $y \leq z$ implies $x \leq z$., it is also true. However it is *not symmetric* as $x \leq y$ does not imply $y \leq x$. Two rules are satisfied but symmetric rule does not. Therefore ≤ is not an equivalence relation.

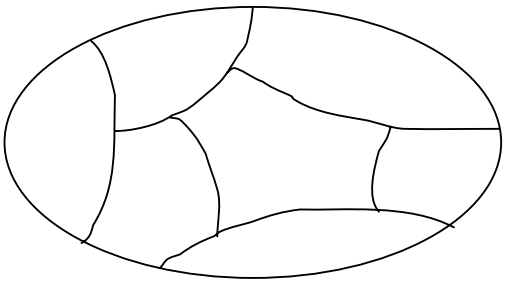
Let's see another example of binary relationship that is also equivalence relation. This is from the electric circuit domain. Electrical connectivity, where all connections are by metal wires, is an equivalence relation. You can make the circuit diagram on the paper including transistor, resistors and capacitors etc. These parts are connected to each other with the metal wire. Now let's apply the rules of equivalence relations on it. It is clearly reflexive, since the component is connected to itself. In a circuit, a transistor is connected to itself. It is symmetric due to the fact that if component *a* is connected to component *b,* then *b* must be electrically connected to *a*. Suppose we have two capacitors *a* and *b*. If capacitor *a* is connected to *b*, capacitor *b* is also connected to *a*. It is also transitive. If component a is connected to component *b* and *b* is connected to *c,* then *a* is connected to *c*. This is also true in electrical connectivity. All the three rules of equivalence relations satisfy in this case. Therefore this is equivalence relation.

Disjoint sets are those in which there is nothing common .

Disjoint Sets

In the beginning of the lecture, it was told that equivalence relationship partitioned the set. Suppose we have a set of elements and a relation which is also equivalence relation. Let's see what it does mathematically to the set. An equivalence relation R over a set S can be viewed as a partitioning of S into disjoint sets. Here we have an equivalence relationship which satisfies the three conditions. Keep in mind the examples of family relationships or electrical circuits. Here the second point is that each set of the partition is called an *equivalence class* of R (all elements that are related).

Consider the diagram below. We have a set in the shape of an ellipse.



This set has been partitioned into multiple sets. All these parts are disjoint sets and belong to an equivalence class.

Let's discuss an example to understand it. Suppose there are many people around you. How can we separate those who are related to each other in this gathering? We make

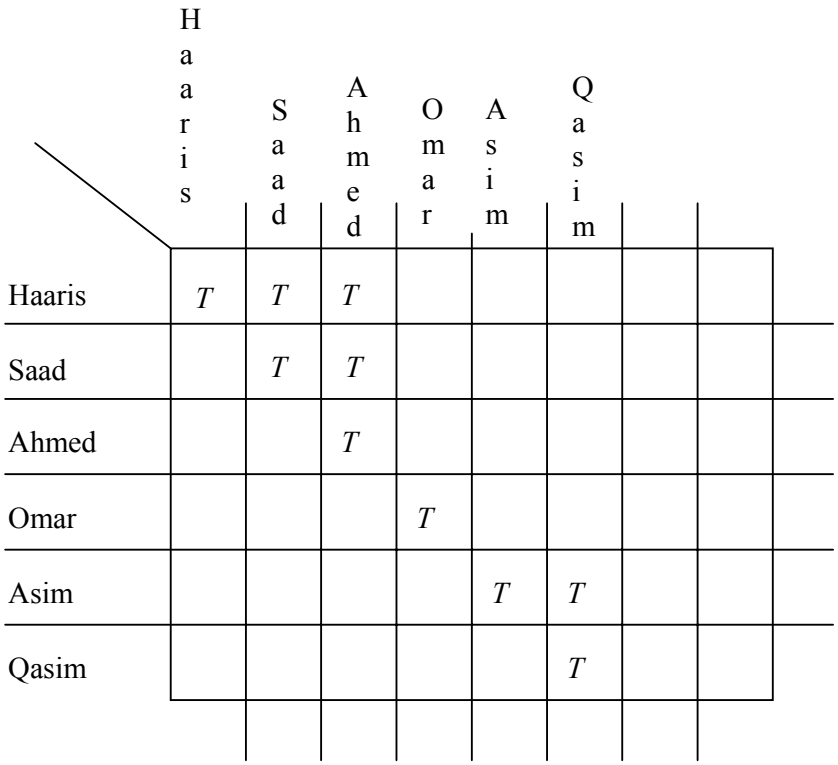
groups of people who are related to each other. The people in one group will say that they are related to each other. Similarly we can have many groups. So every group will say that they are related to each other with family relationship and there is no group that is related to any other group. The people in the group will say that there is not a single person in the other group who is related to them. There is a possibility that a boy from one group marries to a girl from the other group. Now a relation has been established between these two groups. The people in both groups are related to each other due to this marriage and become a bigger family. With the marriages people living in different families are grouped together. We will do operations like this in case of disjoint sets by combining two sets. You must be aware of the union and intersection operations in sets.

Every member of S appears in exactly one equivalence class. We have divided a set into many disjoint sets. One member of the set can appear in only one equivalence class. Keeping in mind the example of family relations, if a person belongs to a group he cannot go to some other group. The second point is to decide if *a R b.* For this purpose*,* we need only to check whether *a* and *b* are in the same equivalence class. Consider the example of pixels. If a pixel *p1* is in a region and want to know the relation of pixel *p2* with it, there is only the need to confirm that these two pixels are in the same region. Lastly, this provides a strategy to solve the equivalence problem. With the help of second point we can get help to solve the equivalence problem. So far, we have not seen the disjoint sets and its data structure.

Dynamic Equivalence Problem

Let's talk about the data structure. Suppose we have a set and converted into disjoint sets. Now we have to decide if these disjoint sets hold equivalence relation or not. Keep in mind the example of family relations in which we had divided the people into groups depending upon their relations. Suppose we have to decide that two persons belong to the same family or not. You have to make this decision at once. We have given a set of people and know the binary relation between them i.e. a person is a cousin of other, a person is brother of other etc. So the relation between two persons is known to us. How we can take that decision? What is the data available to us? We have people (with their names) and the binary relation among them. So we have names and the pair of relations. Think that Haris and Ahmed are related to each other or not i.e. they belong to the same family or not. How can we solve this? One way to find out the relationship is given below.

If the relation is stored as a two-dimensional array of booleans, this can be done in constant time. The problem is that the relation is usually not explicitly defined, but shown in implicit terms. Suppose we are given 1000 names of persons and relations among them. Here we are not talking about the friendship as it is not a family relation. We are only talking about the family relations like son-father, brother-brother, cousin-cousin, brother-sister, etc. Can we deduce some more relations from these given relations? Make a two dimensional array, write the names of persons as the columns and rows headings. Suppose Haris is the name of first column and first row. Saad is the name of 2^{nd} col and 2^{nd} row. Similarly Ahmed, Omar, Asim and Qasim are in the 3^{rd}, 4^{th}, 5^{th}, and 6^{th} columns and rows respectively. The names are arranged in the same way in columns and rows.



Now visit each cell of matrix. If two persons are related, store T(true) in the corresponding column and row. As Haris and Saad are related to each other, so we take the row of Haris and column of Saad and store T in that cell. Also take the row of Saad and column of Ahmed and mark it with T. By now, we do not know that Haris and Ahmed are related so the entry in this regard will be false. We cannot write T(True) here as this relationship has not be stated explicitly. We can deduce that but cannot write it in the matrix. Take all the pairs of relations and fill the matrix. As Haris is related to Haris so we will write T in the Haris row and Haris column. Similarly, the same can be done in the Saad row and Saad column and so on. This will be square matrix as its rows and columns are equal. All of the diagonal entries are T because a person is related to himself. This is a self relation. Can we find out that Haris and Ahmed are related? We can find the answer of this question with the help of two relations between Haris & Saad and Saad & Ahmed. Now we can write T in the Haris row and Ahmed column. The problem is that this information was not given initially but we deduced this from the given information. We have 1000 people so the size of the matrix will be 1000 rows * 1000 columns. Suppose if we have 100,000 people, then the size of the array will be 100,000*100,000. If each entry requires one byte then how much storage will be required to store the whole array? Suppose if we have one million people, the size will be 10 to the power 14. Do we have as much memory available in the computer? We have one solution but it is not efficient in terms of memory. There is always need of efficient solution in terms of space and time. For having a fast algorithm, we can compromise on the space. If we want to conserve memory, a slow algorithm can be used. The algorithm we discussed is fast but utilized too much space.

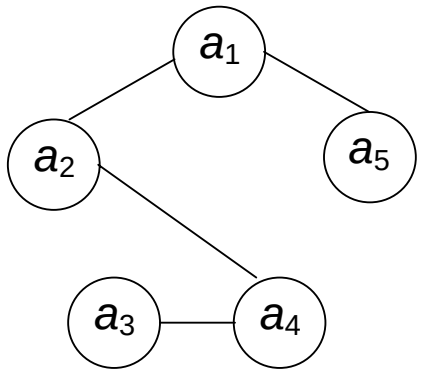
Let's try to find out some more efficient solution of this problem. Consider another example. Suppose the equivalence relation is defined over the five-element set {$a1$,

*a*2, *a*3, *a*4, *a*5}. These elements can be the name of people, pixels, electrical components etc. How many pairs we can have in this set? We have a pair *a1-a1*, *a1-a2*, *a1-a3, a1-a4, a1-a5, a2-a2, a2-a3* etc. The pair *a1-a2* and *a2-a1* are equal due to symmetric. We also find some self-pairs (i.e. *a1-a1, a2-a2* etc). There are 25 pairs of elements, each of which is related or not (30 pairs – 5 self-pairs = 25). These are the total pairs and we may not have as much relations. What will be size of array with five elements? The size will be 5*5 = 25. We are also given relations i.e.

- *a*1 R *a*2,
- *a*3 R *a*4,
- *a*5 R *a*1,
- *a*4 R *a*2,

We have five people in the example and their relation is given in four pairs. If two persons are brothers, then cousin of one brother is also the cousin of other. We can find out this information with the help of a matrix. We would like to be able to infer this information quickly.

We made nodes of each element. These five nodes are as *a1, a2, a3, a4, a5.*



As *a1 R a2*, so we link the nodes *a1* and *a2*. Similarly *a3 R a4*, these two nodes are also linked. Following this pattern, it is established that *a5 R a1* and *a4 R a2*. So we connect these nodes too. It is clear from the figure that all of these nodes are related to each other. If they are related to each other as cousin, then all of these five persons belong to the same family. They need to be in the same set. They are not in disjoint sets. Is *a3 R a5*? You can easily say yes. How you get this information? This relation is not provided in the given four relations. With the above tree or graph we can tell that *a3* is related to *a5*. We can get the information of relationship between different persons using these nodes and may not need the matrix.

We want to get the information soon after the availability of the input data. So the data is processed immediately. The input is initially a collection of *n* sets, each with one element. Suppose if we have 1000 people, then there will be need of 1000 sets having only one person. Are these people related to each other? No, because every person is in different set. This initial representation is that all relations (except reflexive relations) are false. We have made 1000 sets for 1000 people, so only the reflexive relation (every person is related to himself) is true. Now mathematically speaking, each set has a different element so that *S*i ∩ *S*j = Ø which makes the sets

disjoint. A person in one set has no relation with a person in another set, therefore there intersection is null. Now here we have 1000 sets each containing only one person. Only the reflexive relation is true and all the 1000 sets are disjoint. If we take intersection of any two sets that will be null set i.e. there is no common member in them.

Sometimes, we are given some binary relations and are asked to find out the relationship between two persons. In other words, we are not given the details of every pair of 1000 persons. The names of 1000 persons and around 50 relations are provided. With the help of these 50 relations, we can find out more relations between other persons. Such examples were seen above while find out the relationship with the help of graph.

There are two permissible operations in these sets i.e. *find* and *union*. In the *find* method, we are given one element (name of the person) and asked to find which set it belongs to. Initially, we have 1000 sets and asked in which set person 99 is? We can say that every person is in a separate set and person 99 is in set 99. When we get the information of relationships between different persons, the process of joining the sets together can be started. This is the union operation. When we apply union operation on two sets, the members of both sets combined together and form a new set. In this case, there will be no duplicate entry in the new sets as these were disjoint. The definitions of *find* and *union* are:

- *Find* returns the name of the set (equivalence class) that contains a given element, i.e., *S*i = find(*a*)
- *Union* merges two sets to create a new set *S*k = *S*i U *S*j.

We give an element to the *find* method and it returns the name of the set. The method *union* groups the member of two sets into a new set. We will have these two operations in the disjoint abstract data type. If we want to add the relation *a R b*, there is need to see whether *a* and *b* are already related. Here *a* and *b* may be two persons and a relation is given between them. First of all we will see that they are already related or not. This is done by performing *find* on both *a* and *b* to check whether they are in the same set or not. At first, we will send *a* to the *find* method and get the name of its set before sending b to the *find* method. If the name of both sets is same, it means that these two belong to the same set. If they are in the same set, there is a relation between them. We did not get any useful information with this relation. Let's again take the example of the Haris and Saad. We know that Haris is the brother of Saad, so they can be placed into a new set. Afterwards, we get the information that Saad is the brother of Haris.

Now the question arises, if they are not in the same set, what should we do? We will not waste this information and apply *union* which merges the two sets *a* and *b* into a new set. This information will be helpful later on. We keep on building the database with this information. Let's take the example of pixels. Suppose we have two pixels and are told that these two belong to the same region, have same color or these two pixels belong to the liver in CT scan. We will keep these two pixels and put them in one set. We can name that set as *liver*. We will use union operation to merge two sets. The algorithm to do this is frequently known as *Union*/*Find* for this reason.

There are certain points to note here. We do not perform any operations comparing the relative values of set elements. That means that we are not considering that one person is older than the other, younger or one electrical component is bigger than other etc. We merely require knowledge of their location, i.e., which set an element, belongs to.

Let's talk about the algorithm building process. We can assume that all elements are numbered sequentially from 1 to *n*. Initially we have $Si = \{i\}$ for $i = 1$ through *n*. We can give numbers to persons like person1, person2 etc. Consider the example of jail. In a jail, every prisoner is given a number. He is identified by a number, not by the name. So you can say that persons in a cell of jail are related. We will use these numbers to make our algorithm run faster. We can give numbers to every person, every pixel, each electrical component etc. Under this scheme, the member of *S10* is number 10. That number 10 can be of a person, a pixel or an electrical component. We are concerned only with number.

Secondly, the name of the set returned by *find* is fairly arbitrary. All that really matters is that *find*(*x*) = *find*(*y*) if and only if *x* and *y* are in the same set. We will now discuss a solution to the *union*/*find* problem that for any sequence of at most *m* finds and up to *n*-1 unions will require time proportional to (*m* + *n*). We are only storing numbers in the sets and not keeping any other relevant information about the elements. Similarly we are using numbers for the names of the sets. Therefore we may want that the method *find* just returns the number of the set i.e. the person 10 belongs to which set number. The answer may be that the person 10 belongs to set number 10. Similarly the set number of person 99 may be set 99. So if the set numbers of two persons is not equal, we can decide that these two persons do not belong to the same set. With the help of this scheme, we will develop our algorithm that will run very fast. In the next lecture, we will discuss this algorithm and the data structure in detail.

Data Structures

Lecture No. 35

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8

8.2, 8.3

Summary

- Dynamic Equivalence Problem
- Example 1
- Parent Array
  - Initialization
  - Find (i)
  - Union (i, j)
- Example 2
  - Initialization
  - union operation
  - find Operation
- Running Time Analysis

Before talking about the data structure implementation in detail, we will recap the things discussed in the previous lecture. The concepts that came under discussion in the last lecture included the implementation of the disjoint set. It was observed that in case of data elements, we assign a unique number to each element to make sets from these numbers. Moreover, techniques of merger and searching in these sets by using the operations union and find were, talked about. It was seen that if we send a set item or a number to *find* method, it tells us the set of which this item is a member. It will return the name of the set. This name of set may also be a number. We talked about the *union* method in which members of two sets join together to form a new set. The *union* method returns the name or number of this new set.
Now we will discuss the data structure used to implement the disjoint sets, besides ascertaining whether this data structure implements the find and union operations efficiently.

Dynamic Equivalence Problem

We are using sets to store the elements. For this purpose, it is essential to remember which element belongs to which set. We know that the elements in a set are unique and a tree is used to represent a set. Each element in a tree has the same root, so the root can be used to name the set. The item (number) in the root will be unique as the set has unique values of items. We use this root as the name of set for our convenience. Otherwise, we can use any name of our choice. So the element in the root (the number) is used as the name of the set. The *find* operation will return this name. Due to the presence of many sets, there will be a collection of trees. In this collection, each tree will be representing one set. If there are ten elements, we will

have ten sets initially. Thus there will be ten trees at the beginning. In general, we have N elements and initially there will be N trees. Each tree will have one element. Thus there will be N trees of one node. Now here comes a definition for this collection of trees, which states that a collection of trees is called a *forest*.

The trees used for the sets are not necessarily binary trees. That means it is not necessary that each node should have a maximum of two children nodes. Here a node may have more than two children nodes.

To execute the union operation in two sets, we merge the two trees of these sets in such a manner that the root of one tree points to the root of other. So there will be one root, resulting in the merger of the trees. We will consider an example to explain the union operation later.

In the find operation, when we call *find (x)*, it helps us to know which set this *x* belongs to. Internally, we find this *x* in a tree in the *forest*. When this *x* is found in a tree the *find* returns the number at root node (the name of the set) of that tree.

Example 1

Now let's consider an example to understand this phenomenon. Suppose, we have developed a data structure and apply the union and find operations on it. There are eight elements in this data structure i.e.1 to 8. These may be the names of people to which we have assigned these numbers. It may be other distinct eight elements. We will proceed with these numbers and make a set of each number. The following figure shows these numbers in different sets.

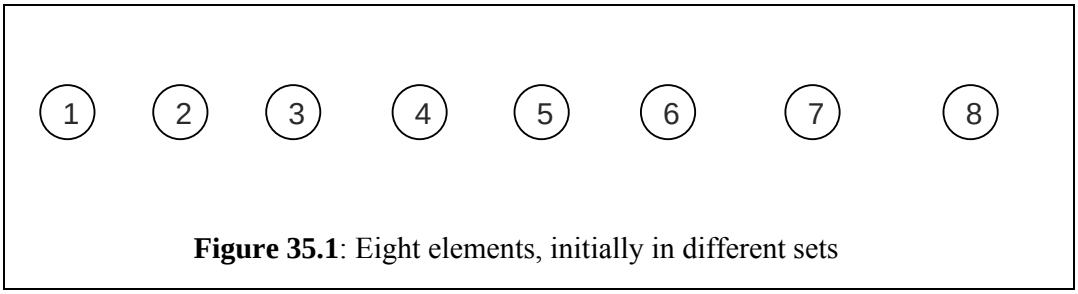


Figure 35.1: Eight elements, initially in different sets

Now we carry out union operation on the sets 5 and 6. The call *union(5,6)* means, merge the sets 5 and 6 and return the new set developed due to the merger of the sets-5 and 6. In the following figure, we see this union. We put the set 6 below set 5, which join together.

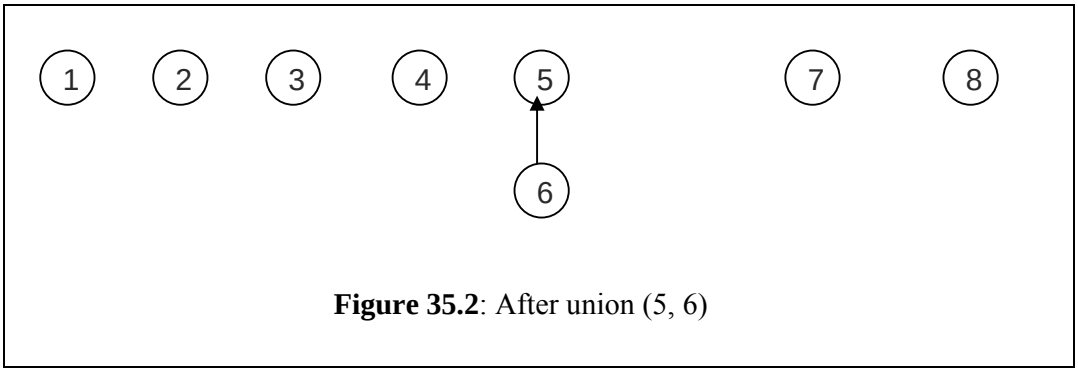


Figure 35.2: After union (5, 6)

After the merger, the new set contains two members 5 and 6. Now the question arises what will be the name of this new set. In the above union operation, the set 6 becomes

the node of 5. It may be in reverse i.e. 5 could be a node of 6. In the above figure, we put 6 as node of 5. Moreover the arrow joining these is from 6 to 5. In the new set formed due to the merger of 5 and 6, it seems that 5 has some superiority. So the name of the new set is 5. We passed two arguments 5 and 6 to the *union* function. And the union made 6 a member of 5. Thus, if we pass S1 and S2 to the *union* function, the *union* will make S2 a member of S1. And the name of the new set will be S1. That means the name of first argument will be the name of the new set.
Now we call *union (7,8)*. After this call, 7 and 8 form a new set in which 8 is a member of 7 and the name of this new set is 7 (that is the first argument in the call). In other words, 7 is root and 8 its child. This is shown in the following figure.

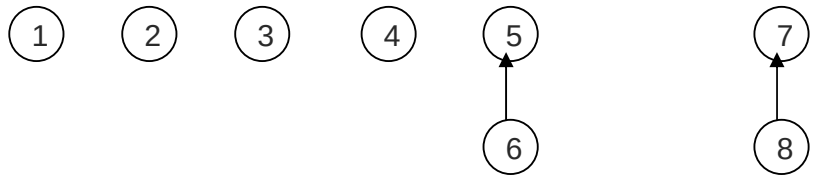


Figure 35.3: After union (7, 8)

Now we call the union function by passing it set 5 and set 7 as arguments i.e. we call *union (5,7)*. Here the sets 5 and 7 have two members each. 5 and 6 are the members of 5 and similarly the two members of 7 are 7 and 8. After merging these sets, the name of the new set will be 5 as stated earlier that the new set will be named after the first argument. The following figure (figure 35.4) shows that now in the set 5, there are four members i.e. 5, 6, 7 and 8.

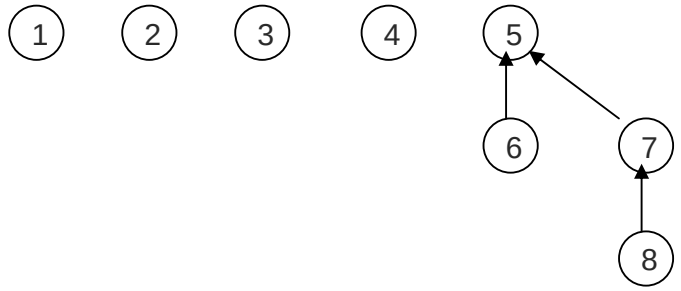


Figure 35.4: After union (5, 7)

We can see that there are four unique set members in this set (i.e. 5).
We will demonstrate the union operation with the help of another example. Suppose, we have made another union that is *union (3,4)*. This call merges the set 4 with the set 3. The following figure shows this.

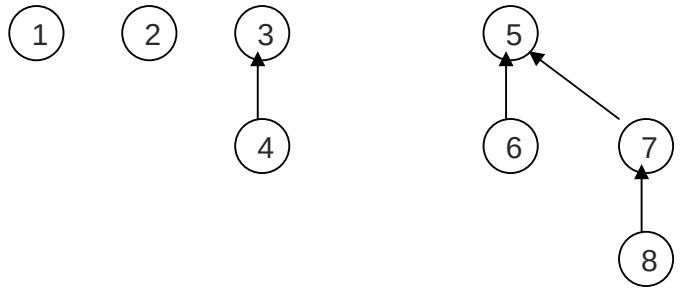


Figure 35.5: After union (3, 4)

In this figure, we see that there are four sets that are 1, 2, 3 and 5. Now we unite the sets 4 and 5 and make a call *union (4,5)*. The following figure shows the outcome of this call. In this figure, the set 5 points to 3 whereas we made a call *union (4,5)*.

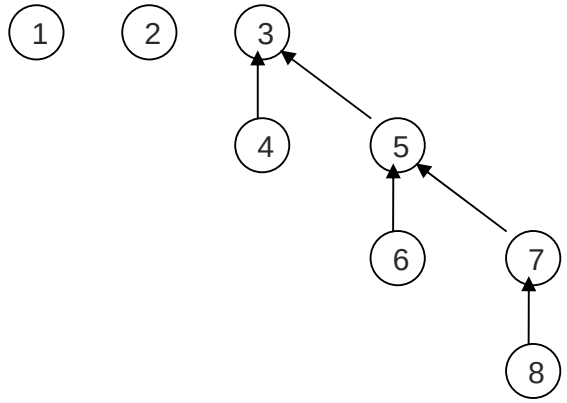


Figure 35.6: After union (4, 5)

So we conclude that it is not necessary that the caller should send the roots to union. It is necessary that the union function should be such that if a caller sends elements of two sets, it should find the sets of those elements, merge them and return the name of new set. Thus our previous call i.e. *union (4,5)* was actually carried out in the way that first the *union* finds the root of 4 that is 3. Then it looks for 5 that itself is a root of its set. After this, it merges both the trees (sets). This merger is shown in the above figure i.e. Figure 35.6.

Up to now, we have come to know that the formation of this tree is not like a binary tree in which we go down ward. Moreover, a binary tree has left and right children that are not seen yet in the tree we developed. This is a tree like structure with some properties.

Let's talk about these properties.

Here we see that typical tree traversal (like inorder, preorder or postorder) is not required. So there is no need for pointers to point to the children. Instead, we need a pointer to parent, as it's an up-tree. We call it up-tree due to the fact that it is such a tree structure in which the pointers are upward. These parent pointers can be stored in an array. Here we have to keep (and find) pointer to the parent of a node unlike a binary tree in which we keep pointer to child nodes. In the array, we will set the parent of root to –1. We can write it as

Parent[i] = -1 // if i is the root

Now we will keep these tree structures (forest) in an array in the same manner. With the merger of the trees, the parents of the nodes will be changed. There may be nodes that have no parent (we have seen this in the previous example). For such nodes, we will keep –1 in the array. This shows that this node has no parent. Moreover, this node will be a root that may be a parent of some other node.

Now we will develop the algorithm for *find* and *union*. Let's consider an example to see the implementation of this disjoint set data structure with an array.

Parent Array

Initialization

We know that at the start we have *n* elements. These *n* elements may be the original names of the elements or unique numbers assigned to them. Now we take an array and with the help of a *for* loop, keep these *n* elements as root of each set in the array. These numbers are used as the index of the array before storing –1 at each location from index zero to *n*. We keep –1 to indicate a number as root. In code, this *for* loop is written as under.

```
for ( i = 0; i < n ; i ++)
        Parent [i] = -1 ;
```

Find (i)

Now look at the following code of a loop. This loop is used to find the parent of an element or the name of the set that contains that element.

```
// traverse to the root (-1)
for(j=i; parent[j] >= 0; j=parent[j])
    ;
return j;
```

In this loop, *i* is an argument passed to the find function. The execution of the loop starts from the value, passed to the find function. We assign this value to a variable *j* and check whether its parent is greater than zero. It means that it is not –1. If it is greater than zero, its parent exists, necessitating the re-initialization of the *j* with this parent of *j* for the next iteration. Thus this loop continues till we find parent of *j* (parent [*j*]) less than zero (i.e. -1). This means that we come to the root before

returning this number *j*.

Union (i, j)

Now let's see the code for the function of union. Here we pass two elements to the function union. The union finds the roots of i and j. If *i* and *j* are disjoint sets, it will merge them. Following is the code of this function.

```
root_i = find(i);
root_j = find(j);
if (root_i != root_j)
      parent[root_j] = root_i;
```

In the code, we see that at first it finds the root of tree in which *i* exists by the find(*i*) method and similarly finds the root of the set containing j by find(*j*). Then there is a check in if statement to see whether these sets (roots) are same or not. If these are not the same, it merges them in such a fashion that the root *i* is set as the parent of root *j*. Thus, in the array where the value of root *j* exists, the value of root *i* becomes there.

Example 2

To understand these concepts, let's consider an example. We re-consider the same previous example of eight numbers and see how the initialization, union and find work.

Initialization

In the following figure (figure 35.7), we have shown the initialization step. Here we make an array of eight locations and have initialized them with –1 as these are the roots of the eight sets. This –1 indicates that there is no parent of this number. We start the index of array from 1 for our convenience. Otherwise, we know that the index of an array starts from zero.

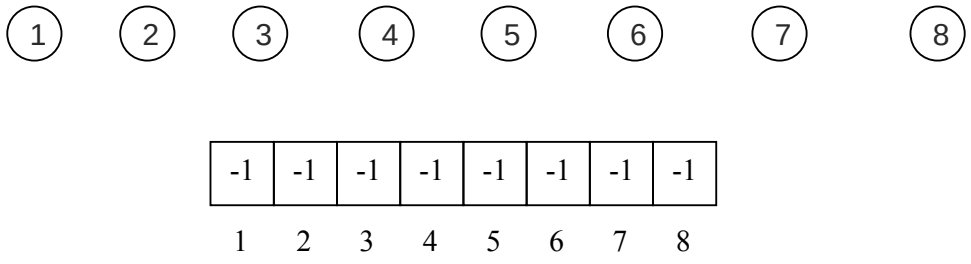


Figure 35.7: Eight elements, initially in different sets

Union Operation

Now we come to the union operation. First of all, we call union (5,6). This union operation forms an up-tree in which 5 is up and 6 down to it. Moreover 6 is pointing

to 5. In the array, we put 5 instead of –1 at the position 6. This shows that now the parent of 6 is 5. The other positions have –1 that indicates that these numbers are the roots of some tree. The only number, not a root now is 6. It is now the child of 5 or in other words, its parent is 5 as shown in the array in the following figure.

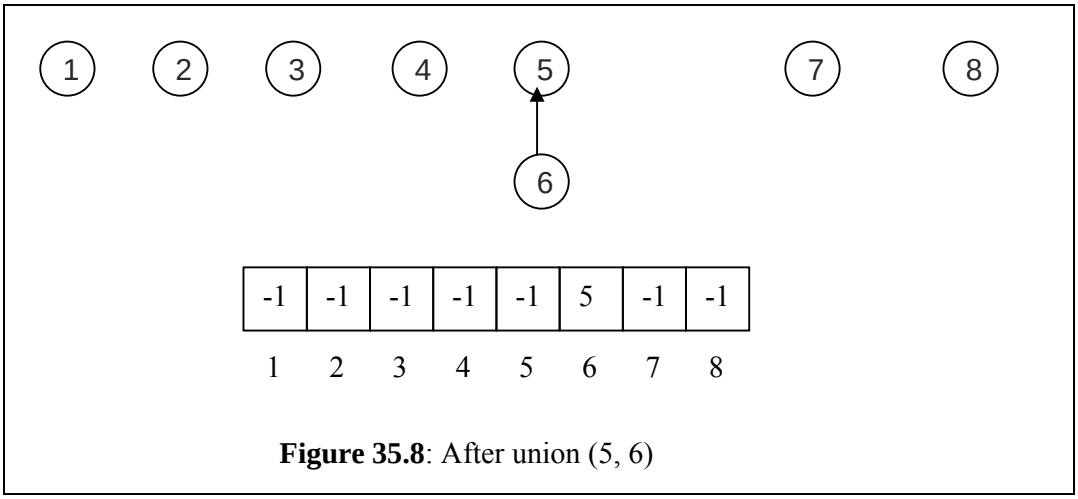


Figure 35.8: After union (5, 6)

Now we carry out the same process (i.e. union) with the numbers 7 and 8 by calling union (7,8). The following figure represents this operation. Here we can see that the parent of 8 is 7 and 7 itself is still a root.

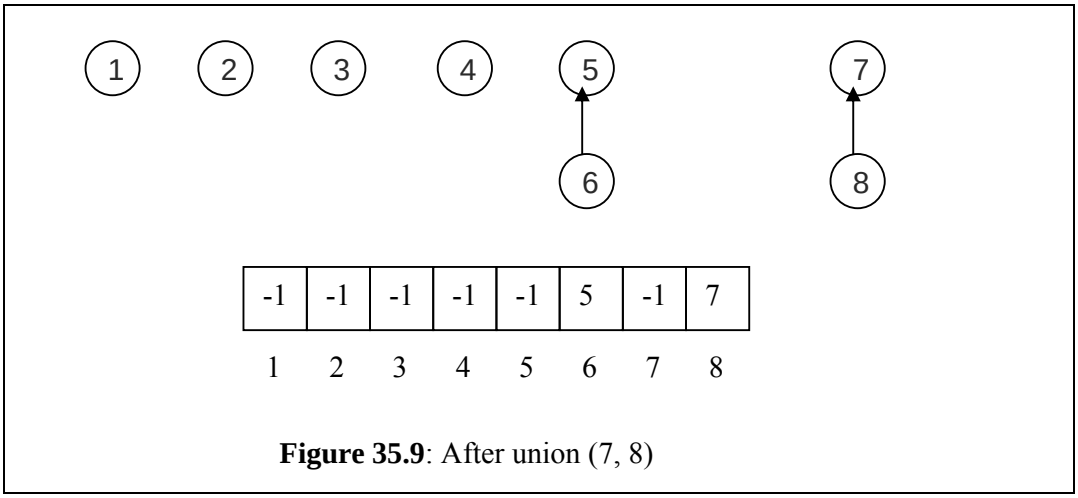


Figure 35.9: After union (7, 8)

Now we execute the *union (5,7)*. The following figure represents the tree and array status after the performance of this union operation.

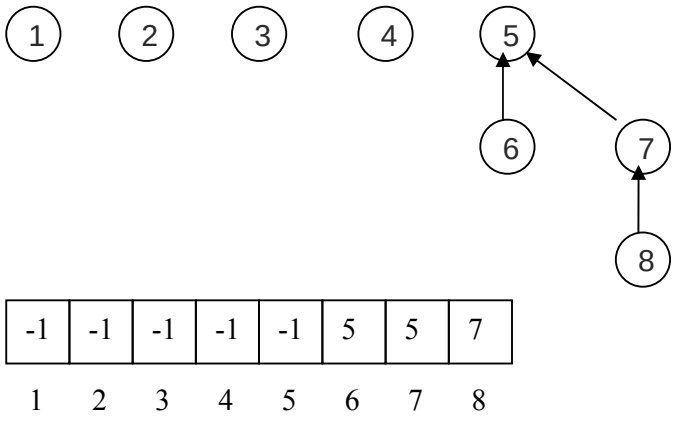


Figure 35.10: After union (5, 7)

The tree in the figure shows that 7 points to 5 now. In the array, we see that the value at position 7 is 5. This means that the parent of 7 is 5 now. Whereas the parent of 5 is still –1 that means it is still a root. Moreover, we can see that the parent of 8 is 7 as before.

Afterwards, we call *union (3,4)*. The effect of this call is shown in the following figure. Here in the array at position 4, there is 3 instead of –1. This shows that the parent of 4 is 3.

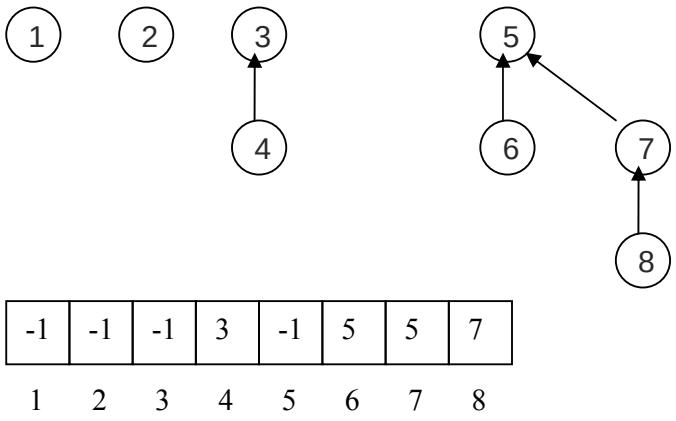


Figure 35.11: After union (3, 4)

By looking at the array only, we can know that how many trees are there in the collection (forest). The number of trees will be equal to the number of –1 in the array. In the above figure, we see that there are four –1 in the array so the number of trees is

four and the trees in the figure confirms this. These four trees are with the roots 1, 2 3 and 5 respectively.

Now we carry out another union operation. In this operation, we do the union of 4 and 5 i.e. union (4, 5). Here the number 4 is not a root. Rather, it is an element of a set. We have already discussed that the union operation finds the set (root) of the element passed to it before putting together those sets. So here the union finds the set containing 4 that is 3. The element 5 itself is a name (root) of a set. Now the union operation merges the two sets by making the second set a member of the first set. In this call (union (4, 5)), the first set is 3 (that is found for the element 4) and second is 5. So the union joins 5 with 3 while 5 is pointing to 3. Due to this union operation, now, in the array, there is 3 at position 5 instead of –1. The 3 at this position 5 indicates that the parent of 5 is 3. The following figure shows the tree and array after the operation union (4,5).

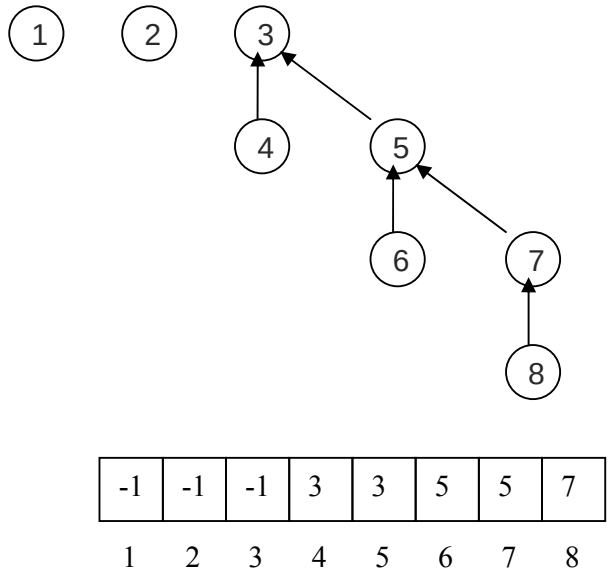


Figure 35.12: After union (4, 5)

These were the union operations while using parent array. Now let's look at the find operation.

Find Operation

To understand the find operation, we make a call *find (8)* in the above forest. This call means that the caller wants to know the set in which number 8 is lying. We know that the root of the tree is also the name of the set. By looking at the trees in the figure below, we come to know that 8 is in the tree with 3 as root. This means 8 is in set 3.

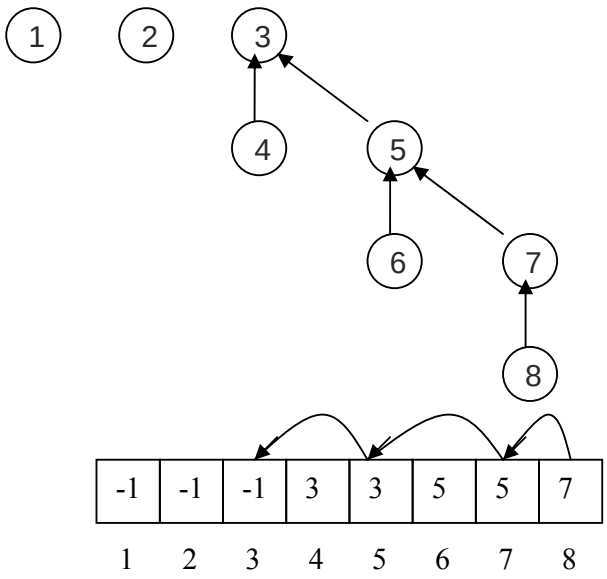


Figure 35.13: find (8)

Thus from the figure, we find the set containing 8. This is being implemented with the parent array to ultimately find it in the array. For this find operation in the array, we have discussed the algorithm in which we used a '*for* loop. This 'for loop' starts from the position that given to the *find* function. Here it is 8. The condition in the for loop was that as long as parent [*j*] is greater than zero, set this parent [*j*] to *j*. Now we execute the loop with the value of *j* equal to 8. First of all, the loop goes to position 8 and looks for the value of parent of 8. This value is 7, which sets the value of *j* to 7 and goes to position 7. At that position, the value of parent of 7 is 5. It goes to position 5. At position 5, the value of parent is 3. So the loop sets the value of *j* equal to 3 and goes to the position 3. Here it finds that the parent of 3 is –1 i.e. less than zero so the loop ends. This position 3 is the root and name of the set as parent of it is –1. Thus the name of the set that contains 8 is 3. The find will return 3 which mean that 8 is a member of set 3.

Similarly, in the above array, we can execute the operation *find (6)*. This will also return 3 as the loop execution will be at the positions 6 before going to 5 and finally to 3. It will end here, as the parent of 3 is –1. Thus *find (6)* returns 3 as the set that contains 6.

Running Time analysis

Now we will discuss how the implementation of disjoint set is better. We must remember that while discussing the implementation of disjoint set, we talked about Boolean matrix. This is a two dimensional structure in which the equivalence relations is kept as a set of Boolean values. Here in the parent array, we also are keeping the same information. The union will be used when the two items are related. In the two-dimensional matrix, we can easily find an item by its index. Now we are

using an array that is a single dimensional structure and seems better with respect to space. We keep all the information in this array which is at first kept in a matrix. Now think about a single dimension array versus a two dimensional array. Suppose we have 1000 set members i.e. names of people. If we make a Boolean matrix for 1000 items, its size will be 1000 x 1000. Thus we need 1000000 locations for Boolean values. In case of an array, the number of locations will be 1000. Thus this use of array i.e.tree like structure is better than two-dimensional array in terms of space to keep disjoint sets and doing union and find operations. Moreover, we do not use pointers (addresses) that are used in C++. We use array indices as pointers. In case of find, we follow some indices (pointers) to find the set containing a particular member. From these points of discussion, we conclude that

- *union* is clearly a constant time operation.
- Running time of *find*(*i*) is proportional to the height of the tree containing node *i*.
- This can be proportional to n in the worst case (but not always)
- Goal: Modify *union* to ensure that heights stay small

We will discuss these points in detail in the next lecture.

Data Structures

Lecture No. 36

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8
8.4, 8.5, 8.6

Summary

- Running Time Analysis
- Union by Size
- Analysis of Union by Size
- Union by Height
- Sprucing up find
- Timing with Optimization

Running Time Analysis

In the previous lecture, we constructed up trees through arrays to implement disjoint sets. u*nion* and *find* methods along with their codes were discussed with the help of figures. We also talked about the ways to improve these methods. As far as the *union* operation is concerned, considering the parent array, it was related to changing the index of the array. The root value of the merging tree was put into that. In case of find, we have to go to the root node following the up pointers.

While studying the balanced trees, we had constructed a binary search tree from the already sorted data. The resultant tree becomes a linked list with height N. Similar kind of risks involved here. Suppose, we have eight members i.e.1, 2, 3, 4, 5, 6, 7 and 8 as discussed in an example in the previous lecture. Suppose their unions are performed as union(1,2), union(2,3), union(3,4) and so on up to union(7,8). When the trees of these unions are merged together, there will be a longer tree of height 8. If we perform a find operation for an element that is present in the lowest node in the tree then N-1 links are traversed. In this particular case, 8-1=7 links.
We know that with reduced tree (with lesser height), the time required for *find* operation will also be reduced. We have to see whether it is possible to reduce the size (in terms of height) of the resultant tree (formed after unions of trees). Therefore, our goal is:

– ***Goal***: Modify *union* to ensure that heights stay small

We had already seen this goal in the last slide of our previous lecture, which is given below in full:

– *union* is clearly a constant-time operation.
– Running time of *find*(*i*) is proportional to the height of the tree containing node *i*.
– This can be proportional to n in the worst case (but not always)

– Goal: Modify *union* to ensure that heights stay small

You might be thinking of employing the balancing technique here. But it will not be sagacious to apply it here. Due to the use of array, the 'balancing' may prove an error-prone operation and is likely to decrease the performance too. We have an easier and appropriate method, called *Union by Size* or *Union by Weight*.

Union by Size

Following are the salient characteristics of this method:

– Maintain sizes (number of nodes) of all trees, and during *union.*
– Make smaller tree, the subtree of the larger one.
– Implementation: for each root node `i`, instead of setting `parent[i]` to `-1`, set it to `-k` if tree rooted at `i` has `k` nodes.
– This is also called *union-by-weight*.

We want to maintain the sizes of the trees as given in the first point. Until now, we are not maintaining the size but only the *parent* node. If the value for *parent* in a node is –1, then this indicates that the node is the *root* node. Consider the code of the *find* operation again:

```
// find(i):
// traverse to the root (-1)
for(j=i; parent[j] >= 0; j=parent[j])
        ;
return j;
```

The terminating condition for loop is checking for non-negative values. At any point when the value of *parent[j]* gets negative, the loop terminates. Note that the condition does not specify exactly the negative number. It may be the number –1, –2 , –3 or some other negative number that can cause the loop to be terminated. That means we can also put the number of nodes (size) in the tree in this place. We can put the number of nodes in a tree in the *root* node in the negative form. It means that if a tree consists of six nodes, its *root* node will be containing –6 in its *parent*. Therefore, the node can be identified as the *root* being the negative number and the magnitude (the absolute value) of the number will be the size (the number of nodes) of the tree.

When the two trees would be combined for union operation, the tree with smaller size will become part of the larger one. This will cause the reduction in tree size. That is why it is called union-by-size or union-by-weight.

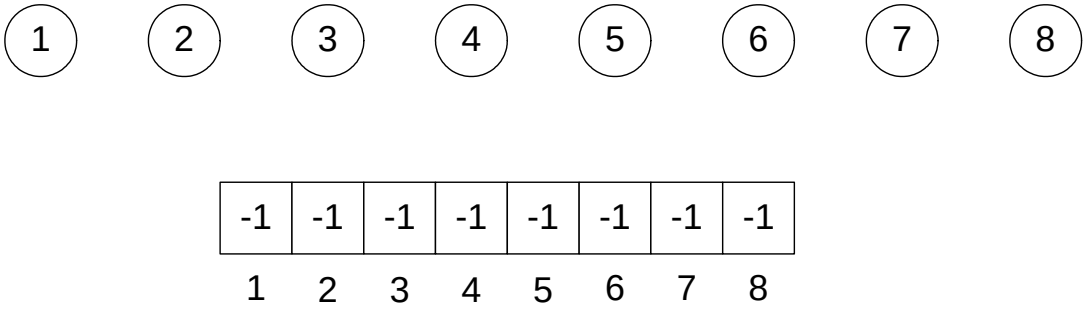
Let's see the pseudo-code of this union operation (quite close to C++ language). This contains the logic for reducing tree size as discussed above:

```
    //union(i,j):
1.  root1 = find(i);
2.  root2 = find(j);
```

```
3.      if (root1 != root2)
4.       if (parent[root1] <= parent[root2])
5.              {
6.                      // first tree has more nodes
7.                      parent[root1] += parent[root2];
8.                      parent[root2] = root1;
9.              }
10.             else
11.             {
12.                     // second tree has more nodes
13.                     parent[root2] += parent[root1];
14.                     parent[root1] = root2;
15.             }
```

This operation performs the *union* of two sets; element i's set and element j's set. The first two lines are finding the *root*s of the sets of both the elements i.e.i and j. Line 3 is performing a check that the *root1* is not equal to *root2*. In case of being unequal, these are merged together. In next statement (line 4), the numbers in *parent* variables of *root* nodes are compared. Note that, the comparison is not on absolute values. So the tree with greater number of nodes would contain the lesser number mathematically. The condition at line 4 will be true, if the tree of *root1* is greater in size than that of *root2*. In line 7, the size of the smaller tree is being added to the size of the greater one. Line 8 is containing the statement that causes the trees to merge. The *parent* of *root2* is being pointed to *root1*. If the *root2* tree is greater in size than the *root1* tree, the if–condition at line 4 returns false and the control is transferred to *else* part of the if-condition. The *parent* of *root1* (the size of the *root1* tree) is added in the *parent* of *root2* in line 13 and the *root1* tree is merged into the *root2* tree using the statement at line 14.

Let's practice this approach using our previous example of array of eight elements.



Eight elements, initially in different sets.

Fig 36.1

These are eight nodes containing initially –1 in the *parent* indicating that each tree contains only one node. Let's unite two trees of 4 and 6. The new tree will be like the one shown in Fig 36.2.

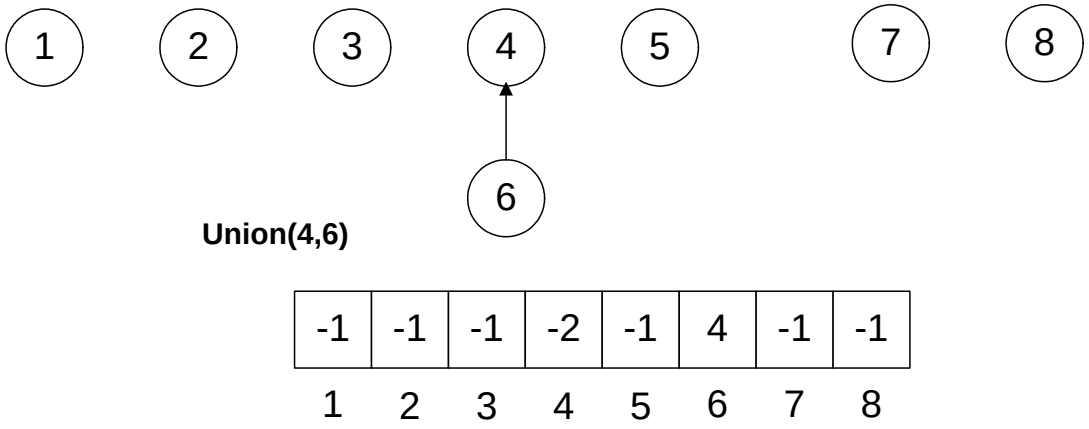


Fig 36.2

From the figure, you can see the node 6 has come down to node 4. The array also depicts that the *parent* at position 4 is containing –2. The number of nodes has become 2. The position 6 is set to 4 indicating that *parent* of 6 is 4. Next, we similarly perform the *union* operation on nodes 2 and 3.

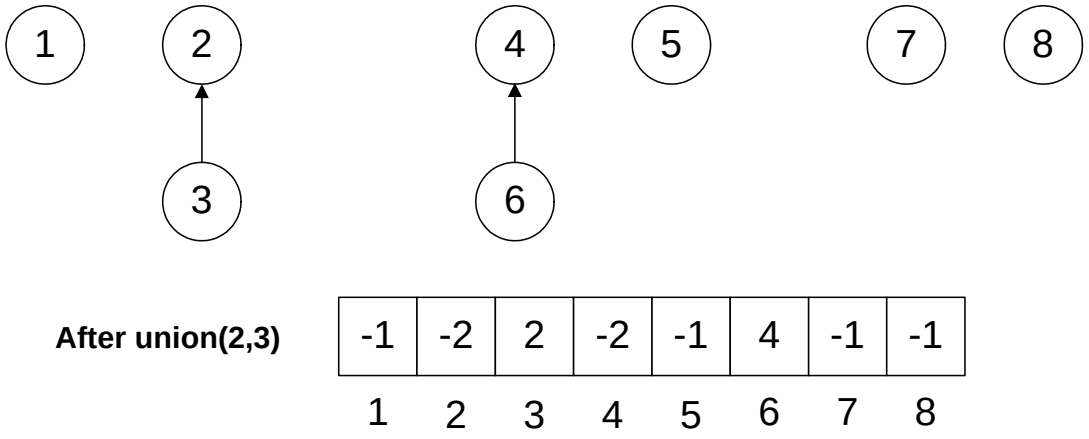


Fig 36.3

Let's perform the *union* operation further and merge the trees 1 and 4.

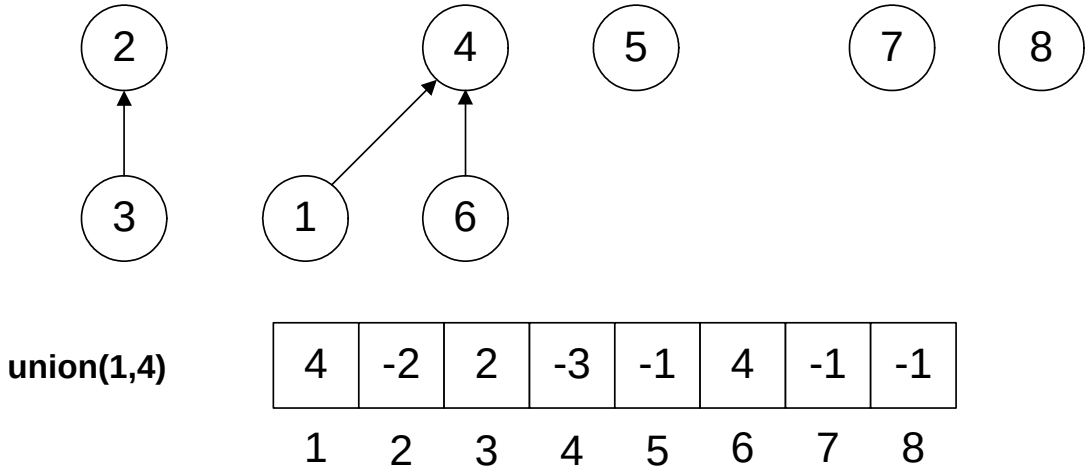


Fig 36.4

1 was a single node tree (-1 in the array at position 1) and 4 was the root node of the tree with two elements (-2 in the array at position 4) 4 and 6. As the tree with root 4 is greater, therefore, node 1 will become part of it. Previously (when the union was not based on weight), it happened contrary to it with second argument tree becoming the part of the first tree.
In this case, the number of levels in the tree still remains the same (of two levels) as that in the greater tree (with root 4). But we apply our previous logic i.e. the number of levels of the trees would have been increased from two to three after the union. Reducing the tree size was our goal of this approach.

In the next step, we merge the trees of 2 and 4. The size of the tree with root 2 is 2 (actually –2 in the parent array) while the size of the tree with root 4 is 3 (actually –3). So the tree of node 2 will be joined in the tree with root 4.

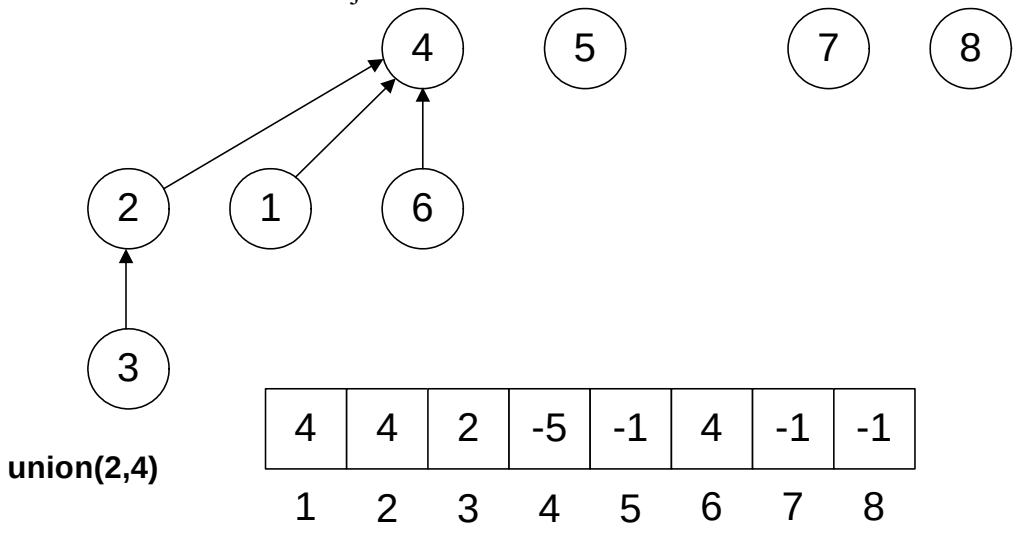


Fig 36.5

The latest values in the array i.e. at position 4, have been updated to –5 (indicating the size of the tree has reached 5). At position 2, the new value is 4, which indicates that the up node of 2 is 4.

Next, we perform the *union(5,4)* operation. We know that the size of the tree with node 4 is 5 (-5 in the array) and node 5 is single node tree. As per our rules of union by size, the node 5 will become part of the tree 4.

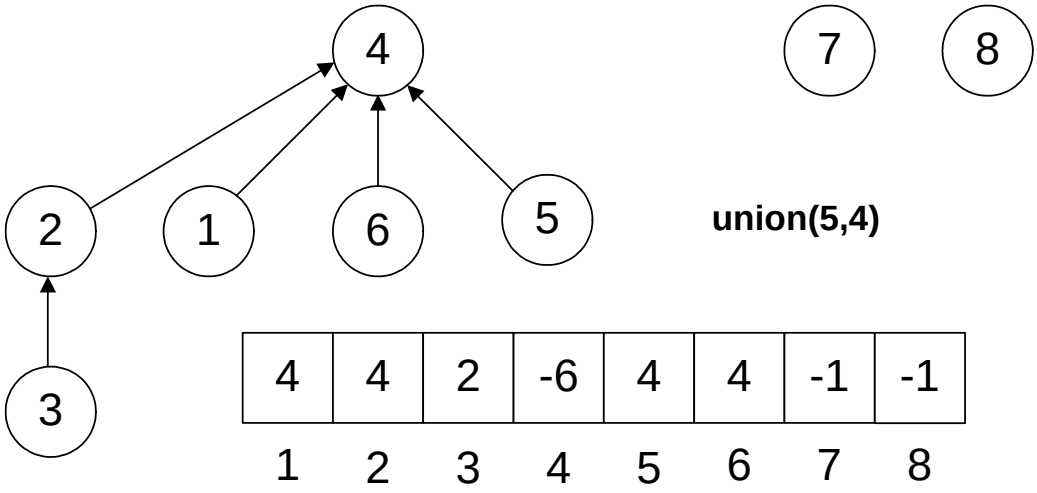


Fig 36.6

The updates inside the array can be seen as the value at position 4 has been changed to –6 and new value at position 5 is 4. This is the tree obtained after applying the *union-by-size approach*. Consider, if we had applied the previous method of union, tree's depth would have been increased more.
It seems that we have achieved our goal of reducing the tree size up to certain extent. Remember, the benefit of reducing the tree height is to increase performance while finding the elements inside the tree.

Analysis of Union by Size

– If unions are done by weight (size), the depth of any element is never greater than $\log_2 n$.

By following the previous method of union where second tree becomes the part of the first tree, the depth of the tree may extend to N. Here N is the number of nodes in the tree. But if we take size into account and perform union by size, the depth of tree is $\log_2 n$ maximum. Suppose the N is 100,000 i.e.for 100,000 nodes. The previous methods may give us a tree with depth level as 100,000. But on the other hand, $\log_2 100000$ is approximately 20.U*nion-by-size* gives us a tree of 20 levels of depth maximum. So this is a significant improvement.

Mathematical proof of this improvement is very complex .We are not covering it here but only providing the logic or reasoning in intuitive proof.

Intuitive Proof:

- Initially, every element is at depth zero.
- When its depth increases as a result of a union operation (it's in the smaller tree), it is placed in a tree that becomes at least twice as large as before (union of two equal size trees).
- How often can each union be carried out? -- $\log_2 n$ times, because after at most $\log_2 n$ unions, the tree will contain all n elements.

Union by Height

- Alternative to *union-by-size* strategy: maintain heights,
- During *union*, make a tree with smaller height a subtree of the other.
- Details are left as an exercise.

This is an alternate way of *union-by-size* that we maintain heights of the trees and join them based on their heights. The tree with smaller height will become part of the one with greater height. This is quite similar with the *union-by-size*. In order to implement Disjoint Set ADT, any of these solutions can work. Both the techniques i.e. *union-by-size* and *union-by-height* are equivalent, although, there are minor differences when analyzed thoroughly.

Now, let's see what can we do for *find* operation to make it faster.

Sprucing up Find

- So far we have tried to optimize *union*.
- Can we optimize *find*?
- Yes, it can be achieved by using *path compression* (or compaction).

Considering performance, we can optimize the *find* method. We have already optimized the trees *union* operation and now we will see how can optimize the *find* operation using the *path compression*.

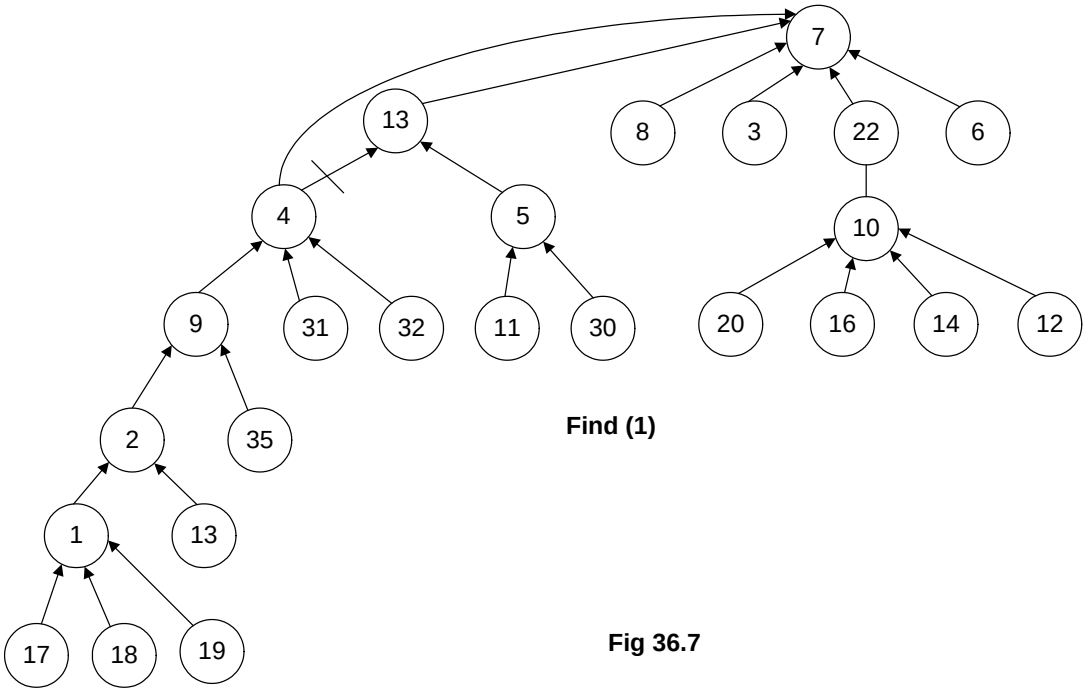
- During *find*(*i*), as we traverse the path from *i* to *root*, update *parent* entries for all these nodes to the *root*.
- This reduces the heights of all these nodes.
- Pay now, and reap the benefits later!
- Subsequent *find* may do less work.

To understand the statements above, let's see the updated code for *find* below:

```
find (i)
{
        if (parent[i] < 0)
                return i;
        else
                return parent[i] = find(parent[i]);
```

```
}
```

We have modified the code in the body of the *find*. This implementation is of recursive nature. *parent[i]* negative means that we have reached the *root* node. Therefore, we are returning i. If this is not the *root* node, then find method is being called recursively. We know that *parent[i]* may be positive or negative. But in this case it will not be negative due to being an index. To understand this recursive call, we elaborate it further with the help of figure. We call the *find* method with argument 1.



Find (1)

Fig 36.7

This tree is bigger as compared to the ones in our previous examples, it has been constructed after performing some *union* operations on trees. Now we have called *find* method with argument 1. We want to find the set with node 1. By following the previous method of *find*, we will traverse the tree from 1 to 2, from 2 to 9, 9 to 4, 4 to 13 and finally from 13 to 7, which is the *root* node, containing the negative number. So the find will return 7.

If we apply recursive mechanism given in the above code, the recursive call will start from *find(1). The parent* (*find(1))* will return 2. The recursive call for 2 will take the control to node 9, then to 4, 13 and then eventually to 7. When *find(7)* is called, we get the negative number and reach the recursion stopping condition. Afterwards, all the recursive calls, on the stack are executed one by one. The call for 13, *find(13)* returns 7. *find(4)* returns 7 because *find(parent(4))* returns 7. That is why the link between 4 and 13 has been dropped and a new link is established between 4 and 7 as shown in the Fig 36.7. So the idea is to make the traversal path shorter from a node to the *root*. The remaining recursive calls will also return 7 as the result of find operation. Therefore, for subsequent calls we establish the links directly from the node to the *root*.

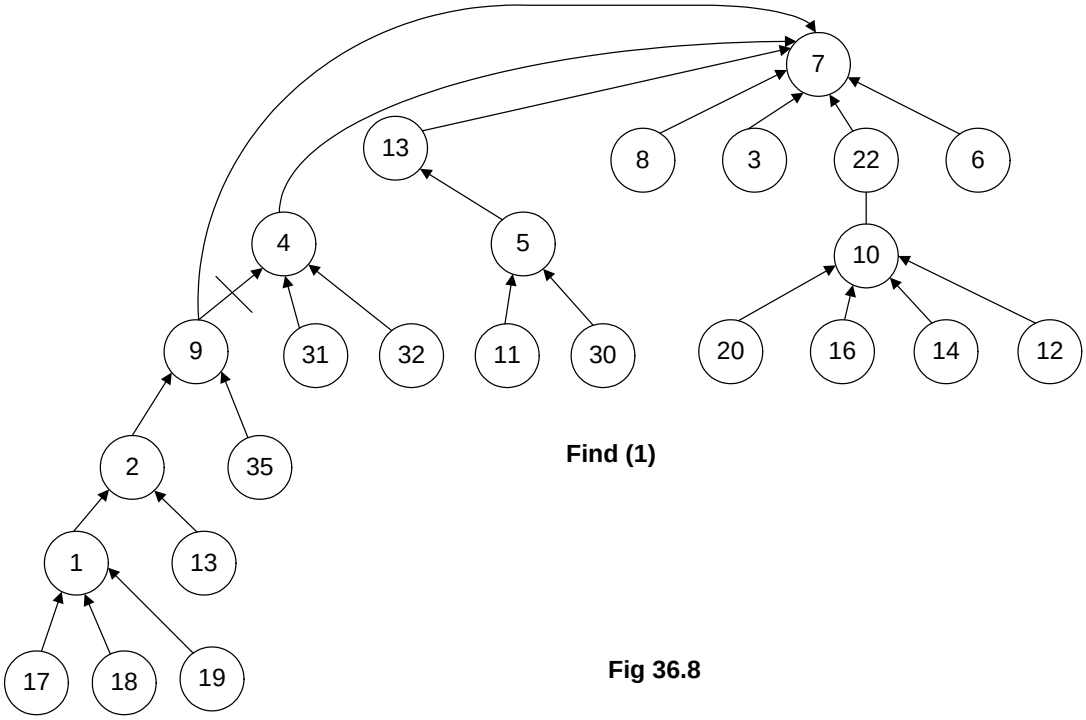


Fig 36.8

Similarly the node 2 is directly connected to the root node and the interlink between the node 2 and 9 is dropped.

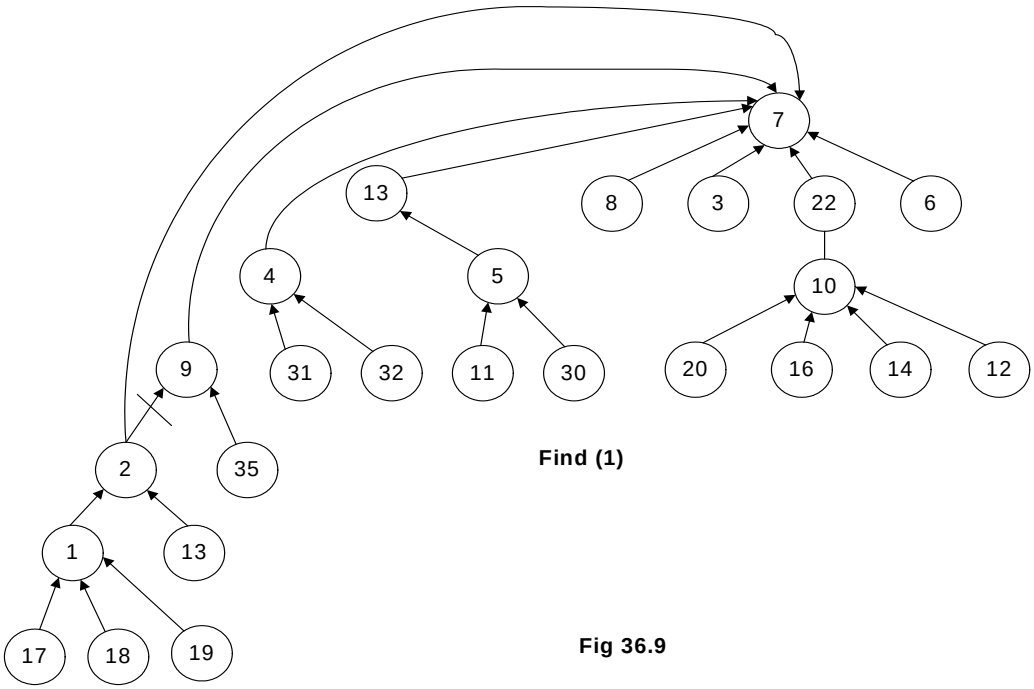
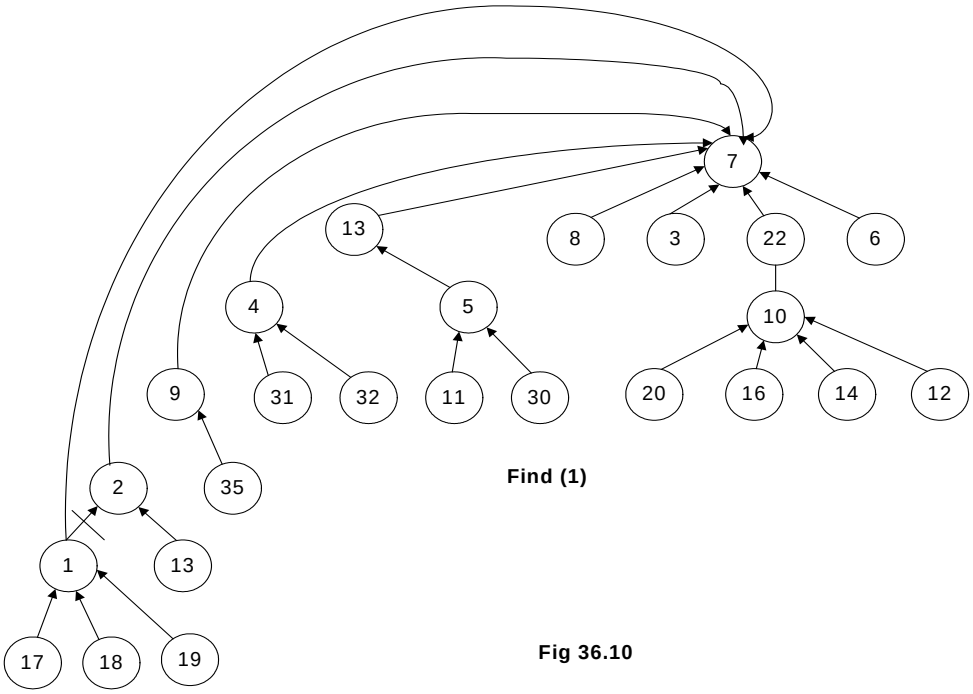


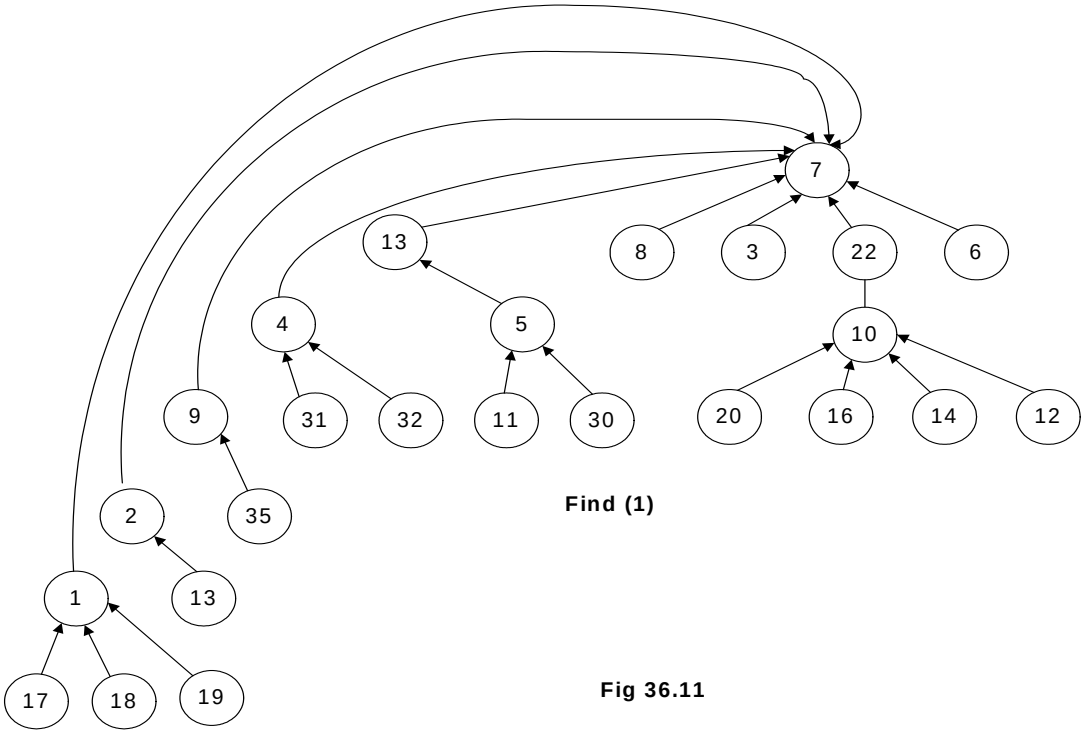
Fig 36.9

The same will happen with 1.



Find (1)

Fig 36.10



Find (1)

Fig 36.11

The *union* operation is based on size or weight but the reducing the in-between links or path compression from nodes to the *root* is done by the *find* method. The *find* method will connect all the nodes on the path to the *root* node directly.
The *path compression* is depicted by the following tree:

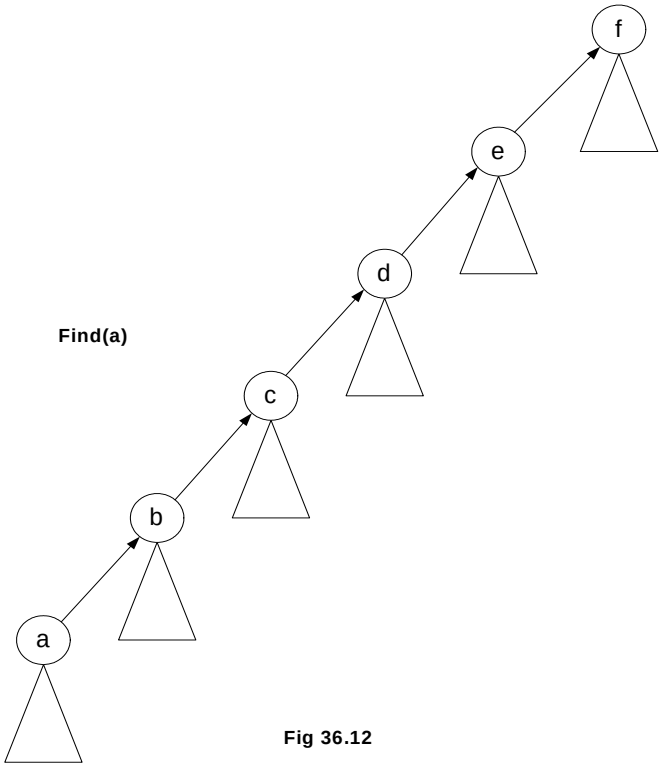


Fig 36.12

If we have called find(a) then see the path from node a to the *root* node f. a is connected to root node f through b, c, d and e nodes. Notice that there may be further subtrees below these nodes. After we apply this logic of *path compression* for *find* operation, we will have the tree as follows:

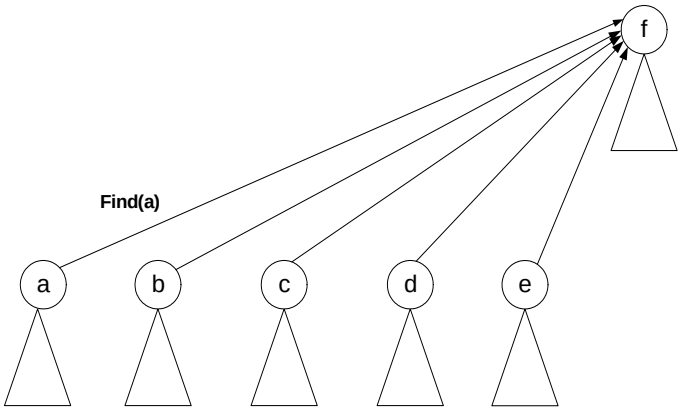


Fig 36.13

We see, how much is the impact on performance in *find* by a theorem.

Timing with Optimization

- Theorem: A sequence of *m union* and *find* operations, *n* of which are *find* operations, can be performed on a disjoint-set forest with union by rank (weight or height) and path compression in worst case time proportional to ($m\tilde{N}(n)$).
- $\tilde{N}(n)$ is the inverse Ackermann's function which grows extremely slowly. For all

practical purposes, $\tilde{N}(n)$ R 4.

– Union-find is essentially proportional to *m* for a sequence of *m* operations, linear in *m*.

There are number of things present in this theorem regarding analysis, which are difficult to cover in this course. We sill study these in course of Algorithm Analysis. At the moment, consider if there are m *union* operations out of which n are *find*s. The average time for union-find will be linear in m and it will grow not quadratically or exponentially. *union* and *find* operations using this data structure are very efficient regarding both space and time.

Data Structures

Lecture No. 37

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 8

Summary

- Review
- Image Segmentation
- Maze Example
- Pseudo Code of the Maze Generation

Review

In the last lecture, we talked about *union* and *find* methods with special reference to the process of optimization in *union*. These methods were demonstrated by reducing size of tree with the help of the techniques-union by size or union by weight. Similarly the tree size was reduced through path optimization in the *find* method. This was due to the fact that we want to reduce the tree traversal for the *find* method. The time required by the *find/union* algorithm is proportional to *m*. If we have *m* union and *n* find, the time required by find is proportional to *m*+*n*. Union is a constant time operation. It just links two trees whereas in the *find* method tree traversal is involved. The disjoint sets are increased and forest keeps on decreasing. The *find* operation takes more time as unions are increased. But on the average, the time required by *find* is proportional to *m*+*n*.

Now we will see some more examples of disjoint sets and *union/find* methods to understand their benefits. In the start, it was told that disjoint sets are used in image segmentation. We have seen the picture of a boat in this regard. We have also talked about the medical scanning to find the human organs with the help of image segmentation. Let's see the usage of *union/find* in image segmentation.

Image segmentation is a major part of image processing. An image is a collection of pixels. A value associated with a pixel can be its intensity. We can take images by cameras or digital cameras and transfer it into the computer. In computer, images are represented as numbers. These numbers may represent the gray level of the image. The zero gray level may represent the black and 255 gray level as white. The numbers between 0 and 255 will represent the gray level between black and white. In the color images, we store three colors i.e. RGB (Red, Green, Blue). By combining these three colors, new ones can be obtained.

Image Segmentation

In image segmentation, we will divide the image into different parts. An image may

be segmented with regard to the intensity of the pixels. We may have groups of pixels having high intensity and those with pixels of low intensity. These pixels are divided on the basis of their threshold value. The pixels of gray level less than 50 can be combined in one group, followed by the pixels of gray level less between 50 and 100 in another group and so on. The pixels can be grouped on the basis of threshold for difference in intensity of neighbors. There are eight neighbors of the pixel i.e. top, bottom, left, right, top-left, top-right, bottom-left and bottom-right. Now we will see the difference of the threshold of a pixel and its neighbors. Depending on the difference value, we will group the pixels. The pixels can be grouped on the basis of texture (i.e. a pattern of pixel intensities). You will study all these in detail in the image processing subject.

Let's see an example. Consider the diagram below:

It seems a sketch or a square of black, gray and white portions. These small squares represent a pixel or picture element. The color of these picture elements may be white, gray or black. It has five rows and columns each. This is a 5 * 5 image. Now we will have a matrix of five rows and five columns. We will assign the values to the elements in the matrix depending on their color. For white color, we use 0, 4 for black and 2 for gray color respectively.

	0	1	2	3	4
0	0	0	0	4	4
1	2	0	4	4	0
2	4	2	2	4	4
3	4	4	0	4	4
4	0	2	2	4	0

When we get the image from a digital device, it is stored in numbers. In the above matrix, we have numbers from 0 to 4. These can be between 0 and 255 or may be more depending upon the digital capturing device. Now there is a raw digital image. We will apply some scheme for segmentation. Suppose we need the pixels having value 4 or more and the pixels with values less than 4 separately. After finding such pixels, put 1 for those that have values more than 4 and put 0 for pixels having less than 4 values. We want to make a binary array by applying the threshold. Let's apply this scheme on it.

0	1	2	3	4

0	0	0	0	1	1
1	0	0	1	1	0
2	1	0	0	1	1
3	1	1	0	1	1
4	0	0	0	1	0

We have applied threshold equal to 4. We have replaced all 4's with 1 and all the values less than 4, have been substituted by zero. Now the image has been converted into a binary form. This may represent that at these points, blood is present or not. The value 4 may represent that there is blood at this point and the value less than 4 represents that there is no blood. It is very easy to a program for this algorithm.

Our objective was to do image segmentation. Suppose we have a robot which captures an image. The image obtained contains 0, 2 and 4 values. To know the values above and below the threshold, it may use our program to get a binary matrix. Similarly, for having the knowledge regarding the region of 1's and region of 0's, the robot can use the algorithm of disjoint sets. In the figure above, you can see that most of the 1's are on the right side of the matrix.

We will visit each row of the matrix. So far, there are 25 sets containing a single element each. Keep in mind the example discussed in *union/find* algorithm. In the zeroth row, we have three 0's. We can apply union on 0's. Yet we are interested in 1's. In the 3rd column, there is an entry of 1. Now the left neighbor of this element is 0 so we do not apply union here and move to the next column. Here, the entry is 1. Its left neighbor is also 1. We apply union here and get a set of two elements. We have traversed the first row completely and moved to the next row. There is 0 in the 0^{th} and 1st column while in the 2^{nd} column; we have an entry of 1. Its left neighbor is 0 so we move to the next column where entry is also 1. We apply union here. There is a set of 1 at the top of this set also. So we take union of these two sets and combine them. There is a set of four 1's, two from row 0 and two from row 1. In the next row, we have 1, 0, 0, 1, 1. In the row 2 column 3, there is 1. At the top of this element, we have 1. We apply union to this 1 and the previous set of four 1's. Now we have a set of five 1's. In the next column again, there is 1, also included in the set as the sixth element. In the row 3, we have 1 at the 0 column. There is 1 at the top of this so we have set of these two 1's. The next element at the row 3 is again 1, which is included in the left side set of 1's. Now we have another set of 1's having 3 elements. The 1's at column 3 and column 4 are included in the set of six 1's making it a set of eight elements. In the last row, we have single 1 at the column 3 which will be included with the eight element set. So there are two sets of 1's in this matrix.

	0	1	2	3	4
0	0	0	0	1	1
1	0	0	1	1	0
2	1	0	0	1	1
3	1	1	0	1	1
4	0	0	0	1	0

We have two disjoint sets and it is clearly visible that where are the entries of 1's and 0's. This is the situation when we have set the threshold of 4 to obtain a binary image. We can change this threshold value.

Let's take the value 2 as threshold value. It means that if the value of pixel is 2 or more than 2, replace it by 1 otherwise 0. The new image will be as:

	0	1	2	3	4
0	0	0	0	1	1
1	1	0	1	1	0
2	1	1	1	1	1
3	1	1	0	1	1
4	0	1	1	1	0

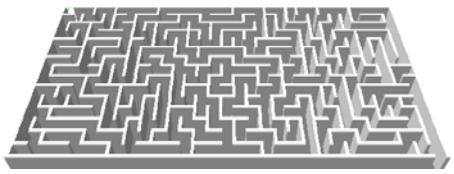
We can apply the union scheme on it to find the region of 1's. Here we have a blob of 1.

	0	1	2	3	4
0	0	0	0	1	1
1	1	0	1	1	0
2	1	1	1	1	1
3	1	1	0	1	1
4	0	1	1	1	0

The region is shaded. The image has been segmented. With the help of *union/find* algorithm, we can very quickly segment the image. The *union/find* algorithm does not require much storage. Initially, we have 25 sets that are stored in an array i.e. the up-tree. We do all the processing in this array without requiring any extra memory. For the image segmentation, disjoint sets and *union-find* algorithm are very useful. In the image analysis course, you will actually apply these on the images.

Maze Example

You have seen the maze game in the newspapers. This is a puzzle game. The user enters from one side and has to find the path to exit. Most of the paths lead to blind alley, forcing the user to go back to square one.



This can be useful in robotics. If the robot is in some room, it finds its path between the different things. If you are told to build mazes to be published in the newspaper, a new maze has to be developed daily. How will you do that? Have you done like this before? Did anyone told you to do like this? Now you have the knowledge of disjoint sets and *union-find* algorithm. We will see that the maze generation is very simple with the help of this algorithm.

Let's take a 5 * 5 grid and generate a 5 * 5 maze. A random maze generator can use *union-find* algorithm. Random means that a new maze should be generated every time.

Consider a 5x5 maze:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Here we have 25 cells. Each cell is isolated by walls from the others. These cells are numbered from 0 to 24. The line between 0 and 1 means that there is a wall between them, inhibiting movement from 0 to 1. Similarly there is a wall between 0 and 5. Take the cell 6, it has walls on 4 sides restricting to move from it to anywhere. The internal cells have walls on all sides. We will remove these walls randomly to establish a path from the first cell to the last cell.

This corresponds to an equivalence relation i.e. two cells are equivalent if they can be reached from each other (walls been removed so there is a path from one to the other). If we remove the wall between two cells, then there can be movement from one cell to the other. In other way, we have established an equivalence relationship between them. This can be understood from the daily life example that when the separation wall of two persons is removed, they become related to each other. In maze generation, we will remove the walls, attaching the cells to each other in some sequence. In the end, we will have a path from the start cell to the end cell.

First of all, we have to decide an entrance and an exit. From the entrance, we will move into the maze and have to reach at the exit point. In our example, the entrance is from the cell 0 and the exit will be at cell 24.

➔

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

➔

How can we generate maze? The algorithm is as follows:

- Randomly remove walls until the entrance and exit cells are in the same set.
- Removal of the wall is the same as doing a union operation.
- Do not remove a randomly chosen wall if the cells it separates are already in the same set.

We will take cells randomly. It means that the probability of each cell is equal. After selecting a cell, we will choose one of its surrounding walls. The internal cells have four walls around them. Now we will randomly choose one wall. In this way, we will choose the neighboring cell. Initially we have 25 sets. Now after removing a wall between 2 cells, we have combined these two cells into one set. Here the union method has been applied to combine these two cells. If these cells are already in the same set, we will do nothing.

We will keep on randomly choosing the cells and removing the walls. The cells will merge together. The elements of the set will keep on growing and at some point, we may have the entrance cell (cell 0) and the exit cell (cell 24) in the same set. When the entrance cell and the exit cell are in the same set, it means that we have a set in which the elements are related to each other and there is no wall between them. In such a situation, we can move from start to the end going through some cells of this set. Now we have at least one path from entrance to exit. There may be other paths in which we have the entrance cell but not the exit. By following this path, you will not reach at the exit as these paths take you to the dead end. As these are disjoint sets, so unless the entrance and exit cell are in the same set, you will not be able to find the solution of the maze.

Pseudo Code of the Maze Generation

Let's take a look at the pseudo code of the maze generation. We will pass it to the size argument to make a maze of this size. We will use the entrance as 0 and exit as *size-1* by default.

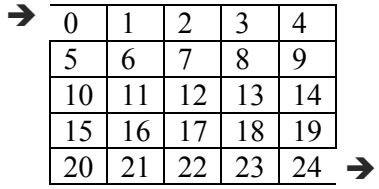
```
MakeMaze(int size) {
        entrance = 0; exit = size-1;
        while (find(entrance) != find(exit)) {
                cell1 = randomly chosen cell
                cell2 = randomly chosen adjacent cell
                if (find(cell1) != find(cell2) {
                        knock down wall between cells
                        union(cell1, cell2)
                }
        }
}
```

After initializing the entrance and exit, we have a while loop. The loop will keep on executing till the time the *find(entrance)* is not equal to *find(exit)*. It means that the loop will continue till the set returned by the *find(entrance)* and *find(exit)* are not same. When the entrance and exit cells are in the same set, the loop will stop. If these cells are not in the same set, we will enter into the loop. At first, we will randomly choose a cell from the available cells e.g. from 0 to 24 from our example. We are not discussing here how to randomly choose a cell in C++. There may be some function available to do this. We store this value in the variable *cell1 and* choose randomly its neighboring cell and store it in *cell2*. This cell may be its top, bottom, left or right cell, if it is internal cell. If the cell is at the top row or top bottom, it will not have four neighbors. In case of being a corner cell, it will have only two neighboring cells. Then we try to combine these two cells into one set. We have an if statement which checks that the set returned by *find(cell1)* is different than the set returned by *find(cell2)*. In this case, we remove the wall between them. Then we apply union on these two cells to combine them into a set.

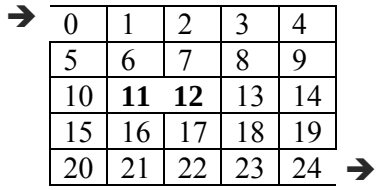
We randomly choose cells. By applying union on them, the sets are joined together and form new sets. This loop will continue till we have a set which contains both entrance and exit cell. This is a very small and easy algorithm. If we have the *union-find* methods, this small piece of code can generate the maze. Let's take a pictorial

look at different stages of maze generation. You can better understand this algorithm with the help of these figures. We are using the disjoint sets and *union-find* algorithm to solve this problem.

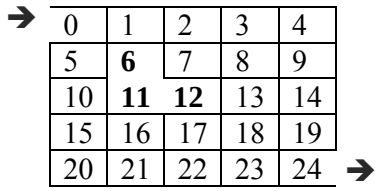
Initially, there are 25 cells. We want to generate a maze from these cells.



Apply the algorithm on it. We randomly choose a cell. Suppose we get *cell 11*. After this, we randomly choose one of its walls. Suppose we get the right wall. Now we will have *cell 11* in the variable *cell1* and *cell 12* in the variable *cell2*.

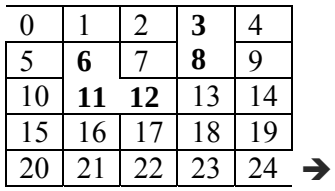


By now, each cell is in its own set. Therefore *find(cell 11)* will return *set_11* and *find(cell 12)* will return *set_12*. The wall between them is removed and union is applied on them. Now we can move from *cell 11* to *cell 12* and vice versa due to the symmetry condition of disjoint sets. We have created a new set (set_11 = {11,12}) that contains *cell 11* and *cell 12* and all other cells are in their own cells.

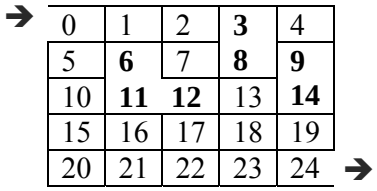


Now we randomly choose the *cell 6* and its bottom wall. Now the *find(cell 6)* will return *set_6*. The *find(cell 11)* will return *set_11* which contains *cell 11* and *cell 12*. The sets returned by the two *find* calls are different so the wall between them is removed and union method applied on them. The set *set_11* now contains three elements *cell 11*, *cell 12* and *cell 6*. These three cells are joined together and we can move into these cells.

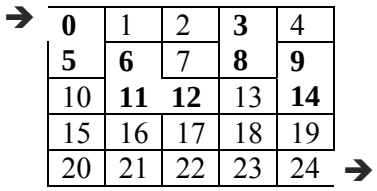
Now we randomly select the *cell 8*. This cell is not neighbor of *set_11* elements. We can randomly choose the *cell 6* again and its bottom wall. However, they are already in the same set so we will do nothing in that case. We randomly choose the *cell 8* and its top wall (cell 3). As these two cells are in different sets, the wall between them is removed so that union method could be applied to combine them into a set (set_8 = {8, 3}).



Now we randomly choose the *cell 14* and its top i.e. *cell 9*. Now we keep on combining cells together but so far entrance and exit cells are not in the same set.



We randomly choose *cell 0* and its bottom wall so the *cell 0* and *cell 5* are combined together.



If you keep on using this algorithm, the walls between cells will keep on vanishing, resulting in the growth of set elements. At some point, the entrance and exit cell will come into one set and there will be a path taking us from the start to exit. There may be some other sets which do not contain the exit cell. Take it as a programming exercise and do it. This is very interesting exercise. Find some information on the internet regarding maze generation. In the next lecture, we will talk about the new data type i.e. table.

Data Structures

Lecture No. 38

Summary

- Table and Dictionaries
- Operations on Table ADT
- Implementation of Table
  - Unsorted Sequential Array
  - Sorted Sequential Array
- Binary Search

We will discuss the concepts of Tables and Dictionaries in this lecture with special reference to the operations on Table ADT and the implementation.

Tables and Dictionaries

The table, an abstract data type, is a collection of rows and columns of information. From rows and columns, we can understand that this is like a two dimensional array. But it is not always a two dimensional array of same data type. Here in a table, the type of information in columns may be different. The data type of first column may be integer while the type of second column is likely to be string. Similarly there may be other different data types in different columns. So the two-dimensional array used for table will have different data types.

A table consists of several columns, known as fields. These fields are some type of information. For example a telephone directory may have three fields i.e. *name*, *address* and *phone number*. On a computer system, the user account may have fields- *user ID*, *password* and *home folder*. Similarly a bank account may have fields like *account number*, *account title*, *account type* and *balance of account*.

Following is a table that contains three fields (We are calling columns as fields). These three fields are *name*, *address* and *phone*. Moreover, there are three rows shown in the table.

Name	Address	Phone
Sohail Aslam	50 Zahoor Elahi Rd, Gulberg-4, Lahore	567-4567
Imran Ahmad	30-T Phase-IV, LCCHS, Lahore	517-2349
Salman Akhtar	131-D Model Town, Lahore	756-5678

Figure 38.1: A table having three fields.

Each row of the table has a set of information in the three fields. The fields of a row are linked to each other. The row of a table is called a record. Thus the above table contains three records. It resembles to the telephone directory. There may be other fields in the phone directory like *father's name* or the *occupation* of the user. Thus the data structure that we form by keeping information in rows and columns is called table.

During the discussion on the concept of table, we have come across the word,

dictionary. We are familiar with the language dictionary, used to look for different words and their meanings. But for a programmer, the dictionary is a data type. This data is in the form of sets of fields. These fields comprise data items.

A major use of table is in Databases where we build and use tables for keeping information. Suppose we are developing payroll software for a company. In this case, a table for employees will be prepared. This table of employees has the name, id number, designation and salary etc of the employees. Similarly there may be other tables in this database e.g. to keep the leave record of the employees. There may be a table to keep the personal information of an employee. Then we use SQL (Structured Query Language) to append, select, search and delete information (records) from a table in the database.

Another application of table is in compilers. In this case, symbol tables are used. We will see in compilers course that symbol table is an important structure in a compiler. In a program, each variable declared has a, type and scope. When the compiler compiles a program, it makes a table of all the variables in the program. In this table, the compiler keeps the name, type and scope of variables. There may be other fields in this table like function names and addresses. In the compiler construction course, we will see that symbol table is a core of compiler.

When we put the information in a table, its order will be such that each row is a record of information. The word *tuple* is used for it in databases. Thus a row is called a record or tuple. As we see there is a number of rows (records) in a table. And a record has some fields.

Now here are some things about tables.

To find an *entry* in the table, we only need to know the contents of one of the fields (not all of them). This field is called *key*. Suppose we want to see information (of a record) in dictionary. To find out a record in the dictionary, we usually know the name of the person about whom information is required. Thus by finding the name of the person, we reach a particular record and get all the fields (information) about that person. These fields may be the address and phone number of the person. Thus in telephone directory, the *key* is the *name* field. Similarly in a user account table, the key is usually *user id*. By getting a user id, we can find its information that consists of other fields from the table. The *key* should be unique so that there is only one record for a particular value of *key*. In the databases, it is known as *primary key*. Thus, in a table, a key uniquely identifies an entry. Suppose if the name is the *key* in telephone book then no two entries in the telephone book have the same name. The *key* uniquely identifies the entries. For example if we use the *name* "imran ahmed" in the following table that contains three entries, the name "imran ahmed" is there only at one place in the following table. In this record, we can also see the address and phone number of "imran ahmed".

Name	Address	Phone
Sohail Aslam	50 Zahoor Elahi Rd, Gulberg-4, Lahore	567-4567
Imran Ahmad	30-T Phase-IV, LCCHS, Lahore	517-2349
Salman Akhtar	131-D Model Town, Lahore	756-5678

Figure 38.2: A table having three fields.

Similarly if we use "salman akhtar" as key we find out one record for this key in the table.

The purpose of this key is to find data. Before finding data, we have to add data. On the addition of the data in the table, the finding of data will be carried through this *key*. We may need to delete some data from the table. For example, in the employees' table, we want to delete the data of an employee who has left the company. To delete the data from the table, the *key* i.e. *name* will be used. However in the employees' table, the *key* may be the *employee id* and not the *name*.

Operations on Table ADT

Now we see the operations (methods) that can be performed with the table abstract data type.

insert

As the name shows this method is used to insert (add) a record in a table. For its execution, a field is designated as *key*. To insert a record (entry) in the table, there is need to know the *key* and the entry. The insert method puts the *key* and the other related fields in a table. Thus we add records in a table.

find

Suppose we have data in the table and want to find some particular information. The find method is given a *key* and it finds the entry associated with the key. In other words, it finds the whole record that has the same *key* value as provided to it. For example, in employees table if *employee id* is the key, we can find the record of an employee whose *employee id* is, say, 15466.

remove

Then there is the remove method that is given a value of the *key to* find and remove the entry associated with that key from the table.

Implementation of Table

Let's talk about why and how we should implement a table. Our choice for implementation of the Table ADT depends on the answers to the following.

- How often entries are inserted, found and removed?
- How many of the possible key values are likely to be used?
- What is the likely pattern of searching for keys? Will most of the accesses be to just one or two key values?
- Is the table small enough to fit into the memory?
- How long will the table exist?

In a table for searching purposes, it is best to store the key and the entry separately (even though the key's value may be inside the entry). Now, suppose we have a record of a person 'Saleem' whose address is '124 Hawkers Lane' while the phone number is '9675846'. Similarly we have another record of a person 'Yunus'. The address and phone fields of this person are '1 Apple crescent' and '622455' respectively. For these records in the table, we will have two parts. One part is the complete entry while the other is the key in which we keep the unique item of the entry. This unique item is twice in a record one as part of the entry and the second in a field i.e. the key field. This key will be used for searching and deletion purposes of

records. With the help of key, we reach at the row and can get other fields of it. We also call *TableNode* to row of the table. Its pictorial representation is given below.

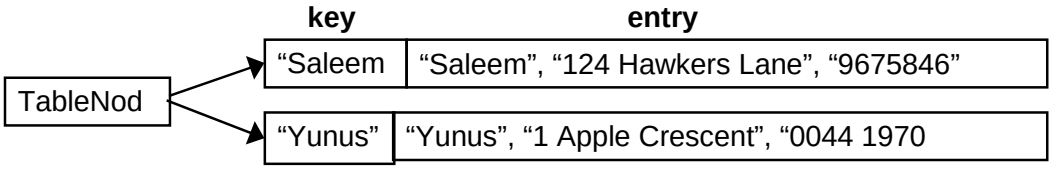


Figure 38.3: key and entry in table

Now we will discuss the implementation of table with different data structures. The first implementation is the unsorted sequential array.

Unsorted Sequential Array

In this implementation, we store the data of table in an array such that TableNodes are stored consecutively in any order. Each element of the row will have a key and entry of the record.

Now let’s think how we can store the data of a table in an array. For a telephone directory, there may be a field name i.e. a string. Then there is ‘address’, which is also a string but of large size as compared to the name. The phone number may be a string or integer. Each record in the directory has these three fields with different types. How can we store this information in an array? Here comes the class. Suppose we make a class *Employee*, with an ultimate aim of developing objects. An object of *Employee* will contain the name, address, designation and salary. Similarly we can make a class of *PhoneDirectoryEntry* that will have name, address and phone number. We can add other fields to these classes. Thus we have *Employee* and *PhoneDirectoryEntry* objects. Now we make an array of objects. In this array, every object is a table entry. To insert a new object (of Employee or PhoneDirectoryEntry), we add it to the back of the array. This insert operation in the array is fast as we directly put the data at the last available position. Thus we add the data to the array as soon as it is available. In other words, we don’t have an order of the data. Now if there are n entries in the array, the find operation searches through the keys one at a time and potentially all of the keys to find a particular key. It has to go through all the entries (i.e. n) if the required key is not in the table. Thus the time of searching will be proportional to the number of entries i.e. n. Similarly the remove method also requires time proportional to n. The remove method, at first, has to find the key, needed to be removed. It consumes the time of find operation that is proportional to the number of entries i.e. n. If the entry is found, the remove method removes it. Obviously it does nothing if the entry is not found.

Here we see that in unsorted sequential array, the insertion of data is fast but the find operation is slow and requires much time. Now the question arises if there is any way to keep an array in which the search operation can be fast? The use of sorted sequential array is a solution to this problem.

Sorted Sequential Array

We have studied in the tree section that binary search tree is used to search the information rapidly. Equally is true about the sorted sequential array. But in this case, we want to put the data in an array and not in a tree. Moreover, we want to search the data in this array very fast. To achieve this objective, we keep the data in the array in a sorted form with a particular order. Now suppose we are putting the information in the *telephoneDirectory* table into an array in a sorted form in a particular order. Here, for example, we put the data alphabetically with respect to name. Thus data of a person whose name starts with 'a' will be at the start of the table and the name starting with 'b' will be after all the names that start with 'a'. Then after the names starting with 'b' there will be names starting with 'c' and so on. Suppose a new name starting from 'c' needs to be inserted. We will put the data in the array and sort the array so that this data can be stored at its position with respect to the alphabetic order. Later in this course, we will read about sorting.
Let's talks about the insert, find and remove methods for sorted data.

insert
For the insertion of a new record in the array, we will have to insert it at a position in the array so that the array should be in sorted form after the insertion. We may need to shift the entries that already exist in the array to find the position of the new entry. For example, if a new entry needs to be inserted at the middle of the array, we will have to shift the entries after the middle position downward. Similarly if we have to add an entry at the start of the array, all the entries will be moved in the array to one position right (down). Thus we see that the *insert* operation is proportional to *n* (number of entries in the table). This means *insert* operation will take considerable time.

find
The *find* operation on a sorted array will search out a particular entry in *log n* time by the binary search. The binary search is a searching algorithm. Recall that in the tree, we also find an item in *log n* time. The same is in the case of sorted array.

remove
The *remove* operation is also proportional to *n*. The *remove* operation first finds the entry that takes *log n* time. While removing the data, it has to shuffle (move) the elements in the array to keep the sorted order. This shuffling is proportional to *n*. Suppose, we remove the first element from the array, then all the elements of the array have to be moved one position to left. Thus *remove* method is proportional to *n*.

Binary Search

The binary search is an algorithm of searching, used with the sorted data. As we have sorted elements in the array, binary search method can be employed to find data in the array. The binary search finds an element in the sorted array in *log n* time. If we have 100000 elements in the array, the log 1000000 will be 20 i.e. very small as compared to 100000. Thus binary search is very fast.
The binary search is like looking up a phone number in the directory or looking up a word in the dictionary. For looking a word in the dictionary, we start from the middle in the dictionary. If the word that we are looking for comes before words on the page, it shows that the word should be before this page. So we look in the first half.

Otherwise, we search for the word in the second half of the dictionary. Suppose the word is in the first half of the dictionary, we consider first half for looking the word. We have no need to look into the second half of the dictionary. Thus the data to be searched becomes half in a step. Now we divide this portion into two halves and look for the word. Here we again come to know that the word is in the first half or in the second half of this portion. The same step is repeated with the part that contains the required word. Finally, we come to the page where the required word exists. We see that in the binary search, the data to be searched becomes half in each step. And we find the entry very fast. The number of maximum steps needed to find an entry is *log n*, where *n* is the total number of entries. Now if we have 100000 entries, the maximum number of attempts (steps) required to find out the entry will be 20 (i.e. log 1000000).

Data Structures

Lecture No. 39

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 10
10.4.2

Summary

- Searching an Array: Binary Search
- Binary Search - Example 1
- Binary Search - Example 2
- Binary Search - Example 3
- Binary Search – C++ Code
- Binary Search – Binary Tree
- Binary Search - Efficiency
- Implementation 3 (of Table ADT): Linked List
- Implementation 4 (of Table ADT): Skip List
- Skip List - Representation
- Skip List - Higher Level Chains
- Skip List - Formally

Searching an Array: Binary Search

In the previous lecture, we had started discussion on Binary Search Tree algorithm. The discussion revealed that if already sorted data is available, then it is better to apply an algorithm of binary search for finding some item inside instead of searching from start to the end in sequence. The application of this algorithm will help get the results very quickly. We also talked about the example of directory of employees of a company where the names of the employee were sorted. For efficient searching, we constructed the binary search tree for the directory and looked for information about an employee named 'Ahmed Faraz'.

We also covered:

– Binary search is like looking up a phone number or a word in the dictionary
– Start in middle of book
– If the name you're looking for, comes before names on the page, search in the first half
– Otherwise, look into the second half

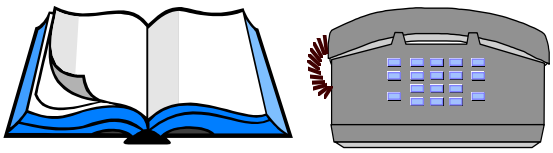


Fig 39.1

The telephone directory is the quotable example to understand the way the binary

search method works.

In this lecture, we will focus on data structures for performing search operation. Consider the data is present in an array as we discussed in the previous lecture. For the first implementation, we supposed that the data is not sorted in the array. For second implementation, we considered that the data inside the array is put in sorted array. The advantage of the effort of putting the data in the array in sorted order pays off when the searches on data items are performed.
Now, let's first see the algorithm (in pseudo code) for this operation below. It is important to mention that this algorithm is independent of data type i.e. the data can be of any type numeric or string.

```
if ( value == middle element )
    value is found
else if ( value < middle element )
        search left half of list with the same method
    else
        search right half of list with the same method
```

The item we are searching for in this algorithm is called *value*. The first comparison of this *value* is made with the *middle element* of the array. If both are equal, it means that we have found our desired search item, which is present in the middle of the array. If this is not the case, then the *value* and the *middle element* are not the same. The *else-if* part of the algorithm is computed, which checks if the *value* is less than the *middle element*. If so, the left half part of the array is searched further in the same fashion (of logically splitting that half part of array into two further halves and applying this algorithm again). This search operation can also be implemented using the recursive algorithm but that will be discussed later. At the moment, we are discussing only the non-recursive algorithm. The last part of the algorithm deals with the case when the *value* is greater than the *middle element.* This processes the right half of the array with the same method.

Let's see this algorithm in action by taking an example of an array of elements:

Binary Search – Example 1

Case 1: ***val == a[mid]***
val = 10
low = 0, high = 8
mid = (0 + 8) / 2 = 4

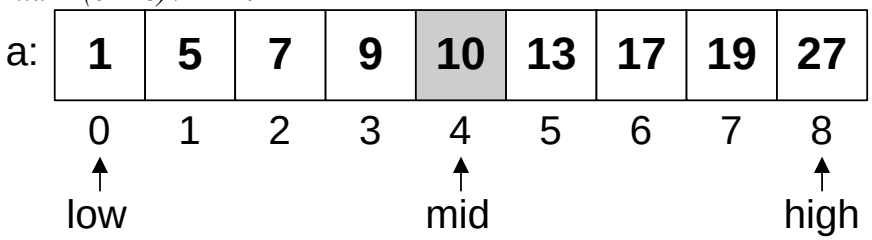


Fig 39.2

You can see an array *a* in the Fig 39.2 with indexes 0 to 8 and values 1, 5, 7, 9, 10, 13, 17, 19 and 27. These values are our data items. Notice that these are sorted in ascending (increasing) order.

You can see in the first line of case 1 that *val = 10*, which indicates that we are searching for value 10. From second line, the range of data to search is from 0 to 8. In this data range, the middle position is calculated by using a simple formula *(low + high)/2*. In this case, it is *mid =(0+8)/2=4*. This is the middle position of the data array. See the array in the above figure Fig 39.2, which shows that the item at array position 4 is 10, exactly the value we are searching for. So, in this case, we have found the value right away in the middle position of the array. The search operation can stop here and an appropriate value can be returned back.

Let's see the case 2 now:

Binary Search – Example 2

Case 2: ***val > a[mid]***
val = 19
low = 0, high = 8
mid = (0 + 8) / 2 = 4
new low = mid + 1 = 5

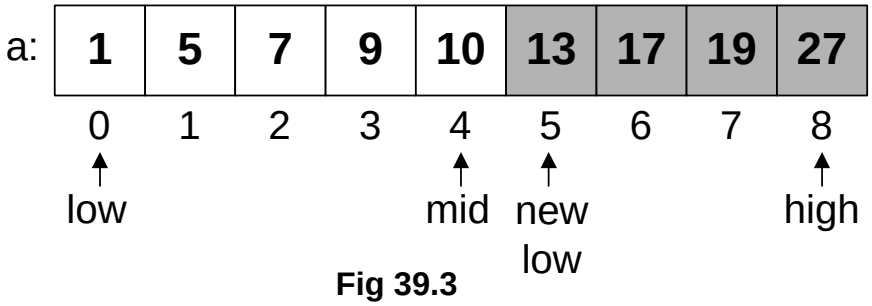


Fig 39.3

The second case is about the scenario when value (*val*) is greater than the middle value (*a[mid]*). The range of data items (*low* and *high*) is the same as that in case 1. Therefore, the middle position (*mid*) is also the same. But the value (*val*) 19 is greater than the value at the middle (mid) 10 of the array. As this array is sorted, therefore, the left half of the array must not contain value 19. At this point of time, our information about val 19 is that it is greater than the middle. So it might be present in the right half of the array. The right half part starts from position 5 to position 8. It is shown in Fig 39.3 that the *new low* is at position 5. With these new low and high positions, the algorithm is applied to this right half again.

Now, we are left with one more case.

Binary Search – Example 3

Case 3: ***val < a[mid]***
val = 7
low = 0, high = 8
mid = (0 + 8) / 2 = 4
new high = mid - 1 =5

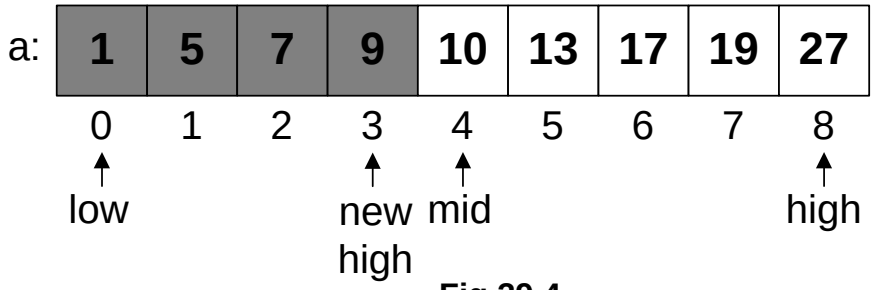


Fig 39.4

The value to be searched (*val*) in this case is 7. The data range is the same starting from *low=0* to *high=8*. Middle is computed in the same manner and the value at the middle position (*mid*) is compared with the *val*. The *val* is less than the value at *mid* position. As the data is sorted, therefore, a value lesser than the one at mid position should be present in the lower half (left half) of the array (if it is there). The left half of the array will start from the same starting position low=0 but the *high* position is going to be changed to *mid-1 i.e. 3*. Now, let's execute this algorithm again on this left half.

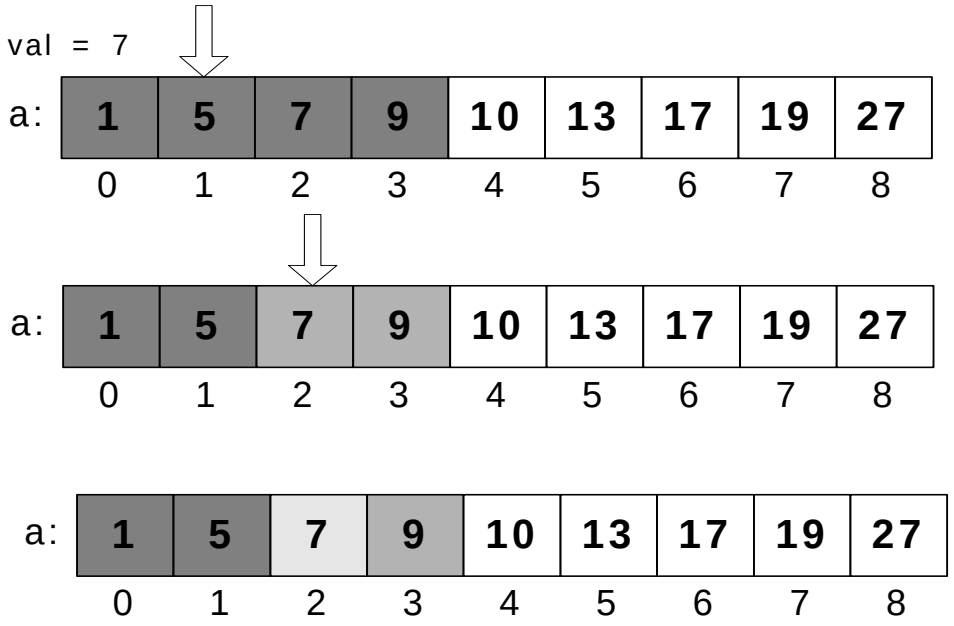


Fig 39.5

Firstly, we will compute the middle of 0 and 3 that results in 1 in this integer division. This is shown in the top array in Fig 39.5. Now, the *val* 7 is compared with the value at the middle position (mid) i.e.5. As 7 is greater than 5, we will process the right half of this data range (which is positioned from *low=0* to *high=3*). The right half of this data range starts from position 2 and ends at position 3. The new data range is *low=2* and *high=3*. The middle position (*mid*) is computed as (2+3)/2=2. The value at the *mid* position is 7. We compare the value at *mid* position (7) to the *val* we are looking for. These are found to be equal and finally we have the desired value.

Our desired number is found within positions- 0 to 8 at position 2. Without applying this binary search algorithm, we might have performed lot more comparisons. You might feel that finding this number 7 sequentially is easier as it is found at position 2 only. But what will happen in case we are searching for number 27. In that case, we

have to compare with each element present in the array to find out the desired number. On the other hand, if this number 27 is searched with the help of the binary search algorithm, it is found in third comparison.
Actually, we have already seen binary search while studying the binary search tree. While comparing the number with *the root* element of the tree, we had come to know that if the number was found smaller than the number in the *root*, we had to switch to left-subtree of the *root* (ensuring that it cannot be found in the right subtree).

Now, let's see the C++ code for this algorithm:

Binary Search – C++ Code

```
int  isPresent(int  *arr, int  val, int  N)
{
  int  low = 0;
  int  high  =  N - 1;
  int  mid;
  while ( low <= high )
  {
                mid  =  ( low + high )/2;
                if (arr[mid] == val)
                  return 1;       // found!
                else if (arr[mid] < val)
                   low  =  mid + 1;
                else
                  high  =  mid - 1;
}
return  0; // not found
```

The name of the routine is *isPresent*, which expects an *int* pointer (an array in actual); an *int* value *val* is required to be searched. Another int value is N that indicates the maximum index value of the array. Inside the body of the function, *int* variables *low*, *high* and *mid* are declared. *low* is initialized to 0 and high is initialized to *N-1*. *while* loop is based on the condition that executes the loop until *low* <= *high*. Inside the loop, very first thing is the calculation of the middle position (*mid*). Then comes the first check inside the loop, it compares the *val* (the value being searched) with the number at middle position (*arr[mid]*). If they are equal, the function returns 1. If this condition returns false, it means that the numbers are unequal. Then comes the turn of another condition. This condition (*arr[mid]* < *val*) is checking if the value at the middle is less than the value being searched. If this is so, the right half of the tree is selected by changing the position of the variable *low* to *mid+1* and processed through the loop again. If both of these conditions return false, the left half of the array is selected by changing the variable *high* to *mid-1*. This left half is processed through the loop again. If the loop terminates and the required value is not found, then 0 is returned as shown in the last statement of this function.
You change this function to return the position of the value if it is found in the array otherwise return –1 to inform about the failure. It is important to note that this function requires the data to be sorted to work properly. Otherwise, it will fail.

This algorithm is depicted figurative in Fig 39.6.

Binary Search – Binary Tree

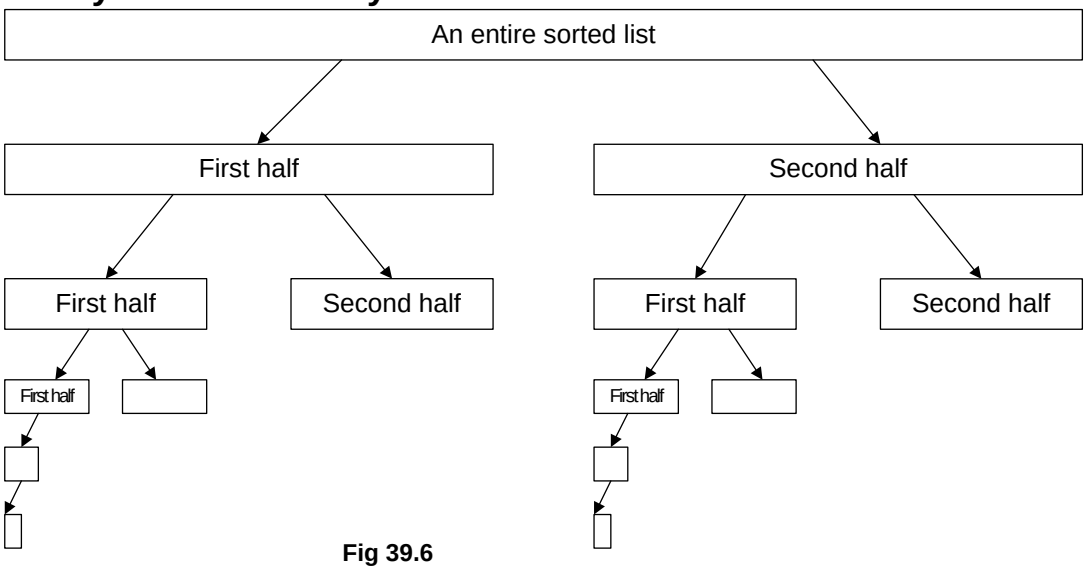


Fig 39.6

– The search divides a list into two small sub-lists till a sub-list is no more divisible.

You might have realized about the good performance of binary trees by just looking at these if you remember the fully balanced trees of N items discussed earlier.

Binary Search - Efficiency

To see the efficiency of this binary search algorithm, consider when we divide the array of N items into two halves first time.

After 1 bisection	*N/2*	*items*
After 2 bisections	$N/4 = N/2^2$	*items*
. . .		
After i bisections	$N/2^i = 1$	*item*

$i = log_2 N$

First half contains N/2 items while the second half also contains around N/2 items. After one of the halves is divided further, each half contains around N/4 elements. At this point, only one of N/4 items half is processed to search further. If we carry out three bisections, each half will contain N/8 items. Similarly for i bisections, we are left with $N/2^i$, which is at one point of time is only one element of the array. So we have the equation here:

$$N/2^i = 1$$

Computing the value of i from this gives us:

$$i = log_2 N$$

Which shows that after maximum $log_2 N$ bisections, either you will be successful in finding your item or fail to do so.

This was our second implementation of table or dictionary abstract data type using sorted sequential array. As discussed at start of it that if we implement table or

dictionary abstract data type using an array, we have to keep the array elements in sorted order. An easier way to sort them can be that whenever we want to *insert* an element in the array, firstly we find its position (in sorted order) in the array and then shift the right side (of that position) elements one position towards right to insert it. In worst case, we might have to shift all the elements one position towards right just to keep the data sorted so this will be proportional to N. Similarly the *remove* operation is also proportional to N. But after keeping the data sorted, the search operation is returned within maximum $log_2 N$ bisections.

Implementation 3 (of Table ADT): Linked List

We might also use linked list to implement the table abstract data type. We can keep the data unsorted and keep on inserting the new coming elements to *front* in the list. It is also possible to keep the data in sorted order. For that, to insert a new element in the list, as we did for array, we will first find its position (in sorted order) in the list and then insert it there. The search operation cannot work in the same fashion here because binary search works only for arrays. Because the linked list may not be contiguous in memory, normally its nodes are scattered through, therefore, binary search cannot work with them.

- *TableNodes are again stored consecutively (unsorted or sorted)*
- ***insert**: add to front; (1or n for a sorted list)*
- ***find**: search through potentially all the keys, one at a time; (n for unsorted or for a sorted list*
- ***remove**: find, remove using pointer alterations; (n)*

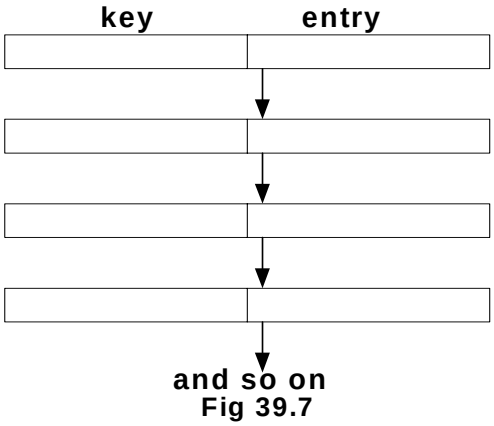


Fig 39.7

Well, linked list is one choice to implement table abstract data type. For unsorted elements, the insertion at *front* operation will take constant time. (as each element is inserted in one go). But if the data inside the list is kept in sorted order then to insert a new element in the list, the entire linked list is traversed through to find the appropriate position for the element.

For *find* operation, all the keys are scanned through whether they are sorted or unsorted. That means the time of *find* is proportional to N.

For *remove* operation, we have to perform *find* first. After that the element is removed

and the links are readjusted accordingly.

We know that when we used sorted array, the *find* operation was optimized. Let's compare the usage of array and linked list for table abstract data type. The fixed size of the array becomes a constraint that it cannot contain elements more than that. Linked list has no such constraint but the *find* operation using linked list becomes slower. Is it possible to speed up this operation of *find* while using linked list? For this, a professor of University of Maryland introduced a new data structure called skip list. Let's discuss little bit about *skip list*.

Implementation 4 (of Table ADT): Skip List

- Overcome basic limitations of previous lists
  - Search and update require linear time
- Fast Searching of Sorted Chain
- Provide alternative to BST (binary search trees) and related tree structures. Balancing can be expensive.
- Relatively recent data structure: Bill Pugh proposed it in 1990.

Important characteristics of skip list are stated above. Now, we see skip list in bit more detail.

Skip List - Representation

Can do better than n comparisons to find element in chain of length *n*

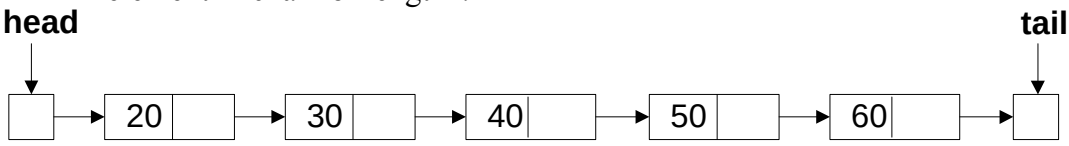


Fig 39.8

As shown in the figure. The *head* and *tail* are special nodes at start and end of the list respectively. If we have to find number 60 in the list then we have no other choice but starting from *head* traverse the subsequent nodes using the *next* pointer until the required node is found or the *tail* is reached. To find 70 in the list, we will scan through the whole list and then get to know that it is not present in it. The professor Pugh suggested something here:

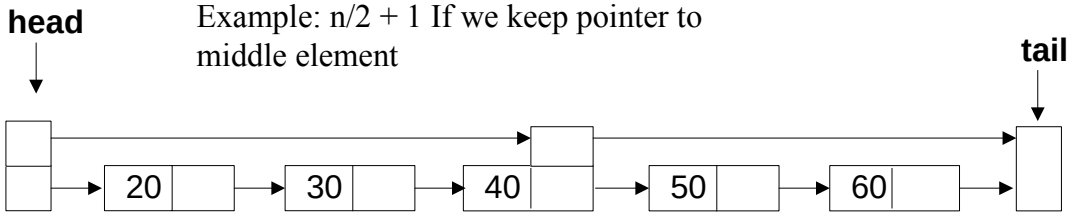


Fig 39.9

Firstly, we use two pointers *head* and *tail*. Secondly, the node in the middle has two *next* pointers; one is the old linked list pointer leading to next node 50 and the second is leading to the *tail* node. Additionally the *head* node also has two pointers, one is the old liked list pointer pointing to the next node 20 and second one is pointing to the middle element's *next* pointer, which is (as told above) further pointing to the *tail* node.
Now, if we want to find element 60 in the above list. Is it possible to search the list in relatively quick manner than the normal linked list shown in Fig 39.8? Yes, it is with the help of the additional pointers we have placed in the list. We will come to the middle of the list first and see that the middle element (40) is smaller than 60, therefore the right half part of the list is selected to process further (as the linked list is sorted). Isn't it the same we did in binary search? It definitely is.

What if we can add additional pointers (links) and boost the performance. See the figure below.

Skip List - Higher Level Chains

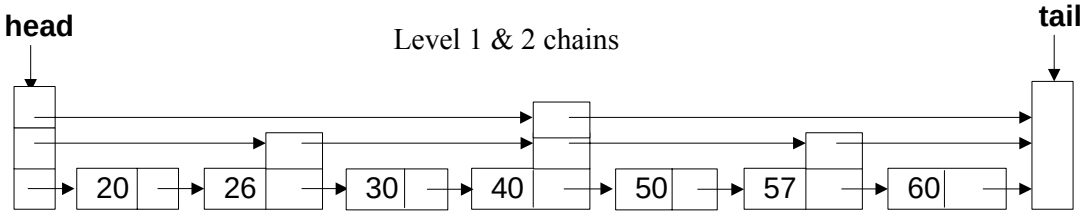


Fig 39.10

– For general n, level 0 chain includes all elements
– level 1 every other element, level 2 chain every fourth, etc.
– level *i*, every 2^i th elementLevel 0 chain is our old linked list chain as shown in Fig 39. Level 1 is new chain added to contain the link of every other node (or alternate node). Level 2 chain contains links to every 4^{th} node. We keep on adding levels of chains so that we generalize that for level i chain includes 2^ith elements. After adding these pointers, the skip list is no more our old linked list; it has become sort of binary tree. Still, there are still few important things to consider.

– *Skip list contains a hierarchy of chains*
– *In general level i contains a subset of elements in level i-1*Skip list becomes a hierarchy of chains and every level contains a subset of element of previous level. Using this kind of skip list data structure, we can find elements in $\log_2 n$ time. But the problem with this is that the frequency of pointers is so high as compared to the size of the data items that it becomes difficult to manage them. The *insert* and *remove* operations on this kind of skip list become very complex because single insertion or removal requires lot of pointers to readjust.
Professor Pugh suggested here that instead of doing leveling in powers of 2, it should be done randomly. Randomness in skip lists is a new topic for us. Let's see a formal definition of skip list.

Skip List - Formally

– A skip list for a set S of distinct (key, element) items is a series of lists S_0, S_1 , … , S_h such that
  - Each list S_i contains the special keys $+\infty$ and $-\infty$
  - List S_0 contains the keys of S in non-decreasing order Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \cdots \supseteq S_h$
  - List S_h contains only the two special keys

You are advised to study skip list from your text books. The idea of randomness is new to us. We will study in the next lecture, how easy and useful becomes the skip list data structure after employing randomness.

Data Structures

Lecture No. 40

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 10

10.4.2

Summary

- Skip List
- Skip List Search
- Insertion in Skip List
- Deletion from Skip List

In the previous lecture, we had started the discussion on the concept of the skip lists. We came across a number of definitions and saw how the use of additional pointers was effective in the list structures. It was evident from the discussion that a programmer prefers to keep the data in the linked list sorted. If the data is not sorted, we cannot use binary search algorithm. And the insert, find and remove methods are proportional to *n*. But in this case, we want to use the binary search on linked list. For this purpose, skip list can be used. In the skip list, there is no condition of the upper limit of the array.

Skip List

A skip list for a set *S* of distinct (key, element) items is a series of lists S_0, S_1 , … , S_h such that

- Each list *Si* contains the special keys +∞ and -∞
- List S_0 contains the keys of *S* in non-decreasing order
- Each list is a subsequence of the previous one, i.e.,
$$S_0 \supseteq S_1 \supseteq \ \ldots \supseteq S_h$$
- List S_h contains only the two special keys

Now let's see an example for the skip list. First of all, we have S0 i.e. a linked list. We did not show the arrows in the list in the figure. The first and last nodes of S0 contain -∞ and +∞ respectively. In the computer, we can put these values by –max(int) and max(int). The values can also be used about which we are sure that these will not be in the data items. However, for the sake of discussion to show these values, the -∞ and +∞ are the best notations. We can see that -∞ is less than any value of data item while +∞ is greater than any value of data item. If we insert any value much ever, it is large the +∞ will be greater than it. Moreover, we see that the numbers in S0 are in the non- decreasing order. This S0 is the first list in which all keys are present.
Now we will take some nodes from this list and link them. That will be not every other node or every fourth node. It may happen this way. However, we will try that the node should not be every other or fourth node. It will be a random selection. Now we see S1 i.e. a subset of S0. In S1 we selected the nodes 23, 31, 34 and 64. We have

chosen these nodes randomly with out any order or preference. Now from this S1, we make S2 by selecting some elements of S1. Here we select only one node i.e. 31 for the list S2. The additional pointer, here, has to move from -∞ to 31 and from 31 to +∞. Now the next list i.e. S3 will be subset of S2. As there is only one node in S2, so in S3, there will only the special keys. In these lists, we use pointers to link all the nodes in S0. Then with additional pointers, we linked these nodes additionally in the other lists. Unlike 2i, there is not every other node in S1 or every fourth node in S2. The following figure represents these lists.

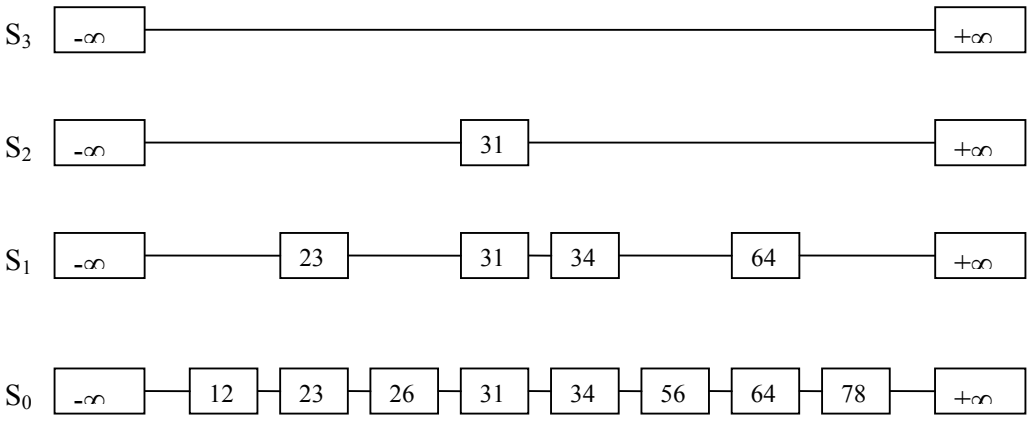


Figure 40.1:

Now we have the list i.e. from S0 to S3. Actually, these list are made during insert operation. We will see the insert method later. Let's first talk about the search method.

Skip List Search

Suppose we have a skip list structure available. Let's see what is its benefit. We started our discussion from the point that we want to keep linked list structure but do not want to search n elements for finding an item. We want to implement the algorithm like binary search on the linked list.

Now a skip list with additional pointers is available. Let's see how search will work with these additional pointers. Suppose we want to search an item x. Then search operation can be described as under.

We search for a key x in the following fashion:

- We start at the first position of the top list
- At the current position p, we compare x with $y \leftarrow key(after(p))$
- $x = y$: we return $element(after(p))$
- $x > y$: we "scan forward"
- $x < y$: we "drop down"
- If we try to drop down past the bottom list, we return *NO_SUCH_KEY*

To search a key *x,* we start at the first position of the top list. For example, we are discussing the top list is S3. We note the current position with *p*. We get the key in the list after the current list (i.e. in which we are currently looking for key) by the *key (after(p))* function. We get this key as *y*. Then we compare the key to be searched for i.e. *x* with this *y*. If *x* is equal to *y* then it means *y* is the element that we are searching for so we return *element(after(p))*. If *x* is greater than *y,* we scan forward and look at the next node. If *x* is less than *y,* we drop down and look in the down lists. No if we drop down and past the bottom list, it means that the element (item) is not there and we return *NO_SUCH_KEY*.

To see the working of this search strategy let's apply it on the skip list, already discussed and shown in the figure 40.1. This figure shows four lists. Remember that these four lists are actually in the one skip list and are made by the additional pointers in the same skip list. There is not such situation thatS1 is developed by extracting the data from S0 and S1 duplicates this data. Actually every node exists once and is pointed by additional pointers. For example, the node 23 exists once but has two next pointers. One is pointing to 26 while the other pointing to 31. In the same way, there are three pointers in node 31, two are to the node 34 and the third is toward the +∞.

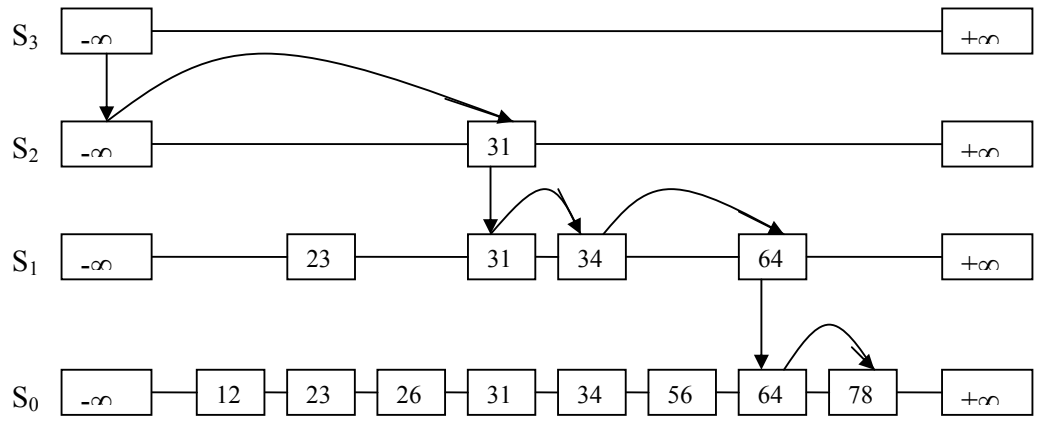


Figure 40.2: Search for 78

Suppose we want to search 78. We start from the first node of the top list i.e. S3. The 78 will be the current node and we denote it with *p*. Now we look at the value in the node after *p*. In the figure, it is +∞. Now as the +∞ is greater than 78, we drop down to S2. Note in the figure that we drop down vertically. We don't go to the first element p in the down list. This process of going down will be discussed later. Now we drop from S3 to S2. This is our current pointer. Now we look for the value in the next node to it. We see that this value is 31. Now 31 is less than 78, so we will do scan forward. The next node is +∞ that is obviously greater than 78. So we drop from here and go to the list S1. In this list, the current position is 34. We compare 78 with this node. As 34 is less than 78, we scan forward in the list. The next node in the list is 64 that is also less than 78. So we look at the next node and note that the next node is +∞ that is greater than 78. Due to this we drop down to list S0. Here we look at the next node of 64 and find that this is 78. Thus at last we reach at the node that we are

searching for. Now if we look at the arrows that are actually the steps to find out the value 78 in the skip list. Then we come to know that these are much less than the links that we have to follow while starting from the -+∞ in the list S0. In S0 we have to traverse the nodes 12, 23, 26, 31, 34, 44, 56, and 64 to reach at 78. This traversal could be larger if there were more nodes before 31. But in our algorithm, we reach at node 31 in one step. Thus we see that the search in the skip list is faster than the search in a linear list. By the steps that we do in searching an element and see that this search is like the binary search scheme. And we will see that this skip list search is also log2 N as in binary search tree. Though the skip list is not a tree, yet its find method works like the binary search in trees. As we know that find method is also involved in the remove method, so remove method also becomes fast as the find method is fast.

Insertion in Skip List

When we are going to insert (add) an item (*x,0*) into a skip list, we use a randomized algorithm. Note that here we are sending the item in a pair. This is due to the fact that we keep the data and the key separately to apply the find and remove methods on table easily. In this pair, *x* is the key, also present in the record. The *0* denotes the data (i.e. whole record). Now the randomized algorithm for insertion of this value is described below.

The first step of the algorithm is that

- We repeatedly toss a coin until we get tails, and we denote with $\boldsymbol{i}$ the number of times the coin came up heads.

This first step describes that we toss a coin while knowing a 50 percent probability for both head and tail. We keep a counter denoted with *i* to count the heads that come up. Now if the head comes up, *i* becomes 1. We again toss the coin if again the head comes up we add 1 to the counter i. We continue this counting until the tail comes up. When tail comes up, we stop tossing and note the value of *i*.
After this, the second step of algorithm comes, stating that

- If $\boldsymbol{i \geq h}$, we add to the skip list new lists $\boldsymbol{S_h}+1, \ldots, \boldsymbol{S_i}+1$, each containing only the two special keys

Here we compare *i* (that is the count of heads came up) with *h* (that is the number of list) if *i* is greater than or equal to *h* then we add new lists to the skip list. These new lists are $\boldsymbol{S_h}+1, \ldots, \boldsymbol{S_i}+1$. Suppose if *i* is 8 and *h* is 4 , we add additional lists S5, S6, S7, S8 and S9. These lists initially will contain the only two special keys that means -∝ and +∝. The next steps are:

- We search for $\boldsymbol{x}$ in the skip list and find the positions $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p_i}$ of the items with largest key less than $\boldsymbol{x}$ in each list $\boldsymbol{S}_0, \boldsymbol{S}_1, \ldots, \boldsymbol{S_i}$
- For $\boldsymbol{j} \leftarrow 0, \ldots, \boldsymbol{i}$, we insert item $(\boldsymbol{x}, \boldsymbol{o})$ into list $\boldsymbol{S_j}$ after position $\boldsymbol{p_j}$

Now we will start from the left most node of the top list i.e. Si and will find the position for the new data item. We find this position as we want to keep the data in the list sorted.
Let's see this insert position with the help of figures. Suppose we have a skip list ,

shown in the figure below. Here are only three items 10, 23 and 36 in it. There are also additional layers S1 and S2. These were made when 10, 23 and 36 were inserted.

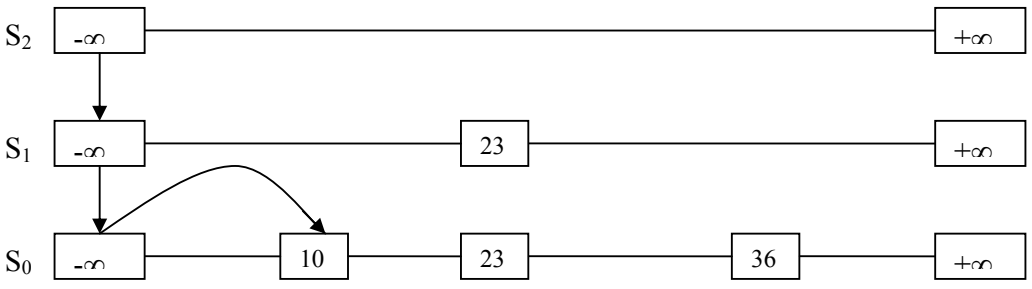


Figure 40.3:

Now we proceed with this list and insert a value in it. The value that we are going to insert is 15. We have tossed the coin and figured out that the value of i is 2. As it is randomized algorithm so in this case, the value of i has become 2. The value of I is the count number of heads before the tail comes up by tossing the coin. From the above figure, we see that the value of h is 2. As h and i are equal, we are not adding S3, S4 and S5 to the list. Rather, we will apply the search algorithm. We start from the left most node of the top list. We call this position p2. We label these positions for their identification. We see that the item being added i.e.15 is less than the +∝, so we drop down to list S1. We denote this step with p1. In this list, the next value is 23 that is also greater than 15. So we again drop down and come in the list S0. Our current position is still the first node in the list as we did not have any scan forward. Now the next node is 10 that is less than 15. So we skip forward. We note this skip forward with p0. Now after p0 the value in next node is 23 that is greater than 15. As we are in the bottom list, there is no more drop down. So here is the position of the new node. This position of new node is after the node 10 and before the node 23. We have labeled the positions po, p1 and p2 to reach there. Now we add the value 15 additionally to the list that we have traversed to find the positions and labeled them to remember. After this we add a list S3 that contains only two keys that are -∝ and +∝, as we have to keep such a list at the top of the skip list. The following figure shows this insertion process.

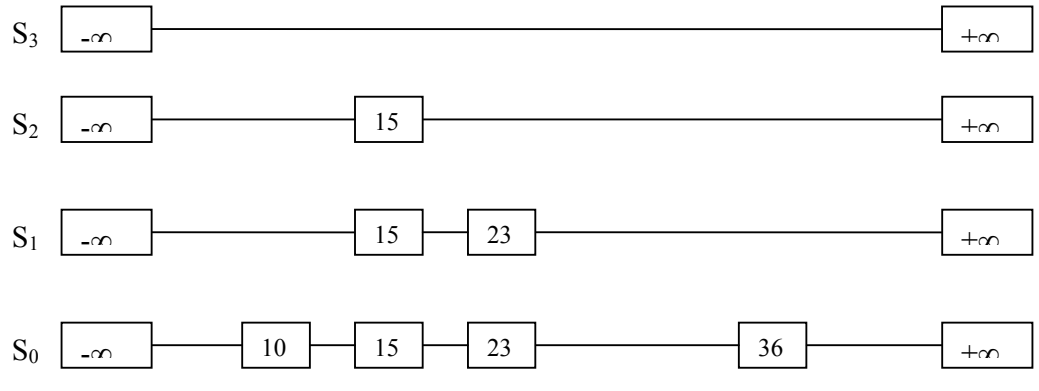


Figure 40.4: Insertion of 15

Here we see that the new inserted node is in three lists. The value that we insert must will be in S0 as it is the data list in which all the data will reside. However, it is also present in the lists that we have traversed to find its position. The value of i in the insertion method is randomly calculated by tossing the coin. How can we toss a coin in computers? There is routine library available in C and C++ that generates a random number. We can give it a range of numbers to generate a number in between them. We can ask it to give us only 0 and 1 and assign 1 to head and 0 to tail respectively. Thus the count number of 1's will give us the number of heads that come up. We can also use some other range like we can say that if the random number is less than some fixed value (whatever we fixed) that it means head otherwise it will mean tail. The algorithms that use random numbers are generally known as randomized algorithms.
So

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution
- It contains statements of the type

```
b ← random()
if  b <= 0.5    // head
        do A …
else            // tail
        do  B …
```

- Its running time depends on the outcome of the coin tosses, i.e, head or tail

In the randomized algorithm, we take a number from the random() function between 0 and 1. This number is up to one decimal place. However, we can keep it up to nay decimal places. As stated above ,after getting a number we check its value. If this is less than or equal to 0.5, we consider it as head and execute the process A. In our algorithm, we increase the value of i by 1. However, if value is greater than 0.5, we consider it as tail and do the process B. In our algorithm, the process B takes place when we note the value of I and stop the tossing. We do this process of tossing in a while loop. The while condition comes false when the tail (i.e. number greater than 0.5) comes. We cannot predict how many times this loop will execute as it depends upon the outcome of the toss. It is also a random number. There may be only one or a large number of heads before the tail comes up.

Deletion from Skip List

In the remove method, we find the item to be removed with the find item and remove it from the list. In the lists where ever this item has links, we bypass them. Thus, the procedure is quite easy. Now let's talk about this method.

To remove an item with key ***x*** from a skip list, we proceed as follows:

We search for ***x*** in the skip list and find the positions ***p***0, ***p***1 , …, ***pi*** of the items with key ***x***, where position ***pj*** is in list ***Sj.*** This means that we look for the links of the item to be removed . We know that an item in the list has necessarily link in the list S0. Moreover it may have links in other lists up to Si or say Sj. After this, we remove positions ***p***0, ***p***1 , …, ***pi*** from the lists ***S***0, ***S***1, … , ***Si.*** We remove all the lists except the list containing only the two special keys.

Let's consider the skip list shown in the figure below.

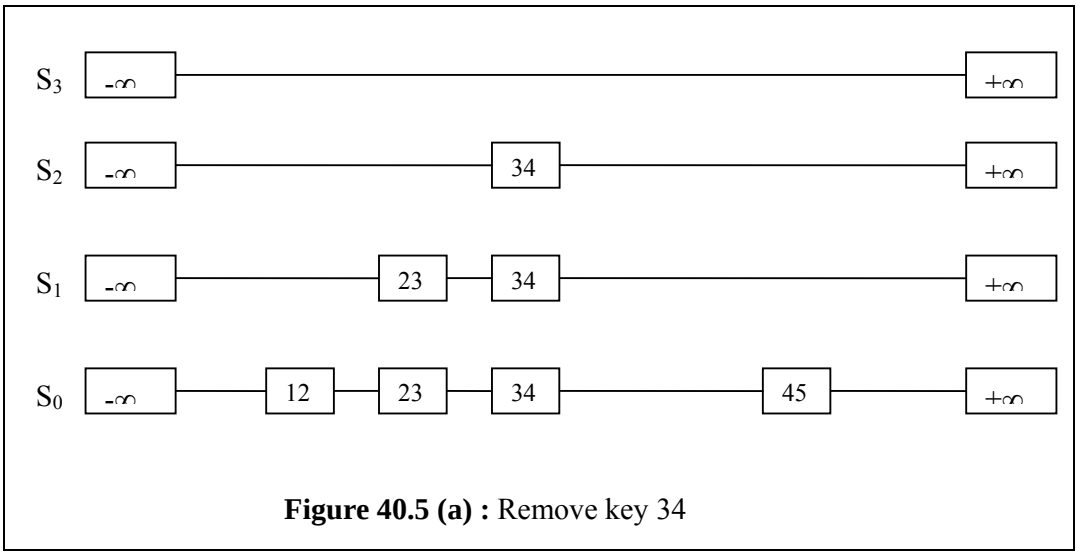


Figure 40.5 (a) : Remove key 34

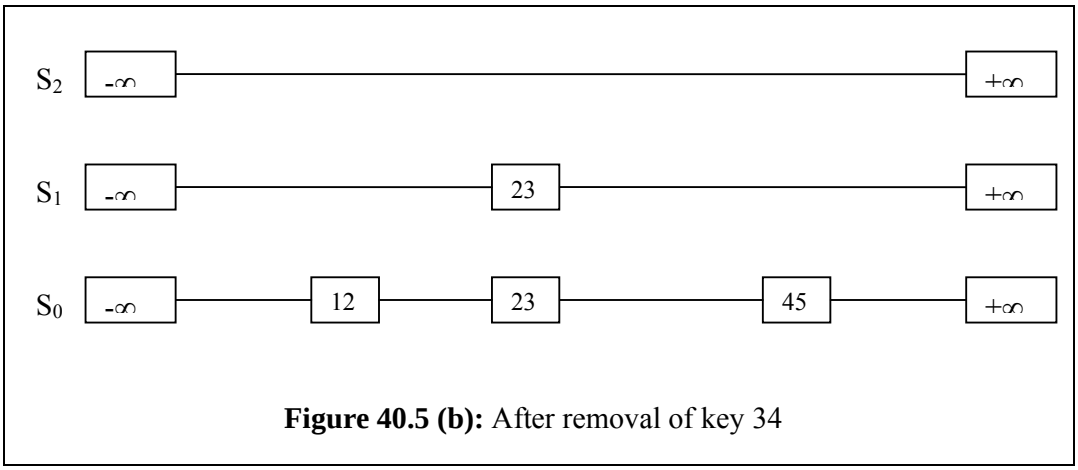


Figure 40.5 (b): After removal of key 34

Suppose we want to remove the node 34 from the list. When this node 34 was inserted, it came not only in list S0 but there were its additional links o in S1 and S2 lists respectively. The list S3 has only the two special keys. Now for finding 34, we

start from the top list. There is +∝ as the next node, so we drop down to the list S2 as +∝ is greater than 34. In S2, we are at node 34. Now we are at the point to remove 34. From here, we go to the remaining lists and reach list S0. We also label the links being traversed. It is evident that we have labeled the links as p2, p1 and p0. Now we remove the node 34 and change the pointers in the lists. We see that in S3, this was the single node. After removing this node, there is only one link that is from -∝ to +∝. The list S3 already has link from -∝ to +∝. Instead of keeping these two i.e. S2 and S3, we keep only S2. Thus we have removed the node 34 from the skip list. We see that the remove method is simple. We don't have randomness and need not tossing. The tossing and random number was only in the case of insertion. In the remove method, we have to change some pointers.

Thus in this data structure, we need not to go for rotations and balancing like AVL tree. In this data structure- Skip-list, we get rid of the limitation of arrays. This way, the search gets very fast. Moreover, we have sorted data and can get it in a sorted form.

Data Structures

Lecture No. 41

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 10
10.4.2

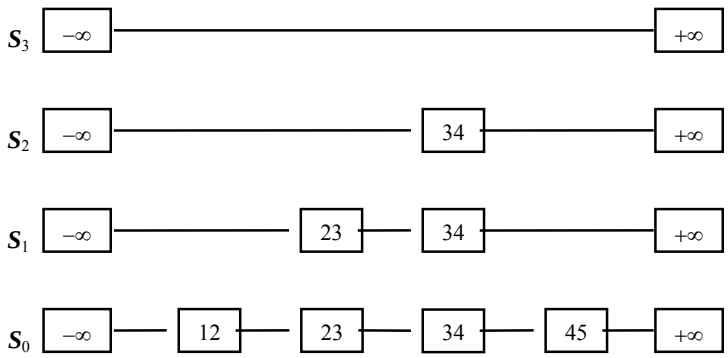
Summary

- Review
- Quad Node
- Performance of Skip Lists
- AVL Tree
- Hashing
- Examples of Hashing

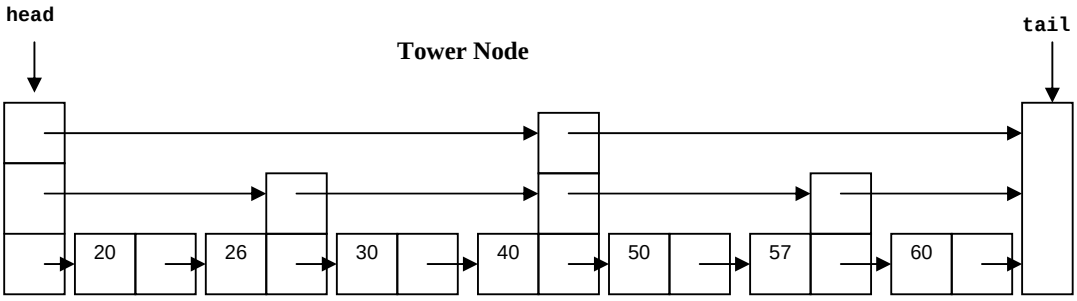
Review

In the previous lecture, we studied three methods of skip list i.e. *insert*, *find* and *remove* and had their pictorial view. Working of these methods was also discussed. With the help of sketches, you must have some idea about the implementation of the extra pointer in the skip list.

Let's discuss its implementation. The skip list is as under:



We have some nodes in this skip list. The data is present at 0, 1st and 2nd levels. The actual values are 12, 23, 34 and 45. The node 34 is present in three nodes. It is not necessary that we want to do the same in implementation. We need a structure with next pointers. Should we copy the data in the same way or not? Let's have a look at the previous example:



Here, the data is 20, 26, 30, 40, 50, 57, 60. At the lowest level, we have a link list. A view of the node 26, node 40 and node 57 reveals that there is an extra next 'pointer'. The head pointer is pointing to a node from where three pointers are pointing at different nodes.

We have seen the implementation of link list. At the time of implementation, there is a data field and a *next* pointer in it. In case of doubly link list, we have a *previous* pointer too. If we add an extra pointer in the node, the above structure can be obtained. It is not necessary that every node contains maximum pointers. For example, in the case of node 26 and node 57, there are two *next* pointers and the node 40 has three next pointers. We will name this node as 'TowerNode'.

TowerNode will have an array of *next* pointers. With the help of this array of pointers, a node can have multiple pointers. Actual number of *next* pointers will be decided by the random procedure. We also need to define *MAXLEVEL* as an upper limit on number of levels in a node. Now we will see when this node is created. A node is created at a time of calling the *insert* method to insert some data in the list. At that occasion, a programmer flips the coin till the time he gets a tail. The number of heads represents the levels. Suppose we want to insert some data and there are heads for six times. Now you know how much next pointers are needed to insert which data. Now we will create a *listNode* from the *TowerNode* factory. We will ask the factory to allocate the place for six *next* pointers dynamically. Keep in mind that the next is an array for which we will allocate the memory dynamically. This is done just due to the fact that we may require different number of *next* pointers at different times. So at the time of creation, the factory will take care of this thing. When we get this node from the factory, it has six next pointers. We will insert the new data item in it. Then in a loop, we will point all these next pointers to next nodes. We have already studied it in the separate lecture on *insert* method.

If your random number generation is not truly so and it gives only the heads. In this case, we may have a very big number of heads and the Tower will be too big, leading to memory allocation problem. Therefore, there is need to impose some upper limit on it. For this purpose, we use *MAXLEVEL*. It is the upper limit for the number of *next* pointers. We may generate some error in our program if this upper limit is crossed.

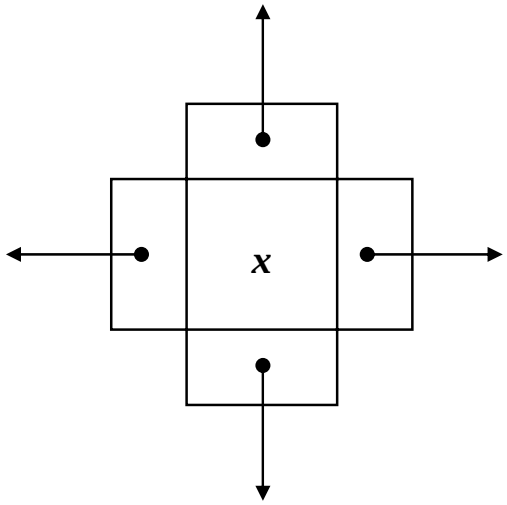
The *next* pointers of a node will point at their own level. Consider the above figure. Suppose we want to insert node 40 in it. Its 0 level pointer is pointing to node 50. The 2nd pointer is pointing to the node 57 while the third pointer pointing to tail. This is the case when we use *TowerNode*. This is one of the solutions of this problem.

Quad Node

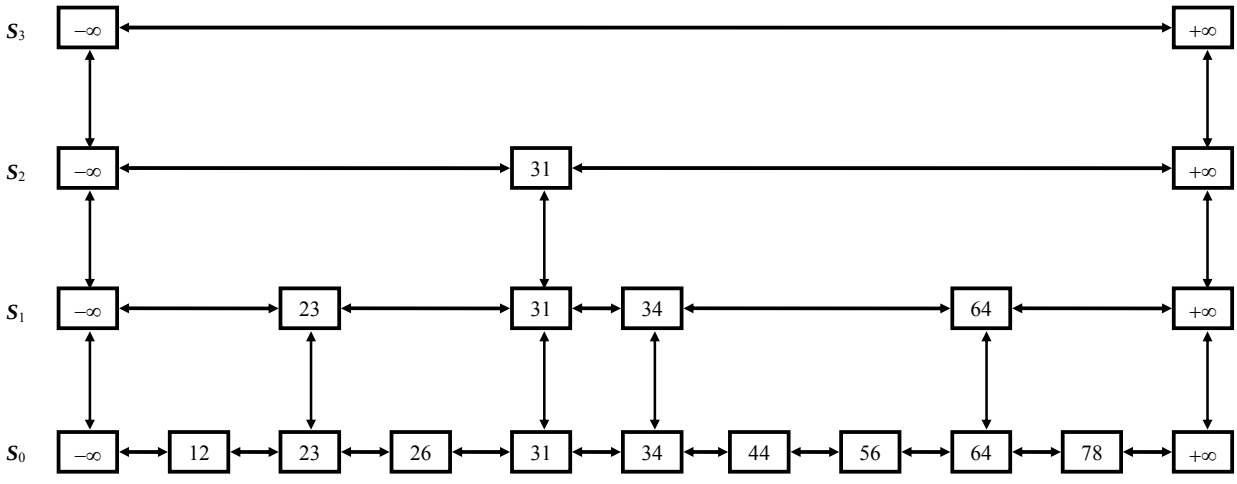
Let's review another method for this solution, called *Quad node*. In this method, we do not have the array of pointers. Rather, there are four *next* pointers. The following details can help us understand it properly.

A quad-node stores:

- item
- link to the node before
- link to the node after
- link to the node below
- link to the node above



This will require copying of the key (item) at different levels. We do not have an array of next pointers in it. So different ways are adopted to create a multilevel node of skip list. While requiring six levels, we will have to create six such nodes and copy the data item *x* in all of these nodes and insert these in link list structure. The following figure depicts it well.



You can see *next* and *previous* and *down* and *up* pointers here. In the bottom layer, the *down* pointer is nil. Similarly the right pointers of right column are nil. In the top

layer, the top pointers are nil. You can see that the values 23, 34, and 64 are copied two times and the value 31 is copied three times. What are the advantages of *quad node*? In *quad node*, we need not to allocate the array for next pointers. Every list node contains four pointers. The *quad node* factory will return a node having four pointers in it. It is our responsibility to link the nodes up, bottom, left and right pointers with other nodes. With the help of previous pointer, we can also move backward in the list. This may be called as doubly skip list.

Performance of Skip Lists

Let's analyze this data structure and see how much time is required for search and deletion process. The analysis is probability-based and needs lot of time. We will do this in some other course. Let's discuss about the performance of the skip list regarding insert, find and remove methods.

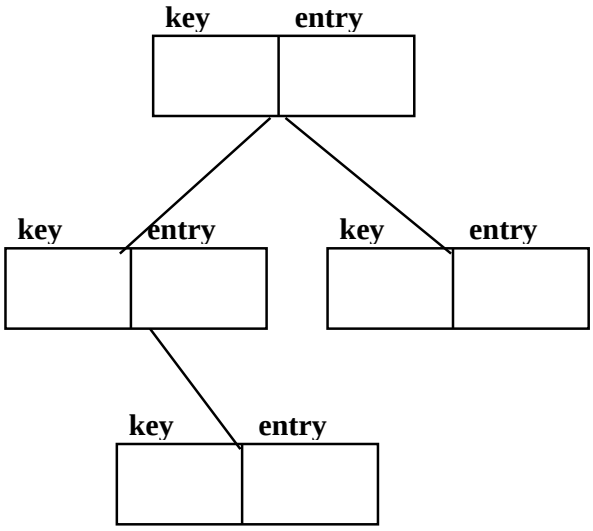
In a skip list, with *n* items the expected space used is proportional to *n*. When we create a skip list, some memory is needed for its items. We also need memory to create a link list at lowest level as well as in other levels. This memory space is proportional to *n* i.e. number of items. For *n* items, we need *n* memory locations. The items may be of integer data type. If we have 100 items, there will be need of 100 memory locations. Then we need space for *next* pointers that necessitates the availability of *2n* memory locations. We have different layers in our structure. We do not make every node as *towerNode* neither we have a formula for this. We randomly select the *towerNode* and their height. The *next* pointers can be up to *maxLevel* but normally these will be few. We do not require pointers at each level. We do not need *20n* or *30n* memory locations. If we combine all these the value will be *15n* to *20n*. Therefore the proportionality constant will be around 15 or 20 but it cant be n^2 or n^3. If this is the case then to store 100 items we do need 100^2 or 100^3 memory locations. There are some algorithms in which we require space in square or cubic times. This is the space requirement and it is sufficient in terms of space requirements. It does not demand too much memory.

Let's see the performance of its methods. The expected search, insertion and deletion time is proportional to *log n*. It looks like binary tree or binary search tree (BST). This structure is proposed while keeping in mind, the binary search tree. You have witnessed that if we have extra nodes, search process can be carried out very fast. We can prove it with the probabilistic analyses of mathematics. We will not do it in this course. This information is available in books and on the internet. All the searches, insertions and deletions are proportional to *log n*. If we have 100,000 nodes, its *log n* will be around 20. We need 20 steps to insert or search or remove an element. In case of insert, we need to search that this element already exists or not. If we allow duplicate entries then a new entry would be inserted after the previous one. In case of delete too, we need to search for the entry before making any deletion. In case of binary search tree, the insertion or deletion is proportional to *log n* when the tree is a balanced tree. This data structure is very efficient. Its implementation is also very simple. As you have already worked with link list, so it will be very easy for you to implement it.

AVL Tree

The insertion, deletion and searches will be performed on the basis of key. In the nodes, we have key and data together. Keep in mind the example of telephone

directory or employee directory. In the key, we have the name of the person and the entry contains the address, telephone number and the remaining information. In our AVL tree, we will store this data in the nodes. Though, the search will be on the key, yet as we already noticed that the insert is proportional to *log n*. Being a balanced tree, it will not become degenerated balance tree. The objective of AVL tree is to make the binary trees balanced. Therefore the find will be *log n*. Similarly the time required for the removal of the node is proportional to *log n*.



We have discussed all the five implementations. In some implementations, time required is proportional to some constant time. In case of a sorted list, we have to search before the insertion. However for an unsorted list, a programmer will insert the item in the start. Similarly we have seen the data structure where insertions, deletions and searches are proportional to *n*. In link list, insertions and deletions are proportional to *n* whereas search is *log n*. It seems that *log n* is the lower limit and we cannot reduce this number more.

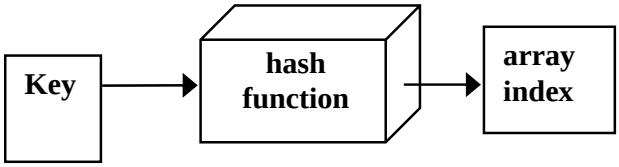
Is it true that we cannot do better than *log n* in case of table? Think about it and send your proposals. So far we have find, remove and insert where time varies between constant and *log n*. It would be nice to have all the three as constant time operations. We do not want to traverse the tree or go into long loops. So it is advisable to find the element in first step. If we want to insert or delete, it should be done in one step. How can we do that? The answer is Hashing.

Hashing

The hashing is an algorithmic procedure and a methodology. It is not a new data structure. It is a way to use the existing data structure. The methods- *find*, *insert* and *remove* of table will get of constant time. You will see that we will be able to do this in a single step. What is its advantage? If we need table data structure in some program, it can be used easily due to being very efficient. Moreover, its operations are of constant time. In the recent lectures, we were talking about the algorithms and procedures rather than data structure. Now we will discuss about the strategies and methodologies. Hashing is also a part of this.

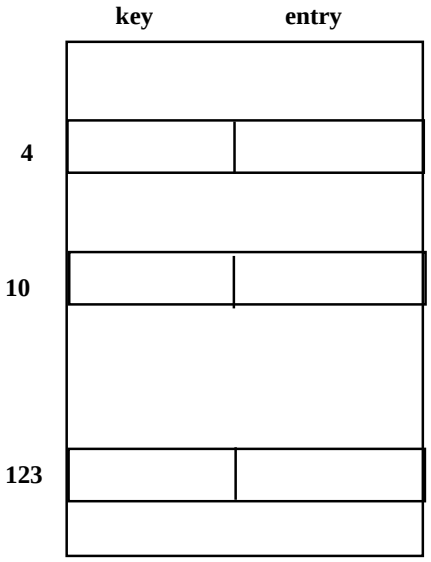
We will store the data in the array but *TableNodes* are not stored consecutively. We are storing the element's data in the *TableNodes*. You have seen the array implementation of the *Table* data structure. We have also seen how to make the data sorted. There will be no gap in the array positions whether we use the sorted or unsorted data. This means that there is some data at the 1st and 2nd position of array and then the third element is stored at the 6th position and 4th and 5th positions are empty. We have not done like this before. In case of link list, it is non-consecutive data structure with respect to memory.

In Hashing, we will internally use array. It may be static or dynamic. But we will not store data in consecutive locations. Their place of storage is calculated using the key and a *hash function.* Hash function is a new thing for you. See the diagram below:



We have a key that may be a name, or roll number or login name etc. We will pass this key to a hash function. This is a mathematical function that will return an array index. In other words, an integer number will be returned. This number will be in some range but not in a sequence.

Keys and entries are scattered throughout the array. Suppose we want to insert the data of our employee. The key is the name of the employee. We will pass this key to the hash function which will return an integer. We will use this number as array index. We will insert the data of the employee at that index.



The insert will calculate place of storage and insert in *TableNode*. When we get a new data item, its key will be generated with the help of hash function to get the array

index. Using this array index, we insert the data in the array. This is certainly a constant time operation. If our hash function is fast, the insert operation will also rapid. It will take only one step to perform this.

Next we have *find* method. It will calculate the place of storage and retrieve the entry. We will get the key and pass it to the hash function and obtain the array index. We get the data element from that array position. If data is not present at that array position, it means data is not found. We do not need to find the data at some other place. In case of binary search tree, we traverse the tree to find the element. Similarly in list structure we continue our search. Therefore find is also a constant time operation with Hashing.

Finally, we have *remove* method. It will calculate the place of storage and set it to null. That means it will pass the key to the hash function and get the array index. Using this array index, it will remove the element.

Examples of Hashing

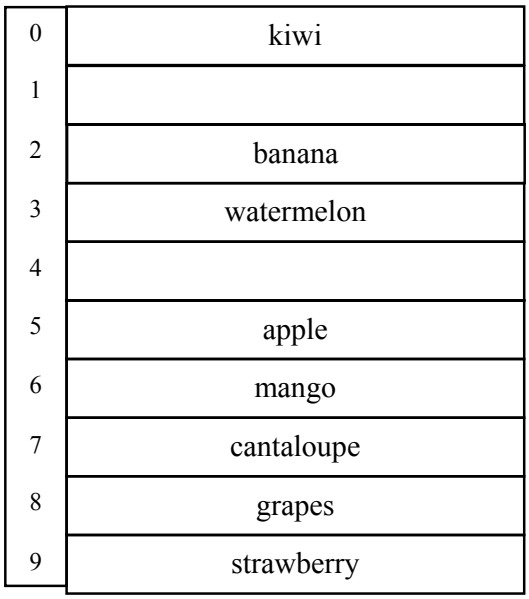
Let's see some examples of hashing and hash functions. With the help of these examples you will easily understand the working of *find*, *insert* and *remove* methods.

Suppose we want to store some data. We have a list of some fruits. The names of fruits are in string. The key is the name of the fruit. We will pass it to the hash function to get the hash key.

Suppose our hash function gave us the following values:

```
HashCode ("apple")        =  5
hashCode ("watermelon")   =  3
hashCode ("grapes")       =  8
hashCode ("cantaloupe")   =  7
hashCode ("kiwi")         =  0
hashCode ("strawberry")   =  9
hashCode ("mango")        =  6
hashCode ("banana")       =  2
```

Our hash function name is *hashCode*. We pass it to the string "apple". Resultantly, it returns a number 5. In case of "watermelon" we get the number 3. In case of "grapes" there is number 8 and so on. Neither we are sending the names of the fruits in some order to the function, nor is function returning the numbers in some order. It seems that some random numbers are returned. We have an array to store these strings. Our array will look like as:



We store the data depending on the indices got from the *hashCode*. The array size is 10. In case of apple, we get the index 5 from *hashCode* so "apple" is stored at array index 5. As we are dealing with strings, so the array will be an array of strings. The "watermelon" is at position 3 and so on every element is at its position. This array will be in the private part of our data structure and the user will not know about it. If our array is *table* then it will look like as under:

```
table[5]  =  "apple"
table[3]  =  "watermelon"
table[8]  =  "grapes"
table[7]  =  "cantaloupe"
table[0]  =  "kiwi"
table[9]  =  "strawberry"
table[6]  =  "mango"
table[2]  =  "banana"
```

We will store our data in the Table array using the string copy. The user is storing the data using the names of the fruits and wants to retrieve or delete the data using the names of fruits. We have used the array for storage purposes but did not store the data consecutively. We store the data using the hash function which provides us the array index. You can note that there are gaps in the array positions.

Similarly we will retrieve the data using the names of fruit and pass it to the *hashCode* to get the index. Then we will retrieve the data at that position. Consider the table array, it seems that we are using the names of fruits as indices.

```
table["apple"]
table["watermelon"]
table["grapes"]
table["cantaloupe"]
table["kiwi"]
```

```
table["strawberry"]
table["mango"]
table["banana"]
```

We are using the array as table [“apple”], table [“watermelon”] and so on. We are not using the numbers as indices here. Internally we are using the integer indices using the *hashCode*. Here we have used the fruit names as indices of the array. This is known as associative array. Now this is the internal details that we are thinking it as associative array or number array.

Let’s discuss about the *hashCode*. How does it work internally? We pass it to strings that may be persons name or name of fruits. How does it generate numbers from these? If the keys are strings, the hash function is some function of the characters in the strings. One possibility is to simply add the ASCII values of the characters. Suppose the mathematical notation of hash function is *h*. It adds all the ASCII values of the string characters. The characters in a string are from 0 to *length - 1*. Then it will take mod of this result with the size of the table. The size of the table is actually the size of our internal array. This formula can be written mathematically as:

$$h(str) = \left(\sum_{i=0}^{length-1} str[i] \right) \% TableSize$$

$$Example: h(ABC) = (65 + 66 + 67) \quad TableSize$$

Suppose we use the string “ABC” and try to find its hash value. The ASCII values of A, B and C are 65, 66 and 67 respectively. Suppose the tableSize is 55. We will add these three numbers and take mod with 55. The result (3.6) will be the hash value. To represent character data in the computer ASCII codes are used. For each character we have a different bit pattern. To memorize this, we use its base 10 values. All the characters on the keyboard like $, %, ‘ have ASCII values. You can find the ASCII table in you book or on the internet.

Let’s see the C++ code of *hashCode* function.

```
int hashCode( char* s )
{
        int i, sum;
        sum = 0;
        for(i=0; i < strlen(s); i++ )
                sum = sum + s[i]; // ascii value
        return sum % TABLESIZE;
}
```

The return type of *hashCode* function is an integer and takes a pointer to character. It declares local variable *i* and *sum*, then initializes the *sum* with zero. We use the *strlen* function to calculate the length of the string. We run a loop from 0 to *length – 1*. In the loop, we start adding the ASCII values of the characters. In C++, characters are

stored as ASCII values. So we directly add *s[i]*. Then in the end, we take mod of sum with *TABLESIZE*. The variable *TABLESIZE* is a constant representing the size of the table.

This is the one of the ways to implement the hash function. This is not the only way of implementing hash function. The hash function is a very important topic. Experts have researched a lot on hash functions. There may be other implementations of hash functions.

Another possibility is to convert the string into some number in some arbitrary base *b* (*b* also might be a prime number). The formula is as:

$$h(str) = \left(\sum_{i=0}^{length-1} str[i] \times b^i \right) \% T$$

$$Example: h(ABC) = (65b^0 + 66b^1 + 67b^2) \% T$$

We are taking the ASCII value and multiply it by *b* to the power of *i*. Then we accumulate these numbers. In the end, we take the mod of this summation with *tableSize* to get the result. The *b* may be some number. For example, we can take *b* as a prime number and take 7 or 11 etc. Let's take the value of *b* as 7. If we want to get the hash value of ABC using this formula:

H(ABC) = (65 * 7 ^0 + 66 * 7^1 + 67 * 7^2) mod 55 = 45

We are free to implement the hash function. The only condition is that it accepts a string and returns an integer.

If the keys are integers, *key%T* is generally a good hash function, unless the data has some undesirable features. For example, if *T = 10* and all keys end in zeros, then *key%T = 0* for all keys. Suppose we have employee ID i.e. an integer. The employee ID may be in hundreds of thousand. Here the table size is 10. In this case, we will take mod of the employee ID with the table size to get the hash value. Here, the entire employee IDs end in zero. What will be the remainder when we divide this number with 10? It will be 0 for all employees. So this hash function cannot work with this data. In general, to avoid situations like this, *T* should be a prime number.

Data Structures

Lecture No. 42

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 5
5.1, 5.2, 5.4.1, 5.4.2

Summary

- Collision
- Linear Probing

In the previous lecture, we were discussing about the hash functions. The hash algorithm depends on the hash function. The hash function generates the array index to enable us to insert data in the table array. We have seen two examples of hash functions in the previous lecture. Both the functions use the ASCII values of characters to generate the index. Here the question arises how can we implement the hash function in case of having integer data? We may have employee ID, user ID or student ID as integers. Here we may take mod with some number or table size and the result is used as an array index.

If the keys are integers then *key%T* is generally a good hash function unless the data has some undesirable features. If we want to store the employee record, user record or student record in the table, this can be done through hash function. We take the mod of the value with the *T*. The value of *T* may be 10, 15 or 100 depending on the requirements. There may be some problem. For example, if *T = 10* and all keys end in zeros, then *key%T = 0* for all keys. The hash function gives 0 for all the keys, used as array index. Now it is a problem. We cannot store our values as all the records have same index i.e. 0. In general, to avoid such situations, *T* should be a prime number. Internally, we have to store the data in the array and there is complete freedom to use this array by taking the size of our own choice.

We have to store some data in the array. As the array is private, we will decide about its size on our own. We will take the size of the array in prime numbers. To store 100 records, we will take prime number near 100. We will select this prime number as *MAXTABLESIZE*. Then we will use this number in our hash function. This will help resolve the problem arising due to situation where all keys end with 0. Using the prime number, the values from the hash function will not be 0 for all the keys.

With the help of prime number, we cannot solve this problem completely. Similarly, it cannot be made sure that the values from the hash function are unique for all the keys. Sometimes, we may have same index for two different keys. This phenomenon is known as collision i.e. the hash values are same of two different keys. How can we solve this collision problem?

Collision

Collision takes place when two or more keys (data items) produce the same index.

Let's see the previous example of storing the names of fruits. Suppose our hash function gives us the following values:

```
hash("apple")       =  5
hash("watermelon")  =  3
hash("grapes")      =  8
hash("cantaloupe")  =  7
hash("kiwi")        =  0
hash("strawberry")  =  9
hash("mango")       =  6
hash("banana")      =  2
```

We store these data items in the respective index in the array. In the above example, the index given by the hash function does not collide with any other entry. Suppose we want to add another fruit "honeydew" in it. When "honeydew" is passed to the hash function, we get the value 6 i.e.

hash("honeydew") = 6

This is not the responsibility of the hash function to see the data in the array before generating the index. Hash function is generally a mathematical formula that takes the keys and returns a number. It is responsibility of the caller to find its solution. We have already "mango" at position 6. The user of our ADT calls the insert function giving the value "honeydew". We call the hash function to find out its index that comes out to be 6. Now the problem is this that position 6 is already occupied. What should we do to avoid it?

There are a lot of solutions of this problem. These solutions can be divided into two main categories. One type of solutions is the changing of the hash function. Even with the introduction of a new function, it is not guaranteed that there will be no collision with future data. Second option is that we live with the collision and do something to resolve it.

The definition of collision is:

"When two values hash to the same array location, this is called a *collision"*

We cannot say that the usage of this hash function will not result in collision

especially when the data is changing. Collisions are normally treated as a phenomenon of "first come, first served", the first value that hashes to the location gets it. We have to find something to do with the second and subsequent values that hash to the same location.

As we have seen above in the example that at position 6, there is the data item *mango* while another item *honeydew* is trying to acquire the same position. First come first served means that the *mango* will remain at its position and we will do something for *honeydew*. We will see three solutions for this problem:

Solution #1: Search for an empty location

- Can stop searching when we find the value or an empty location.
- Search must be wrap-around at the end.

Let's apply this solution on the above example. We have *mango* at position 6. Now the position for *honeydew* is also 6. We find another empty location in the array. Keep in mind that in hashing, we do not store data at consecutive positions. Rather, data is scattered. We know that there are some empty locations in the array. We do not want to refuse the user that we cannot store the data due to the occupation of the position by some other item. Therefore, we will store the data item at the empty location. We will see an example in a short while.

Solution #2: Use a second hash function

- ...and a third, and a fourth, and a fifth, ...

We have a primary hash function, we pass it to the string *honeydew*. It returns 6. As the position 6 is occupied, so we call another hash function that is implemented in a different way. This hash function will return an integer. If this location is empty, we will store our data here and the problem is solved. But if the array location at the index number returned by this function is also occupied. Then we will call the third hash function. There is a possibility of collision so we will have three, four, five or more hash functions. When there is a collision, we call these hash functions one by one till the time, there is an index with empty location. We will store the data at this empty location.

Solution #3: Use the array location as the header of a linked list of values that hash to this location

In this solution, we will not store the data directly in the array. Our array will be an array of pointers to *TableNode*. We will create a list node and store the data in it. The array will have a pointer to the node. If we want to store some data at location 6, we will store a pointer at location 6 that points to the node containing the data. When we have another data at the position 6, we create a list node and attach it with the previous node. There is a possibility of having a link list from each location of the table. At least, it will have one node.

There are some more strategies for the resolution of this problem. But we will study only the above-mentioned three solutions. Now we will see how we can implement the methods of *Table* and *Dictionary ADT* using the hashing function. Firstly, we will see how the *insert*, *find* and *remove* methods will work with the first solution.

Linear Probing

The first solution is known as open addressing. When there is a collision, we try to find some other place in our array. This approach of handling collisions is called *open addressing*; it is also known as *closed hashing*. The word open is used with addressing and the word closed is used with hashing. Be careful when naming these. More formally, cells at $h_0(x)$, $h_1(x)$, $h_2(x)$, … are tried in succession where

$$hi(x) = (hash(x) + f(i)) \bmod TableSize, \qquad \text{with } f(0) = 0.$$

Here *hash* is our hash function. If there is some collision, we add *f(i)* value to it before taking its mod with *TableSize*. The function, *f*, is the collision resolution strategy.

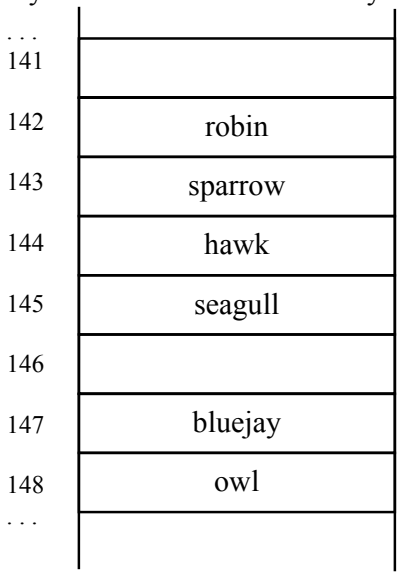
We use $f(i) = i$, i.e., *f* is a linear function of *i*. Thus

$$location(x) = (hash(x) + i) \bmod TableSize$$

The function *f* can be any function. So our first implementation of *f* is that whatever integer (i) we gave to it, it returns it back. That is f(i) is a linear function. We can implement function *f* as it returns the square of *i*. That will be quadratic function. Similarly, we can have cubic functions. We are free to implement it.

In the above example, we are given *x* for insertion. This can be string such as *honeydew*. We call the *hash(x)* that returns a number. We will see that this location is empty or not. In case of collision, we add *i* to this number. The value of *i* can be 0,1,2,3 etc. We will take the mod of this value with *TableSize* and try to store the data there. The collision resolution strategy is called *linear probing as* it scans the array sequentially (with wrap around) in search of an empty cell. Let's consider an example to understand how it works. Keep in mind that when there is a collision, some other location in the array is found. This is known as linear probing. Here we are trying to probe the array linearly that where is the empty location.

Let's see an example. Here we have names of birds being stored in the array. We have a larger array. We already have some data in the array that is as under:



This is a part of array. There is some data before and after these locations. As shown above, the locations 142, 143, 144, 145, 147 and 148 contains data. Suppose we want to add *seagull* to this hash table. With respect to ADT, someone is using this *Table* or *Dictionary* and calls the *insert* method of this *Table*. Now we have to use the *seagull* as key and have to store its associated data. We will call the *hashCode* function to get the array index and pass it the string "seagull" as:

hashCode("seagull") = 143

Now the location 143 is not empty and we have a bird, *sparrow* at this location. That is *table[143]* is not empty. The *seagull* is different than *sparrow* i.e. *table[143] != seagull*. If it already exists and we are not allowing duplicates, we can say that the data already exists. This is not the case here. We will check the next position, 143+1 (here i is 1) i.e. 144. At location 144, we have a bird *hawk*. So this position is also not empty and not equal to *seagull*. Here also we have different data. Now we check the next position 143+2 (here i is 2) i.e. 145. This location is empty so we store our data at this location. We have two collisions here at 143 and 144 positions. We have found the empty position in the array with single jumps. First we add 1, then 2 and so on.

Suppose we want to add some existing data. As collisions are happening here, so when we get the index value from hash function, the data may not be at that position. We have to follow this linear chain. Suppose you want to *add hawk* to this hash table. The table is as under:

. . .	
141	
142	robin
143	sparrow
144	hawk
145	seagull
146	
147	bluejay
148	owl
. . .	

We know that *hawk* is already in the table. Also *seagull* is added in the table. We will see that that data that we want to insert already exists or not. First of all, we call the hash function that is *hashCode* as:

hashCode("hawk") = 143

We use this value as the table index to insert this data in the table. The *table[143]* is not empty. Then we check the data at position 143. We have stored *sparrow* here so *table[143] != hawk*. At this point, we cannot decide whether the data item *hawk* exists in the table or not. As we are doing linear probing and there are chances that this data may exist at some other location. Therefore we will check it further taking any final decision. Now we check the next position i.e. 144. It is also not empty but the data here is *hawk* that is *table[144] == hawk*. We have two options here. One is that *hawk* is already in the table, so do nothing. The other option is that user might want to modify the data associated with *hawk*.

The size of our internal array is fixed and we store it in the constant *TableSize*. In linear probing, at the time of collisions, we add one to the index and check that location. If it is also not empty, we add 2 and check that position. Suppose we keep on incrementing the array index and reach at the end of the table. We were unable to find the space and reached the last location of the array. Now what does it mean? Is the table completely filled? There are chances that we started from the position 143. Suppose that the table size is 500. We were unable to find any empty location. Can we say that the array locations before 143 are also not available? There is a possibility that we find some empty spaces there. We will not stop at the end of the table and go to the start of this table array. We will start linear probing from the start and if we find some empty spaces, the data will be stored there. This process is called wrap around. There is a chance that in wrap around, we could not find space and come back to the 143 position. The solution of this problem is to get a larger array. We may want to create an array of size 1000 or some prime number larger than 500. Then copy all the data in this new array. This new data may be inserted with the help of linear probing. The other solution is that we may want to keep multiple arrays. In this case, when one array is filled we go to the second array.

Let's see the example of wrap around process. Suppose we want to add *cardinal* to this hash table. We will use the above table to insert this data. Also suppose that 148 is the last position of this table. We call the *hashCode* function as:

hashCode("cardinal") = 147

We get the array index as 147. This position is already filled in the table. The data at this position is *bluejay*. So we will add 1 to the index and check the 148 position. This position is also filled. We do not stop here and go back to the start of the array. We cannot increment the array index further. We will treat the table as circular; after 148 comes 0. Now we will check the position 0. If it is occupied, we will see the position 1 then 2 and so on. We have seen the *insert* method using the linear probing. Now we will see the *find* method.

Suppose we want to find *hawk* in this hash table. We call the *hashCode* as:

hashCode("hawk") = 143

The *table[143]* is not empty. However, the data at this position is not *hawk*. So with the linear probing procedure, we add 1 to the index and check the 144 position. This position is also not empty. We compare it with *hawk* that is true. We use the same procedure for looking things up in the table as generally done for inserting them.

Let's have a look on the delete procedure. If an item is placed in *array[hash(key)+4]*, the item just before it is deleted. How will probe determine that the "hole" does not indicate the item is not in the array? We may have three states for each location as:

- Occupied
- Empty (never used)
- Deleted (previously used)

Using the linear probe, we insert data where we get some empty space in the array. Firstly, we try to insert it at the index position given by the hash function. If it is occupied, we move to the next position and so on. In this way, we have a chain to follow. Now if an element of the chain is deleted, how can we know that it was filled previously. As we have seen in the above example that *seagull* collided twice before getting a space in the array. Despite having a hash value of 143, it was stored at 145. The three locations 143, 144 and 145 have different data values but are a part of a chain. These three names collide with each other with respect to hash function. Suppose we delete the 2nd name i.e. *hawk* due to some reason. Now there are two names, which collide. A space has been created due to the deletion of *hawk*. If we try to search *seagull* using the linear probing, there will be a hash value of 143. It is filled but has different data. We move to the next position that is empty. How can linear probing know that do not stop here because there was some data previously at this position, which has not been deleted. And there is some data after this position, which is part of this chain due to the collisions and that data is *seagull*.

To deal with such situations, we keep three states in our array. 1) It is empty (never used) and has no data in it. 2) It is filled with some legal data and is occupied and 3) it had some data which is now deleted and currently it is empty. We will use this state if we have to go to next position using the linear probing.

One problem with linear probing technique is the tendency to form "clusters". A cluster is a group of items not containing any open slots. The bigger a cluster gets, the more likely the new values will hash into the cluster, and make it even bigger. Clusters cause efficiency to degrade. Suppose we want to add another bird in our table. The hash function returns the index as 143. Now we have already three items that collided at 143. The array locations at 143, 144, 145 are already occupied. So we will insert this new data at 146 using the linear probing. The data is getting gathered instead of scattering because linear probing inserts the data in the next position. It seems as the normal use of the array in which we insert data in the array from first position then next position and so on. It may depend on our data or our hash function. This gathering of data is called clustering.

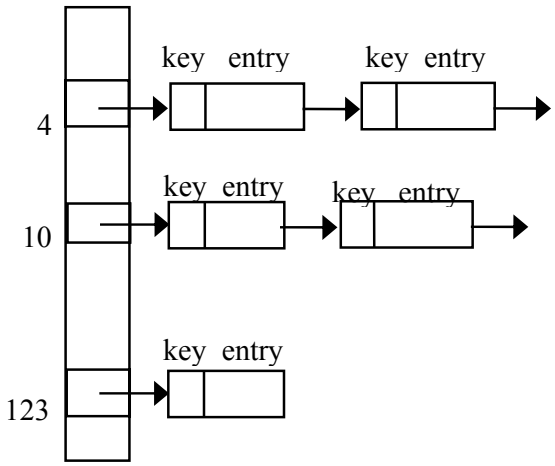
We were trying to store the data in the array in a constant time or in a single step. Similarly the *find* and *remove* methods should be of constant time. That attribute has now been lost. What should we do now? One of the solutions is quadratic probing.

Quadratic probing uses different formula:

- Use $F(i) = i^2$ (square of i) to resolve collisions
- If hash function resolves to H and a search in cell H is inconclusive, try $H + 1^2$, $H + 2^2$, $H + 3^2$, …

In the quadratic probing when a collision happens we try to find the empty location at index + $1^{2.}$ If it is filled then we add 2^2 and so on. Let's take the above example. We want to insert seagull. The hash function generates the value 143 and this location is already occupied. Now we will start probing. First of all, we add 1^2 in the 143 and have 144. It is also not empty. Now we will add 2^2 in the 143 and have 147. If it is also occupied, we will add 3^2 in 143 and we have 152. The data is now getting scattered. Unfortunately, there are some problems with quadratic probing also.

Let's discuss the third solution that we will use the link list for the collision resolution. Each table position is a linked list. When we are going to insert new data, we will insert the keys and entries anywhere in the list (front easiest). It is shown in the below diagram.



On the left side, we have vertical array that contains the pointers in it. When we insert the first item, we attach a list node. When we have collision, the new item is inserted in the start of the link list. Suppose an item is stored at position 4 and we have another data, requiring the position 4. So there is a collision and we will insert the data at the start of the list.

Let's compare the link list and open addressing.

- Advantages over open addressing:
  - Simpler insertion and removal
  - Array size is not a limitation
- Disadvantage
  - Memory overhead is large if entries are small.

The problem in linear probing is that when our array is full what we should do. This problem can be solved using the link list.

In the next lecture, we will continue our discussion on hashing. We will see an animation. Hashing is very important methodology and can be used in data structures besides *Table* and *Dictionary*.

Data Structures

Lecture No. 43

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 5, 7
5.4, 5.5, 5.6, 7.1

Summary

- Hashing Animation
- Applications of Hashing
- When Hashing is Suitable?
- Sorting
- Elementary Selection Algorithms
- Selection Sort

Hashing Animation

In the previous lecture, we discussed about collision strategies in hashing. We studied three solutions, Linear Probing, Quadratic Probing and Linked List chaining. Hashing is vast research field, which covers hash functions, storage and collision issues etc. At the moment, we will see hashing in implementation of table ADT. Operations of *insert*, *delete* and *find* are performed in constant time using this hashing strategy. Constant time means the time does not increase with the increase in data volume. However, if collisions start happening then the time does not remain constant. Especially if we see *linear probing*, we had to insert the data by sorting the array sequentially. Similar was the case with *quadratic*. In case of *linked list*, we start constructing linked list that takes time and memory. But later we will see some situations where hashing is very useful.

Today, we will study these three strategies of hash implementation using animations. These animations will be provided to you in a *Java* program. It is important to mention here that all the data structures and algorithms we have studied already can be implemented using any of the languages of *C/C++* or *Java*. However, it is an important decision to choose the programming language because every language has its strong area, where it has better application. *Java* has become very popular because of its facilities for *Internet*. As you already know *C++*, therefore, *Java* is easy to learn for you. The syntax is quite similar. If we show you the java code you will say it is C++.

Let's see the hashing animation. This animation will be shown in the browser. This is an applet written in java language. We will see linear probing, quadratic probing and link list chaining in it. This is an example of how do we solve collision.
We have an array shown in four different columns. The size of the array is 100 and the index is from 0 to 99. Each element of the array has two locations so we can store 200 elements in it. When we have first collision the program will use the 2^{nd} part of the array location. When there is a 2^{nd} collision the data is stored using the linear probing. At the top right corner we have hash function *x* and its definition is "mod

100". That is when a number is passed to it, it will take mod with 100 and return the result which is used as index of the array.

In this example we are using numbers only and not dealing with the characters. We also have a statistical table on the right side. This program will generate 100 random numbers and using the hash function it will store these in the array. The numbers will be stored in the array at different locations. In the bottom left we have hashing algorithms. We have chosen the linear probing. Now the program will try to solve this problem using the linear probing. Press the run button to start it. It is selecting the numbers randomly, dividing it by 100 and the remainder is the hash value which is used as array index.

Here we have number 506. It is divided by 100 and the remainder is 6. It means that this number will be stored at the sixth array location. Similarly we have a number 206, now its remainder is also 6. As location 6 is already occupied so we will store 206 at the 2^{nd} part of the location 6. Now we have the number 806. Its remainder is also 6. As both the parts of location 6 are occupied. Using the linear probing we will store it in the next array location i.e. 7. If we have another number having the remainder as 6, we will store it at the 2^{nd} part of location 7. If we have number 807, its remainder is 7. The location 7 is already occupied due to the linear probing. Therefore the number 807 will be stored using the linear probing in the location 8. Now you can understand how the numbers are stored in the array. You can also see some clustering effect in it. See the location 63. All the numbers having remainder as 63 are clustered around location 63.

Let's change the collision resolution algorithm to quadratic probing. Run the animation again. Now we have array size as 75 and the array is shown in three columns. Each location of the array can store two numbers. In quadratic probing we add square of one first i.e. 1 and then the square of two and so on in case of collisions. Here we have used a different hash function. We will take the mod with 75. When the both parts of the array location is filled we will use the quadratic probing to store the next numbers. Analyze the numbers and see where the collisions have happened.

Lets see the animation using the linked list chaining. Now the hash function uses 50 to take mod with the numbers. So far pointers are not shown. When both parts of the location are filled, we will see the link list appearing. We have four numbers having remainder 0. The two numbers will be stored in the array and the next two will be stored using the link list which is attached at the 0 location.

We are not covering the hashing topic in much depth here as it is done in algorithms and analysis of algorithms domain. This domain is not part of this course. For the time being, we will see the usage of hashing. For certain situations, table ADT can be used, which internally would be using hashing.

Applications of Hashing

Let's see few examples of those applications where hashing is highly useful. The hashing can be applied in table ADT or you can apply hashing using your array to store and retrieve data.

– *Compilers use hash tables to keep track of declared variables (symbol table).*

Compilers use hash tables in order to implement *symbol tables*. A *symbol table* is an important part of compilation process. Compiler puts variables inside symbol table during this process. Compiler has to keep track of different attributes of variables. The name of the variable, its type, scope and function name where it is declared etc is put into the symbol table. If you consider the operations on symbol table, those can be insertion of variable information, search of a variable information or deletion of a variable. Two of these *insert* and *find* are mostly used operations. You might have understood already that a variable name will be parameter (or the key) to these operations. But there is one slight problem that if you named a variable as *x* outside of a code block and inside that code block, you declared another variable of the similar type and name then only name cannot be the key and scope is the only differentiating factor. Supposing that all the variables inside the program have unique names, variable name can be used as the key. Compiler will insert the variable information by calling the *insert* function and by passing in the variable information. It retrieves the variable value by passing in the variable name. Well, this exercise is related to your Compiler Construction course where you will construct you own language compiler.

Another usage of hashing is given below:

– *A hash table can be used for on-line spelling checkers — if misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time.*

You must have used spell checkers in a word processing program like MS Word. That spell checker finds out mistakes, provides you correct options and prompts you to choose any of the synonyms. You can also set the correct the words automatically.

Hashing can be used to find the spelling mistakes. For that you first take all the words from the dictionary of spoken English and construct a hash table of those. To find the spelling mistakes, you will take first word from the text that is being checked and compare it with all the words present inside the hash table. If the word is not found in the hash table then there is a high probability that the word is incorrect, although there is a low probability that the word is correct but it is not present in the dictionary. Based on the high probability a message can be displayed to the user of the application that the word is wrong. MS Word does the same. As far the automatic correct feature is concerned, it is another algorithm, which we are not going to discuss here.
Let's see few more examples in this connection.

– *Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again.*

Normal computer games are graphical, there are positions that are chosen by the computer or by the player. Consider the game of chess where one player has chosen one position again. Here we can use the positions of the pieces (64 pieces) at that time as the key to store it in the hash table. If our program wants to analyze that if a player has encountered the similar situation before, can pass in the positions of the pieces to the function *find*. Inside the function, when the positions are hashed again then the previously present index is returned, which shows that the similar situation has been encountered before.

See another example below:

- *Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different.*

Sometimes in your applications, you don't want to know which value is smaller or bigger but you are only interested in knowing if they are equal or not. For this, we can use hashing. If the two data items don't collide, then their hash values will be different. Based on this two values are said to be unequal.

Above was the situation when hashing can be useful. You may like to know in what circumstances hashing is not a good solution to apply.

When Hashing is Suitable?

- *Hash tables are very good if there is a need for many searches in a reasonably stable table.*

We have just seen the excellent example of reasonably stable hash table when we discussed hash table for English dictionary. We had constructed a hash table of all the words inside dictionary and were looking for different words in it. So majorly, there were frequent look up operations and insertions were in very minor frequency.

- *Hash tables are not so good if there are many insertions and deletions, or if table traversals are needed — in this case, AVL trees are better.*

In some applications, it is required to frequently read and write data. In these kinds of applications hash table might not be a good solution, AVL tree might be a good option. But bear in mind that there are no hard and fast statistics to go for hash table and then to AVL tree. You have to be a good software engineer to choose relevant data structure.

- *Also, hashing is very slow for any operations which require the entries to be sorted*
    - *e.g. Find the minimum key*

At times, you do other operations of *insert*, *delete* and *find* but additionally, you require the data in sorted order or the minimum or maximum value. We have discussed it many times that we insert data in the array of hash table without any sort order and it is scattered through the array in such a fashion that there are holes in the array. In these circumstances, the hash table is not useful. You might be remembering from the animation we saw earlier on in this lecture that there was no real sequence of filling of array. Some clusters were formed because of collision but there was no order as such. So hashing is not really useful in these circumstances.

We are finishing with our discussion on hashing. The important thing is how we thought about one data structure and internally we implemented in six different ways. You must be remembering that as long as the interface of the data structures remains the same, different internal implementations does not really matter from the client perspective. Occasionally, somebody might be interested in knowing the internal implementation of your data structure because that might be important for him in order to use your data structure.

Let's move on to the next topic of Sorting. It is very vast topic and cannot be covered in this course thoroughly.

Sorting

Sorting means to put the data in a certain order or sequence. We have discussed sorting before in different scattered through topics in this course but it has not been discussed so far as a separate topic. You must be remembering that when we traverse the binary search tree in in-order way, the obtained data happens to be sorted. Similarly, we saw other data structures, where we used to keep data in sorted order. In case of min-heap if we keep on removing elements one by one, we get data in sorted order.

Sorting is so useful that in 80-90% of computer applications, sorting is there in one form or the other. Normally, sorting and searching go together. Lot of research has been done on sorting; you can get lot of stuff on it from different sources. Very efficient algorithms have already been developed for it. Moreover, a vast Mathematical analysis has been performed of these algorithms. If you want to expose yourself, how these analyses are performed and what Mathematical tools and procedures are employed for performing analysis then sorting is very useful topic for you.

Sorting Integers

– *How to sort integers in this array?*

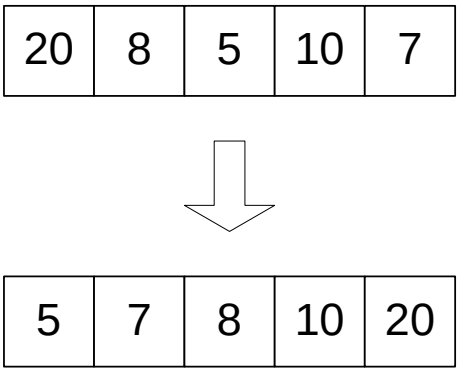


Fig 43.1

We want to sort the numbers given in the above array. Apparently, this operation may seem very simple. But think about it, if you are given a very large volume of data (may be million of numbers) then you may realize that there has to be an efficient mechanism to perform this operation. Firstly, let's put the problem in words:

We have a very large array of numbers. We want to sort the numbers inside the array in ascending order such that the minimum number of the array will be the first element of it and the largest element will be the last element at the end of the array.
Let's go to the algorithms of sorting. Point to be noted here that we are going to study algorithms of sorting; we are not talking about data structures. Until now, you might

have realized that algorithms go along data structures. We use a data structure to contain data and we use algorithms to perform certain operations or actions on that data.

Elementary Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort

These algorithms have been put as elementary because these are very simple. They will act as our baseline and we will compare them with other algorithms in order to find a better algorithm.

Selection Sort

- Main idea:
    - find the smallest element
    - put it in the first position
    - find the next smallest element
    - put it in the second position

…

- And so on, until you get to the end of the list

This technique is so simple that you might have found it yourself already. You search the whole array and find the smallest number. The smallest number is put on the first position of the array while the previous element in this position is moved somewhere else. Find the second smallest number and then put that number in the second position in the array, again the previous number in that position is shifted somewhere else. We keep on performing this activity again and again and eventually we get the array sorted. This technique is called selection sort because we select elements for their sorted positions.

In the next lecture, we will see how we can optimize this sorting operation. You read about sorting in your textbooks and from the Internet.

Data Structures

Lecture No. 44

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 7

7.1, 7.2

Summary

- Selection Sort
  - Selection Sort analysis
- Insertion Sort
  - Insertion Sort Analysis
- Bubble Sort
  - Bubble Sort analysis
- Summary
- N $\log_2$ (N) Algorithms

This is the sequel of the previous lecture in which we talked about the sort algorithms. There are three elementary sorting methods being discussed in this hand out. These are- selection sort, insertion sort and bubble sort. To begin with, we will talk about the selection sort algorithm.

Selection Sort

Suppose we have an array with different numbers. For sorting it in an ascending order, selection sorting will be applied on it in the following manner. We find the smallest number in the array and bring it to the position 1. We do the same process with the remaining part of the array and bring the smallest number among the remaining array to the next position. This process continues till the time all the positions of the array are filled with the elements. Thus the main idea of selection sort is that

- find the smallest element
- put it in the first position
- find the next smallest element in the remaining elements
- put it in the second position
- …
- And so on, until we get to the end of the array

Let's understand this algorithm by considering an example with the help of figures. Consider an array that has four numbers i.e. 19, 5, 7 and 12 respectively. Now we want to sort this array in ascending order. To sort the array, selection algorithm will be applied on it.

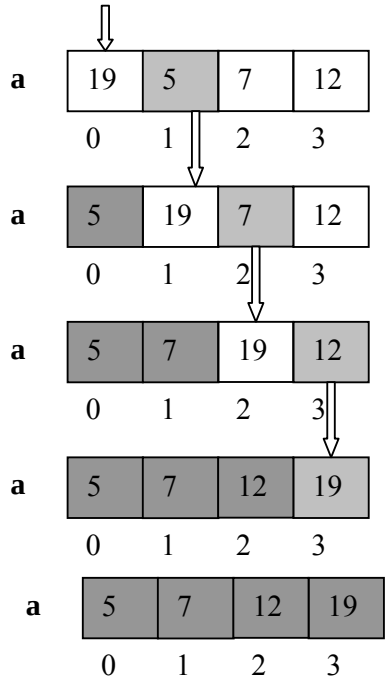


Figure 44.1: Selection Sort

The above pictorial representation explains the selection sort. It describes that at the start, we begin the search for the smallest number from the first position of the array i.e. from the index zero. We find that 5 is the smallest number and bring it to the first position i.e. index 0. Later, number 19 is put at the position that was occupied by number 5. Thus in a sense, we swap these numbers. Now 5, the smallest number in the array, has come at its final position i.e. index 0.

As index 0 has the proper number, so we start our search from position 2 i.e. index 1. Now we look at the remaining elements 19, 7, 12 and find the smallest number among them. Here 7 is the smallest so we change its position with 19 to bring 7 at its position. Thus 5 and 7 have come at their final positions. Now there are two elements are left behind i.e. 12 and 19. We search the smallest among these and find that 12 is the smallest. So we swap it with 19 to bring it at index 2 i.e. its final position. Now there is last element remaining and obviously it is at its position as there is no element to compare with it. The array is now in the sorted form with ascending numbers.

The point to remember in the selection search is that at the beginning, we start search for the smallest number from index 0 (i.e. first position). After it we start search from the index 1 (i.e. position 2). After each search one number gets its final position so we start the next search from the next position to it. Thus we do the multiple passes of the array to sort it. First, we pass through the n elements of the array and search the n-1 elements and then n-2. Thus at last, we come to the single and last element of the array.
Now let's see the code of this algorithm in C++.

```
void selectionSort(int *arr, int N)
```

```
{
        int posmin, count, tmp ;
        for (count=0;count<N;count++)
        {
                posmin = findIndexMin(arr, count,
N) ;
                tmp=arr[posmin] ;
                arr[posmin]=arr[count] ;
                arr[count]=tmp ;
        }
}

int findIndexMin (int *arr, int start,          int N)
{
        int posmin=start ;
        int index ;
        for(index=start; index < N; index++)
                if (arr[index]<arr[posmin])
                        posmin=index ;
        return posmin ;
}
```

In the above code, we write the function *selectionSort*. This function takes an array of integers as ***arr* and the size of array as *N*. There are local variables declared in function as *posmin, count* and *tmp*. Then there is a '*for* loop' that starts from zero and goes to *N-1*. This is due to the fact that the index of array of N elements is from zero to N-1. The condition *count* < *N* indicates that loop will execute as long as *count* is less than *N* and will exit when count gets *N*. Now in the loop, we calculate the *posmin* with a function i.e. *findIndexMin*. This *findIndexMin* method is written below in the code. This routine or method works in the way that we pass to it the whole array i.e. *arr*, the value of *count* (what it is in that execution of loop) and size of the array i.e. *N*. This routine starts the search from the index equal to value of *count* and goes to Nth position, returning the position of the minimum (smallest) element. Now in the first routine, we get this position value in the variable *posmin*. Thus the code line:

```
posmin = findIndexMin(arr, count, N) ;
```

gets the position of the smallest number returned by the method findIndexMin in the variable *posmin*. After finding this position, we do the swapping of this *posmin* with the *count* position. Thus we put the value of position *posmin* in the *count* position.

The *findIndexMin* routine is such that it takes *arr, start* and *N* as arguments. It starts searching from the *start* position in the *for* loop, finds the position of the minimum value and returns that position.

This sorting algorithm is also known as the in-place sorting algorithm as there is no need of additional storage to carry out this sorting. The pictorial representation of the swap action of this algorithm will be as under:

We want to sort the following array that has five elements. We start the search from

the first element and find that the smallest element is 5. As we have started our search from index 0 (that is the *count* in our above code) so we swap 5 with the value at index 0 i.e. 20. After this swap, 5comes at the position of 20 while 20 goes to the position of 5. Afterwards, we start the search from index 1. We find that 7 is the smallest number among the remaining elements. It swaps its position with 8. After this, in the remaining three elements, 8 is the smallest. This number 8 swaps its position with 20. At the end, 10 is the smallest number among 10 and 20. So there is no need of swapping as 10 has already got its position.

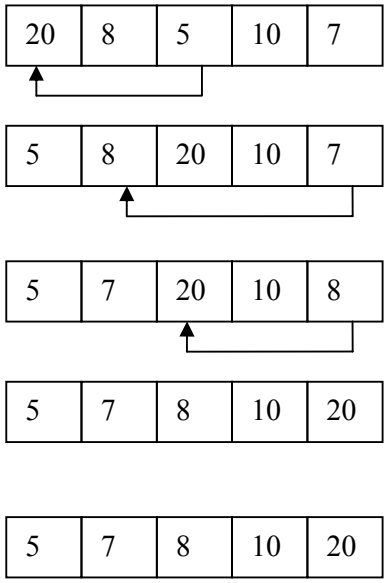


Figure 44.2: Swap Action (selection sorting)

Selection Sort Analysis

We have seen in the code for the algorithm that there are two loops. The loop in the *selectionSort* method passes the array to search the smallest element and the second loop that is in the findIndexMin method finds the position (index) of the smallest value. To find the first smallest element, we have to go through the N elements of the array. For the purpose of finding the second smallest element, we have to search the N-1 elements. During this search process, we have to find two elements for the second last smallest element. And obviously in the end, there is one element that is at its proper position, necessitating no search. We have to do all these searches. These are N, N-1, N-2 ……2, 1 for the first, second, third ……second last and last element respectively. Now if we want to find the total searches, the addition of all these searches together will help us get a sum total as given in the following equation.

$$\text{Total searches} = 1 + 2 + 3 + \ldots\ldots + (N-2) + (N-1) + N$$
$$= N(N+1) / 2$$
$$= (N^2 + N) / 2$$

Suppose if N is 10, then according to this formula, the total searches will be (100 + 10) / 2 i.e. 55. If N is 100, the total searches will be (10000 + 100) / 2 i.e. 5050. Similarly if N is 1 million, N^2 is going to be very large as compared to N. Thus there we can ignore N and can say that the time (total searches) is proportional to N^2.

This means that larger the N, greater will be the performance of selection with respect to $N^{2.}$

Insertion Sort

The main idea of insertion sort is

- Start by considering the first two elements of the array data. If found out of order, swap them
- Consider the third element; insert it into the proper position among the first three elements.
- Consider the fourth element; insert it into the proper position among the first four elements.
- … …

This algorithm is not something uncommon to the persons who know card playing. In the game of cards, a player gets 13 cards. He keeps them in the sorted order in his hand for his ease. A player looks at the first two cards, sorts them and keeps the smaller card first and then the second. Suppose that two cards were 9 and 8, the player swap them and keep 8 before 9. Now he takes the third card. Suppose, it is 10, then it is in its position. If this card is of number 2, the player will pick it up and put it on the start of the cards. Then he looks at the fourth card and inserts it in the first three cards (that he has sorted) at a proper place. He repeats the same process with all the cards and finally gets the cards in a sorted order. Thus in this algorithm, we keep the left part of the array sorted and take element from the right and insert it in the left part at its proper place. Due to this process of insertion, it is called insertion sorting.

Let's consider the array that we have used in the selection sort and sort it now with the insertion sorting. The following figure shows the insertion sort of the array pictorially.

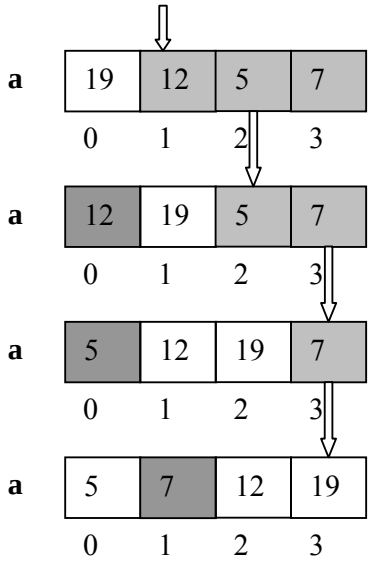


Figure 44.3: Insertion Sort

The array consists of the elements 19, 12, 5 and 7. We take the first two numbers i.e. 19 and 12. As we see 12 is less than 19, so we swap their positions. Thus 12 comes at index 0 and 19 goes to index 1. Now we pick the third number i.e. 5. We have to find the position of this number by comparing it with the two already sorted numbers. These numbers are 12 and 19. We see that 5 is smaller than these two. So it should come before these two numbers. Thus the proper position of 5 is index 0. To insert it at index 0, we shift the numbers 12 and 19 before inserting 5 at index 0. Thus 5 has come at its position. Now we pick the number 7 and find its position between 5 and 12. To insert 7 after 5 and before 12, we have to shift the numbers 12 and 19 to the right. After this shifting, we put number 7 at its position. Now the whole array has been sorted so the process stops here.

Following is the code of the insertion sort in C++.

```
void insertionSort(int *arr, int N)
{
int pos, count, val;
        for(count=1; count < N; count++)
        {
                val = arr[count];
                for(pos=count-1;   pos   >=   0;
pos--)
                if (arr[pos] > val)
                        arr[pos+1]=arr[pos];
                  else  break;

         arr[pos+1] = val;
    }
}
```

In this sorting function, we start the process from index 0 and 1 and swap them. Afterwards, we go to the third position and put it into its proper position. While inserting a number in the sorted portion of the array, we have to shift the elements. This shifting is an additional overhead of this sorting algorithm. Due to this shifting, the sorting requires more time. This algorithm is also an in place algorithm as we don't need any additional storage. At the most, we need some variables of local nature, normally used in programs.
From the above code, we see that the name of this sorting routine is *insertionSort*. It takes an array and its size as arguments. There are local variables *pos*, *count* and *val* declared in this function. After this there is a *for* loop that starts from the *count* having value equal to 1. Now we put the value of *count* index (i.e. arr[count]) in variable *val*. This value has to find its place in the left sorted portion of the array. To find that position we have to execute one more *for* loop. This loop starts from *count-1* and goes to down to zero. In the body of the loop we compare the value of *val* with the value at current position in the array i.e. arr[pos]. If *val* is less than the *arr[pos]* i.e. value at current index. It means that the *val* has to go to the left position of *arr[pos]*.*So* we shift the *arr[pos]* to right to create place for the new value i.e. *val*. When the loop exits, we put this value *val* at arr[pos + 1]. Thus we have inserted the number in the array at its proper position.

Following is the step by step explanation for the insertion sort of the above example with same previous array.

First of all we take the first two elements 19 and 12. As 12 is less than 19, we do right shift on 19. And put 12 at its position i.e. index 0. Afterwards, we go to index 2. There is 5 at this position. As we see that 5 is less than the other elements on the left side of array, it has to come at the first position. To bring 5 to first position, the number 12 and 19 has to be shifted to right. After this shifting, the position of index 0 becomes empty and we put 5 there. Finally, there is number 7. The position of 7 will be between 5 and 12. For this, we have to shift 12 and 19 towards right so that the place for 7 could be created. We put 7 at that place. Thus the array is sorted now.

Insertion Sort Analysis

Let's analyze that when the value of N increases. How much time for insertion sort increases? In the code of insertion sort, we have seen that there are outer and inner loops. Due to these two loops, we can understand that it is also like N^2 algorithm. In the sort process, there may be a situation that every iteration inserts an element at the start of the array by shifting all sorted elements along. Now if we have to bring the second element to its position, there will be need of shifting the first element. This means that we have to shift one element. Similarly, for placing the third element at the start position (we are discussing the worst case scenario in which at every iteration the element has to go to the first position), we have to shift two elements. Thus we sum up all the shiftings, the total becomes 2 + 3 + 4 +……. + N-1 + N.
The summation can be written as follows.

$$\text{Total} = (2 + N)(N - 1) / 2$$
$$= O(N^2)$$

From this expression, we see that when the value of N increases, the value of N^2 will dominate. It will increase significantly with respect to N. Thus we see that insertion

sort is also an N^2 algorithm like selection sort.

Bubble Sort

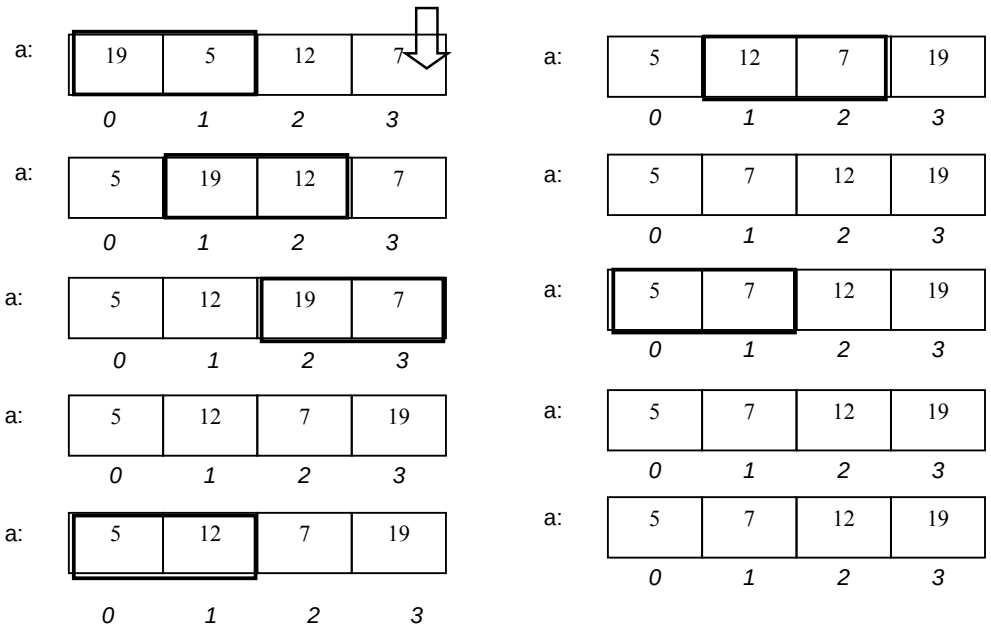
The third sorting algorithm is bubble sort. The basic idea of this algorithm is that we bring the smaller elements upward in the array step by step and as a result, the larger elements go downward. If we think about array as a vertical one, we do bubble sort. The smaller elements come upward and the larger elements go downward in the array. Thus it seems a bubbling phenomenon. Due to this bubbling nature, this is called the bubble sort. Thus the basic idea is that the lighter bubbles (smaller numbers) rise to the top. This is for the sorting in ascending order. We can do this in the reverse order for the descending order.

The steps in the bubble sort can be described as below

- Exchange neighboring items until the largest item reaches the end of the array
- Repeat the above step for the rest of the array

In this sort algorithm, we do not search the array for the smallest number like in the other two algorithms. Also we do not insert the element by shifting the other elements. In this algorithm, we do pair-wise swapping. We will take first the elements and swap the smaller with the larger number. Then we do the swap between the next pair. By repeating this process, the larger number will be going to the end of the array and smaller elements come to the start of the array.

Let's try to understand this phenomenon with the help of figures how bubble sort works. Consider the same previous array that has elements 19, 12, 5 and 7.



First of all, we compare the first pair i.e. 19 and 5. As 5 is less than 19, we swap these

elements. Now 5 is at its place and we take the next pair. This pair is 19, 12 and not 12, 7. In this pair 12 is less than 19, we swap 12 and 19. After this, the next pair is 19, 7. Here 7 is less than 19 so we swap it. Now 7 is at its place as compared to 19 but it is not at its final position. The element 19 is at its final position. Now we repeat the pair wise swapping on the array from index 0 to 2 as the value at index 3 is at its position. So we compare 5 and 12. As 5 is less than 12 so it is at its place (that is before 12) and we need not to swap them. Now we take the next pair that is 12 and 7. In this pair, 7 is less than 12 so we swap these elements. Now 7 is at its position with respect to the pair 12 and 7. Thus we have sorted the array up to index 2 as 12 is now at its final position. The element 19 is already at its final position. Note that here in the bubble sort, we are not using additional storage (array). Rather, we are replacing the elements in the same array. Thus bubble sort is also an in place algorithm. Now as index 2 and 3 have their final values, we do the swap process up to the index 1. Here, the first pair is 5 and 7 and in this pair, we need no swapping as 5 is less than 7 and is at its position (i.e. before 7). Thus 7 is also at its final position and the array is sorted.

Following is the code of bubble sort algorithm in C++.

```
void bubbleSort(int *arr, int N)
{
        int i, temp, bound = N-1;
        int swapped = 1;
        while (swapped > 0 )
        {
                swapped = 0;
                for(i=0; i < bound; i++)
                    if ( arr[i] >  arr[i+1] )
                        {
                        temp = arr[i];
                        arr[i] = arr[i+1];
                        arr[i+1] = temp;
                        swapped = i;
                          }
       bound = swapped;
    }
}
```

In line with the previous two sort methods, the *bubbleSort* method also takes an array and size of the array as arguments. There is *i, temp, bound* and *swapped* variables declared in the function. We initialize the variable *bound* with N–1. This N-1 is our upper limit for the swapping process. The outer loop that is the *while* loop executes as long as swapping is being done. In the loop, we initialize the swapped variable with zero. When it is not changed in the *for* loop, it means that the array is now in sorted form and we exit the loop. The inner *for* loop executes from zero to *bound-1*. In this loop, the if statement compares the value at index i and i+1. If I (element on left side in the array) is greater than the element at i+1 (element on right side in the array) then we swap these elements. We assign the value of i to the swapped variable that being greater than zero indicates that swapping has been done. Then after the *for* loop, we put the value of swapped variable in the bound to know that up to this index, swapping has taken place. After the *for* loop, if the value of *swapped* is not zero, the

while loop will continue execution. Thus the *while* loop will continue till the time, the swapping is taking place.
Now let's see the time complexity of bubble sort algorithm.

Bubble Sort Analysis

In this algorithm, we see that there is an outer loop and an inner loop in the code. The outer loop executes N times, as it has to pass through the whole array. Then the inner loop executes for N times at first, then for N-1 and for N-2 times. Thus its range decreases with each of the iteration of the outer loop. In the first iteration, we do the swapping up to N elements. And as a result the largest elements come at the last position. The next iteration passes through the N-1 elements. Thus the part of the array in which swapping is being done decreases after iteration. At the end, there remains only one element where no swapping is required. Now if we sum up these iterations i.e. 1 + 2 + 3 + …… + N-1 + N. Then this summation becomes N (1 + N) / 2 = O (N^2). Thus in this equation, the term N^2 dominates as the value of N increases. It becomes ignorable as compared to N^2. Thus when the value of N increases, the time complexity of this algorithm increases proportional to N^2.

Summary

Now considering the above three algorithms, we see that these algorithms are easy to understand. Coding for these algorithms is also easy. These three algorithms are in place algorithms. There is no need of extra storage for sorting an array by these algorithms. With respect to the time complexity, these algorithms are proportional to N^2. Here N is the number of elements. So we can see that as the value of N increases, the performance time of these algorithms increases considerably as it is proportional to N^2. Thus these algorithms are expensive with respect to time performance. There are algorithms that have the time complexity proportional to N $\log_2$ (N). The following table shows the respective values of N^2 and N $\log_2$(N) for some values of N.

N	N^2	N Log_2 (N)
10	100	33.21
100	10000	664.38
1000	1000000	9965.78
10000	100000000	132877.12
100000	10000000000	1660964.04
1000000	1E+12	19931568.57

From this table we can see that for a particular value of N, the value of N^2 is very large as compared to the value of N $\log_2$ (N). Thus we see that the algorithms whose time complexity is proportional to N^2 are much time consuming as compared to the algorithms the time complexity of which is proportional to N $\log_2$ (N). Thus we see that the N $\log_2$ (N) algorithms are better than the N^2 algorithms.

N $\log_2$ (N) Algorithms

Now let's see the algorithms that are N $\log_2$ (N) algorithms. These include the following algorithms.

- Merge Sort
- Quick Sort
- Heap Sort

These three algorithms fall under 'divide and conquer category'. The divide and conquer strategy is well known in wars. The philosophy of this strategy is ,' divide your enemy into parts and then conquer these parts'. To conquer these parts is easy, as these parts cannot resist or react like a big united enemy. The same philosophy is applied in the above algorithms. To understand the divide and conquer strategy in sorting algorithm, let's consider an example. Suppose we have an unsorted array of numbers is given below.

10	12	8	4	2	11	7	5

Now we split this array into two parts shown in the following figure.

10	12	8	4	**2**	**11**	**7**	**5**

Now we have two parts of the array. We sort these parts separately. Suppose we sort these parts with an elementary sort algorithm. These parts may be sorted in the following manner.

4	8	10	12	**2**	**5**	**7**	**11**

After this we merge these two parts and get the sorted array as shown below.

2	**4**	**5**	**7**	**8**	**10**	**11**	**12**

Data Structures

Lecture No. 45

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 7 7.6, 7.7

Summary

- Divide and Conquer
- Mergesort
- mergeArrays
- Mergesort and Linked Lists
- Mergesort Analysis
- Quicksort
- Course Overview

Divide and Conquer

In the previous lecture, we had started discussing three new sorting algorithms; *merge sort, quick sort* and *heap sort*. All of these three algorithms take time proportional to $nlog_2n$. Our elementary three sorting algorithms were taking n^2 time; therefore, these new algorithms with $nlog_2n$ time are faster. In search operation, we were trying to reduce the time from n to log_2n.

Let's discuss these sorting algorithms; merge sort, quick sort and heap sort in detail. We had started our discussion from *divide and conquer* rule where we also saw an example. Instead of sorting a whole array, we will divide it in two parts, each part is sorted separately and then they are merged into a single array.

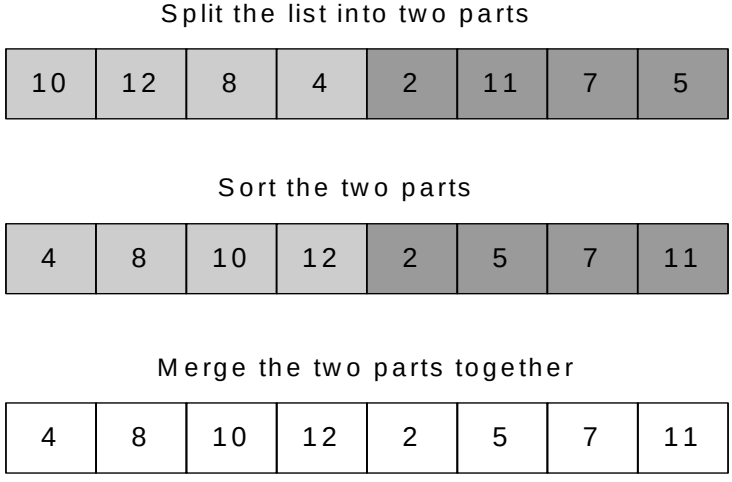


Fig 45.1

Let' see few analysis to confirm the usefulness of the divide and conquer technique.

- *To sort the halves approximate time is* $(n/2)^2+(n/2)^2$
- *To merge the two halves approximate time is n*
- *So, for n=100, divide and conquer takes approximately:*

$= (100/2)^2 + (100/2)^2 + 100$
$= 2500 + 2500 + 100$
$= 5100$

We know that elementary three sorting algorithms were taking approximately n^2 time. Suppose we are using insertion sort of those elementary algorithms. We divide the list into two halves then the time will be approximately $(n/2)^2+(n/2)^2$. The time required for merging operation is approximately *n*. This operation contains a simple loop that goes to *n*.

Suppose that n is 100. Considering if we apply insertion sort algorithm on it then the time taken will be approximately $(100)^2 = 10000$. Now, if we apply divide and conquer technique on it. Then for first half approximate time will be $(100/2)^2$. Similarly for second half it will be $(100/2)^2$. The merging approximate time will be *100*. So the whole operation of sorting using this divide and conquer technique in insertion sort will take around $(100/2)^2 + (100/2)^2+100 = 5100$. Clearly the time spent (5100) after applying divide and conquer mechanism is significantly lesser than the previous time (10000). It is reduced approximately to half of the previous time. This example shows the usefulness of divide and conquer technique.

By looking at the benefit after dividing the list into two halves, some further questions arise:

- Why not divide the halves in half?
- The quarters in half?
- And so on . . .
- When should we stop?
  At $n = 1$

So we stop subdividing the list when we reach to the single element level. This divide and conquer strategy is not a thing, we have already prepared binary search tree on the same lines. One side of the tree contains the greater elements than the root and other part contains the smaller elements. Especially, when performing binary search in an array, we had started our search from mid of it. Subdivided the array and kept on comparing and dividing the array until we got success or failure. The subdivision process may prolong to individual element of the array.

Recall Binary Search

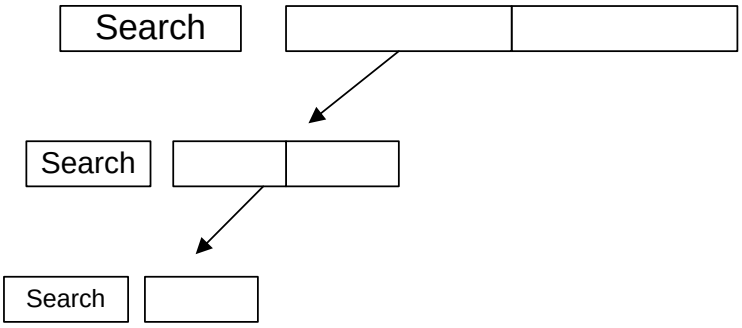


Fig 45.2

Hence, we used to perform binary search on the same lines of divide and conquer strategy. Remember, we applied binary search on sorted array. From this one can realize that sorting facilitates in searching operations.

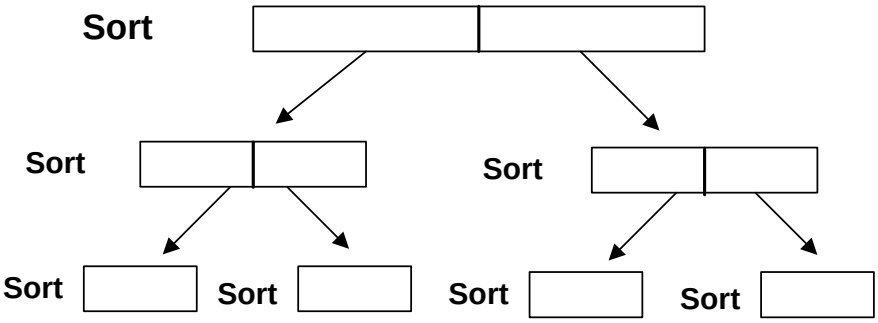


Fig 45.3

In figure 45.3, in order to sort the list, we have divided the list in the two parts and each part is subdivided into further subparts. At end each part is consisting of either single element or maximum two elements. If we have two numbers to sort, we can compare them (or sort them) with single if statement. After sorting individual subparts, we start merging them in the upward direction as shown in the figure Fig 45.4.

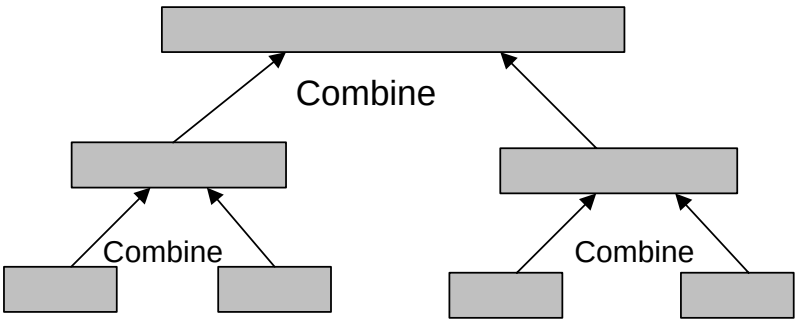


Fig 45.4

In Fig 45.4, we have four sorted smaller parts. We combine them to become two sorted parts and two sorted parts are further combined or merged to become one

sorted list.

Mergesort

- ***Mergesort*** *is a divide and conquer algorithm that does exactly that.*
- *It splits the list in half*
- *Mergesorts the two halves*
- *Then merges the two sorted halves together*
- *Mergesort can be implemented recursively*

Let's see the mergsort algorithm, how does that work.

- *The mergesort algorithm involves three steps:*
  - *If the number of items to sort is 0 or 1, return*
  - *Recursively sort the first and second halves separately*
  - *Merge the two sorted halves into a sorted group*If the data is consisting of 0 or 1 element then there is nothing required to be done further. If the number of elements is greater than 1 then apply divide and conquer strategy in order to sort them. Divide the list into two halves and sort them separately using recursion. After the halves (subparts) have been sorted, merge them together to get a list of sorted elements.

We will discuss recursive sorting in a moment, before that let's see the merging operation using pictures. We have two sorted array and another empty array whose size is equal to the sum of sizes of two sorted arrays.

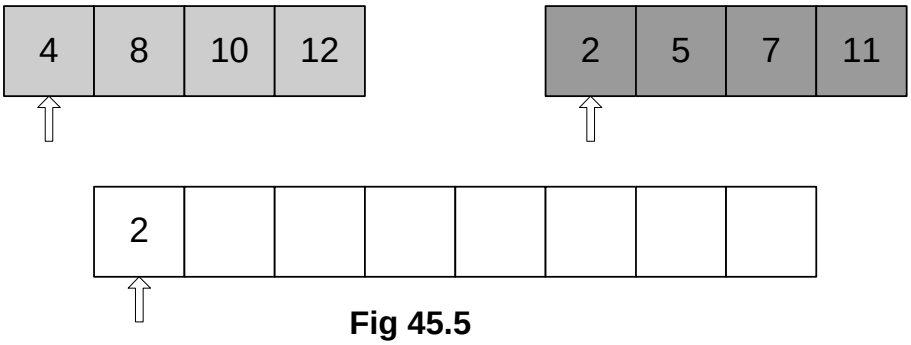


Fig 45.5

You can see from Fig 45.5, array 1 on the top left is containing 4 elements, top right is also containing 4 elements and both of them are sorted internally. Third array is containing 4+4 = 8 elements.

Initially, very first elements (present at the starting index 0 of array) of both the arrays are compared and the smaller of them is placed in the initial position of the third array. You can see from Fig 45.5 that elements 4 and 2 are compared pointed to by the indexes (actually arrays current indexes). The smaller of them is 2, therefore, it is placed in the initial position in the third array. A pointer (the current index of array) is also shown for third array that will move forward as the array is filled in. The smaller number was from right array, therefore, its pointer is moved forward one position as shown in Fig 45.6.

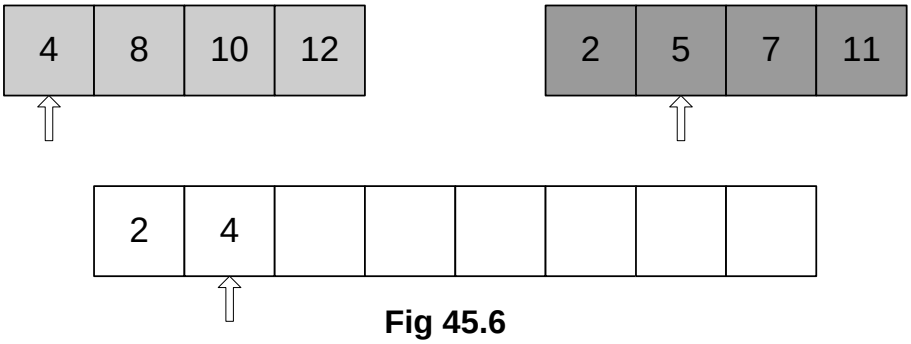


Fig 45.6

This time the numbers at current positions in both the arrays are compared. As 4 is smaller of the two numbers, therefore, it is put in the third array and third array's pointer is moved one position forward. Also because this time, the number has been chosen from left array, therefore, its pointer is also moved forward. The updated figure is shown in Fig 45.7.

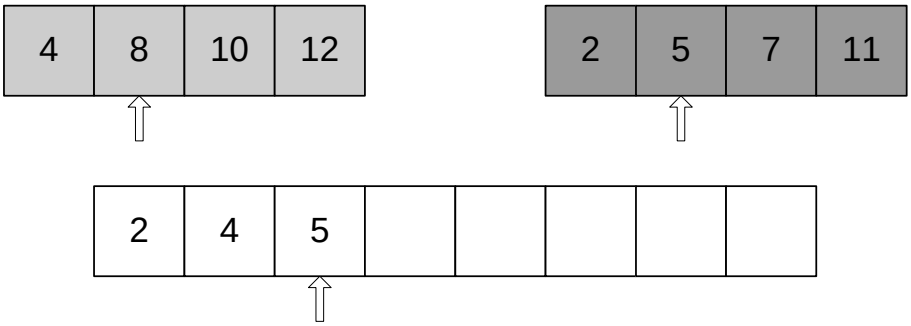


Fig 45.7

Next, numbers 8 and 5 are compared. As 5 is smaller of the two, it is put in the third array. The changed positions of pointers and the next comparison are shown in Fig 45.8.

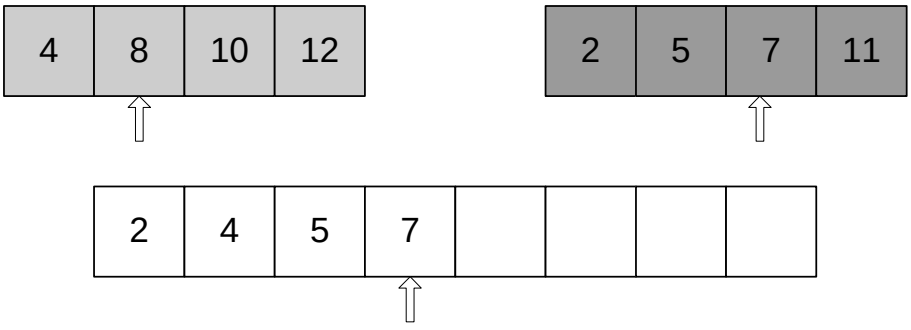


Fig 45.8

Fig 45.9 has showed the situation after next comparison done in the similar manner.

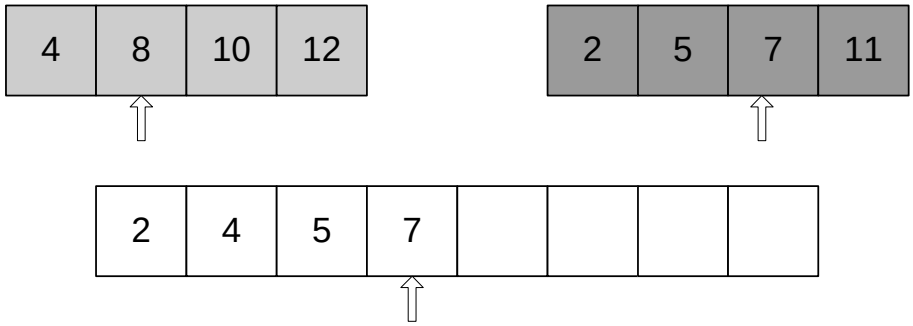


Fig 45.9

By now, you must have understood how the merging operation works. Remember, we do merging when we have two parts sorted already.
Consider this operation in terms of time and complexity. It is performed using a simple loop that manipulates three arrays. An index is used for first array, which starts from the initial position to the size of the array. Similar an index is used for second array, which starts from the initial position and ends at the end of the array. A third index is used for third array that sizes to the sum of the maximum values of both previous indexes. This is simple single loop operation, which is not complex.

Now, let's see sorting using recursion now pictorially.

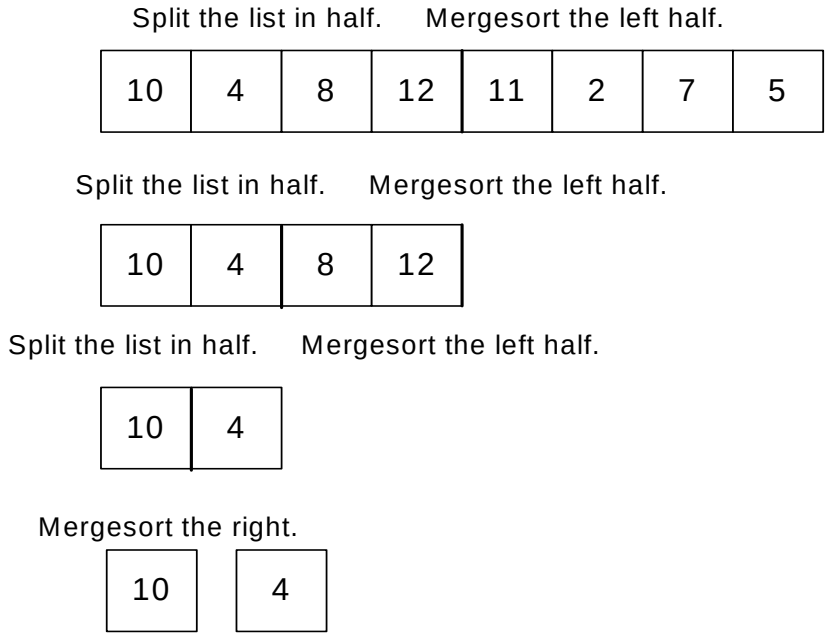


Fig 45.10

At the top of Fig 45.10 is an array, which is not sorted. In order to sort it using recursive mechanism, we divide the array into two parts logically. Only logical partitions of the array are required bases on size and index of the arrays. The left half of the array containing 10,4,8 and 12 is processed further for sorting. So actually, while calling the mergesort algorithm recursively, only that particular half of the array

is a passed as an argument. This is spitted further into two parts; 10, 4 and 8, 12. 10, 4 half is processed further by calling the *mergesort* recursively and we get both numbers 10 and 4 as separate halves. Now these halves are numbers and they cannot be subdivided therefore recursive call to *mergesort* will stop here. These numbers (new halves) 10 and 4 are sorted individually, so they are merged. When they are merged, they become as 4,10 as shown in Fig 45.11.

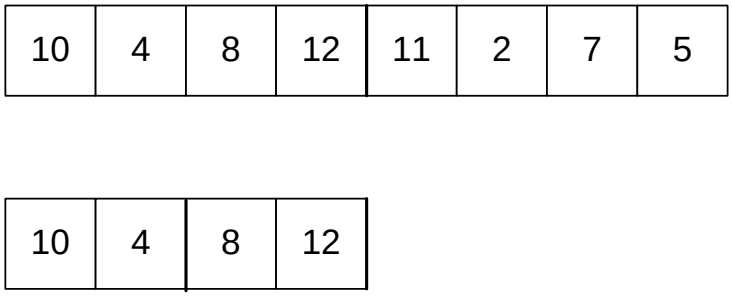


Fig 45.11

The recursive call to *mergesort* has done its work for left half that consisted of 10 and 4. Now, we apply the same technique (the recursive calls to *mergesort*) to the right half that was consisting of 8 and 12. 8 and 12 are spitted into separate numbers as indicated in Fig 45.11. Further division of them is not possible, therefore, they are merged as shown in Fig 45.12.

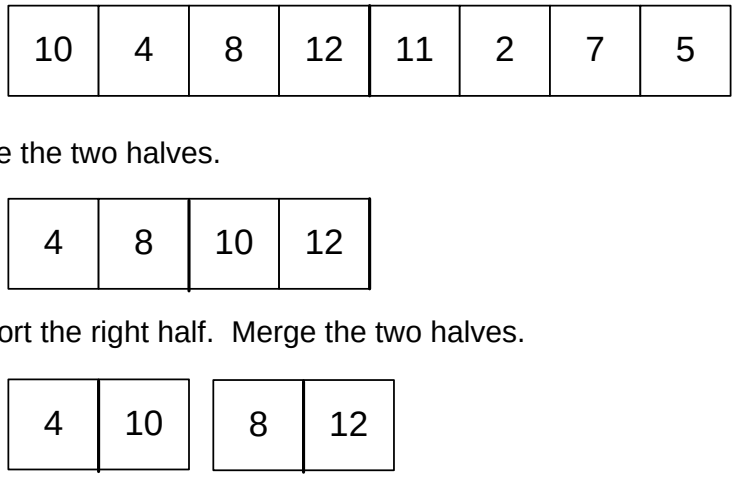


Fig 45.12

At this point in time, we have two parts one is containing elements as 4, 10 and other as 8, 12. These two parts are merged. The merged half is shown in Fig 45.12. Note that it is completely sorted as 4,8,10 and 12. This completes the operation on the left half of the array. We do similar process with the right half now, which is consisting of 11, 2, 7 and 5.

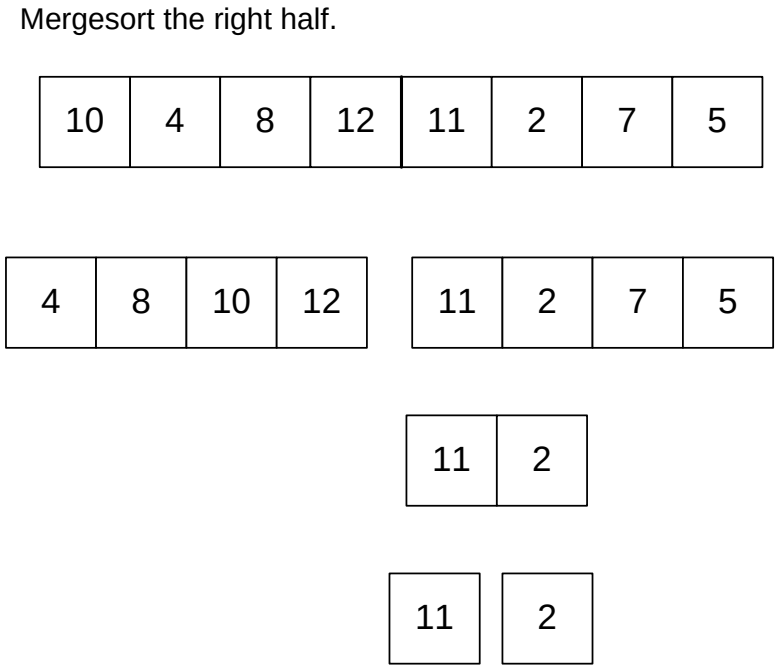


Fig 45.13

Similarly, left half of it 11,2 is processed recursively. It is divided into two halves and we have two parts as two individual numbers. The recursive call for this part stops here and the merging process starts. When the parts 11 and 2 are merged, they become one sorted part as shown in Fig 45.14.

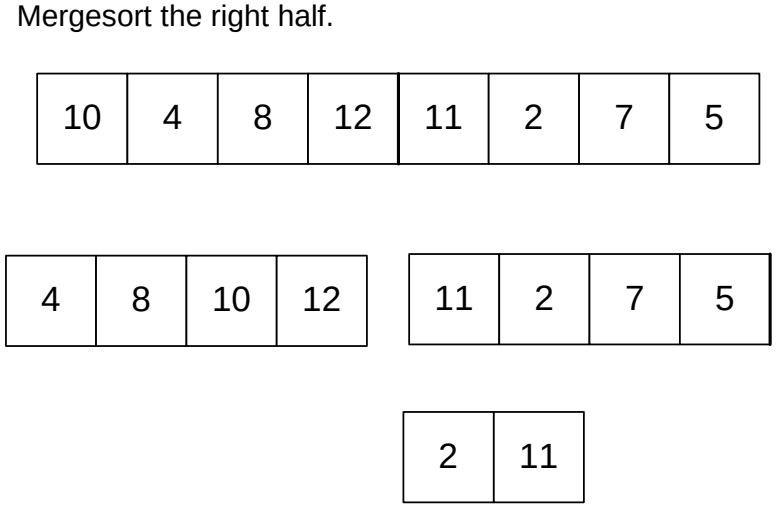


Fig 45.14

Now the same procedure is applied on right half consisting of 7 and 5. By applying recursive mechanism, it is further subdivided into two individual number parts.

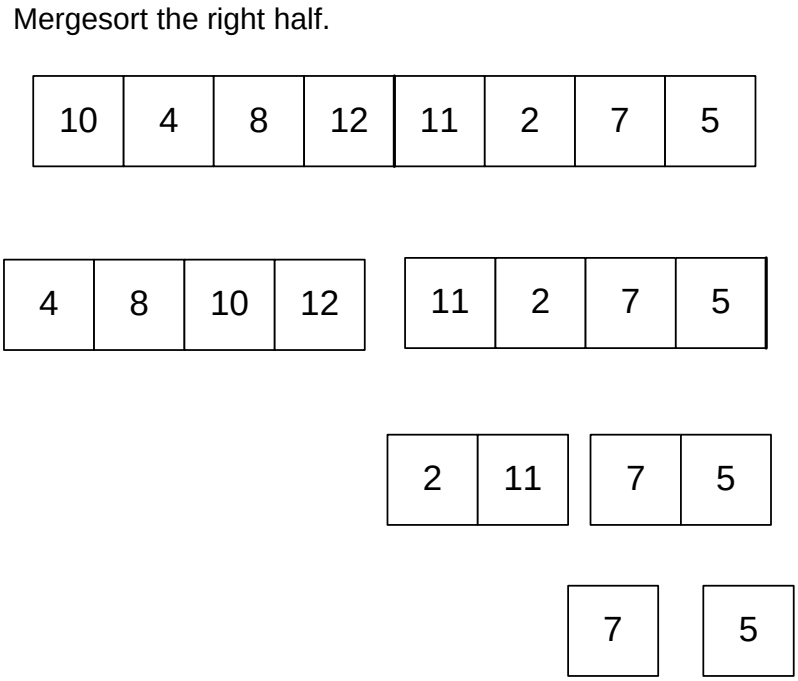


Fig 45.15

The merging operation starts, the resultant is shown in the Fig 45.16. After merging, the new part has become 5,7.

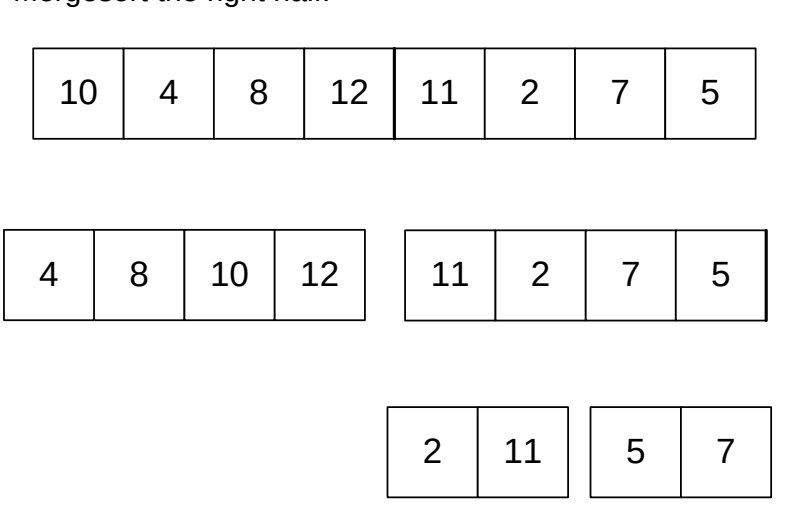


Fig 45.16

When 5,7 merged part is further merged with already merged part 2,11. The new half

becomes as 2,5,7,and 11 as shown in Fig 45.17.

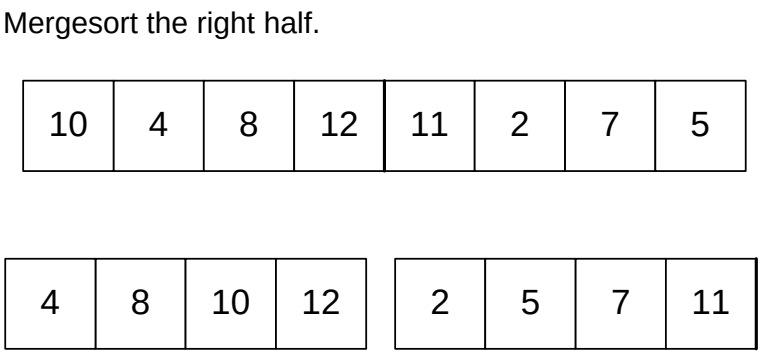


Fig 45.17

Now, we can merge the two biggest halves to get the array in sorted order. After merging, the resulted sorted array is shown in Fig 45.18.

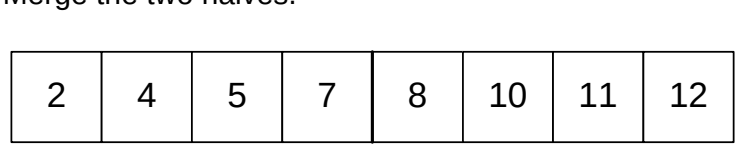


Fig 45.18

Let's see the C++ code for this sorting algorithm. We are not going to write a class of it but at the moment, we are only writing it as a procedure. We can use it standalone or later on, we can also change it to make it a class member.

```
void  mergeSort(float  array[], int  size)
{
        int * tmpArrayPtr = new int[size];
        if (tmpArrayPtr != NULL)
                mergeSortRec(array, size, tmpArrayPtr);
        else
        {
                cout << "Not enough memory to sort list.\n");
                return;
        }
        delete [] tmpArrayPtr;
}
void mergeSortRec(int array[], int size, int tmp[])
{
        int  i;
        int  mid = size/2;

        if (size > 1)
        {
```

```
            mergeSortRec(array, mid, tmp);
            mergeSortRec(array+mid, size-mid, tmp);
            mergeArrays(array, mid, array+mid, size-mid, tmp);
            for (i = 0; i < size; i++)
                  array[i]  =  tmp[i];
      }
}
```

The name of the procedure is *mergeSort*, it is accepting first argument as an array of *float*s and as second argument as an *int*. The second parameter is containing the size of the array. Inside the function, we have created a temporary array *tmpArrayPtr* dynamically of the same size as the size passed in the second argument. This array will be used for merging operation. In the next *if* statement *if (tmpArrayPtr != NULL)*, allocation for dynamic array is being checked, if it is allocated then another function *mergeSortRec* is called. Its first argument is *float*'s array *array* , second argument is *int size*, and third argument is *tmpArrayPtr*.
In *mergeSortRec*, we divided the *size* of the array by 2 to get the mid of the array. If the *size* of the array is greater than 1 then inside the if-statement, *mergeSortRec* is being called recursively two times. In first recursive call, *mergeSortRec(array, mid, tmp);* we provided the array from index 0 to *mid*. For second recursive call, *mergeSortRec(array+mid, size-mid, tmp);*we provided the array starting from one position ahead of *mid* position to the end of the array, however, the size of the passed array is *size-mid*. The first recursive call will keep on executing by further subdividing the left part of the half every time until the size of the array reduces to 1. Similarly, the second recursive call keeps on working on the right half of the array until the size of the array reduces to 1. After both of these recursive calls will be completed then the merging operation is performed by calling *mergeArrays*. At end, inside the for loop, the sorted data is copied from *tmp* array to the actual array *array*.

mergeArrays

Let's see the merging of two array while using their indexes as i and j.

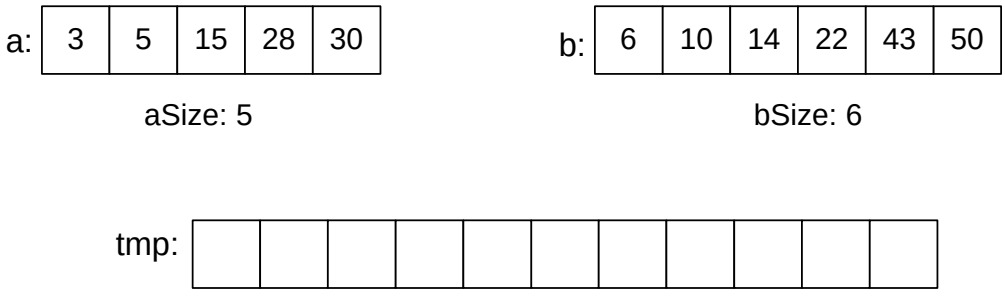
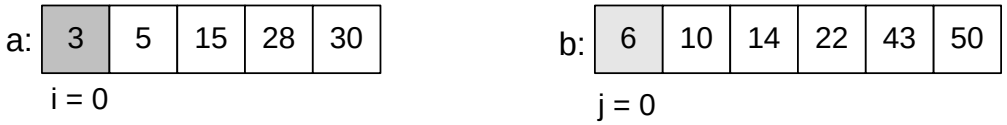


Fig 45.19

We are doing the same operation of merging of arrays but with little more detail here. One the left is an array *a* consisting of 2,5,15,28 and 30 and on the right is another array *b* consisting of 6,10,14,22,43 and 50. The size of array *a* is 6 while *b*'s size is 6. We create a temporary array named *tmp*, whose size is 11. We have used indexes *i*, *j* and *k* for arrays *a*, *b* and *tmp* respectively and each one of them has been initialized with 0.



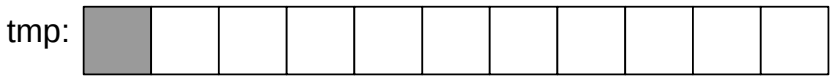
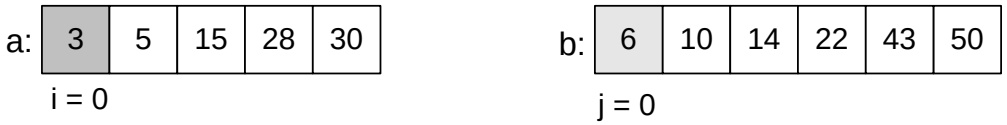


Fig 45.20

We compare the initial elements 3 and 6 of the arrays *a* and *b*. As 3 is smaller than 6, therefore, it is put in the temporary array *tmp*'s starting location where *k=0*.



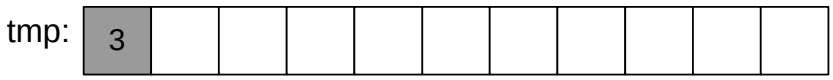
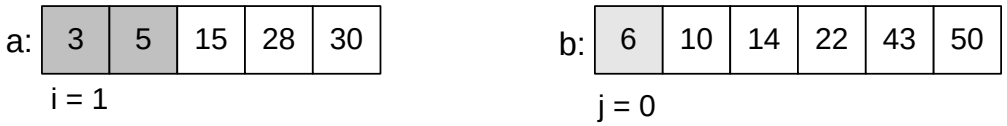


Fig 45.21

i is incremented and now elements 5 and 6 are compared. 5 being smaller of these take the place in the *tmp* array as shown in Fig 45.22.



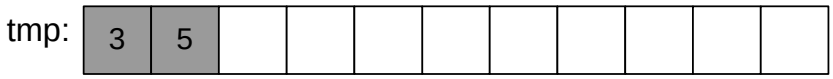


Fig 45.22

i is incremented again and it reaches the element 15. Now, elements 15 and 6 are compared. Obviously 6 is smaller than 15, therefore, 6 will take the place in the *tmp* array as shown in Fig 45.23.

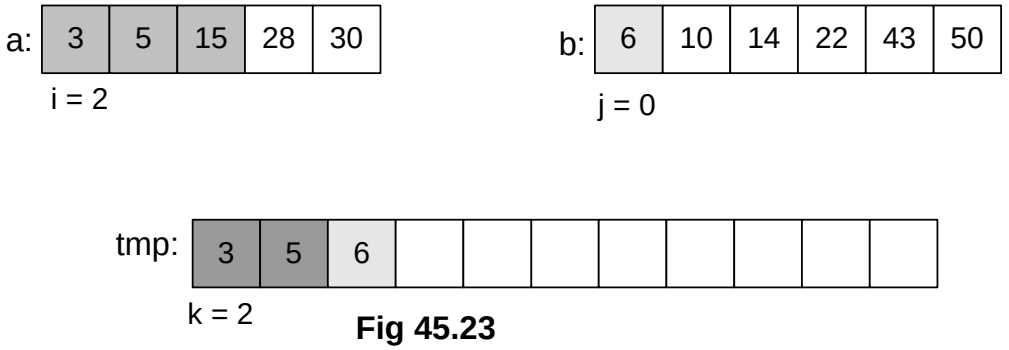


Fig 45.23

Because this time the element was taken from right array with index *j*, therefore, this time index *j* will be incremented. Keep an eye on the index *k* that is increasing after every iteration. So this time, the comparison is made between 15 and 10. 10 being smaller will take the place in *tmp* array as shown in Fig 45.24.

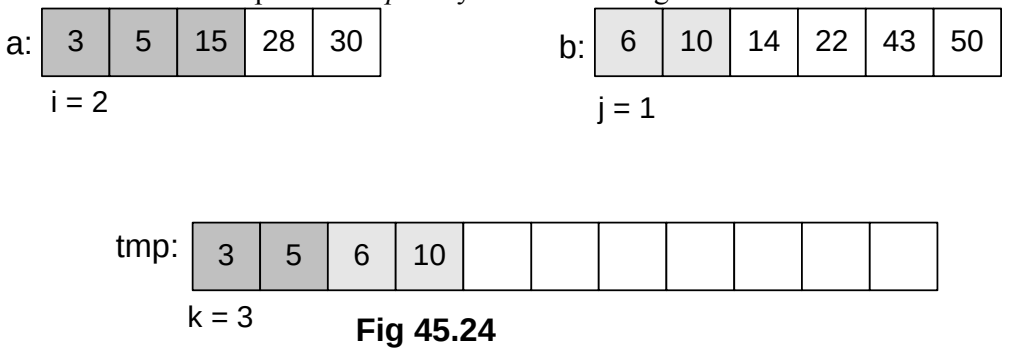


Fig 45.24

Again, *j* will be incremented because last element was chosen from this array. Elements 15 and 14 are compared. Because 14 smaller, so it is moved to *tmp* array in the next position.

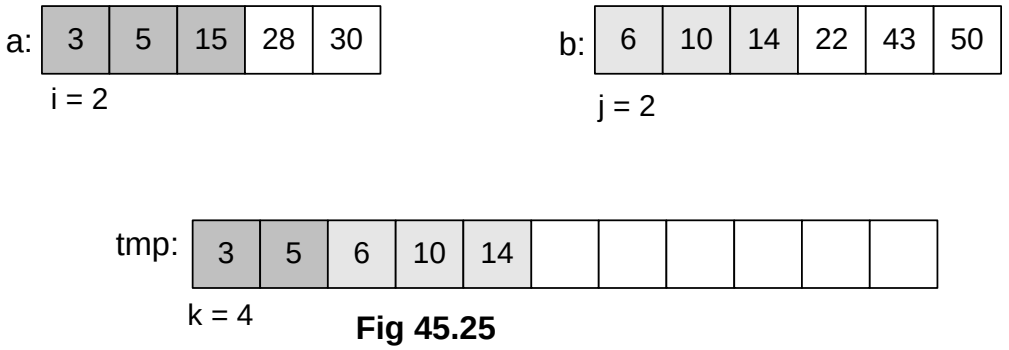


Fig 45.25

With these movements, the value of *k* is increasing after each iteration. Next, 22 is compared with 15, 15 being smaller of two is put into the *tmp* array as shown in Fig 45.26.

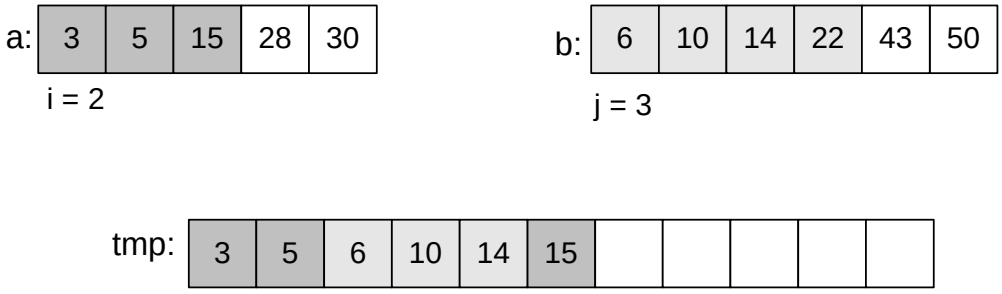


Fig 45.26

By now , you must have understood how this is working, let's see the next iterations pictures below:

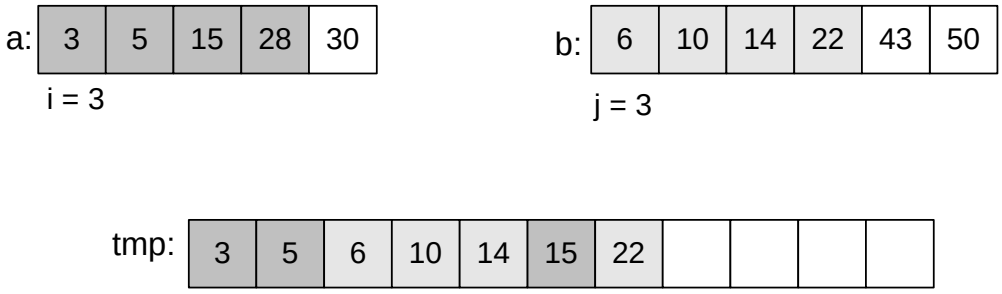


Fig 45.27

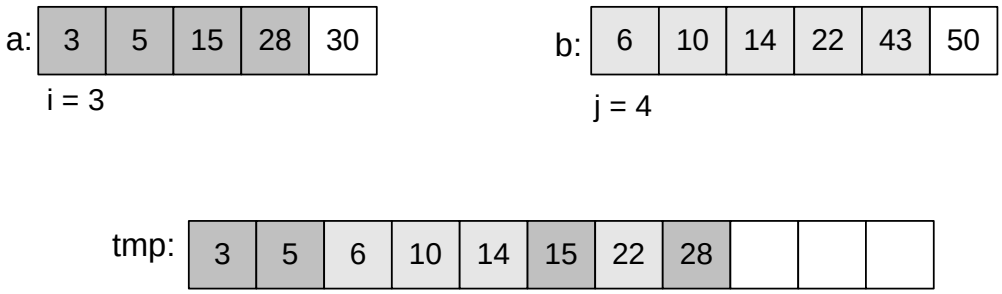


Fig 45.28

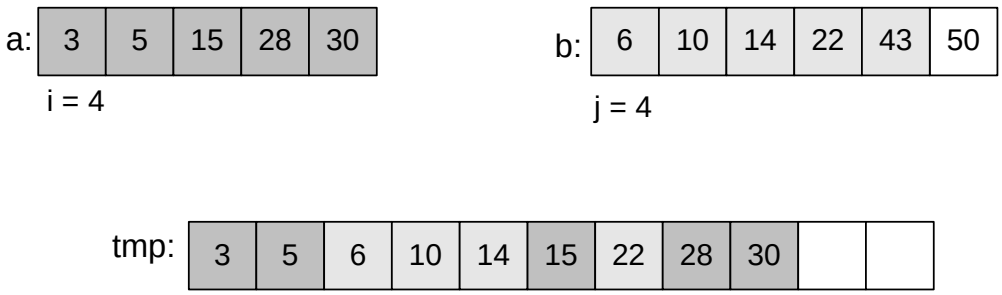


Fig 45.29

Note that array *a* is all used up now, so the comparison operation will end here. All the remaining elements of array *b* are incorporated in *tmp* array straightaway as

shown in Fig 45.30. Hence, we have the resultant merged array *tmp* as shown in the figure.

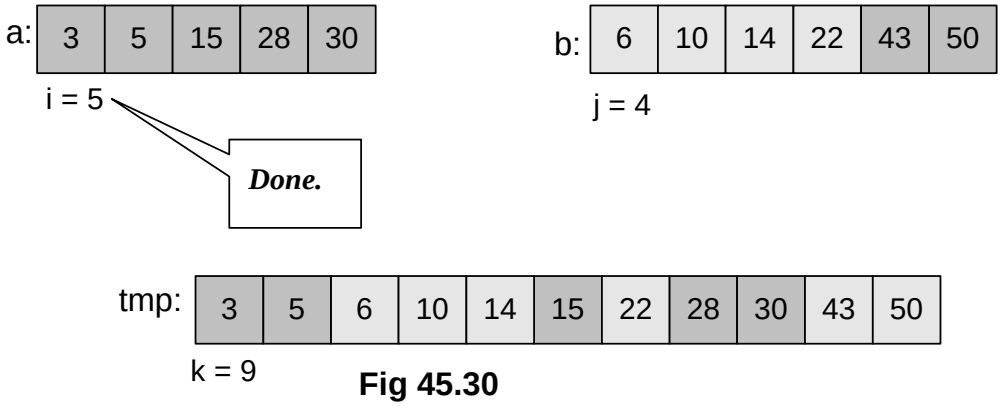


Fig 45.30

Mergesort and Linked Lists

Merge sort works with arrays as well as linked lists. Now we see, how a linked list is sorted. Suppose we have a singly linked list as shown in figure Fig 45.31. We can divide the list into two halves as we are aware of the size of the linked list. Each half is processed recursively for sorting. Both of the sorted resultant halves are merged together.

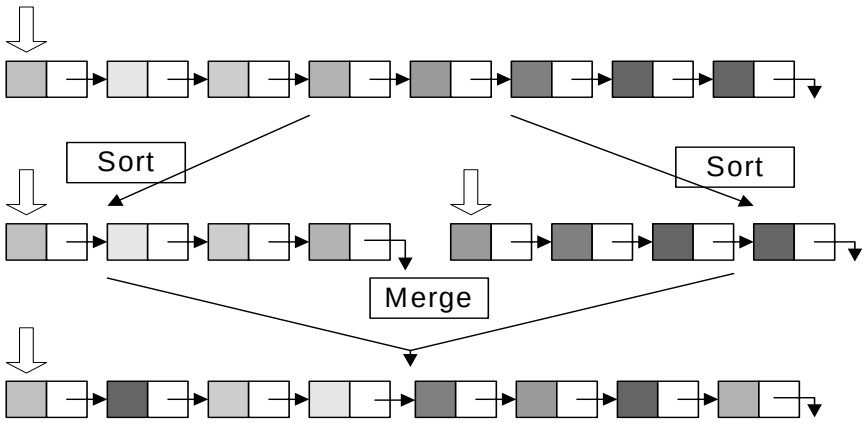


Fig 45.31

Mergesort Analysis

As you have seen, we used an additional temporary array while performing the merging operation. At the end of the merging operation, elements are copied from temporary array to the original array. The merge sort algorithm is not an inplace sorting algorithm because it requires an additional temporary array of the same size as the size of the array under process for sorting. This algorithm is still a good sorting algorithm, we see this fact from its analysis.

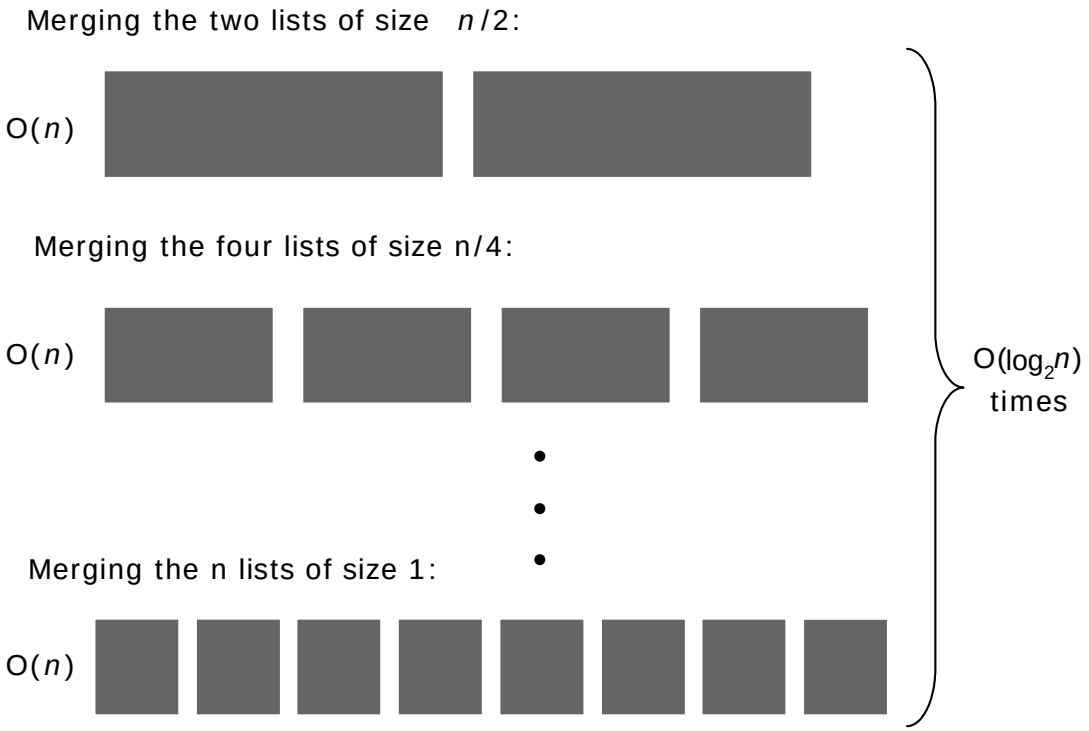


Fig 45.32

- *Mergesort is O(n log₂ n)*
- *Space?*
- *The other sorts we have looked at (insertion, selection) are* ***in-place*** *(only require a constant amount of extra space)*
- *Mergesort requires O(n) extra space for merging*

As shown in Fig 45.32, the array has been divided into two halves. We know that merging operation time is proportional to *n*, as it is done in a single loop regardless of the number of equal parts of the original array. We also know that this dividing the array into halves is similar mechanism as we do in a binary tree and a complete or perfect balance tree has $log_2 n$ number of levels. Because of these two factors, the merge sort algorithm is called $nlog_2 n$ algorithm. Now, let's discuss about quick sort algorithm, which is not only an $nlog_2 n$ algorithm but also an inplace algorithm. As you might have guessed, we don't need an additional array while using this algorithm.

Quicksort

- ***Quicksort*** *is another divide and conquer algorithm.*
- *Quicksort is based on the idea of partitioning (splitting) the list around a pivot or split value.*

Quicksort is also a divide and conquer algorithm. We see pictorially, how the quick sort algorithm works. Suppose we have an array as shown in the figure Fig 45.33.

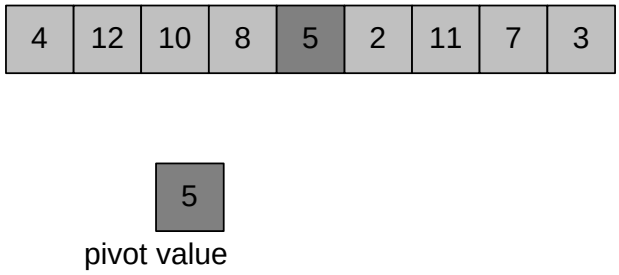


Fig 45.33

We select an element from the array and call it the *pivot*. In this array, the *pivot* is the middle element 5 of the array. Now, we swap this with the last element 3 of the array. The updated figure of the array is shown in Fig 45.34.

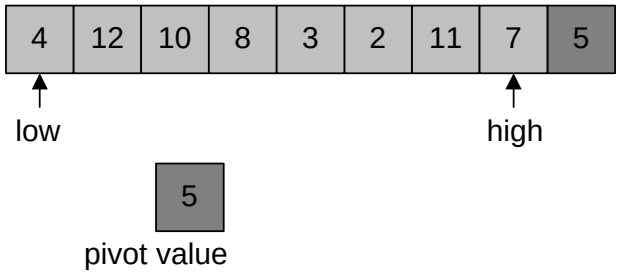


Fig 45.34

As shown in Fig 45.34, we used two indexes *low* and *high*. The index *low* is started from 0^{th} position of the array and goes towards right until $n\text{-}1^{th}$ position. Inside this loop, an element that is bigger than the *pivot* is searched. The *low* index is incremented further as 4 is less than 5.

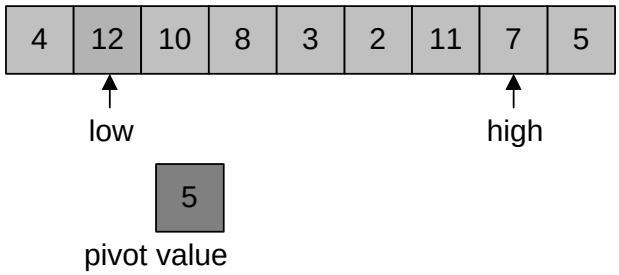


Fig 45.35

low is pointing to element 12 and it is stopped here as 12 is greater than 5. Now, we start from the other end, the *high* index is moved towards left from $n\text{-}1^{th}$ position to 0.

While coming from right to left, we search such an element that is smaller than 5. Elements 7 and 11 towards left are greater than 5, therefore, the *high* pointer is advanced further towards left. *high* index is stopped at the next position as next element 2 is smaller than 5. Following figure Fig 45.36 depicts the latest situation.

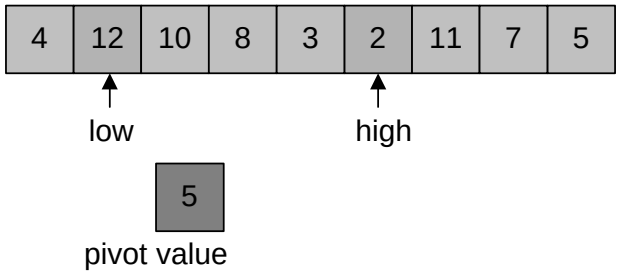


Fig 45.36

Both of the indexes have been stopped, *low* is stopped at a number 12 that is greater than the *pivot* and *high* is stopped at number 2 that is smaller than the *pivot*. In the next step, we swap both of these elements as shown in Fig 45.37.

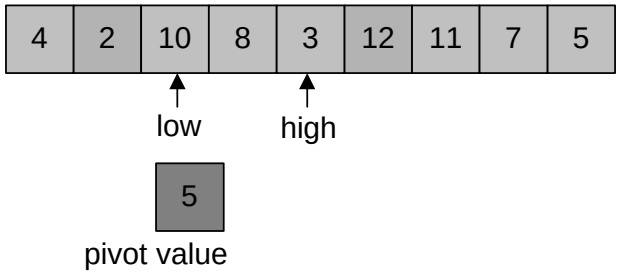


Fig 45.37

Note that our *pivot* element 5 is still there at its original position. We again go to index *low* and start moving towards right, trying to find a number that is greater than the *pivot* element 5. It immediately finds the next number 10 greater than 5. Similarly, the *high* is moved towards left in search to find an element smaller than the *pivot* element 5. The very next element 3 is smaller than 5, therefore, the *high* index stops here. These elements 10 and 3 are swapped, the latest situation is shown in Fig 45.38.

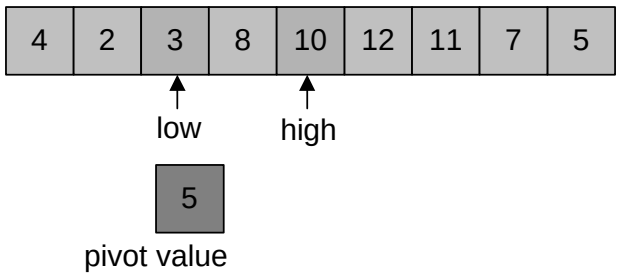


Fig 45.38

Now, in the next iteration both *low* and *high* indexes cross each other. When the *high* pointer crosses the *low* pointer, we stop it moving further as shown in Fig 45.39 and

swap the element at the crossing position (which is 8) with the *pivot* number as shown in Fig 45.40.

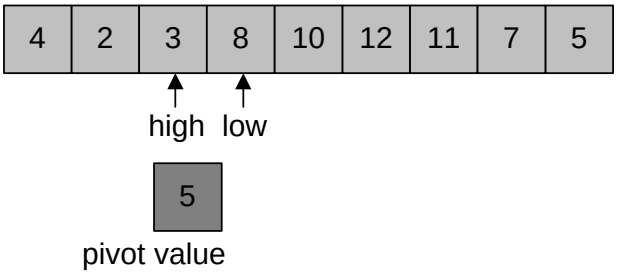


Fig 45.39

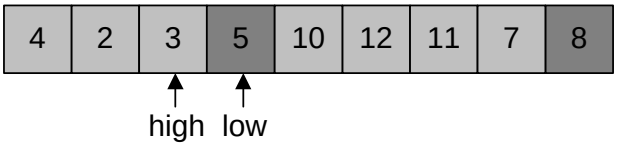


Fig 45.40

This array is not sorted yet but element 5 has found its destination. The numbers on the left of 5 should be smaller than 5 and on right should be greater than it and we can see in Fig 45.40 that this actually is the case here. Notice that smaller numbers are on left and greater numbers are on right of 5 but they are not sorted internally.
Next, we recursively quick sort the left and right parts to get the whole array sorted.

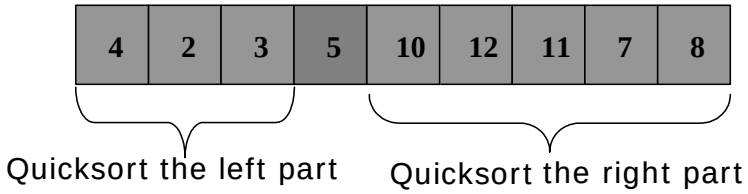


Fig 45.41

Now, we see the C++ code of quick sort.

```
void quickSort(int array[], int size)
{
  int index;
  if (size > 1)
  {
    index = partition(array, size);
    quickSort(array, index);
    quickSort(array+index+1, size - index-1);
  }
}
int partition(int array[], int size)
{
```

```
  int k;
  int mid = size/2;
  int index = 0;
  swap(array, array+mid);
  for (k = 1; k < size; k++){
     if (array[k] < array[0]){
        index++;
        swap(array+k, array+index);
     }
  }
  swap(array, array+index);
  return index;
}
```

An array and its size are passed as arguments to the *quickSort* function. The function declared a local variable *index* and the size of the array is checked in the next statement. If the size of the array is more than 1 then the function does the recursive calling mechanism to sort the array. It divides the array into two parts by choosing the pivot element. In the subsequent calls, firstly the left side is sorted and then the right side using the recursive mechanism. Quicksort algorithm is very elegant and it can sort an array of any size efficiently. It is considered one of the best general purpose algorithms for sorting. This normally is preferred sorting method being $nlog_2n$ and inplace algorithm. You are advised to read more about this algorithm from your text books and try to do it as an exercise.

Today's lecture being the last lecture of the course, let's have a short review of it.

Course Overview

We had started this course while keeping the objectives of data structures in mind that appropriate data structures are applied in different applications in order to make them work efficiently. Secondly, the applications use data structures that are not memory hungry. In the initial stages, we discussed *array* data structure. After we found one significant drawback of arrays; their fixed size, we switched our focus to *linked list* and different other data structures. However, in the meantime, we started realizing the significance of algorithms; without them data structures are not really useful rather I should say complete.

We also studied *stack* and *queue* data structures. We implemented them with *array* and *linked list* data structures. With their help, we wrote such applications which seemed difficult apparently. I hope, by now, you must have understood the role of stack in computer's runtime environment. Also *Queue* data structure was found very helpful in Simulations.

Later on, we also came across the situations when we started thinking that linear data structures had to be tailored in order to achieve our goals with our work, we started studying trees then. Binary tree was found specifically very useful. It also had a degenerate example, when we constructed AVL trees in order to balance the binary search tree. We also formed threaded binary trees. We studied union/find data structure, which was an up tree. At that time, we had already started putting special importance on algorithms. We found one important fact that it is not necessary that for every application we should use a new data structure. We formed new Abstract Data Types (ADT) using the existing data structures. For dictionary or table data

structure, we majorly worked with ADTs when we implemented them in six different ways. We also studied Skip List within the topic of Table ADT, which is a very recent data structure. After that we discussed about Hashing. Hashing was a purely algorithmic procedure and there was nothing much as a data structure.

In future, you will more realize the importance of algorithm. While solving your problems, you will choose an algorithm to solve your problem and that algorithm will bring along some data structure along. Actually, data structure becomes a companion to an algorithm. For example, in order to build your symbol table while constructing your own compiler, you will use hashing. For searches, trees will be employed.

One important fact here is that the data structures and algorithms covered in this course are not complete in the sense that you don't need any other data structure except them. One example is Graph, which is not discussed much in this course. Graph data structures are primarily important from algorithmic point of view.

Now, you should examine yourself, what have you learned in this course. As a software engineer, you have learned data structures and algorithmic skills to increase your domain knowledge of design choices. You can apply these design choices in order to resolve different design problems of your applications that you will come across in your student and professional life.