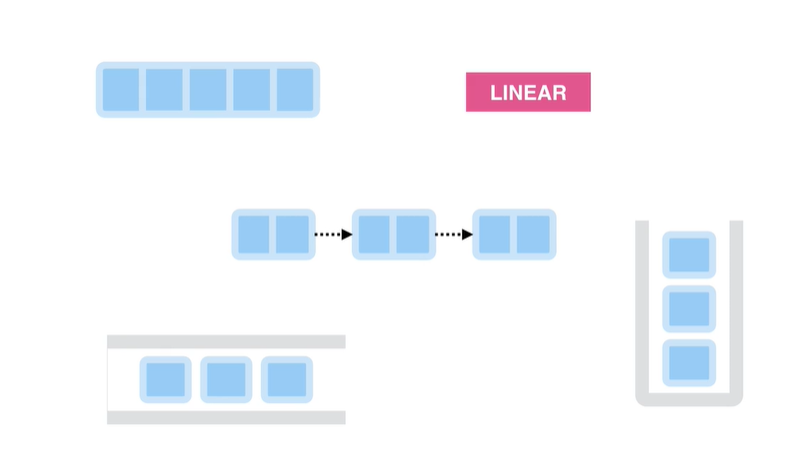
**Tree**

**V-1-Introduction:-**

So far we learned linear data structure.



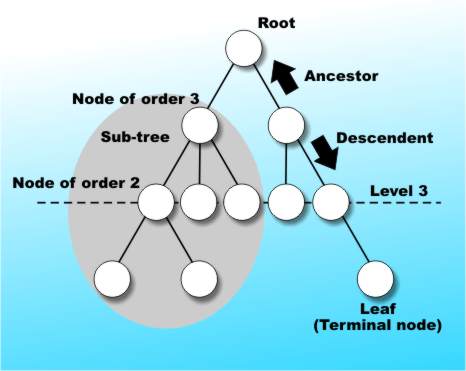
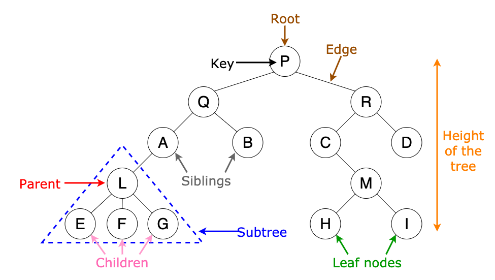
Stack

Queue

Linked List

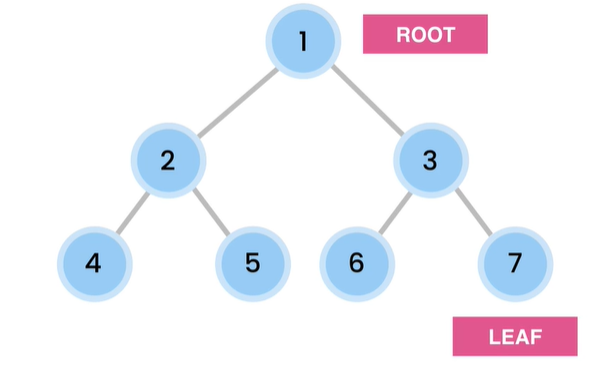
Array

Tree is one of the Non-Linear Data Structure.



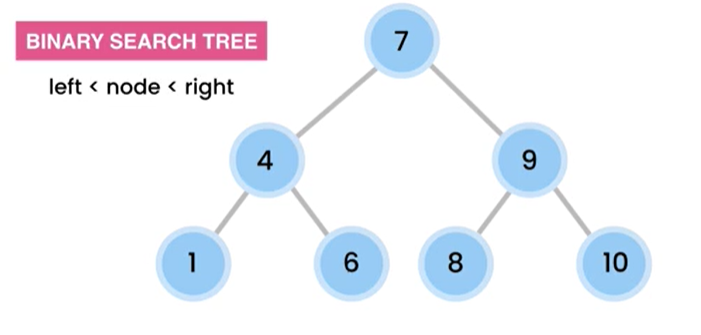
**V-2-What are Trees:-**

Tree is a Data Structure where we store values in a Hierarchy. The picture below is a binary tree. Cause each parent has at most two children.



In chrome it store pass in tree Structure

***Now BST(Binary Search Tree):-***



Any Node is greater

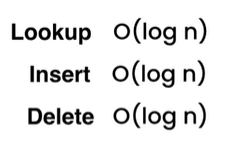
than left subtree and

Less than then the

Right subtree.[4<7<9]

7 is greater than 1,4,6

7 is less than 8,9,10



Tree structure operation always make

It half to work any operation. That’s

Why it is faster than array,LinkedList.

**V\_3\_E\_1)**

Binary Search Trees

Add the following numbers to a binary search tree. Remember, in a binary search tree:

left < parent

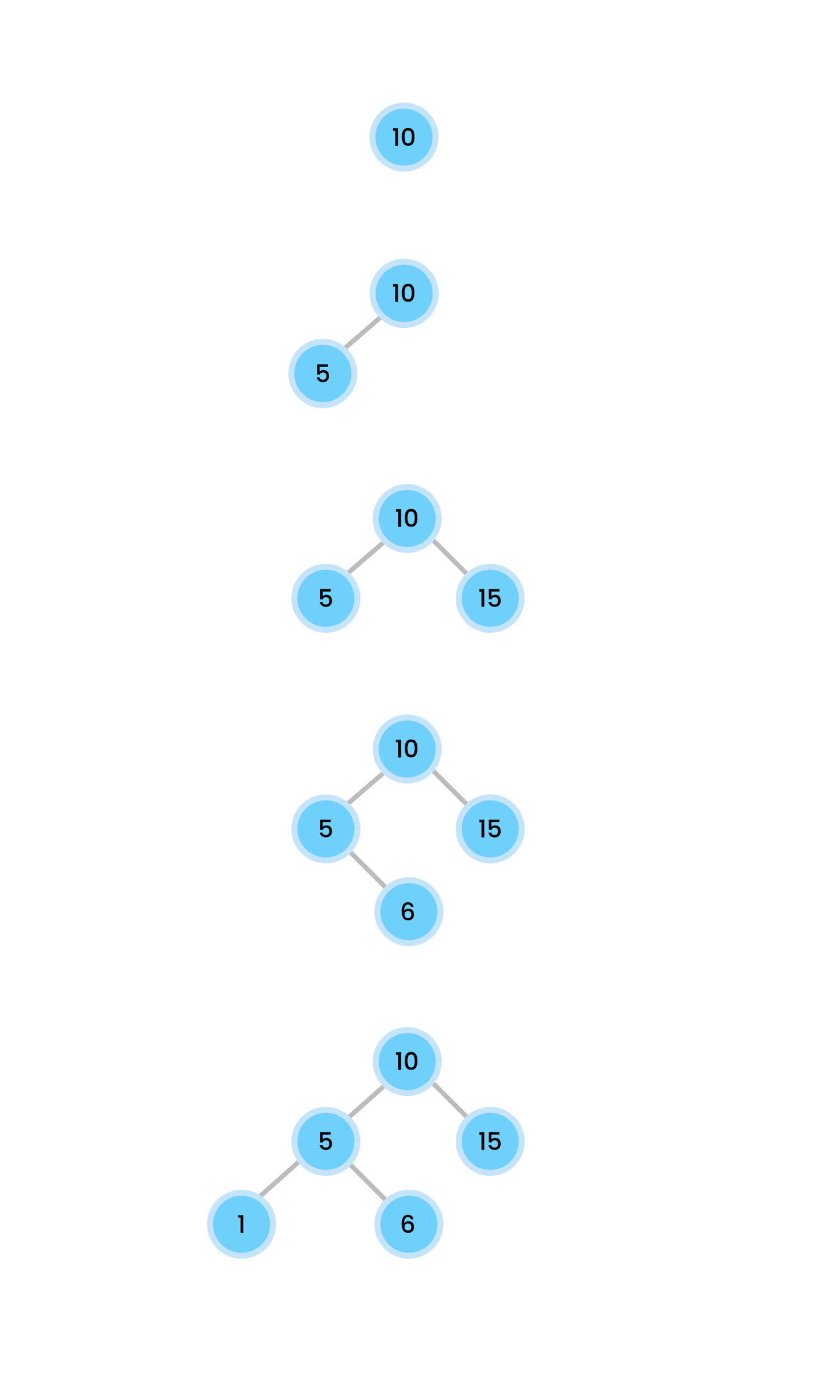
right > parent

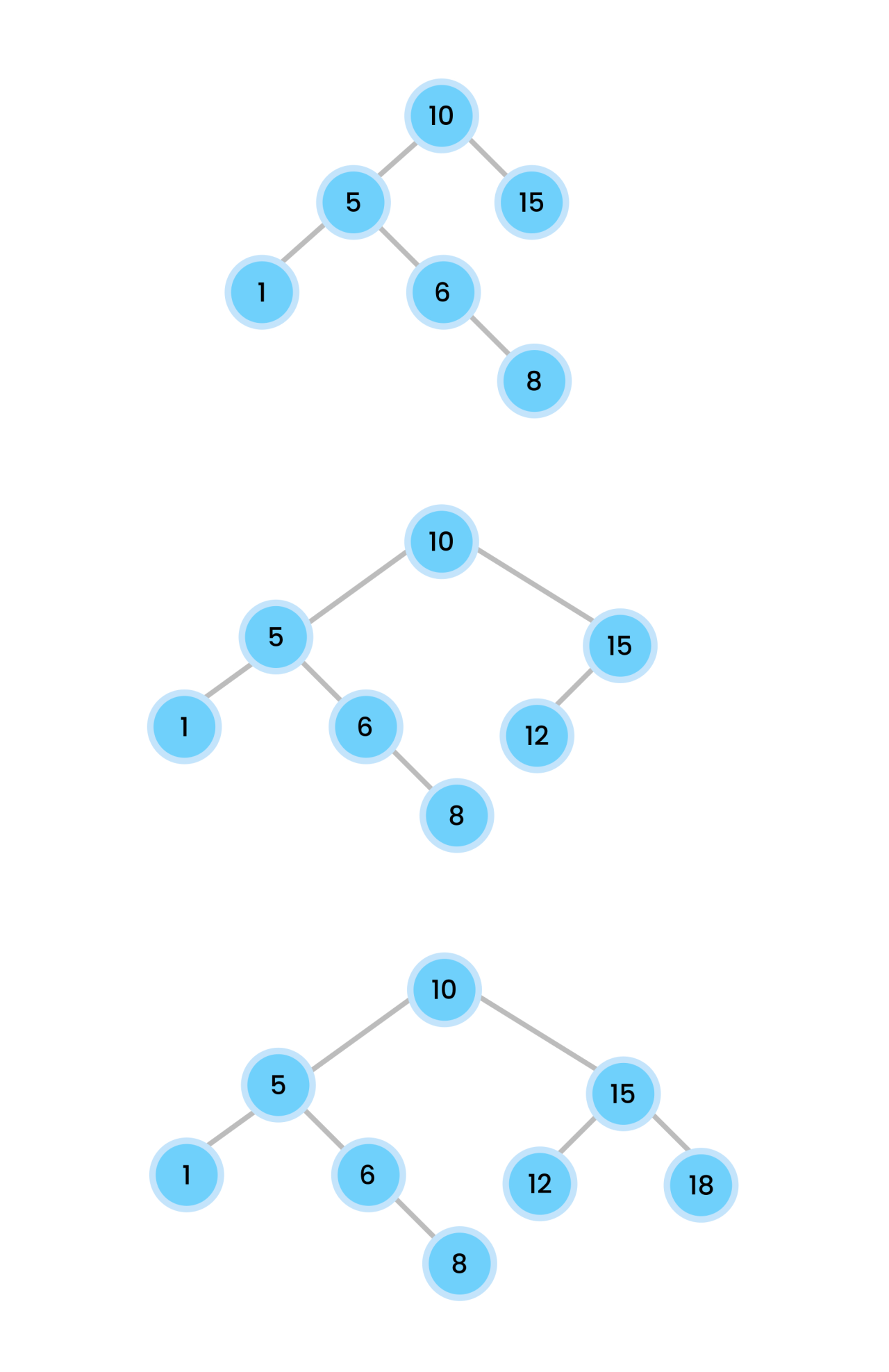
[10, 5, 15, 6, 1, 8, 12, 18, 17]

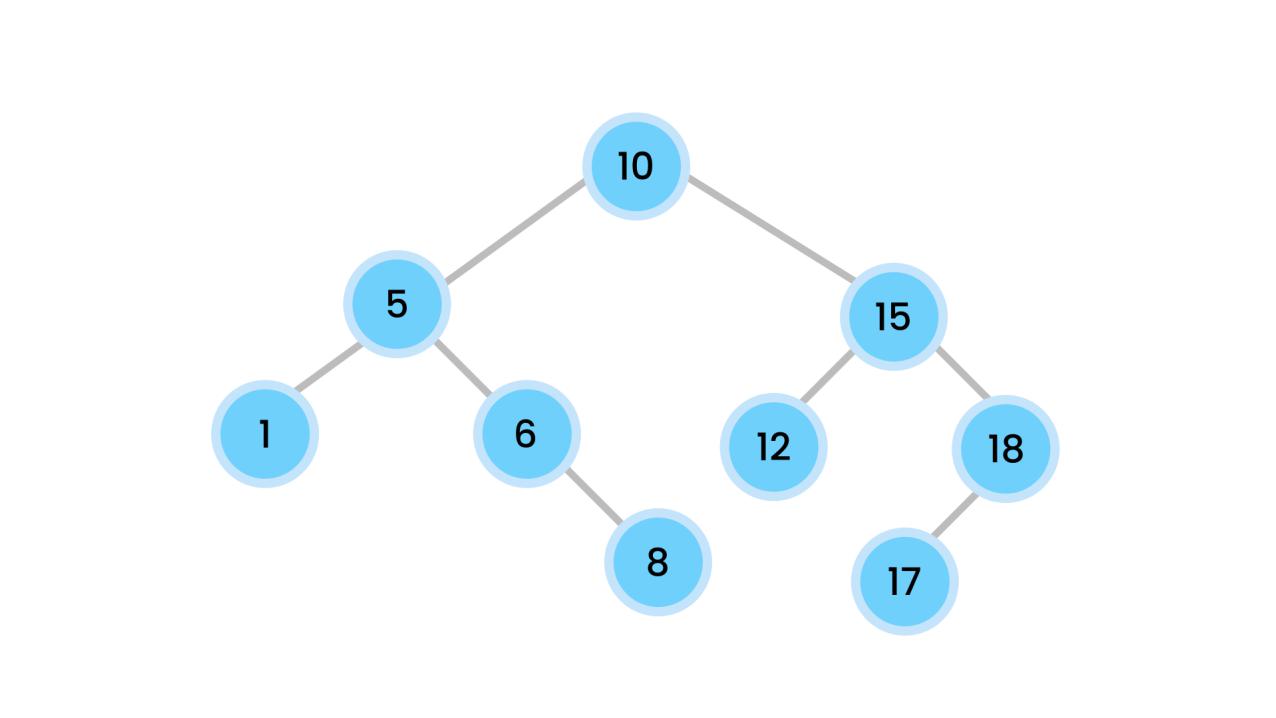
**Solution:**

You can also use the following tool to see how a binary search tree evolves as you add new items:

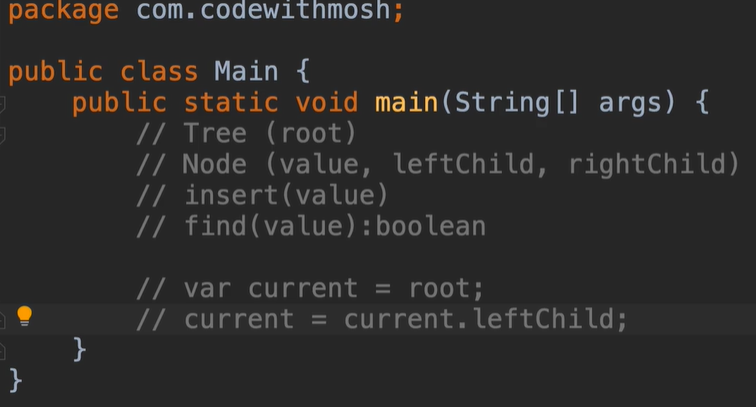
<https://visualgo.net/en/bst>







**V-4-Build A TREE:-**

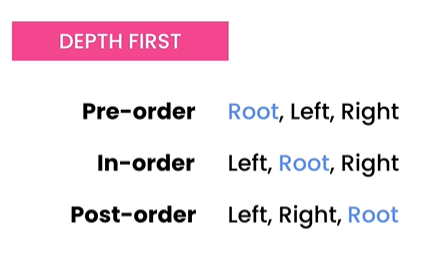


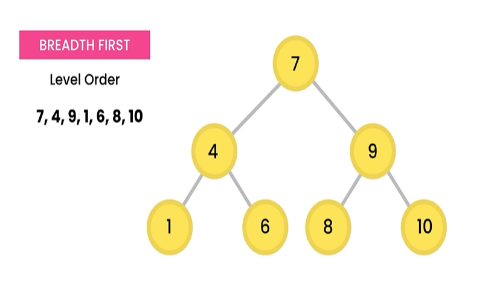
**V-5-Insert item in a tree:-**

public void insert(int value){  
 var node = new Node(value);//creating the node with given value  
 if(root==null){//if root isn't defined yet then make the node as root  
 root = node;  
 return;//we need to immediately return  
 }  
 //if root node is defined already then  
 var current = root;  
 while(true){//insert nodes by left<root<right (by comparing values)  
 if(value< current.value){  
 if(current.leftchild==null){//if leftchild is null then insert(set) leftchild here(current is the parent of the leftchild)  
 current.leftchild=node;  
 break;  
 }  
 current = current.leftchild;  
 }  
 else{  
 if(current.rightchild==null){//same logic as rightchild  
 current.rightchild=node;  
 break;  
 }  
 current = current.rightchild;  
 }  
 }  
  
}

**V-6-Find item in a tree:-**

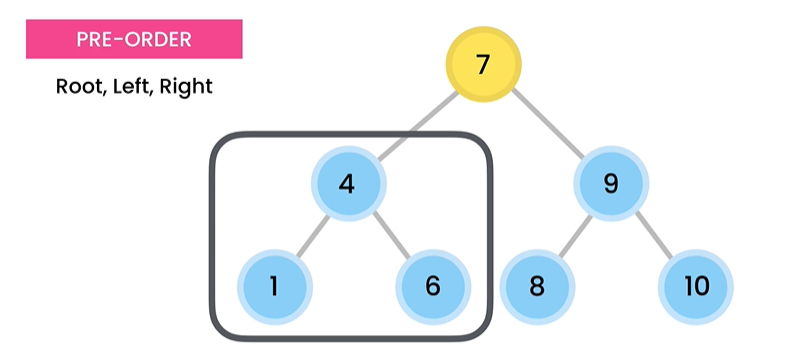
public boolean find(int value){  
 var current = root;  
 while(current!=null){  
 if(value<current.value)  
 current=current.leftchild;  
 else if (value> current.value)  
 current=current.rightchild;  
 else  
 return true;  
 }  
 return false;  
}

**V-8-Traversing Trees:-**

We can traverse a tree in two ways. 1. BFS(Level Order Traversal) 2. DFS

**LEVEL**

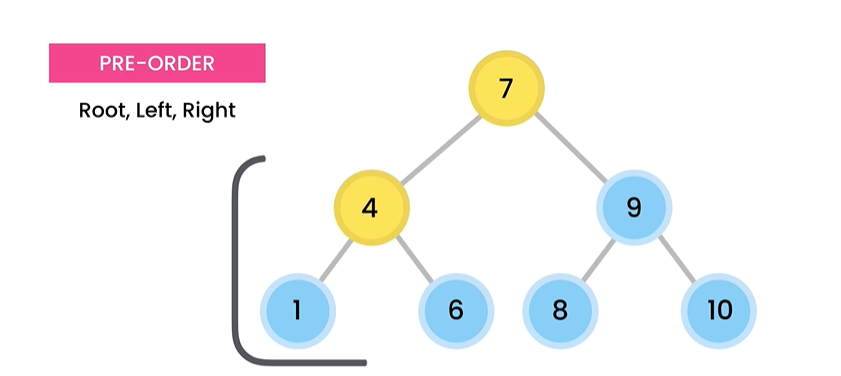
*Pre-Order-Traversl: ( Step by Step)*



left subtree.

Visit 7, cause this is root, then

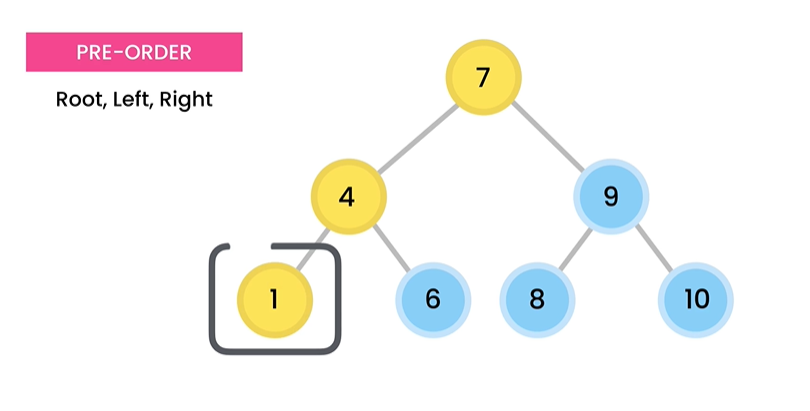
Go to left subtree.



Visit 4, cause this is root of

1 & 6 , then

Go to left subtree.



Visit 1, cause this is the

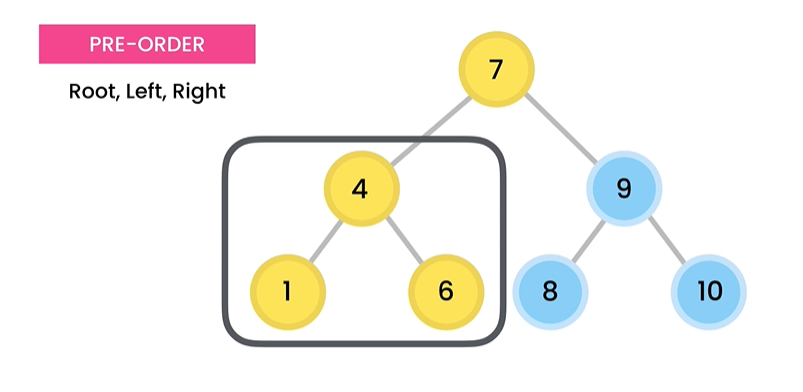
Left subtree of node 4.

but after that we

have to come back to node 4

cause there is no more

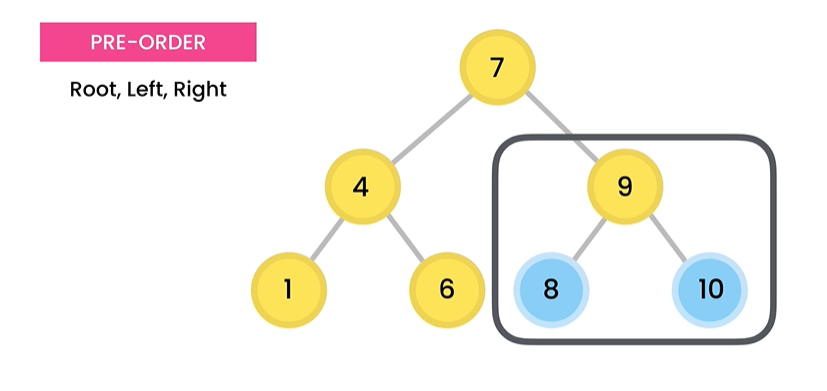
left subtree for node 1.



Visit 6, cause *root(4)* of node 6

and the left subtree of root

node(4) is visited already.

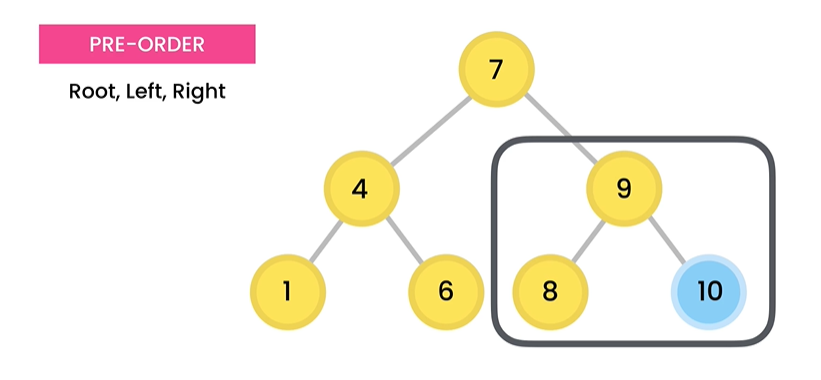


Visit 9, cause

root node (7) and its left

subtree is already visited. Now

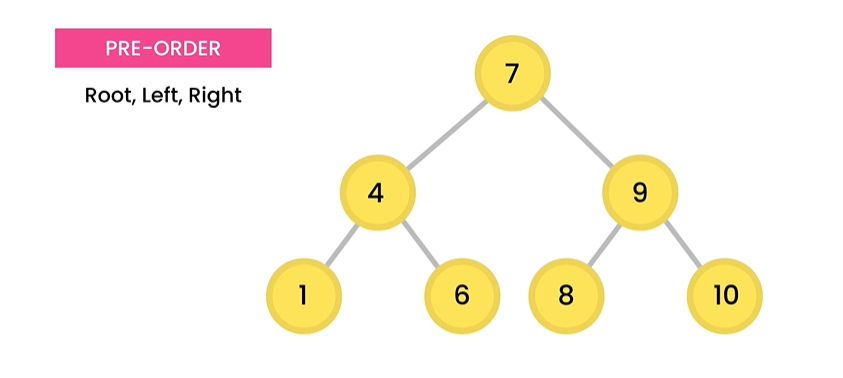
it is time for right subtree.



Visit 8, cause

this is the left subtree of root

node (9) .



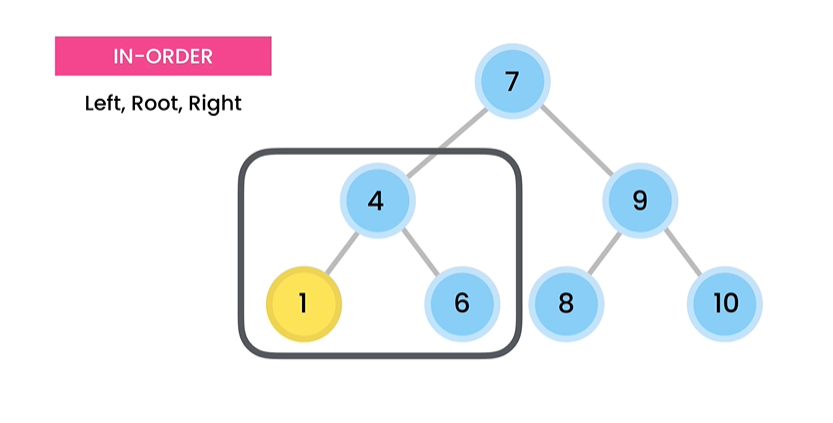
Visit 10, cause

Root node 9 and its left subtree

is already visited.

We are done for pre-order traversal . The order for pre order traversal is **7 -> 4 -> 1 -> 6 -> 9 -> 8 -> 10.**

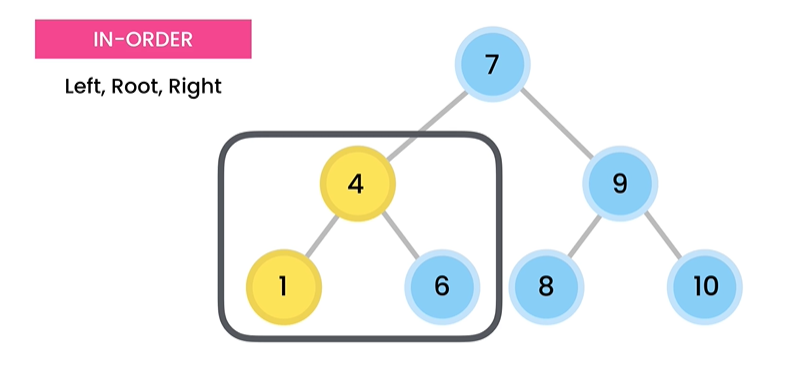
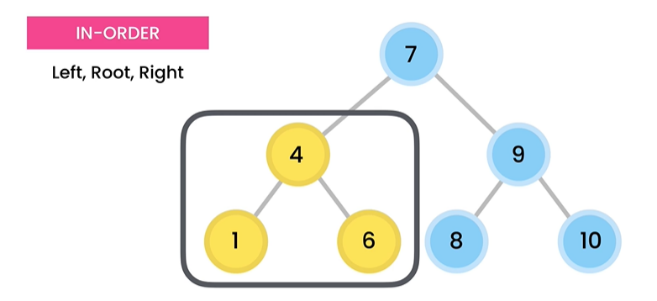
*In-Order-Traversal:- (Step By Step)*



Visit 1, cause

this is the leftmost node possible

for the whole tree.



Visit 4, cause

this is the root

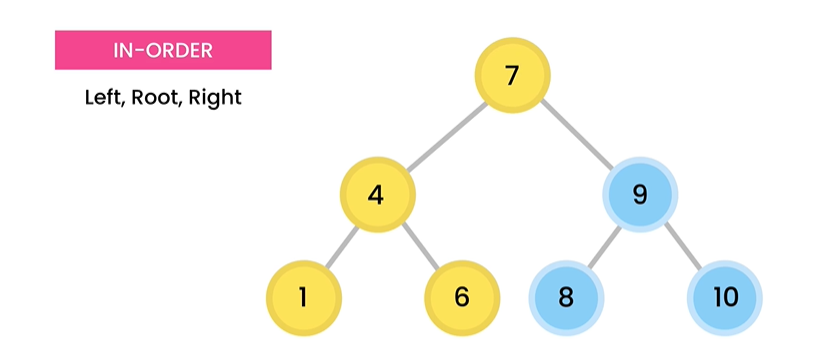
of the leftmost node 1.

for the whole tree.

Visit 6, cause

left node 1 and root node 4 is

visited already.



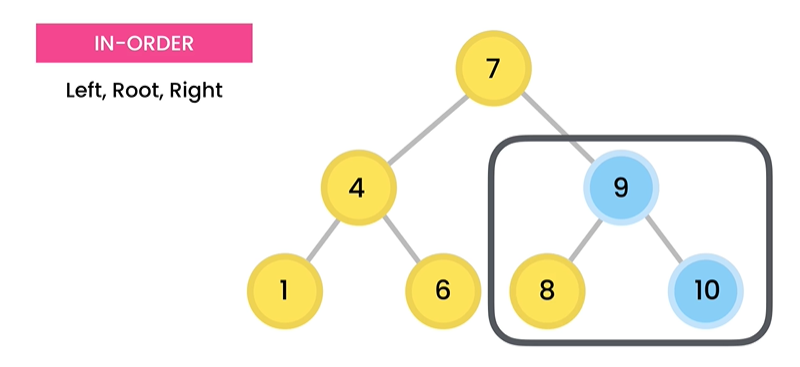
Visit 7, cause

left subtree for root node 7

is already visited.

left subtree for

root node 7



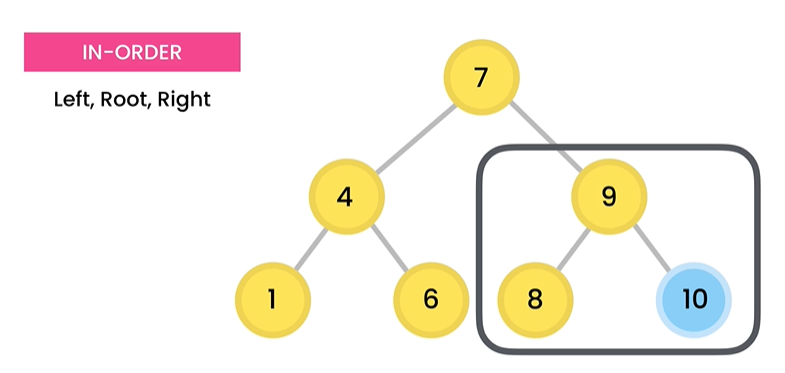
Visit 8, cause

Right subtree of root node 7

has another root 9 , where

8 is at the leftchild of root node 9.

HAS

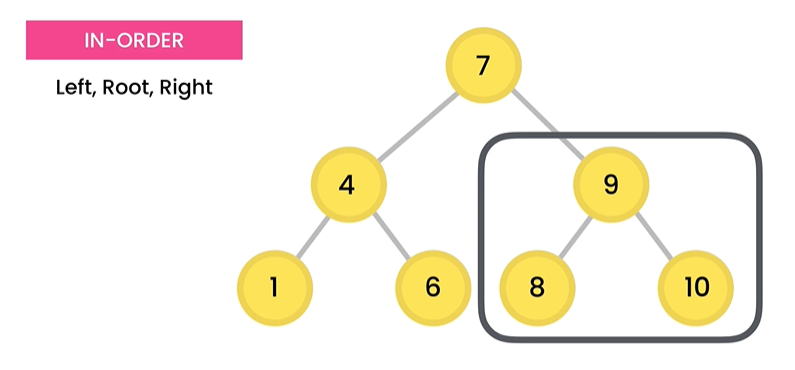


Visit 9, cause

Left node of root node 9 is

Visited already.

HAS



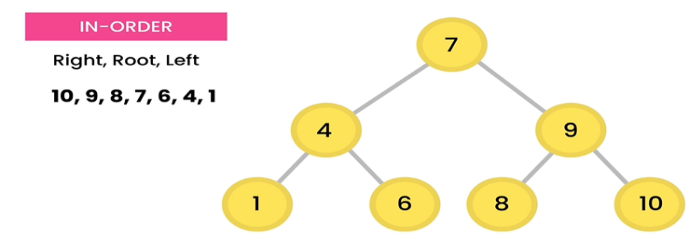
Visit 10, cause

Left node and root node 9 is

Visited already.

HAS

We are done for in-order traversal . The order for in order traversal is **1 -> 4 -> 6 -> 7 -> 8 -> 9 -> 10.**

****Magic – sorting - (We always get the numbers in ascending order for in-order traversal)**

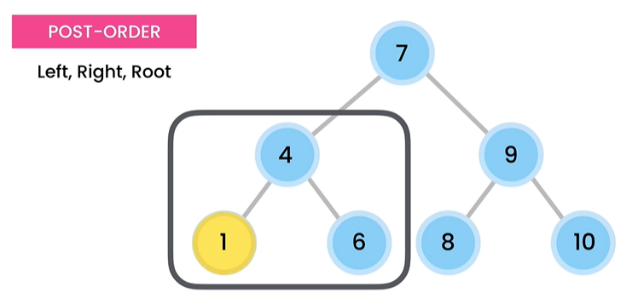
If we swap the order of visit to

Right>Root>Left

Then we got the nodes

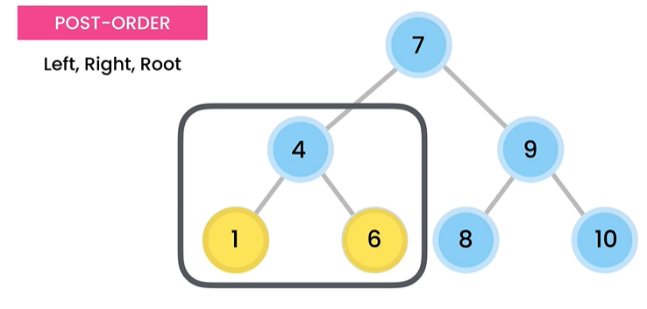
In descending order (Sorted)

HAS

*Post-Order-Traversal:- (Step By Step)*

visit 1, cause it is the leftmost node.

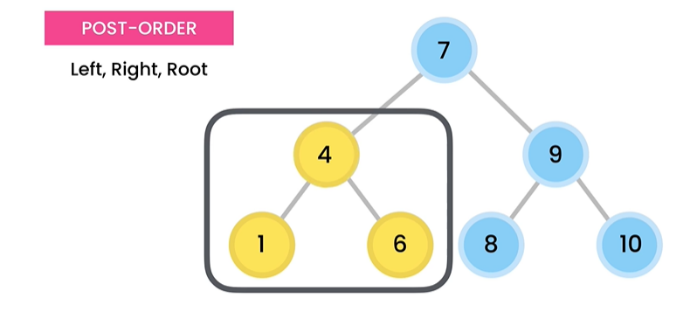
HAS



visit 6, cause we visited the left node

of root node 4.

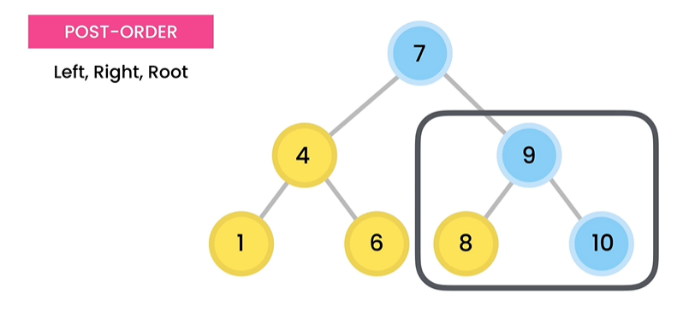
HAS



visit 4, cause we visited the left node

& right node of root node 4.

HAS



left subtree

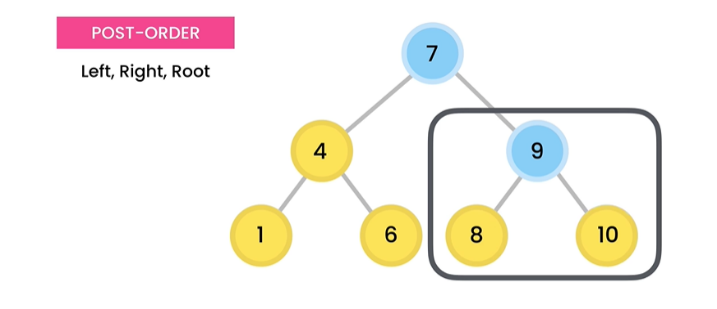
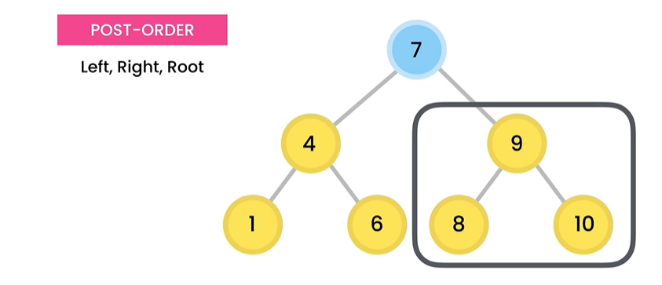
of root node 7.

HAS

visit 8, cause we visited the left subtree

of root node 7.

HAS



visit 10, cause we visited the left node

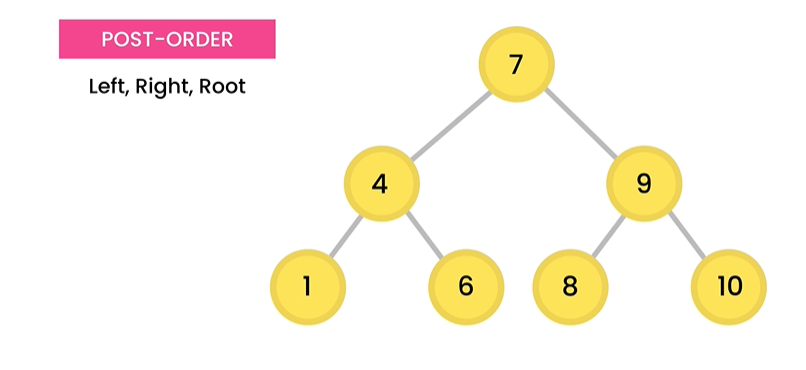
of root node 9.

HAS

visit 9, cause we visited the left node

& right node of root node 9.

HAS

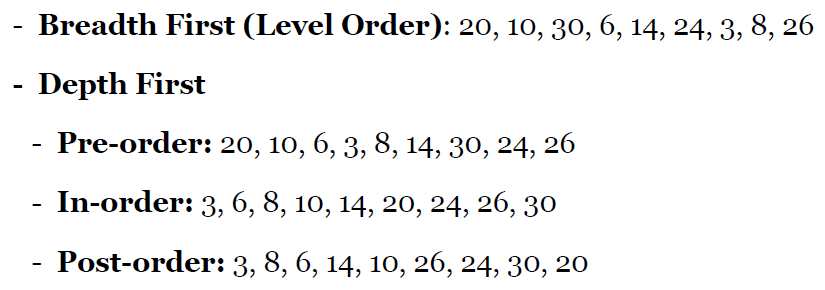
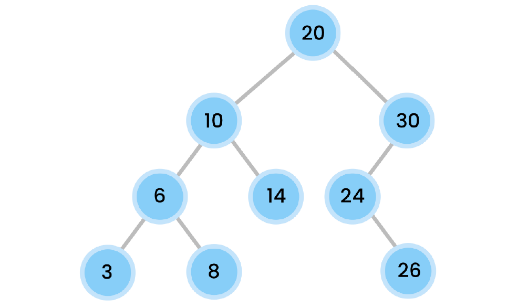


visit 7, cause we visited the both

the left & right subtree of root node 7.

HAS

We are done for post-order traversal. The order for in order traversal is **1 -> 6 -> 4 -> 8 -> 10 -> 9 -> 7.**

**Exercise(BFS & DFS):**

**V-9-Recursion:-**

public static int factorial(int n){  
 // 4! = 4 x 3 x 2 x 1  
 // 3! = 3 x 2 x 1  
 var factorial = 1;  
 for (int i=n ; i>1 ;i--)//using traditional for loop  
 factorial\*=i;  
 return factorial;  
}  
public static int factorialUsingRecursion(int n){  
 // 4! = 4 x 3!  
 // 3! = 3 x 2 x 1  
 // 2! = 2 x 1  
 // n! = n x (n-1)!  
 // 0! = 1 [this will be the case where recursion for factorial should be stopped  
 if(n==0) return 1;//base case to stop recursion otherwise it will give stack overflow  
 return n\**factorialUsingRecursion*(n-1);  
}

**V-10-DFS-(Pre,In,Post):-**

public void traversPreOrder(){  
 traversPreOrder(root);  
}  
private void traversPreOrder(Node root){  
 if(root==null)  
 return;  
 System.*out*.println(root.value);  
 traversPreOrder(root.leftchild);  
 traversPreOrder(root.rightchild);  
}  
public void traversInOrder(){  
 traversInOrder(root);  
}  
private void traversInOrder(Node root){  
 if(root==null)  
 return;  
 traversInOrder(root.leftchild);  
 System.*out*.println(root.value);  
 traversInOrder(root.rightchild);  
}  
public void traversPostOrder(){  
 traversPostOrder(root);  
}  
private void traversPostOrder(Node root){  
 if(root==null)  
 return;  
 traversPostOrder(root.leftchild);  
 traversPostOrder(root.rightchild);  
 System.*out*.println(root.value);  
}

**V-11-Depth & Height of Trees:-**

VisiT Finding Depth of any nodes in a tree:- (2 Ways):-

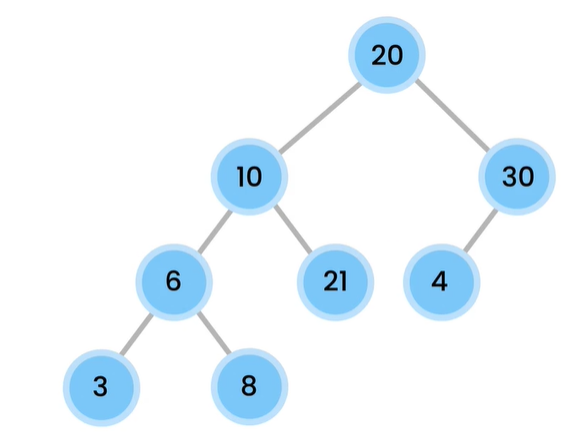
1. Count nodes (from root to that node-1). Ex-

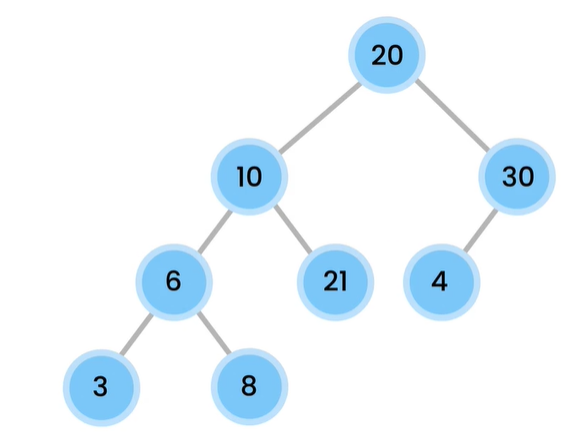
Depth(3) = (20->10->6->3)-1= 4-1 = 3.

2. Count the edges from root to that node. Ex-

Depth(3) = (20->10->6->3) = 3. [edges (20->10,10->6,6->3).

1

****

****

VisiT Finding Height of any nodes in a tree:-

Xzckj Height of the root node is also called the height of the tree

The picture(Tree) beside has a height of 3.As, Height(root) = 3.

1. Count edges from leaf node to that particular node but

Strictly followed by the longest path. Ex-

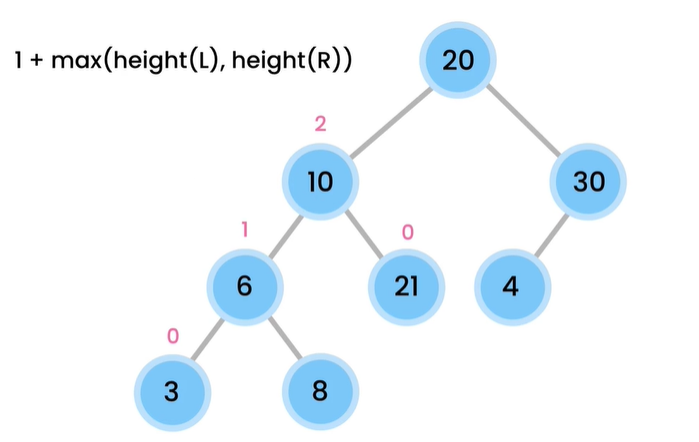
Height(3) = Height(8)= Height(21) = Height(4) = 0 , cause they are leaf nodes.

Height(10) = 2. Because in that particular subtree [marked as green circle] has 3 leaf nodes for node 10 we can choose. They are 3,8,21[marked as red circle.] But we have count the edges from any of these leaf nodes[3,8,21] so that the path can be maximum. This is like post-order traversal. As we count the root node last.

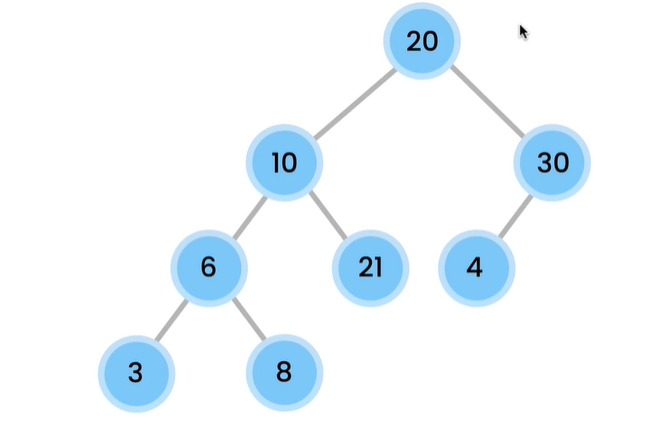
1

Height Formula

For Any Node.



public int height(){  
 return height(root);  
}  
private int height(Node root){  
 if(root==null) return -1;  
 if(isLeaf(root))  
 return 0;  
return 1+Math.*max*(height(root.leftchild),height(root.rightchild));  
}

**V-12-Minimum Value in a Binary Tree(also maximum by me):-**

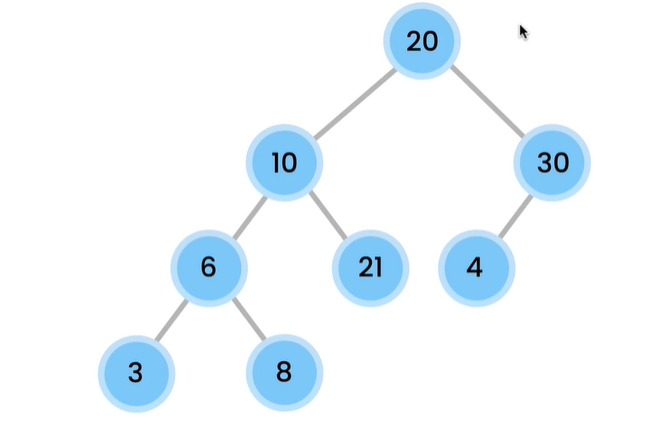


FIG:**A** FIG:**B**

Binary Tree but not BST(Binary Search Tree) BST(Binary Search Tree) + Binary Tree.

Fig A is not BST cause in BST (left Subtree<root<right Subtree) we have 21 in left subtree which is greater than the root(20) & also we have 4 in right subtree which is less than the root(20). But if we see Fig B it followed BST rule left Subtree<root<right Subtree.

private boolean isLeaf(Node node){  
 return node.leftchild == null && node.rightchild==null;  
}  
public int min2(){  
 //O(log n) cause it will works with half of the elements  
 //if the tree is BST , then only we can apply this method to find minimum  
 //cause in leftmost node carry the minimum value in a BST  
 root\_null\_throw\_exception();  
 var current = root;  
 var last=current;  
 while(current!=null){  
 last = current;  
 current = current.leftchild;  
 }  
 return last.value;  
}  
public int max2(){  
 //O(log n) cause it will works with half of the elements  
 //if the tree is BST , then only we can apply this method to find maximum  
 //cause in rightmost node carry the maximum value in a BST  
 root\_null\_throw\_exception();  
 var current = root;  
 var last=current;  
 while(current!=null){  
 last = current;  
 current = current.rightchild;  
 }  
 return last.value;  
}  
public int min(){  
 root\_null\_throw\_exception();  
 return min(root);  
}

private int min(Node root){ // 0(n) cause it is like post-order traversal, we have to iterate through every nodes  
 //this works for Binary Tree+ BST(Binary Search Tree)  
 if(isLeaf(root)) return root.value;  
 var left = min(root.leftchild);  
 var right = min(root.rightchild);  
 return Math.*min*(Math.*min*(left,right), root.value);  
}  
public int max(){  
 root\_null\_throw\_exception();  
 return max(root);  
}  
private int max(Node root){//this works for Binary Tree+ BST(Binary Search Tree)  
 if(isLeaf(root)) return root.value;  
 var left = max(root.leftchild);  
 var right = max(root.rightchild);  
 return Math.*max*(Math.*max*(left,right), root.value);  
}  
private void root\_null\_throw\_exception(){  
 if(root==null)  
 throw new IllegalStateException();  
}

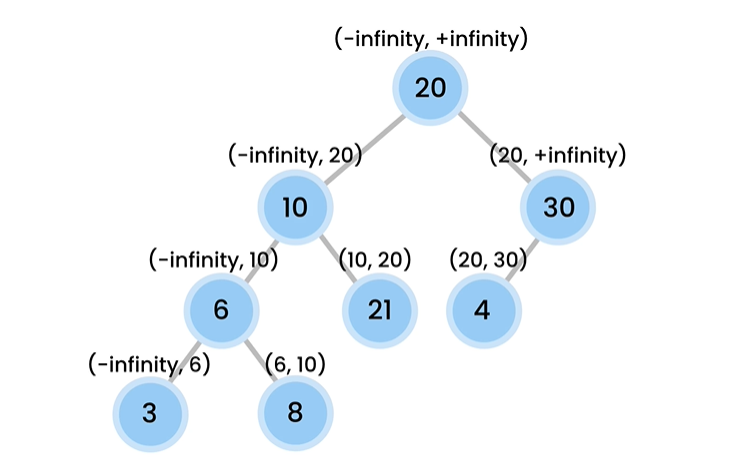
**V-(13+14)-Equality Checking:-**

private void root\_null\_throw\_exception(){  
 if(root==null)  
 throw new IllegalStateException();  
}  
public boolean equals(Tree other){  
 if(other == null) return false;  
return equals(root,other.root);  
}  
private boolean equals(Node first,Node second){  
 //1st case, if both trees are null then they are equal  
if(first== null && second==null) return true;  
//2nd case, if both nodes are not null , then check root and left+right subtrees  
if(first!=null && second!=null)  
 return first.value == second.value &&  
 equals(first.leftchild,second.leftchild) &&  
 equals(first.rightchild, second.rightchild);  
return false;

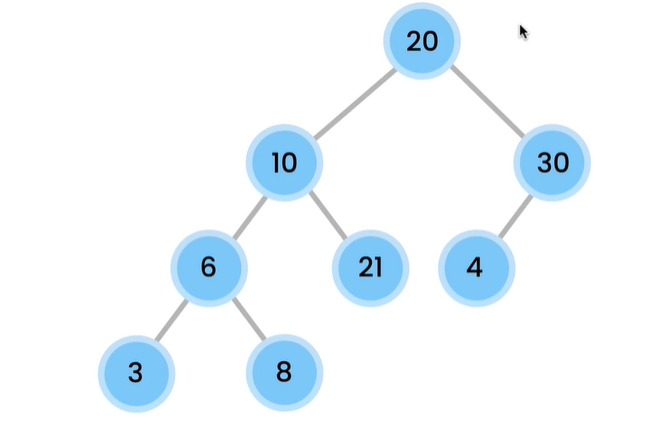
//if one of the tree root is null but other is not(3rd case)  
}

**V-(15+16)-Validity Checking of BST-(Pre-Order Traversal Exercise):-**

**The exercise is pre-order traversal. Because we have to define root first, then its left child, after that it’s right child. The green marked below is the valid range for every node value in BST.**



public boolean istreeBST(){  
 return istreeBST(root,Integer.*MIN\_VALUE*,Integer.*MAX\_VALUE*);  
}  
private boolean istreeBST(Node root,int min,int max){  
 //if tree has nothing then nothing to check  
 if(root==null)  
 return true;  
 //We know in BST left(min) <root< right(max) , if break this then false  
 if(root.value>max || root.value<min)  
 return false;  
 //if above two of the cases aren't then we have to check for every node and left and right child value  
 //if we have a root of 10 , then its left child must be in range of -infinity to 9(root-1)  
 // & right child must be in range of root+1 to infinity, that mean for root 10  
 // right child must be at least 11.  
 return istreeBST(root.leftchild,min, root.value-1)  
 && istreeBST(root.rightchild, root.value+1,max);  
}

**V-(17+18)-Nodes at K distance from a root:-**

Distance = 0

Distance = 1

Distance = 3

Distance = 2

The Green Marked labeling is the distances calculated from root node. If distance = 0 , then print only the root node & that is 20. If distance = 1, then print 10,3. If distance = 2, then print 6,21,4. If distance is 3 , then print 3,8.

public ArrayList<Integer> getNodesAtDistance(int distance){  
 var list = new ArrayList<Integer>();  
 getNodesAtDistance(root,distance,list);  
 return list;  
}  
private void getNodesAtDistance(Node root,int distance,ArrayList<Integer> list){  
if(root==null) return;  
if(distance==0){  
 list.add(root.value);  
 return;  
}  
getNodesAtDistance(root.leftchild,distance-1,list);  
getNodesAtDistance(root.rightchild,distance-1,list);  
}

**V-(19)-Nodes at K distance from a root:-**

public void traverseLevelOrder(){  
 for (int i=0;i<=height();i++){  
 for(var item :getNodesAtDistance(i))  
 System.*out*.println(item);  
 }  
}

EXERCISE:-

1- Implement a method to calculate the size of a binary tree.

ANS:-

public int size() {  
 return size(root);  
}  
private int size(Node root) {  
 if (root == null)  
 return 0;  
  
 if (isLeaf(root))  
 return 1;  
  
 return 1 + size(root.leftchild) + size(root.rightchild);  
}

2- Implement a method to count the number of leaves in a binary tree.

Ans:-

public int countLeaves() {  
 return countLeaves(root);  
}  
  
private int countLeaves(Node root) {  
 if (root == null)  
 return 0;  
  
 if (isLeaf(root))  
 return 1;  
  
 return countLeaves(root.leftchild) + countLeaves(root.rightchild);  
}

3- Implement a method to return the maximum value in a binary search

tree using recursion.

Ans:-

public int max3() {  
 if (root == null)  
 throw new IllegalStateException();  
 return max\_3(root);  
}  
private int max\_3(Node root) {  
 if (root.rightchild == null)  
 return root.value;  
 return max(root.rightchild);  
}

4- Implement a method to check for the existence of a value in a binarytree using recursion. Compare this method with the find() method. The

find() method does the same job using iteration.

Ans:-

public boolean contains(int value) {  
 return contains(root, value);  
}  
private boolean contains(Node root, int value) {  
 if (root == null)  
 return false;  
  
 if (root.value == value)  
 return true;  
  
 return contains(root.leftchild, value) || contains(root.rightchild, value);  
}

5- Implement a method to check to see if two values are siblings in a

binary tree.

Ans:-

public boolean areSibling(int first, int second) {  
 return areSibling(root, first, second);  
}  
private boolean areSibling(Node root, int first, int second) {  
 if (root == null)  
 return false;  
  
 var areSibling = false;  
 if (root.leftchild != null && root.rightchild != null) {  
 areSibling = (root.leftchild.value == first && root.rightchild.value == second) ||  
 (root.rightchild.value == first && root.leftchild.value == second);  
 }  
  
 return areSibling ||  
 areSibling(root.leftchild, first, second) ||  
 areSibling(root.rightchild, first, second);  
}

6- Implement a method to return the ancestors of a value in a

List<Integer>.

[Helping to get Idea-{ <https://youtu.be/qjD-CmuC0MQ> }]

Ans:-

public List<Integer> getAncestors(int value) {  
 var list = new ArrayList<Integer>();  
 getAncestors(root, value, list);  
 return list;  
}  
private boolean getAncestors(Node root, int value, List<Integer> list) {  
 // We should traverse the tree until we find the target value. If  
 // find the target value, we return true without adding the current node  
 // to the list; otherwise, if we ask for ancestors of 5, 5 will be also  
 // added to the list.  
 if (root == null)  
 return false;  
  
 if (root.value == value)  
 return true;  
  
 // If we find the target value in the left or right sub-trees, that means  
 // the current node (root) is one of the ancestors. So we add it to the list.  
 if (getAncestors(root.leftchild, value, list) ||  
 getAncestors(root.rightchild, value, list)) {  
 list.add(root.value);  
 return true;  
 }  
  
 return false;  
}

Ex-7 . Is the tree balance or not?

Ans:-

public boolean isBalanced() {  
 return isBalanced(root);  
}  
  
private boolean isBalanced(Node root) {  
 if (root == null)  
 return true;  
  
 var balanceFactor = height(root.leftchild) - height(root.rightchild);  
  
 return Math.*abs*(balanceFactor) <= 1 &&  
 isBalanced(root.leftchild) &&  
 isBalanced(root.rightchild);  
}

Ex-8 . Is the tree perfect or not?

[hint :- If all parents had exactly two child in a tree then we called it a pefect binary Tree]

[Resource(for more):- <https://www.programiz.com/dsa/perfect-binary-tree> ]

Ans:-

public boolean isPerfect() {  
 return size() == (Math.*pow*(2, height() + 1) - 1);  
}

Summary:-

1. A tree can have at most two children.

2. A node without any children is called a leaf node.

3. The height of a leaf node is 0. As we go up in the tree from leaf node the height increases.[The height of a tree is the height of a root node.]

4. The depth of the root node is 0. As we go down from root node, the depth of nodes will be increased.[More accurately the edges from root node to that node]

5. In BST [left Subtree<root<Right Subtree]

