

Data Structures and Algorithms

# Lecture 8

# SECTION 1

## TREES

# Tree Data Structure

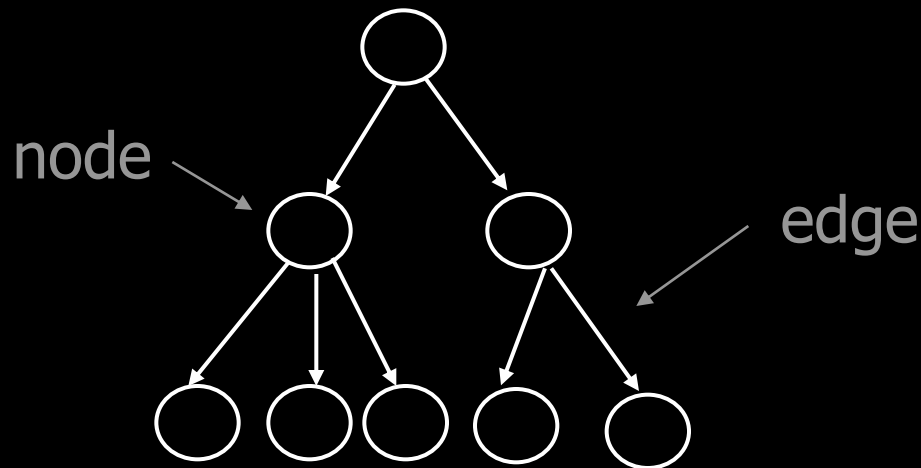
- A linear linked list will not be able to capture the tree-like relationship with ease.
- Shortly, we will see that for applications that require searching, linear data structures are not suitable.
- We will focus our attention on *binary trees*.

# Binary Tree

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

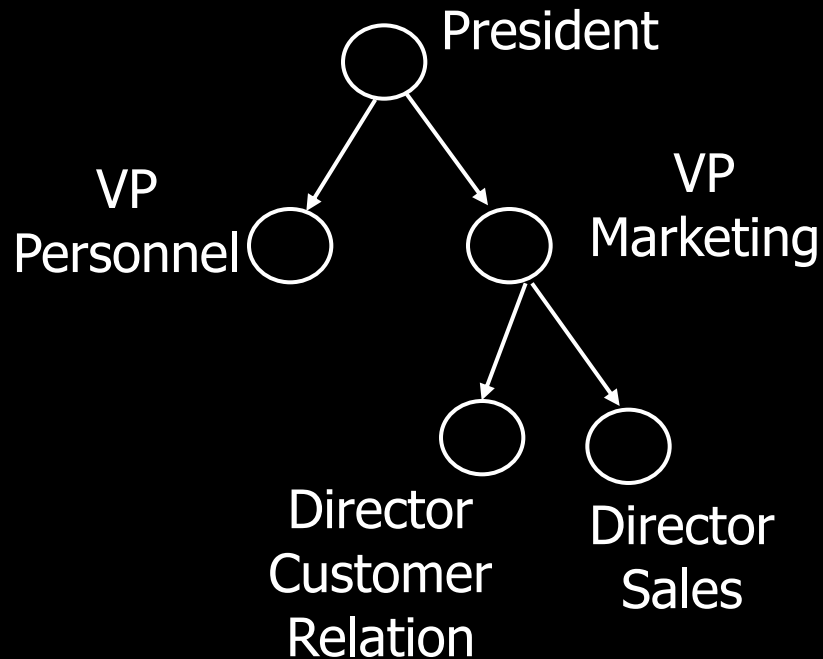
# What is a tree?

- Trees are structures used to represent hierarchical relationship
- Each tree consists of nodes and edges
- Each node represents an object
- Each edge represents the relationship between two nodes.

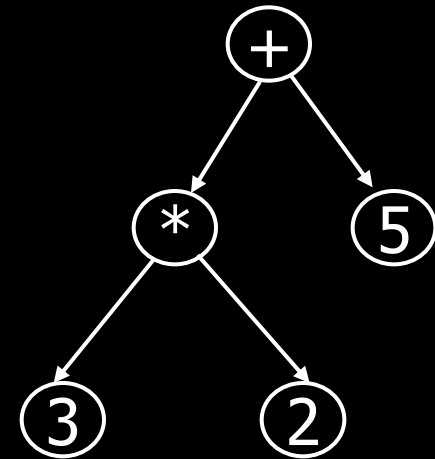


# Some applications of Trees

## Organization Chart

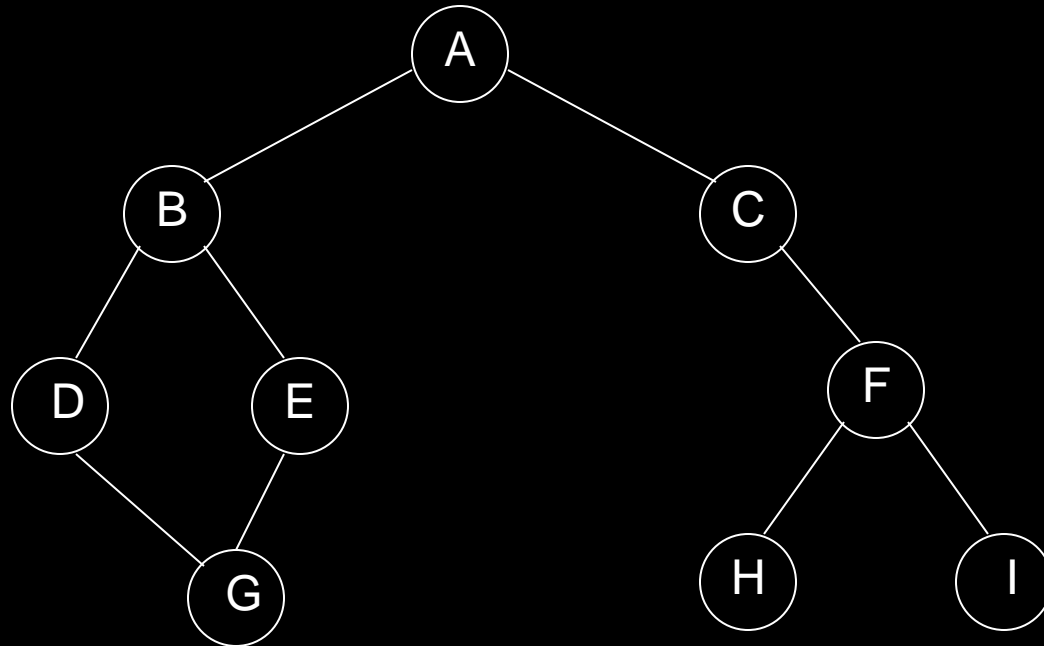


## Expression Tree



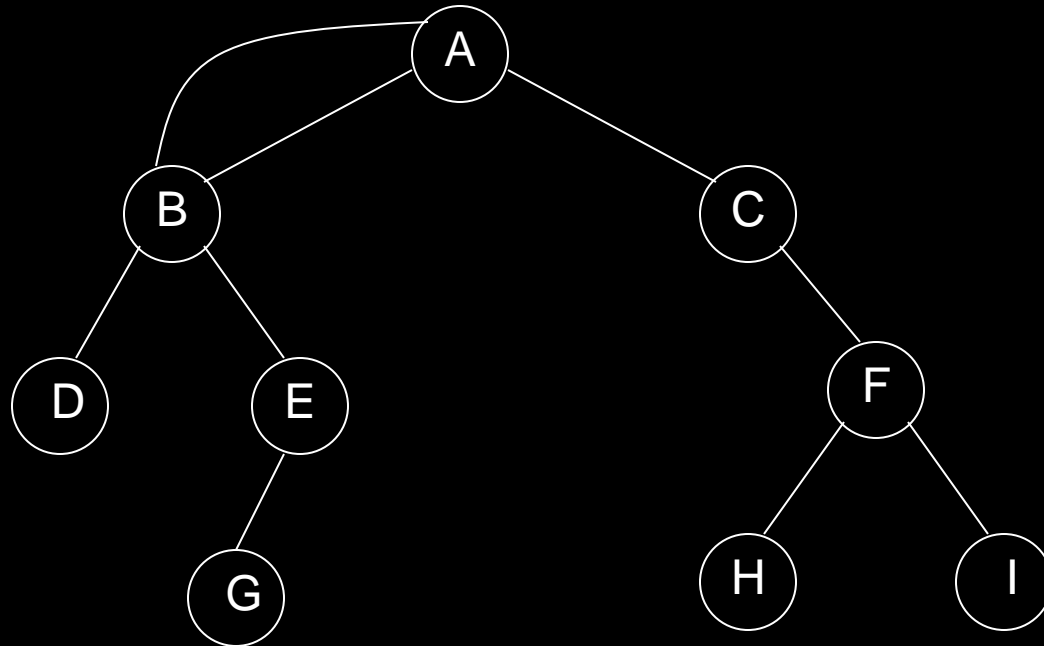
# Not a Tree

- Structures that are not trees.



# Not a Tree

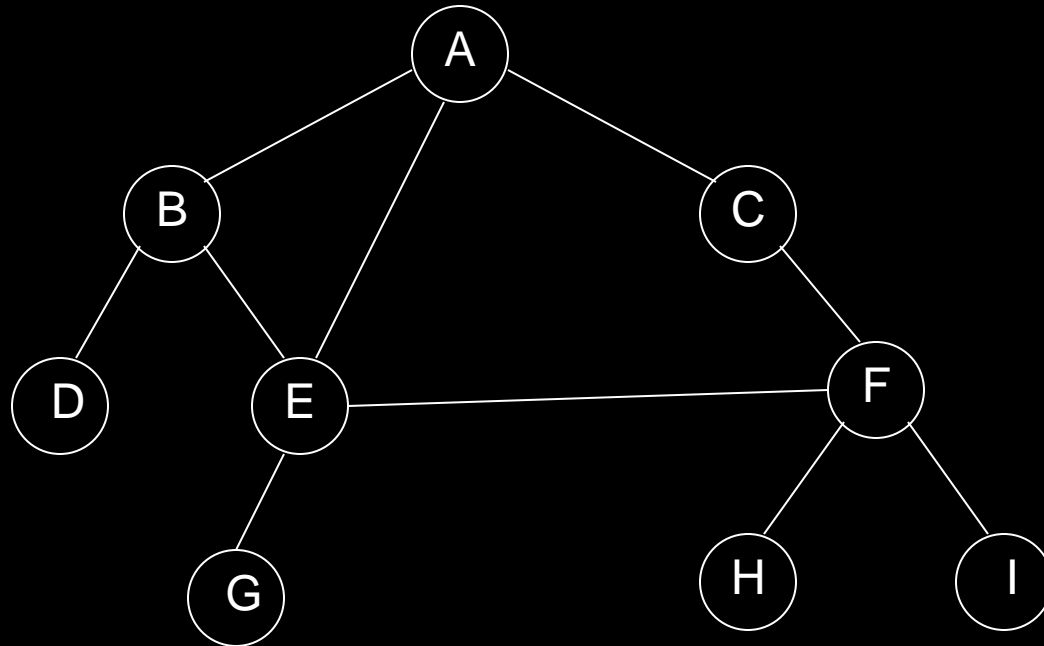
- Structures that are not trees.





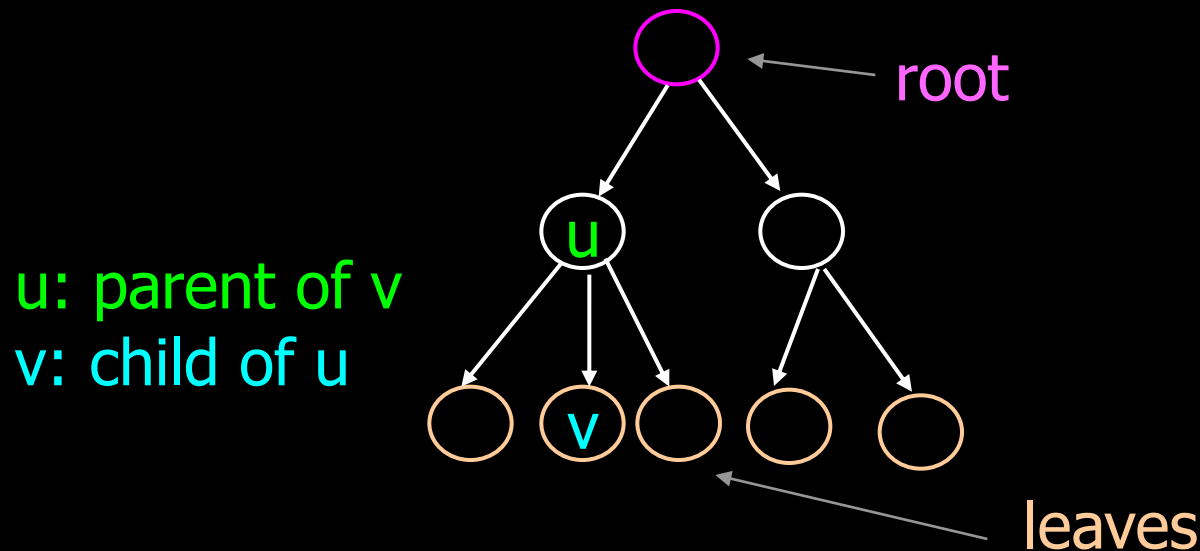
# Not a Tree

- Structures that are not trees.



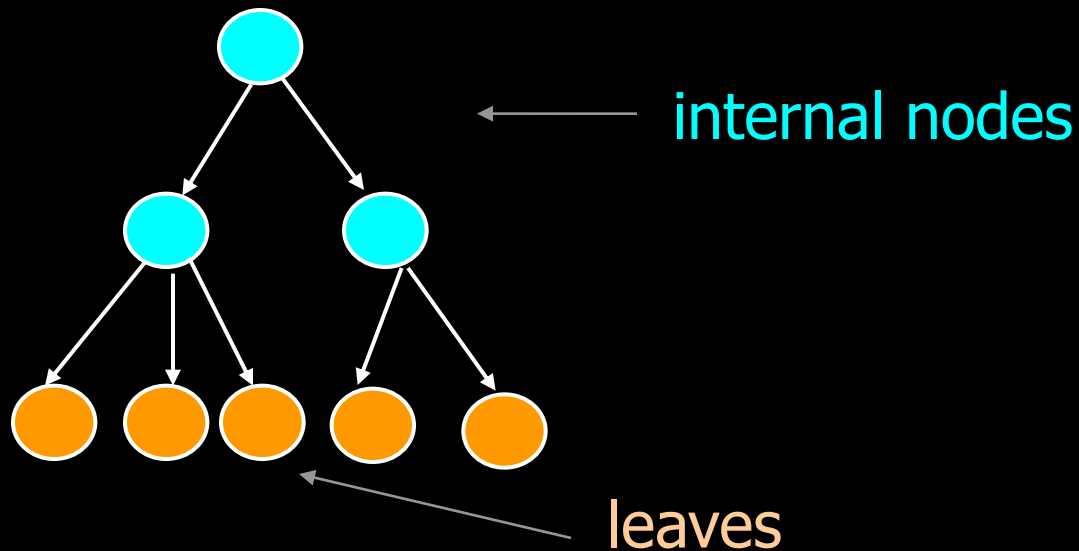
# Terminology I

- For any two nodes  $u$  and  $v$ , if there is an edge pointing from  $u$  to  $v$ ,  $u$  is called the **parent** of  $v$  while  $v$  is called the **child** of  $u$ . Such edge is denoted as  $(u, v)$ .
- In a tree, there is exactly one node without parent, which is called the **root**. The nodes without children are called **leaves**.



# Terminology II

- In a tree, the nodes without children are called **leaves**. Otherwise, they are called **internal nodes**.



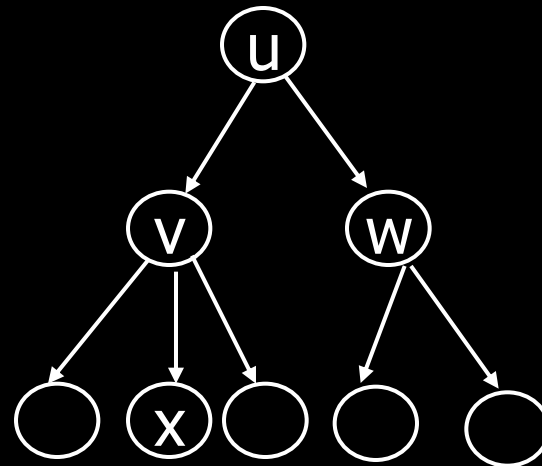
# Terminology III

- If two nodes have the same parent, they are **siblings**.
- A node  $u$  is an **ancestor** of  $v$  if  $u$  is parent of  $v$  or parent of parent of  $v$  or ...
- A node  $v$  is a **descendent** of  $u$  if  $v$  is child of  $u$  or child of child of  $u$  or ...

**$v$  and  $w$  are siblings**

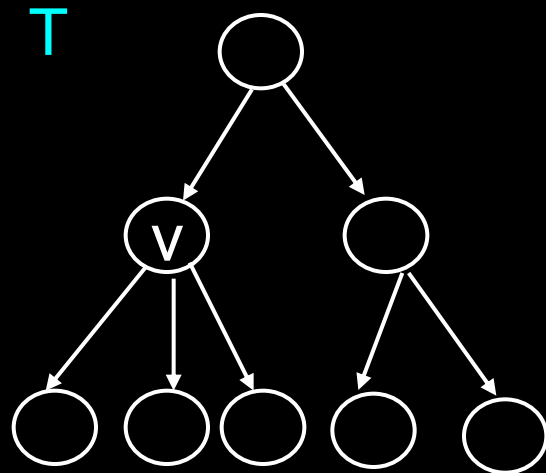
**$u$  and  $v$  are ancestors of  $x$**

**$v$  and  $x$  are descendants of  $u$**

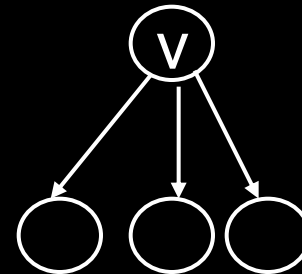


# Terminology IV

- A **subtree** is any node together with all its descendants.

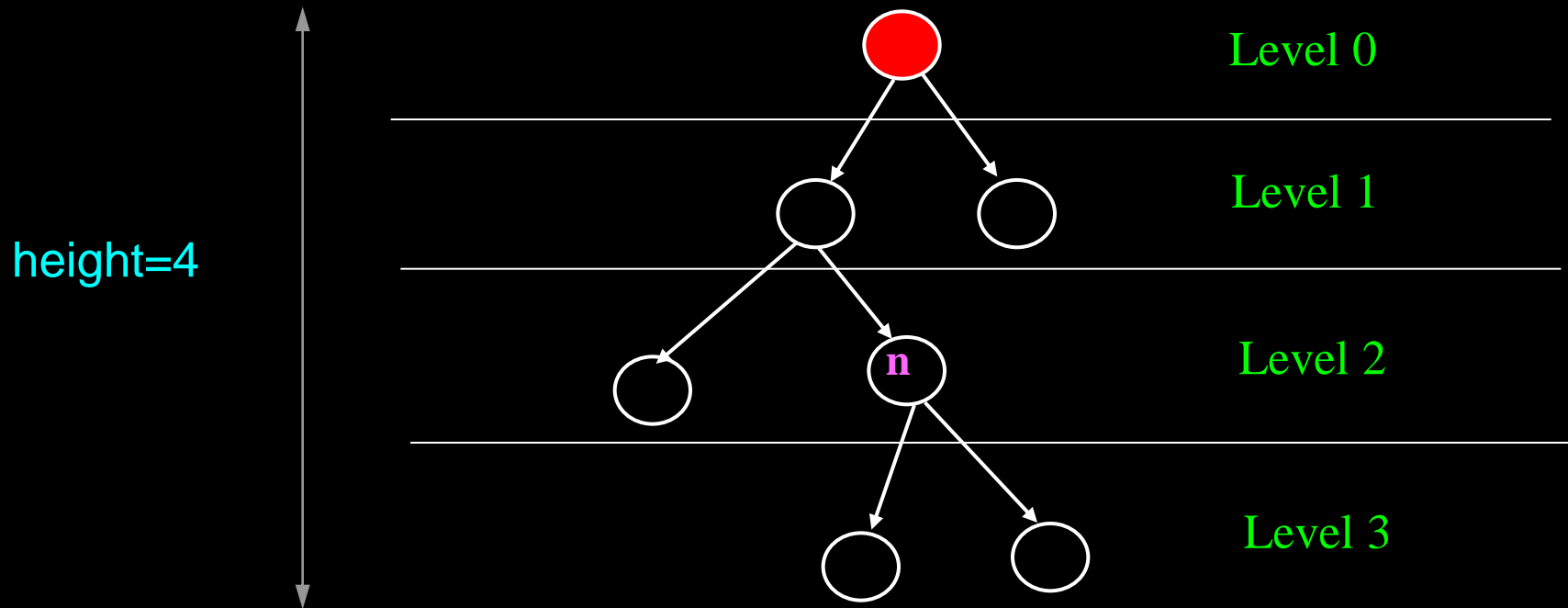


A subtree of T



# Terminology V

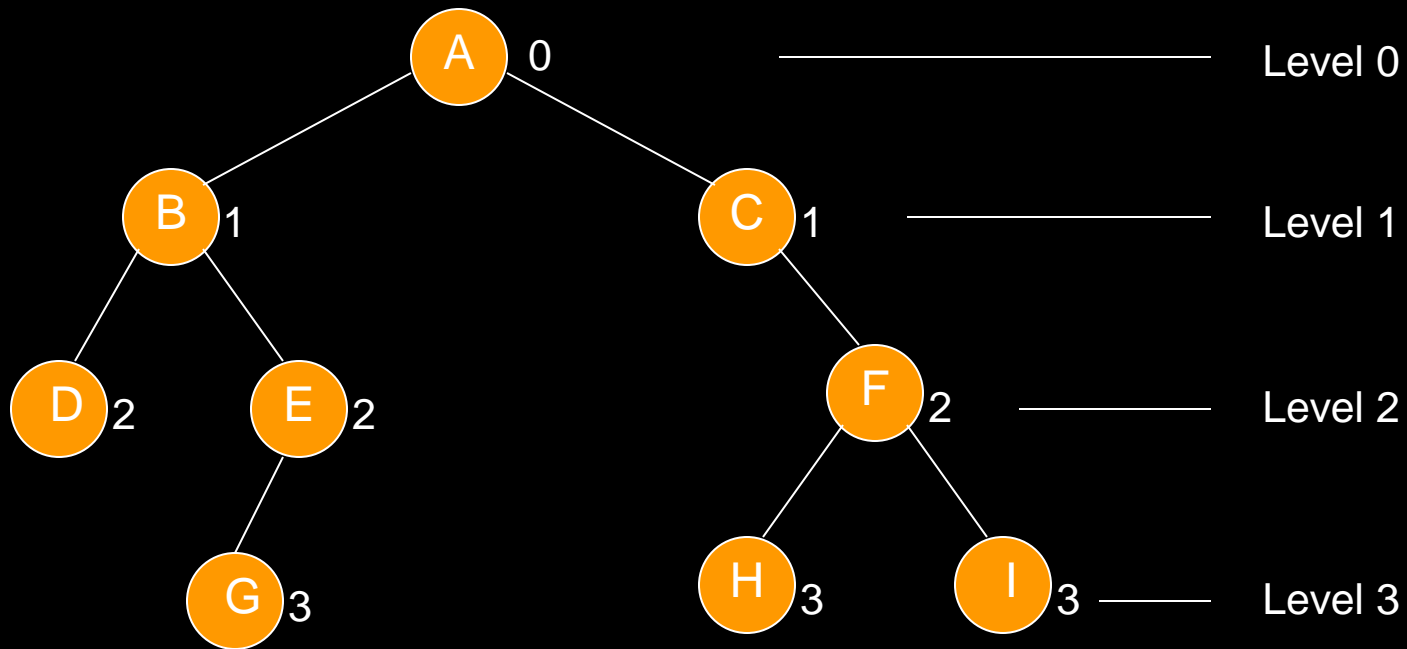
- **Level of a node  $n$ :** number of nodes on the path from root to node  $n$
- **Height of a tree:** maximum level among all of its node
- **The *depth* of tree** is the maximum level of any leaf in the tree.



# Terminology V

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

# Terminology V



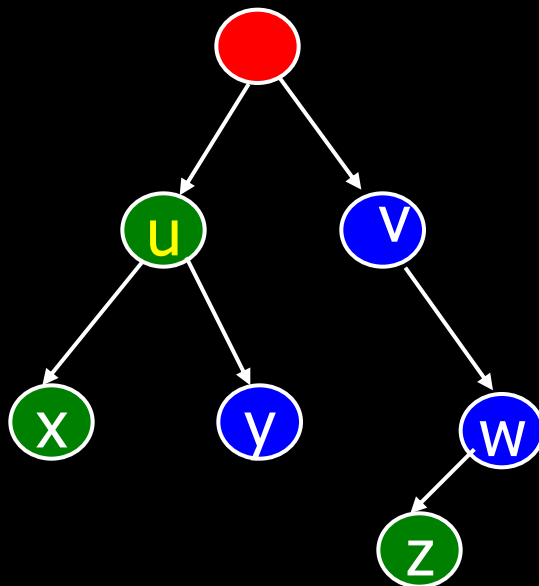


# Terminology VI & VII

- The **Degree** of a Node is the number of it's children
- A tree is said to be **full** if all of it's internal nodes have the same degree and all of it's leaves are on the same level

# Binary Tree

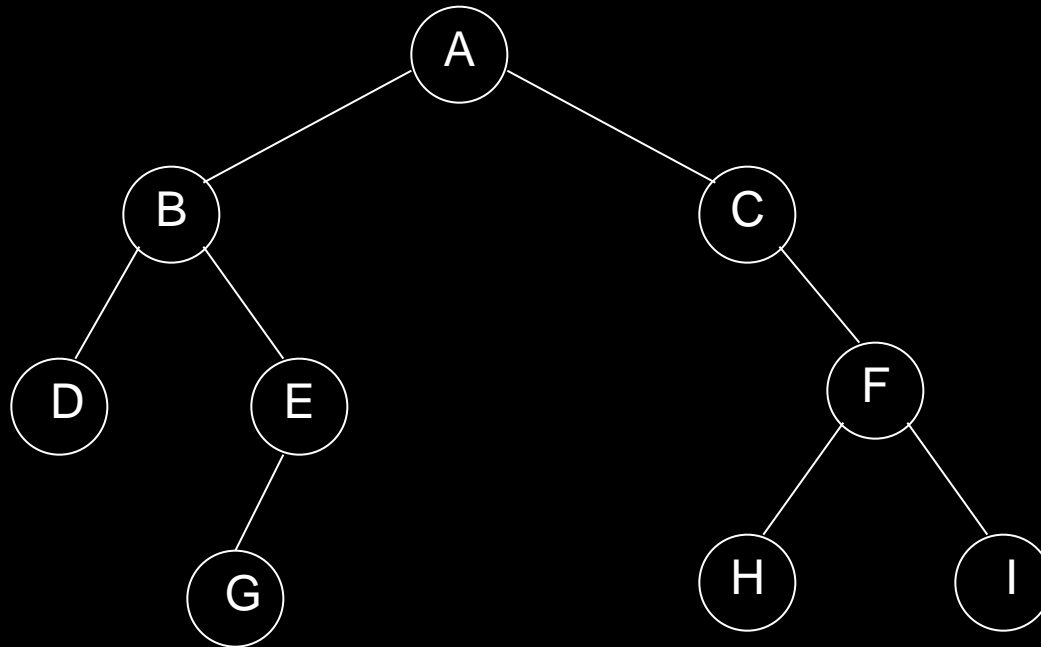
- Binary Tree: Tree in which every node has at most 2 children
- Left child of u: the child on the left of u
- Right child of u: the child on the right of u



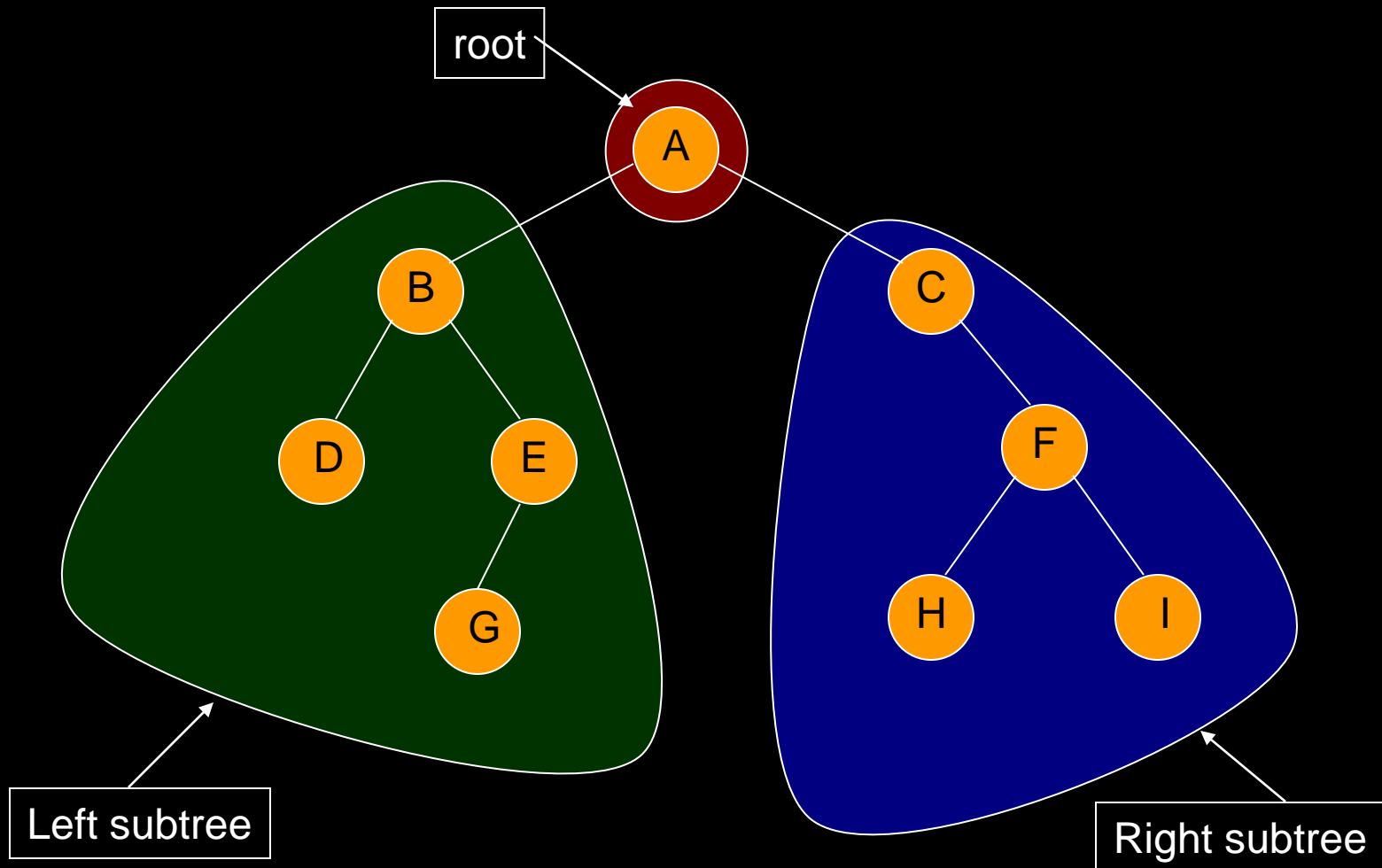
x: left child of u  
y: right child of u  
w: right child of v  
z: left child of w

# Binary Tree

- Binary tree with 9 nodes.

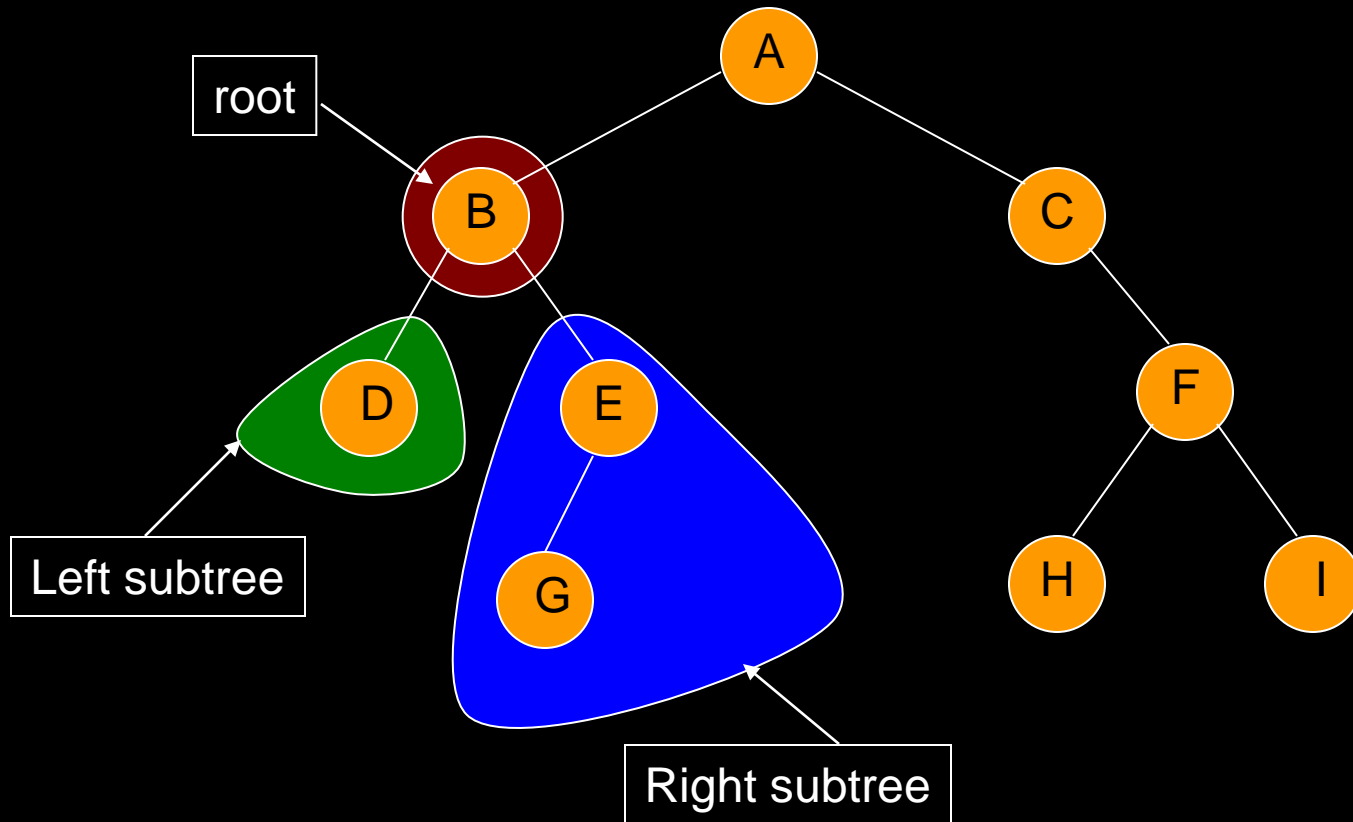


# Binary Tree



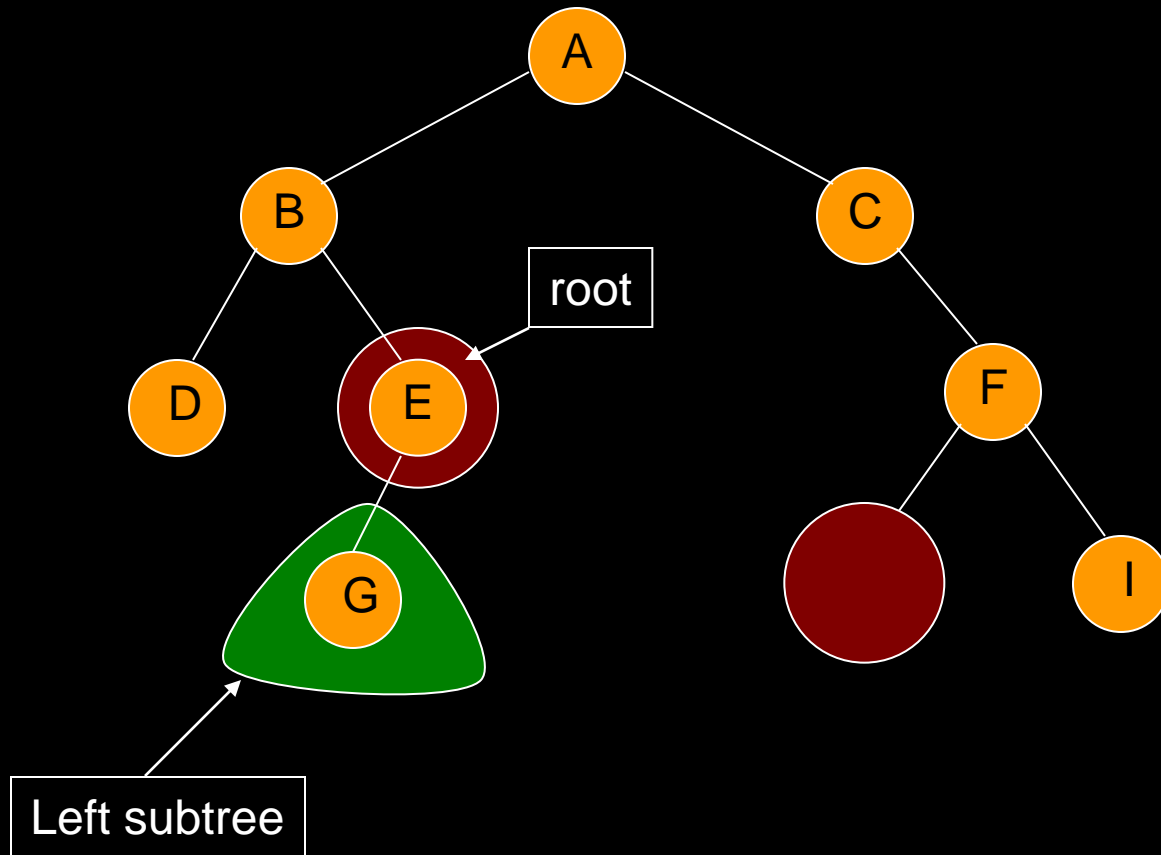
# Binary Tree

- Recursive definition



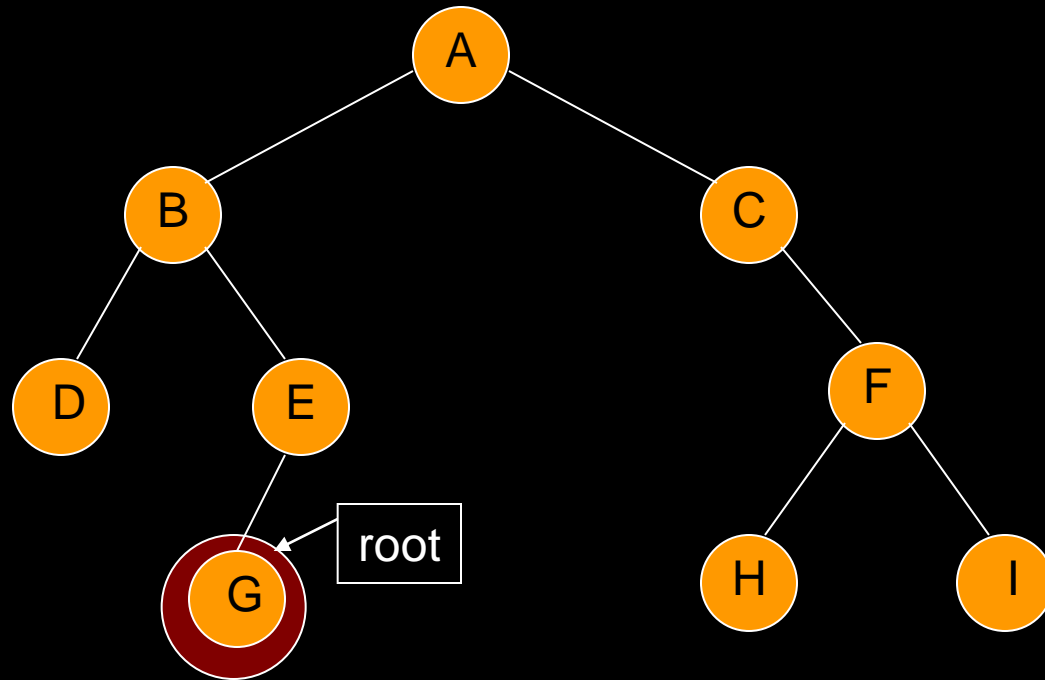
# Binary Tree

- Recursive definition



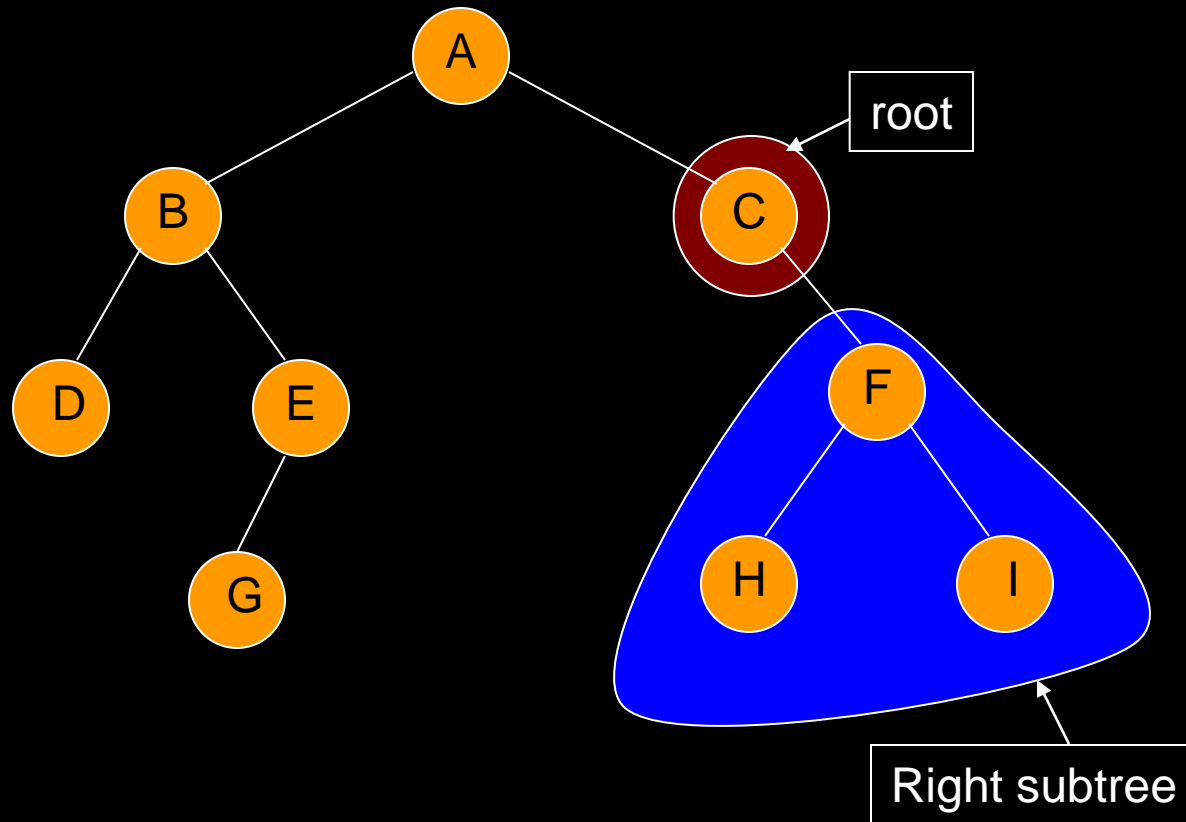
# Binary Tree

- Recursive definition



# Binary Tree

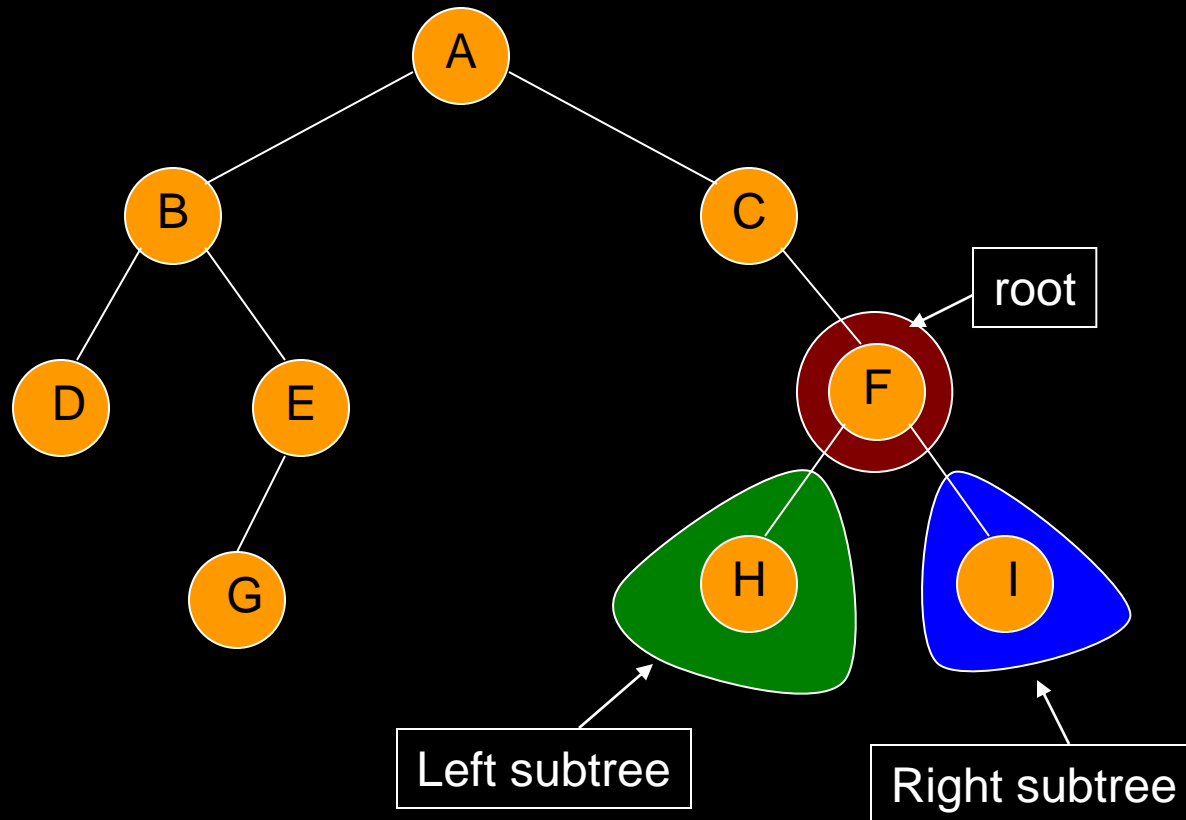
- Recursive definition





# Binary Tree

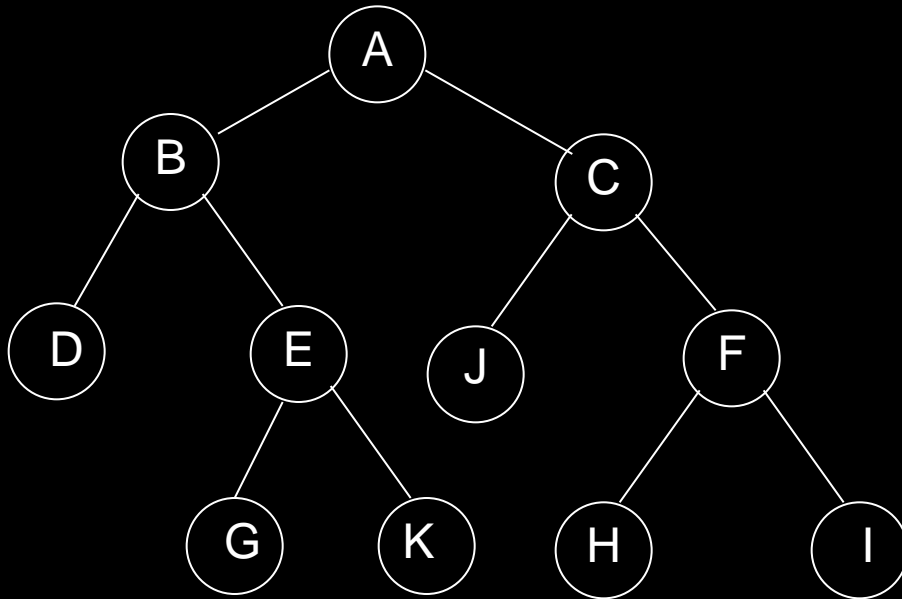
- Recursive definition



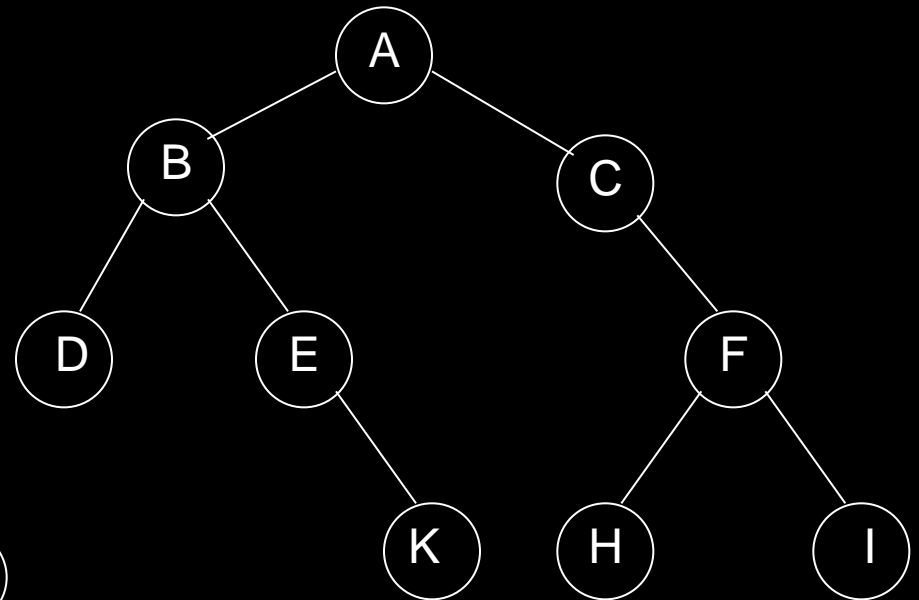
# Strict Binary Tree

- If every non-leaf node in a binary tree has non-empty left and right subtrees, the tree is termed a *strict binary tree*.

strict binary tree

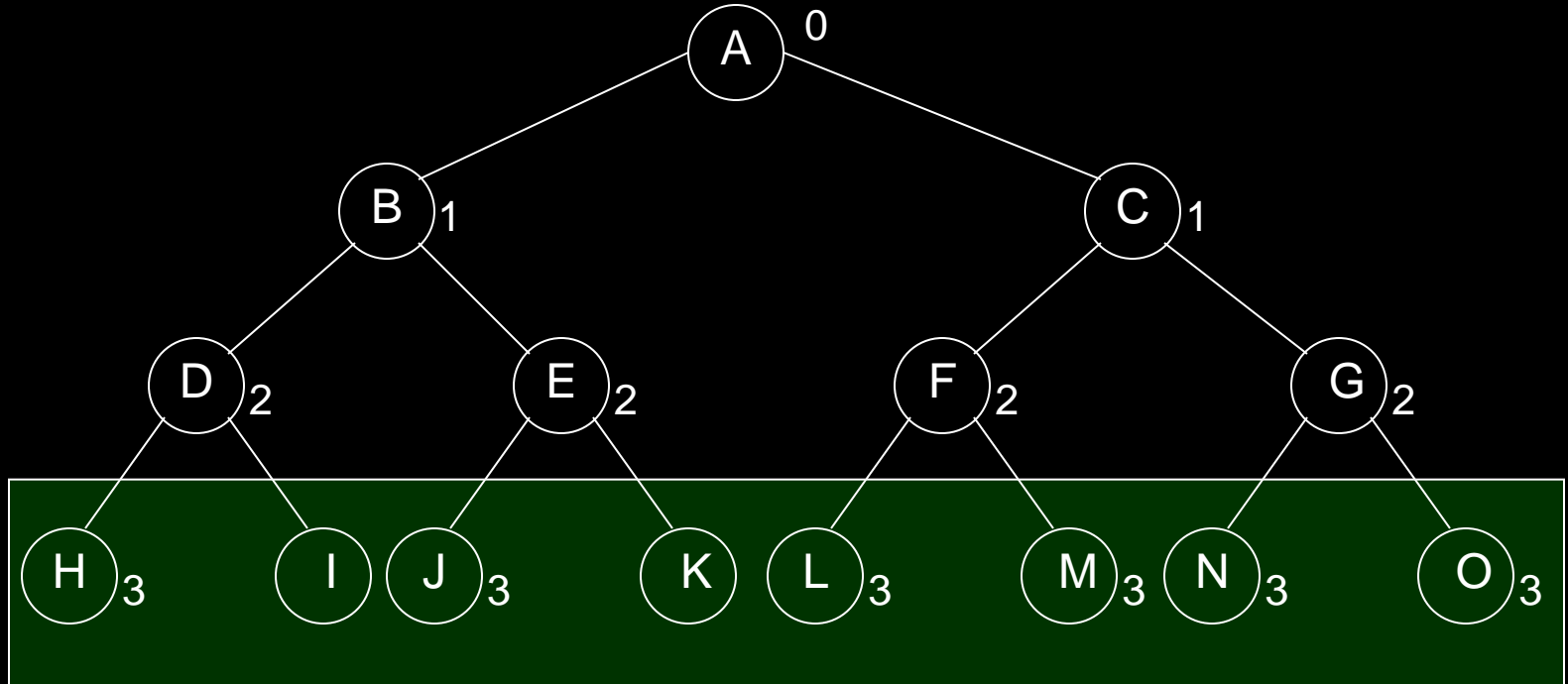


Not a strict binary tree

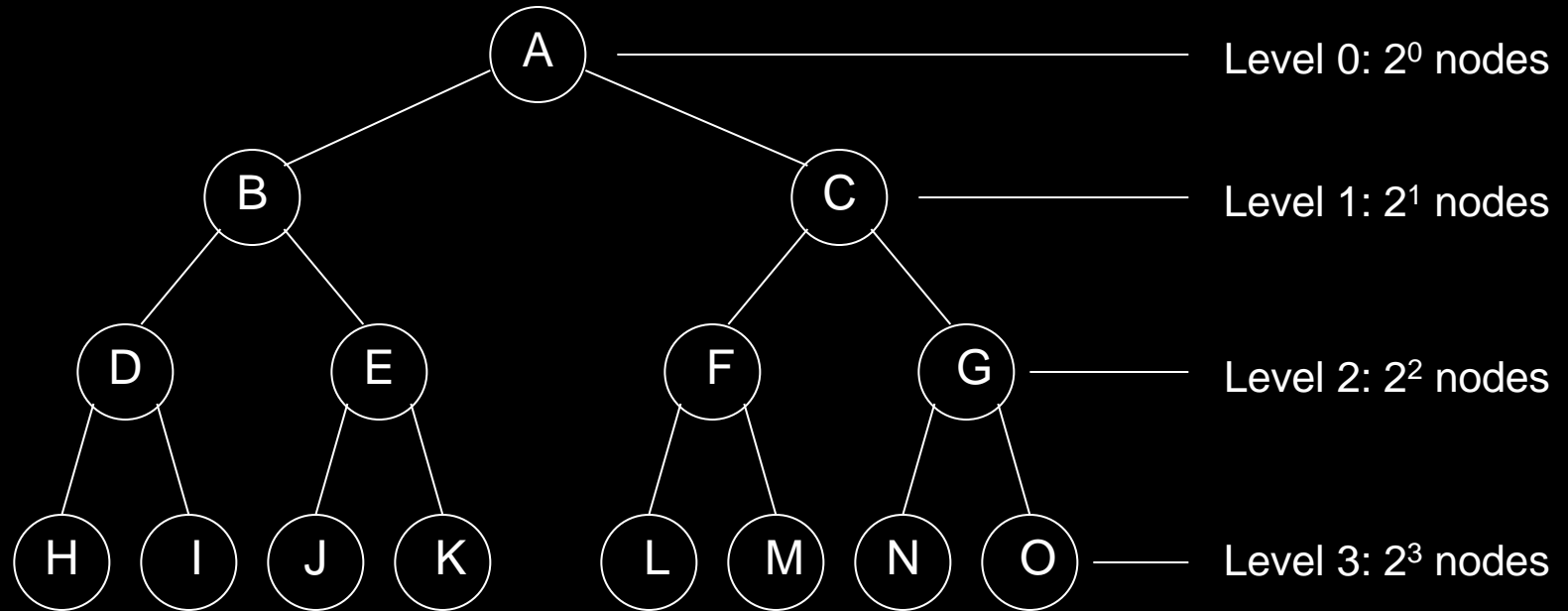


# Full Binary Tree

- A *Full binary tree* of depth  $d$  is the strictly binary all of whose leaves are at level  $d$ .



# Full Binary Tree



# Full Binary Tree

At level  $k$ , there are  $2^k$  nodes.

Total number of nodes in the tree of depth  $d$ :

$$2^0 + 2^1 + 2^2 + \dots + 2^d = \sum_{j=0}^d 2^j = 2^{d+1} - 1$$

- In a complete binary tree, there are  $2^d$  leaf nodes and  $(2^d - 1)$  non-leaf (inner) nodes.

# Full Binary Tree

If the tree is built out of 'n' nodes then

$$\begin{aligned} & n = 2^{d+1} - 1 \\ \text{or} \quad & \log_2(n+1) = d+1 \\ \text{or} \quad & d = \log_2(n+1) - 1 \end{aligned}$$

i.e., the depth of the complete binary tree built using 'n' nodes will be  $\log_2(n+1) - 1$ .

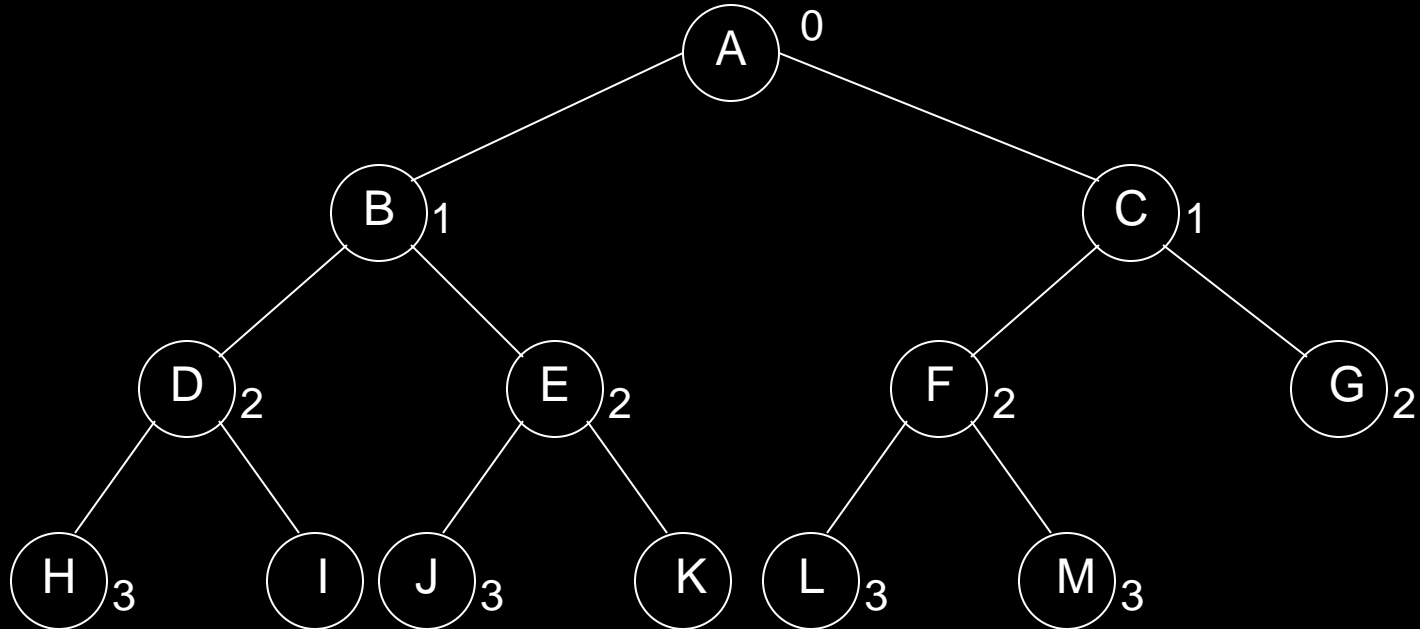
For example, for  $n=100,000$ ,  $\log_2(100001)$  is less than 20; the tree would be 20 levels deep.

The significance of this shallowness will become evident later.

# Complete Binary Tree

A *Complete Binary Tree* is either

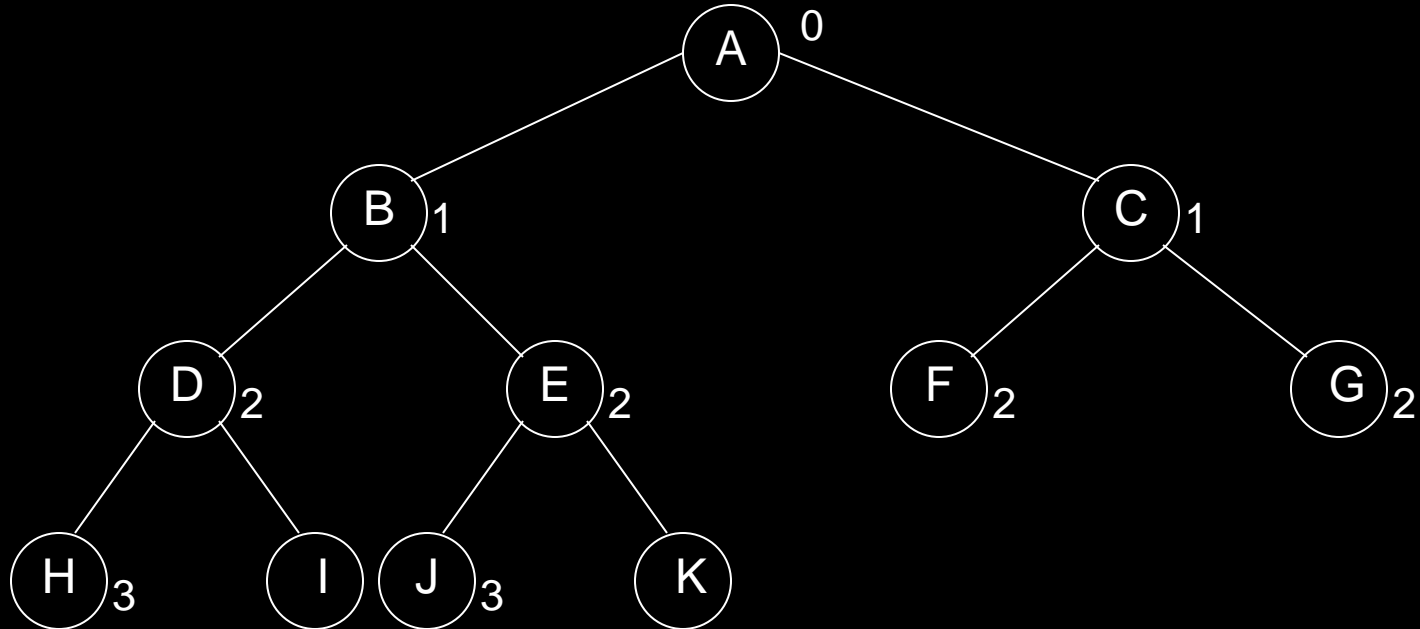
- Full Binary tree
- Can be made into full tree by adding leaves only in an uninterrupted segment of nodes in the right of the bottom level



# Complete Binary Tree

A *Complete Binary Tree* is either

- Full Binary tree
- Can be made into full tree by adding leaves only in an uninterrupted segment of nodes in the right of the bottom level

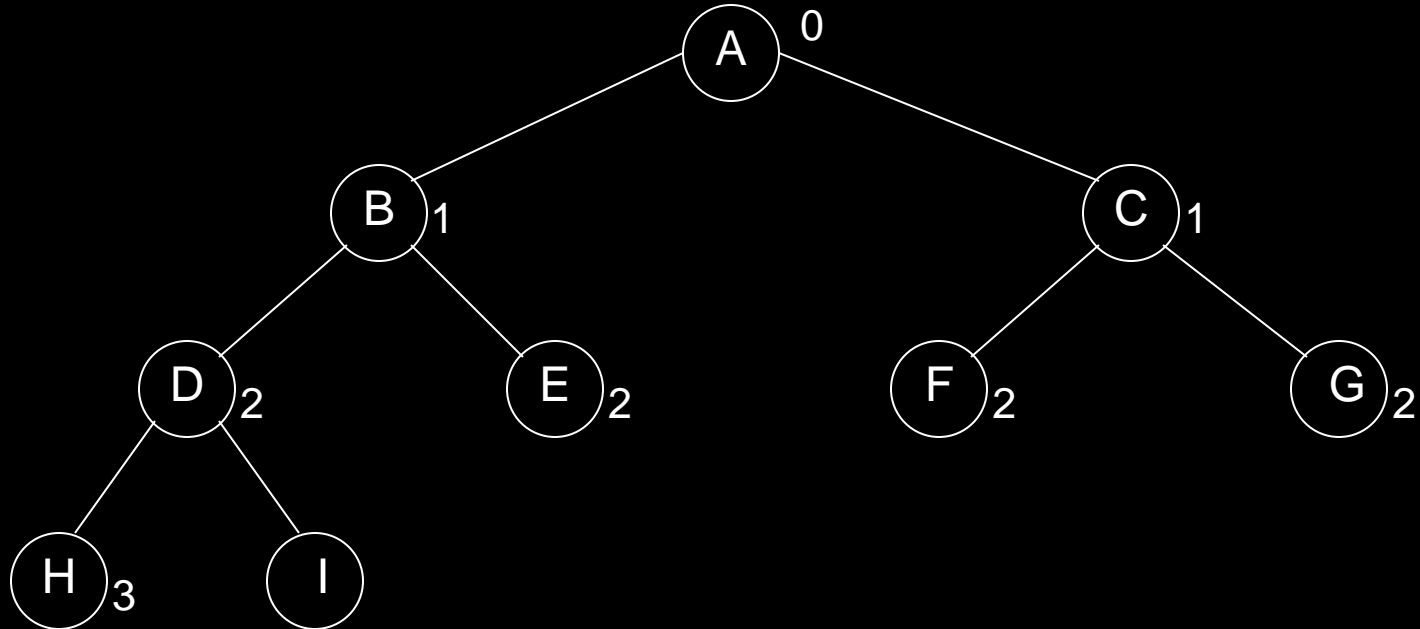




# Complete Binary Tree

A *Complete Binary Tree* is either

- Full Binary tree
- Can be made into full tree by adding leaves only in an uninterrupted segment of nodes in the right of the bottom level



# Binary Search Tree

A special kind of binary tree in which:

1. Each node contains a distinct data value,
2. The key values in the tree can be compared using “greater than” and “less than”, and
3. The key value of each node in the tree is **less than every key value in its right subtree**, and **greater than every key value in its left subtree**.

# Operations on Binary Tree

- There are a number of operations that can be defined for a binary tree.
- If  $p$  is pointing to a node in an existing tree then
  - $\text{left}(p)$  returns pointer to the left subtree
  - $\text{right}(p)$  returns pointer to right subtree
  - $\text{parent}(p)$  returns the father of  $p$
  - $\text{brother}(p)$  returns brother of  $p$ .
  - $\text{info}(p)$  returns content of the node.

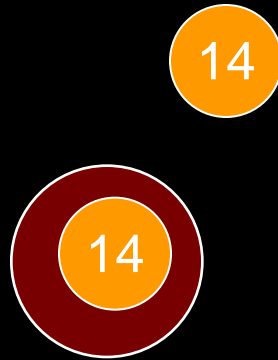
# Operations on Binary Tree

- In order to construct a binary tree, the following can be useful:
- `setLeft(p,x)` creates the left child node of p. The child node contains the info 'x'.
- `setRight(p,x)` creates the right child node of p. The child node contains the info 'x'.

# Building Binary Search Tree

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

# Building Binary Search Tree



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

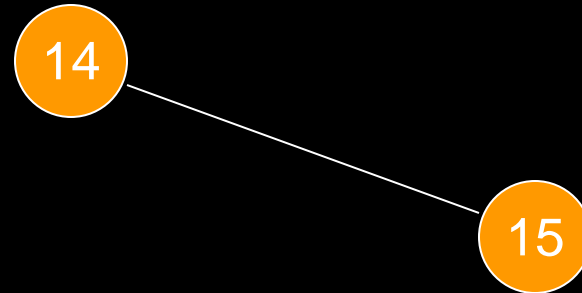
# Building Binary Search Tree

15

14

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

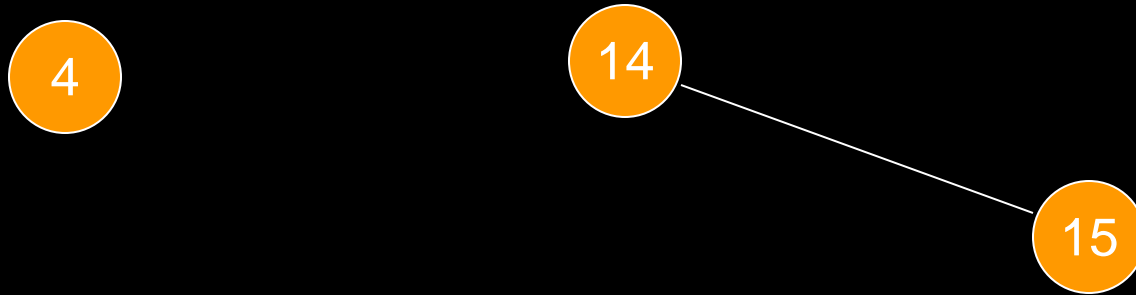
# Building Binary Search Tree



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

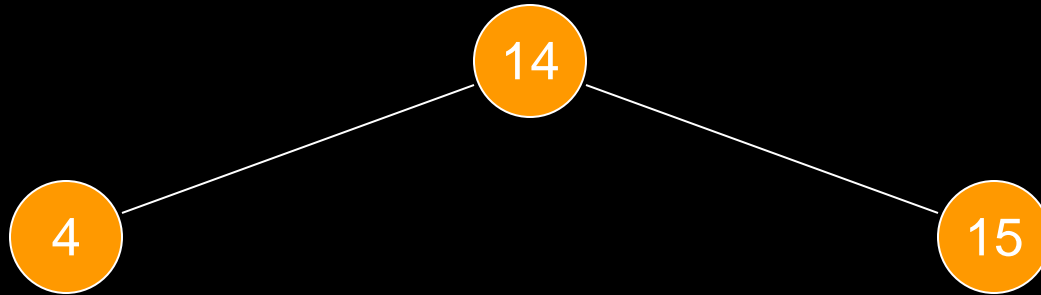


# Building Binary Search Tree



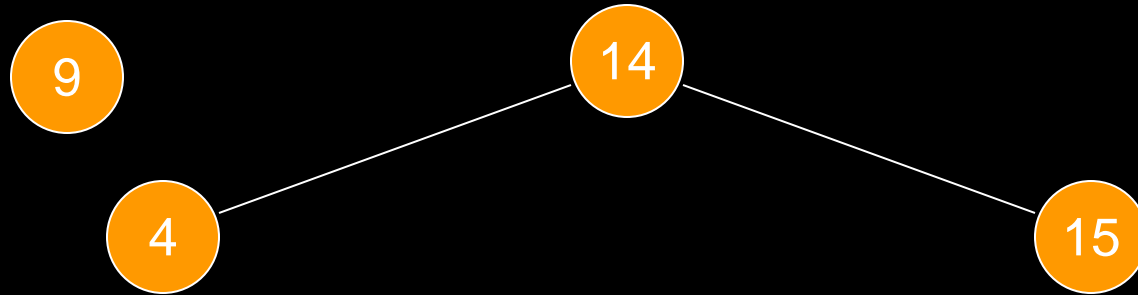
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



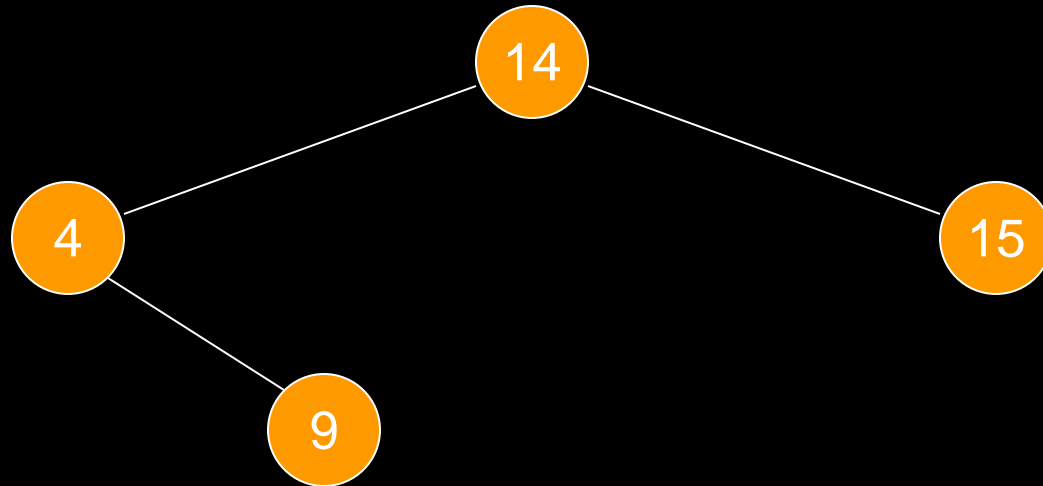
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



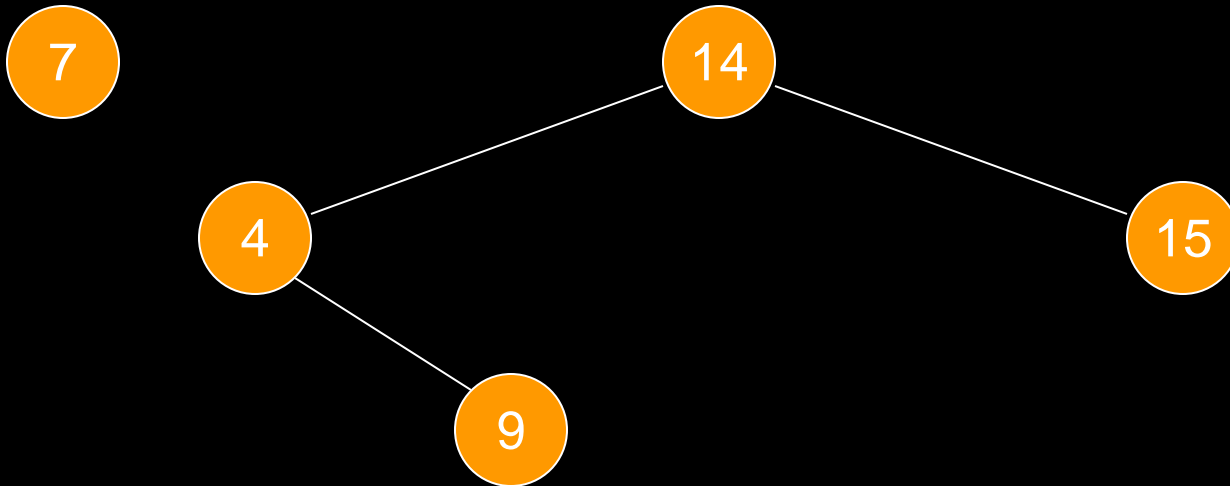
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



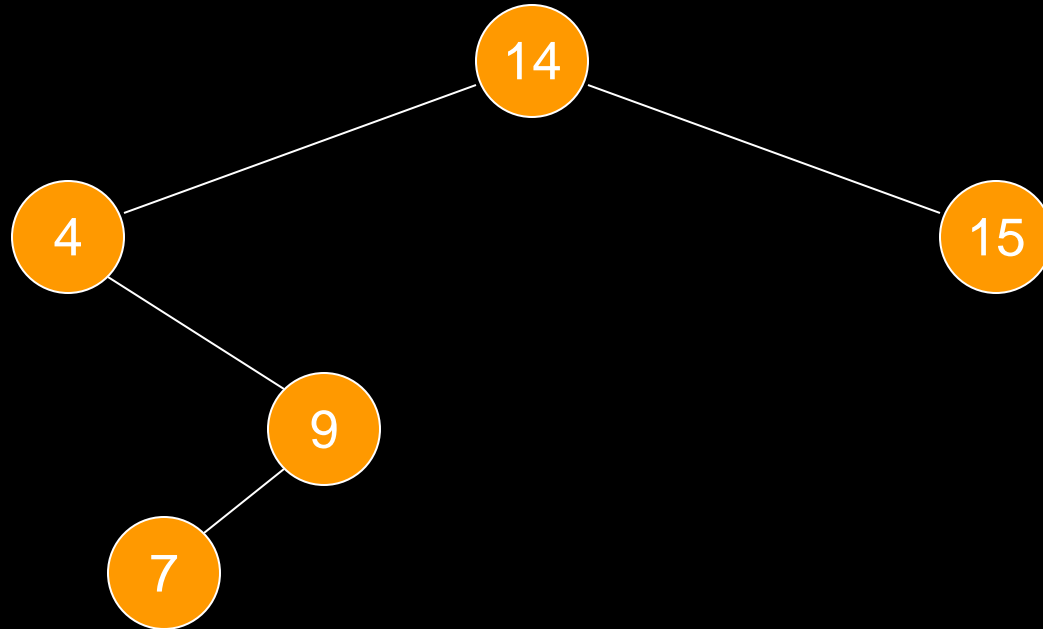
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



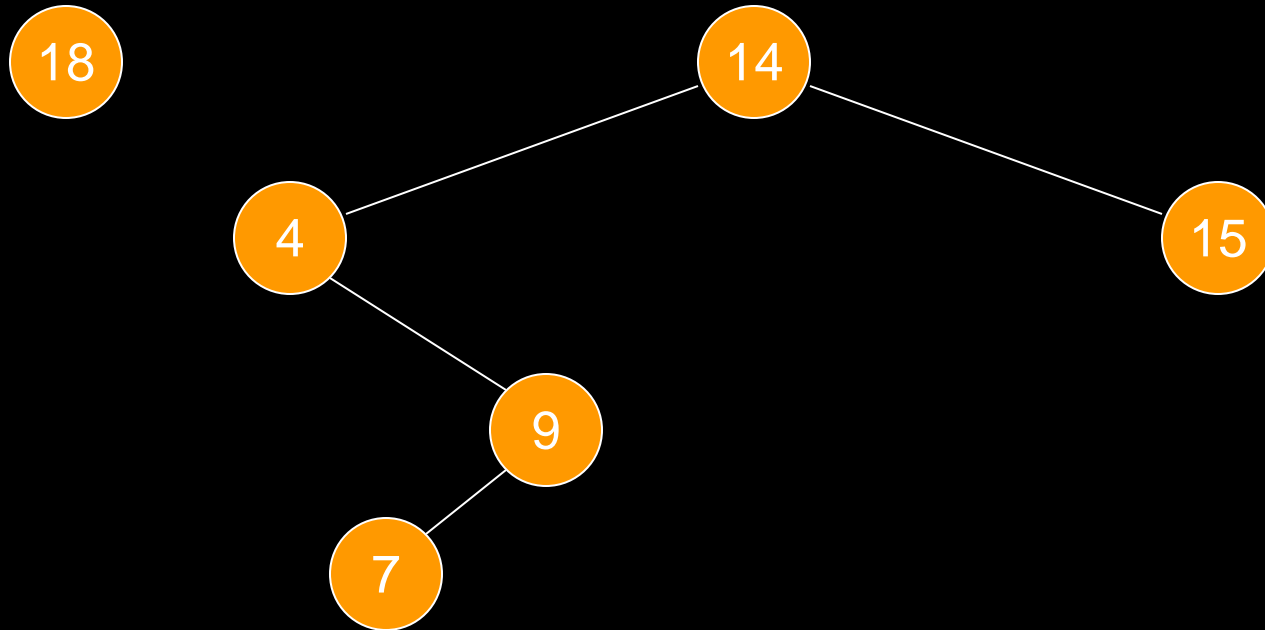
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



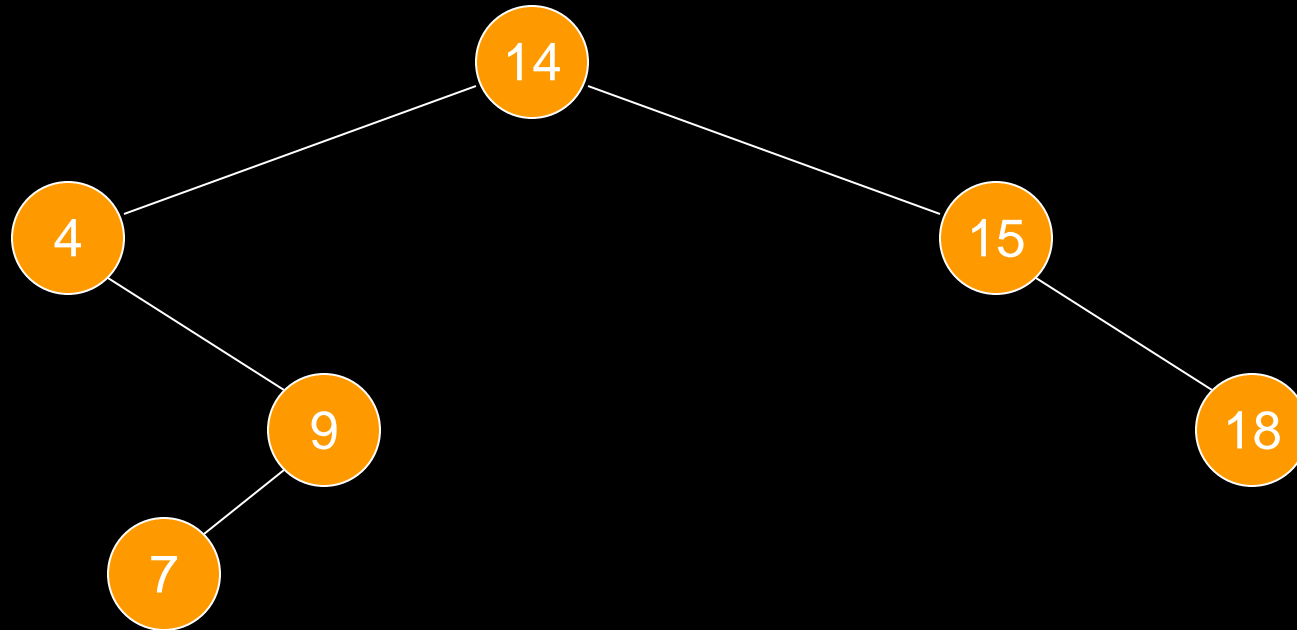
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



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

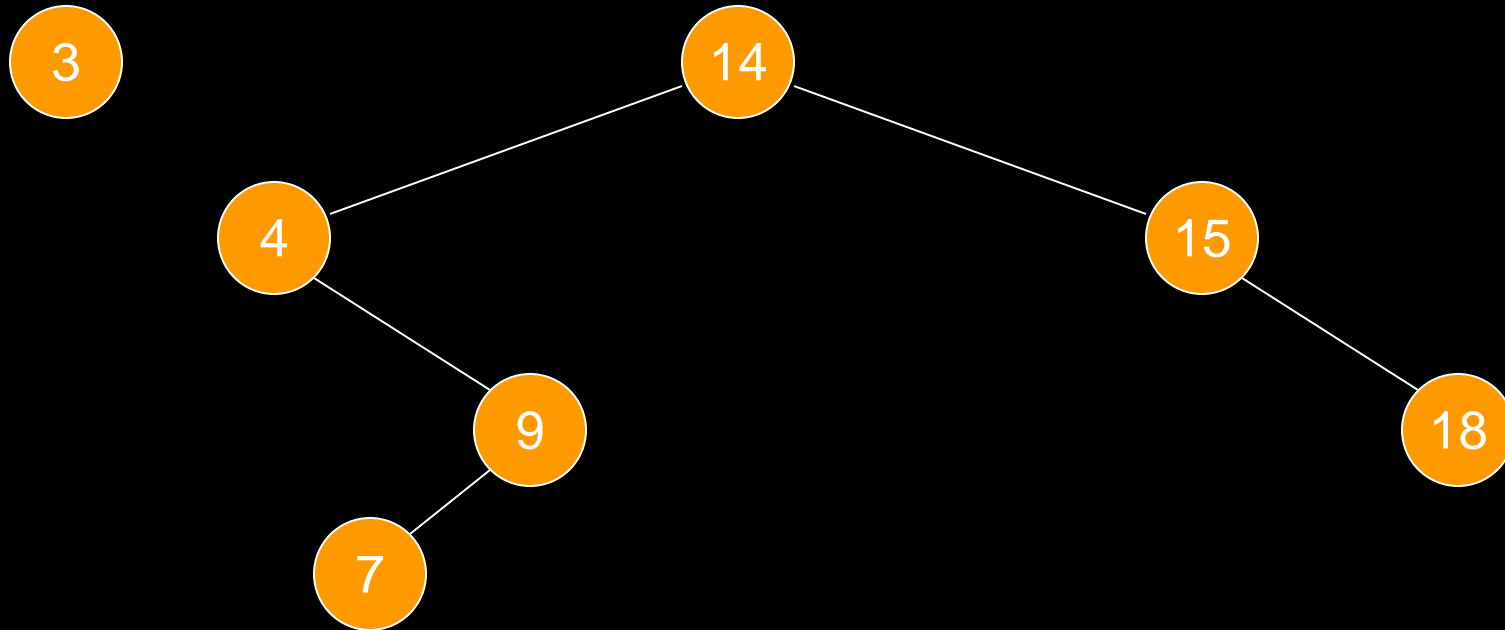
# Building Binary Search Tree



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

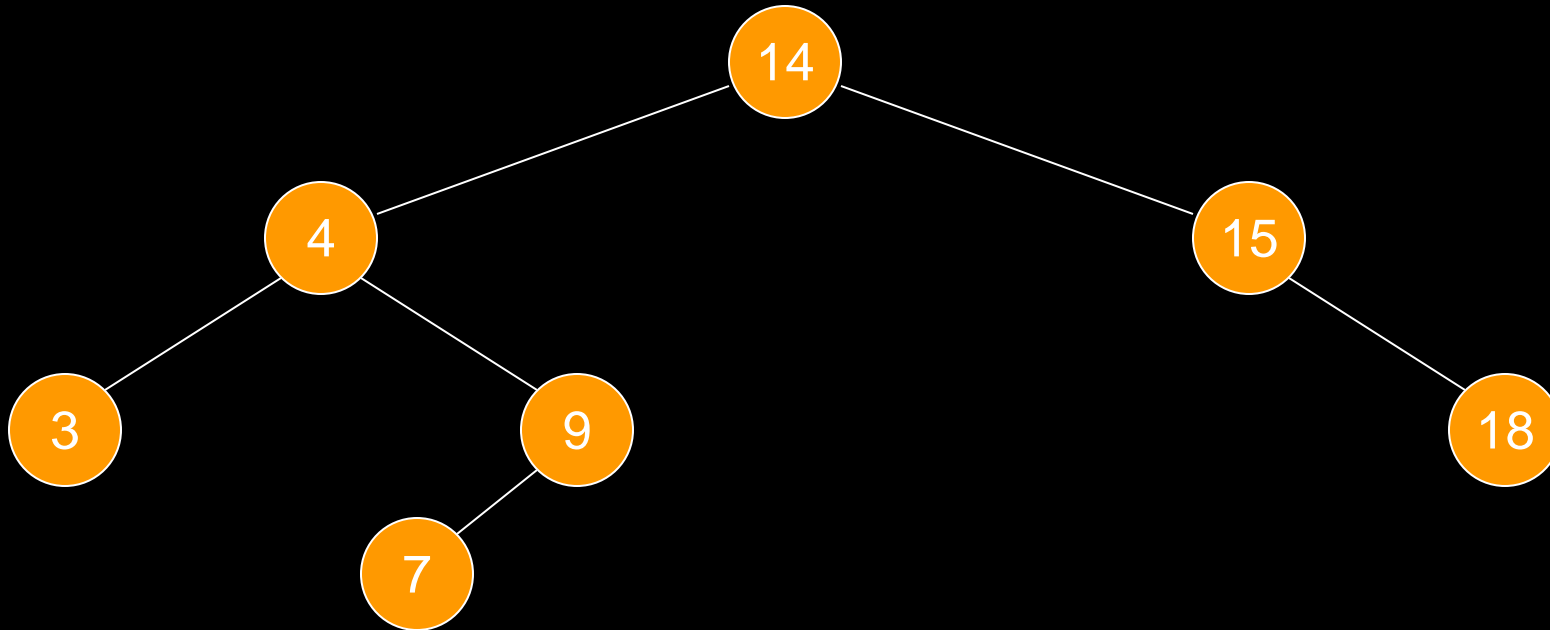


# Building Binary Search Tree



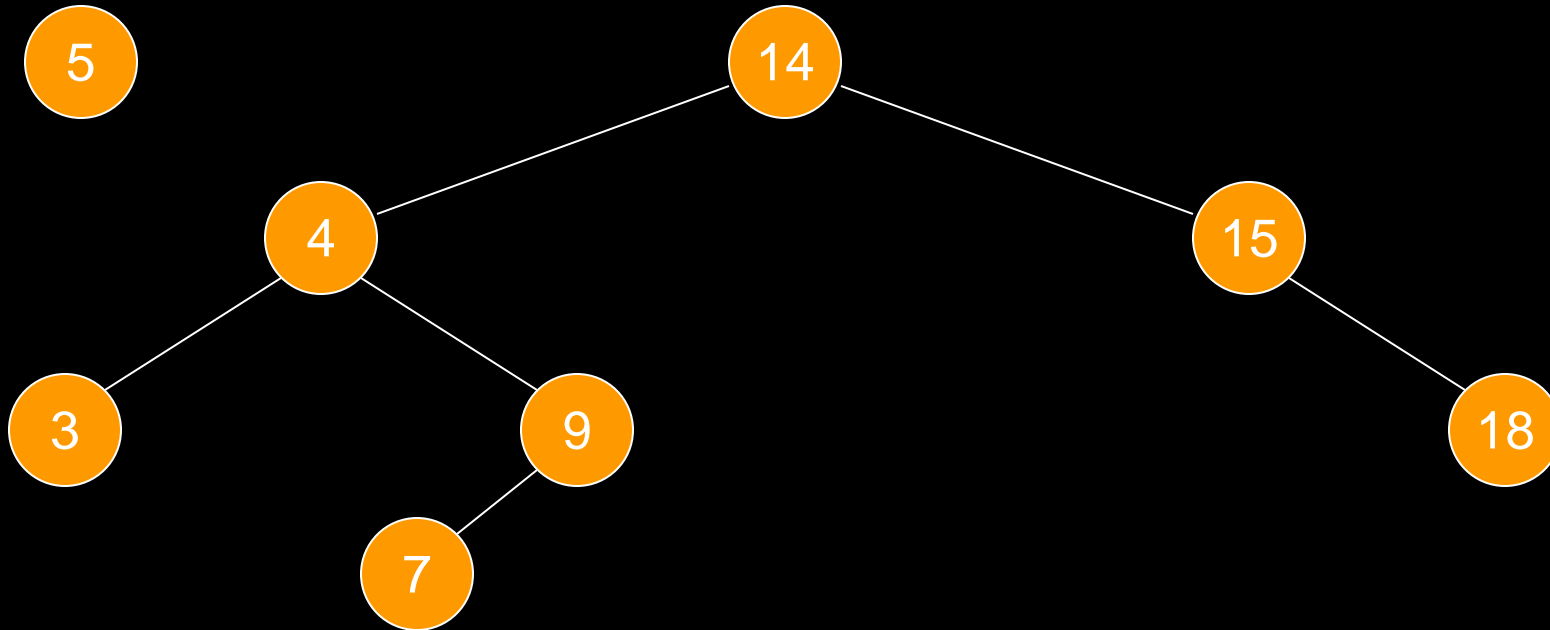
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



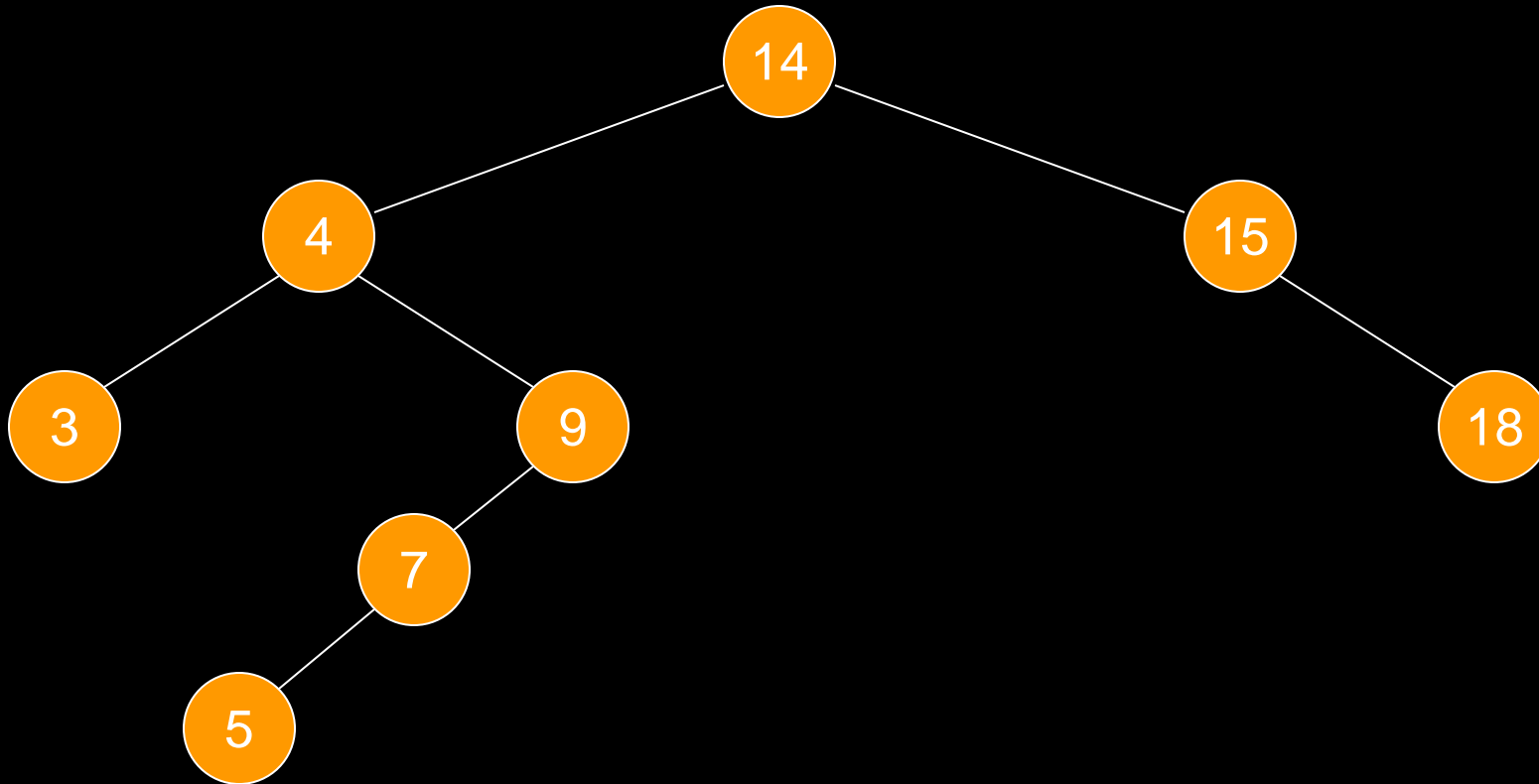
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



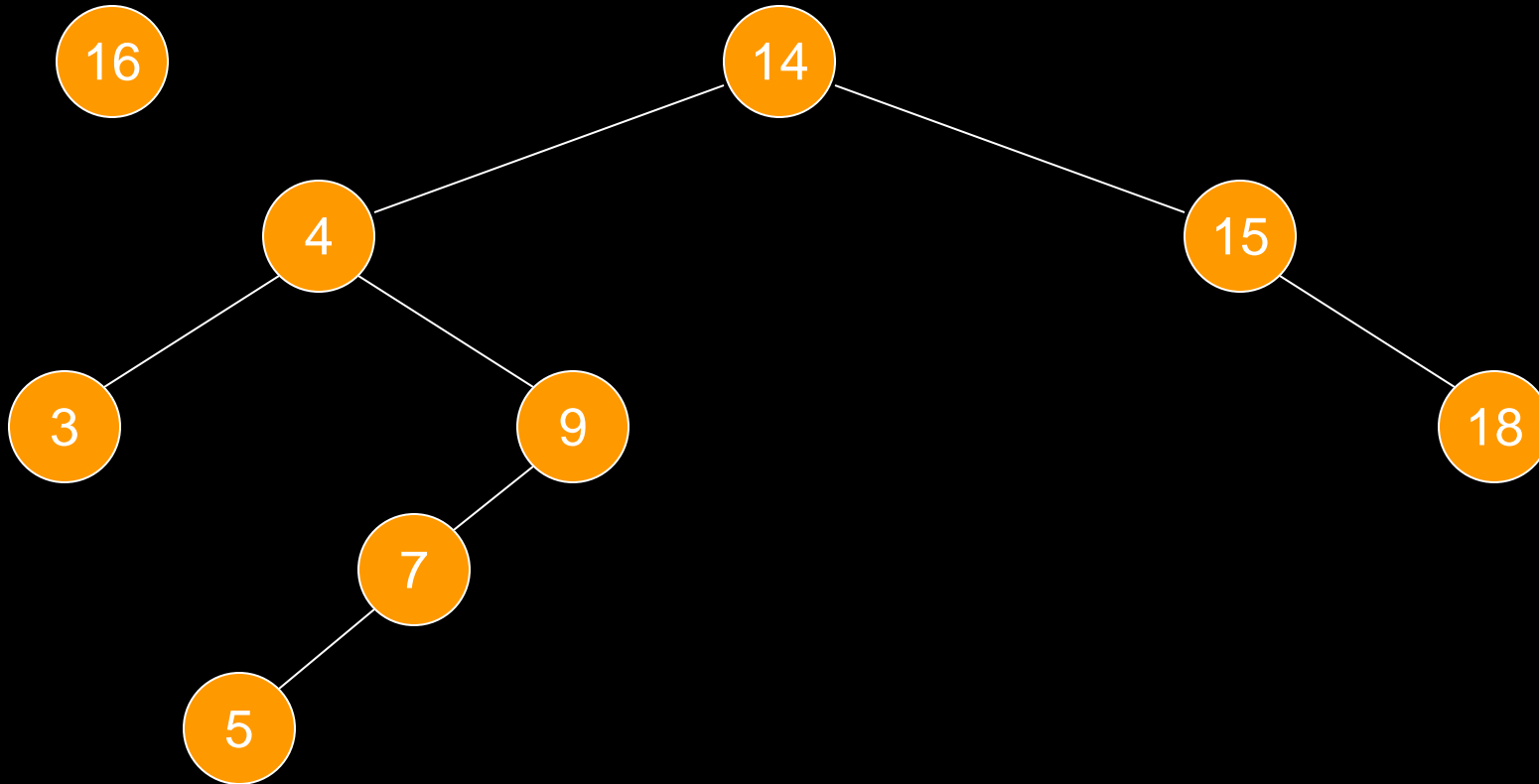
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



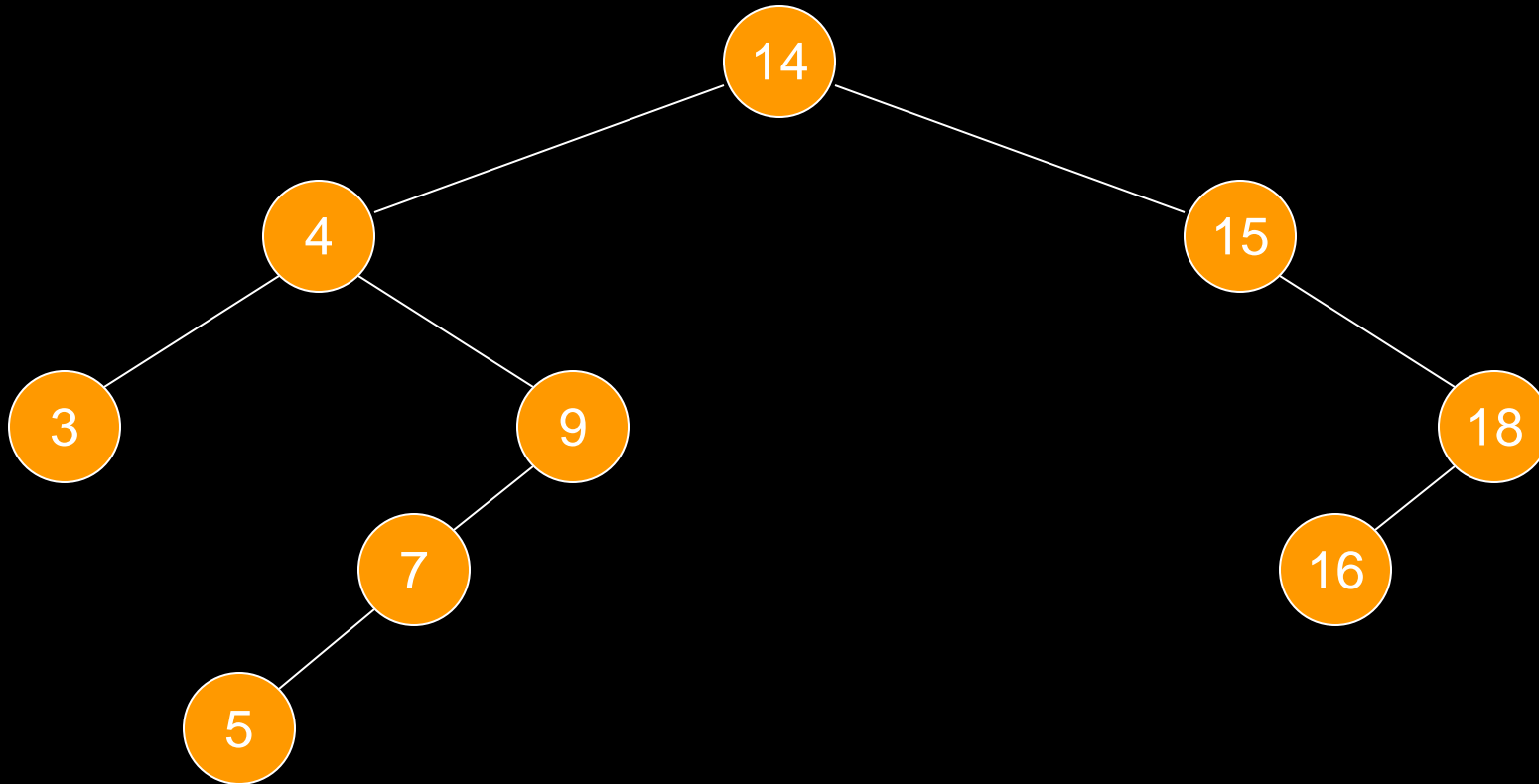
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



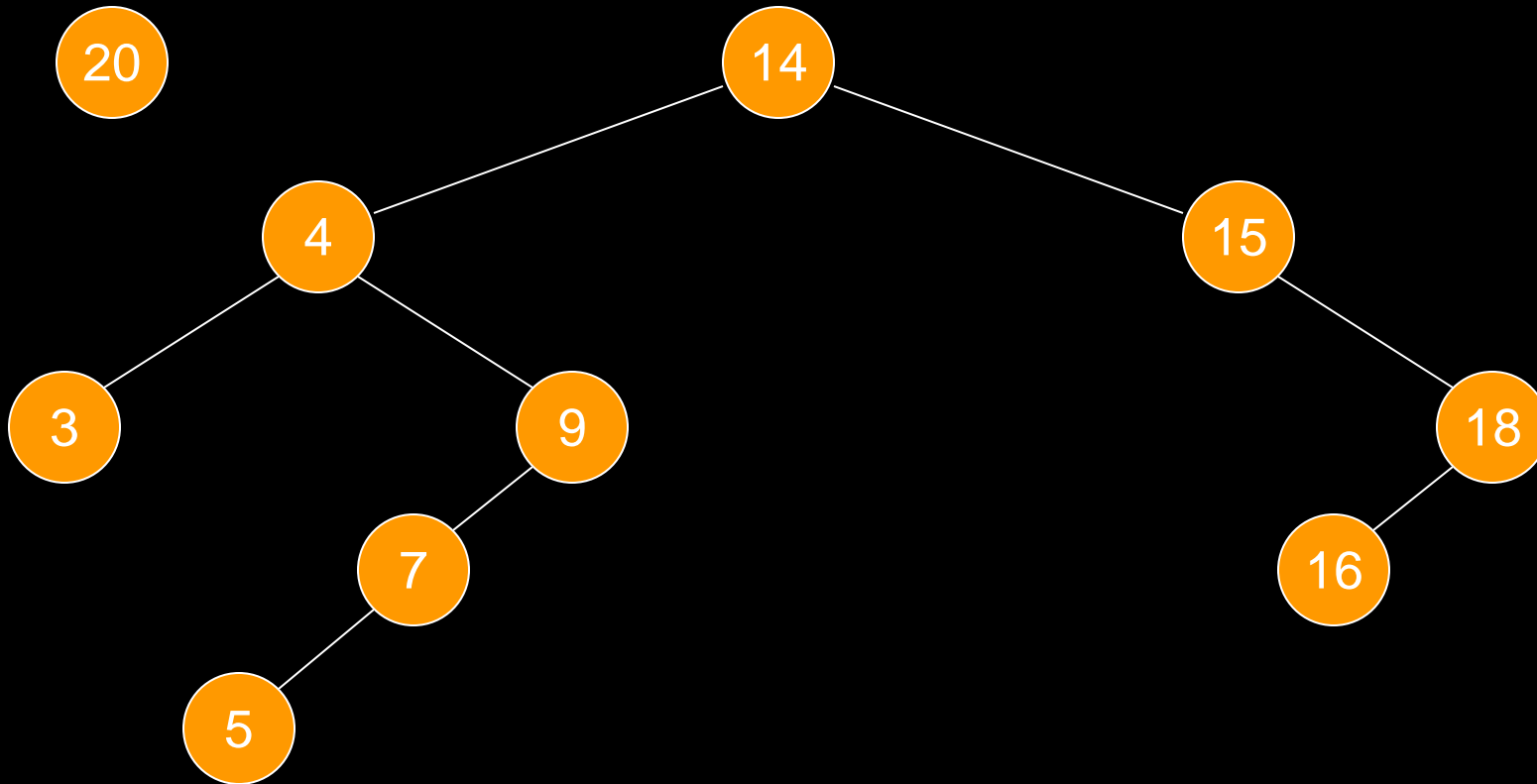
14, 15, 4, 9, 7, 18, 3, 5, 16, , 20, 17

# Building Binary Search Tree



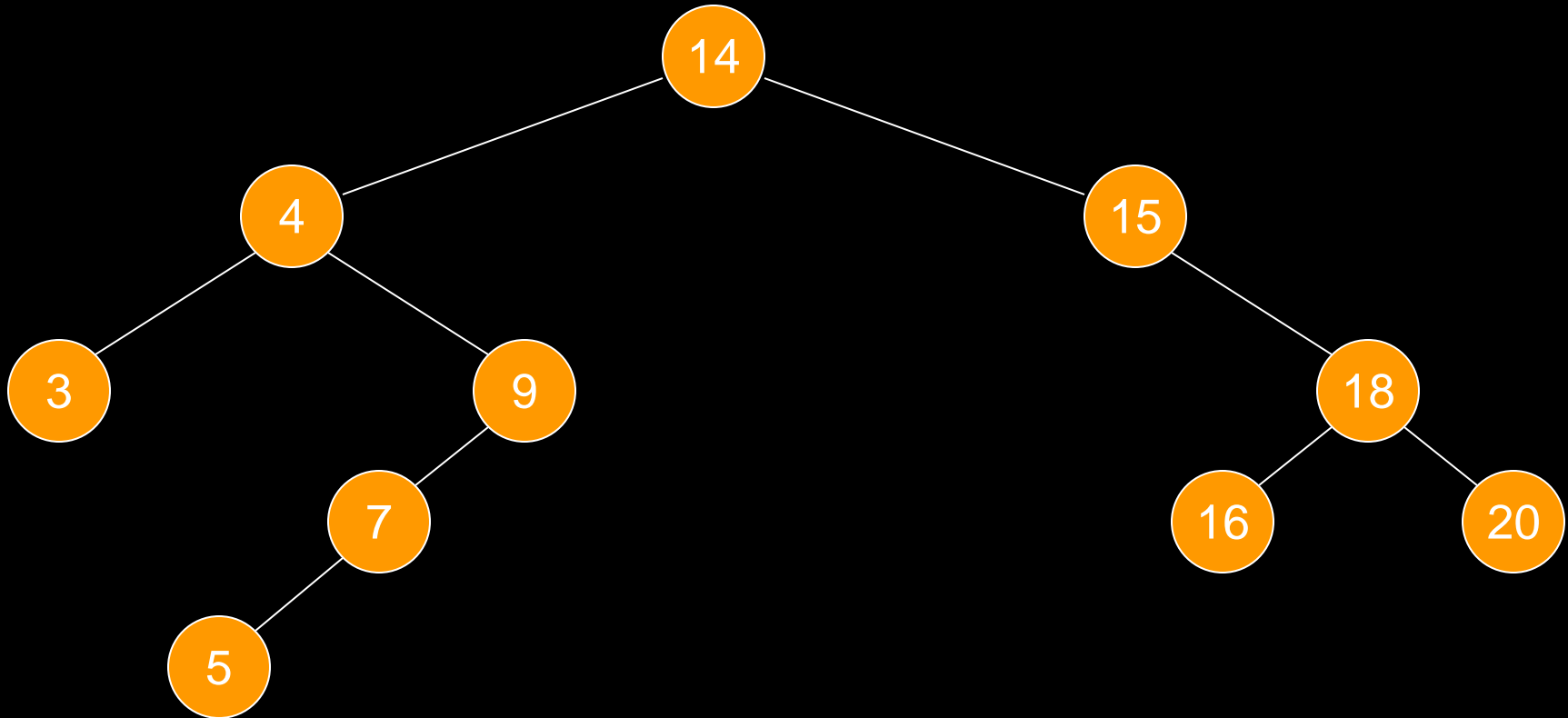
14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17

# Building Binary Search Tree



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

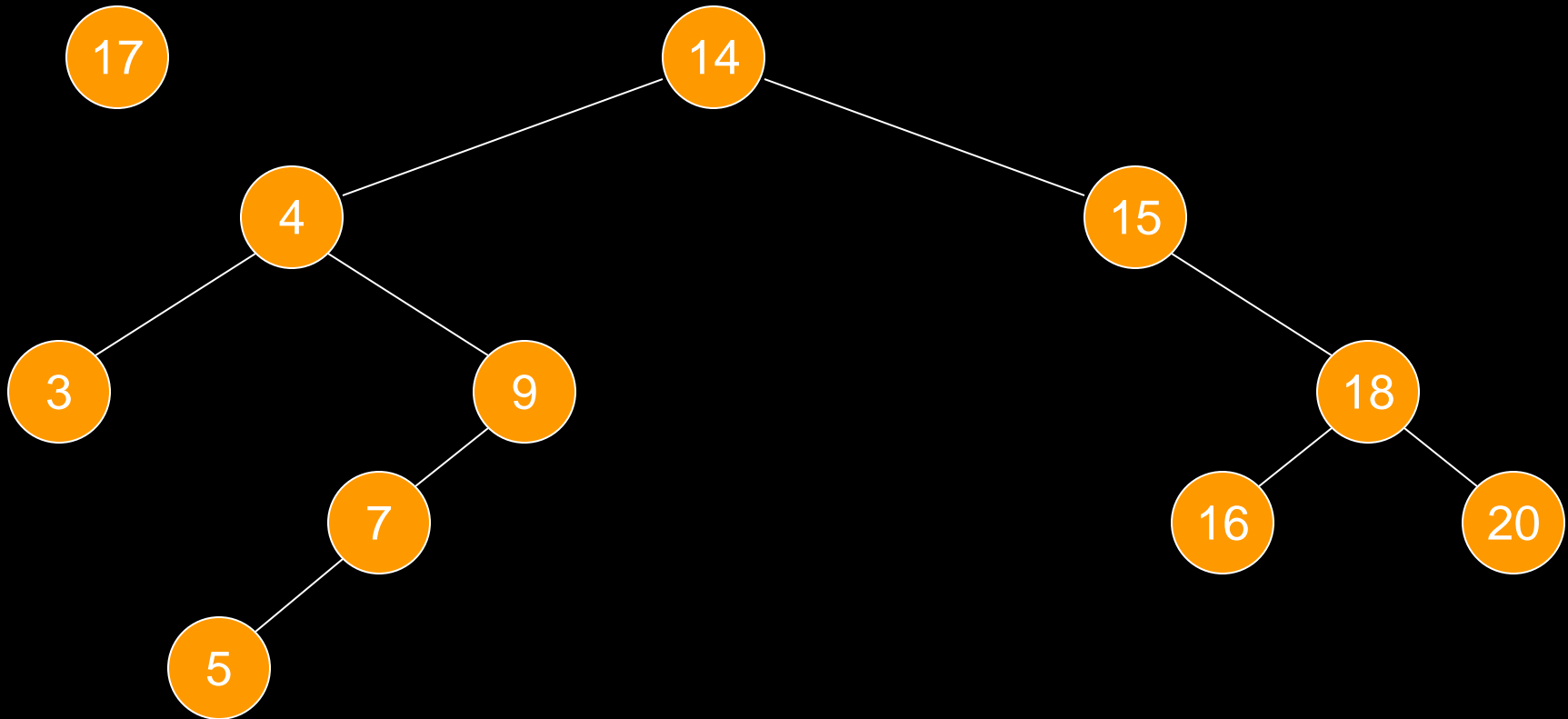
# Building Binary Search Tree



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

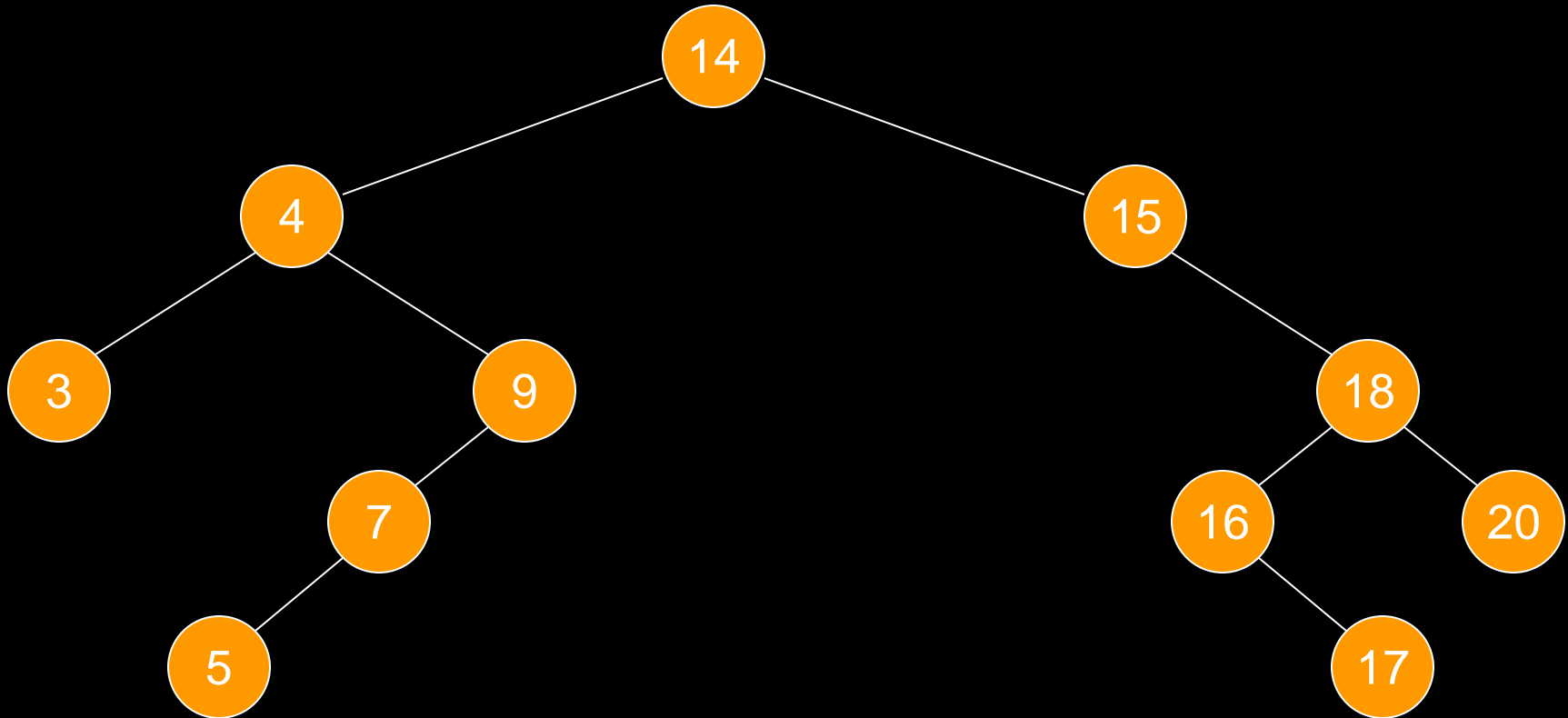


# Building Binary Search Tree



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

# Building Binary Search Tree



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

# Applications of Binary Trees

- A binary tree is a useful data structure when two-way decisions must be made at each point in a process.
- For example, suppose we wanted to find all duplicates in a list of numbers:

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

# Applications of Binary Trees

- One way of finding duplicates is to compare each number with all those that precede it.

---

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



---

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



# Searching for Duplicates

- If the list of numbers is large and is growing, this procedure involves a large number of comparisons.
- A linked list could handle the growth but the comparisons would still be large.
- The number of comparisons can be drastically reduced by using a binary tree.
- The tree grows dynamically like the linked list.

# Searching for Duplicates

- The binary tree is built in a special way.
- The first number in the list is placed in a node that is designated as the root of a binary tree.
- Initially, both left and right subtrees of the root are empty.
- We take the next number and compare it with the number placed in the root.
- If it is the same then we have a duplicate.

# Searching for Duplicates

- Otherwise, we create a new tree node and put the new number in it.
- The new node is made the left child of the root node if the second number is less than the one in the root.
- The new node is made the right child if the number is greater than the one in the root.

# Searching for Duplicates



14

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



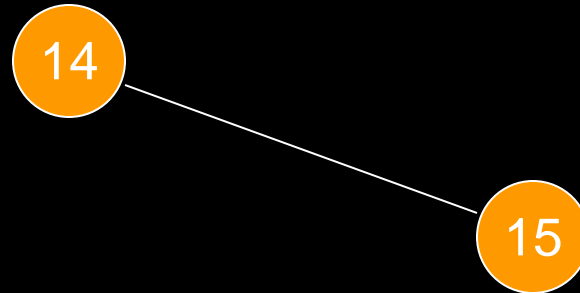
# Searching for Duplicates

15

14

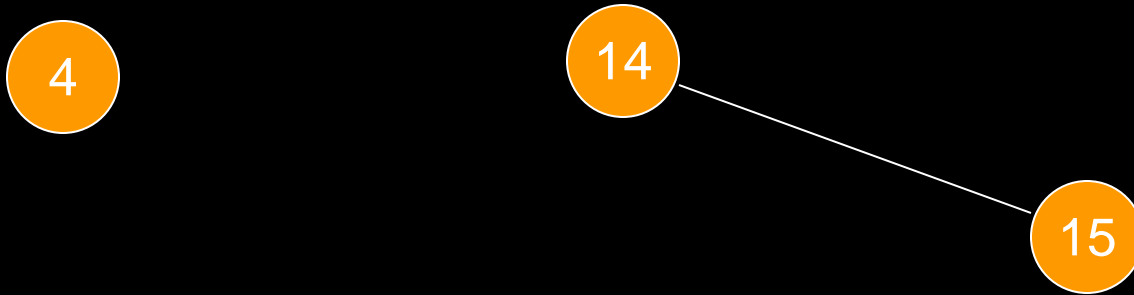
15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



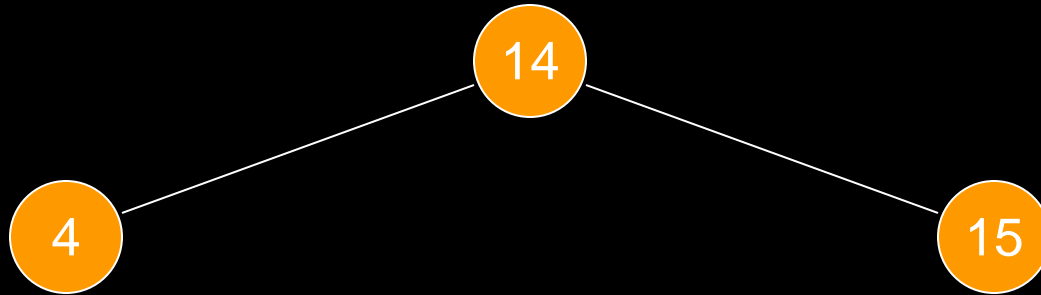
15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



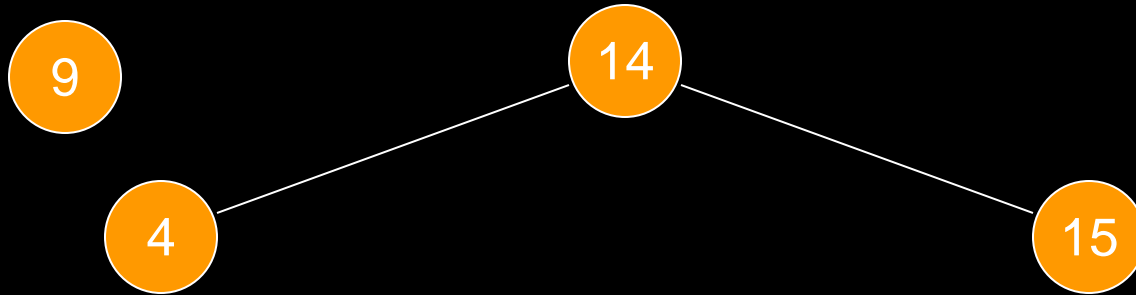
4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



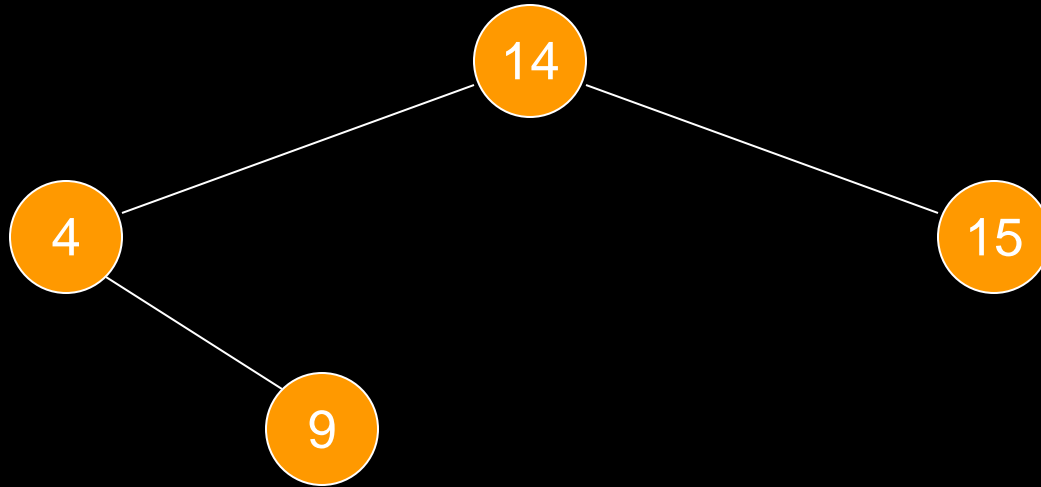
4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



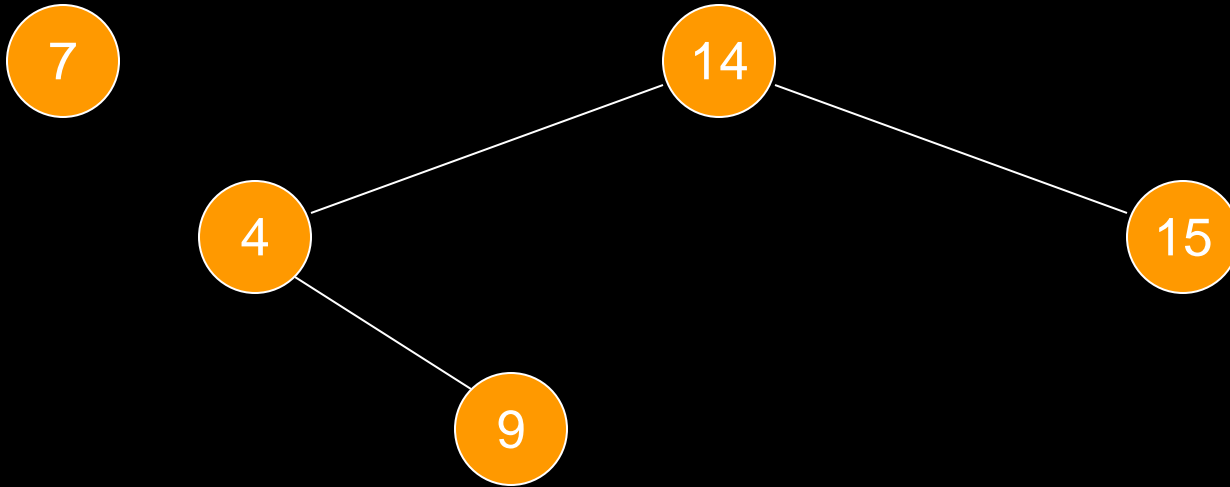
9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



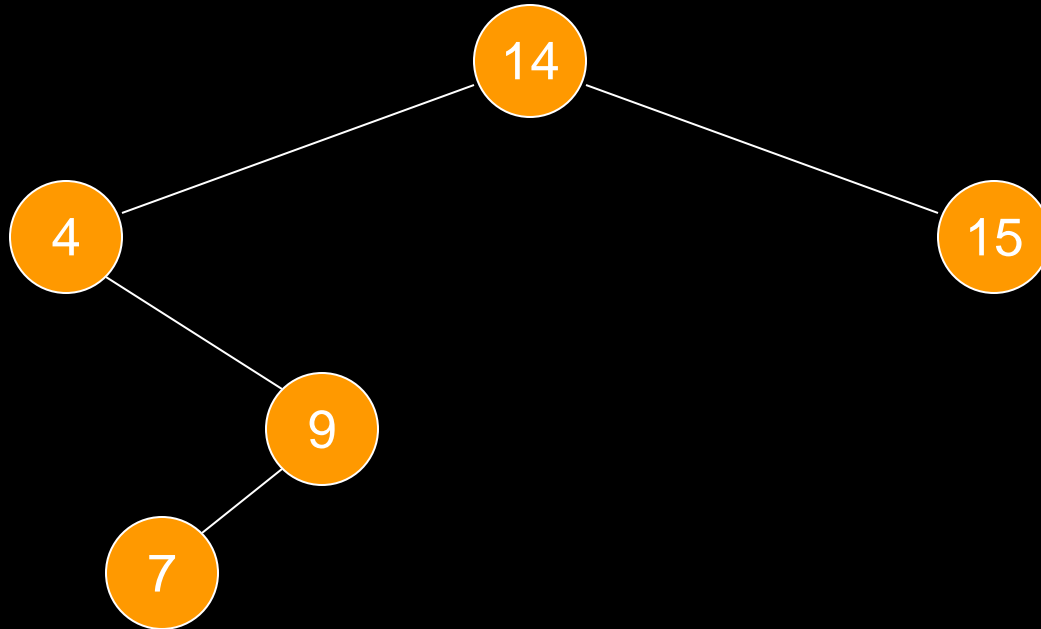
9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



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

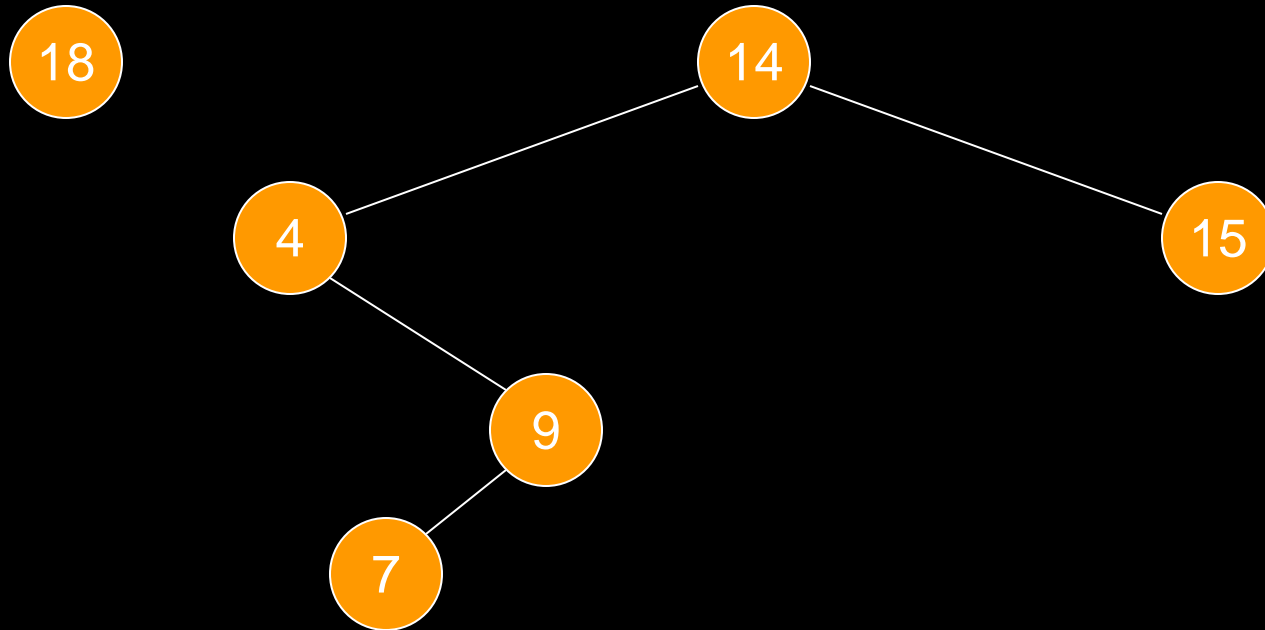
# Searching for Duplicates



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

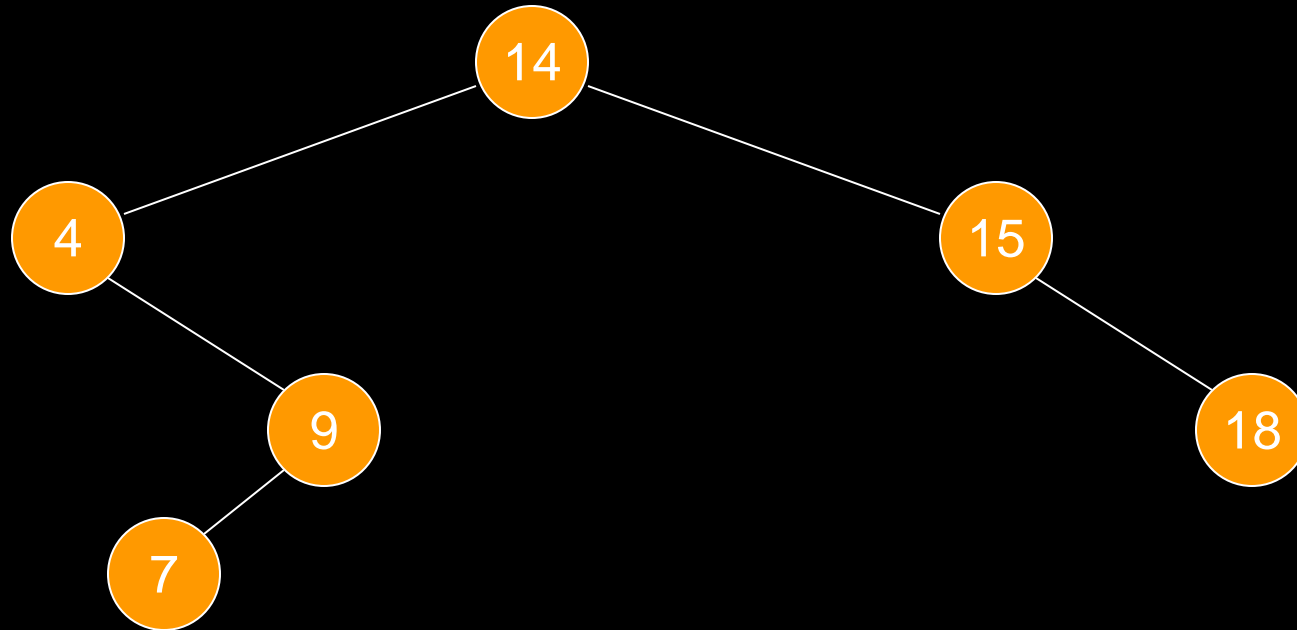


# Searching for Duplicates



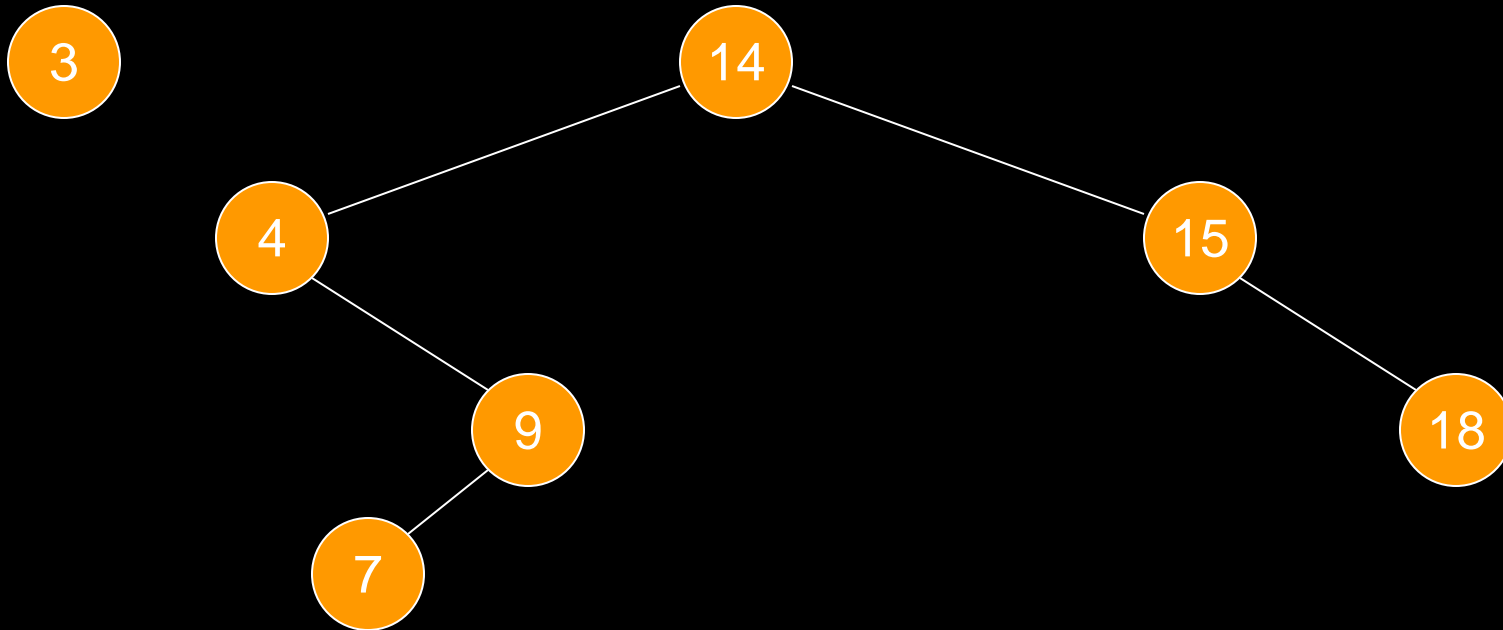
18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



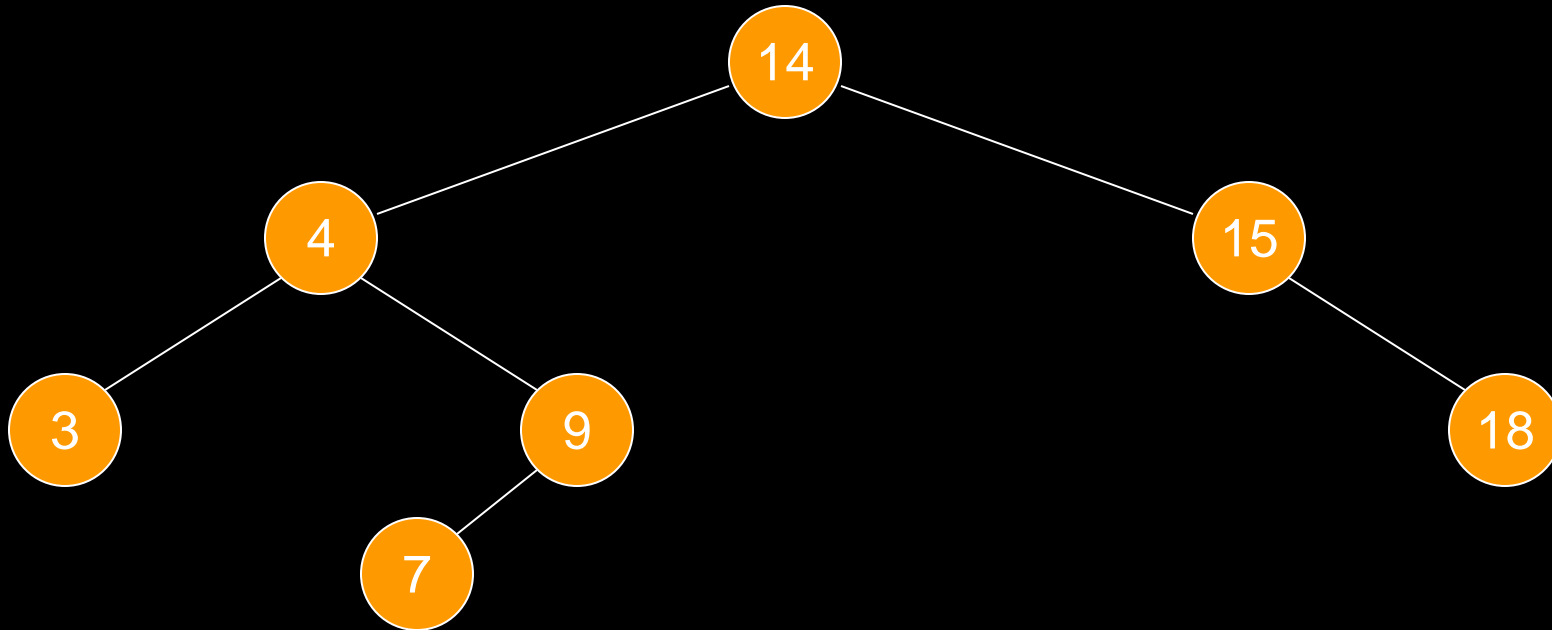
18, 3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



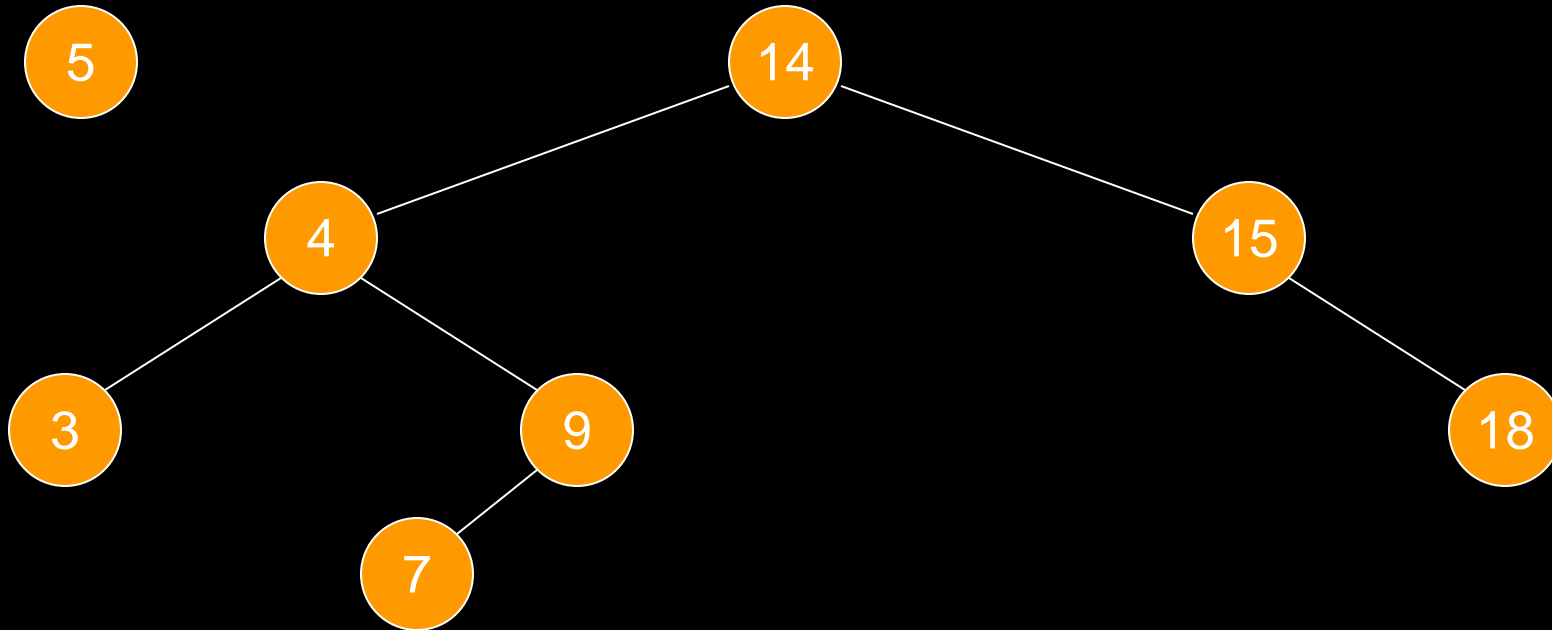
3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



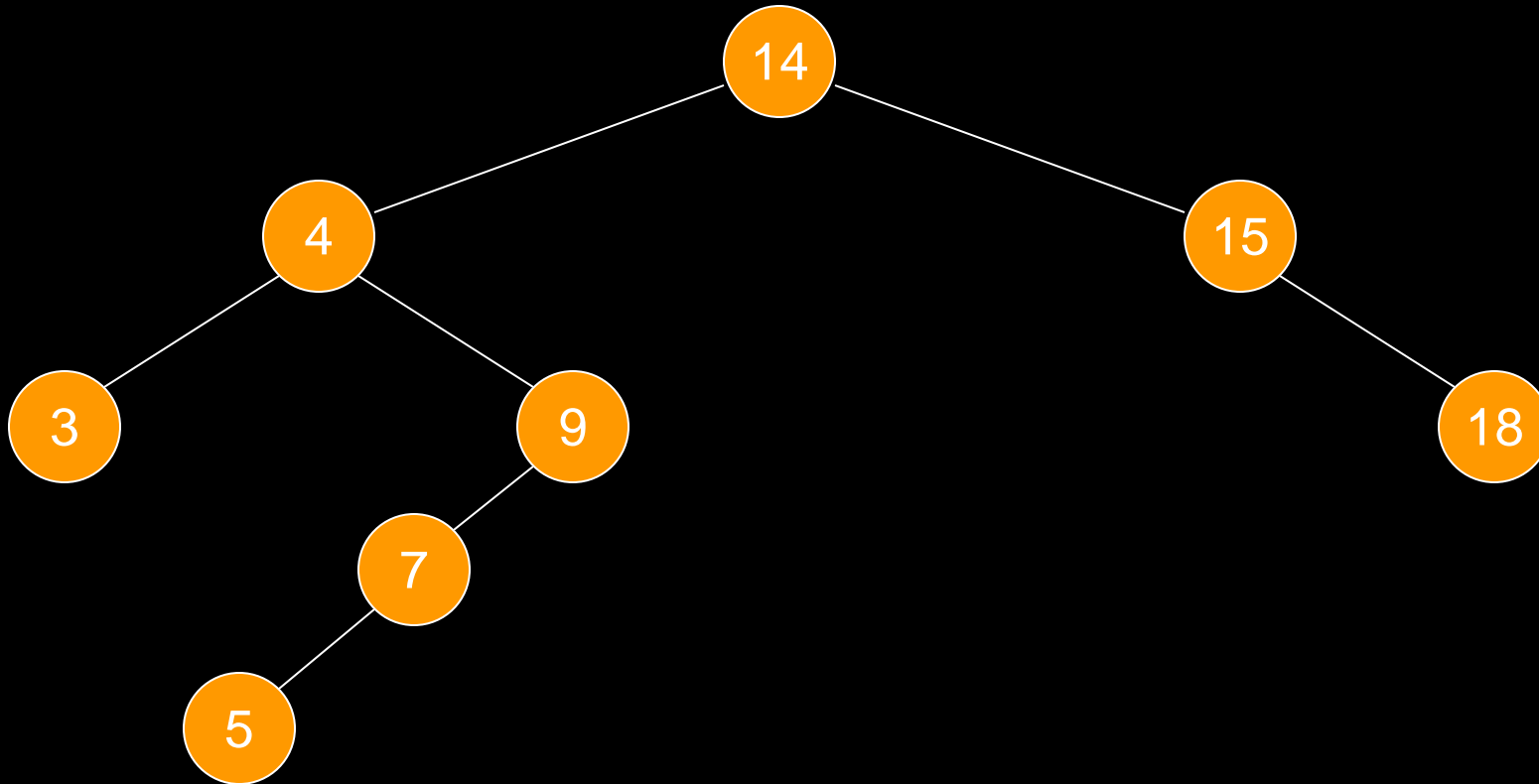
3, 5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



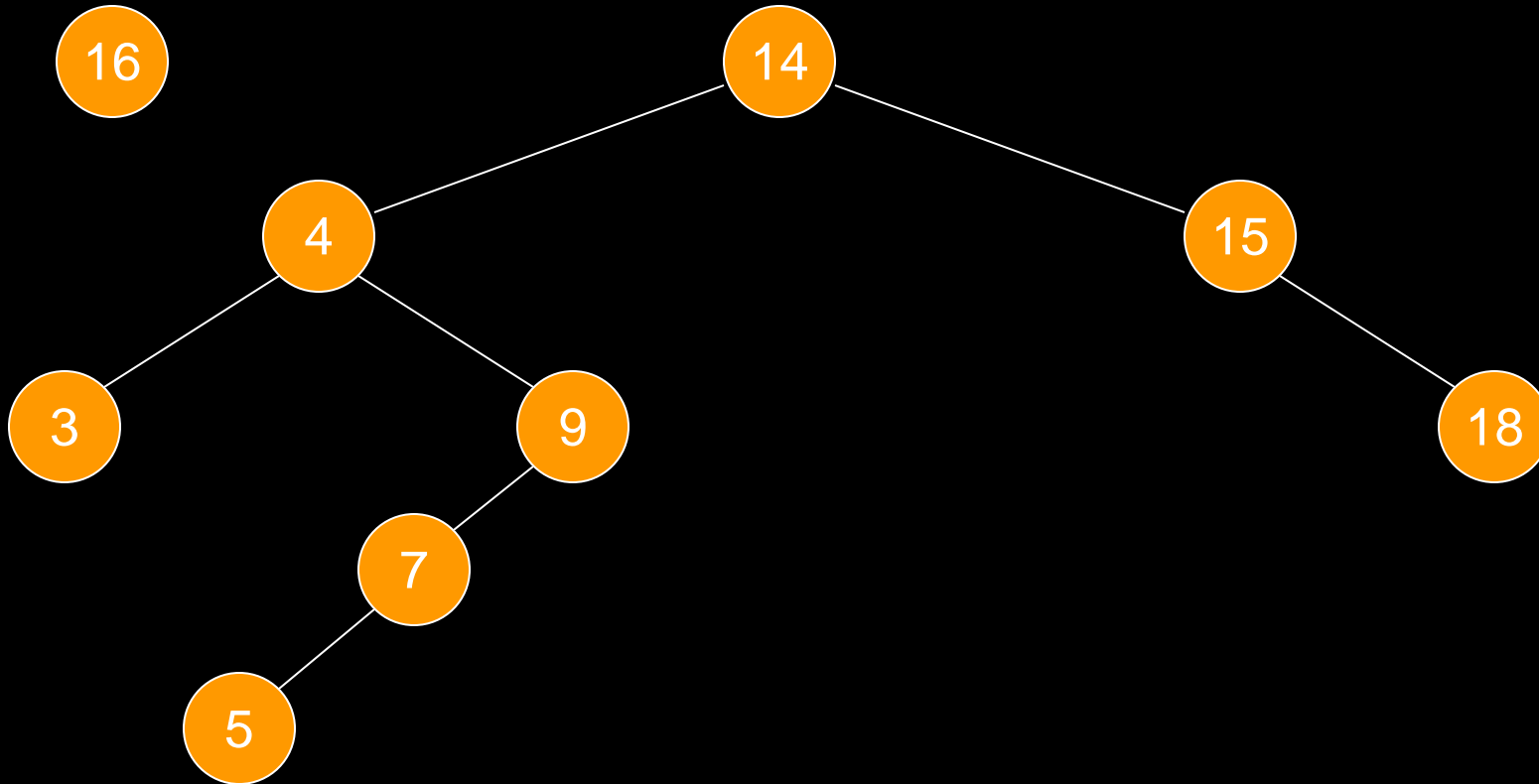
5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



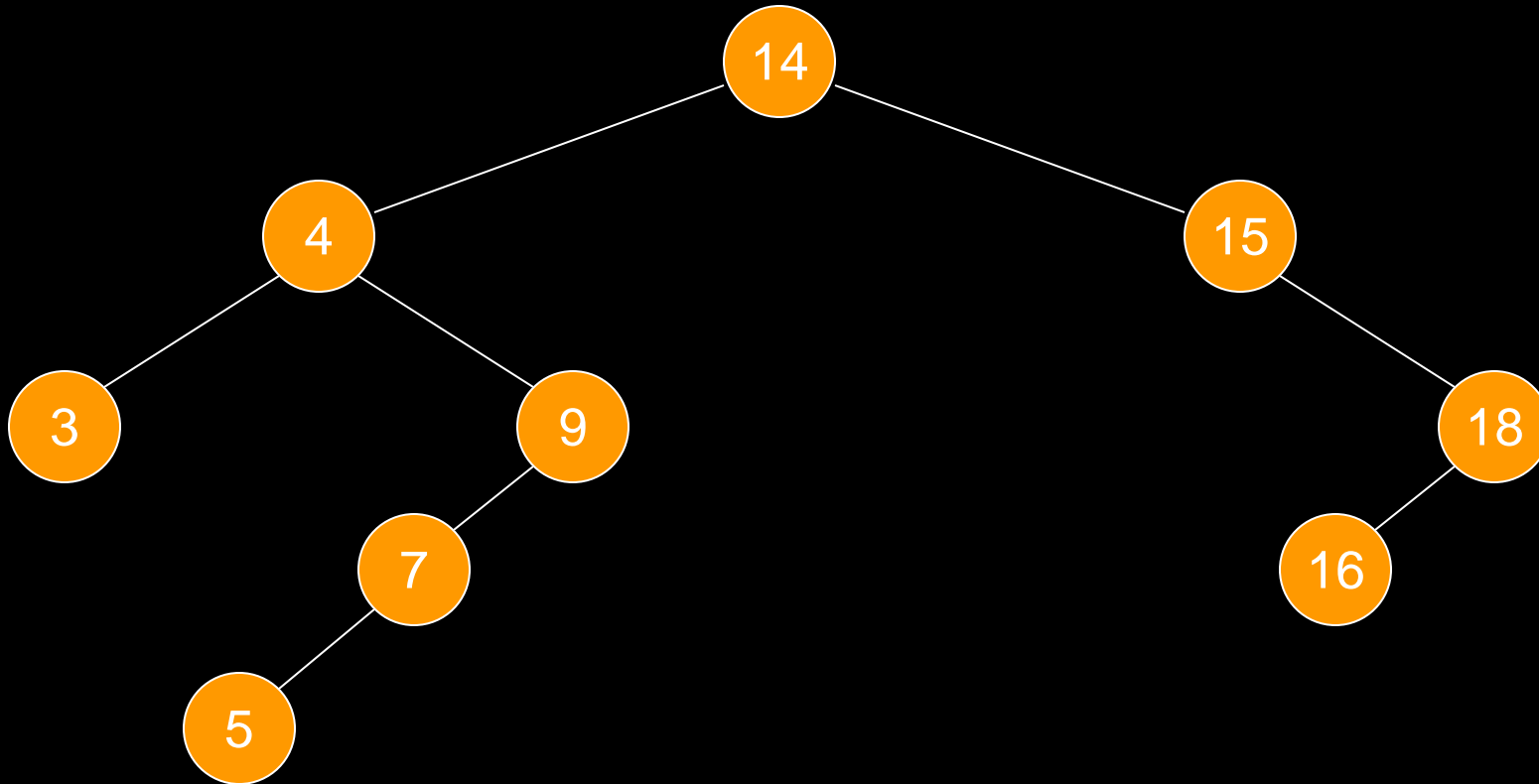
5, 16, 4, 20, 17, 9, 14, 5

# Searching for Duplicates



16, 4, 20, 17, 9, 14, 5

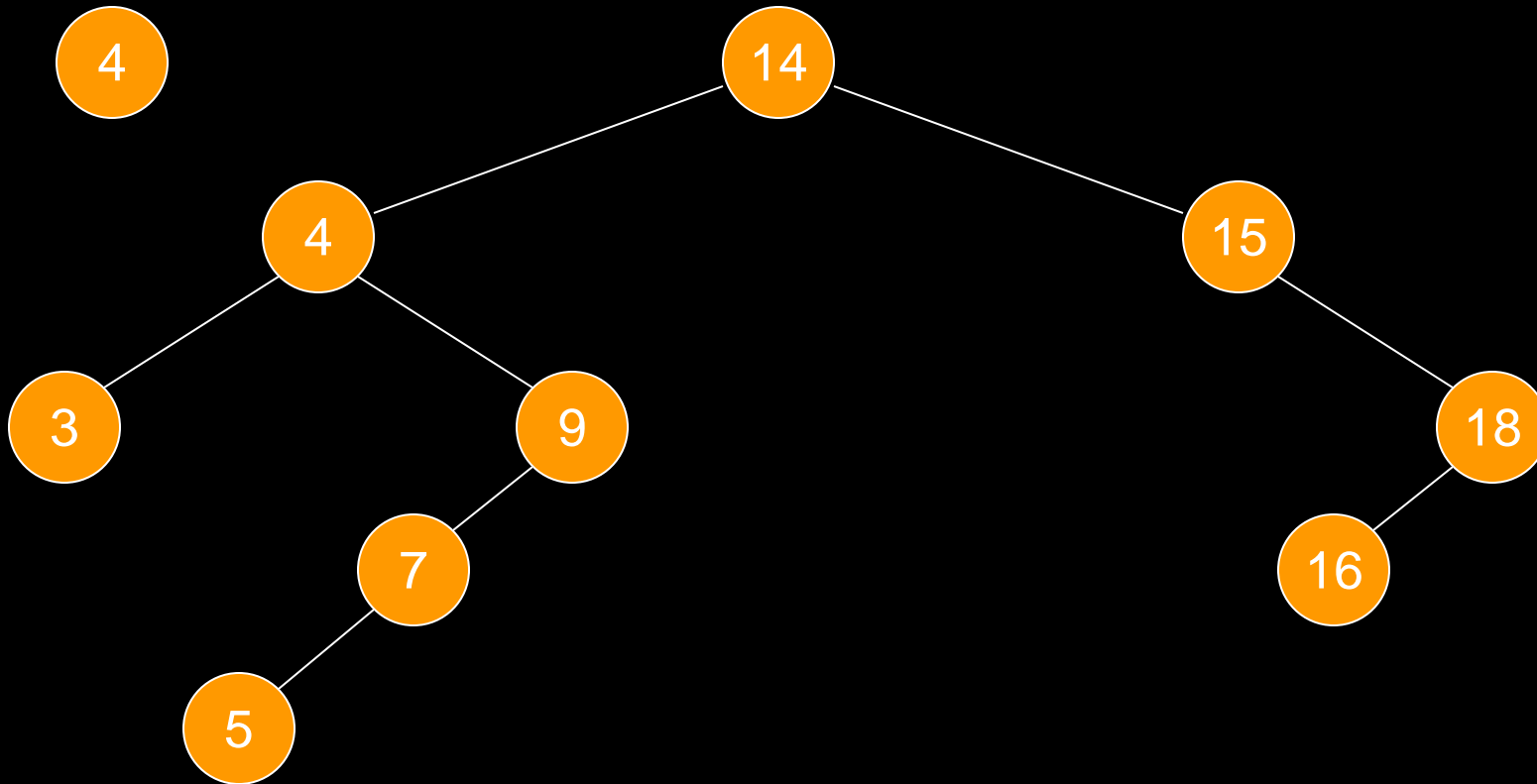
# Searching for Duplicates



16, 4, 20, 17, 9, 14, 5

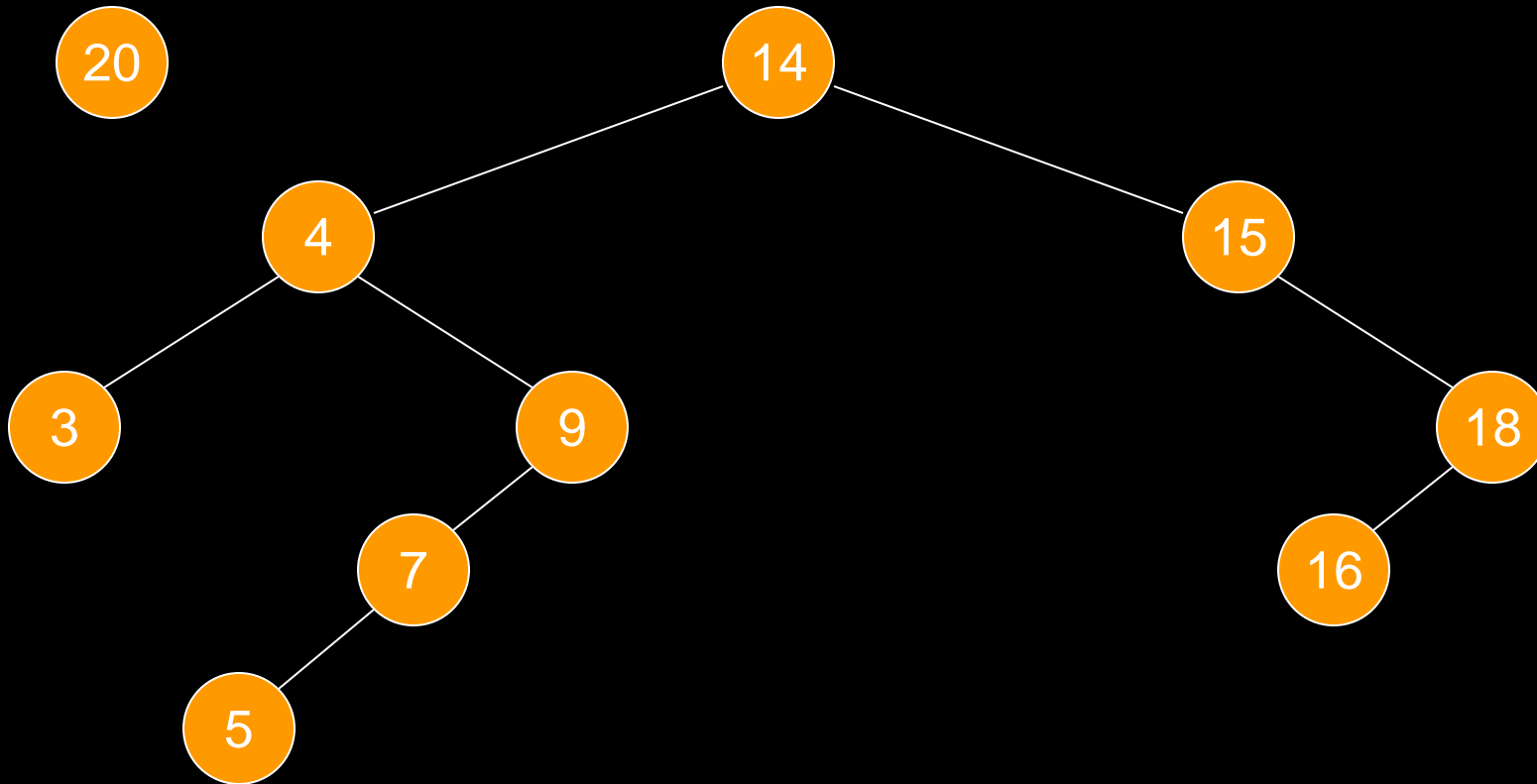


# Searching for Duplicates



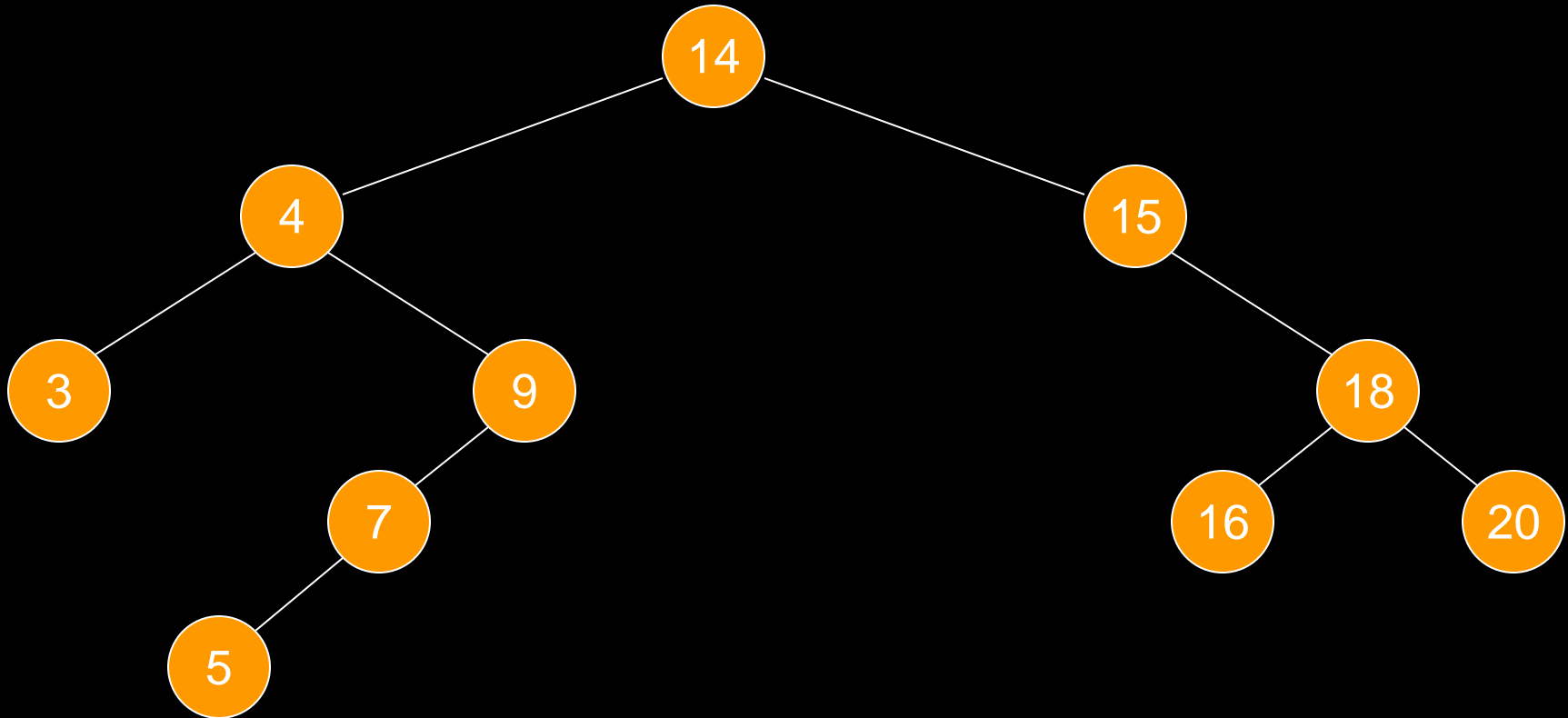
4, 20, 17, 9, 14, 5

# Searching for Duplicates



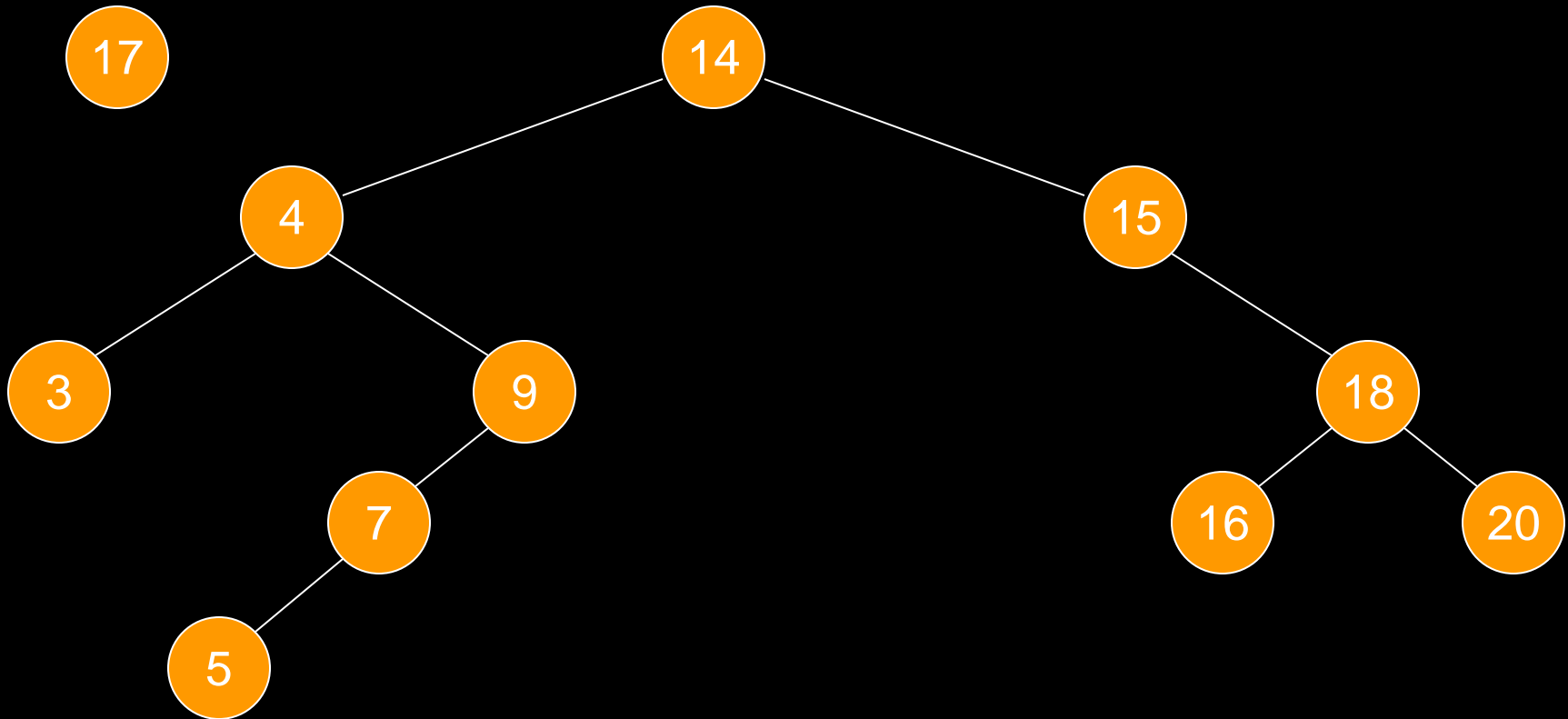
20, 17, 9, 14, 5

# Searching for Duplicates



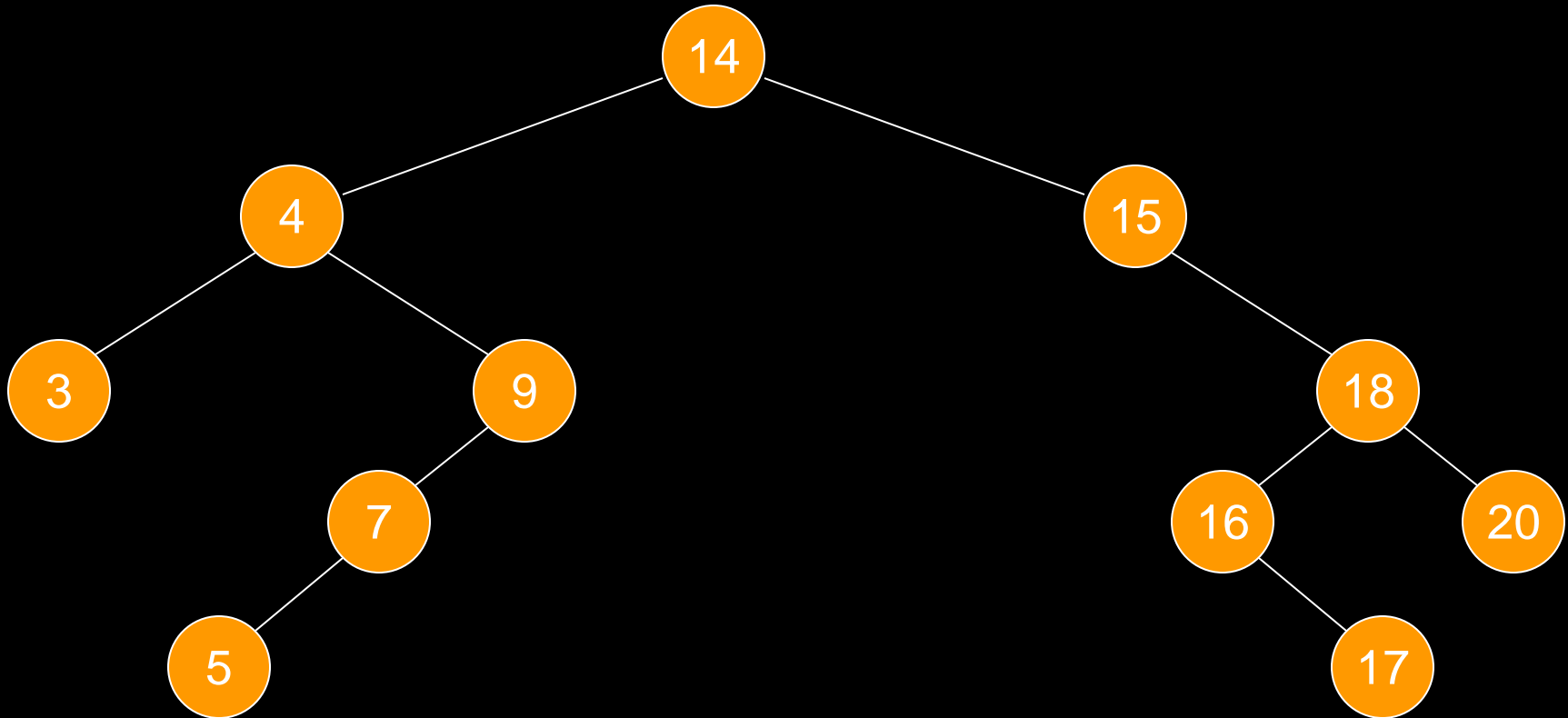
20, 17, 9, 14, 5

# Searching for Duplicates



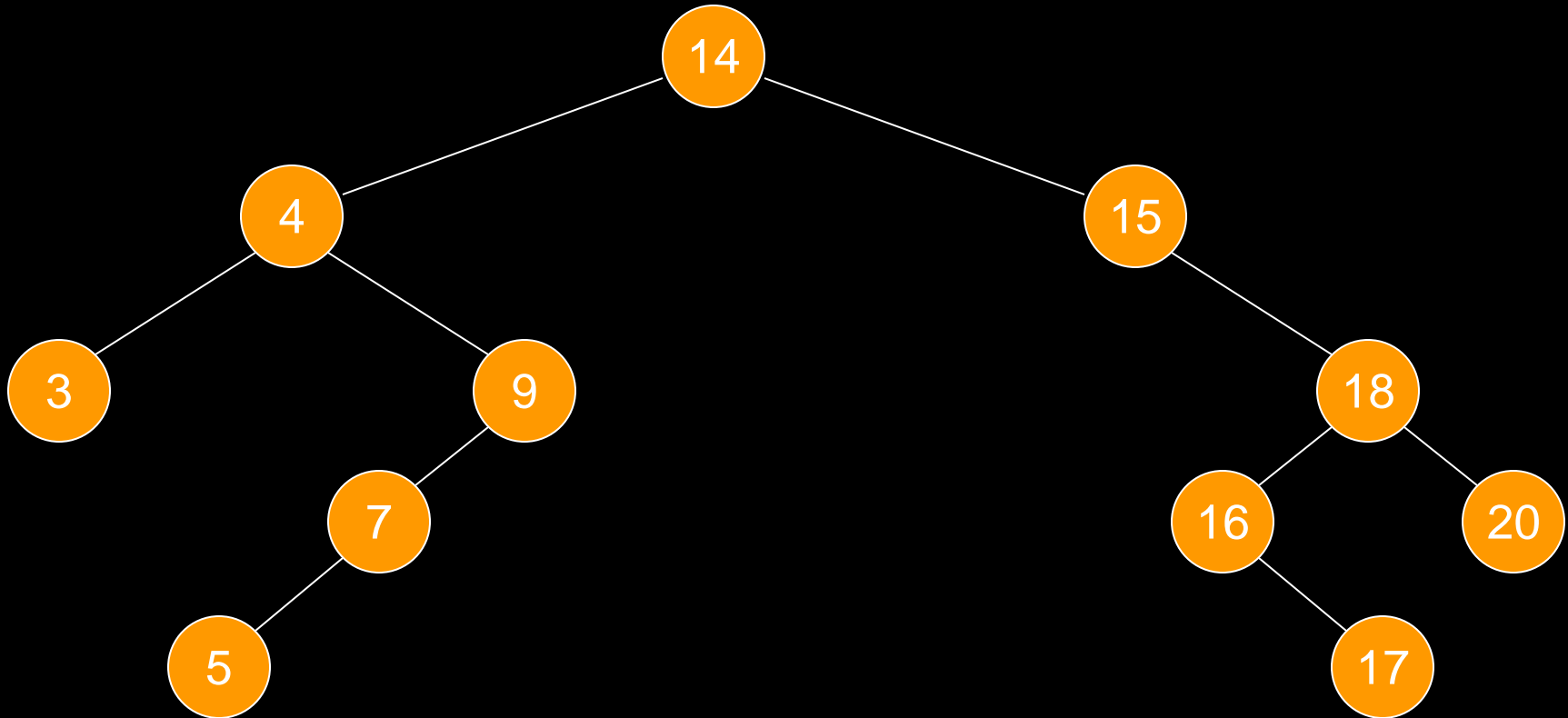
17, 9, 14, 5

# Searching for Duplicates



17, 9, 14, 5

# Searching for Duplicates



9, 14, 5

# Representation of a Binary Tree

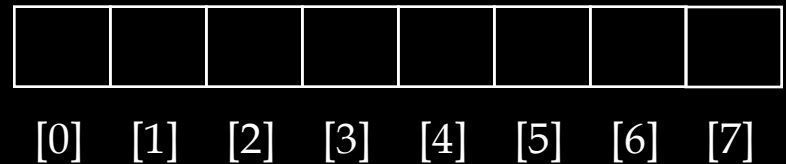
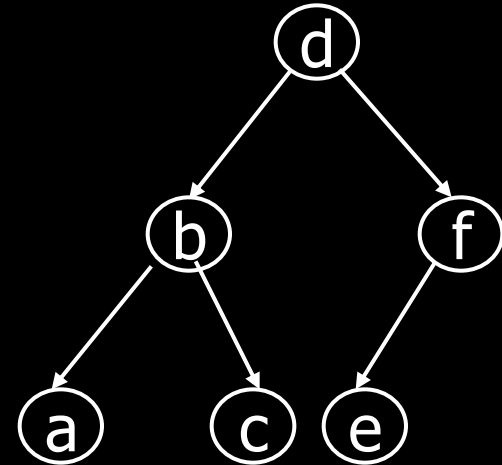
- An array-based representation
- A reference-based representation

# An array-based representation

For a Binary Tree of Depth  $d$

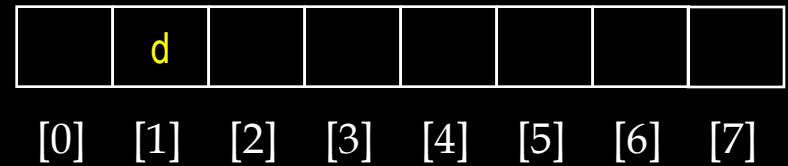
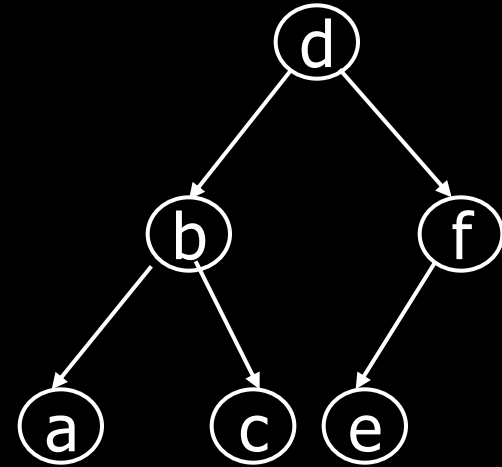
Array must be of size  $2^d$

1. The Root will be stored in location **1**
2. The **left Child** of the Node stored at location  **$i$**  is stored at location  **$2i$**
3. The **Right Child** of the Node stored at location  **$i$**  is stored at location  **$2i+1$**

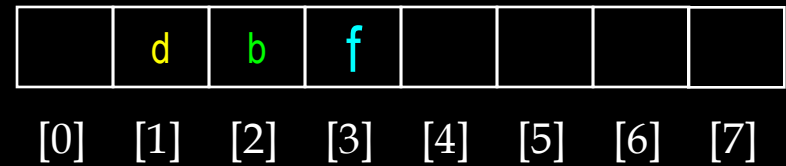
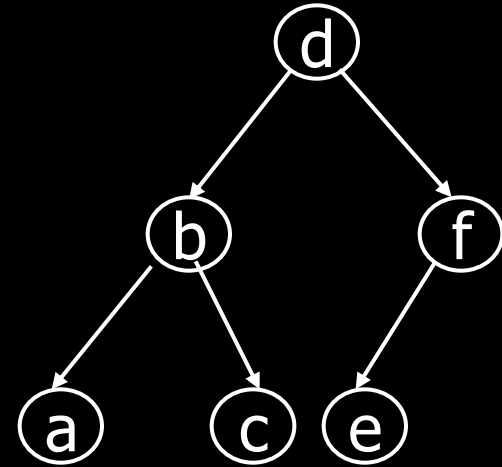




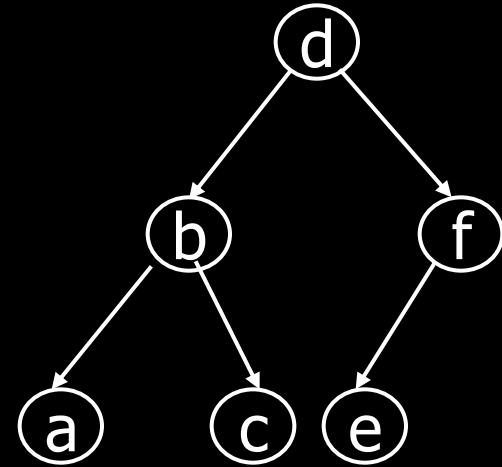
# An array-based representation



# An array-based representation

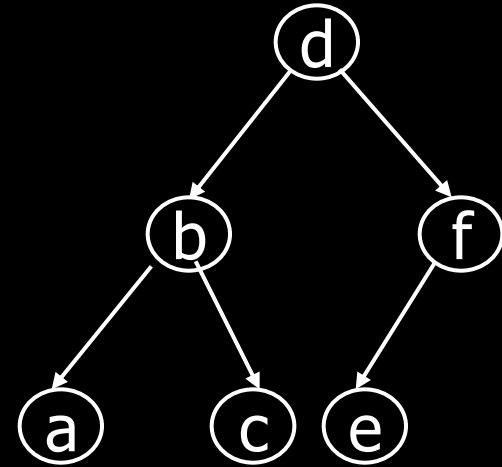


# An array-based representation



	d	b	f	a	c		
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

# An array-based representation



	d	b	f	a	c	e	
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

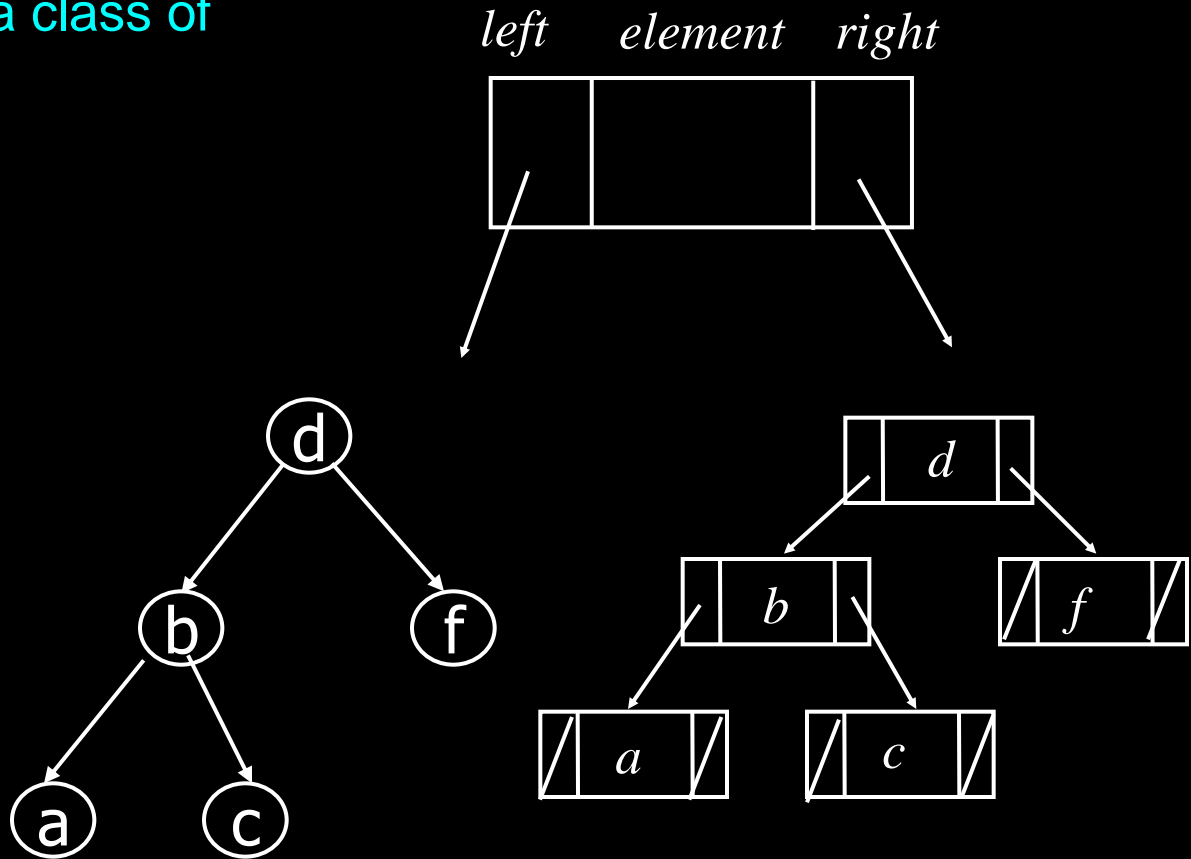
# Reference Based/Linked Representation

You can code this with a class of three fields:

Object element;

BinaryNode left;

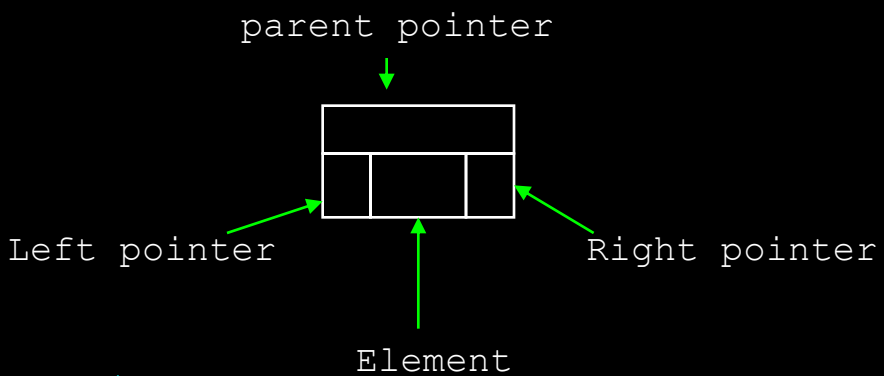
BinaryNode right;



# Building Binary Search Trees – Node - Example

```
class BStree {
private:
    class TreeNode {
    private:
        int data;
        TreeNode *left,*right,*parent;
    public:
        int getdata() { return data; };
        void setdata(int data) { this->data = data;};
        TreeNode* getleft() { return left; };
        void setleft(TreeNode *left) { this->left = left; };
        TreeNode* getright() { return right; };
        void setright(TreeNode *right) { this->right = right; };
        TreeNode* getparent() { return parent; };
        void setparent(TreeNode *parent) { this->parent = parent; };
    };

    int bstsize, dupcount;
    TreeNode *root,*current,*temp;
```



# Building Binary Search Trees – constructor - Example

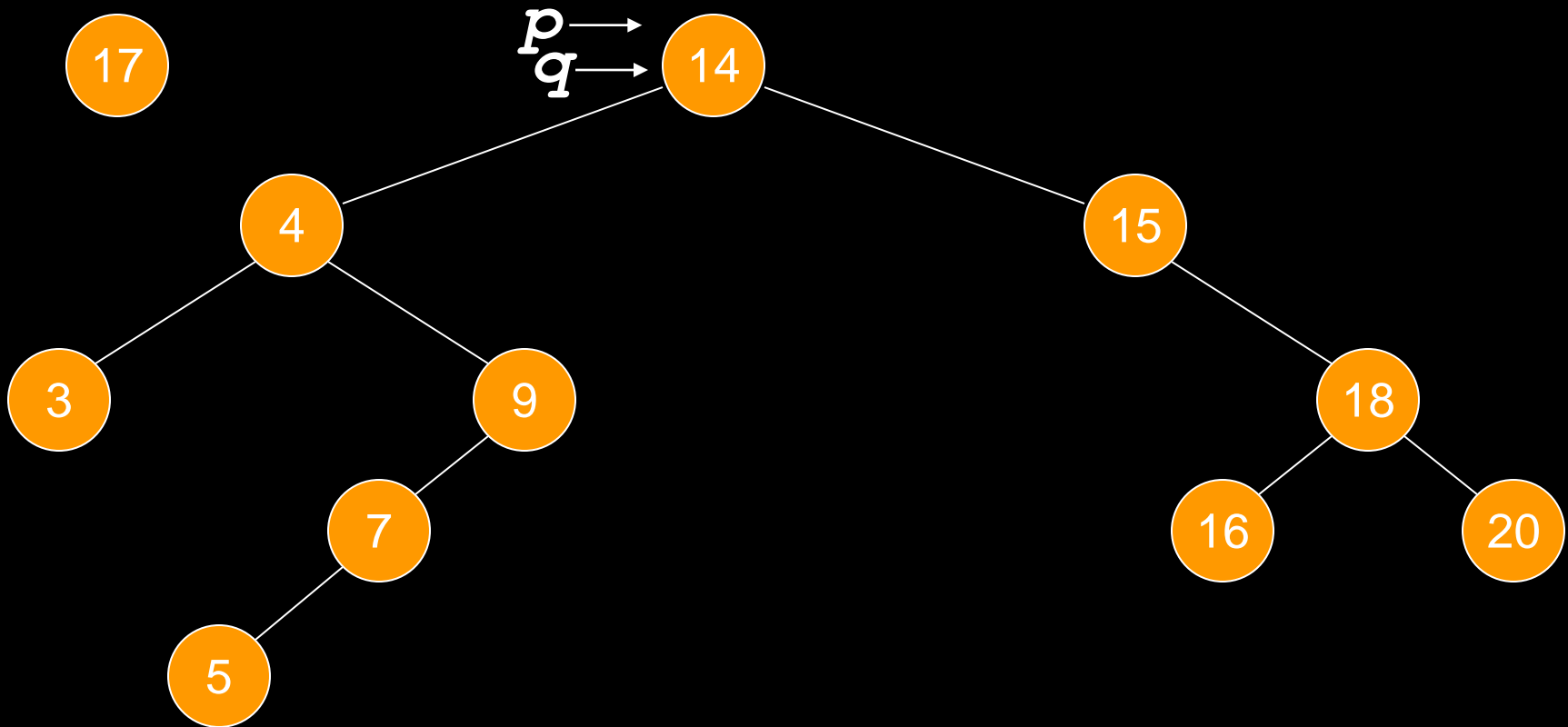
```
public:
    // Constructor
    BStree() {
        root = NULL;
        current = NULL;
        temp = NULL;
        bstsize = 0;
        dupcount = 0;
```

# Building Binary Search Trees – insert - Example

```
int insert(int data) {  
    TreeNode *newNode=new TreeNode;  // create new node  
    newNode->setdata(data);  
    newNode->setleft(NULL); newNode->setright(NULL); newNode->setparent(NULL);  
    // if tree is empty  
    if (root==NULL) { root=newNode; current=newNode; bstsize=1; return 0;}  
    else if (root!=NULL) //if not empty, find proper location and point current to it  
        { temp = current = root;  
          while( (data!=temp->getdata()) && (current!=NULL) )  
              { temp = current;  
                if ( data<temp->getdata() )    current = temp->getleft();  
                else                          current = temp->getright();          }  
          if( data == temp->getdata() )    //duplicate data, trash new node  
              { dupcount++; delete newNode; }  
          else if( data<temp->getdata() ) //otherwise link new node to the tree  
              { temp->setleft( newNode ); newNode->setparent(temp);bstsize++;}  
          else      { temp->setright( newNode ); newNode->setparent(temp);bstsize++;}  
              }  
    return 0;  
}
```

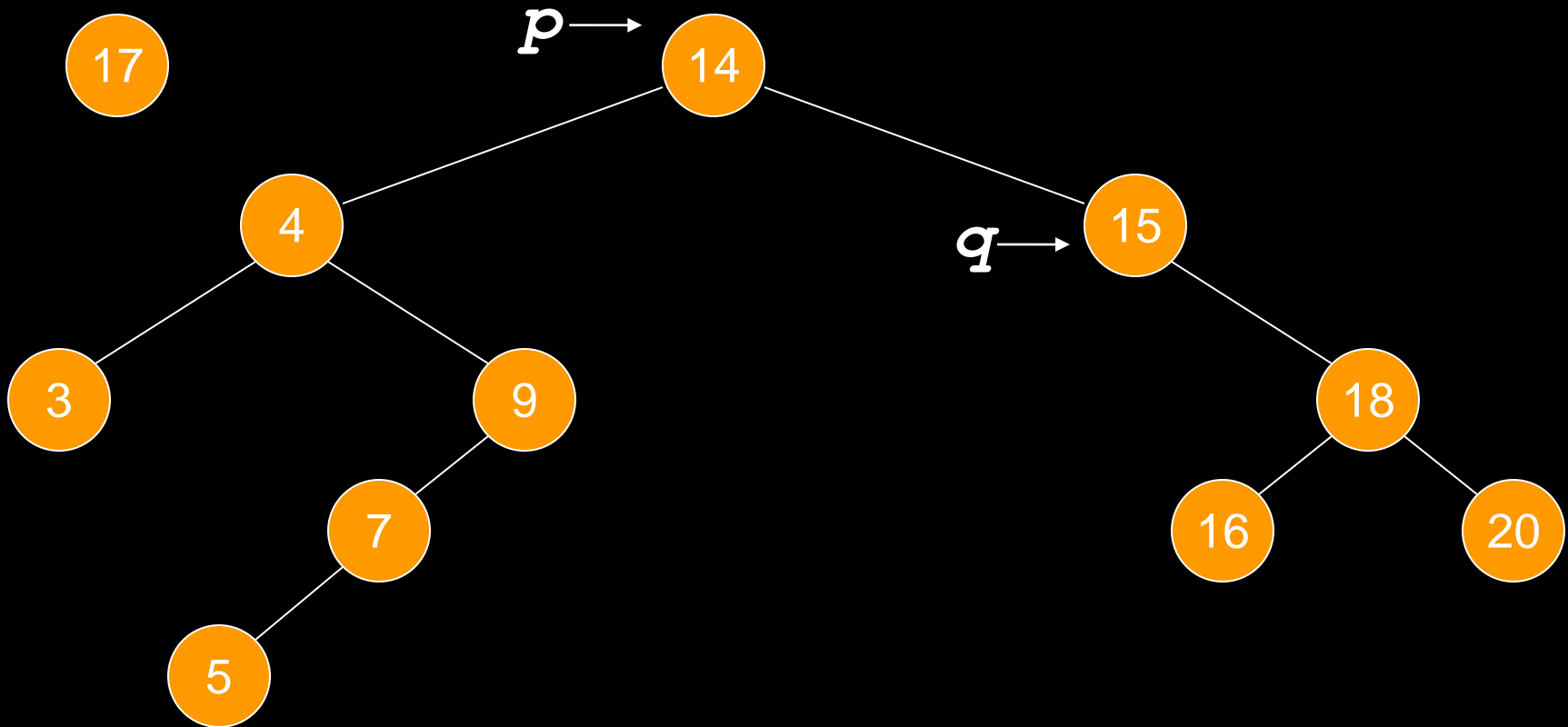


# Trace of insert



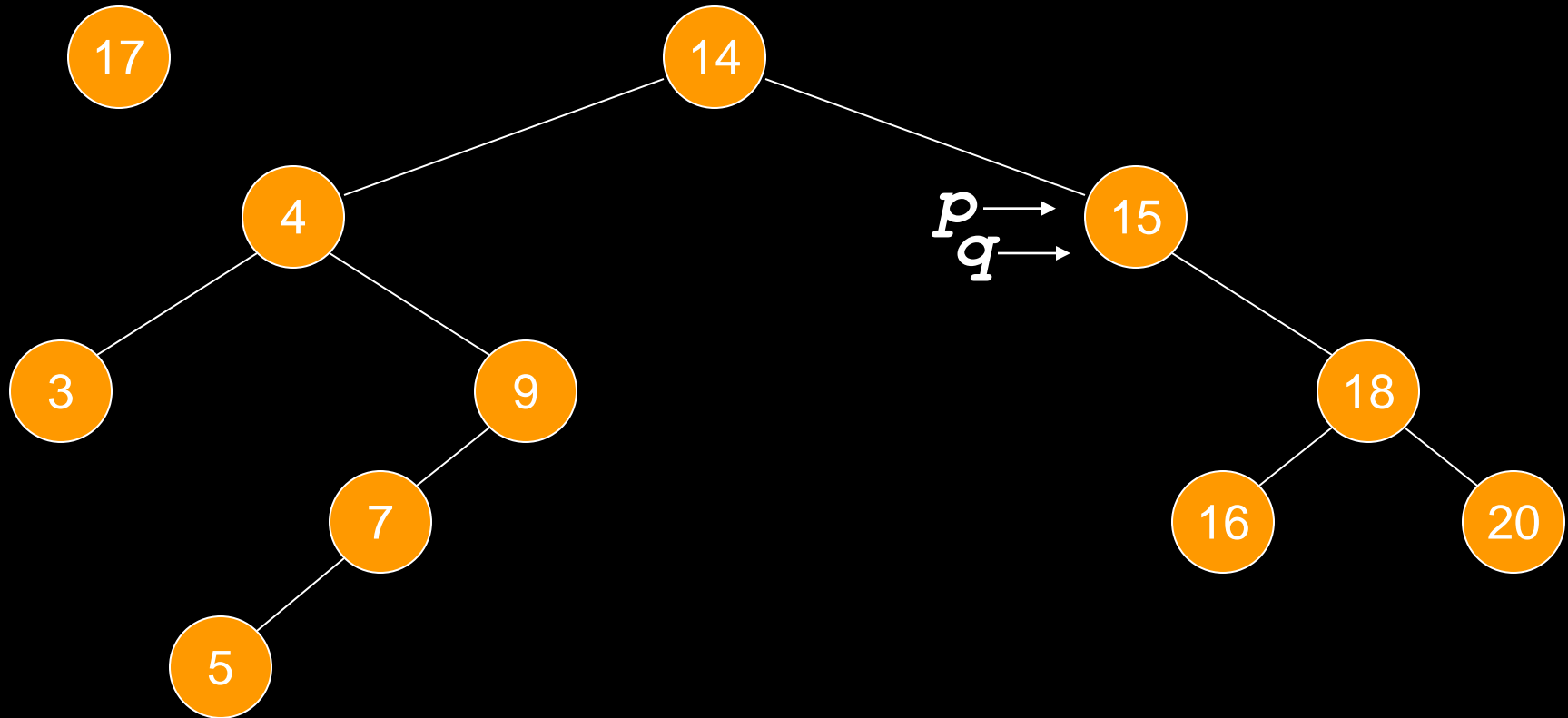
17, 9, 14, 5

# Trace of insert

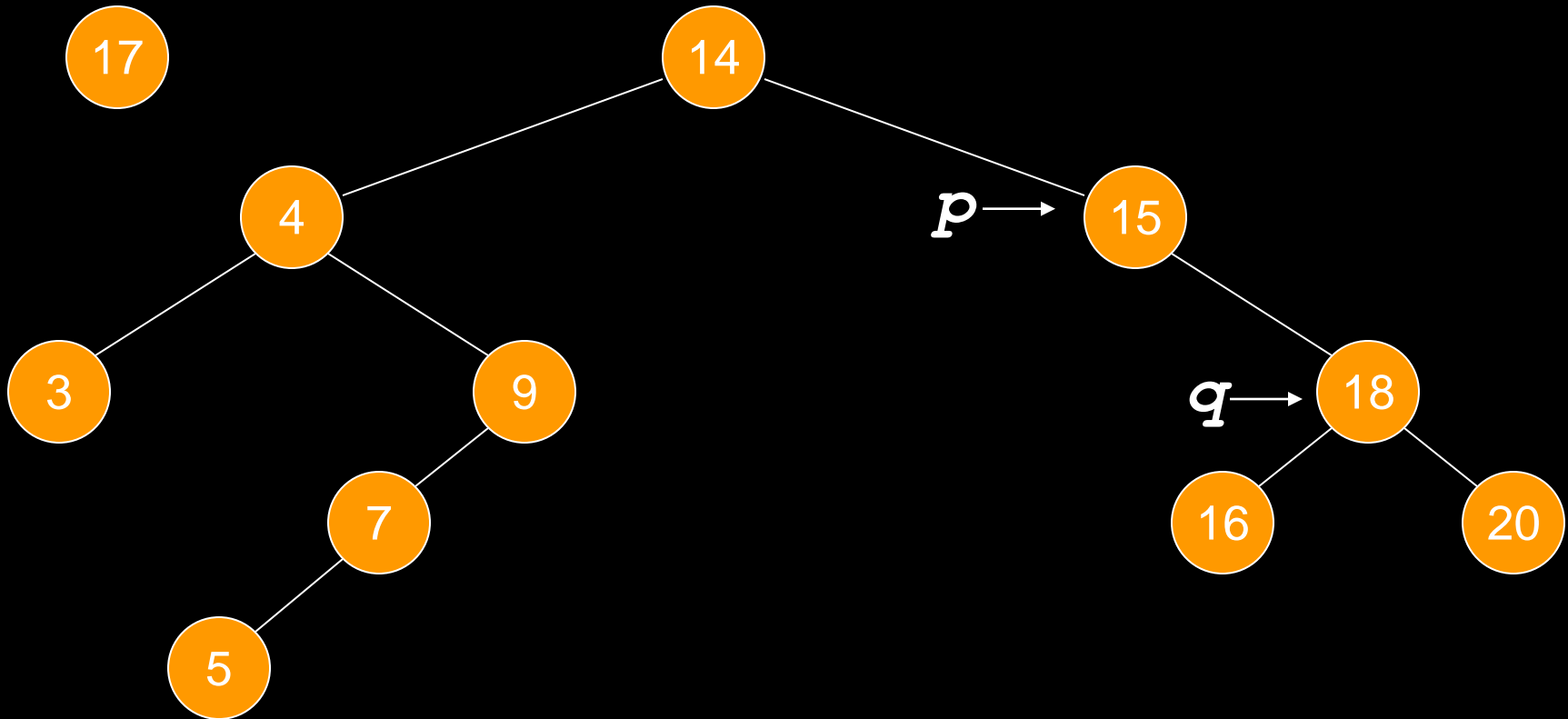


17, 9, 14, 5

# Trace of insert

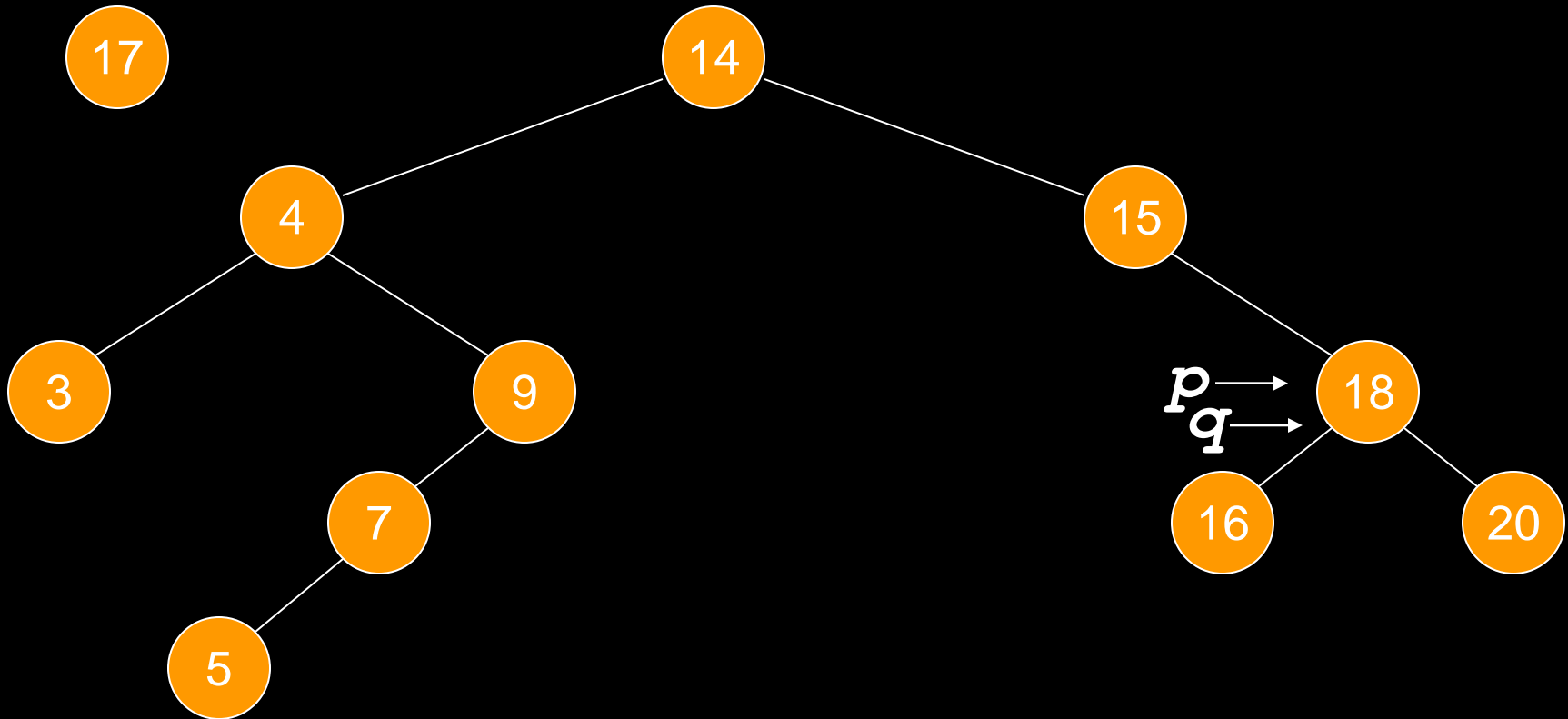


# Trace of insert



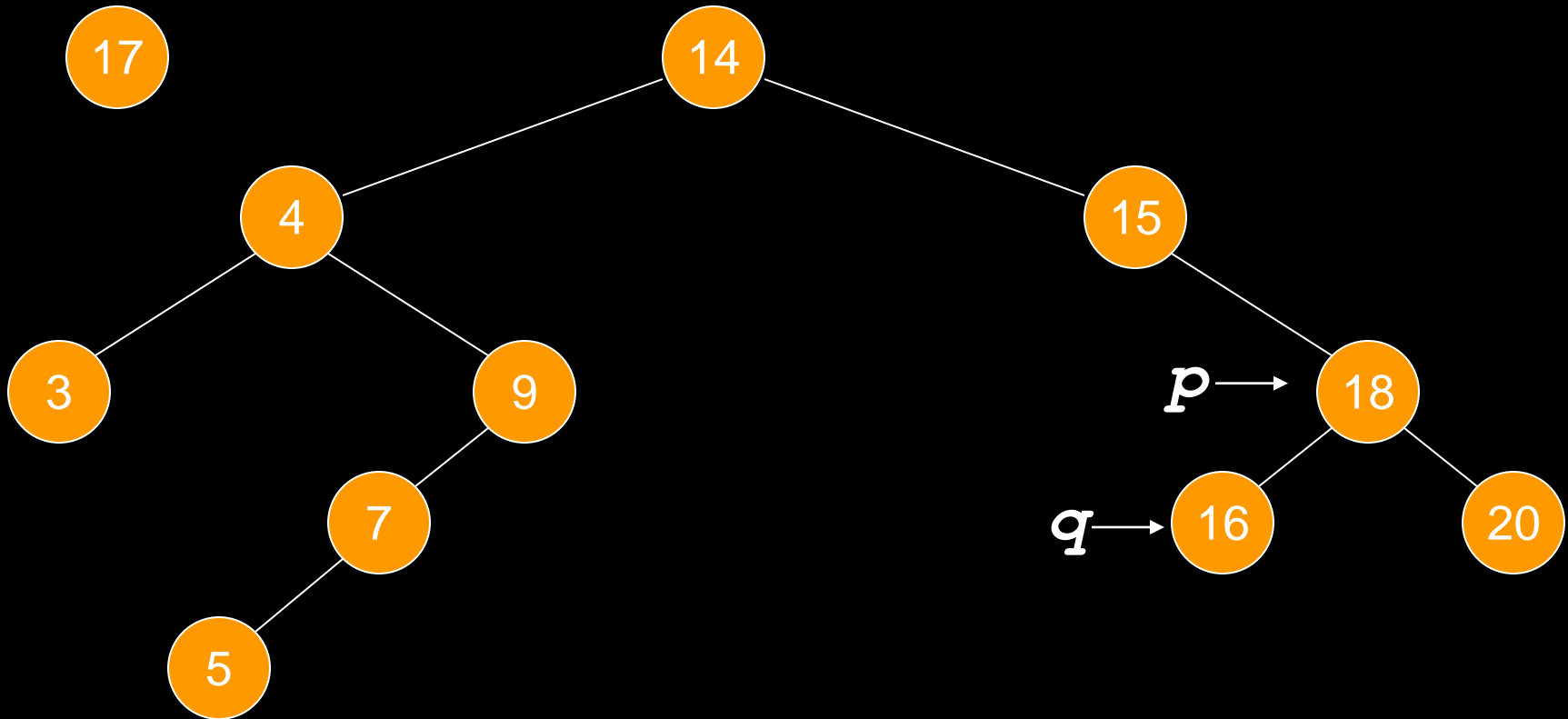
17, 9, 14, 5

# Trace of insert



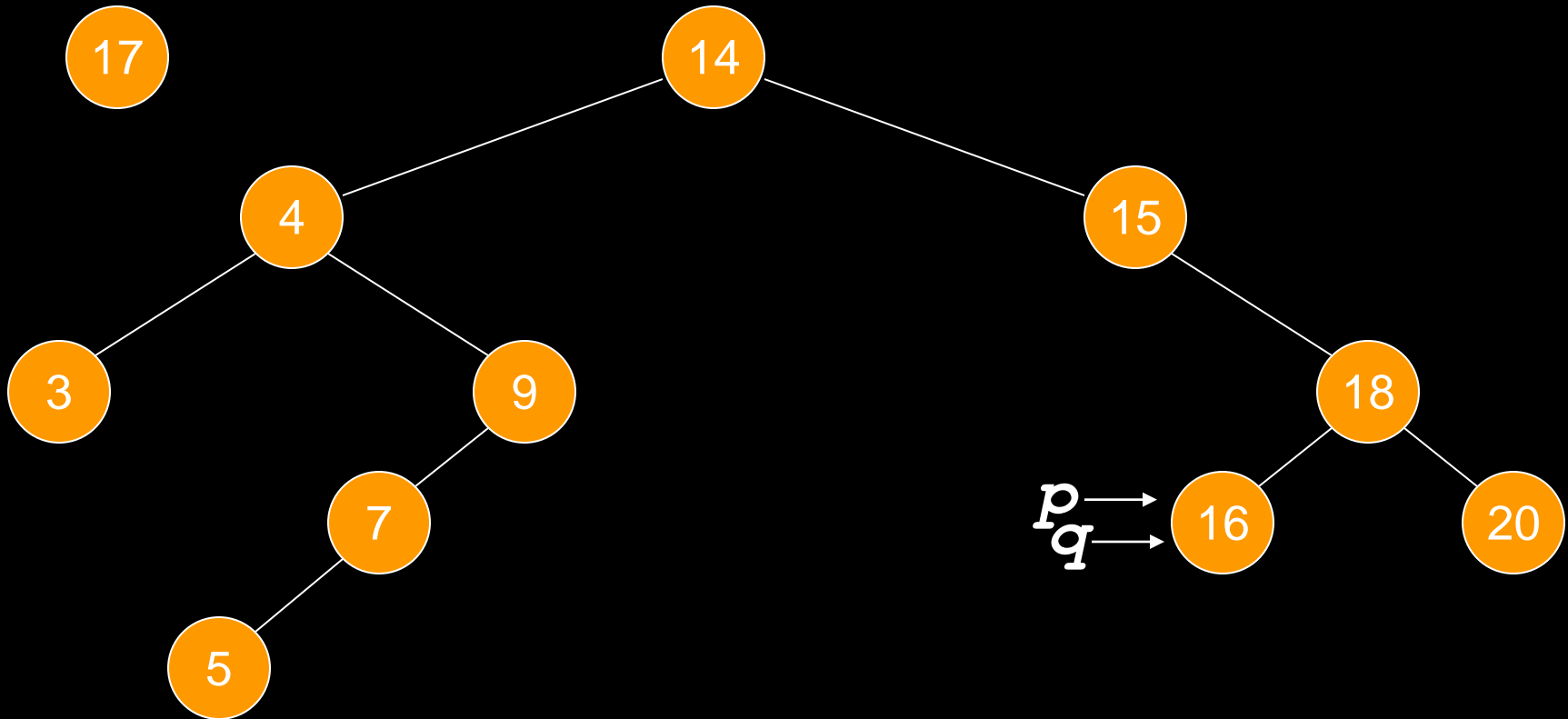
17, 9, 14, 5

# Trace of insert



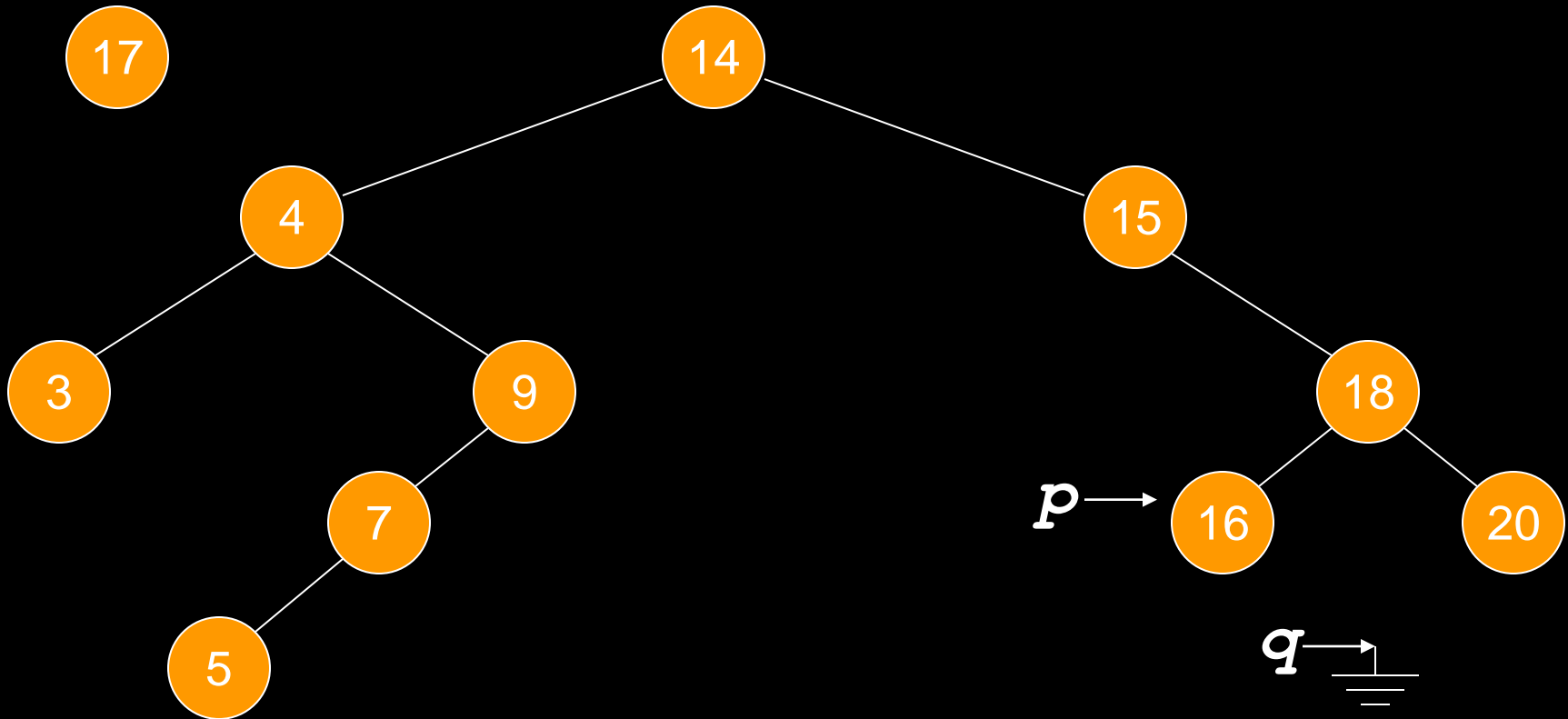
17, 9, 14, 5

# Trace of insert



17, 9, 14, 5

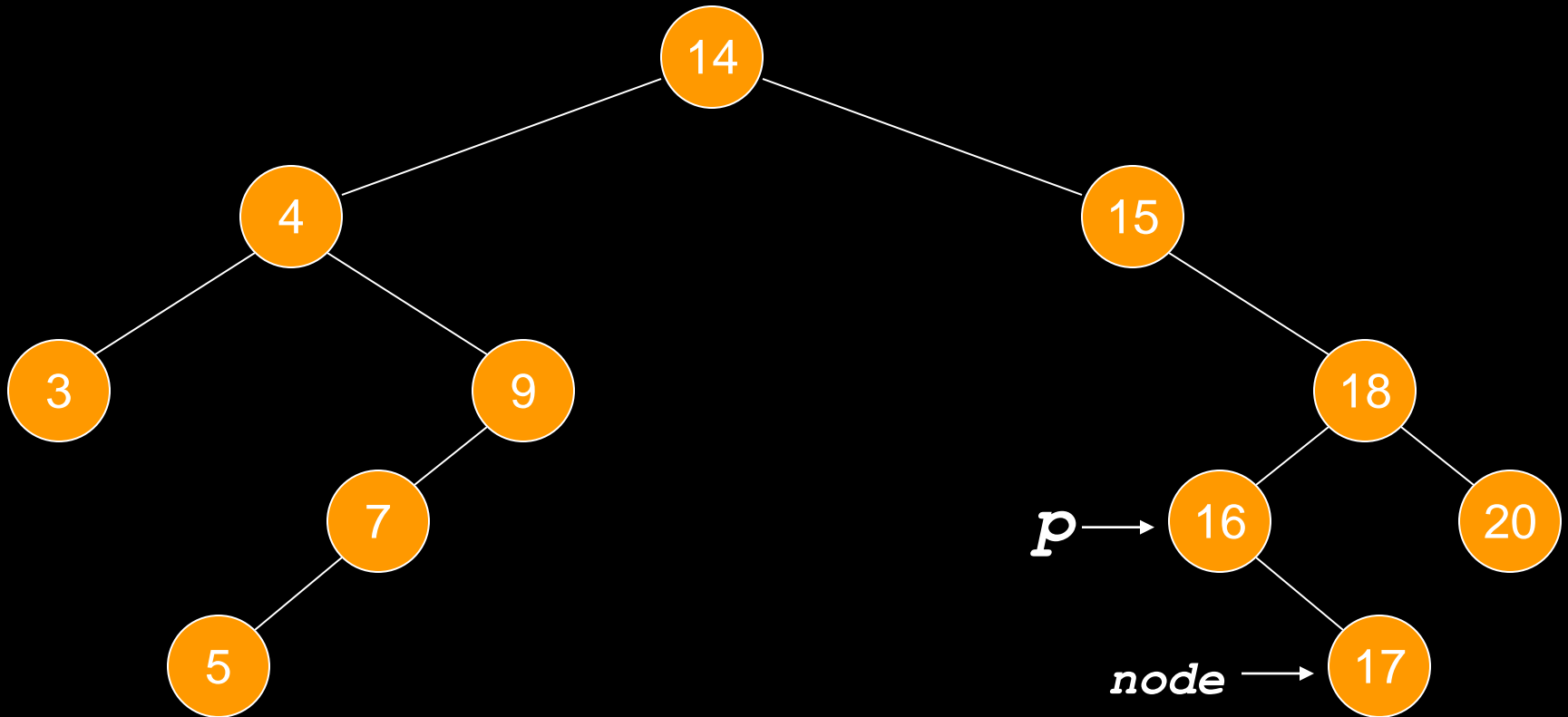
# Trace of insert



17, 9, 14, 5



# Trace of insert



17, 9, 14, 5

`p->setRight( node );`

# Cost of Search

- Given that a binary tree is level  $d$  deep. How long does it take to find out whether a number is already present?
- Consider the insert(17) in the example tree.
- Each time around the while loop, we did one comparison.
- After the comparison, we moved a level *down*.

# Cost of Search

- With the binary tree in place, we can write a routine *find(x)* that returns true if the number  $x$  is present in the tree, false otherwise.
- How many comparison are needed to find out if  $x$  is present in the tree?
- We do one comparison at each level of the tree until either  $x$  is found or  $q$  becomes NULL.

# Cost of Search

- If the binary tree is built out of  $n$  numbers, how many comparisons are needed to find out if a number  $x$  is in the tree?
- Recall that the depth of the complete binary tree built using ' $n$ ' nodes will be  $\log_2(n+1) - 1$ .
- For example, for  $n=100,000$ ,  $\log_2(100001)$  is less than 20; the tree would be 20 levels deep.

# Cost of Search

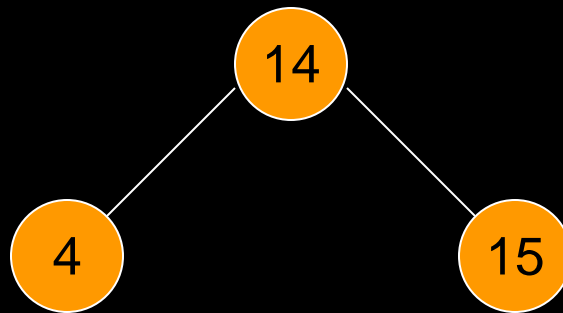
- If the tree is complete binary or nearly complete, searching through 100,000 numbers will require a maximum of 20 comparisons.
- Or in general, approximately  $\log_2(n)$ .
- Compare this with a linked list of 100,000 numbers. The comparisons required could be a maximum of  $n$ .

# Traversing a Binary Tree

- Suppose we have a binary tree, ordered (BST) or unordered.
- We want to print all the values stored in the nodes of the tree.
- In what order should we print them?

# Traversing a Binary Tree

- Ways to print a 3 node tree:



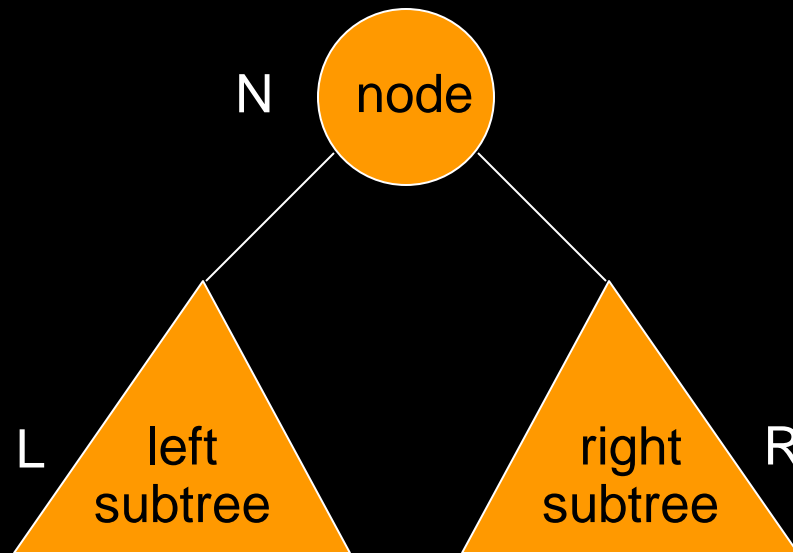
(4, 14, 15), (4,15,14)

(14,4,15), (14,15,4)

(15,4,14), (15,14,4)

# Traversing a Binary Tree

- In case of the general binary tree:



(L,N,R), (L,R,N)

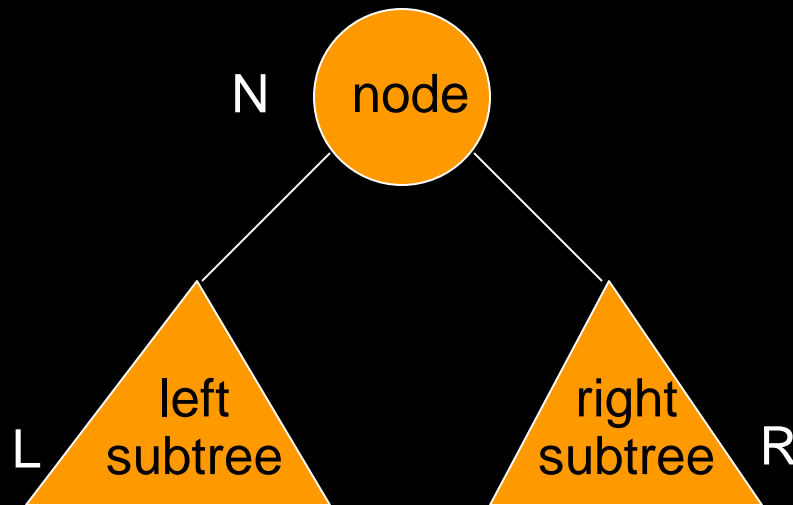
(N,L,R), (N,R,L)

(R,L,N), (R,N,L)



# Traversing a Binary Tree

- Three common ways



Preorder: (N,L,R)

Inorder: (L,N,R)

Postorder: (L,R,N)

# Traversing a Binary Tree

- There are three main ways to traverse a tree:
  - Pre-order:
    - (1) visit node,
    - (2) recursively visit left subtree,
    - (3) recursively visit right subtree
  - In-order:
    - (1) recursively visit left subtree,
    - (2) visit node,
    - (3) recursively right subtree
  - Post-order:
    - (1) recursively visit left subtree,
    - (2) recursively visit right subtree,
    - (3) visit node
- In different situations, we use different traversal algorithms.

# Traversing a Binary Tree

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

# Traversing a Binary Tree

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

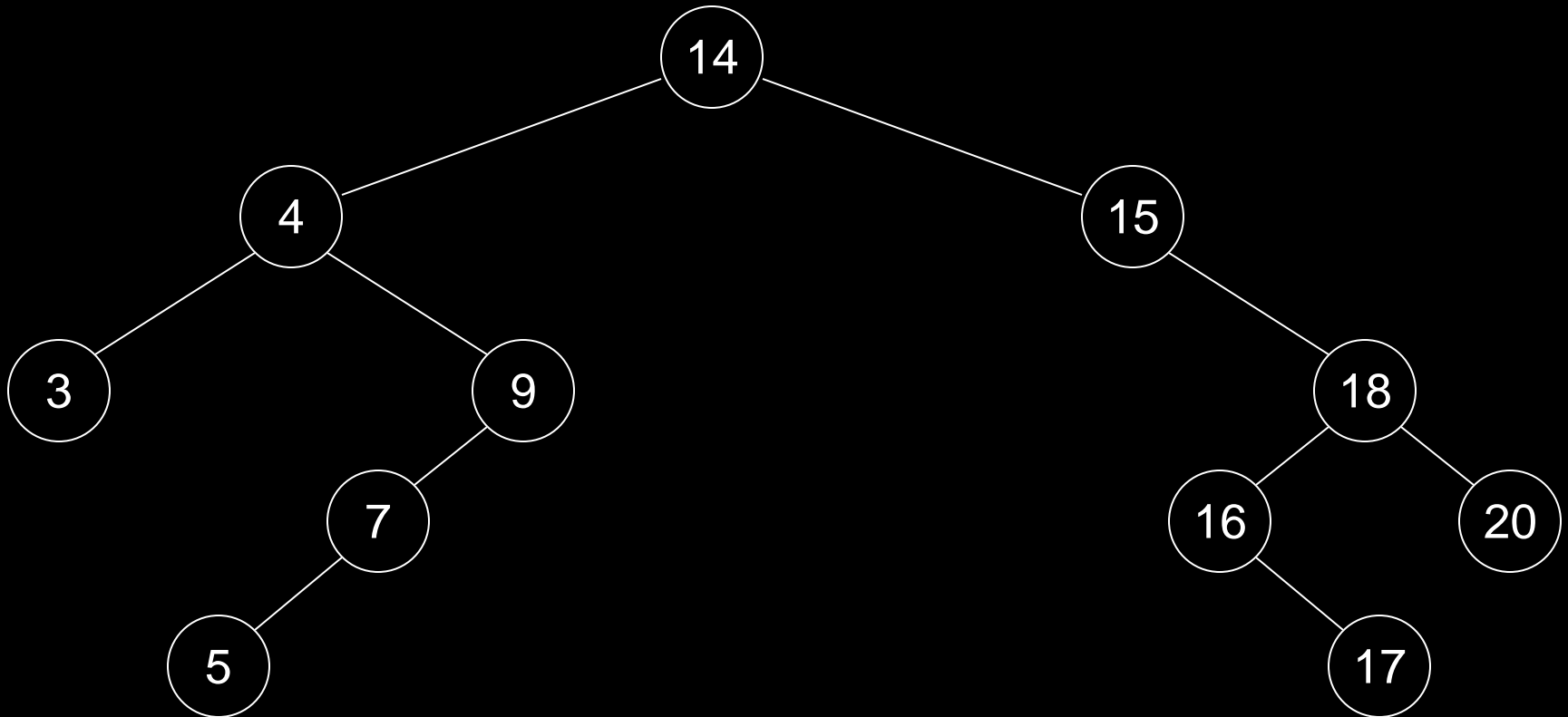
# Traversing a Binary Tree

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

# Traversing a Binary Tree

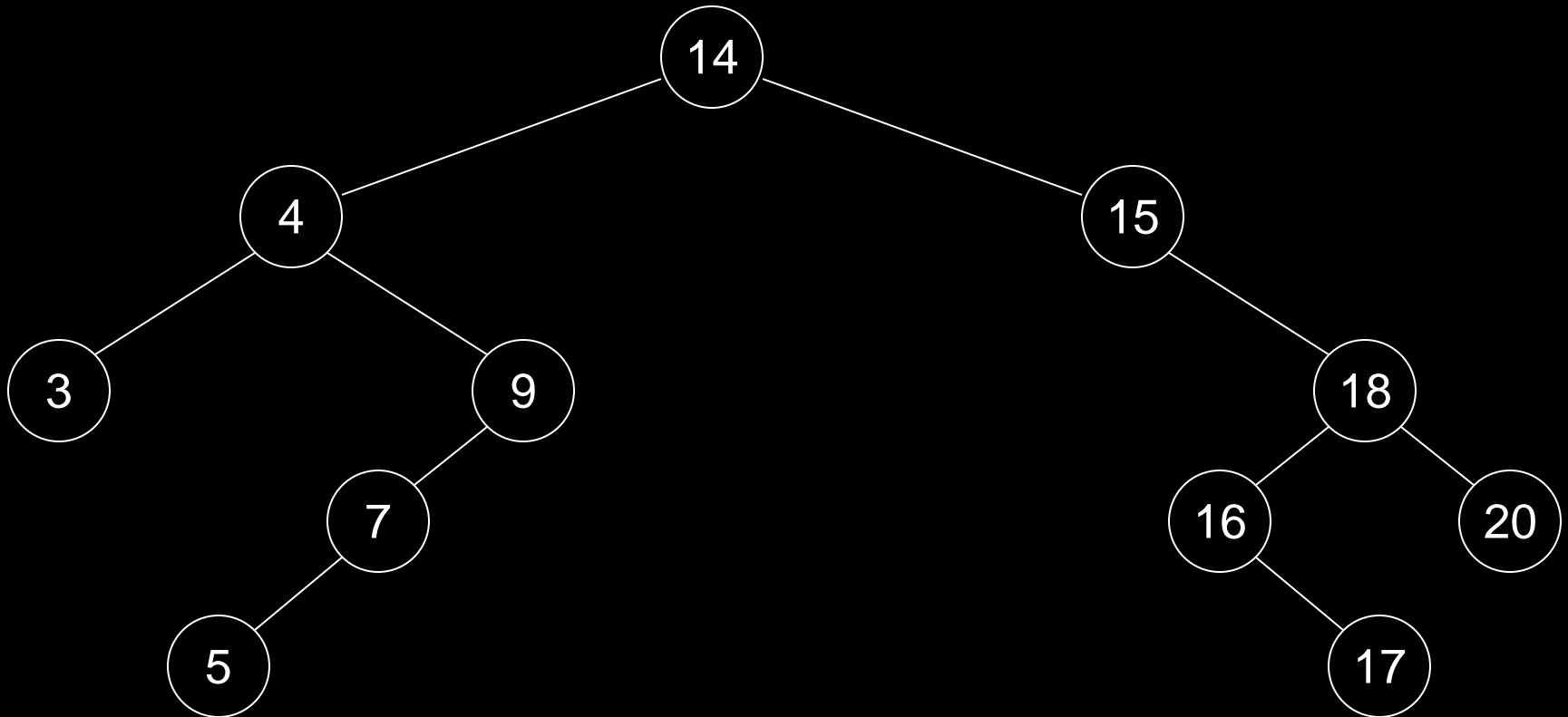
```
cout << "inorder: ";    preorder( root );  
cout << "inorder: ";    inorder( root );  
cout << "postorder: ";  postorder( root );
```

# Traversing a Binary Tree



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

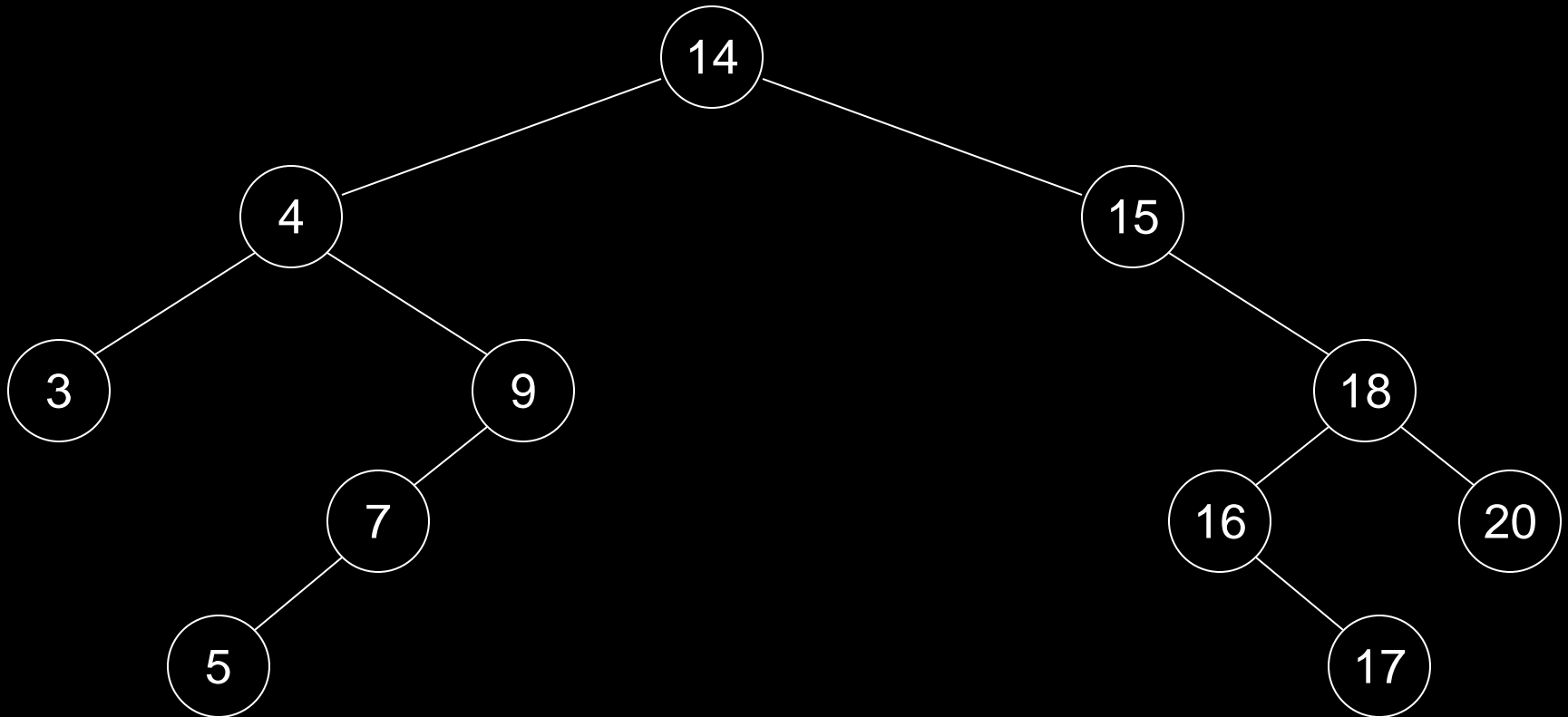
# Traversing a Binary Tree



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

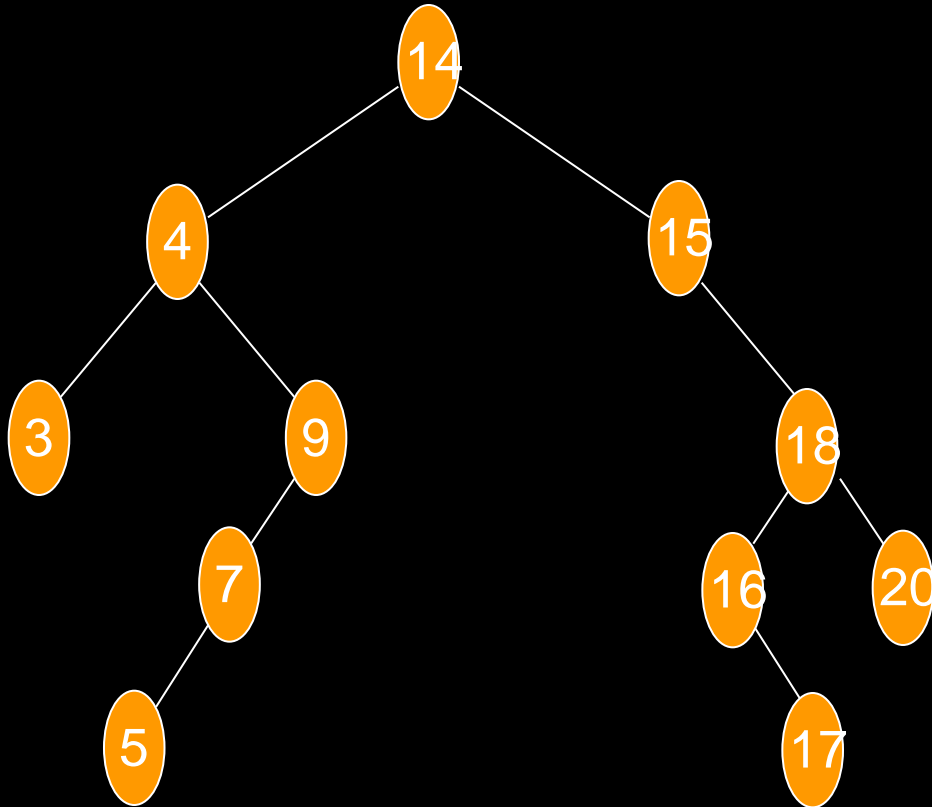


# Traversing a Binary Tree



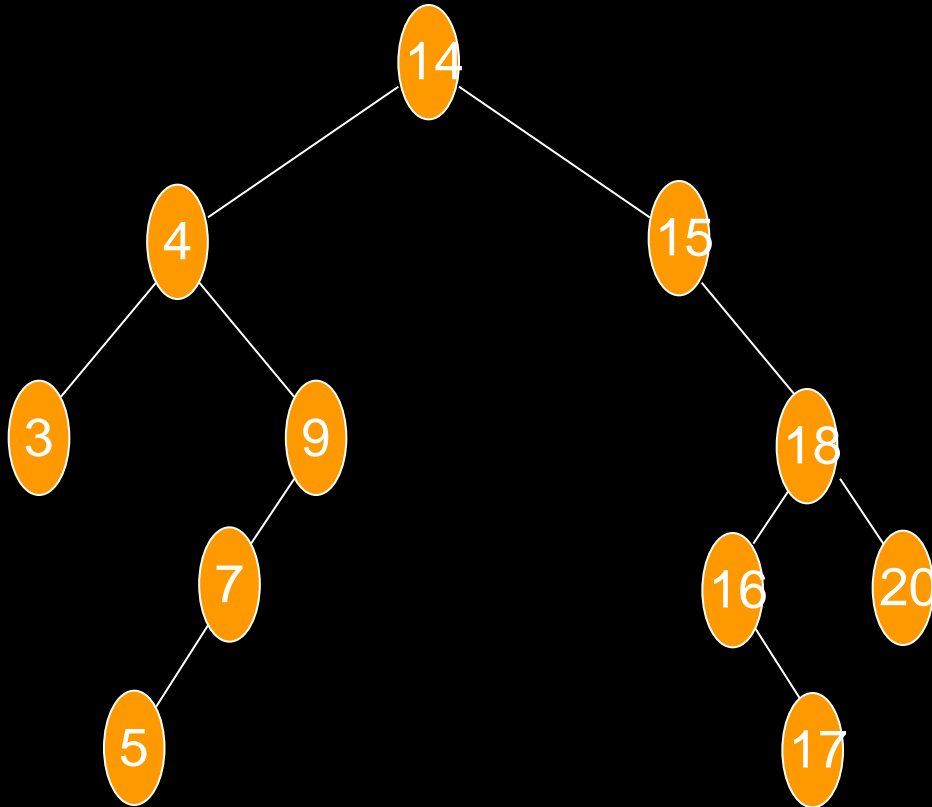
Postorder: 3 5 7 9 4 17 16 20 18 15 14

# Recursion: preorder



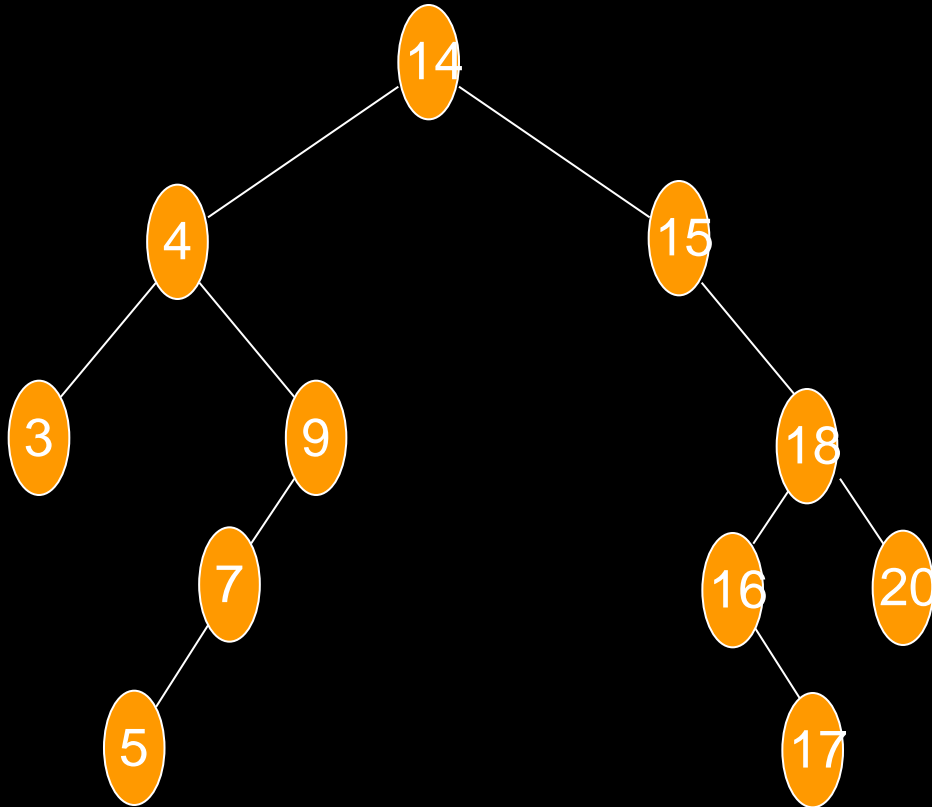
```
preorder(14)
14
  ..preorder(4)
  4
    ....preorder(3)
    3
      .....preorder(null)
      .....preorder(null)
      ....preorder(9)
      9
        .....preorder(7)
        7
          .....preorder(5)
          5
            .....preorder(null)
            .....preorder(null)
            .....preorder(null)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

# Recursion: preorder



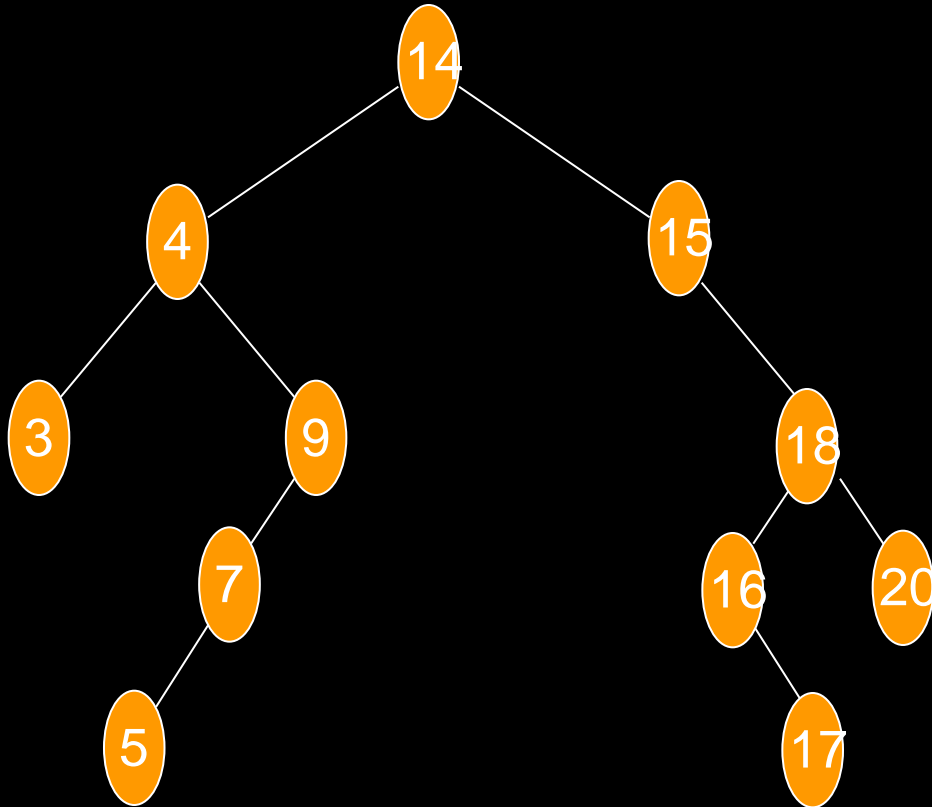
```
..preorder(15)
15
....preorder(null)
....preorder(18)
18
.....preorder(16)
16
.....preorder(null)
.....preorder(17)
17
.....preorder(null)
.....preorder(null)
.....preorder(20)
20
.....preorder(null)
.....preorder(null)
```

# Recursion: inorder



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

# Recursion: inorder



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

# BST– Other Functions - Example

```
TreeNode* returnroot() {return root;}           //return root node address

int totalnodes() {return bstsize;}              //return total nodes in BST

int duplicate() {return dupcount;}              // return total number of duplicates
                                              // encountered during building of BST

int isLeaf() {if ((left==NULL)&&(right==NULL)) return 1;
              else return 0; }

};
```

Write a function “**depth()**” which returns the depth of current node (at which the current pointer points)

Pototype: **int depth(TreeNode\*);**

Write a function “**brother()**” which returns the pointer to brother of current node

ProtoType: **TreeNode\* brother(TreeNode\*);**

Write a function “**search()**” which searches a value in BST, if found then return pointer to it, else return NULL

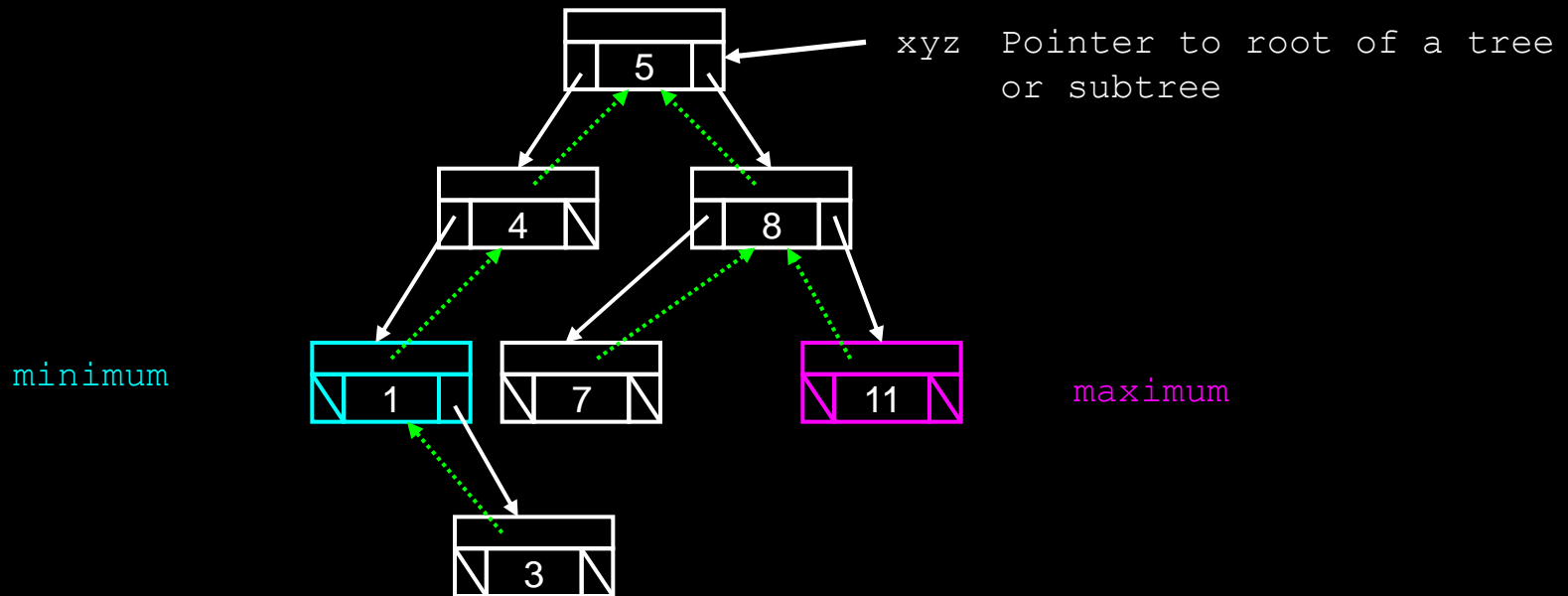
ProtoType: **TreeNode\* search(T);** (very similar to insert( ) function)

Write two functions “**isLeft()**” which returns 1 if current node is left child of parent, otherwise return 0

Write a function “**isRight()**” which returns 1 if current node is right child of parent, otherwise return 0

# Finding Minimum & Maximum element in BST – Basic Idea

- The binary-search-tree property guarantees that:
  - The **minimum** is located at the **left-most** node.
  - The **maximum** is located at the **right-most** node.



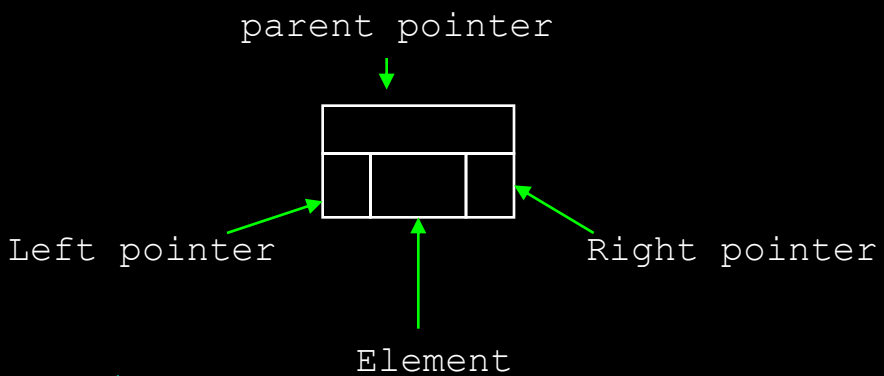
```
treeNode* minimum(treeNode* xyz)
{
    while( xyz->getLeft() != NULL )
        { xyz = xyz->getLeft() };
    return xyz;
}
```

```
treeNode* maximum(treeNode* xyz)
{
    while( xyz->getRight() != NULL )
        { xyz = xyz->getRight() };
    return xyz;
}
```

# Building Binary Search Trees – Node - Example

```
class BStree {
private:
    class TreeNode {
    private:
        int data;
        TreeNode *left,*right,*parent;
    public:
        int getdata() { return data; };
        void setdata(int data) { this->data = data;};
        TreeNode* getleft() { return left; };
        void setleft(TreeNode *left) { this->left = left; };
        TreeNode* getright() { return right; };
        void setright(TreeNode *right) { this->right = right; };
        TreeNode* getparent() { return parent; };
        void setparent(TreeNode *parent) { this->parent = parent; };
    };

    int bstsize, dupcount;
    TreeNode *root,*current,*temp;
```





# Building Binary Search Trees – constructor - Example

```
public:
    // Constructor
    BStree() {
        root = NULL;
        current = NULL;
        temp = NULL;
        bstsize = 0;
        dupcount = 0;
```

# Building Binary Search Trees – insert - Example

```
int insert(int data) {  
    TreeNode *newNode=new TreeNode; // create new node  
    newNode->setdata(data);  
    newNode->setleft(NULL); newNode->setright(NULL); newNode->setparent(NULL);  
// if tree is empty  
    if (root==NULL) { root=newNode; current=newNode; bstsize=1; return 0;}  
    else if (root!=NULL) //if not empty, find proper location and point current to it  
        { temp = current = root;  
          while( (data!=temp->getdata()) && (current!=NULL) )  
              { temp = current;  
                if ( data<temp->getdata() )    current = temp->getleft();  
                else                          current = temp->getright();          }  
          if( data == temp->getdata() )    //duplicate data, trash new node  
              { dupcount++; delete newNode; }  
          else if( data<temp->getdata() ) //otherwise link new node to the tree  
              { temp->setleft( newNode ); newNode->setparent(temp);bstsize++;}  
          else      { temp->setright( newNode ); newNode->setparent(temp);bstsize++;}  
              }  
    return 0;  
}
```

# BST Class – Example-

```
void pre(TreeNode* TNode)
{
    if( TNode != NULL )
    {
        cout << (TNode->getdata())<<" ";
        pre(TNode->getleft());
        pre(TNode->getright());
    }
}

void in(TreeNode* TNode)
{
    if( TNode != NULL )
    {
        in(TNode->getleft());
        cout << (TNode->getdata())<<" ";
        in(TNode->getright());
    }
}

void post(TreeNode* TNode)
{
    if( TNode != NULL )
    {
        post(TNode->getleft());
        post(TNode->getright());
        cout << (TNode->getdata())<<" ";
    }
}
```

# BST Class – Example-

```
TreeNode* returnroot() {return root;}
```

```
int totalnodes() {return bstsize;}
```

```
int duplicate() {return dupcount;}
```

```
int isLeaf() {if ((left==NULL)&&(right==NULL)) return 1;  
              else return 0; }
```

```
};
```

# BST Class – Client Program Example-

```
#include <iostream>

int main()
{
    int x[15] = { 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5 };
    BStree T;
    for(int i=0; i<15; i++ )
    {
        T.insert(x[i] );
    }
    cout << "\nTotal Duplicate Encountered = " << T.duplicate();
    cout << "\nTotal Nodes = " << T.totalnodes();
    cout << "\nPreOrder Traversal: "; T.pre(T.returnroot());
    cout << "\nInOrder Traversal: "; T.in(T.returnroot());
    cout << "\nPostOrder Traversal: "; T.post(T.returnroot());
    cout<<endl;
    system("pause");
    return 0;
}
```

# Removing a Node in our BST class – Basic Idea

Removing an item disrupts the tree structure

We can't just delete a node like we could with a linked list. The tree node could have one or two children, so it's not immediately clear what should become of them.

Basic idea:

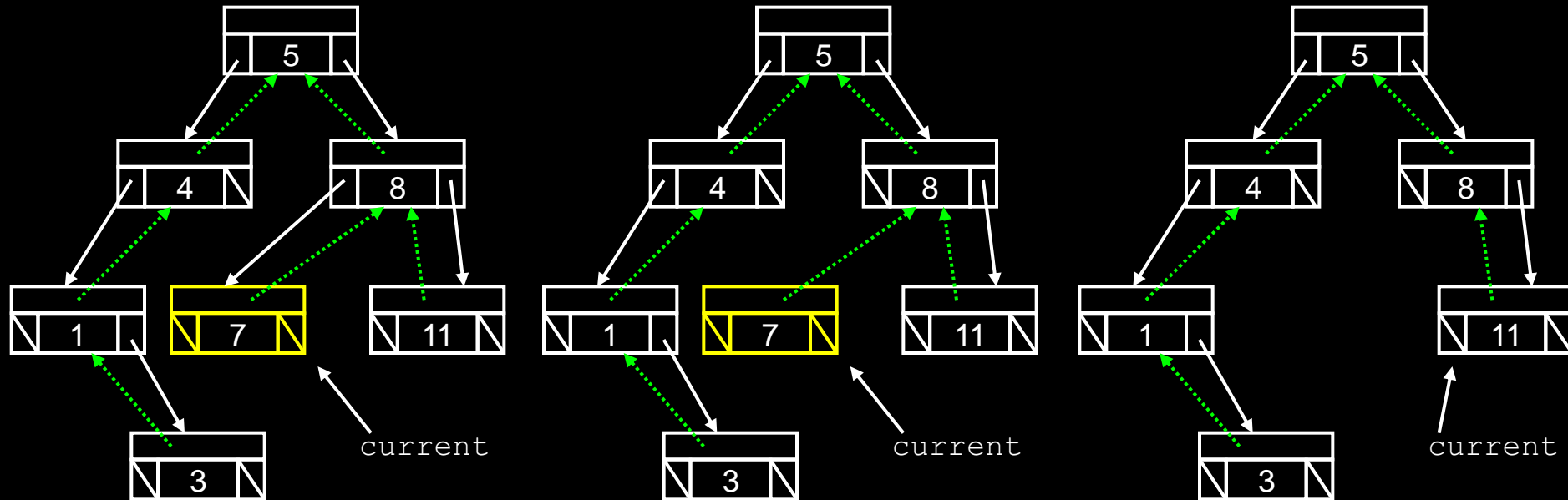
- a. **Find the node** (use search()) that is to be removed.
- b. Then **"fix"** the tree so that it is still a binary search tree.

Three cases:

1. **node to be removed has no children**
2. **node to be removed has one child subtree**
3. **node to be removed has two child subtrees**

# Removing a Node in our BST class – Basic Idea

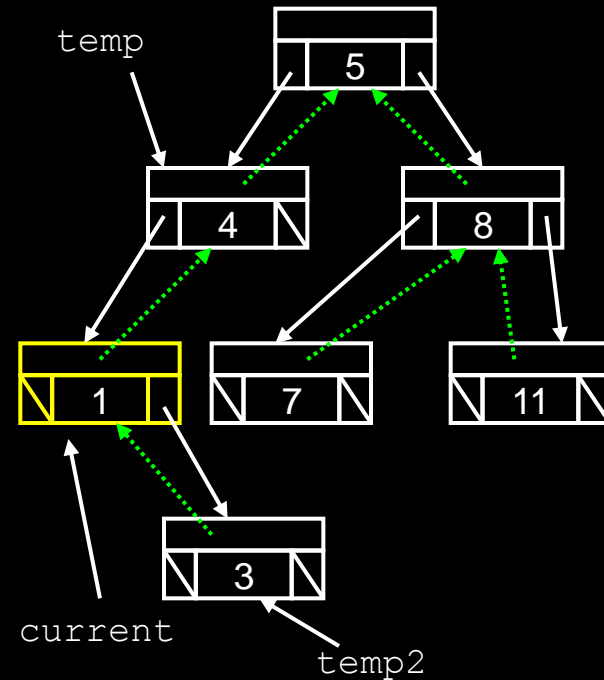
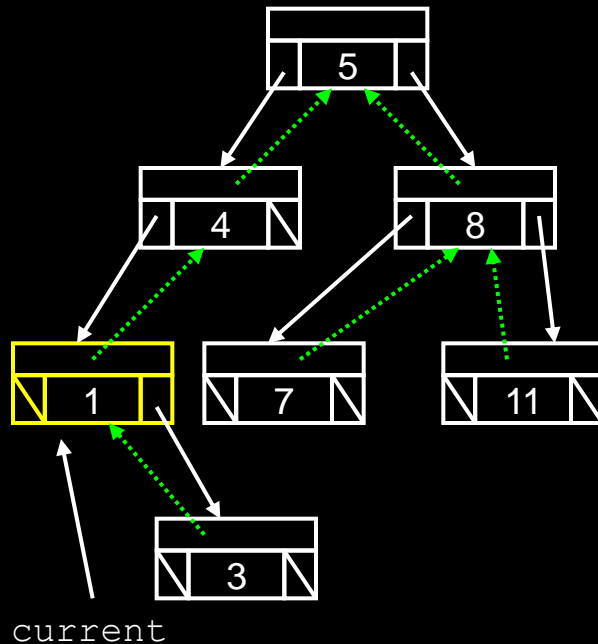
## 1. node to be removed has no children



1. use function “isLeaf()” to determine the current node has no children
2. use functions “isLeft()” and “isRight()” to determine whether current node is left or right child,
3. set the respective pointer field of parent to NULL.
4. delete the node pointed by current

# Removing a Node in our BST class – Basic Idea

2. node to be removed has one child subtree: replace the removed node with its subtree

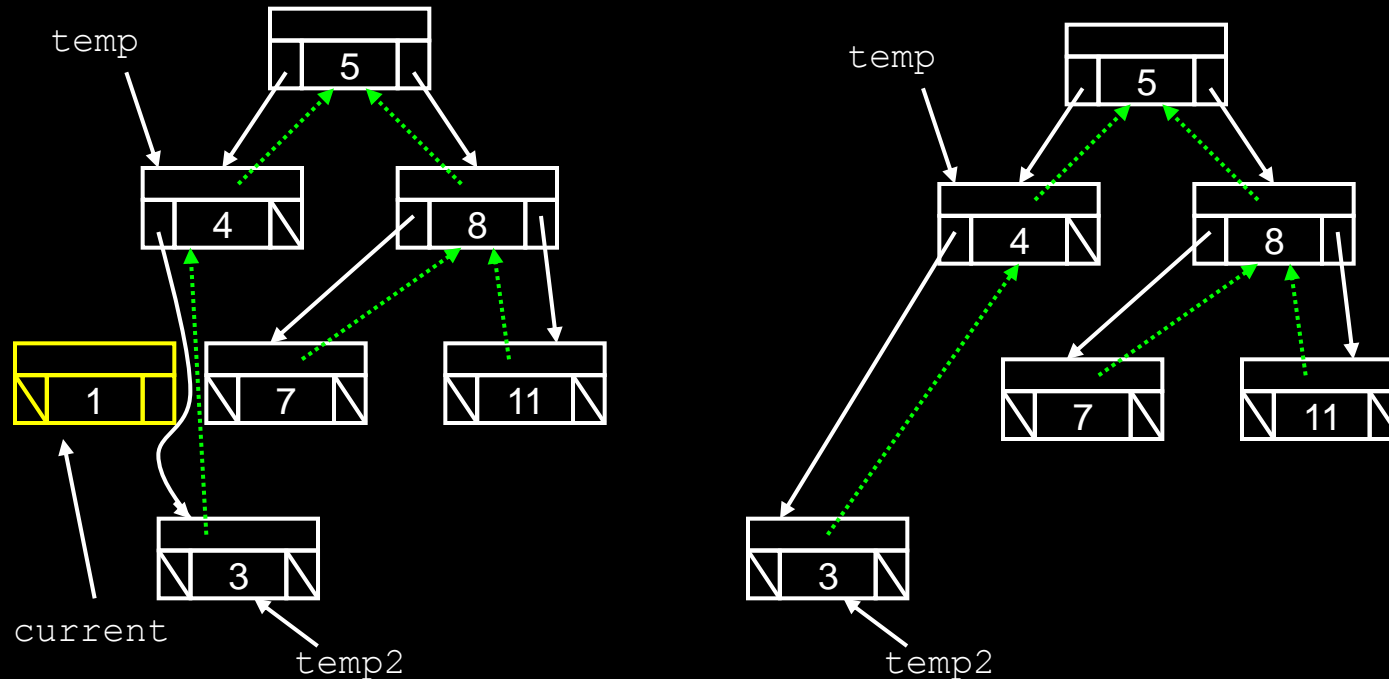


1. `else if ((current->getleft() == NULL) || (current->getRight() == NULL))`
2. Determine whether left or right child, in this case left child
3. Point temp to parent, declare another pointer temp2 and point it to current's child
3. `temp->setLeft(temp2); temp2->setParent(temp);`
4. Delete the node pointed by current



# Removing a Node in our BST class – Basic Idea

2. node to be removed has one child subtree: (cont..) replace the removed node with its subtree

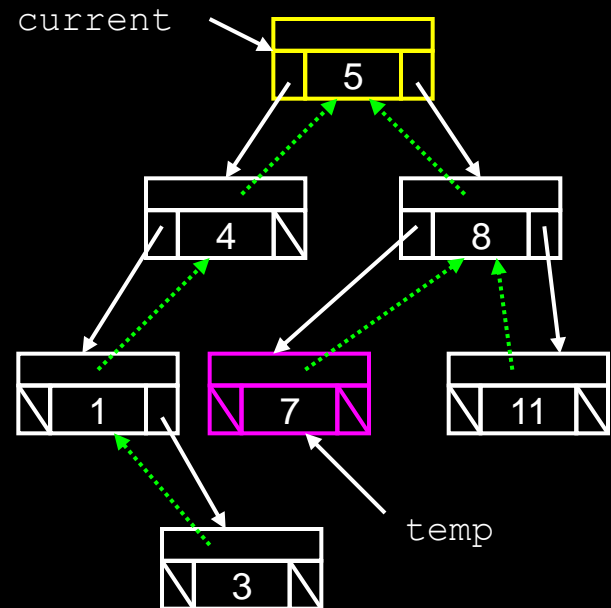
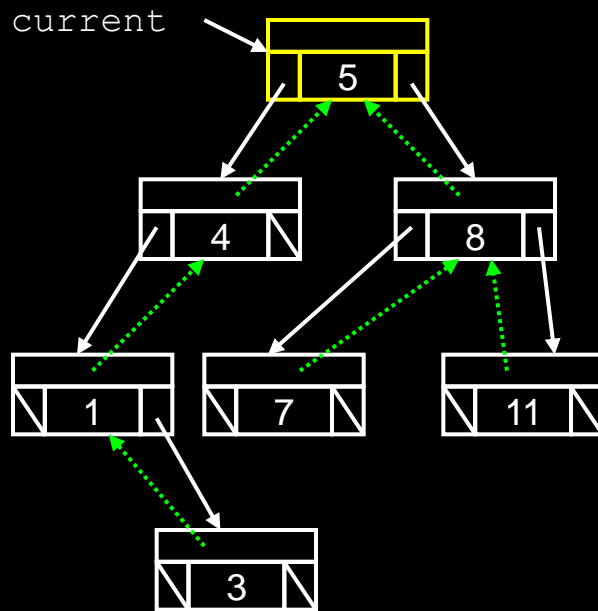


1. `else if ((current->getleft() == NULL) || (current->getRight() == NULL))`
2. Determine whether left or right child, in this case left child
3. Point `temp` to parent, declare another pointer `temp2` and point it to `current`'s child
3. `temp->setLeft(temp2); temp2->setParent(temp);`
4. Delete the node pointed by `current`

# Removing a Node in our BST class – Basic Idea

## 3. node to be removed has two child subtrees:

replace the node with its successor (leftmost element of right subtree), then remove the successor from the tree

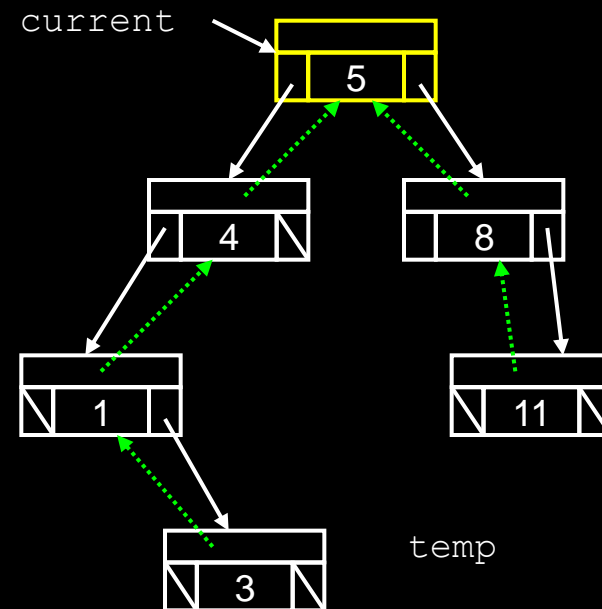
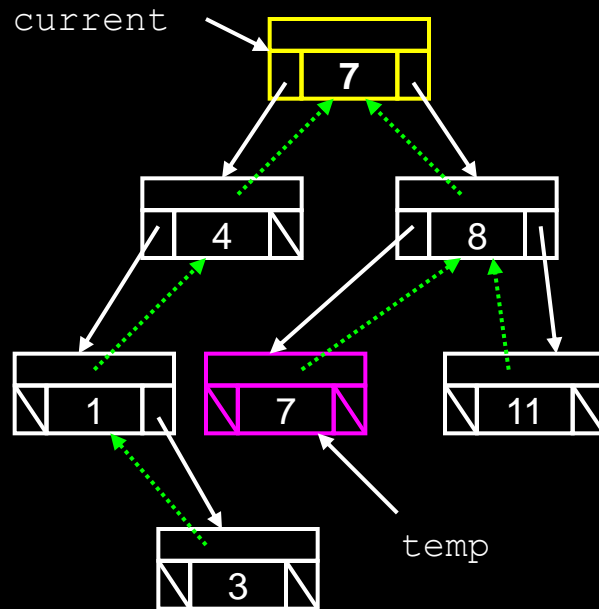


```
1. if ((current->getLeft() != NULL) && (current->getRight() != NULL))
2. temp = current->getRight()
   while( temp->getLeft() != NULL ) { temp=temp->getLeft() }
3. current->setData(temp->getData());
4. delete temp;
```

# Removing a Node in our BST class – Basic Idea

## 3. node to be removed has two child subtrees: (cont..)

replace the node with its successor (leftmost element of right subtree), then remove the successor from the tree



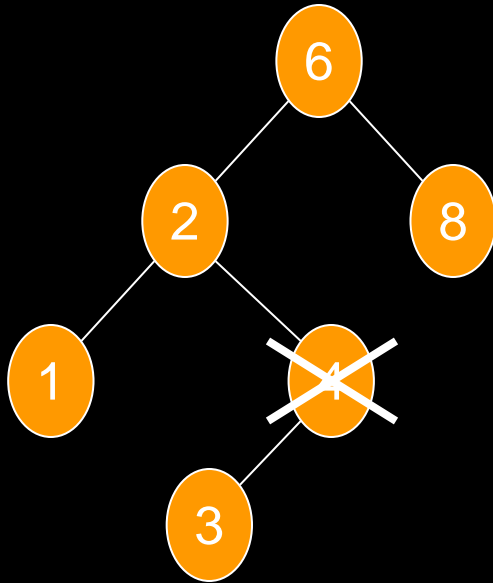
```
1. if ((current->getleft() != NULL) && (current->getRight() != NULL))
2. temp = current->getRight()
   while( (temp->getLeft() != NULL) ) { temp=temp->getLeft() }
3. current->setData(temp->getData());
4. delete temp;
```

# Deleting a node in BST

- As is common with many data structures, the hardest operation is deletion.
- Once we have found the node to be deleted, we need to consider several possibilities.
- If the node is a *leaf*, it can be deleted immediately.

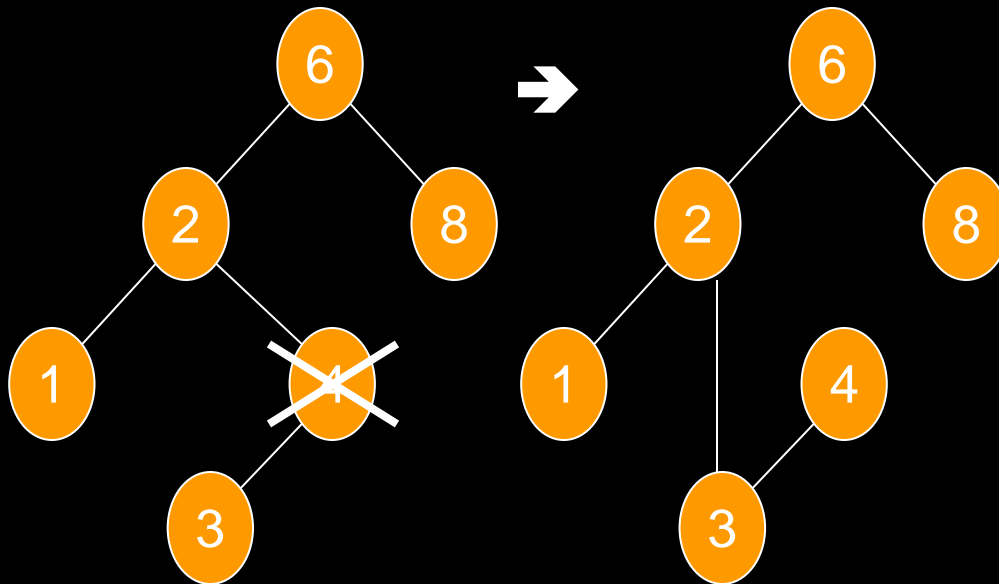
# Deleting a node in BST

- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to inorder successor.



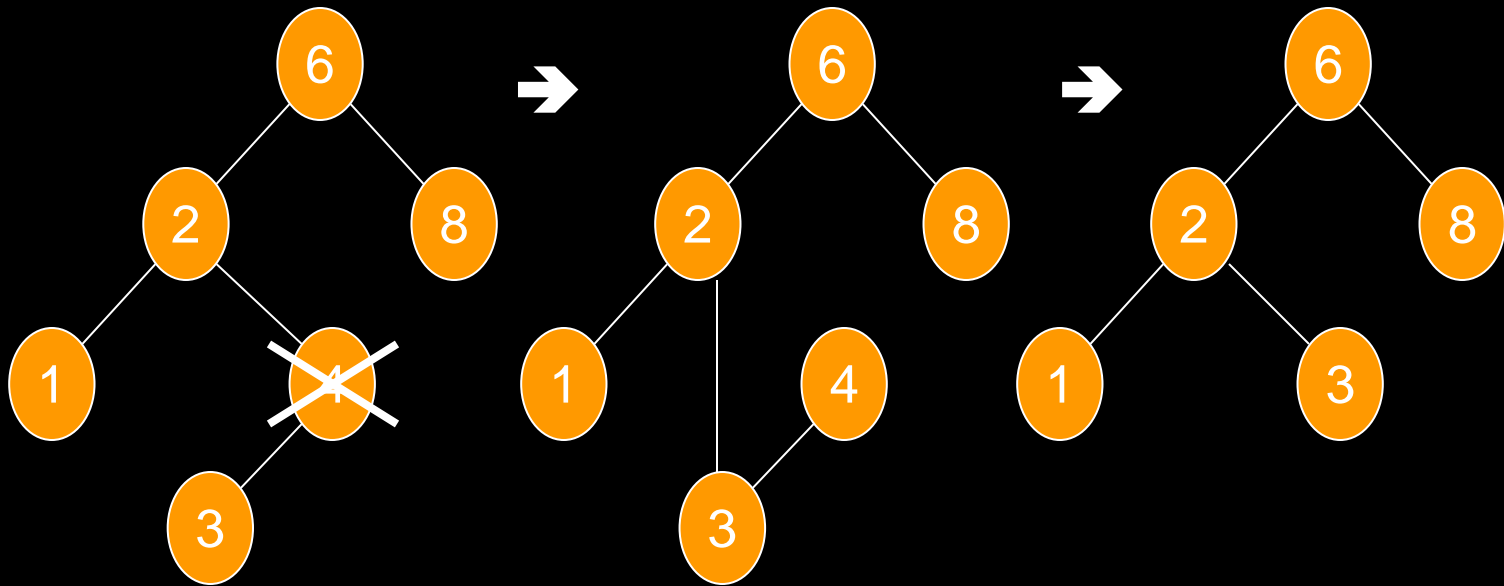
# Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.



# Deleting a node in BST

- The inorder traversal order has to be maintained after the delete.



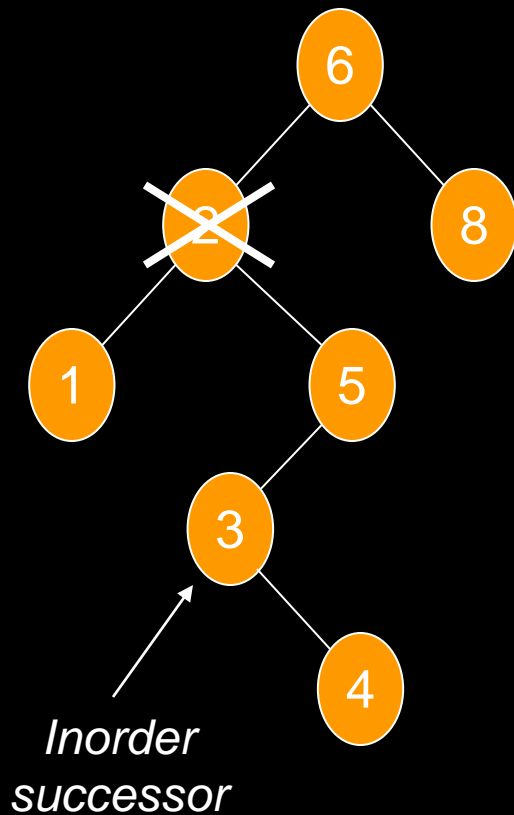
# Deleting a node in BST

- The complicated case is when the node to be deleted has both left and right subtrees.
- The strategy is to replace the data of this node with the smallest data of the right subtree and recursively delete that node.



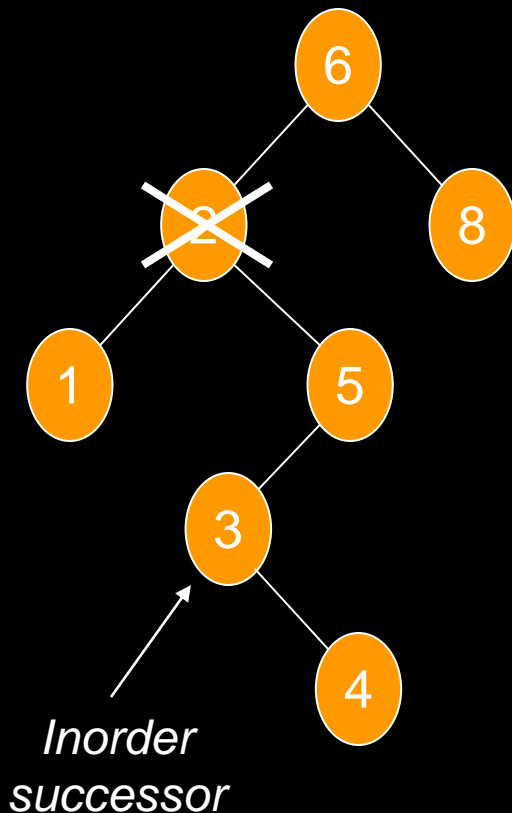
# Deleting a node in BST

Delete(2): locate inorder successor



# Deleting a node in BST

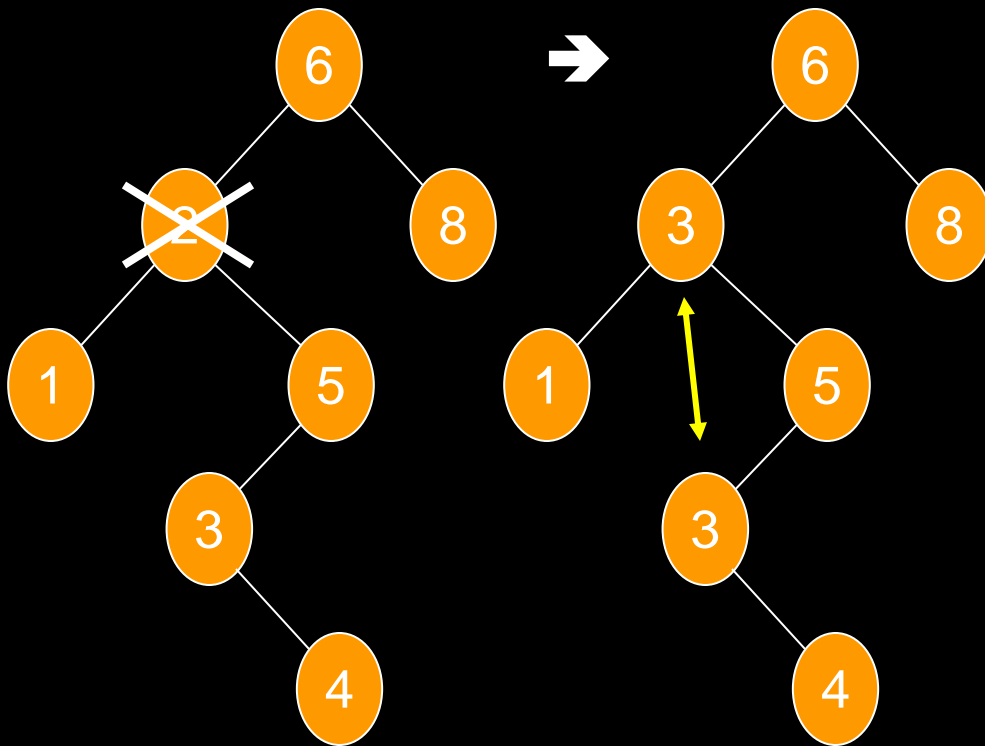
Delete(2): locate inorder successor



- Inorder successor will be the left-most node in the right subtree of 2.
- The inorder successor will not have a left child because if it did, that child would be the left-most node.

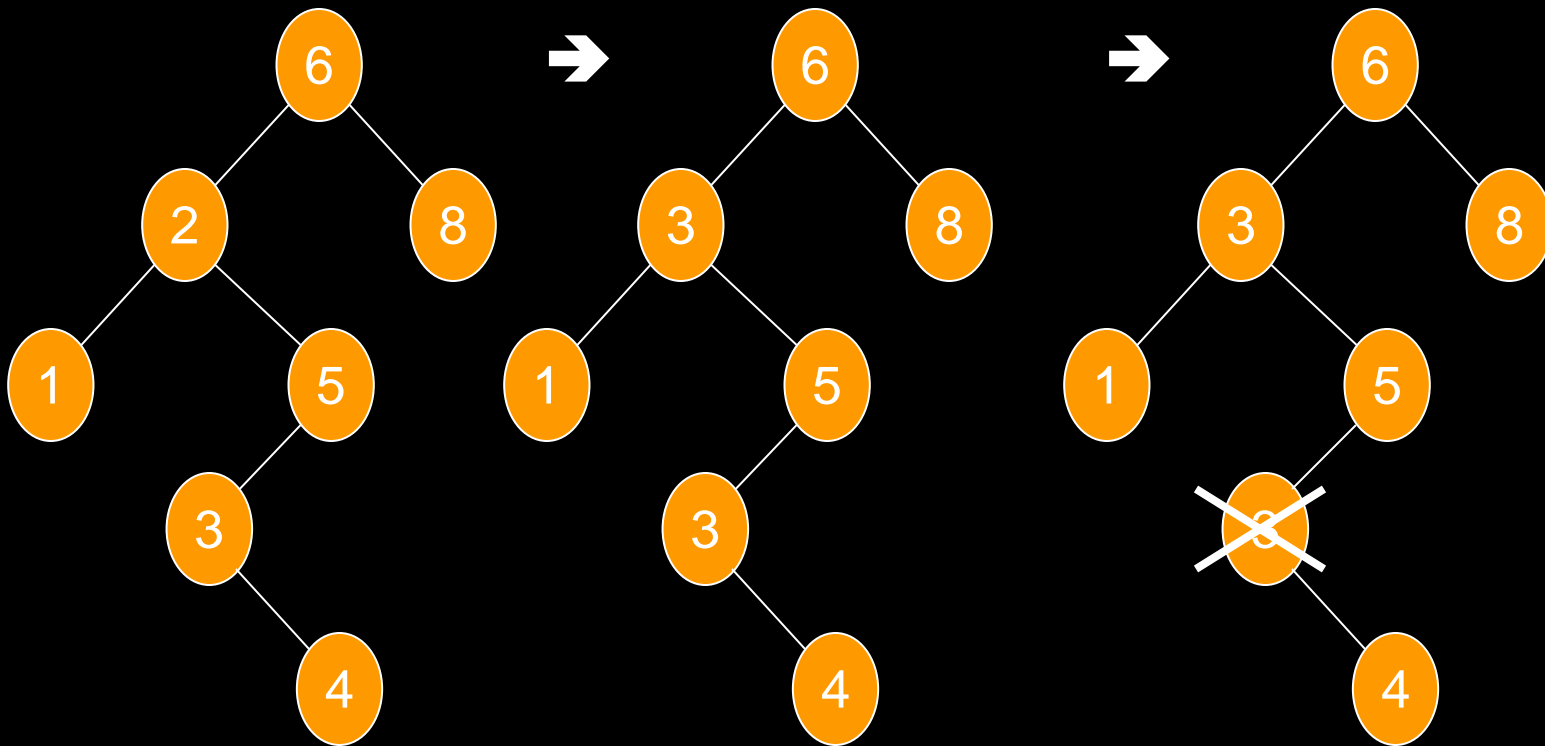
# Deleting a node in BST

Delete(2): copy data from inorder successor



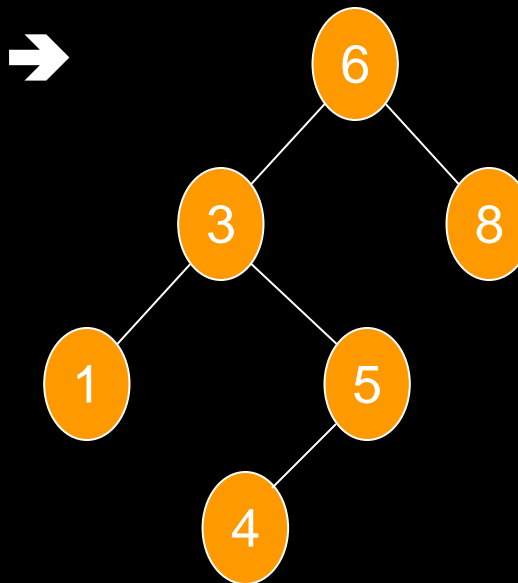
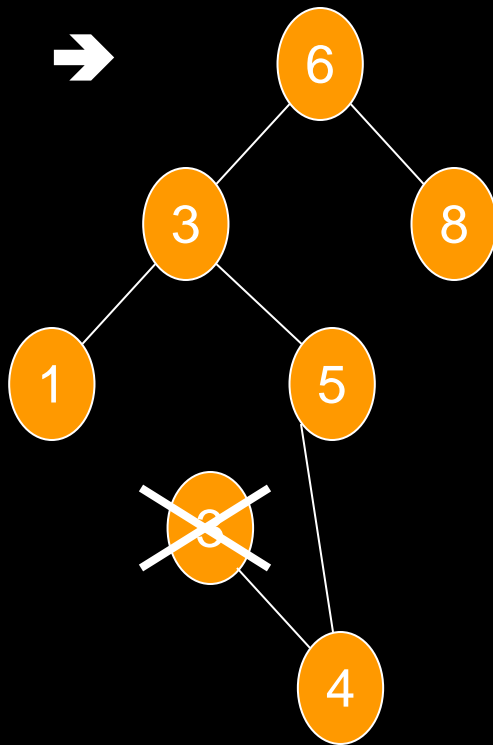
# Deleting a node in BST

Delete(2): remove the inorder successor



# Deleting a node in BST

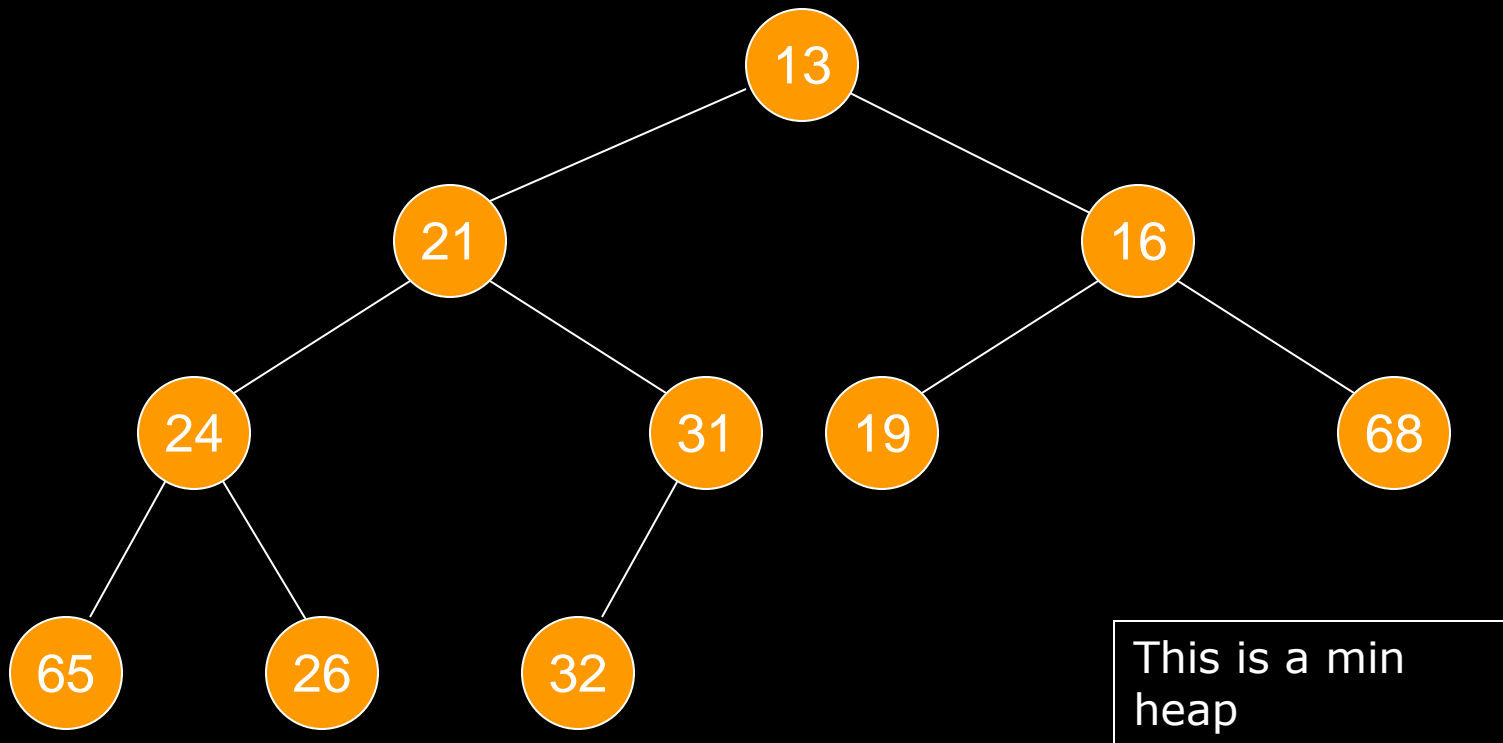
Delete(2)



# Heap

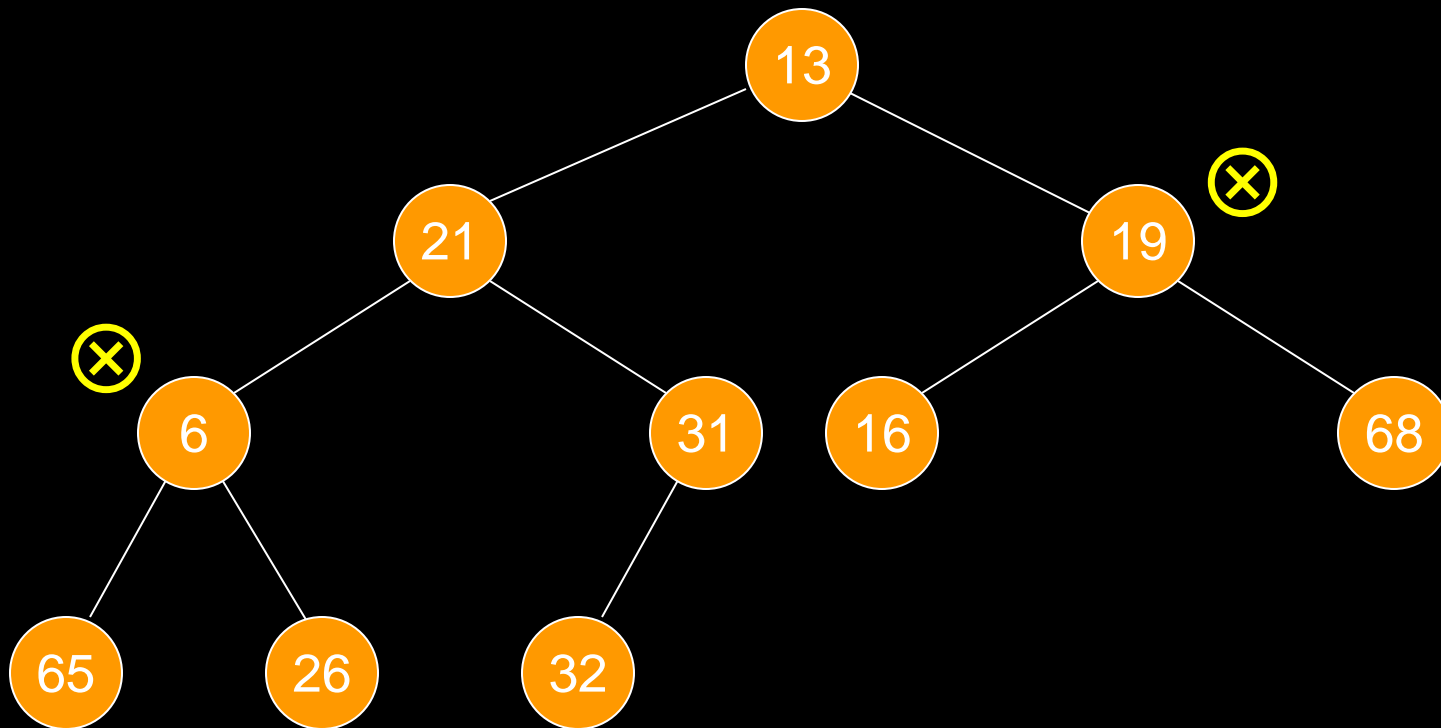
- A heap is a complete binary tree that conforms to the min or max heap order.
- The *heap order* property: in a (min) heap, for every node  $X$ , the key in the parent is smaller than (or equal to) the key in  $X$ .
- Or, the parent node has key smaller than or equal to both of its children nodes.

# Heap



# Heap

Not a heap: heap property violated

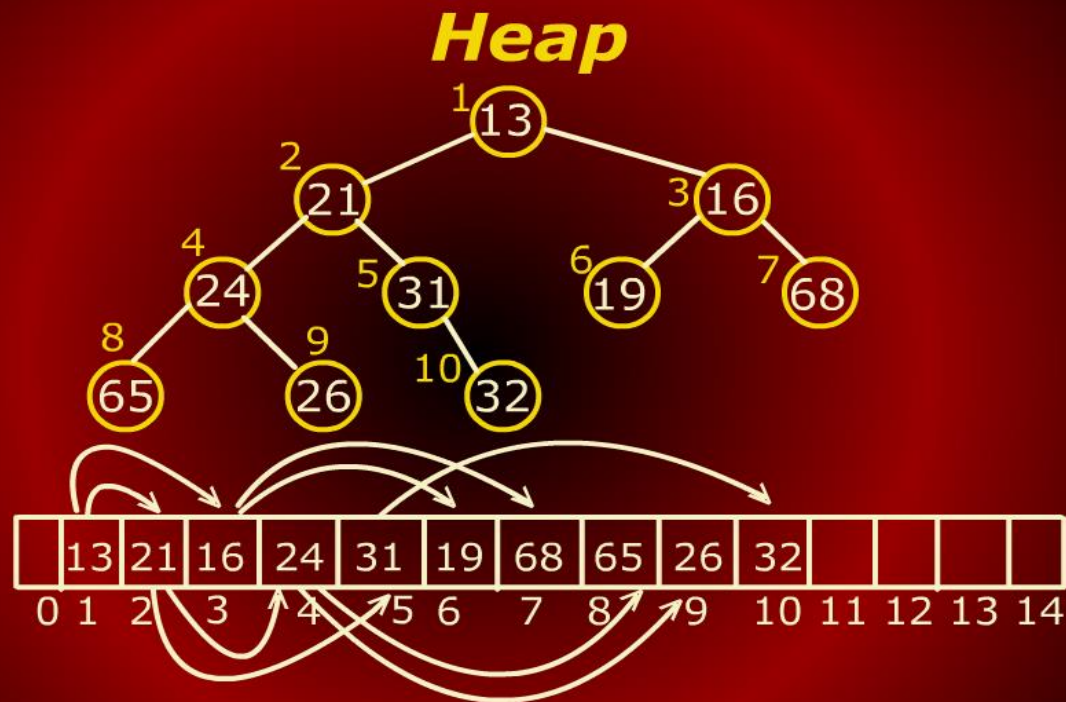




# Heap

- 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.

# Heap



# Heap

## *Heap*

Access to nodes involves simple arithmetic operations:

- $\text{left}(i)$ : returns  $2i$ , index of left child of node  $i$ .
- $\text{right}(i)$ : returns  $2i + 1$ , the right child.
- $\text{parent}(i)$  returns  $\lfloor i/2 \rfloor$ , the parent of  $i$ .

# Heap

## *Heapsort Algorithm*

- We build a max heap out of the given array of numbers  $A[1..n]$ .
- We repeatedly extract the the maximum item from the heap.
- Once the max item is removed, we are left with a hole at the root.
- To fix this, we will replace it with the last leaf in tree.

# Heap

## *Heap Sort*

HEAPSORT( array A, int n)

1 BUILD-HEAP(A, n)

2  $m \leftarrow n$

3 **while** ( $m \geq 2$ )

4 **do** SWAP(A[1],A[m])

5      $m \leftarrow m - 1$

6     HEAPIFY(A, 1,m)

# Heap

## *Heapify*

HEAPIFY( array A, int i, int n)

1  $l \leftarrow \text{LEFT}(i)$

2  $r \leftarrow \text{RIGHT}(i)$

3  $\text{max} \leftarrow i$

4 **if** ( $l \leq n$ ) and ( $A[l] > A[\text{max}]$ )

5   **then**  $\text{max} \leftarrow l$

6 **if** ( $r \leq n$ ) and ( $A[r] > A[\text{max}]$ )

7   **then**  $\text{max} \leftarrow r$

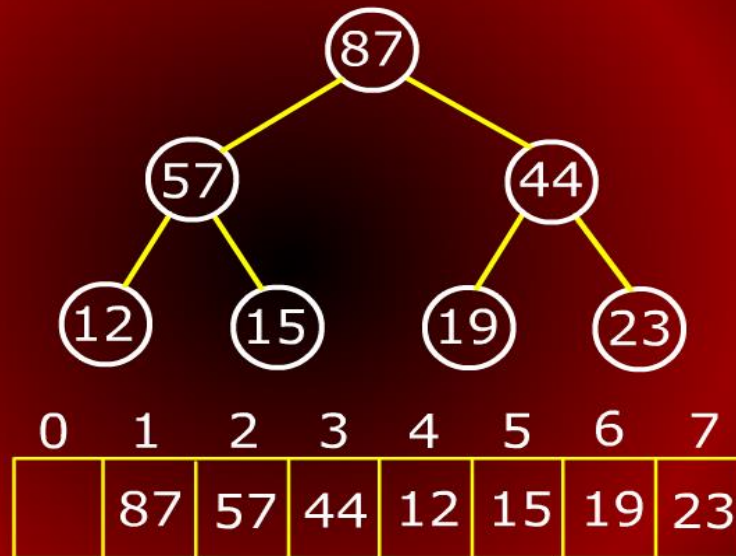
# Heap

## *Heapify*

```
3  max  $\leftarrow$  i
4  if ( $l \leq m$ ) and ( $A[l] > A[\text{max}]$ )
5    then max  $\leftarrow$  l
6  if ( $r \leq m$ ) and ( $A[r] > A[\text{max}]$ )
7    then max  $\leftarrow$  r
8  if (max  $\neq$  i)
9    then SWAP( $A[i], A[\text{max}]$ )
10 HEAPIFY( $A, \text{max}, m$ )
```

# Heap

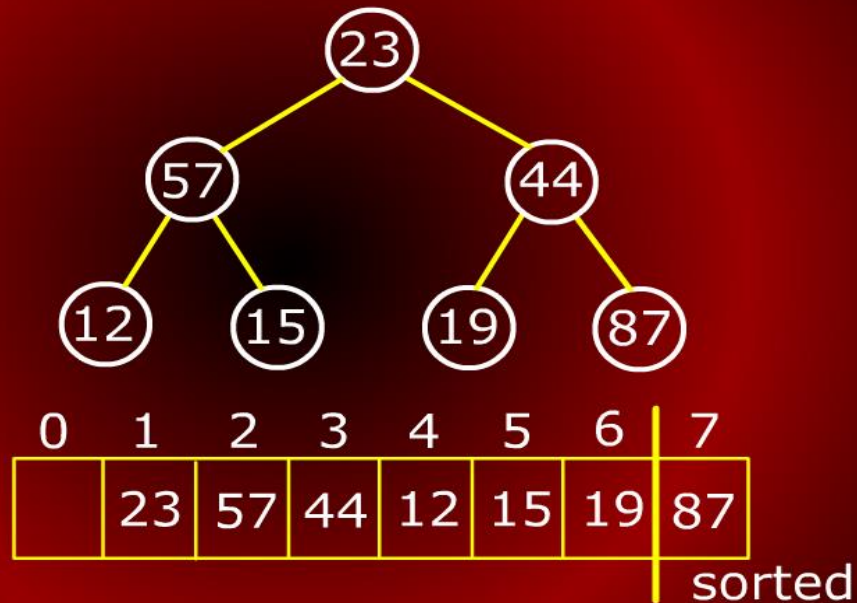
## *Heapsort Trace*





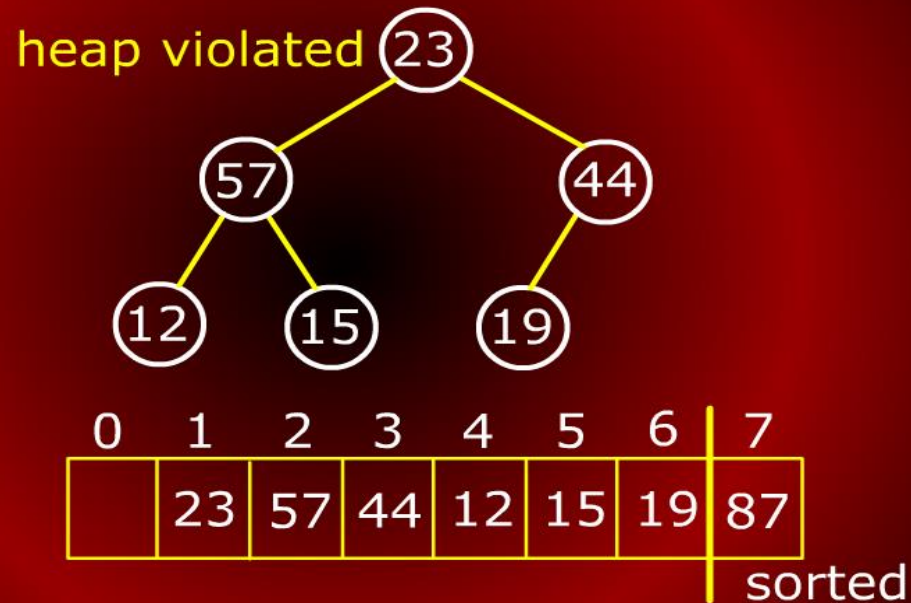
# Heap

## *Heapsort Trace*



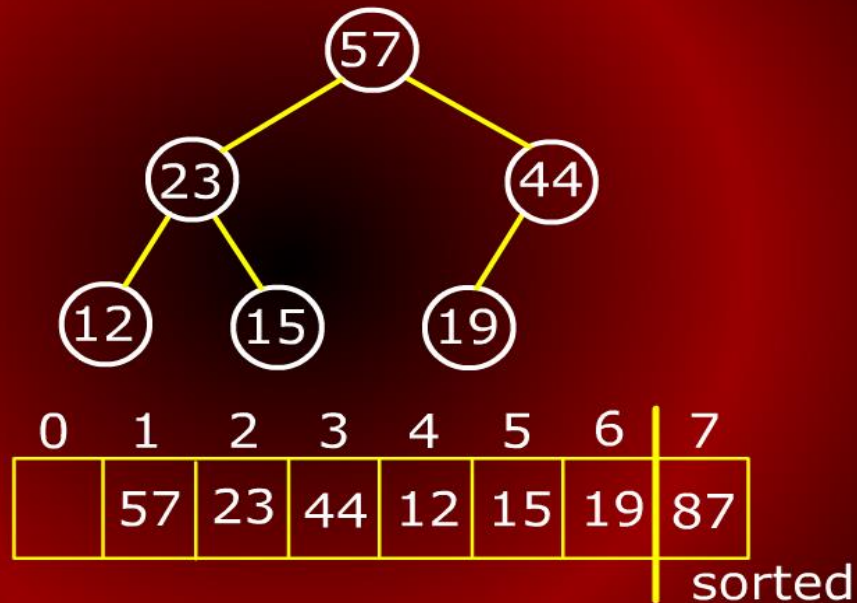
# Heap

## *Heapsort Trace*



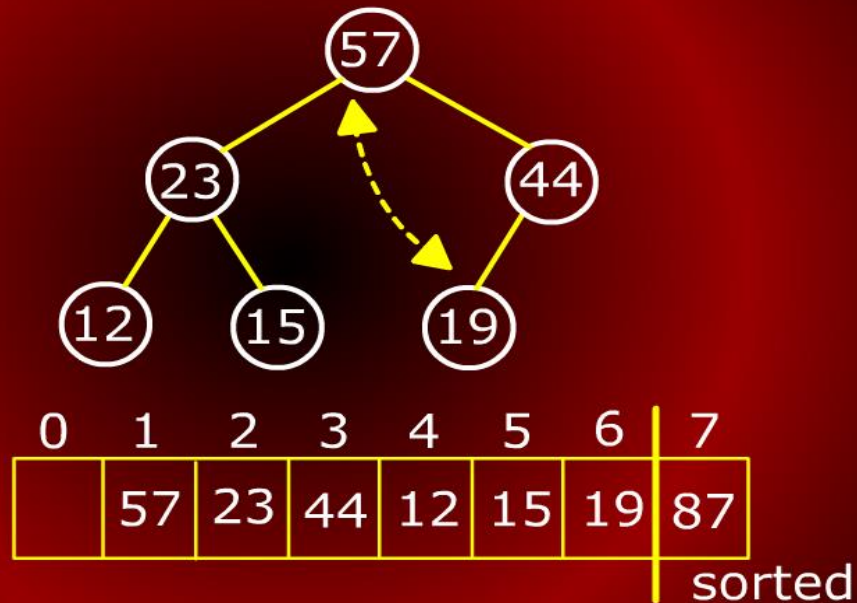
# Heap

## *Heapsort Trace*



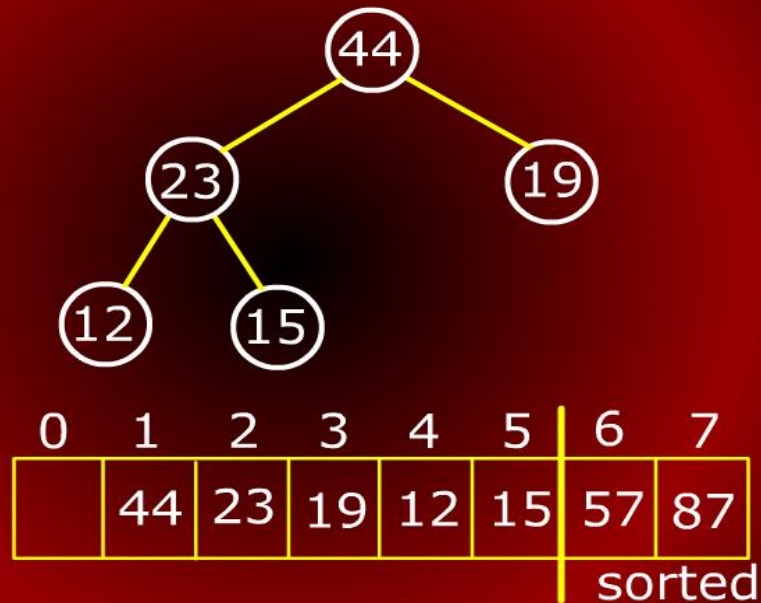
# Heap

## *Heapsort Trace*



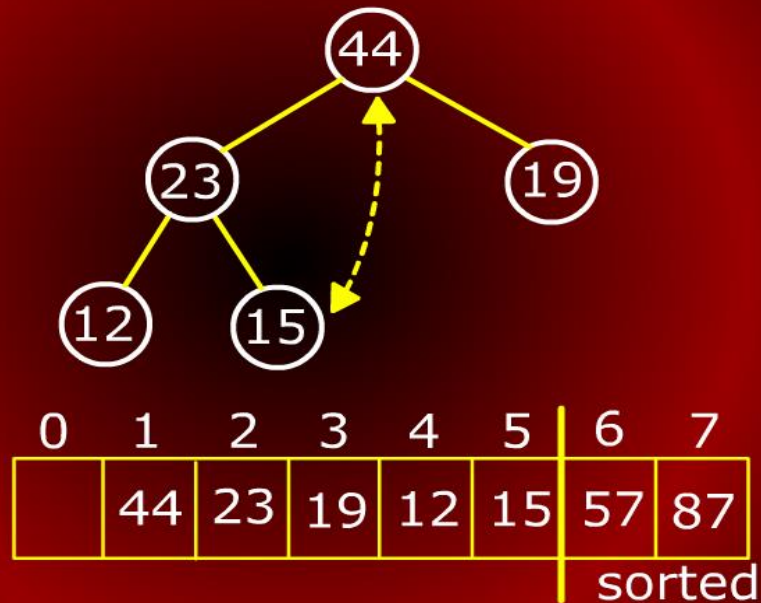
# Heap

## *Heapsort Trace*



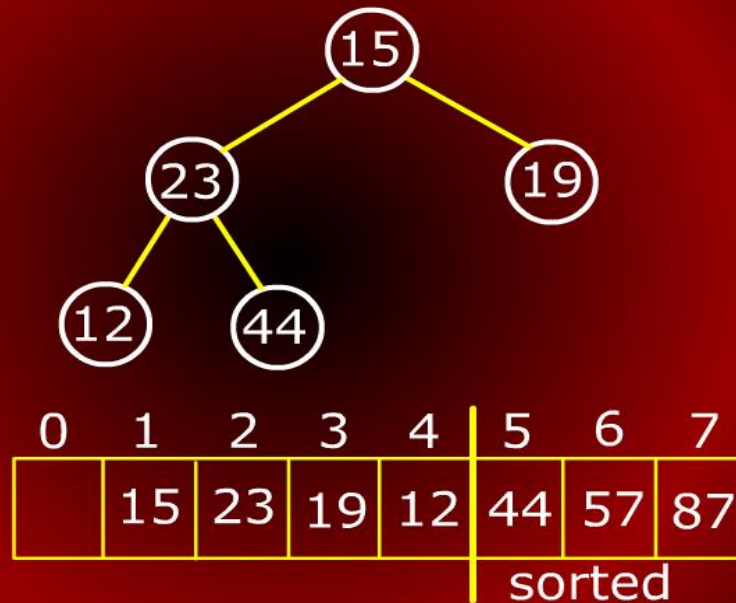
# Heap

## *Heapsort Trace*



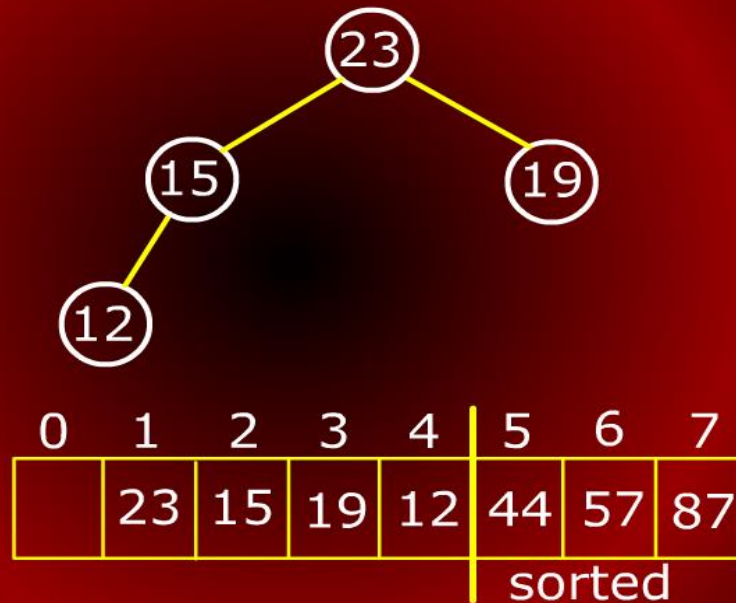
# Heap

## *Heapsort Trace*



# Heap

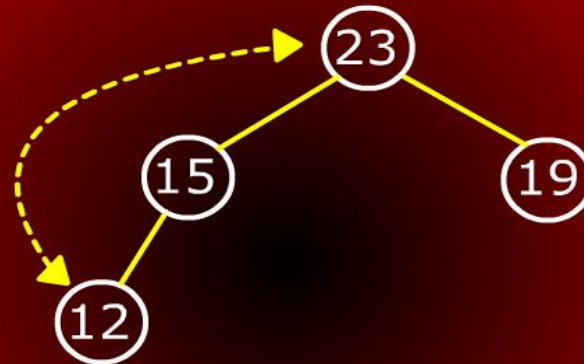
## *Heapsort Trace*





# Heap

## *Heapsort Trace*



0	1	2	3	4	5	6	7
	23	15	19	12	44	57	87
					sorted		

# Heap

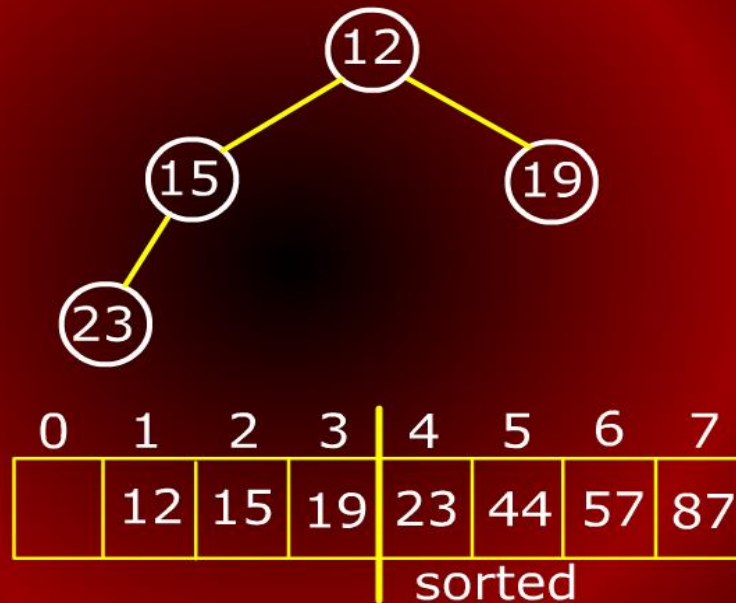
## *Heapsort Trace*



0	1	2	3	4	5	6	7
	19	15	12	23	44	57	87
				sorted			

# Heap

## *Heapsort Trace*



# Heap

## *Heapsort Trace*



0	1	2	3	4	5	6	7
	19	15	12	23	44	57	87
				sorted			

# Heap

## *Heapsort Trace*



0	1	2	3	4	5	6	7
	15	12	19	23	44	57	87

sorted

# Heap

## *Heapsort Trace*

0	1	2	3	4	5	6	7
	12	15	19	23	44	57	87
	sorted						