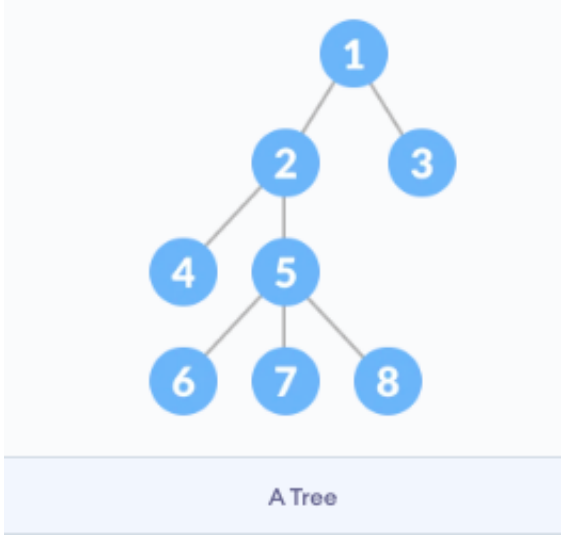
AGENDA

- Trees, Types of trees
- Binary Trees, Operations on Trees
- Tree representations and tree traversals
- Application of binary trees
- Binary search trees
- Operations and Traversals on BST – Applications
- Introduction of Height balanced trees, AVL trees

TREES

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.
- Stores data in the form of nodes and nodes are connected to each other using edges.
- The data in a tree are not stored in a sequential manner.
- Instead, they are arranged on multiple levels (hierarchical structure).

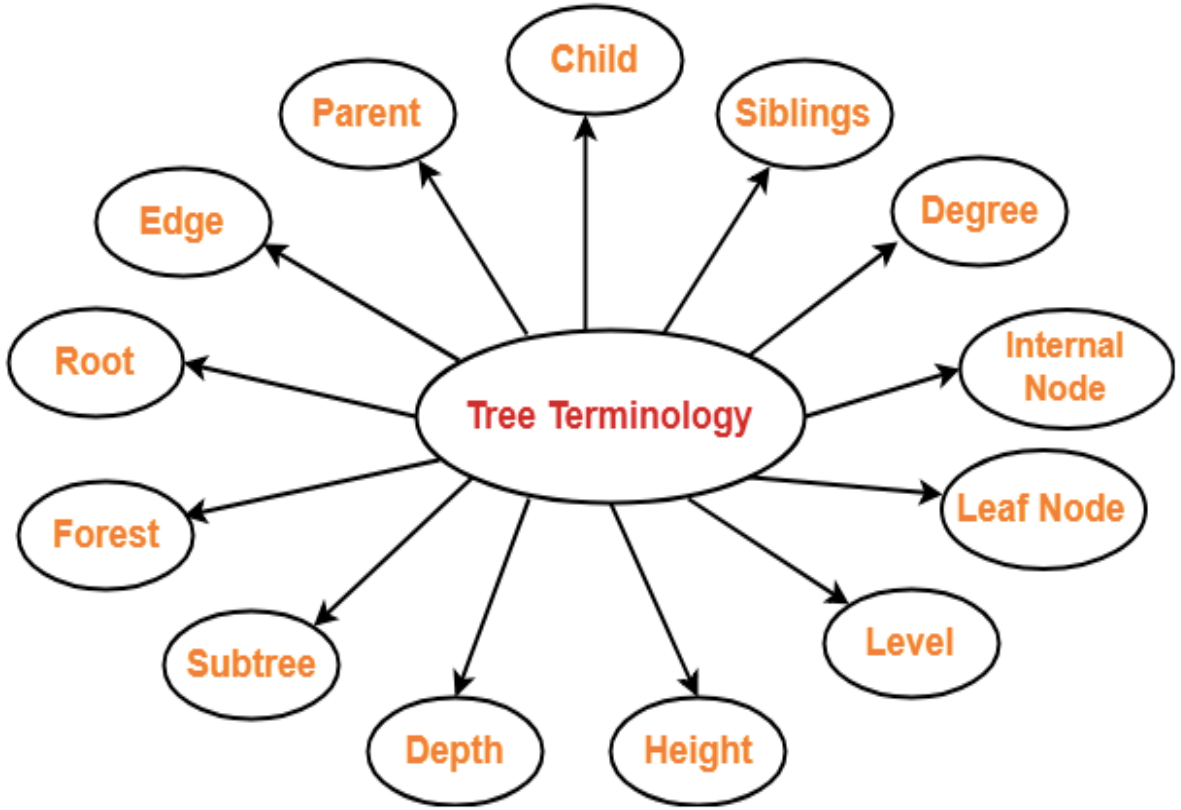


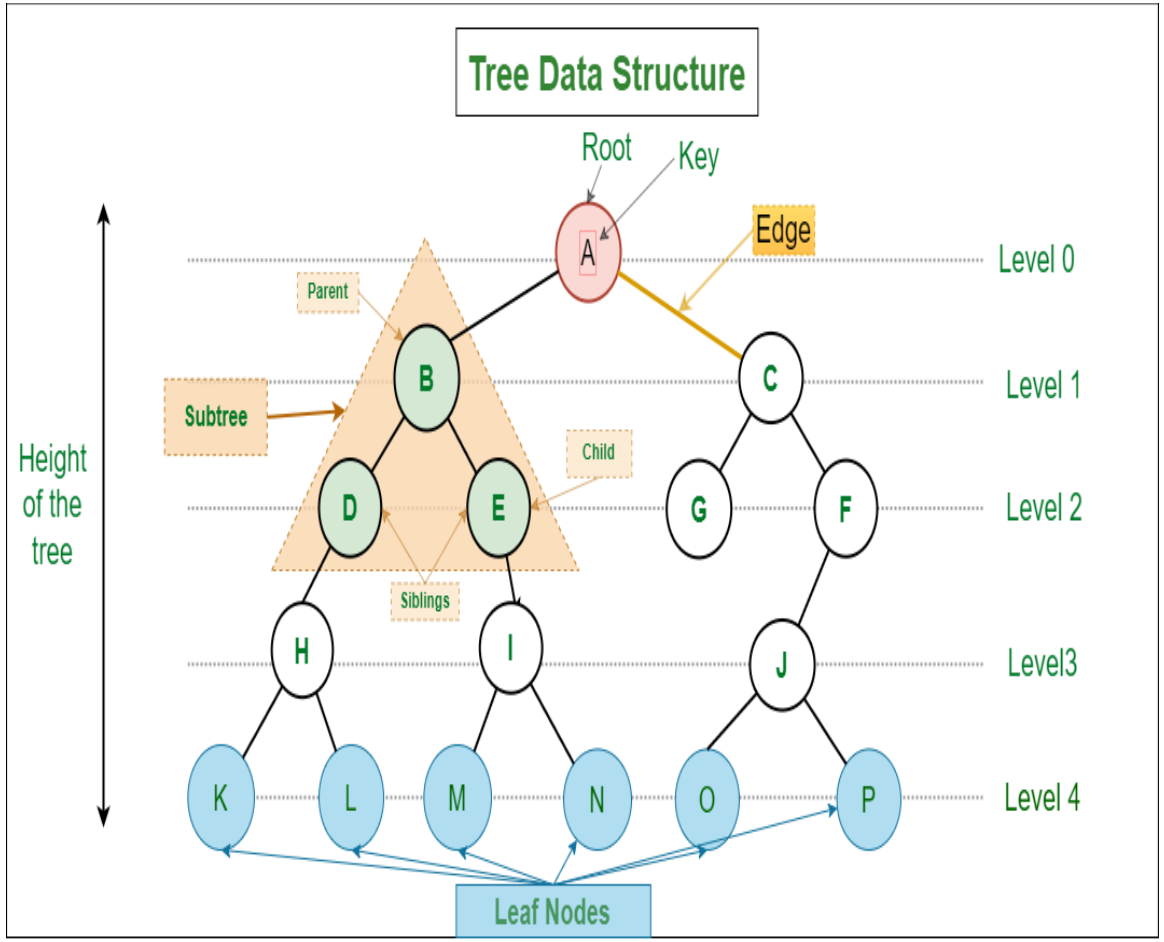
A Tree

Advantages of Tree Data Structure

- Tree offer **Efficient Searching** Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
- Trees provide a hierarchical representation of data, making it **easy to organize and navigate** large amounts of information.
- The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

TERMINOLOGIES





Root

- The first node from where the tree originates is called as a **root node**.
- node without parent (A)

Edge

The connecting link between any two nodes is called as an **edge**.

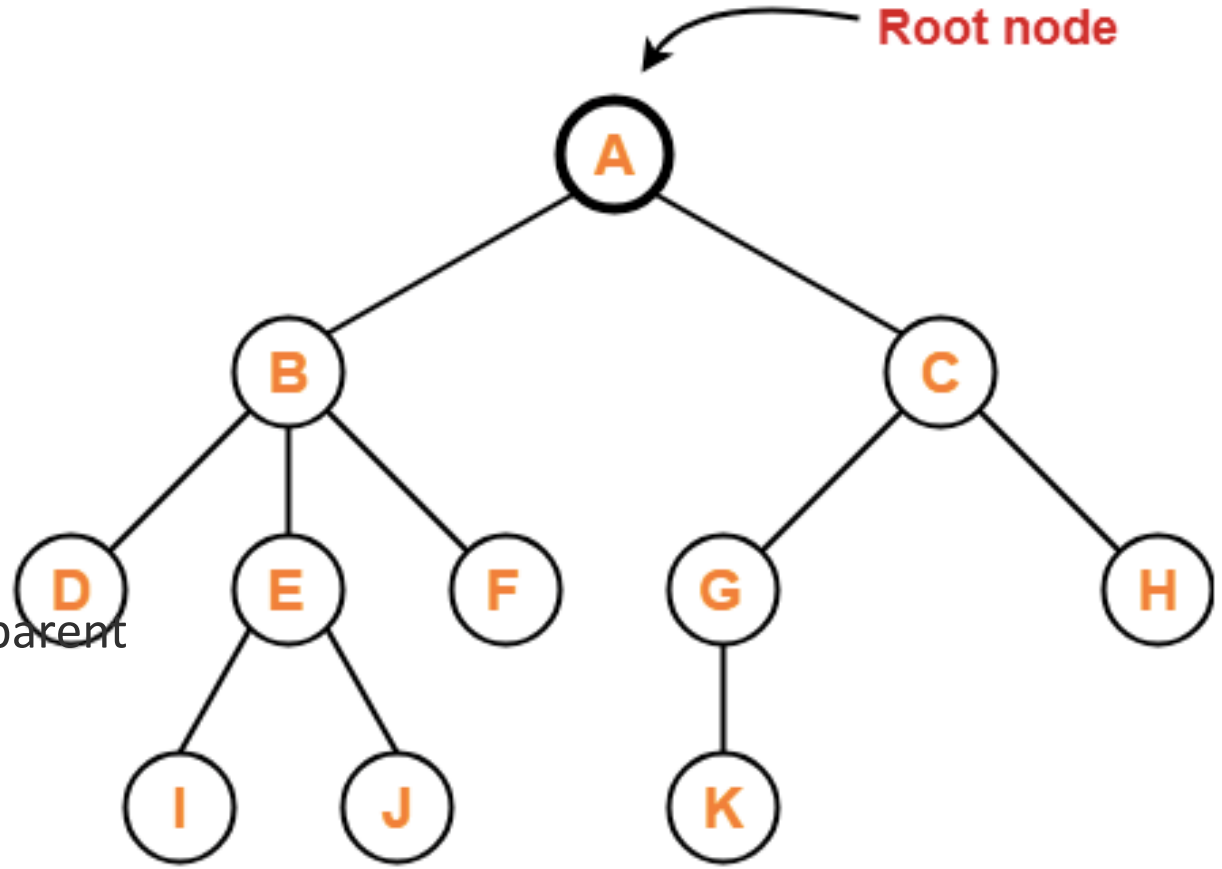
•In a tree with n number of nodes, there are exactly (n-1) number of edges.

Parent

- The node which has one or more children is called as a parent node.
- Node B is the parent of nodes D, E and F

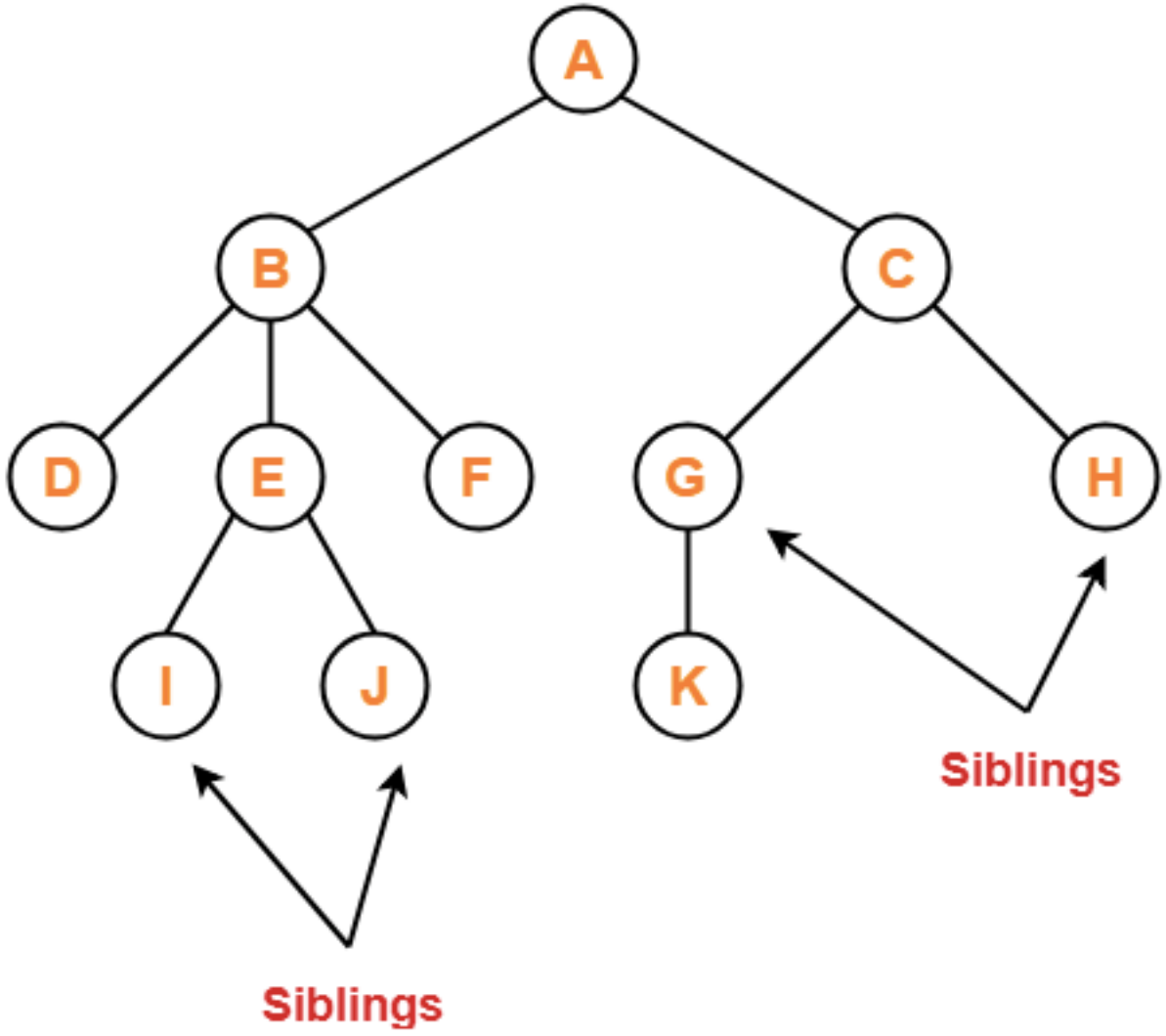
Child

- The node which is a descendant of some node is called as a **child node**.
- Nodes D, E and F are the children of node B



Siblings

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

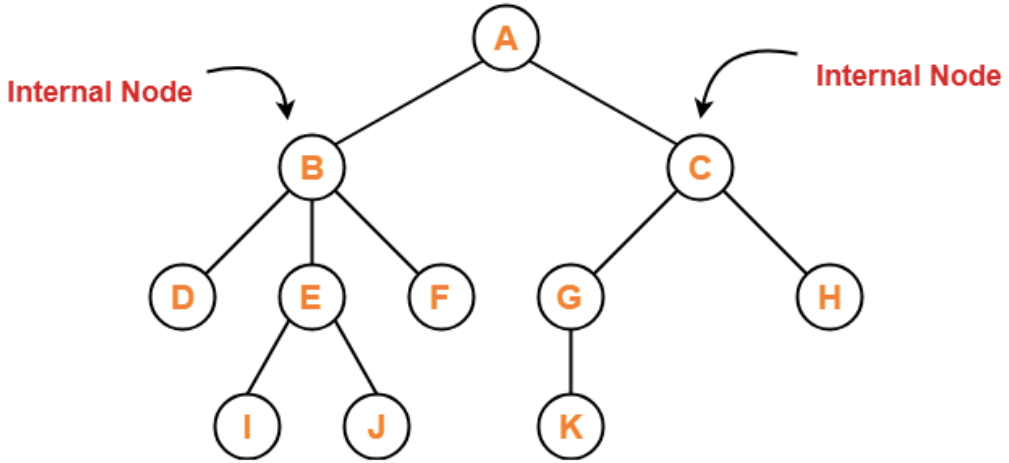


Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

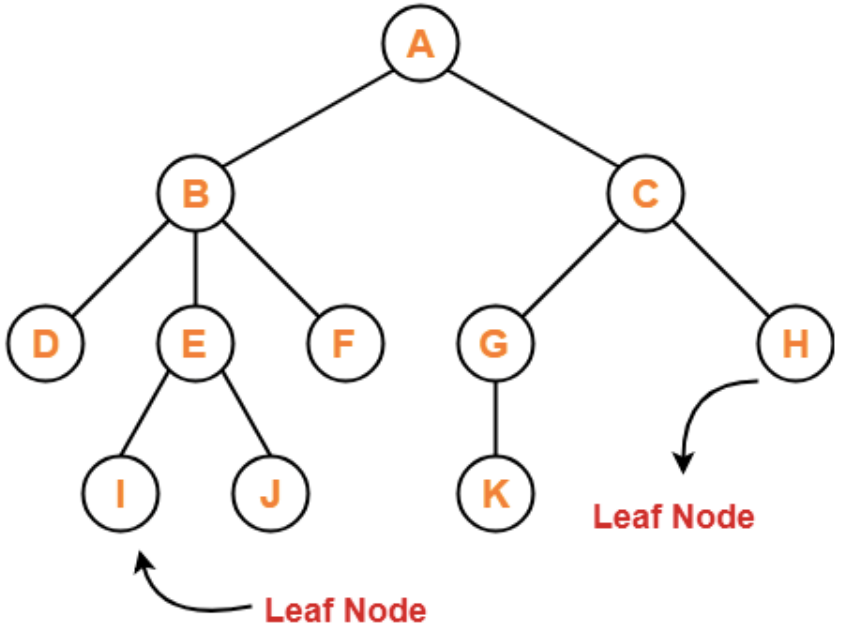
Internal Node-

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node



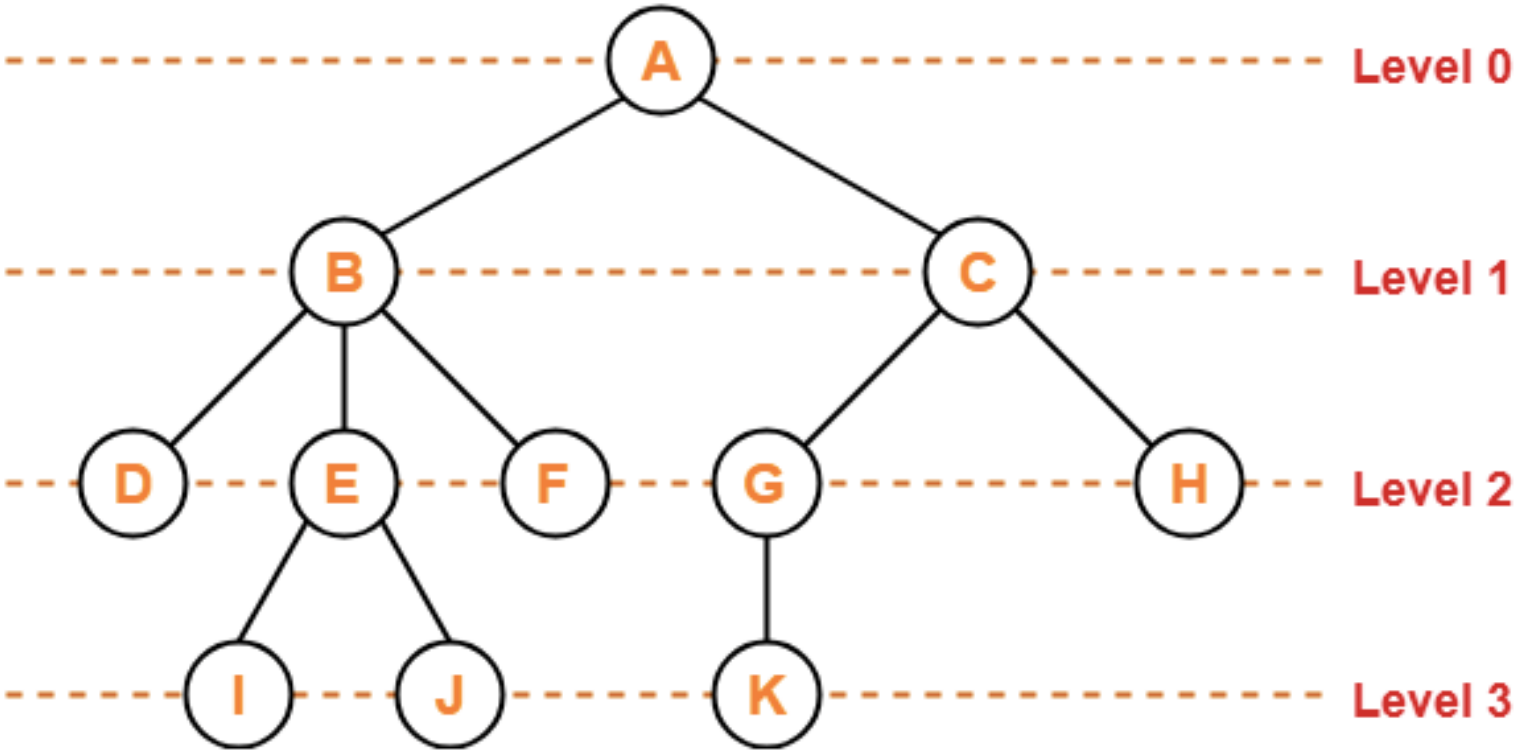
Leaf Node

- The node which does not have any child is called as a **leaf node**.



Level

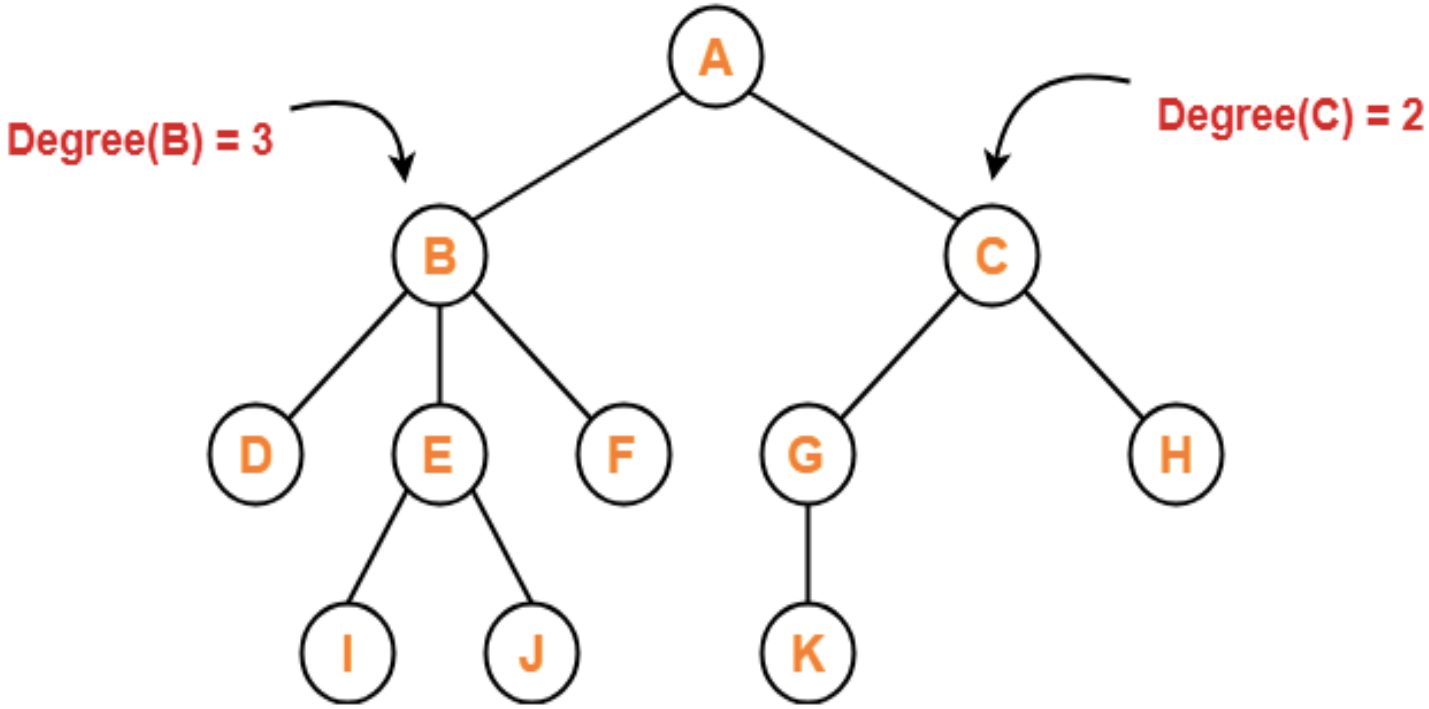
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



Degree-

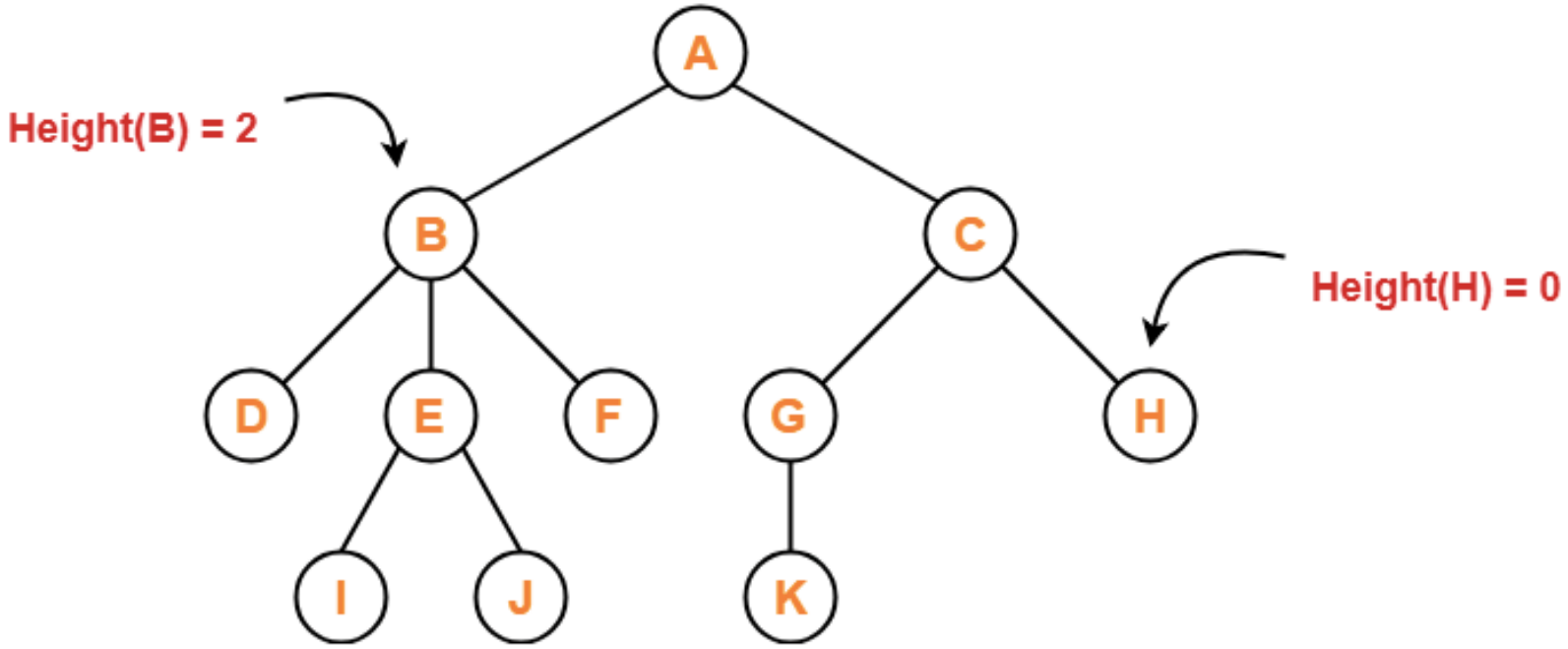
- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Here,
- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0



Height

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0



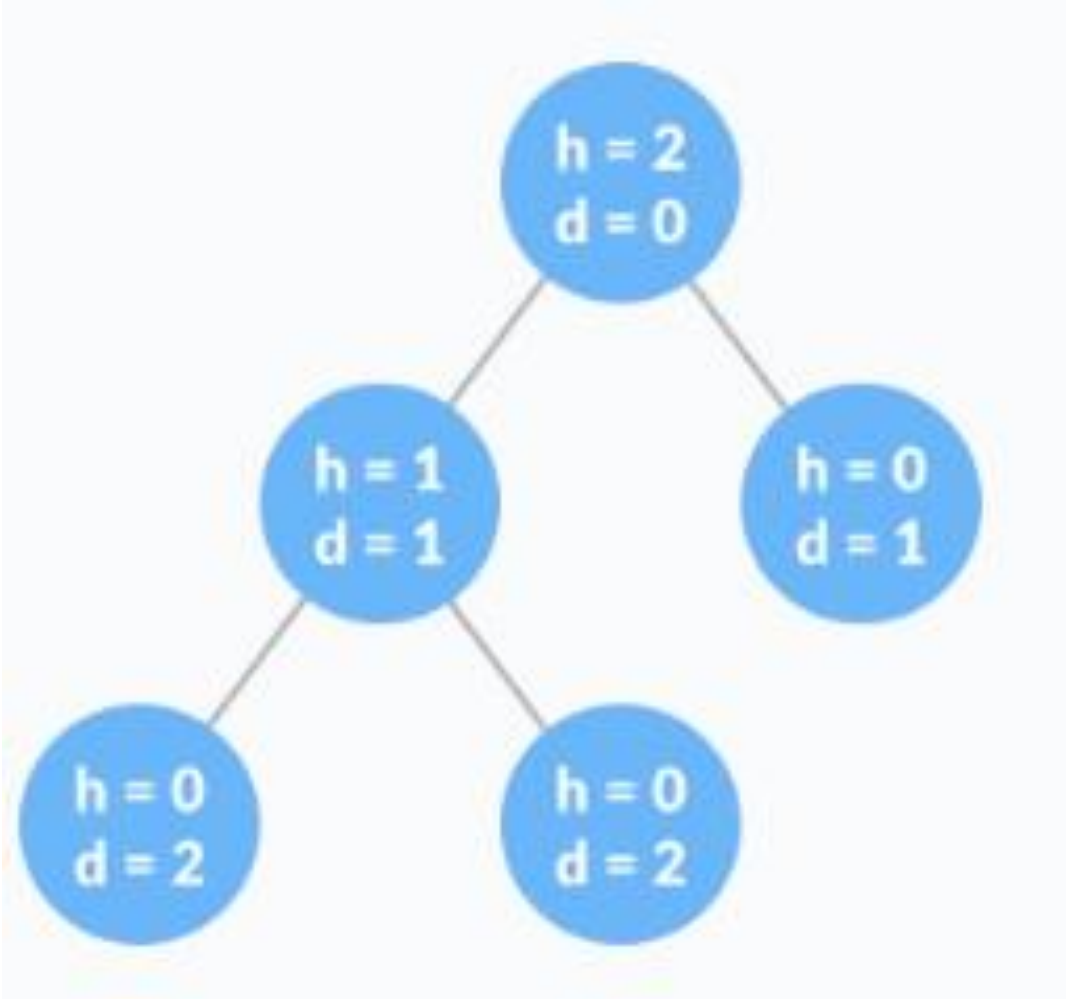
Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

The height of a Tree is the height of the root node or the depth of the deepest node.

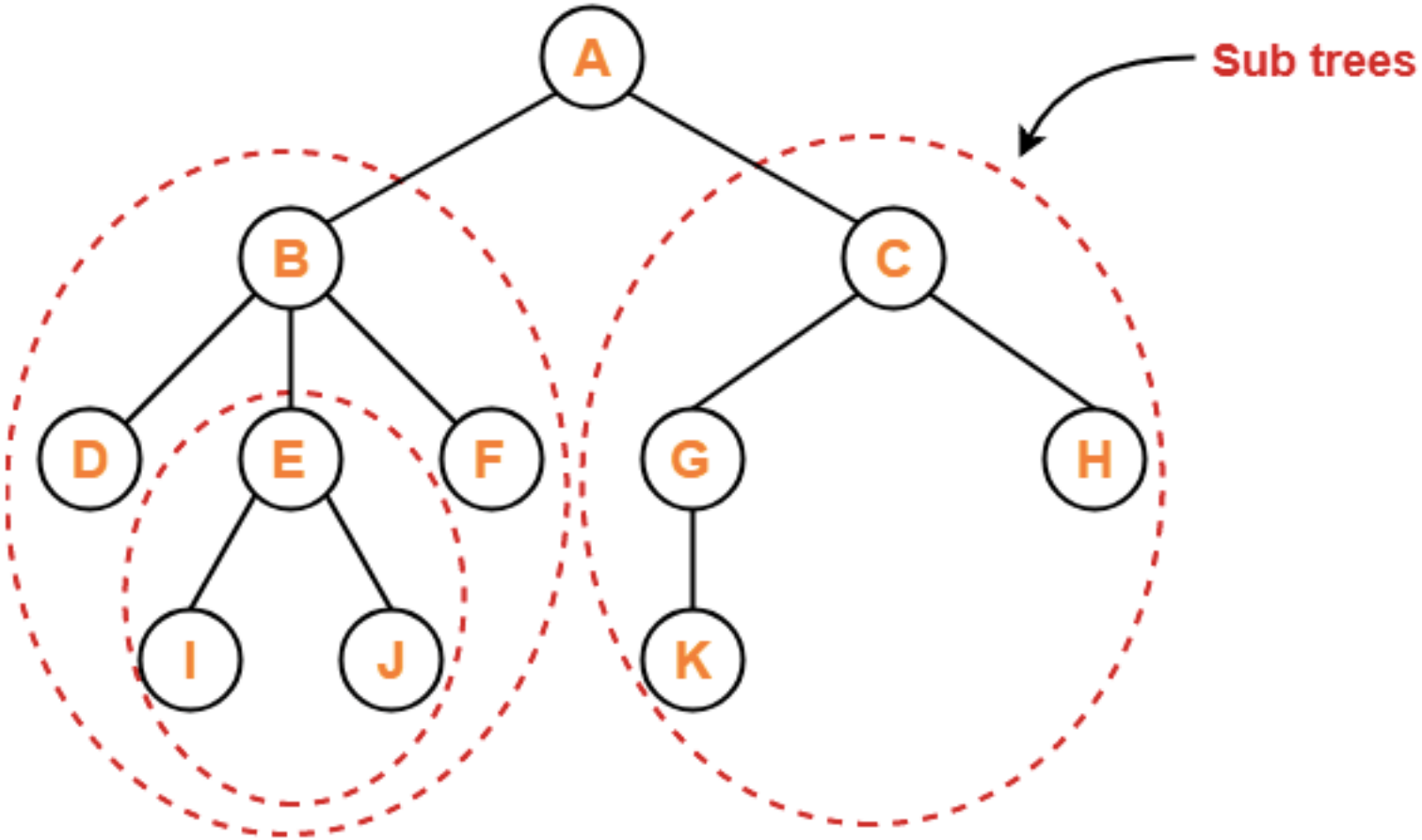
Depth

- The depth of a node is the number of edges from the root to the node.



Subtree

- Every child node forms a subtree on its parent node.

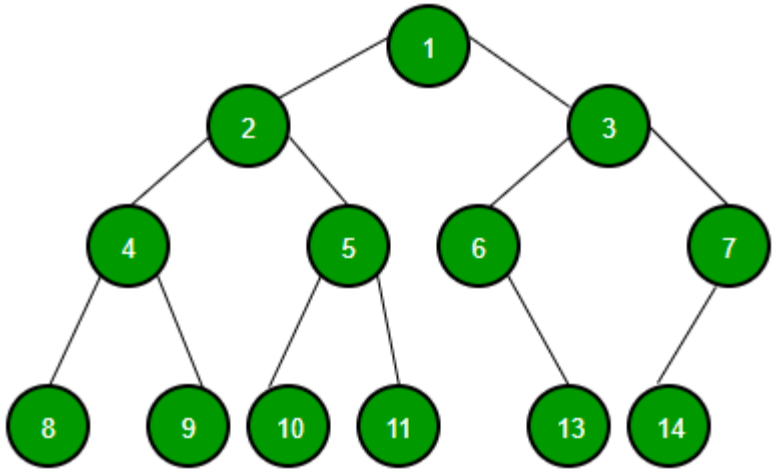


TYPES

- Binary Tree
- Binary Search Tree
- AVL Tree

BINARY TREE

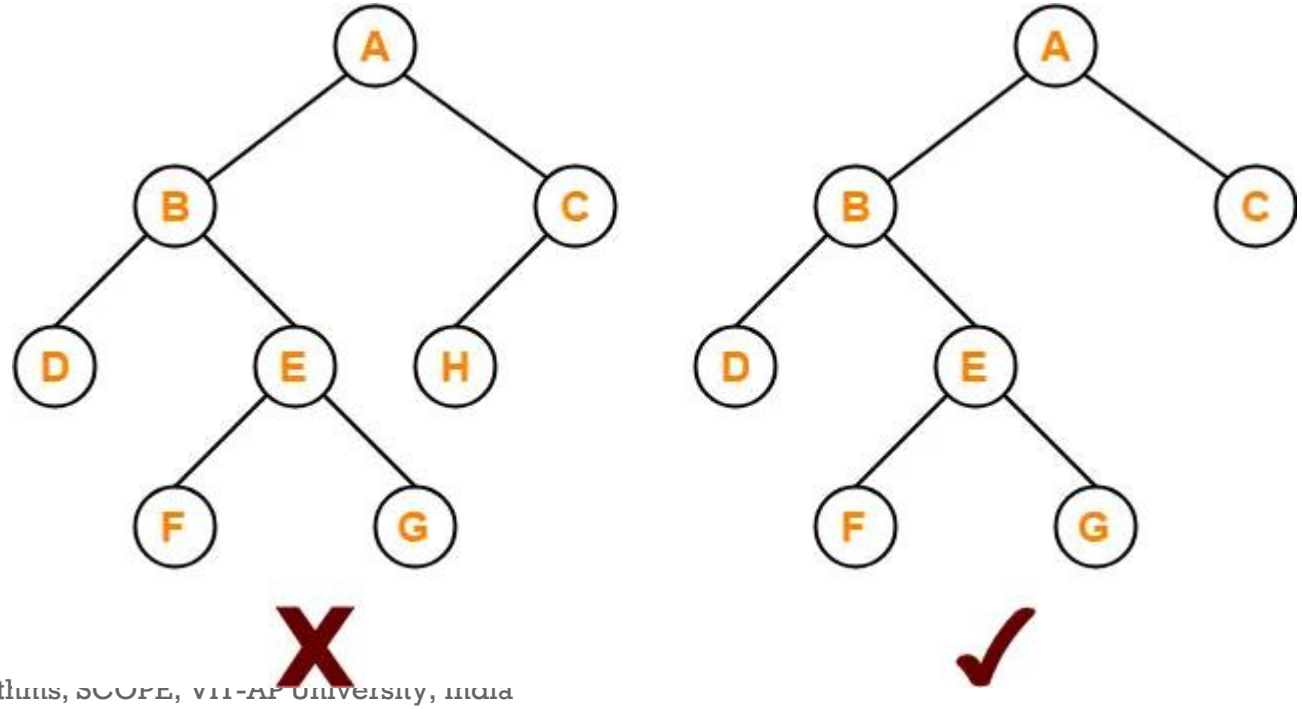
- A binary tree is a tree data structure in which each parent node can have at most two children.
- Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.
- Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



TYPES BINARY TREES

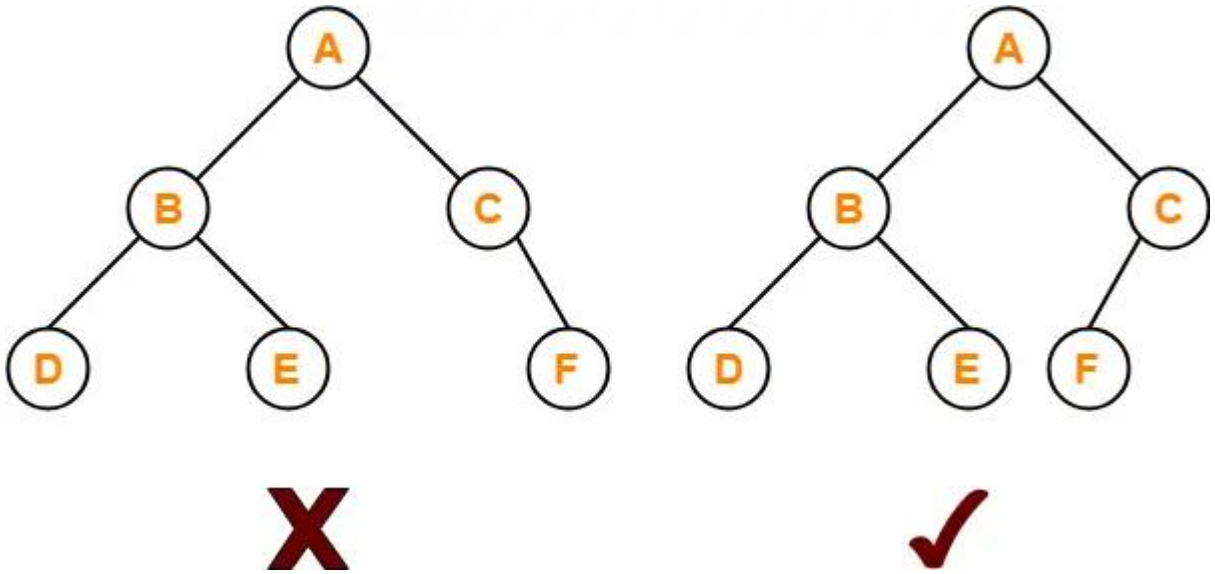
1. Full/Strictly Binary Tree

- A full Binary tree is a binary tree in which every parent node/internal node has either two or no children.



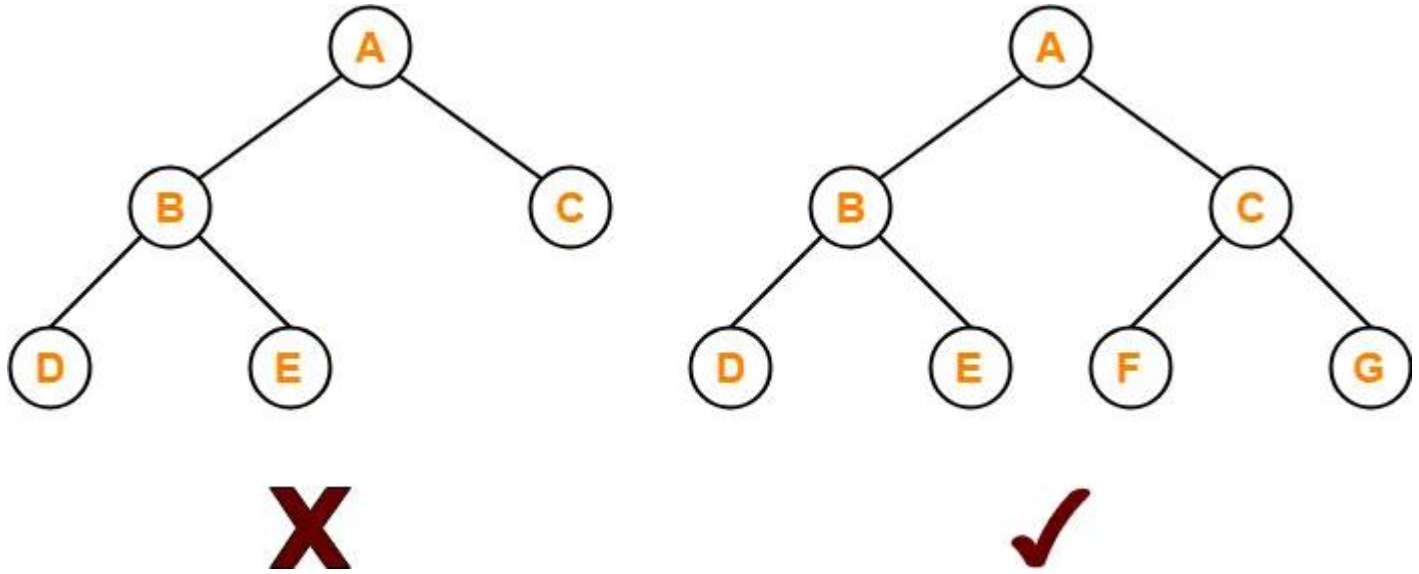
2. Complete Binary Tree

- A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.



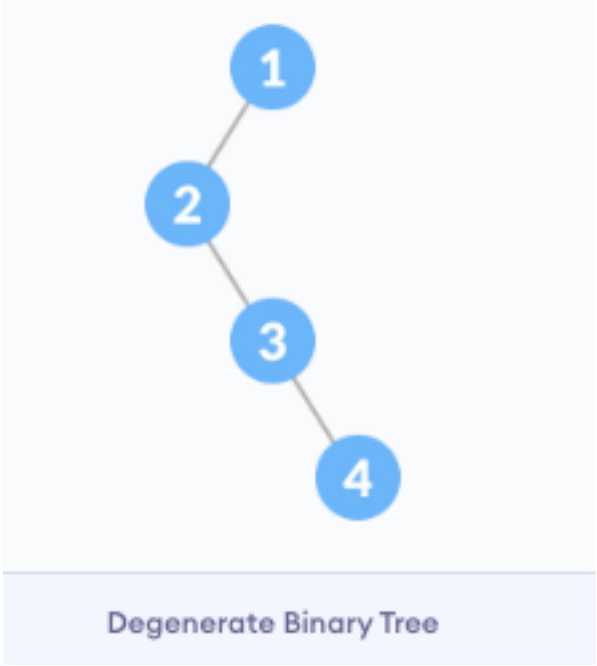
3. Perfect Binary Tree

- A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



4. Degenerate or Pathological Tree

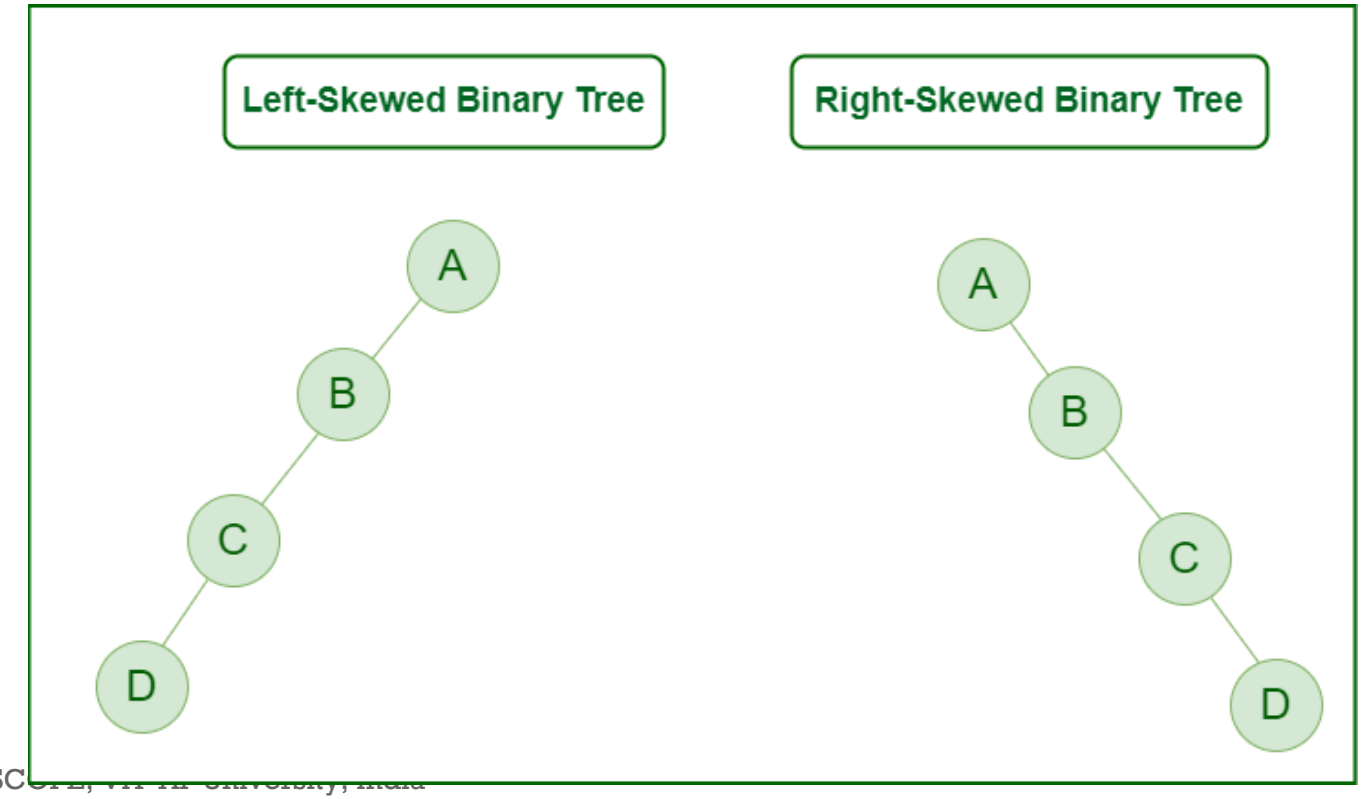
- A degenerate tree is the tree having a single child either left or right.



Degenerate Binary Tree

5. Skewed Binary Tree

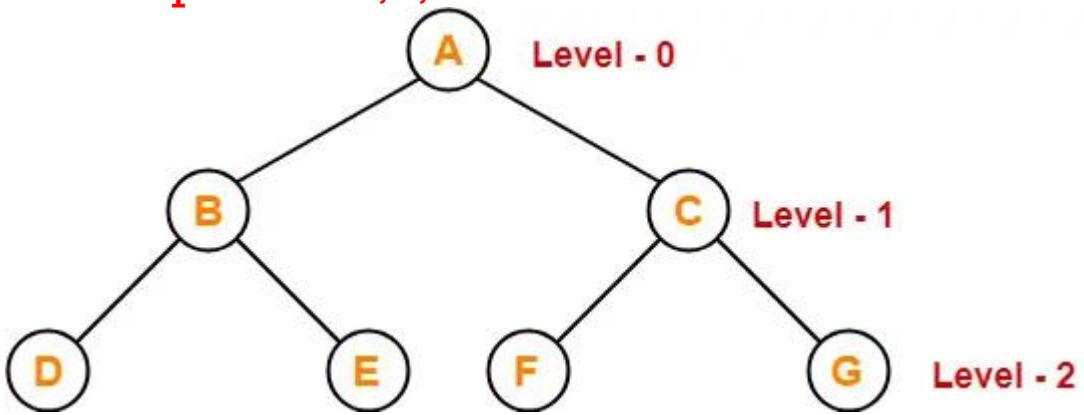
- A skewed binary tree is a degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.



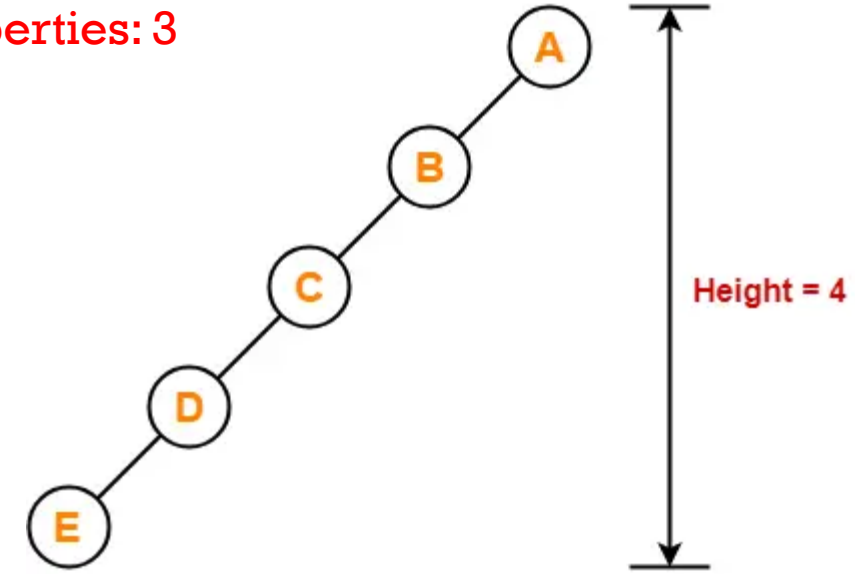
BINARY TREE PROPERTIES

1. The maximum number of nodes at level 'L' of a binary tree is 2^L
2. The Maximum number of nodes in a binary tree of height 'h' is $2^{h+1} - 1$
3. Minimum number of nodes in a binary tree of height H is $H + 1$
4. Total Number of leaf nodes in a Binary Tree= Total Number of nodes with 2 children + 1

Properties: 1,2,4



Properties: 3



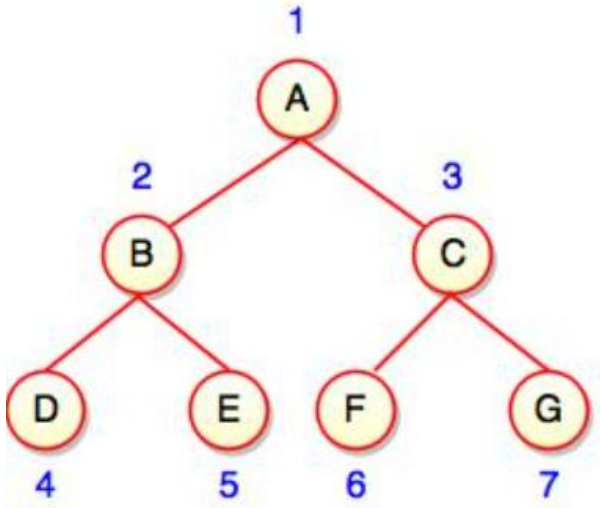
TREE REPRESENTATIONS

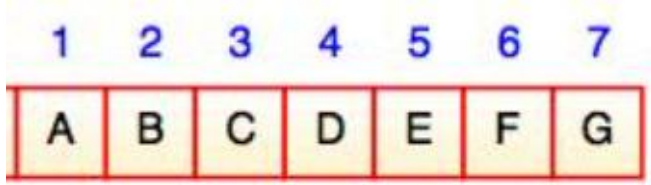
- A binary tree data structure is represented using two methods. Those methods are as follows...

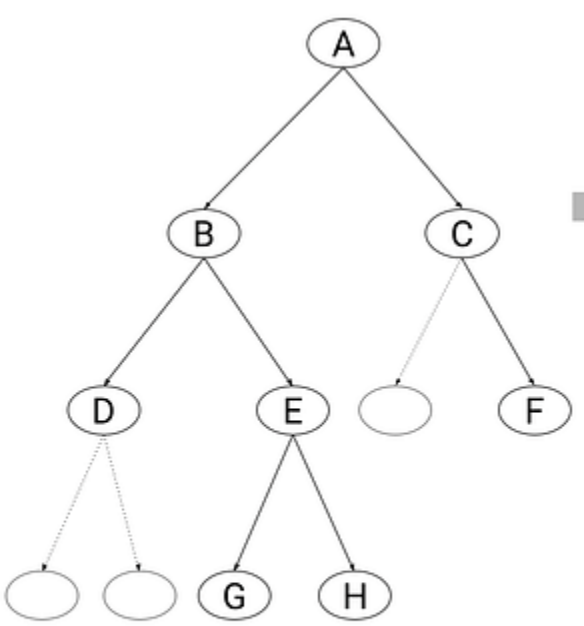
1. **Array Representation**
2. **Linked List Representation**

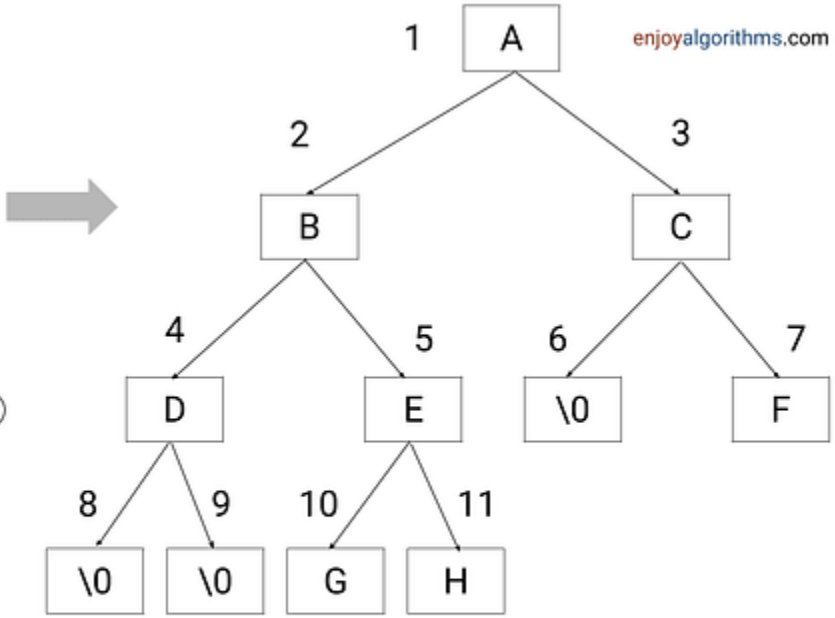
ARRAY REPRESENTATION OF BINARY TREE

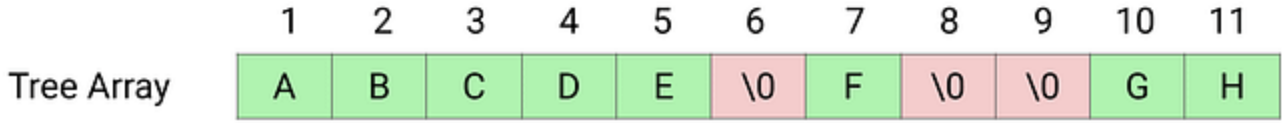
- In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
- a node i, the parent is i/2, the left child is 2i and the right child is 2i+1.





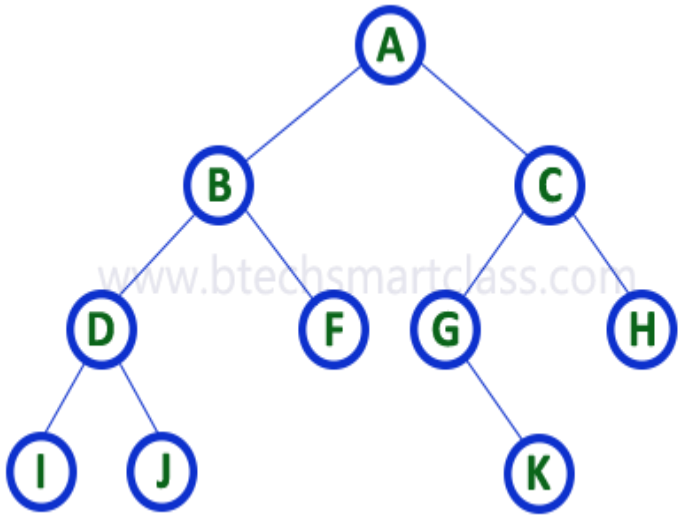


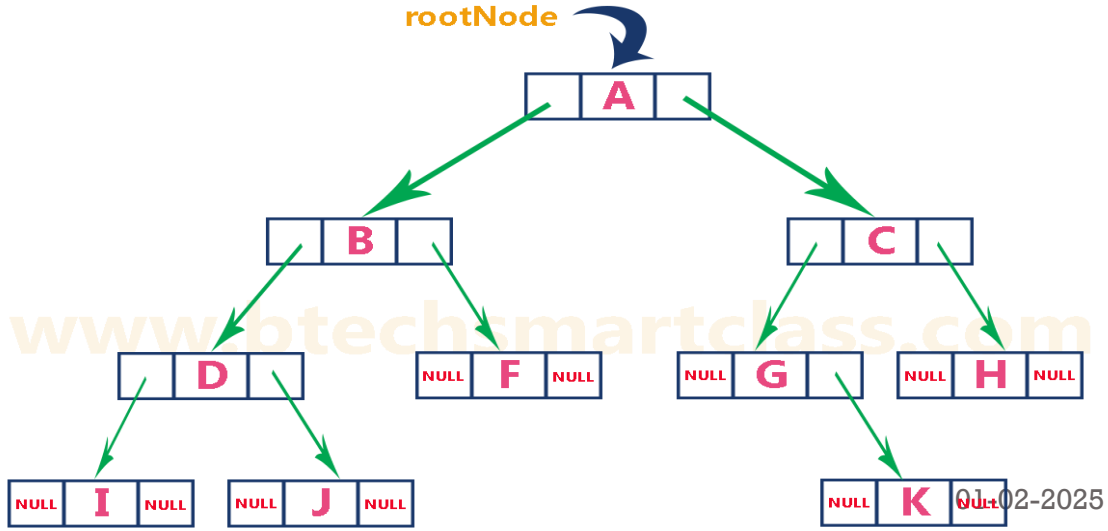




LINKED LIST REPRESENTATION

- We use a double linked list to represent a binary tree.
- In a double linked list, every node consists of three fields.
- First field for storing left child address,
- second for storing actual data and third for storing right child address.





TREE TRAVERSAL

- Tree traversal is a process of visiting each node of a tree data structure.
- Linear data structures like arrays, stacks, queues, and linked list have only one way to traverse the data.
- But a hierarchical data structure like a tree can be traversed in different ways.

3 TYPES OF TREE TRAVERSAL ALGORITHMS

- Preorder traversal (root left right)
- Inorder traversal (left root right)
- Postorder traversal (left-right root)

PREORDER TRAVERSAL

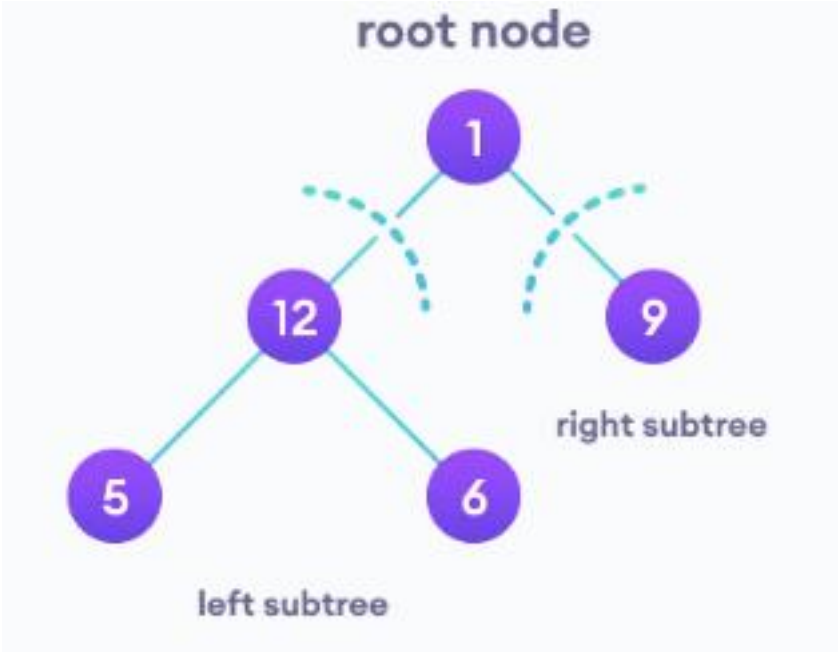
- This technique follows the 'root left right' policy.
- It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed.
- As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

Algorithm:

Step 1 - Visit the root node.

Step 2 - Visit all the nodes in the left subtree

Step 3 - Visit all the nodes in the right subtree



Pseudocode:

```
display(root.data)
preorder(root.left)
preorder(root.right)
```

Preorder: 1 -> 12 -> 5 -> 6 -> 9

INORDER TRAVERSAL

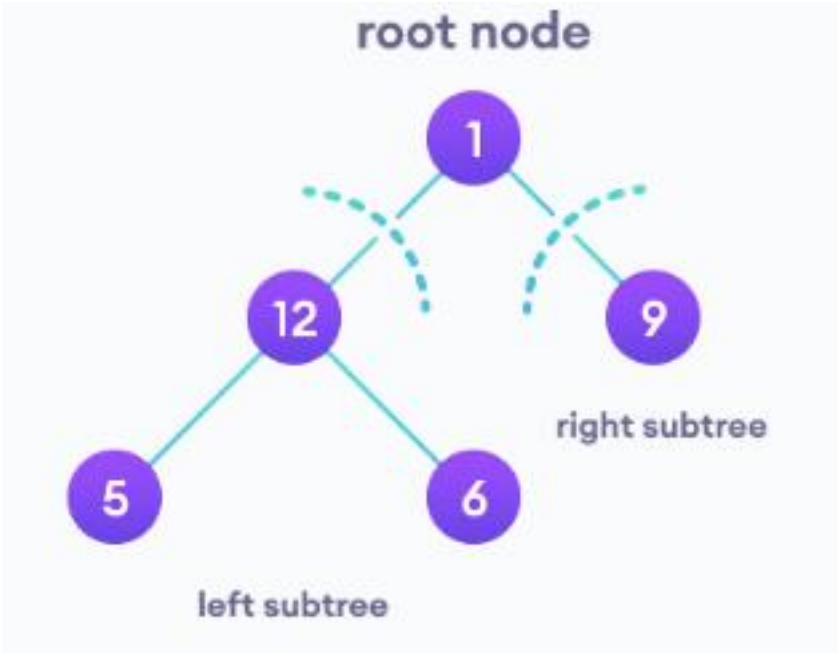
- This technique follows the 'left root right' policy.
- It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed.
- As the root node is traversed between the left and right subtree, it is named inorder traversal.

Algorithm:

Step 1 - Visit all the nodes in the left subtree

Step 2 - Visit the root node.

Step 3 - Visit all the nodes in the right subtree



Pseudocode:

```
inorder(root.left)
display(root.data)
inorder(root.right)
```

Inorder: 5 -> 12 -> 6 -> 1 -> 9

POSTORDER TRAVERSAL

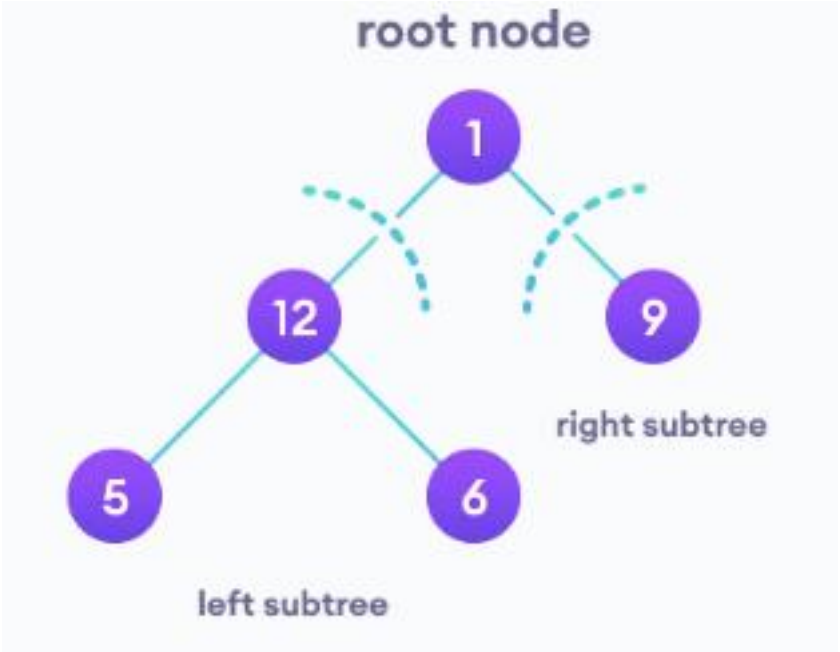
- This technique follows the 'left-right root' policy.
- It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed.
- As the root node is traversed after (or post) the left and right subtree, it is called Postorder traversal.

Algorithm:

Step 1 - Visit all the nodes in the left subtree

Step 2 - Visit all the nodes in the right subtree

Step 3 - Visit the root node.



Pseudocode:

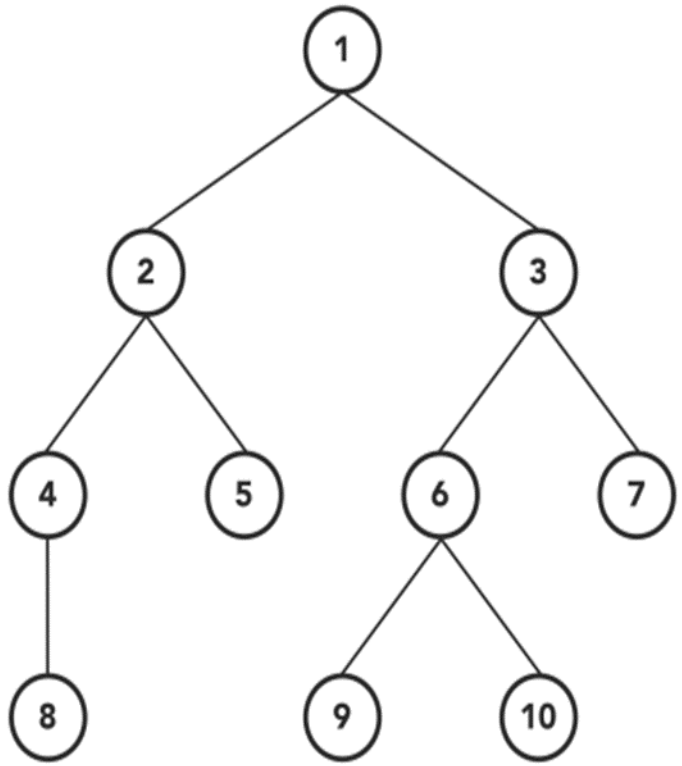
```
postorder(root.left)
postorder(root.right)
display(root.data)
```

Preorder: 5 -> 6 -> 12 -> 9 -> 1

Inorder: 'left root right

Preorder: 'root left right

Postorder: 'left-right root

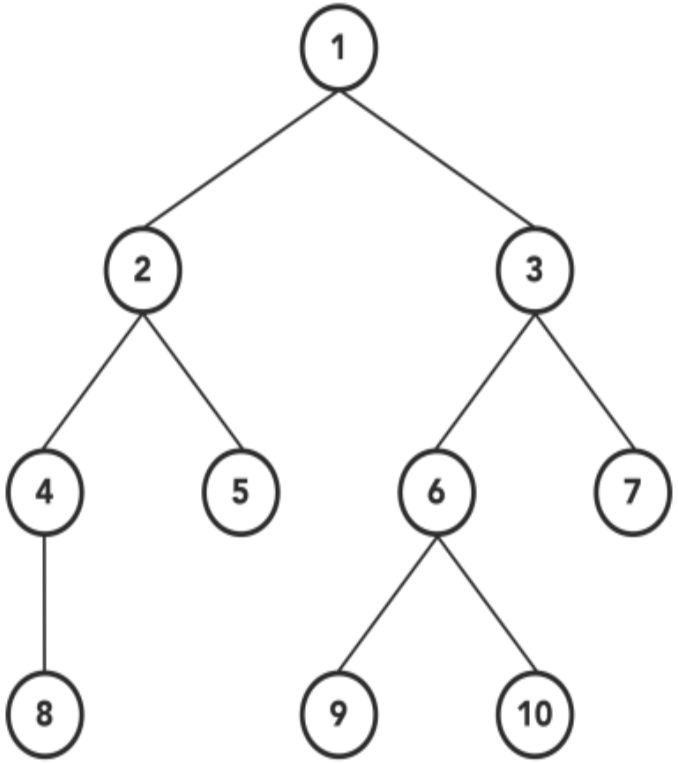


Preorder: 1 2 4 8 5 3 6 9 10 7

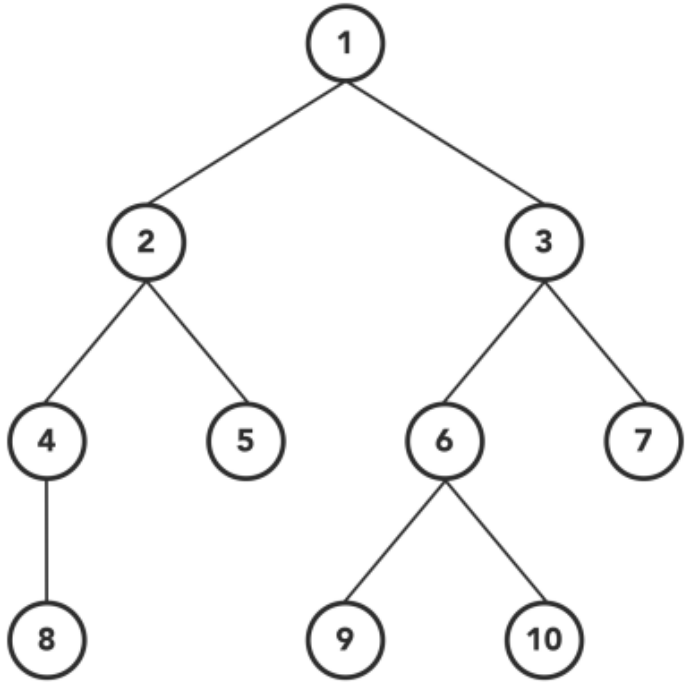
Inorder: 8 4 2 5 1 9 6 10 3 7

Postorder: 8 4 5 2 9 10 6 7 3 1

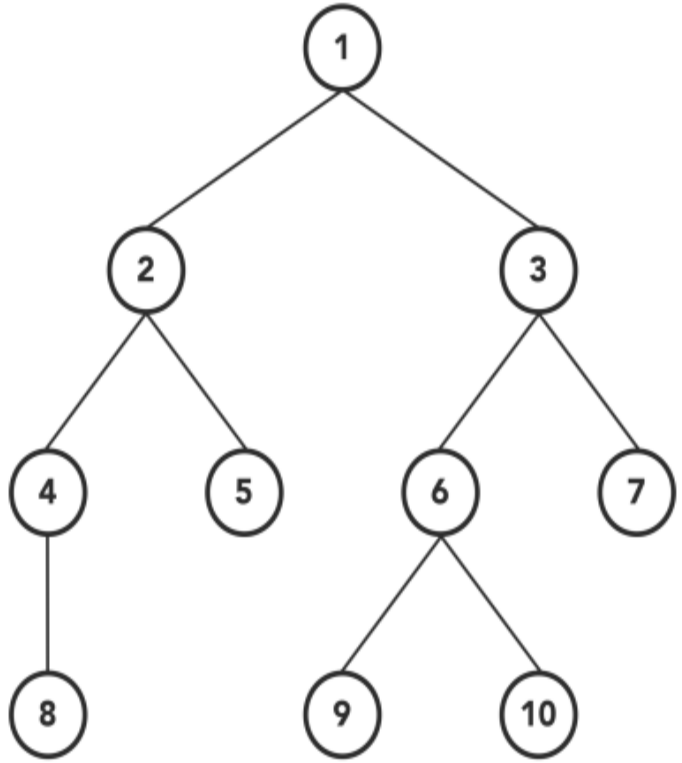
Inorder



Preorder



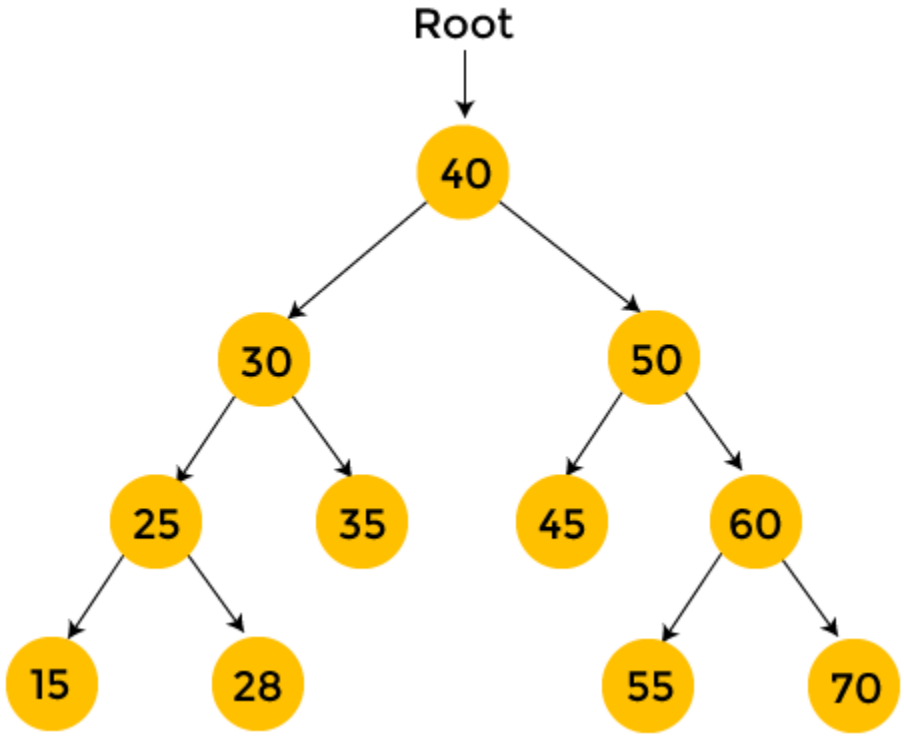
Postorder



Preorder: 'root left right

Inorder: 'left root right

Postorder: 'left-right root



Preorder: 40 30 25 15 28 35 50 45 60 55 70

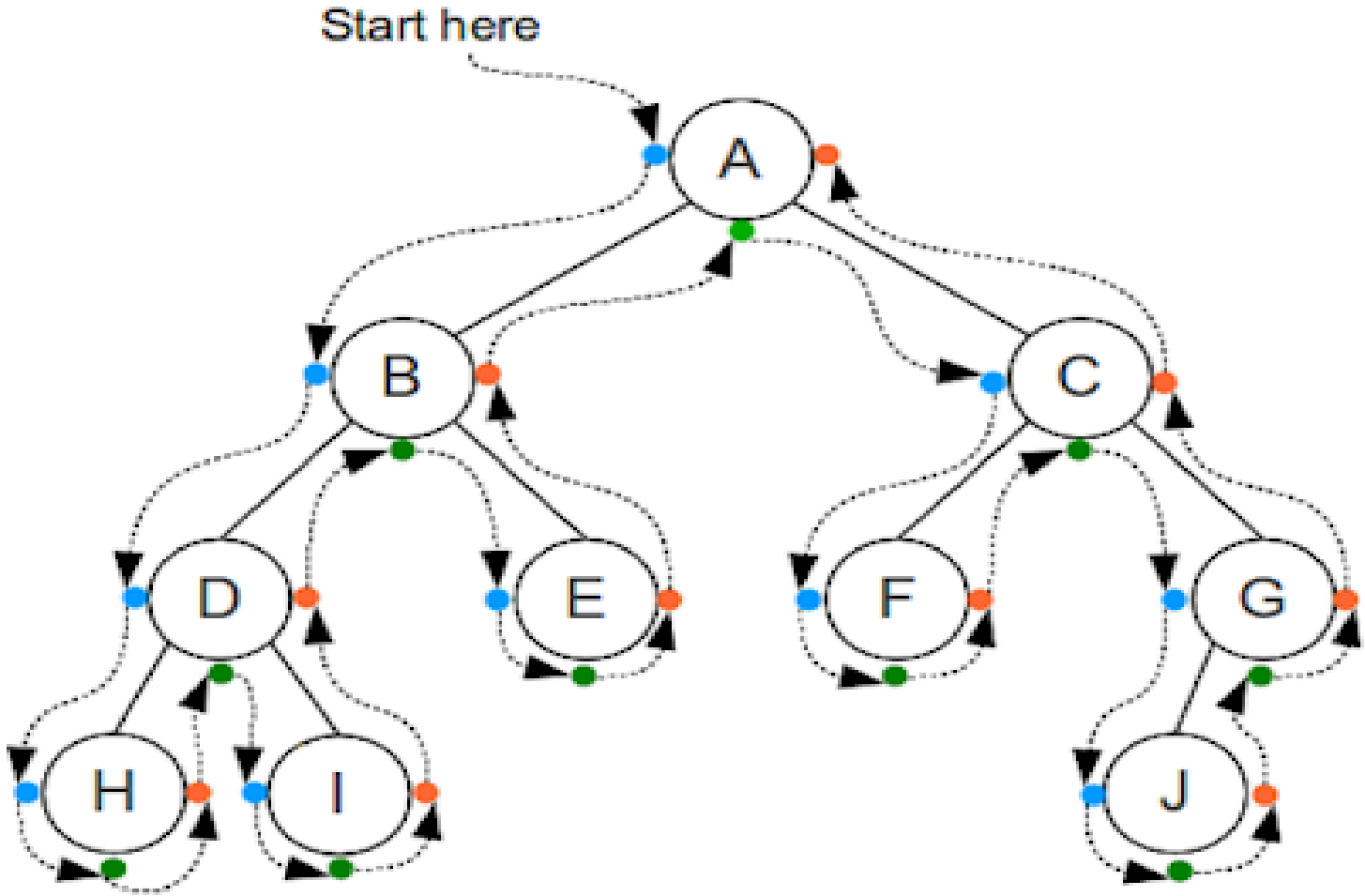
Inorder: 15 25 28 30 35 40 45 50 55 60 70

Postorder: 15 28 25 35 30 45 55 70 60 50 40

Preorder Traversal (Data Structures) - javatpoint

Inorder Traversal (Data Structures) - javatpoint

Postorder Traversal (Data Structures) - javatpoint

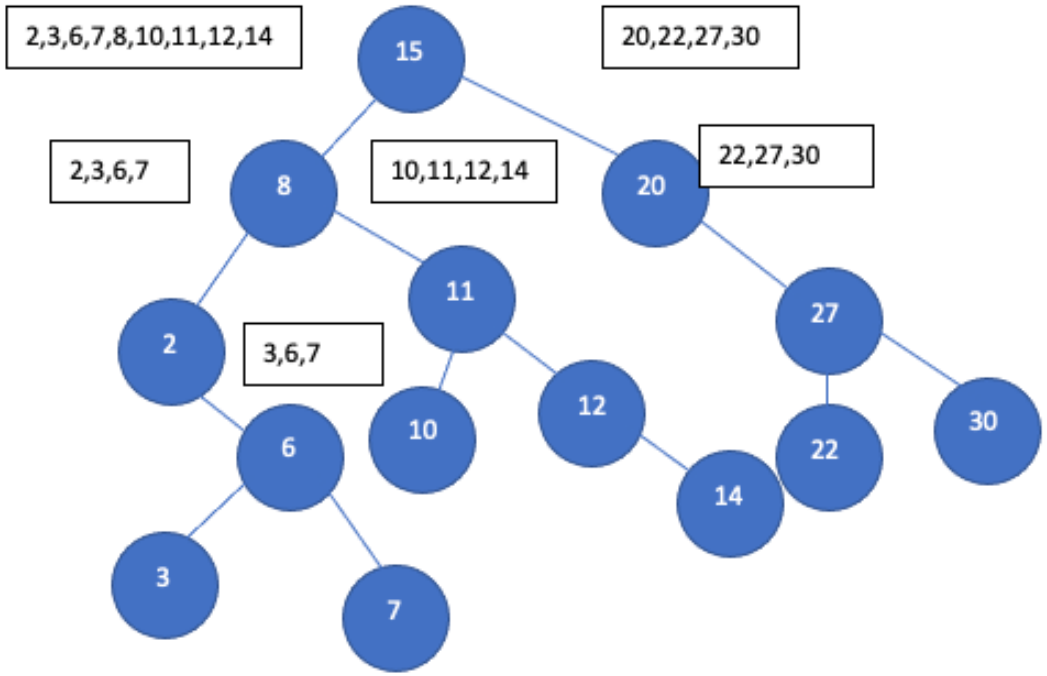


Pre-Order	ABDHIECFGJ
In-Order	HDIBEAFCJG
Post-Order	HIDEBFJGCA

BINARY TREE CONSTRUCTION FROM PREORDER AND INORDER TRAVERSAL

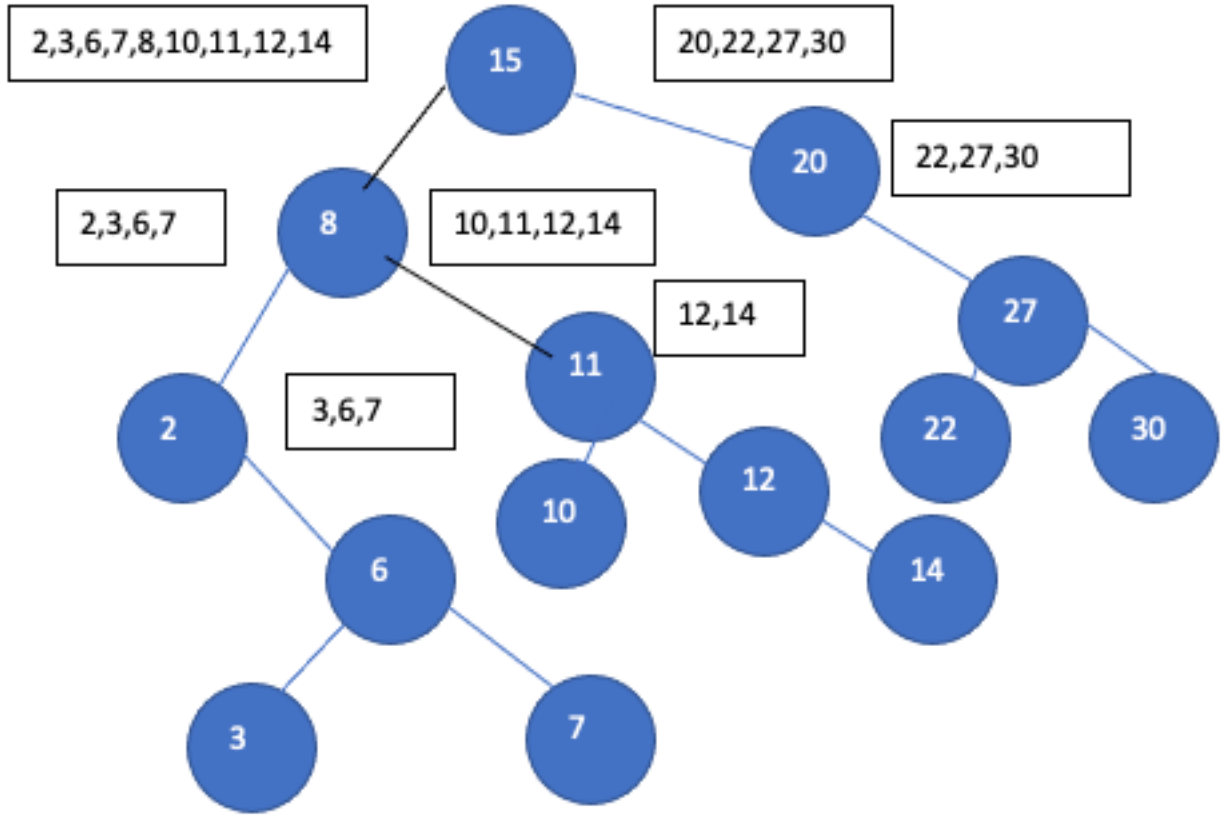
Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30 (Left Root Right)

Preorder: 15 8 2 6 3 7 11 10 12 14 20 27 22 30 (Root Left Right) (Scan from Left to Right)



Binary Tree construction from Postorder and Inorder traversal

- Inorder: 2 3 6 7 8 10 11 12 14 15 20 22 27 30 (Left Root Right)
- Postorder: 3 7 6 2 10 14 12 11 8 22 30 27 20 15 (Left Right Root) (Scan from Right to Left)

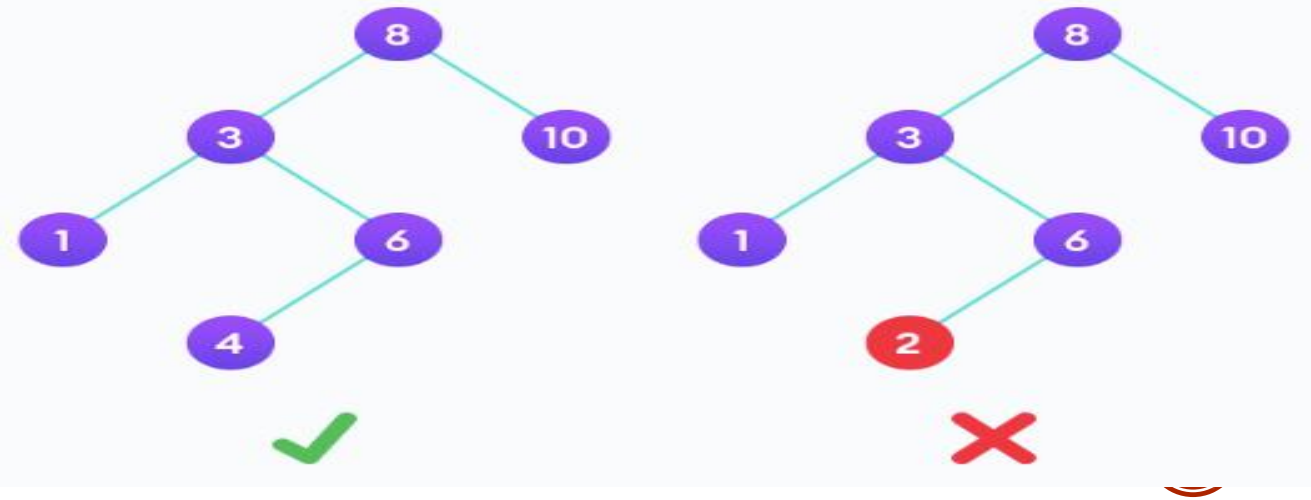


Binary Tree construction from Postorder and Inorder traversal

1. The postorder traversal of a binary tree is 8, 9, 6, 7, 4, 5, 2, 3, 1. The inorder traversal of the same tree is 8, 6, 9, 4, 7, 2, 5, 1, 3. The height of a tree is the length of the longest path from the root to any leaf. The height of the binary tree above is _____ .

BINARY SEARCH TREES (BST)

- A binary search tree (BST) is a data structure used in computer science and mathematics for organizing and storing a collection of elements, such as numbers or other data types.
- It is a type of binary tree with the following properties:
  1. Each node in the tree has at most two children, which are referred to as the left child and the right child.
  2. The nodes in the left subtree of a node have values less than or equal to the value of that node.
  3. The nodes in the right subtree of a node have values greater than the value of that node.
     - These properties make binary search trees particularly useful for searching, inserting, and deleting elements efficiently.



BST OPERATIONS

- Insertion
- Search
- Deletion
- **Traversal**

INSERTION

Algorithm:

1.Start at the root of the tree.

2.If the tree is empty (root is null), create a new node with the desired value and make it the root of the tree.

3.If the tree is not empty, compare the value you want to insert with the value of the current node.

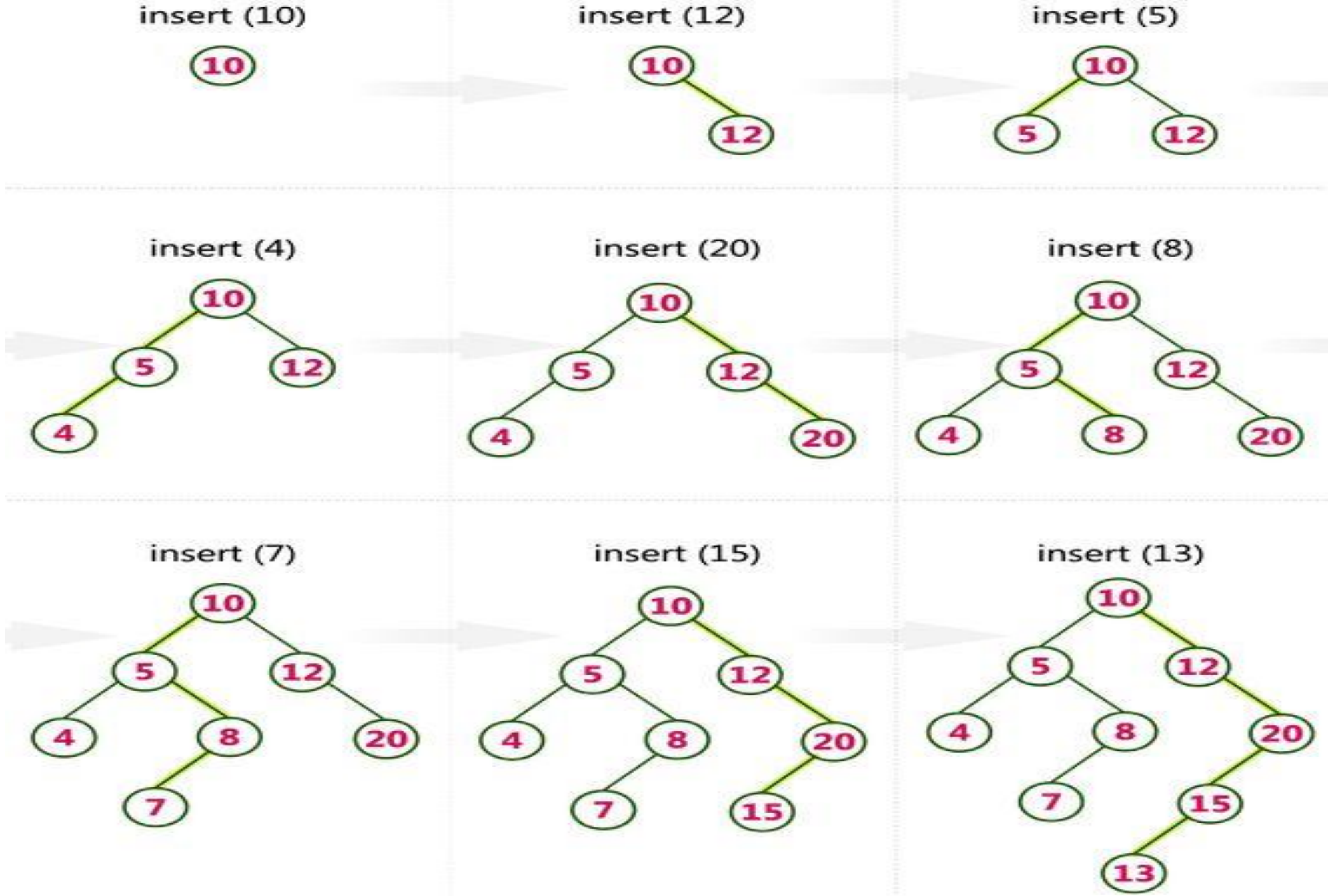
a. If the value to insert is less than the current node's value, move to the left child (if it exists).

b. If the value to insert is greater than or equal to the current node's value, move to the right child (if it exists).

4.Repeat step 3 until you reach a null (empty) child. This is the location where the new node should be inserted.

5.Create a new node with the desired value and make it the left child of the current node if the value is less than the current node's value, or the right child if it's greater.

INSERTION


insert (10)
10
insert (12)
10
12
insert (5)
10
5
12
insert (4)
10
5
12
4
insert (20)
10
5
12
4
20
insert (8)
10
5
12
4
8
20
insert (7)
10
5
12
4
8
20
7
insert (15)
10
5
12
4
8
20
7
15
insert (13)
10
5
12
4
8
20
7
15
13

```
class Node {
    int key;
    Node left, right;

    public Node(int item) {
        key = item;
        left = right = null;
    }
}
class BinarySearchTree {
    Node root;

    BinarySearchTree() {
        root = null;
    }
    void insert(int key) {
        root = insertRec(root, key);
    }
Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }
        if (key < root.key) {
            root.left = insertRec(root.left, key);
        } else if (key > root.key) {
            root.right = insertRec(root.right, key);
        }
        return root;
    }
```

```
void inOrder() {
        inOrderRec(root);
    }
    void inOrderRec(Node root) {
        if (root != null) {
            inOrderRec(root.left);
            System.out.print(root.key + " ");
            inOrderRec(root.right);
        }
    }
public static void main(String[] args) {
        BinarySearchTree bst = new
BinarySearchTree();

        // Insert elements into the BST
        bst.insert(50);
        bst.insert(30);
        bst.insert(20);
        bst.insert(40);
        bst.insert(70);
        bst.insert(60);
        bst.insert(80);

System.out.println("In-order traversal of the BST:");
        bst.inOrder();
    }
}
```

In-order traversal of the BST:

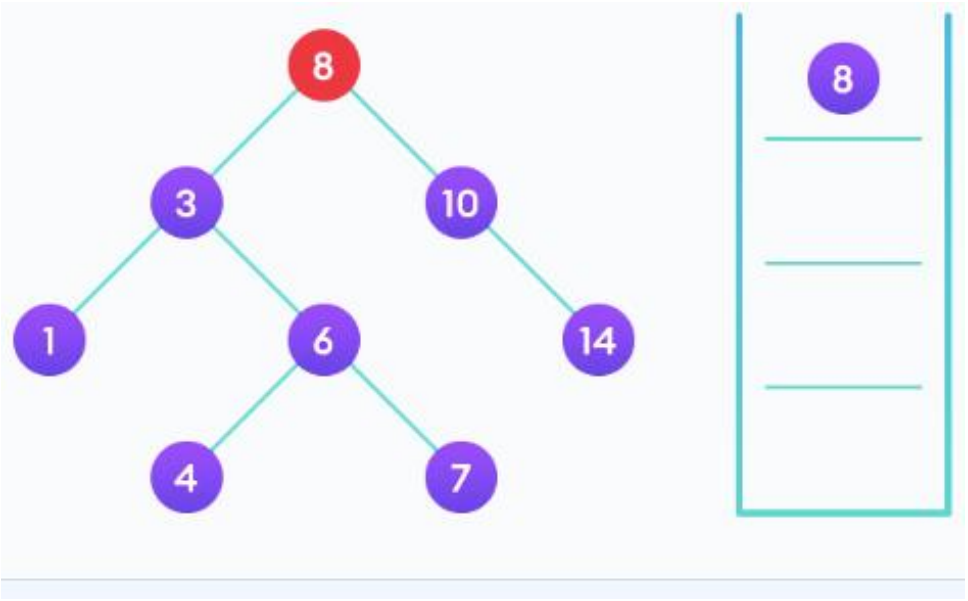
20 30 40 50 60 70 80

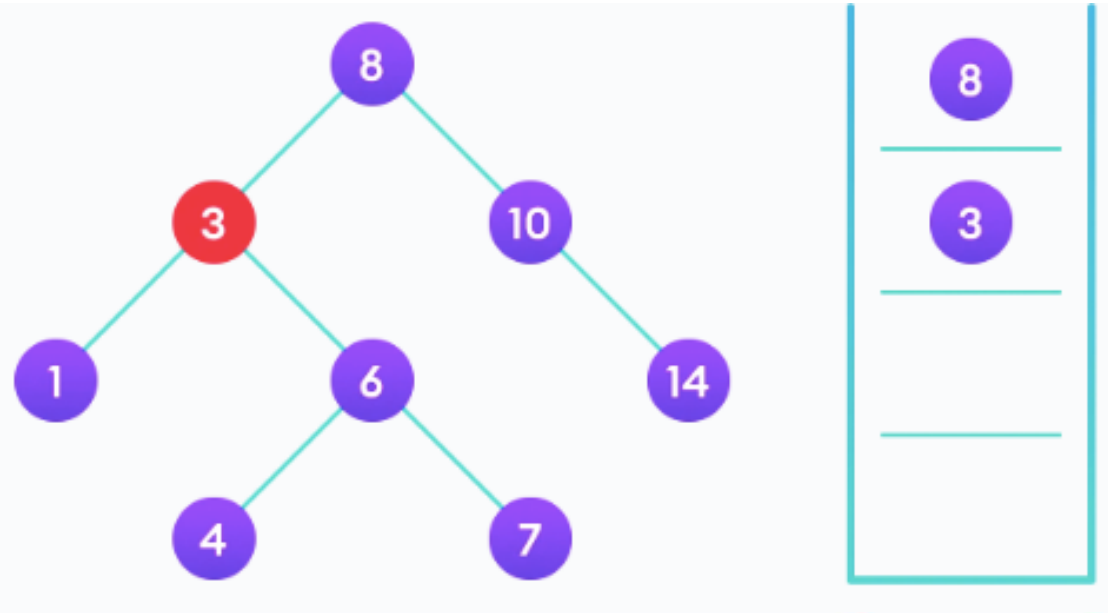
SEARCH

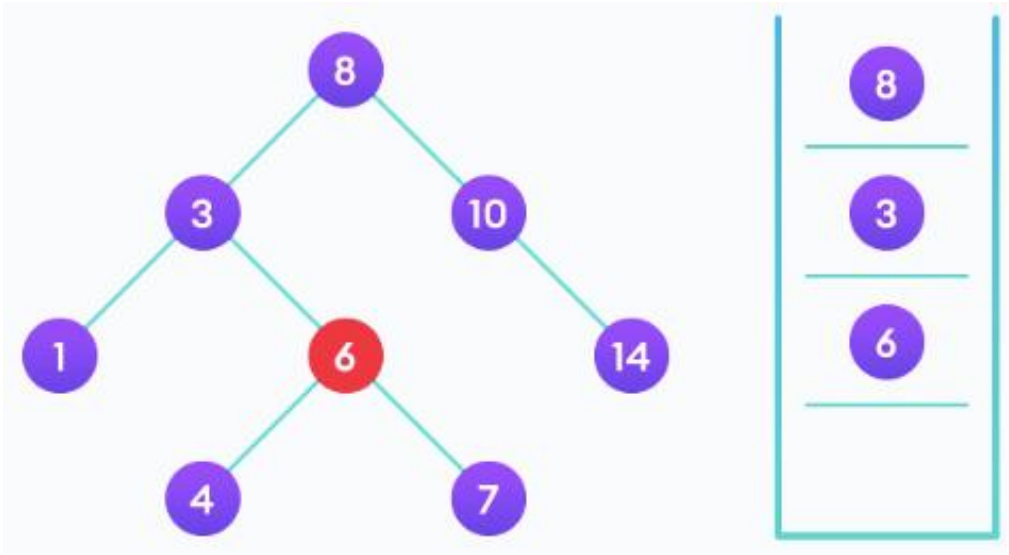
- The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.
- If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

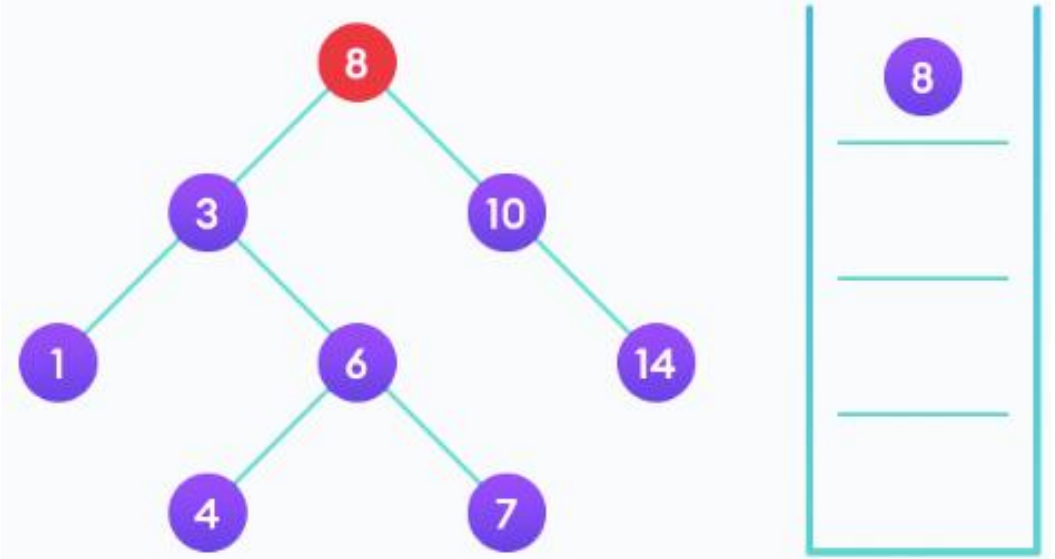
Algorithm:

```
If root == NULL
return NULL;
If number == root->data
return root->data;
If number < root->data
return search(root->left)
If number > root->data
return search(root->right)
```

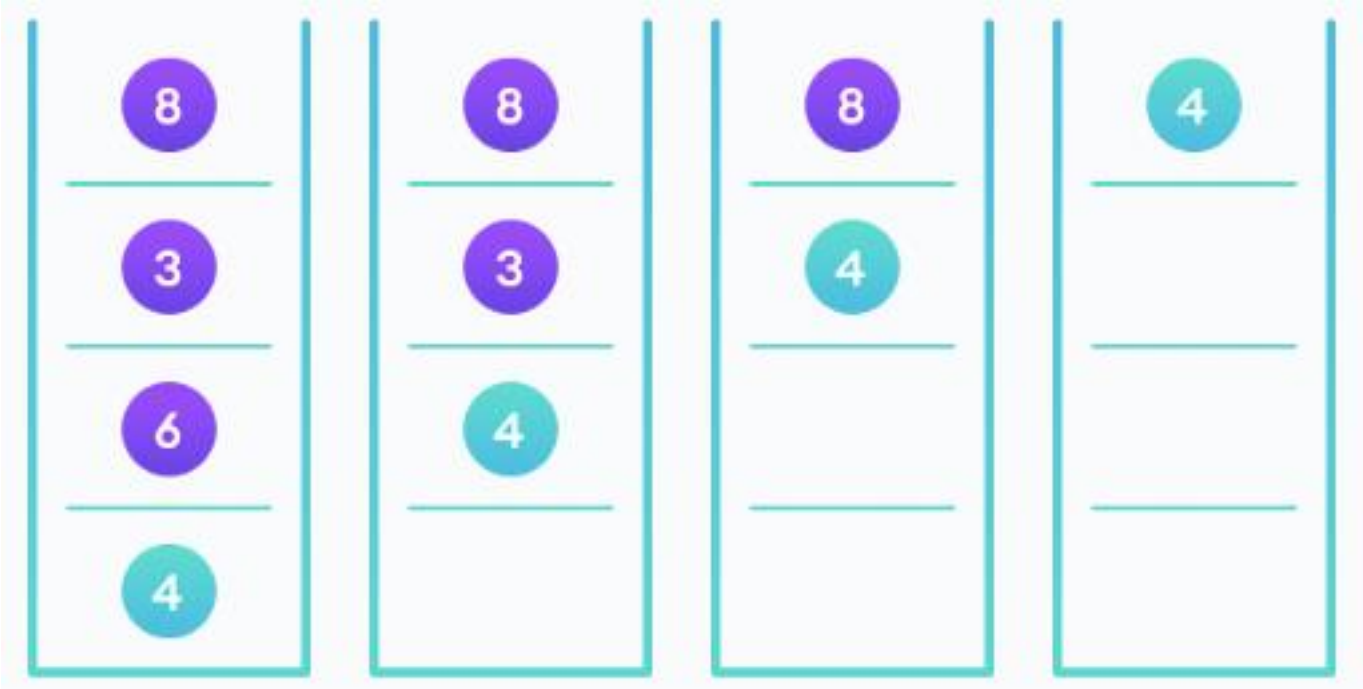

8
3
10
1
6
14
4
7
8


8
3
10
1
6
14
4
7
8
3

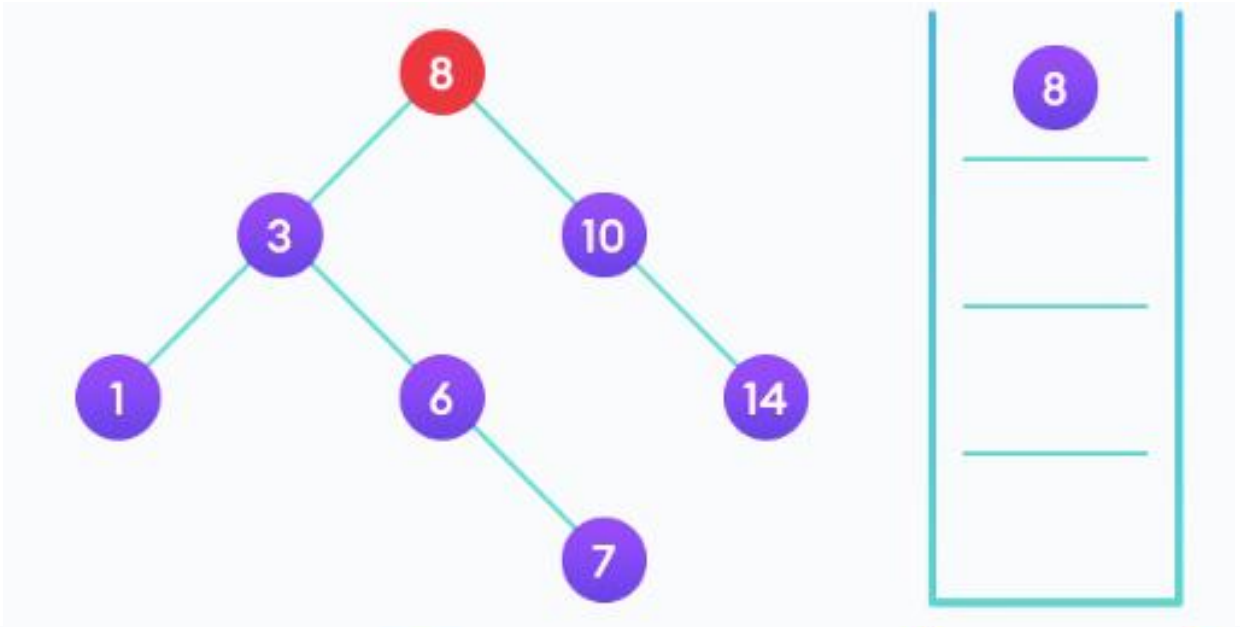

8
3
10
1
6
14
4
7
8
3
6

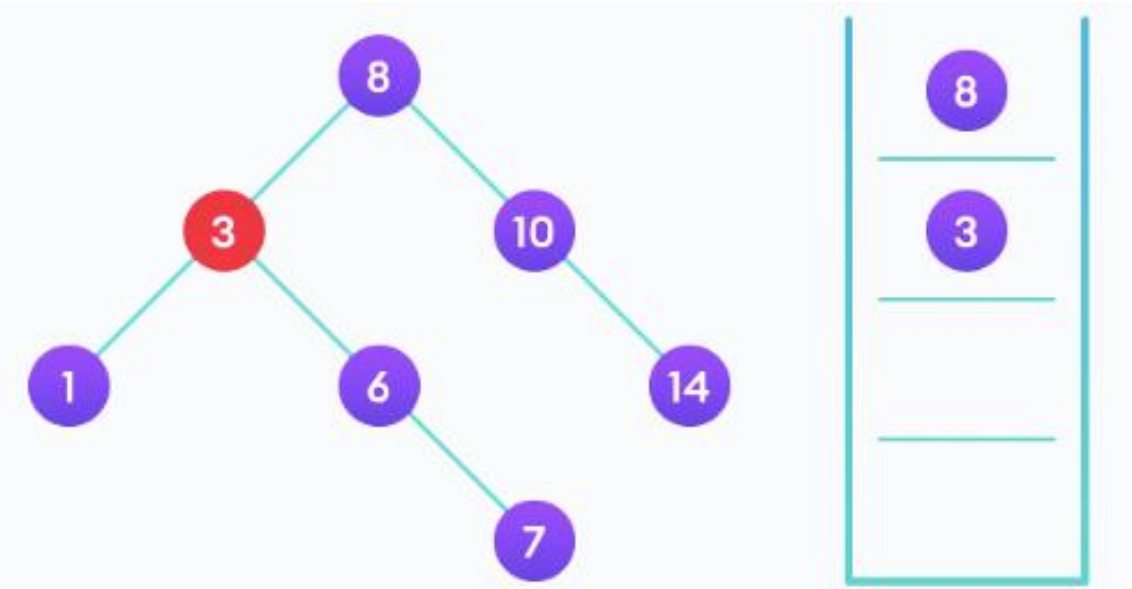

8
3
10
1
6
14
4
7
8

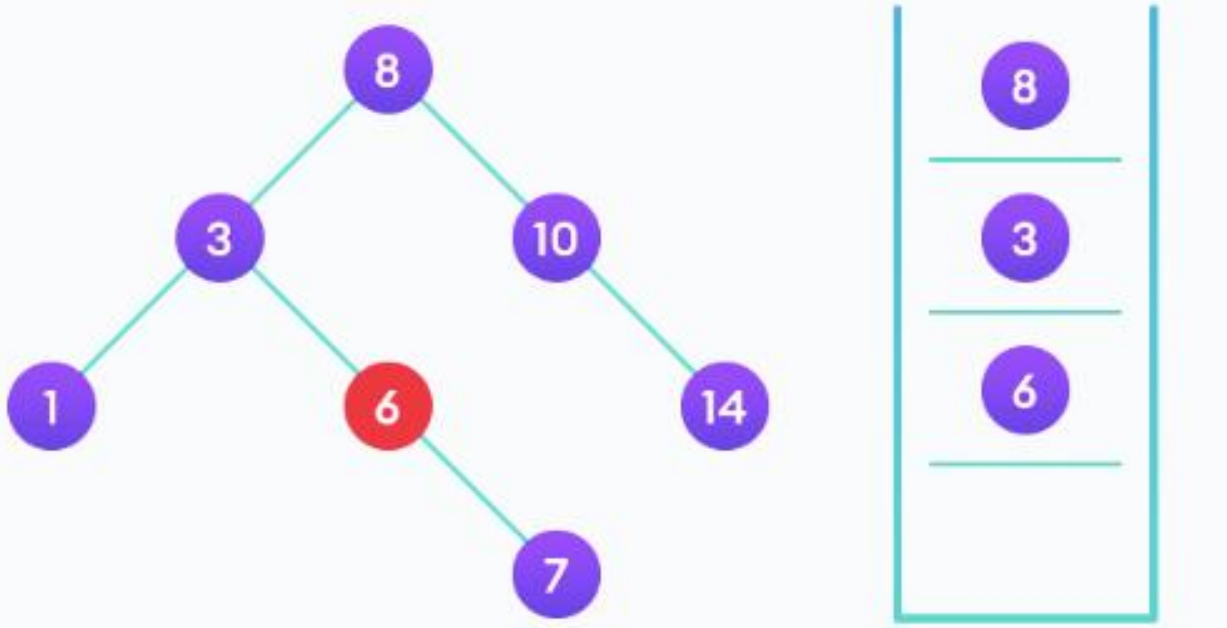
If we use recursion here, their positions will be retained as they are without any changes/updates

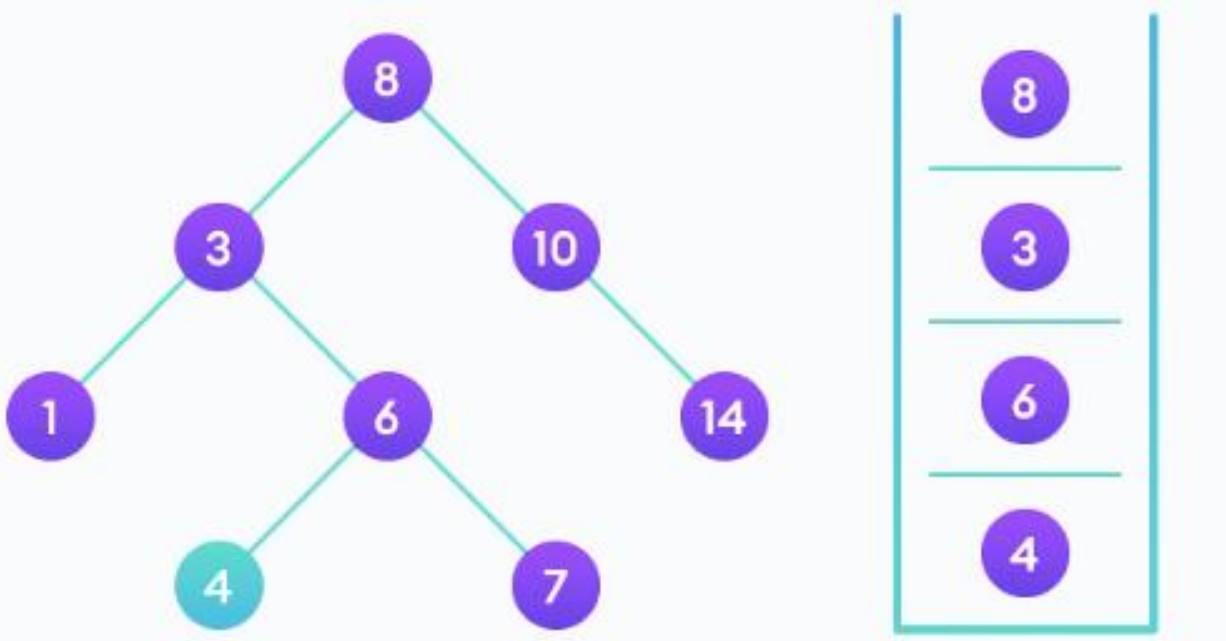


If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

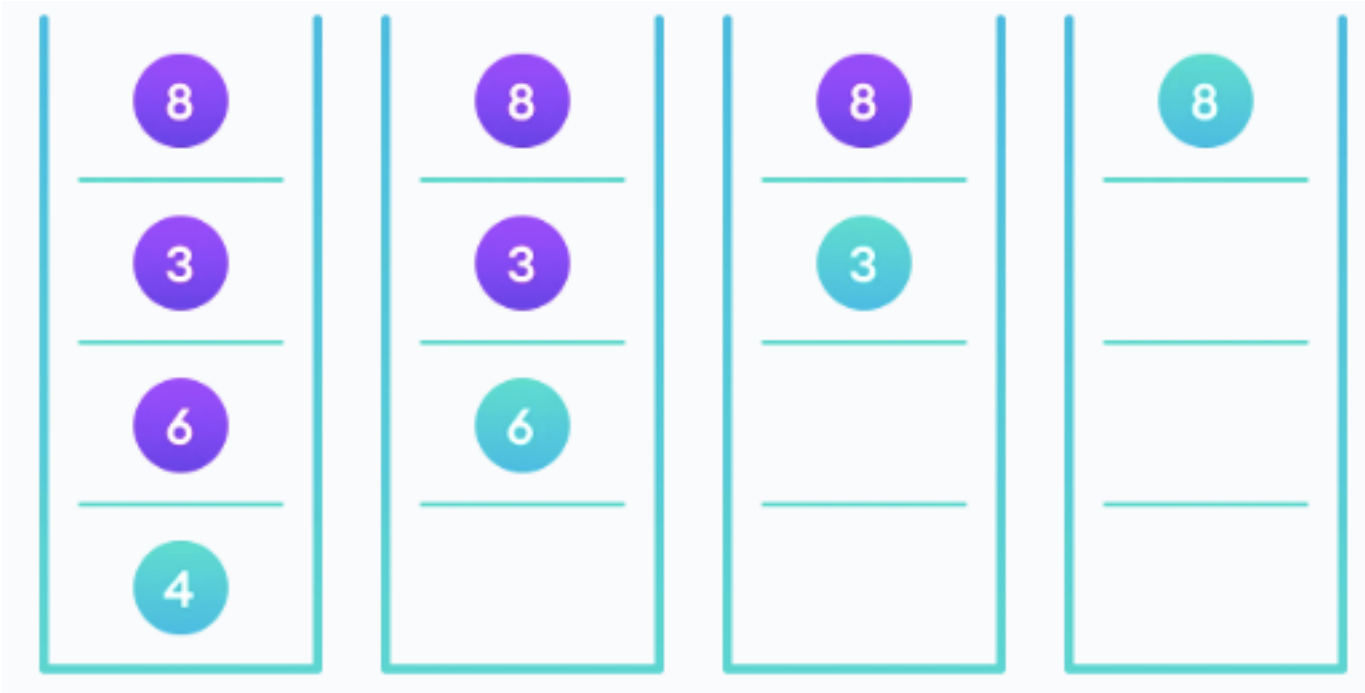

8
3
10
1
6
14
7
8


8
3
10
1
6
14
7
8
3


8
3
10
1
6
14
7
8
3
6


8
3
10
1
6
14
4
7
8
3
6
4

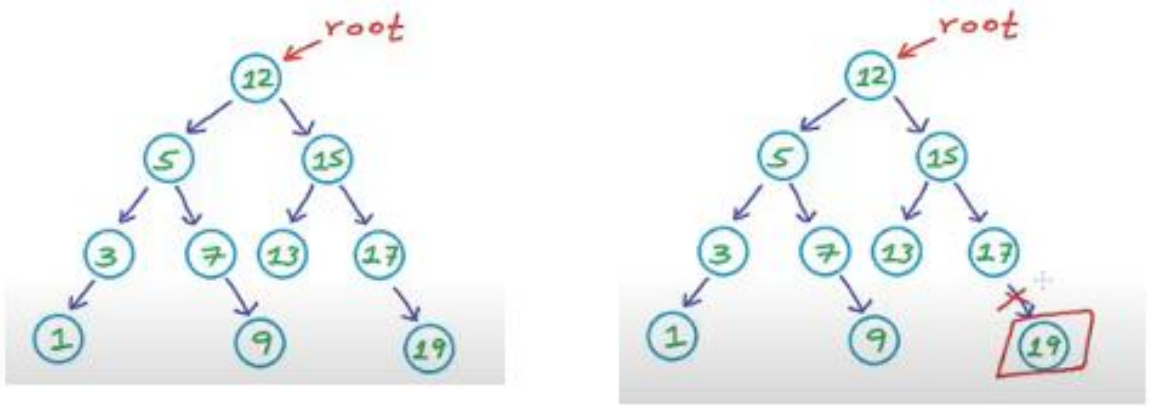
If we use recursion here, their positions will be retained as they are without any changes/updates



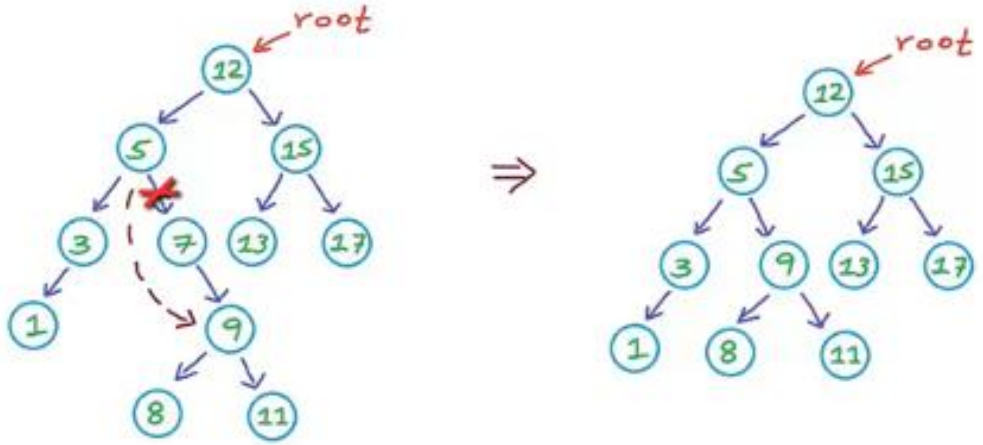
DELETION

- There are three cases for deleting a node from a binary search tree.

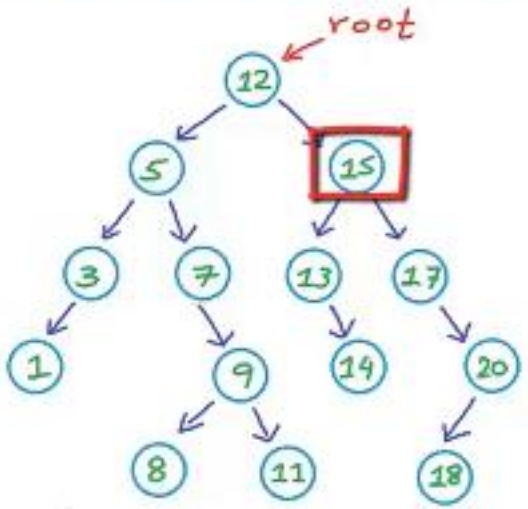
Case 1: Deletion of Leaf Nodes (No child)



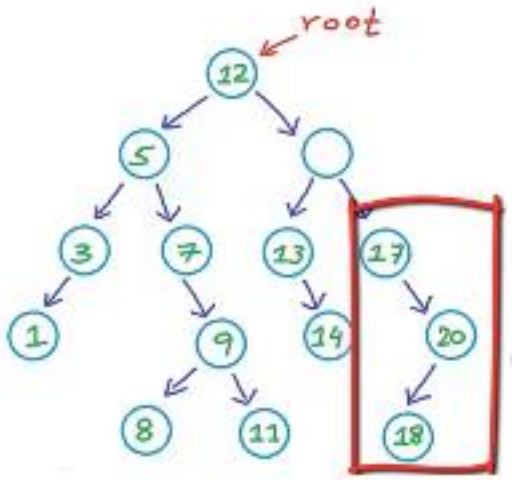
Case 2: Deletion of Nodes with one child

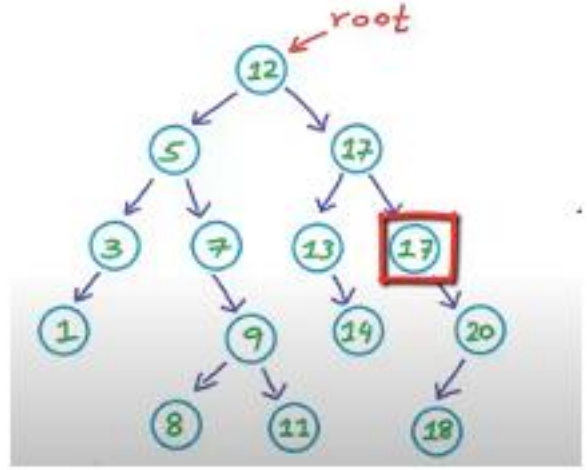


Case 3: Deletion of Nodes with two children

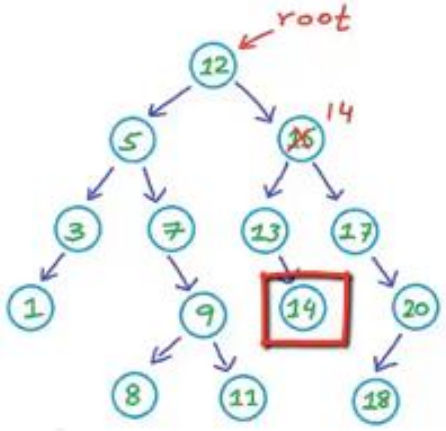


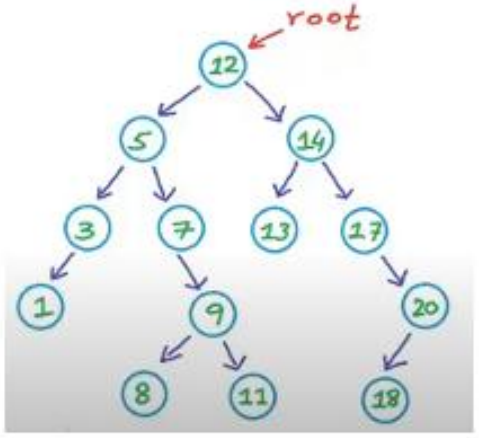
Option 1: Find minimum in the right sub tree

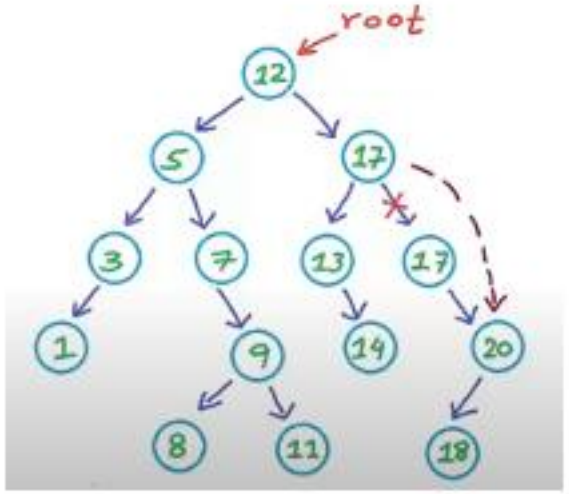


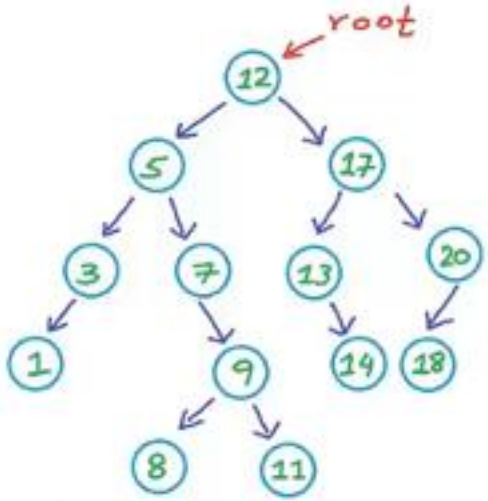


Option 2: Find the maximum in the left sub tree









```
import java.util.*;

public class BSTree

{
        static BTreeNode insert(BTreeNode root,int data)

        {
                if(root==null)

                {
                        BTreeNode btnode=new BTreeNode();

                        btnode.data=data;

                        btnode.left=null;

                        btnode.right=null;

                        root=btnode;
                        return root;
                }
                if(data<=root.data)
                {
                        root.left=insert(root.left,data);
                }
                else if(data>root.data)
                {
                        root.right=insert(root.right,data);
                }
                return root;
        }
```

```
static BTreeNode deleteNode(BTreeNode root,int data)
        {
                if(root==null)
                return null;
                        if(root.data==data)
                        {
                        if(root.right==null && root.left==null)
                        return null;
                        if(root.left==null && root.right!=null)
                        return root.right;
                        if(root.right==null && root.left !=null)
                        return root.left;
                        BTreeNode temp=successor(root.right,root);
                        root.data=temp.data;
                        }
                        else if(data<=root.data)
                        {
                        root.left=deleteNode(root.left,data);
                        }
                        else
                        {
                        root.right=deleteNode(root.right,data);
                        }
        return root;
        }
```

```
static BTreeNode successor(BTreeNode root,BTreeNode parent)
        {
                BTreeNode pre=parent,cur=root;
                        while(cur.left!=null)
                        {
                                pre=cur;
                                cur=cur.left;
                        }
                        deleteNode(pre,cur.data);
                return cur;
        }
        static void inorder(BTreeNode root)
        {
                if(root==null)
                        return;
                inorder(root.left);
                System.out.print(root.data+" ");
                inorder(root.right);
        }
```

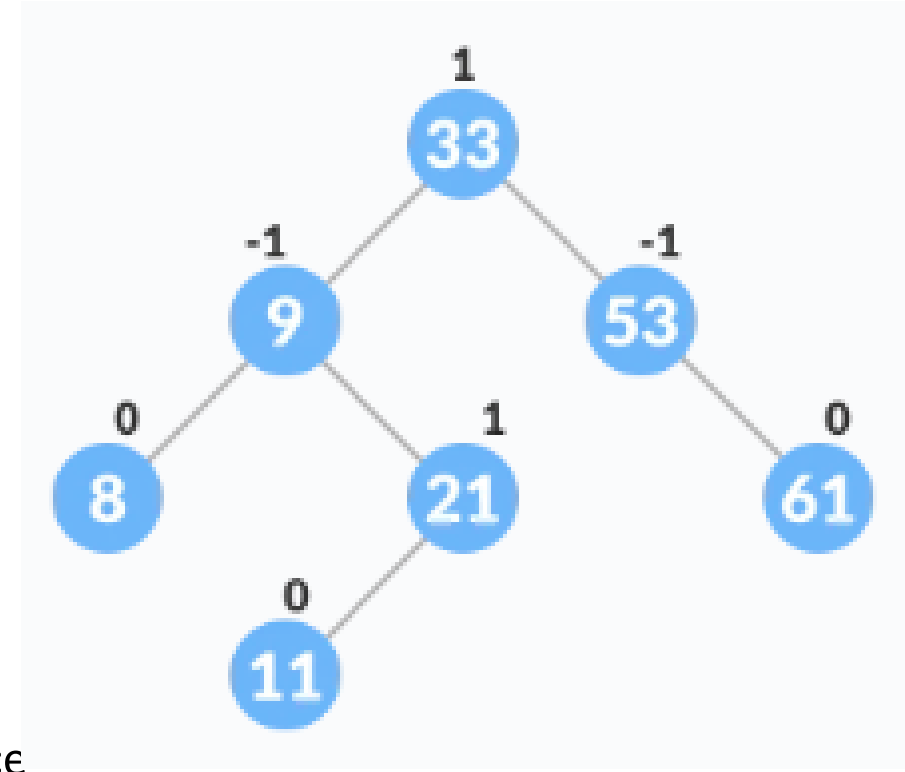
```
public static void main(String args[])
        {
                BTreeNode root=null;
                root=insert(root,8);
                System.out.println(root);
                root=insert(root,9);
                root=insert(root,23);
                root=insert(root,4);
                root=insert(root,12);
                root=insert(root,11);
                root=insert(root,13);
                root=insert(root,10);
                System.out.println("\nInorder");
                inorder(root);
                root=deleteNode(root,4);
                System.out.println("\nInorder");
                inorder(root);
                root=deleteNode(root ,9);
                System.out.println("\nInorder");
                inorder(root);
                root=deleteNode(root,12);
                System.out.println("\nInorder");
                inorder(root);
                System.out.println(root);
                root=deleteNode(root,8);
                System.out.println("\nInorder");
                inorder(root);
                System.out.println(root);

        }

}
```

AVL TREES

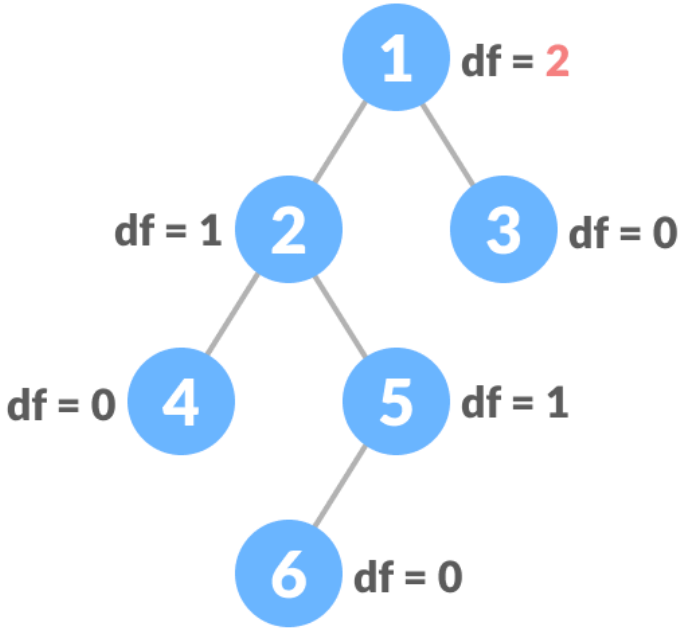
- AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.
- AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.



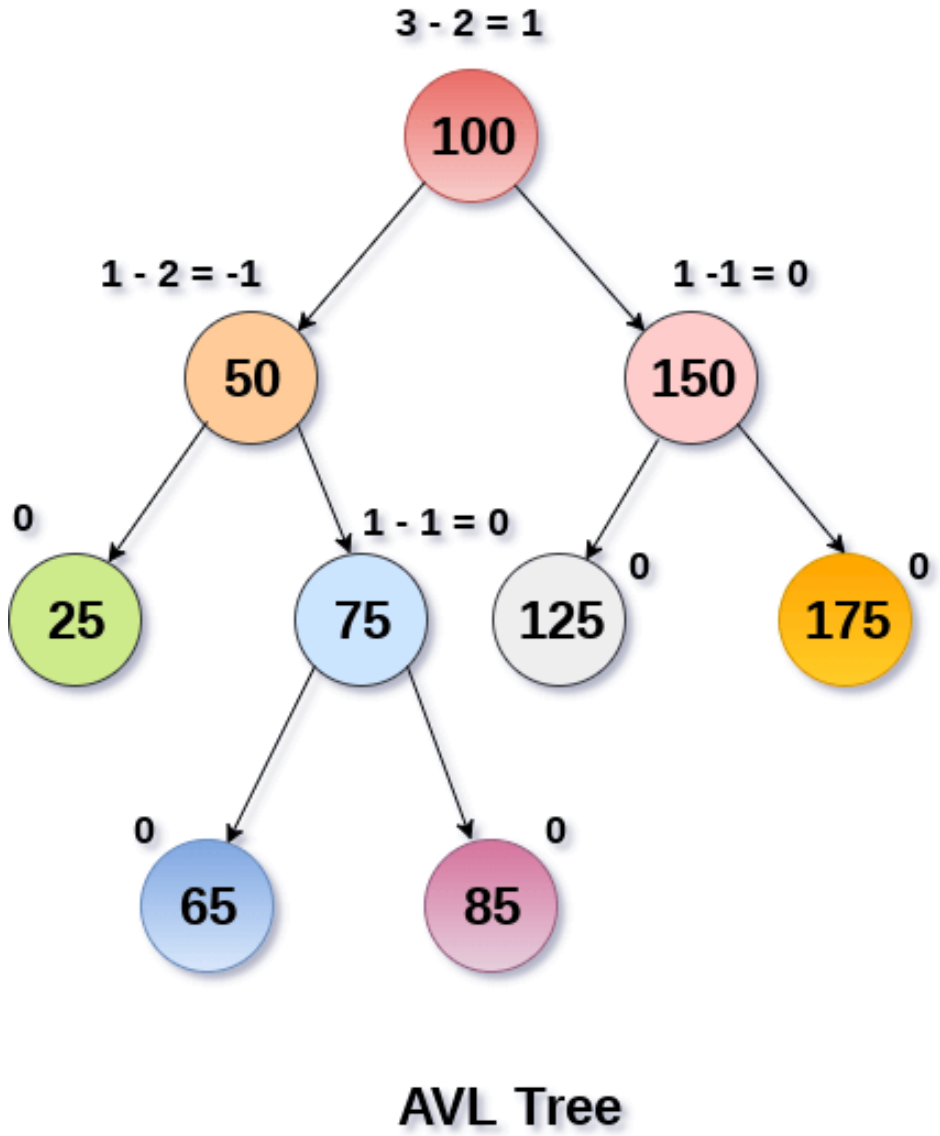
Balance Factor

- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.
- Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)
- The self balancing property of an AVL tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

Example – Height calculation of each nodes: Not a BST & It is not balanced = Not a AVL Tree



df = |height of left child - height of right child|



AVL Tree

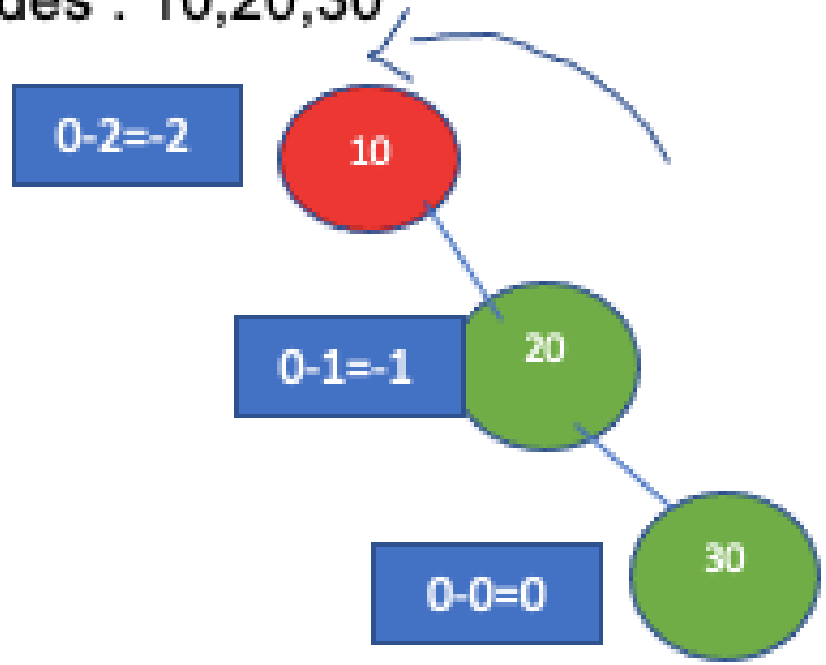
OPERATIONS

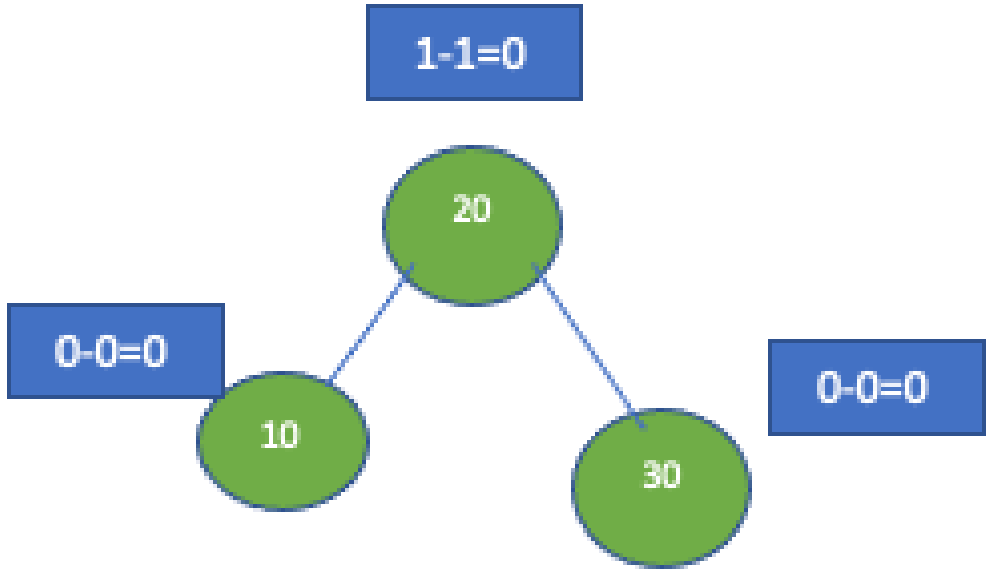
- Rotation
- Insertion
- Deletion

ROTATION

Case 1 : Left Rotation

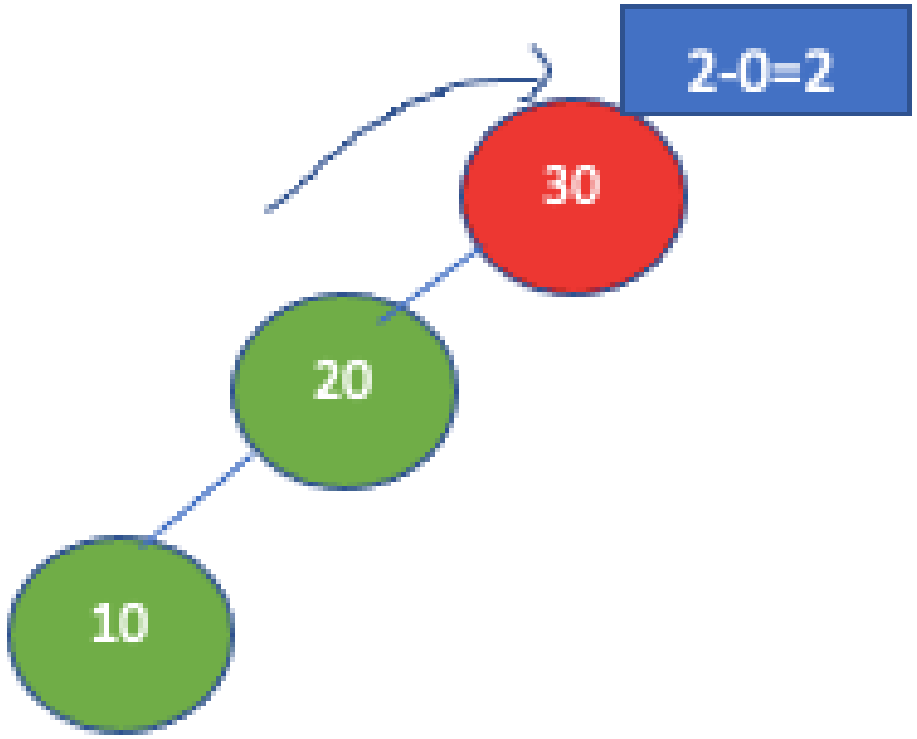
Nodes : 10,20,30

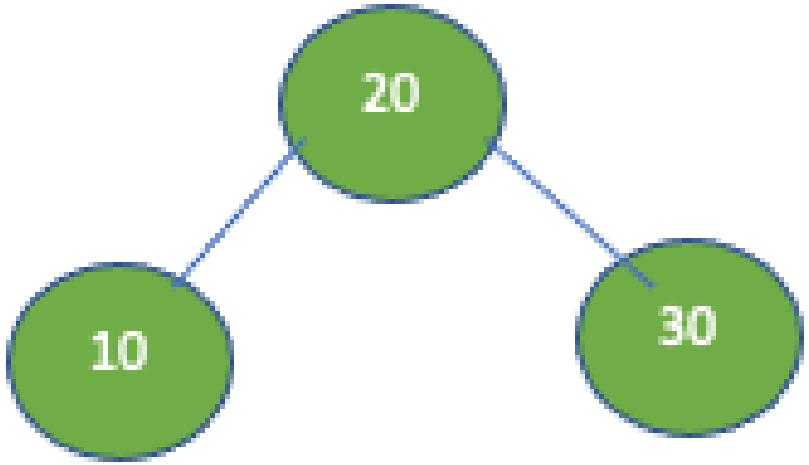




Case 2 : Right Rotation

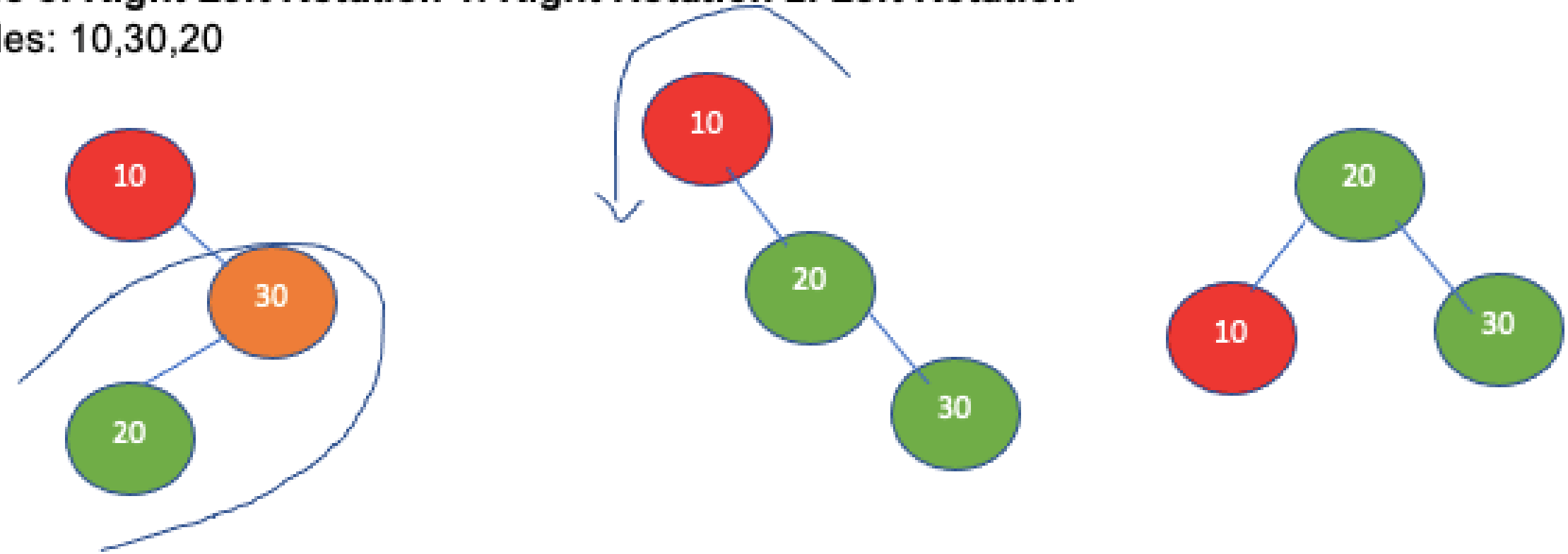
Nodes :30,20,10





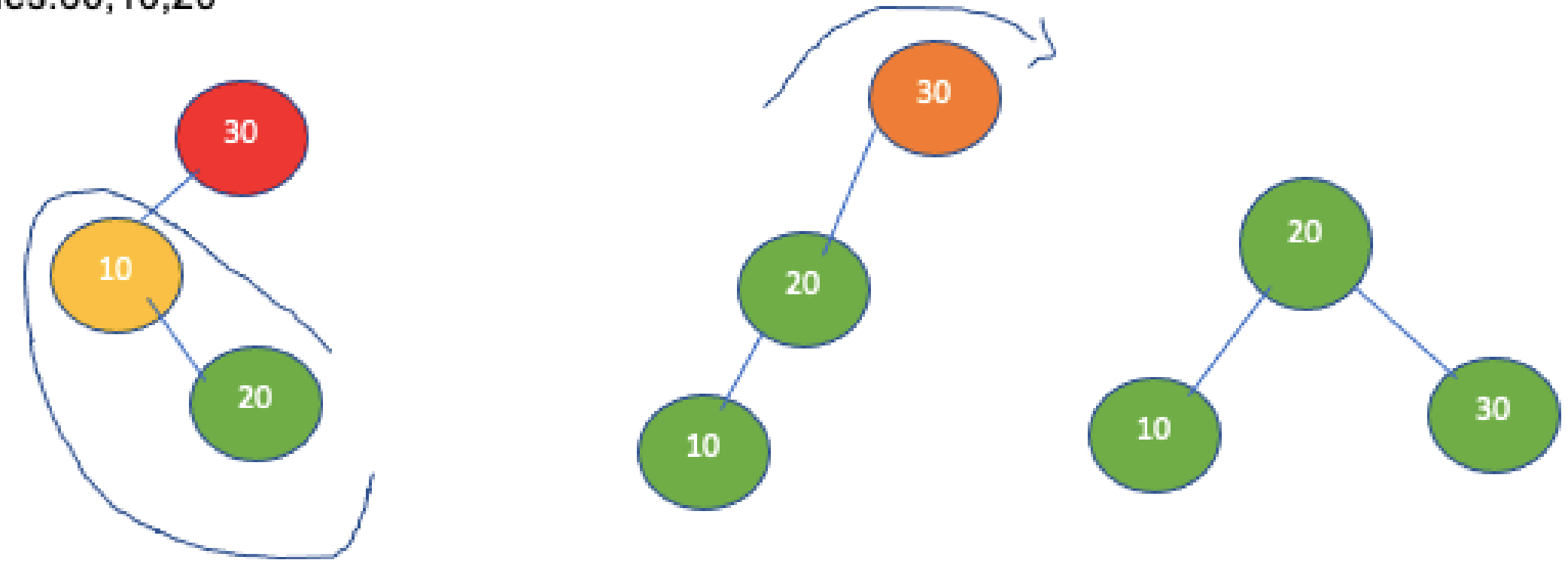
Case 3: Right Left Rotation 1. Right Rotation 2. Left Rotation

Nodes: 10,30,20

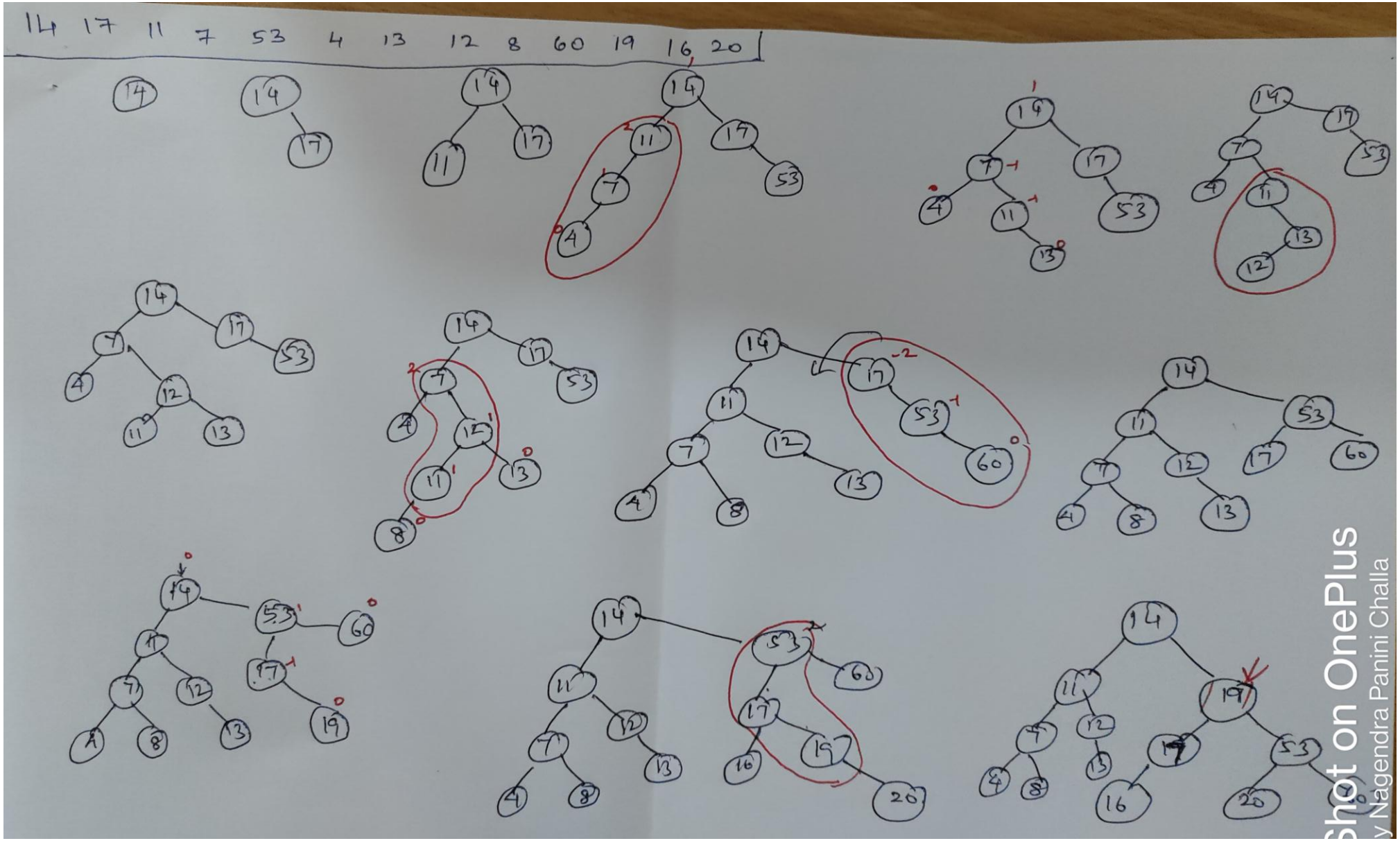


Case 4: Left Right Rotation 1.Left rotation 2. Right rotation 10 20 30

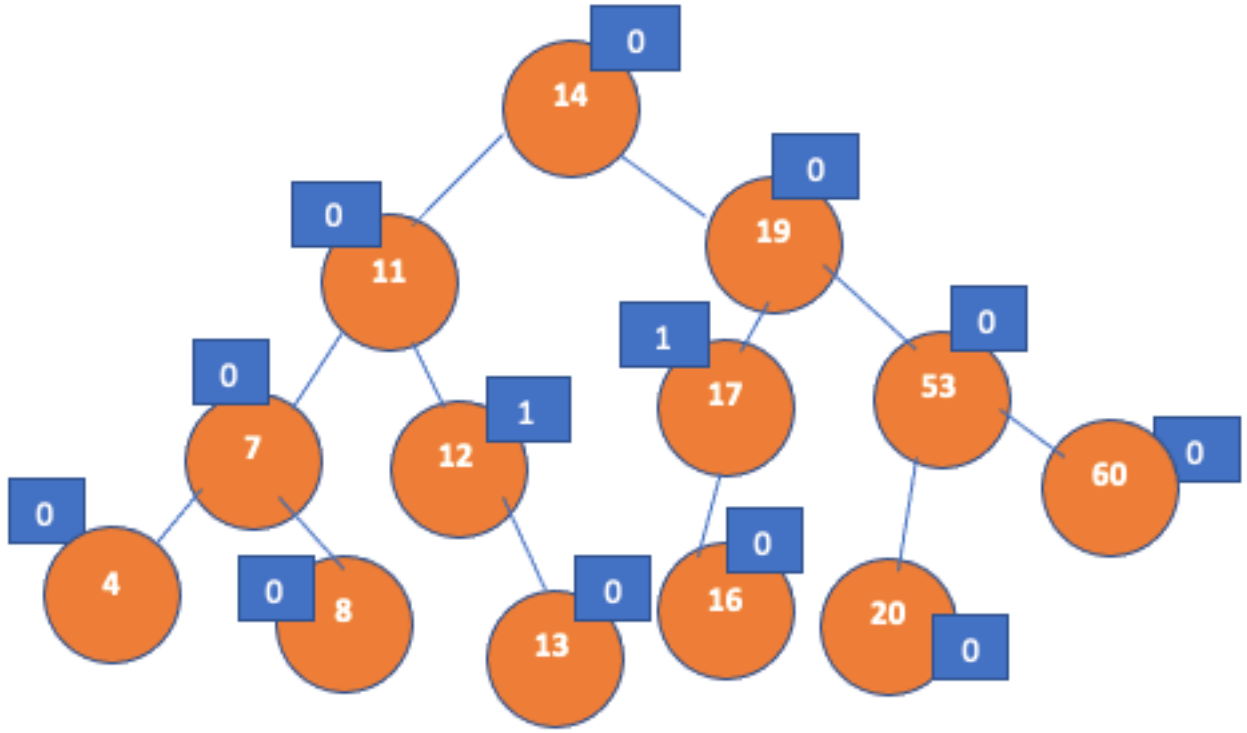
Nodes:30,10,20



INSERTION

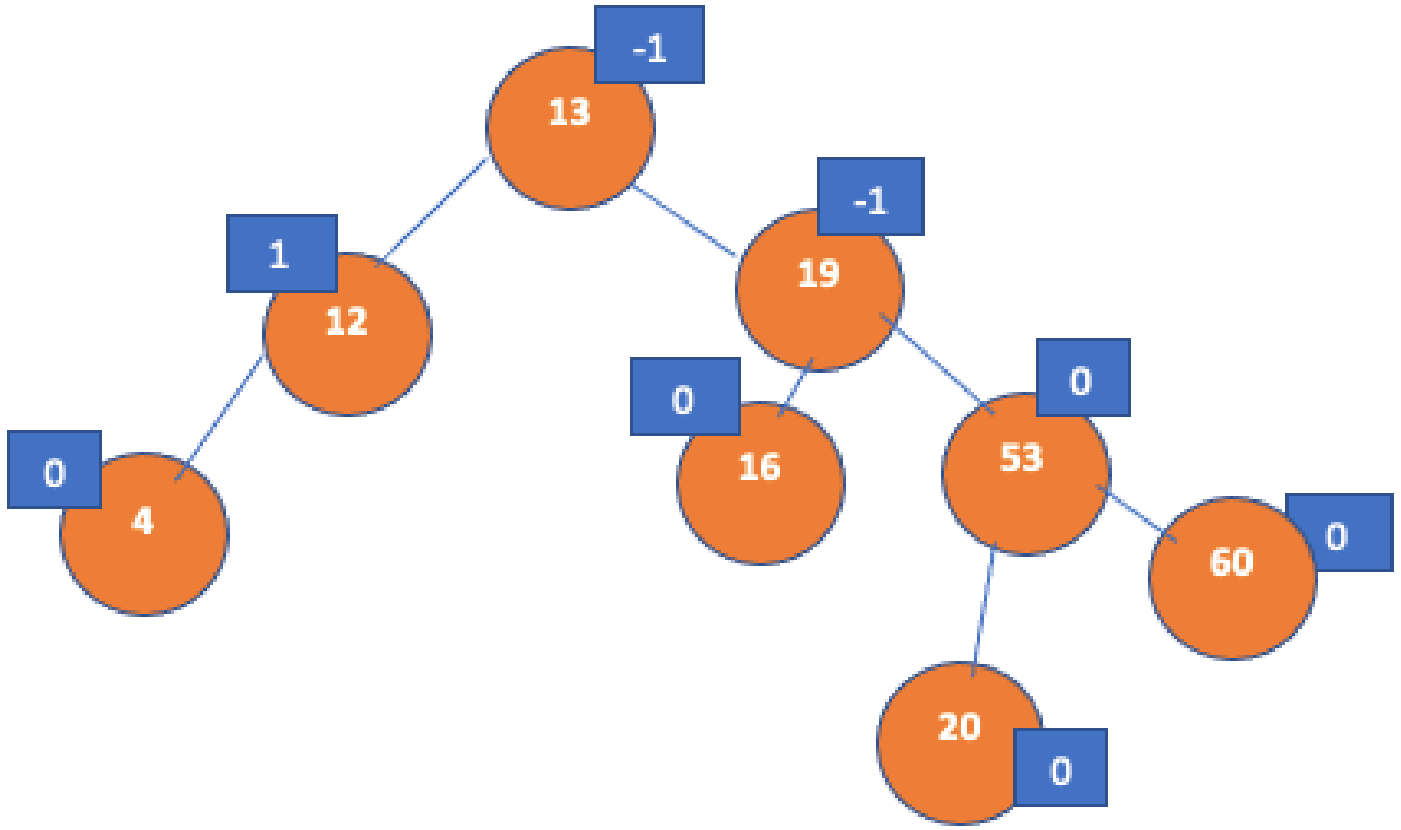

14 17 11 7 53 4 13 12 8 60 19 16 20
Shot on OnePlus
by Nagendra Panini Challa

Nodes: 14,17,11,7,53,4,13,12,8,60,19,16,20



DELETION

Nodes: 8,7,11,14,17



```
// AVL tree implementation in Java
// Create node
class Node
{
int item, height;
Node left, right;
Node(int d)
{
item = d;
height = 1;
} }
// Tree class
class AVLTree
{ Node root;
int height(Node N)
{ if (N == null)
return 0;
return N.height;
}
int max(int a, int b)
{ return (a > b) ? a : b;
}
```

```
Node rightRotate(Node y)
{ Node x = y.left;
Node T2 = x.right;
x.right = y;
y.left = T2;
y.height = max(height(y.left), height(y.right)) + 1;
x.height = max(height(x.left), height(x.right)) + 1;
return x;
} Node leftRotate(Node x)
{
Node y = x.right;
Node T2 = y.left;
y.left = x;
x.right = T2;
x.height = max(height(x.left), height(x.right)) + 1;
y.height = max(height(y.left), height(y.right)) + 1;
return y;
}
```

```
// Get balance factor of a node
int getBalanceFactor(Node N)
{ if (N == null)
return 0;
return height(N.left) - height(N.right);
} // Insert a node
Node insertNode(Node node, int item)
{
// Find the position and insert the node
if (node == null)
return (new Node(item));
if (item < node.item)
node.left = insertNode(node.left, item);
else if (item > node.item)
node.right = insertNode(node.right, item);
else
return node;
```

```
// Update the balance factor of each node
// And, balance the tree
node.height = 1 + max(height(node.left), height(node.right));
int balanceFactor = getBalanceFactor(node);
if (balanceFactor > 1)
{ if (item < node.left.item)
{ return rightRotate(node);
} else if (item > node.left.item)
{ node.left = leftRotate(node.left);
return rightRotate(node);
} }
if (balanceFactor < -1)
{ if (item > node.right.item)
{ return leftRotate(node);
} else if (item < node.right.item)
{ node.right = rightRotate(node.right);
return leftRotate(node);
} } return node;
} Node nodeWithMimumValue(Node node)
{ Node current = node;
while (current.left != null)
current = current.left;
return current;
}
```

```
// Delete a node
Node deleteNode(Node root, int item)
{
// Find the node to be deleted and remove it
if (root == null)
return root;
if (item < root.item)
root.left = deleteNode(root.left, item);
else if (item > root.item)
root.right = deleteNode(root.right, item);
else {
if ((root.left == null) || (root.right == null))
{ Node temp = null;
if (temp == root.left)
temp = root.right;
else temp = root.left;
if (temp == null)
{ temp = root;
root = null; }
else root = temp; }
else {
Node temp = nodeWithMimumValue(root.right);
root.item = temp.item;
root.right = deleteNode(root.right, temp.item);
} }
if (root == null)
return root;
```

```
// Update the balance factor of each node and balance the tree

root.height = max(height(root.left), height(root.right)) + 1;
int balanceFactor = getBalanceFactor(root);
if (balanceFactor > 1)
{
if (getBalanceFactor(root.left) >= 0)
{ return rightRotate(root);
} else {
root.left = leftRotate(root.left);
return rightRotate(root);
} }
if (balanceFactor < -1)
{ if (getBalanceFactor(root.right) <= 0)
{ return leftRotate(root);
} else {
root.right = rightRotate(root.right);
return leftRotate(root);
} } return root; }
void preOrder(Node node)
{ if (node != null)
{ System.out.print(node.item + " ");
preOrder(node.left);
preOrder(node.right); } }
```

```
// Print the tree
private void printTree(Node currPtr, String indent, boolean last)
{
if (currPtr != null)
{ System.out.print(indent);
if (last)
{
System.out.print("R----");
indent += " ";
} else {
System.out.print("L----");
indent += "|  ";
}
System.out.println(currPtr.item);
printTree(currPtr.left, indent, false);
printTree(currPtr.right, indent, true);
} }
```

```
// Main Function

public static void main(String[] args) { AVLTree tree =
new AVLTree(); tree.root = tree.insertNode(tree.root,
33); tree.root = tree.insertNode(tree.root, 13); tree.root =
tree.insertNode(tree.root, 53); tree.root =
tree.insertNode(tree.root, 9); tree.root =
tree.insertNode(tree.root, 21); tree.root =
tree.insertNode(tree.root, 61); tree.root =
tree.insertNode(tree.root, 8); tree.root =
tree.insertNode(tree.root, 11);
tree.printTree(tree.root, "", true);
tree.root = tree.deleteNode(tree.root, 13);
System.out.println("After Deletion: ");
tree.printTree(tree.root, "", true); } }
```