



# Learning & Growing With Team

## Help & Testing Services Any Field

### CS301-Data Structures

### PPT Slide Merged By LGWT

### Final Term Lecture 19 - 45

You Can Discuss Your Problems in these groups:

Whatsapp Study Groups: [Handouts & Files](#) [Study Group](#) [Community Link](#)

If Any Link Have Been Not Working then you can direct contact me on whatsapp

<https://wa.me/923196248868?text=Asslamualikum%21%20Can%20You%20Send%20me%20Group%20Link>

**Paid Services Available:**

- + Assignments
- + Quiz
- + GDB
- + Mid & Final Exam Preparation

**[CLICK ME FOR PAID SERVICES](#)**

For More Notes & Files Join Our Whatsapp Group <https://chat.whatsapp.com/CpYyqhErHGX3E8PJ9Aj2XR>

# The `const` Keyword

- The answer is that, yes, we don't want the function to change the parameter, but neither do we want to use up time and memory creating and storing an entire copy of it.
- So, we make the original object available to the called function by using pass-by-reference.
- We also mark it constant so that the function will not alter it, even by mistake.

Lecture No.19

Data Structure

Dr. Sohail Aslam

# The `const` Keyword

- *Use 2:* The `const` keyword appears at the end of class member's function signature:

```
EType& findMin( ) const;
```

- Such a function cannot change or write to member variables of that class.
- This type of usage often appears in functions that are suppose to read and return member variables.



# The `const` Keyword

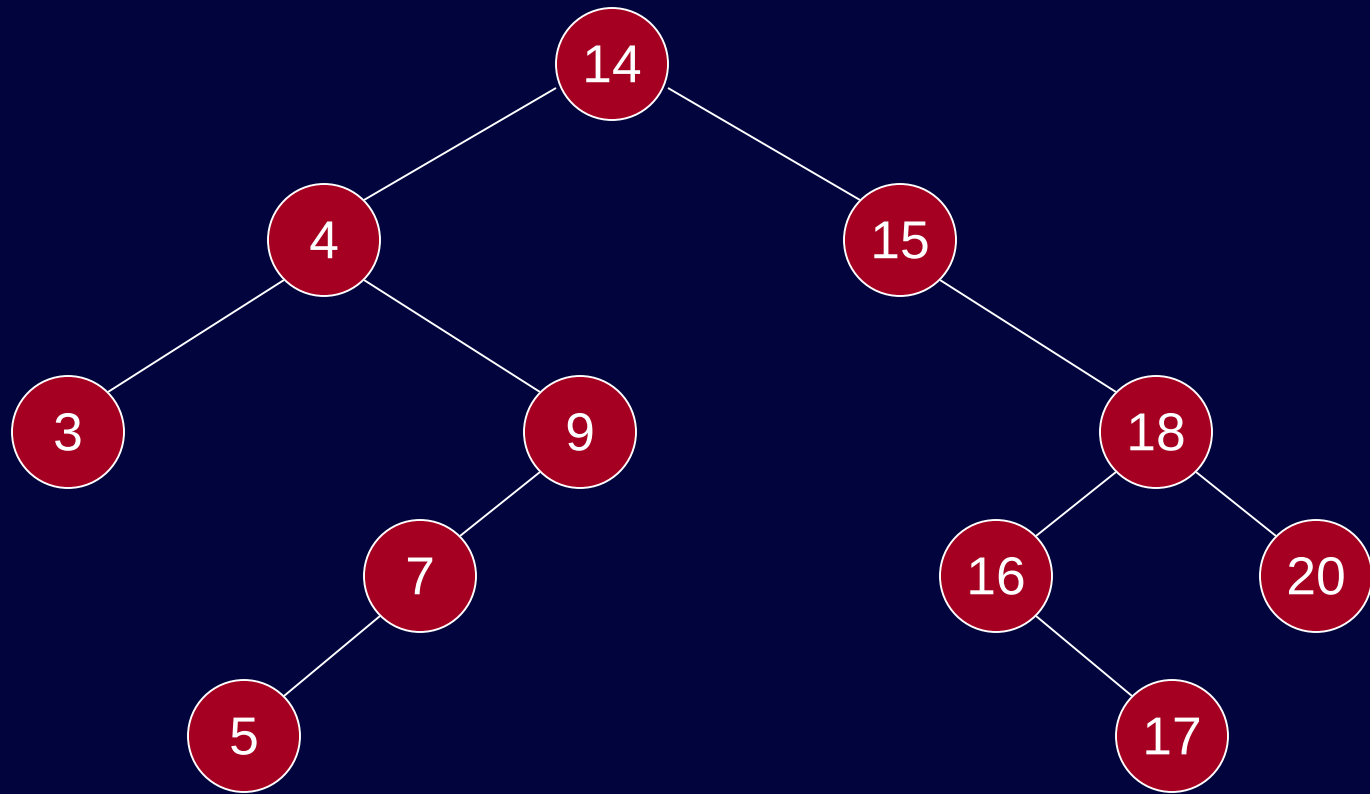
- *Use 3:* The `const` keyword appears at the beginning of the return type in function signature:

```
const EType& findMin( ) const;
```

- Means, whatever is returned is constant.
- The purpose is typically to protect a reference variable.
- This also avoids returning a copy of an object.

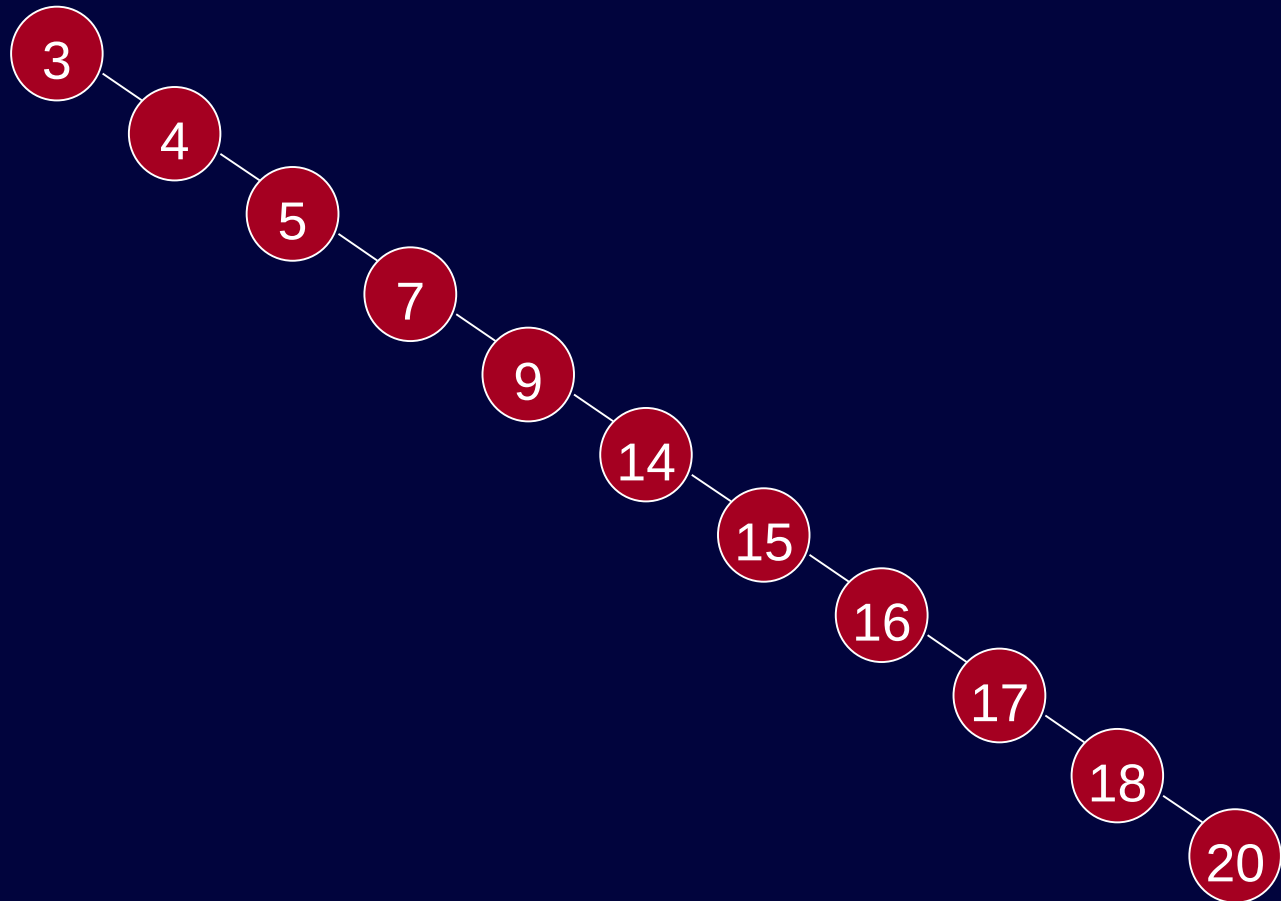
# Degenerate Binary Search Tree

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



# Degenerate Binary Search Tree

BST for 3 4 5 7 9 14 15 16 17 18 20



# Degenerate Binary Search Tree

BST for 3 4 5 7 9 14 15 16 17 18 20

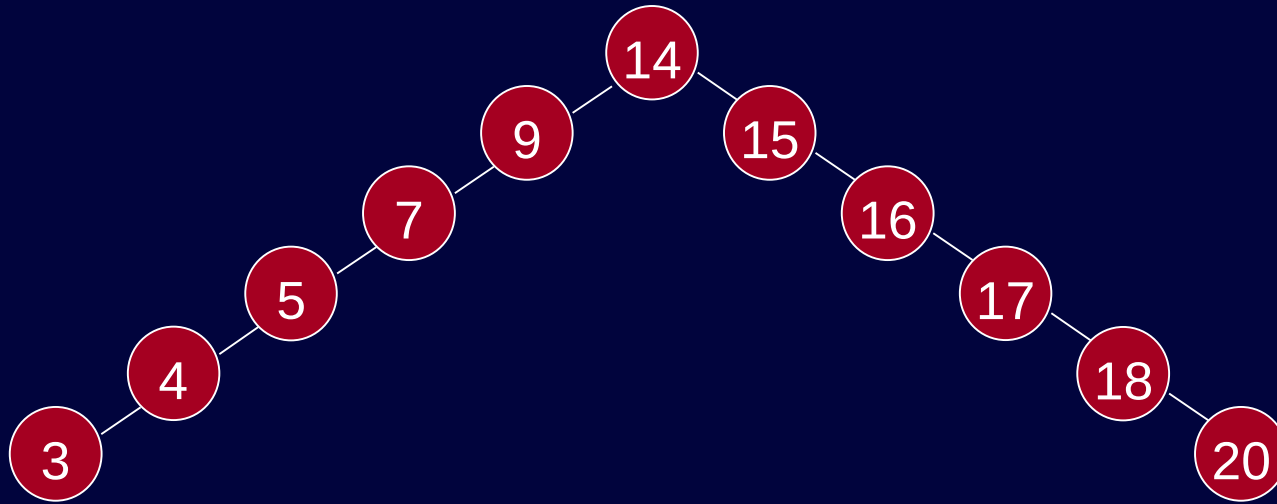


Linked List!

# Balanced BST

- We should keep the tree *balanced*.
- One idea would be to have the left and right subtrees have the same height

# Balanced BST



Does not force the tree to be *shallow*.

# Balanced BST

- 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 do this, there must have  $(2^{d+1} - 1)$  data items, where  $d$  is the depth of the tree.
- This is too rigid a condition.

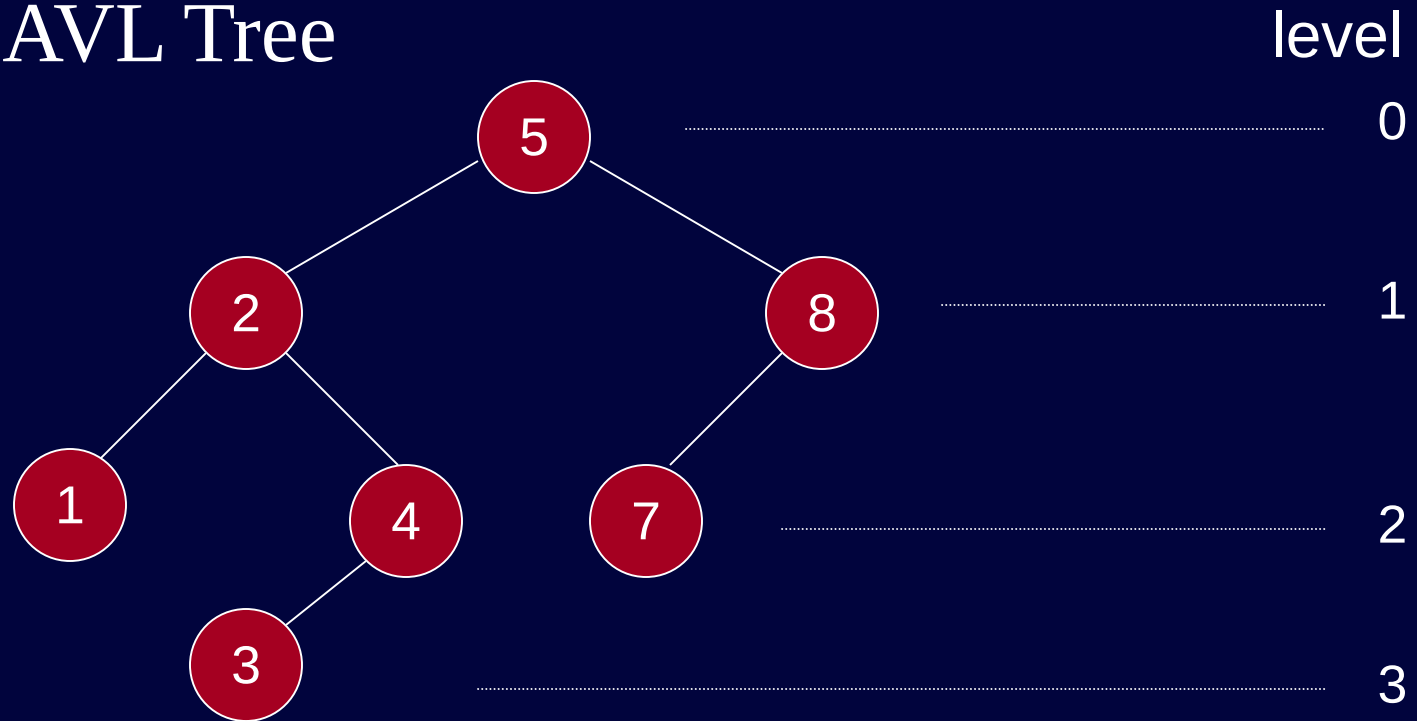
# AVL Tree

- AVL (Adelson-Velskii and Landis) tree.
- An AVL tree is identical to a BST except
  - height of the left and right subtrees can differ by at most 1.
  - height of an empty tree is defined to be  $(-1)$ .



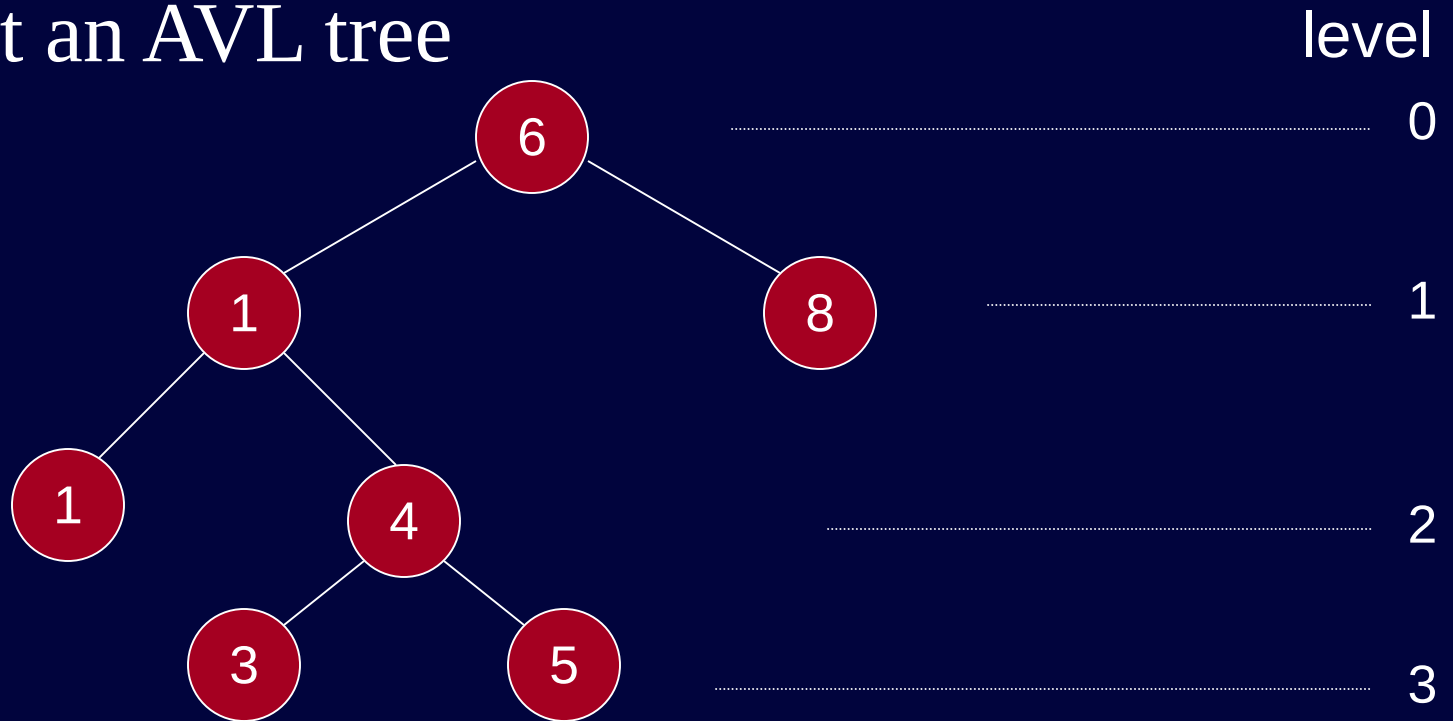
# AVL Tree

- An AVL Tree



# AVL Tree

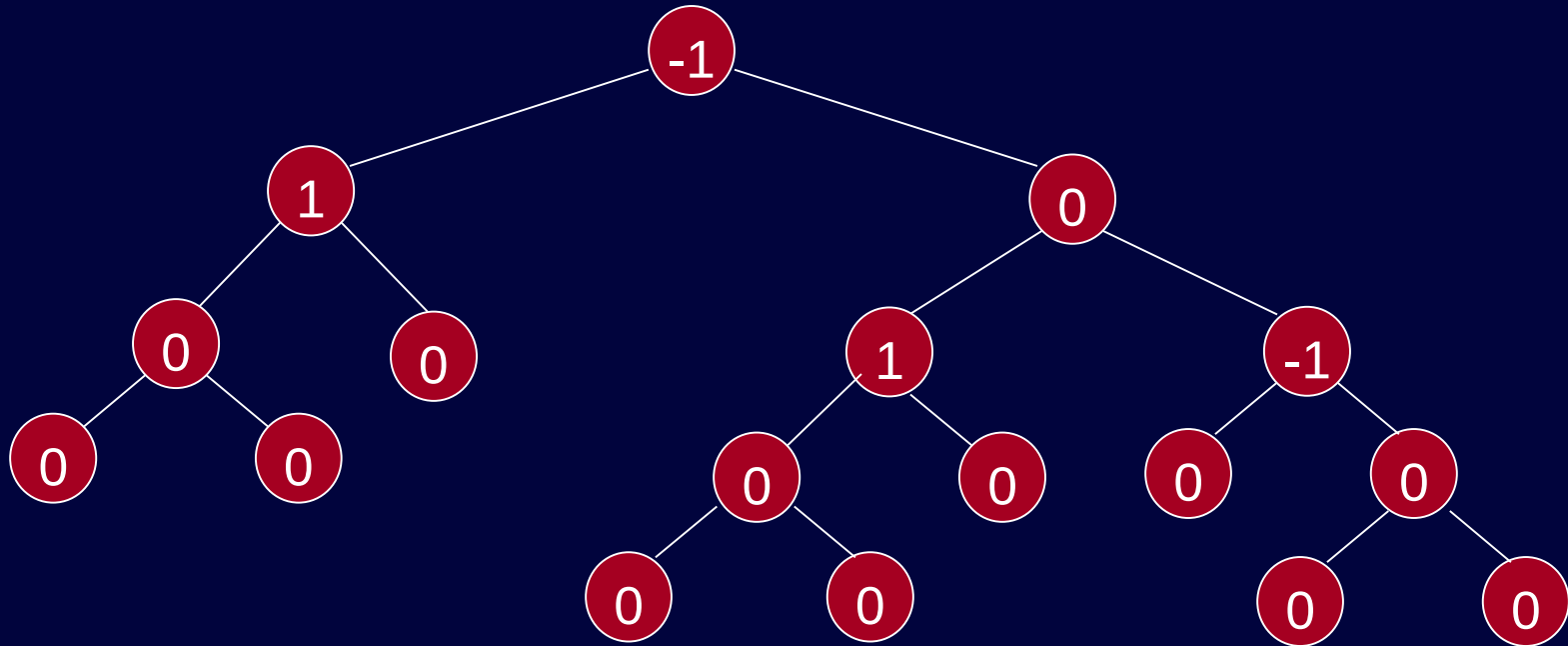
- Not an AVL tree



# Balanced Binary Tree

- The *height* of a binary tree is the maximum level of its leaves (also called the depth).
- 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. Each node has an indicated balance of 1, 0, or  $-1$ .

# Balanced Binary Tree



Lecture No.20

Data Structures

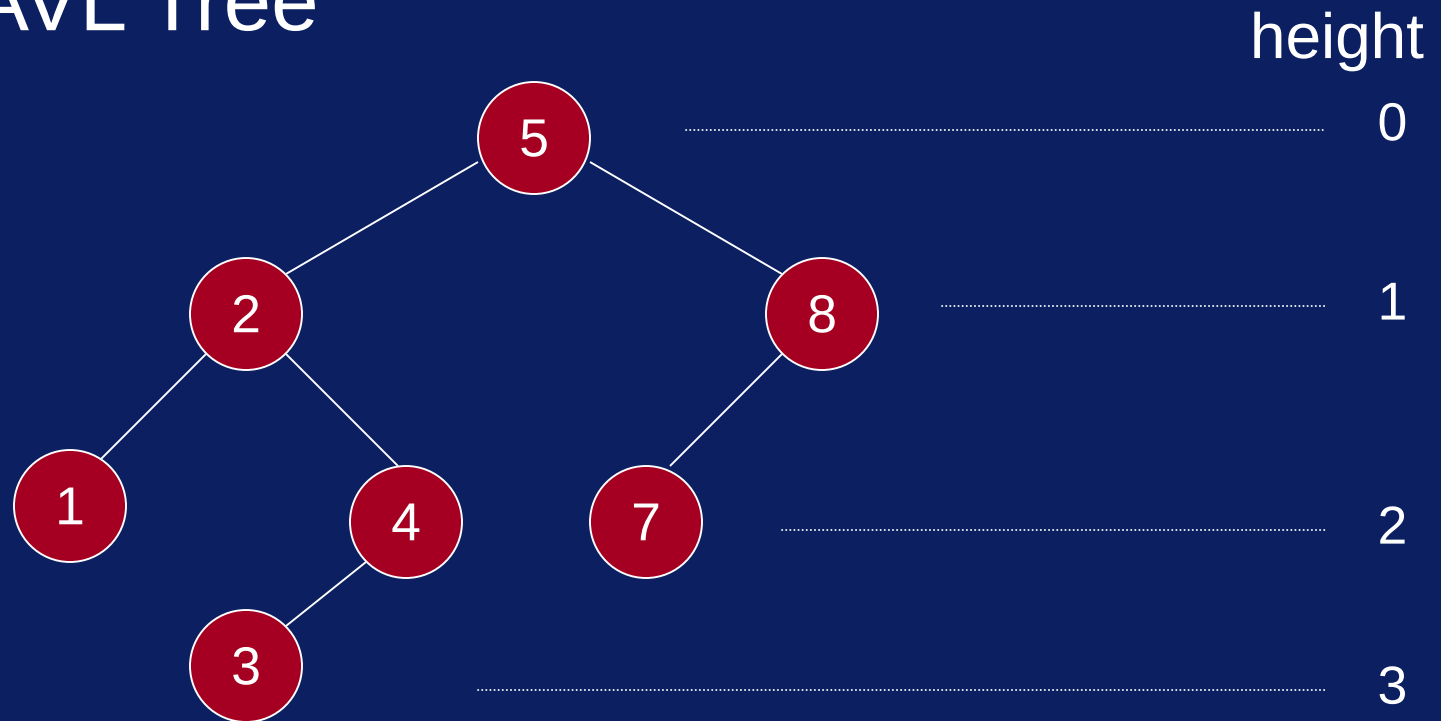
Dr. Sohail Aslam

# AVL Tree

- AVL (Adelson-Velskii and Landis) tree.
- An AVL tree is identical to a BST except
  - height of the left and right subtrees can differ by at most 1.
  - height of an empty tree is defined to be  $(-1)$ .

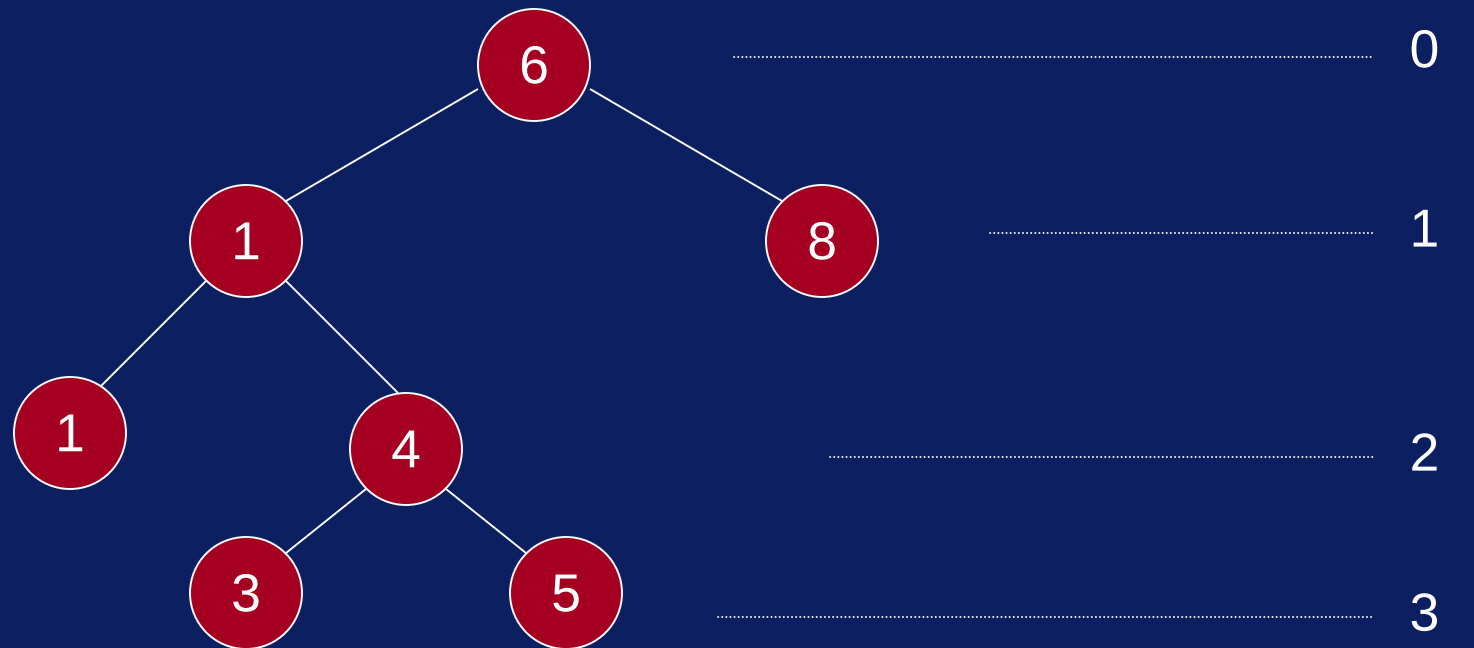
# AVL Tree

- An AVL Tree



# AVL Tree

- Not an AVL tree

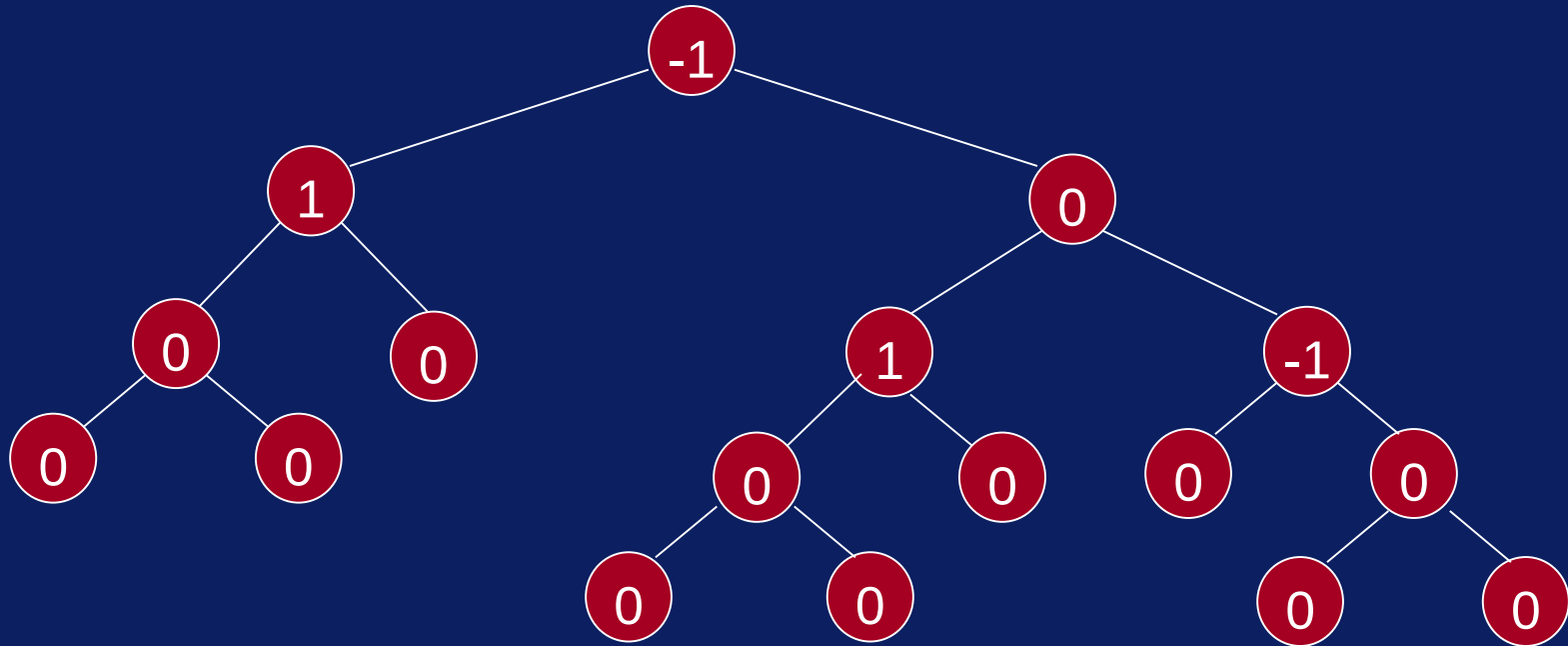




# Balanced Binary Tree

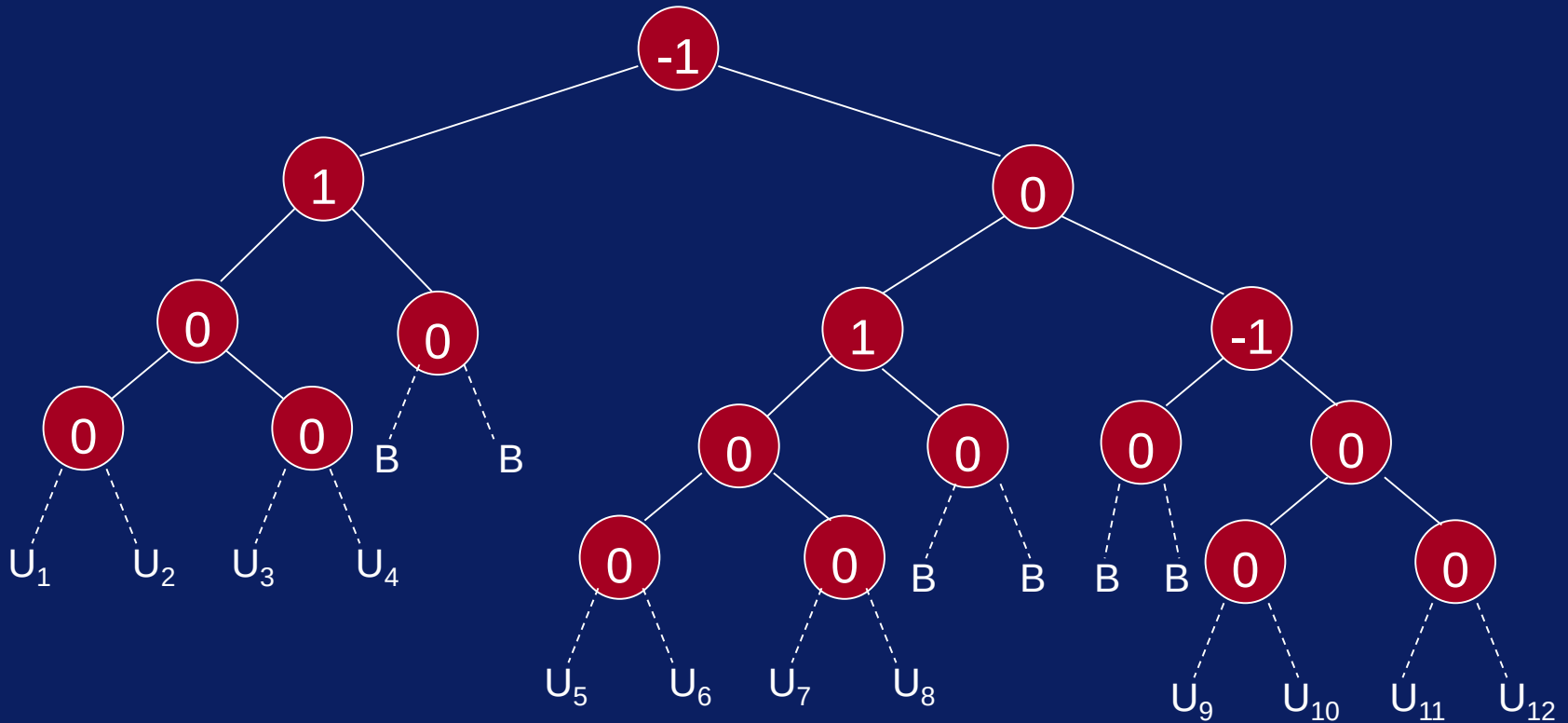
- The *height* of a binary tree is the maximum level of its leaves (also called the depth).
- The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.
- Here, for example, is a balanced tree. Each node has an indicated balance of 1, 0, or  $-1$ .

# Balanced Binary Tree



# Balanced Binary Tree

Insertions and effect on balance

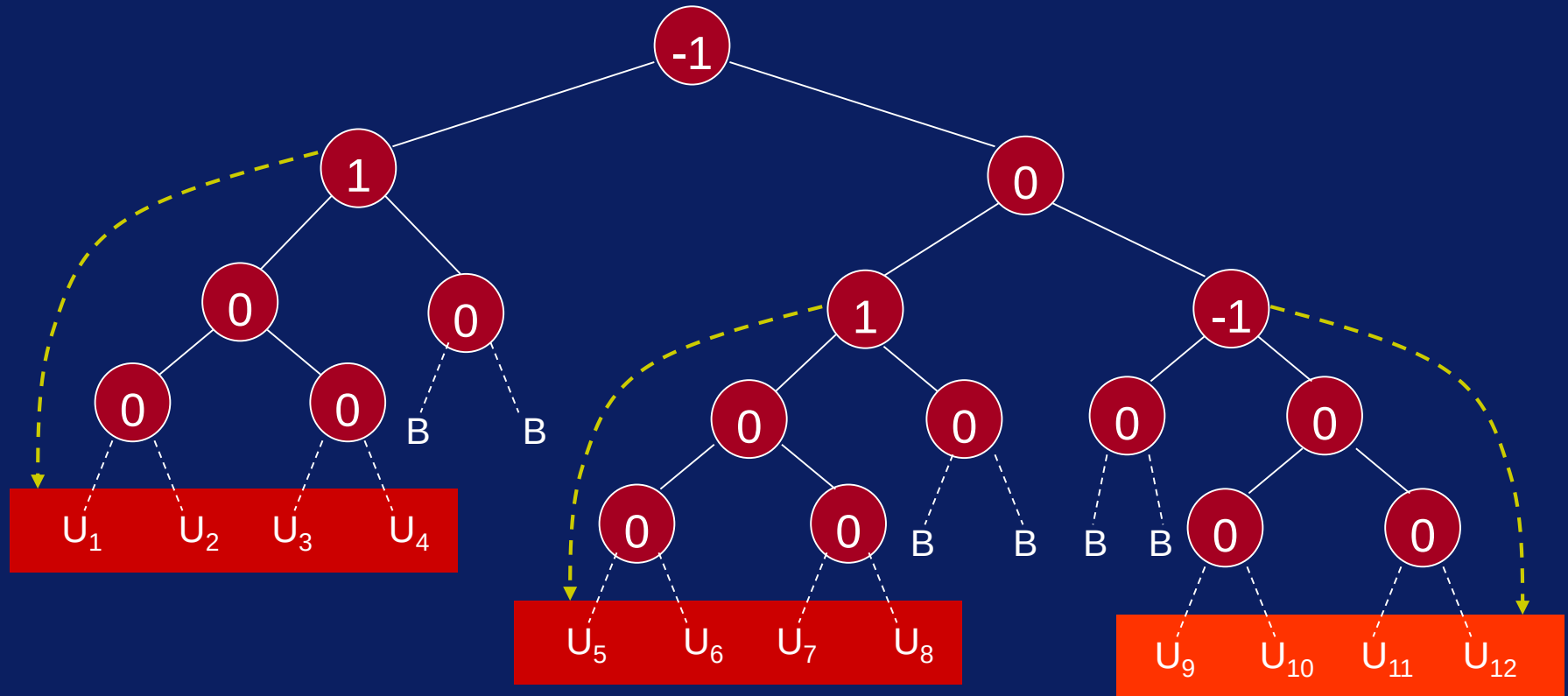


# Balanced Binary Tree

- Tree becomes unbalanced only if the newly inserted node
  - is a left descendant of a node that previously had a balance of 1 ( $U_1$  to  $U_8$ ),
  - or is a descendant of a node that previously had a balance of  $-1$  ( $U_9$  to  $U_{12}$ )

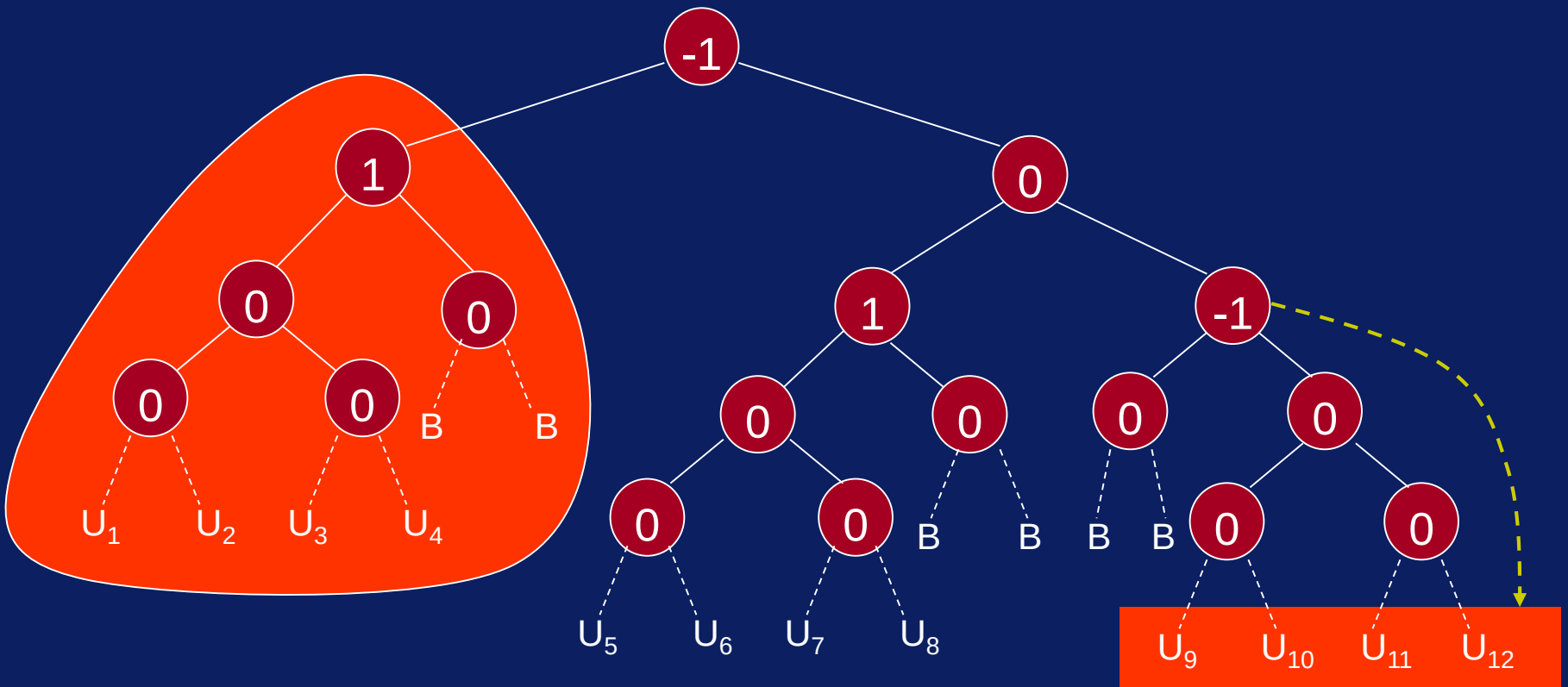
# Balanced Binary Tree

Insertions and effect on balance

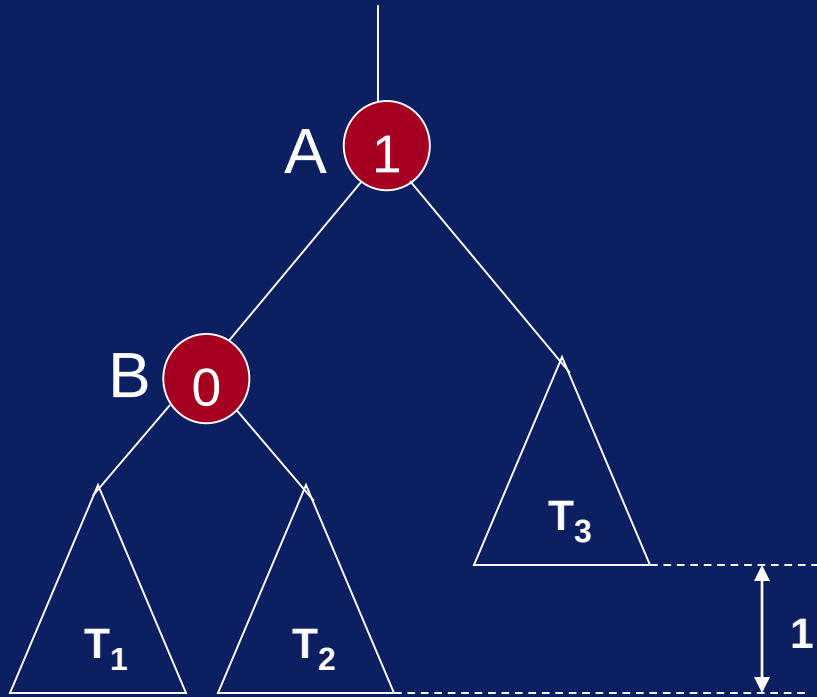


# Balanced Binary Tree

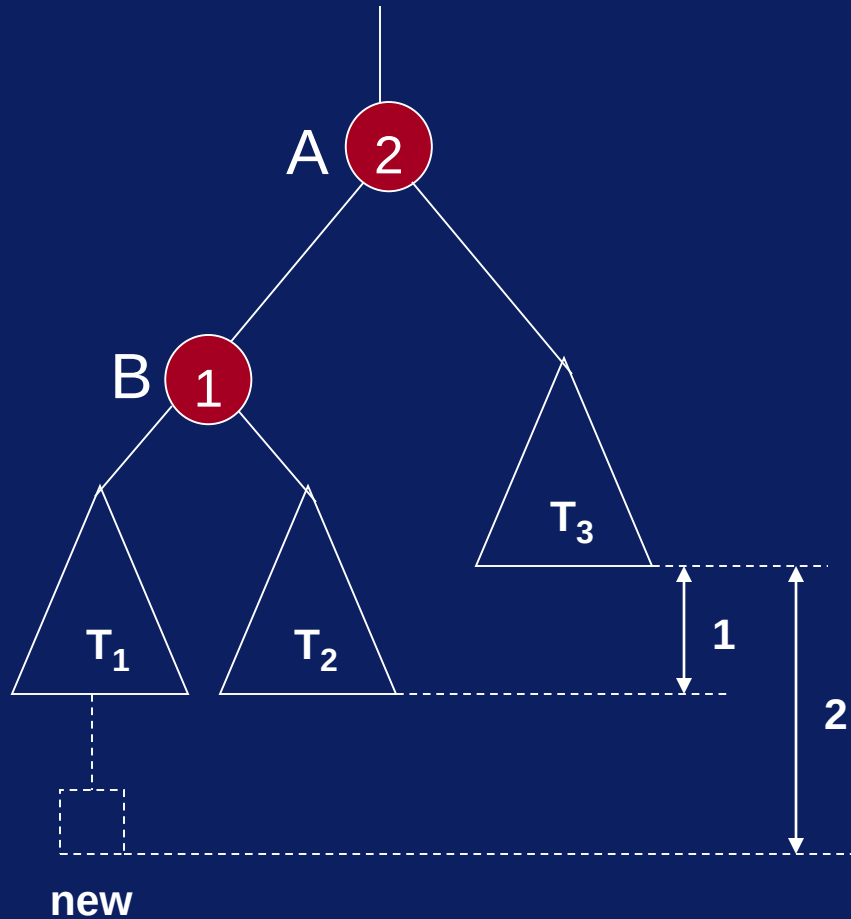
Consider the case of node that was previously 1



# Inserting New Node in AVL Tree

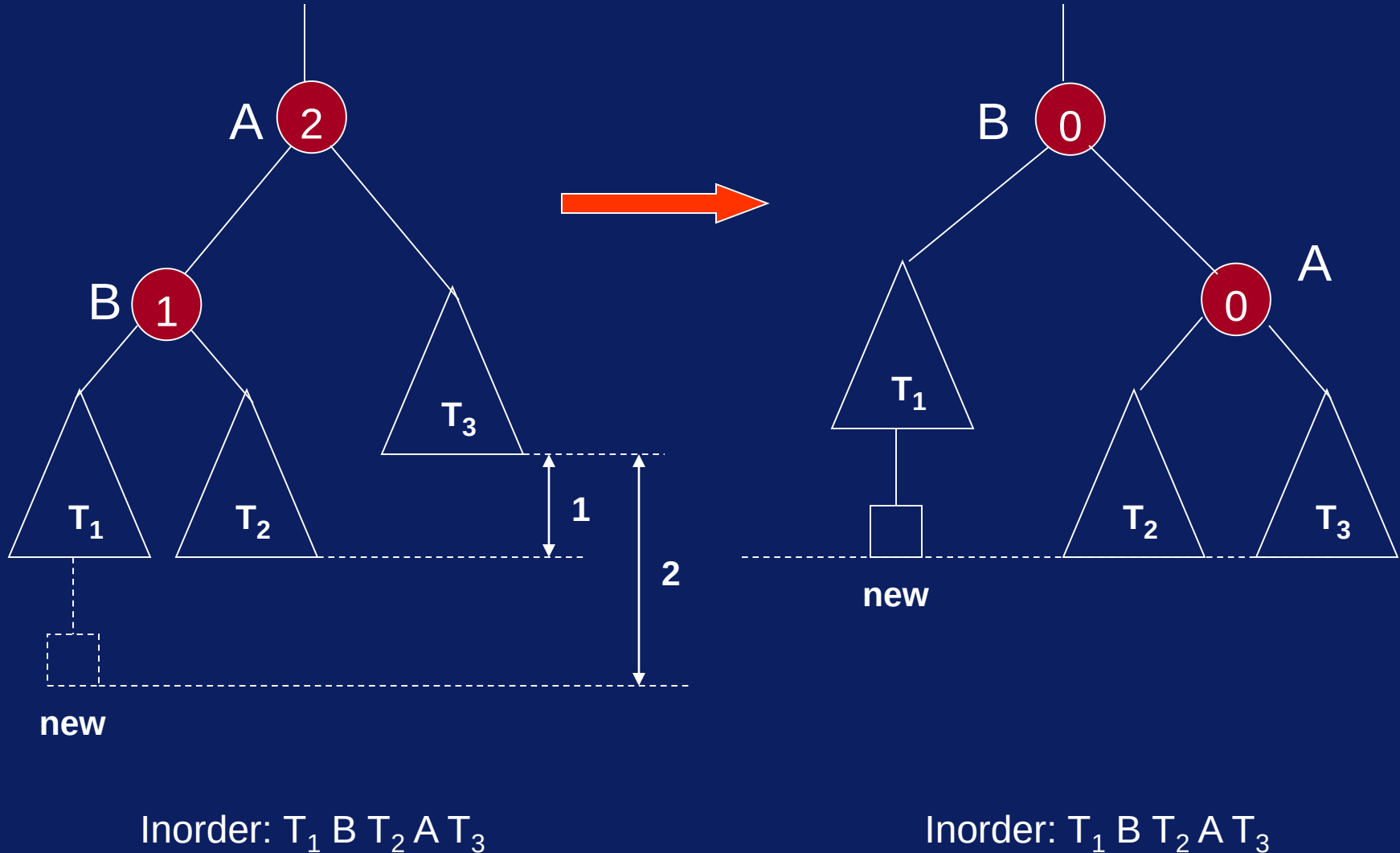


# Inserting New Node in AVL Tree





# Inserting New Node in AVL Tree



# AVL Tree Building Example

- Let us work through an example that inserts numbers in a balanced search tree.
- We will check the balance after each insert and rebalance if necessary using rotations.

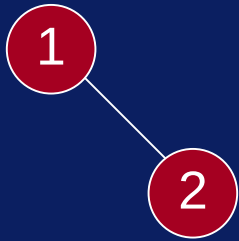
# AVL Tree Building Example

Insert(1)



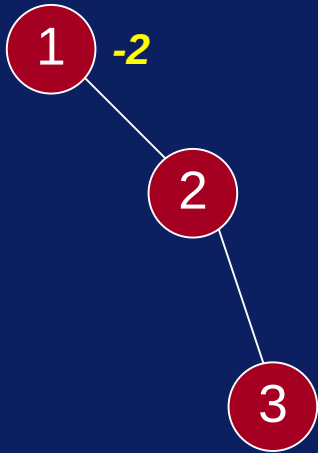
# AVL Tree Building Example

Insert(2)



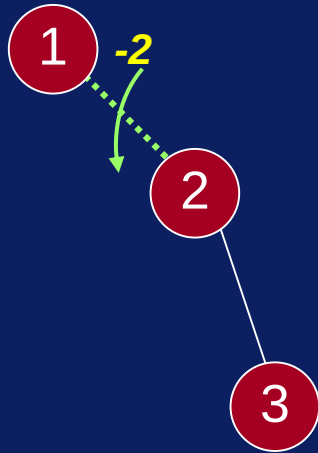
# AVL Tree Building Example

Insert(3) single left rotation



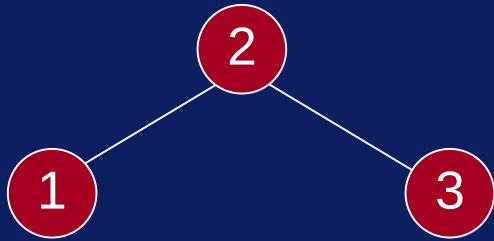
# AVL Tree Building Example

Insert(3) single left rotation



# AVL Tree Building Example

Insert(3)



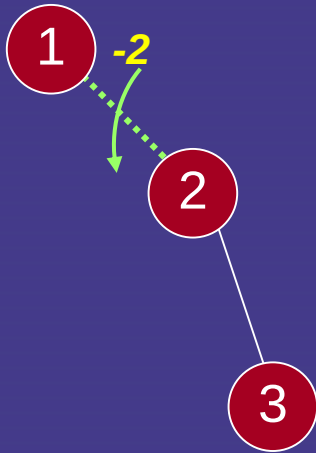
# Data Structures

Lecture 21  
Sohail Aslam



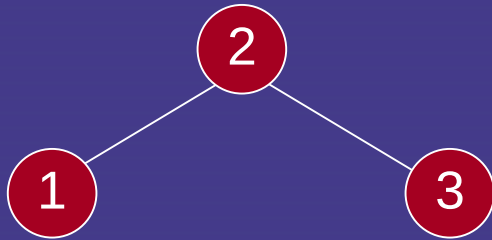
# AVL Tree Building Example

Insert(3) single left rotation



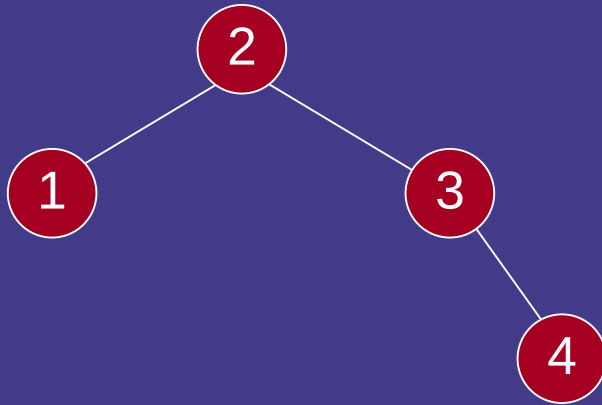
# AVL Tree Building Example

Insert(3)



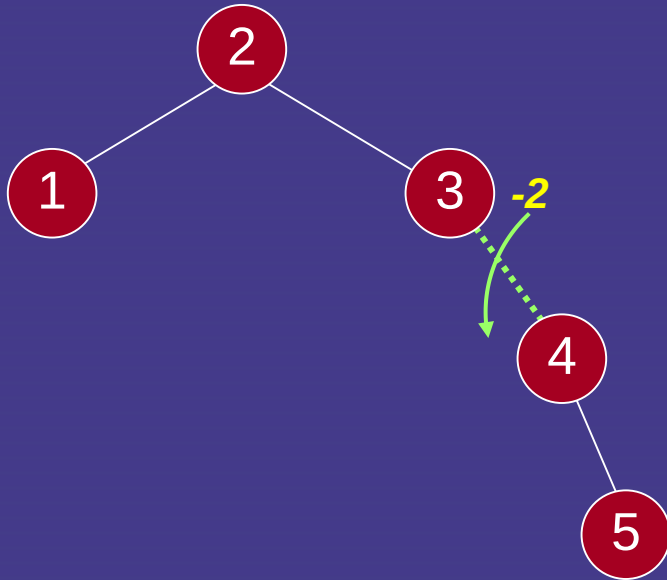
# AVL Tree Building Example

Insert(4)



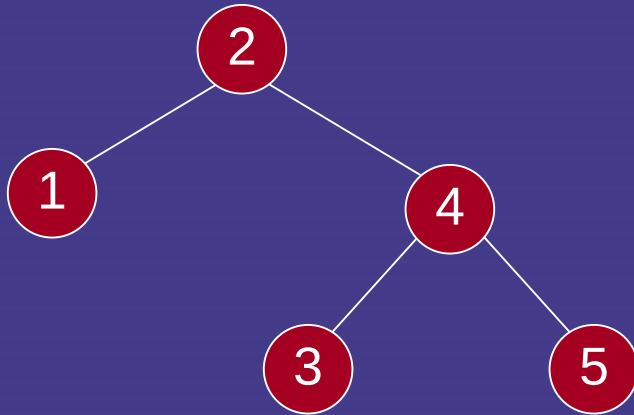
# AVL Tree Building Example

Insert(5)



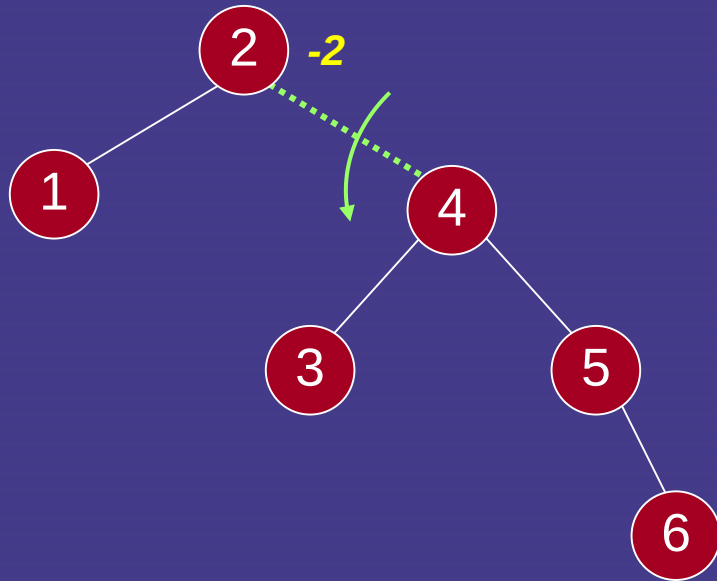
# AVL Tree Building Example

Insert(5)



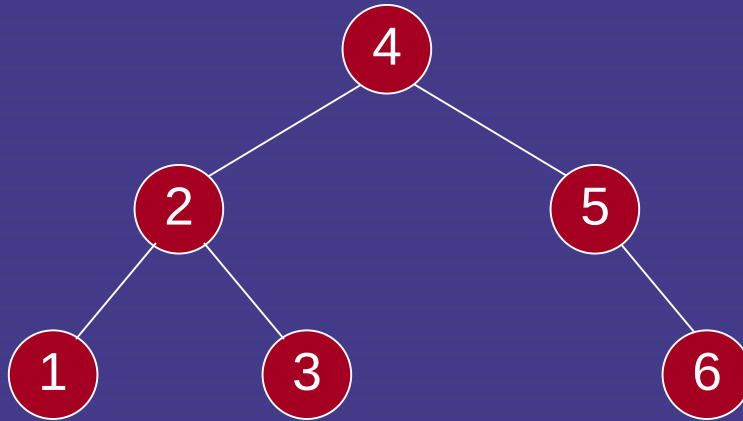
# AVL Tree Building Example

Insert(6)



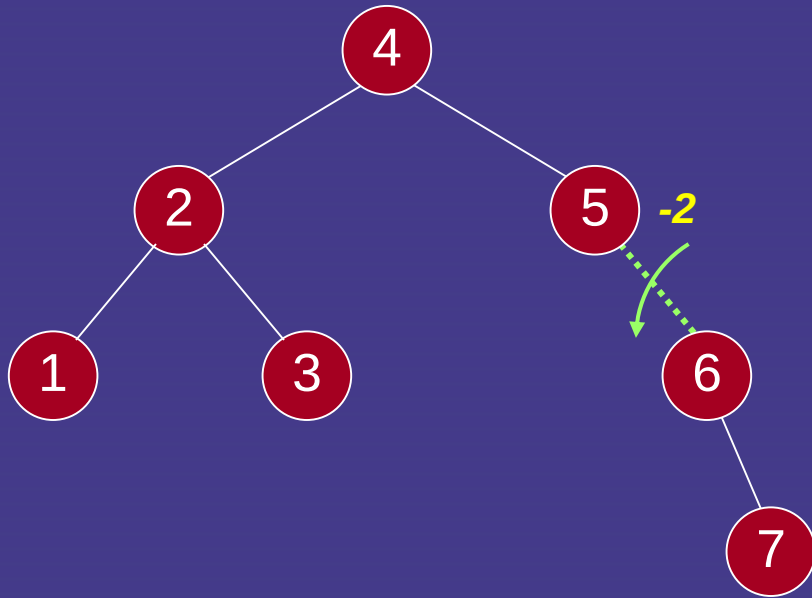
# AVL Tree Building Example

Insert(6)



# AVL Tree Building Example

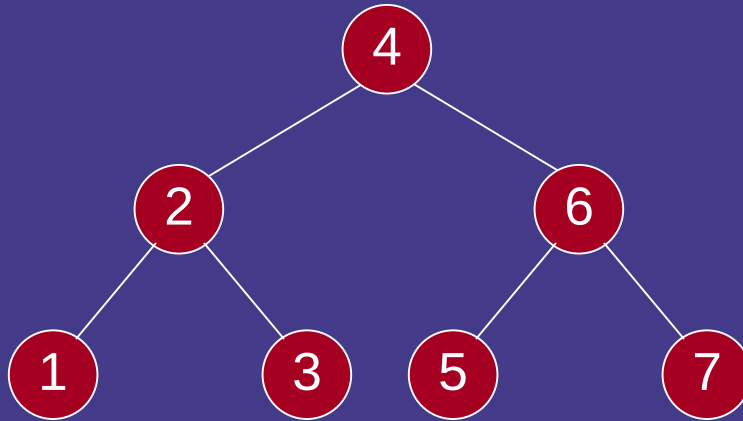
Insert(7)





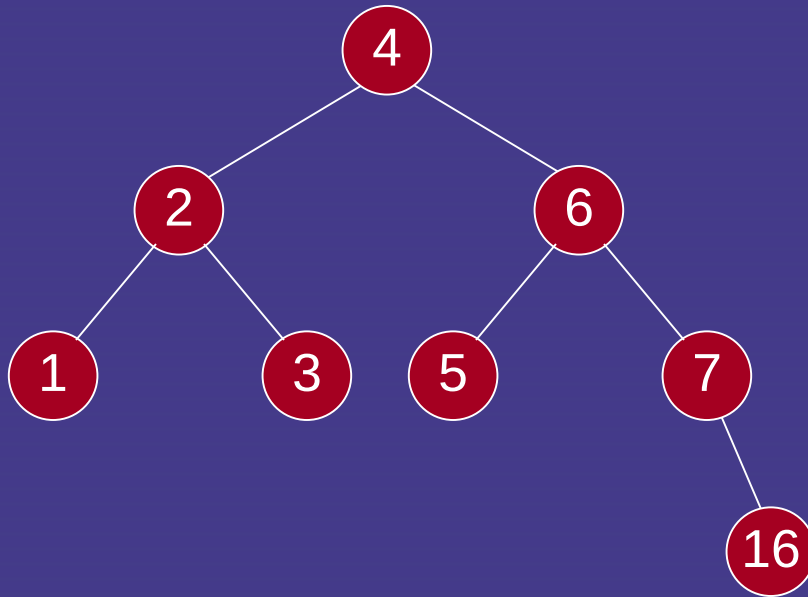
# AVL Tree Building Example

Insert(7)



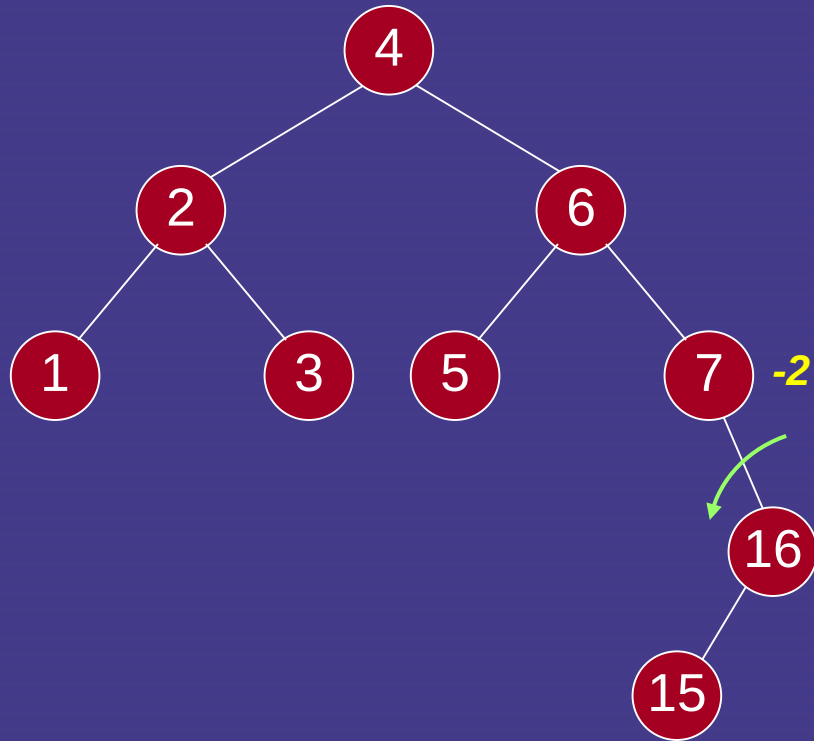
# AVL Tree Building Example

Insert(16)



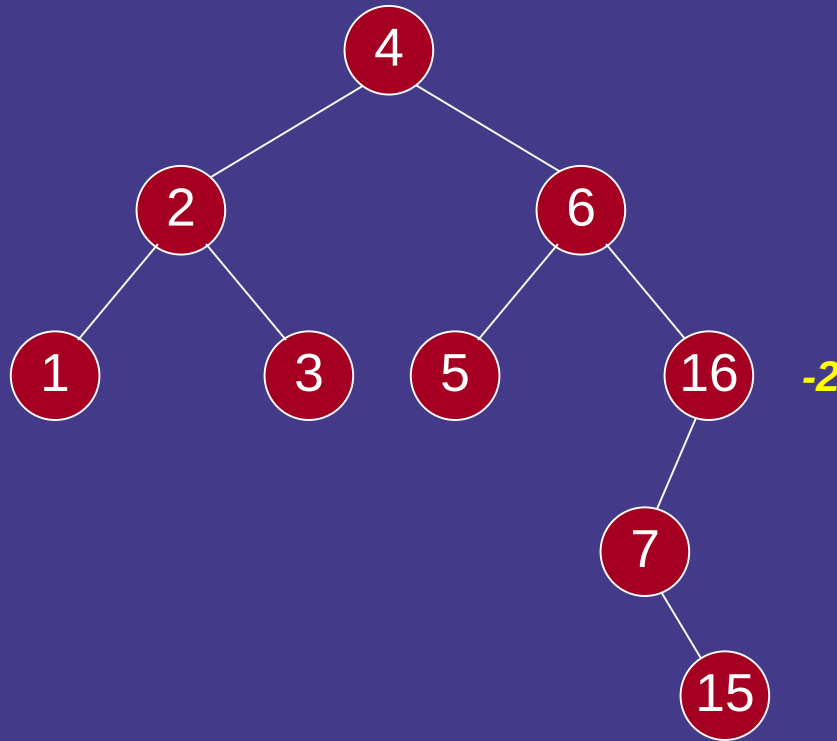
# AVL Tree Building Example

Insert(15)



# AVL Tree Building Example

Insert(15)



# Cases for Rotation

- Single rotation does not seem to restore the balance.
- The problem is the node 15 is in an inner subtree that is too deep.
- Let us revisit the rotations.

# Cases for Rotation

- Let us call the node that must be rebalanced  $\mathcal{Q}$ .
- Since any node has at most two children, and a height imbalance requires that  $\mathcal{Q}$ 's two subtrees differ by two (or  $-2$ ), the violation will occur in four cases:

# Cases for Rotation

1. An insertion into left subtree of the left child of  $\mathfrak{Q}$ .
2. An insertion into right subtree of the left child of  $\mathfrak{Q}$ .
3. An insertion into left subtree of the right child of  $\mathfrak{Q}$ .
4. An insertion into right subtree of the right child of  $\mathfrak{Q}$ .

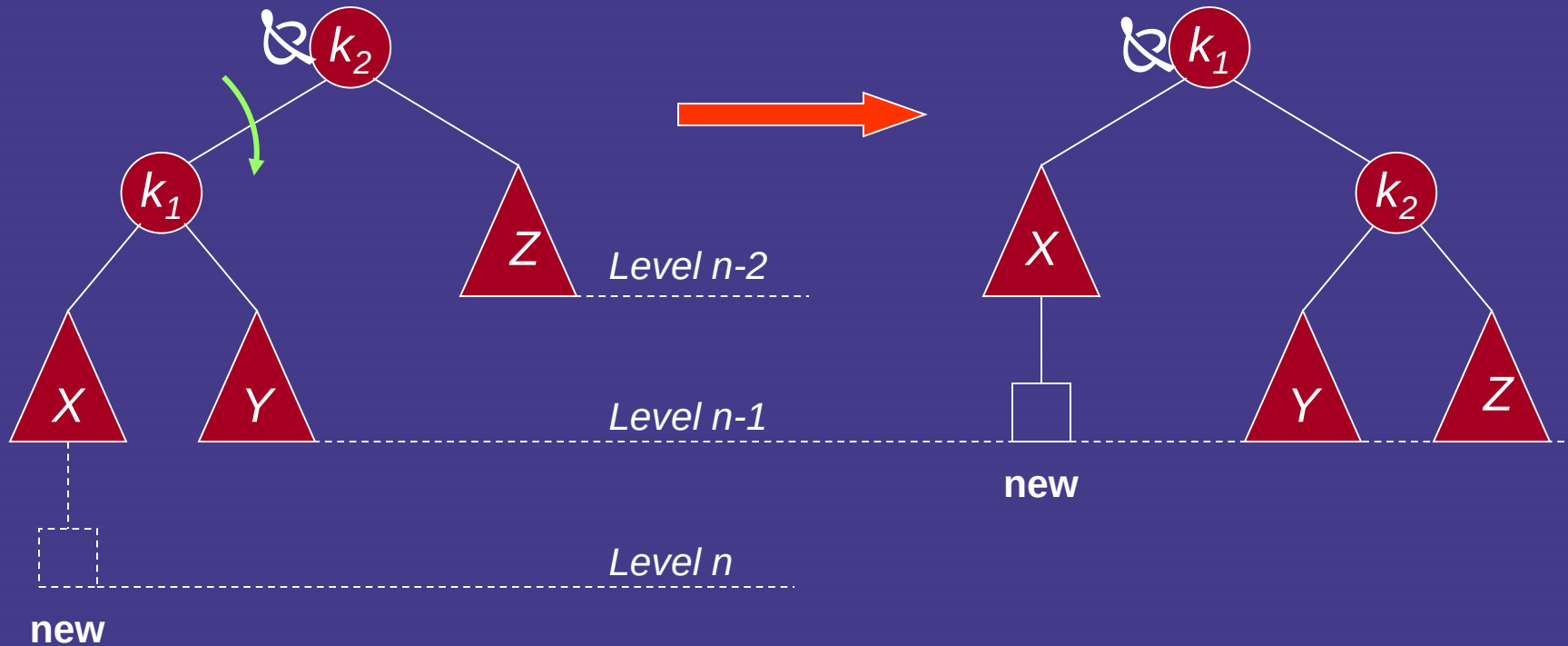
# Cases for Rotation

- 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 single rotation cannot fix.



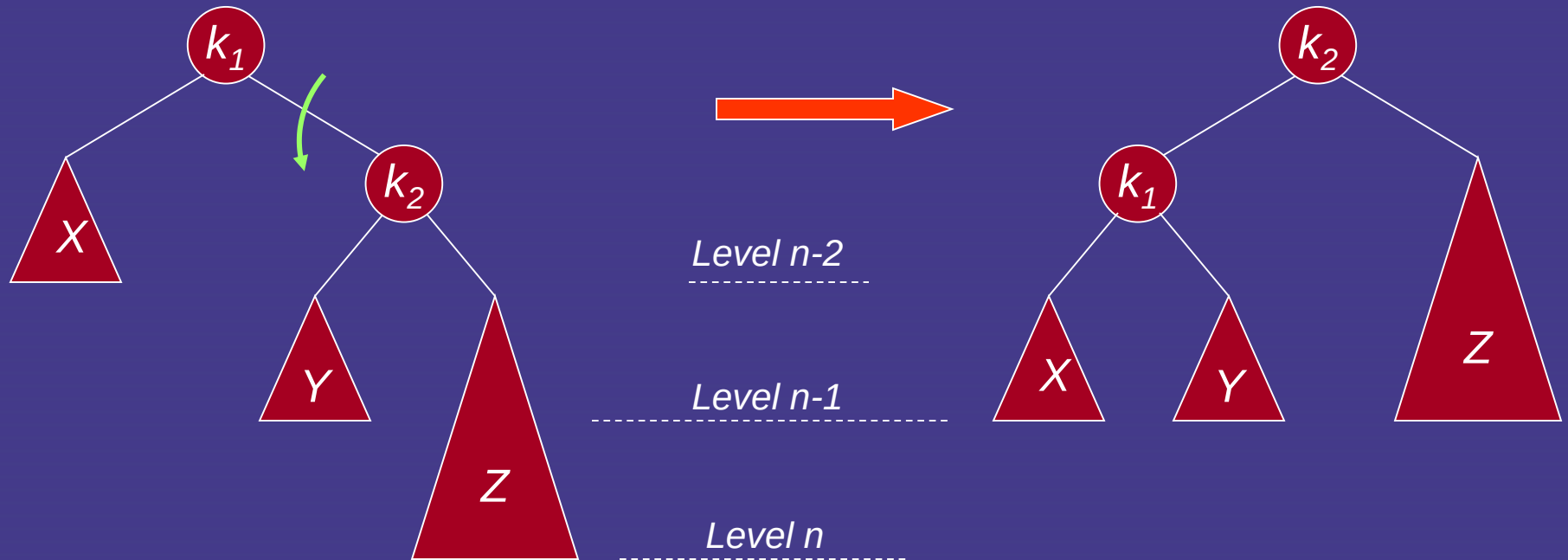
# Cases for Rotation

- Single right rotation to fix case 1.



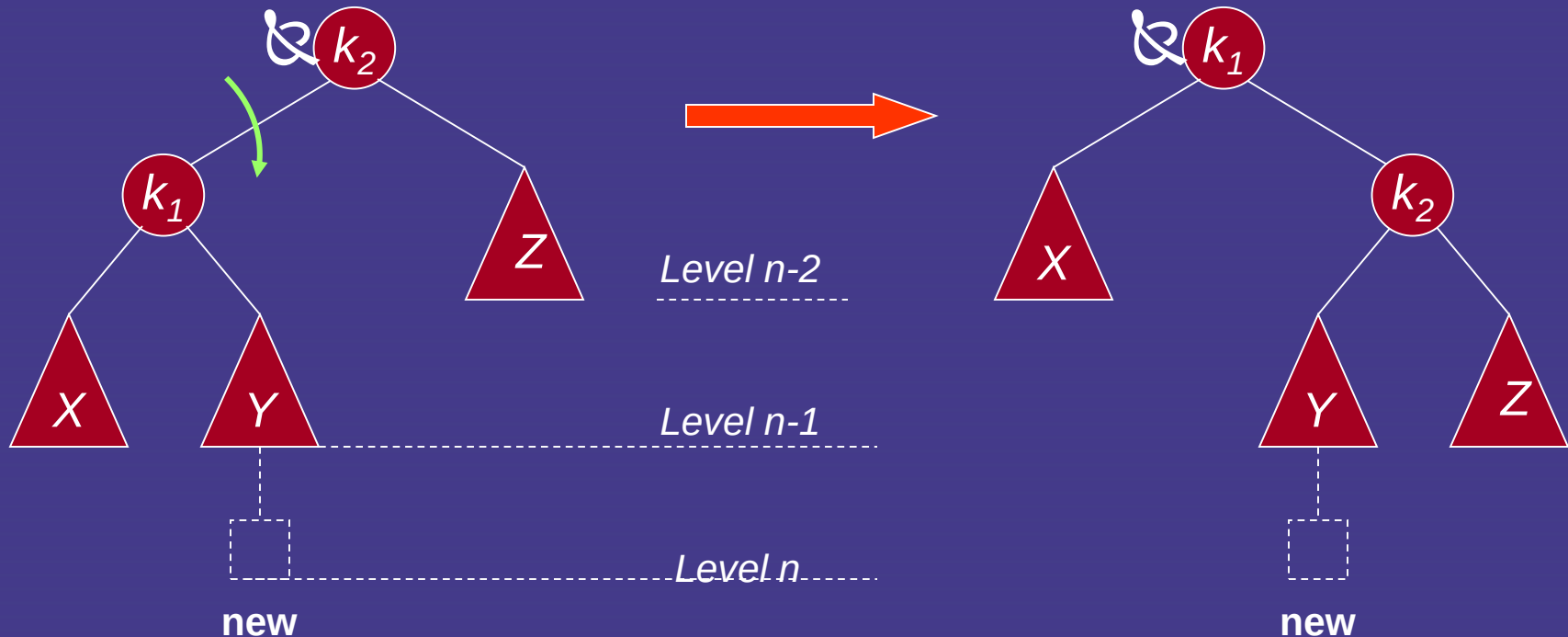
# Cases for Rotation

- Single left rotation to fix case 4.



# Cases for Rotation

- Single right rotation *fails* to fix case 2.

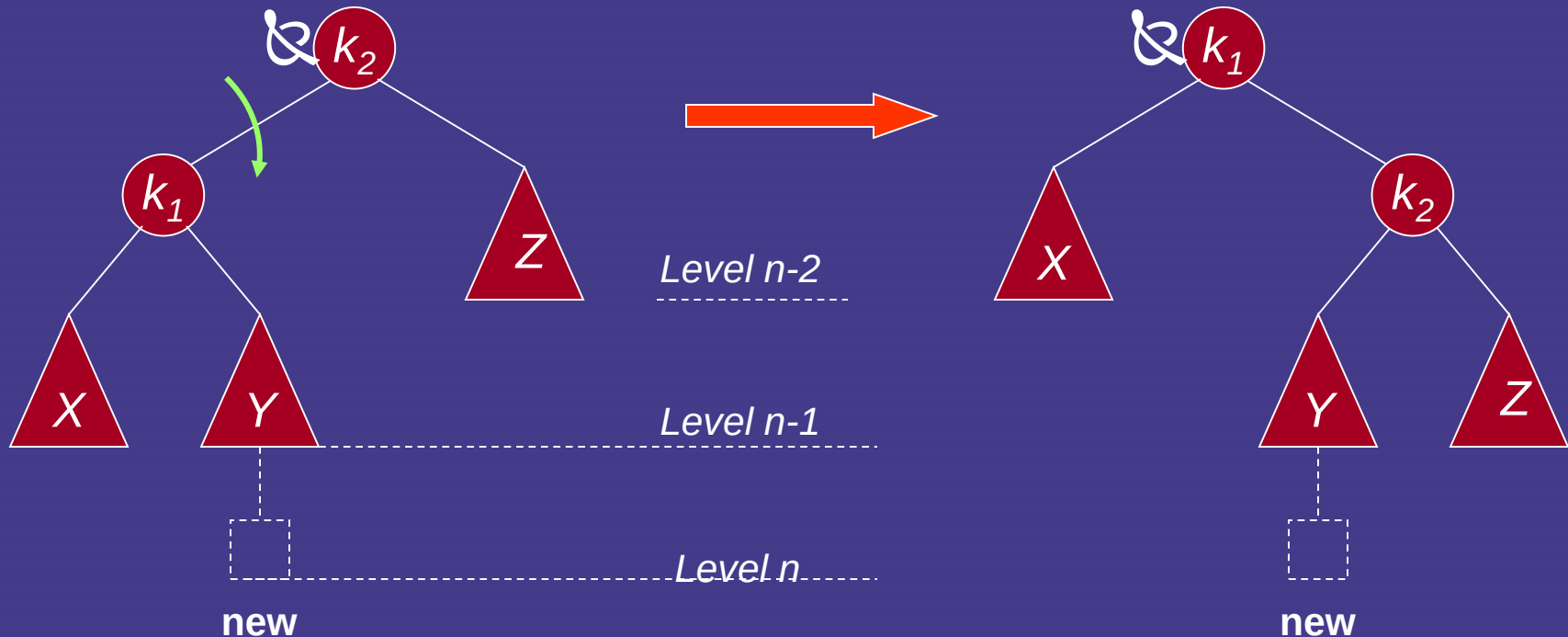


# Data Structures

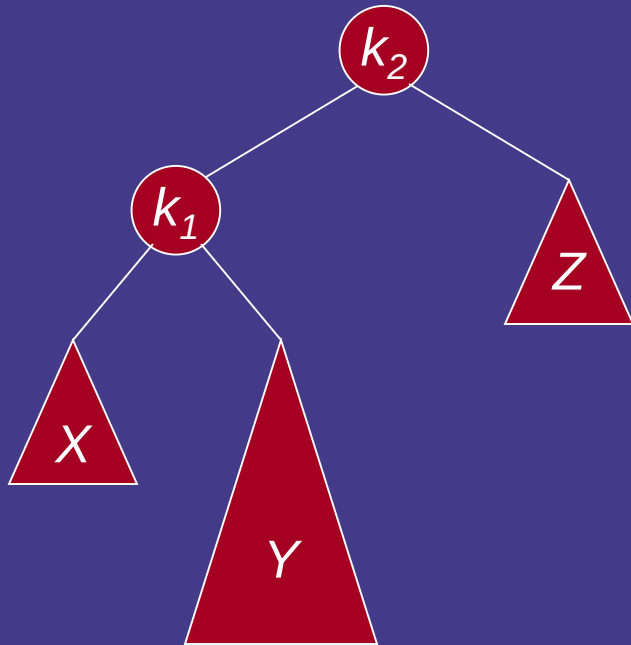
Lecture 22  
Sohail Aslam

# Cases for Rotation

- Single right rotation *fails* to fix case 2.

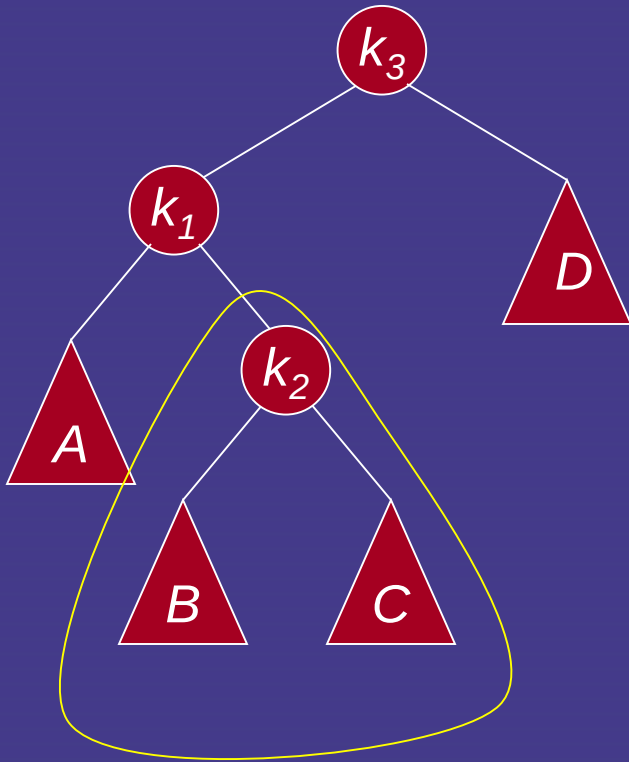


# Cases for Rotation



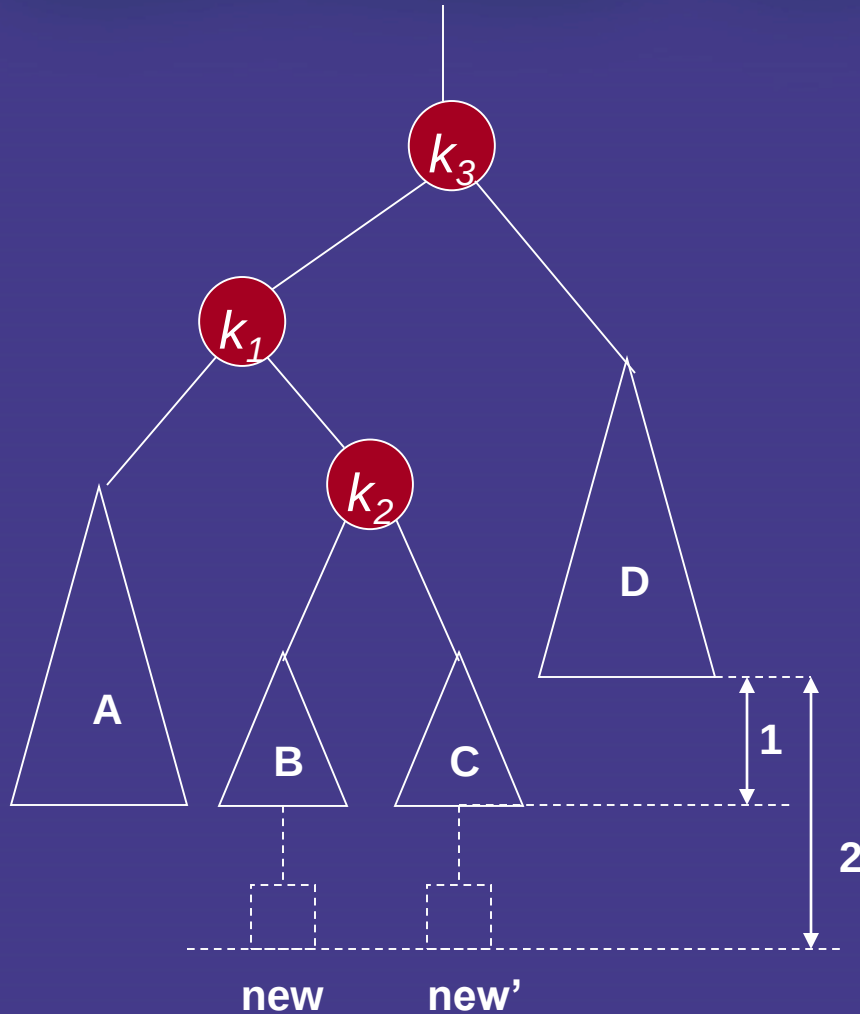
- Y is non-empty because the new node was inserted in Y.
- Thus Y has a root and two subtrees.
- View the entire tree with four subtrees connected with 3 nodes.

# Cases for Rotation



- Y is non-empty because the new node was inserted in Y.
- Thus Y has a root and two subtrees.
- View the entire tree with four subtrees connected with 3 nodes.

# Cases for Rotation

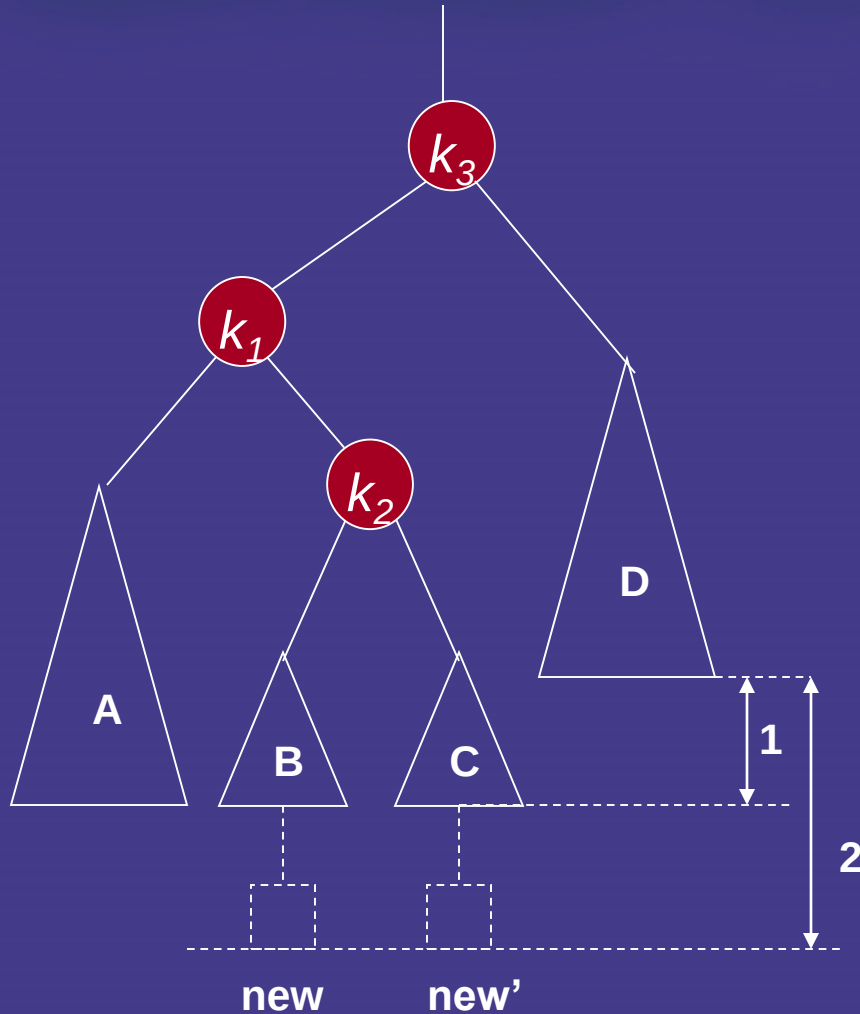


New node inserted at either of the two spots

- Exactly one of tree  $B$  or  $C$  is two levels deeper than  $D$ ; we are not sure which one.
- Good thing: it does not matter.
- To rebalance,  $k_3$  cannot be left as the root.



# Cases for Rotation

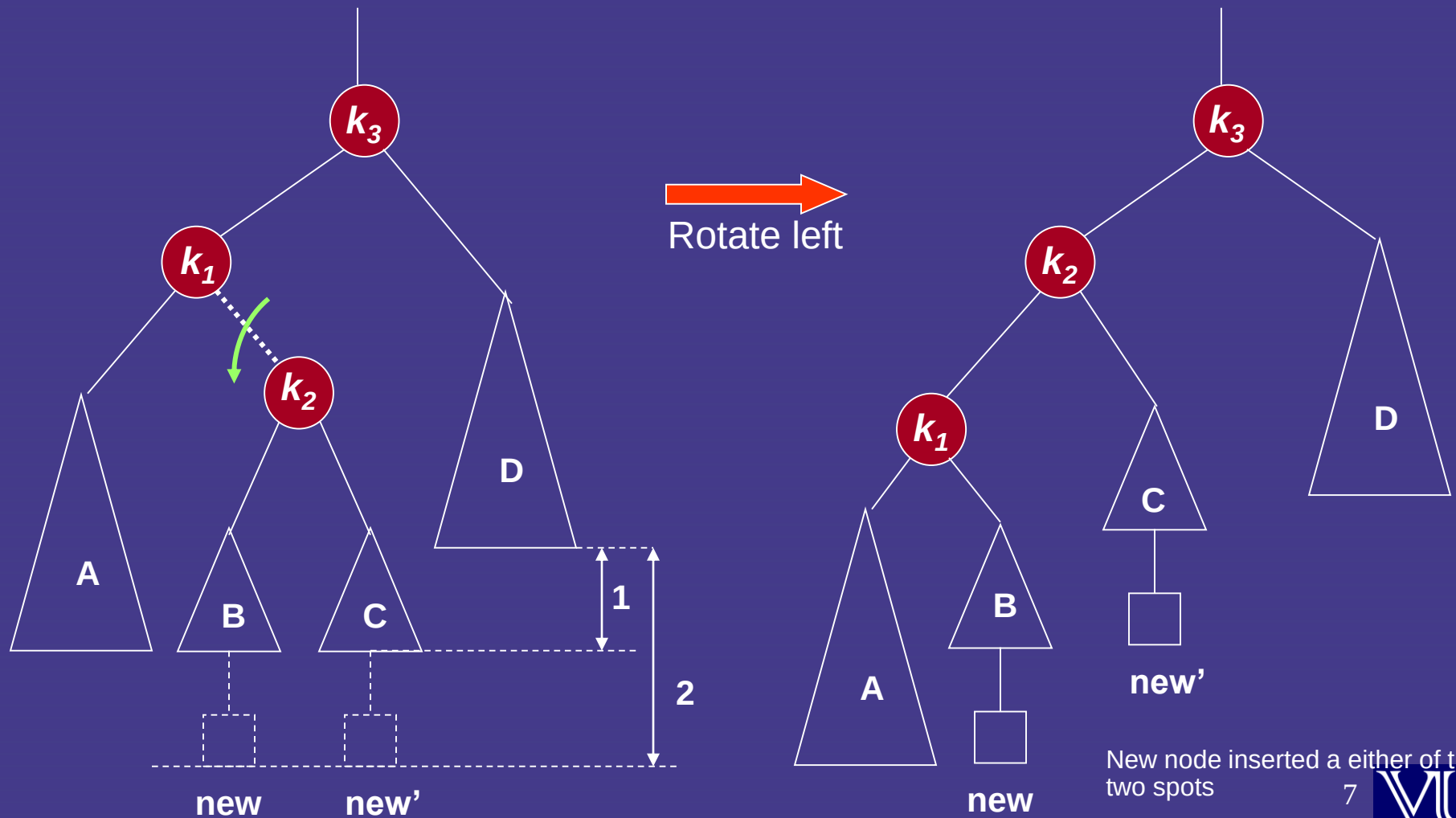


New node inserted at either of the two spots

- A rotation between  $k_3$  and  $k_1$  ( $k_3$  was  $k_2$  then) was shown to not work.
- The only alternative is to place  $k_2$  as the new root.
- This forces  $k_1$  to be  $k_2$ 's left child and  $k_3$  to be its right child.

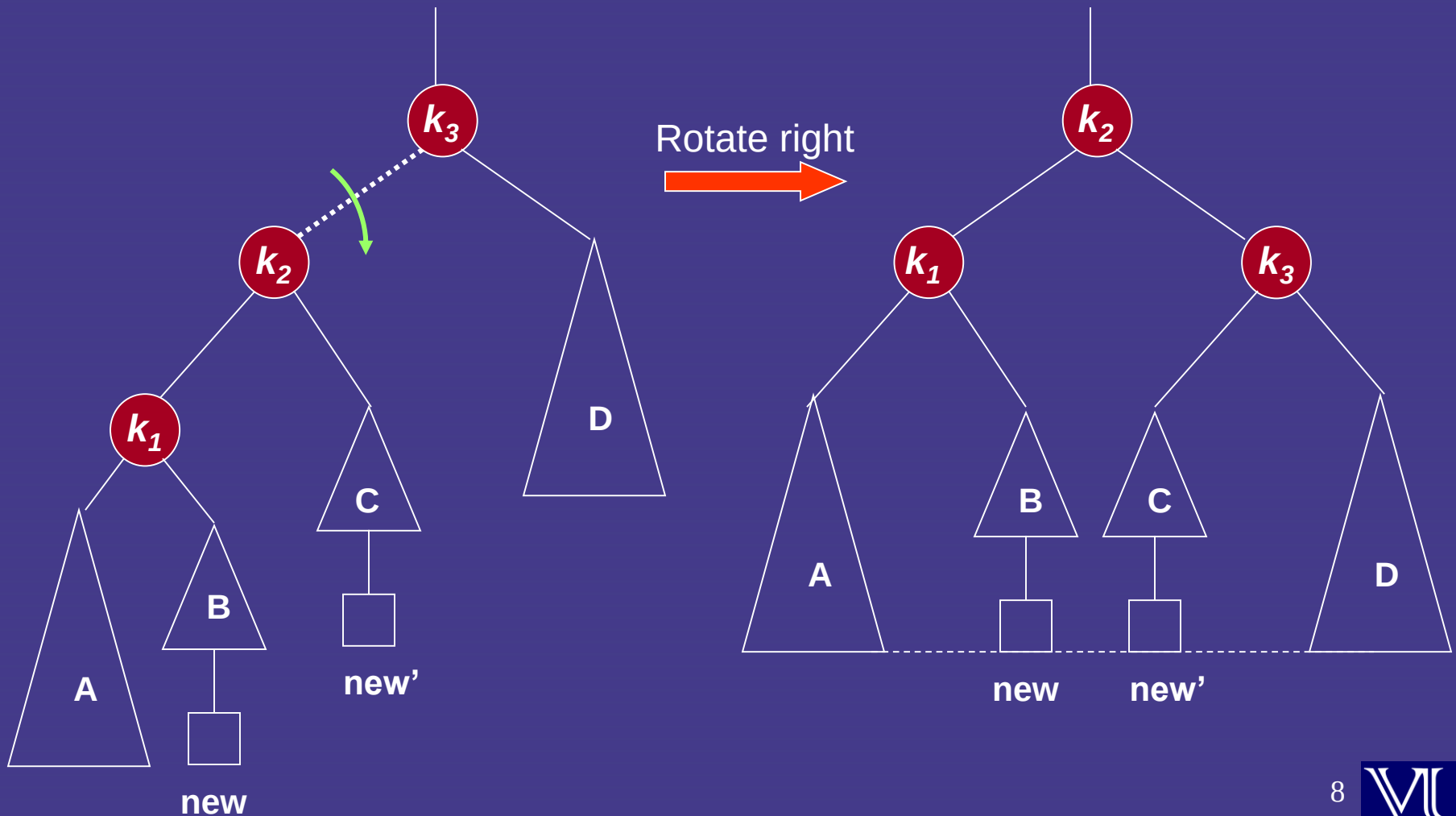
# Cases for Rotation

- Left-right *double* rotation to fix case 2.



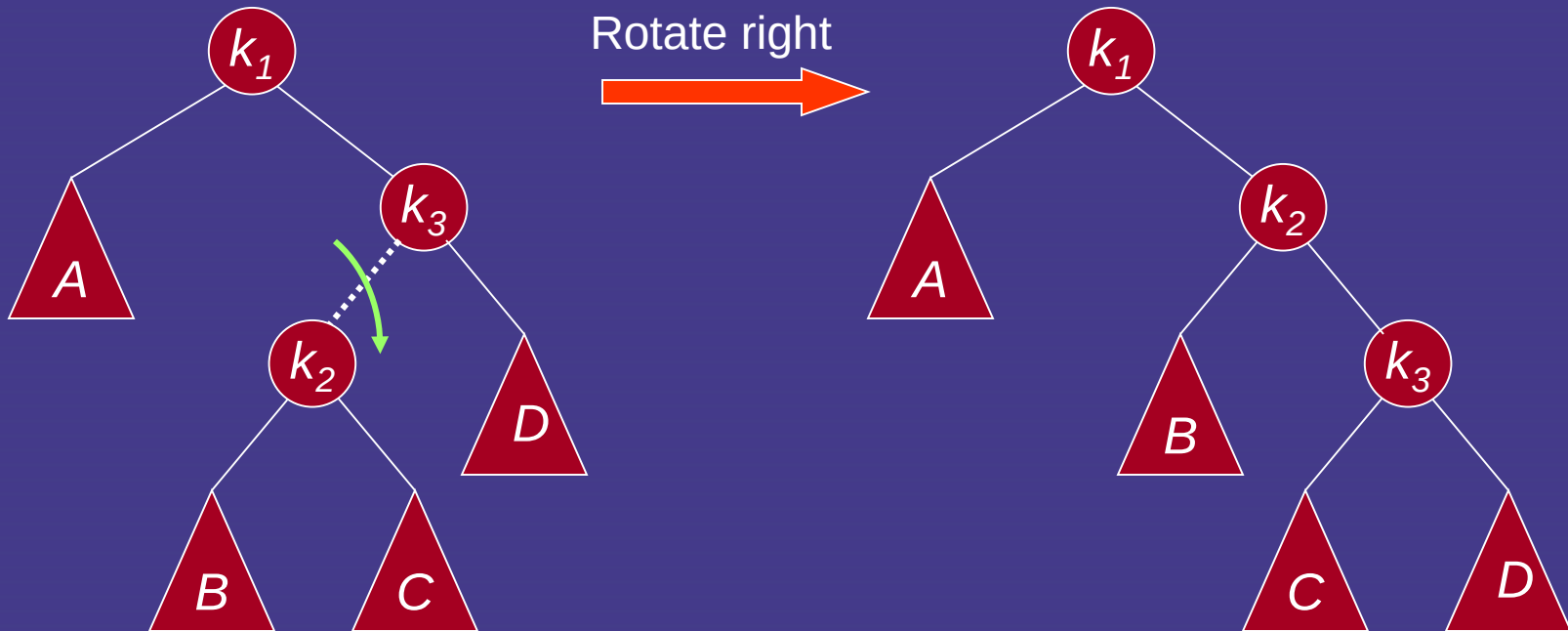
# Cases for Rotation

- Left-right *double* rotation to fix case 2.



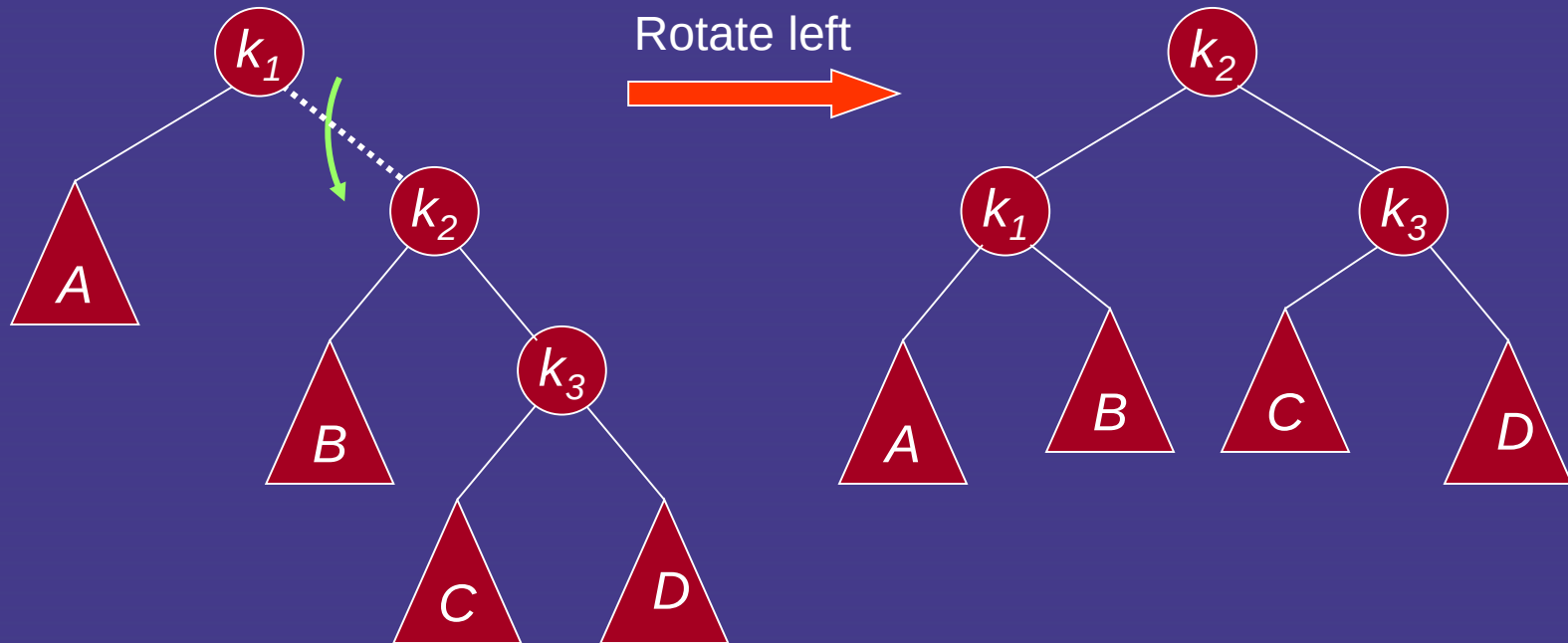
# Cases for Rotation

- Right-left *double* rotation to fix case 3.



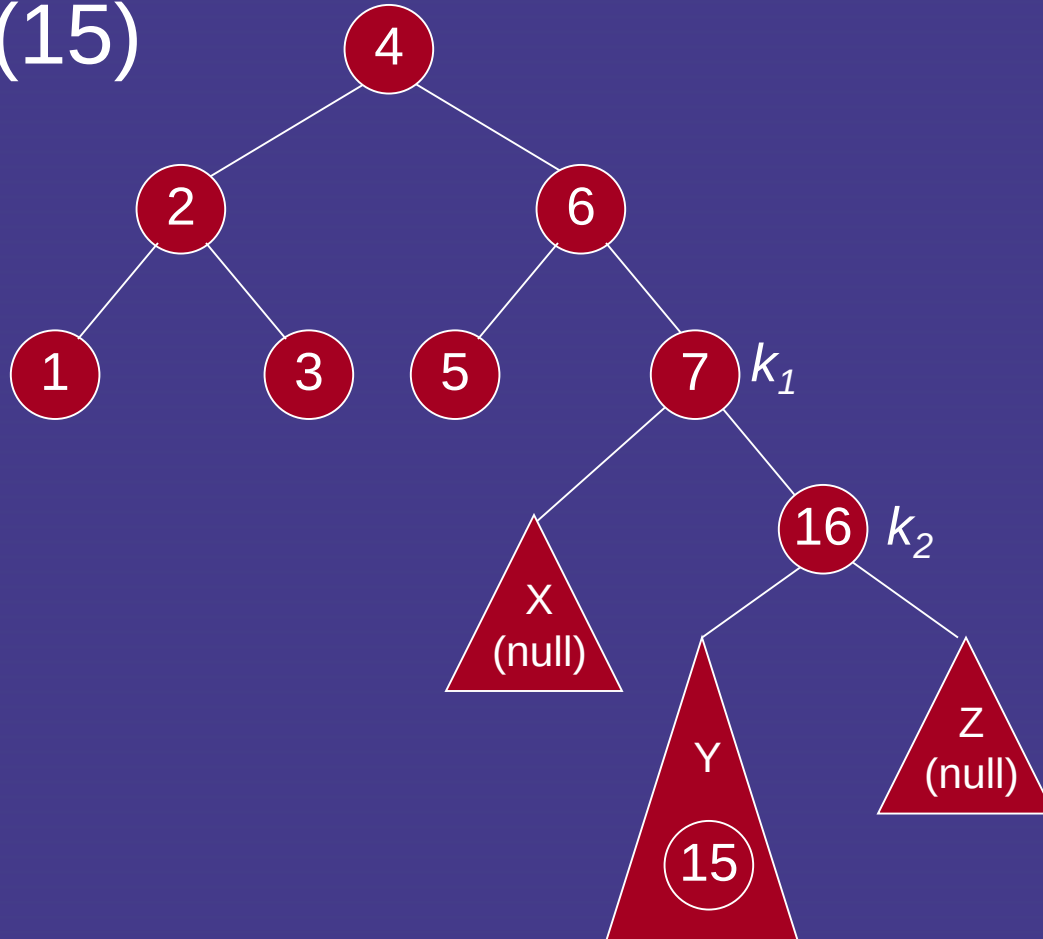
# Cases for Rotation

- Right-left *double* rotation to fix case 3.



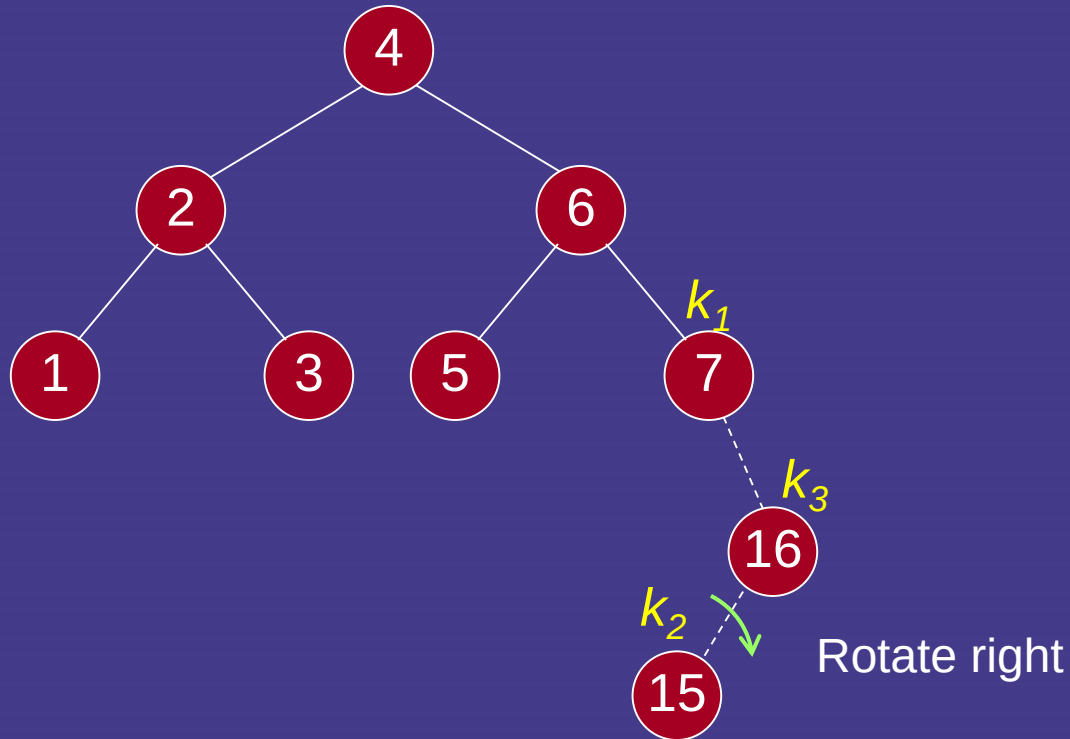
# AVL Tree Building Example

Insert(15)



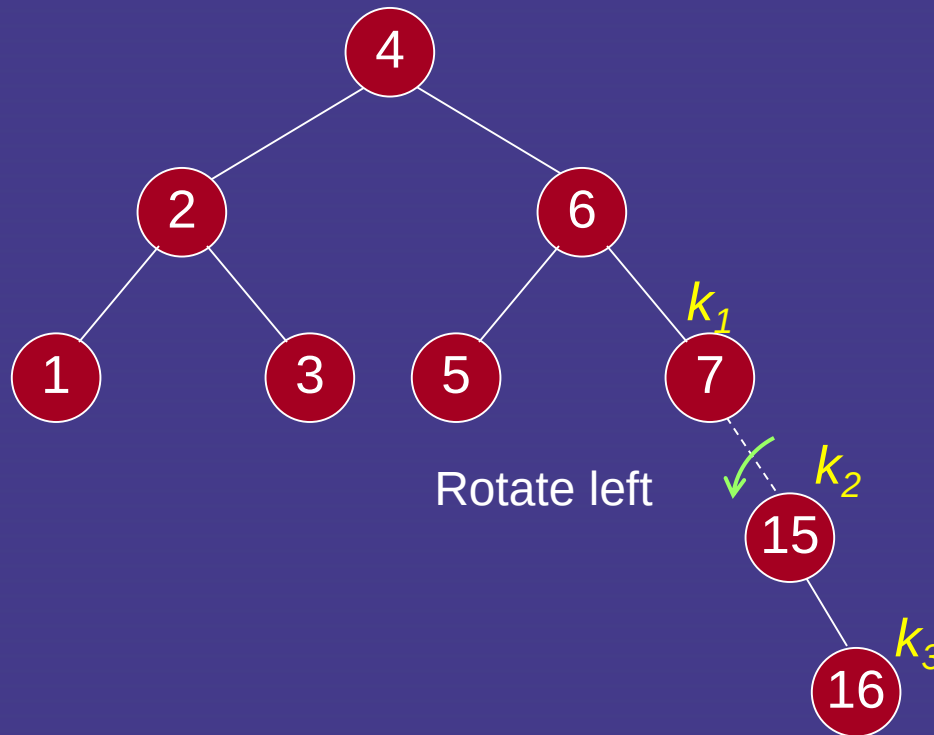
# AVL Tree Building Example

Insert(15) right-left double rotation



# AVL Tree Building Example

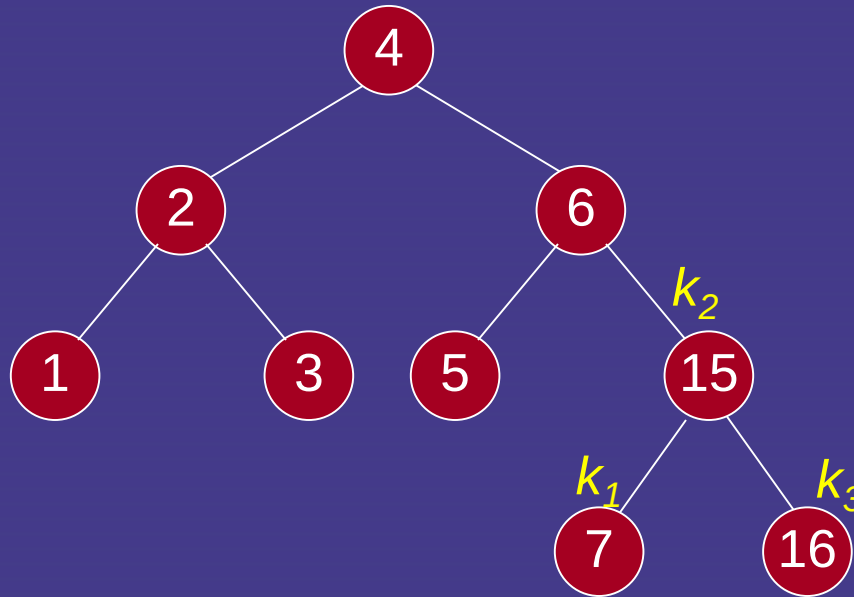
Insert(15) right-left double rotation





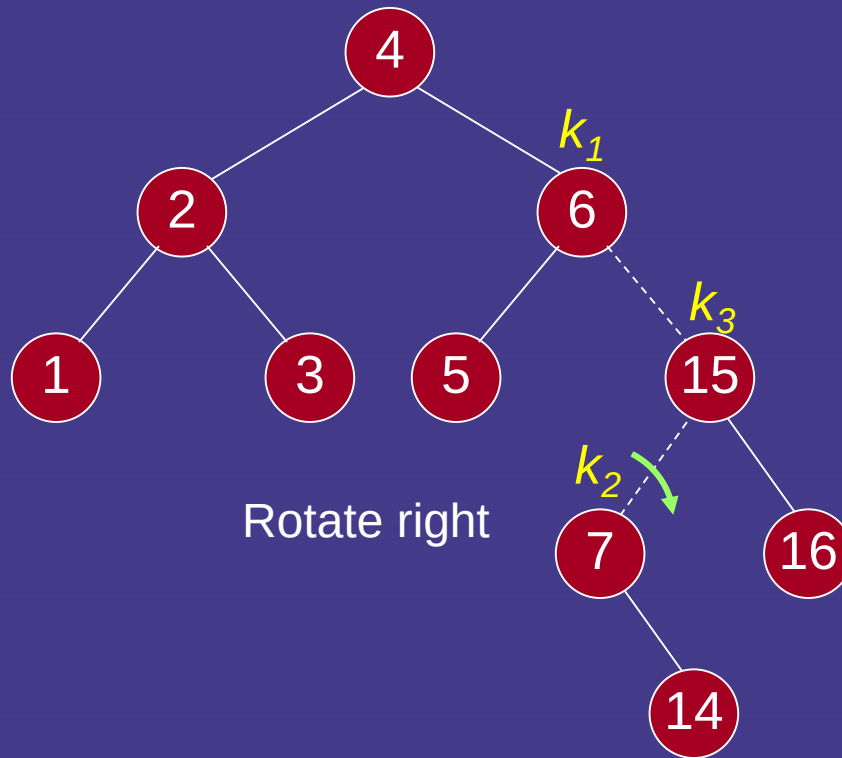
# AVL Tree Building Example

Insert(15) right-left double rotation



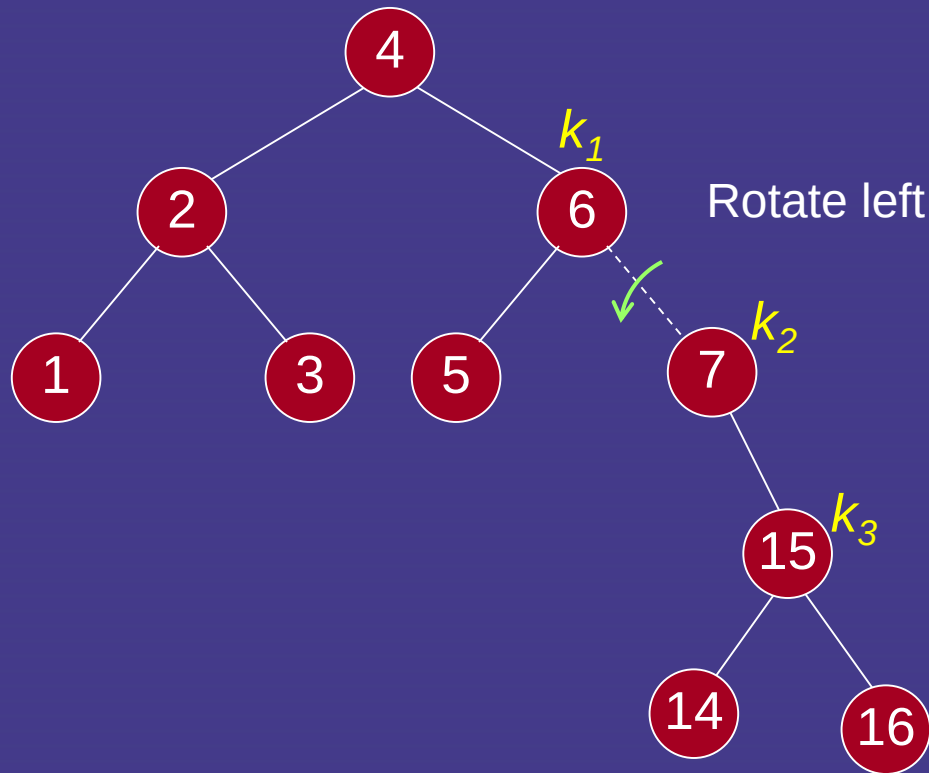
# AVL Tree Building Example

Insert(14): right-left double rotation



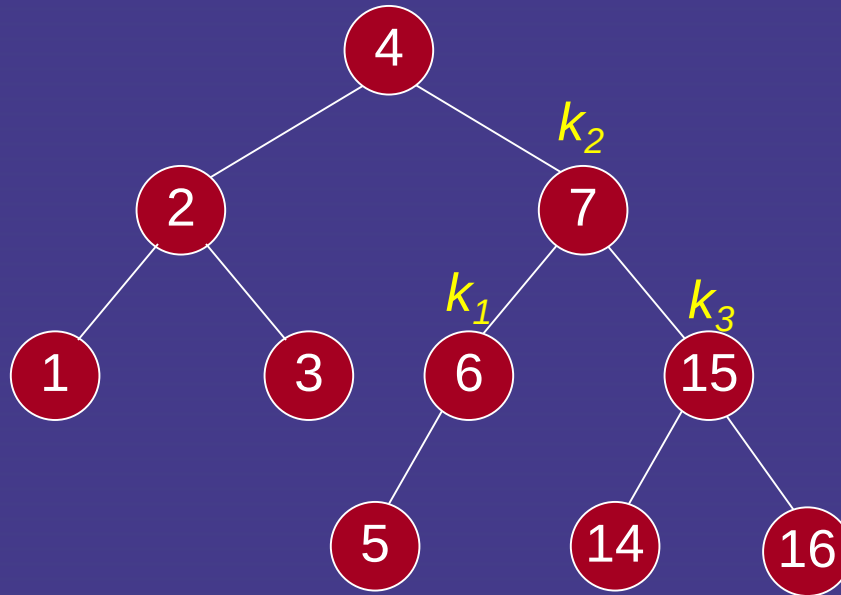
# AVL Tree Building Example

Insert(14): right-left double rotation



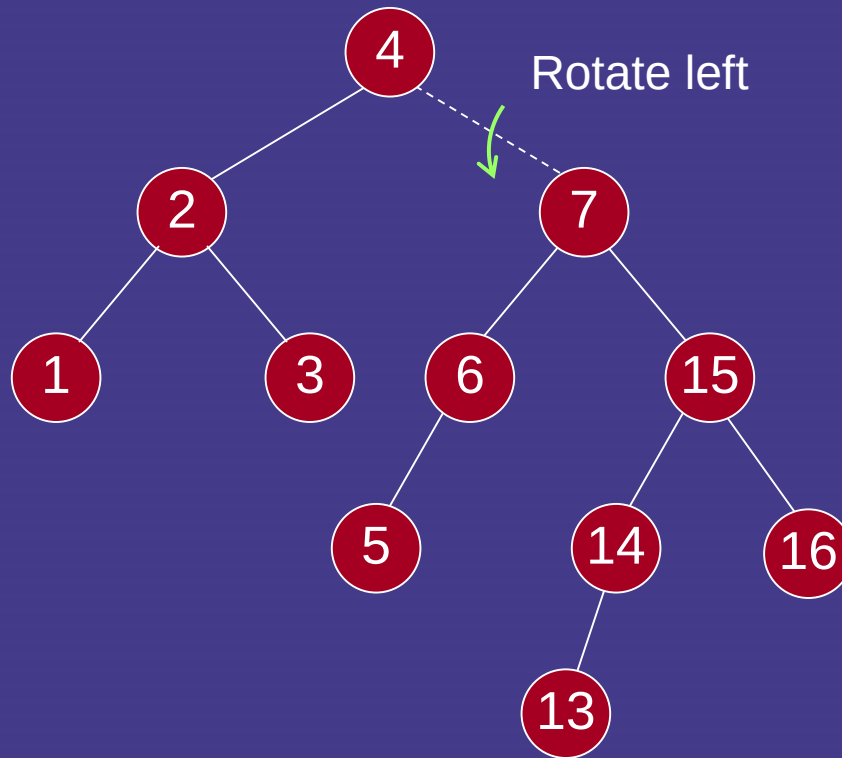
# AVL Tree Building Example

Insert(14): right-left double rotation



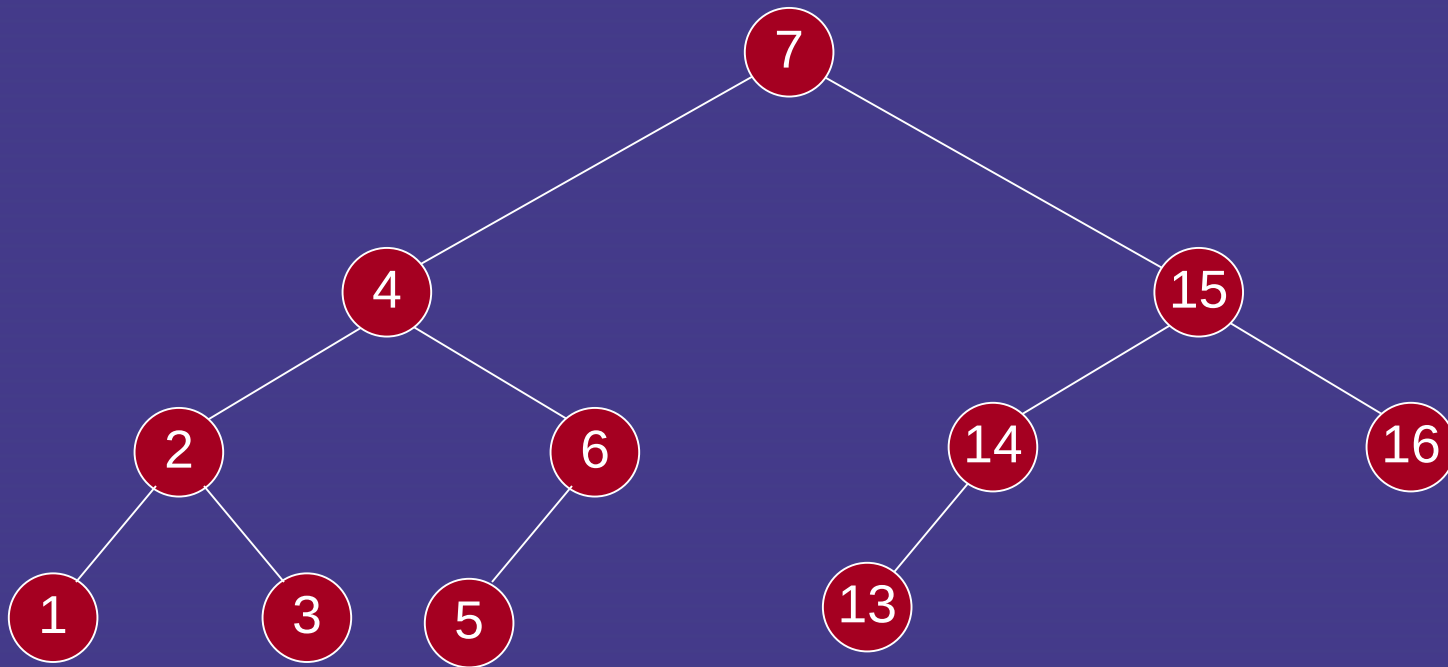
# AVL Tree Building Example

Insert(13): single rotation



# AVL Tree Building Example

Insert(13): single rotation



# C++ code for insert

► `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() )  
                root = singleRightRotation( root );  
            else  
                root = doubleLeftRightRotation( root );  
    }  
}
```

# C++ code for insert

```
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() )
                root = singleRightRotation( root );
            else
                root = doubleLeftRightRotation( root );
    }
}
```



# C++ code for insert

```
TreeNode<int>*
```

```
avlInsert(TreeNode<int>* root, int info)
```

```
{
```

```
    if( info < root->getInfo() ){
```

Root of left subtree may change



```
        root->setLeft(avlInsert(root->getLeft(), info));
```

```
        int htdiff = height(root->getLeft()) -  
                    height(root->getRight());
```

```
        if( htdiff == 2 )
```

```
            if( info < root->getLeft()->getInfo() )
```

```
                root = singleRightRotation( root );
```

```
            else
```

```
                root = doubleLeftRightRotation( root );
```

```
    }
```

# C++ code for insert

```
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() )
                root = singleRightRotation( root );
            else
                root = doubleLeftRightRotation( root );
    }
}
```

# C++ code for insert

```
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() )
                root = singleRightRotation( root );
            else
                root = doubleLeftRightRotation( root );
    }
}
```

# C++ code for insert

```
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() )
                root = singleRightRotation( root );
            else
                root = doubleLeftRightRotation( root );
    }
}
```

➡

Outside case

inside case

# C++ code for insert

```
    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;
```

```
}
```

# C++ code for insert

```
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;
```

```
}
```

# C++ code for insert

```
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;
```

```
}
```

# C++ code for insert

```
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;
```

```
}
```



# C++ code for insert

```
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;
```

```
}
```

# C++ code for insert

```
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;
```

```
}
```

# C++ code for insert

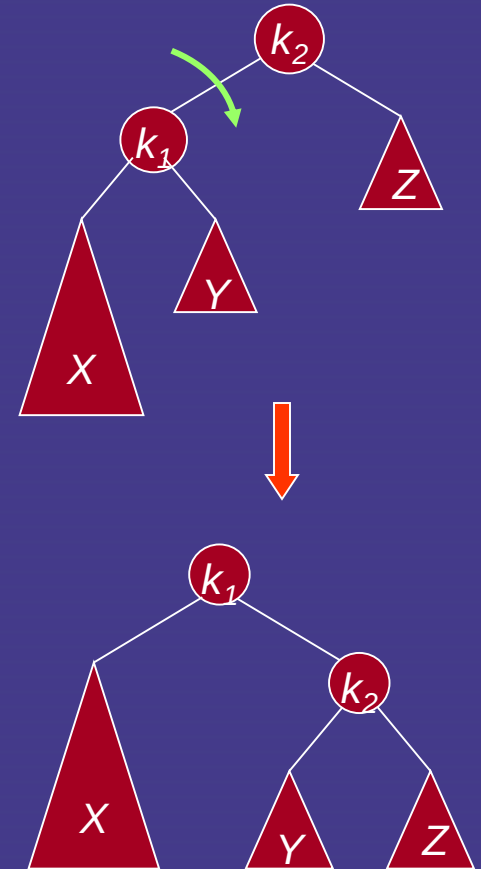
```
else if(info > root->getInfo() ) {
    root->setRight(avlInsert(root->getRight(),info));
    int htdiff = height(root->getRight()) -
                height(root->getLeft());
    if( htdiff == 2 )
        if( info > root->getRight()->getInfo() )
            root = singleLeftRotation( root );
        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;
}
```

# singleRightRotation

```
► 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;
}
```

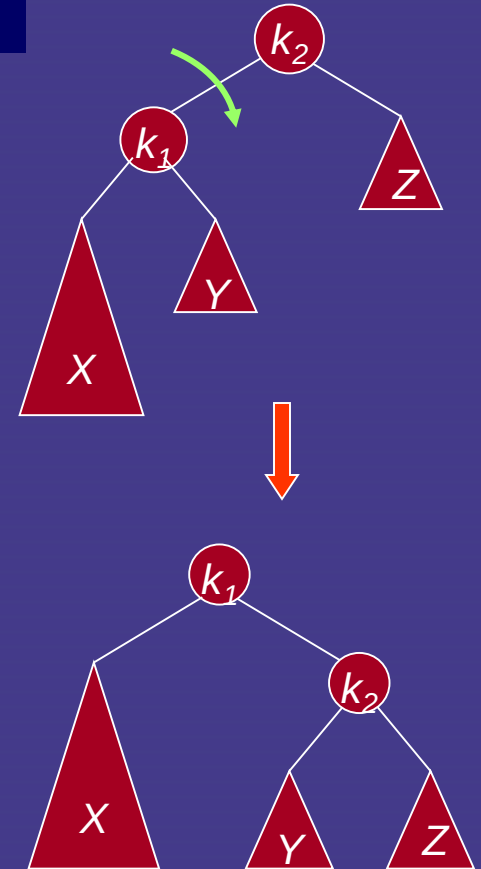


# singleRightRotation

```
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;
}
```

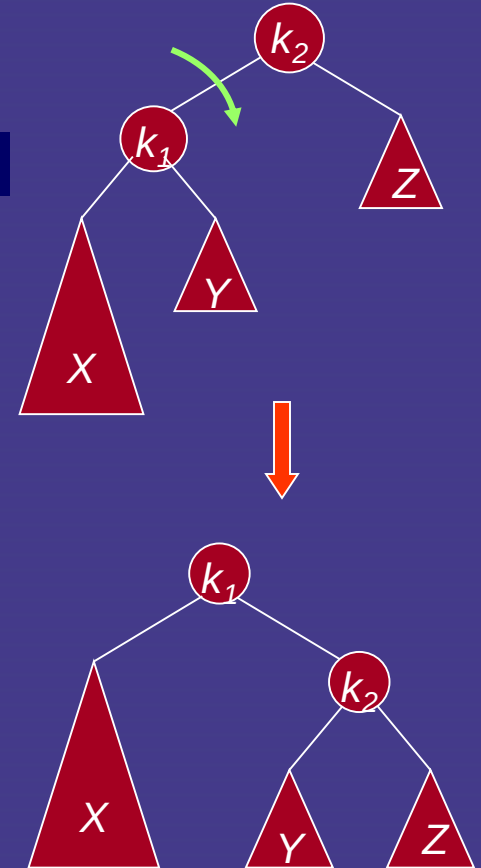


# singleRightRotation

```
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;
}
```

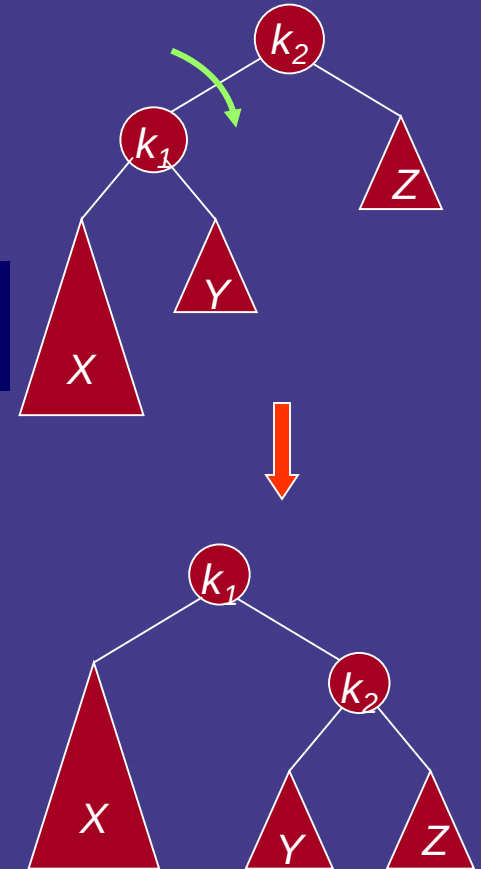


# singleRightRotation

```
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;
}
```

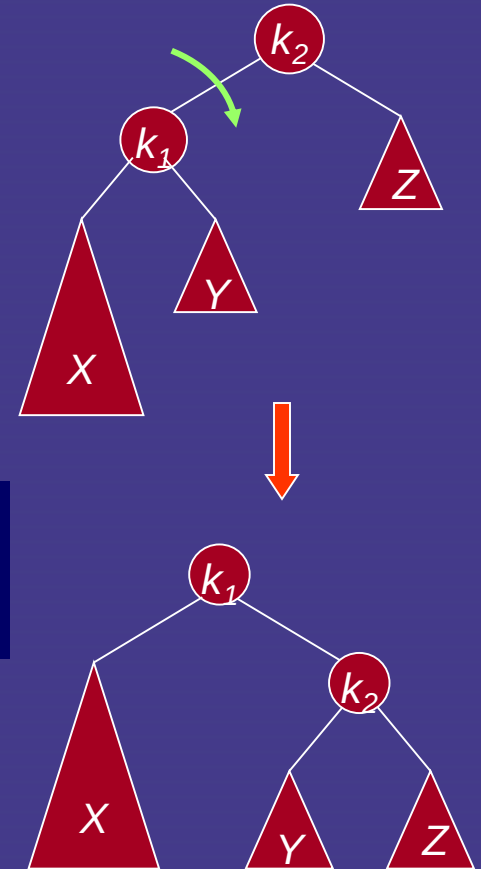


# singleRightRotation

```
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;
}
```



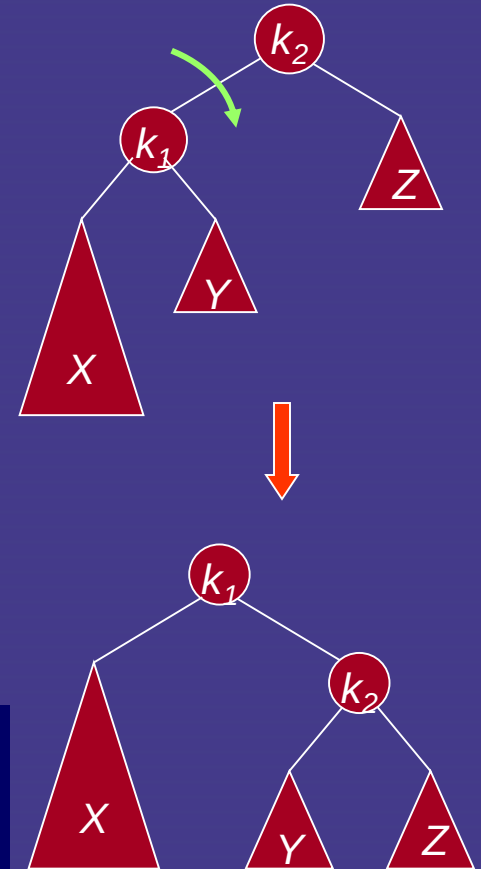


# singleRightRotation

```
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;
}
```

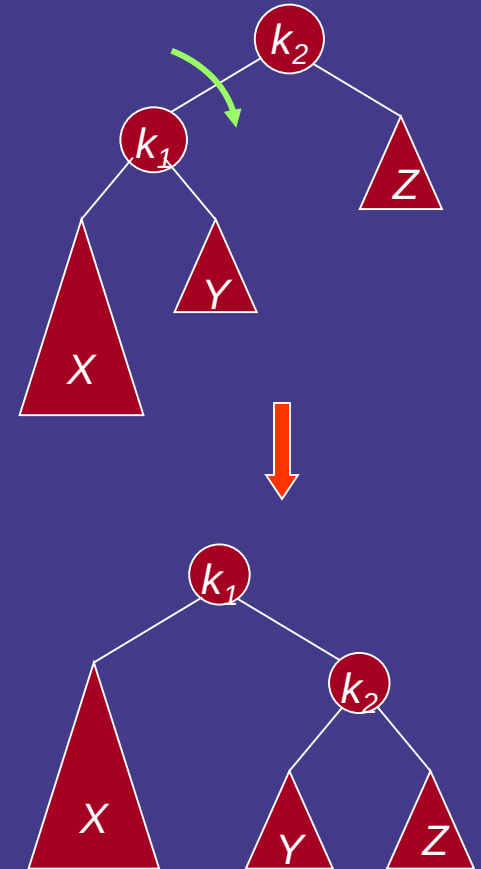


# singleRightRotation

```
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;
}
```



# height

```
int height( TreeNode<int>* node )
```

```
{
```

```
    if( node != NULL ) return node->getHeight();
```

```
    return -1;
```

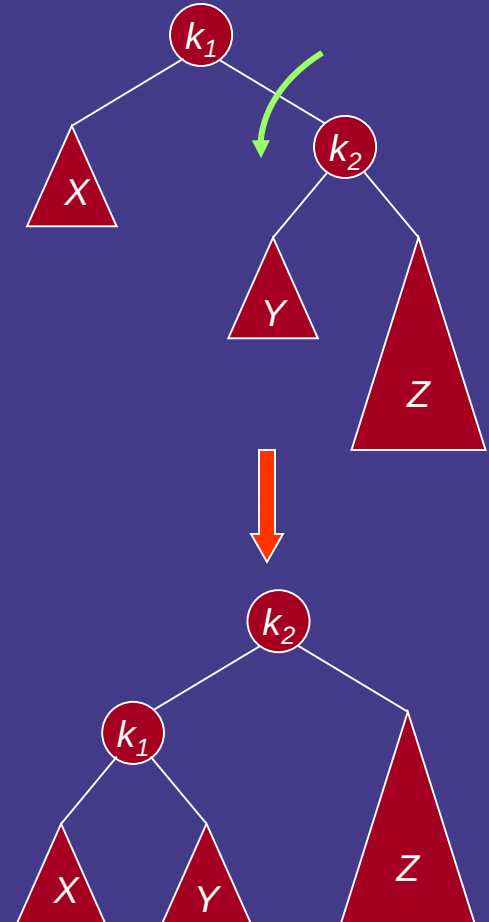
```
}
```

# height

```
int height( TreeNode<int>* node )  
{  
    if( node != NULL ) return node->getHeight();  
    return -1;  
}
```

# singleLeftRotation

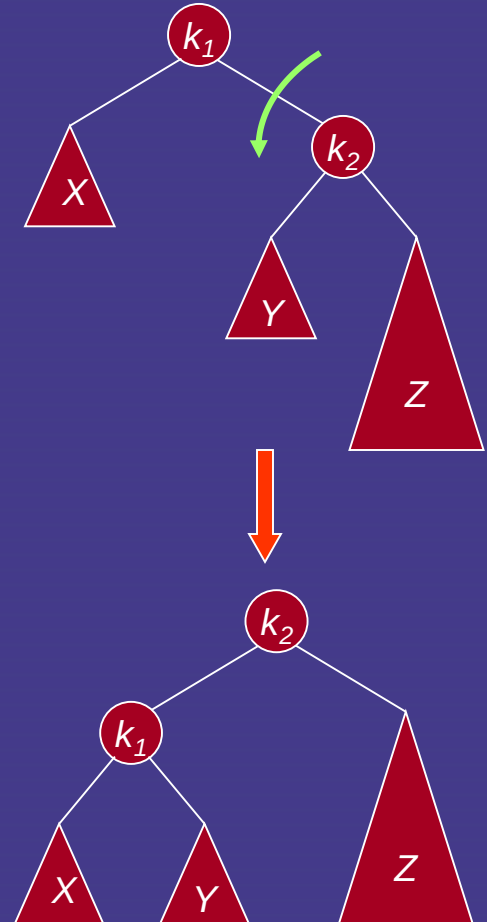
```
► TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )  
{  
    if( k1 == NULL ) return NULL;  
    // k2 is now the new root  
    TreeNode<int>* k2 = k1->getRight();  
    k1->setRight( k2->getLeft() ); // Y  
    k2->setLeft( k1 );  
    // reassign heights. First k1 (demoted)  
    int h = 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;  
  
    }
```



# singleLeftRotation

```
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 );

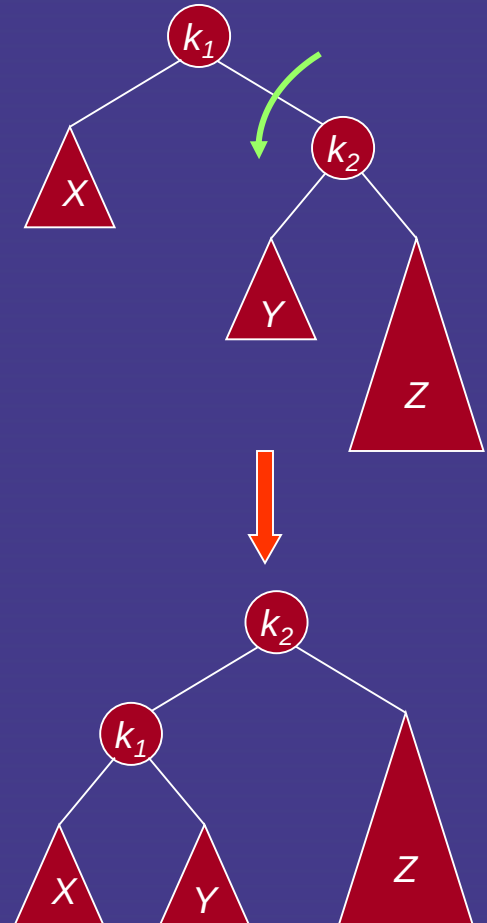
    return k2;
}
```



# singleLeftRotation

```
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 );

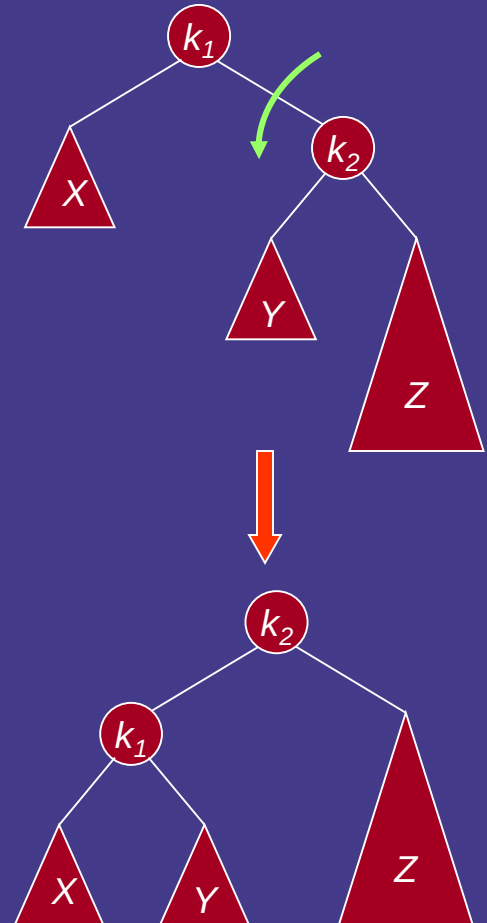
    return k2;
}
```



# singleLeftRotation

```
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;
}
```

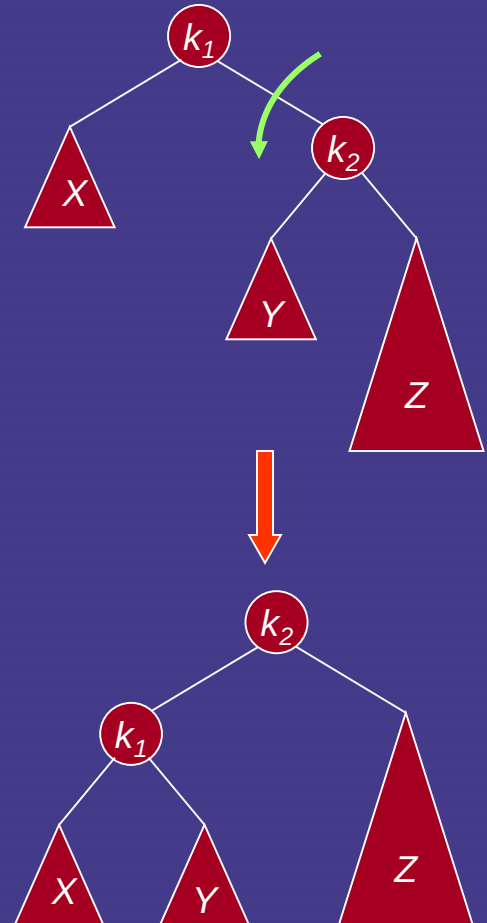




# singleLeftRotation

```
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 );

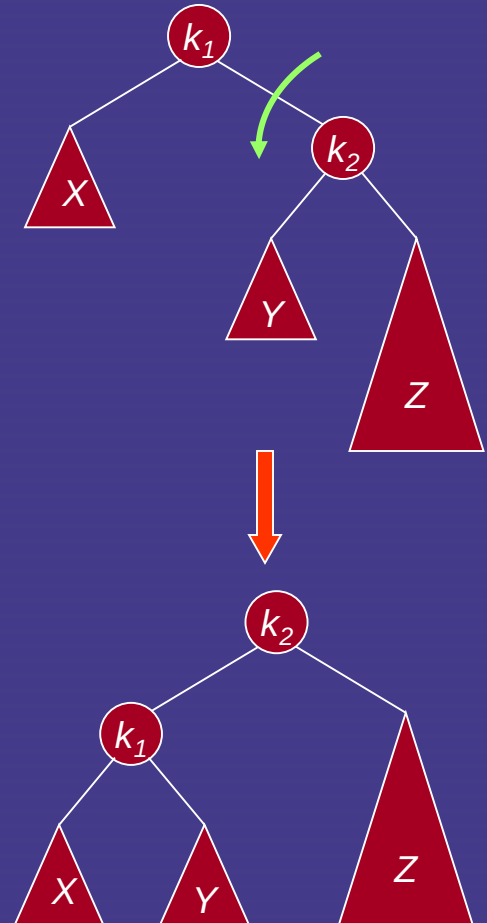
    return k2;
}
```



# singleLeftRotation

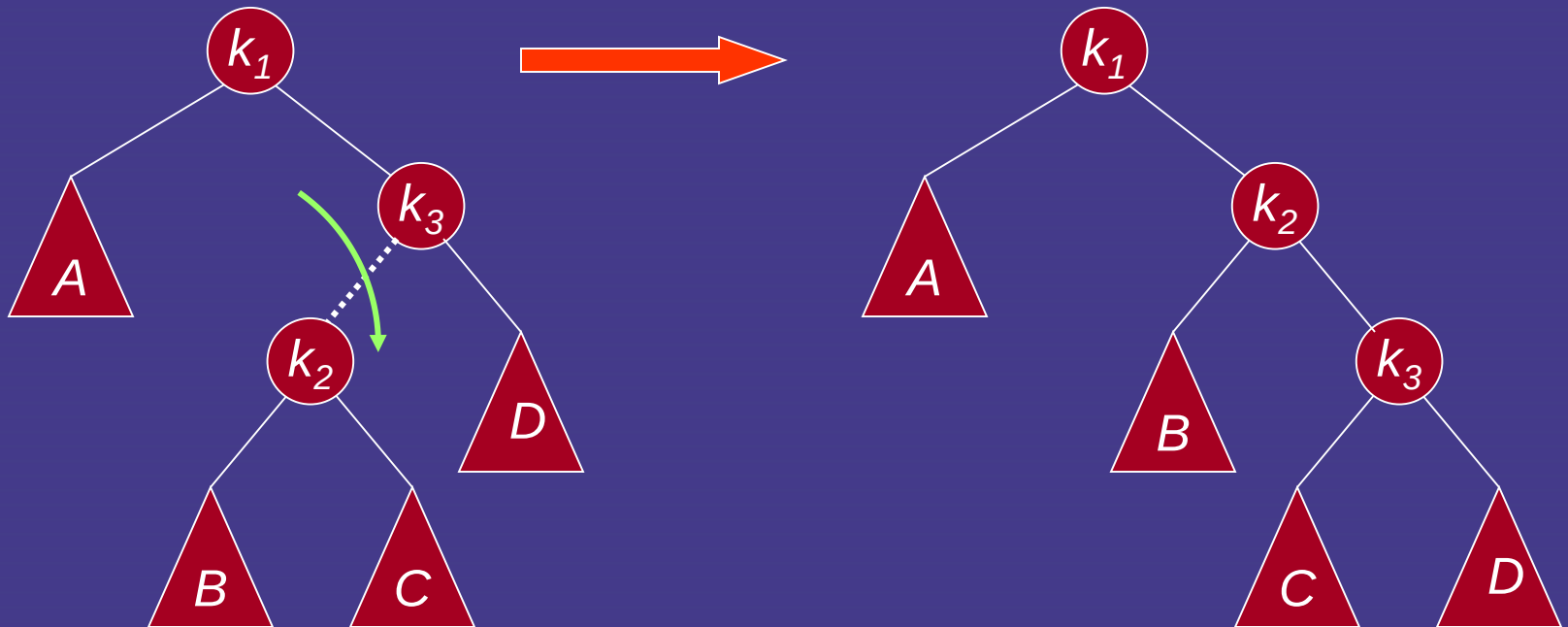
```
TreeNode<int>* singleLeftRotation( TreeNode<int>* k1 )
{
    if( k1 == NULL ) return NULL;
    // k2 is now the new root
    TreeNode<int>* k2 = k1->getRight();
    k1->setRight( k2->getLeft() ); // Y
    k2->setLeft( k1 );
    // reassign heights. First k1 (demoted)
    int h = Max(height(k1->getLeft()),
                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;
}
```



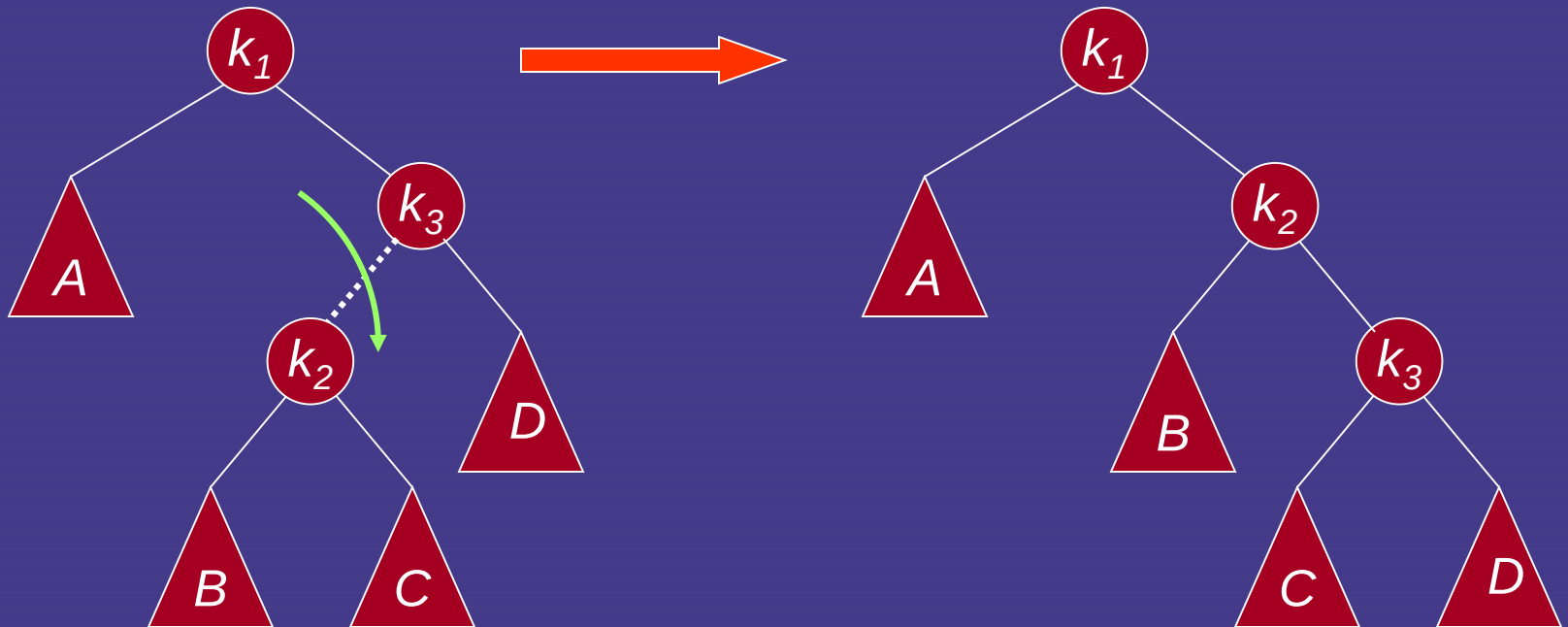
# doubleRightLeftRotation

```
► 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);  
}
```



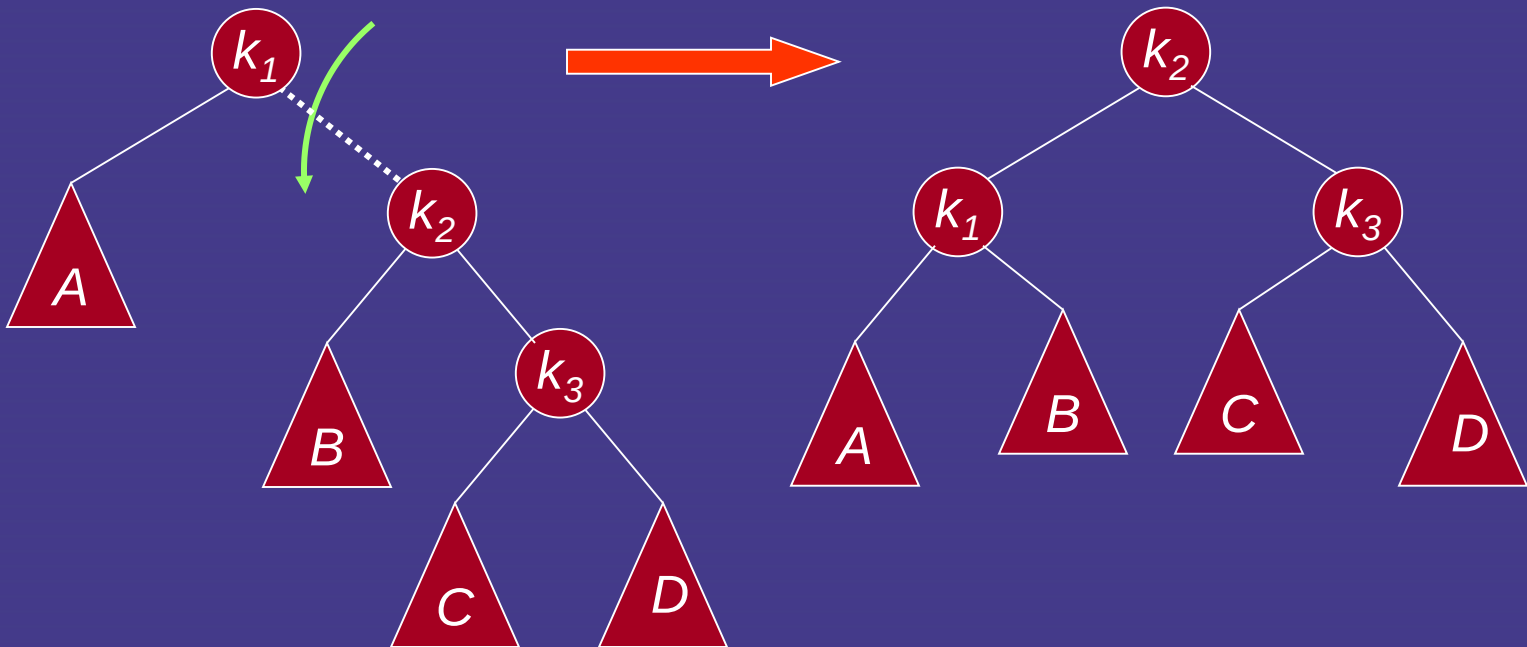
# doubleRightLeftRotation

```
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);
}
```



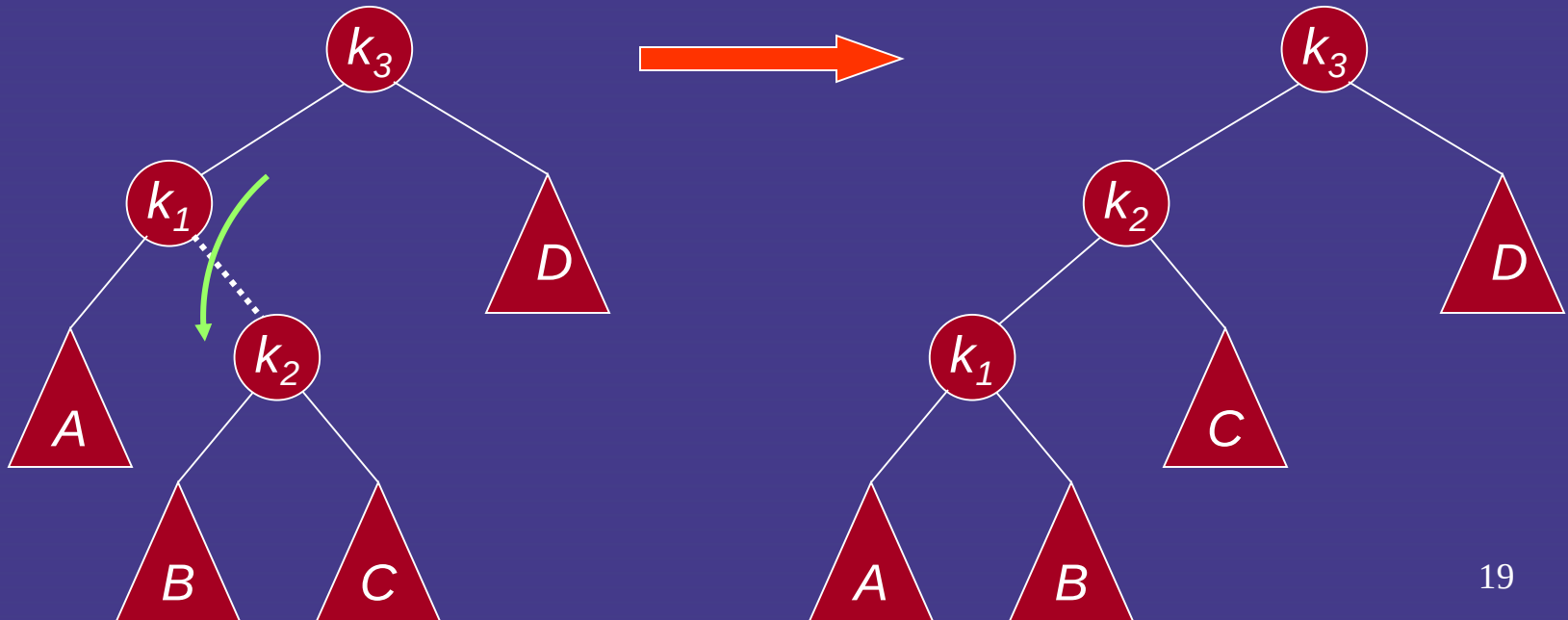
# doubleRightLeftRotation

```
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);
}
```



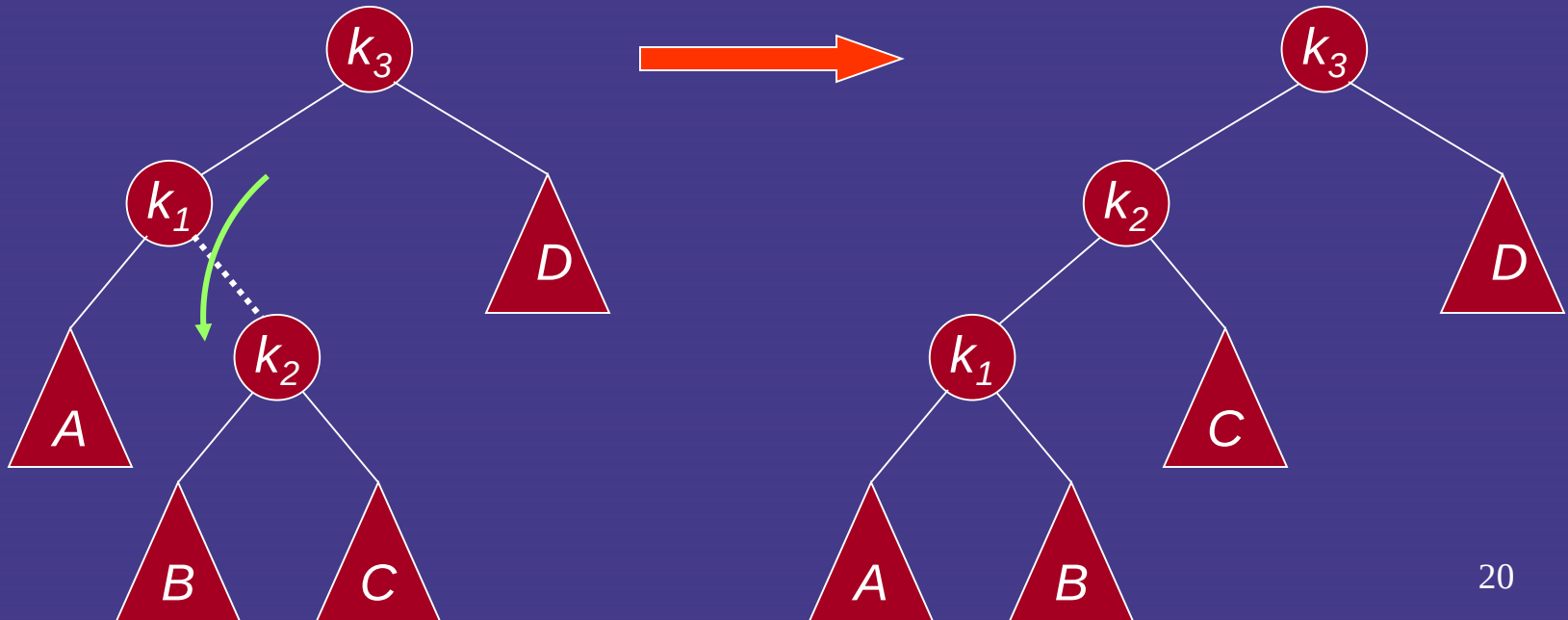
# doubleRightLeftRotation

```
► 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);  
}
```



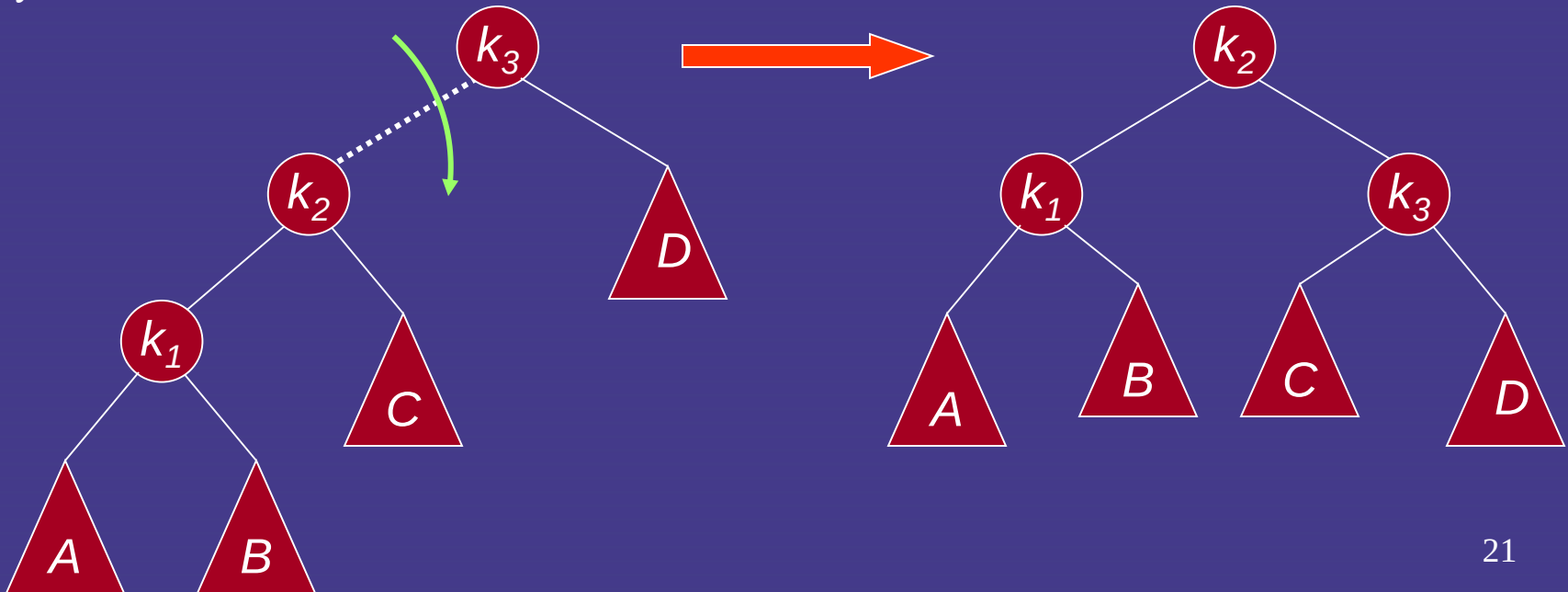
# doubleRightLeftRotation

```
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);
}
```



# doubleRightLeftRotation

```
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);
}
```





# Deletion in AVL Tree

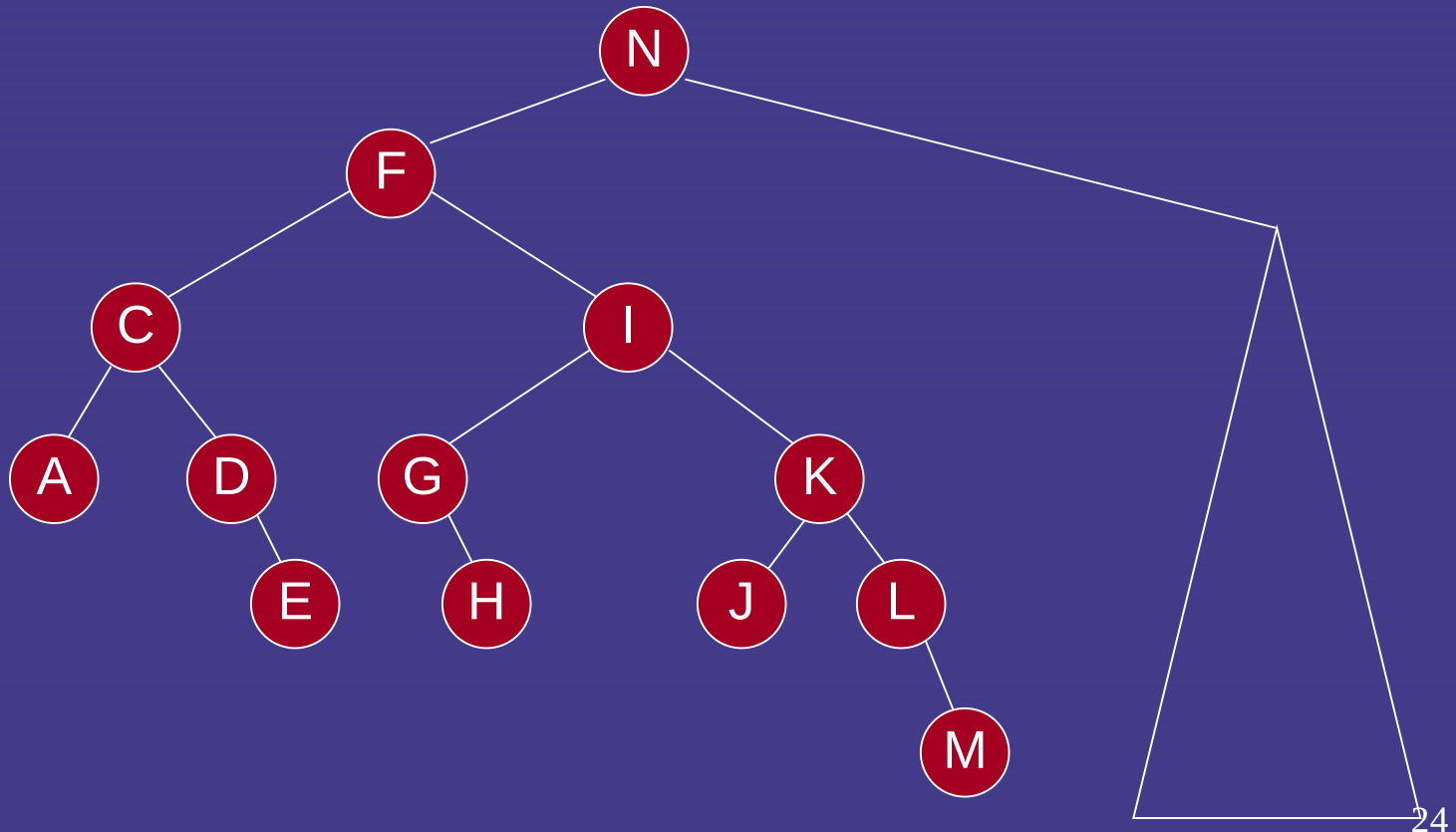
- Delete is the inverse of insert: given a value  $X$  and an AVL tree  $T$ , delete the node containing  $X$  and rebalance the tree, if necessary.
- Turns out that deletion of a node is considerably more complex than insert

# Deletion in AVL Tree

- Insertion in a height-balanced tree requires at most one single rotation or one double rotation.
- We can use rotations to restore the balance when we do a deletion.
- We may have to do a rotation at every level of the tree:  $\log_2 N$  rotations in the worst case.

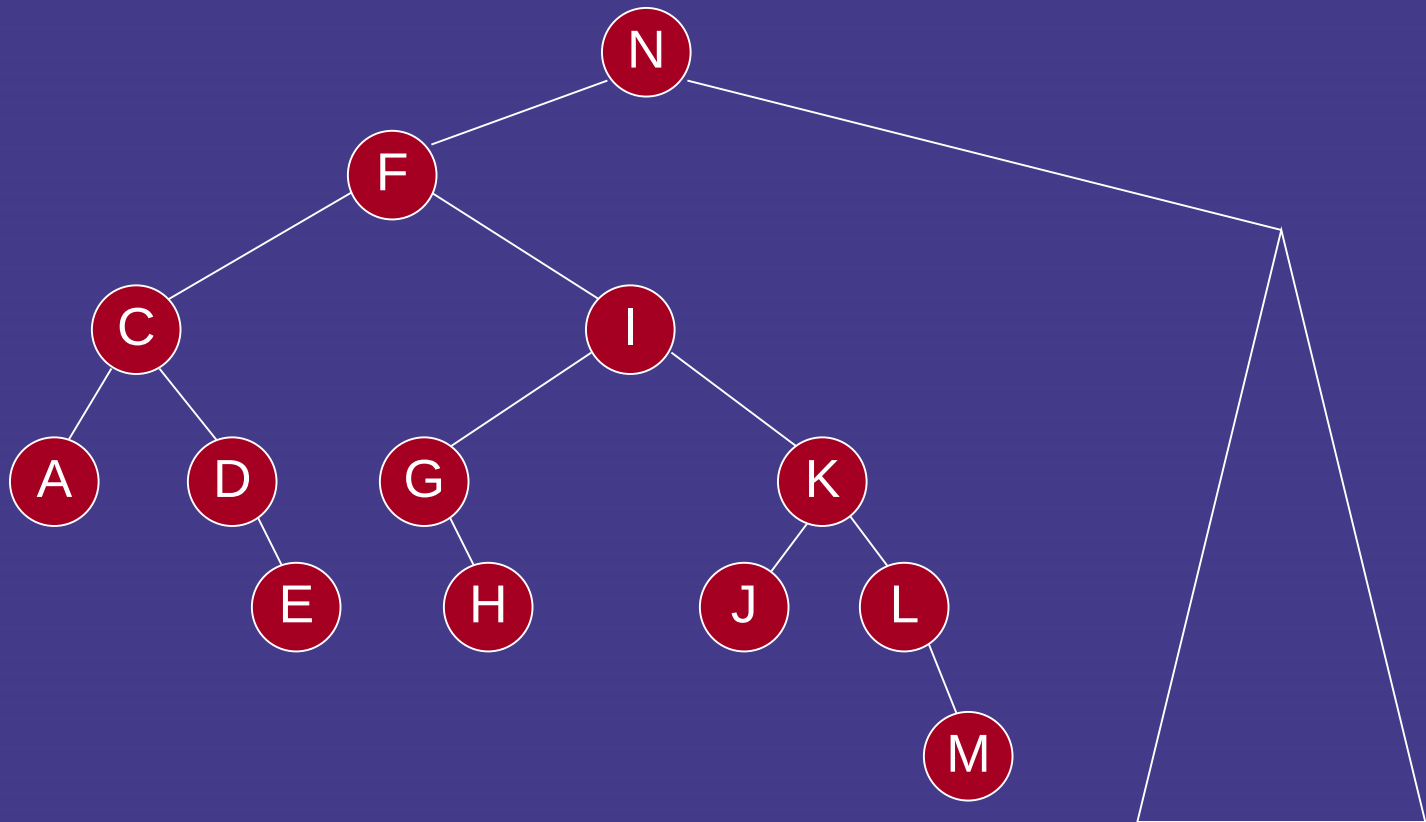
# Deletion in AVL Tree

- Here is a tree that causes this worse case number of rotations when we delete A. At every node in N's left subtree, the left subtree is one shorter than the right subtree.



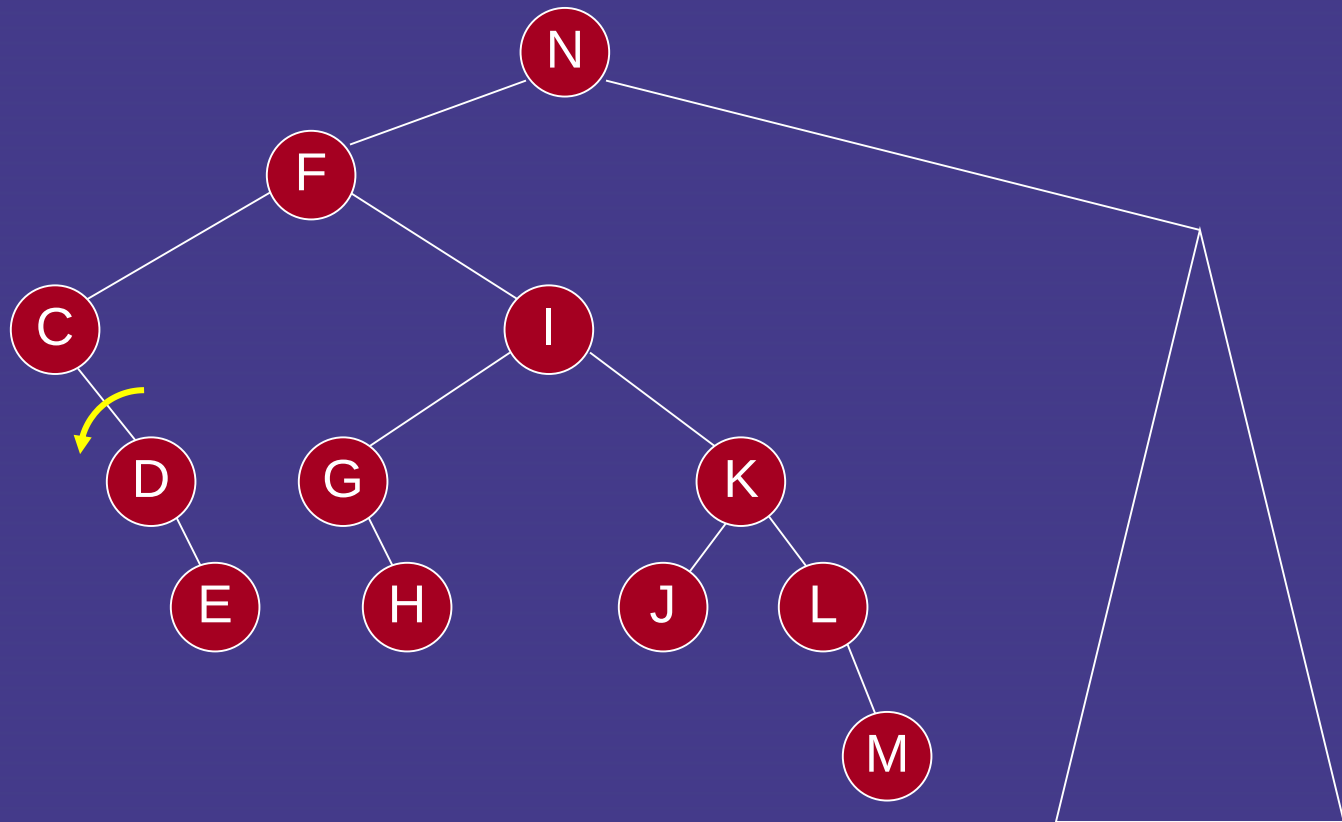
# Deletion in AVL Tree

- Deleting A upsets balance at C. When rotate (D up, C down) to fix this



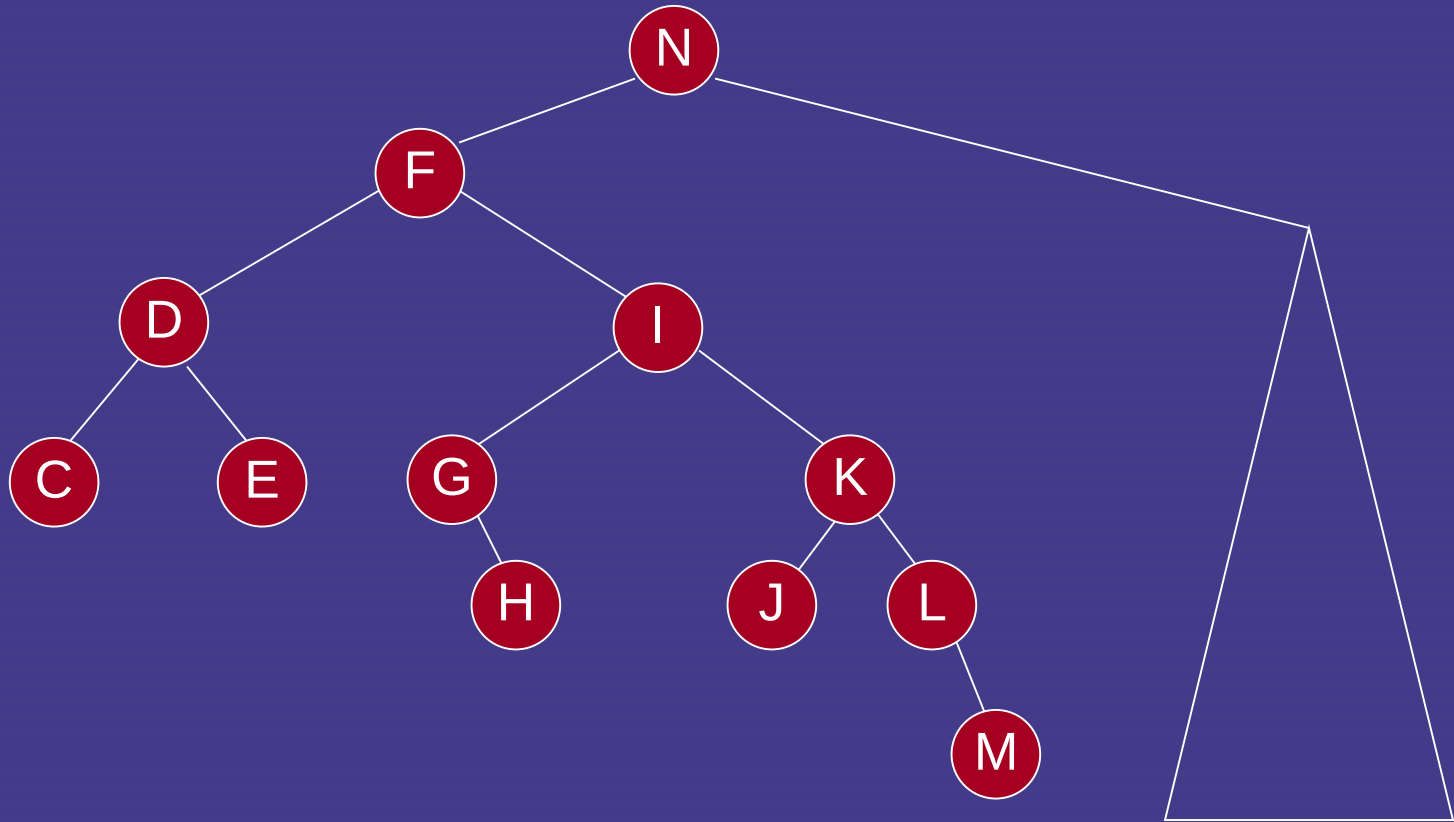
# Deletion in AVL Tree

- Deleting A upsets balance at C. When rotate (D up, C down) to fix this



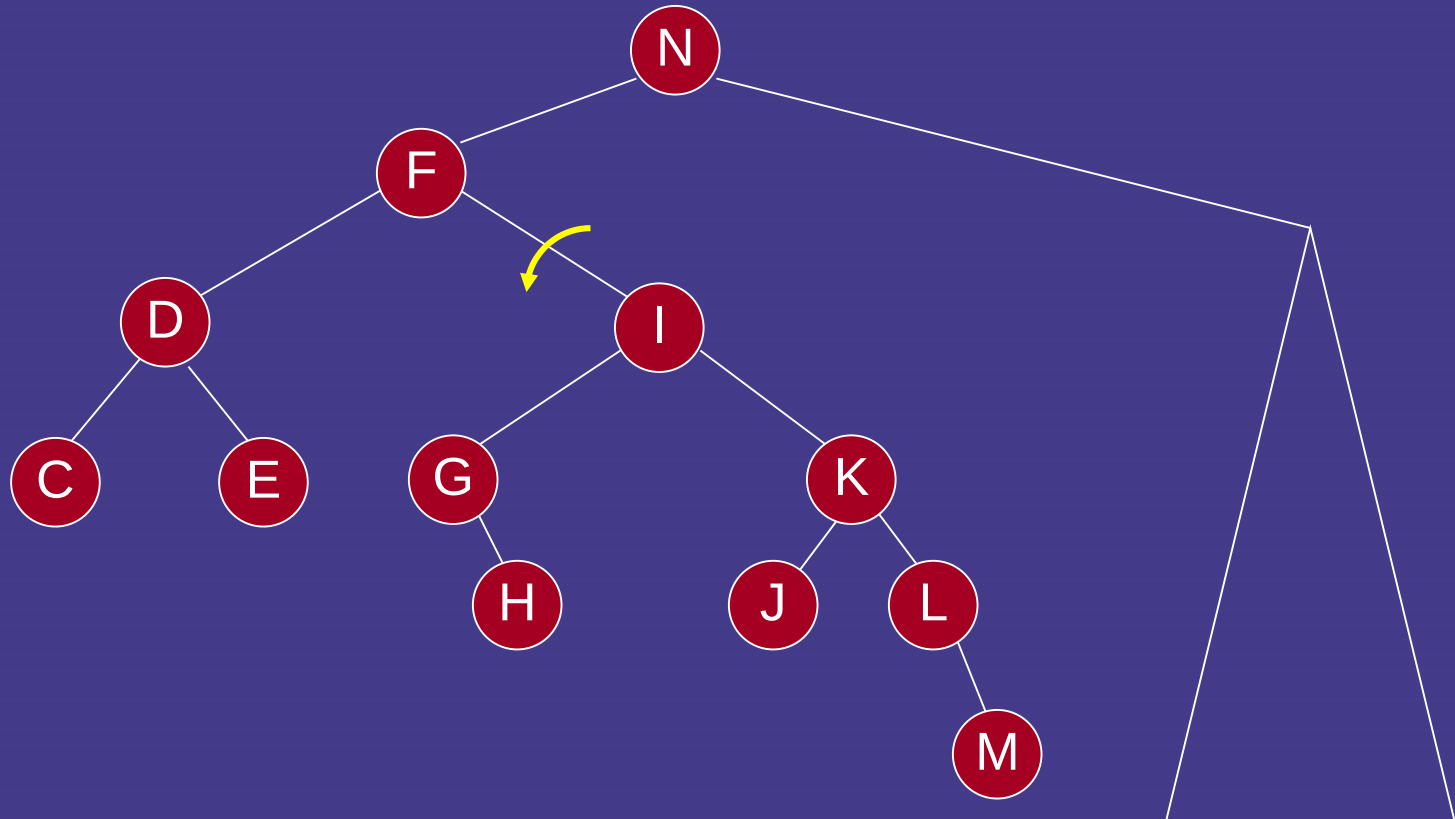
# Deletion in AVL Tree

- The whole of F's left subtree gets shorter. We fix this by rotation about F-I: F down, I up.



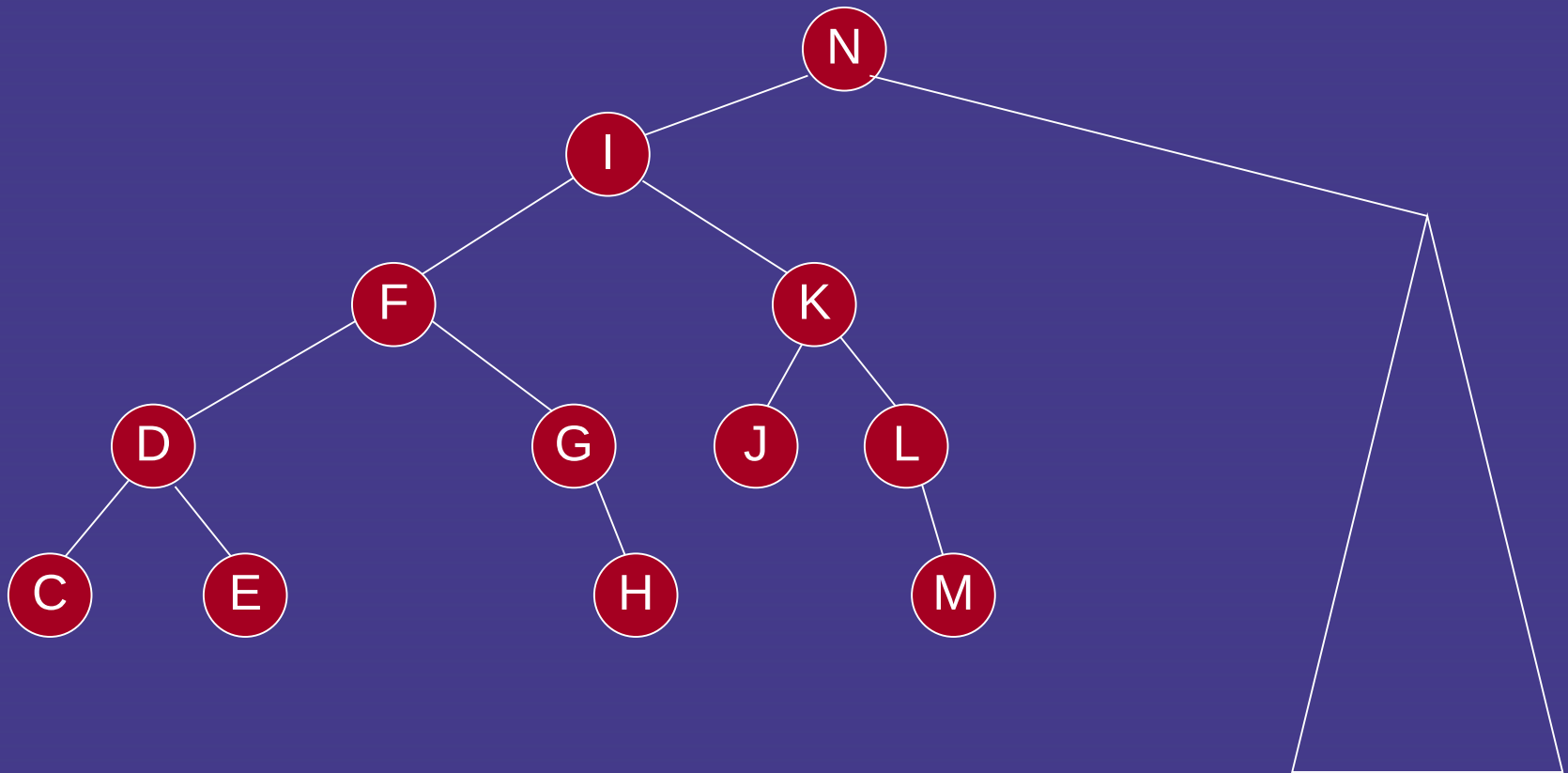
# Deletion in AVL Tree

- The whole of F's left subtree gets shorter. We fix this by rotation about F-I: F down, I up.



# Deletion in AVL Tree

- This could cause imbalance at N.
- The rotations propagated to the root.





# Deletion in AVL Tree

## *Procedure*

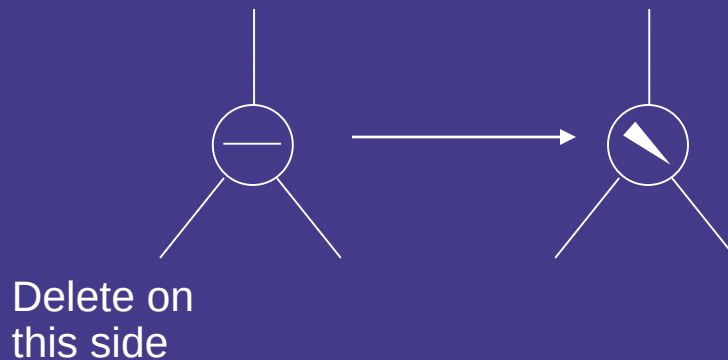
- Delete the node as in binary search tree (BST).
- The node deleted will be either a leaf or have just one subtree.
- Since this is an AVL tree, if the deleted node has one subtree, that subtree contains only one node (why?)
- Traverse up the tree from the deleted node checking the balance of each node.

# Deletion in AVL Tree

- There are 5 cases to consider.
- Let us go through the cases graphically and determine what action to take.
- We will not develop the C++ code for deleteNode in AVL tree. This will be left as an exercise.

# Deletion in AVL Tree

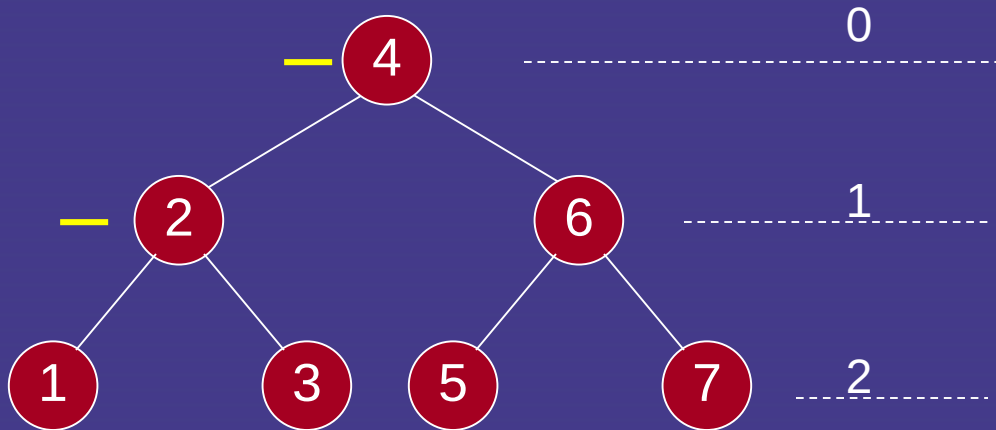
*Case 1a:* the parent of the deleted node had a balance of 0 and the node was deleted in the parent's *left* subtree.



*Action:* change the balance of the parent node and stop. No further effect on balance of any higher node.

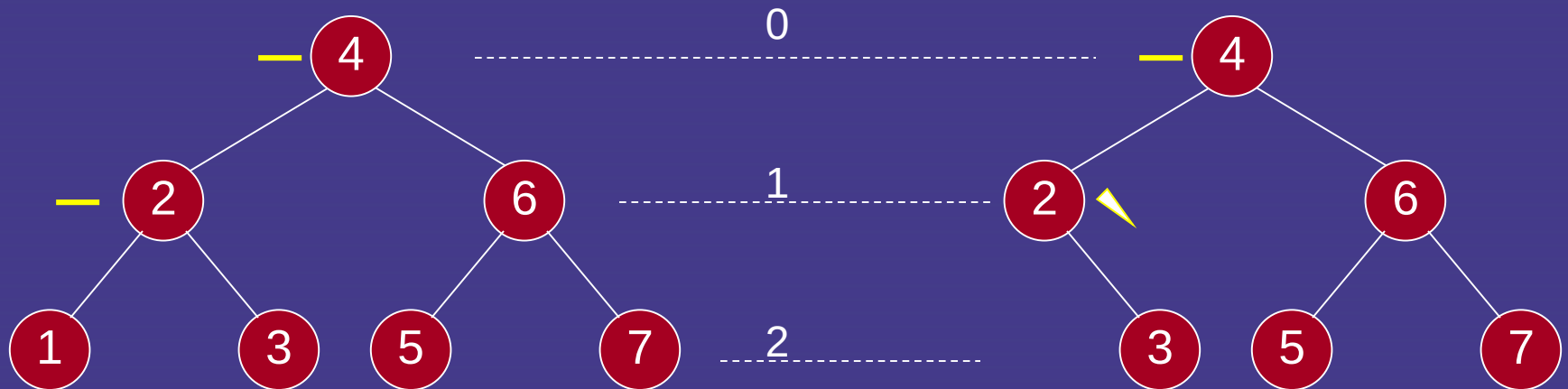
# Deletion in AVL Tree

Here is why; the height of left tree does not change.



# Deletion in AVL Tree

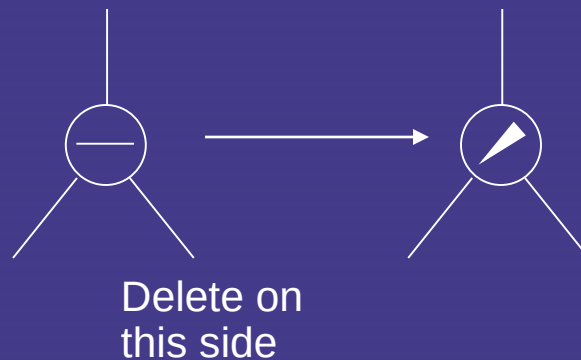
Here is why; the height of left tree does not change.



**remove(1)**

# Deletion in AVL Tree

*Case 1b:* the parent of the deleted node had a balance of 0 and the node was deleted in the parent's *right* subtree.



*Action:* (same as 1a) change the balance of the parent node and stop. No further effect on balance of any higher node.

# Deletion in AVL Tree

*Case 2a:* the parent of the deleted node had a balance of 1 and the node was deleted in the parent's *left* subtree.



*Action:* change the balance of the parent node. May have caused imbalance in higher nodes so continue up the tree.

# Deletion in AVL Tree

- There are 5 cases to consider.
- Let us go through the cases graphically and determine what action to take.
- We will not develop the C++ code for deleteNode in AVL tree. This will be left as an exercise.



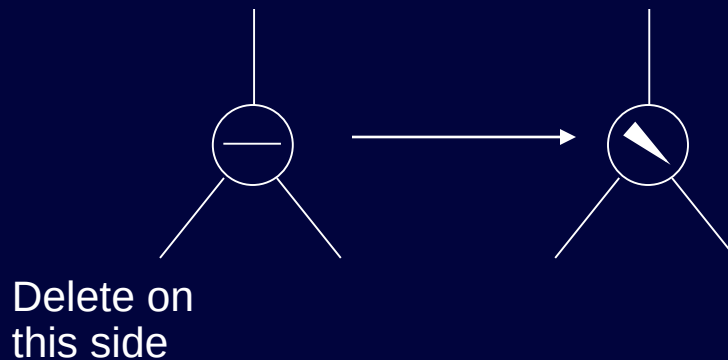
Lecture No.24

Data Structure

Dr. Sohail Aslam

# Deletion in AVL Tree

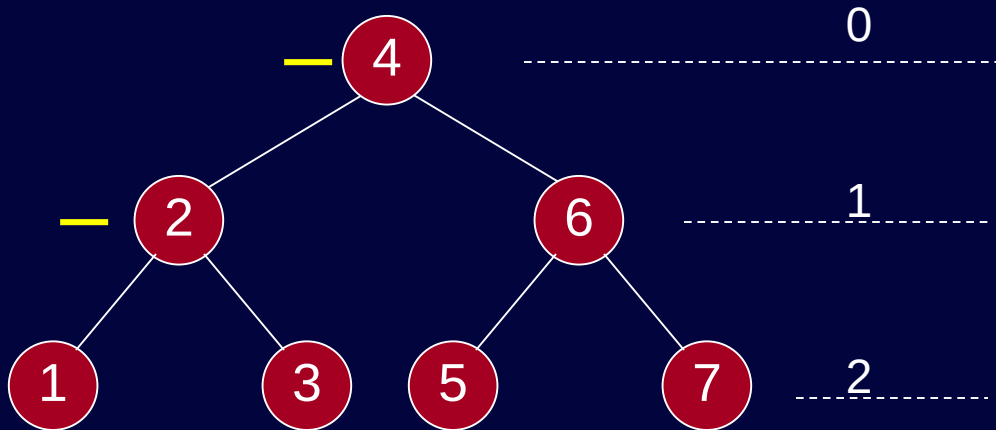
*Case 1a:* the parent of the deleted node had a balance of 0 and the node was deleted in the parent's *left* subtree.



*Action:* change the balance of the parent node and stop. No further effect on balance of any higher node.

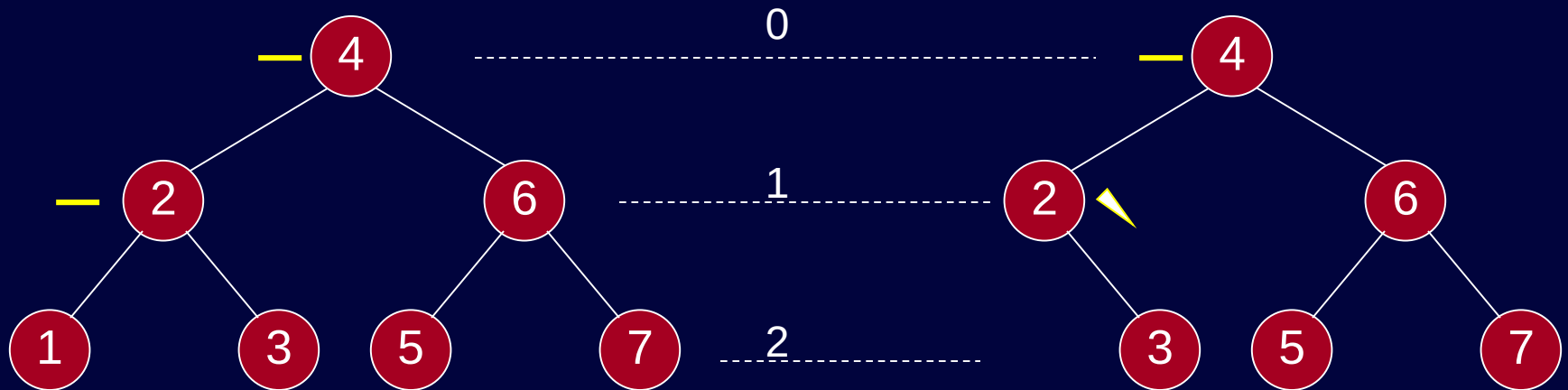
# Deletion in AVL Tree

Here is why; the height of left tree does not change.



# Deletion in AVL Tree

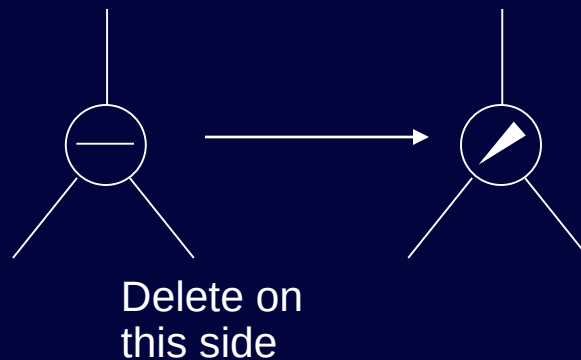
Here is why; the height of left tree does not change.



**remove(1)**

# Deletion in AVL Tree

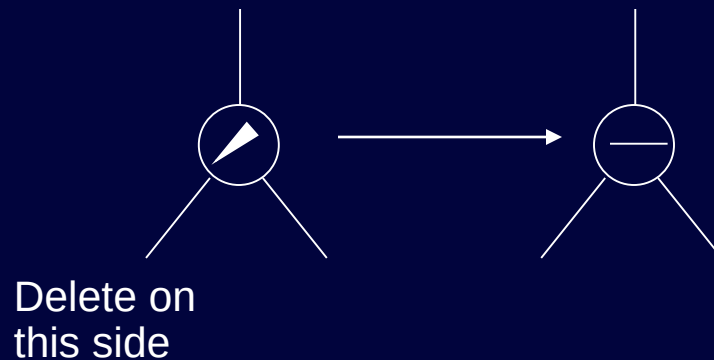
*Case 1b:* the parent of the deleted node had a balance of 0 and the node was deleted in the parent's *right* subtree.



*Action:* (same as 1a) change the balance of the parent node and stop. No further effect on balance of any higher node.

# Deletion in AVL Tree

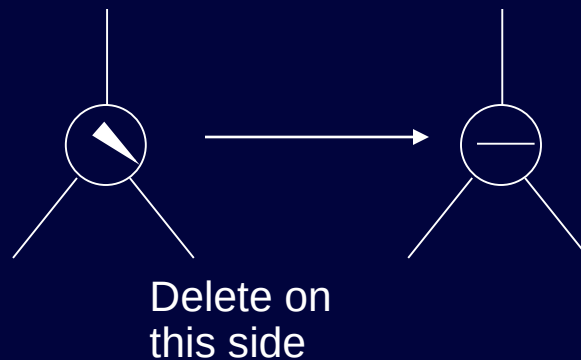
*Case 2a:* the parent of the deleted node had a balance of 1 and the node was deleted in the parent's *left* subtree.



*Action:* change the balance of the parent node. May have caused imbalance in higher nodes so continue up the tree.

# Deletion in AVL Tree

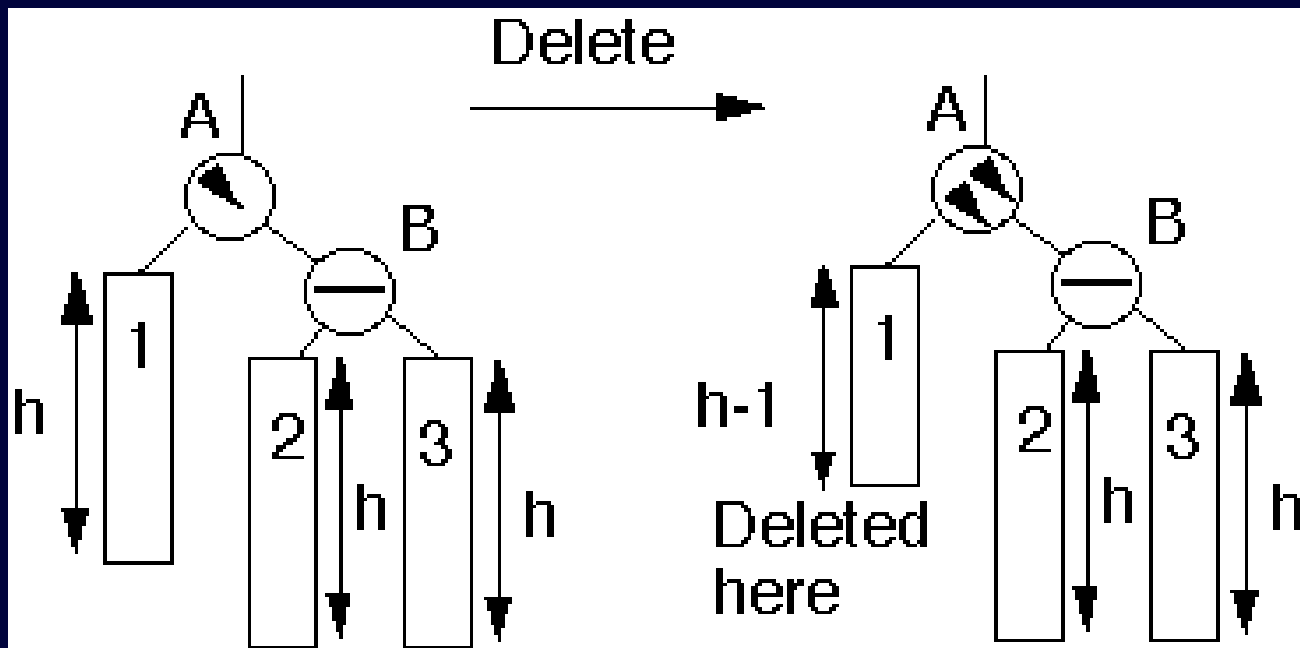
*Case 2b:* the parent of the deleted node had a balance of -1 and the node was deleted in the parent's *right* subtree.



*Action:* same as 2a: change the balance of the parent node. May have caused imbalance in higher nodes so continue up the tree.

# Deletion in AVL Tree

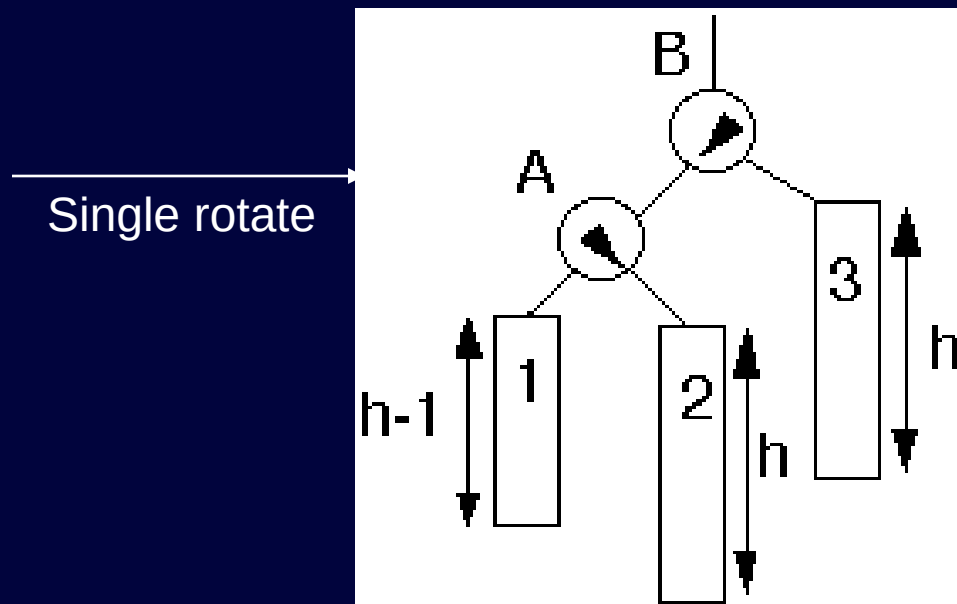
Case 3a: the parent had balance of -1 and the node was deleted in the parent's *left* subtree, right subtree was balanced.





# Deletion in AVL Tree

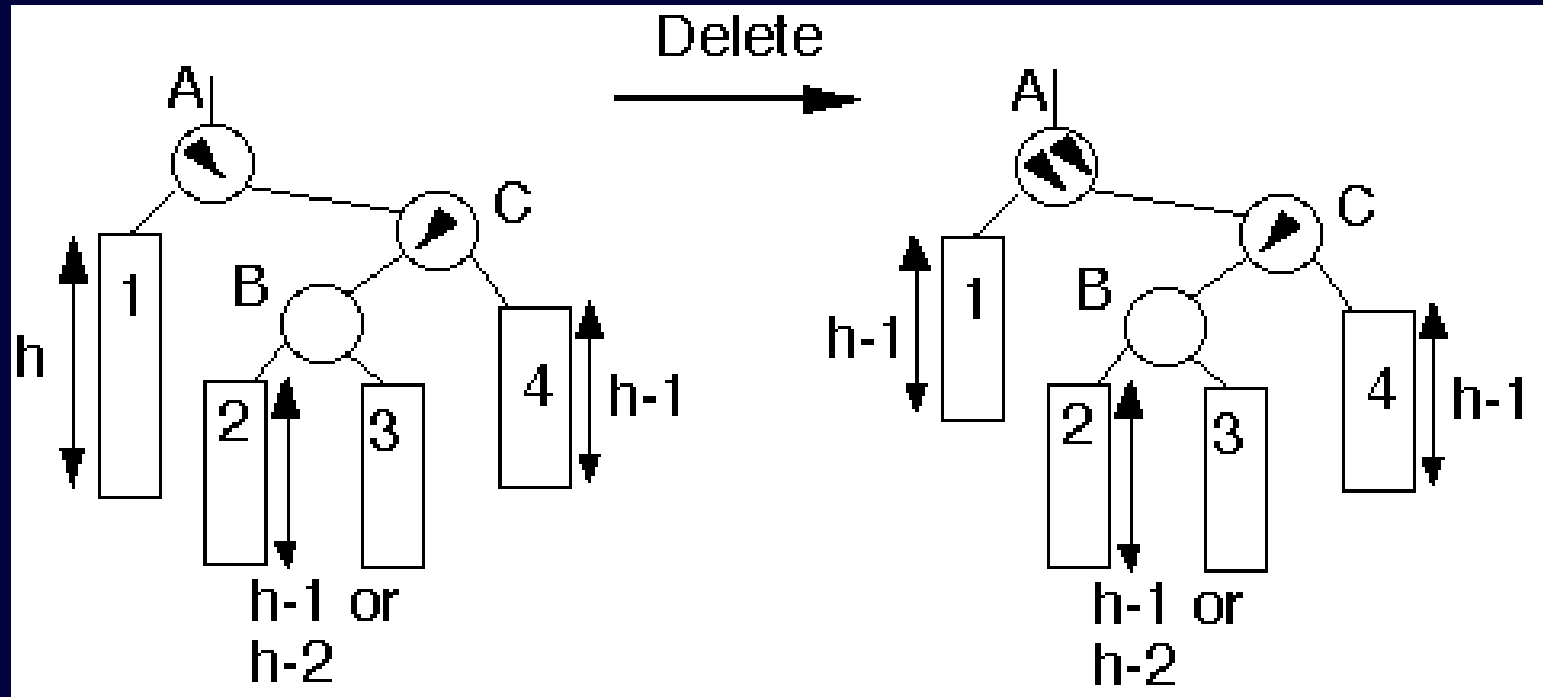
Case 3a: the parent had balance of -1 and the node was deleted in the parent's *left* subtree, right subtree was balanced.



*Action:* perform single rotation, adjust balance. No effect on balance of higher nodes so stop here.

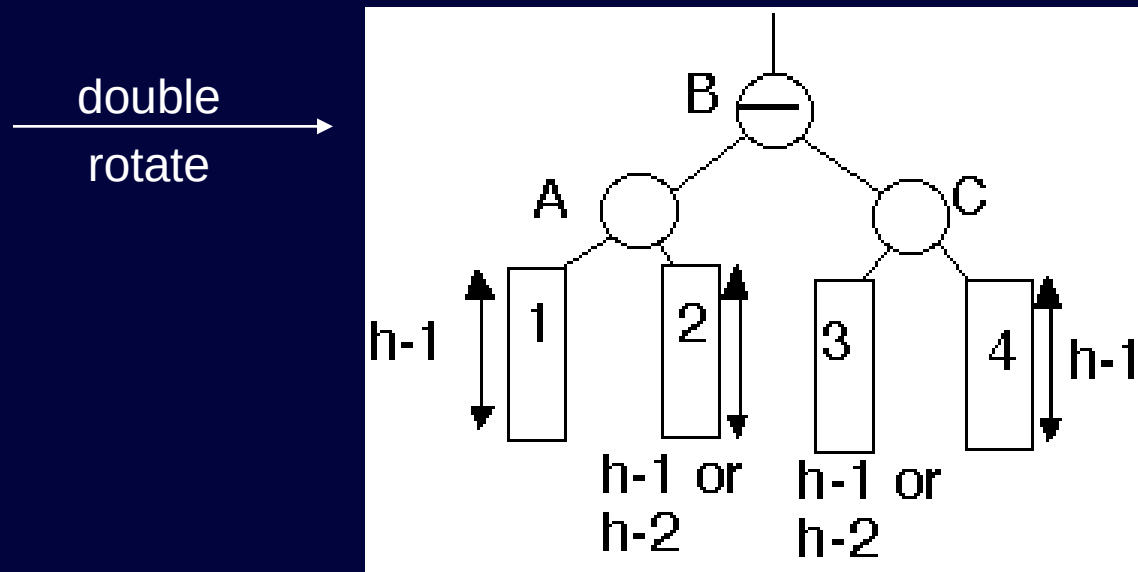
# Deletion in AVL Tree

Case 4a: parent had balance of -1 and the node was deleted in the parent's *left* subtree, right subtree was unbalanced.



# Deletion in AVL Tree

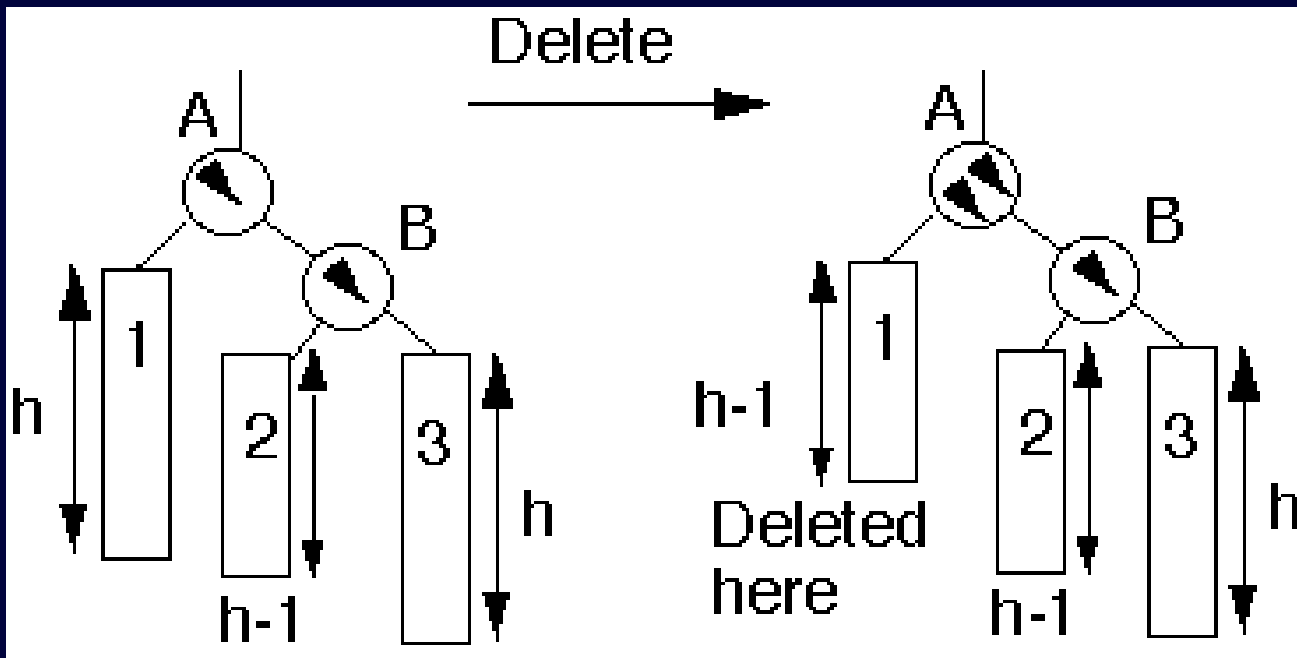
*Case 4a:* parent had balance of -1 and the node was deleted in the parent's *left* subtree, right subtree was unbalanced.



*Action:* Double rotation at B. May have effected the balance of higher nodes, so continue up the tree.

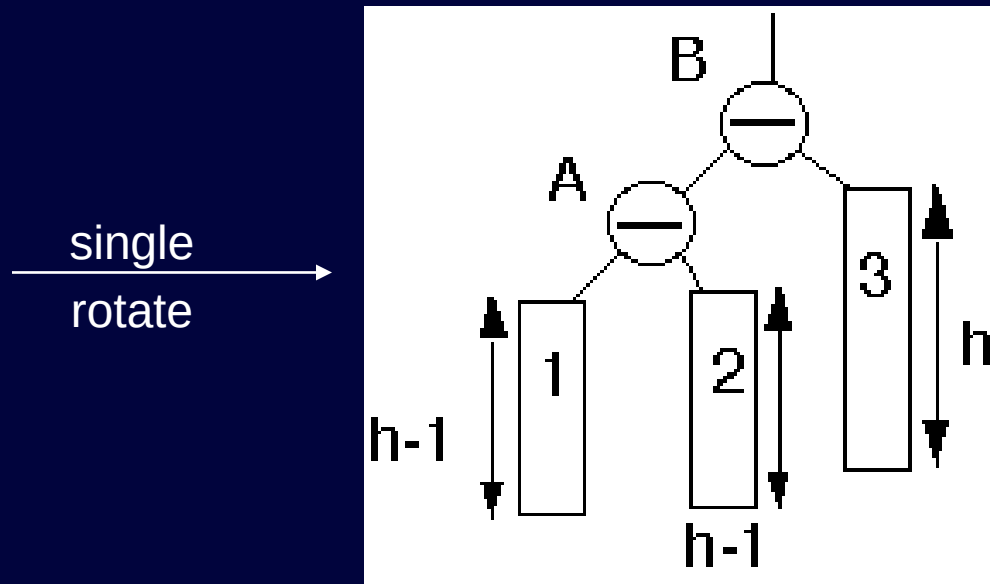
# Deletion in AVL Tree

Case 5a: parent had balance of -1 and the node was deleted in the parent's *left* subtree, right subtree was unbalanced.



# Deletion in AVL Tree

*Case 5a*: parent had balance of -1 and the node was deleted in the parent's *left* subtree, right subtree was unbalanced.



*Action*: Single rotation at B. May have effected the balance of higher nodes, so continue up the tree.

# Other Uses of Binary Trees

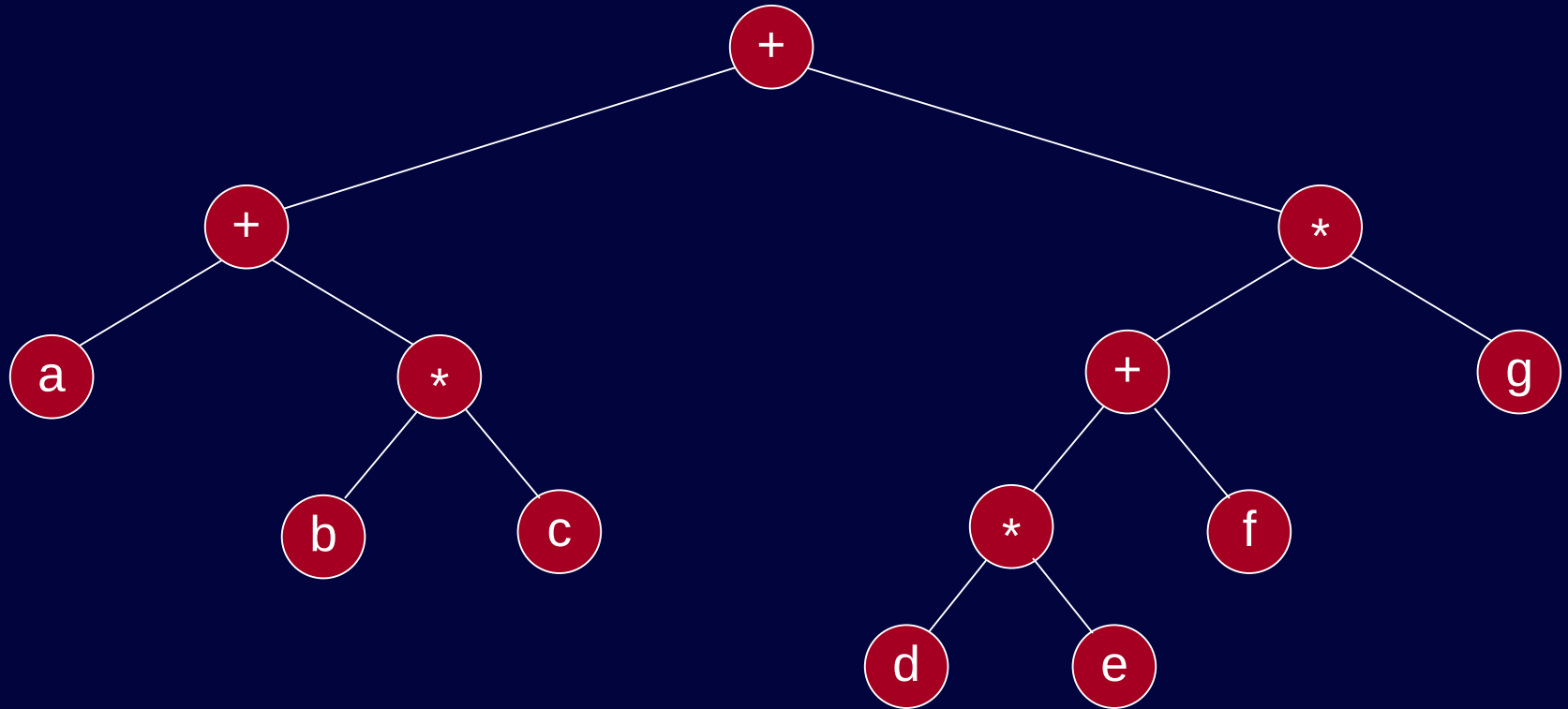
## Expression Trees

# Expression Trees

- *Expression trees*, and the more general parse trees and abstract syntax trees are significant components of compilers.
- Let us consider the expression tree.

# Expression Tree

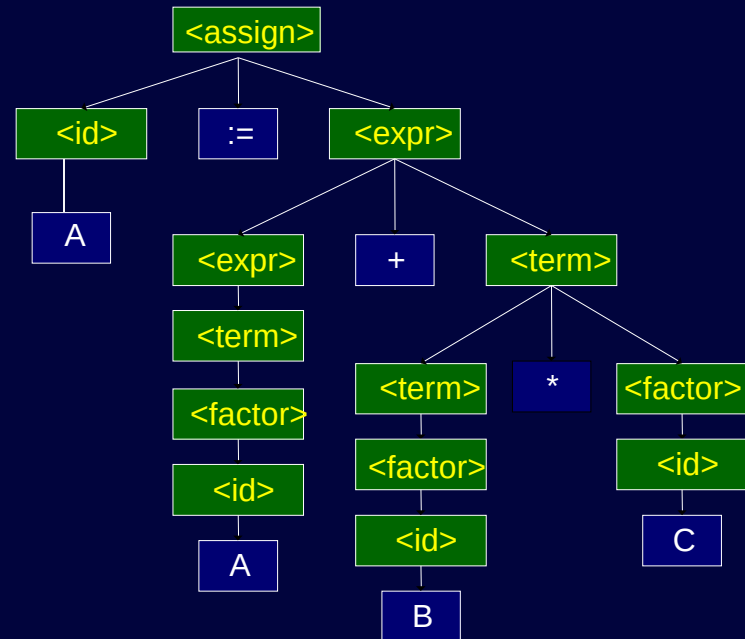
$(a + b * c) + ((d * e + f) * g)$





# Parse Tree in Compiler

$A := A + B * C$



Expression grammar

$\langle\text{assign}\rangle \rightarrow \langle\text{id}\rangle := \langle\text{expr}\rangle$

$\langle\text{id}\rangle \rightarrow A \mid B \mid C$

$\langle\text{expr}\rangle \rightarrow \langle\text{expr}\rangle + \langle\text{term}\rangle \mid$   
 $\langle\text{term}\rangle$

$\langle\text{term}\rangle \rightarrow \langle\text{term}\rangle * \langle\text{factor}\rangle \mid$   
 $\langle\text{factor}\rangle$

$\langle\text{factor}\rangle \rightarrow ( \langle\text{expr}\rangle ) \mid \langle\text{id}\rangle$

# Parse Tree for an SQL query

Consider querying a movie database

*Find the titles for movies with stars born in 1960*

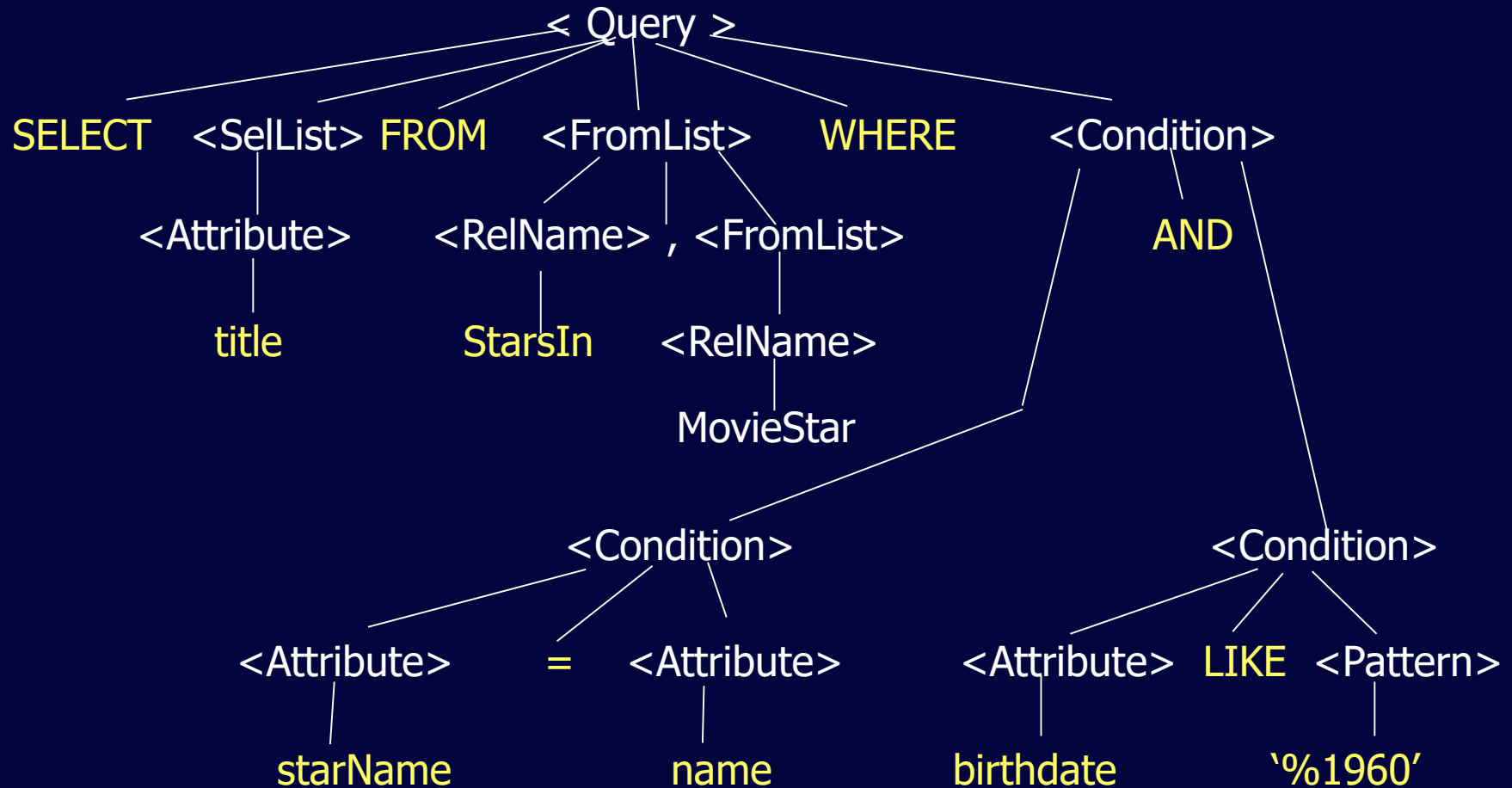
The database has tables

StarsIn(title, year, starName)

MovieStar(name, address, gender, birthdate)

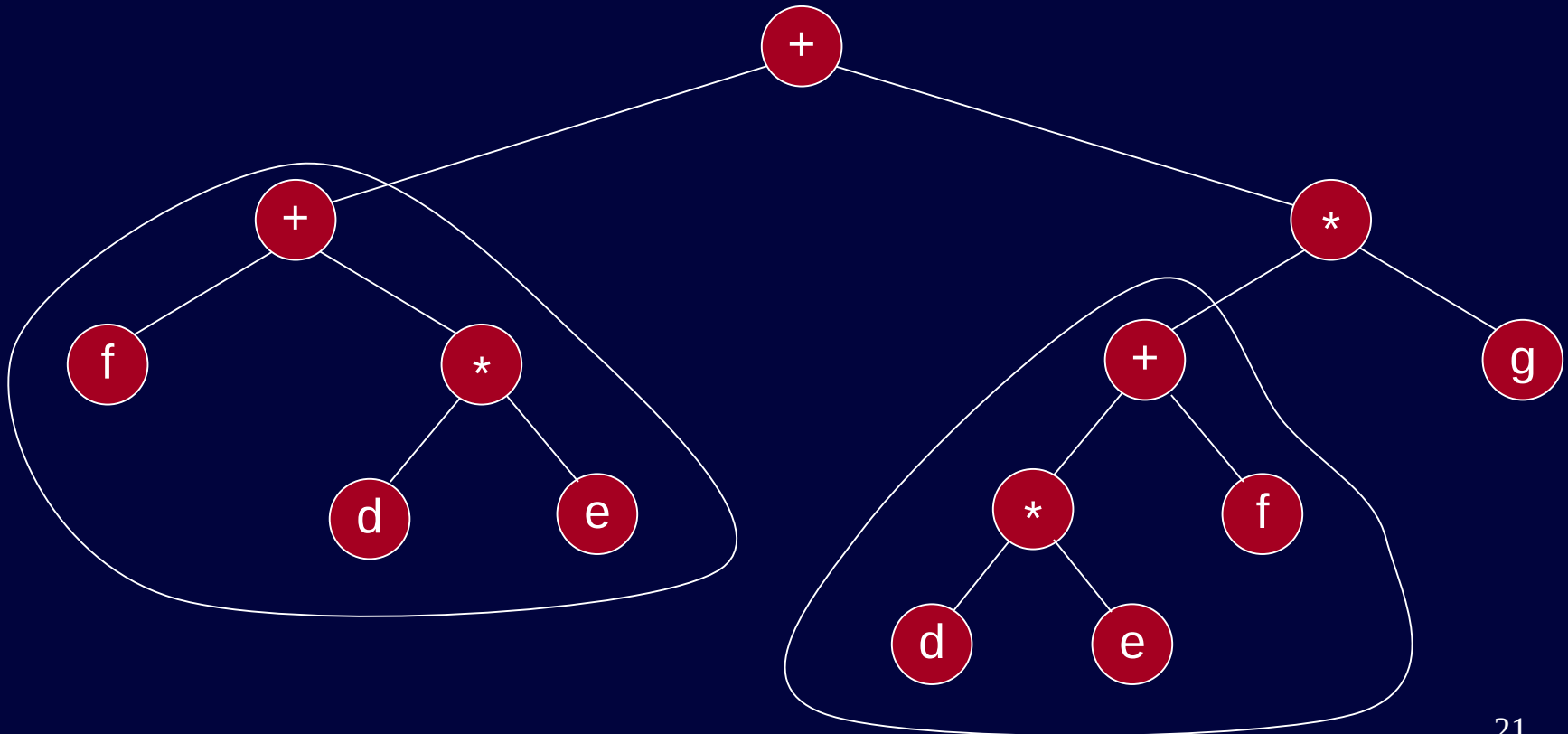
```
SELECT title
FROM   StarsIn, MovieStar
WHERE  starName = name AND birthdate LIKE '%1960' ;
```

# SQL Parse Tree



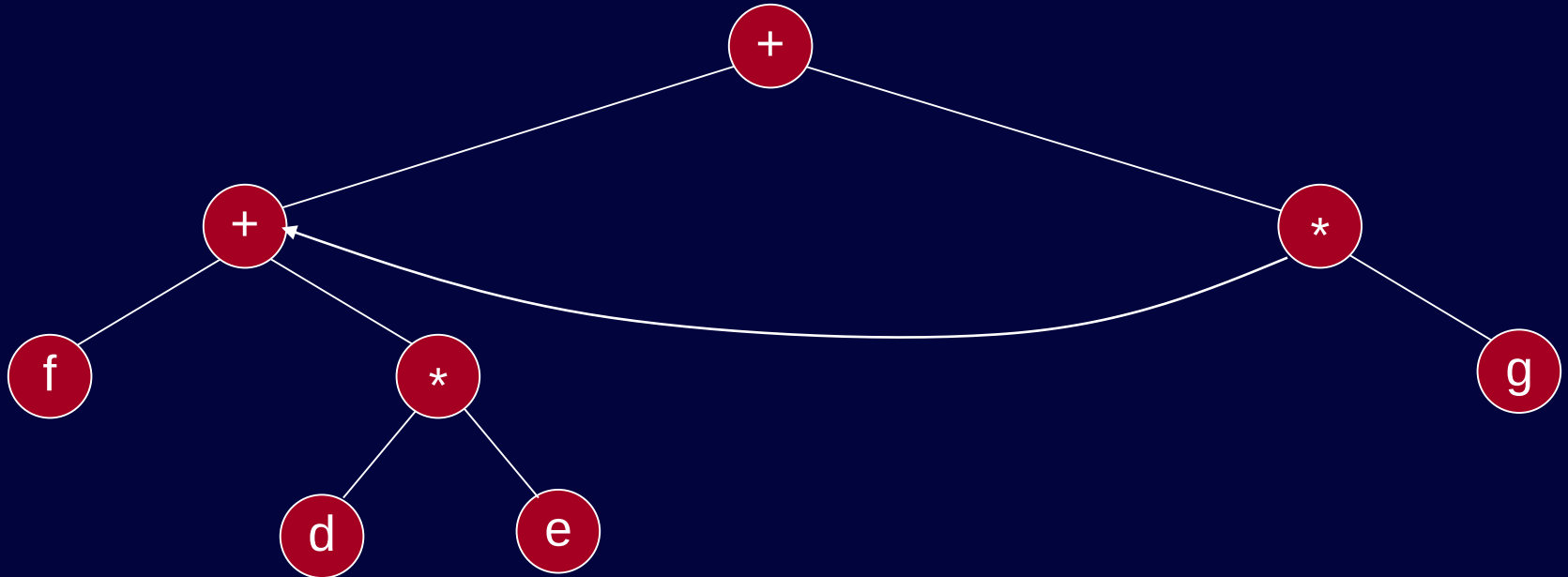
# Compiler Optimization

Common subexpression:  
 $(f + d * e) + ((d * e + f) * g)$



# Compiler Optimization

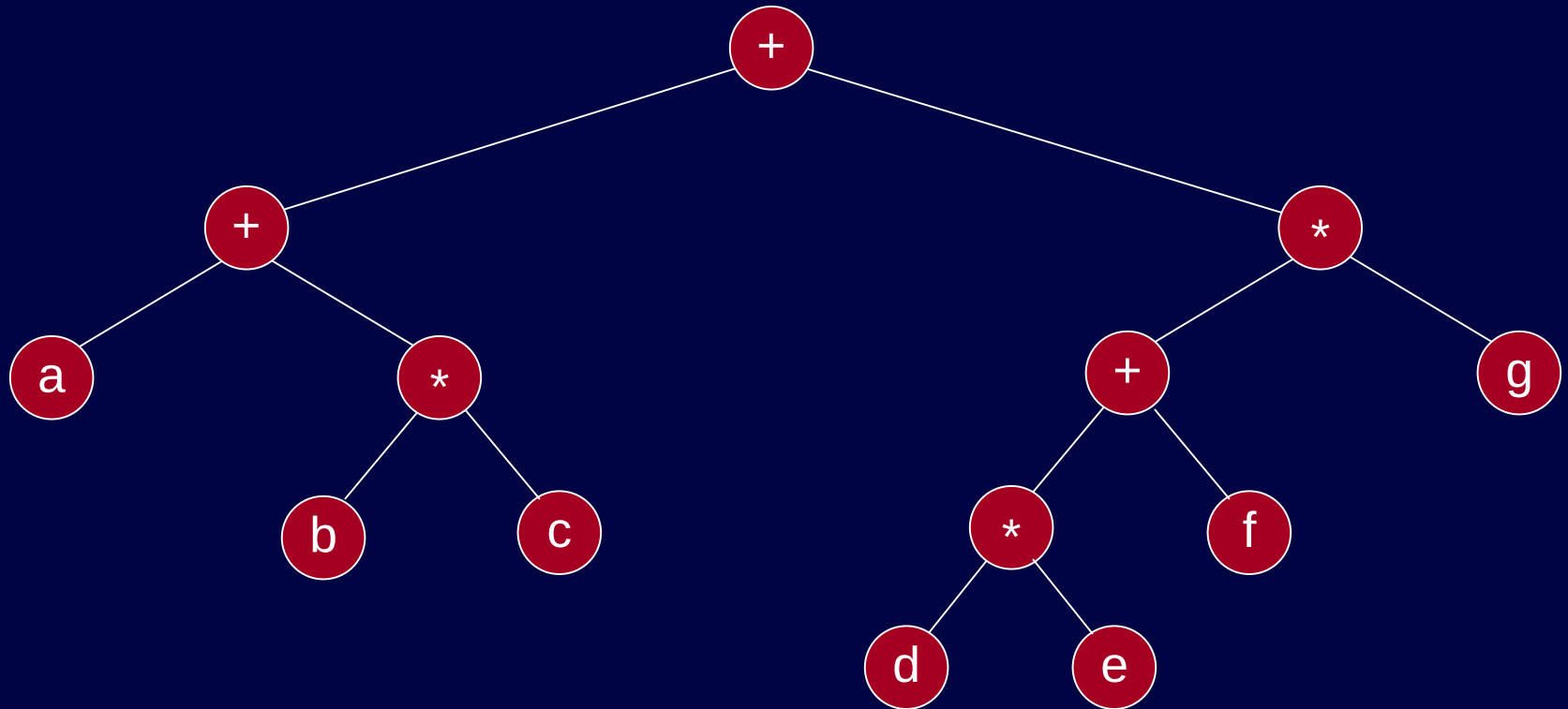
(Common subexpression:  
 $(f + d * e) + ((d * e + f) * g)$ )



Graph!

# Expression Tree

- The inner nodes contain operators while leaf nodes contain operands.



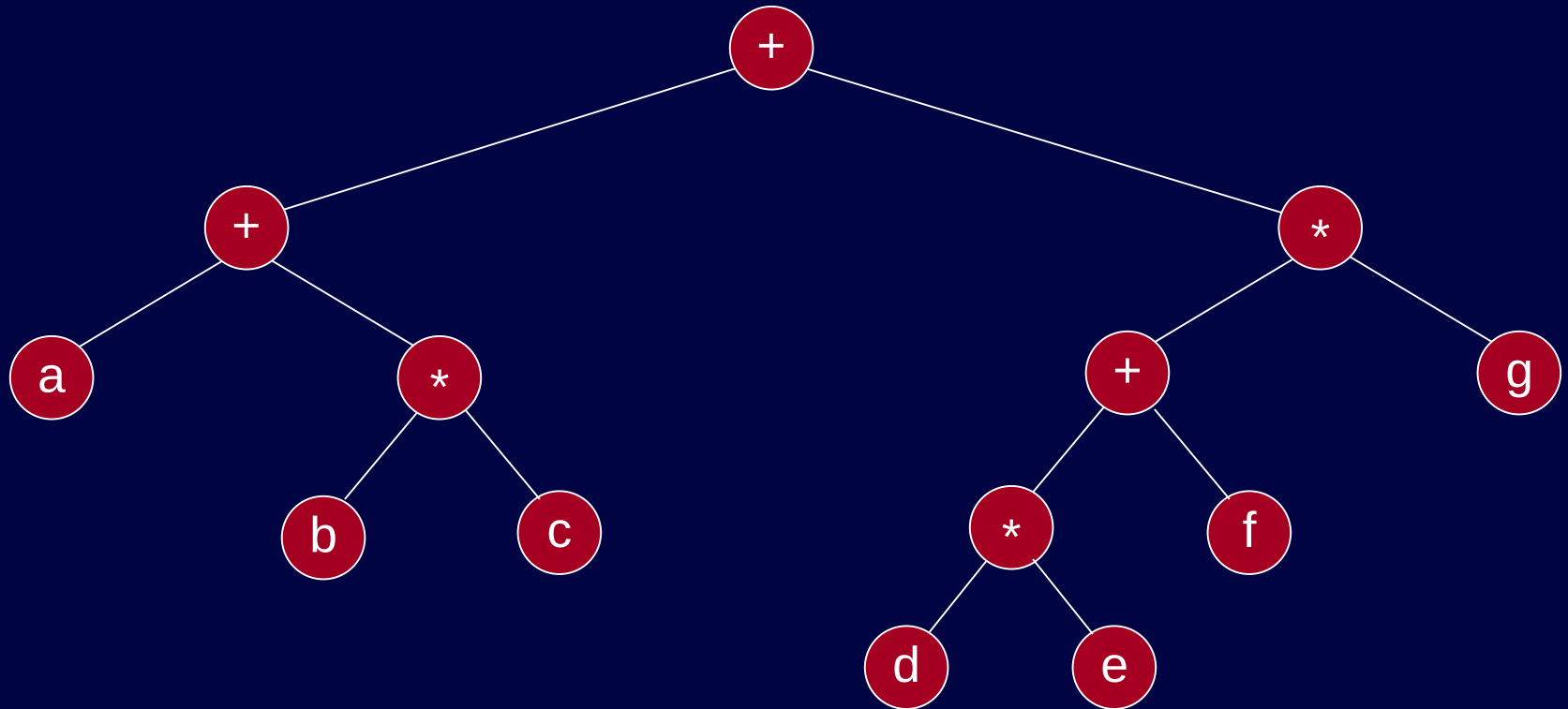
# Lecture No.25

## Data Structures

Dr. Sohail Aslam

# Expression Tree

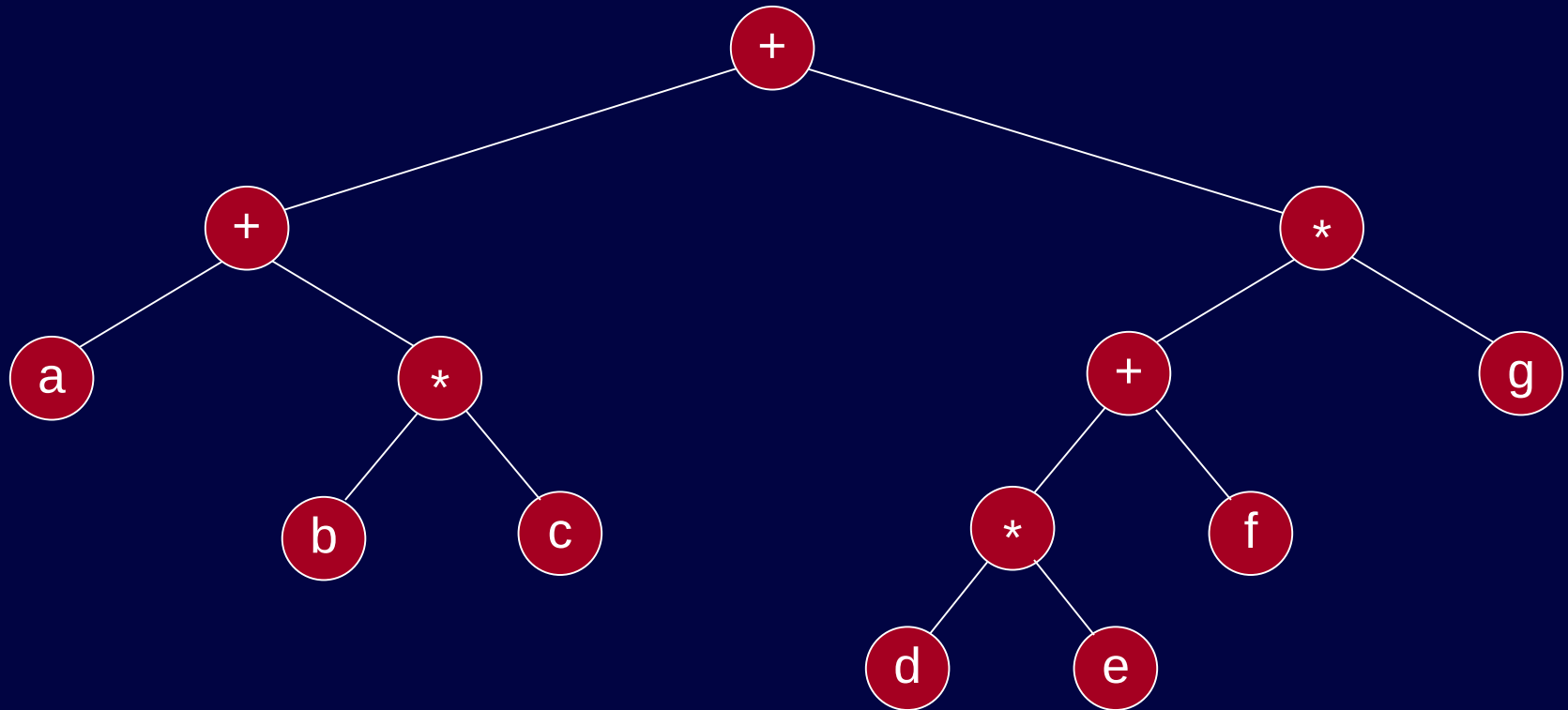
- The tree is binary because the operators are binary.





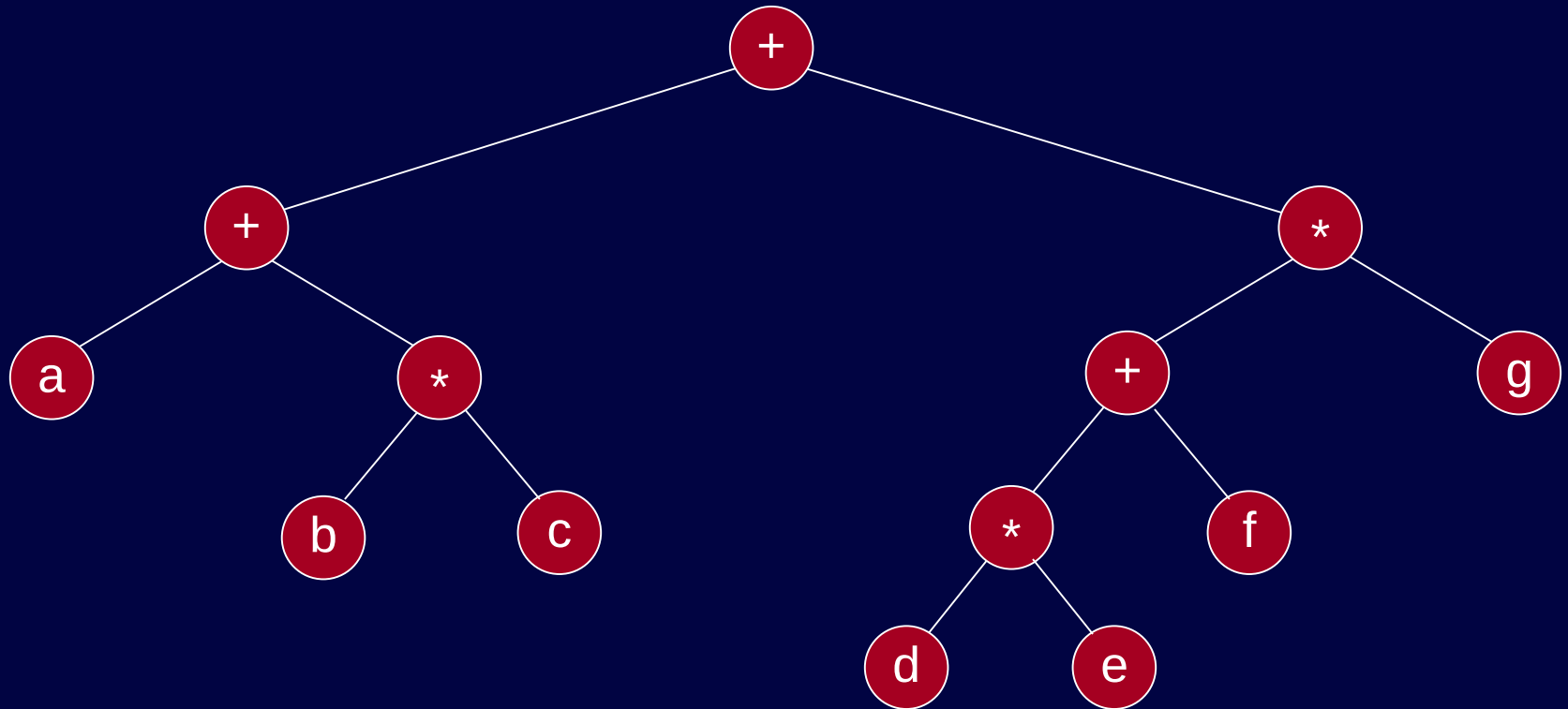
# Expression Tree

- This is not necessary. A unary operator (!, e.g.) will have only one subtree.



# Expression Tree

- Inorder traversal yields:  $a+b*c+d*e+f*g$

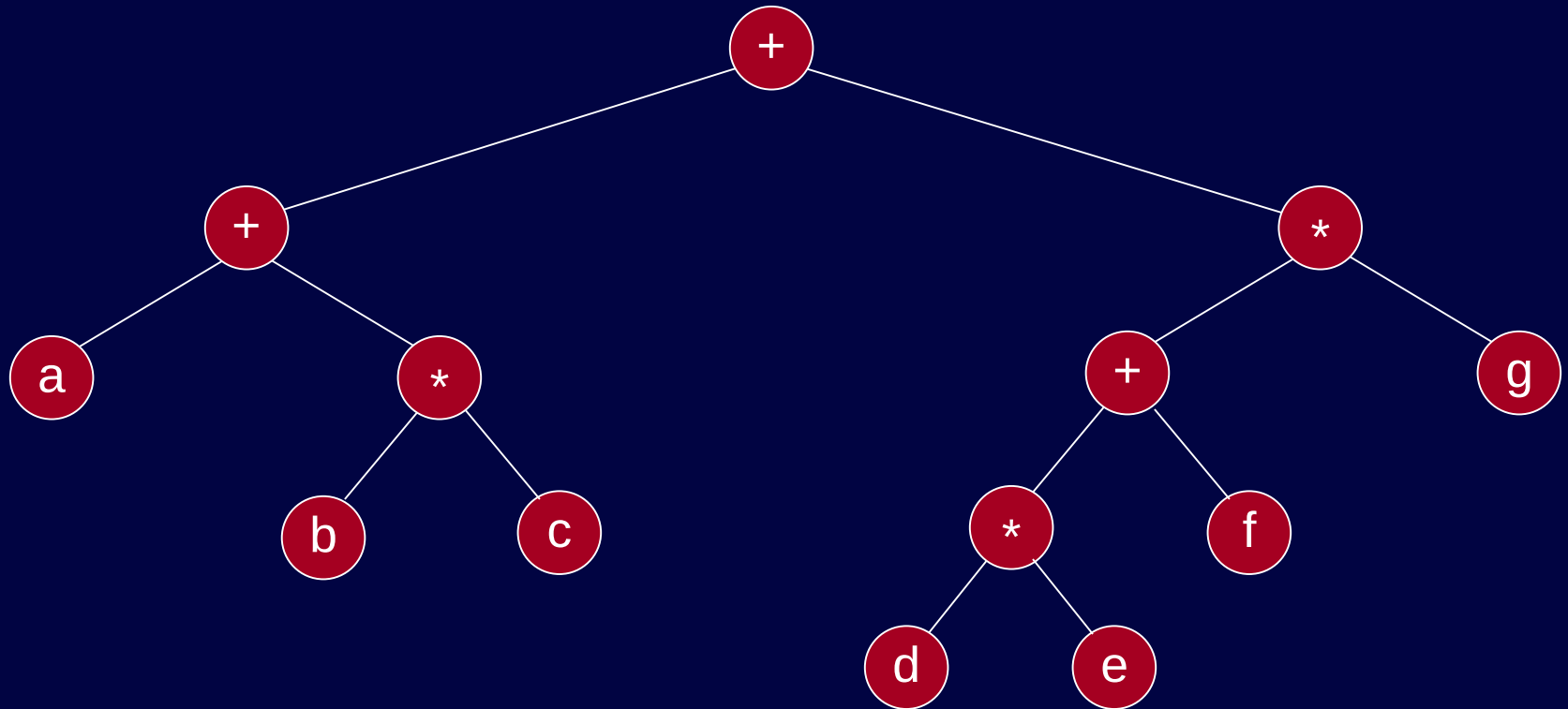


# Enforcing Parenthesis

```
/* inorder traversal routine using the
   parenthesis */
void inorder(TreeNode<int>* treeNode)
{
    if( treeNode != NULL )
    {
        if(treeNode->getLeft() != NULL && treeNode->getRight() != NULL) //if not leaf
            cout<<" (" ;
        inorder(treeNode->getLeft());
        cout << *(treeNode->getInfo())<<" ";
        inorder(treeNode->getRight());
        if(treeNode->getLeft() != NULL && treeNode->getRight() != NULL) //if not leaf
            cout<<") " ;
    }
}
```

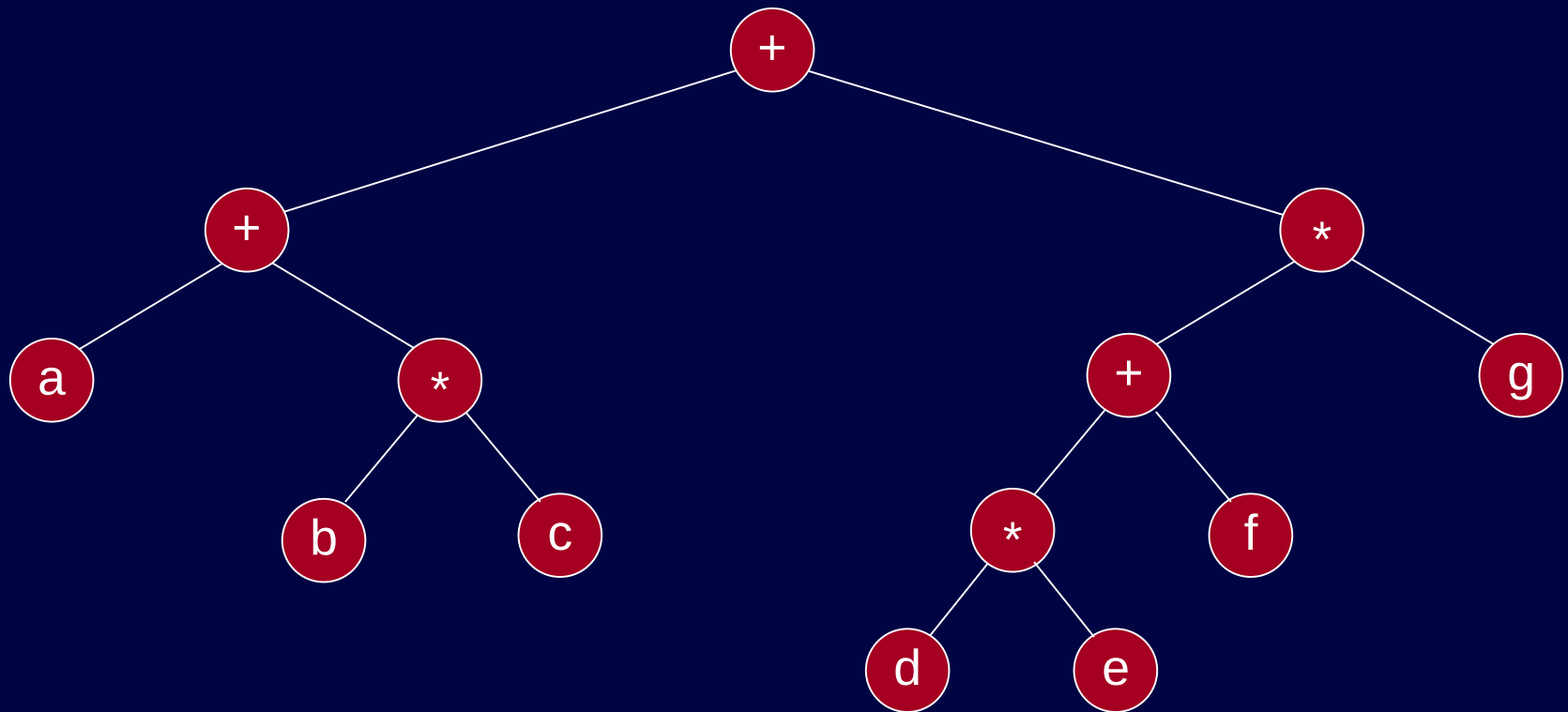
# Expression Tree

- Inorder :  $(a+(b*c))+(((d*e)+f)*g)$



# Expression Tree

- Postorder traversal:  $a\ b\ c\ *\ +\ d\ e\ *\ f\ +\ g\ *\ +$   
which is the postfix form.



# Constructing Expression Tree

- Algorithm to convert postfix expression into an expression tree.
- We already have an expression to convert an infix expression to postfix.
- Read a symbol from the postfix expression.
- If symbol is an operand, put it in a one node tree and push it on a stack.
- 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.

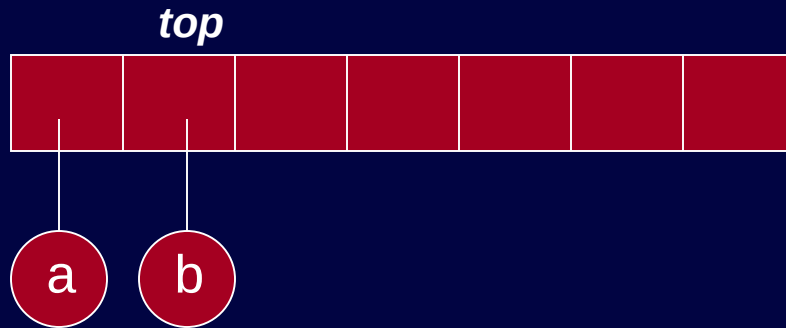
# Constructing Expression Tree

▪ a b + c d e + \* \*



# Constructing Expression Tree

■  $a b + c d e + * *$



If symbol is an operand, put it in a one node tree and push it on a stack.

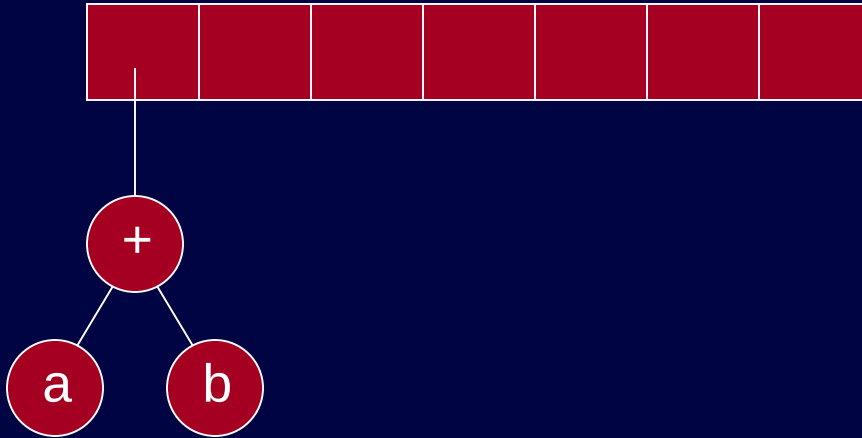
---

*Stack is growing left to right*



# Constructing Expression Tree

■ a b + c d e + \* \*



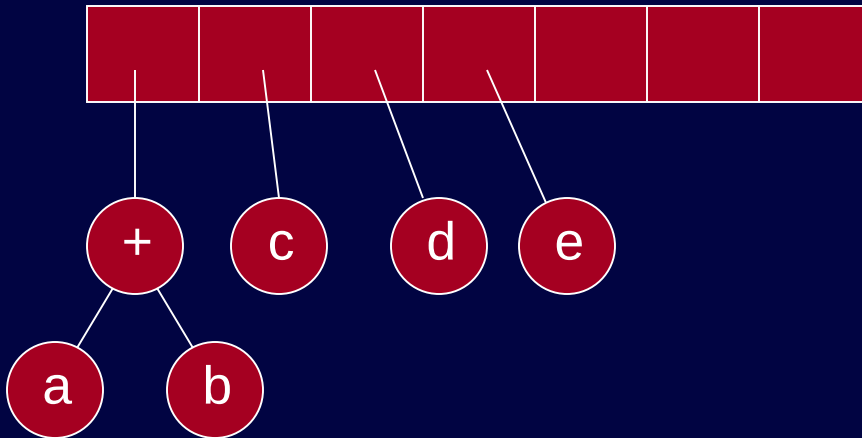
If symbol is an operator, pop two trees from the stack, form a new tree with operator as the root and  $T_1$  and  $T_2$  as left and right subtrees and push this tree on the stack.

---

*Stack is growing left to right*

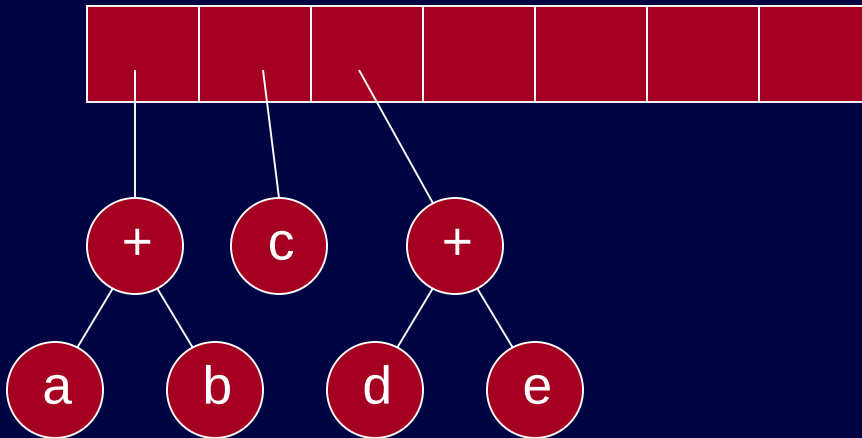
# Constructing Expression Tree

■  $a b + c d e + * *$



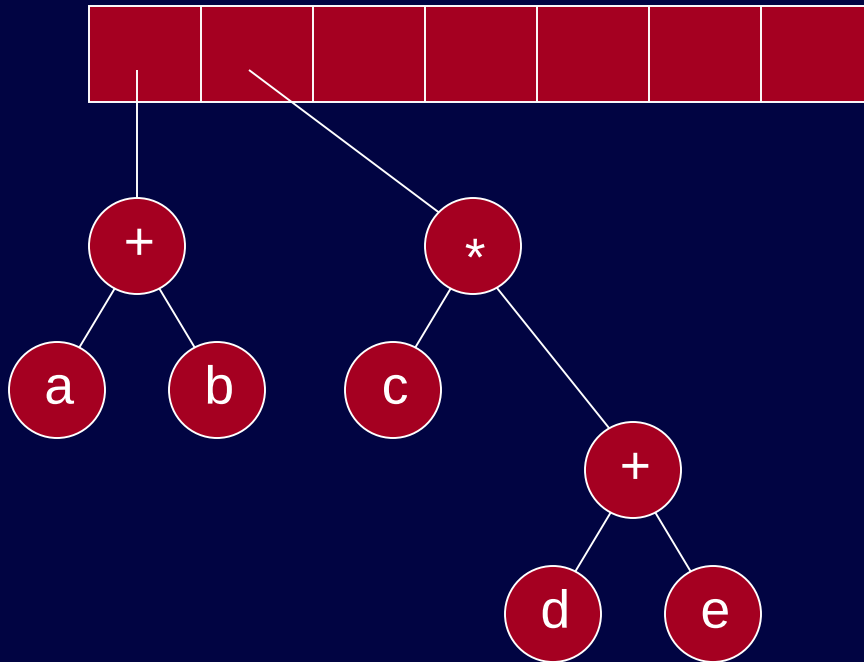
# Constructing Expression Tree

■  $a b + c d e + * *$



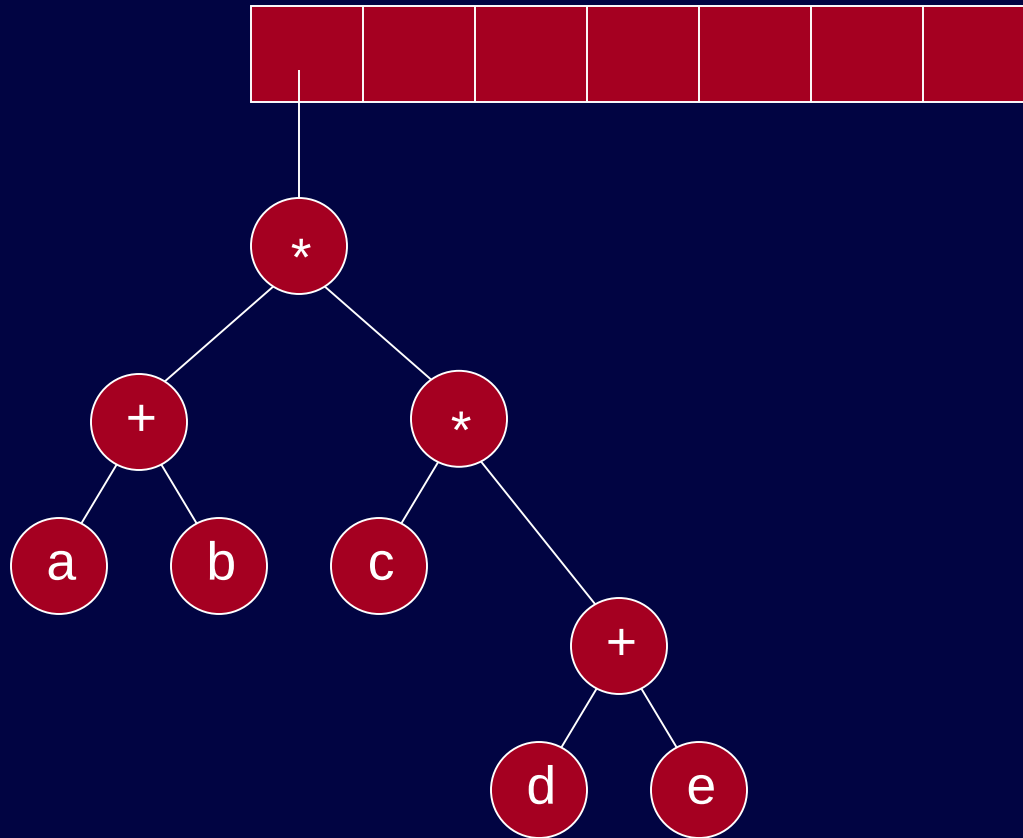
# Constructing Expression Tree

■  $a b + c d e + * *$



# Constructing Expression Tree

■  $a b + c d e + * *$



# Other Uses of Binary Trees

Huffman Encoding

# 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 to simply send less data.
- Improvements with regard to the transmission media has led to increase in the rate.
- The other options is to send less data by means of data compression.
- Compression methods are used for text, images, voice and other types of data (space probes).

# Huffman Encoding

- Huffman code is method for the compression for 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.



# Huffman Encoding

- To understand Huffman encoding, it is best to use a simple example.
- Encoding the 32-character phrase:  
*"traversing threaded binary trees"*,
- If we send the phrase as a message in a network using standard 8-bit ASCII codes, we would have to send  $8 \times 32 = 256$  bits.
- Using the Huffman algorithm, we can send the message with only 116 bits.

# Huffman Encoding

- 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, and if there is a tie, pick randomly amongst those with equal frequencies.

# Huffman Encoding

- Make a new node out of these two, and make the two nodes 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 until only one node, the root, remains.

# Huffman Encoding

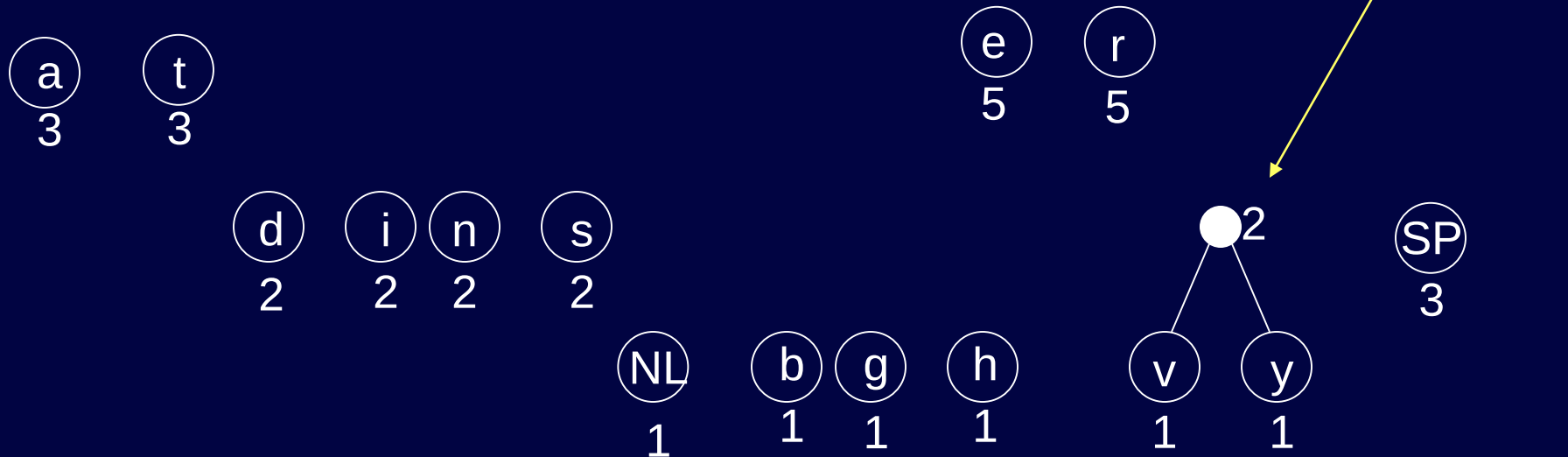
Original text:

*traversing threaded binary trees*

size: 33 characters (space and newline)

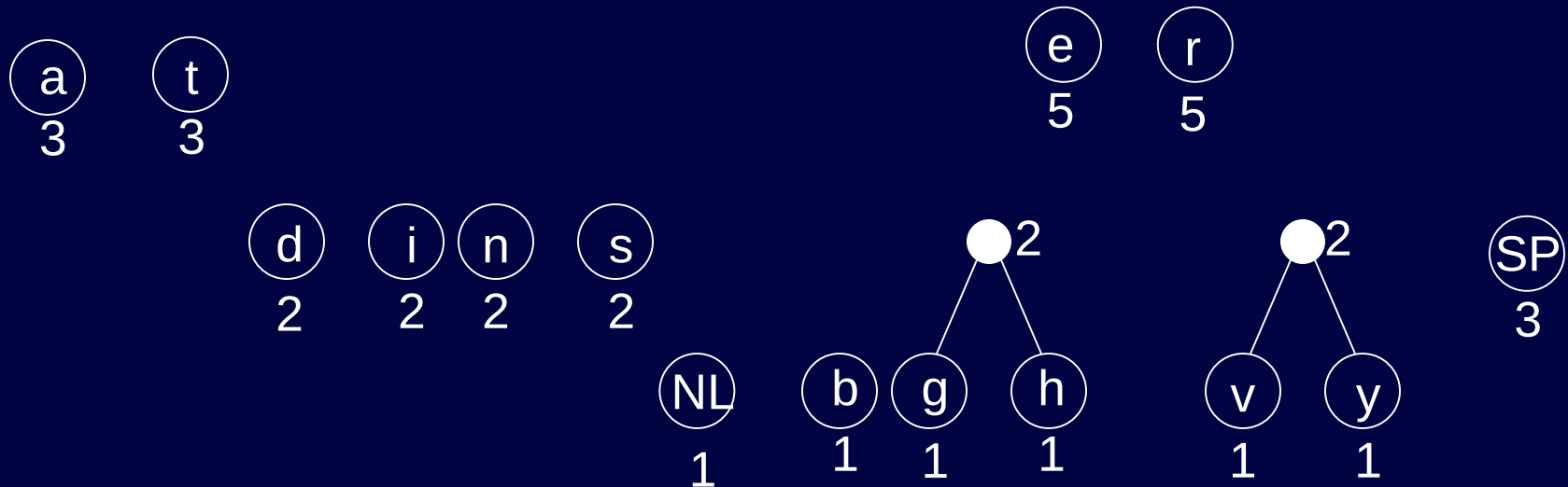
|      |   |     |   |
|------|---|-----|---|
| NL : | 1 | i : | 2 |
| SP : | 3 | n : | 2 |
| a :  | 3 | r : | 5 |
| b :  | 1 | s : | 2 |
| d :  | 2 | t : | 3 |
| e :  | 5 | v : | 1 |
| g :  | 1 | y : | 1 |
| h :  | 1 |     |   |

# Huffman Encoding

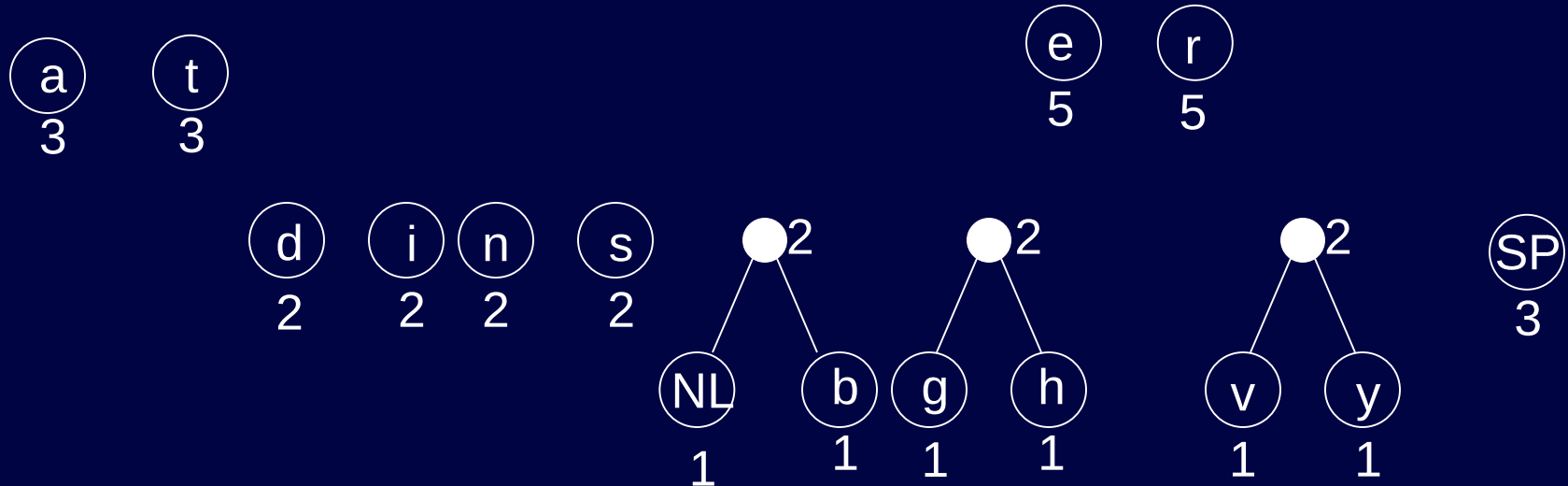


# Huffman Encoding

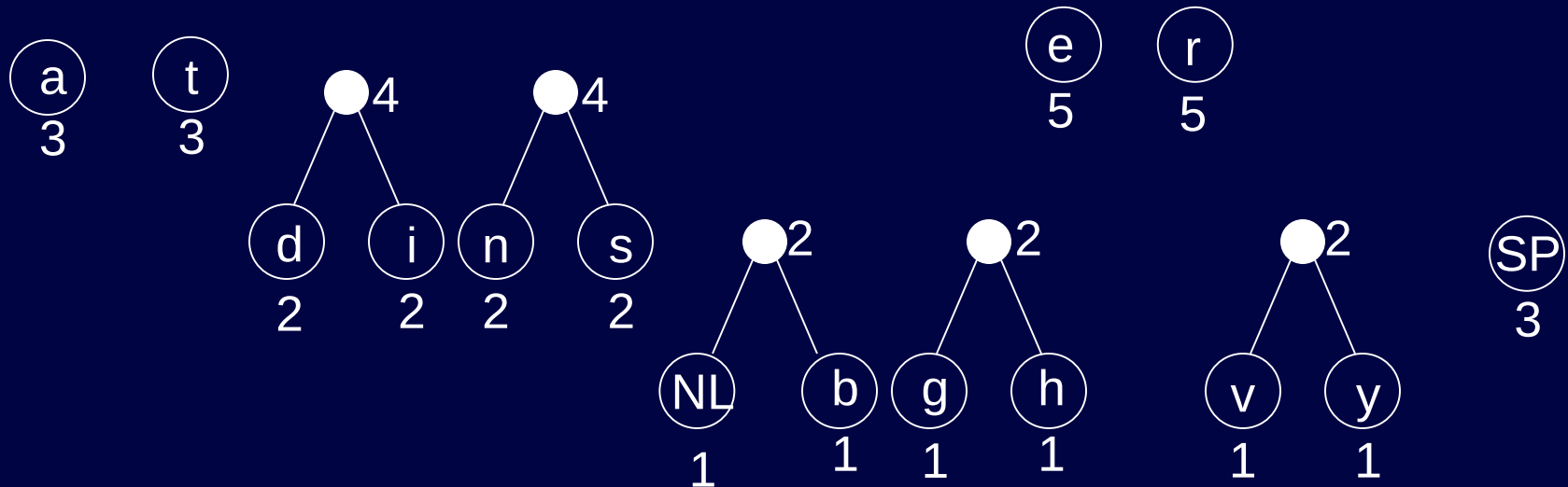
There a number of ways to combine nodes. We have chosen just one such way.



# Huffman Encoding

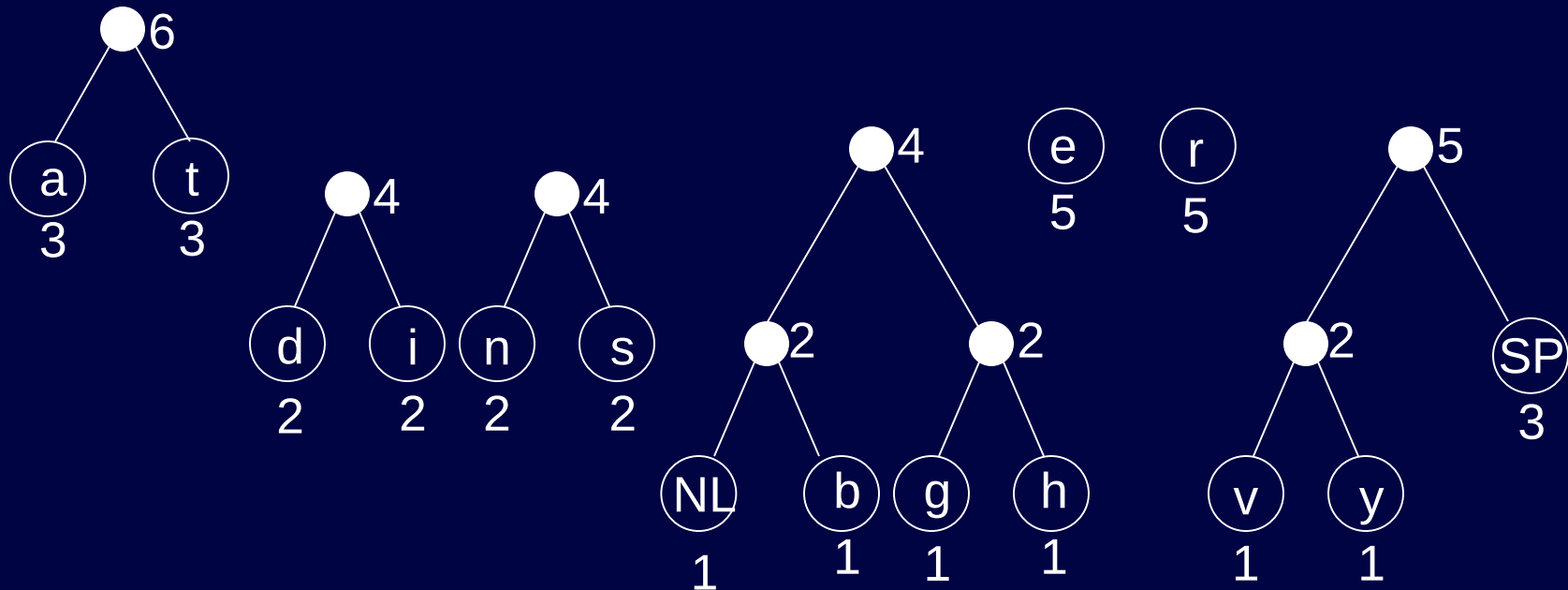


# Huffman Encoding

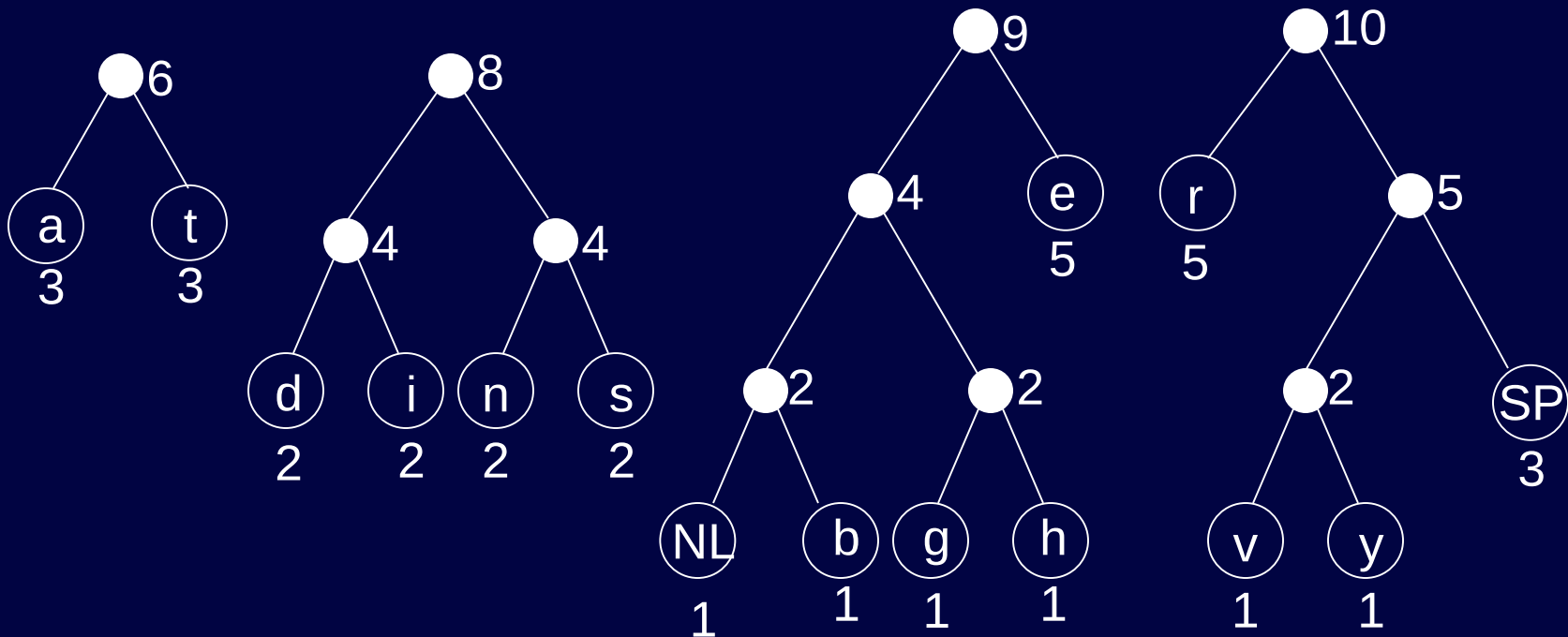




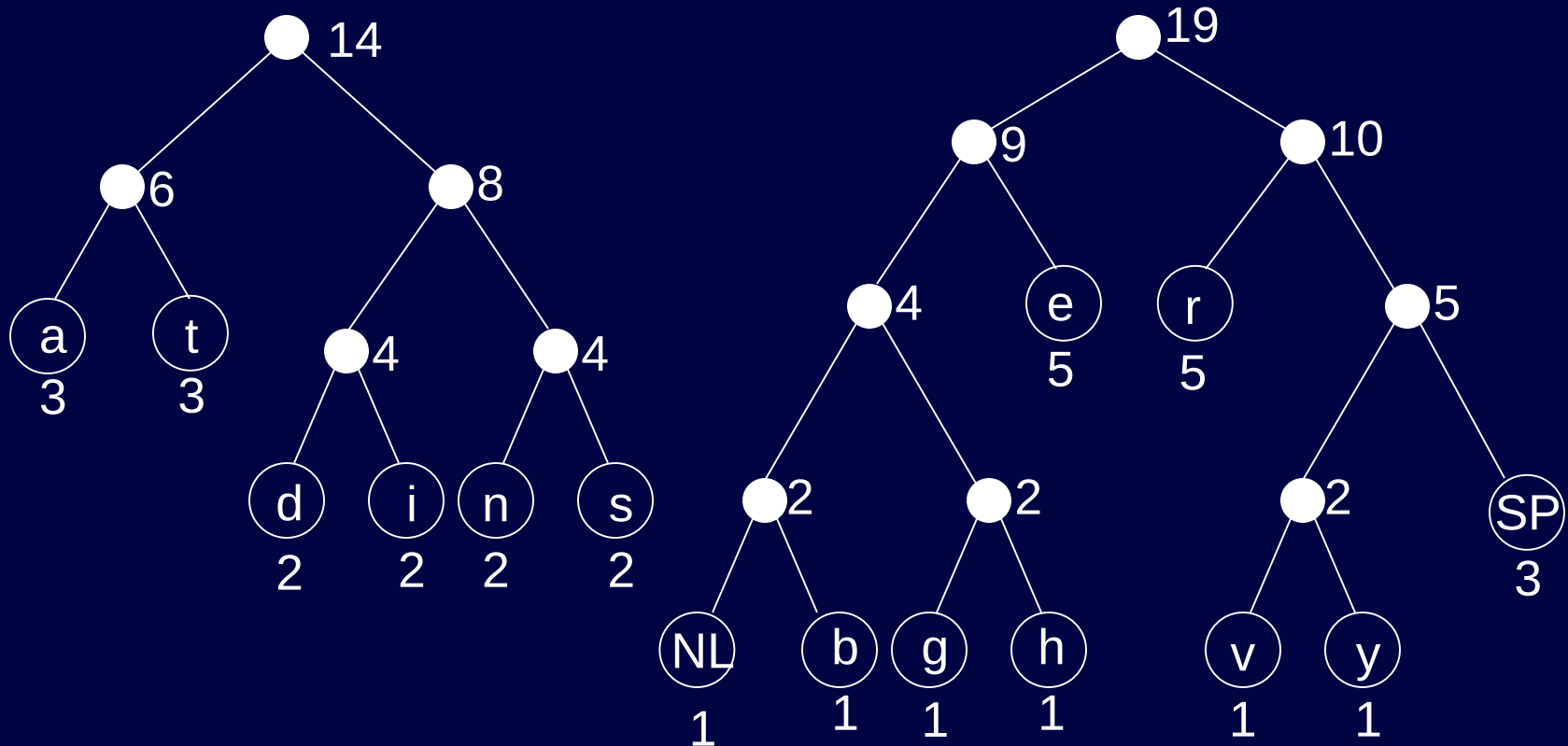
# Huffman Encoding



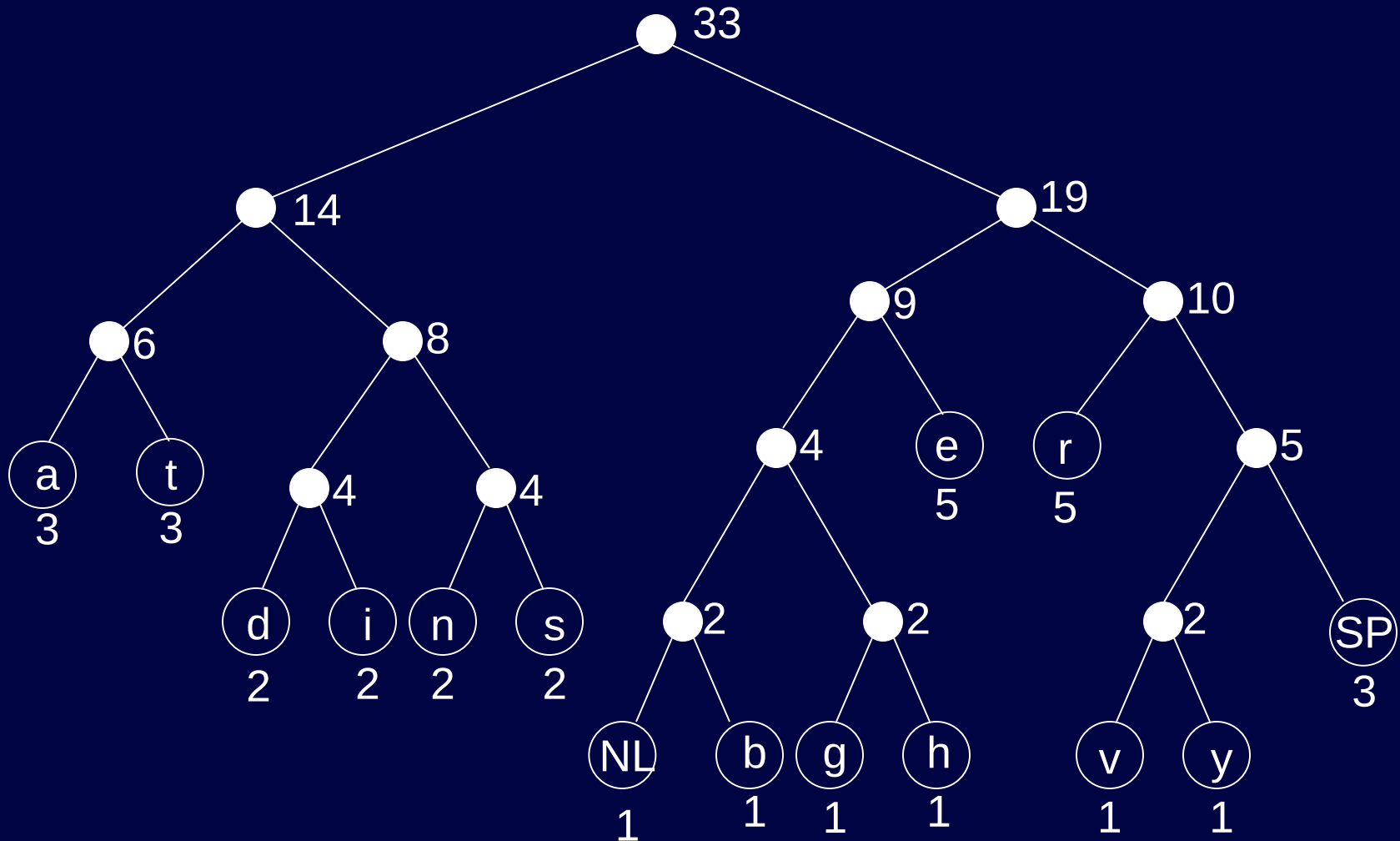
# Huffman Encoding



# Huffman Encoding



# Huffman Encoding



# Huffman Encoding

- Huffman code is method for the compression for 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.

# Lecture No.26

## Data Structures

Dr. Sohail Aslam

# Huffman Encoding

- To understand Huffman encoding, it is best to use a simple example.
- Encoding the 32-character phrase:  
*"traversing threaded binary trees"*,
- If we send the phrase as a message in a network using standard 8-bit ASCII codes, we would have to send  $8 \times 32 = 256$  bits.
- Using the Huffman algorithm, we can send the message with only 116 bits.

# Huffman Encoding

- 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, and if there is a tie, pick randomly amongst those with equal frequencies.



# Huffman Encoding

- Make a new node out of these two, and make the two nodes 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 until only one node, the root, remains.

# Huffman Encoding

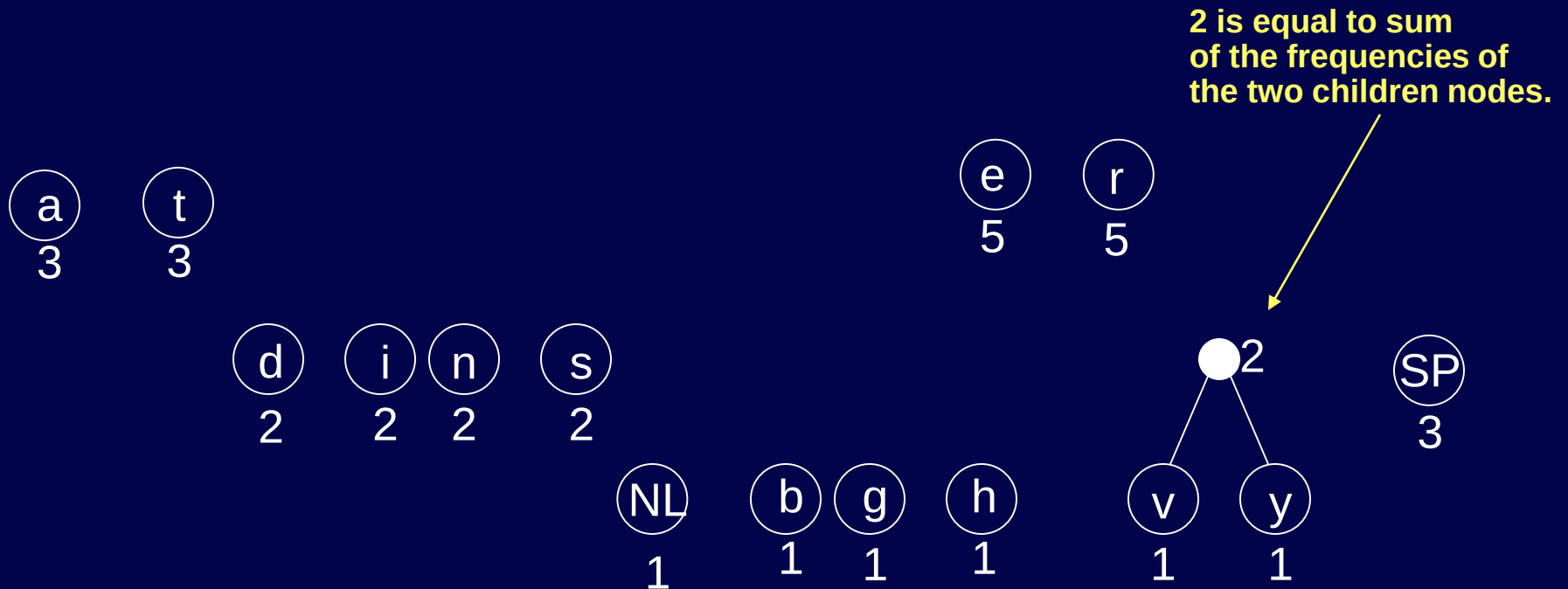
Original text:

*traversing threaded binary trees*

size: 33 characters (space and newline)

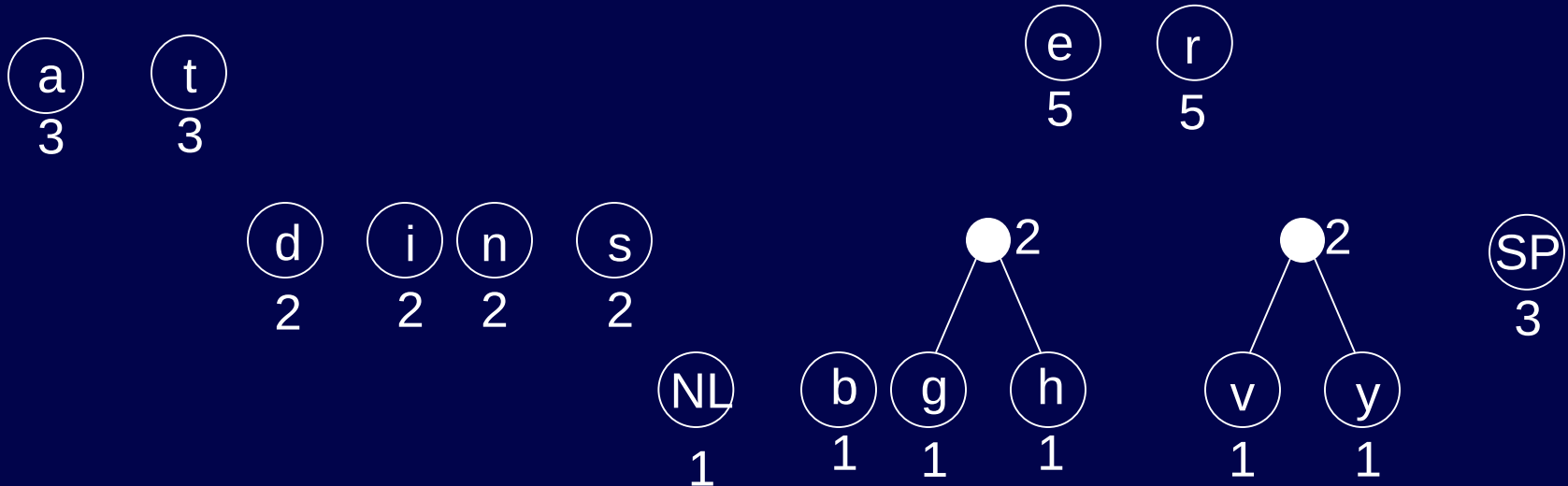
|      |   |     |   |
|------|---|-----|---|
| NL : | 1 | i : | 2 |
| SP : | 3 | n : | 2 |
| a :  | 3 | r : | 5 |
| b :  | 1 | s : | 2 |
| d :  | 2 | t : | 3 |
| e :  | 5 | v : | 1 |
| g :  | 1 | y : | 1 |
| h :  | 1 |     |   |

# Huffman Encoding

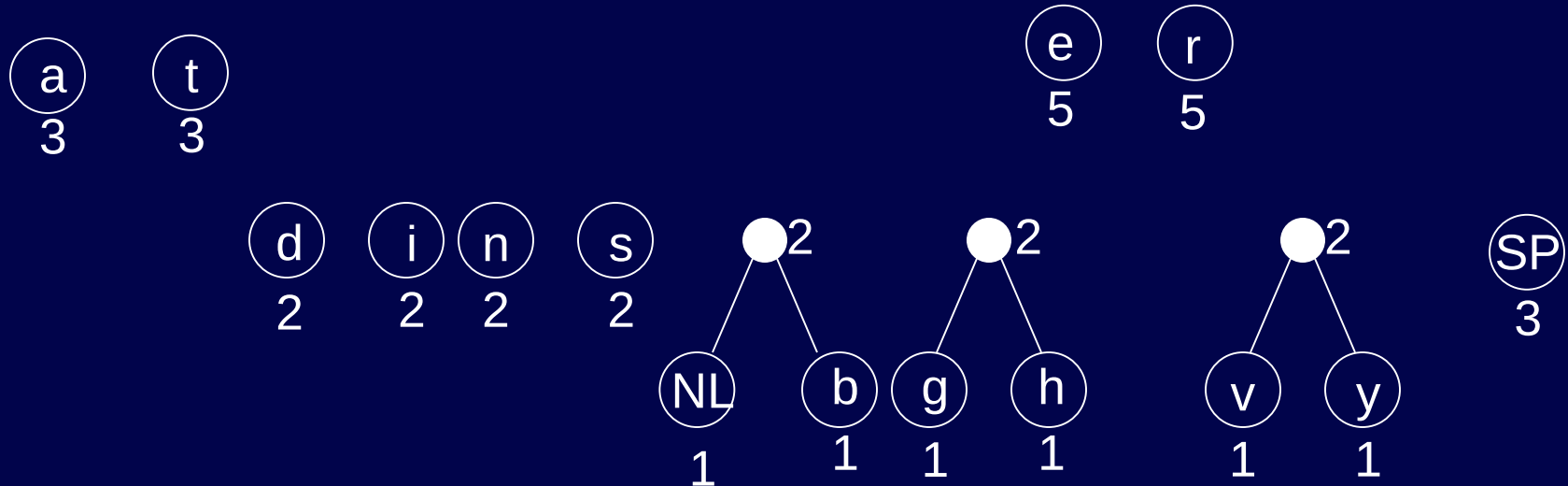


# Huffman Encoding

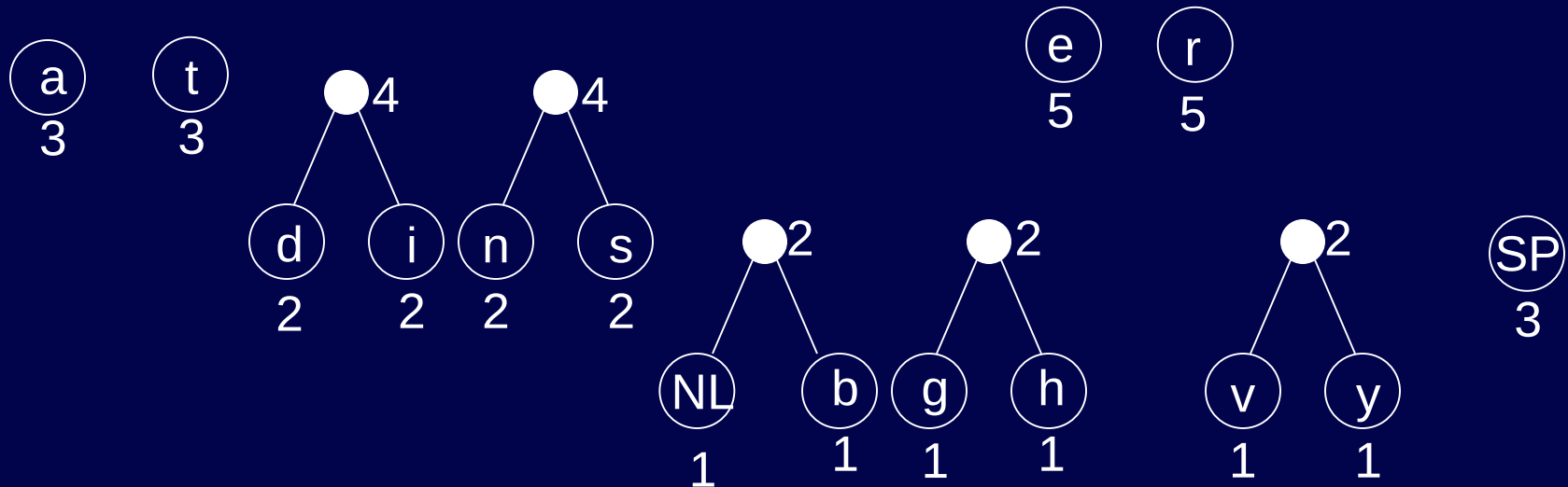
There a number of ways to combine nodes. We have chosen just one such way.



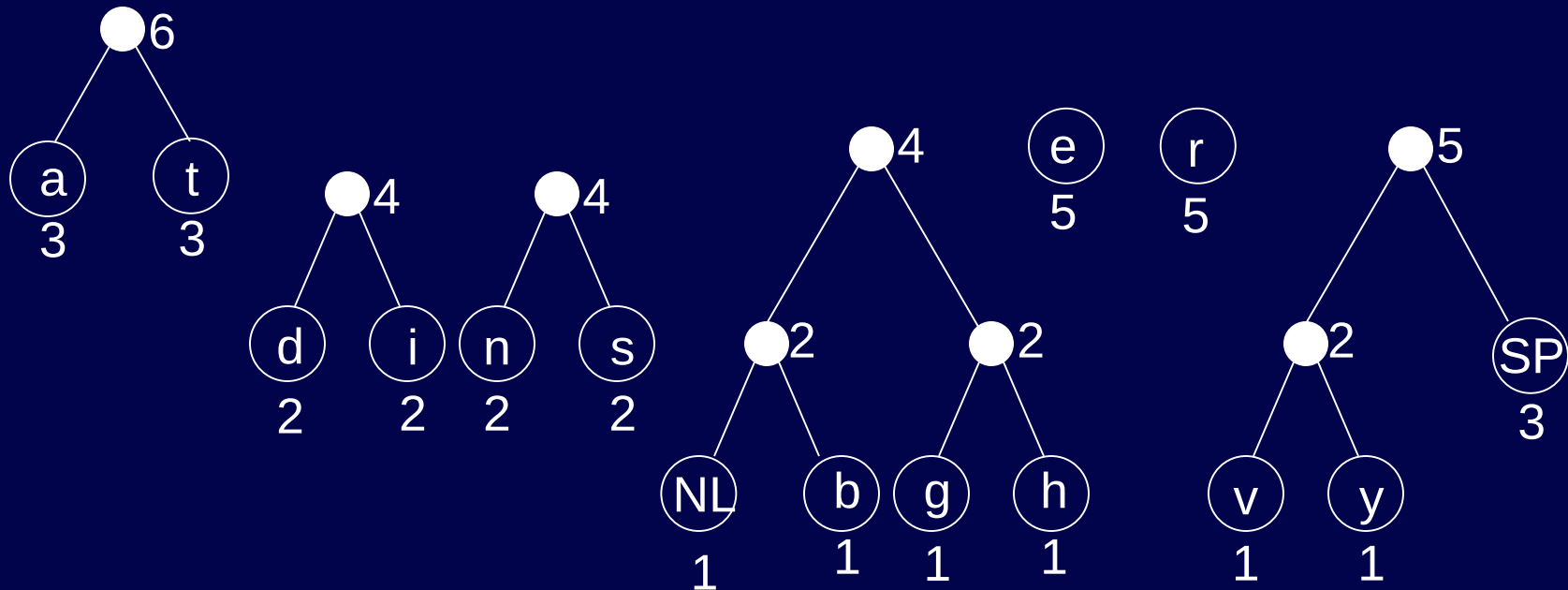
# Huffman Encoding



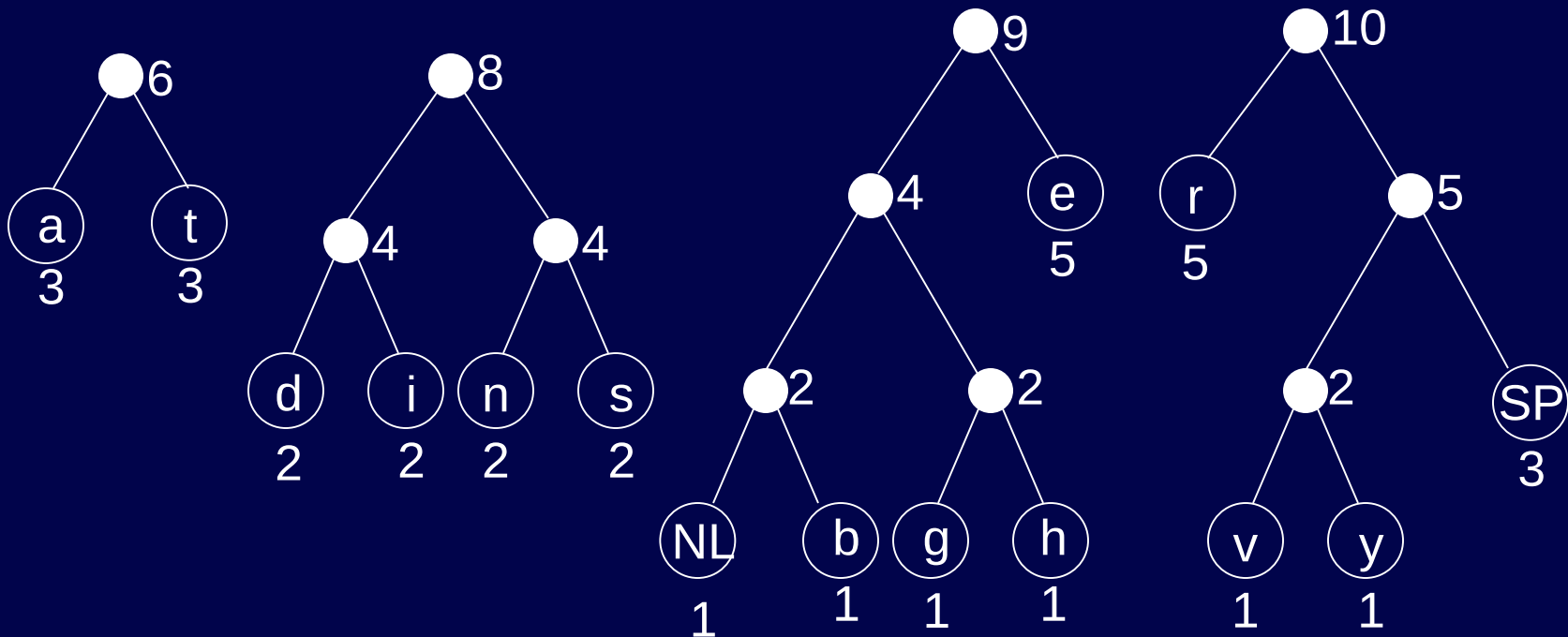
# Huffman Encoding



# Huffman Encoding

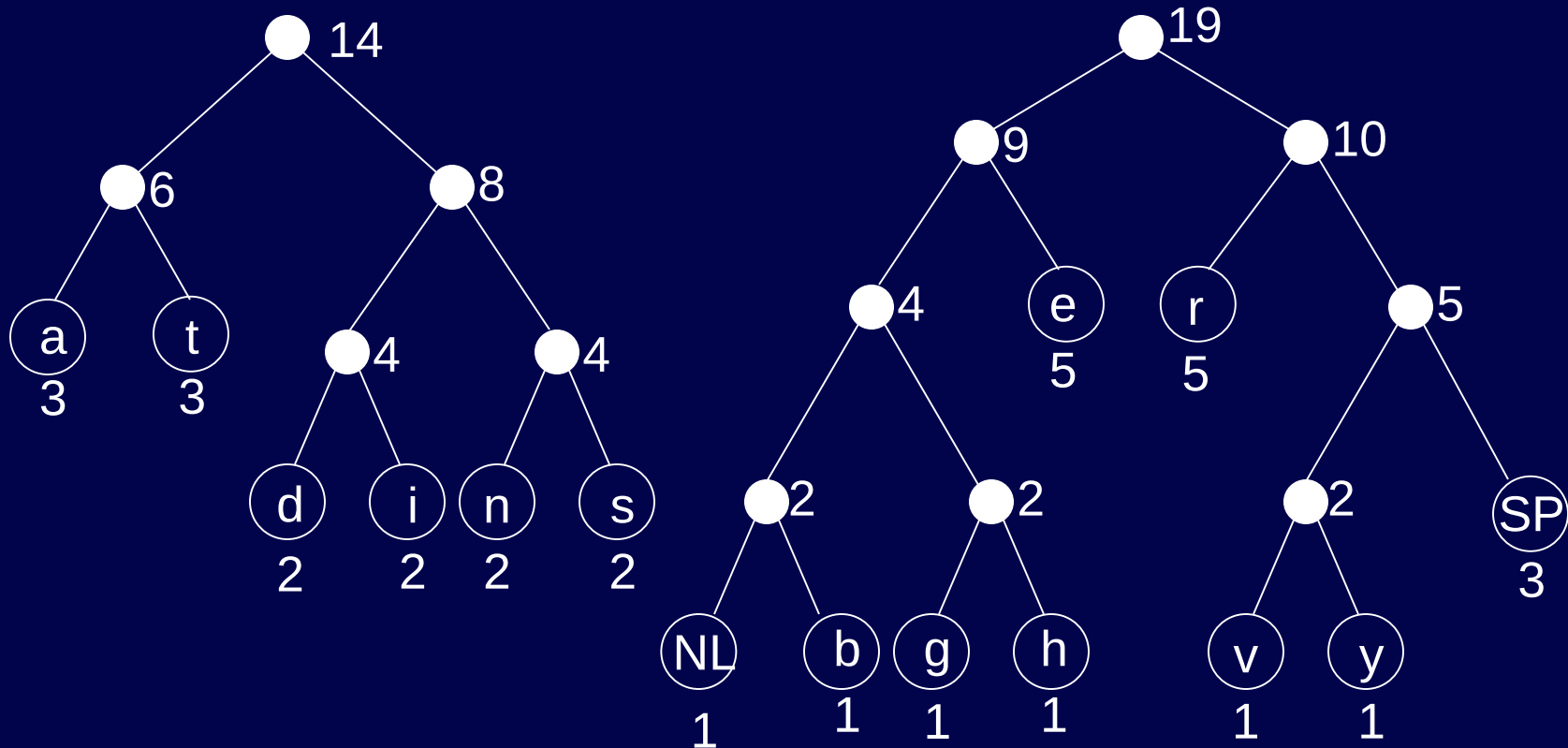


# Huffman Encoding

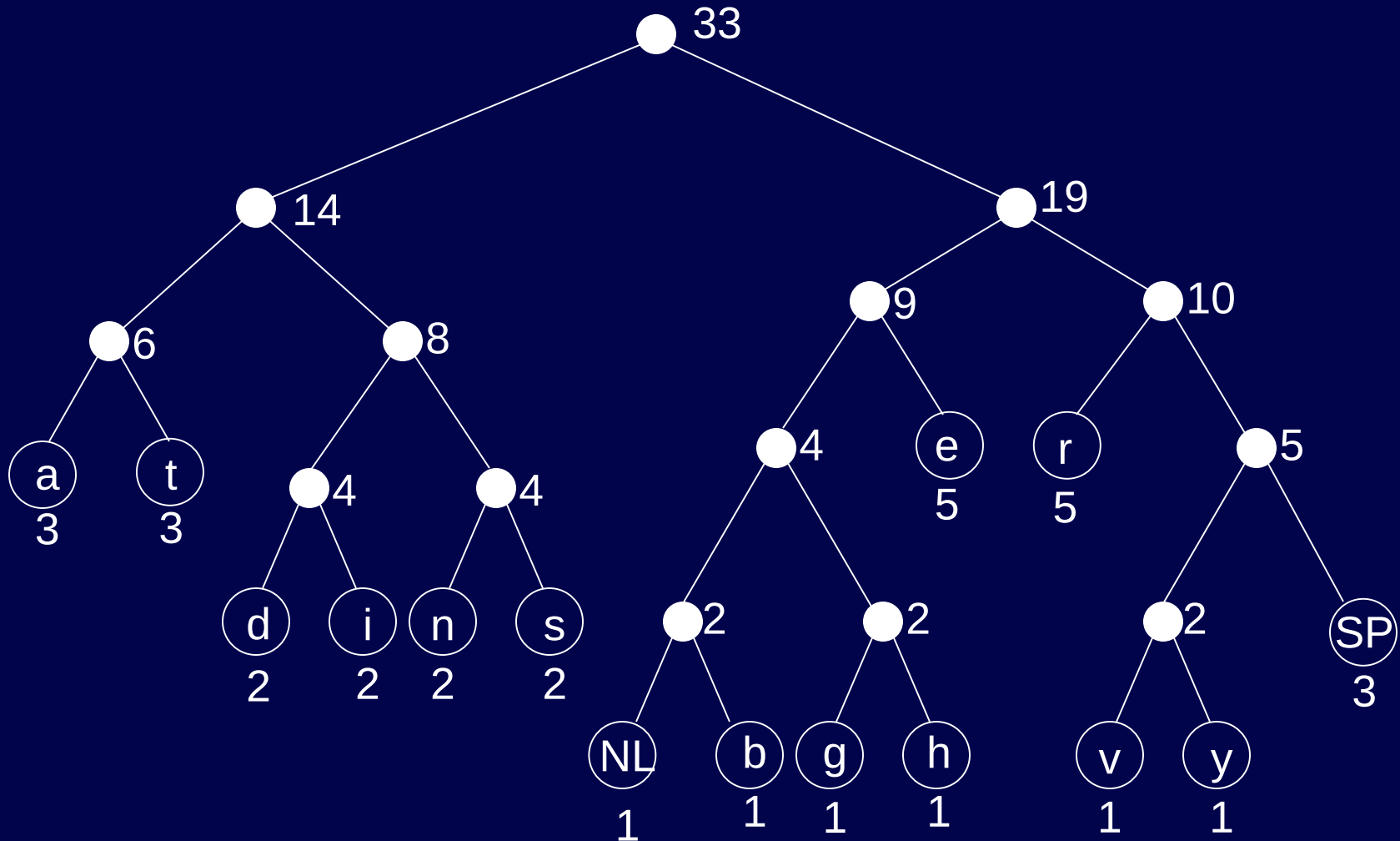




# Huffman Encoding



# Huffman Encoding



# Huffman Encoding

- 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, and if there is a tie, pick randomly amongst those with equal frequencies.

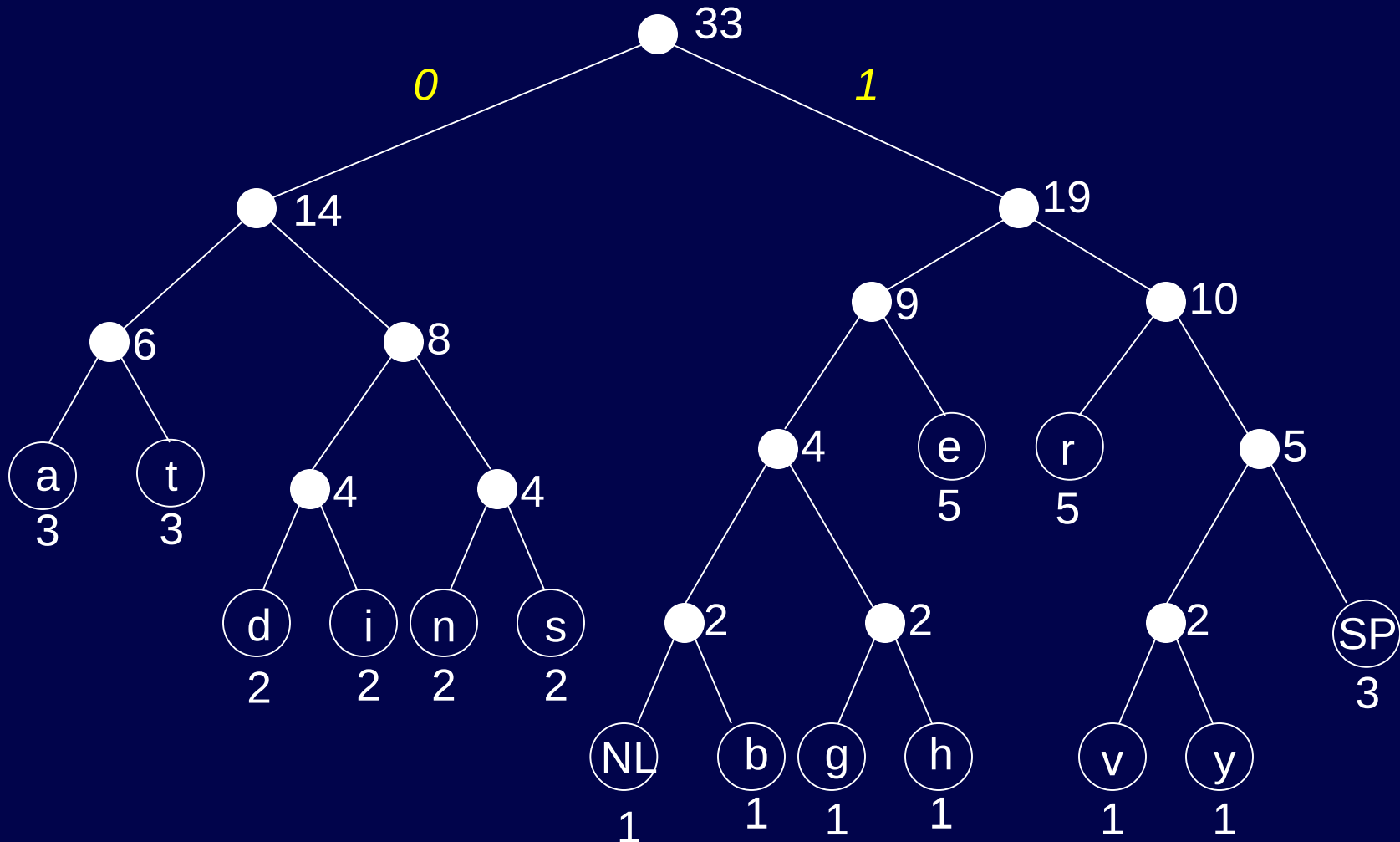
# Huffman Encoding

- Make a new node out of these two, and make the two nodes 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 until only one node, the root, remains.

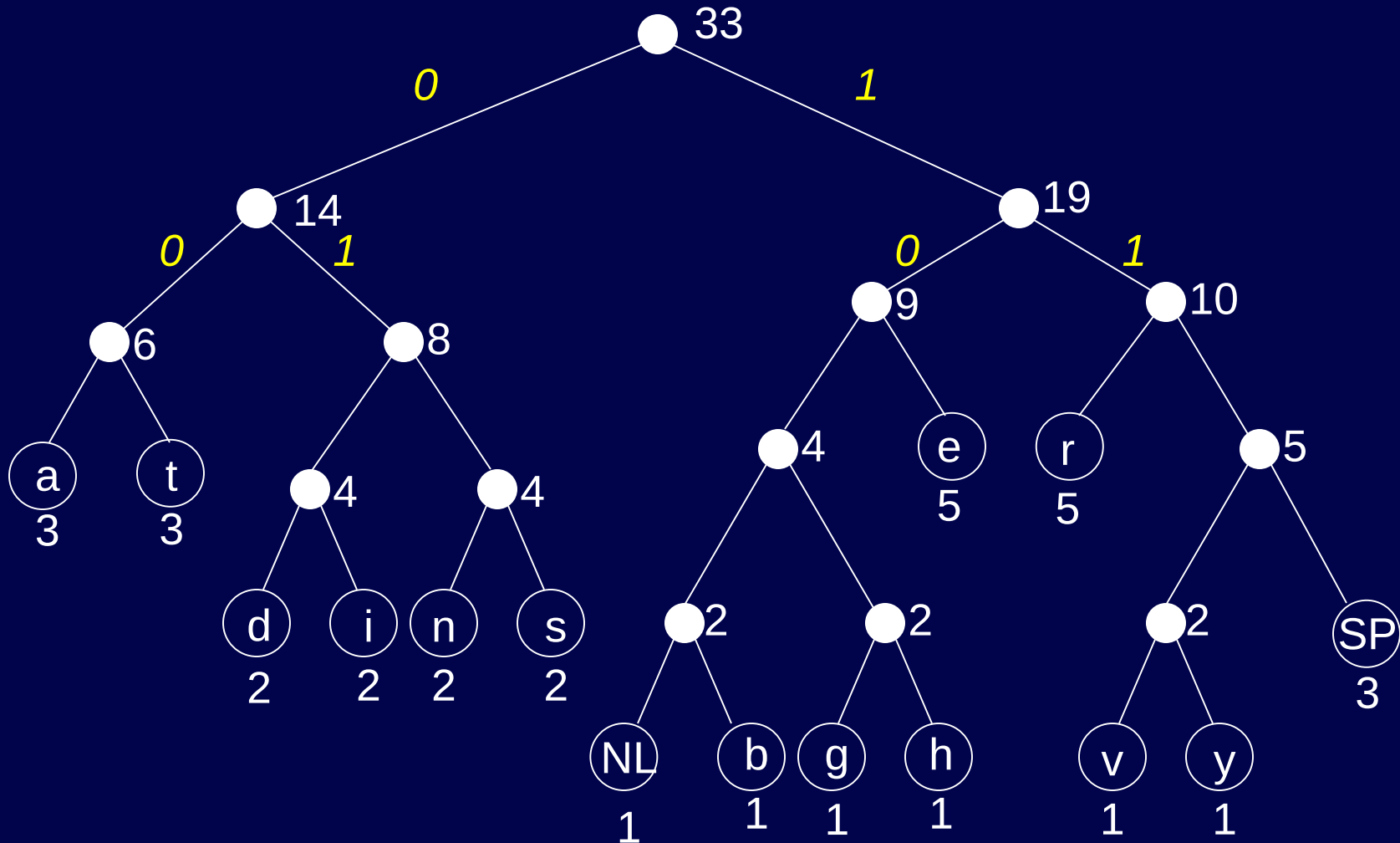
# Huffman Encoding

- 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.

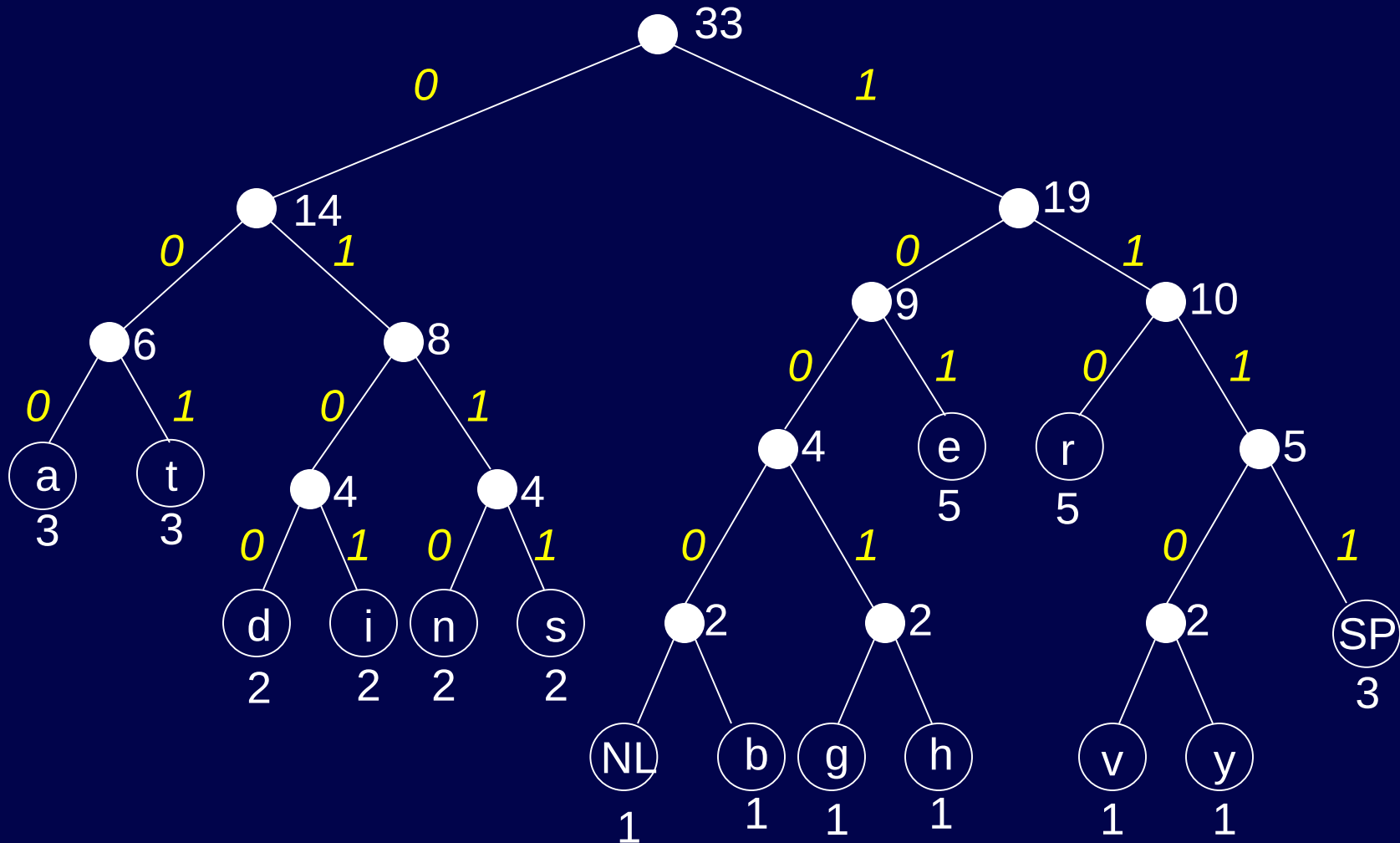
# Huffman Encoding



# Huffman Encoding

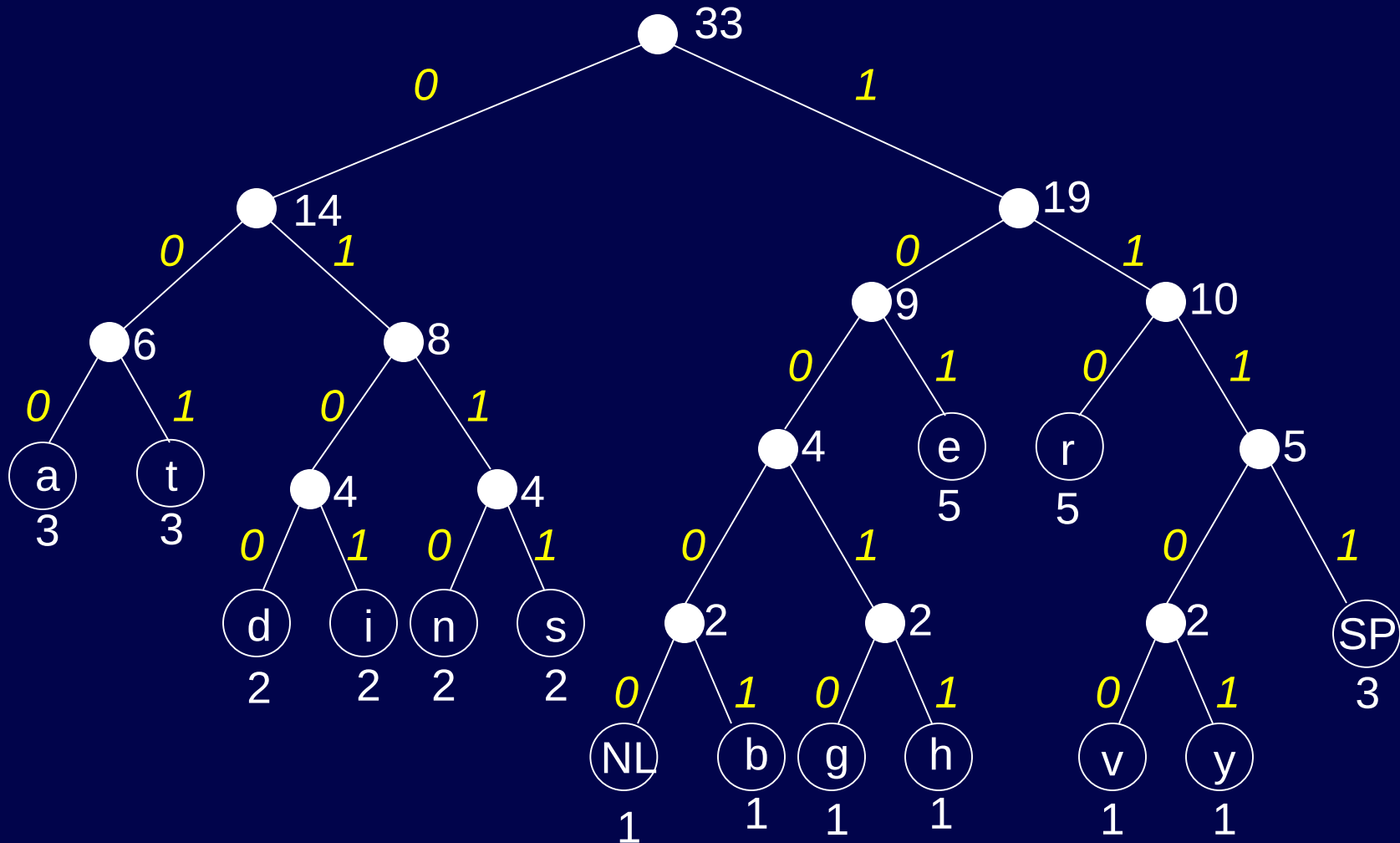


# Huffman Encoding





# Huffman Encoding



# Huffman Encoding

## Huffman character codes

|    |   |       |
|----|---|-------|
| NL | ⇒ | 10000 |
| SP | ⇒ | 1111  |
| a  | ⇒ | 000   |
| b  | ⇒ | 10001 |
| d  | ⇒ | 0100  |
| e  | ⇒ | 101   |
| g  | ⇒ | 10010 |
| h  | ⇒ | 10011 |
| i  | ⇒ | 0101  |
| n  | ⇒ | 0110  |
| r  | ⇒ | 110   |
| s  | ⇒ | 0111  |
| t  | ⇒ | 001   |
| v  | ⇒ | 11100 |
| y  | ⇒ | 11101 |

- Notice that the code is variable length.
- Letters with higher frequencies have shorter codes.
- The tree could have been built in a number of ways; each would yielded different codes but the code would still be minimal.

# Huffman Encoding

# Original: *traversing threaded binary trees*

# Encoded:

t r a v e

001110000111001011100111010101101001  
011110011001111010100001001010100111  
110000101011000011011101111100111010  
110101110000

# Huffman Encoding

Original: *traversing threaded binary trees*  
With 8 bits per character, length is 264.

Encoded:

```
001110000111001011100111010101101001  
011110011001111010100001001010100111  
110000101011000011011101111100111010  
110101110000
```

Compressed into 122 bits, 54% reduction.

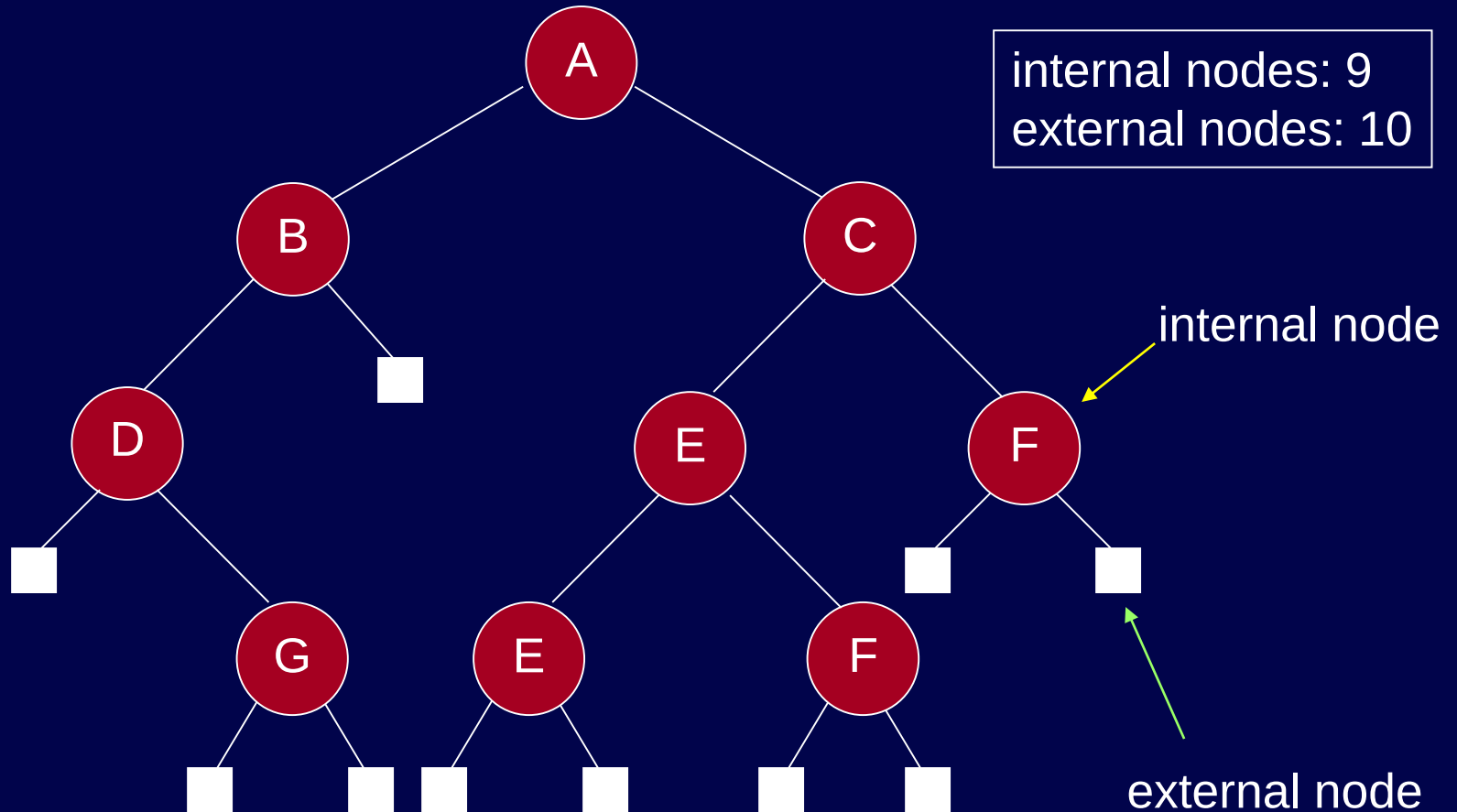
# Mathematical Properties of Binary Trees

# Properties of Binary Tree

Property: *A binary tree with  $N$  internal nodes has  $N+1$  external nodes.*

# Properties of Binary Tree

*A binary tree with  $N$  internal nodes has  $N+1$  external nodes.*



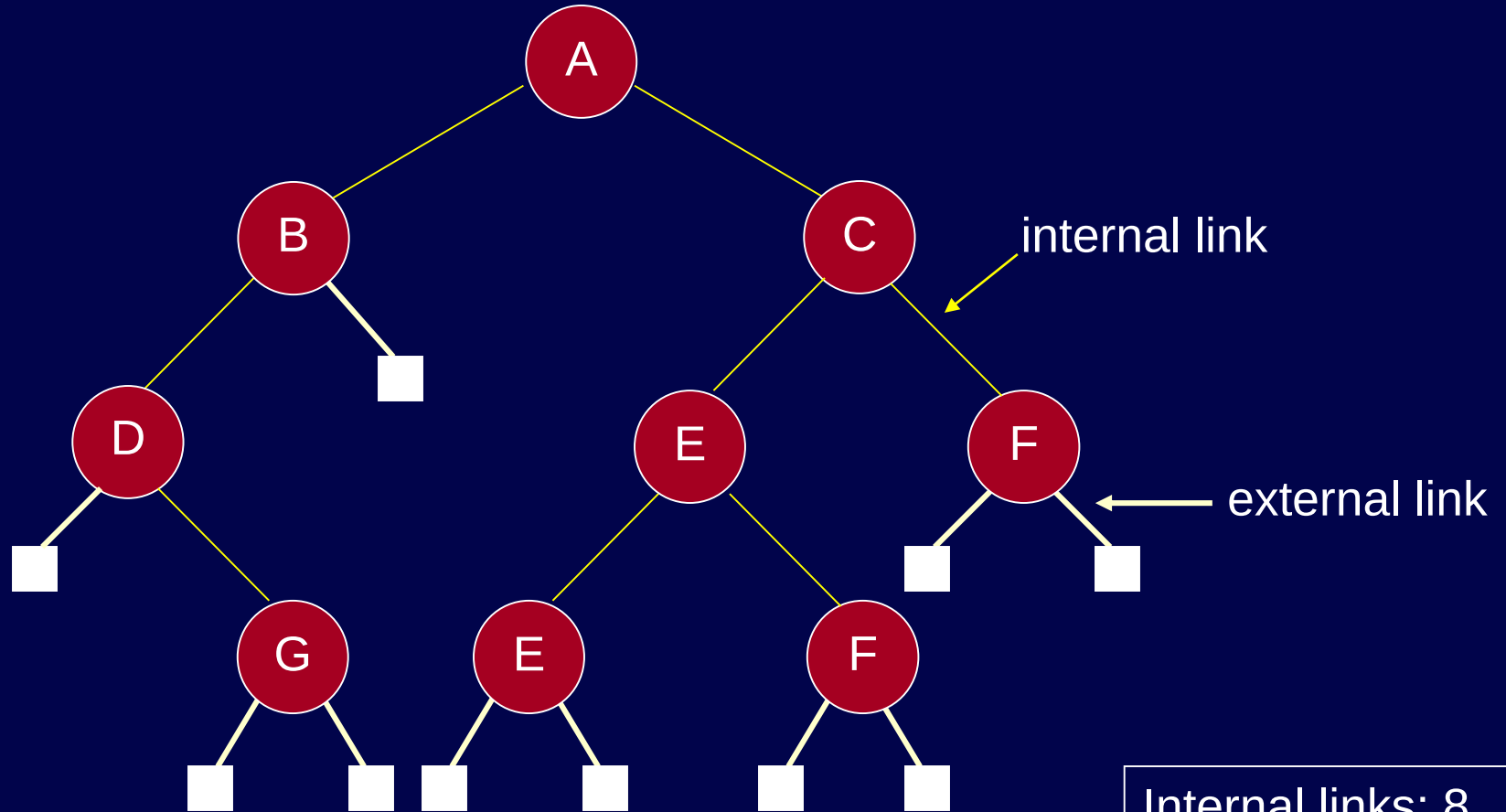
# Properties of Binary Tree

*Property: A binary tree with  $N$  internal nodes has  $2N$  links:  $N-1$  links to internal nodes and  $N+1$  links to external nodes.*



# Threaded Binary Tree

Property: A binary tree with  $N$  internal nodes has  $2N$  links:  $N-1$  links to internal nodes and  $N+1$  links to external nodes.



Internal links: 8  
External links: 10

# Properties of Binary Tree

*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 parent, so there are  $N-1$  links connecting internal nodes.
- Similarly, each of the  $N+1$  external nodes has one link to its parent.
- Thus  $N-1+N+1=2N$  links.

# Data Structures

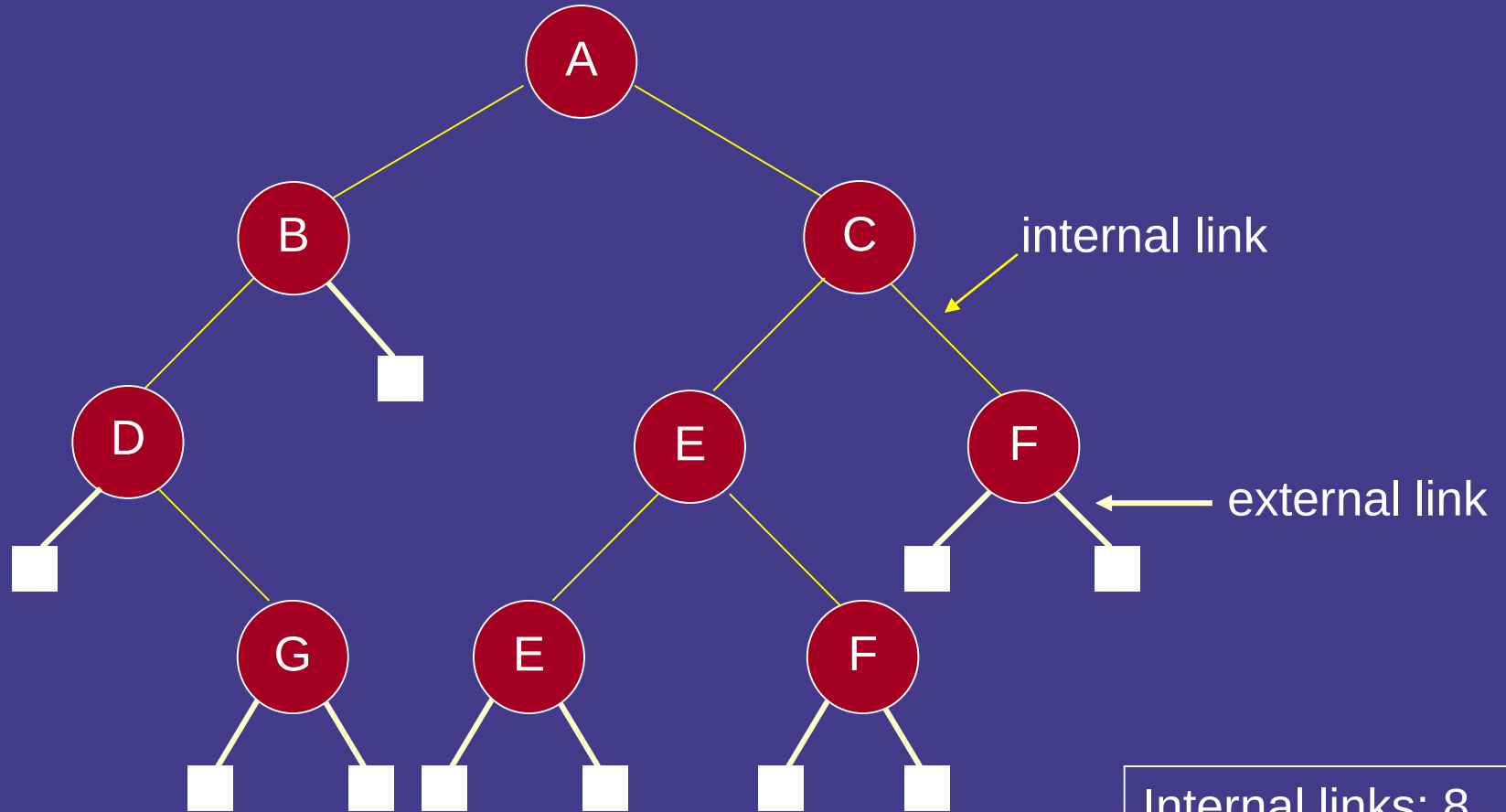
Lecture 27  
Sohail Aslam

# Properties of Binary Tree

*Property: A binary tree with  $N$  internal nodes has  $2N$  links:  $N-1$  links to internal nodes and  $N+1$  links to external nodes.*

# Threaded Binary Tree

Property: A binary tree with  $N$  internal nodes has  $2N$  links:  $N-1$  links to internal nodes and  $N+1$  links to external nodes.



Internal links: 8  
External links: 10

# Properties of Binary Tree

*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 parent, so there are  $N-1$  links connecting internal nodes.
- Similarly, each of the  $N+1$  external nodes has one link to its parent.
- Thus  $N-1+N+1=2N$  links.

# Threaded Binary Trees

# Threaded Binary Tree

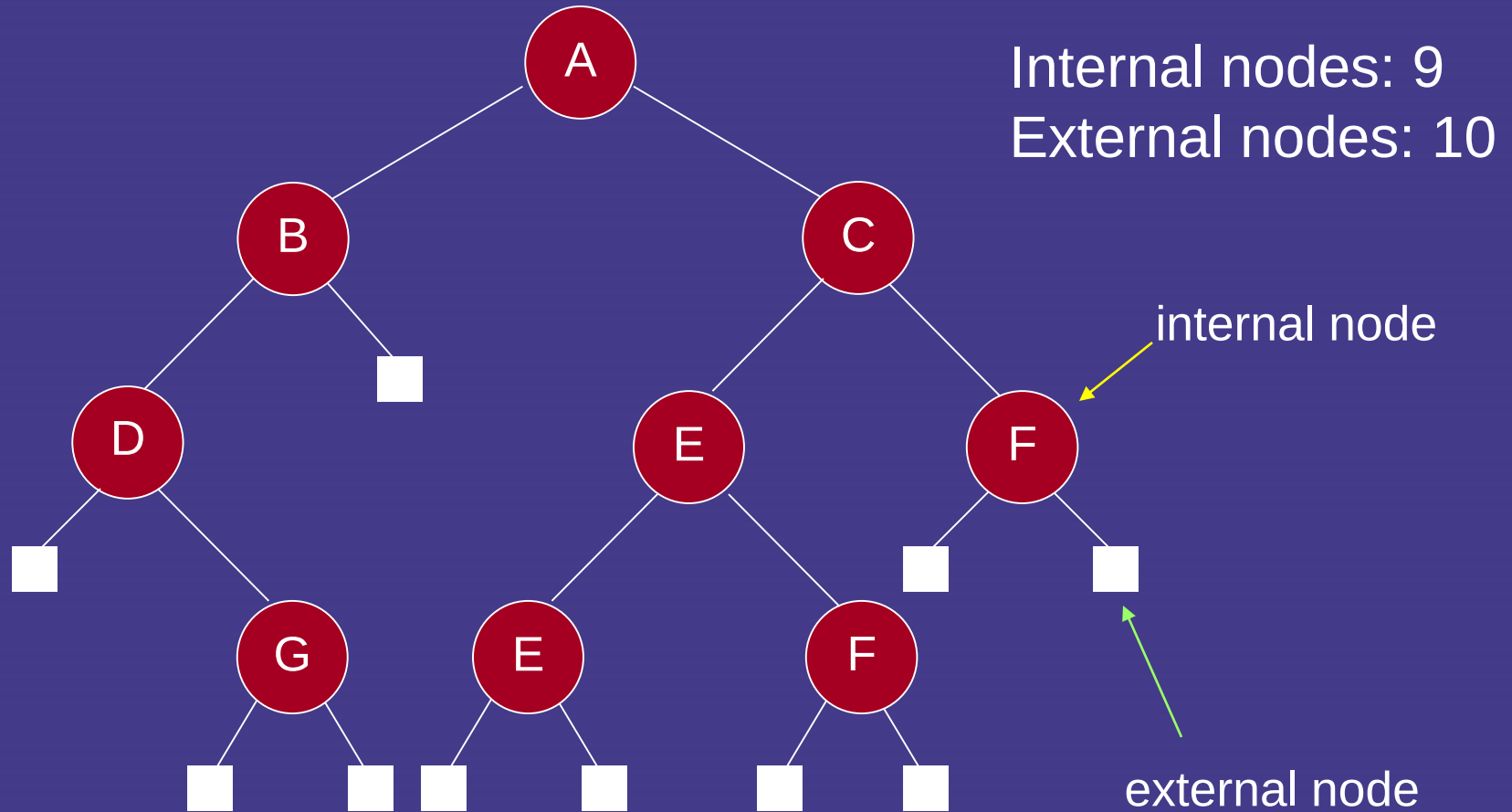
- In many applications binary tree traversals are carried out repeatedly.
- The overhead of stack operations during recursive calls can be costly.
- The same would true if we use a non-recursive but stack-driven traversal procedure
- It would be useful to modify the tree data structure which represents the binary tree so as to speed up, say, the inorder traversal process: make it "stack-free".



# Threaded Binary Tree

- 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.

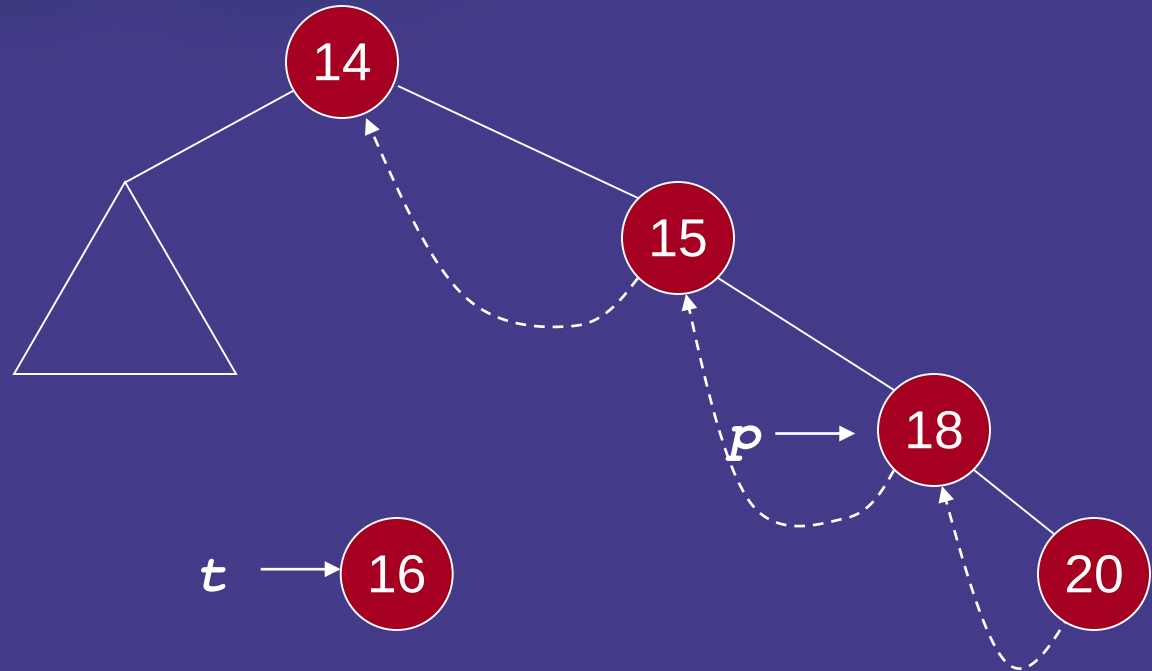
# Threaded Binary Tree



# Threaded Binary Tree

- The *threaded tree* data structure will replace these NULL pointers with pointers to the inorder successor (predecessor) of a node as appropriate.
- 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.

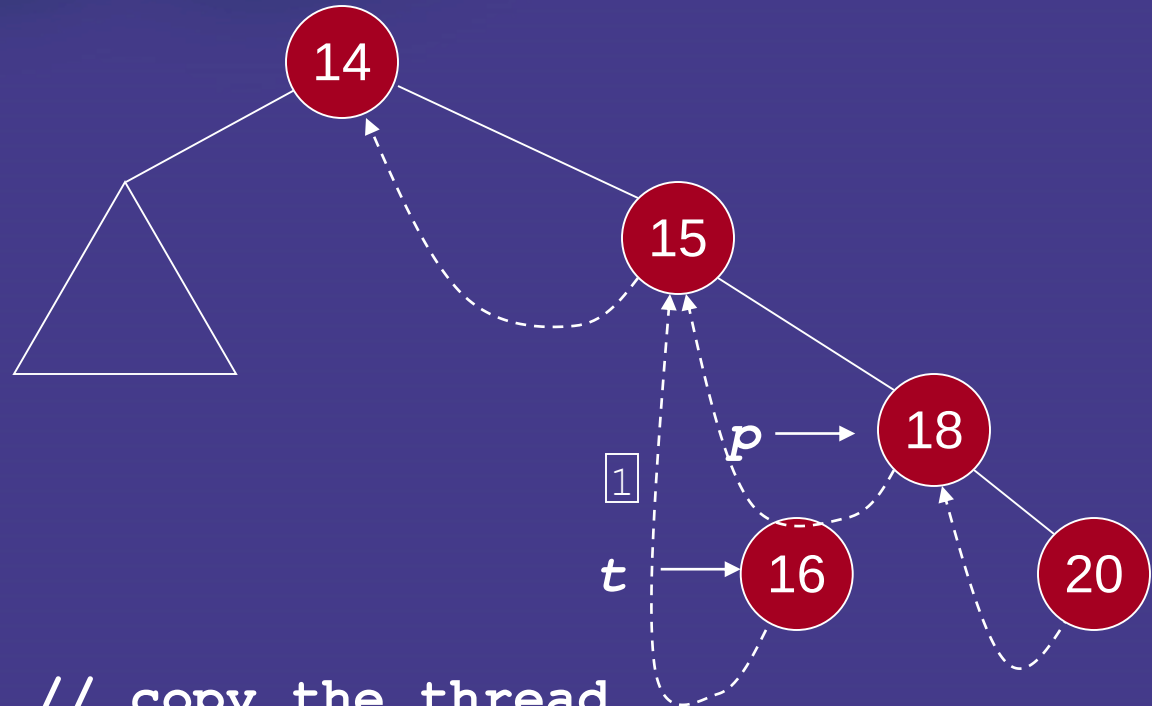
# Adding threads during insert



```
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;
```

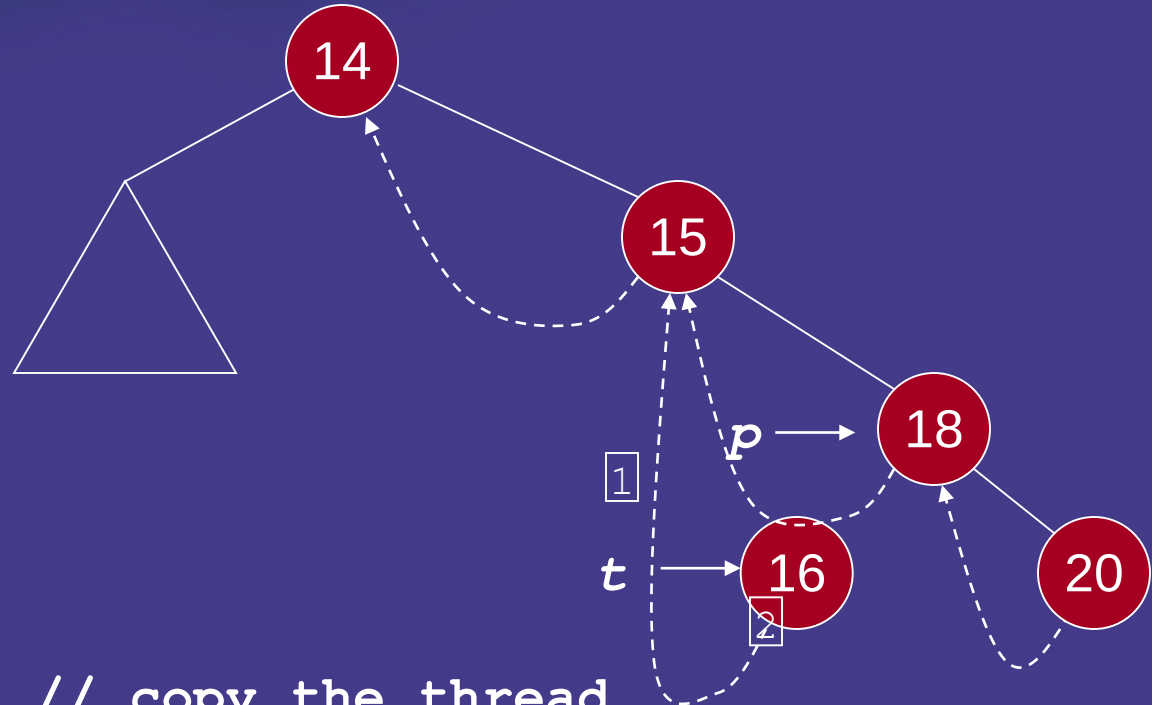
# Adding threads during insert



```
1. 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;
```

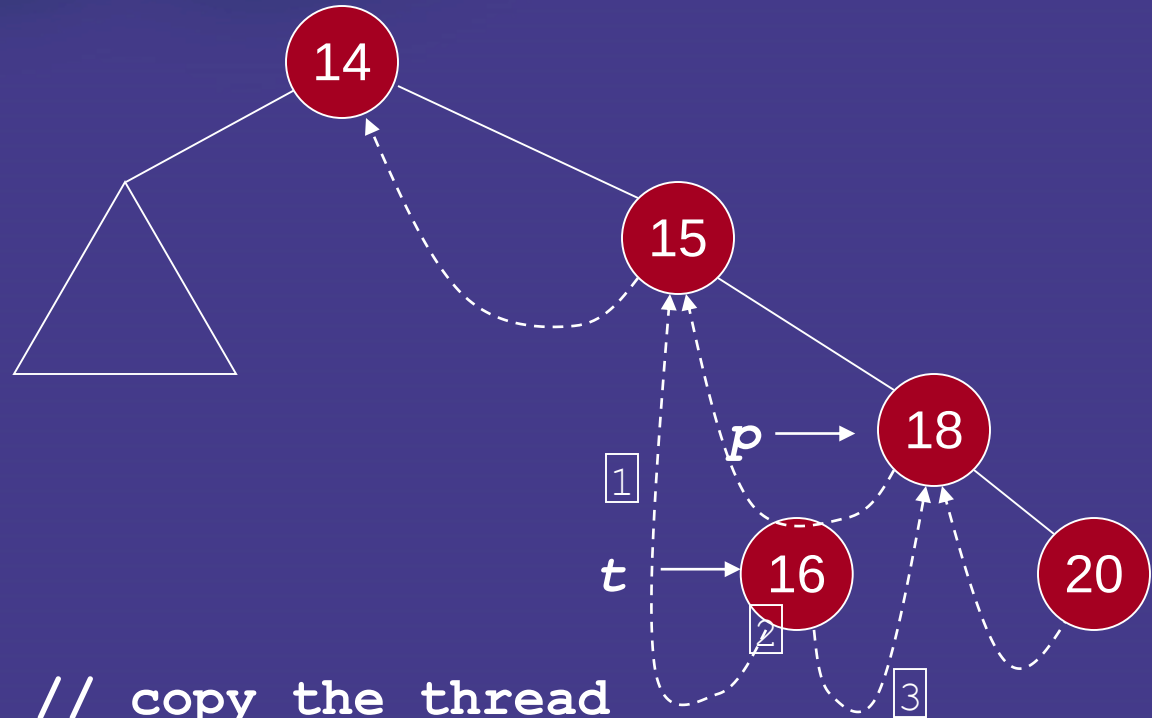
# Adding threads during insert



```
1. t->L = p->L; // copy the thread
2. t->LTH = thread;
t->R = p; // *p is successor of *t
t->RTH = thread;
```

```
p->L = t; // attach the new leaf
p->LTH = child;
```

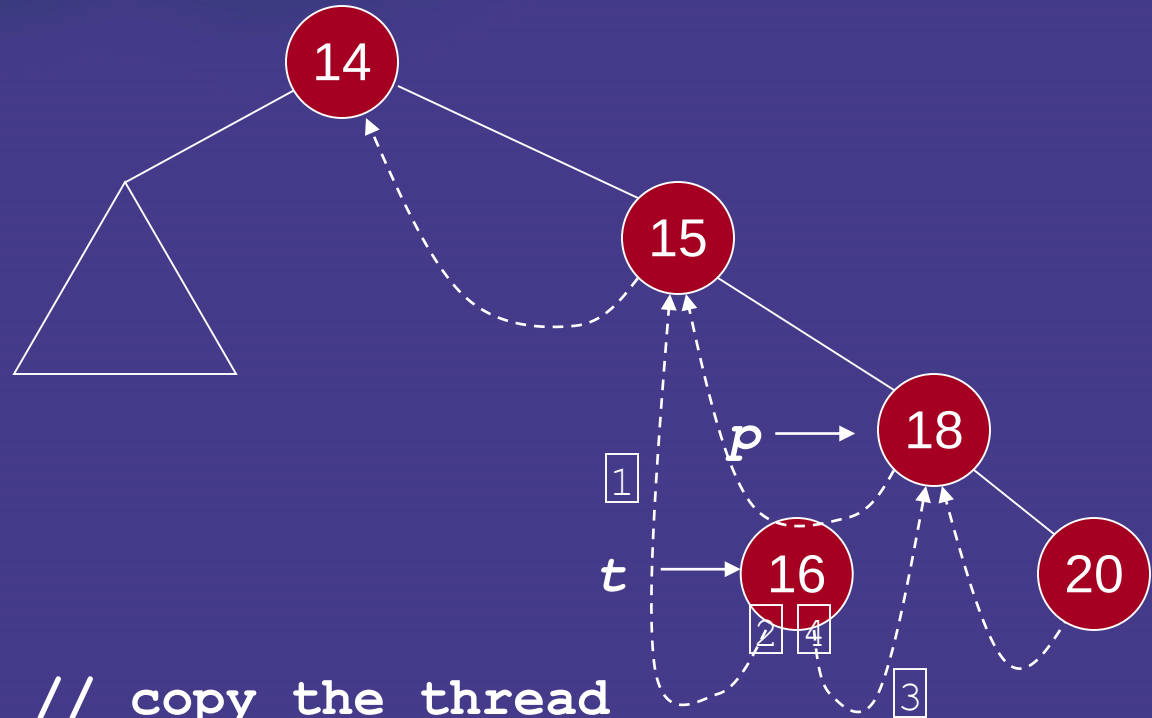
# Adding threads during insert



```
1. t->L = p->L; // copy the thread
2. t->LTH = thread;
3. t->R = p; // *p is successor of *t
t->RTH = thread;
```

```
p->L = t; // attach the new leaf
p->LTH = child;
```

# Adding threads during insert

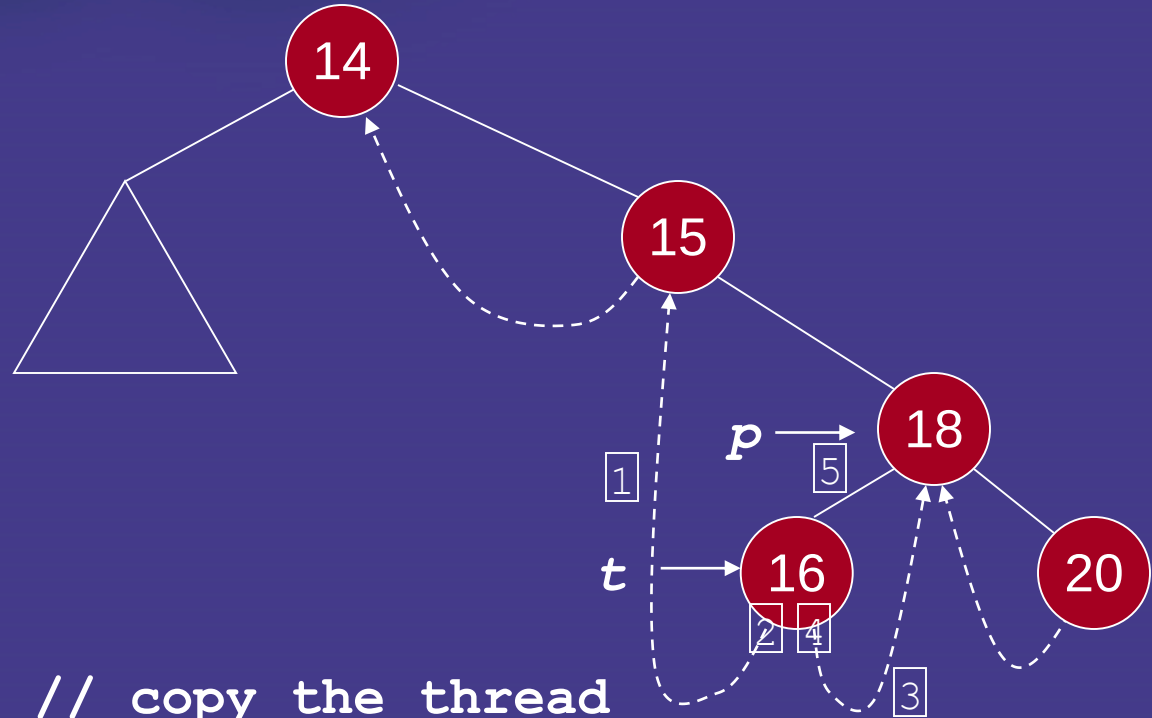


1. `t->L = p->L; // copy the thread`
2. `t->LTH = thread;`
3. `t->R = p; // *p is successor of *t`
4. `t->RTH = thread;`

```
p->L = t; // attach the new leaf
p->LTH = child;
```



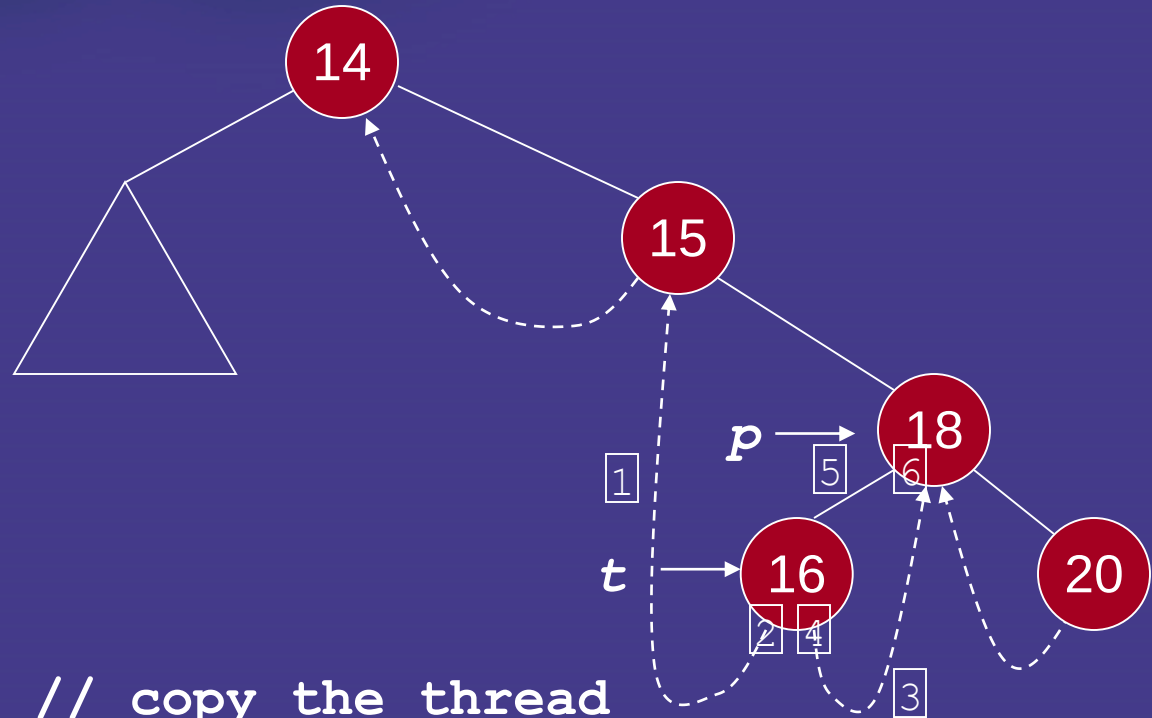
# Adding threads during insert



```
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;
```

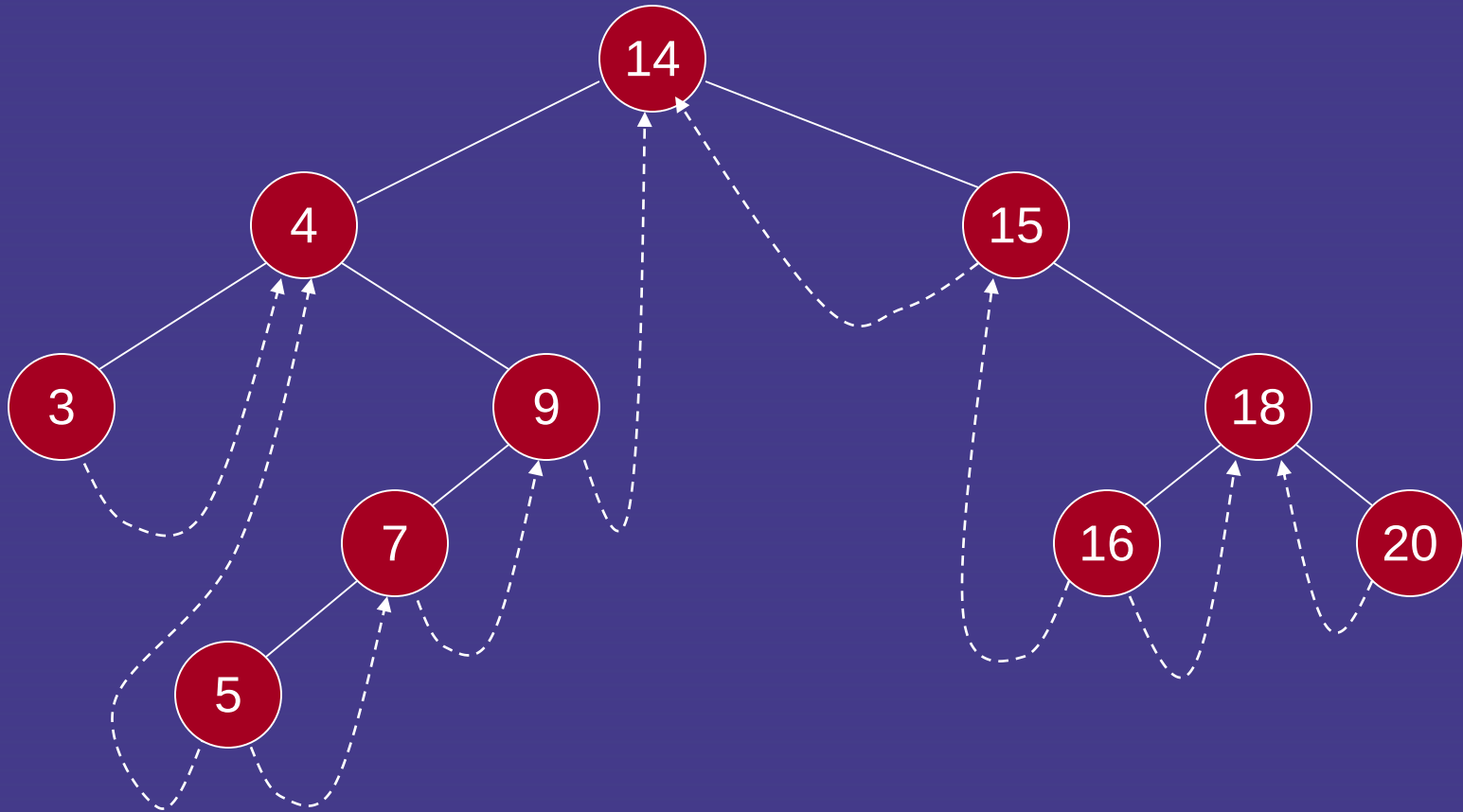
```
5. p->L = t; // attach the new leaf
p->LTH = child;
```

# Adding threads during insert



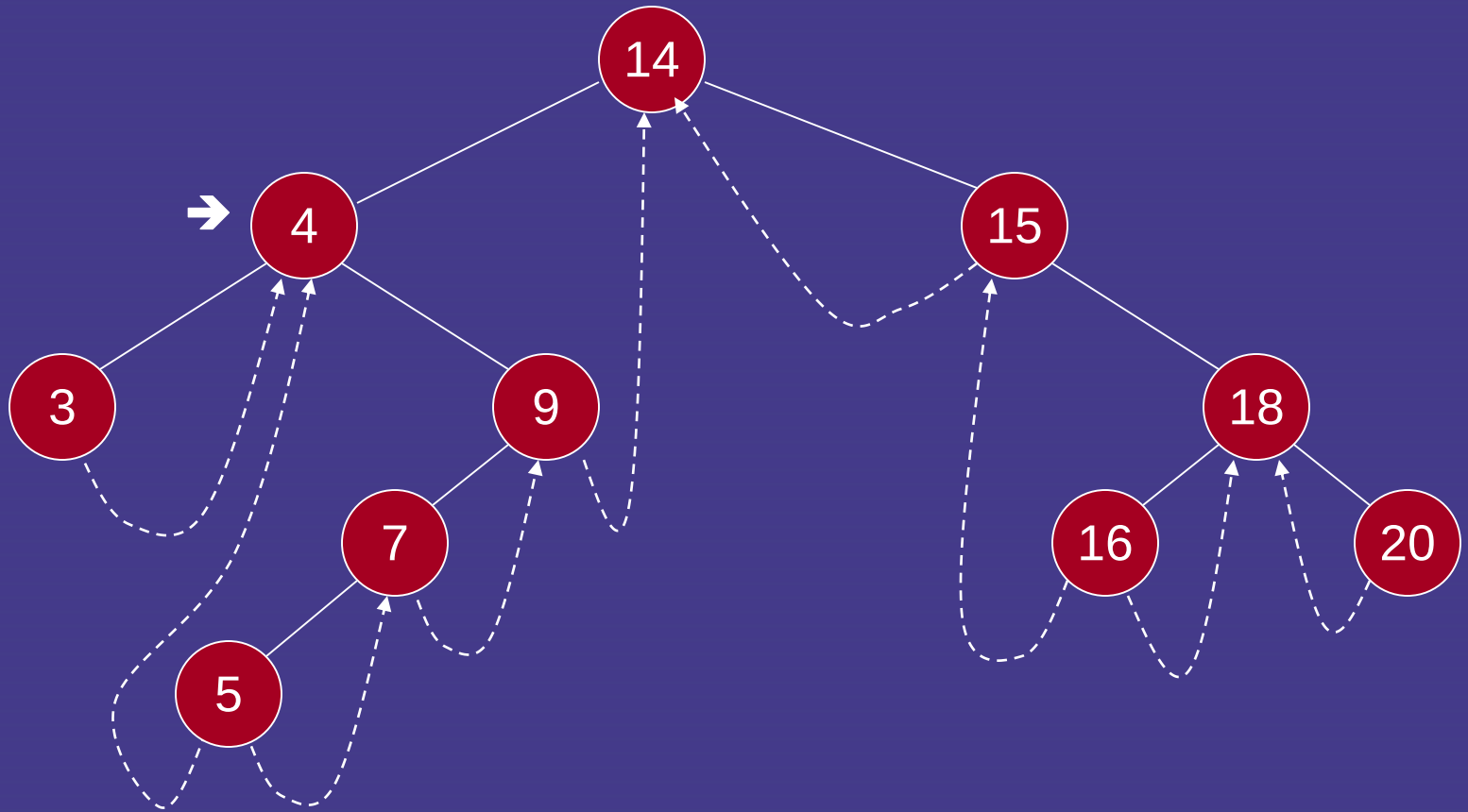
1.  $t \rightarrow L = p \rightarrow L$ ; // copy the thread
2.  $t \rightarrow LTH = \text{thread}$ ;
3.  $t \rightarrow R = p$ ; //  $*p$  is successor of  $*t$
4.  $t \rightarrow RTH = \text{thread}$ ;
5.  $p \rightarrow L = t$ ; // attach the new leaf
6.  $p \rightarrow LTH = \text{child}$ ;

# Threaded Binary Tree



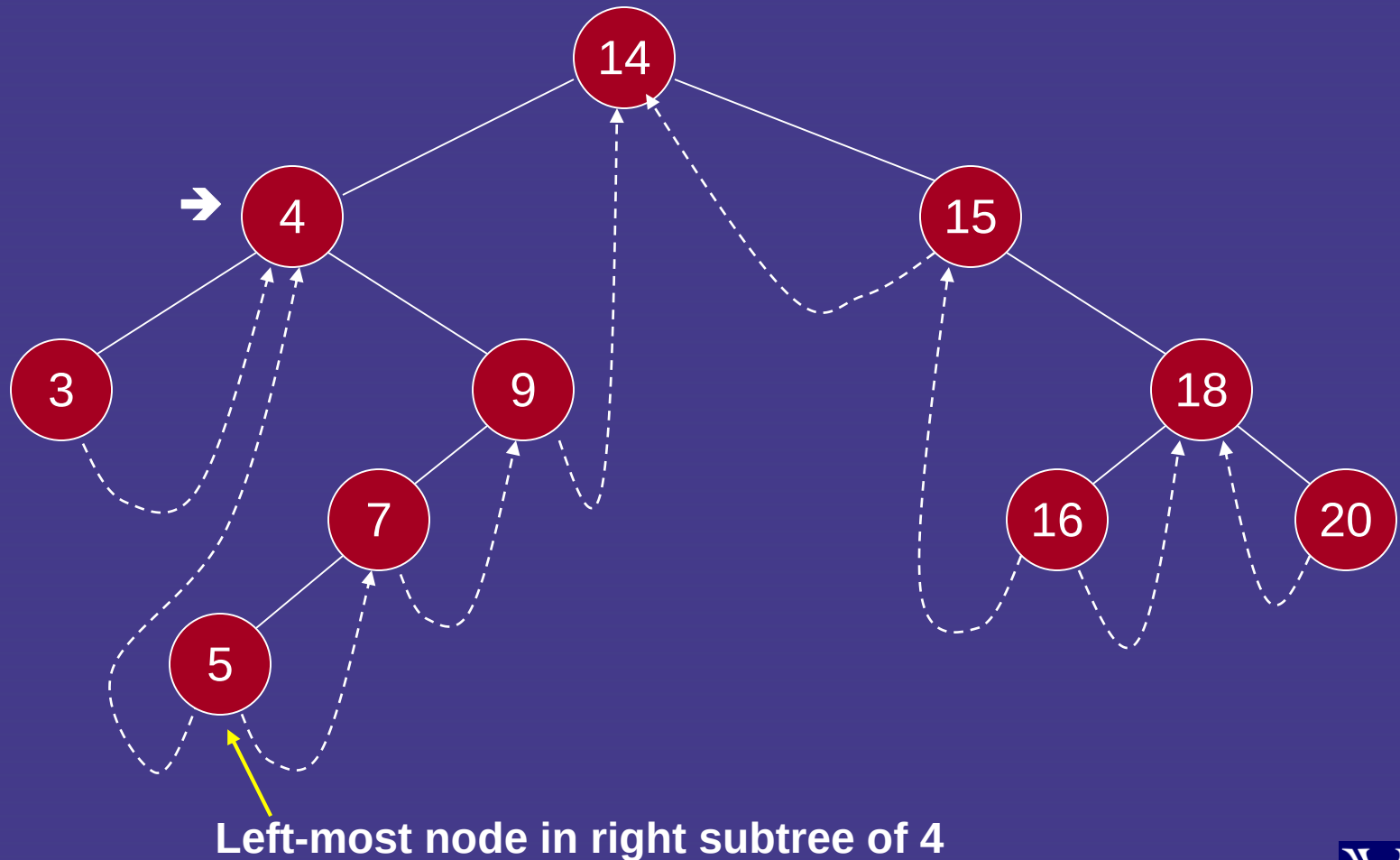
# Where is inorder successor?

Inorder successor of 4.



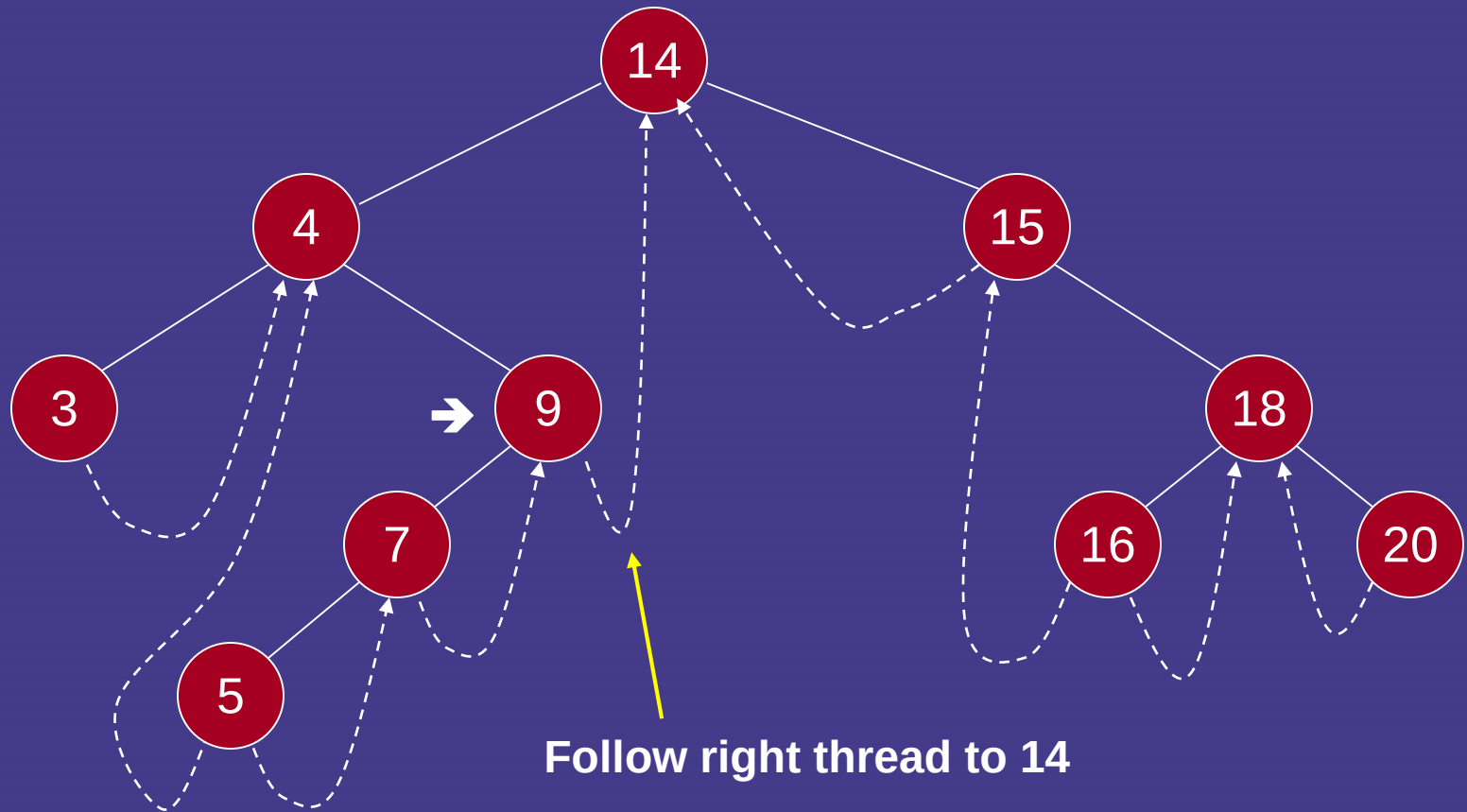
# Where is inorder successor?

Inorder successor of 4.



# Where is inorder successor?

Inorder successor of 9.



# Routine: nextInorder

```
TreeNode* nextInorder(TreeNode* p)
{
    if(p->RTH == thread)
        return(p->R) ;
    else {
        p = p->R;
        while(p->LTH == child)
            p = p->L;
        return p;
    }
}
```

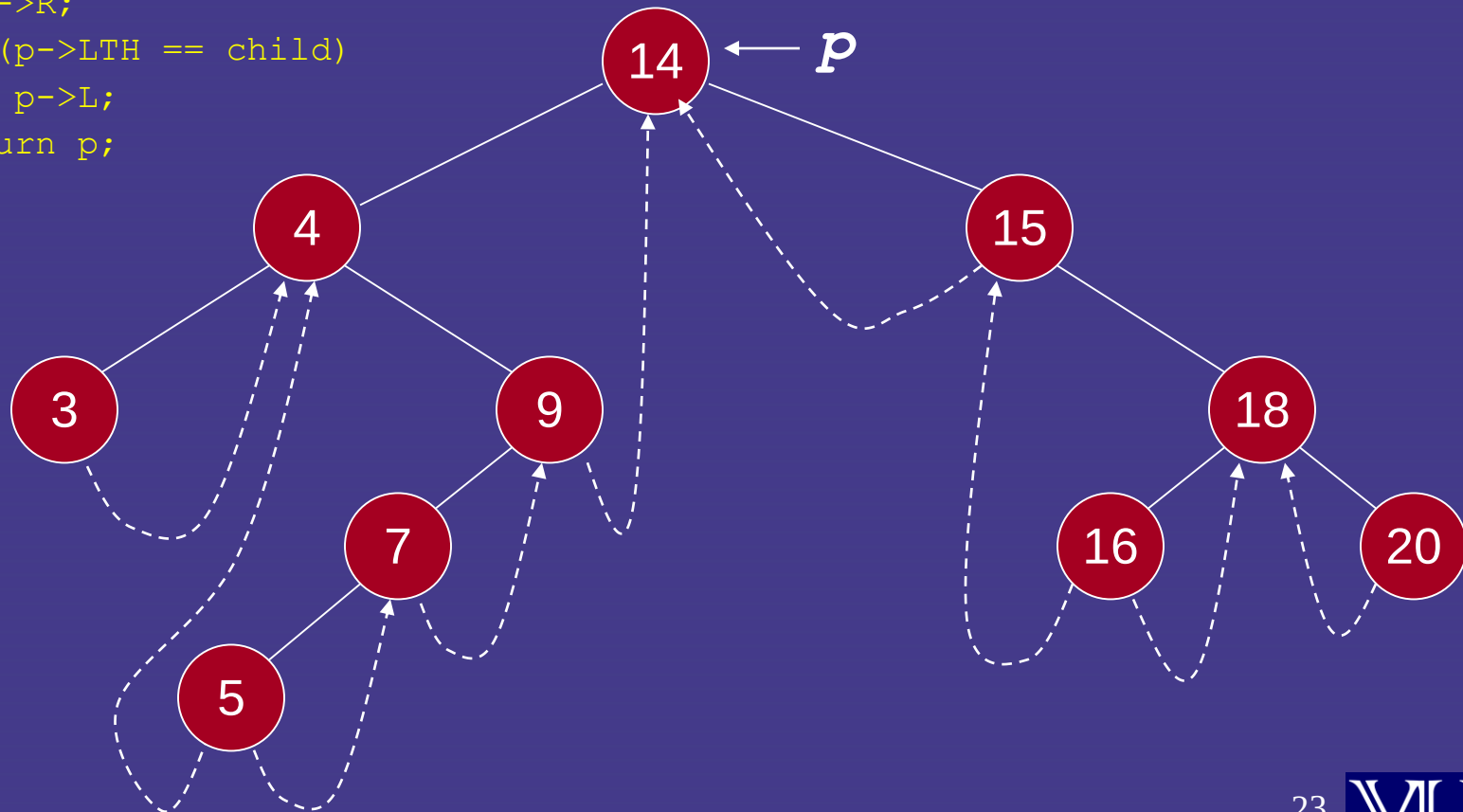
# Inorder traversal

- 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*.
- 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.



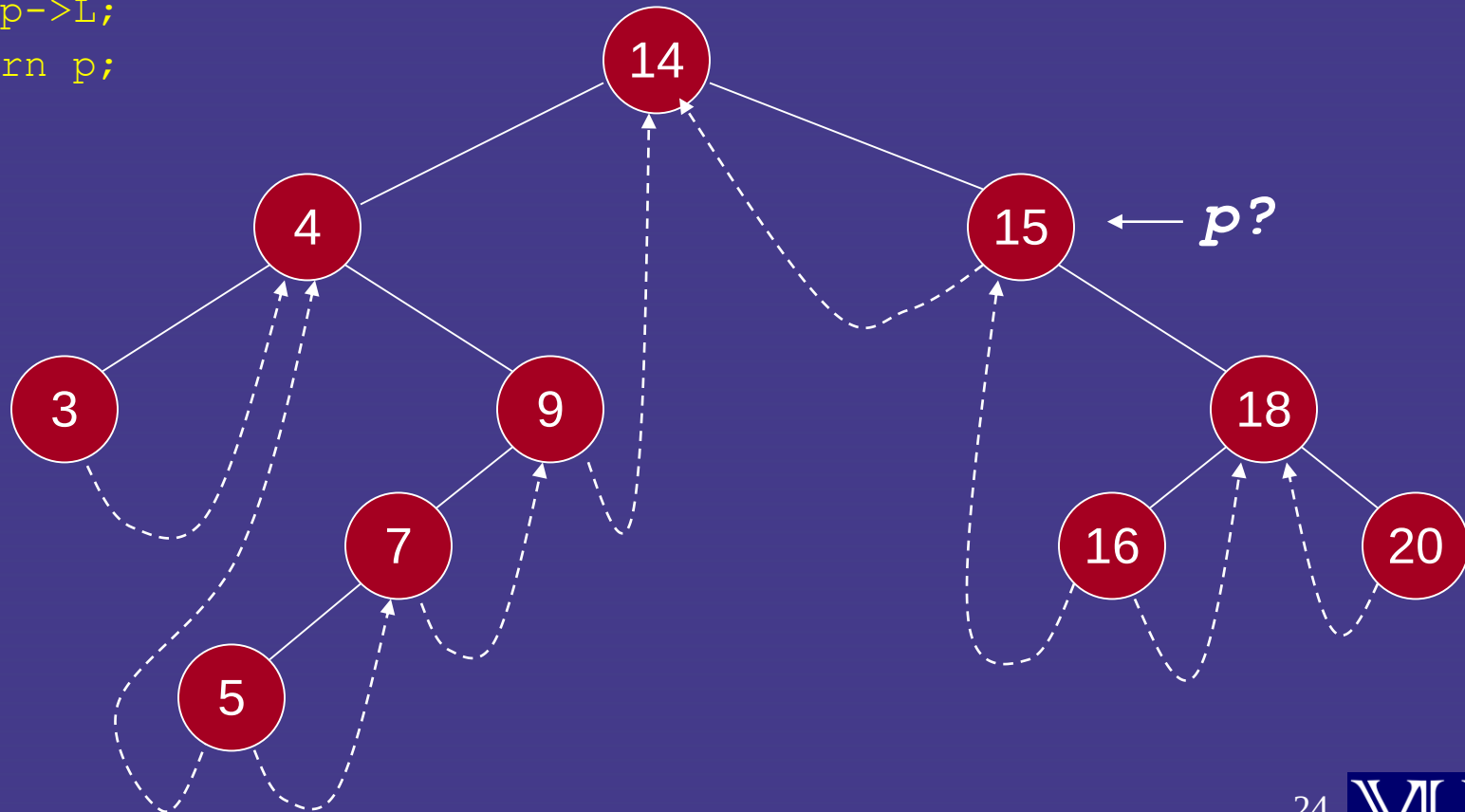
# Calling nextInorder with root

```
TreeNode* nextInorder(TreeNode* p){  
    if(p->RTH == thread) return(p->R);  
    else {  
        p = p->R;  
        while(p->LTH == child)  
            p = p->L;  
        return p;  
    }  
}
```



# Calling nextInorder with root

```
TreeNode* nextInorder(TreeNode* p){  
    if(p->RTH == thread) return(p->R);  
    else {  
        p = p->R;  
        while(p->LTH == child)  
            p = p->L;  
        return p;  
    }  
}
```

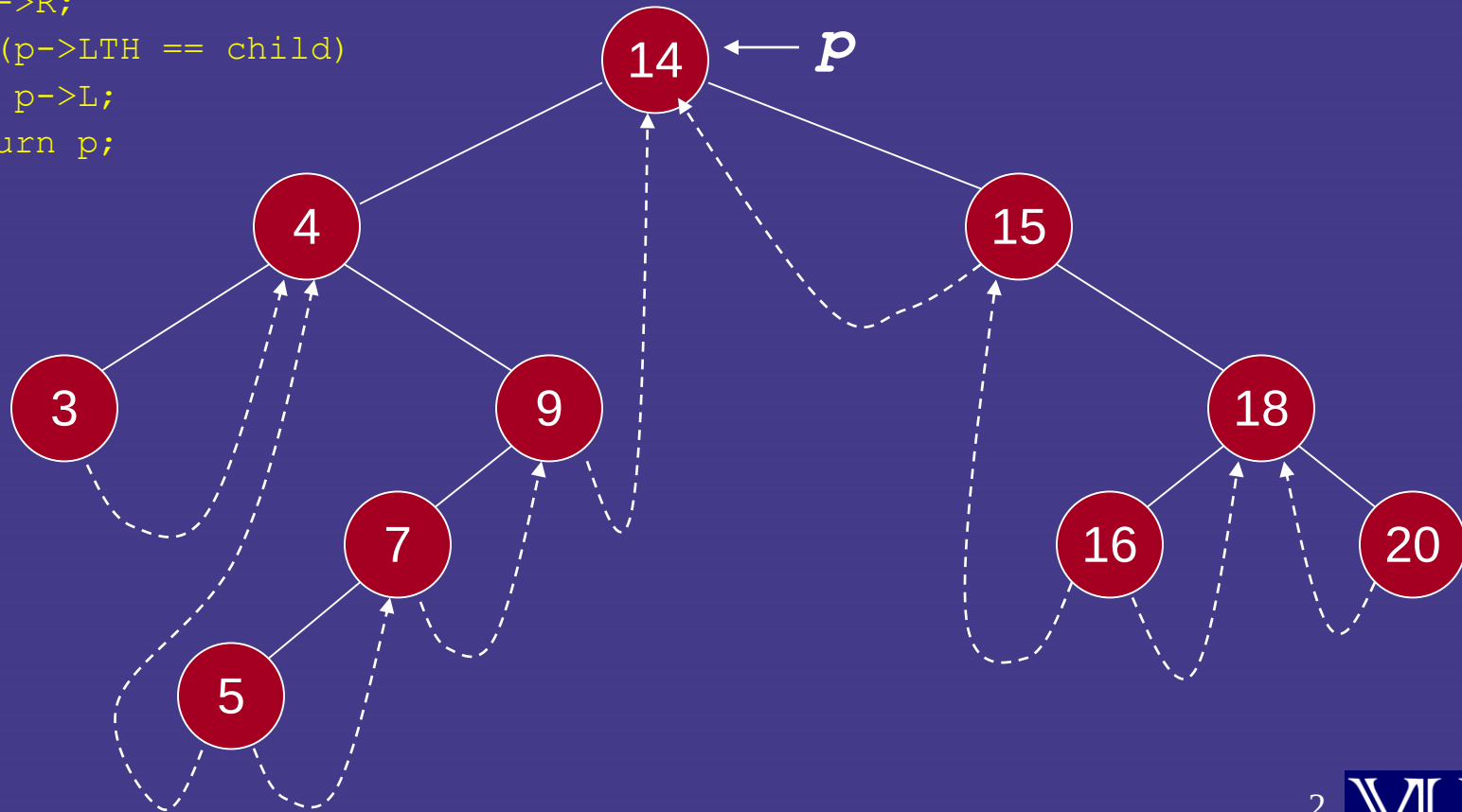


# Data Structures

Lecture 28  
Sohail Aslam

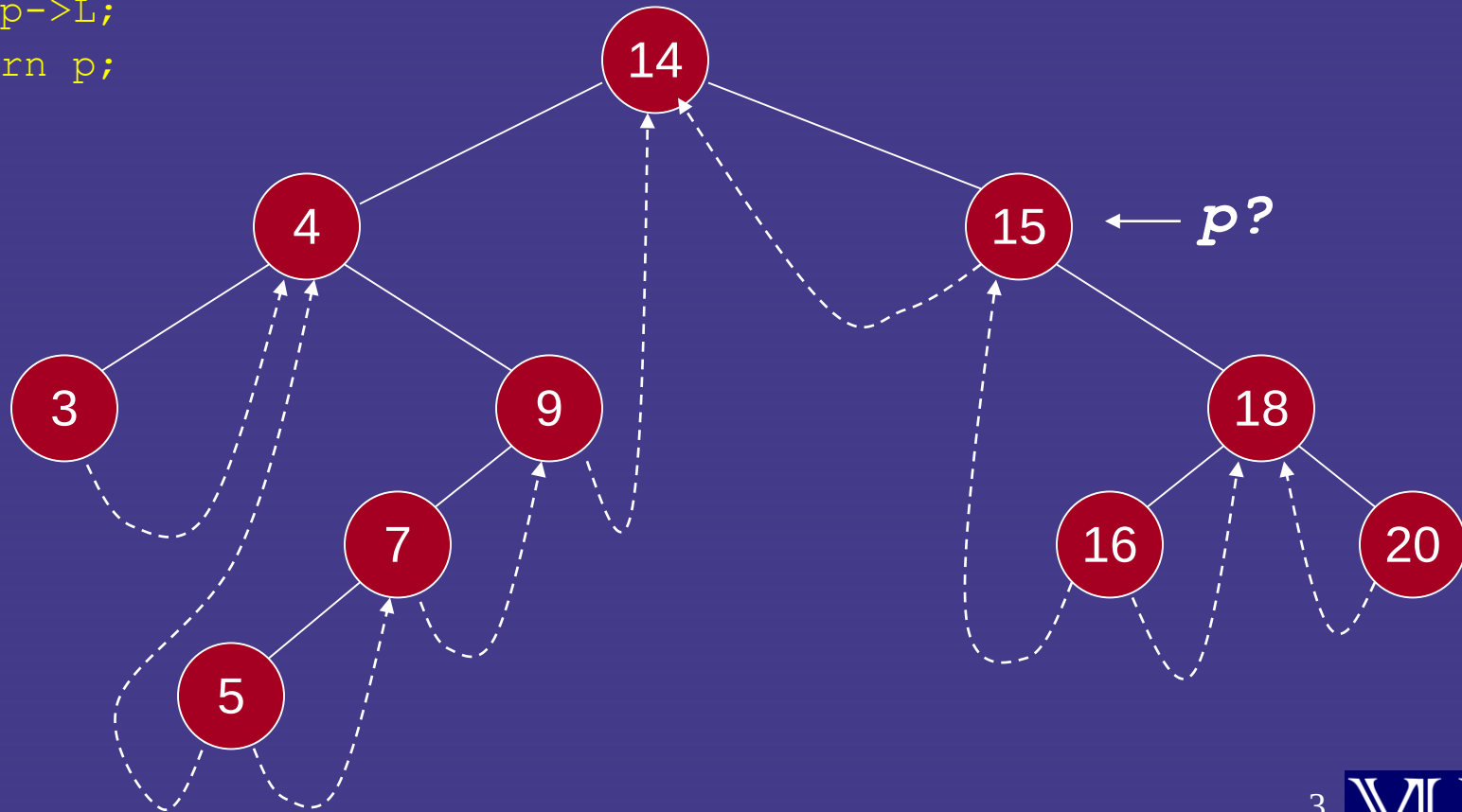
# Calling nextInorder with root

```
TreeNode* nextInorder(TreeNode* p){  
    if(p->RTH == thread) return(p->R);  
    else {  
        p = p->R;  
        while(p->LTH == child)  
            p = p->L;  
        return p;  
    }  
}
```



# Calling nextInorder with root

```
TreeNode* nextInorder(TreeNode* p){  
    if(p->RTH == thread) return(p->R);  
    else {  
        p = p->R;  
        while(p->LTH == child)  
            p = p->L;  
        return p;  
    }  
}
```



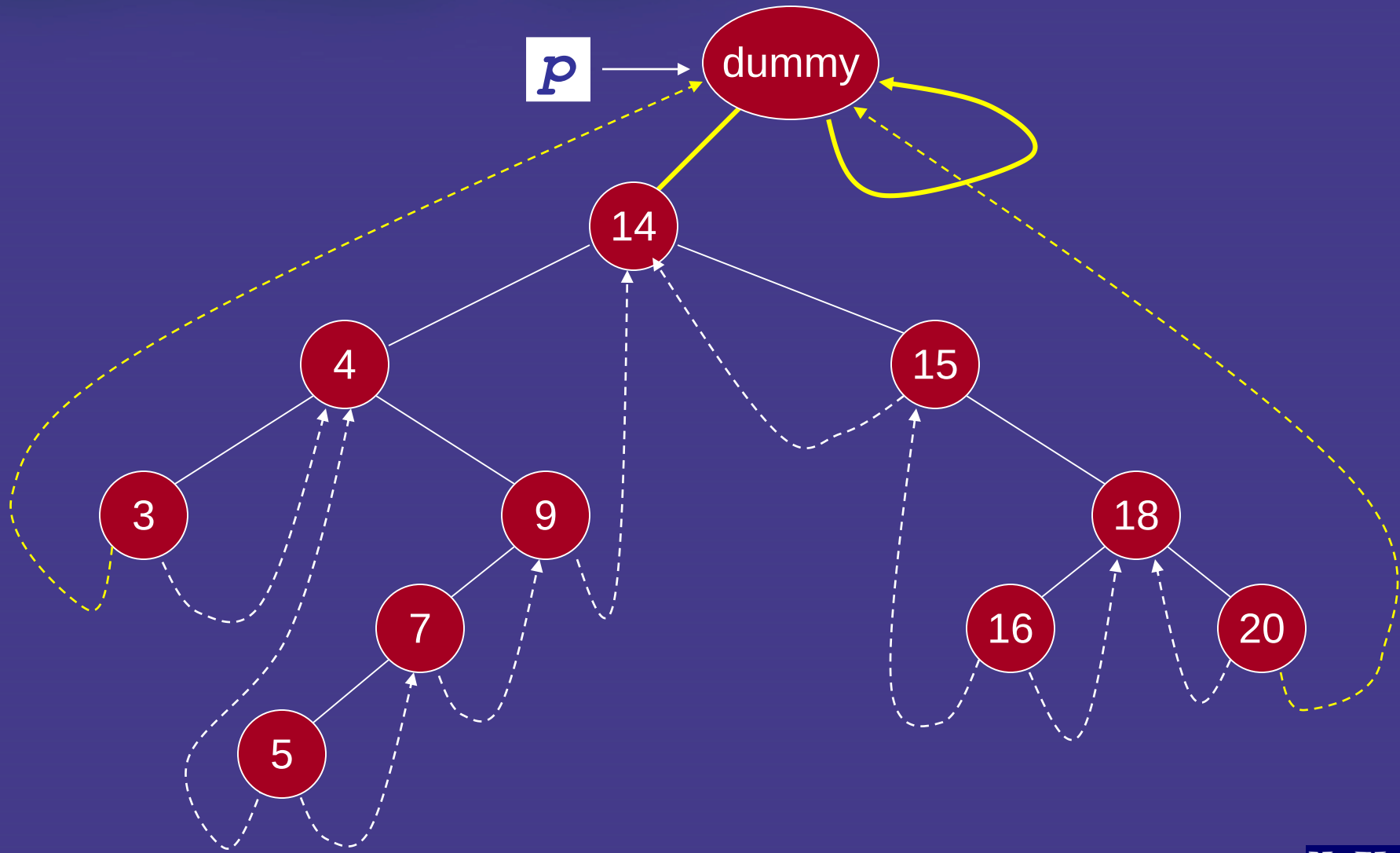


# Inorder traversal

```
void fastInorder(TreeNode* p)
{
    while( (p=nexInorder(p)) != dummy)
        cout << p->getInfo();
}
```

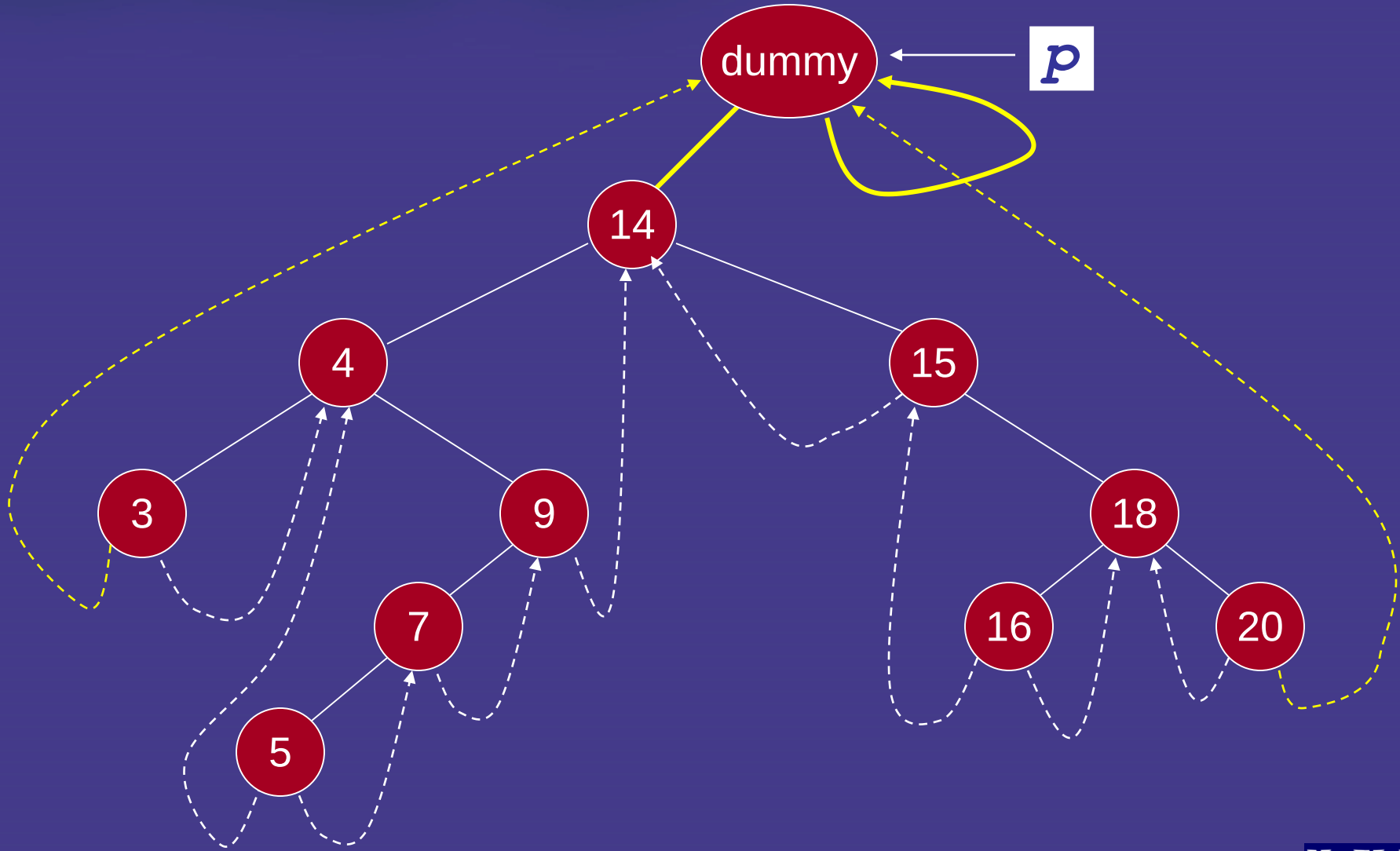
- Start the inorder traversal by calling  
**fastInorder(dummy) .**

# Trace or nextInorder

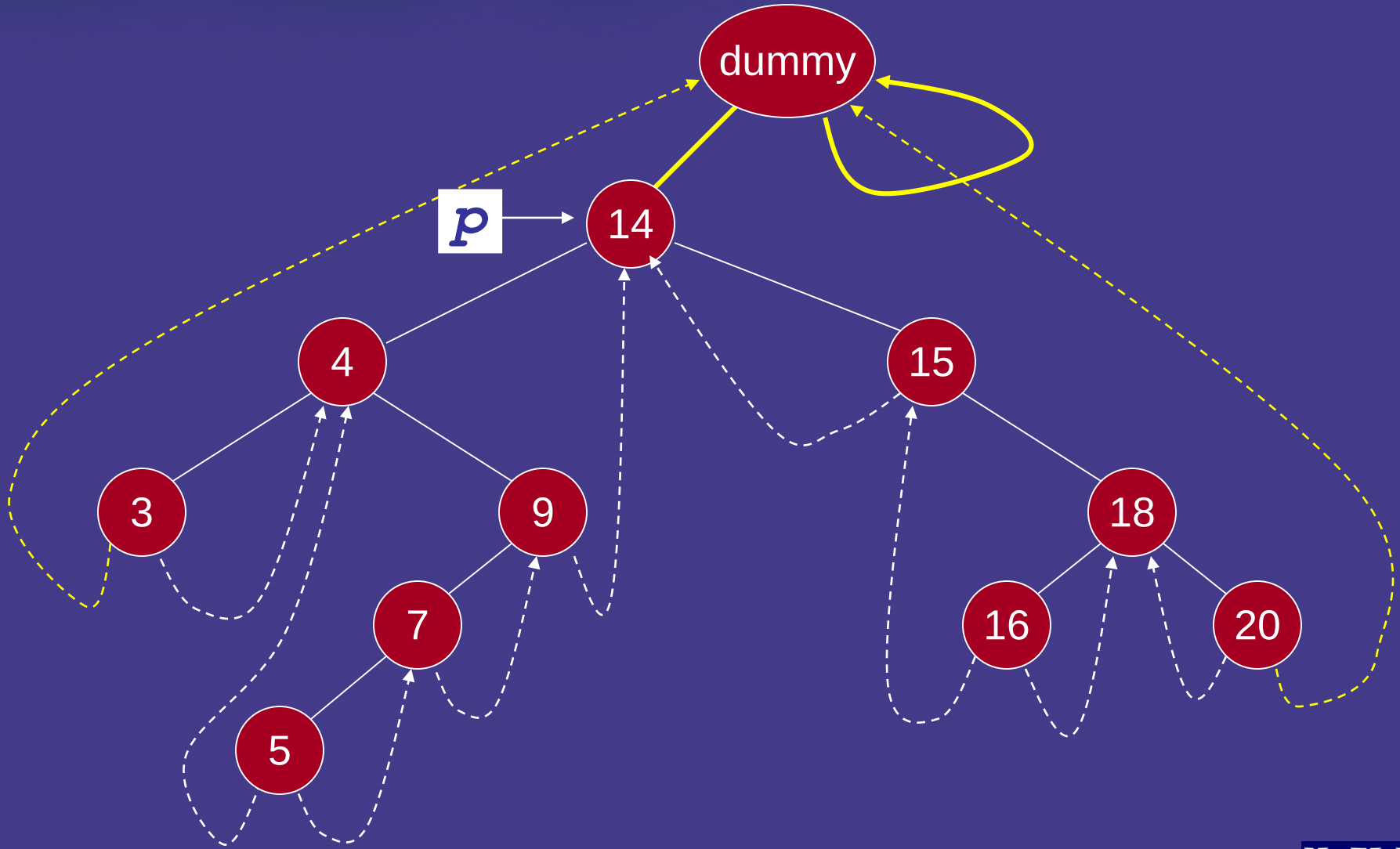




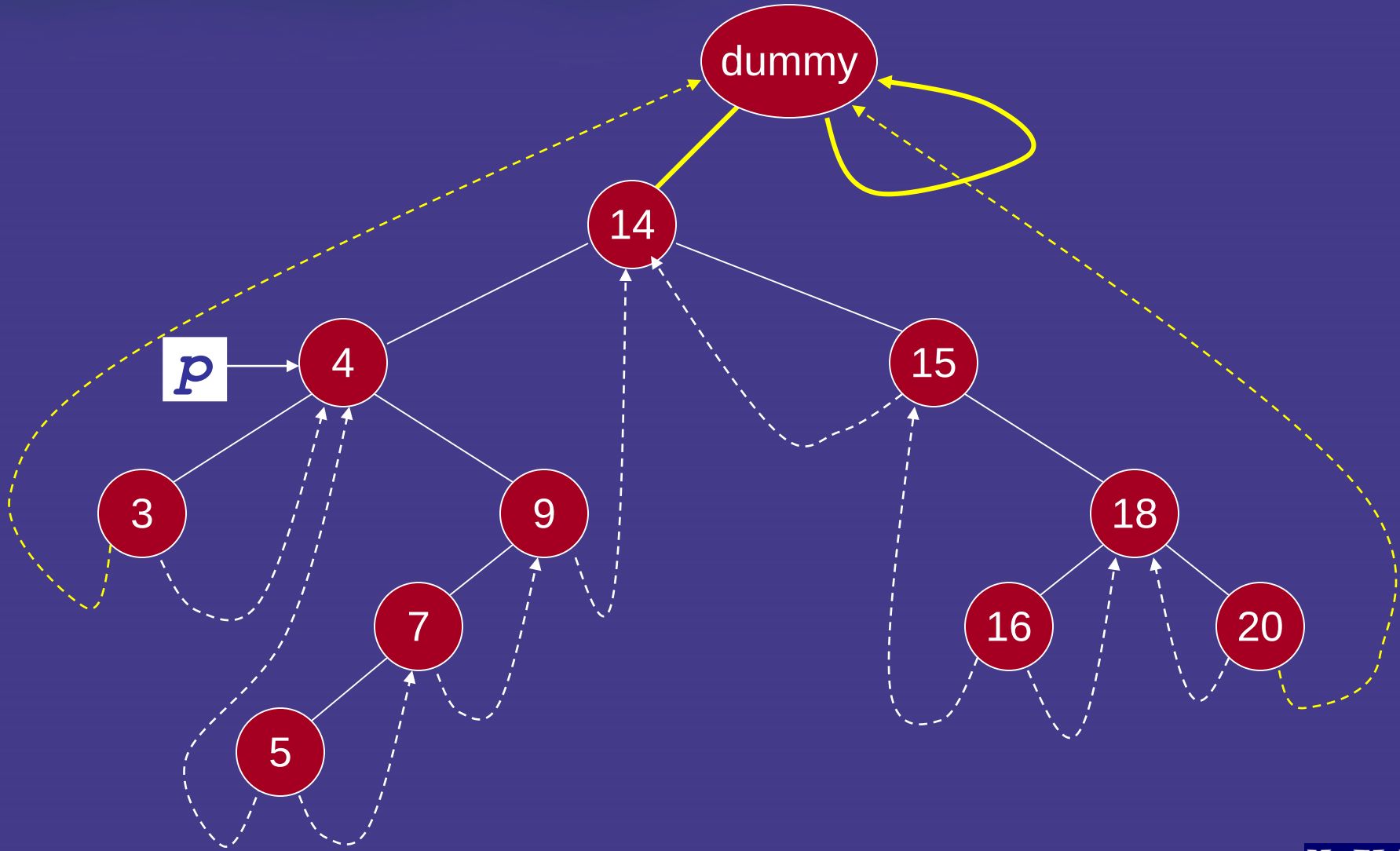
# Trace or nextInorder



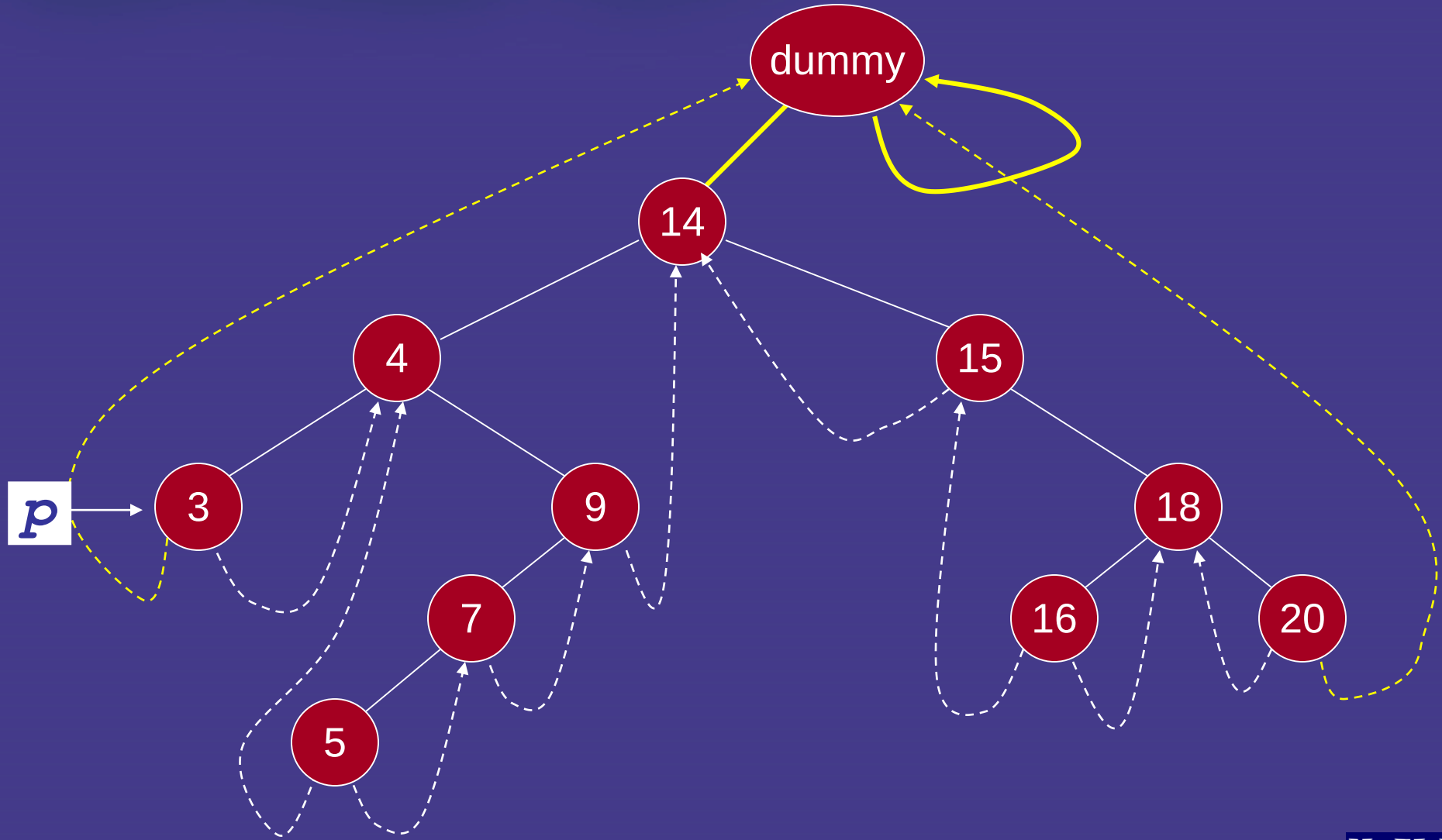
# Trace or nextInorder



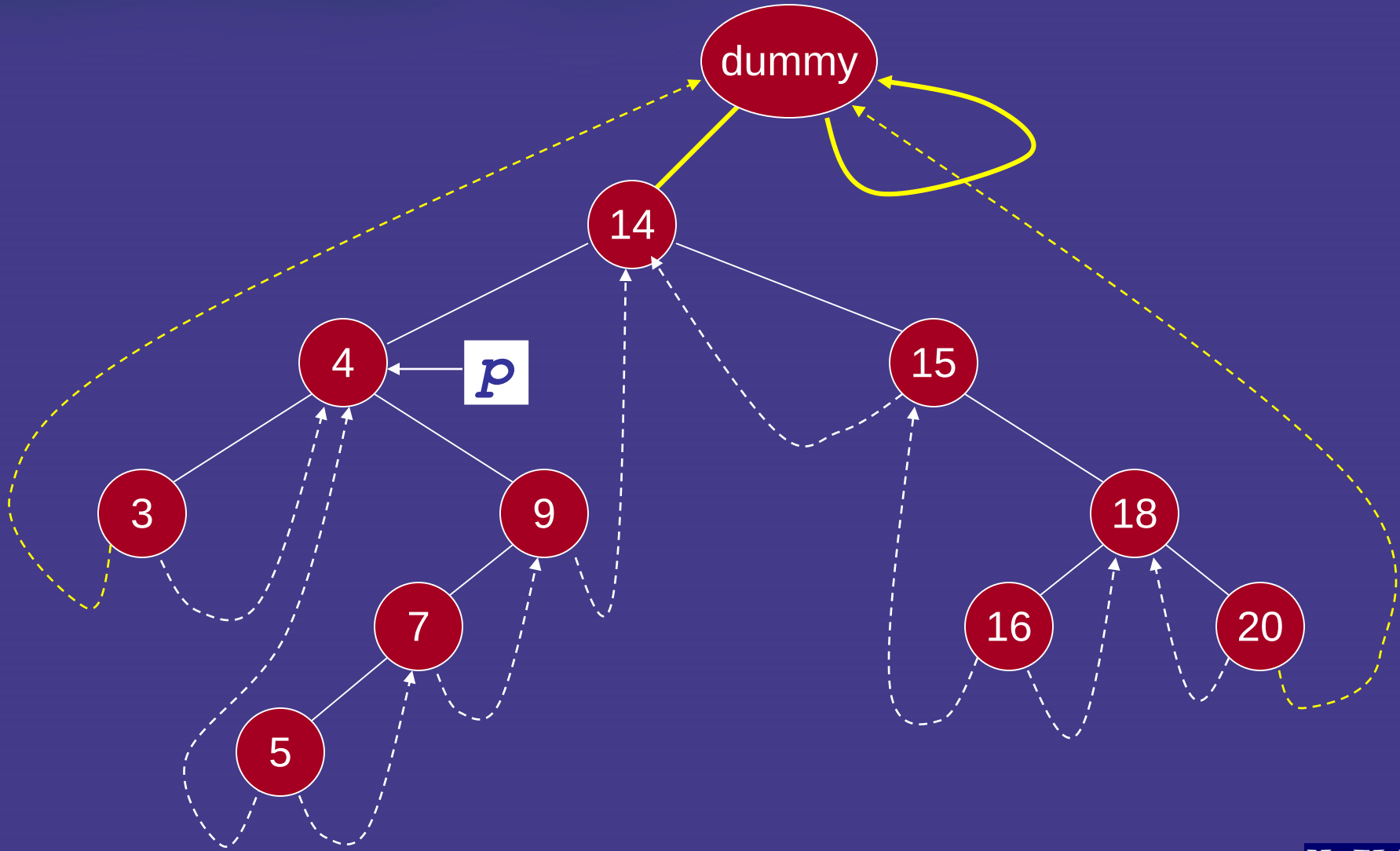
# Trace or nextInorder



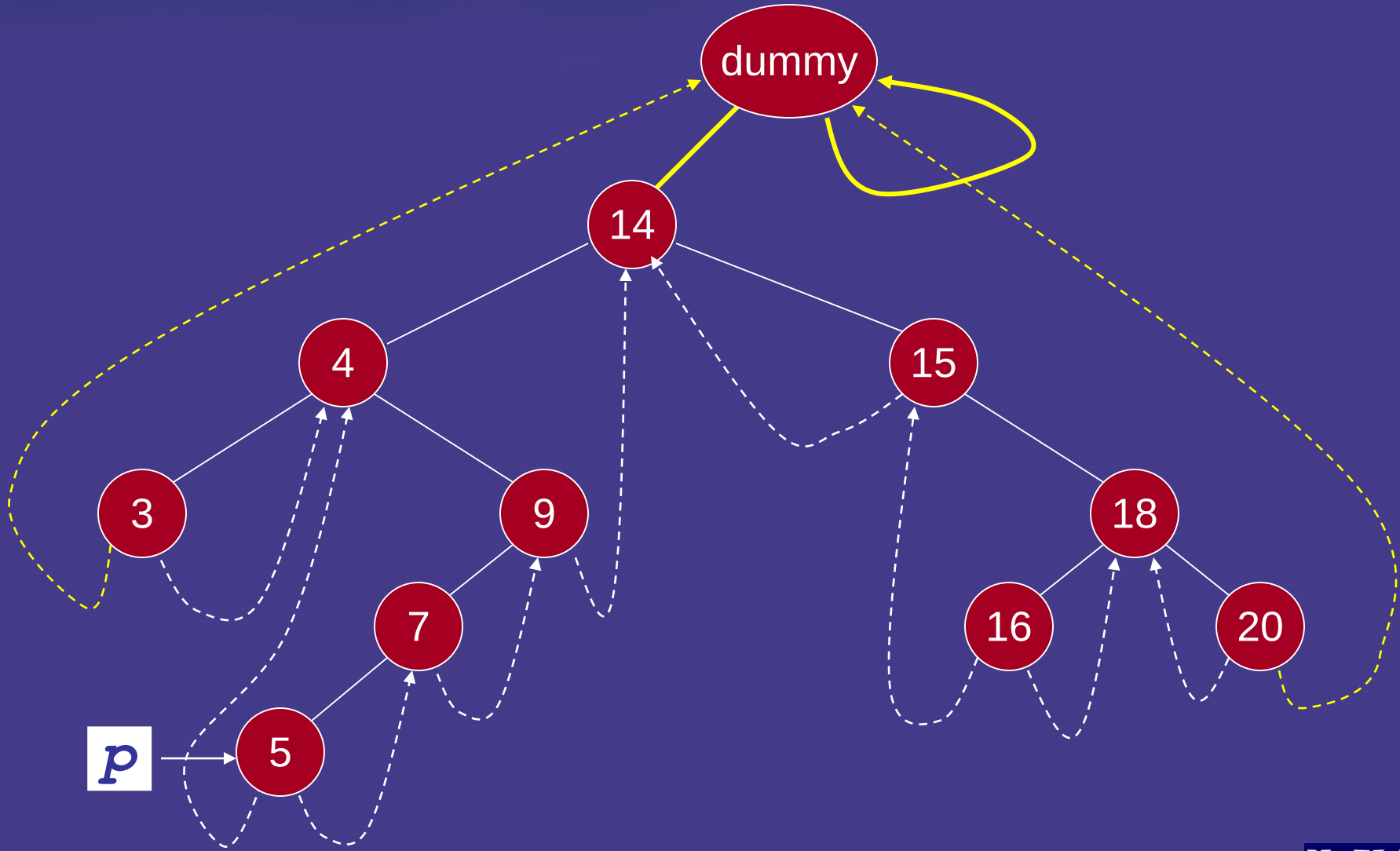
# Trace or nextInorder



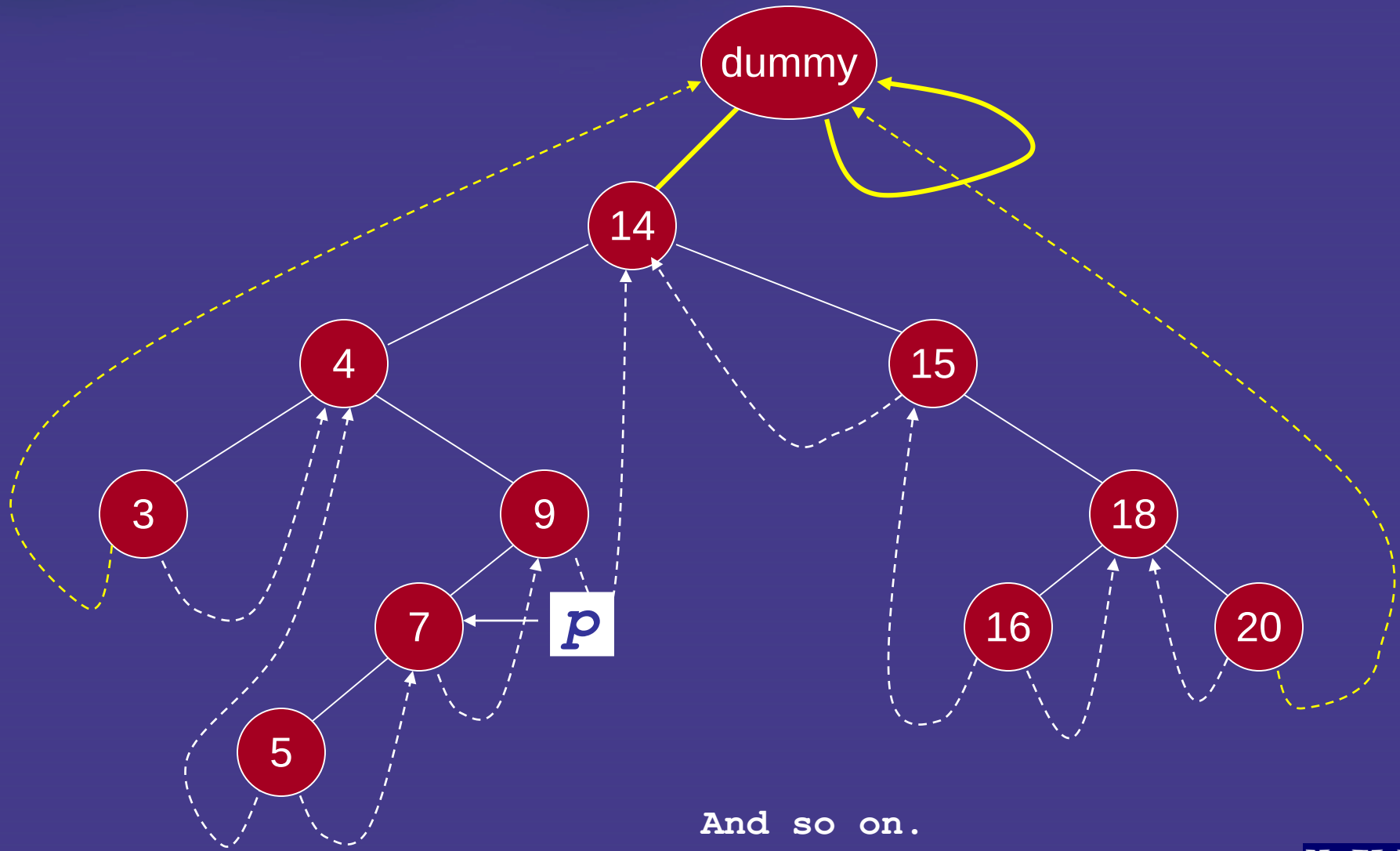
# Trace or nextInorder



# Trace or nextInorder



# Trace or nextInorder



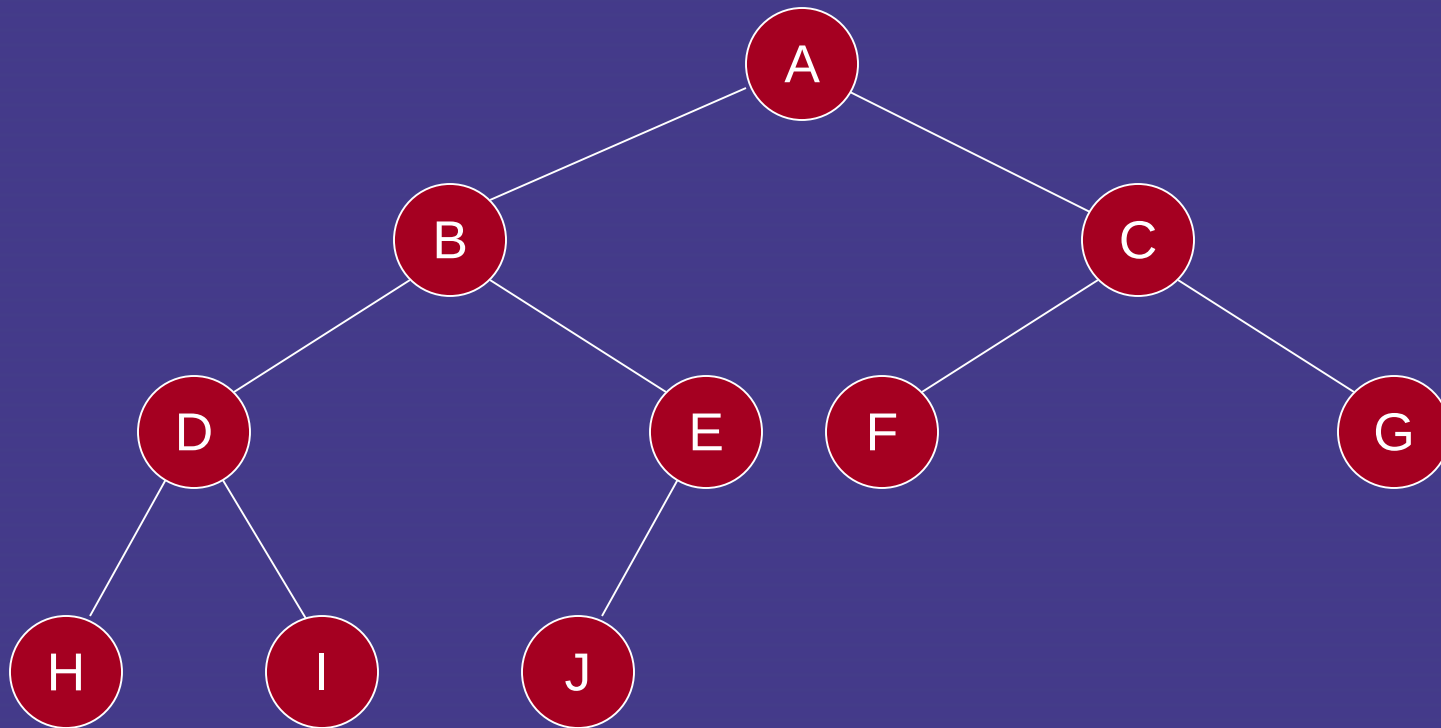
# Complete Binary Tree



# 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.
- Different definition of complete binary tree than an earlier definition we had.
- The earlier definition could be called a perfect binary tree.

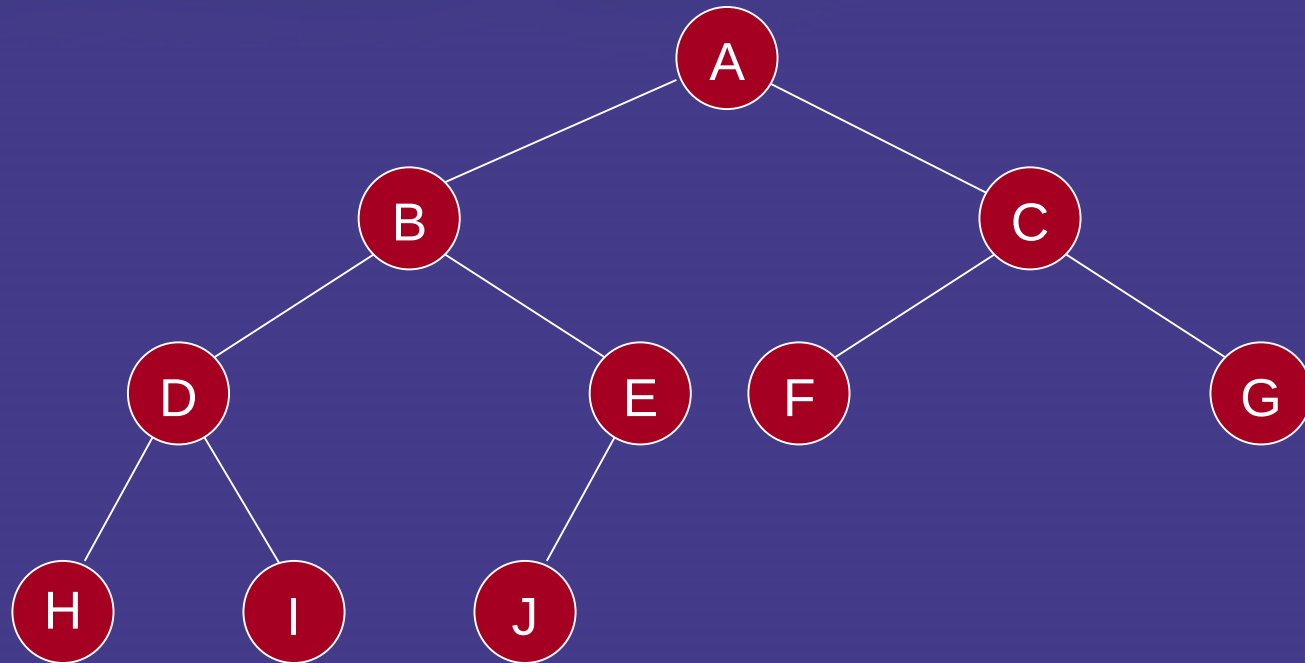
# Complete Binary Tree



# Complete Binary Tree

- Recall that such a tree of height  $h$  has between  $2^h$  to  $2^{h+1} - 1$  nodes.
- The height of such a tree is thus  $\lceil \log_2 N \rceil$  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.

# Complete Binary Tree



|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
|   | A | B | C | D | E | F | G | H | I | J  |    |    |    |    |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

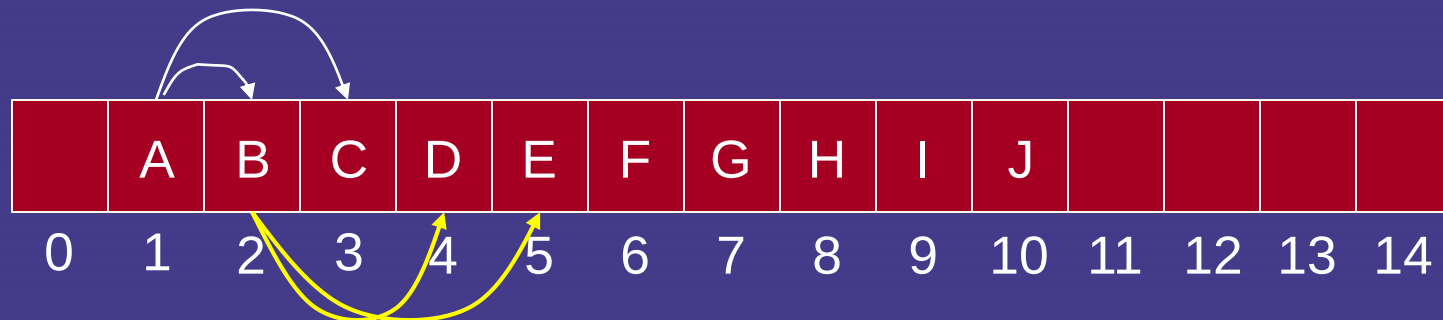
# Complete Binary Tree

- For any array element at position  $i$ , the left child is at  $2i$ , the right child is at  $(2i + 1)$  and the parent is at  $\lfloor \frac{i}{2} \rfloor$ .



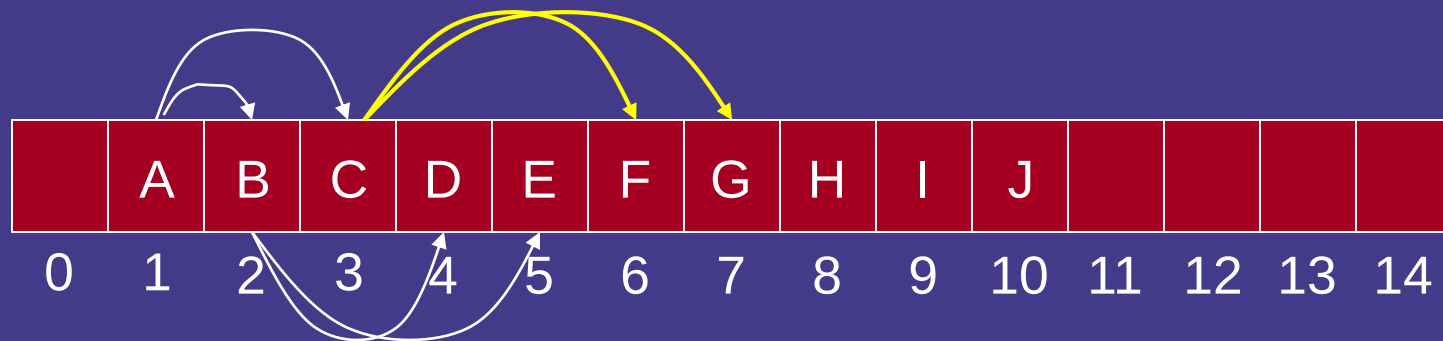
# Complete Binary Tree

- For any array element at position  $i$ , the left child is at  $2i$ , the right child is at  $(2i + 1)$  and the parent is at  $\lfloor \frac{i}{2} \rfloor$ .



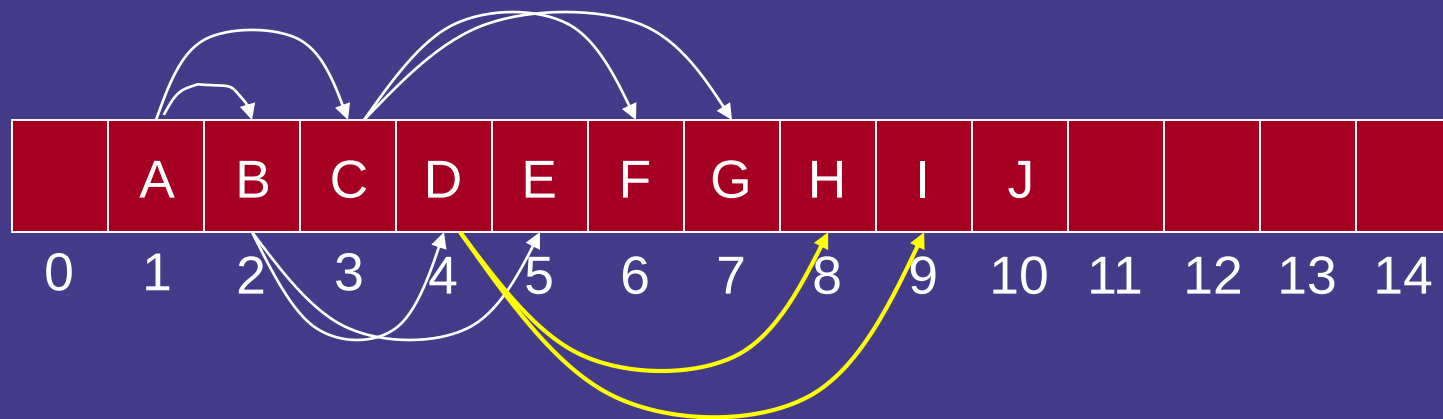
# Complete Binary Tree

- For any array element at position  $i$ , the left child is at  $2i$ , the right child is at  $(2i + 1)$  and the parent is at  $\lfloor \frac{i}{2} \rfloor$ .



# Complete Binary Tree

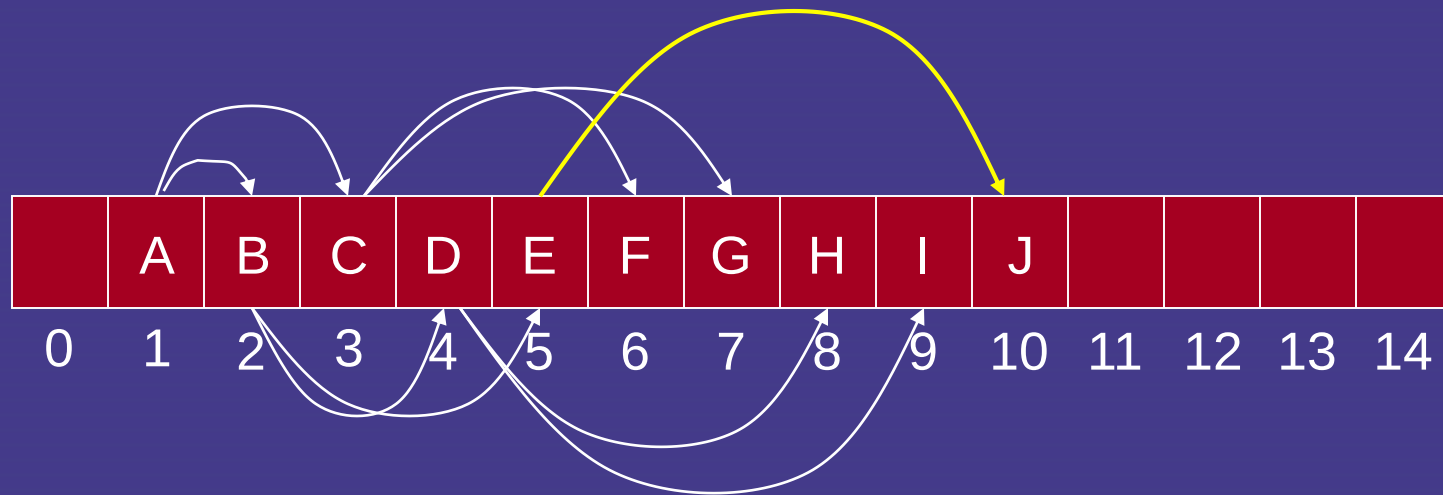
- For any array element at position  $i$ , the left child is at  $2i$ , the right child is at  $(2i + 1)$  and the parent is at  $\lfloor \frac{i}{2} \rfloor$ .



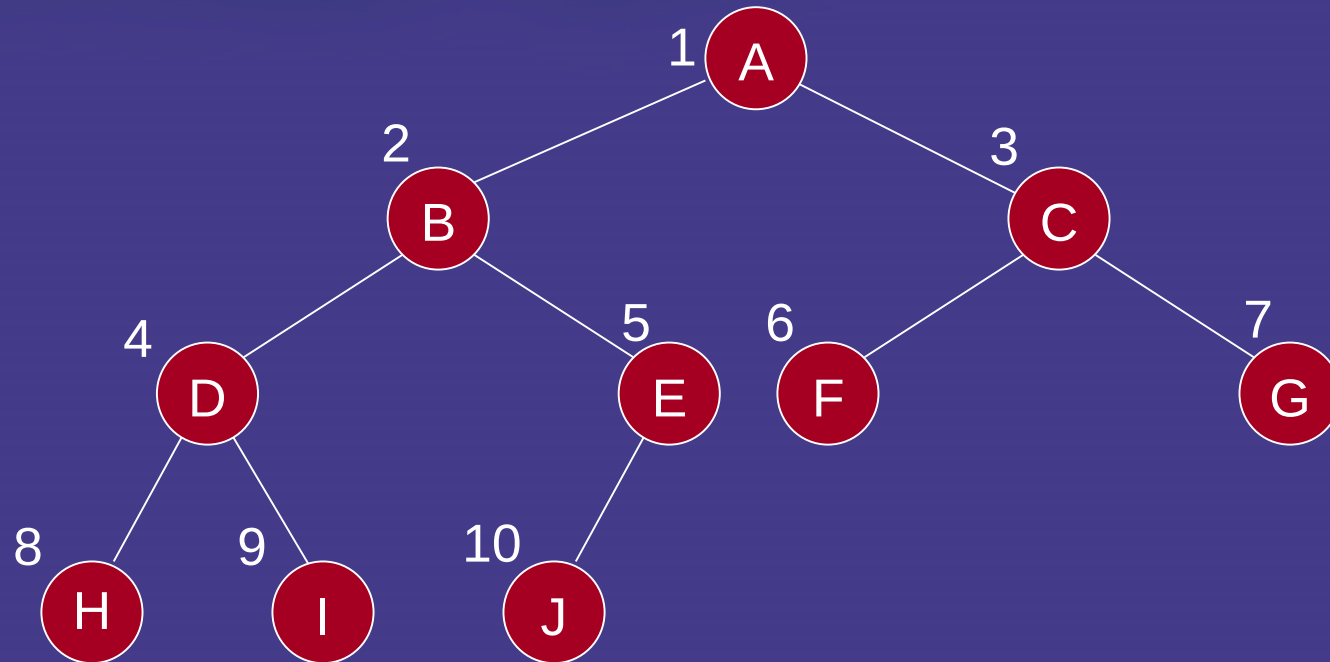


# Complete Binary Tree

- For any array element at position  $i$ , the left child is at  $2i$ , the right child is at  $(2i + 1)$  and the parent is at  $\lfloor i/2 \rfloor$ .



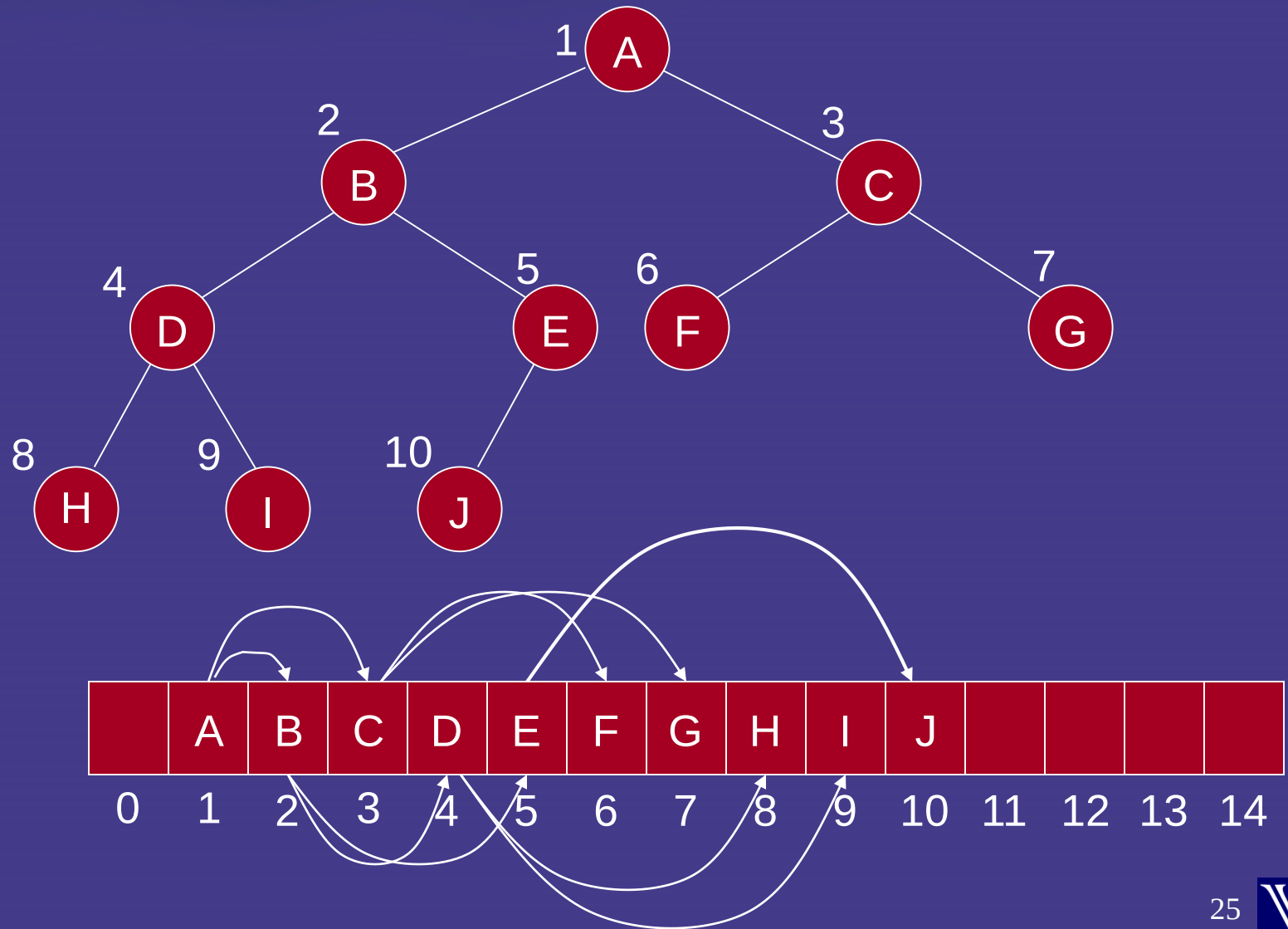
# Complete Binary Tree



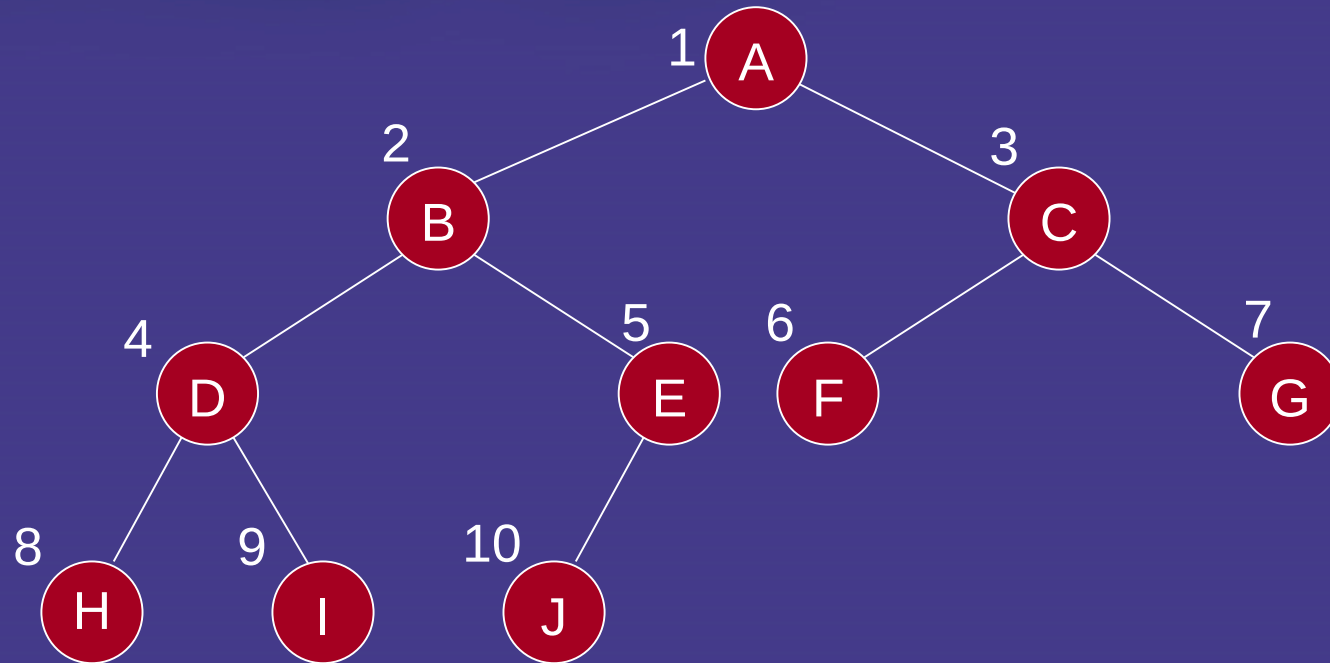
|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
|   | A | B | C | D | E | F | G | H | I | J  |    |    |    |    |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Level-order numbers  array index

# Complete Binary Tree



# Complete Binary Tree



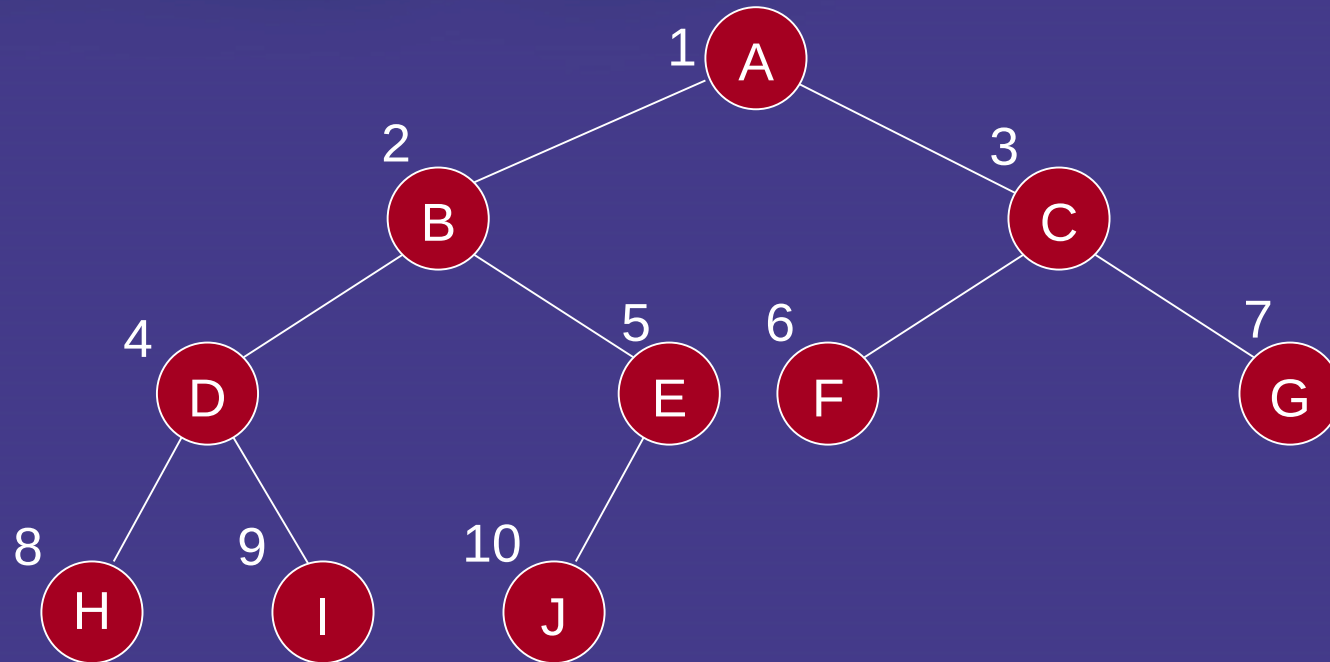
|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
|   | A | B | C | D | E | F | G | H | I | J  |    |    |    |    |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

- Question: why don't we store all binary trees in arrays?  
Why use pointers?

# Data Structures

Lecture 29  
Sohail Aslam

# Complete Binary Tree



|   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|
|   | A | B | C | D | E | F | G | H | I | J  |    |    |    |    |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

- Question: why don't we store all binary trees in arrays?  
Why use pointers?

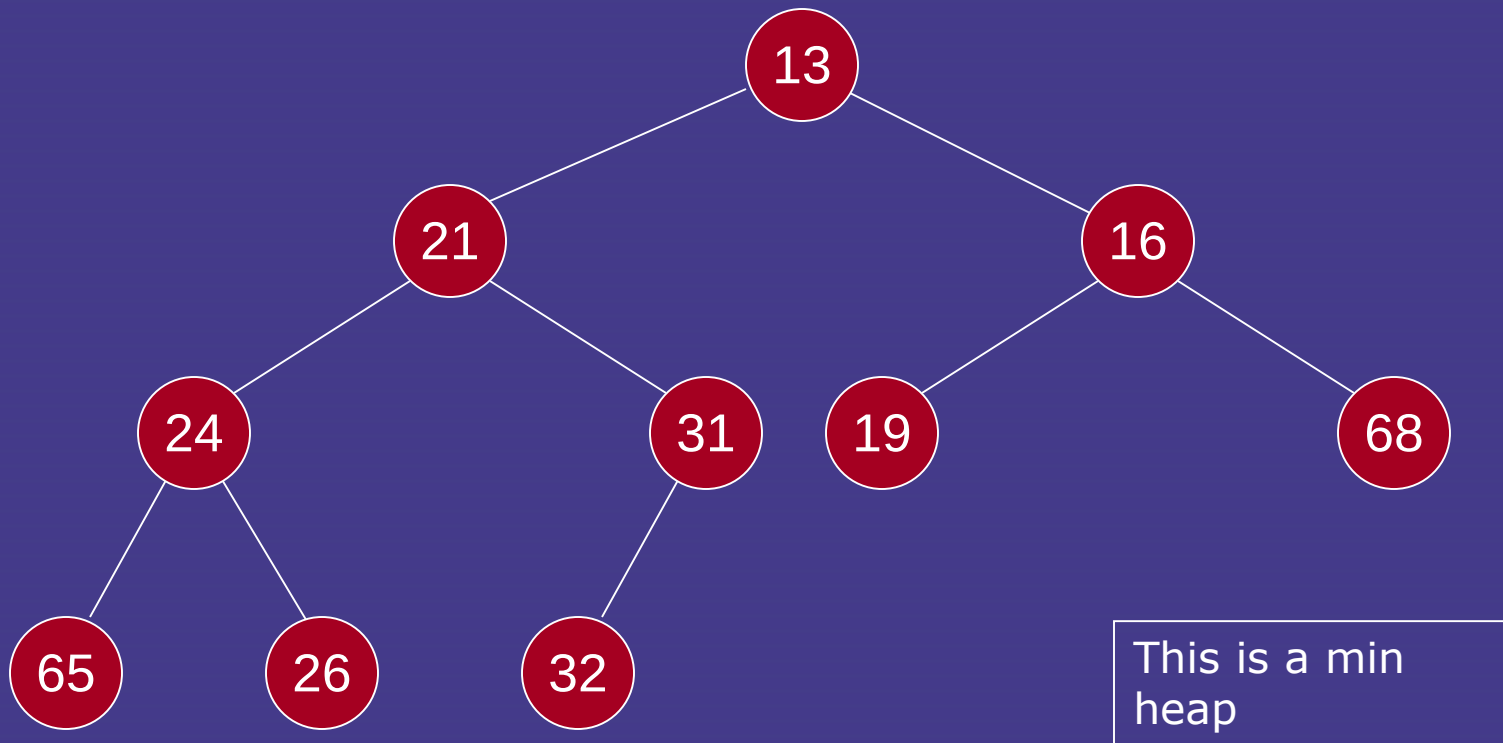
# The Heap ADT

# Heap

- A heap is a complete binary tree that conforms to the heap order.
- The *heap order* property: in a (min) heap, for every node  $X$ , the key in the parent is smaller than (or equal to) the key in  $X$ .
- Or, the parent node has key smaller than or equal to both of its children nodes.

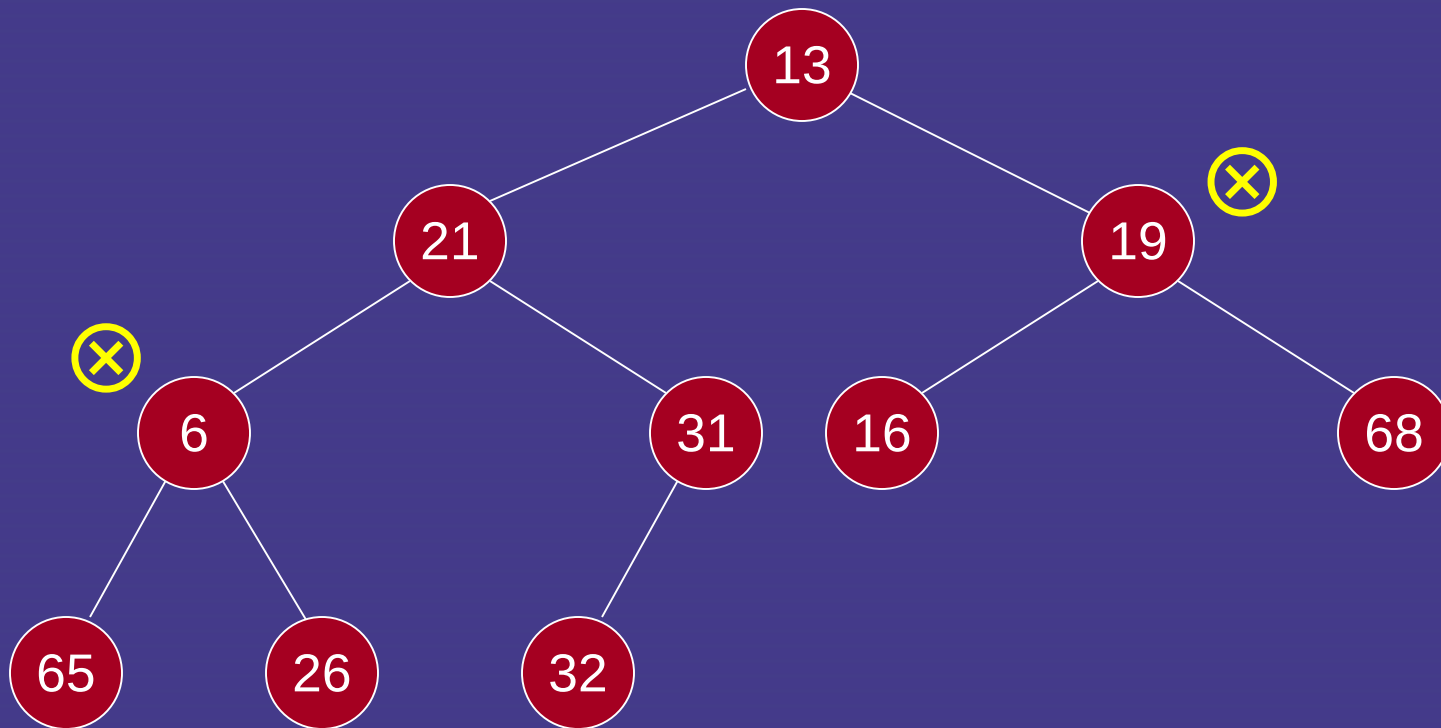


# Heap



# Heap

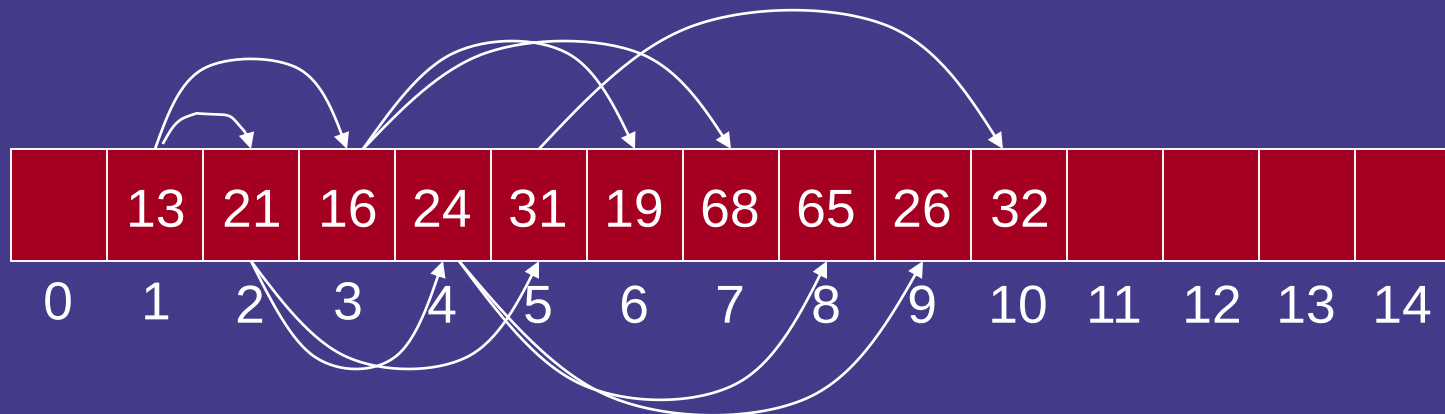
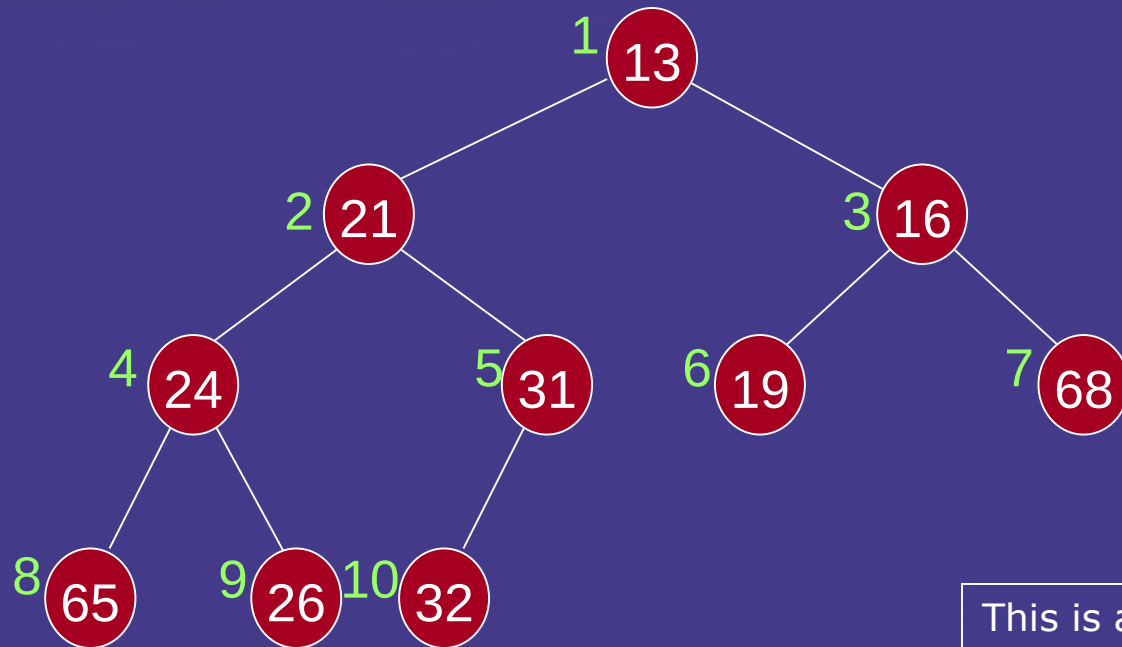
Not a heap: heap property violated



# Heap

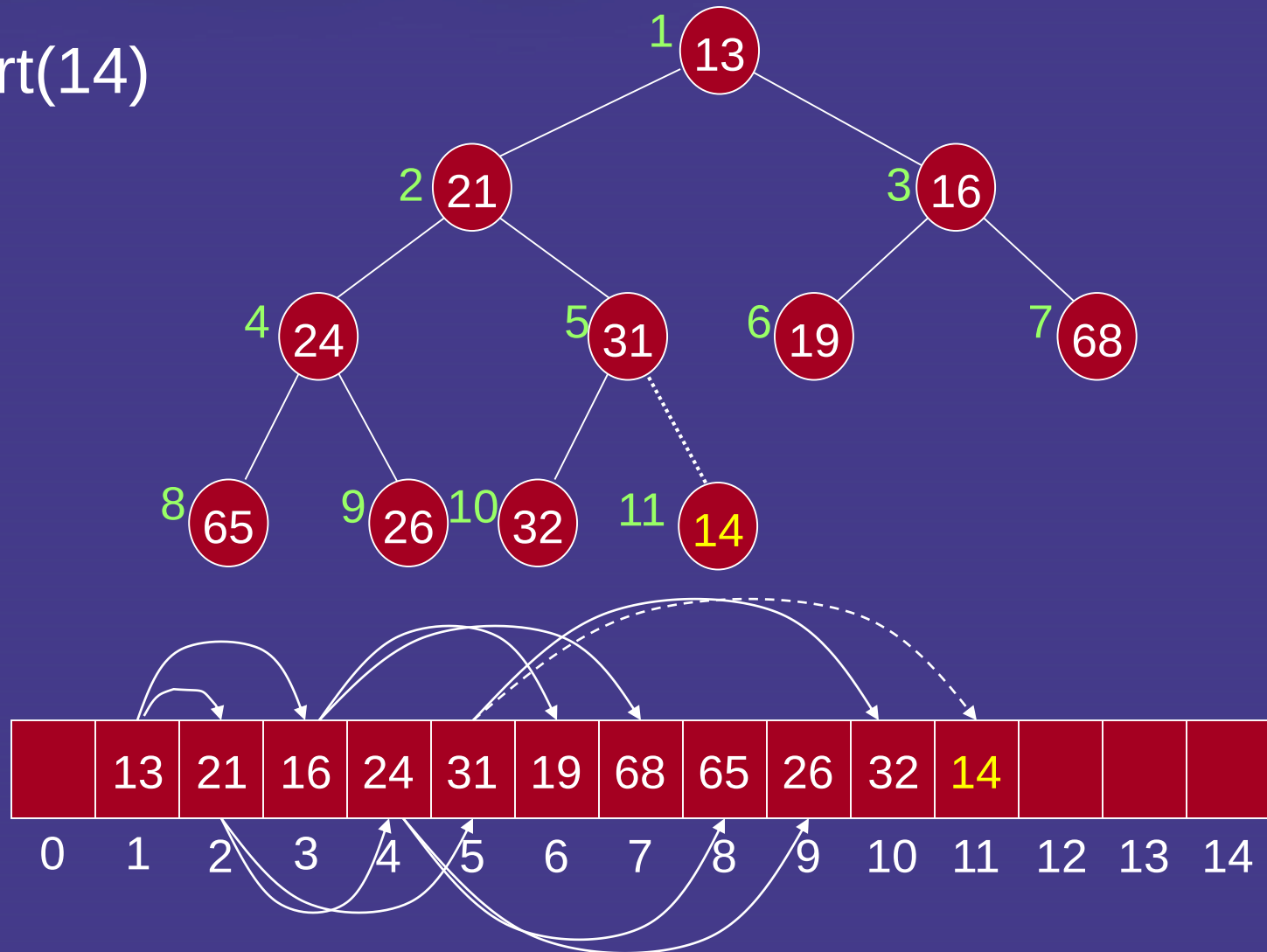
- Analogously, we can define a max-heap, where the parent has a key larger than the its two children.
- Thus the largest key would be in the root.

# Inserting into a Heap



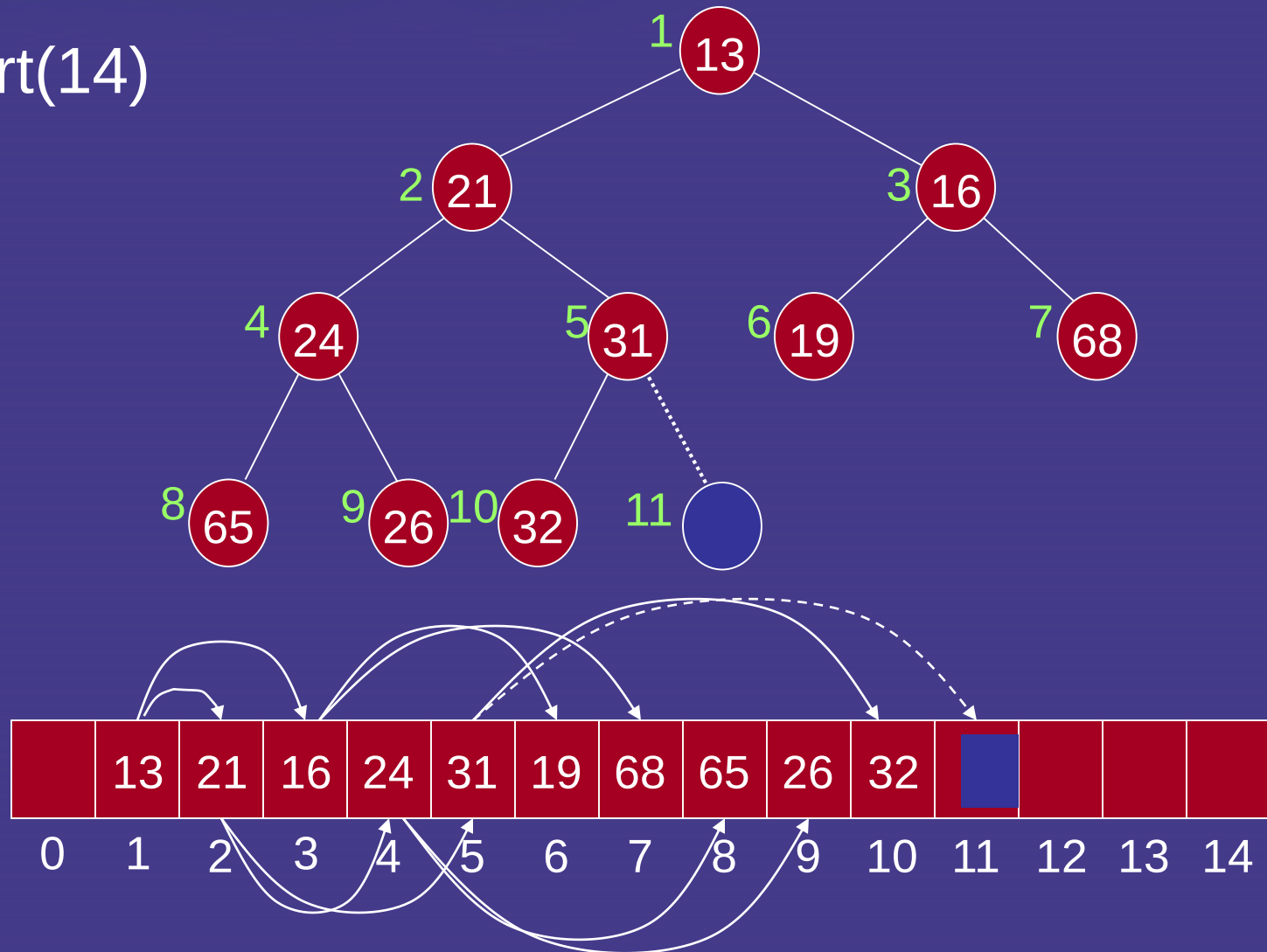
# Inserting into a Heap

insert(14)



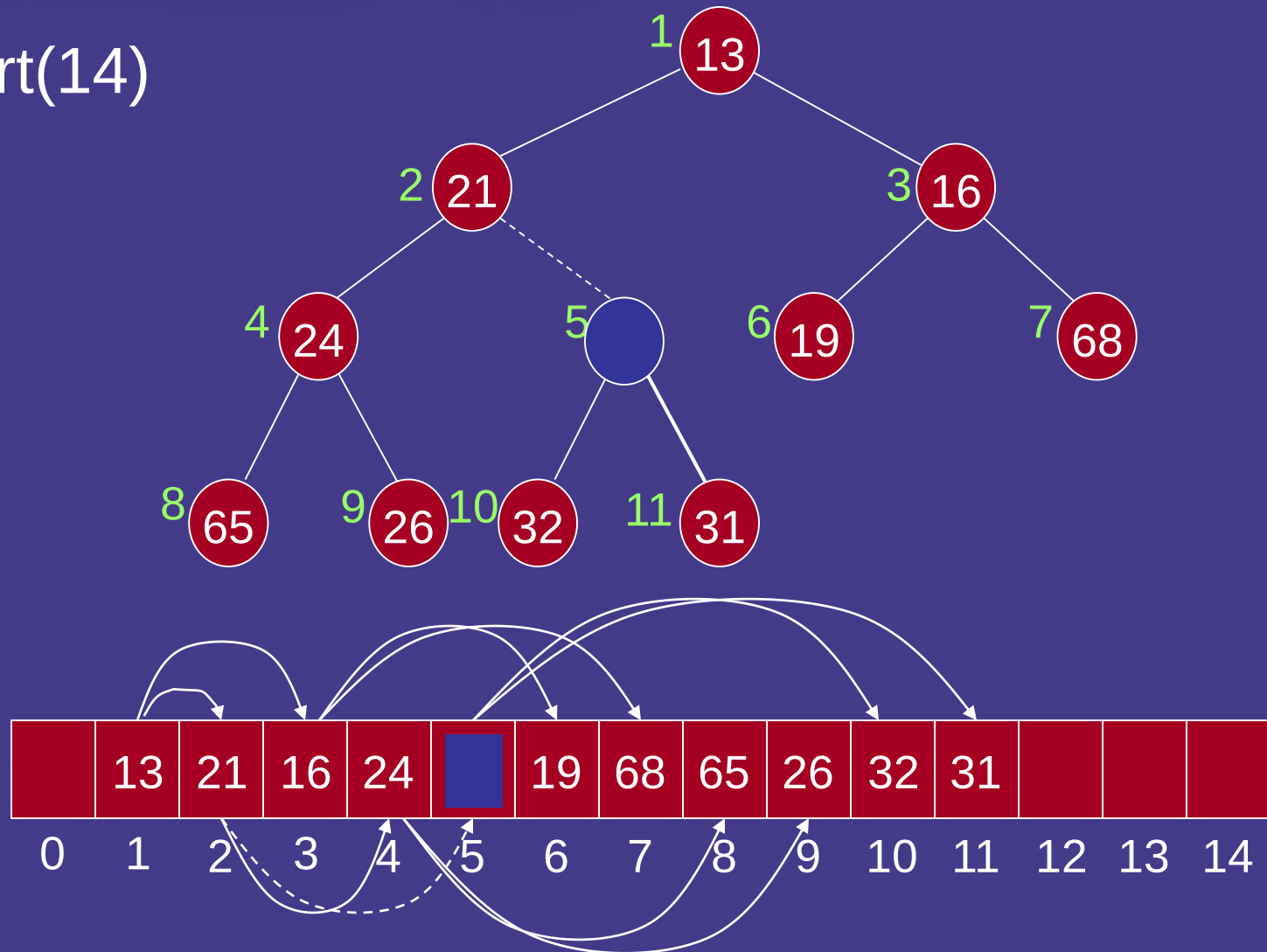
# Inserting into a Heap

insert(14)



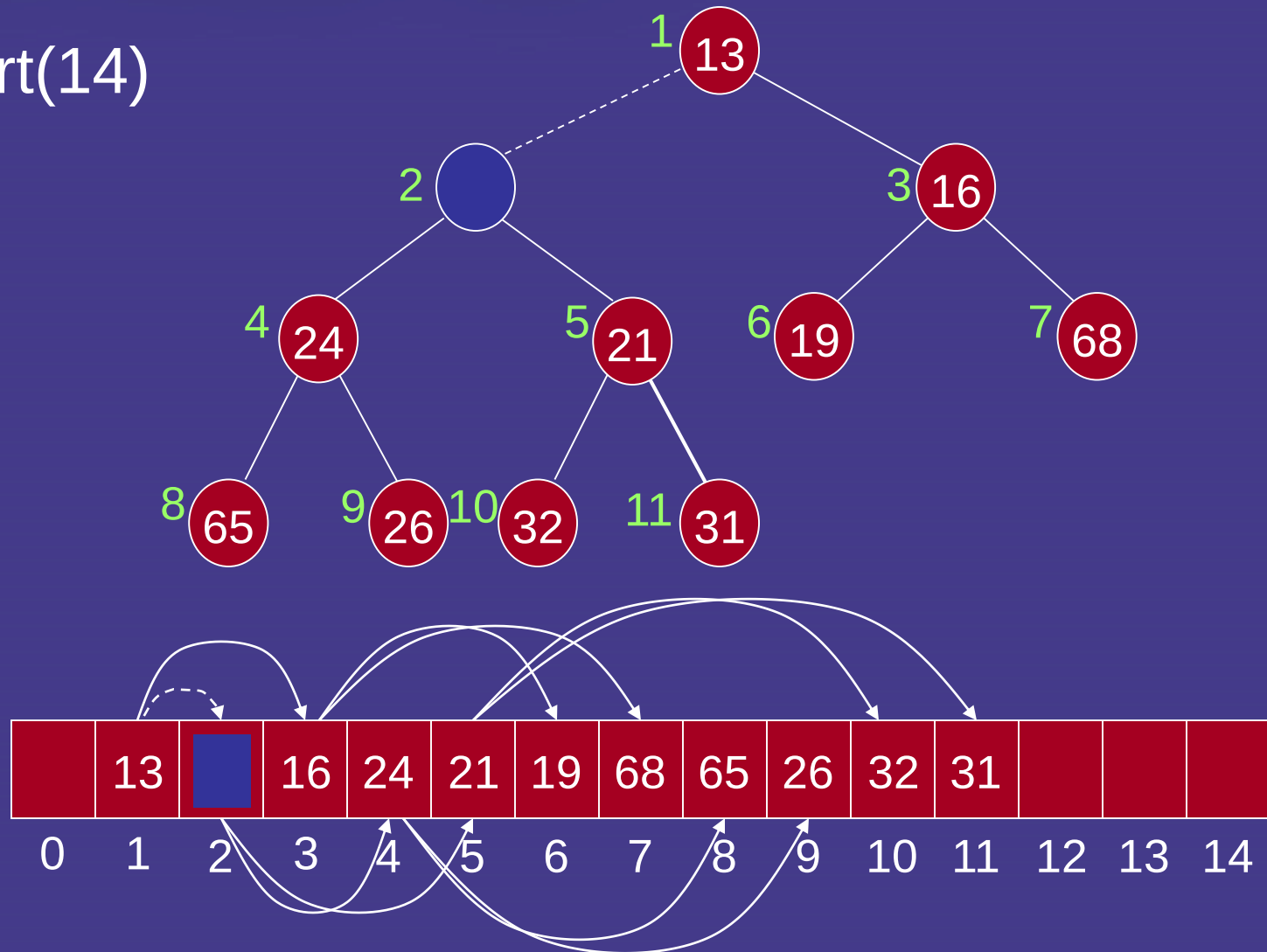
# Inserting into a Heap

insert(14)



# Inserting into a Heap

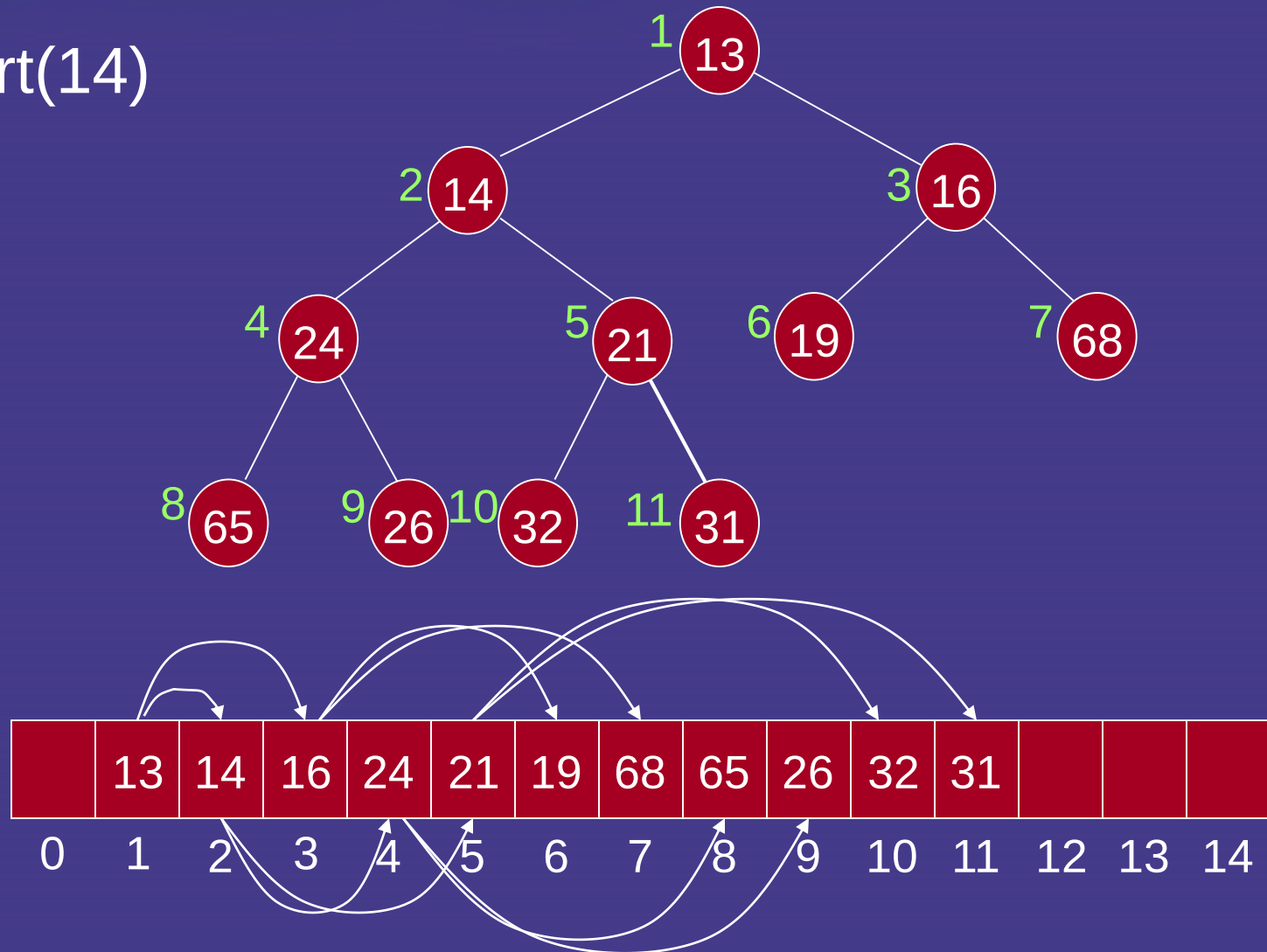
insert(14)





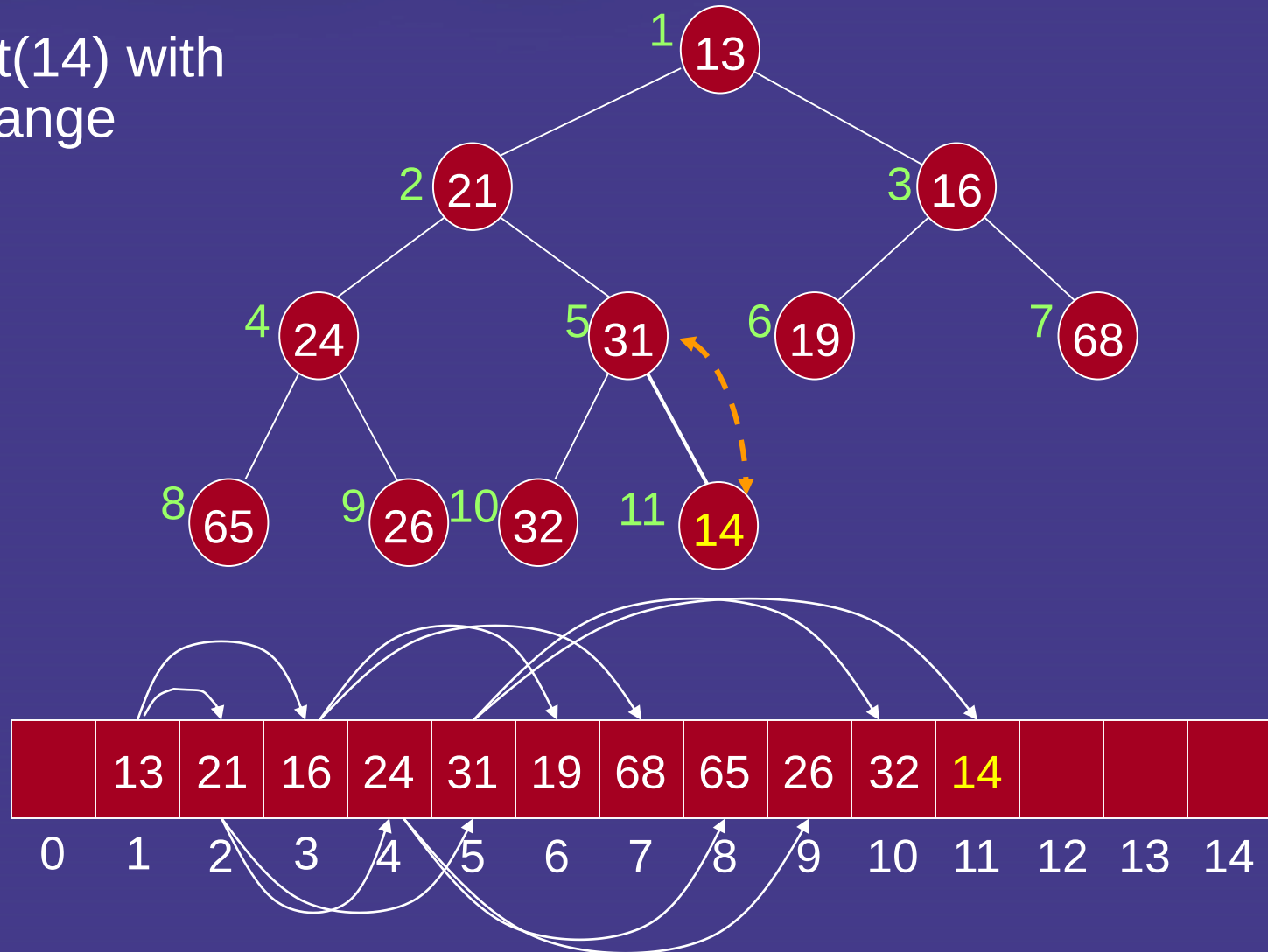
# Inserting into a Heap

insert(14)



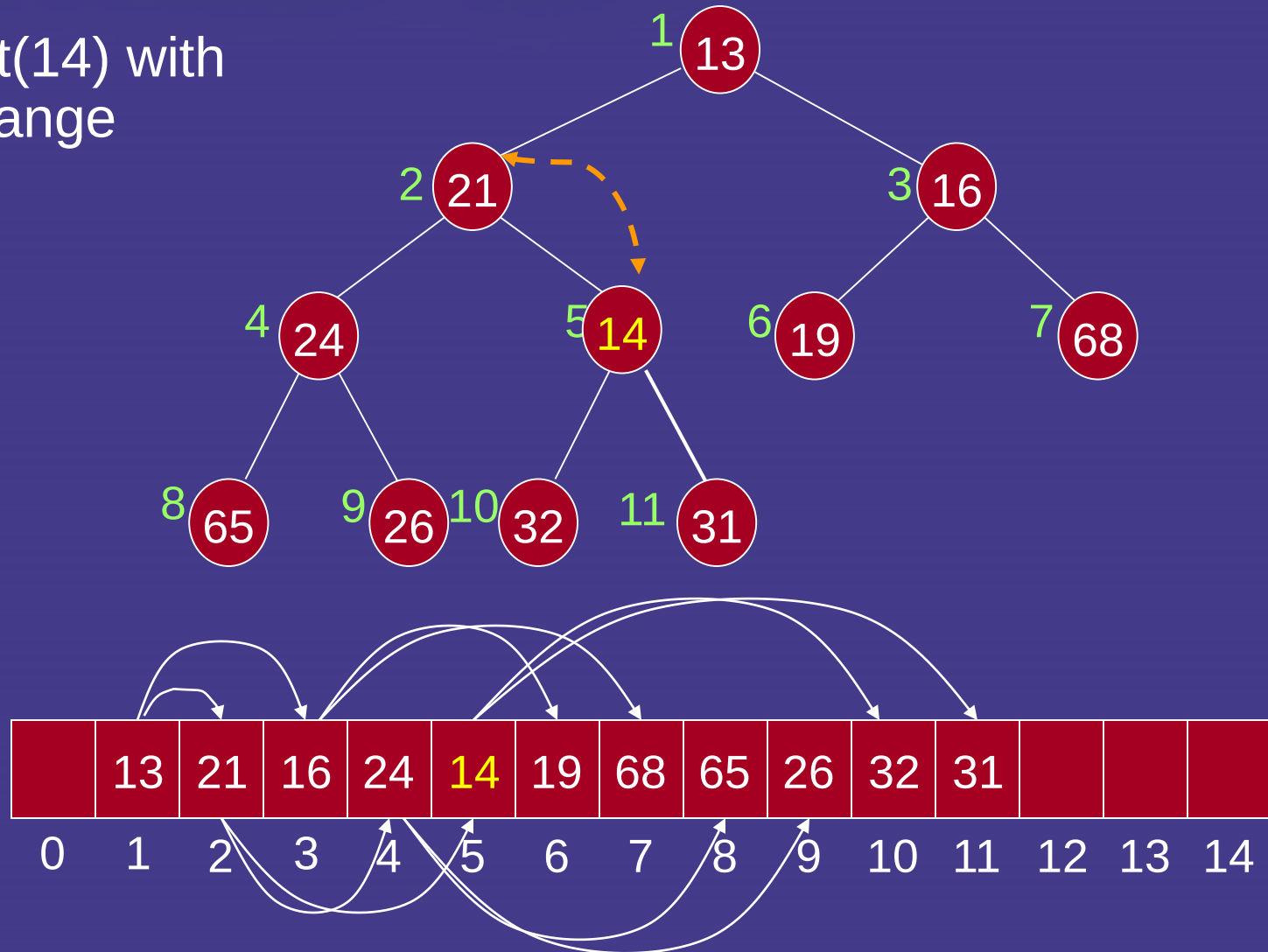
# Inserting into a Heap

insert(14) with  
exchange



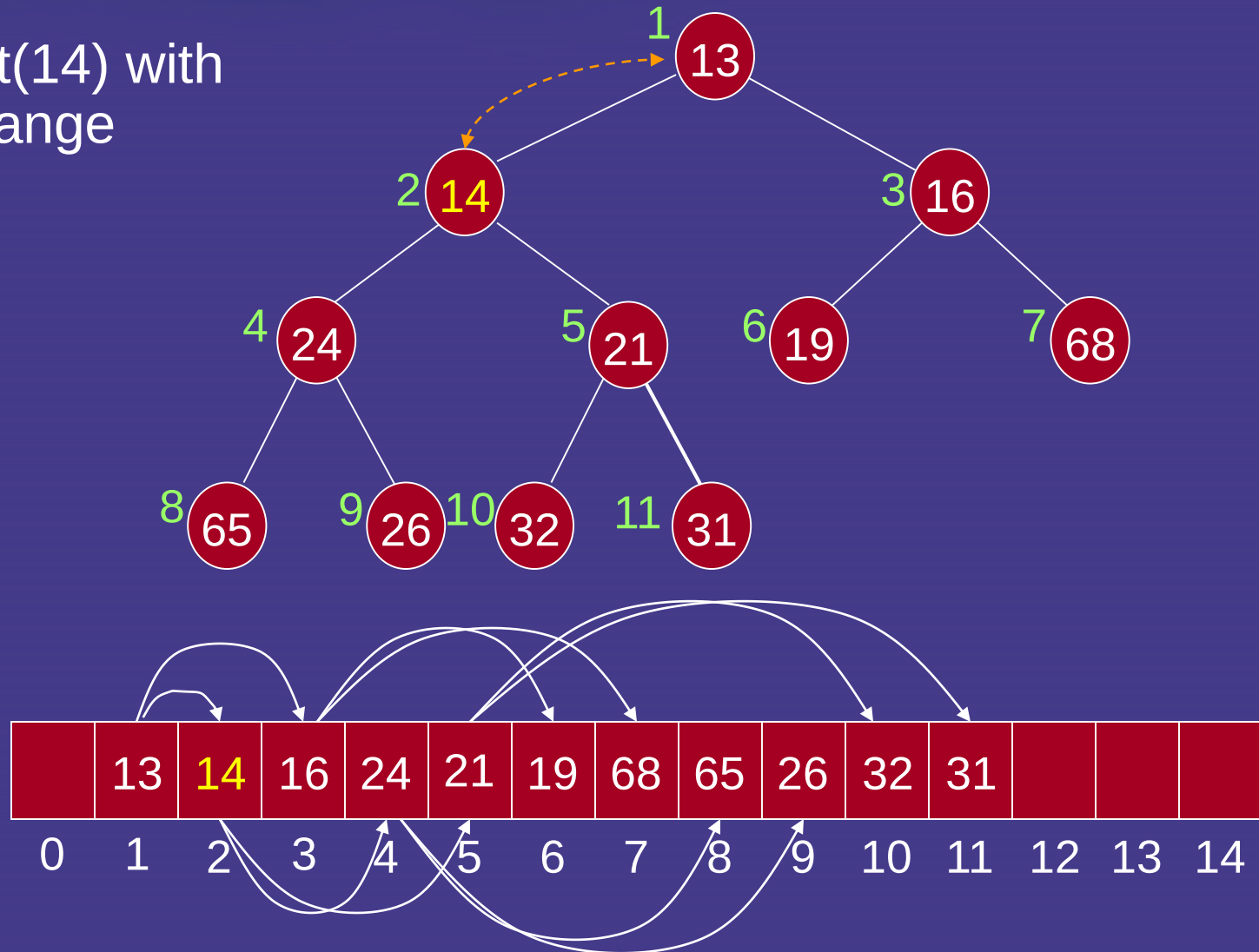
# Inserting into a Heap

insert(14) with  
exchange



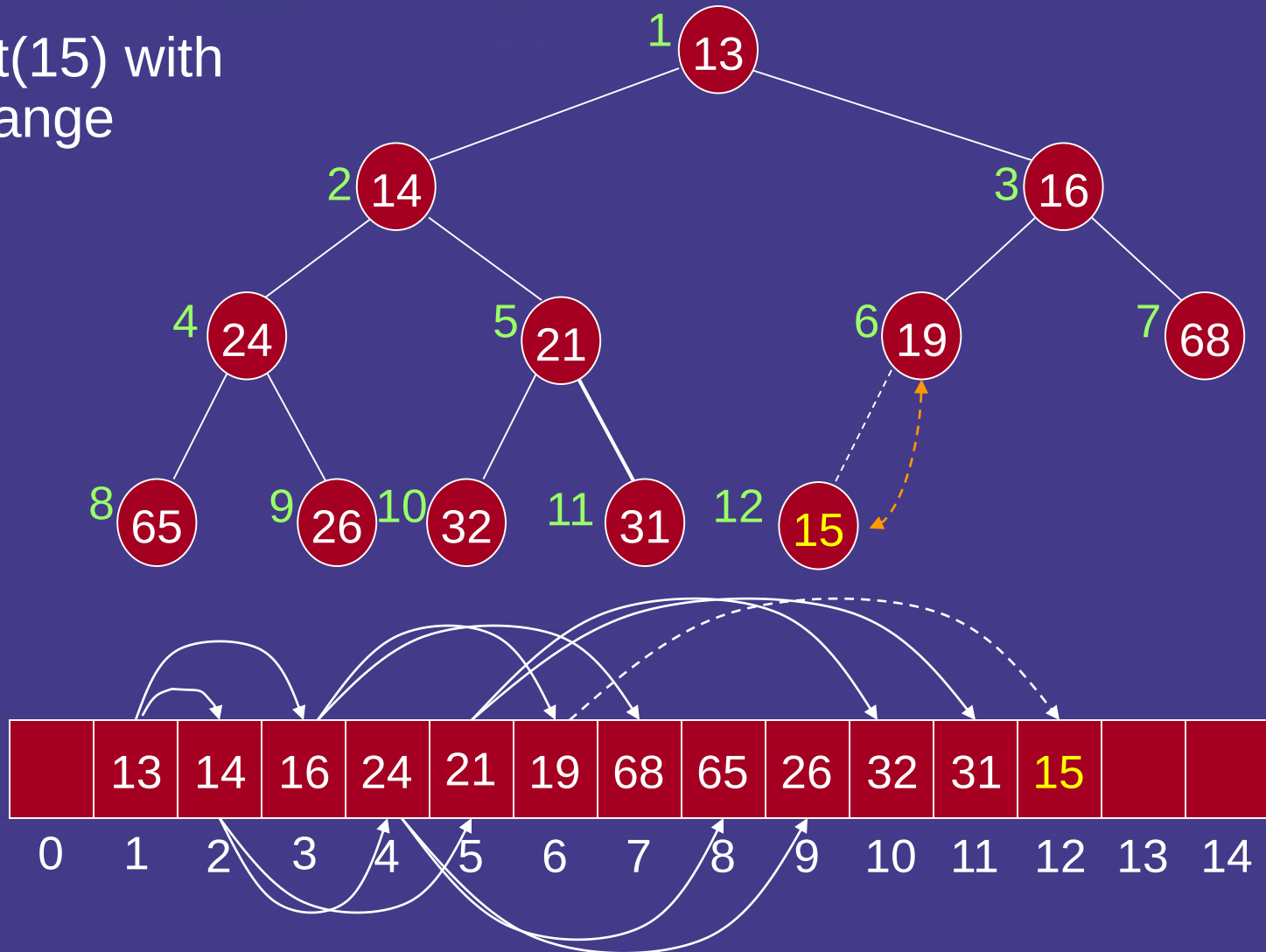
# Inserting into a Heap

insert(14) with  
exchange



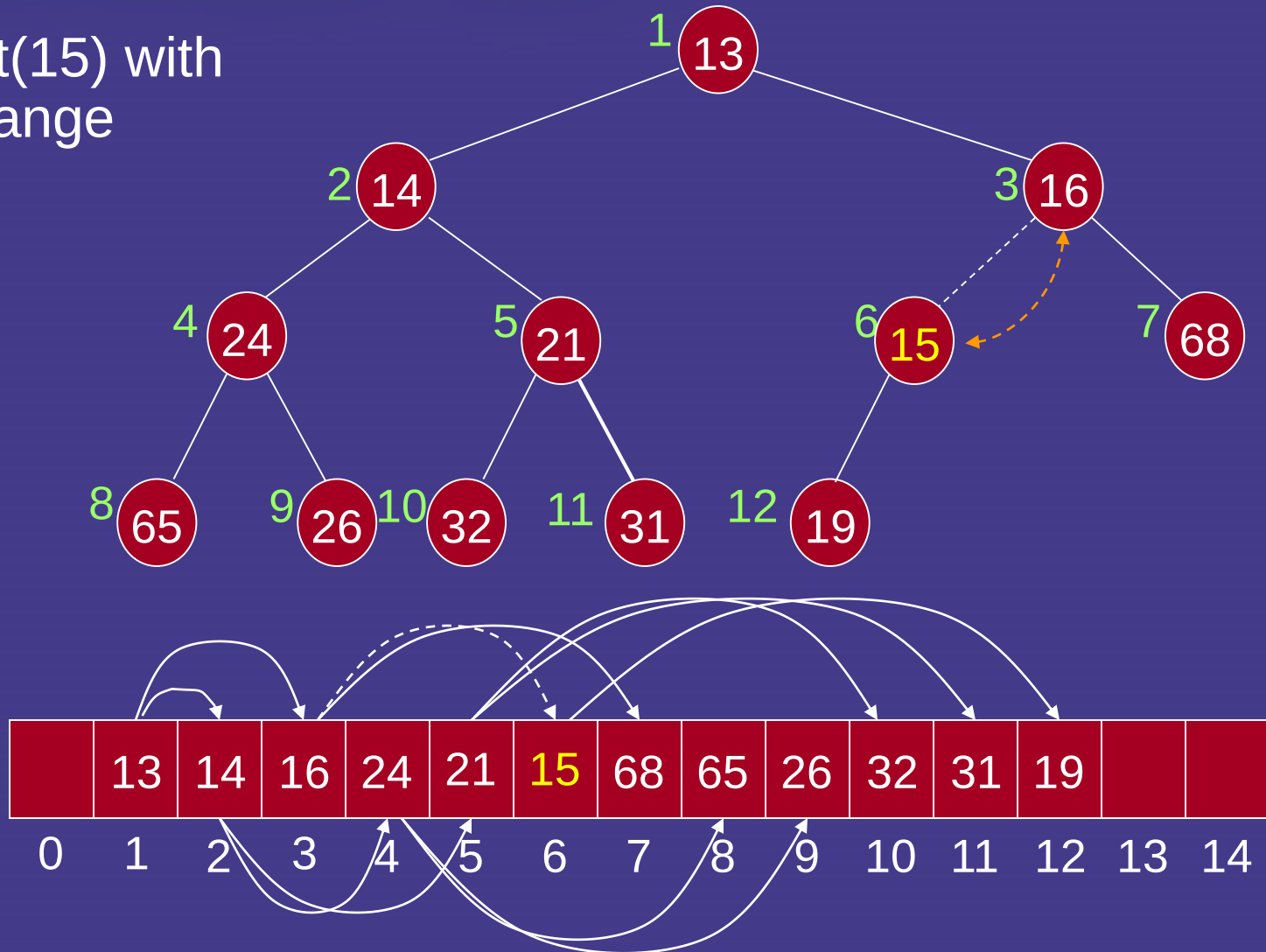
# Inserting into a Heap

insert(15) with  
exchange



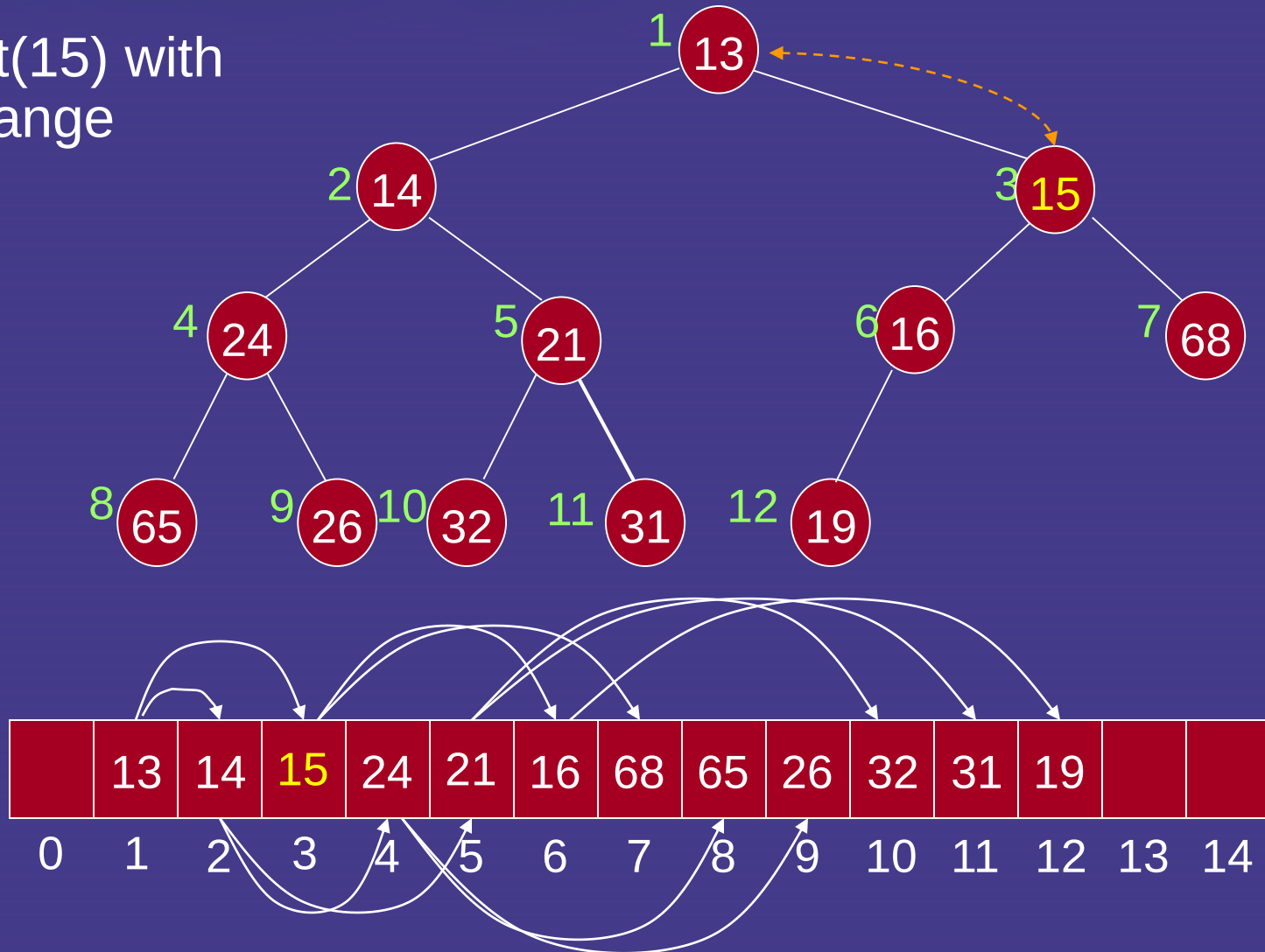
# Inserting into a Heap

insert(15) with  
exchange



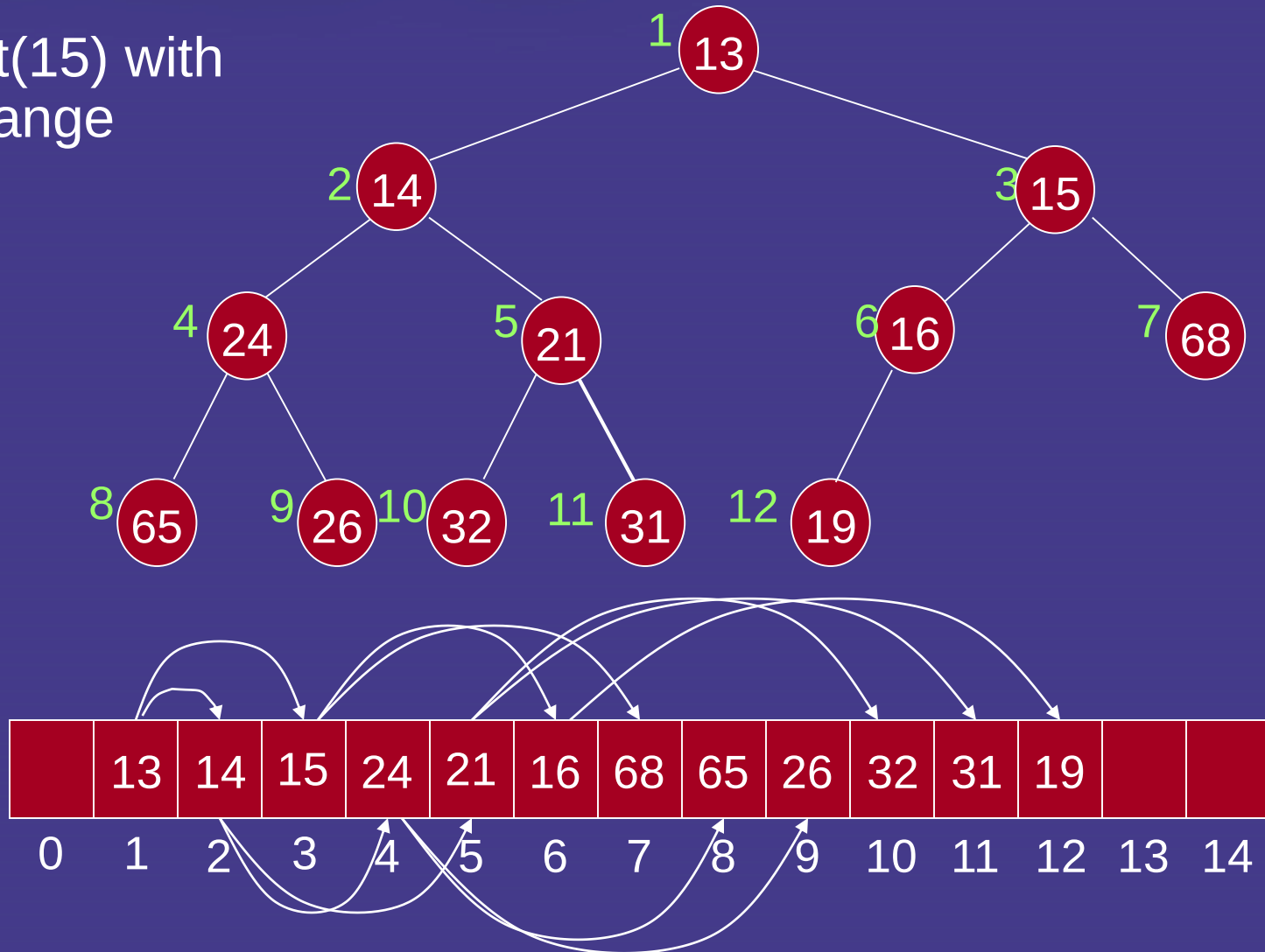
# Inserting into a Heap

insert(15) with  
exchange



# Inserting into a Heap

insert(15) with  
exchange



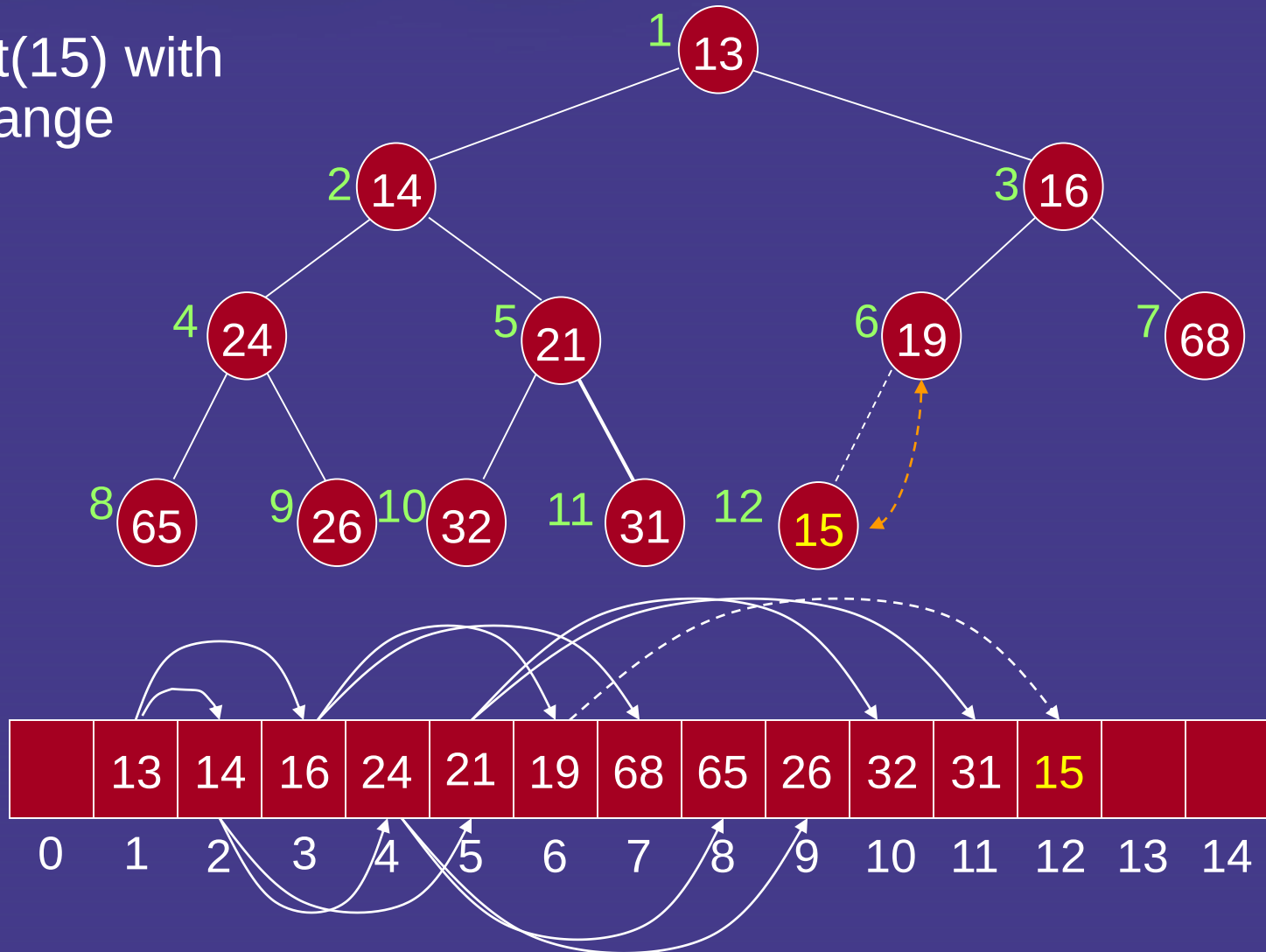


# Data Structures

Lecture 30  
Sohail Aslam

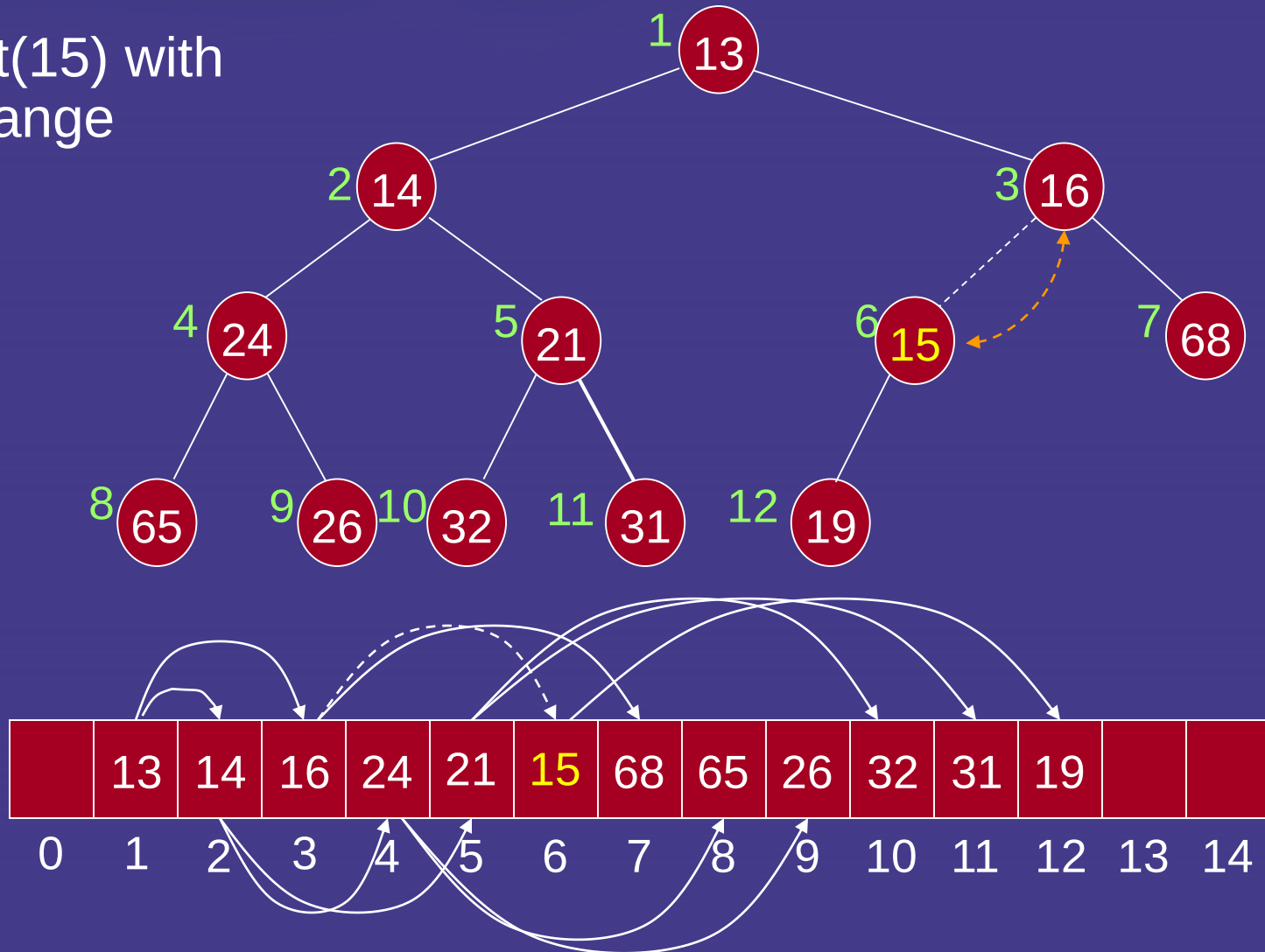
# Inserting into a Heap

insert(15) with  
exchange



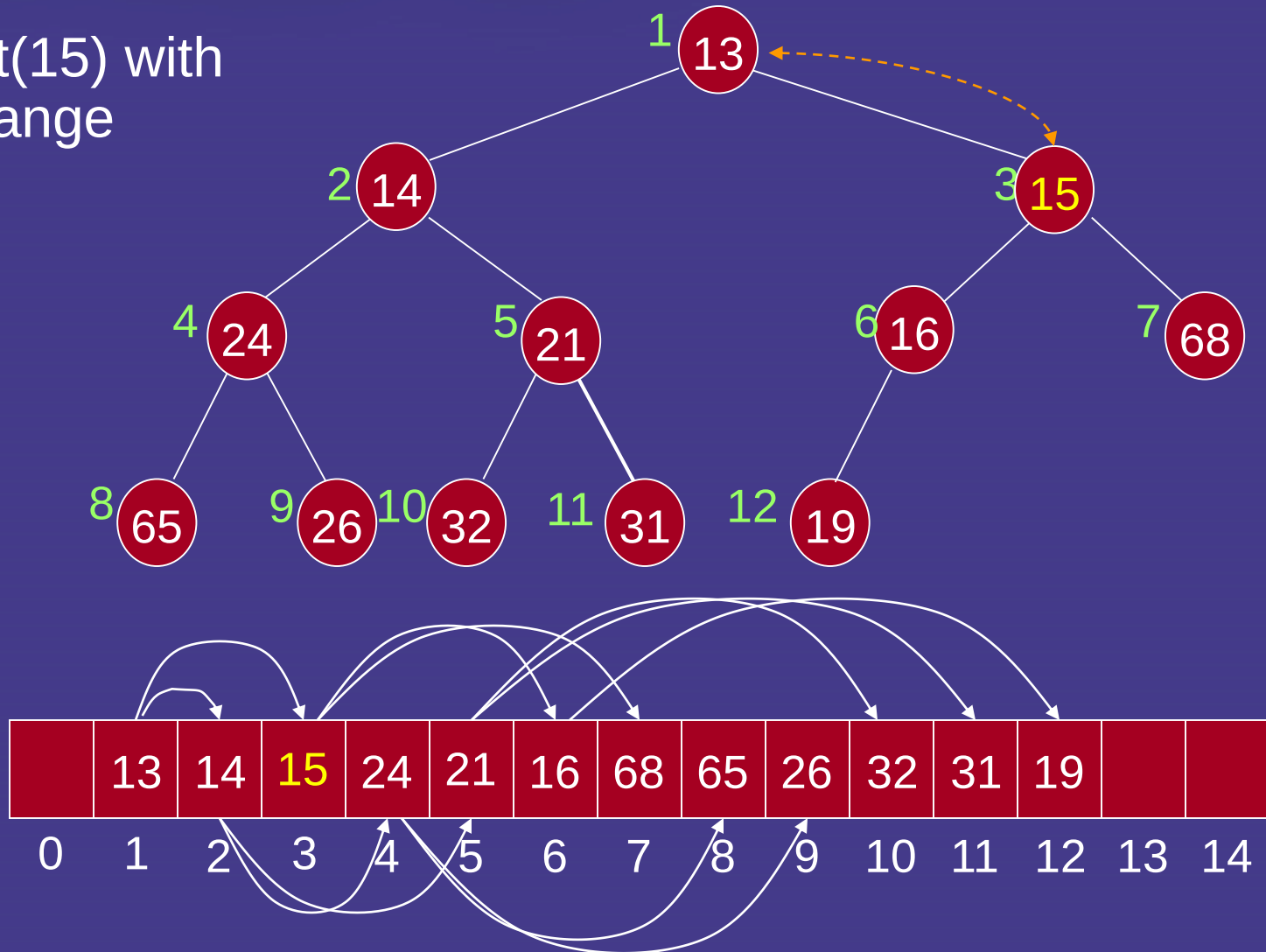
# Inserting into a Heap

insert(15) with  
exchange



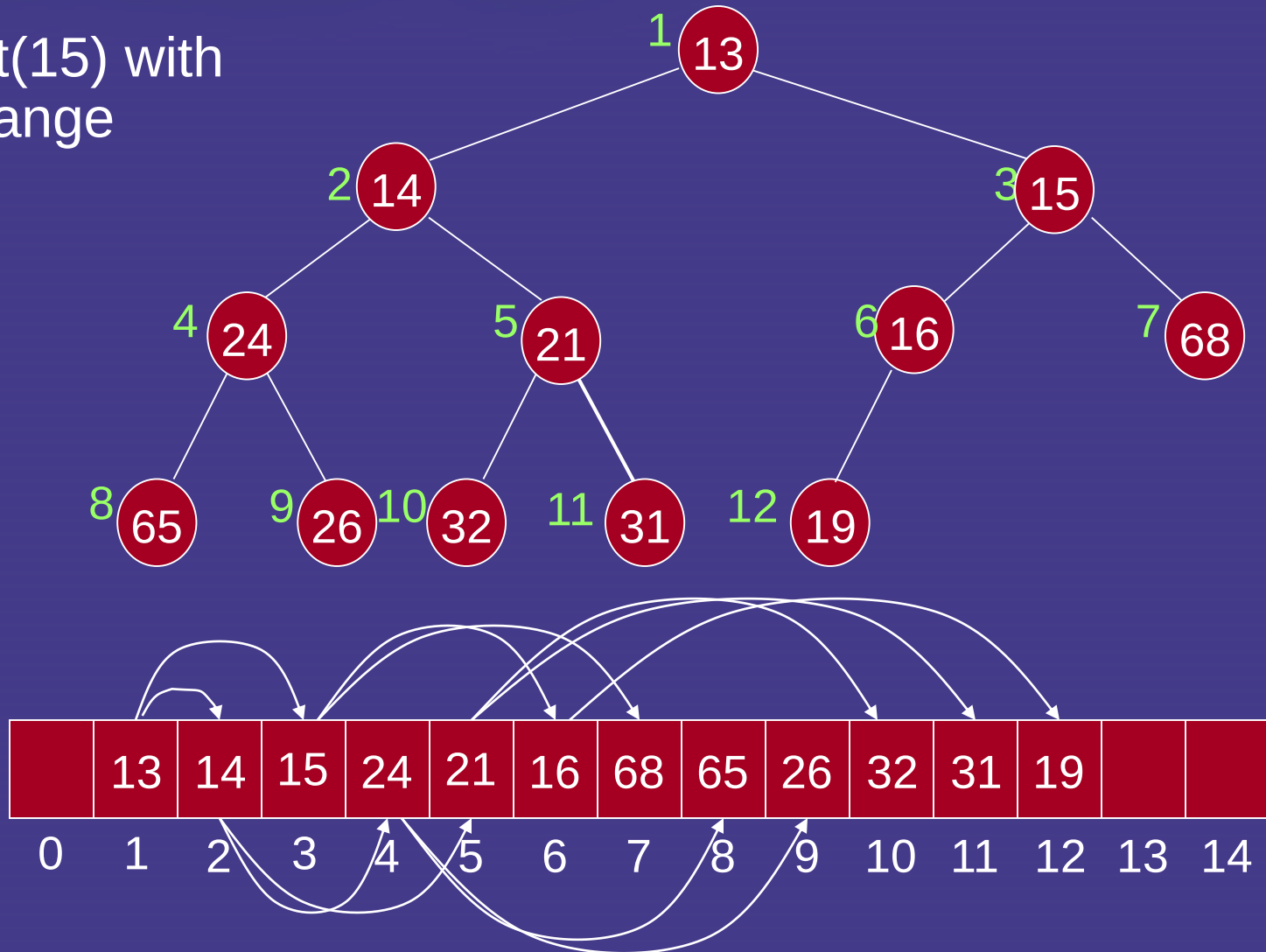
# Inserting into a Heap

insert(15) with  
exchange



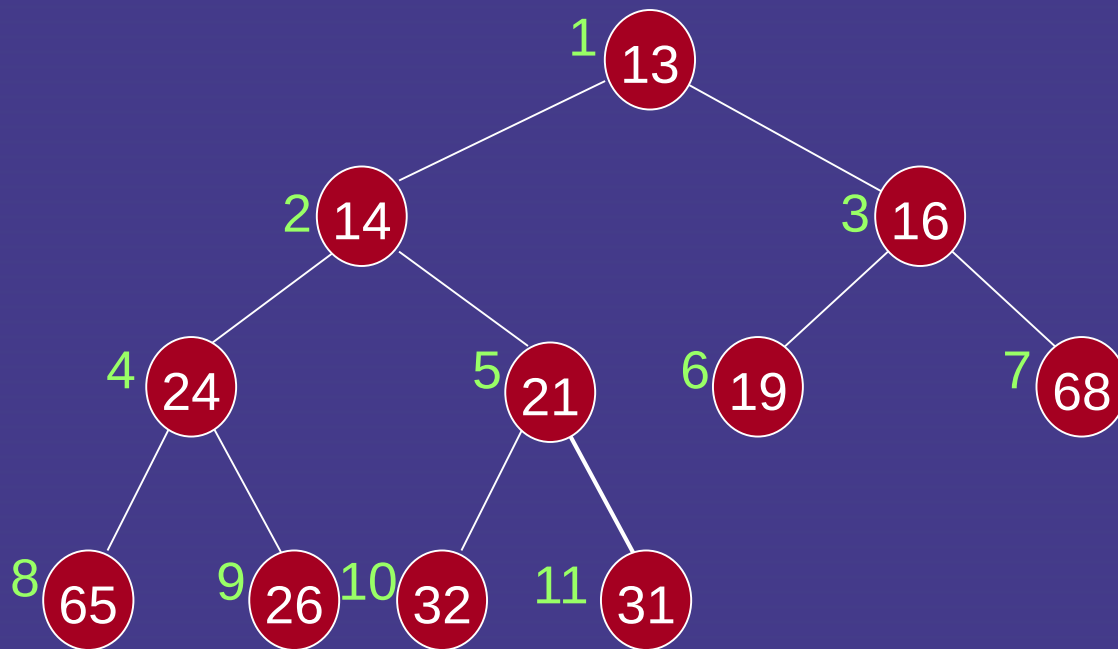
# Inserting into a Heap

insert(15) with  
exchange



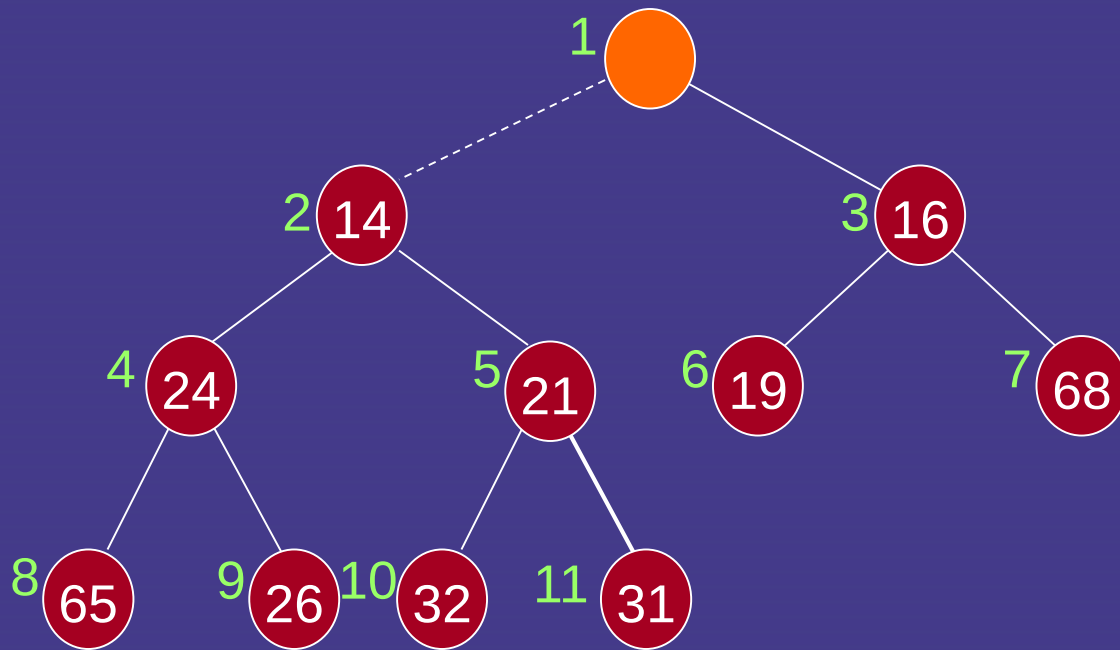
# DeleteMin

- Finding the minimum is easy; it is at the top of the heap.
- Deleting it (or removing it) causes a hole which needs to be filled.



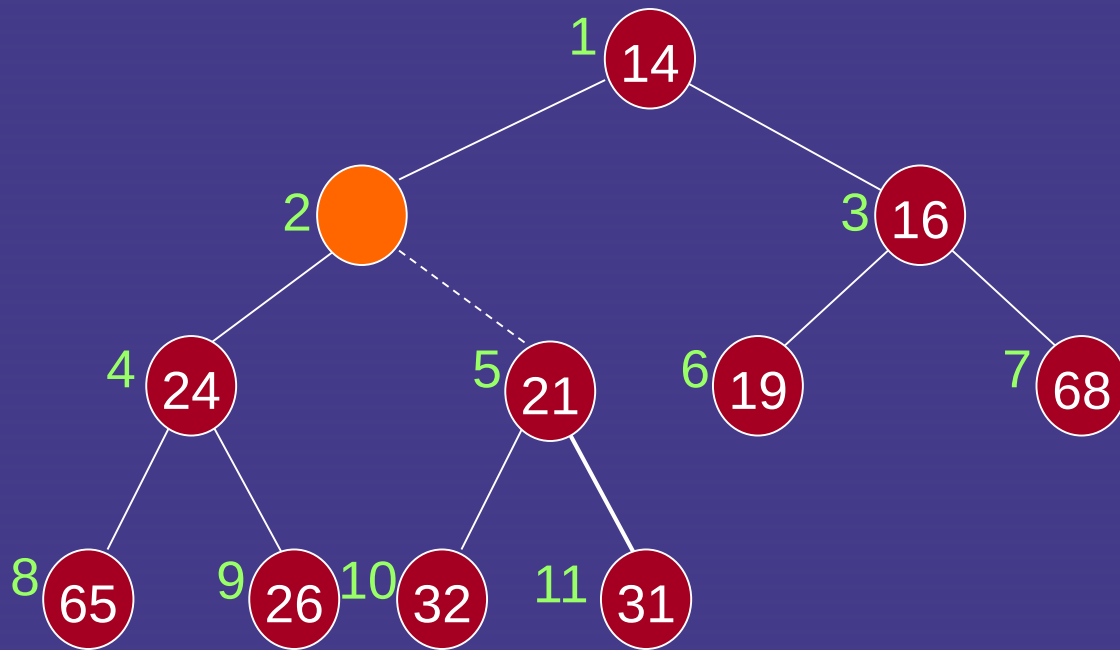
# DeleteMin

deleteMin()



# DeleteMin

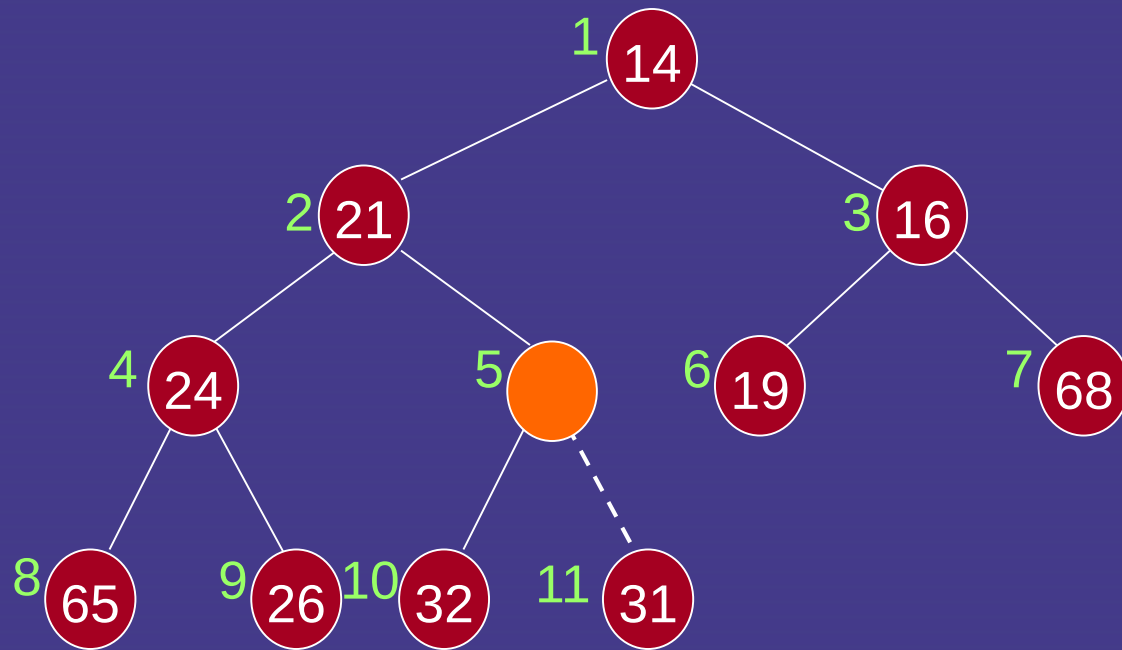
deleteMin()





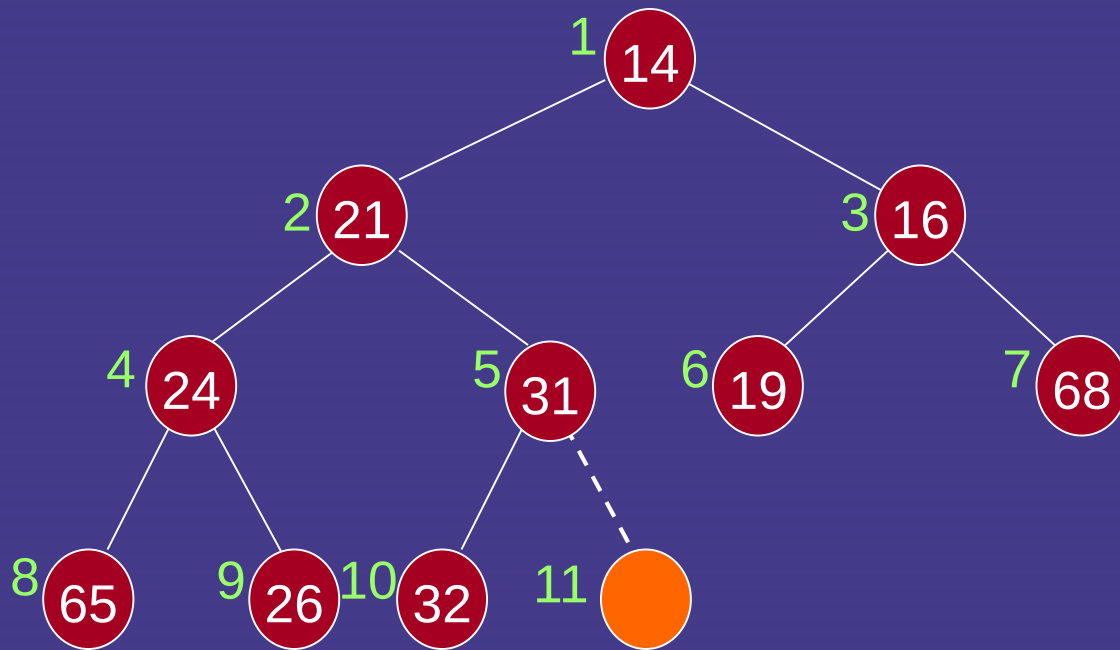
# DeleteMin

deleteMin()



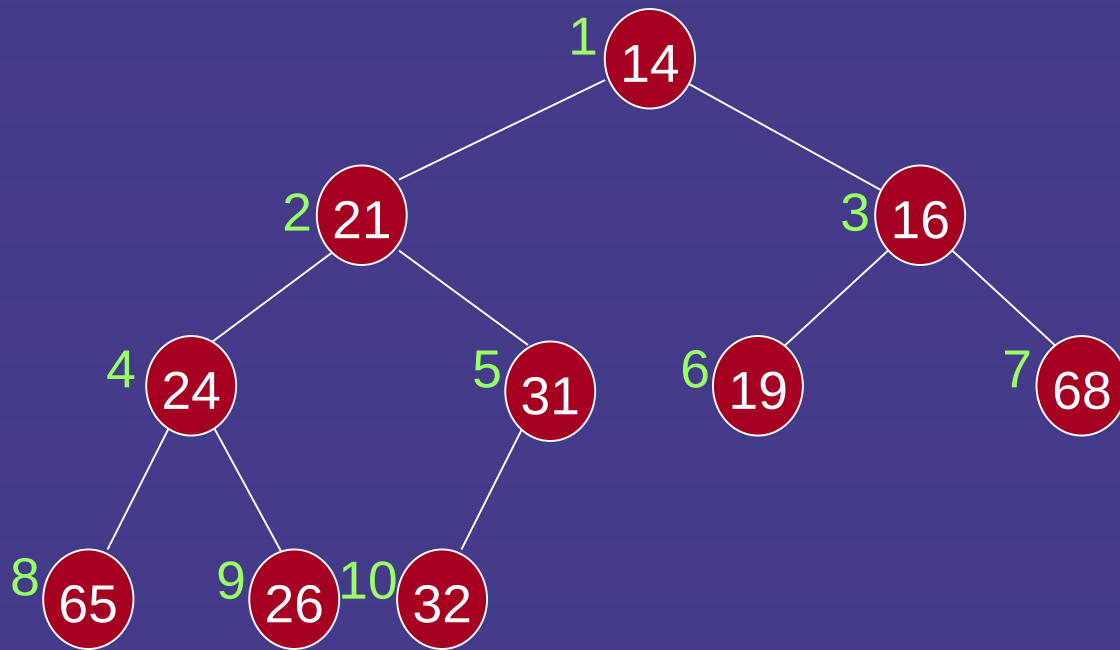
# DeleteMin

deleteMin()



# DeleteMin

deleteMin(): heap size is reduced by 1.



# BuildHeap

- Suppose we are given as input  $N$  keys (or items) and we want 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  $\log_2 N$  (if new key percolates all the way up to the root).

# BuildHeap

- The worst time for building a heap of  $N$  keys could be  $N \log_2 N$ .
- It turns out that we can build a heap in linear time.

# BuildHeap

- Suppose we have a method *percolateDown(p)* which moves down the key in node *p* downwards.
- This is what was happening in *deleteMin*.

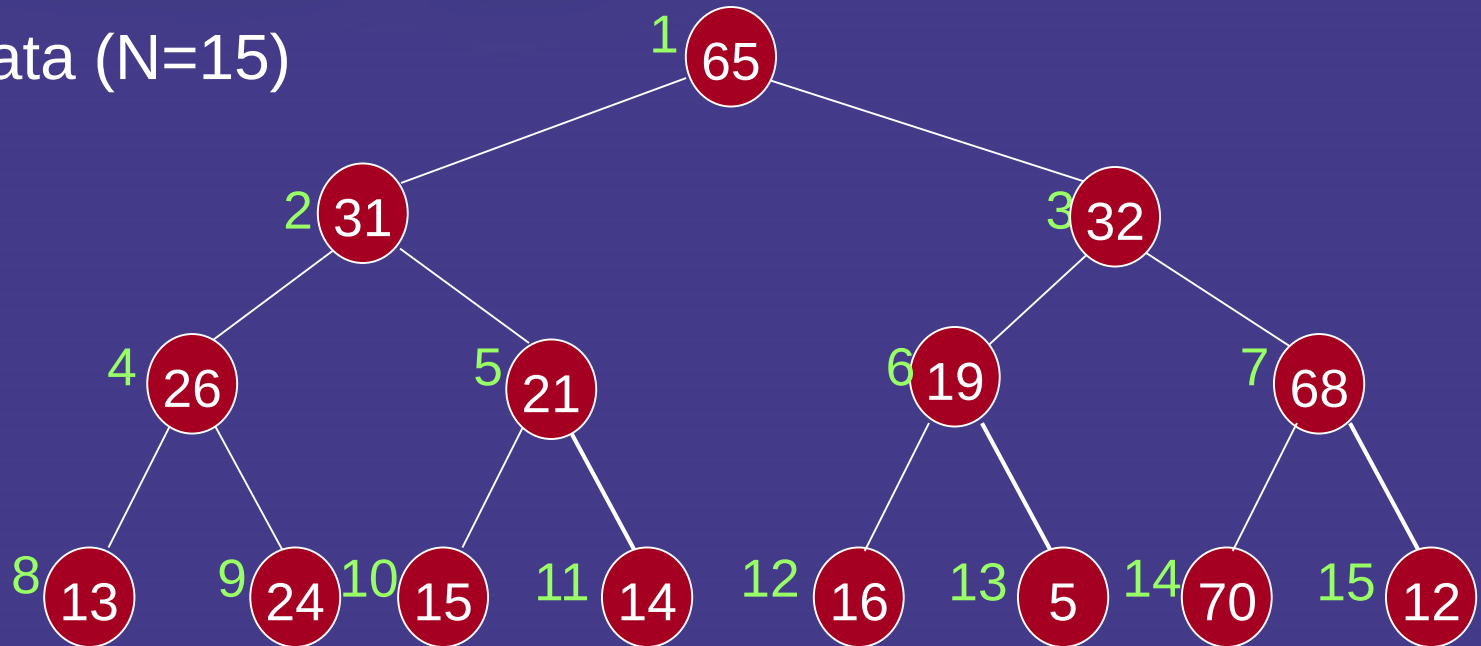
# BuildHeap

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 |

# BuildHeap

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 |



# BuildHeap

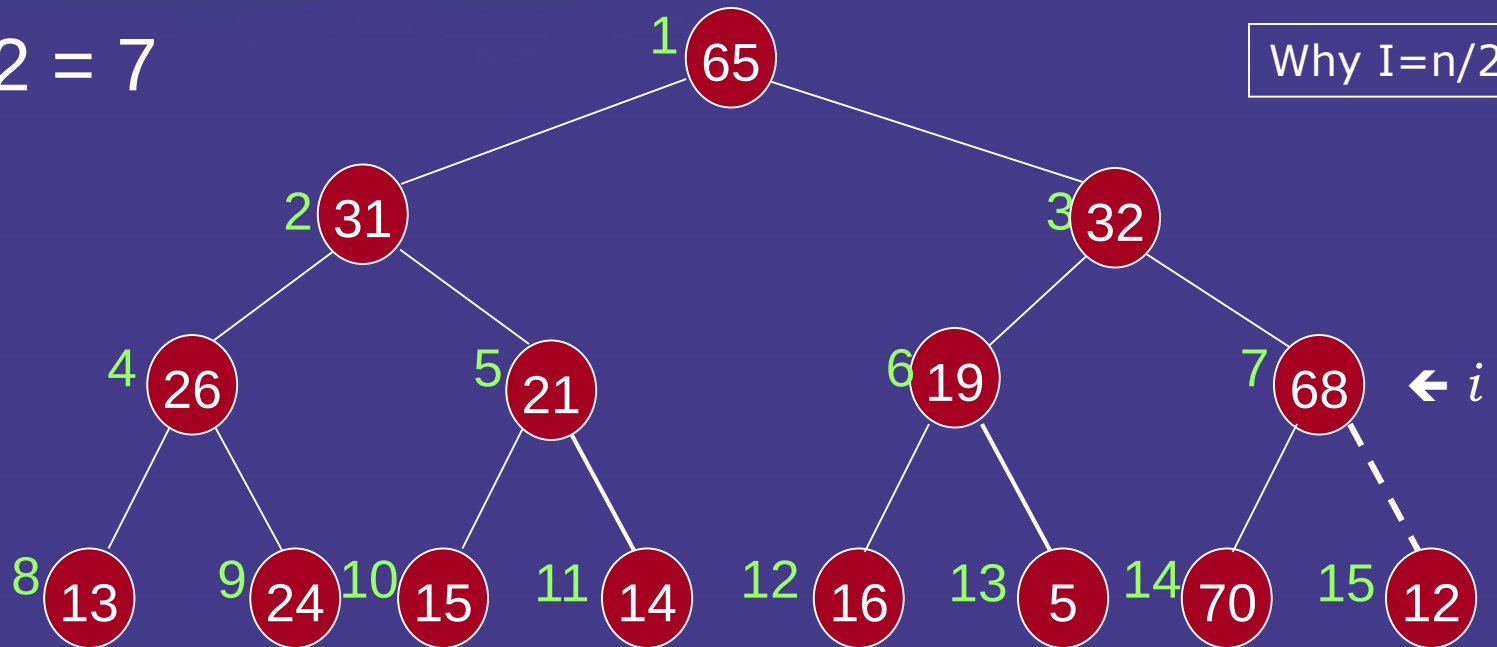
- The general algorithm is to place the  $N$  keys in an array and consider it to be an unordered binary tree.
- The following algorithm will build a heap out of  $N$  keys.

```
for( i = N/2; i > 0; i-- )  
    percolateDown(i) ;
```

# BuildHeap

$$i = 15/2 = 7$$

Why  $I=n/2$ ?

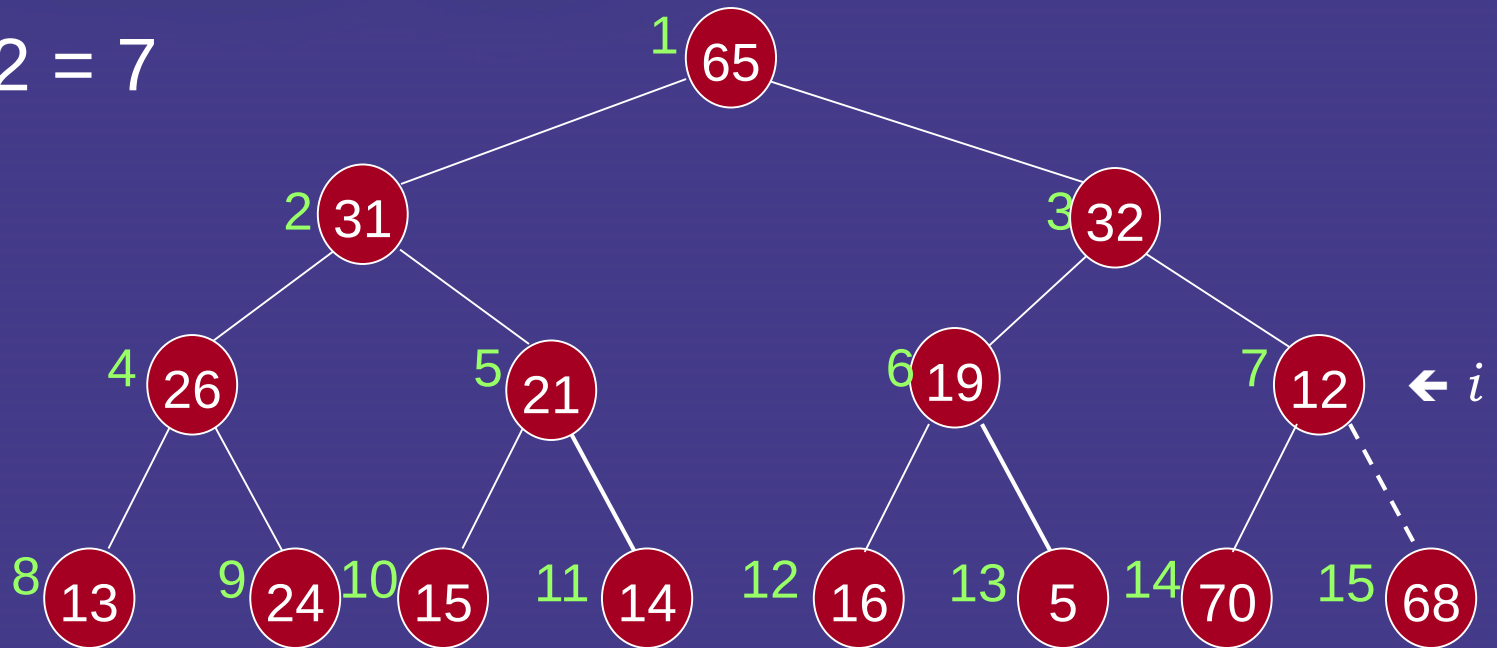


$i$

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 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 |

# BuildHeap

$$i = 15/2 = 7$$

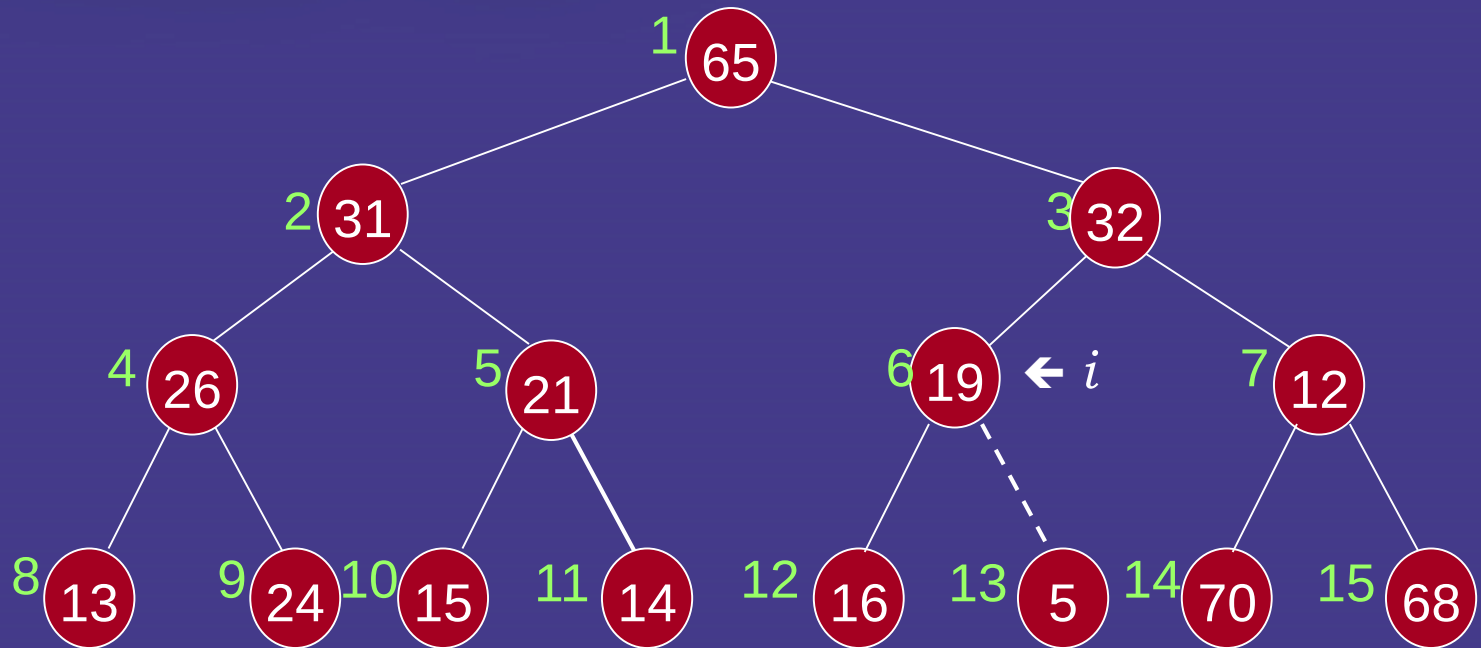


$i$

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 26 | 21 | 19 | 12 | 13 | 24 | 15 | 14 | 16 | 5  | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

$i = 6$

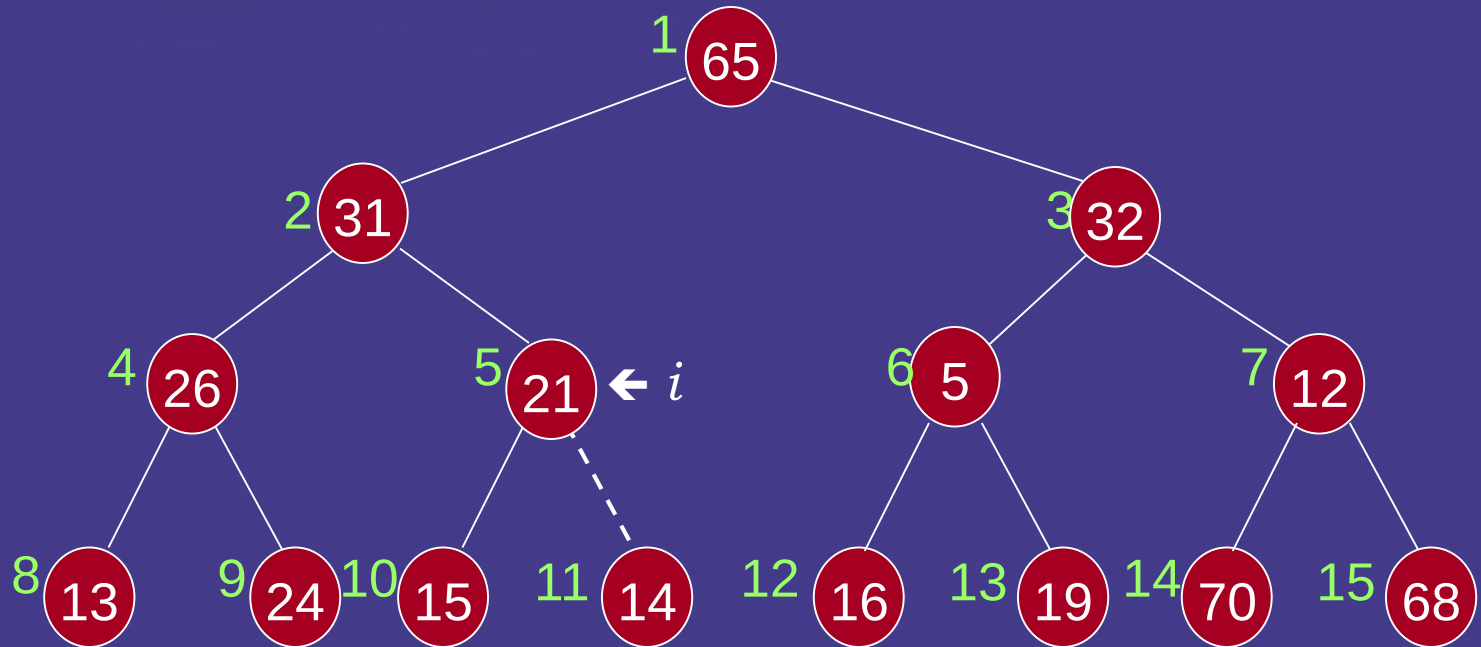


$i$

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 26 | 21 | 19 | 12 | 13 | 24 | 15 | 14 | 16 | 5  | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

$i = 5$



$i$

|   |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 26 | 21 | 5 | 12 | 13 | 24 | 15 | 14 | 16 | 19 | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

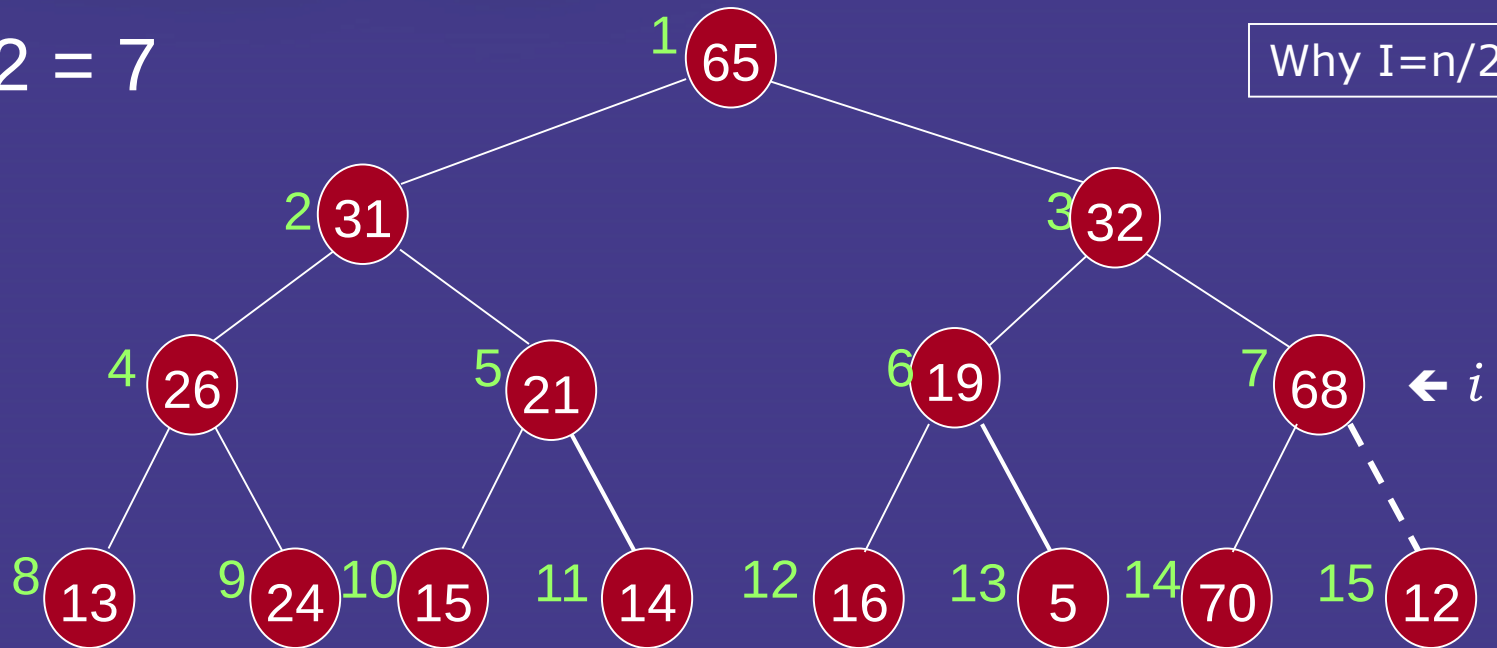
- The general algorithm is to place the  $N$  keys in an array and consider it to be an unordered binary tree.
- The following algorithm will build a heap out of  $N$  keys.

```
for( i = N/2; i > 0; i-- )  
    percolateDown(i) ;
```

# BuildHeap

$$i = 15/2 = 7$$

Why  $I=n/2$ ?

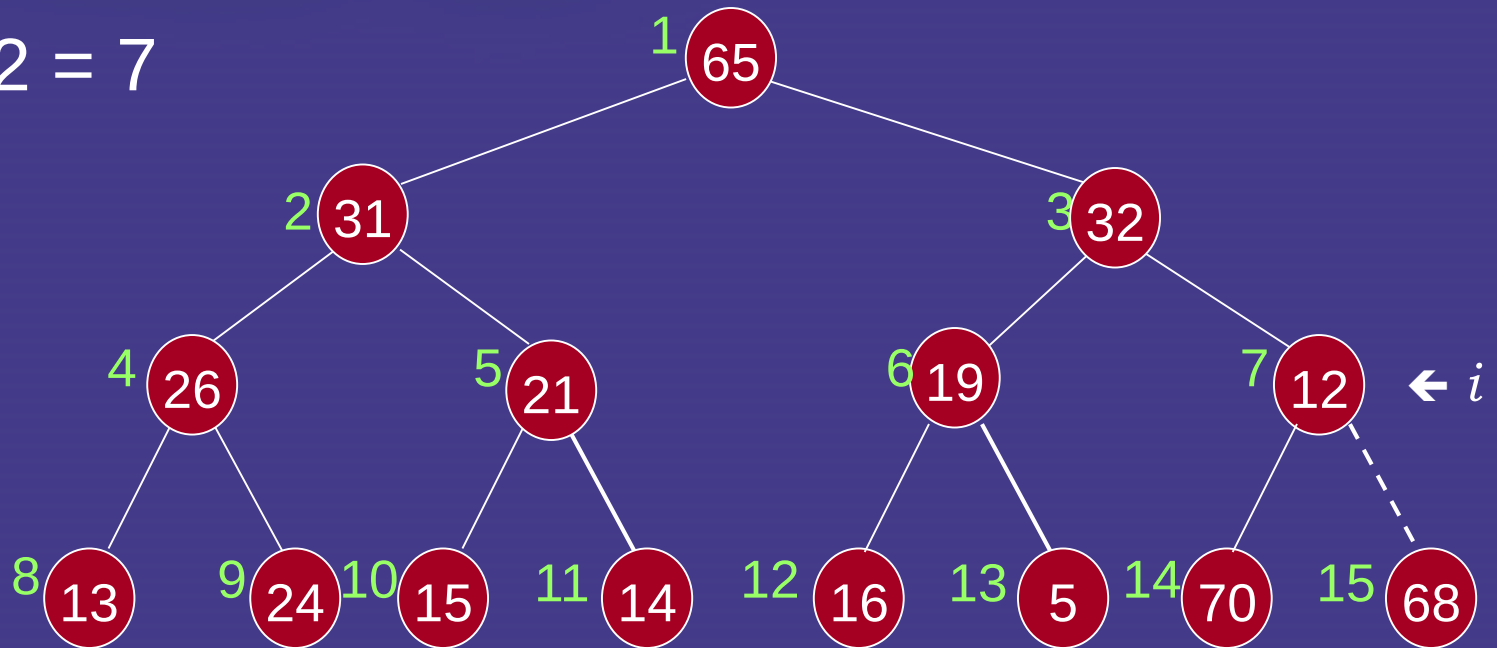


$i$

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 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 |

# BuildHeap

$$i = 15/2 = 7$$



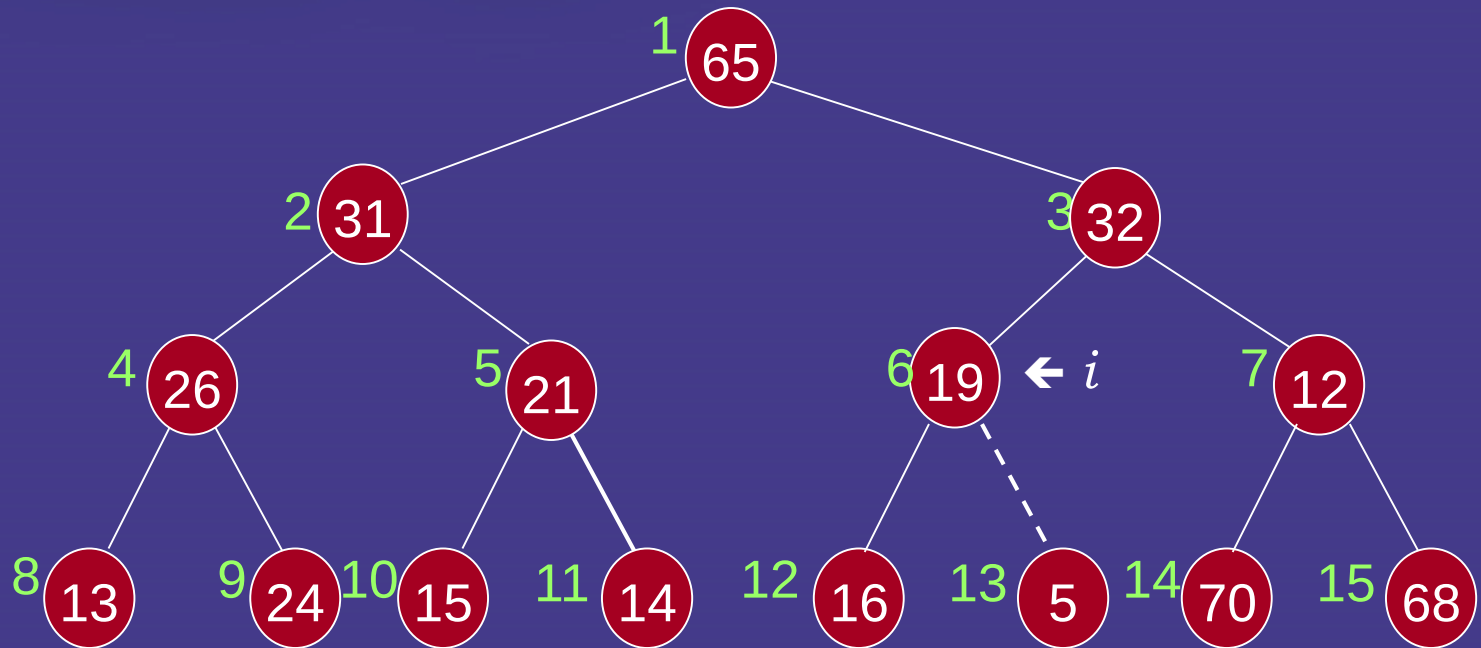
$i$

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 26 | 21 | 19 | 12 | 13 | 24 | 15 | 14 | 16 | 5  | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |



# BuildHeap

$i = 6$

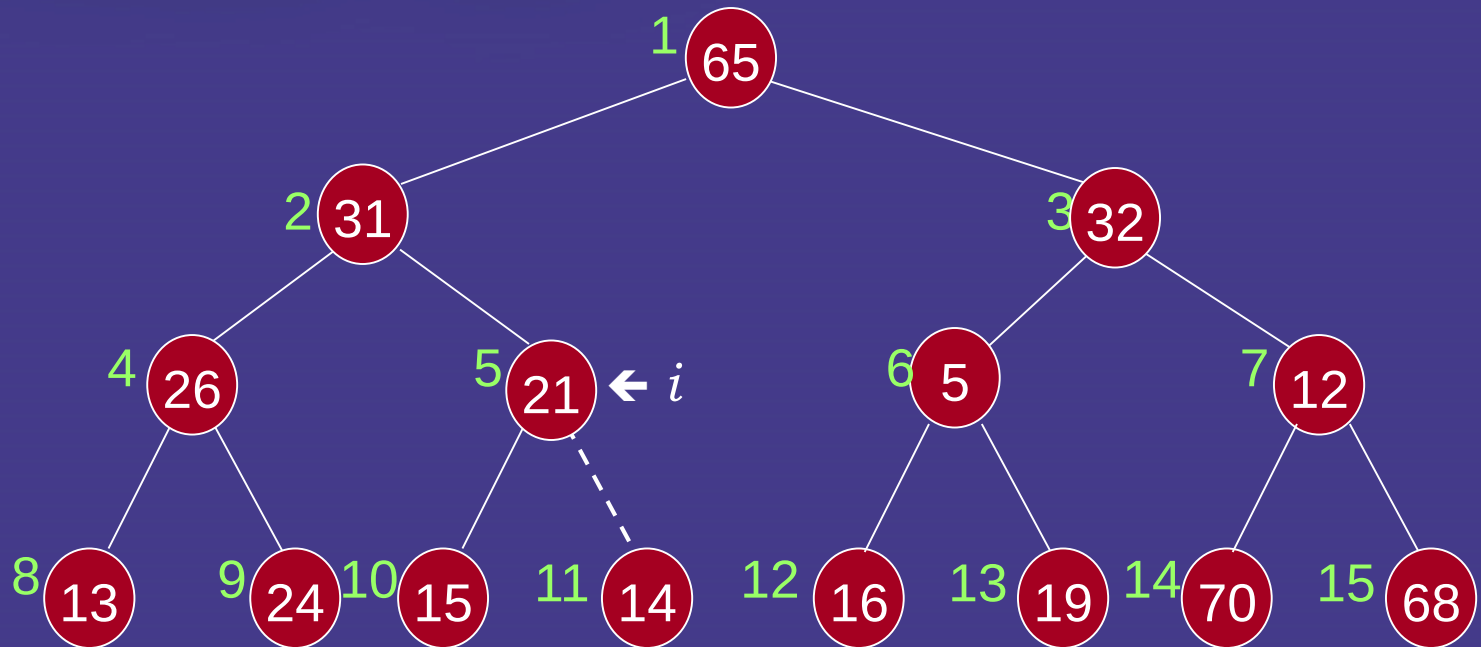


$i$

|   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 26 | 21 | 19 | 12 | 13 | 24 | 15 | 14 | 16 | 5  | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

$i = 5$

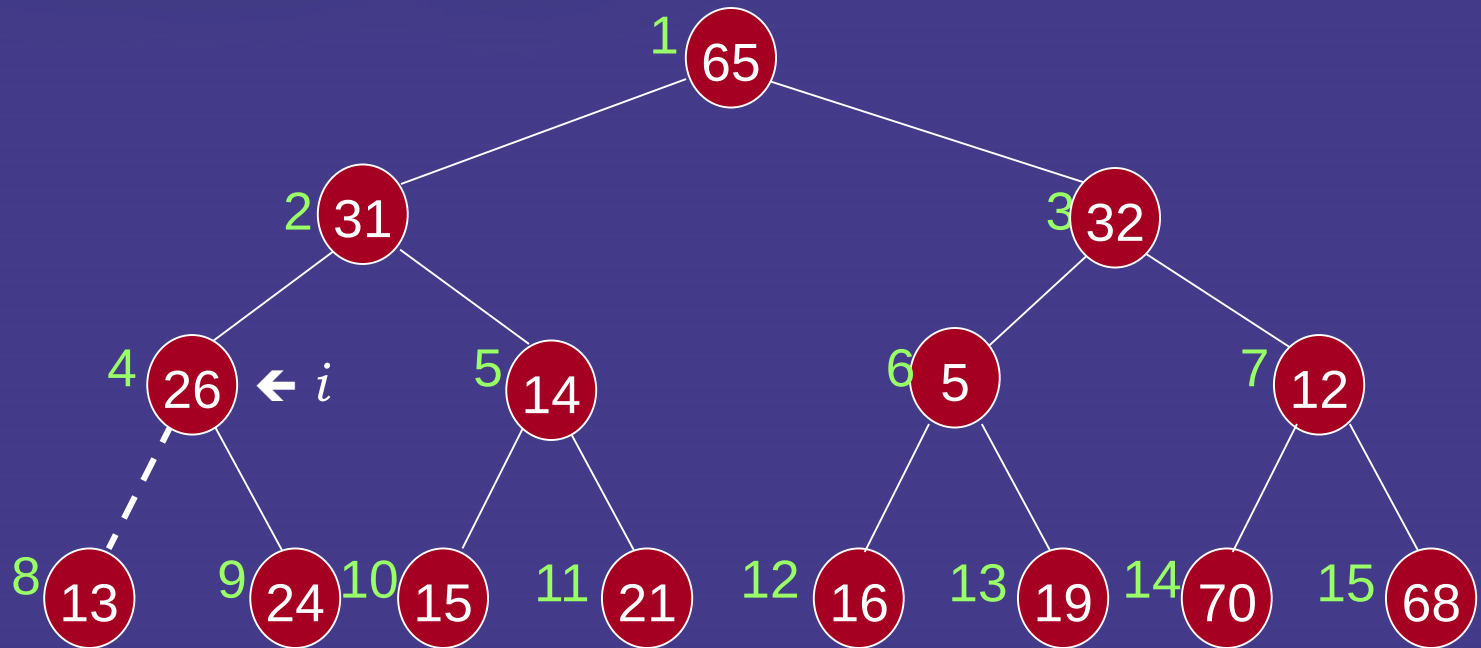


$i$

|   |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 26 | 21 | 5 | 12 | 13 | 24 | 15 | 14 | 16 | 19 | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

$i = 4$

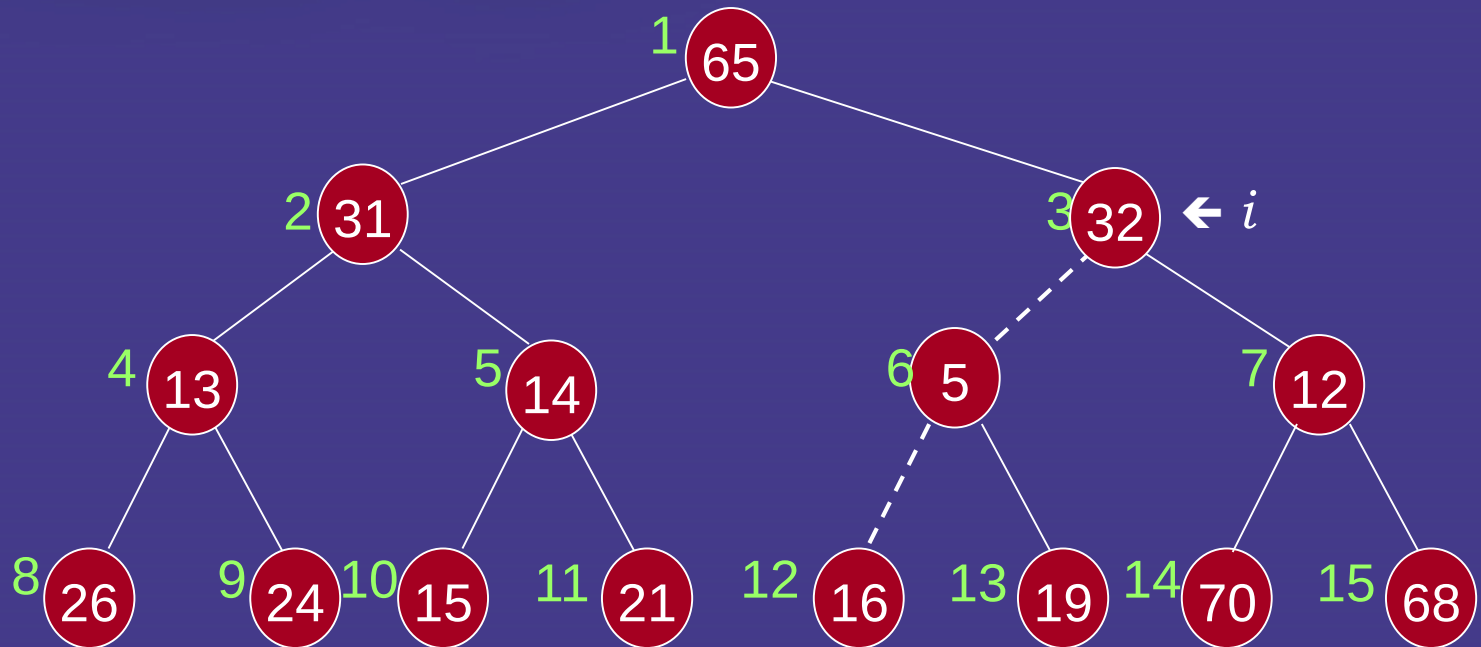


$i$

|   |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 26 | 14 | 5 | 12 | 13 | 24 | 15 | 21 | 16 | 19 | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

$i = 3$

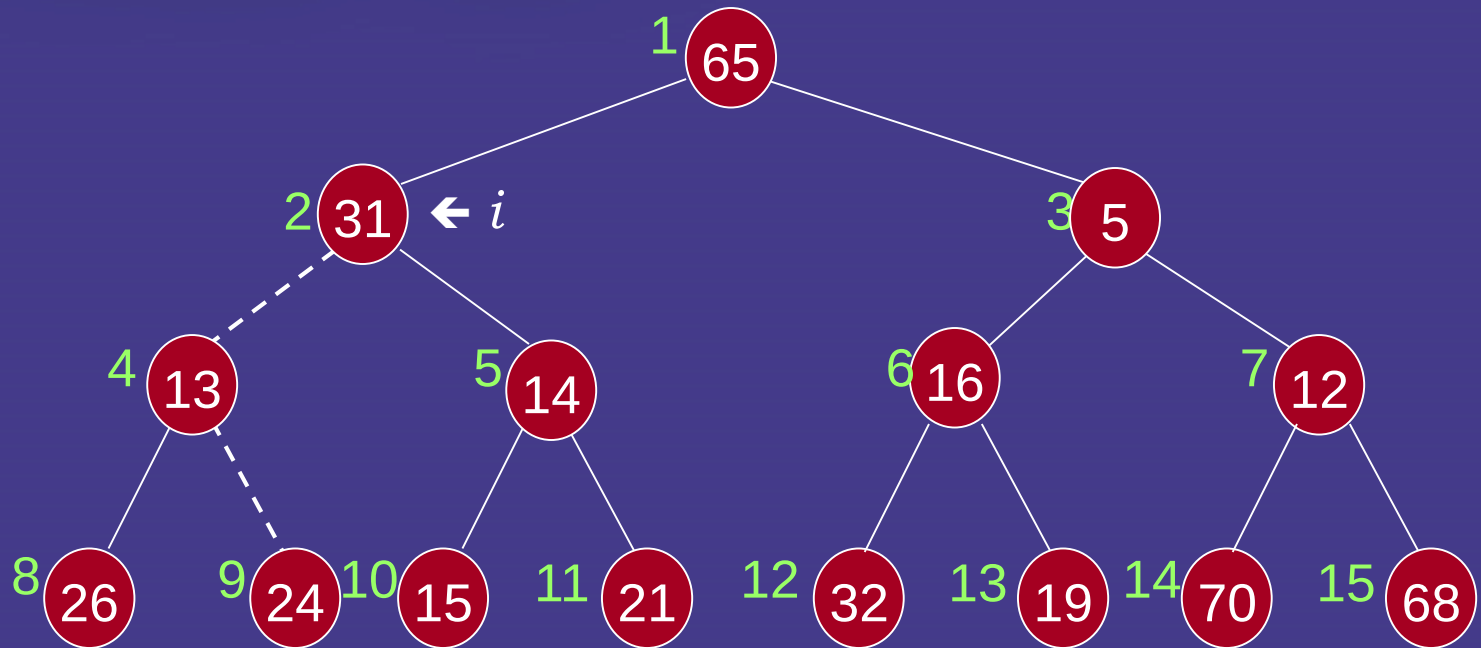


$i$

|   |    |    |    |    |    |   |    |    |    |    |    |    |    |    |    |
|---|----|----|----|----|----|---|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 32 | 13 | 14 | 5 | 12 | 26 | 24 | 15 | 21 | 16 | 19 | 70 | 68 |
| 0 | 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

$i = 2$

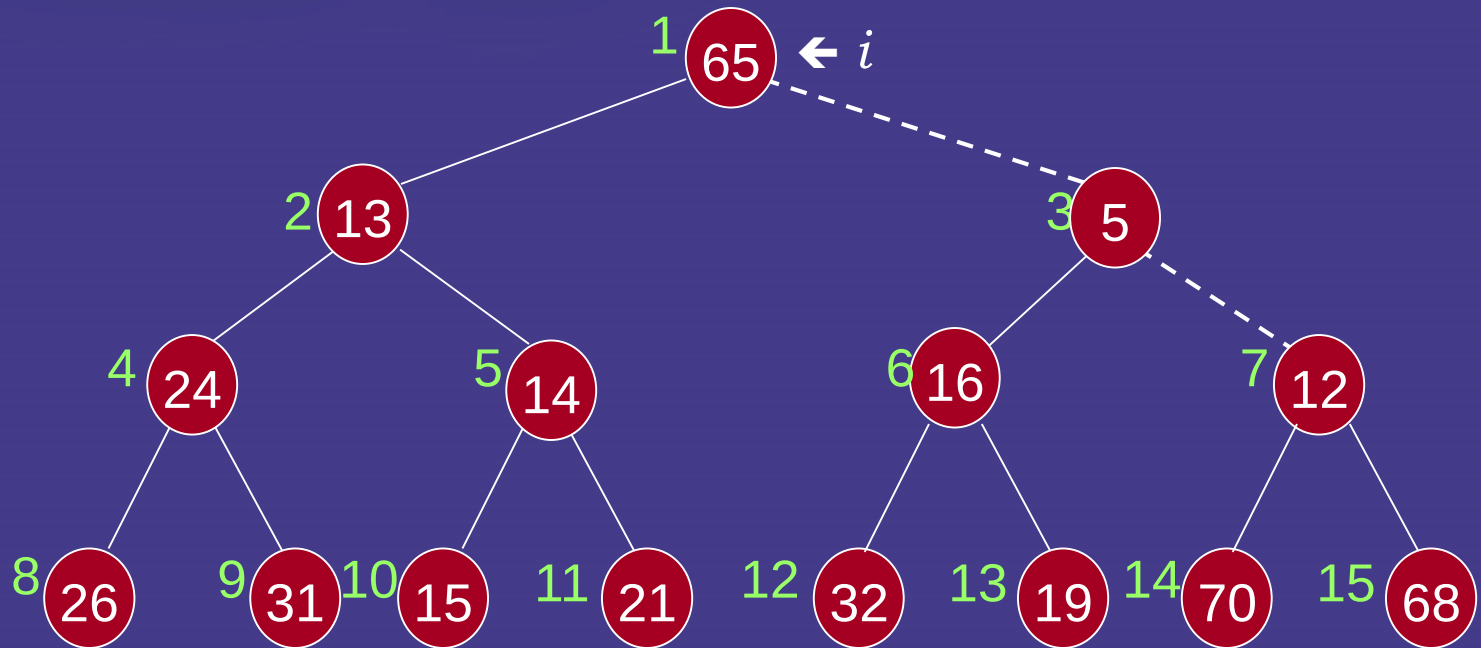


$i$

|   |    |    |   |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 65 | 31 | 5 | 13 | 14 | 16 | 12 | 26 | 24 | 15 | 21 | 32 | 19 | 70 | 68 |
| 0 | 1  | 2  | 3 | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

$i = 1$

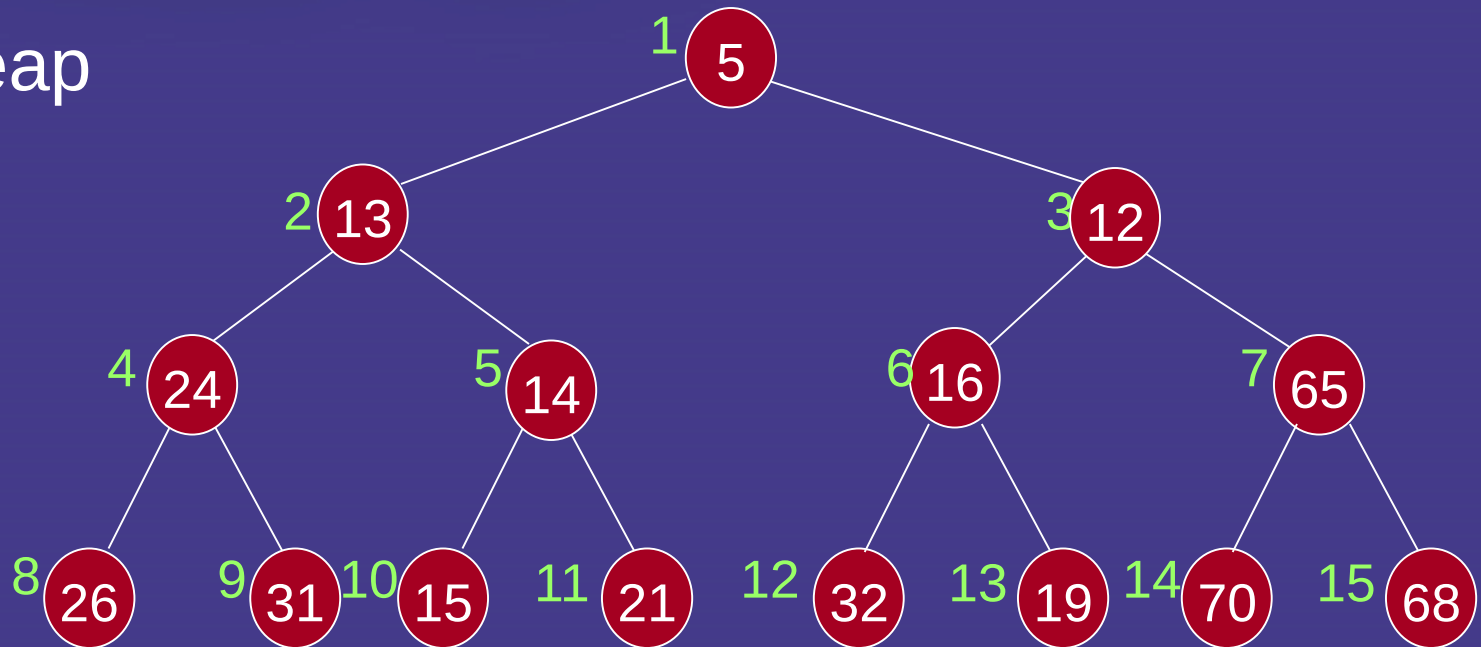


$i$

|   |    |    |   |    |    |    |    |    |    |    |    |    |    |    |    |
|---|----|----|---|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 65 | 13 | 5 | 24 | 14 | 16 | 12 | 26 | 31 | 15 | 21 | 32 | 19 | 70 | 68 |
| 0 | 1  | 2  | 3 | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# BuildHeap

Min heap



|   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
|   | 5 | 13 | 12 | 24 | 14 | 16 | 65 | 26 | 31 | 15 | 21 | 32 | 19 | 70 | 68 |
| 0 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |

# Other Heap Operations

- *decreaseKey(p, delta)*:  
lowers the value of the key at position 'p' by the amount 'delta'. Since this might violate the heap order, the heap must be reorganized with percolate up (in min heap) or down (in max heap).
- *increaseKey(p, delta)*:  
opposite of decreaseKey.
- *remove(p)*:  
removes the node at position p from the heap. This is done by first decreaseKey(p,  $\infty$ ) and then performing deleteMin().



# Heap code in C++

```
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( );
};
```

# Heap code in C++

```
private:
    int    currentSize; // Number of elements in heap
    eType* array;       // The heap array
    int    capacity;

    void percolateDown( int hole );
};
```

# Heap code in C++

```
#include "Heap.h"

template <class eType>
Heap<eType>::Heap( int capacity )
{
    array = new eType[capacity + 1];
    currentSize=0;
}
```

# Heap code in C++

```
// 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;
}
```

# Heap code in C++

```
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 );
}
```

# Heap code in C++

```
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 );
}
```

Lecture No.32

Data Structure

Dr. Sohail Aslam

# Heap code in C++

*// 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;
```

```
}
```



# Heap code in C++

```
template <class eType>
const eType& Heap<eType>::getMin( )
{
    if( !isEmpty( ) )
        return array[ 1 ];
}
```

```
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 );
}
```

# Heap code in C++

```
template <class eType>
bool Heap<eType>::isEmpty( )
{
    return currentSize == 0;
}
template <class eType>
bool Heap<eType>::isFull( )
{
    return currentSize == capacity;
}
template <class eType>
int Heap<eType>::getSize( )
{
    return currentSize;
}
```

# BuildHeap in Linear Time

- How is buildHeap a linear time algorithm? I.e., better than  $N \log_2 N$ ?
- We need to show that the sum of heights is a linear function of  $N$  (number of nodes).

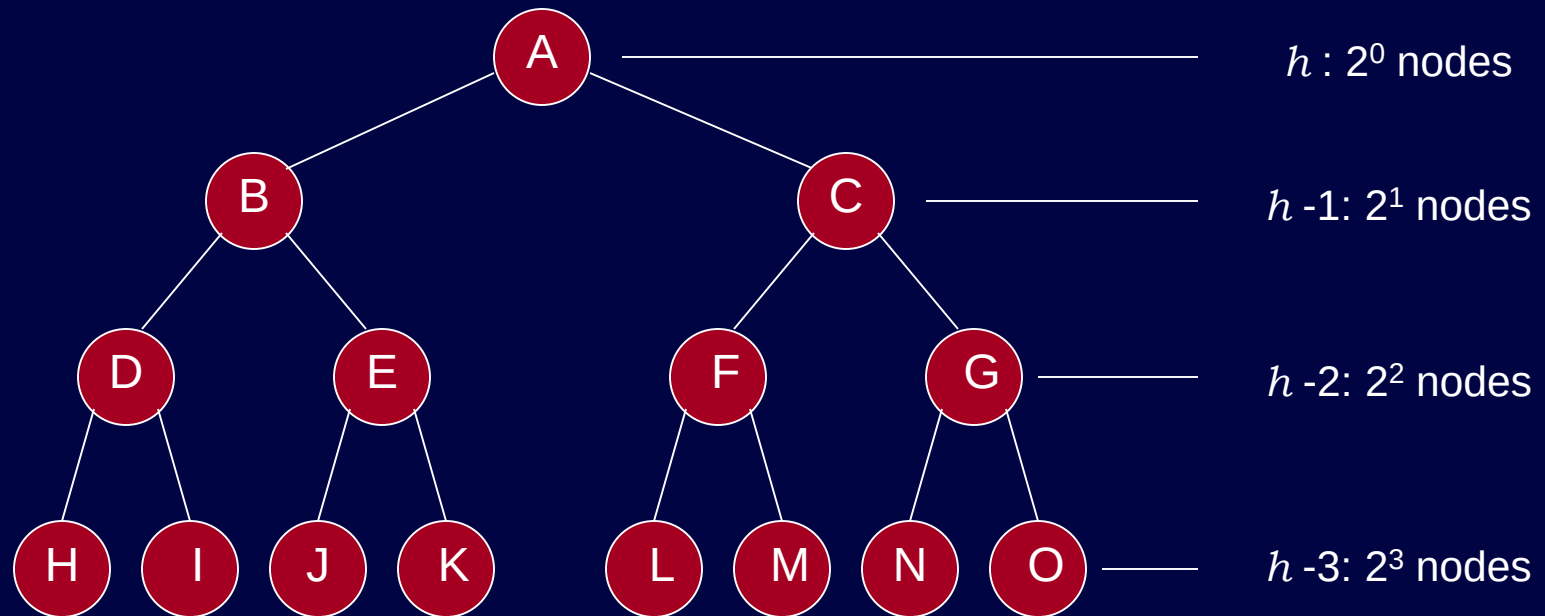
*Theorem:*

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  $N - h - 1$ .

# BuildHeap in Linear Time

It is easy to see that this tree consists of  $(2^0)$  node at height  $h$ ,  $2^1$  nodes at height  $h-1$ ,  $2^2$  at  $h-2$  and, in general,  $2^i$  nodes at  $h-i$ .

# Complete Binary Tree



# BuildHeap in Linear Time

The sum of the heights of all the nodes is then

$$\begin{aligned} S &= \sum_{i=0}^{h-1} 2^i (h-i), \text{ for } i = 0 \text{ to } h-1 \\ &= h + 2(h-1) + 4(h-2) + 8(h-3) + \dots + 2^{h-1} \quad (1) \end{aligned}$$

Multiplying by 2 gives the equation

$$2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \dots + 2^h \quad (2)$$

Subtract the two equations to get

$$\begin{aligned} S &= -h + 2 + 4 + 8 + 16 + \dots + 2^{h-1} + 2^h \\ &= (2^{h+1} - 1) - (h+1) \end{aligned}$$

Which proves the theorem.

# BuildHeap in Linear Time

Since a complete binary tree has between  $2^h$  and  $2^{h+1}$  nodes

$$S = (2^{h+1} - 1) - (h+1)$$

$$\star N - \log_2(N+1)$$

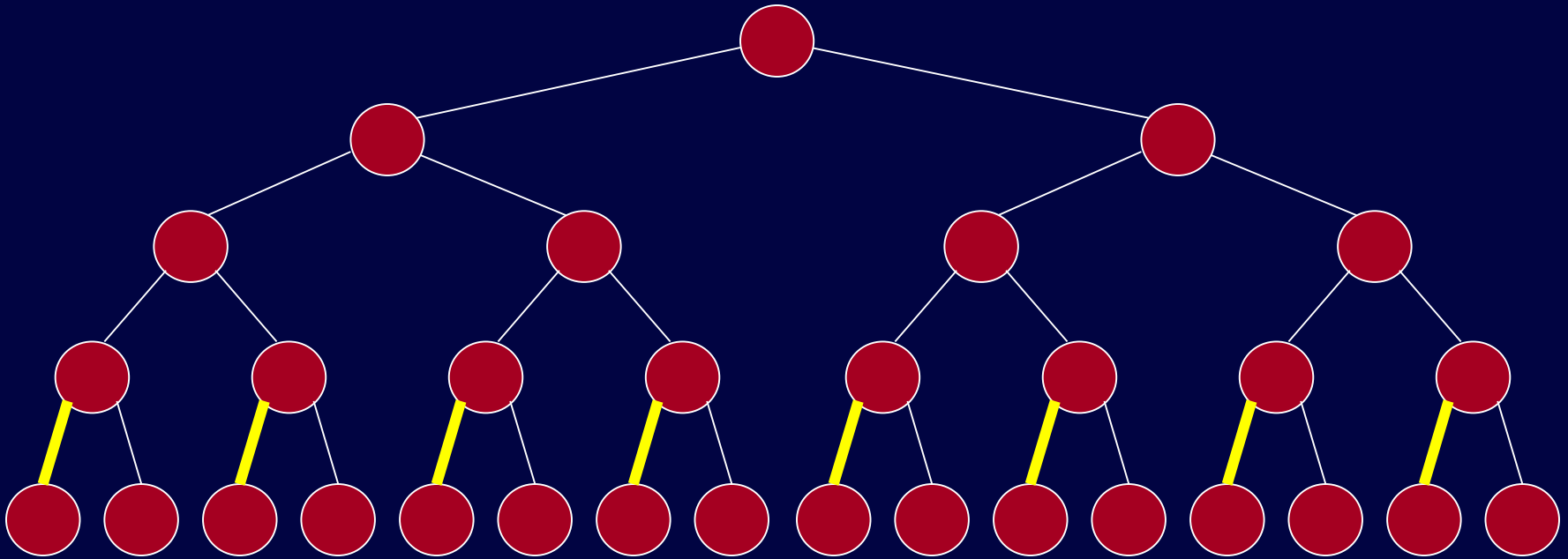
Clearly, as  $N$  gets larger, the  $\log_2(N+1)$  term becomes insignificant and  $S$  becomes a function of  $N$ .

# BuildHeap in Linear Time

- Another way to prove the theorem.
- The *height* of a node in the tree = the number of edges on the longest downward path to a leaf
- The height of a tree = the height of its root
- For any node in the tree that has some height  $h$ , darken  $h$  tree edges
  - Go down 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.

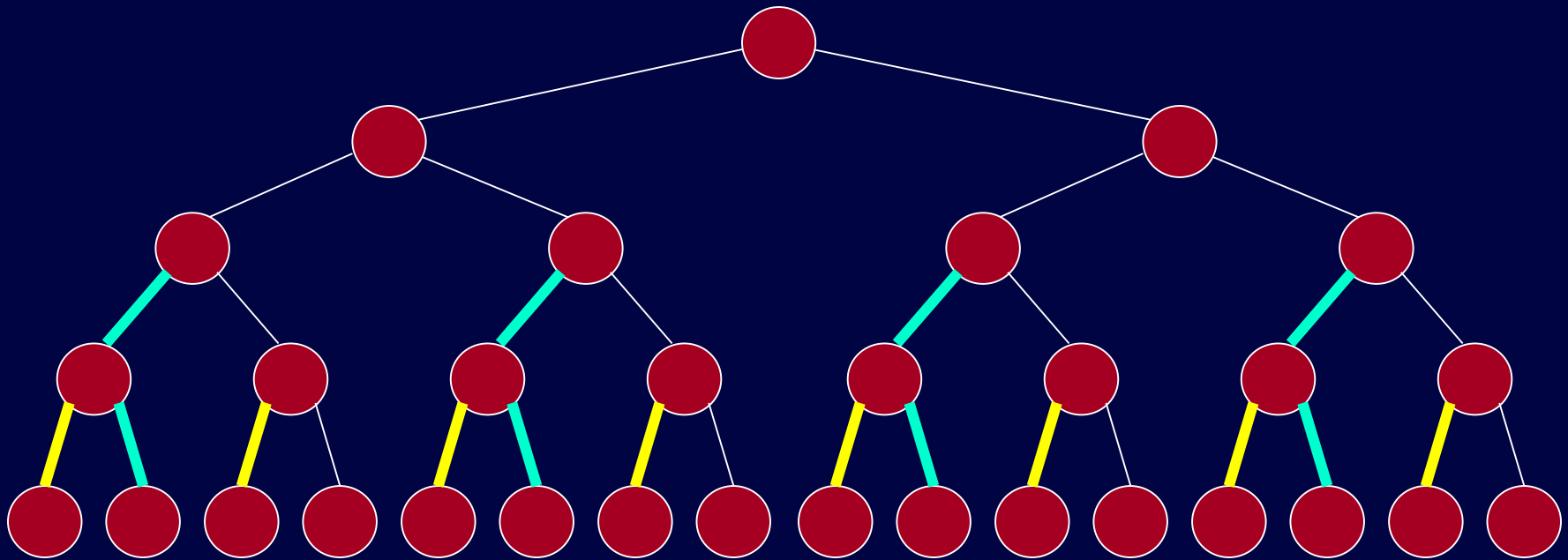


# Height 1 Nodes



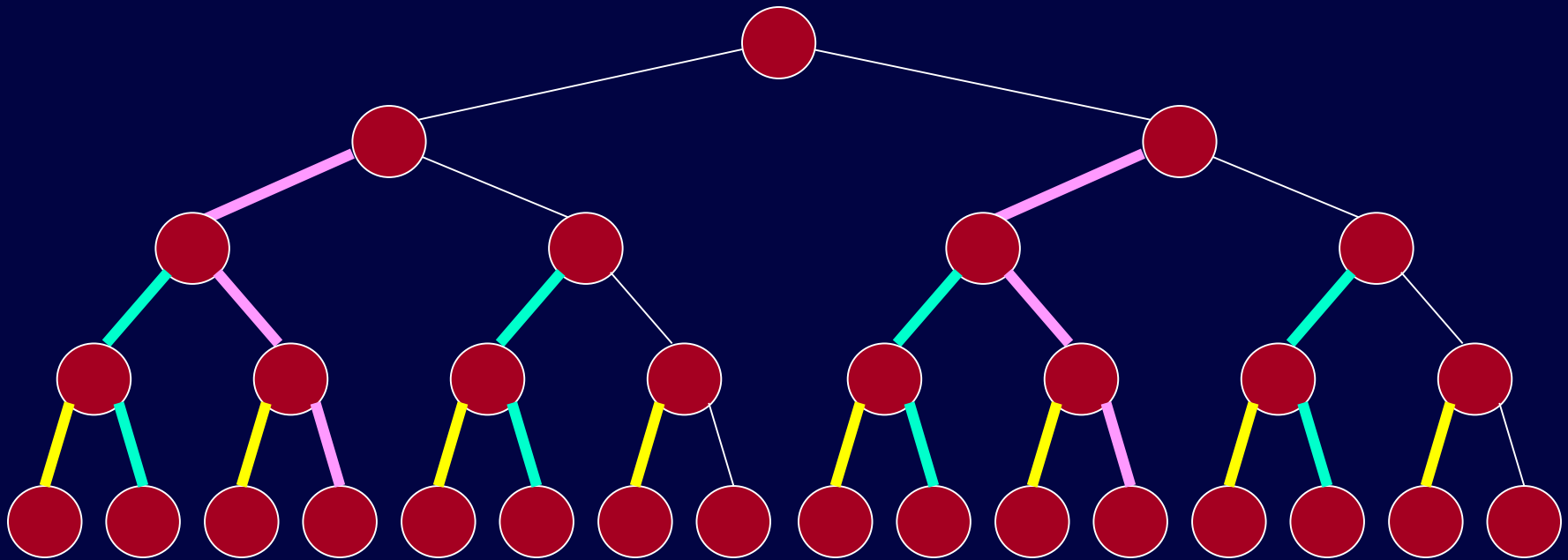
Marking the left edges for height 1 nodes

# Height 2 Nodes



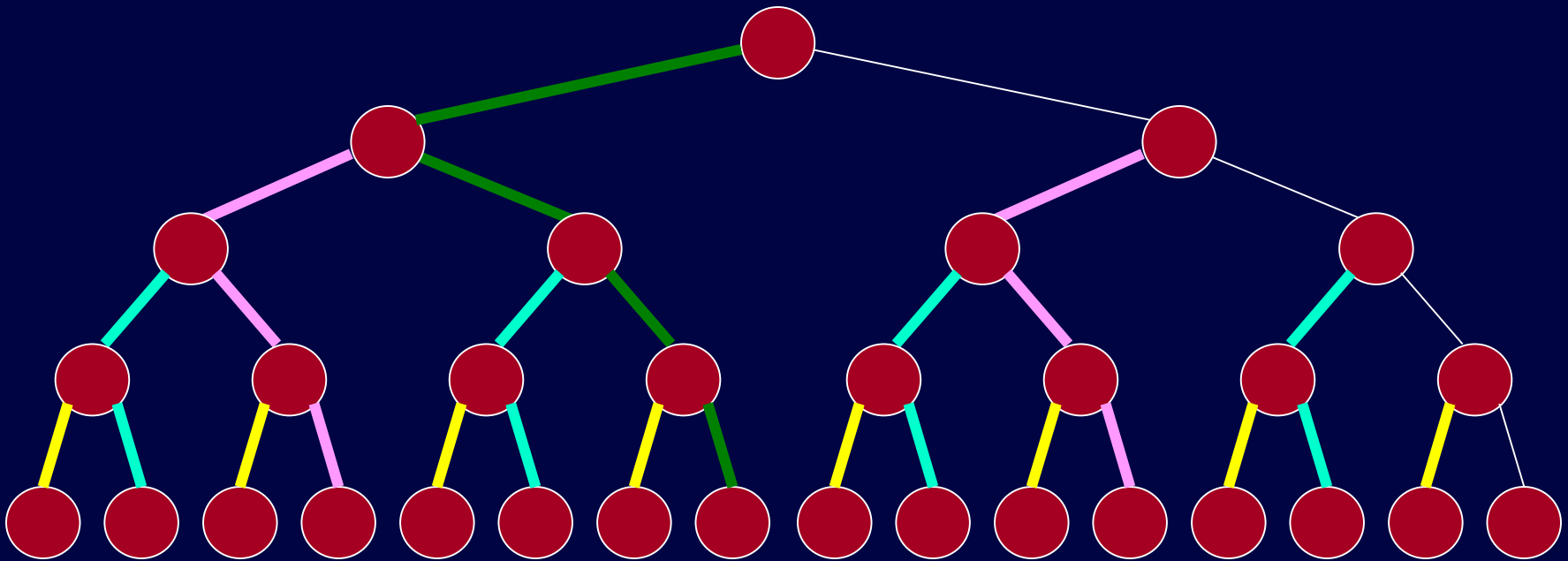
Marking the first left edge and the subsequent right edge for height 2 nodes

# Height 3 Nodes



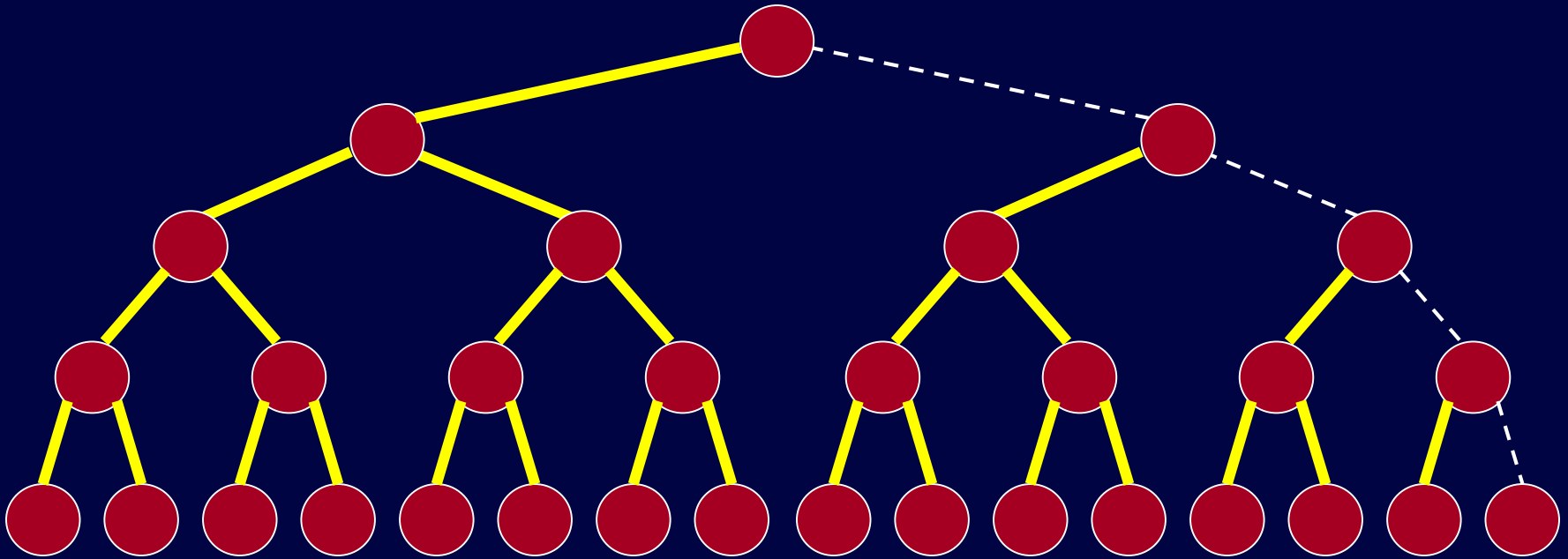
Marking the first left edge and the subsequent two right edges for height 3 nodes

# Height 4 Nodes



Marking the first left edge and the subsequent three right edges for height 4 nodes

# Theorem



N=31, treeEdges=30, H=4, dottedEdges=4 (H).  
Darkened Edges = 26 = N-H-1 (31-4-1)

# Priority Queue Using Heap

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

class PriorityQueue
{
public:
    PriorityQueue() {
        heap = new Heap<Event>( PQMAX );
    };
    ~PriorityQueue() {
        delete heap;
    };
};
```

Lecture No.33

Data Structure

Dr. Sohail Aslam

# Priority Queue Using Heap

```
Event* remove()  
{  
    if( !heap->isEmpty() ) {  
        Event* e;  
        heap->deleteMin( e );  
        return e;  
    }  
    return (Event*)NULL;  
    cout << "remove - queue is empty." << endl;  
};
```



# Priority Queue Using Heap

```
int insert(Event* e)
{
    if( !heap->isFull() ) {
        heap->insert( e );
        return 1;
    }
    cout << "insert queue is full." << endl;
    return 0;
};

int full(void) {
    return heap->isFull();
};

int length() { return heap->getSize(); };
};
```

# 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.
- One way is to put these  $N$  elements in an array and sort it. The  $k^{\text{th}}$  smallest of these is at the  $k^{\text{th}}$  position.

# The Selection Problem

- 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 interesting case is  $k = \lceil N/2 \rceil$ , since this is known as the *median*.

# HeapSort

- 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 refine this idea to obtain a fast sorting algorithm called *heapsort*.

# Disjoint Set ADT

- 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 gained by updates of the form “Haaris is related to Saad”.

# Disjoint Set ADT

- Key property: If Haaris is related to Saad and Saad is related to Ahmad, then Haaris is related to Ahmad.
- Once we have relationships information, we would like to answer queries like “Is Haaris related to Ahmad?”

# Disjoint Set ADT

## *Blob Coloring*

- A well-known low-level computer vision problem for black and white images is the following:

Gather together all the picture elements (pixels) that belong to the same "blobs", and give each pixel in each different blob an identical label.

# Disjoint Set ADT

## *Blob Coloring*

- Thus in the following image, there are five blobs.



- We want to *partition* the pixels into *disjoint sets*, one set per blob.



# Disjoint Set ADT

The image segmentation problem.



# Equivalence Relations

- A binary relation  $\sim$  over a set  $S$  is called an *equivalence relation* if it has following properties
  1. Reflexivity: for all element  $x \in S$ ,  $x \sim x$
  2. Symmetry: for all elements  $x$  and  $y$ ,  $x \sim y$  if and only if  $y \sim x$
  3. Transitivity: for all elements  $x$ ,  $y$  and  $z$ , if  $x \sim y$  and  $y \sim z$  then  $x \sim z$
- The relation “is related to” is an equivalence relation over the set of people

# Equivalence Relations

- The  $\odot$  relationship is *not* an equivalence relation.
- It is reflexive, since  $x \odot x$ ,
- and transitive, since  $x \odot y$  and  $y \odot z$  implies  $x \odot z$ ,
- it is *not symmetric* since  $x \odot y$  does not imply  $y \odot x$ .

# Equivalence Relations

- A binary relation  $\sim$  over a set  $S$  is called an *equivalence relation* if it has following properties
  1. Reflexivity: for all element  $x \in S$ ,  $x \sim x$
  2. Symmetry: for all elements  $x$  and  $y$ ,  $x \sim y$  if and only if  $y \sim x$
  3. Transitivity: for all elements  $x$ ,  $y$  and  $z$ , if  $x \sim y$  and  $y \sim z$  then  $x \sim z$
- The relation “is related to” is an equivalence relation over the set of people

Lecture No.34

Data Structure

Dr. Sohail Aslam

# Equivalence Relations

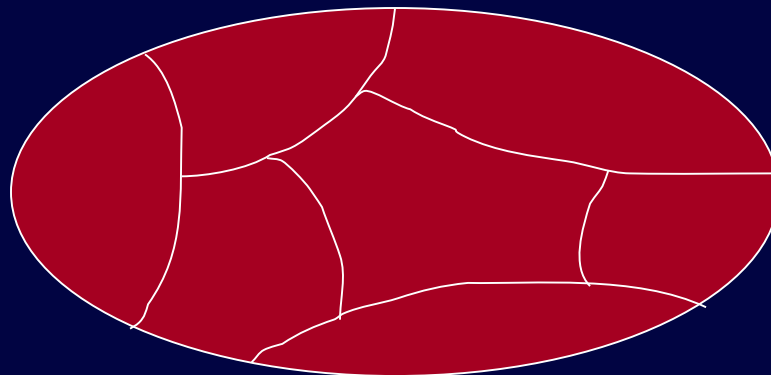
- The  $\odot$  relationship is *not* an equivalence relation.
- It is reflexive, since  $x \odot x$ ,
- and transitive, since  $x \odot y$  and  $y \odot z$  implies  $x \odot z$ ,
- it is *not symmetric* since  $x \odot y$  does not imply  $y \odot x$ .

# Equivalence Relations

- Electrical connectivity, where all connections are by metal wires, is an equivalence relation.
- It is clearly reflexive, since any component is connected to itself.
- It is symmetric because if component  $a$  is connected to component  $b$  then  $b$  must be electrically connected to  $a$ .
- It is transitive, since if component  $a$  is connected to component  $b$  and  $b$  is connected to  $c$  then  $a$  is connected to  $c$ .

# The Disjoint Sets View

- An equivalence relation  $\sim$  over a set  $S$  can be viewed as a partitioning of  $S$  into disjoint sets.
- Each set of the partition is called an *equivalence class* of  $\sim$  (all elements that are related).





# The Disjoint Sets View

- Every member of  $S$  appears in exactly one equivalence class.
- To decide if  $a \sim b$ , we need only to check whether  $a$  and  $b$  are in the same equivalence class.
- This provides a strategy to solve the equivalence problem.

# Dynamic Equivalence Problem

- If the relation is stored as a two-dimensional array of booleans, then, of course, this can be done in constant time.
- The problem is that the relation is usually not explicitly defined, but, rather, implicitly defined.

|        | Haaris   | Saad     | Ahmed    | Omar     | Asim     | Qasim    |  |  |
|--------|----------|----------|----------|----------|----------|----------|--|--|
| Haaris | <i>T</i> | <i>T</i> | <i>T</i> |          |          |          |  |  |
| Saad   |          | <i>T</i> | <i>T</i> |          |          |          |  |  |
| Ahmed  |          |          | <i>T</i> |          |          |          |  |  |
| Omar   |          |          |          | <i>T</i> |          |          |  |  |
| Asim   |          |          |          |          | <i>T</i> | <i>T</i> |  |  |
| Qasim  |          |          |          |          |          | <i>T</i> |  |  |

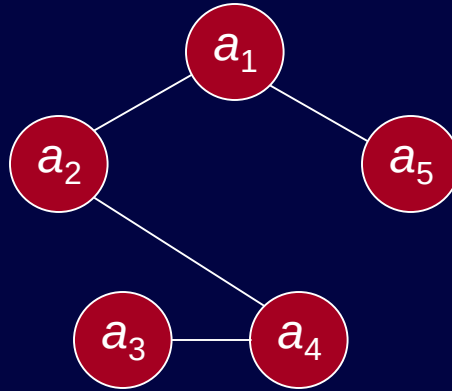
# Dynamic Equivalence Problem

- As an example, suppose the equivalence relation is defined over the five-element set  $\{a_1, a_2, a_3, a_4, a_5\}$ .
- There are 25 pairs of elements, each of which is related or not. (30 pairs – 5 self-pairs=25).

# Dynamic Equivalence Problem

- However, given the information

- $a_1 \blacksquare a_2$ ,
- $a_3 \blacksquare a_4$ ,
- $a_5 \blacksquare a_1$ ,
- $a_4 \blacksquare a_2$ ,



Implies that all pairs are related.

- We would like to be able to infer this information quickly.

# Dynamic Equivalence Problem

- The input is initially a collection of  $n$  sets, each with one element.
- This initial representation is that all relations (except reflexive relations) are false.
- Each set has a different element so that  $S_i \cap S_j = \emptyset$  ; this makes the sets *disjoint*.

# Dynamic Equivalence Problem

- There are two permissible operations: *find* and *union*.
- *Find* returns the name of the set (equivalence class) that contains a given element, i.e.,  $S_i \ni \text{find}(a)$
- *Union* merges two sets to create a new set  $S_k = S_i \times S_j$ .

# Dynamic Equivalence Problem

- If we want to add the relation  $a \blacksquare b$ , then we first see if  $a$  and  $b$  are already related.
- This is done by performing *finds* on both  $a$  and  $b$  to check whether they are in the same set.
- If they are not, we apply *union* which merges the two sets  $a$  and  $b$  belong to.
- The algorithm to do this is frequently known as *Union/Find* for this reason.

# Dynamic Equivalence Problem

- Notice that we do not perform any operations comparing the relative values of set elements.
- We merely require knowledge of their location, i.e., which set an element belongs to.
- We can thus assume that all elements are numbered sequentially from 1 to  $n$ .
- Thus, initially we have  $S_i = \{i\}$  for  $i = 1$  through  $n$ .



# Dynamic Equivalence Problem

- Our second observation is that 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  $(m+n)$ .

# Dynamic Equivalence Problem

- We will use a tree to represent each set, since each element in a tree has the same root.
- The root can be used to name the set.
- There will be a collection of trees, each tree representing one set. A collection of trees is called a *forest*.

Lecture No.35

Data Structure

Dr. Sohail Aslam

# Dynamic Equivalence Problem

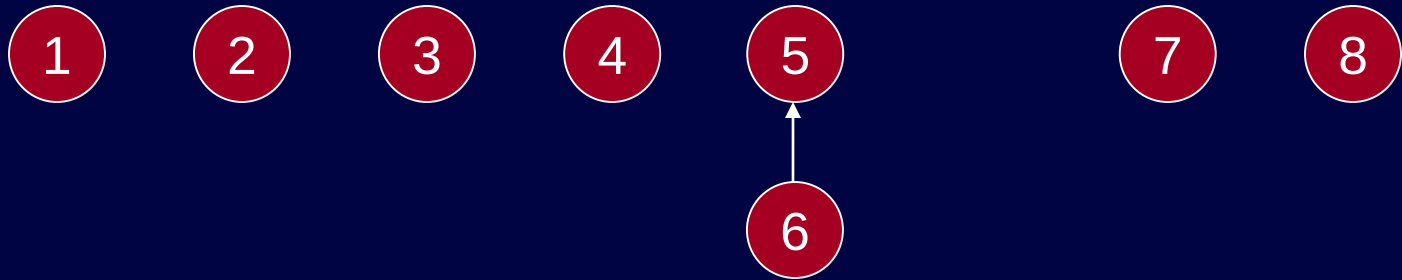
- The trees we will use are not necessarily binary.
- To perform union of two sets, we merge the two trees by making the root of one point to the root of the other.
- A *find*( $x$ ) on element  $x$  is performed by returning the root of the tree containing  $x$ .

# Dynamic Equivalence Problem



Eight elements, initially in different sets.

# Dynamic Equivalence Problem



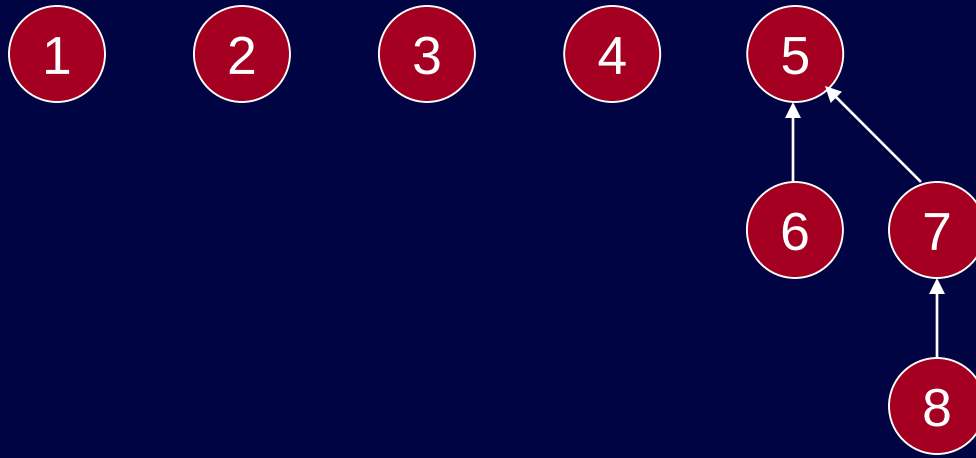
After union(5,6)

# Dynamic Equivalence Problem



After union(7,8)

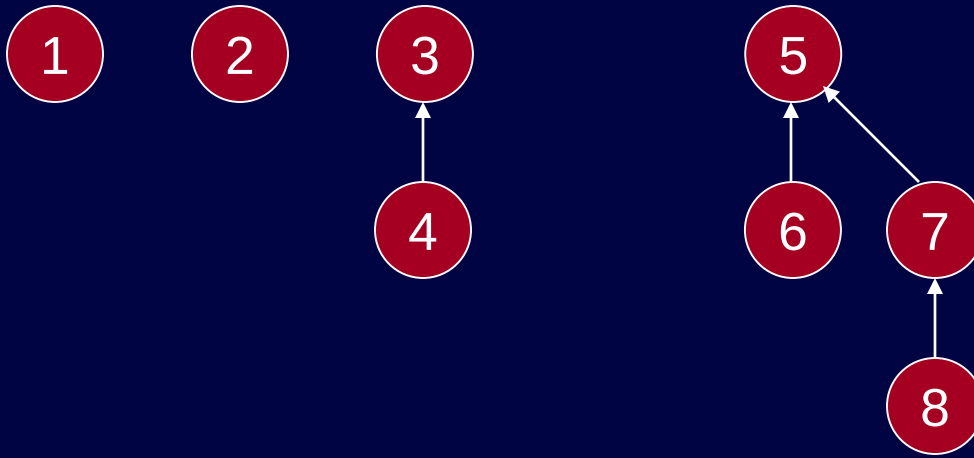
# Dynamic Equivalence Problem



After union(5,7)

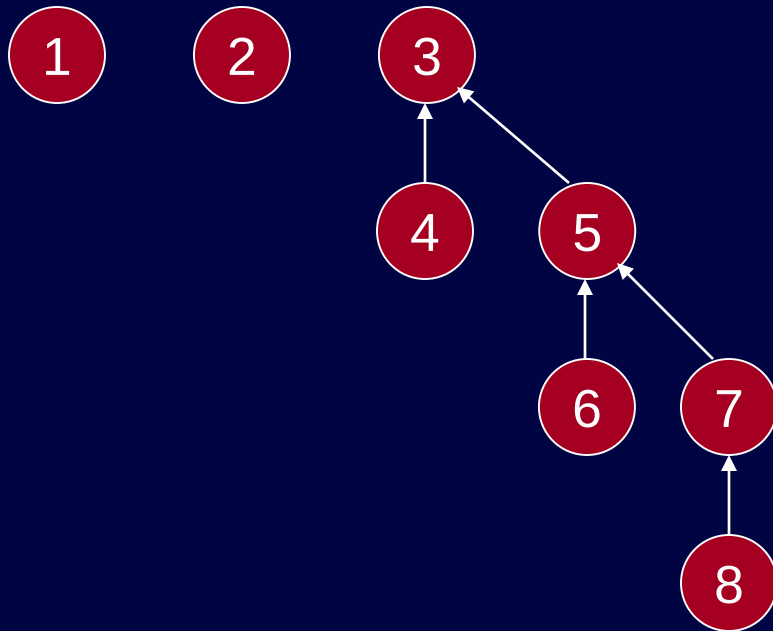


# Dynamic Equivalence Problem



After union(3,4)

# Dynamic Equivalence Problem



After union(4,5)

# Dynamic Equivalence Problem

- Typical tree traversal not required, so no need for pointers to children, instead we need a pointer to parent – an *up-tree*
- Parent pointers can be stored in an array: *parent[i]* (set to -1 if i is root).
- The algorithm for *find* and *union* can thus be:

# Dynamic Equivalence Problem

Initialization:

```
for (i=0; i < n; i++)  
    parent[i] = -1;
```

find(i):

```
// traverse to the root (-1)  
for(j=i; parent[j] >= 0; j=parent[j])  
    ;  
return j;
```

# Dynamic Equivalence Problem

union(i,j):

```
    root_i = find(i);
```

```
    root_j = find(j);
```

```
    if (root_i != root_j)
```

```
        parent[root_j] = root_i;
```

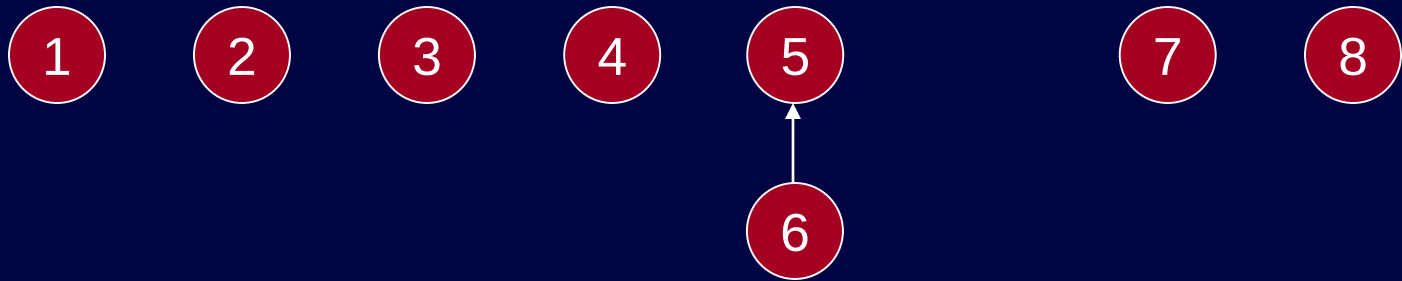
# Parent Array



|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |

Eight elements, initially in different sets.

# Parent Array



|    |    |    |    |    |   |    |    |
|----|----|----|----|----|---|----|----|
| -1 | -1 | -1 | -1 | -1 | 5 | -1 | -1 |
| 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  |

After union(5,6)

# Parent Array



|    |    |    |    |    |   |    |   |
|----|----|----|----|----|---|----|---|
| -1 | -1 | -1 | -1 | -1 | 5 | -1 | 7 |
| 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8 |

After union(7,8)



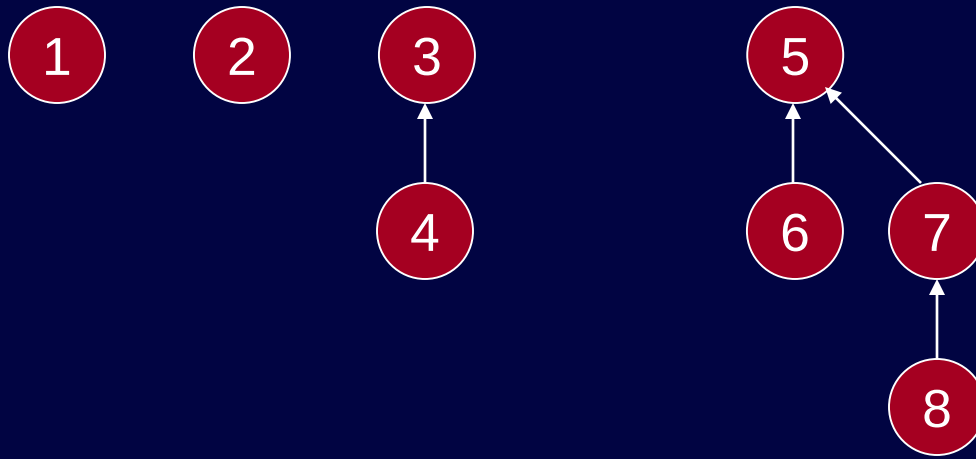
# Parent Array



|    |    |    |    |    |   |   |   |
|----|----|----|----|----|---|---|---|
| -1 | -1 | -1 | -1 | -1 | 5 | 5 | 7 |
| 1  | 2  | 3  | 4  | 5  | 6 | 7 | 8 |

After union(5,7)

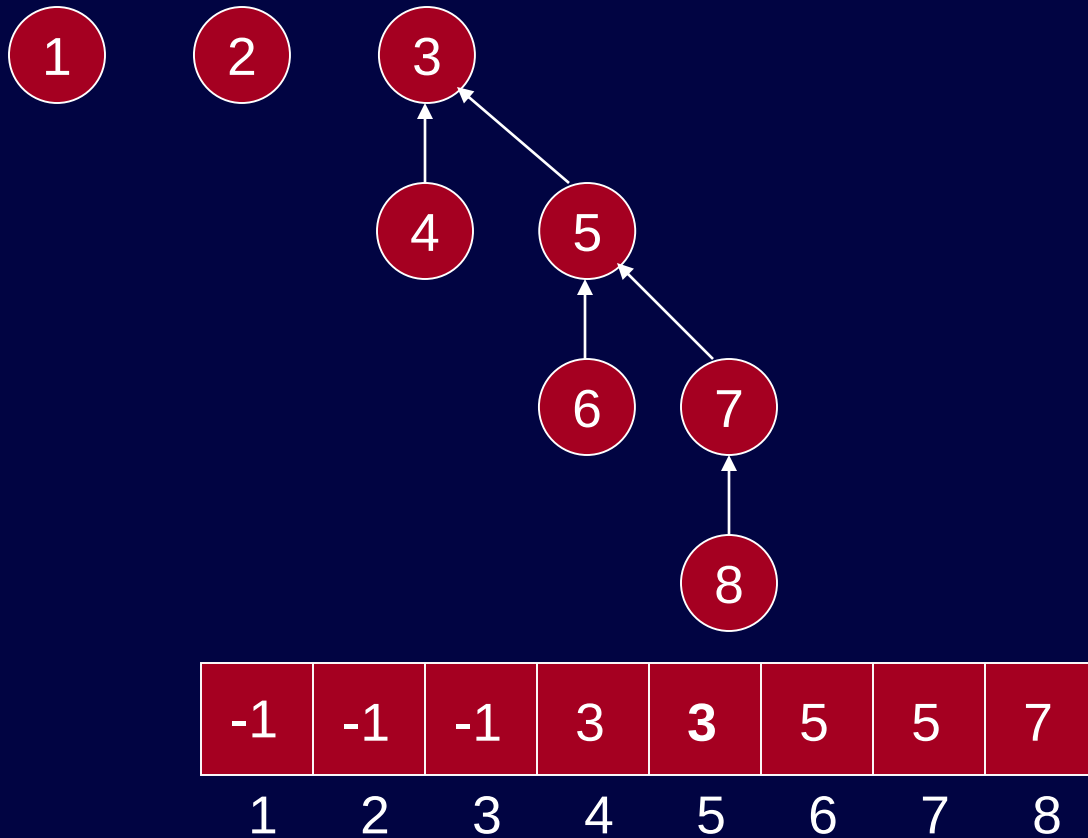
# Parent Array



|    |    |    |   |    |   |   |   |
|----|----|----|---|----|---|---|---|
| -1 | -1 | -1 | 3 | -1 | 5 | 5 | 7 |
| 1  | 2  | 3  | 4 | 5  | 6 | 7 | 8 |

After union(3,4)

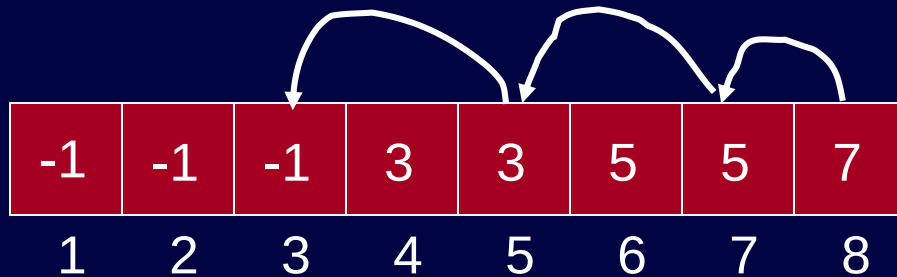
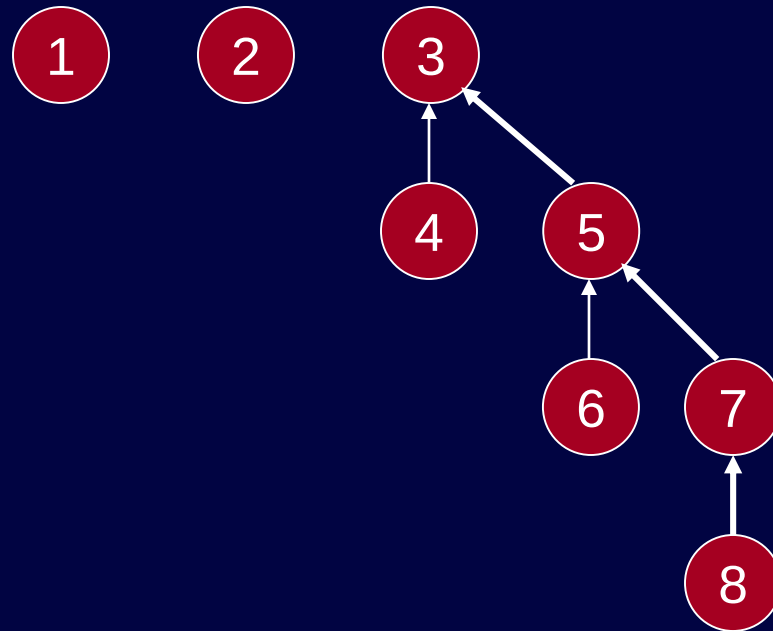
# Parent Array



After union(4,5)

# Parent Array

Find(8)



# Running Time Analysis

- *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

# Running Time Analysis

- *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

Lecture No.36

# Data Structures

Dr. Sohail Aslam

# *Union by Size*

- 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 also called *union-by-weight*.



# Union by Size

```
union(i,j) :  
    root1 = find(i);  
    root2 = find(j);  
    if (root1 != root2)  
        if (parent[root1] <= parent[root2])  
        {  
            // first tree has more nodes  
            parent[root1] += parent[root2];  
            parent[root2] = root1;  
        }  
        else { // second tree has more nodes  
            parent[root2] += parent[root1];  
            parent[root1] = root2;  
        }  
    }
```

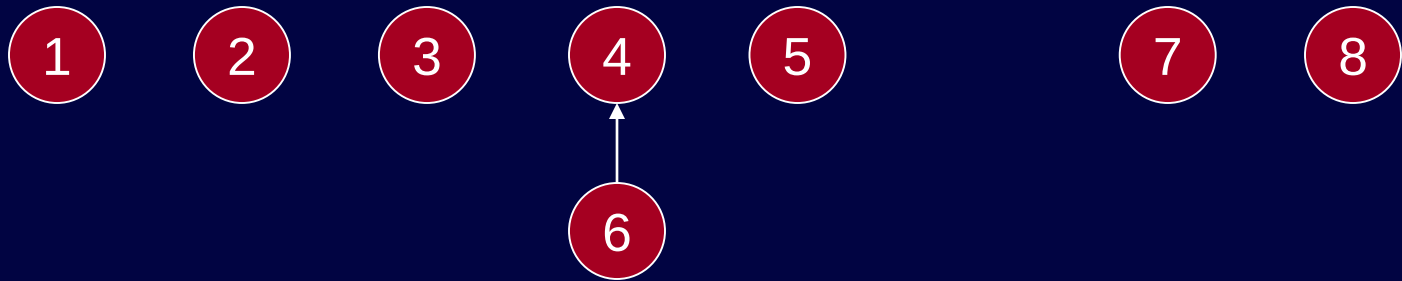
# *Union by Size*



|    |    |    |    |    |    |    |    |
|----|----|----|----|----|----|----|----|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |

Eight elements, initially in different sets.

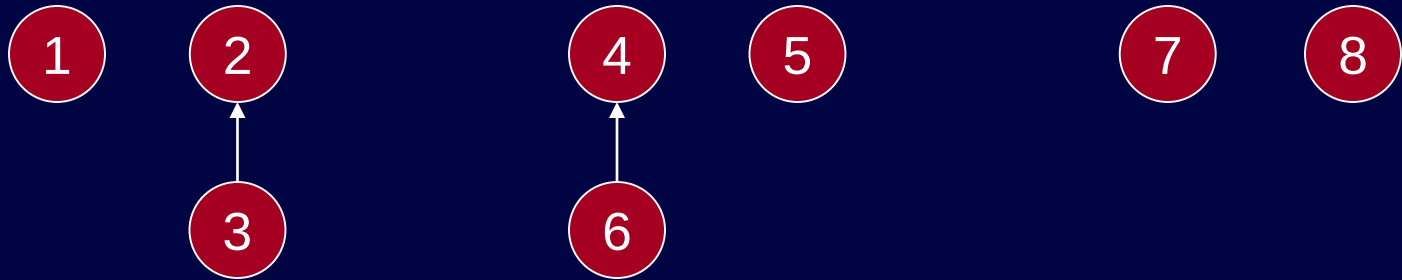
# *Union by Size*



|    |    |    |    |    |   |    |    |
|----|----|----|----|----|---|----|----|
| -1 | -1 | -1 | -2 | -1 | 4 | -1 | -1 |
| 1  | 2  | 3  | 4  | 5  | 6 | 7  | 8  |

Union(4,6)

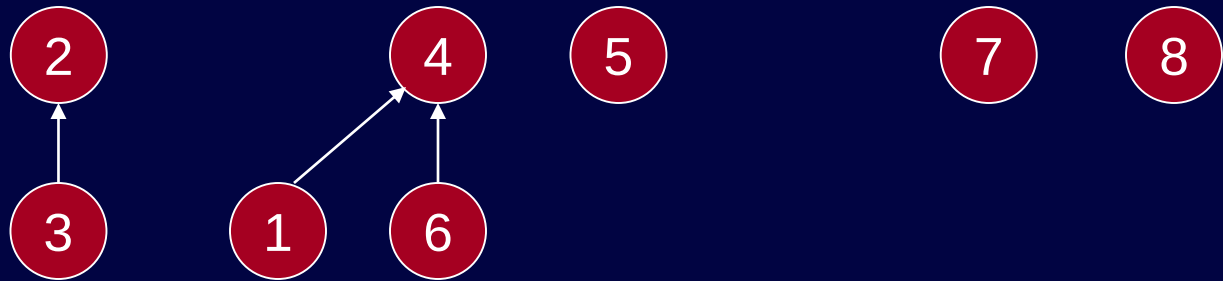
# *Union by Size*



|    |    |   |    |    |   |    |    |
|----|----|---|----|----|---|----|----|
| -1 | -2 | 2 | -2 | -1 | 4 | -1 | -1 |
| 1  | 2  | 3 | 4  | 5  | 6 | 7  | 8  |

Union(2,3)

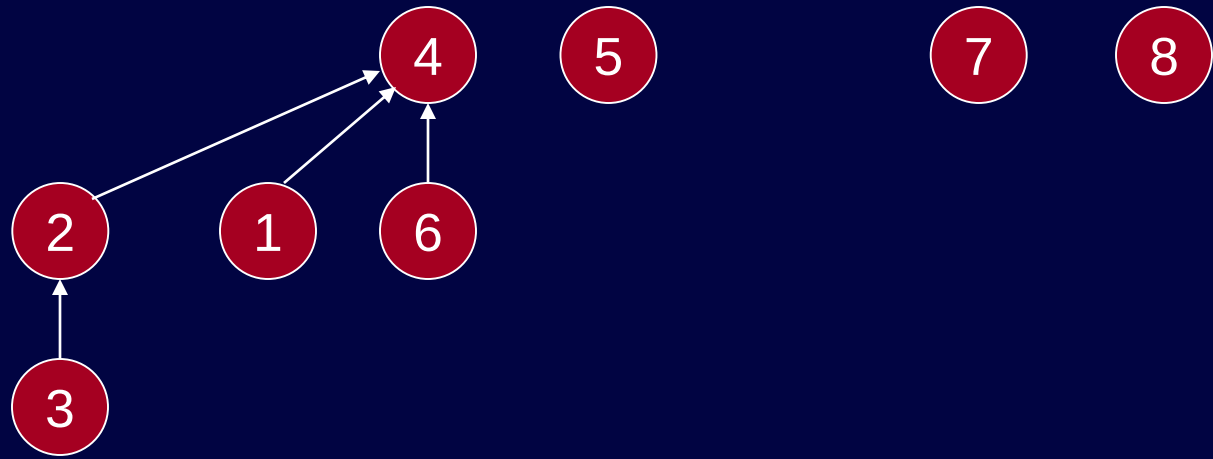
# Union by Size



|   |    |   |    |    |   |    |    |
|---|----|---|----|----|---|----|----|
| 4 | -2 | 2 | -3 | -1 | 4 | -1 | -1 |
| 1 | 2  | 3 | 4  | 5  | 6 | 7  | 8  |

Union(1,4)

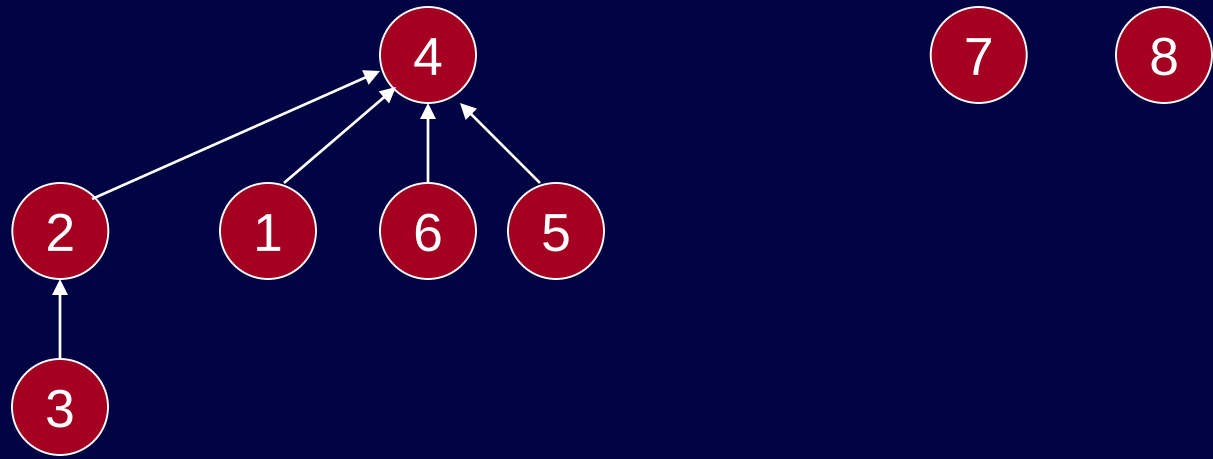
# Union by Size



|   |   |   |    |    |   |    |    |
|---|---|---|----|----|---|----|----|
| 4 | 4 | 2 | -5 | -1 | 4 | -1 | -1 |
| 1 | 2 | 3 | 4  | 5  | 6 | 7  | 8  |

Union(2,4)

# Union by Size



|   |   |   |    |   |   |    |    |
|---|---|---|----|---|---|----|----|
| 4 | 4 | 2 | -6 | 4 | 4 | -1 | -1 |
| 1 | 2 | 3 | 4  | 5 | 6 | 7  | 8  |

Union(5,4)

# Analysis of *Union* by Size

- If unions are done by weight (size), the depth of any element is never greater than  $\log_2 n$ .



# Analysis of *Union* by Size

- 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 done? --  $\log_2 n$  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.

# Sprucing up *Find*

- So far we have tried to optimize *union*.
- Can we optimize *find*?
- Yes, using *path compression* (or compaction).

# Sprucing up *Find*

- 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 benefits later!
- Subsequent *find* may do less work

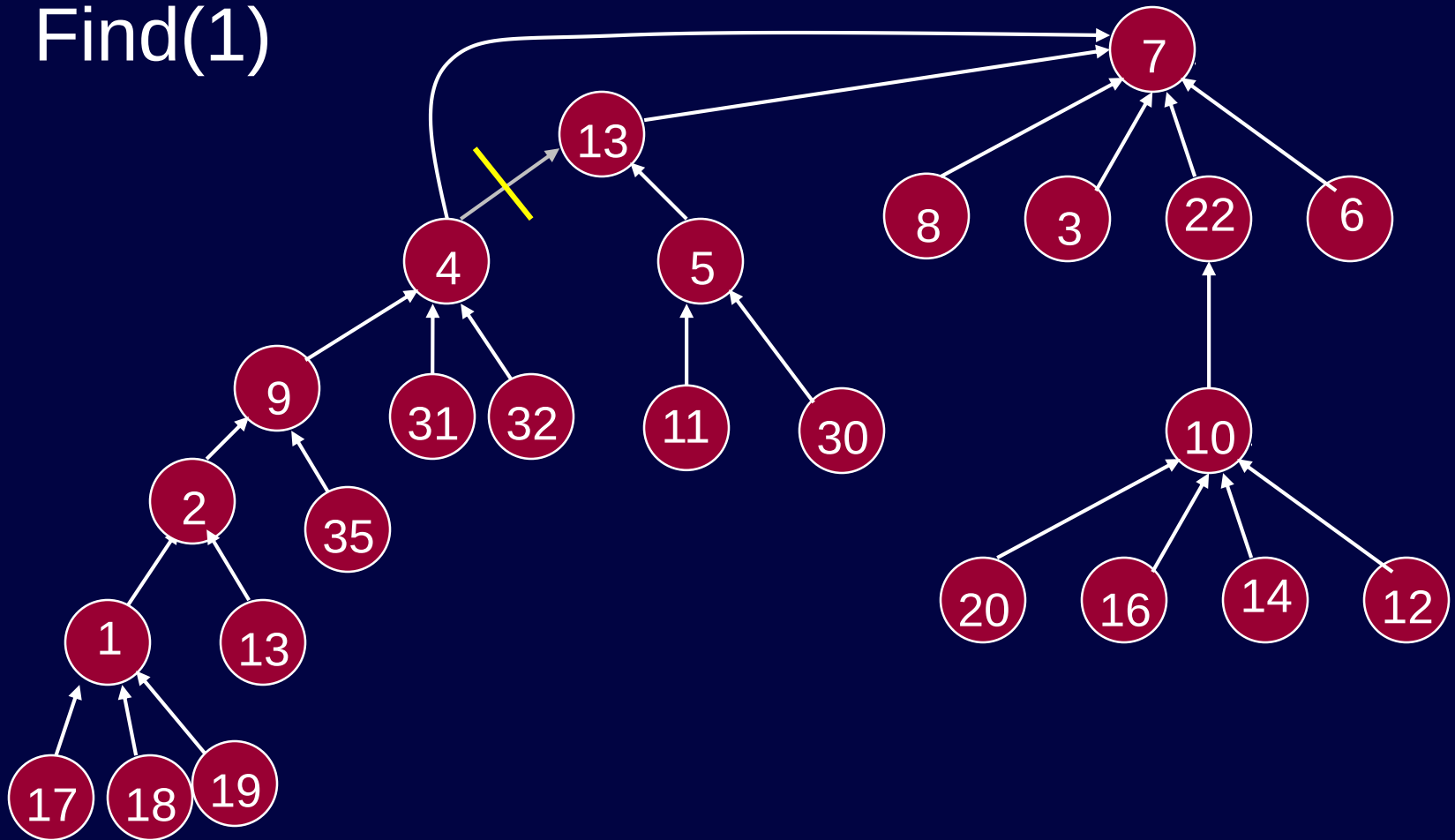
# Sprucing up *Find*

- Updated code for *find*

```
find (i)
{
    if (parent[i] < 0)
        return i;
    else
        return parent[i] = find(parent[i]);
}
```

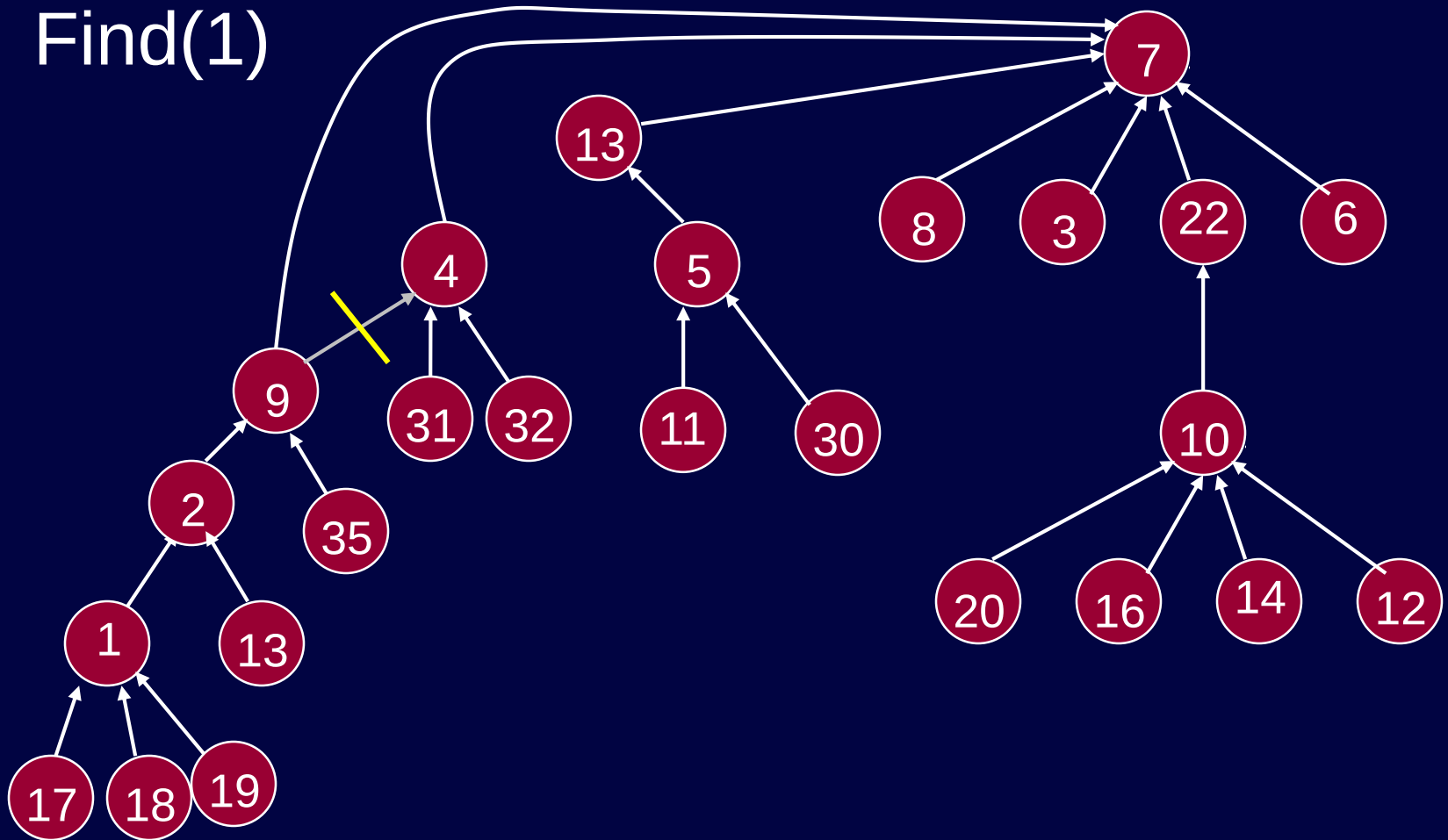
# Path Compression

- Find(1)



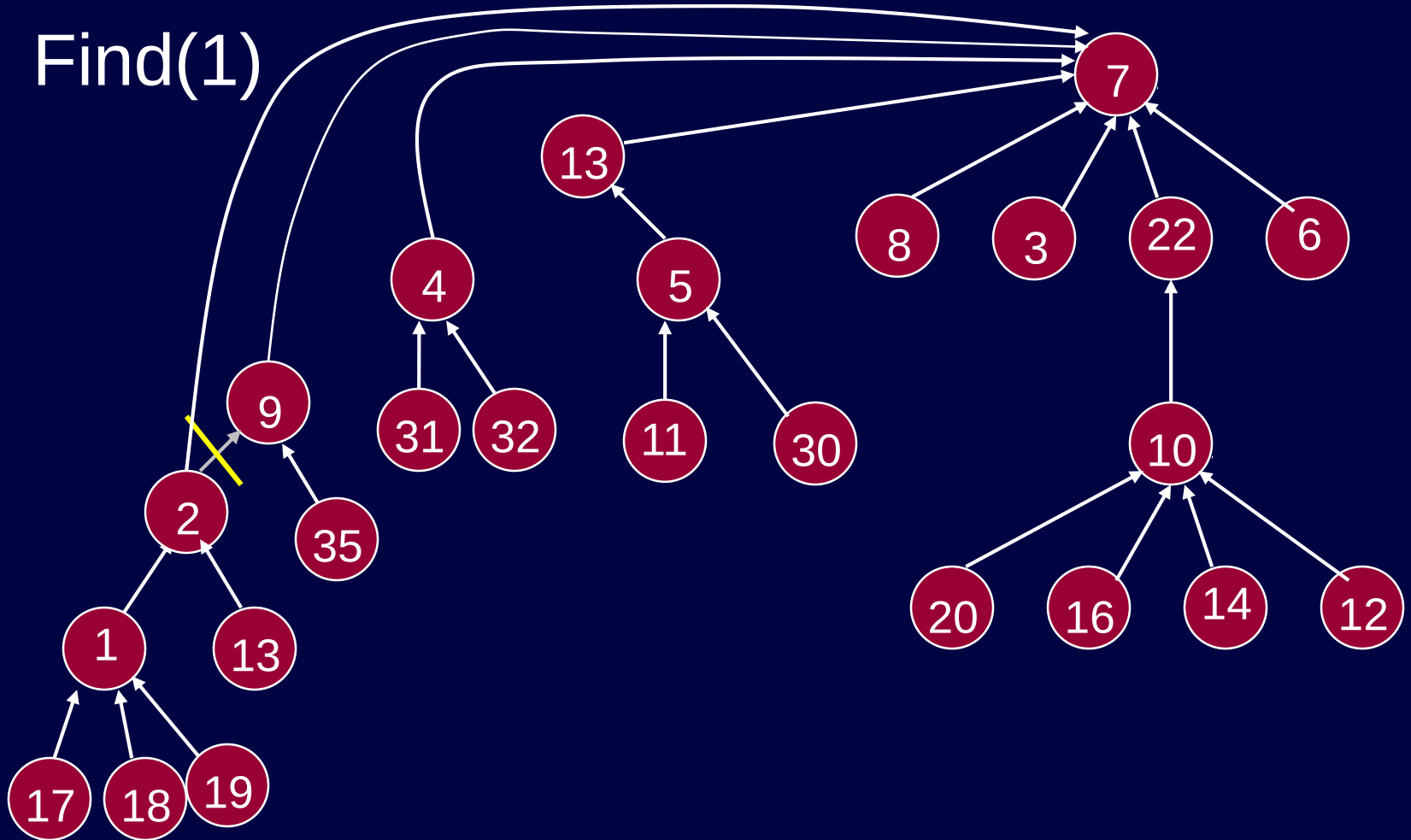
# Path Compression

- Find(1)



# Path Compression

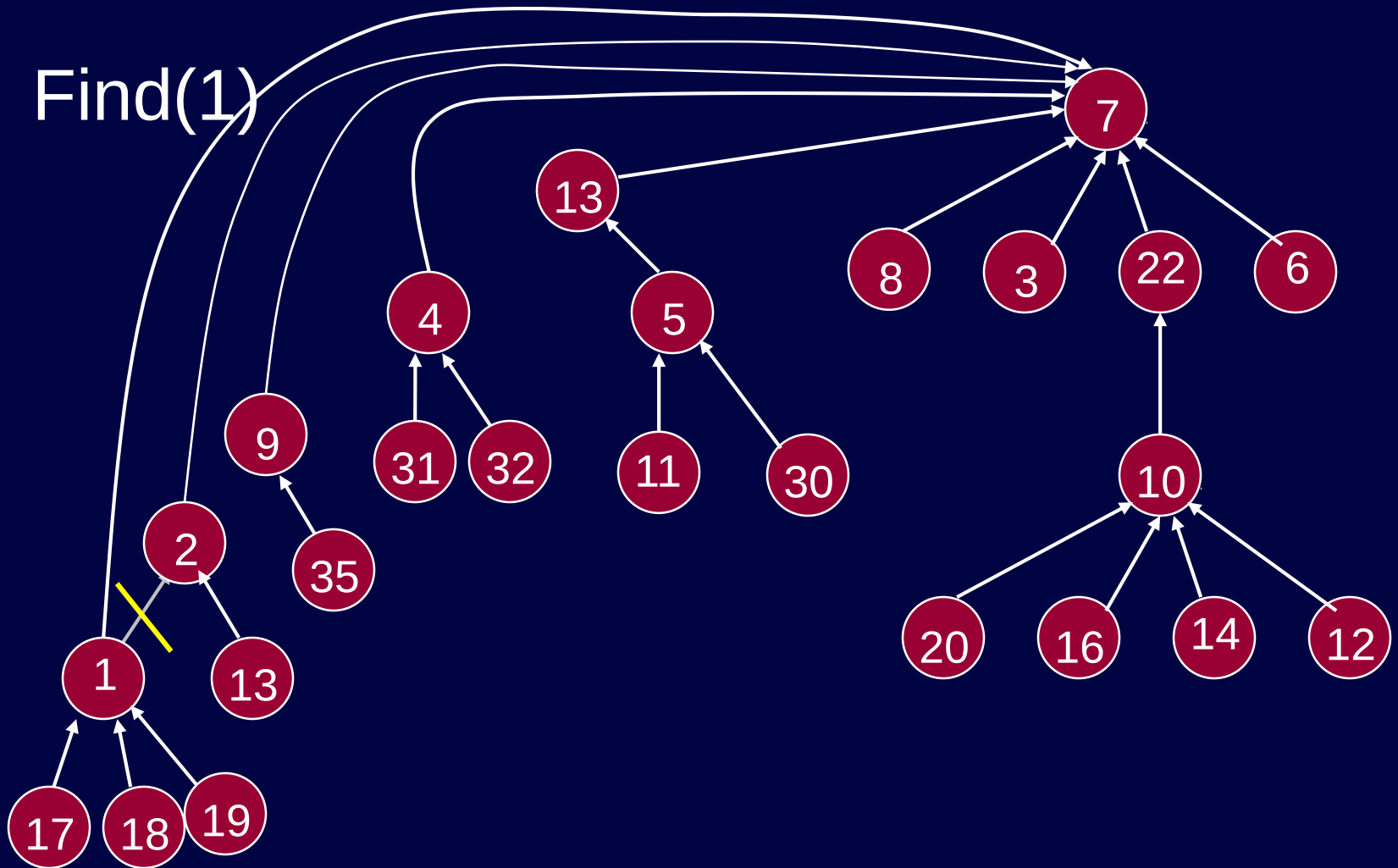
- Find(1)





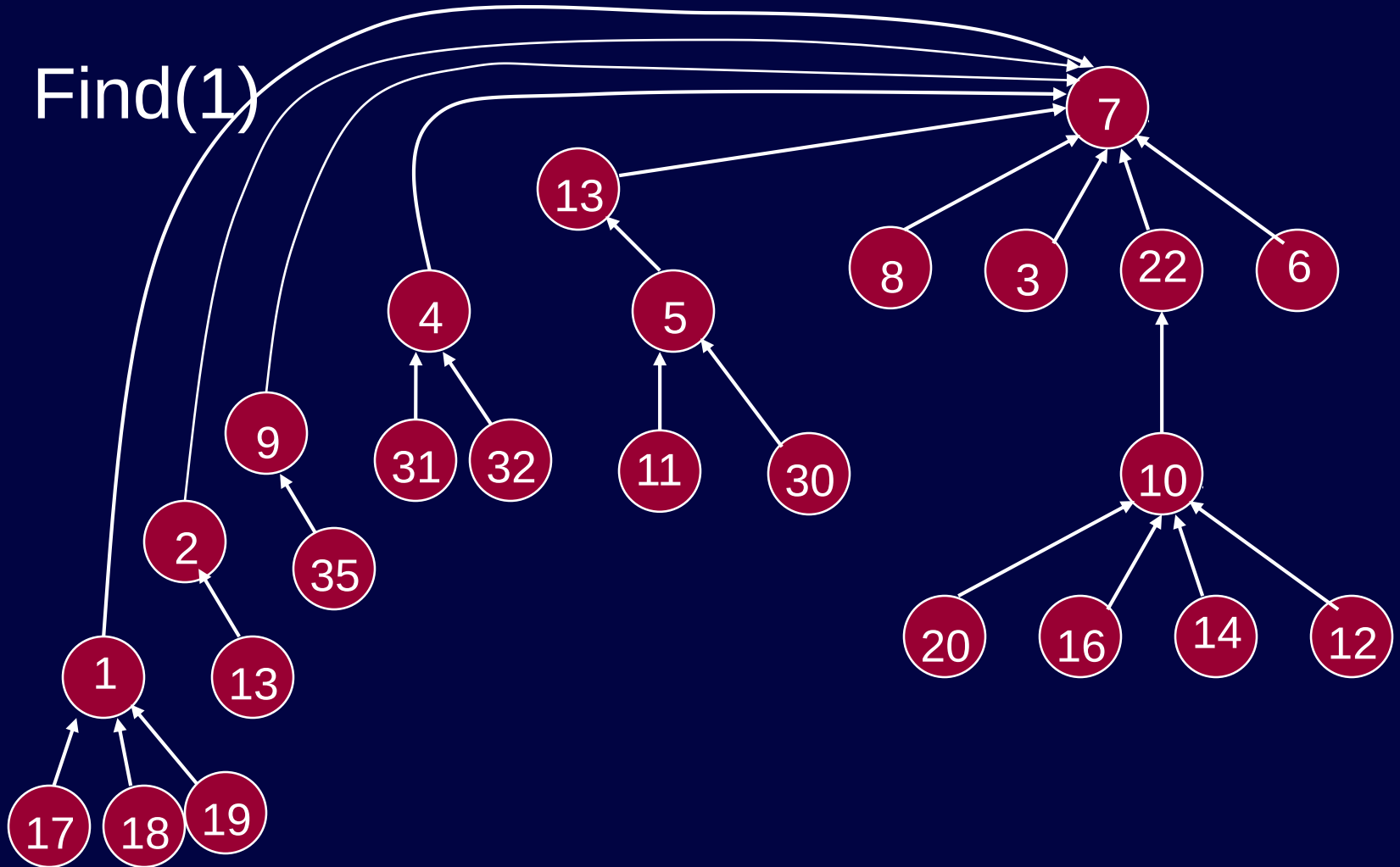
# Path Compression

■ Find(1)



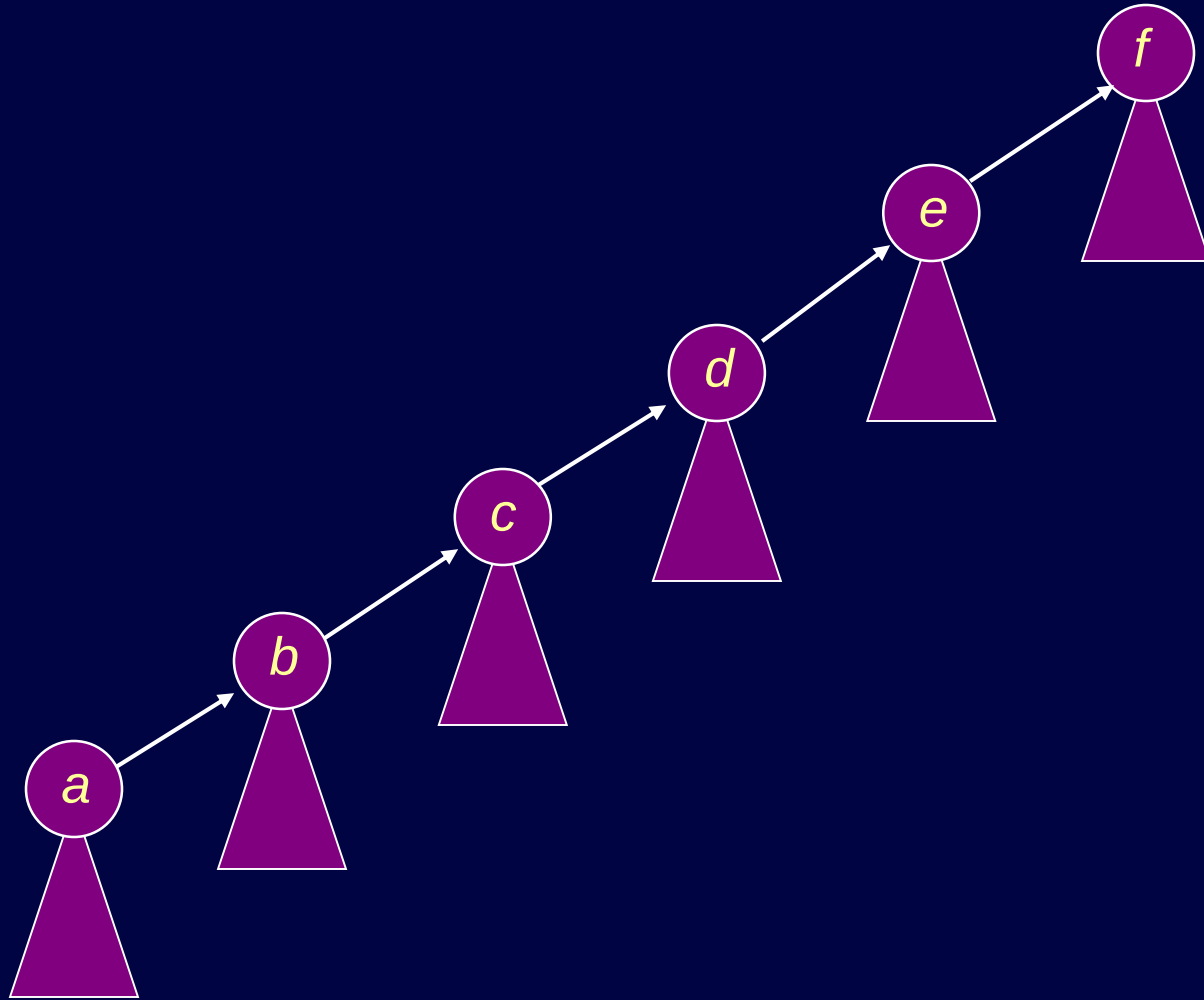
# Path Compression

- Find(1)



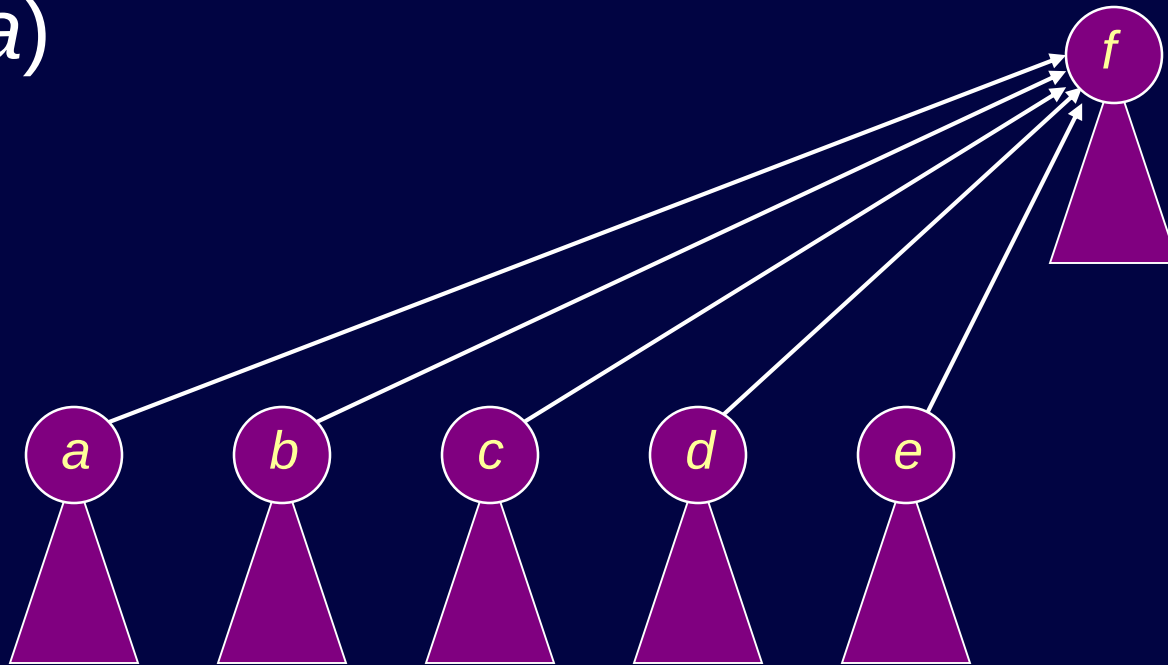
# Path Compression

- *Find(a)*



# Path Compression

- *Find(a)*



# 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  $(m \& n)$ .
- $\&(n)$  is the inverse Ackermann's function which grows extremely slowly. For all practical puposes,  $\&(n) \leq 4$ .
- Union-find is essentially proportional to  $m$  for a sequence of  $m$  operations, linear in  $m$ .

# 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  $(m \& n)$ .
- $\&(n)$  is the inverse Ackermann's function which grows extremely slowly. For all practical puposes,  $\&(n) \leq 4$ .
- Union-find is essentially proportional to  $m$  for a sequence of  $m$  operations, linear in  $m$ .

Lecture No.37

# Data Structures

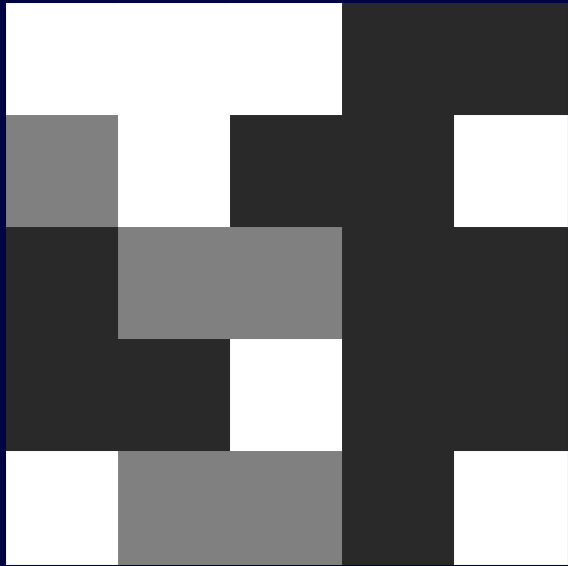
Dr. Sohail Aslam

# Image Segmentation

- Inclusion criteria for pixels
  - use pixel intensity,
  - threshold of intensity,
  - threshold for difference in intensity of neighbors,
  - texture (ie. a pattern of pixel intensities)



# Image Segmentation



|   | 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 |

# Image Segmentation

|   | 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 |

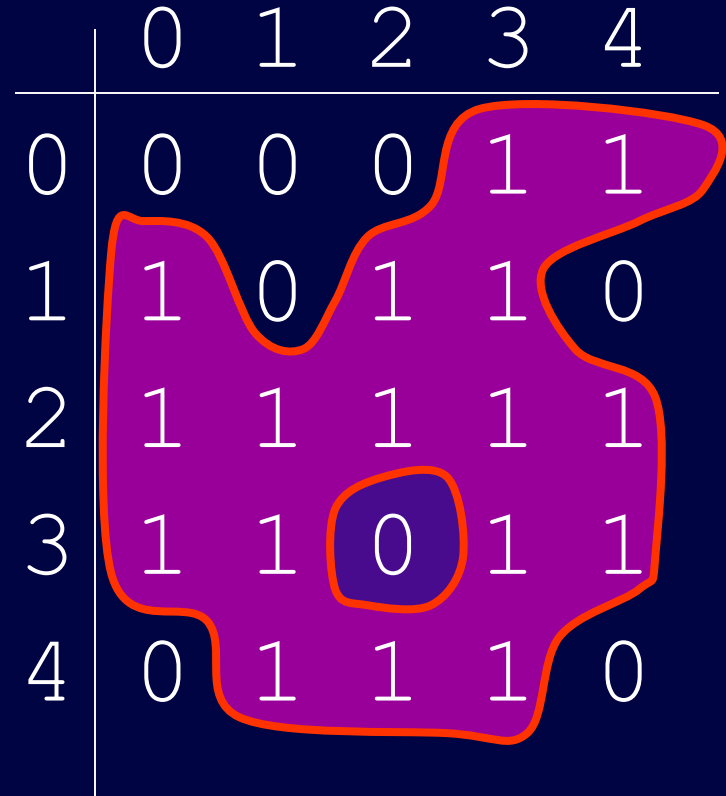
|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 1 | 1 |
| 3 | 1 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 1 | 0 |

Threshold=4

# Image Segmentation

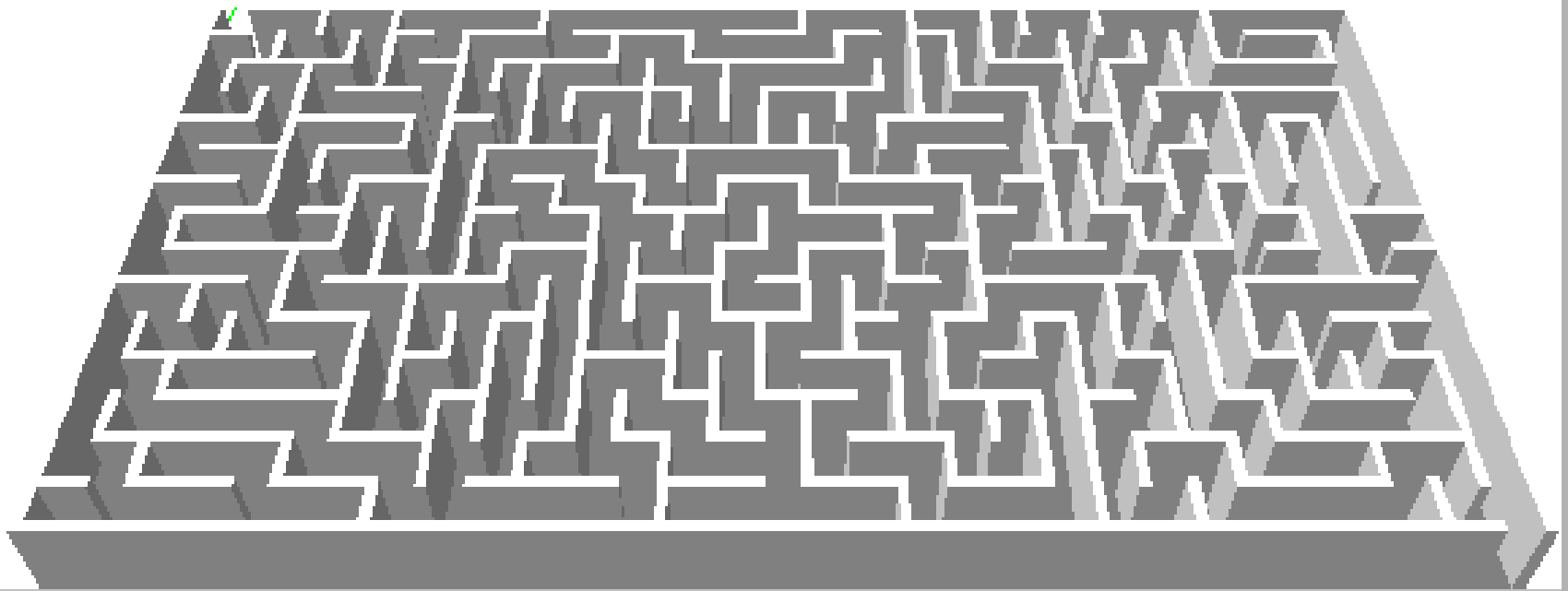
|   | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 4 | 4 |
| 1 | 2 | 0 | 4 | 4 | 0 |
| 2 | 4 | 2 | 2 | 4 | 4 |
| 3 | 4 | 4 | 0 | 4 | 4 |
| 4 | 0 | 2 | 2 | 4 | 0 |

|   | 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 |



Threshold=2

# Maze Generation



# Maze Generation

- A random maze generator can use union-find. Consider a 5x5 maze:

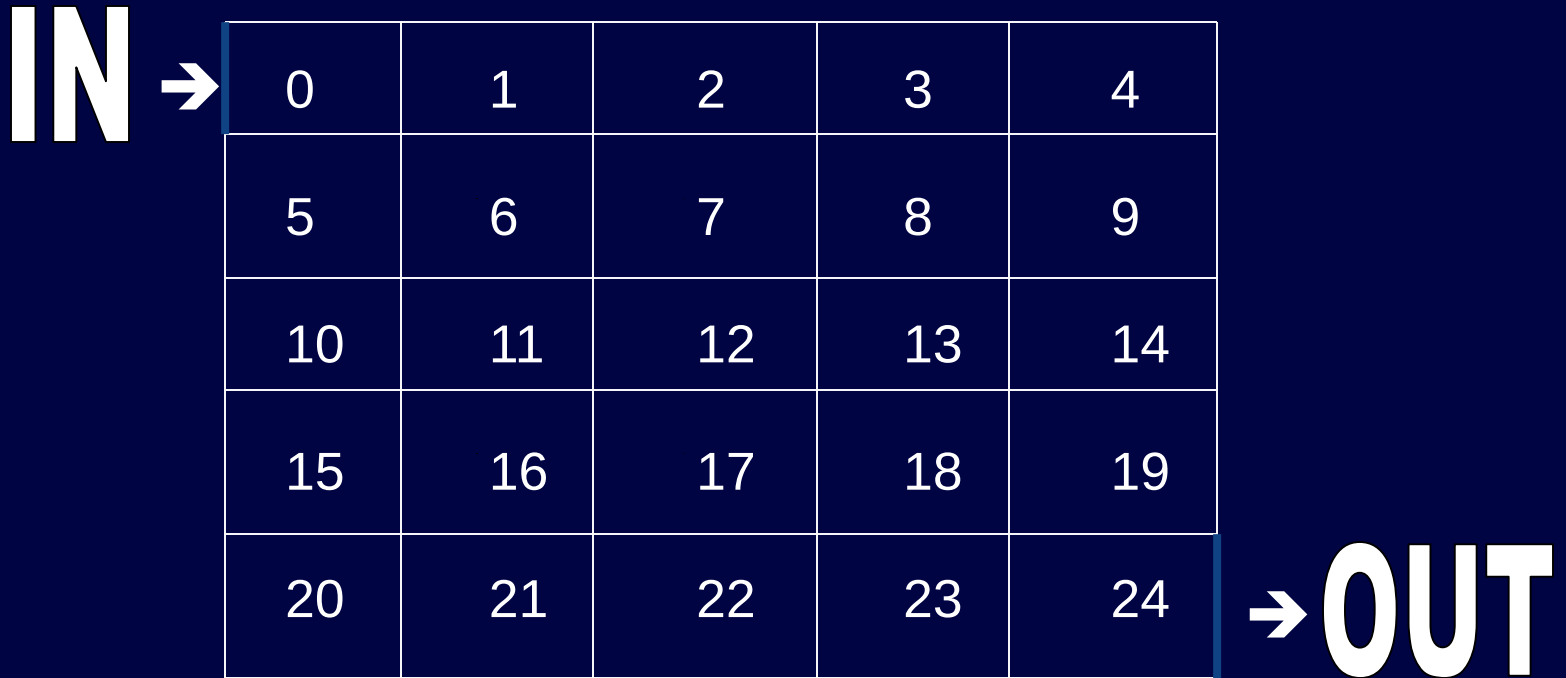
|    |    |    |    |    |
|----|----|----|----|----|
| 0  | 1  | 2  | 3  | 4  |
| 5  | 6  | 7  | 8  | 9  |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

# Maze Generator

- Initially, 25 cells, each isolated by walls from the others.
- This corresponds to an equivalence relation -- 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).

# Maze Generator

- To start, choose an entrance and an exit.



# Maze Generator

- Randomly remove walls until the entrance and exit cells are in the same set.
- Removing a 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.

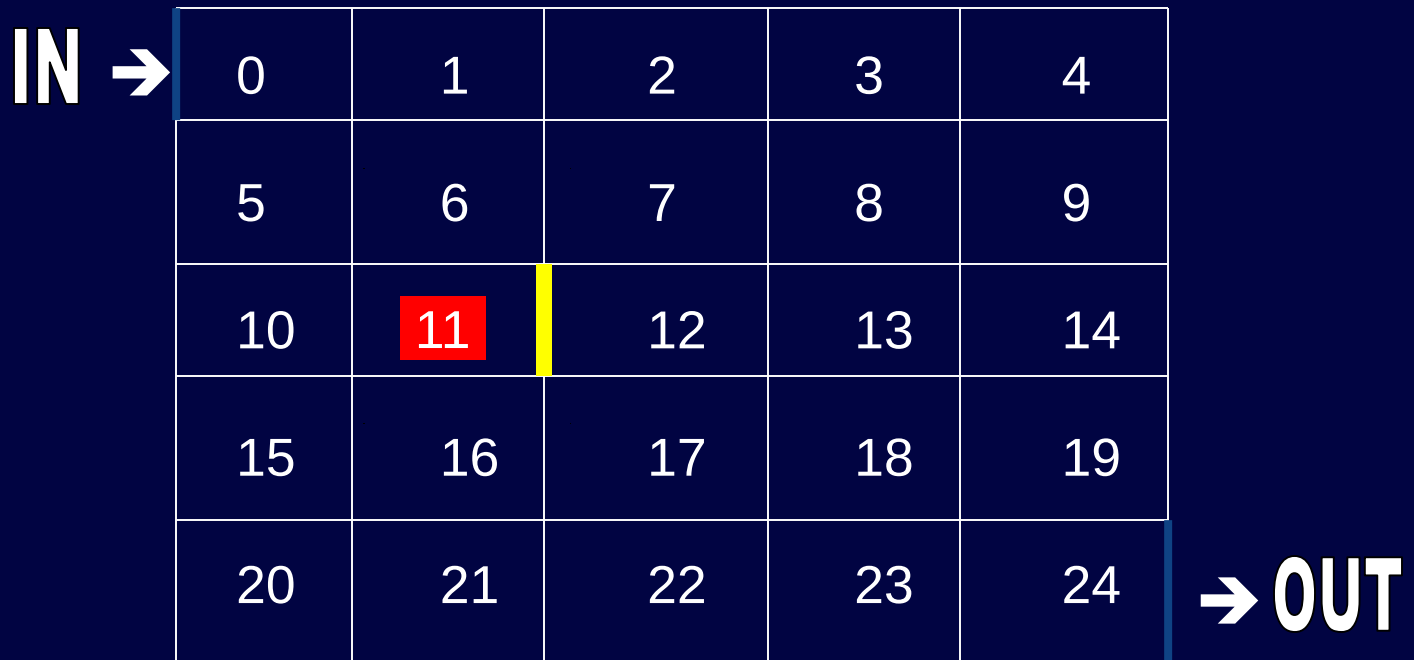


# MakeMaze

```
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)  
        }  
    }  
}
```

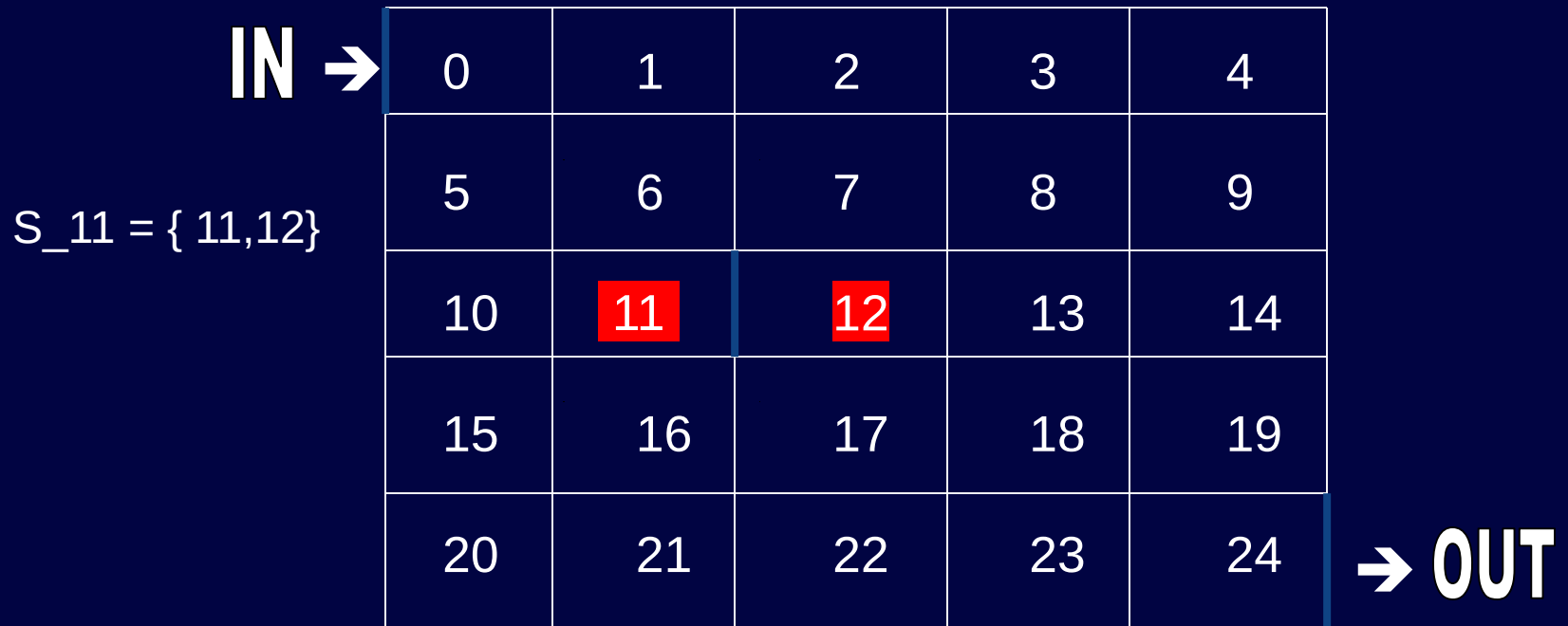
# Maze Generator

Cell 11, right wall chosen randomly



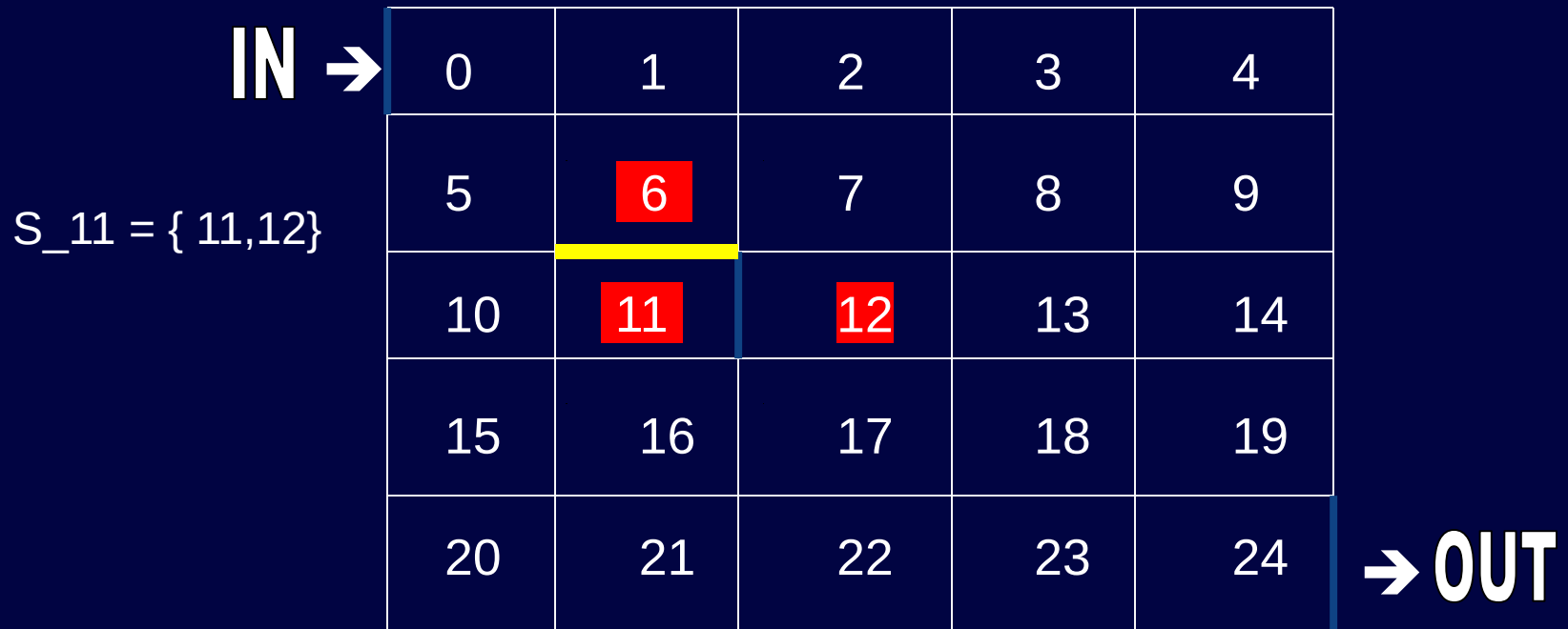
# Maze Generator

Cell 11, right wall chosen randomly



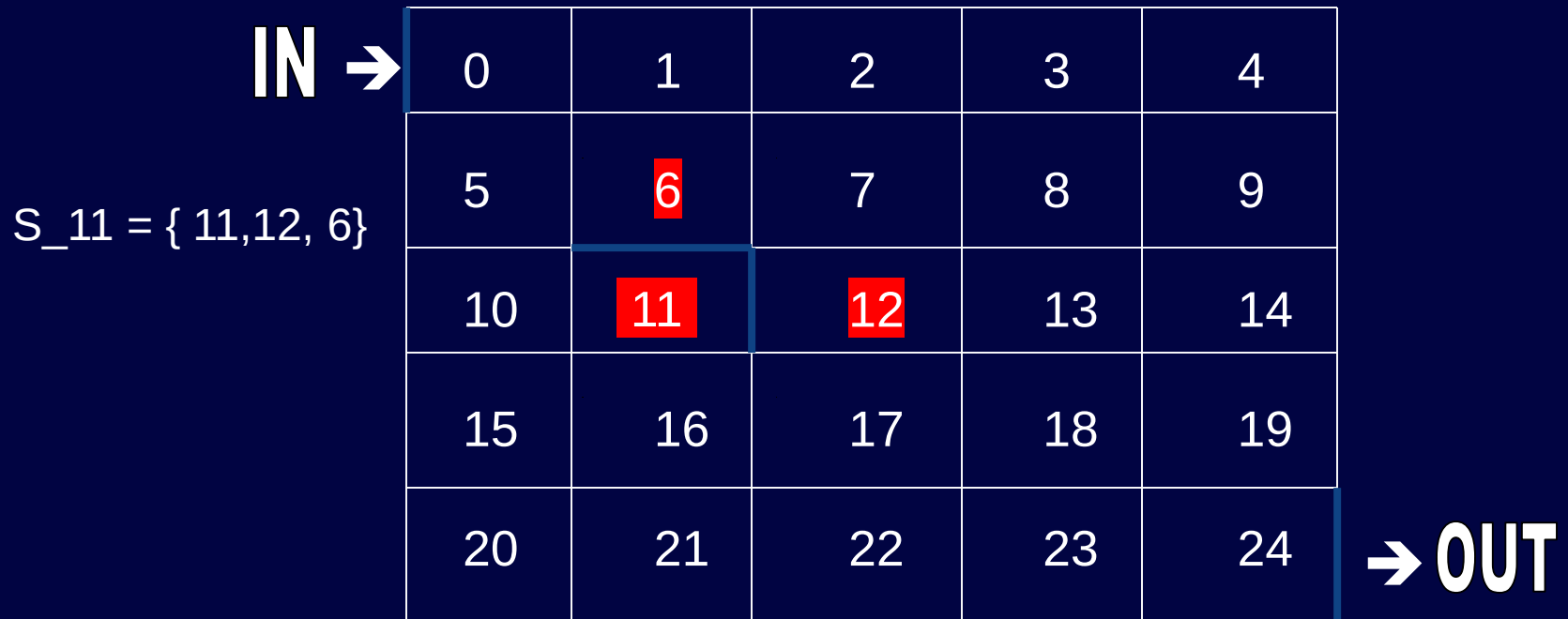
# Maze Generator

Cell 6, bottom wall chosen randomly



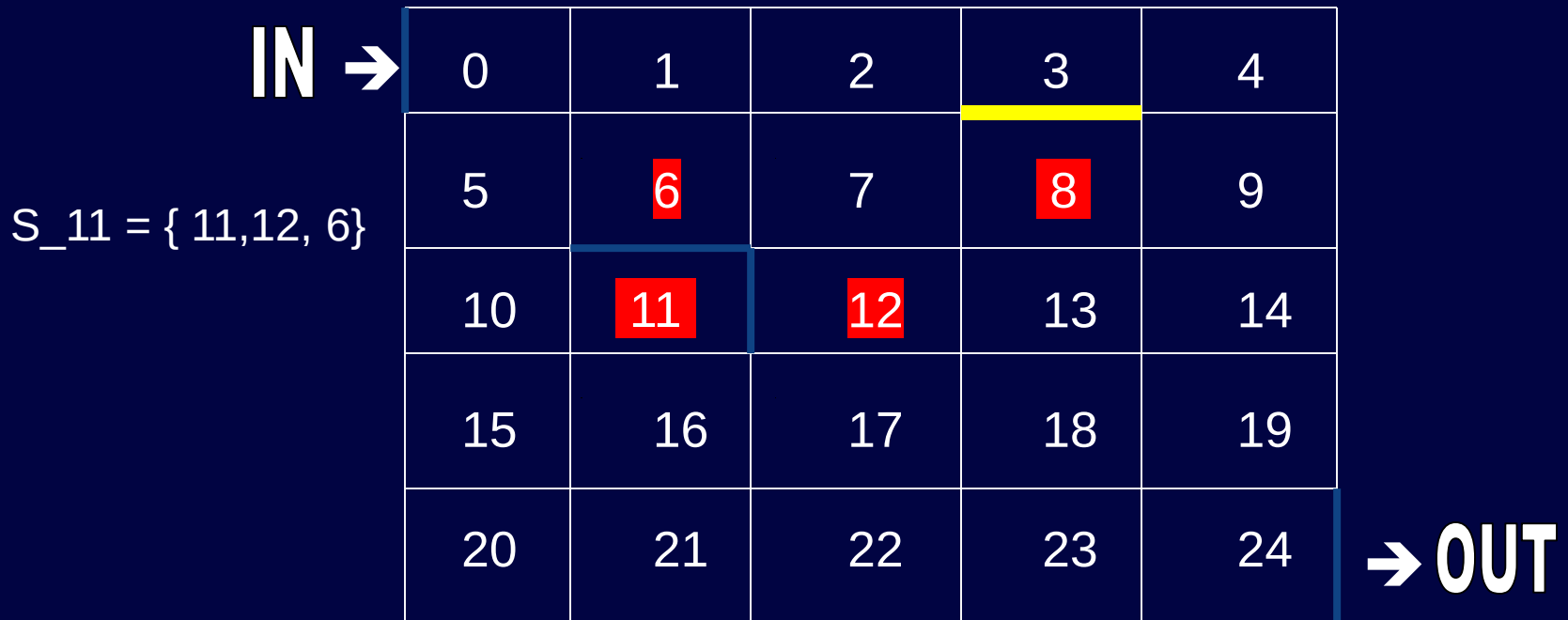
# Maze Generator

Cell 6, bottom wall chosen randomly



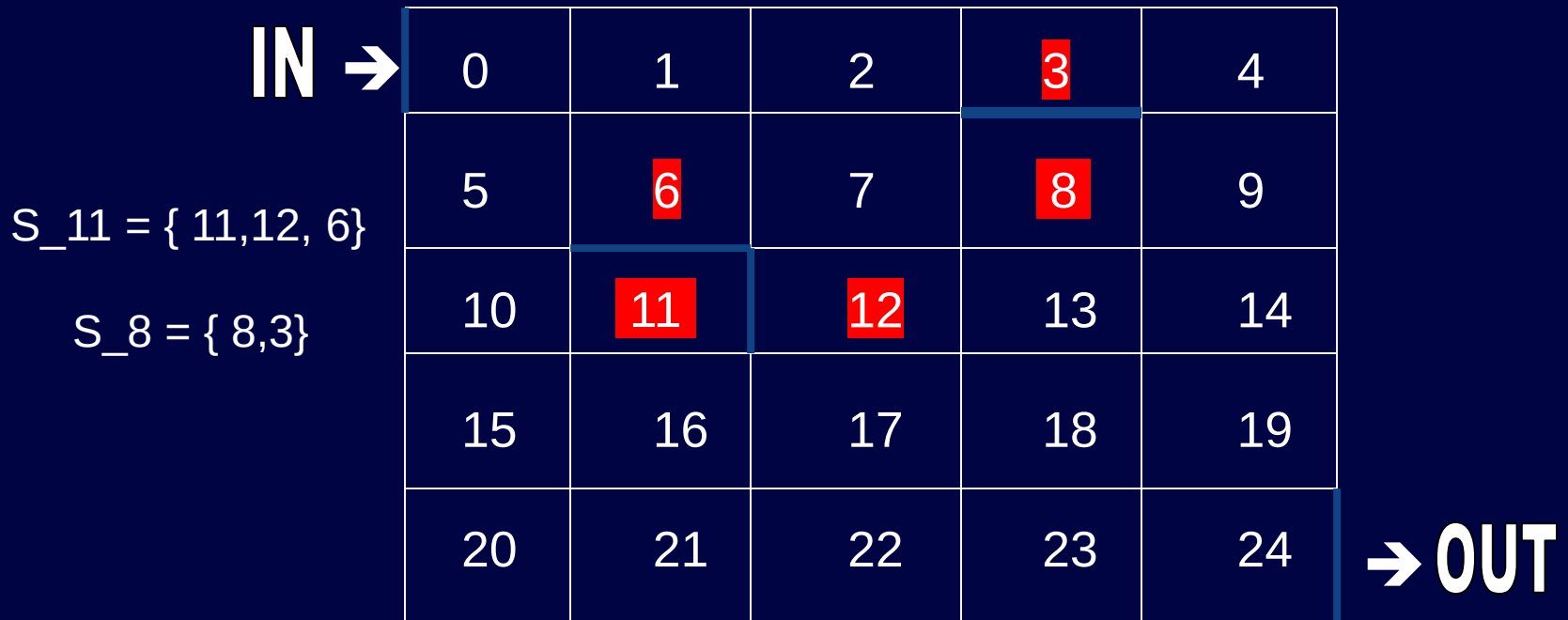
# Maze Generator

Cell 8, top wall chosen randomly



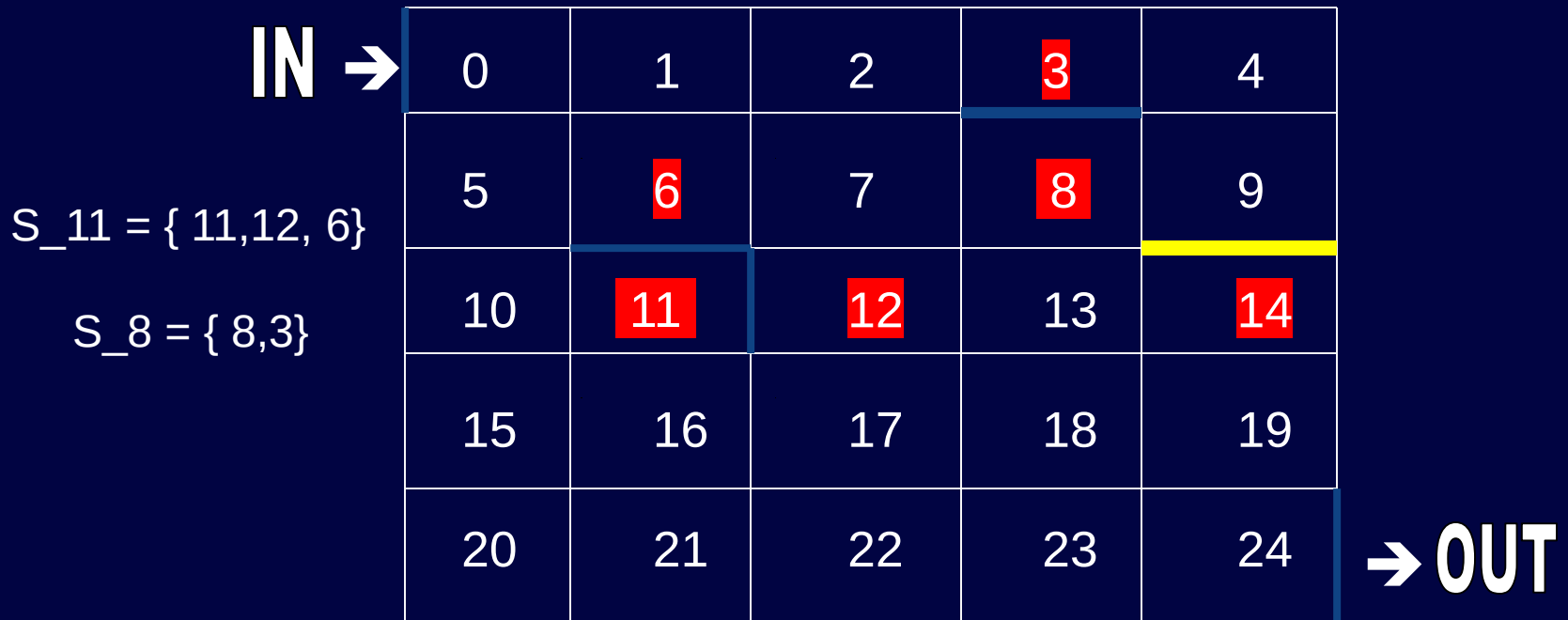
# Maze Generator

Cell 8, top wall chosen randomly



# Maze Generator

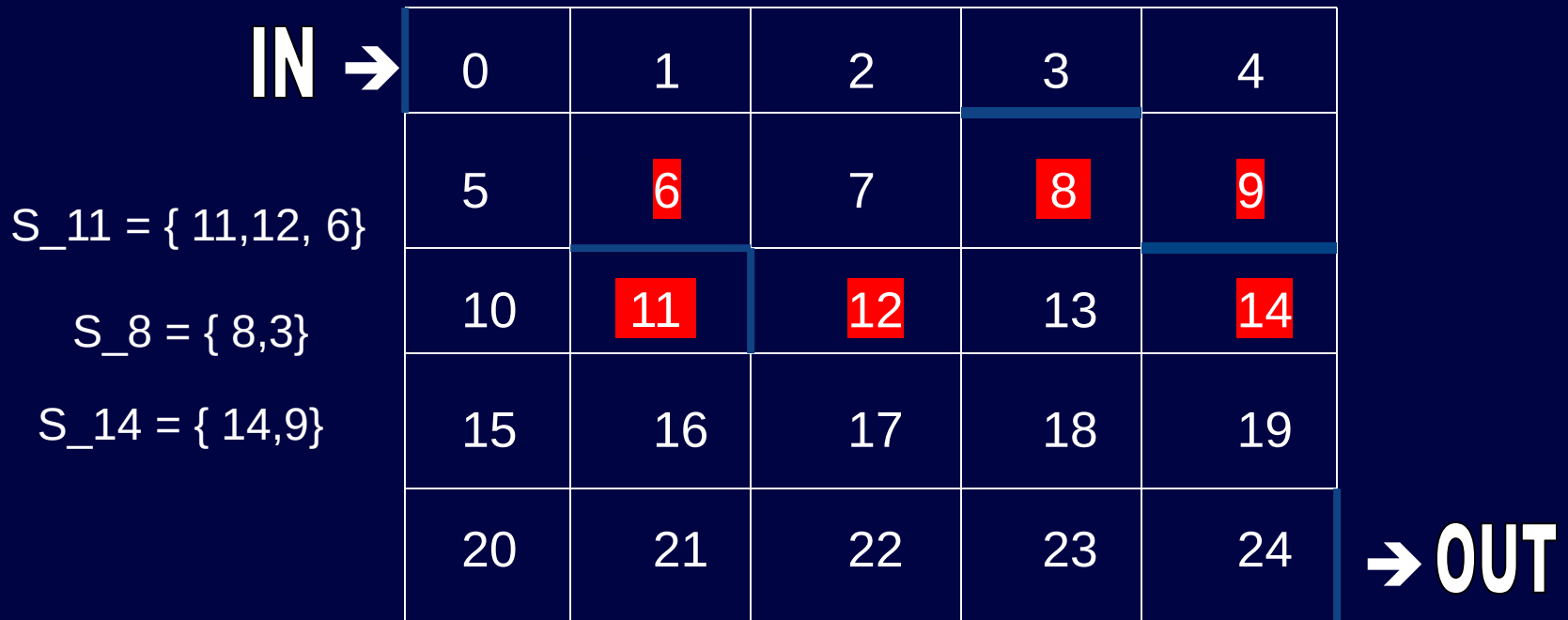
Cell 14, top wall chosen randomly





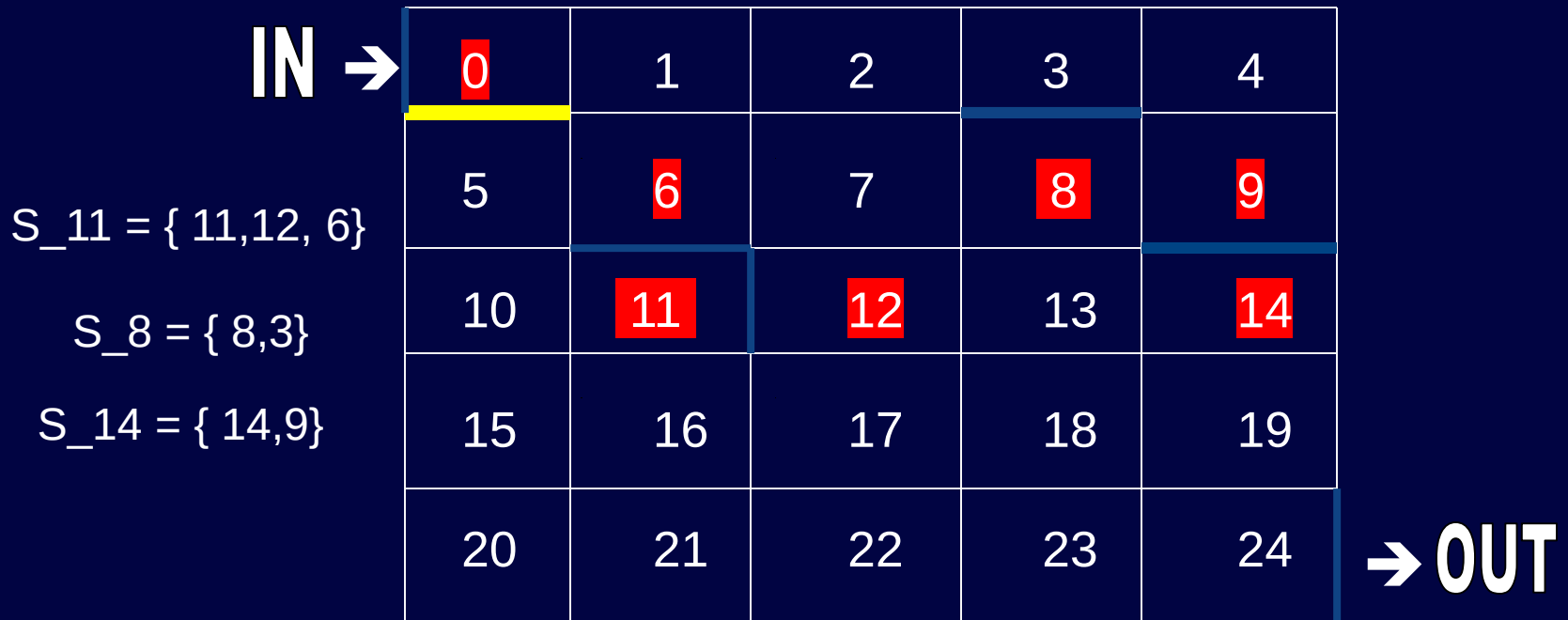
# Maze Generator

Cell 14, top wall chosen randomly



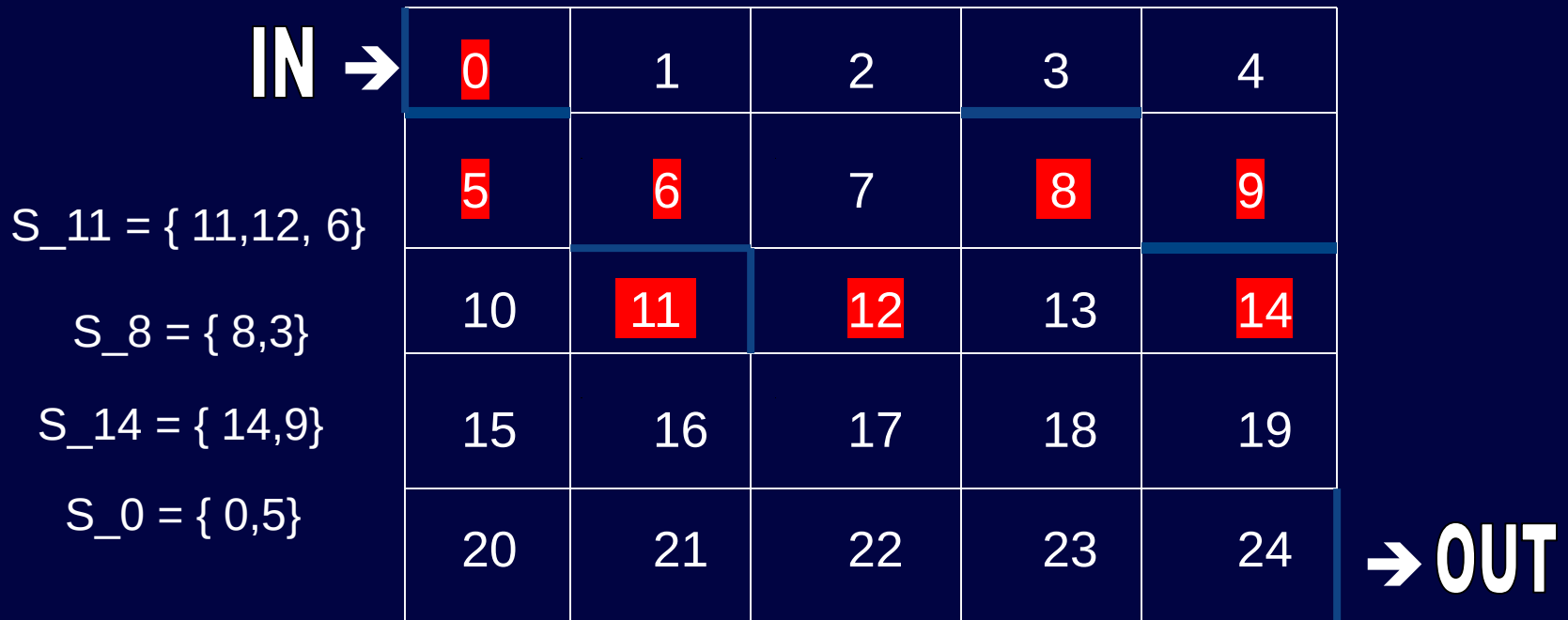
# Maze Generator

Cell 0, bottom wall chosen randomly



# Maze Generator

Cell 0, bottom wall chosen randomly



# Tables and Dictionaries

# Tables: rows & columns of information

- A *table* has several *fields* (types of information)
  - A telephone book may have fields *name*, *address*, *phone number*
  - A user account table may have fields *user id*, *password*, *home folder*

| Name          | Address                               | Phone    |
|---------------|---------------------------------------|----------|
| Sohail Aslam  | 50 Zahoor Elahi Rd, Gulberg-4, Lahore | 576-3205 |
| Imran Ahmad   | 30-T Phase-IV, LCCHS, Lahore          | 572-4409 |
| Salman Akhtar | 131-D Model Town, Lahore              | 784-3753 |

# Tables: rows & columns of information

- To find an *entry* in the table, you only need know the contents of one of the fields (not all of them).
- This field is the *key*
  - In a telephone book, the key is usually “name”
  - In a user account table, the key is usually “user id”

# Tables: rows & columns of information

- Ideally, a key *uniquely identifies* an entry
  - If the key is “name” and no two entries in the telephone book have the same name, the key uniquely identifies the entries

| Name          | Address                               | Phone    |
|---------------|---------------------------------------|----------|
| Sohail Aslam  | 50 Zahoor Elahi Rd, Gulberg-4, Lahore | 576-3205 |
| Imran Ahmad   | 30-T Phase-IV, LCCHS, Lahore          | 572-4409 |
| Salman Akhtar | 131-D Model Town, Lahore              | 784-3753 |

# The Table ADT: operations

- insert: given a key and an entry, inserts the entry into the table
- find: given a key, finds the entry associated with the key
- remove: given a key, finds the entry associated with the key, *and* removes it



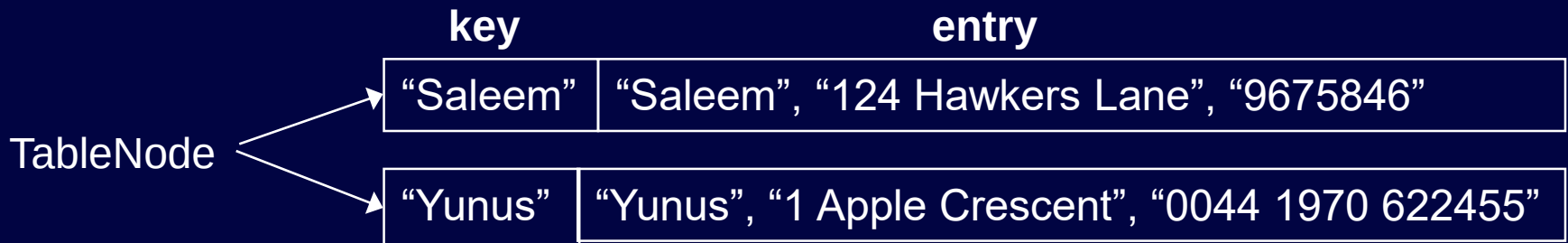
# How should we implement a table?

*Our choice of representation for the Table ADT depends on the answers to the following*

- How often are entries inserted and removed?
- How many of the possible key values are likely to be used?
- What is the likely pattern of searching for keys?  
E.g. Will most of the accesses be to just one or two key values?
- Is the table small enough to fit into memory?
- How long will the table exist?

# TableNode: a key and its entry

- 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)



# Implementation 1: unsorted sequential array

- An array in which TableNodes are stored consecutively in *any* order
- insert: add to back of array;  
(1)
- find: search through the keys one at a time, potentially all of the keys; ( $n$ )
- remove: find + replace removed node with last node; ( $n$ )

|   | key       | entry |
|---|-----------|-------|
| 0 |           |       |
| 1 |           |       |
| 2 |           |       |
| 3 |           |       |
| ⋮ | and so on |       |

# Implementation 2:sorted sequential array

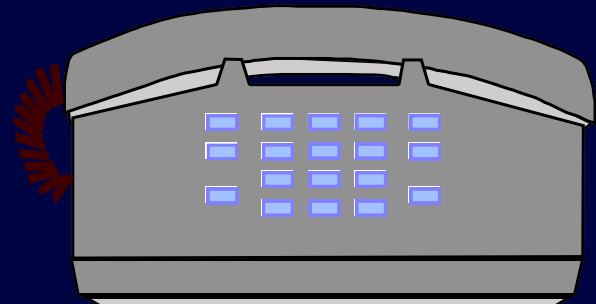
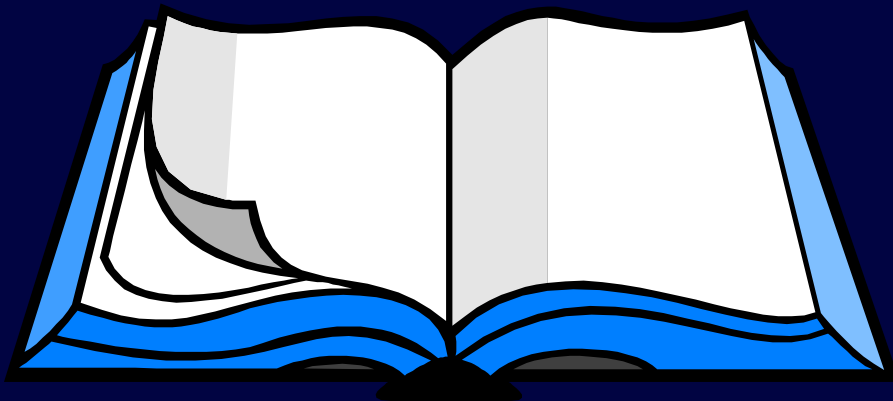
- An array in which TableNodes are stored consecutively, *sorted* by key
- insert: add in sorted order; ( $n$ )
- find: binary search; ( $\log n$ )
- remove: find, remove node and shuffle down; ( $n$ )

We can use binary search because the array elements are sorted

|   | key       | entry |
|---|-----------|-------|
| 0 |           |       |
| 1 |           |       |
| 2 |           |       |
| 3 |           |       |
| ⋮ | and so on |       |

# Searching an Array: Binary Search

- Binary search is like looking up a phone number or a word in the dictionary
  - Start in middle of book
  - If name you're looking for comes before names on page, look in first half
  - Otherwise, look in second half



# Binary Search

*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*

# Binary Search

**Case 1:**  $\text{val} == \text{a}[\text{mid}]$

$\text{val} = 10$

$\text{low} = 0, \text{high} = 8$

$\text{mid} = (0 + 8) / 2 = 4$



# Binary Search -- Example 2

**Case 2:**  $val > a[mid]$

$val = 19$

$low = 0, high = 8$

$mid = (0 + 8) / 2 = 4$

$new\ low = mid + 1 = 5$





# Binary Search -- Example 3

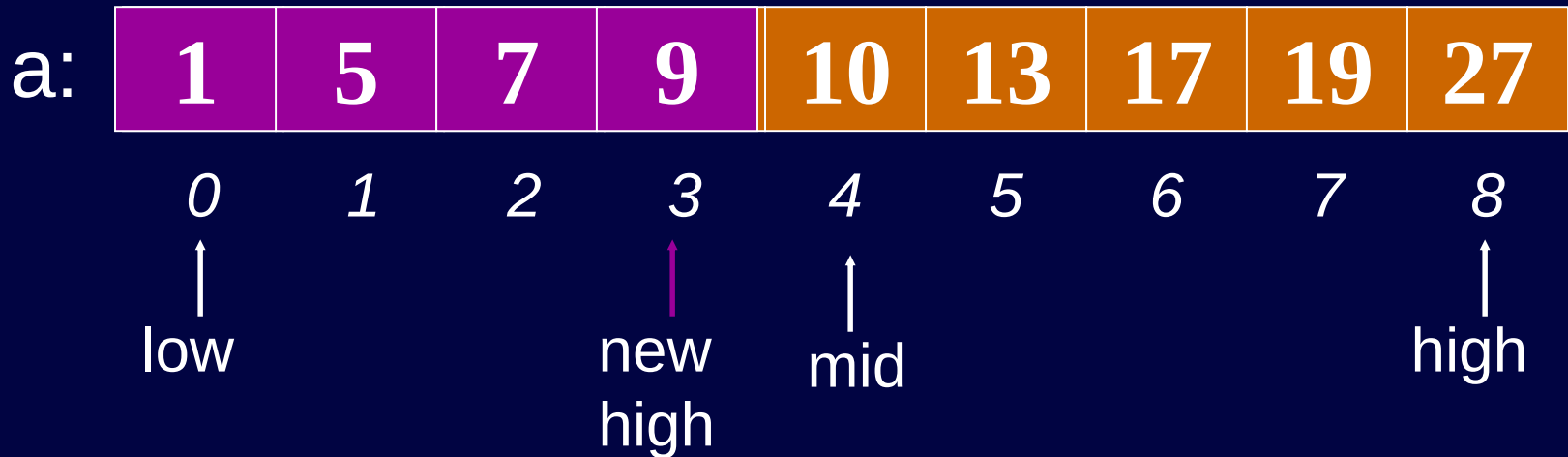
**Case 3:**  $val < a[mid]$

$val = 7$

$low = 0, high = 8$

$mid = (0 + 8) / 2 = 4$

$new\ high = mid - 1 = 3$



# Binary Search -- Example 3 (cont)

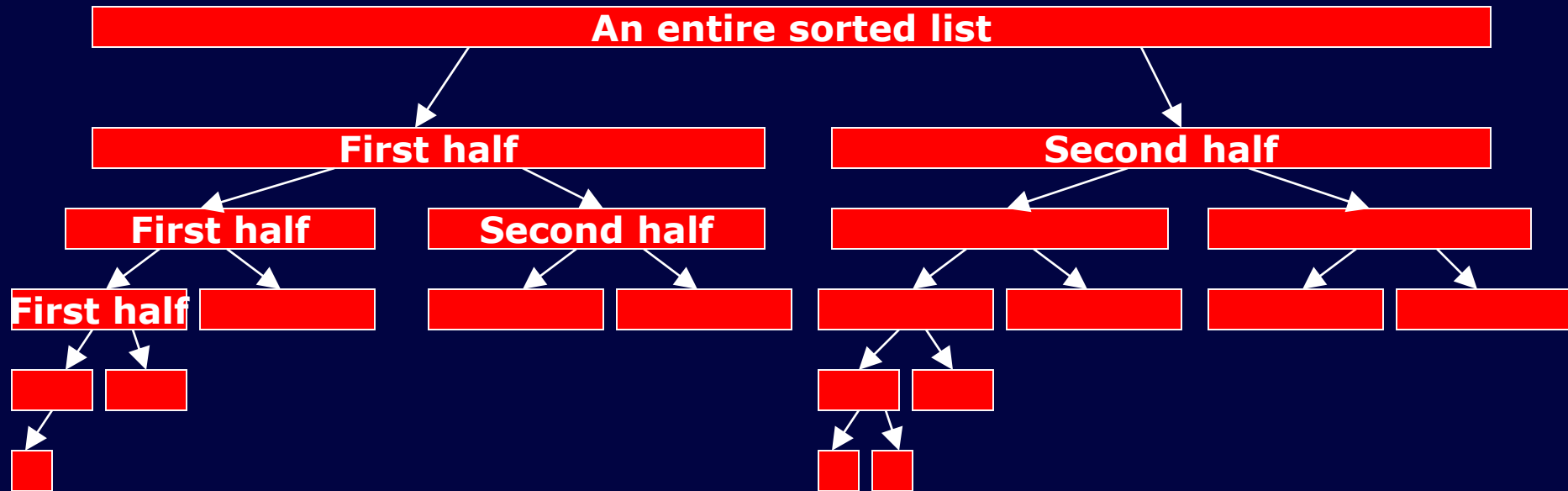
val = 7



# 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
}
```

# Binary Search: binary tree



- The search divides a list into two small sub-lists till a sub-list is no more divisible.

# Binary Search Efficiency

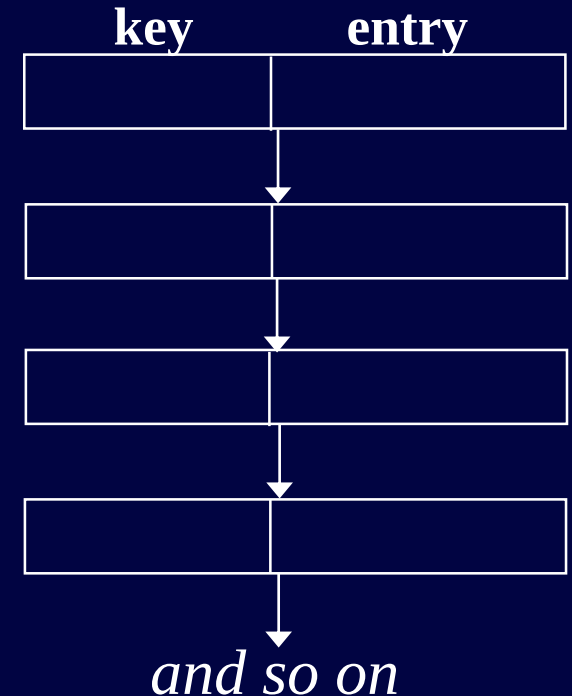
- After 1 bisection  $N/2$  items
- After 2 bisections  $N/4 = N/2^2$  items
- . . .
- After  $i$  bisections  $N/2^i = 1$  item

---

$$i = \log_2 N$$

# Implementation 3: linked list

- TableNodes are again stored consecutively (unsorted or sorted)
- insert: add to front; (*1 or  $n$  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; ( *$n$* )

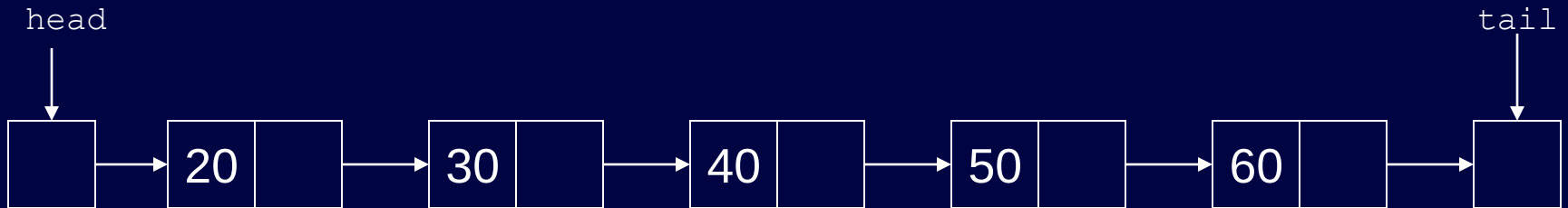


# Implementation 4: Skip List

- 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.

# Skip List Representation

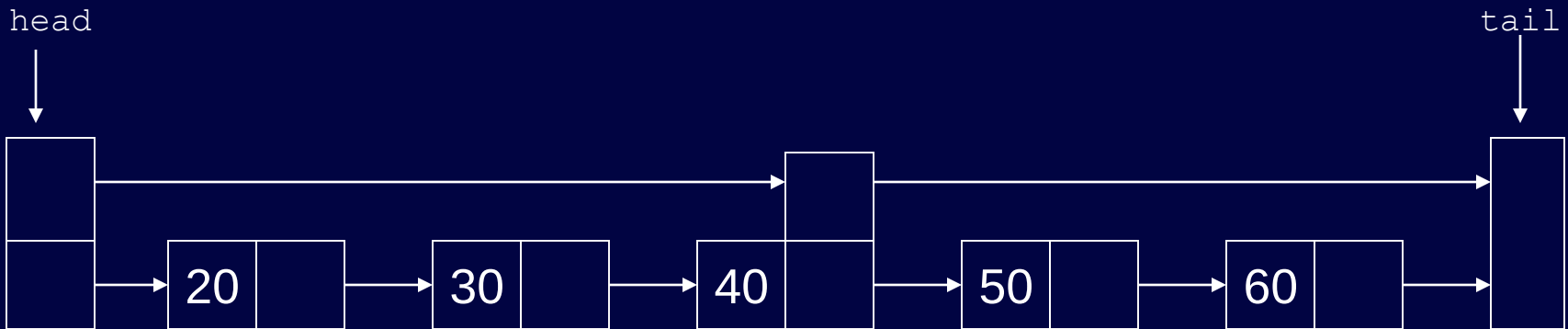
- Can do better than  $n$  comparisons to find element in chain of length  $n$



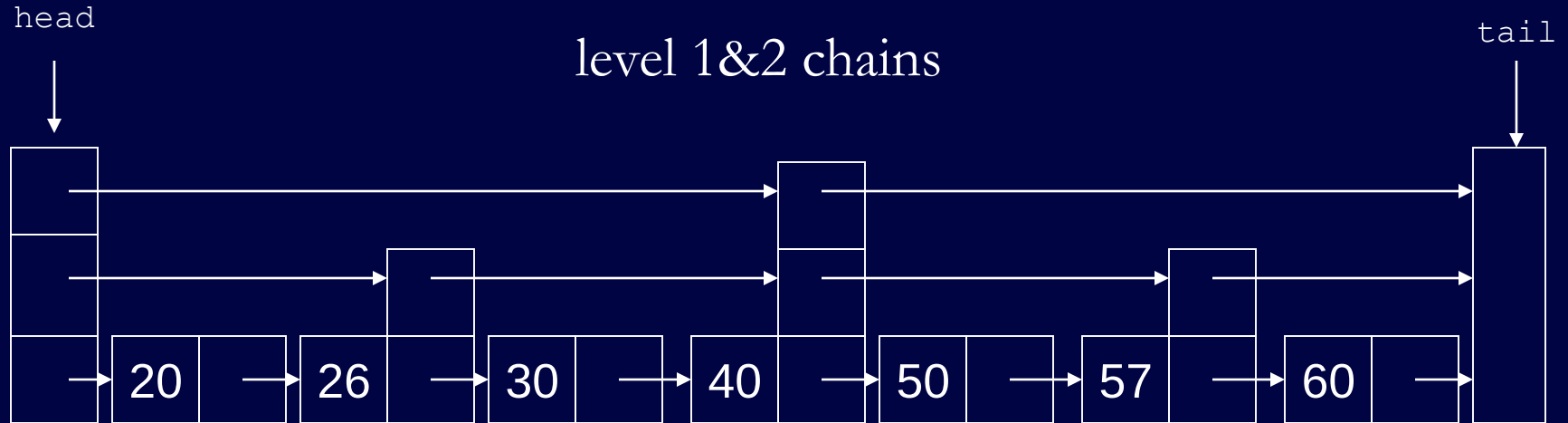


# Skip List Representation

- Example:  $n/2 + 1$  if we keep pointer to middle element

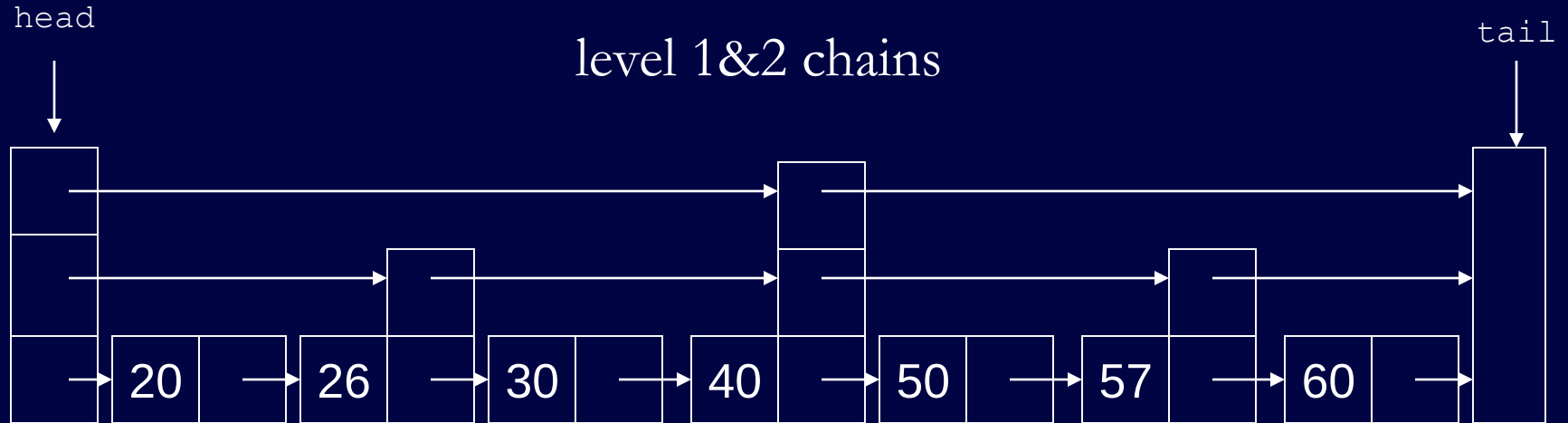


# Higher Level Chains



- 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 element

# Higher Level Chains



- Skip list contains a hierarchy of chains
- In general level  $i$  contains a subset of elements in level  $i-1$

# Skip List: formally

A skip list for a set  $S$  of distinct (key, element) items is a series of lists  $S_0, S_1, \dots, S_h$  such that

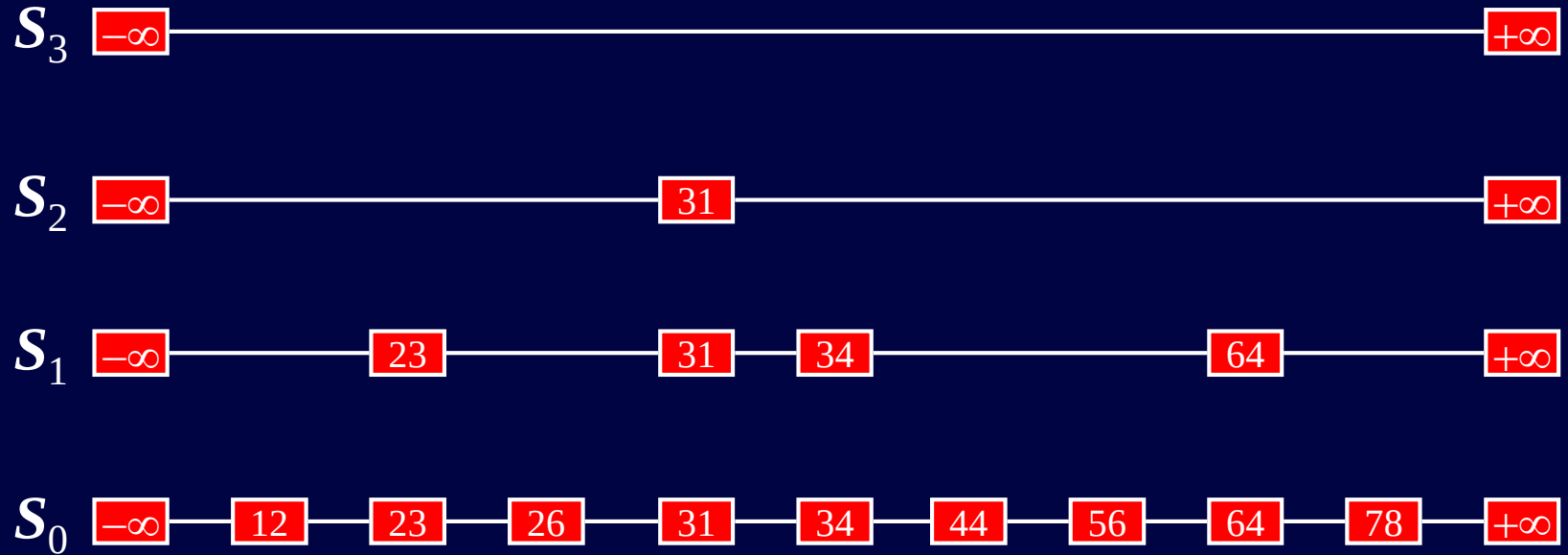
- Each list  $S_i$  contains the special keys  $+\infty$  and  $-\infty$
- List  $S_0$  contains the keys of  $S$  in nondecreasing order
- Each list is a subsequence of the previous one, i.e.,
$$S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$$
- List  $S_h$  contains only the two special keys

Lecture No.38

Data Structure

Dr. Sohail Aslam

# Skip List: formally



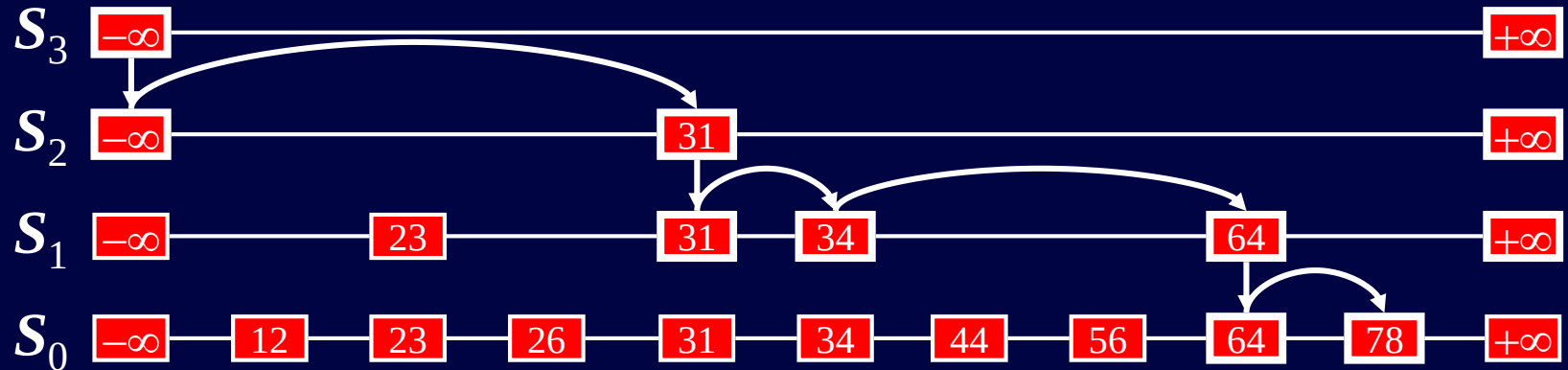
# Skip List: Search

We search for a key  $x$  as follows:

- We start at the first position of the top list
- At the current position  $p$ , we compare  $x$  with  $y \leftarrow \text{key}(\text{after}(p))$
- $x = y$ : we return  $\text{element}(\text{after}(p))$
- $x > y$ : we “scan forward”
- $x < y$ : we “drop down”
- If we try to drop down past the bottom list, we return *NO\_SUCH\_KEY*

# Skip List: Search

Example: search for 78





# Skip List: Insertion

To insert an item  $(x, o)$  into a skip list, we use a randomized algorithm:

- We repeatedly toss a coin until we get tails, and we denote with  $i$  the number of times the coin came up heads
- If  $i \geq h$ , we add to the skip list new lists  $S_{h+1}, \dots, S_{i+1}$ , each containing only the two special keys

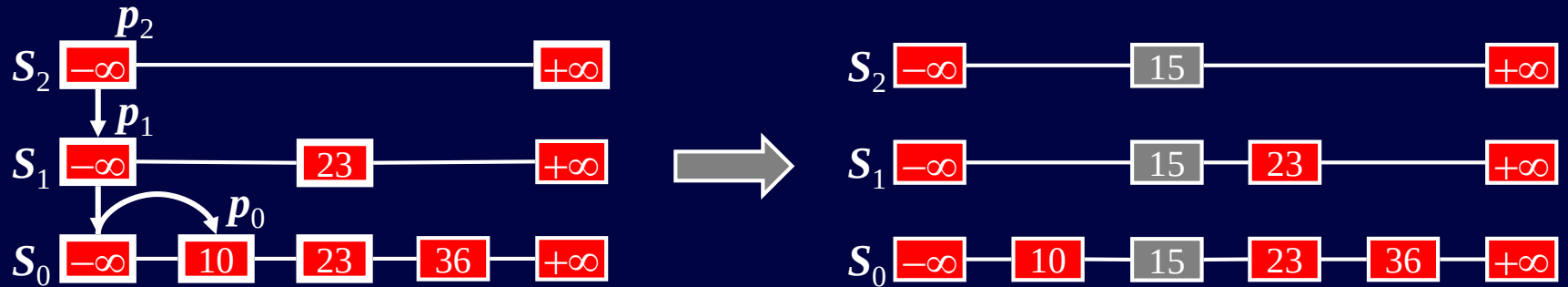
# Skip List: Insertion

To insert an item  $(x, o)$  into a skip list, we use a randomized algorithm: (cont)

- We search for  $x$  in the skip list and find the positions  $p_0, p_1, \dots, p_i$  of the items with largest key less than  $x$  in each list  $S_0, S_1, \dots, S_i$
- For  $j \leftarrow 0, \dots, i$ , we insert item  $(x, o)$  into list  $S_j$  after position  $p_j$

# Skip List: Insertion

- Example: insert key 15, with  $i = 2$



# Randomized Algorithms

- 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 outcomes of the coin tosses, i.e, head or tail

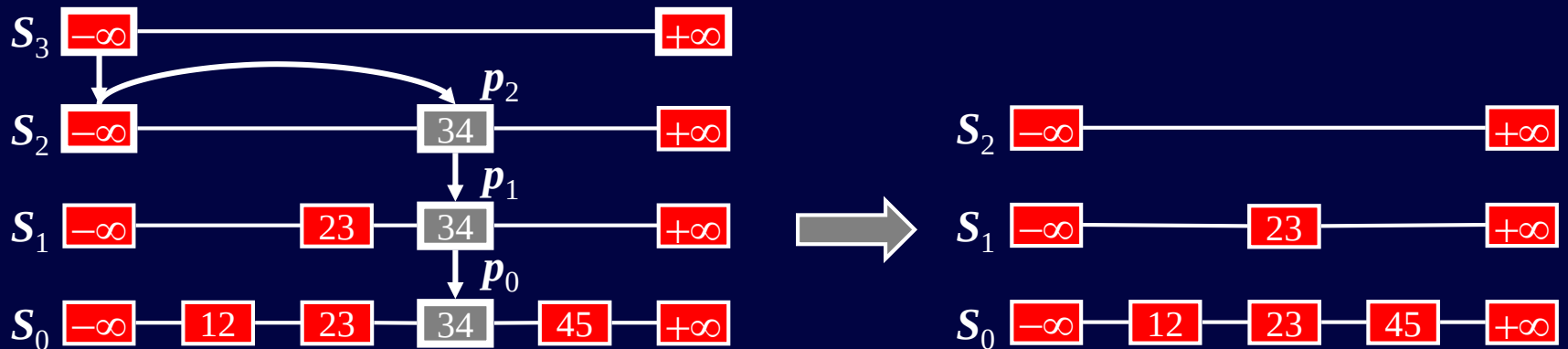
# Skip List: Deletion

To remove an item with key  $x$  from a skip list, we proceed as follows:

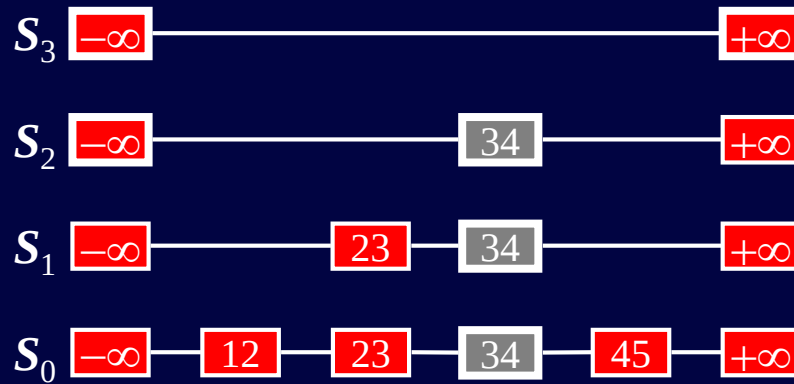
- We search for  $x$  in the skip list and find the positions  $p_0, p_1, \dots, p_i$  of the items with key  $x$ , where position  $p_j$  is in list  $S_j$
- We remove positions  $p_0, p_1, \dots, p_i$  from the lists  $S_0, S_1, \dots, S_i$
- We remove all but one list containing only the two special keys

# Skip List: Deletion

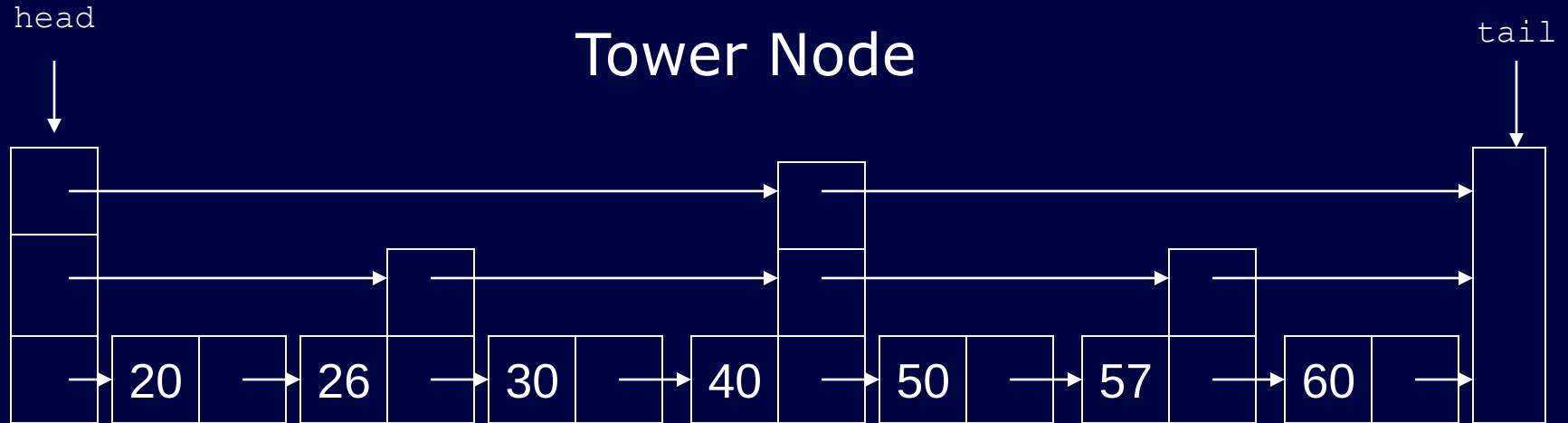
- Example: remove key 34



# Skip List: Implementation



# Implementation: TowerNode

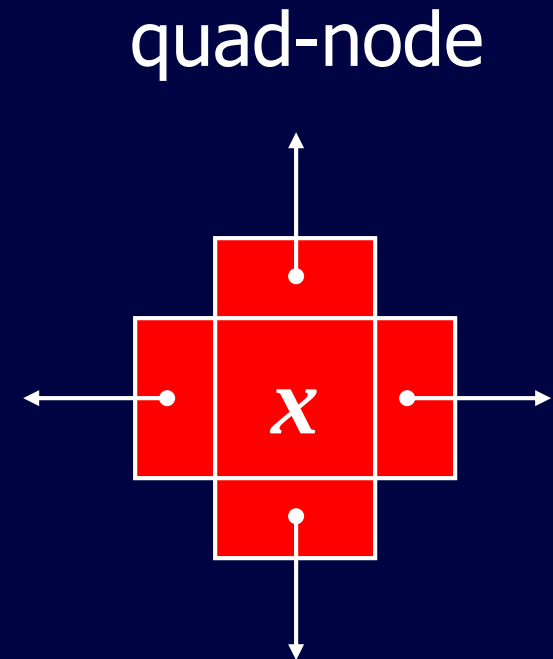


- TowerNode will have array of next pointers.
- Actual number of next pointers will be decided by the random procedure.
- Define MAXLEVEL as an upper limit on number of levels in a node.

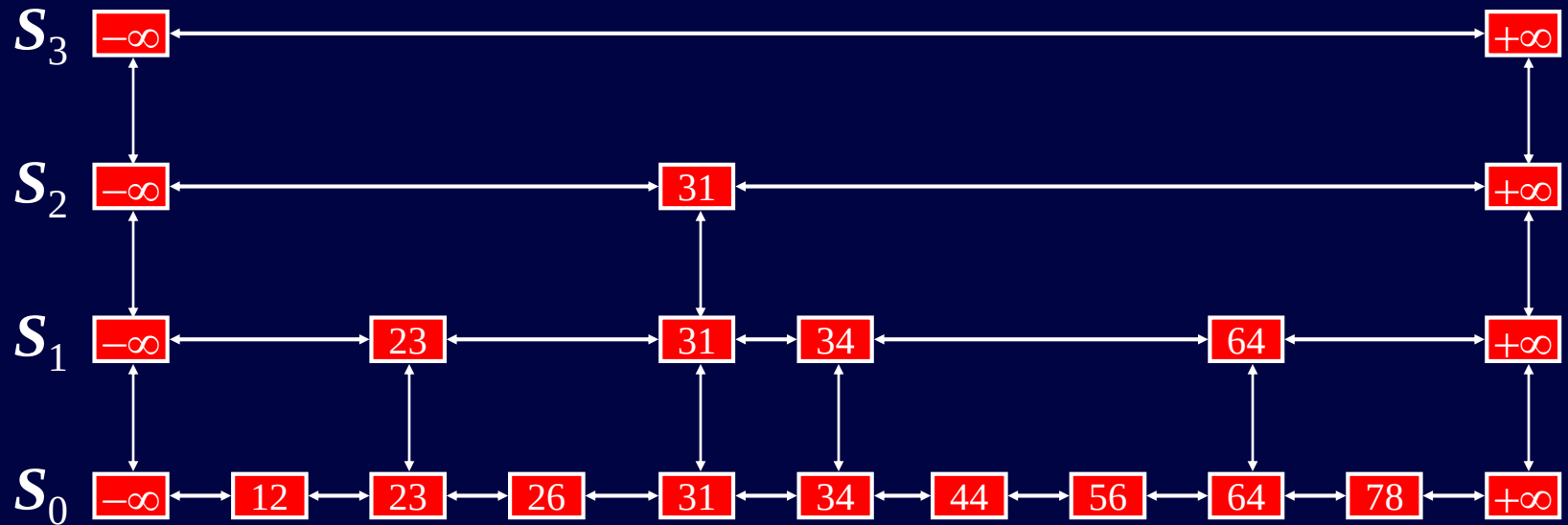


# Implementation: QuadNode

- 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 the key (jitem) at different levels



# Skip Lists with Quad Nodes

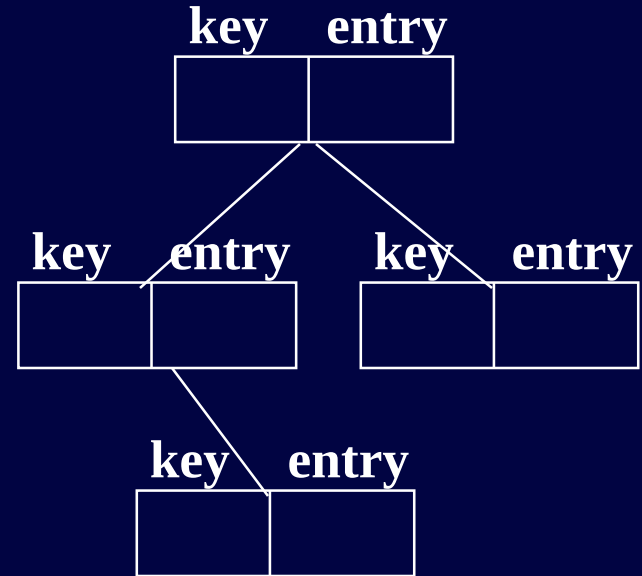


# Performance of Skip Lists

- In a skip list with  $n$  items
  - The expected space used is proportional to  $n$ .
  - The expected search, insertion and deletion time is proportional to  $\log n$ .
- Skip lists are fast and simple to implement in practice

# Implementation 5: AVL tree

- An AVL tree, ordered by key
- insert: a standard insert; ( $\log n$ )
- find: a standard find (without removing, of course); ( $\log n$ )
- remove: a standard remove; ( $\log n$ )



*and so on*

# Anything better?

- So far we have find, remove and insert where time varies between constant  $\log n$ .
- It would be nice to have all three as constant time operations!

# Implementation 6: Hashing

- An *array* in which TableNodes are **not** stored consecutively
- Their place of storage is calculated using the key and a *hash function*



- Keys and entries are scattered throughout the array.

|     | key | entry |
|-----|-----|-------|
|     |     |       |
| 4   |     |       |
|     |     |       |
| 10  |     |       |
|     |     |       |
| 123 |     |       |
|     |     |       |

# Hashing

- insert: calculate place of storage, insert TableNode; (1)
- find: calculate place of storage, retrieve entry; (1)
- remove: calculate place of storage, set it to null; (1)

All are constant time (1) !

|     | key | entry |
|-----|-----|-------|
|     |     |       |
| 4   |     |       |
|     |     |       |
| 10  |     |       |
|     |     |       |
| 123 |     |       |
|     |     |       |

# Hashing

- We use an array of some fixed size  $T$  to hold the data.  $T$  is typically prime.
- Each key is mapped into some number in the range  $0$  to  $T-1$  using a *hash function*, which ideally should be efficient to compute.



# Example: fruits

- 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`

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |

# Example

- Store data in a table array:

```
table[5] = "apple"  
table[3] = "watermelon"  
table[8] = "grapes"  
table[7] = "cantaloupe"  
table[0] = "kiwi"  
table[9] = "strawberry"  
table[6] = "mango"  
table[2] = "banana"
```

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |

# Example

- Associative array:

```
table["apple"]  
table["watermelon"]  
table["grapes"]  
table["cantaloupe"]  
table["kiwi"]  
table["strawberry"]  
table["mango"]  
table["banana"]
```

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |

# Example Hash Functions

- 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:

$$h(str) = \left( \sum_{i=0}^{length-1} str[i] \right) \% TableSize$$

$$Example : h(ABC) = (65 + 66 + 67) \% TableSize$$

# Finding the hash 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;
}
```

# Example Hash Functions

- Another possibility is to convert the string into some number in some arbitrary base  $b$  ( $b$  also might be a prime number):

$$h(str) = \left( \sum_{i=0}^{length-1} str[i] \times b^i \right) \% T$$

$$Example : h(ABC) = (65b^0 + 66b^1 + 67b^2) \% T$$

# Example Hash Functions

- If the keys are integers then  $key \% T$  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.
- In general, to avoid situations like this,  $T$  should be a prime number.

# Collision

Suppose our hash function gave us the following values:

- $\text{hash}(\text{"apple"}) = 5$   
 $\text{hash}(\text{"watermelon"}) = 3$   
 $\text{hash}(\text{"grapes"}) = 8$   
 $\text{hash}(\text{"cantaloupe"}) = 7$   
 $\text{hash}(\text{"kiwi"}) = 0$   
 $\text{hash}(\text{"strawberry"}) = 9$   
 $\text{hash}(\text{"mango"}) = 6$   
 $\text{hash}(\text{"banana"}) = 2$

**$\text{hash}(\text{"honeydew"}) = 6$**

- Now what?

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |



# Collision

- When two values hash to the same array location, this is called a *collision*
- Collisions are normally treated as “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 this same location.

# Solution for Handling collisions

- **Solution #1:** Search from there for an empty location
  - Can stop searching when we find the value *or* an empty location.
  - Search must be wrap-around at the end.

# Solution for Handling collisions

- **Solution #2:** Use a second hash function
  - ...and a third, and a fourth, and a fifth, ...

# Solution for Handling collisions

- **Solution #3:** Use the array location as the header of a linked list of values that hash to this location

# Solution 1: Open Addressing

- This approach of handling collisions is called *open addressing*; it is also known as *closed hashing*.
- More formally, cells at  $h_0(x)$ ,  $h_1(x)$ ,  $h_2(x)$ , ... are tried in succession where

$$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{TableSize},$$

with  $f(0) = 0$ .

- The function,  $f$ , is the collision resolution strategy.

# Linear Probing

- We use  $f(i) = i$ , i.e.,  $f$  is a linear function of  $i$ . Thus

$$location(x) = (hash(x) + i) \bmod TableSize$$

- The collision resolution strategy is called *linear probing* because it scans the array sequentially (with wrap around) in search of an empty cell.

# Linear Probing: insert

- Suppose we want to *add seagull* to this hash table
- Also suppose:
  - `hashCode("seagull") = 143`
  - `table[143]` is not empty
  - `table[143] != seagull`
  - `table[144]` is not empty
  - `table[144] != seagull`
  - `table[145]` is empty
- Therefore, put *seagull* at location 145

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |
| ... |         |

# Linear Probing: insert

- Suppose you want to *add hawk* to this hash table
- Also suppose
  - `hashCode("hawk") = 143`
  - `table[143]` is not empty
  - `table[143] != hawk`
  - `table[144]` is not empty
  - `table[144] == hawk`
- *hawk* is already in the table, so do nothing.

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |
| ... |         |



# Linear Probing: insert

- Suppose:
  - You want to add **cardinal** to this hash table
  - **hashCode("cardinal") = 147**
  - The last location is 148
  - 147 and 148 are occupied
- Solution:
  - Treat the table as circular; after 148 comes 0
  - Hence, **cardinal** goes in location 0 (or 1, or 2, or ...)

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |

# Linear Probing: find

- Suppose we want to find **hawk** in this hash table
- We proceed as follows:
  - `hashCode("hawk") = 143`
  - `table[143]` is not empty
  - `table[143] != hawk`
  - `table[144]` is not empty
  - `table[144] == hawk` (found!)
- We use the same procedure for looking things up in the table as we do for inserting them

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |
| ... |         |

# Linear Probing and Deletion

- If an item is placed in `array[hash(key) + 4]`, then the item just before it is deleted
- How will probe determine that the “hole” does not indicate the item is not in the array?
- Have three states for each location
  - Occupied
  - Empty (never used)
  - Deleted (previously used)

# Clustering

- 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 it is that new values will hash into the cluster, and make it ever bigger.
- Clusters cause efficiency to degrade.

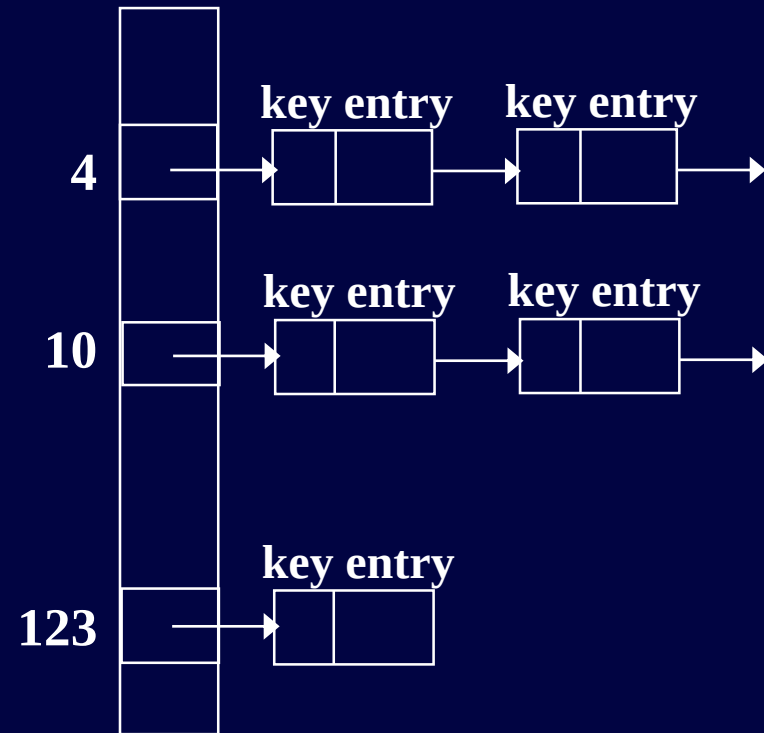
# Quadratic Probing

- Quadratic probing uses different formula:
  - Use  $F(i) = i^2$  to resolve collisions
  - If hash function resolves to  $H$  and a search in cell  $H$  is inconclusive, try  $H + 1^2$ ,  $H + 2^2$ ,  $H + 3^2$ , ...
- Probe
  - array[hash(key)+1<sup>2</sup>], then
  - array[hash(key)+2<sup>2</sup>], then
  - array[hash(key)+3<sup>2</sup>], and so on
  - Virtually eliminates primary clusters

# Collision resolution: chaining

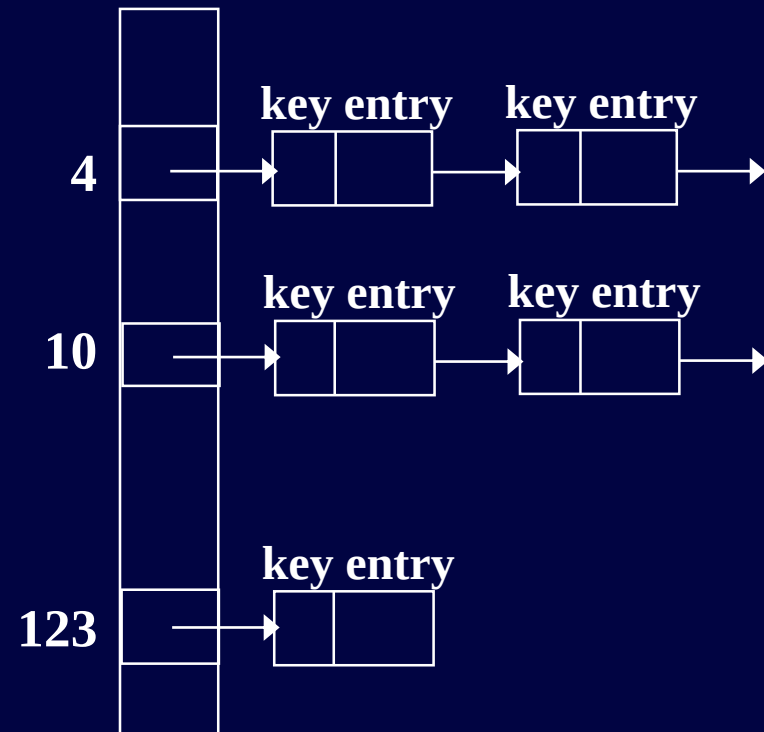
- Each table position is a linked list
- Add the keys and entries anywhere in the list (front easiest)

*No need to change position!*



# Collision resolution: chaining

- Advantages over open addressing:
  - Simpler insertion and removal
  - Array size is not a limitation
- Disadvantage
  - Memory overhead is large if entries are small.



# Applications of Hashing

- Compilers use hash tables to keep track of declared variables (symbol table).
- 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.



# Applications of Hashing

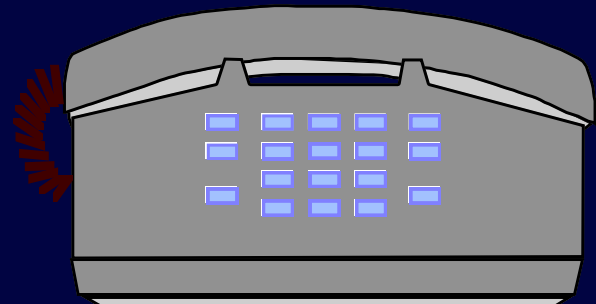
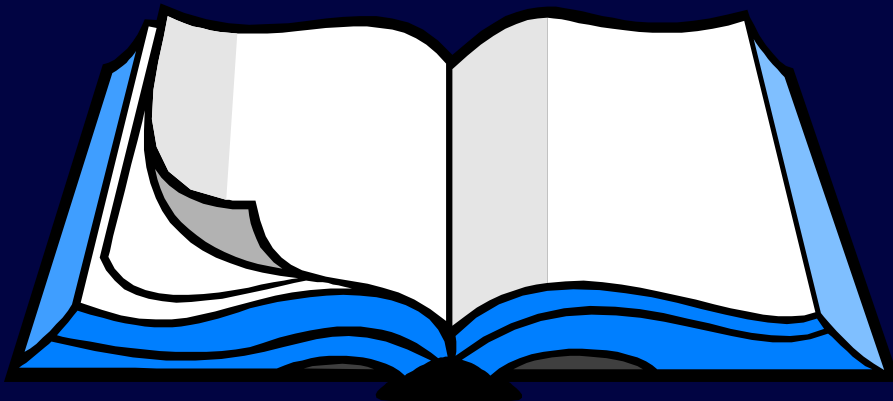
- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again.
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different.

# When is hashing suitable?

- Hash tables are very good if there is a need for many searches in a reasonably stable table.
- 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.
- Also, hashing is very slow for any operations which require the entries to be sorted
  - e.g. Find the minimum key

# Searching an Array: Binary Search

- Binary search is like looking up a phone number or a word in the dictionary
  - Start in middle of book
  - If name you're looking for comes before names on page, look in first half
  - Otherwise, look in second half



Lecture No.39

Data Structure

Dr. Sohail Aslam

# Binary Search

*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*

# Binary Search

**Case 1:**  $\text{val} == \text{a}[\text{mid}]$

$\text{val} = 10$

$\text{low} = 0, \text{high} = 8$

$\text{mid} = (0 + 8) / 2 = 4$



# Binary Search -- Example 2

**Case 2:**  $val > a[mid]$

$val = 19$

$low = 0, high = 8$

$mid = (0 + 8) / 2 = 4$

$new\ low = mid + 1 = 5$



# Binary Search -- Example 3

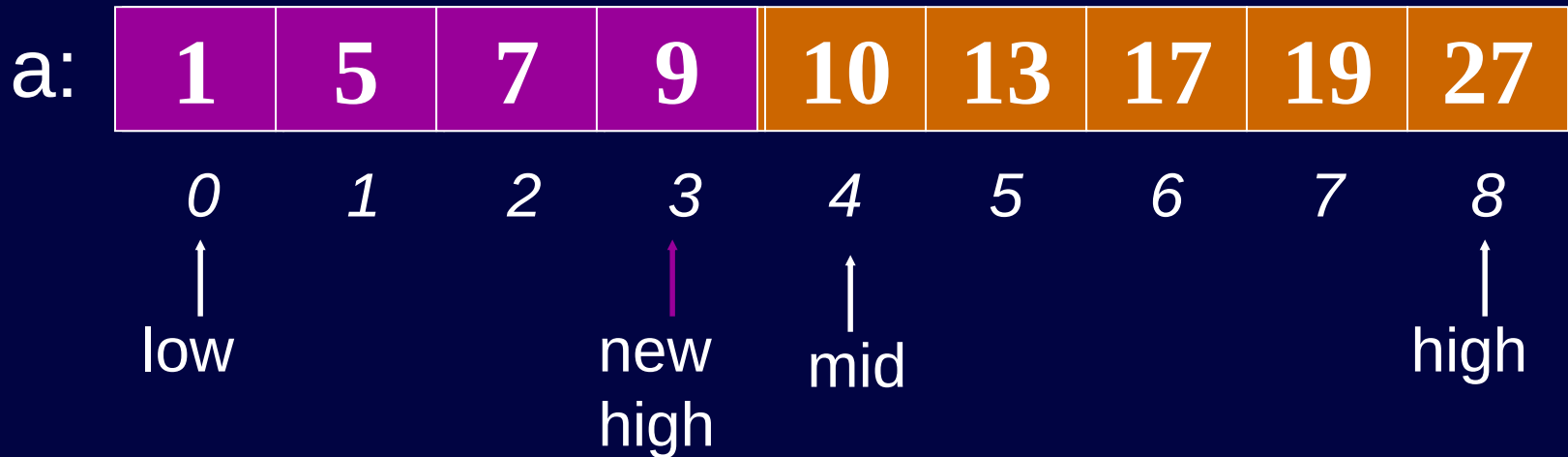
**Case 3:**  $val < a[mid]$

$val = 7$

$low = 0, high = 8$

$mid = (0 + 8) / 2 = 4$

$new\ high = mid - 1 = 3$





# Binary Search -- Example 3 (cont)

val = 7



a:

|   |   |   |   |    |    |    |    |    |
|---|---|---|---|----|----|----|----|----|
| 1 | 5 | 7 | 9 | 10 | 13 | 17 | 19 | 27 |
| 0 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  |



a:

|   |   |   |   |    |    |    |    |    |
|---|---|---|---|----|----|----|----|----|
| 1 | 5 | 7 | 9 | 10 | 13 | 17 | 19 | 27 |
| 0 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  |

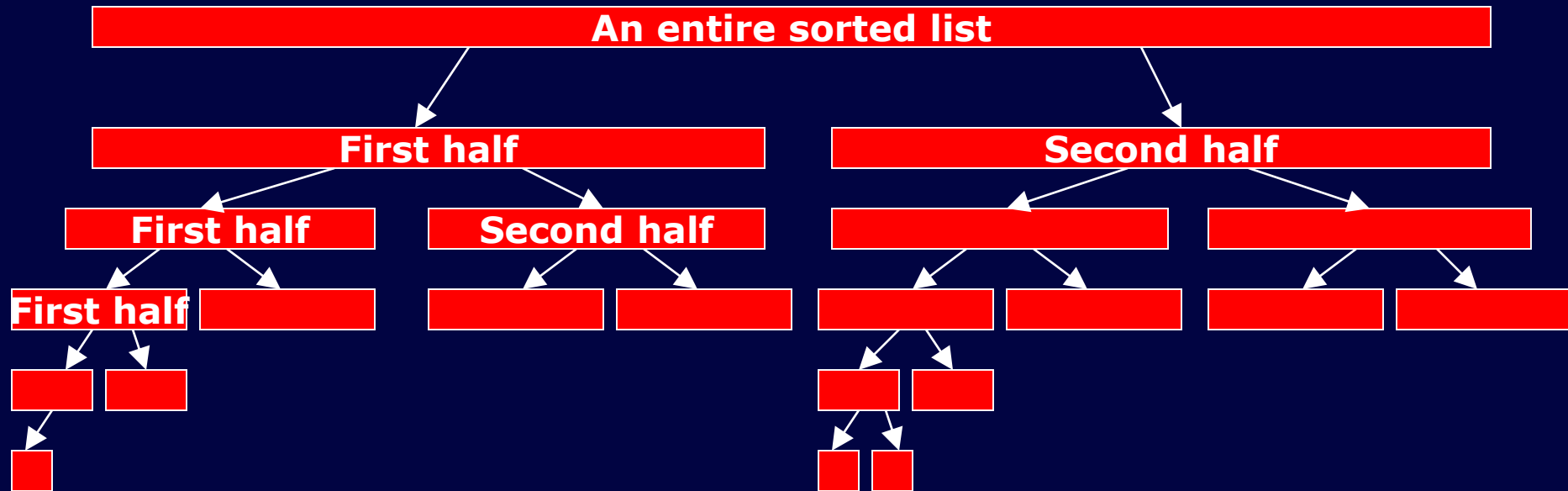
a:

|   |   |   |   |    |    |    |    |    |
|---|---|---|---|----|----|----|----|----|
| 1 | 5 | 7 | 9 | 10 | 13 | 17 | 19 | 27 |
| 0 | 1 | 2 | 3 | 4  | 5  | 6  | 7  | 8  |

# 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
}
```

# Binary Search: binary tree



- The search divides a list into two small sub-lists till a sub-list is no more divisible.

# Binary Search Efficiency

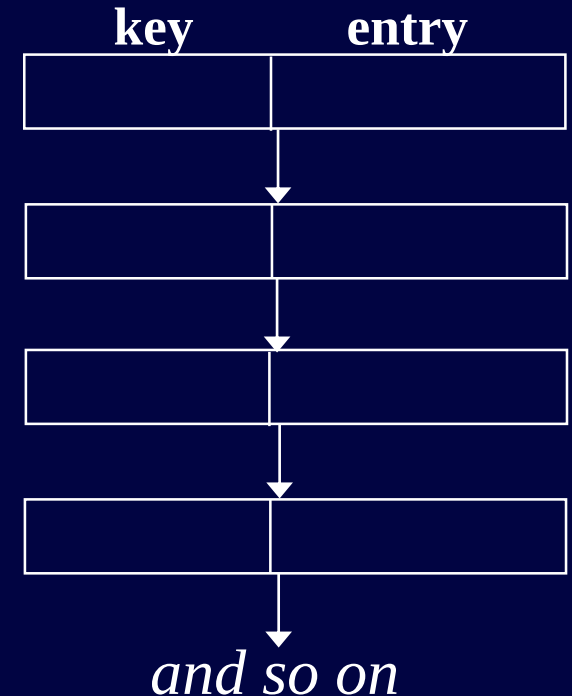
- After 1 bisection  $N/2$  items
- After 2 bisections  $N/4 = N/2^2$  items
- . . .
- After  $i$  bisections  $N/2^i = 1$  item

---

$$i = \log_2 N$$

# Implementation 3: linked list

- TableNodes are again stored consecutively (unsorted or sorted)
- insert: add to front; (*1 or  $n$  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; ( *$n$* )

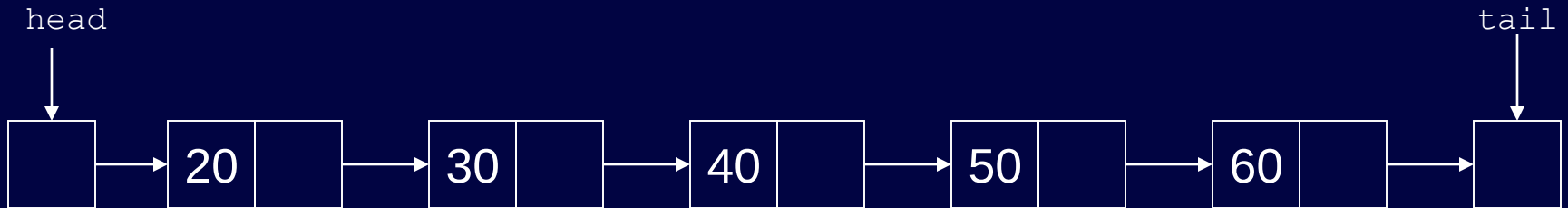


# Implementation 4: Skip List

- 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.

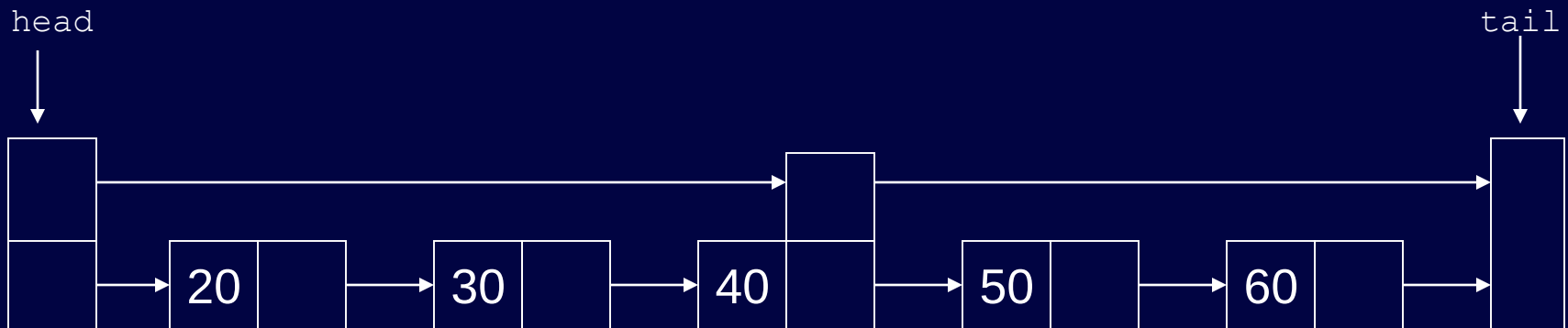
# Skip List Representation

- Can do better than  $n$  comparisons to find element in chain of length  $n$



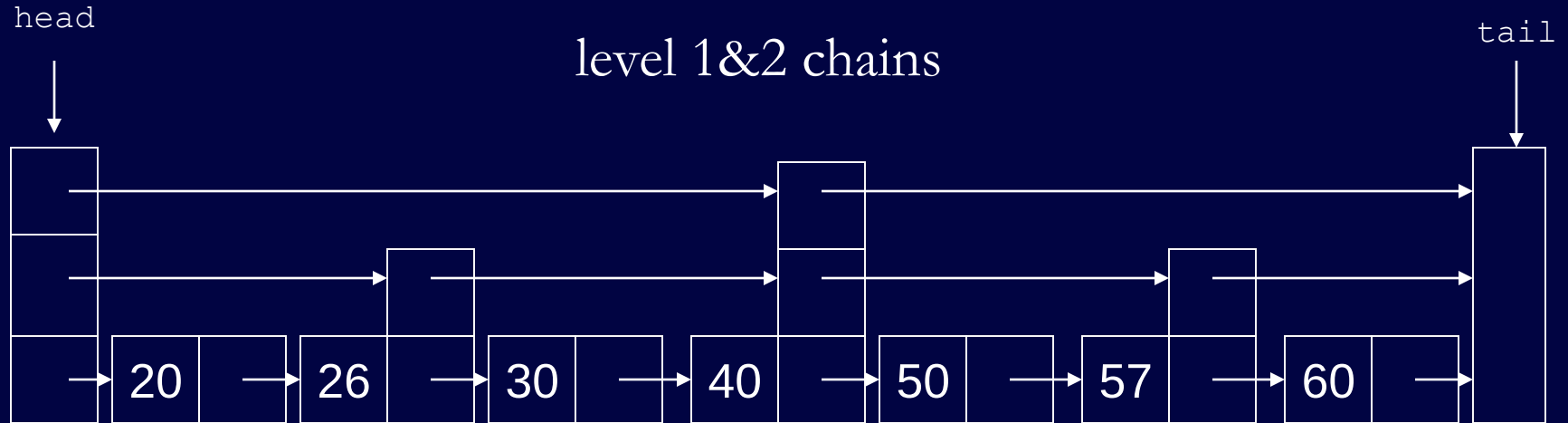
# Skip List Representation

- Example:  $n/2 + 1$  if we keep pointer to middle element



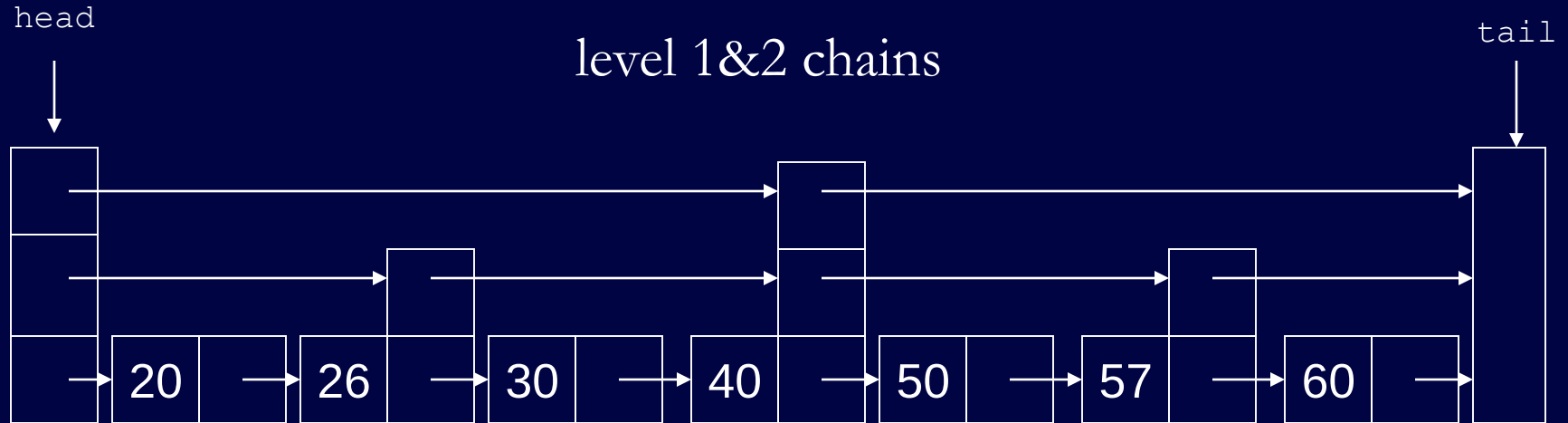


# Higher Level Chains



- 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 element

# Higher Level Chains



- Skip list contains a hierarchy of chains
- In general level  $i$  contains a subset of elements in level  $i-1$

# Skip List: formally

A skip list for a set  $S$  of distinct (key, element) items is a series of lists  $S_0, S_1, \dots, S_h$  such that

- Each list  $S_i$  contains the special keys  $+\infty$  and  $-\infty$
- List  $S_0$  contains the keys of  $S$  in nondecreasing order
- Each list is a subsequence of the previous one, i.e.,
$$S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$$
- List  $S_h$  contains only the two special keys

# Skip List: formally

A skip list for a set  $S$  of distinct (key, element) items is a series of lists  $S_0, S_1, \dots, S_h$  such that

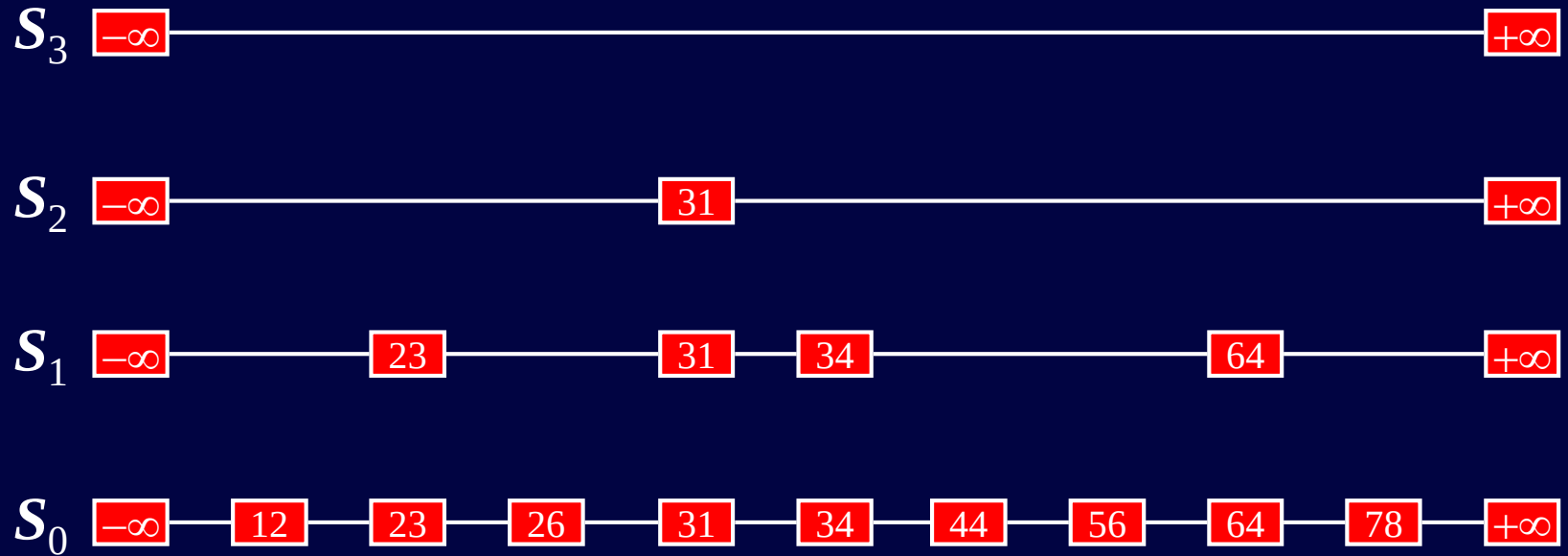
- Each list  $S_i$  contains the special keys  $+\infty$  and  $-\infty$
- List  $S_0$  contains the keys of  $S$  in nondecreasing order
- Each list is a subsequence of the previous one, i.e.,
$$S_0 \supseteq S_1 \supseteq \dots \supseteq S_h$$
- List  $S_h$  contains only the two special keys

Lecture No.40

Data Structure

Dr. Sohail Aslam

# Skip List: formally



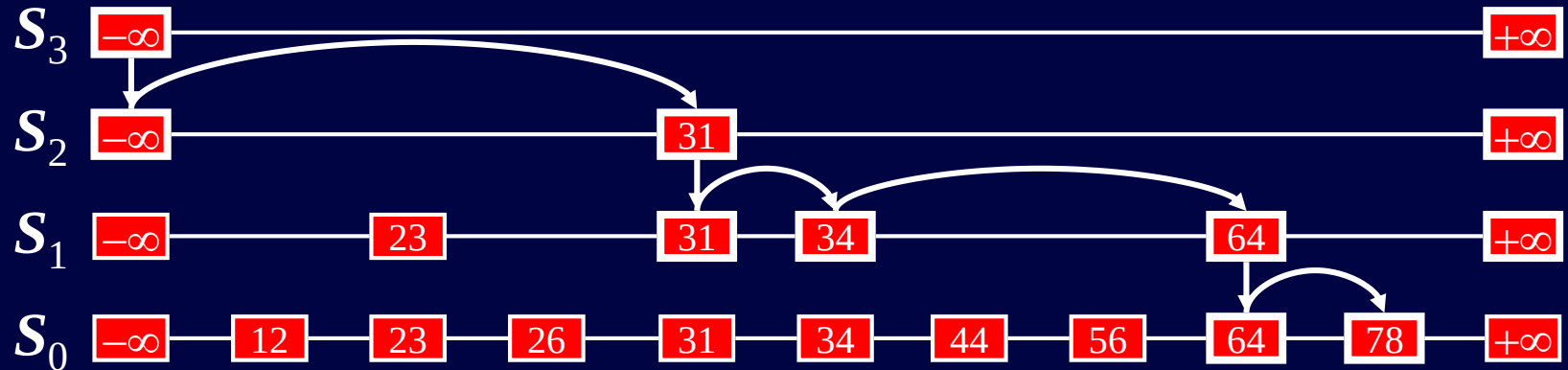
# Skip List: Search

We search for a key  $x$  as follows:

- We start at the first position of the top list
- At the current position  $p$ , we compare  $x$  with  $y \leftarrow \text{key}(\text{after}(p))$
- $x = y$ : we return  $\text{element}(\text{after}(p))$
- $x > y$ : we “scan forward”
- $x < y$ : we “drop down”
- If we try to drop down past the bottom list, we return *NO\_SUCH\_KEY*

# Skip List: Search

Example: search for 78





# Skip List: Insertion

To insert an item  $(x, o)$  into a skip list, we use a randomized algorithm:

- We repeatedly toss a coin until we get tails, and we denote with  $i$  the number of times the coin came up heads
- If  $i \geq h$ , we add to the skip list new lists  $S_{h+1}, \dots, S_{i+1}$ , each containing only the two special keys

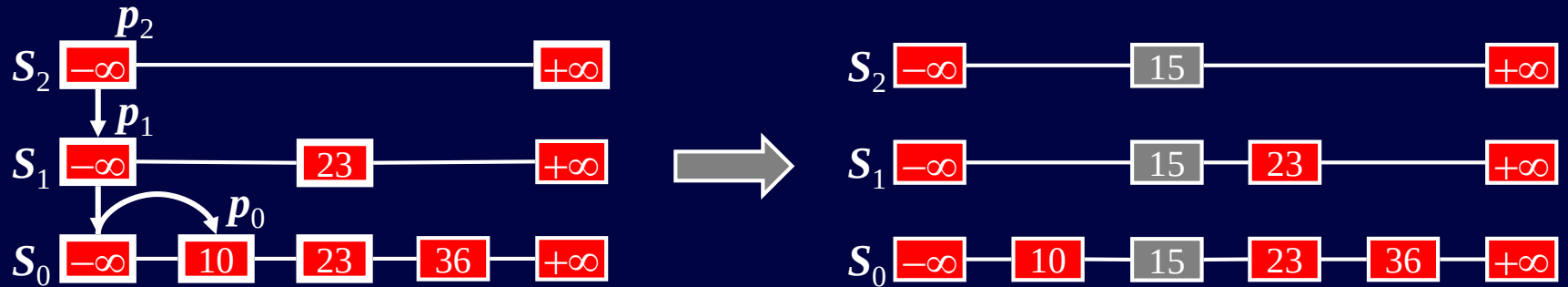
# Skip List: Insertion

To insert an item  $(x, o)$  into a skip list, we use a randomized algorithm: (cont)

- We search for  $x$  in the skip list and find the positions  $p_0, p_1, \dots, p_i$  of the items with largest key less than  $x$  in each list  $S_0, S_1, \dots, S_i$
- For  $j \leftarrow 0, \dots, i$ , we insert item  $(x, o)$  into list  $S_j$  after position  $p_j$

# Skip List: Insertion

- Example: insert key 15, with  $i = 2$



# Randomized Algorithms

- 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 outcomes of the coin tosses, i.e, head or tail

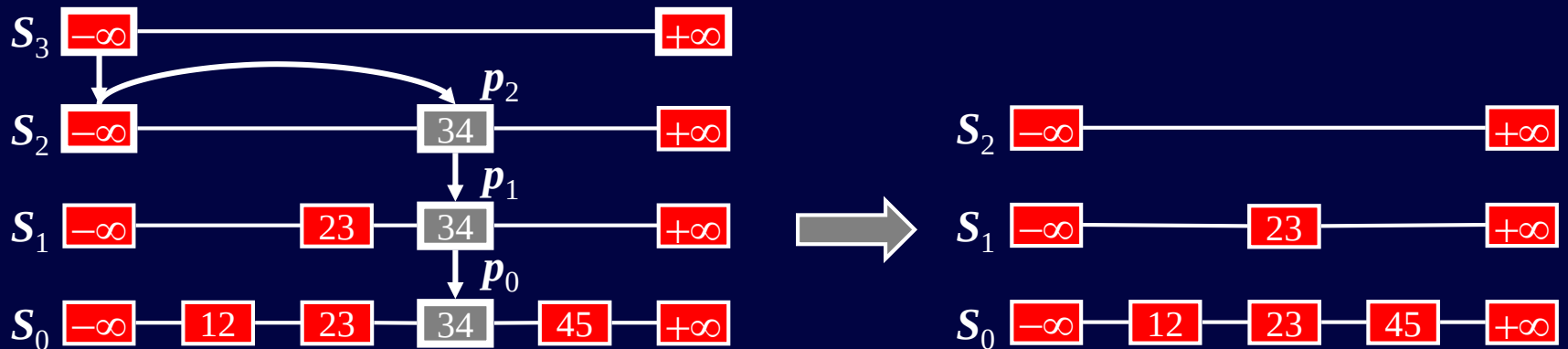
# Skip List: Deletion

To remove an item with key  $x$  from a skip list, we proceed as follows:

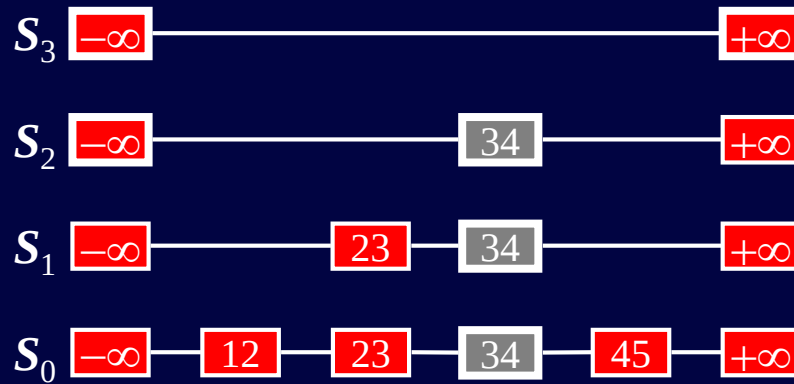
- We search for  $x$  in the skip list and find the positions  $p_0, p_1, \dots, p_i$  of the items with key  $x$ , where position  $p_j$  is in list  $S_j$
- We remove positions  $p_0, p_1, \dots, p_i$  from the lists  $S_0, S_1, \dots, S_i$
- We remove all but one list containing only the two special keys

# Skip List: Deletion

- Example: remove key 34



# Skip List: Implementation



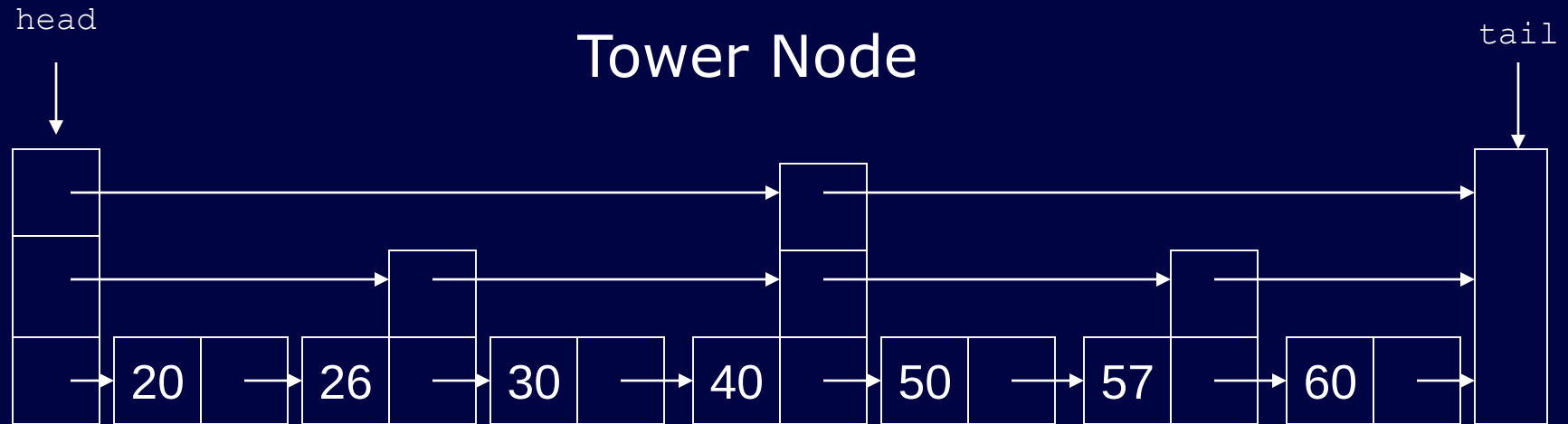
Lecture No.41

Data Structure

Dr. Sohail Aslam



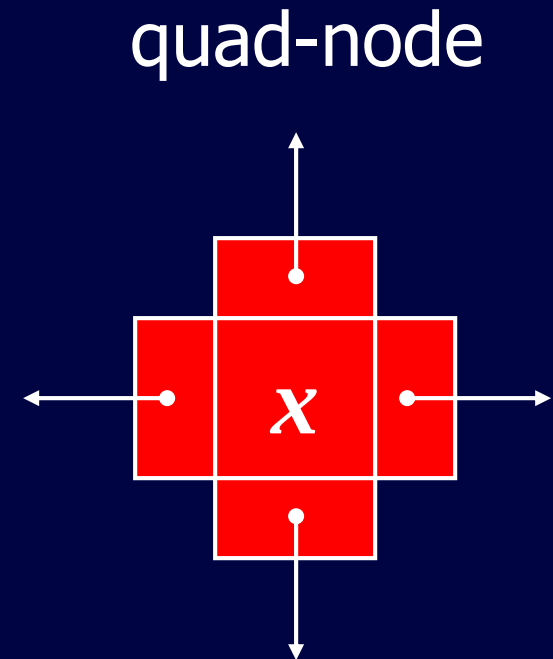
# Implementation: TowerNode



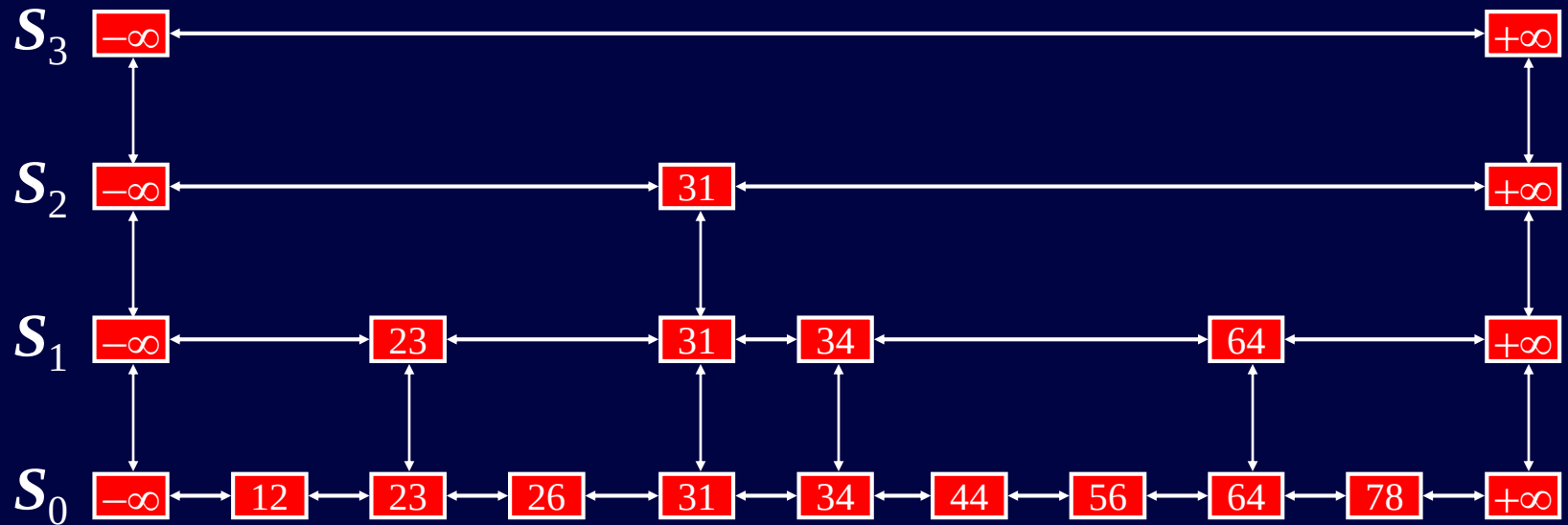
- TowerNode will have array of next pointers.
- Actual number of next pointers will be decided by the random procedure.
- Define MAXLEVEL as an upper limit on number of levels in a node.

# Implementation: QuadNode

- 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 the key (item) at different levels



# Skip Lists with Quad Nodes

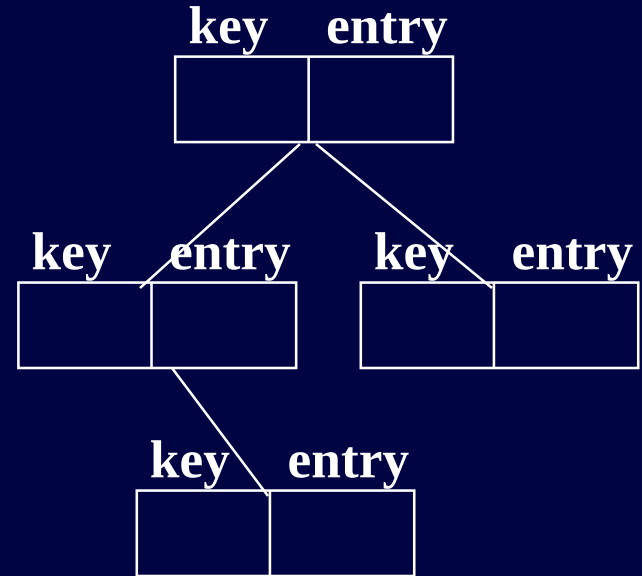


# Performance of Skip Lists

- In a skip list with  $n$  items
  - The expected space used is proportional to  $n$ .
  - The expected search, insertion and deletion time is proportional to  $\log n$ .
- Skip lists are fast and simple to implement in practice

# Implementation 5: AVL tree

- An AVL tree, ordered by key
- insert: a standard insert; ( $\log n$ )
- find: a standard find (without removing, of course); ( $\log n$ )
- remove: a standard remove; ( $\log n$ )



*and so on*

# Anything better?

- So far we have find, remove and insert where time varies between constant  $\log n$ .
- It would be nice to have all three as constant time operations!

# Implementation 6: Hashing

- An *array* in which TableNodes are **not** stored consecutively
- Their place of storage is calculated using the key and a *hash function*



- Keys and entries are scattered throughout the array.

|     | key | entry |
|-----|-----|-------|
|     |     |       |
| 4   |     |       |
|     |     |       |
| 10  |     |       |
|     |     |       |
| 123 |     |       |
|     |     |       |

# Hashing

- insert: calculate place of storage, insert TableNode; (1)
- find: calculate place of storage, retrieve entry; (1)
- remove: calculate place of storage, set it to null; (1)

All are constant time (1) !

|     | key | entry |
|-----|-----|-------|
|     |     |       |
| 4   |     |       |
|     |     |       |
| 10  |     |       |
|     |     |       |
| 123 |     |       |
|     |     |       |



# Hashing

- We use an array of some fixed size  $T$  to hold the data.  $T$  is typically prime.
- Each key is mapped into some number in the range  $0$  to  $T-1$  using a *hash function*, which ideally should be efficient to compute.

# Example: fruits

- 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`

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |

# Example

- Store data in a table array:

```
table[5] = "apple"  
table[3] = "watermelon"  
table[8] = "grapes"  
table[7] = "cantaloupe"  
table[0] = "kiwi"  
table[9] = "strawberry"  
table[6] = "mango"  
table[2] = "banana"
```

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |

# Example

- Associative array:

```
table["apple"]  
table["watermelon"]  
table["grapes"]  
table["cantaloupe"]  
table["kiwi"]  
table["strawberry"]  
table["mango"]  
table["banana"]
```

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |

# Example Hash Functions

- 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:

$$h(str) = \left( \sum_{i=0}^{length-1} str[i] \right) \% TableSize$$

$$Example : h(ABC) = (65 + 66 + 67) \% TableSize$$

# Finding the hash 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;
}
```

# Example Hash Functions

- Another possibility is to convert the string into some number in some arbitrary base  $b$  ( $b$  also might be a prime number):

$$h(str) = \left( \sum_{i=0}^{length-1} str[i] \times b^i \right) \% T$$

$$Example : h(ABC) = (65b^0 + 66b^1 + 67b^2) \% T$$

# Example Hash Functions

- If the keys are integers then  $key \% T$  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.
- In general, to avoid situations like this,  $T$  should be a prime number.



# Lecture No.42

## Data Structures

Dr. Sohail Aslam

# Example Hash Functions

- If the keys are integers then  $key \% T$  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.
- In general, to avoid situations like this,  $T$  should be a prime number.

# Collision

Suppose our hash function gave us the following values:

- $\text{hash}(\text{"apple"}) = 5$   
 $\text{hash}(\text{"watermelon"}) = 3$   
 $\text{hash}(\text{"grapes"}) = 8$   
 $\text{hash}(\text{"cantaloupe"}) = 7$   
 $\text{hash}(\text{"kiwi"}) = 0$   
 $\text{hash}(\text{"strawberry"}) = 9$   
 $\text{hash}(\text{"mango"}) = 6$   
 $\text{hash}(\text{"banana"}) = 2$

**$\text{hash}(\text{"honeydew"}) = 6$**

- Now what?

|   |            |
|---|------------|
| 0 | kiwi       |
| 1 |            |
| 2 | banana     |
| 3 | watermelon |
| 4 |            |
| 5 | apple      |
| 6 | mango      |
| 7 | cantaloupe |
| 8 | grapes     |
| 9 | strawberry |

# Collision

- When two values hash to the same array location, this is called a *collision*
- Collisions are normally treated as “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 this same location.

# Solution for Handling collisions

- **Solution #1:** Search from there for an empty location
  - Can stop searching when we find the value *or* an empty location.
  - Search must be wrap-around at the end.

# Solution for Handling collisions

- **Solution #2:** Use a second hash function
  - ...and a third, and a fourth, and a fifth, ...

# Solution for Handling collisions

- **Solution #3:** Use the array location as the header of a linked list of values that hash to this location

# Solution 1: Open Addressing

- This approach of handling collisions is called *open addressing*; it is also known as *closed hashing*.
- More formally, cells at  $h_0(x)$ ,  $h_1(x)$ ,  $h_2(x)$ , ... are tried in succession where

$$h_i(x) = (\text{hash}(x) + f(i)) \bmod \text{TableSize},$$

with  $f(0) = 0$ .

- The function,  $f$ , is the collision resolution strategy.



# Linear Probing

- We use  $f(i) = i$ , i.e.,  $f$  is a linear function of  $i$ . Thus

$$location(x) = (hash(x) + i) \bmod TableSize$$

- The collision resolution strategy is called *linear probing* because it scans the array sequentially (with wrap around) in search of an empty cell.

# Linear Probing: insert

- Suppose we want to *add seagull* to this hash table
- Also suppose:
  - `hashCode("seagull") = 143`
  - `table[143]` is not empty
  - `table[143] != seagull`
  - `table[144]` is not empty
  - `table[144] != seagull`
  - `table[145]` is empty
- Therefore, put *seagull* at location 145

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |
| ... |         |

# Linear Probing: insert

- Suppose you want to *add hawk* to this hash table
- Also suppose
  - `hashCode("hawk") = 143`
  - `table[143]` is not empty
  - `table[143] != hawk`
  - `table[144]` is not empty
  - `table[144] == hawk`
- *hawk* is already in the table, so do nothing.

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |
| ... |         |

# Linear Probing: insert

- Suppose:
  - You want to add **cardinal** to this hash table
  - `hashCode("cardinal") = 147`
  - The last location is 148
  - 147 and 148 are occupied
- Solution:
  - Treat the table as circular; after 148 comes 0
  - Hence, **cardinal** goes in location 0 (or 1, or 2, or ...)

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |

# Linear Probing: find

- Suppose we want to find **hawk** in this hash table
- We proceed as follows:
  - `hashCode("hawk") = 143`
  - `table[143]` is not empty
  - `table[143] != hawk`
  - `table[144]` is not empty
  - `table[144] == hawk` (found!)
- We use the same procedure for looking things up in the table as we do for inserting them

|     |         |
|-----|---------|
| ... |         |
| 141 |         |
| 142 | robin   |
| 143 | sparrow |
| 144 | hawk    |
| 145 | seagull |
| 146 |         |
| 147 | bluejay |
| 148 | owl     |
| ... |         |

# Linear Probing and Deletion

- If an item is placed in `array[hash(key) + 4]`, then the item just before it is deleted
- How will probe determine that the “hole” does not indicate the item is not in the array?
- Have three states for each location
  - Occupied
  - Empty (never used)
  - Deleted (previously used)

# Clustering

- 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 it is that new values will hash into the cluster, and make it ever bigger.
- Clusters cause efficiency to degrade.

# Quadratic Probing

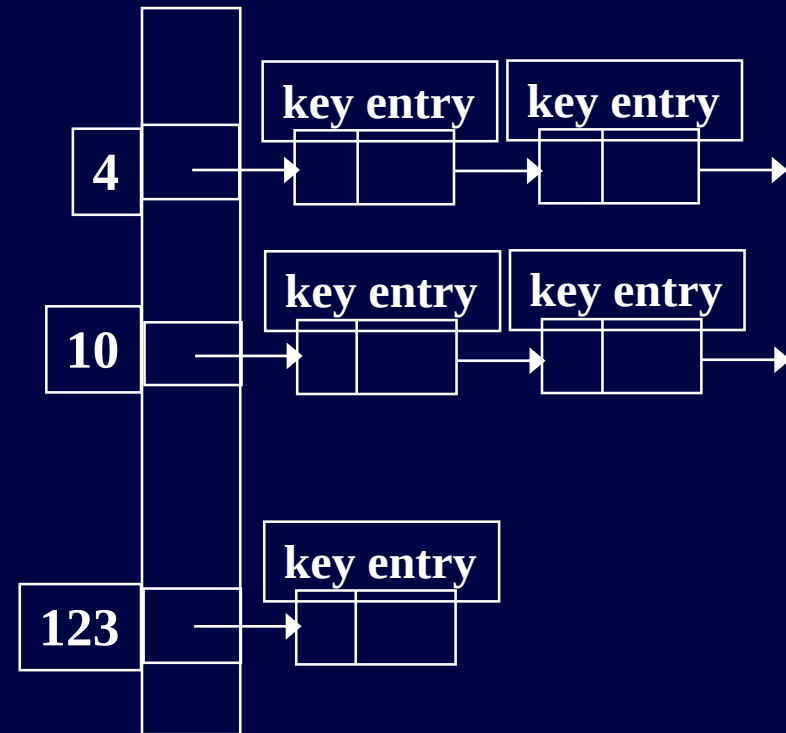
- Quadratic probing uses different formula:
  - Use  $F(i) = i^2$  to resolve collisions
  - If hash function resolves to  $H$  and a search in cell  $H$  is inconclusive, try  $H + 1^2$ ,  $H + 2^2$ ,  $H + 3^2$ , ...
- Probe
  - array[hash(key)+1<sup>2</sup>], then
  - array[hash(key)+2<sup>2</sup>], then
  - array[hash(key)+3<sup>2</sup>], and so on
  - Virtually eliminates primary clusters



# Collision resolution: chaining

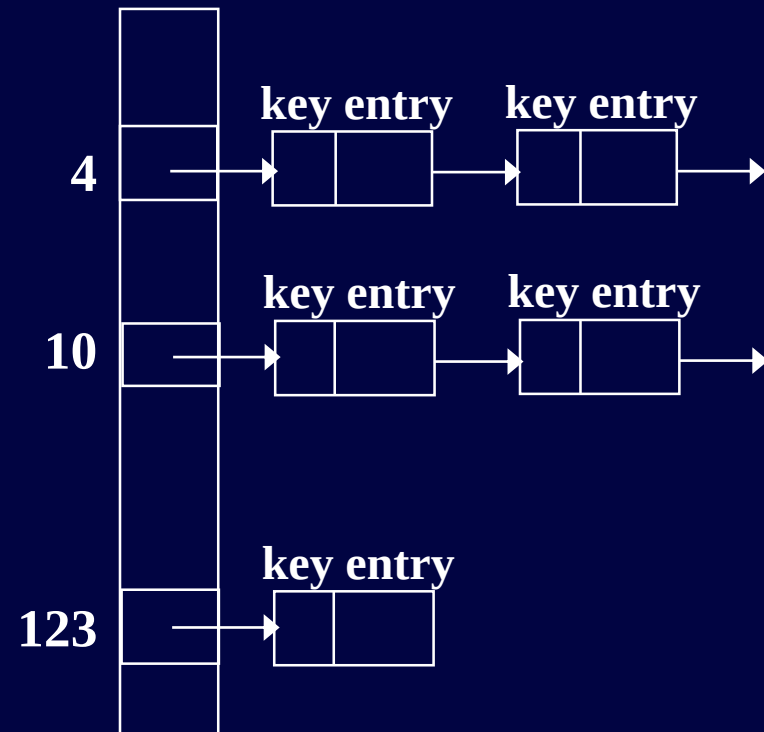
- Each table position is a linked list
- Add the keys and entries anywhere in the list (front easiest)

*No need to change position!*



# Collision resolution: chaining

- Advantages over open addressing:
  - Simpler insertion and removal
  - Array size is not a limitation
- Disadvantage
  - Memory overhead is large if entries are small.



# Lecture No.43

## Data Structures

Dr. Sohail Aslam

# Recap

- Collision Strategies in Hashing
- Linear Probing
- Quadratic Probing
- Linked List chaining

# Hashing Animation

- Let's see the hashing animation. 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 2nd part of the array location. When there is a 2nd 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 2nd 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 2nd 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.

# Hashing Animation

- 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

- Compilers use hash tables to keep track of declared variables (symbol table).
- 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.

# Applications of Hashing

- Game playing programs use hash tables to store seen positions, thereby saving computation time if the position is encountered again.
- Hash functions can be used to quickly check for inequality — if two elements hash to different values they must be different.

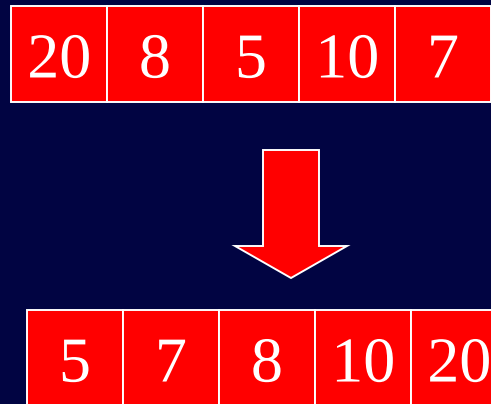


# When is hashing suitable?

- Hash tables are very good if there is a need for many searches in a reasonably stable table.
- 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.
- Also, hashing is very slow for any operations which require the entries to be sorted
  - e.g. Find the minimum key

# Sorting Integers

How to sort the integers in this array?



# Elementary Sorting Algorithms

- Selection Sort
- Insertion Sort
- Bubble Sort

# 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

# Sorting

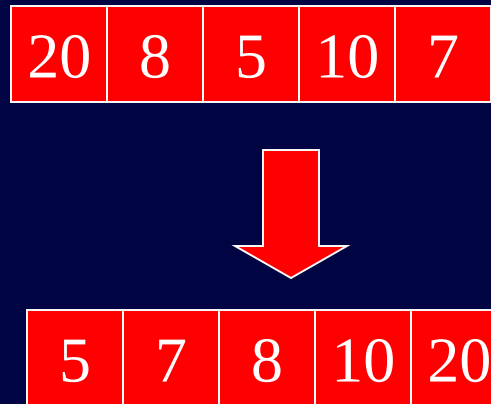
Lecture No.44

Data Structure

Dr. Sohail Aslam

# Sorting Integers

How to sort the integers in this array?



# Elementary Sorting Algorithms

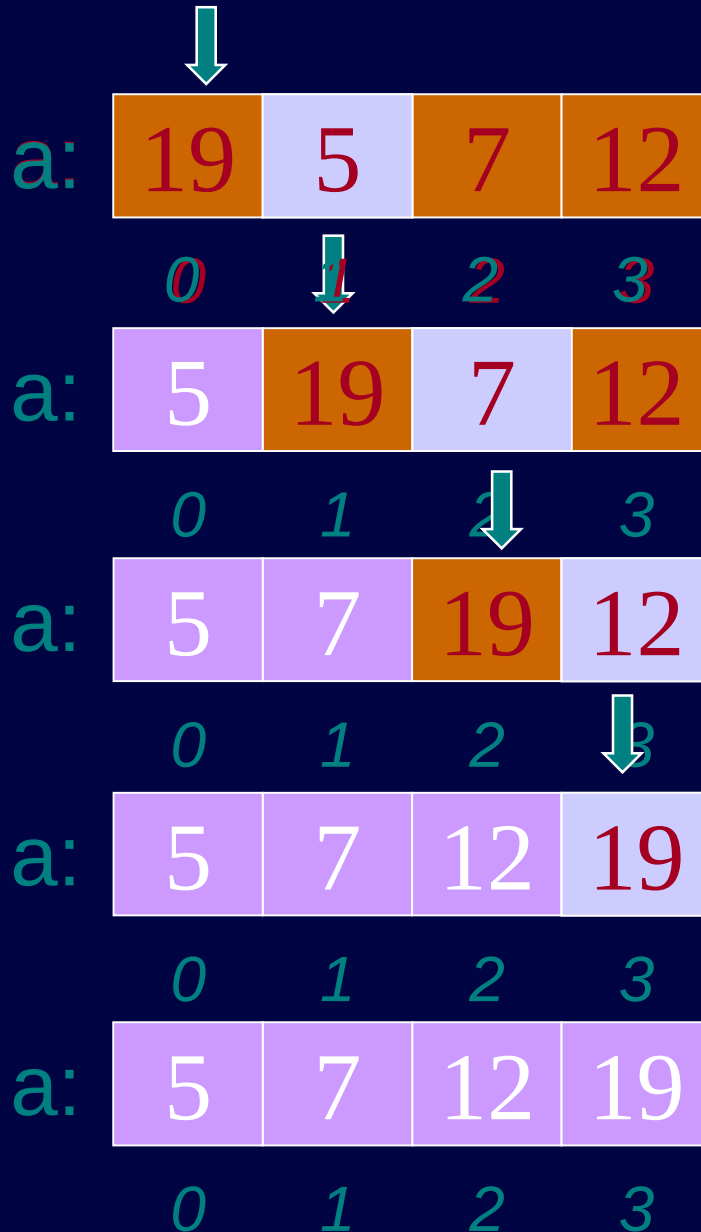
- Selection Sort
- Insertion Sort
- Bubble Sort



# 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

# Selection Sort -- Example



## Selection Sort: Code

```
void selectionSort(int *arr, int N)
{
    int posmin, count, tmp;
    for(count=0;count<N;count++)
    {
        posmin = findIndexMin(arr,count,N) ;

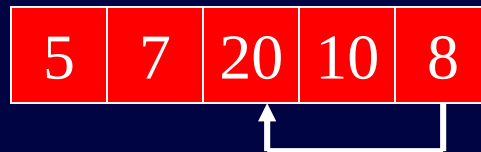
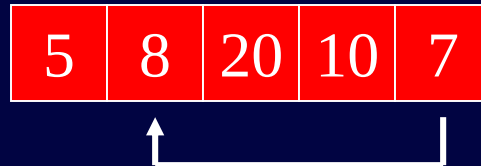
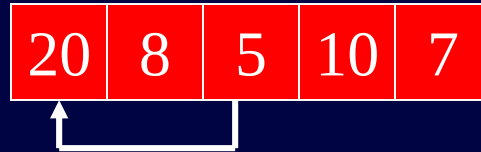
        tmp=arr[posmin] ;
        arr[posmin]=arr[count] ;
        arr[count]=tmp;
    }
}
```

## Selection Sort: Code

```
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;
}
```

# Swap Action (SelectionSorting)



# Selection Sort Analysis

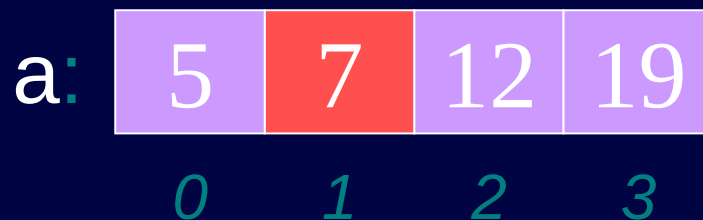
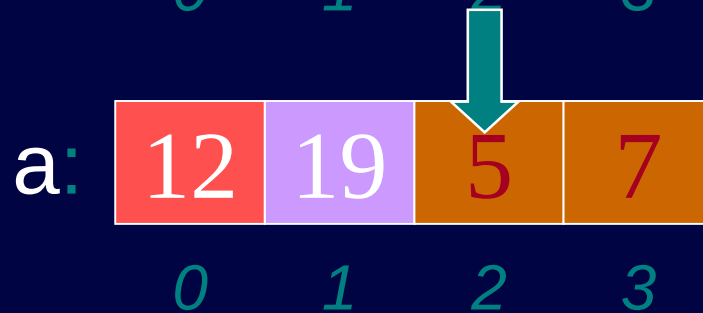
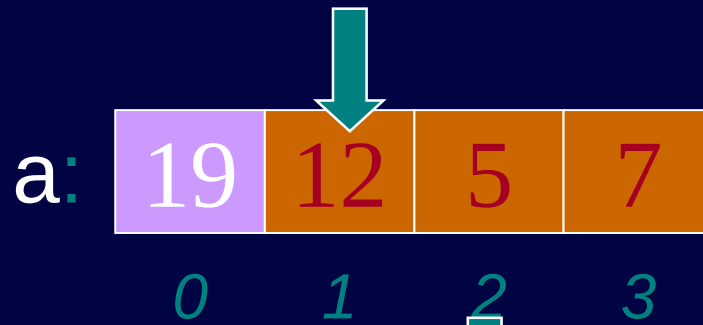
- What is the time complexity of this algorithm?
- Worst case == Best case == Average case
- Each iteration performs a linear search on the rest of the array

|                       |                             |   |
|-----------------------|-----------------------------|---|
| • first element       | N                           | + |
| • second element      | N-1                         | + |
| • ...                 |                             |   |
| • penultimate element | 2                           | + |
| • last element        | 1                           |   |
| <hr/>                 |                             |   |
| • Total               | $N(N+1)/2$<br>$= (N^2+N)/2$ |   |

# Insertion Sort

- Basic idea (*sorting cards*):
  - Starts by considering the first two elements of the array data, if out of order, swap them
  - Consider the third element, insert it into the proper position among the first three elements.
  - Consider the forth element, insert it into the proper position among the first four elements.
  - ... ..

# Insertion Sort -- Example





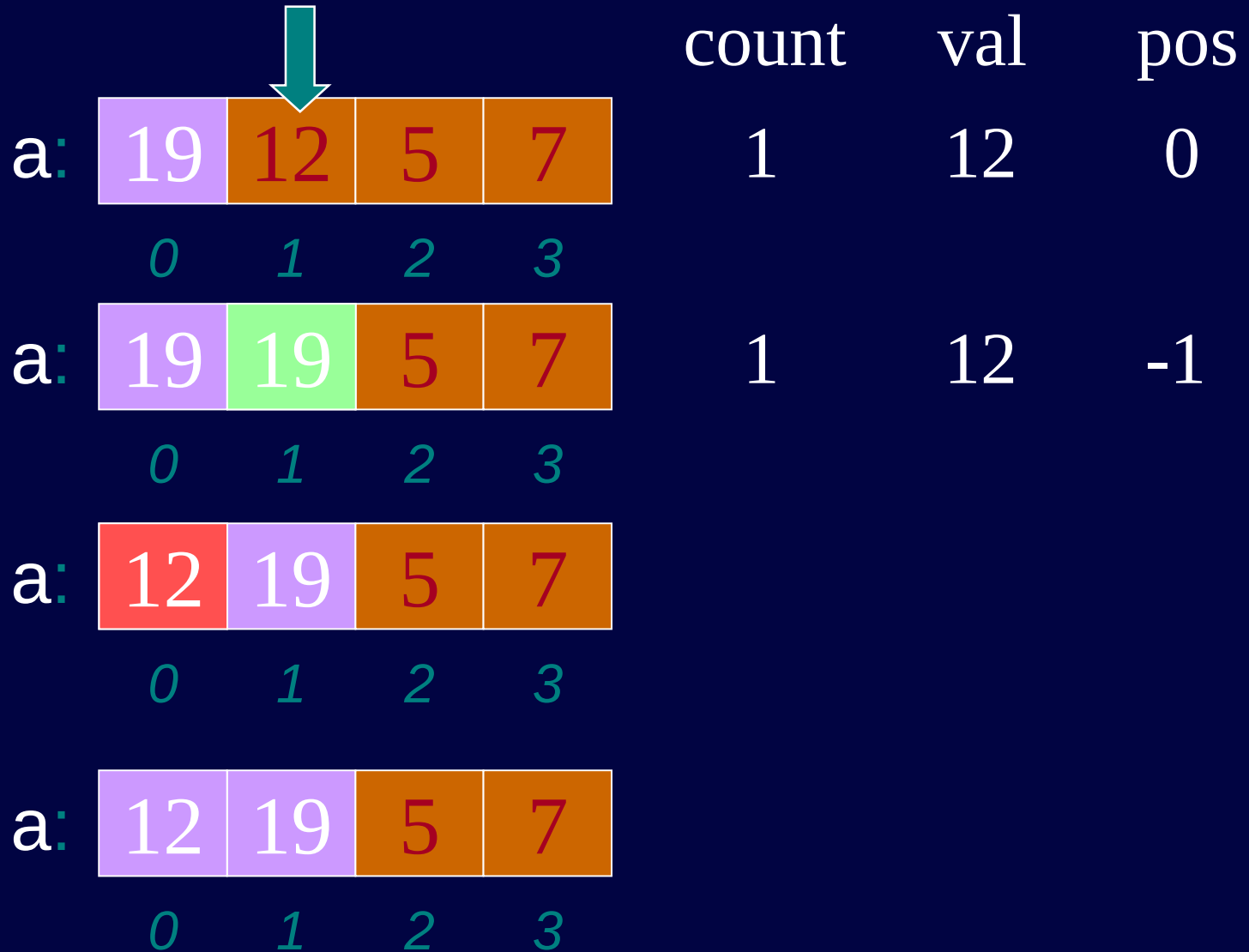
# Insertion Sort: Code

```
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;
    }
}
```

# Insertion Sort -- animation



# Insertion Sort -- animation (cont)

|    |                                                                                                                            | count | val | pos |
|----|----------------------------------------------------------------------------------------------------------------------------|-------|-----|-----|
| a: | <div><div>12</div><div>19</div><div>5</div><div>7</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div>  | 2     | 5   | 1   |
| a: | <div><div>12</div><div>19</div><div>19</div><div>7</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div> | 2     | 5   | 0   |
| a: | <div><div>12</div><div>12</div><div>19</div><div>7</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div> | 2     | 5   | -1  |
| a: | <div><div>5</div><div>12</div><div>19</div><div>7</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div>  |       |     |     |
| a: | <div><div>5</div><div>12</div><div>19</div><div>7</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div>  |       |     |     |

# Insertion Sort -- animation (cont)

|    |                                                                                                                            | count | val | pos |
|----|----------------------------------------------------------------------------------------------------------------------------|-------|-----|-----|
| a: | <div><div>5</div><div>12</div><div>19</div><div>7</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div>  | 3     | 7   | 2   |
| a: | <div><div>5</div><div>12</div><div>19</div><div>19</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div> | 3     | 7   | 1   |
| a: | <div><div>5</div><div>12</div><div>12</div><div>19</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div> | 3     | 7   | 0   |
| a: | <div><div>5</div><div>7</div><div>12</div><div>19</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div>  |       |     |     |
| a: | <div><div>5</div><div>7</div><div>12</div><div>19</div></div> <div><div>0</div><div>1</div><div>2</div><div>3</div></div>  |       |     |     |

# Insertion Sort Analysis

- What is the time complexity of this algorithm?
- Worst case > Average case > Best case
- Each iteration inserts an element at the start of the array, shifting all sorted elements along

- second element                      2            +

- ...

- penultimate element    N-1    +

- last element                      N

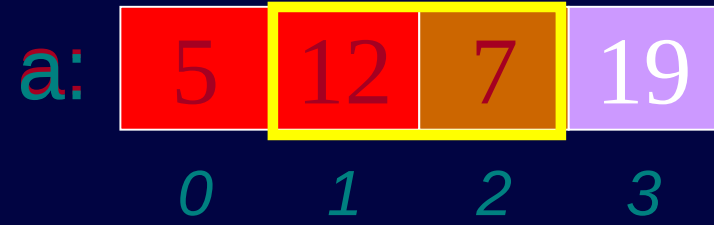
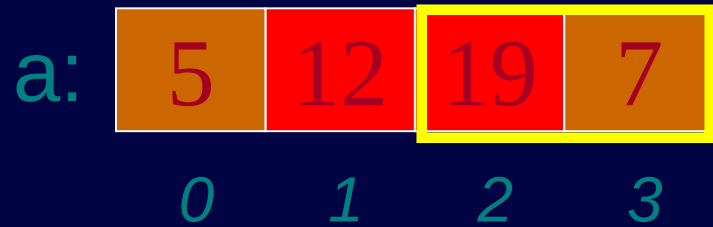
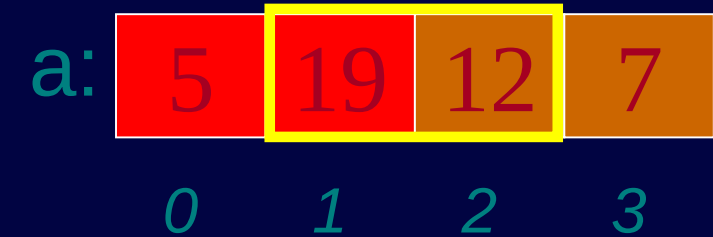
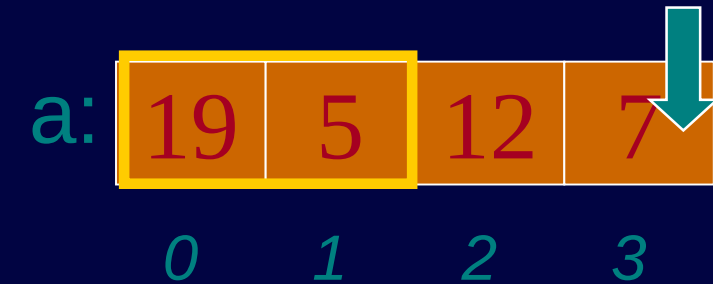
---

- Total                                       $(2+N)(N-1)/2 = O(N^2)$

# Bubble Sort

- Basic idea (*lighter bubbles rise to the top*):
  - Exchange neighbouring items until the largest item reaches the end of the array
  - Repeat for the rest of the array

# Bubble Sort -- Example



# Bubble Sort: Code

```
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;  
    }  
}
```





# Summary

- Insertion, Selection and Bubble sort:
  - Worst case time complexity is proportional to  $N^2$ .

Best sorting routines are  $N \log(N)$

# NLogN Algorithms

- Divide and Conquer
- Merge Sort
- Quick Sort
- Heap Sort

# Divide and Conquer

**What if we split the list into two parts?**

|    |    |   |   |   |    |   |   |
|----|----|---|---|---|----|---|---|
| 10 | 12 | 8 | 4 | 2 | 11 | 7 | 5 |
|----|----|---|---|---|----|---|---|

# Divide and Conquer

**Sort the two parts:**

|   |   |    |    |   |   |   |    |
|---|---|----|----|---|---|---|----|
| 4 | 8 | 10 | 12 | 2 | 5 | 7 | 11 |
|---|---|----|----|---|---|---|----|

# Divide and Conquer

**Then merge the two parts together:**

|   |   |   |   |   |    |    |    |
|---|---|---|---|---|----|----|----|
| 2 | 4 | 5 | 7 | 8 | 10 | 11 | 12 |
|---|---|---|---|---|----|----|----|

# Lecture No.45

## Data Structures

Dr. Sohail Aslam

# Divide and Conquer

**What if we split the list into two parts?**

|    |    |   |   |   |    |   |   |
|----|----|---|---|---|----|---|---|
| 10 | 12 | 8 | 4 | 2 | 11 | 7 | 5 |
|----|----|---|---|---|----|---|---|



# Divide and Conquer

**Sort the two parts:**

|   |   |    |    |   |   |   |    |
|---|---|----|----|---|---|---|----|
| 4 | 8 | 10 | 12 | 2 | 5 | 7 | 11 |
|---|---|----|----|---|---|---|----|

# Divide and Conquer

**Then merge the two parts together:**

|   |   |   |   |   |    |    |    |
|---|---|---|---|---|----|----|----|
| 2 | 4 | 5 | 7 | 8 | 10 | 11 | 12 |
|---|---|---|---|---|----|----|----|

# Analysis

- To sort the halves  $\star (n/2)^2 + (n/2)^2$
- To merge the two halves  $\star n$
- So, for  $n=100$ , divide and conquer takes:

$$= (100/2)^2 + (100/2)^2 + 100$$

$$= 2500 + 2500 + 100$$

$$= 5100 \quad (n^2 = 10,000)$$

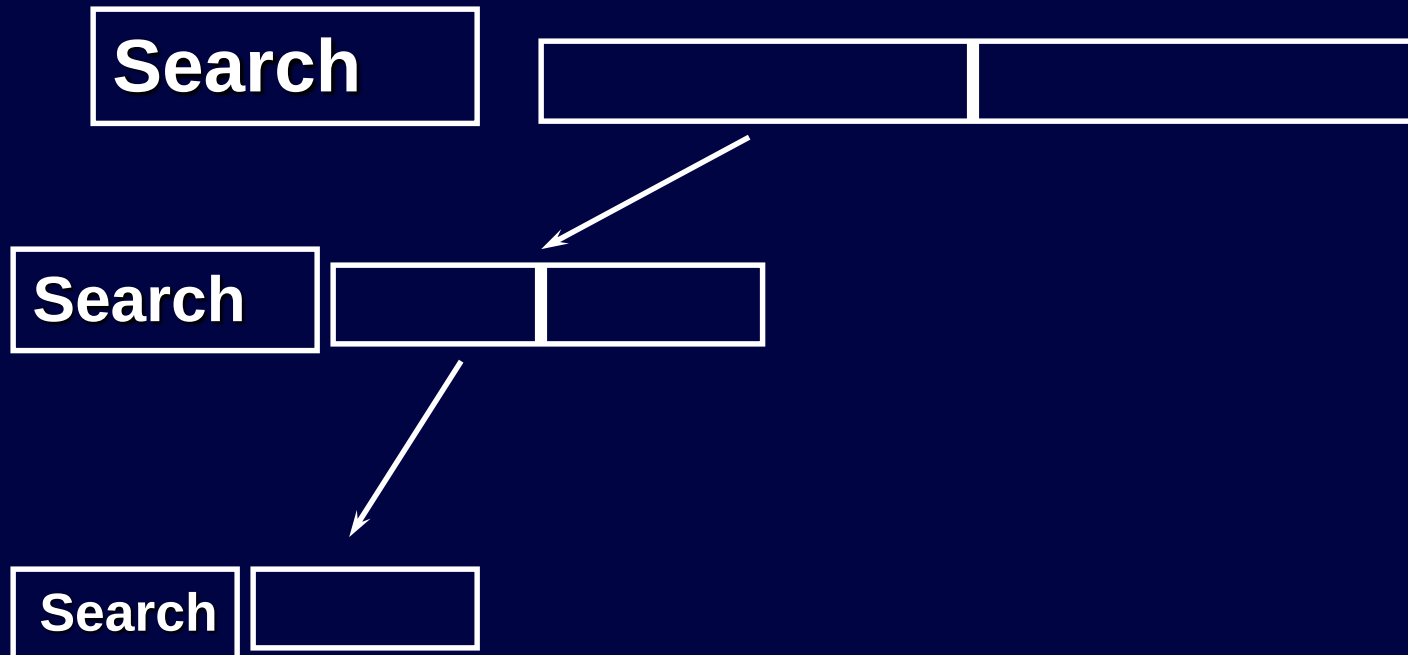
# Divide and Conquer

- Why not divide the halves in half?
- The quarters in half?
- And so on . . .
- When should we stop?

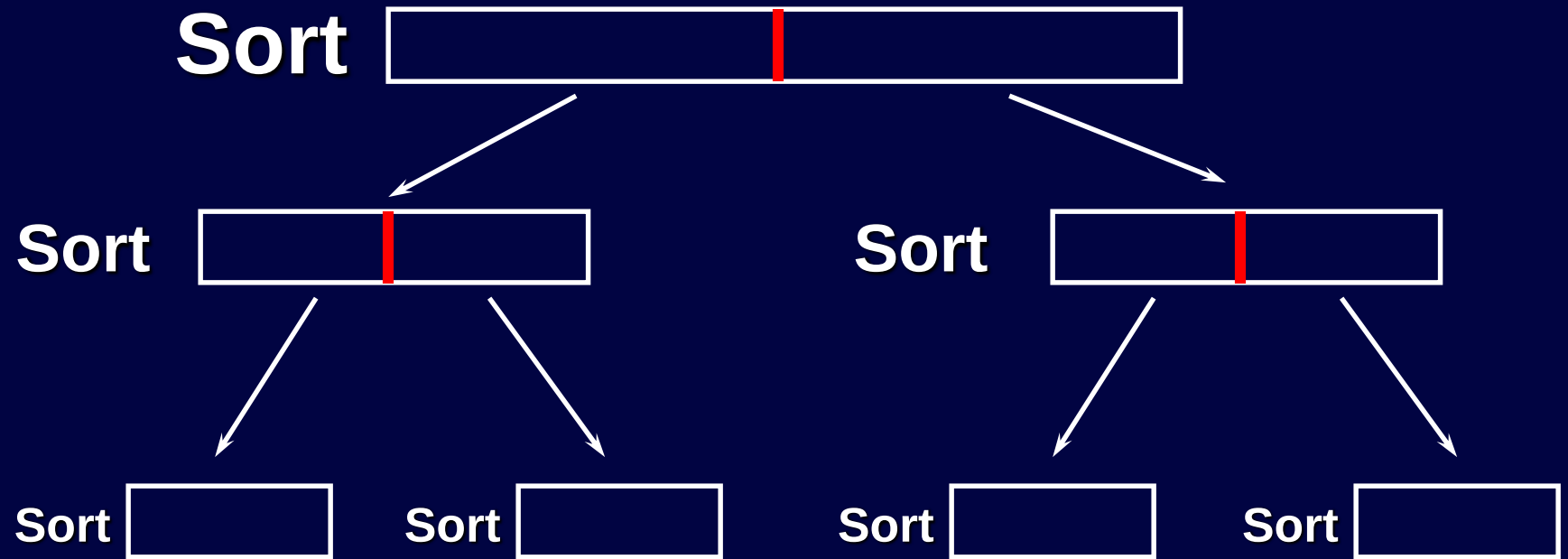
At  $n = 1$

# Divide and Conquer

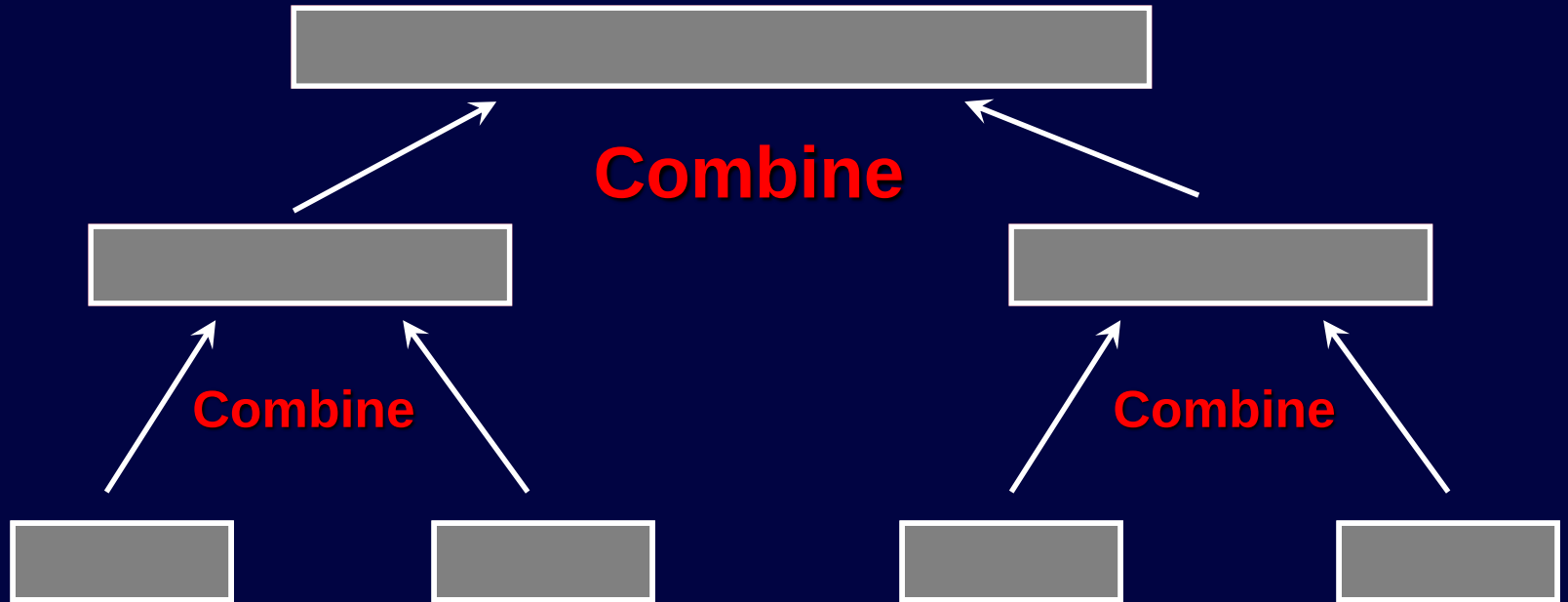
***Recall: Binary Search***



# Divide and Conquer



# Divide and Conquer



# Mergesort

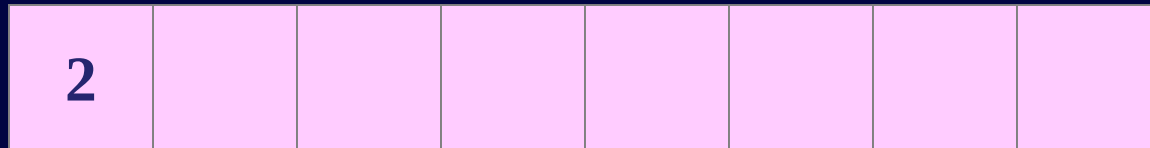
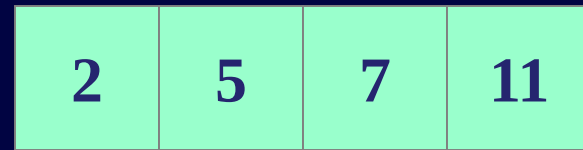
- ***Mergesort*** is a *divide and conquer* algorithm that does exactly that.
- It splits the list in half
- Mergesorts the two halves
- Then merges the two sorted halves together
- Mergesort can be implemented recursively



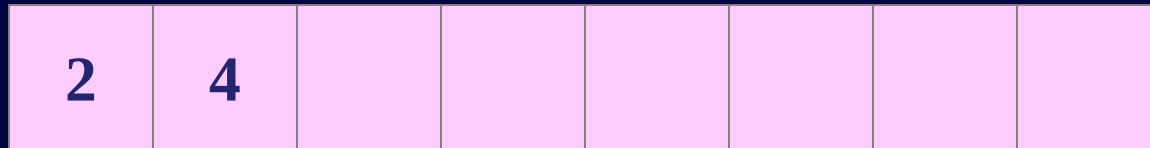
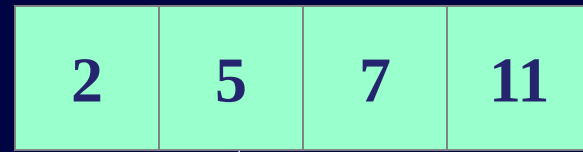
# Mergesort

- 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

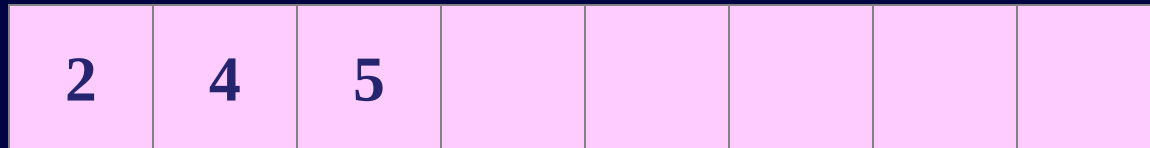
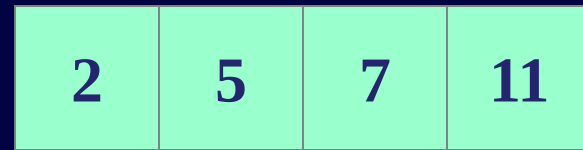
# Merging: animation



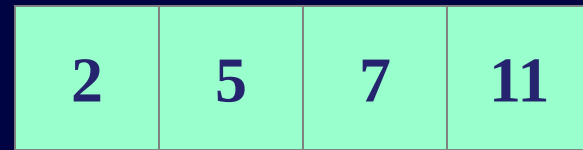
# Merging: animation



# Merging: animation



# Merging



# Mergesort

**Split the list in half.**

**Mergesort the left half.**

|    |   |   |    |    |   |   |   |
|----|---|---|----|----|---|---|---|
| 10 | 4 | 8 | 12 | 11 | 2 | 7 | 5 |
|----|---|---|----|----|---|---|---|

**Split the list in half.**

**Mergesort the left half.**

|    |   |   |    |
|----|---|---|----|
| 10 | 4 | 8 | 12 |
|----|---|---|----|

**Split the list in half.**

**Mergesort the left half.**

|    |   |
|----|---|
| 10 | 4 |
|----|---|

**Mergesort the right.**

|    |   |
|----|---|
| 10 | 4 |
|----|---|

# Mergesort

|    |   |   |    |    |   |   |   |
|----|---|---|----|----|---|---|---|
| 10 | 4 | 8 | 12 | 11 | 2 | 7 | 5 |
|----|---|---|----|----|---|---|---|

|    |   |   |    |
|----|---|---|----|
| 10 | 4 | 8 | 12 |
|----|---|---|----|

**Mergesort the right half. Merge the two halves.**

|   |    |   |    |
|---|----|---|----|
| 4 | 10 | 8 | 12 |
|---|----|---|----|

**Merge the two halves.**

|   |    |
|---|----|
| 8 | 12 |
|---|----|

# Mergesort

|    |   |   |    |    |   |   |   |
|----|---|---|----|----|---|---|---|
| 10 | 4 | 8 | 12 | 11 | 2 | 7 | 5 |
|----|---|---|----|----|---|---|---|

**Merge the two halves.**

|   |   |    |    |
|---|---|----|----|
| 4 | 8 | 10 | 12 |
|---|---|----|----|

**Mergesort the right half. Merge the two halves.**

|   |    |   |    |
|---|----|---|----|
| 4 | 10 | 8 | 12 |
|---|----|---|----|



# Mergesort

Mergesort the right half.

|   |   |    |    |    |   |   |   |
|---|---|----|----|----|---|---|---|
| 4 | 8 | 10 | 12 | 11 | 2 | 7 | 5 |
|---|---|----|----|----|---|---|---|

|    |   |   |   |
|----|---|---|---|
| 11 | 2 | 7 | 5 |
|----|---|---|---|

|    |   |
|----|---|
| 11 | 2 |
|----|---|

|    |   |
|----|---|
| 11 | 2 |
|----|---|

# Mergesort

**Mergesort the right half.**

|   |   |    |    |    |   |   |   |
|---|---|----|----|----|---|---|---|
| 4 | 8 | 10 | 12 | 11 | 2 | 7 | 5 |
|---|---|----|----|----|---|---|---|

|   |    |   |   |
|---|----|---|---|
| 2 | 11 | 7 | 5 |
|---|----|---|---|

|   |    |
|---|----|
| 2 | 11 |
|---|----|

# Mergesort

Mergesort the right half.

|   |   |    |    |    |   |   |   |
|---|---|----|----|----|---|---|---|
| 4 | 8 | 10 | 12 | 11 | 2 | 7 | 5 |
|---|---|----|----|----|---|---|---|

|   |    |   |   |
|---|----|---|---|
| 2 | 11 | 7 | 5 |
|---|----|---|---|

|   |   |
|---|---|
| 5 | 7 |
|---|---|

|   |   |
|---|---|
| 7 | 5 |
|---|---|

# Mergesort

Mergesort the right half.

|   |   |    |    |    |   |   |   |
|---|---|----|----|----|---|---|---|
| 4 | 8 | 10 | 12 | 11 | 2 | 7 | 5 |
|---|---|----|----|----|---|---|---|

|   |    |   |   |
|---|----|---|---|
| 2 | 11 | 7 | 5 |
|---|----|---|---|

# Mergesort

**Mergesort the right half.**

|   |   |    |    |   |   |   |    |
|---|---|----|----|---|---|---|----|
| 4 | 8 | 10 | 12 | 2 | 5 | 7 | 11 |
|---|---|----|----|---|---|---|----|

# Mergesort

**Merge the two halves.**

|   |   |   |   |   |    |    |    |
|---|---|---|---|---|----|----|----|
| 2 | 4 | 5 | 7 | 8 | 10 | 11 | 12 |
|---|---|---|---|---|----|----|----|

# Mergesort

```
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;
}
```

# Mergesort

```
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];
    }
}
```



# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**aSize: 5**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**bSize:** 6

[illegible]

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=0**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=0**

**tmp:**

|  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|--|--|

**k=0**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=0**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=0**

**tmp:**

|   |  |  |  |  |  |  |  |  |  |  |
|---|--|--|--|--|--|--|--|--|--|--|
| 3 |  |  |  |  |  |  |  |  |  |  |
|---|--|--|--|--|--|--|--|--|--|--|

**k=0**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=1**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=0**

**tmp:**

|   |   |  |  |  |  |  |  |  |  |  |
|---|---|--|--|--|--|--|--|--|--|--|
| 3 | 5 |  |  |  |  |  |  |  |  |  |
|---|---|--|--|--|--|--|--|--|--|--|

**k=1**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=2**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=0**

**tmp:**

|   |   |   |  |  |  |  |  |  |  |  |
|---|---|---|--|--|--|--|--|--|--|--|
| 3 | 5 | 6 |  |  |  |  |  |  |  |  |
|---|---|---|--|--|--|--|--|--|--|--|

**k=2**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=2**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=1**

**tmp:**

|   |   |   |    |  |  |  |  |  |  |  |
|---|---|---|----|--|--|--|--|--|--|--|
| 3 | 5 | 6 | 10 |  |  |  |  |  |  |  |
|---|---|---|----|--|--|--|--|--|--|--|

**k=3**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=2**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=2**

**tmp:**

|   |   |   |    |    |  |  |  |  |  |  |
|---|---|---|----|----|--|--|--|--|--|--|
| 3 | 5 | 6 | 10 | 14 |  |  |  |  |  |  |
|---|---|---|----|----|--|--|--|--|--|--|

**k=4**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=2**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=3**

**tmp:**

|   |   |   |    |    |    |  |  |  |  |  |
|---|---|---|----|----|----|--|--|--|--|--|
| 3 | 5 | 6 | 10 | 14 | 15 |  |  |  |  |  |
|---|---|---|----|----|----|--|--|--|--|--|

**k=5**



# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=3**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=3**

**tmp:**

|   |   |   |    |    |    |    |  |  |  |  |
|---|---|---|----|----|----|----|--|--|--|--|
| 3 | 5 | 6 | 10 | 14 | 15 | 22 |  |  |  |  |
|---|---|---|----|----|----|----|--|--|--|--|

**k=6**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=3**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=4**

**tmp:**

|   |   |   |    |    |    |    |    |  |  |  |
|---|---|---|----|----|----|----|----|--|--|--|
| 3 | 5 | 6 | 10 | 14 | 15 | 22 | 28 |  |  |  |
|---|---|---|----|----|----|----|----|--|--|--|

**k=7**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=4**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=4**

**tmp:**

|   |   |   |    |    |    |    |    |    |  |  |
|---|---|---|----|----|----|----|----|----|--|--|
| 3 | 5 | 6 | 10 | 14 | 15 | 22 | 28 | 30 |  |  |
|---|---|---|----|----|----|----|----|----|--|--|

**k=8**

# mergeArrays

**a:**

|   |   |    |    |    |
|---|---|----|----|----|
| 3 | 5 | 15 | 28 | 30 |
|---|---|----|----|----|

**i=5**

**b:**

|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 6 | 10 | 14 | 22 | 43 | 50 |
|---|----|----|----|----|----|

**j=4**

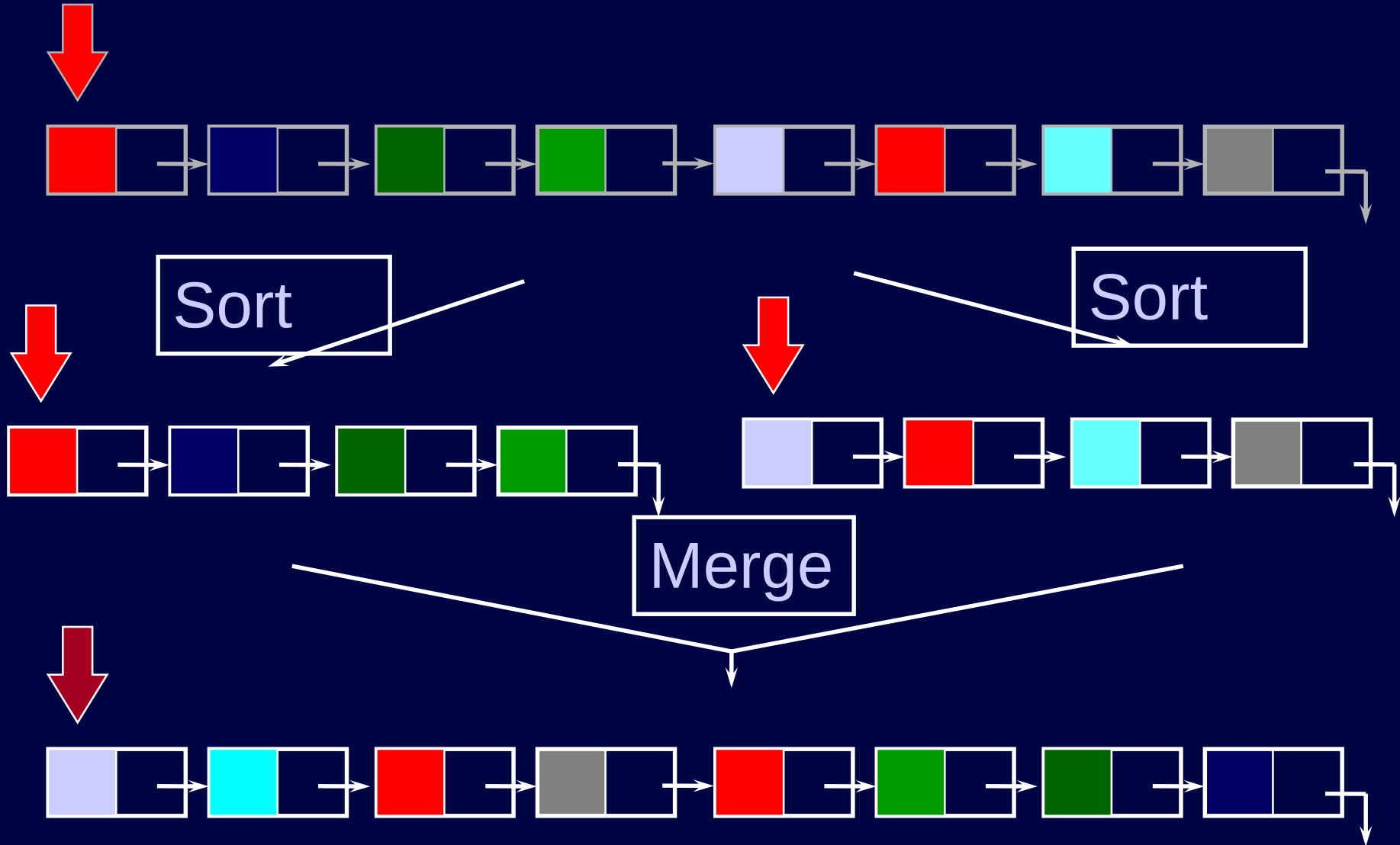
***Done.***

**tmp:**

|   |   |   |    |    |    |    |    |    |    |    |
|---|---|---|----|----|----|----|----|----|----|----|
| 3 | 5 | 6 | 10 | 14 | 15 | 22 | 28 | 30 | 43 | 50 |
|---|---|---|----|----|----|----|----|----|----|----|

**k=9**

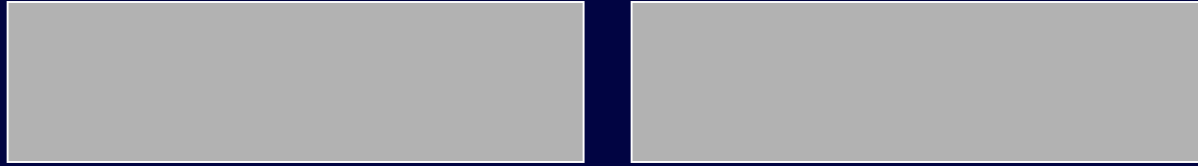
# Merge Sort and Linked Lists



# Mergesort Analysis

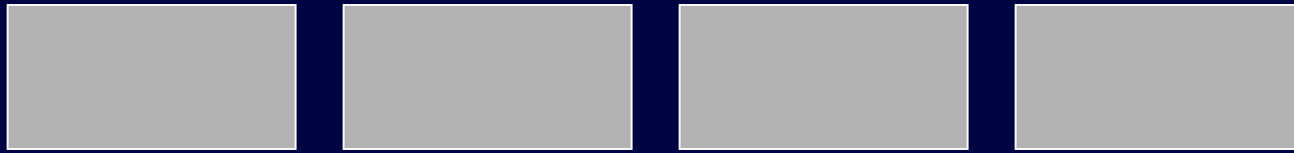
Merging the two lists of size  $n/2$ :

$O(n)$



Merging the four lists of size  $n/4$ :

$O(n)$



Merging the  $n$  lists of size 1:

$O(n)$



$O(\lg n)$   
times

- Mergesort is  $O(n \lg n)$
- 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

# Mergesort Analysis

- Mergesort is  $O(n \lg n)$
- 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

# 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

First the list is partitioned around a pivot value. Pivot can be chosen from the beginning, end or middle of list):

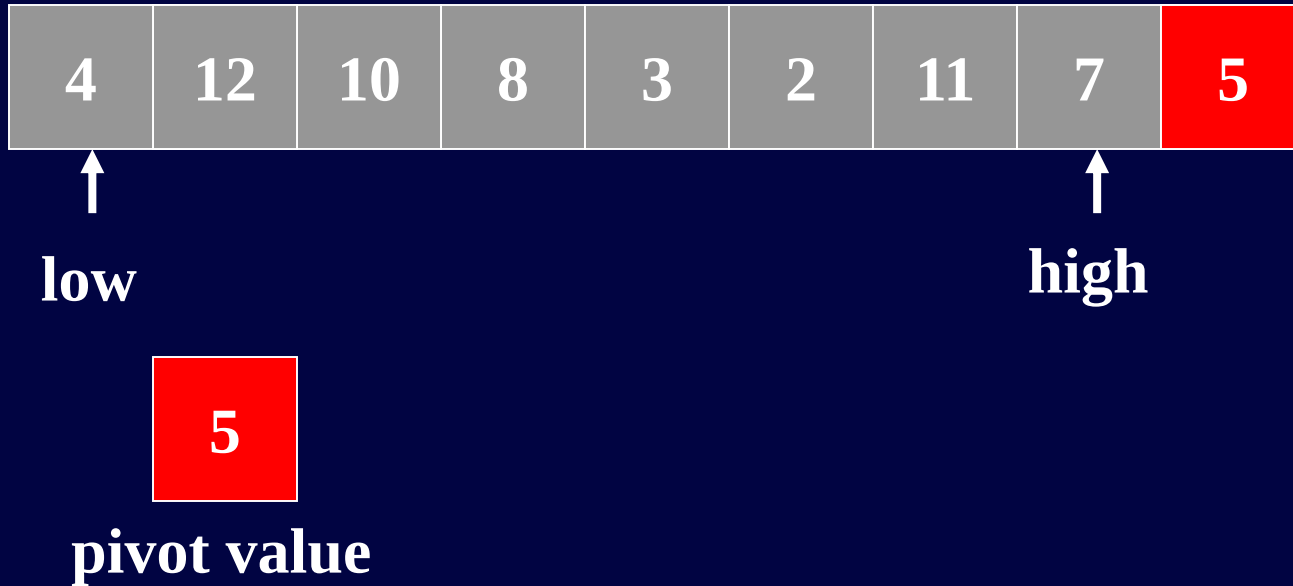
|   |    |    |   |   |   |    |   |   |
|---|----|----|---|---|---|----|---|---|
| 4 | 12 | 10 | 8 | 5 | 2 | 11 | 7 | 3 |
|---|----|----|---|---|---|----|---|---|

5

**pivot value**

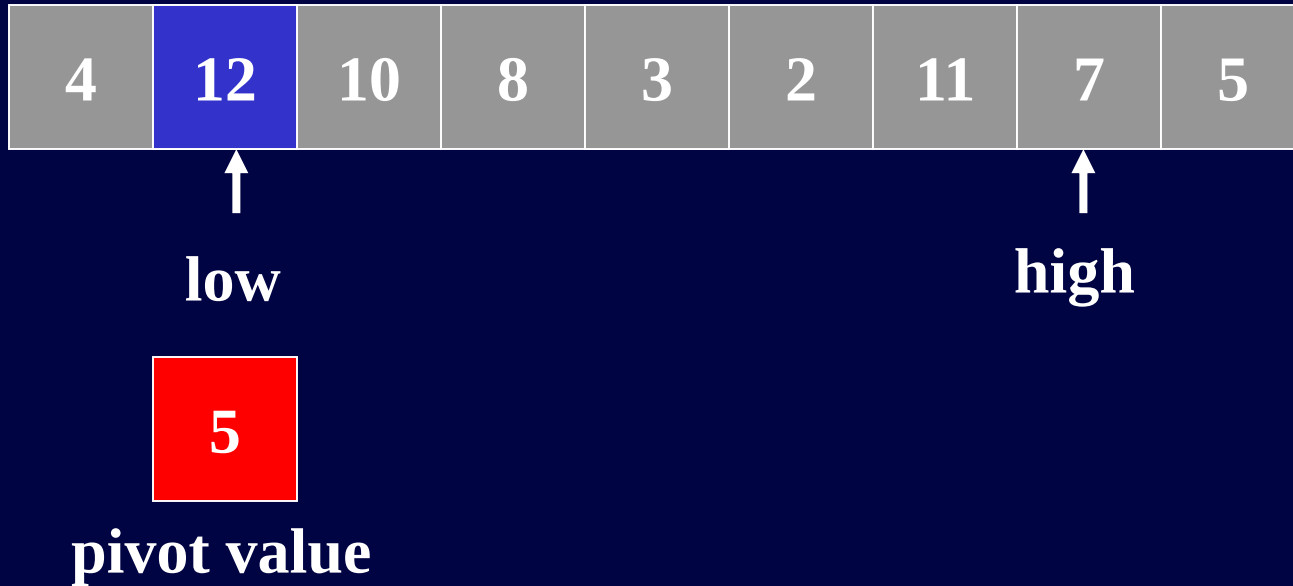
# Quicksort

The pivot is swapped to the last position and the remaining elements are compared starting at the ends.



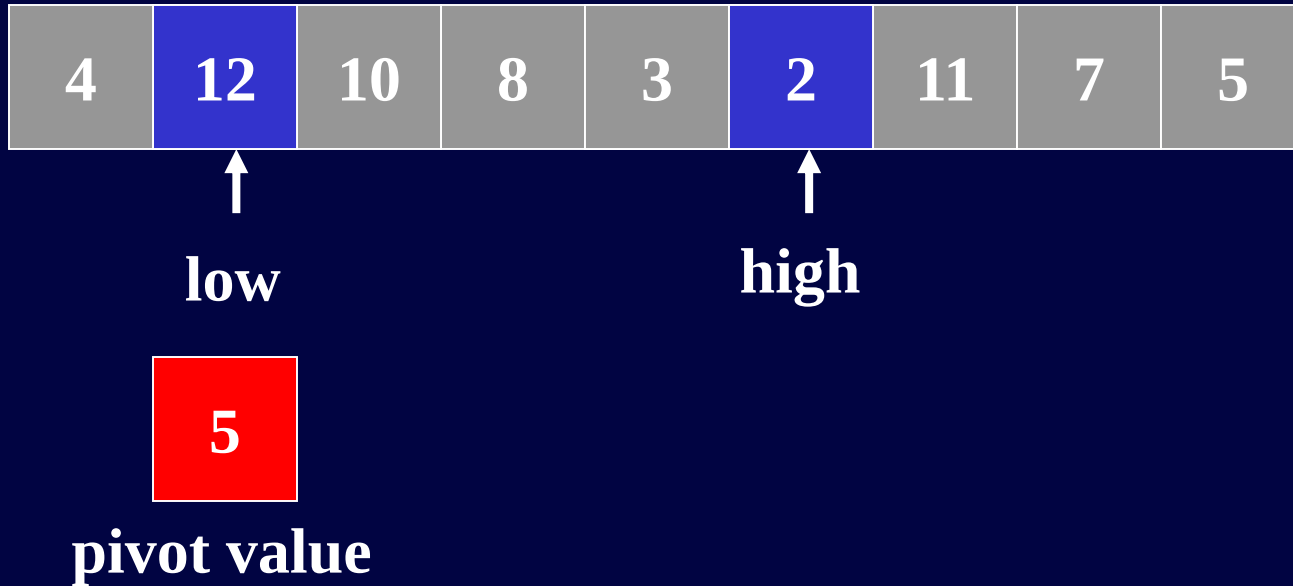
# Quicksort

Then the low index moves right until it is at an element that is larger than the pivot value (i.e., it is on the wrong side)



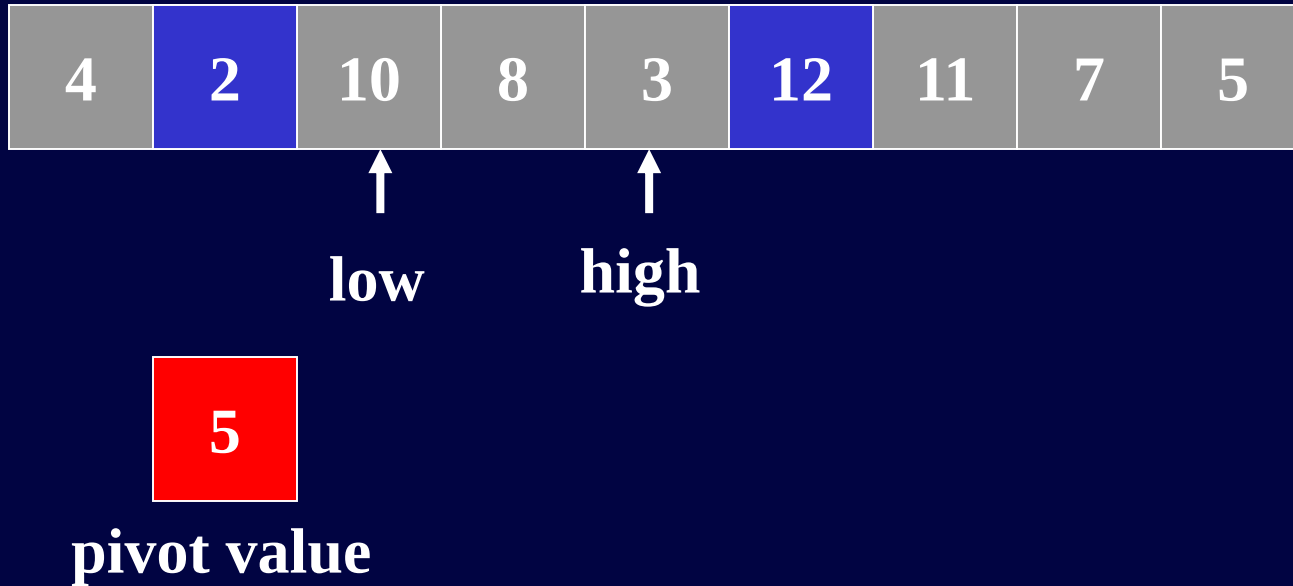
# Quicksort

Then the high index moves left until it is at an element that is smaller than the pivot value (i.e., it is on the wrong side)



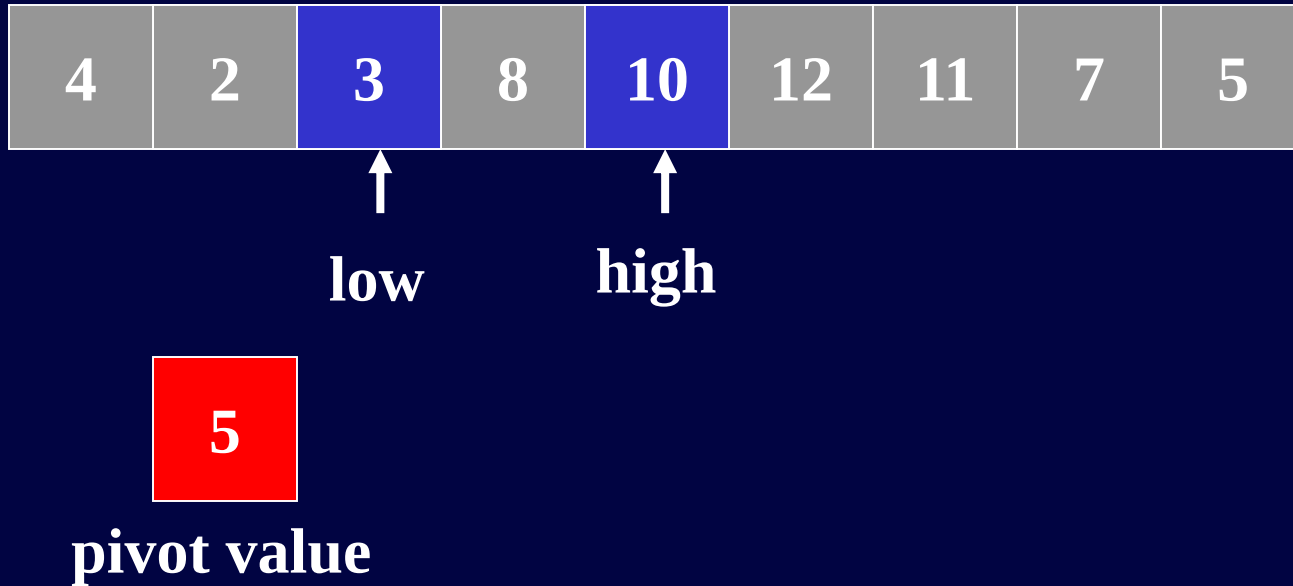
# Quicksort

Then the two values are swapped and the index values are updated:



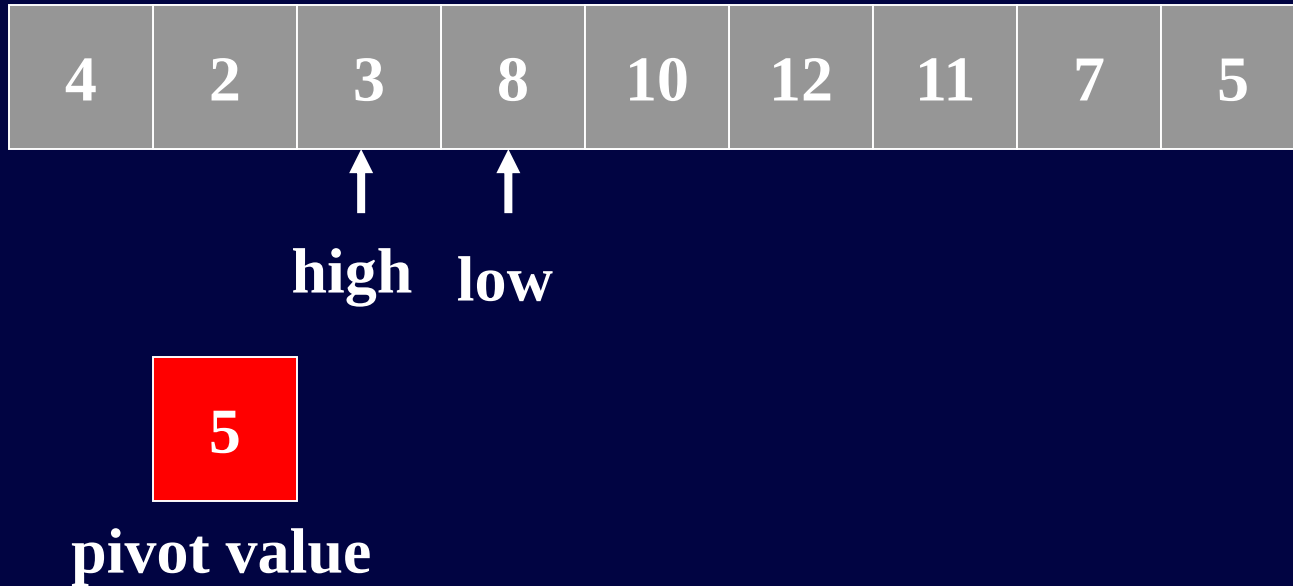
# Quicksort

This continues until the two index values pass each other:



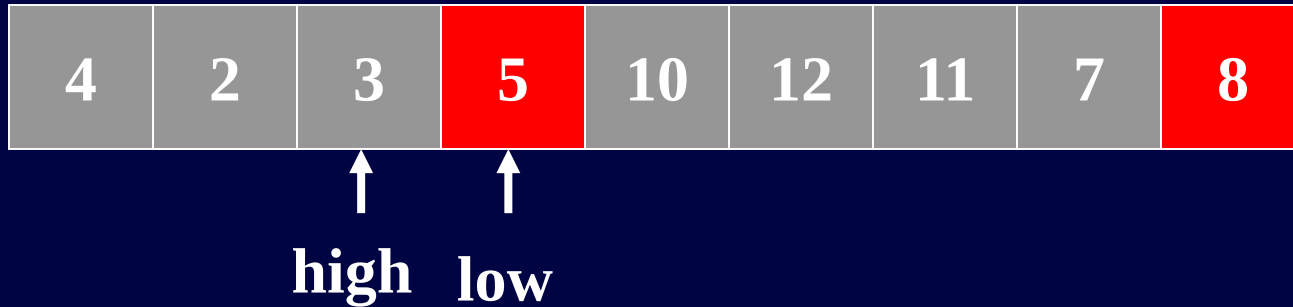
# Quicksort

This continues until the two index values pass each other:



# Quicksort

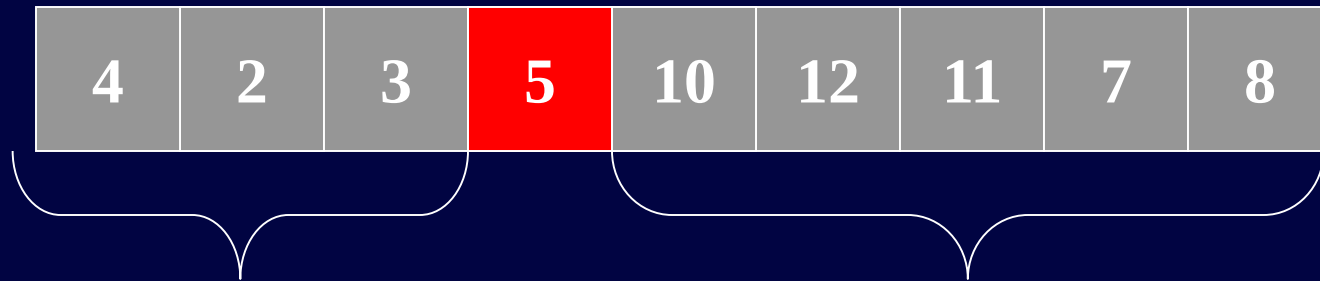
Then the pivot value is swapped into position:





# Quicksort

Recursively quicksort the two parts:



Quicksort the left part

Quicksort the right part

# Quicksort

```
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);
    }
}
```

# Quicksort

```
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;
}
```

# Data Structures-Course Recap

- Arrays
- Link Lists
- Stacks
- Queues
- Binary Trees
- Sorting