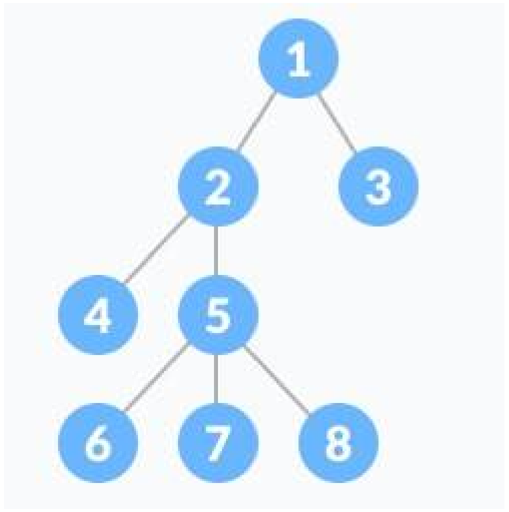
# DATA STRUCTURES –Trees & Hashing

UNIT-4&5
Dr. SELVA KUMAR S
ASSISTANT PROFESSOR
B.M.S. COLLEGE OF ENGINEERING

## Trees -Introduction

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.



## Properties of Trees

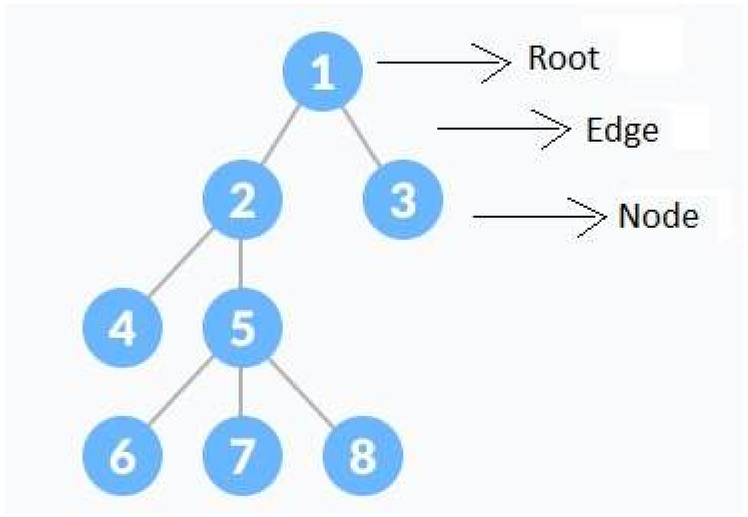
- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has n-1 edges.
- A graph is a tree if and if only if it is minimally connected.
- Any connected graph with n vertices and n-1 edges is a tree.

# Tree Applications

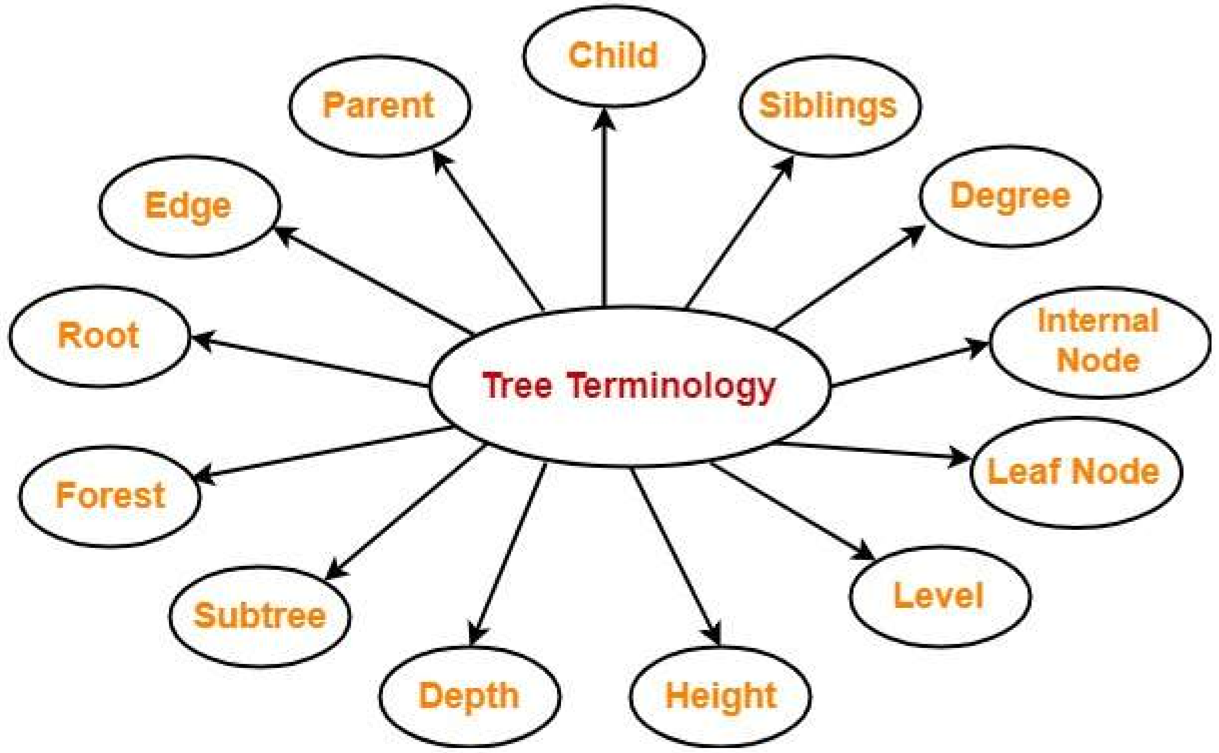
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

# Tree terminologies

- Node-A nodeisanentitythatcontainsakeyorvalueandpointers to its child nodes.
  - The last nodes of each path are calledleaf nodes or external nodesthat do not contain a link/pointer to child nodes.
  - The node having at least a child node is called aninternal node.
- Edge -It is the link between any two nodes.
- Root - It is the topmost node of a tree.

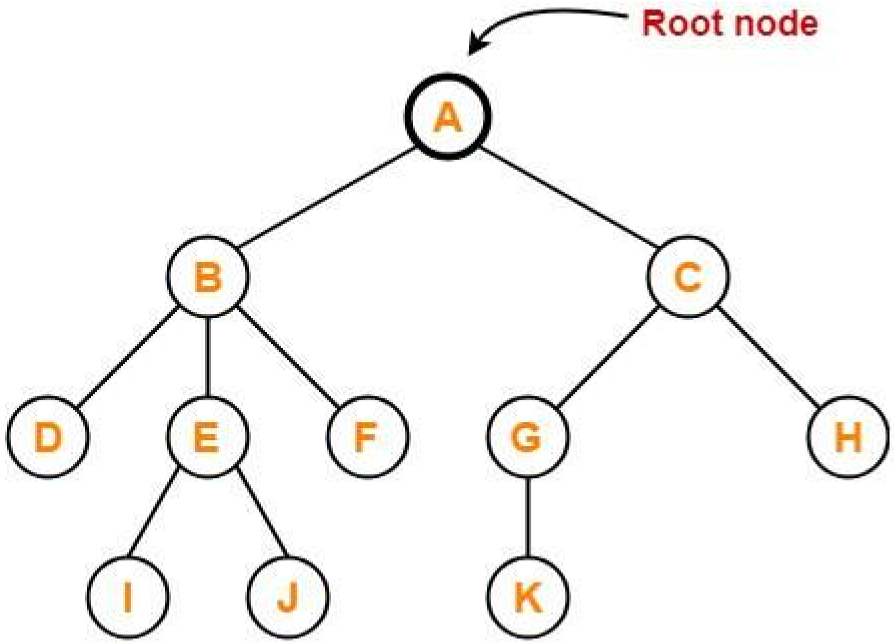


# Tree terminologies



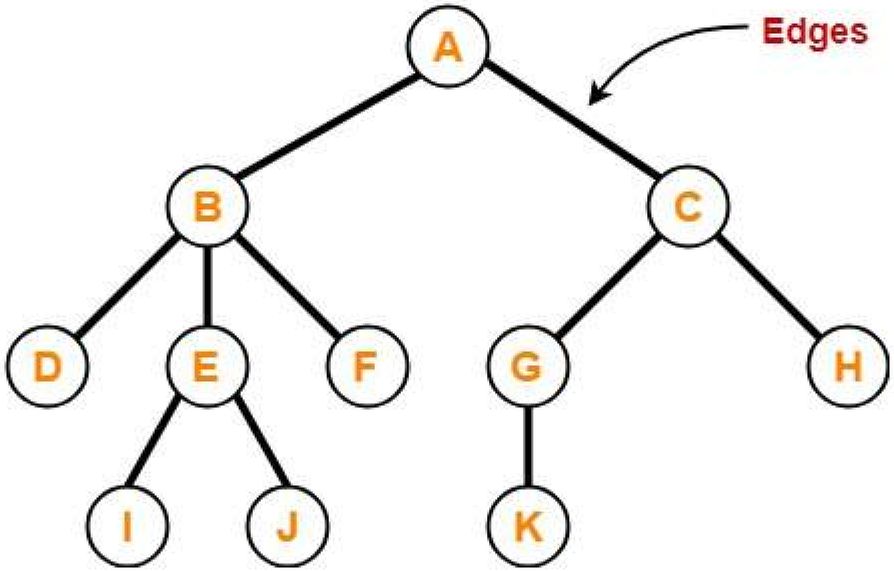
# Root

- The first node from where the tree originates is called as aroot node.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



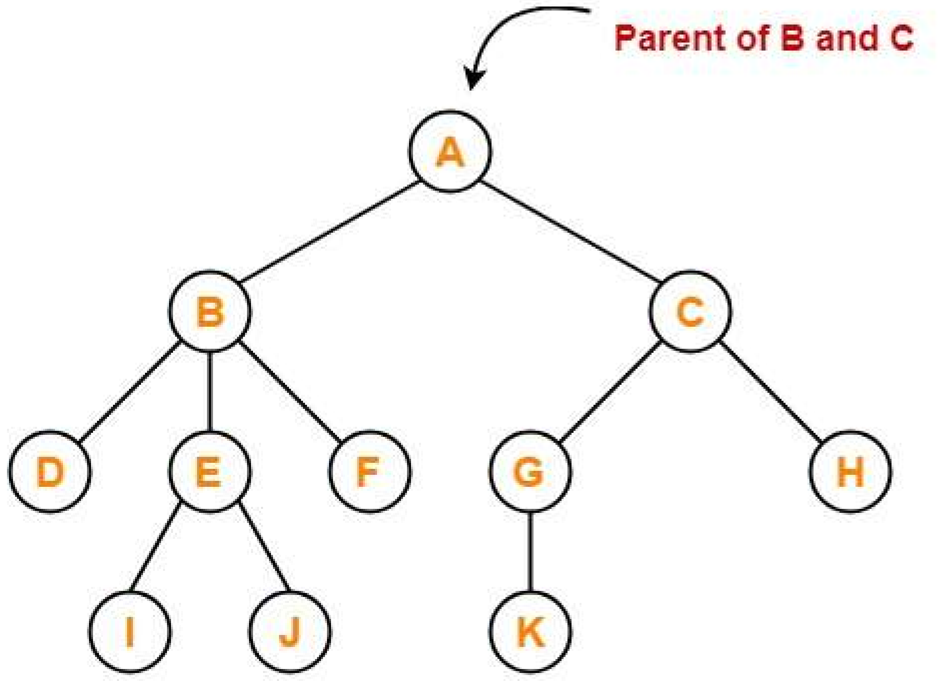
# Edge

- The connecting link betweenany two nodes is called as an edge.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.



# Parent

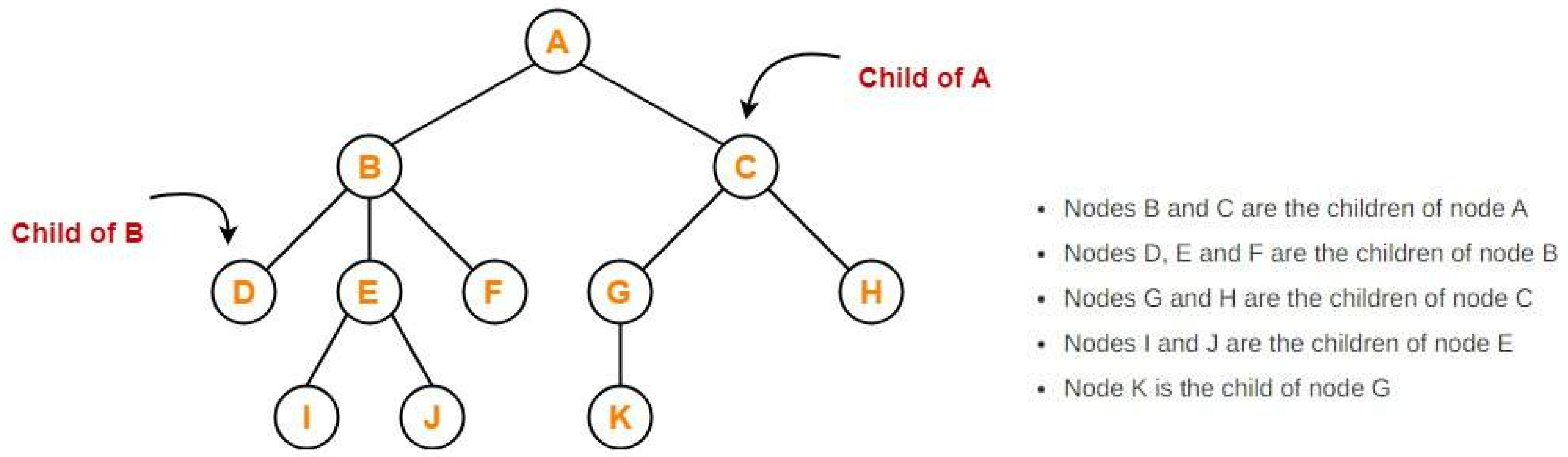
- The node which has a branchfromittoanyother node is called as aparent node.
- Inotherwords,thenodewhichhasoneormorechildreniscalledas a parent node.
- In a tree, a parent node can have any number of child nodes.



- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

## Child

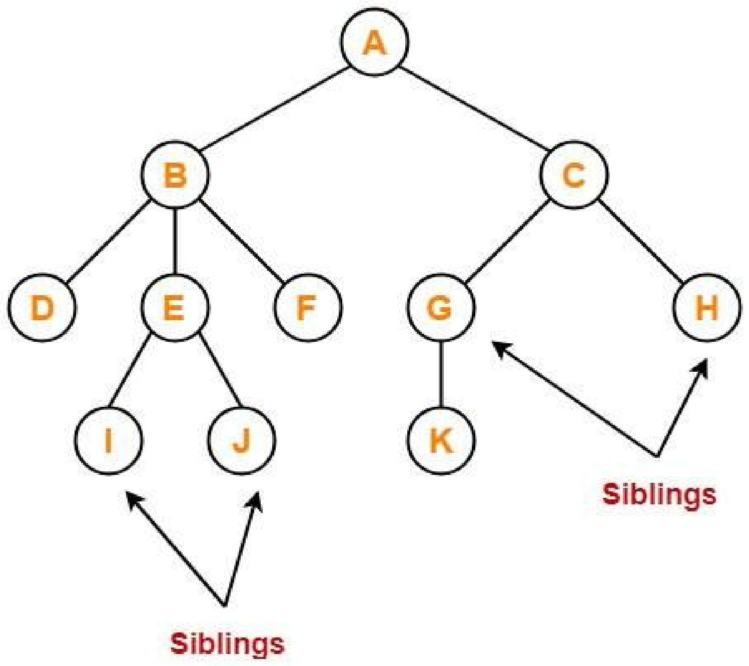
- The node which is a descendant of some node is called as achild node.
- All the nodes except root node are child nodes.



- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

## Siblings

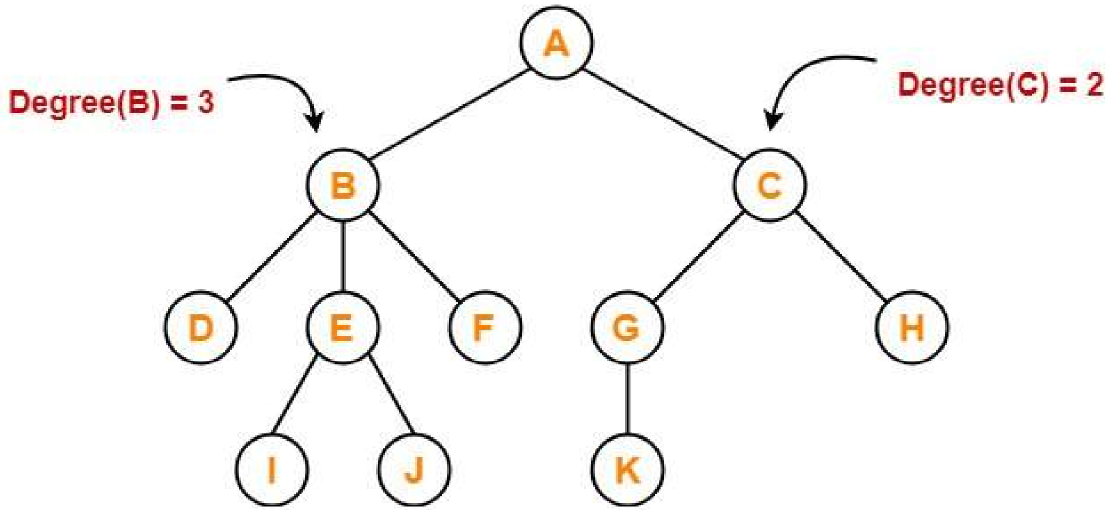
- Nodes which belongtothesameparent are called assiblings.
- In other words, nodes with the same parent are sibling nodes.



- 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

# Degree

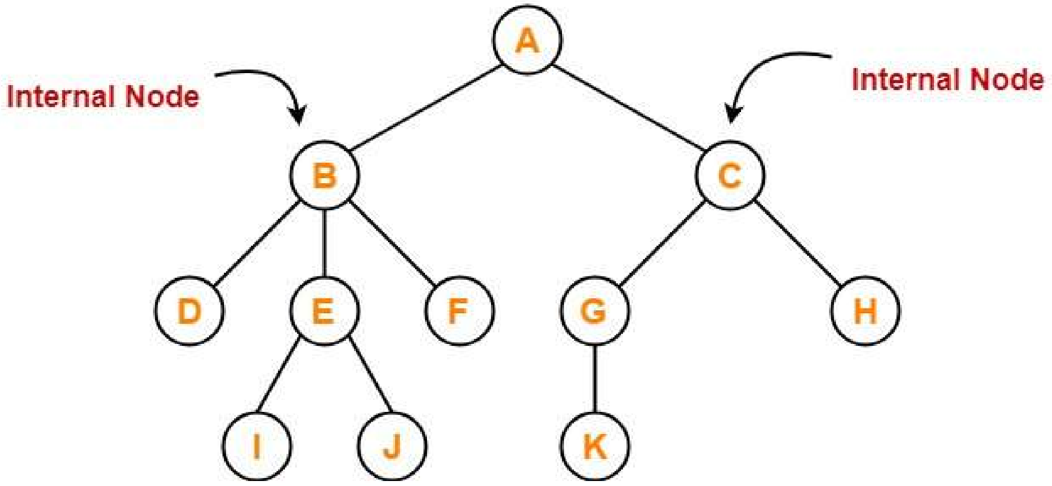
- Degree of a nodeisthetotalnumberof children of that node.
- Degree of a treeis the highest degree of a node among all the nodes in the tree.



- 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

# Internal node

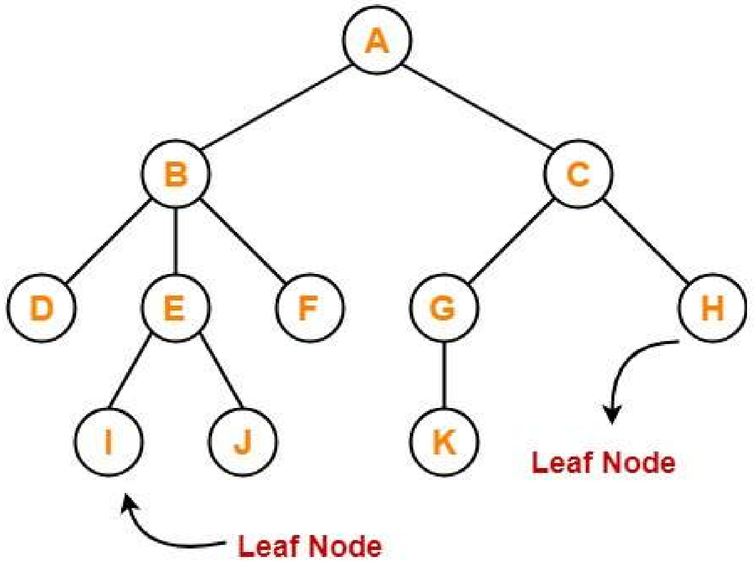
- The node whichhasat leastonechildiscalled as aninternal node.
- Internal nodes are also called asnon-terminal nodes.
- Every non-leaf node is an internal node.



Here, nodes A, B, C, E and G are internal nodes.

# Leaf node

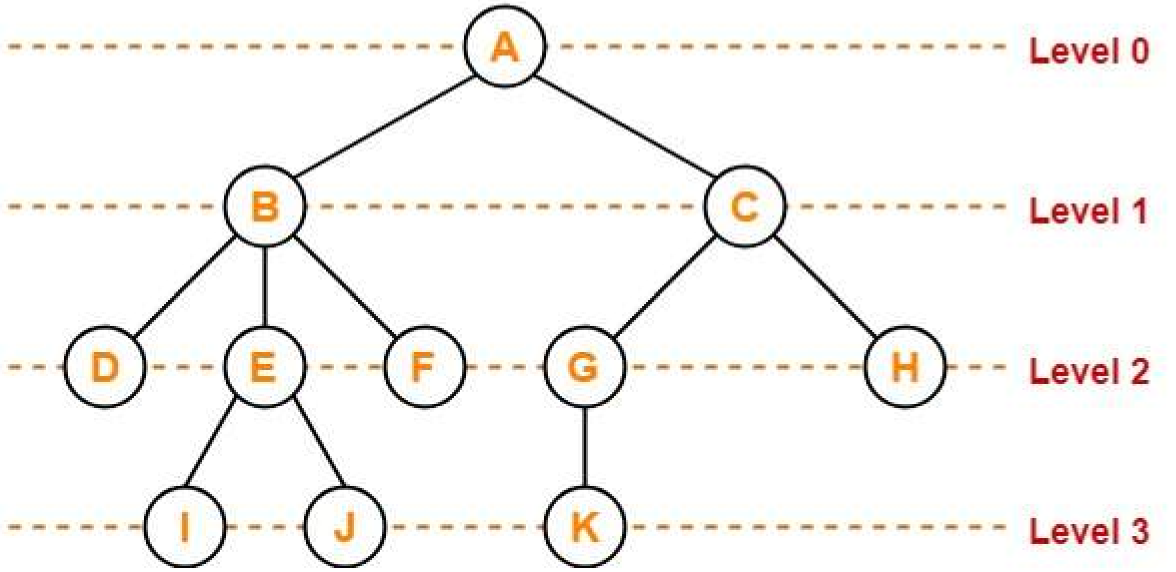
- The node which doesnot haveanychildis called as aleaf node.
- Leaf nodes are also called asexternal nodesorterminal nodes.



Here, nodes D, I, J, F, K and H are leaf nodes.

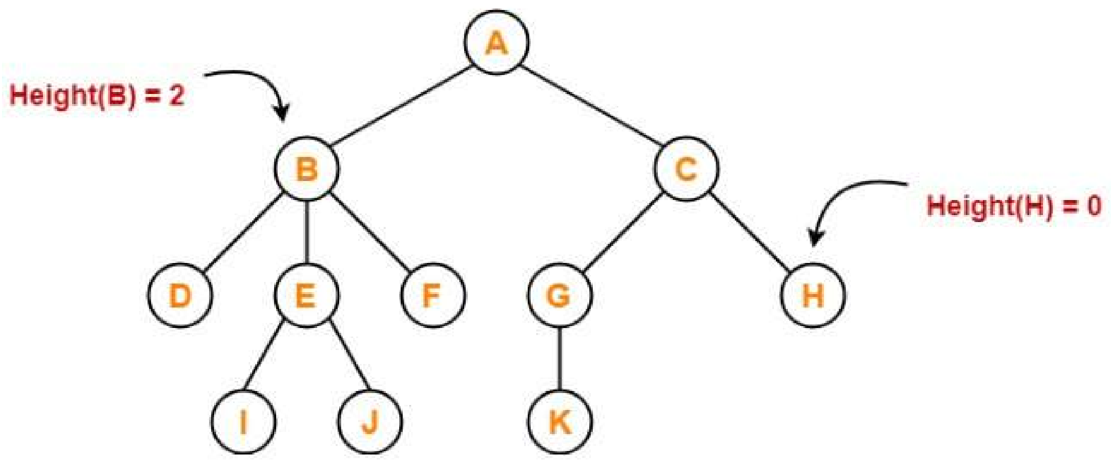
## Level

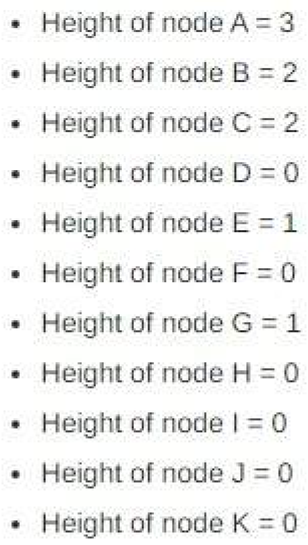
- In a tree, each step fromtop to bottom is called aslevel of a tree.
- The level count starts with 0 and increments by 1 at each level or step.



# Height

- Total number of edges that lieson the longest path from any leaf node to a particular node is called asheight of that node.
- Height of a treeis the height of root node.
- Height of all leaf nodes = 0

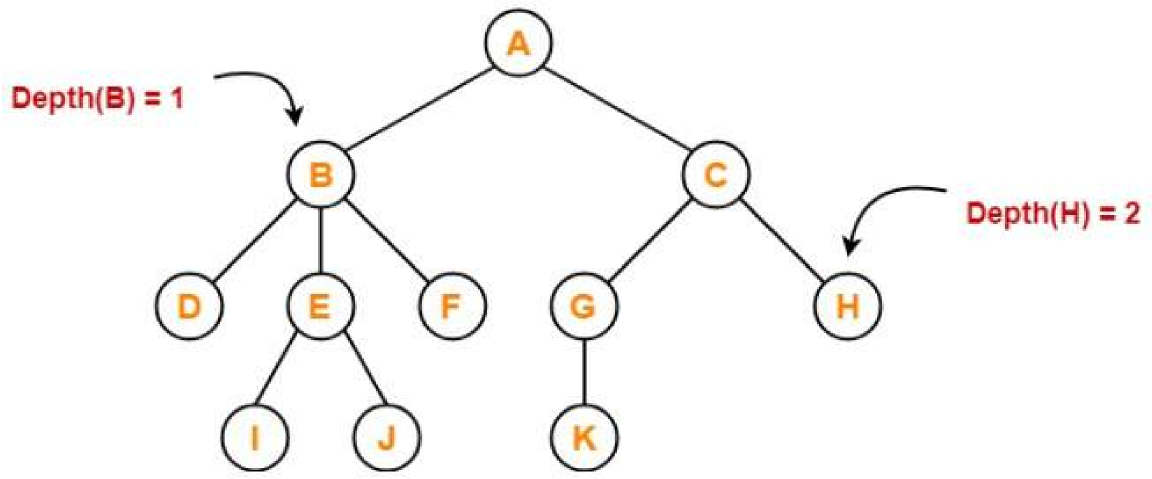




- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

# Depth

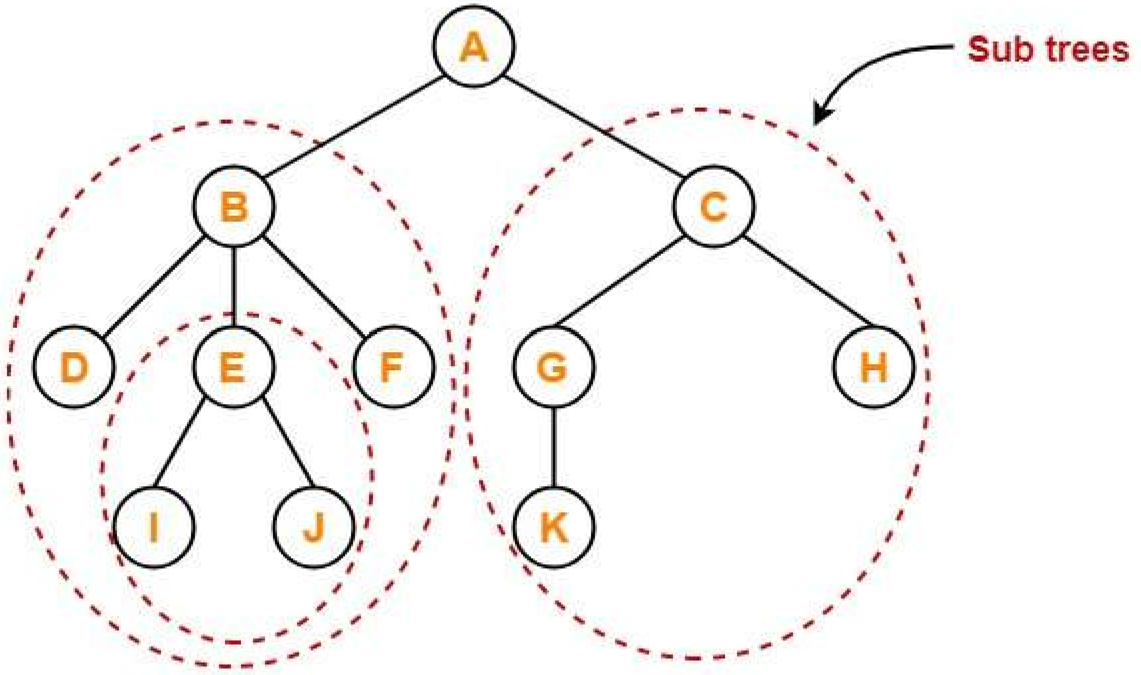
- Total number of edges from rootnode toaparticularnode is called asdepth of that node.
- Depth of a treeis the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



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

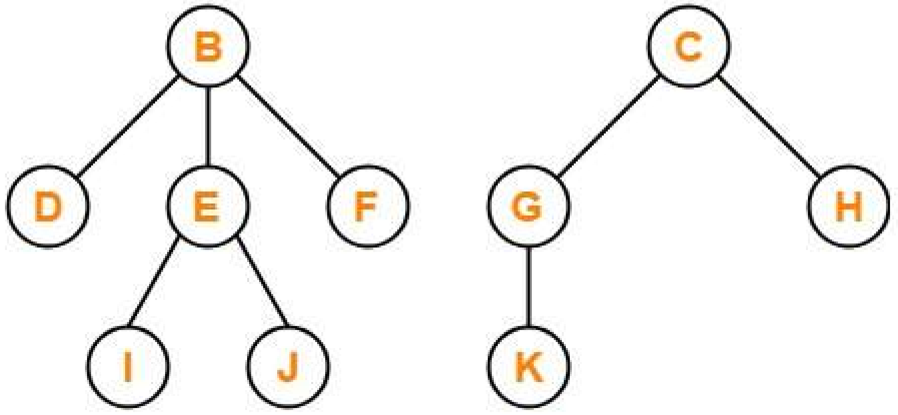
# Subtree

- In a tree, each child fromanodeformsasubtreerecursively.
- Every child node forms a subtree on its parent node.



## Forest

- A forest is a set of disjoint trees.



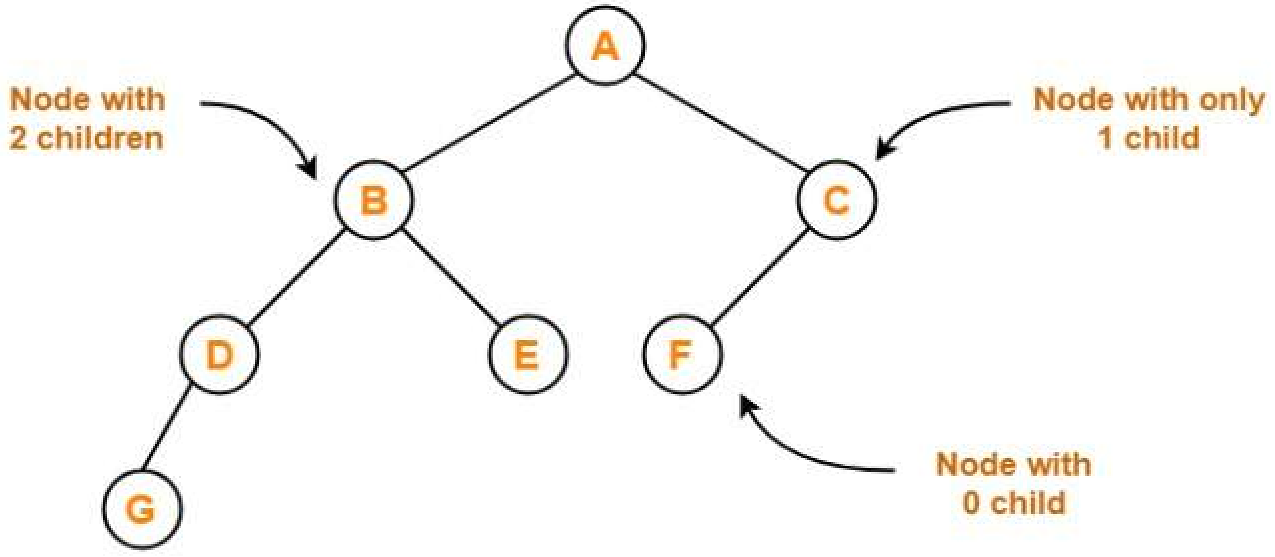
Forest

# Types of Tree

- General Tree
- Binary Tree
- Binary Search Tree
- AVL Tree
- Red-Black Tree
- N-aryTree

# Binary Tree

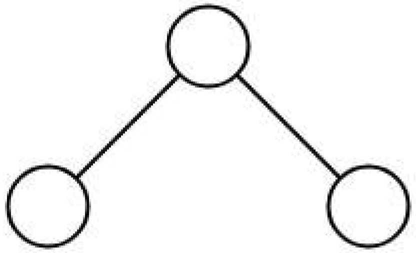
- Binary tree is a special tree data structurein which each node can have at most 2 children.
- Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.



Binary Tree Example

## Unlabeled Binary Tree

- Abinary tree is unlabeled if its nodes are not assigned any label.

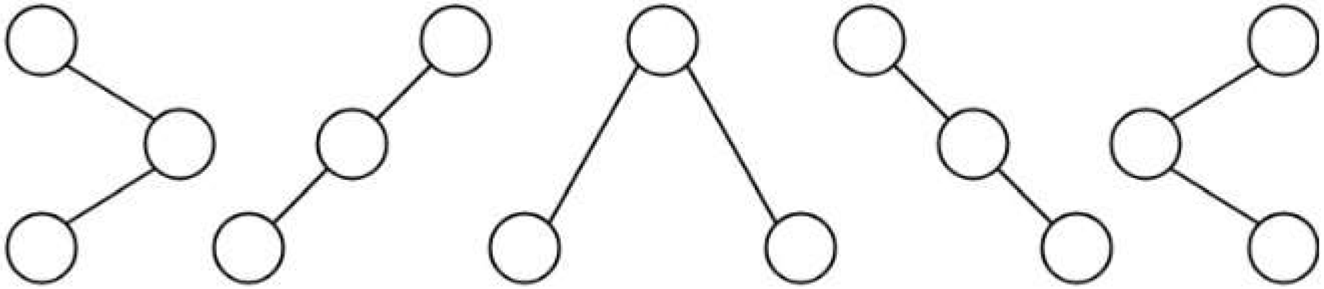


Unlabeled Binary Tree

Number of different Binary Trees possible with 'n' unlabeled nodes $= \dfrac{{}^{2n}C_n}{n+1}$

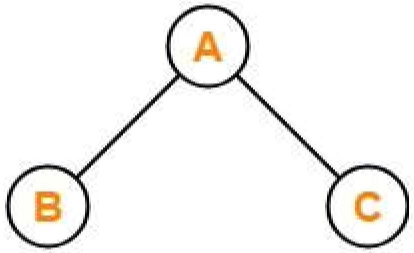
## Example

- Consider we want todrawallthebinarytrees possible
- Number of binary trees possible with 3 unlabeled nodes
- =2 x 3C3/ (3 + 1)
- =6C3/ 4
- = 5


Binary Trees Possible With 3 Unlabeled Nodes

# Labeled Binary Tree

- Abinary tree is labelled if all its nodes are assigned a label.

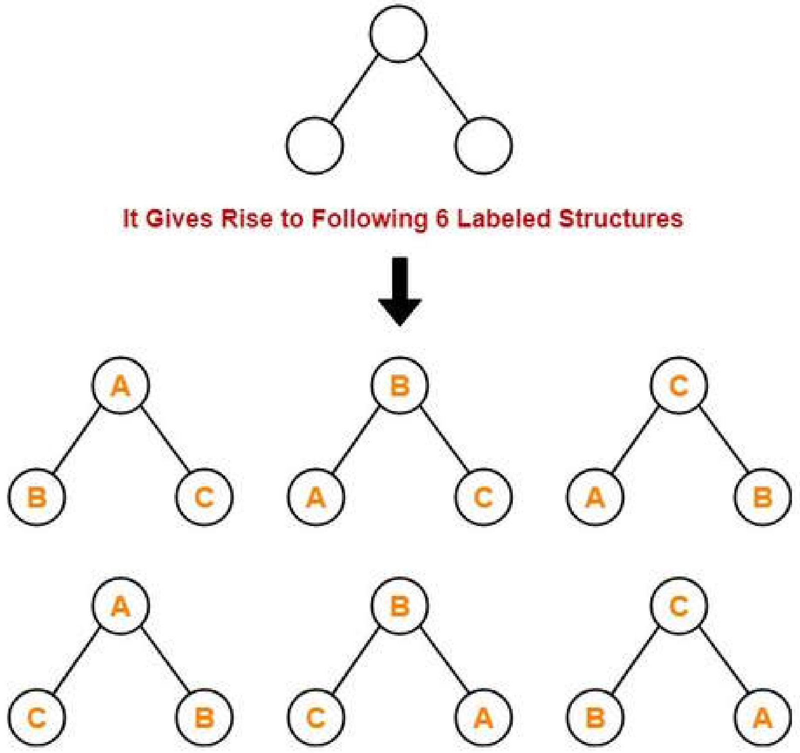


Labeled Binary Tree

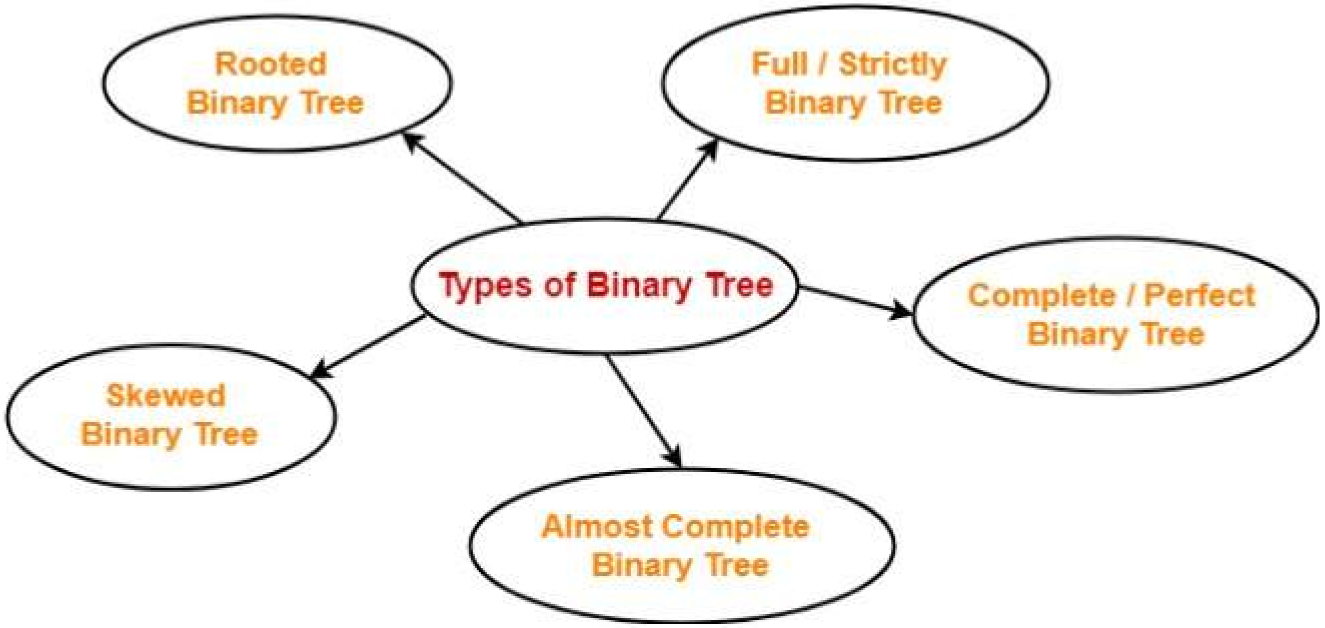
Number of different Binary Trees possible with 'n' labeled nodes $= \frac{{}^{2n}C_n}{n+1} \times n!$

# Example

- Consider we want to drawallthebinarytreespossible with 3 labeled nodes.
- Number of binary trees possible with 3 labelednodes
- ={2 x 3C3/ (3 + 1) } x 3!
- ={6C3/ 4 } x 6
- =5 x 6
- =30

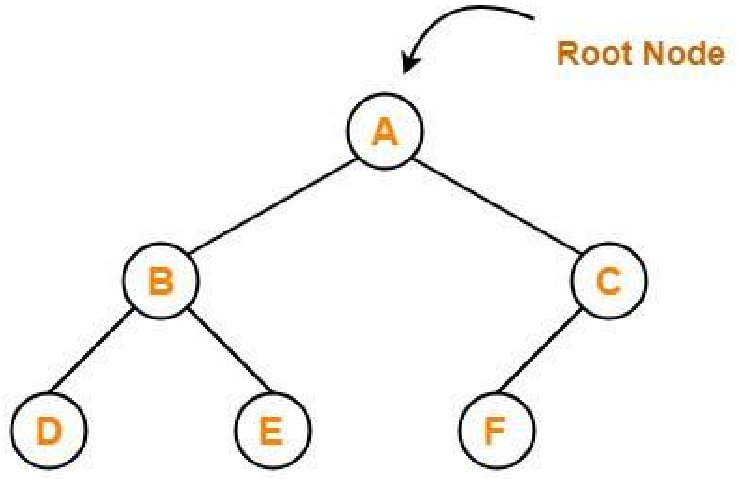


# Types of Binary Trees


Rooted Binary Tree
Full / Strictly Binary Tree
Types of Binary Tree
Complete / Perfect Binary Tree
Skewed Binary Tree
Almost Complete Binary Tree

## Rooted Binary Tree

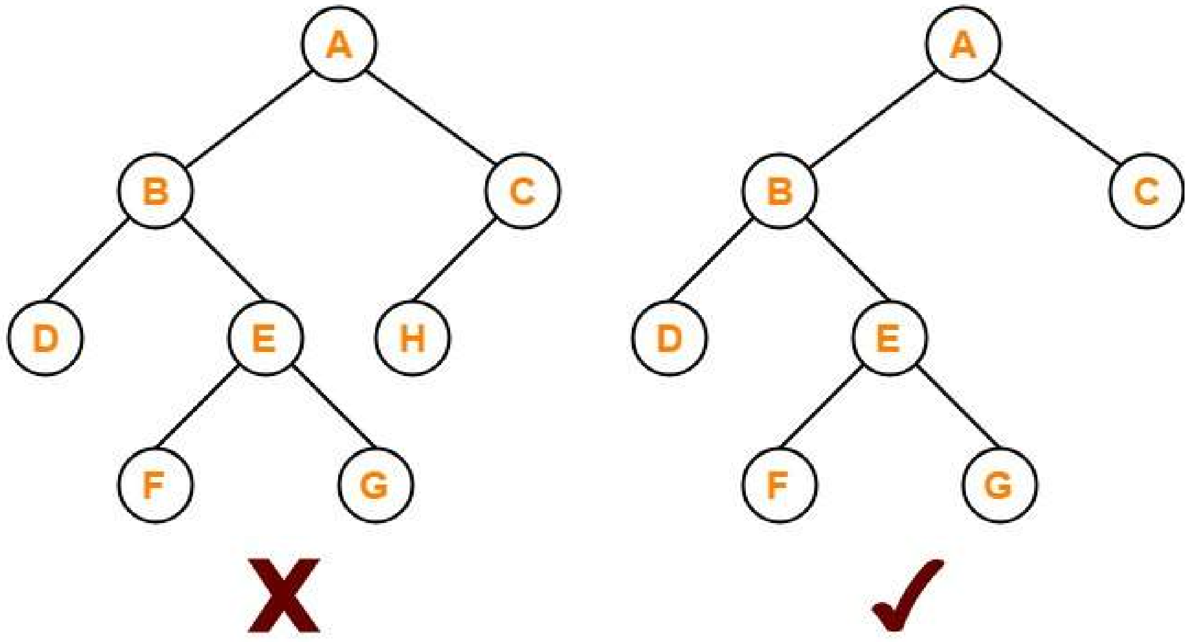
- Arooted binary treeis a binary tree that satisfies the following 2 properties:
  - Ithas arootnode.
  - Each node has at most 2 children.



Rooted Binary Tree

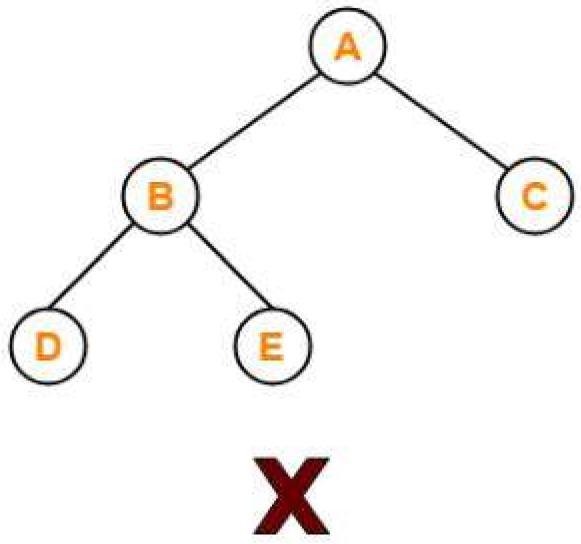
## Full/Strictly Binary Tree

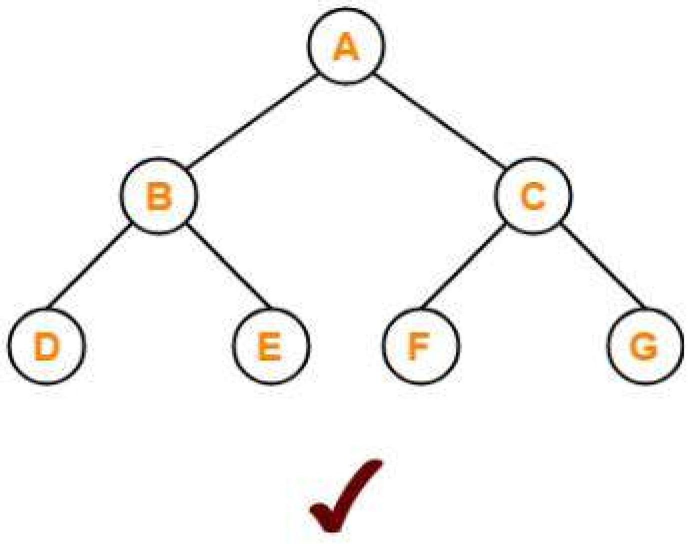
- A binary tree in which every node has either 0 or 2 children is called as aFull binary tree.
- Full binary tree is also called asStrictly binary tree.



# Complete /Perfect Binary Tree

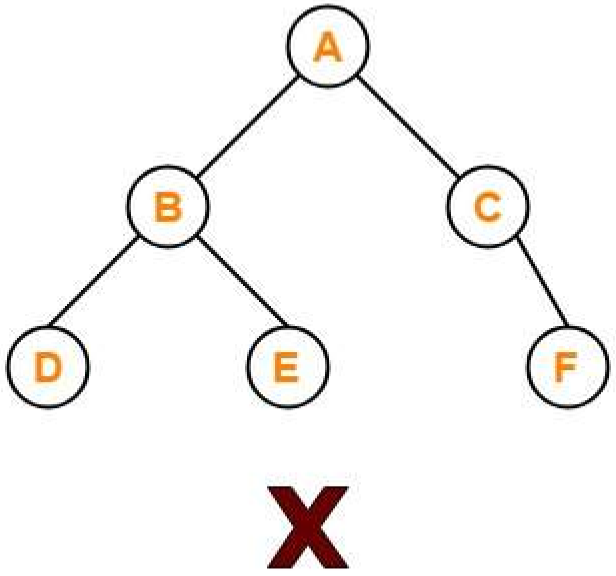
- Acomplete binary treeis a binary tree that satisfies the following 2 properties:
  - Every internalnodehas exactly 2 children.
  - All the leaf nodes are at the same level.

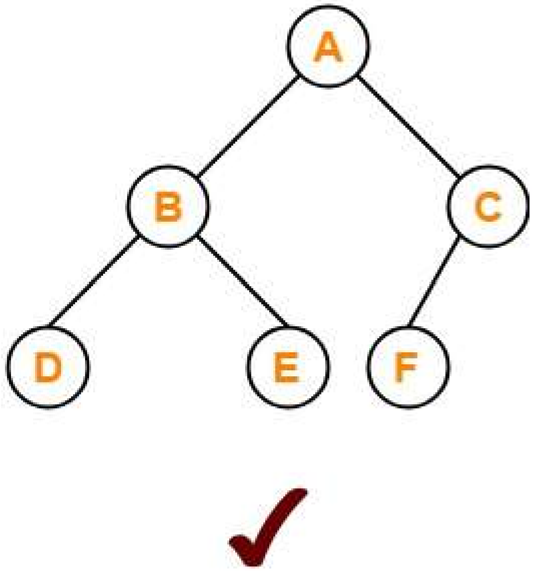




# Almost Complete Binary Tree

- Analmostcompletebinarytreeisabinarytreethatsatisfies the following 2 properties-
  - All the levels arecompletely filled except possibly the last level.
  - The last level must be strictly filled from left to right.



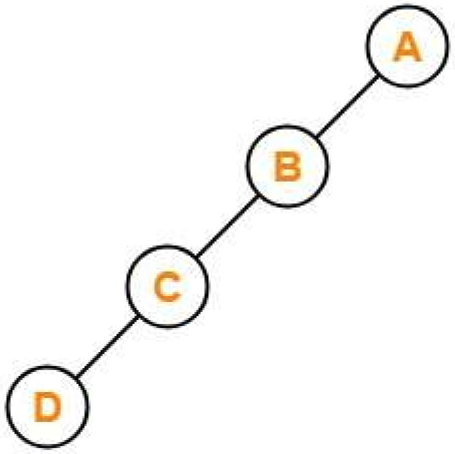


# Skewed Binary Tree

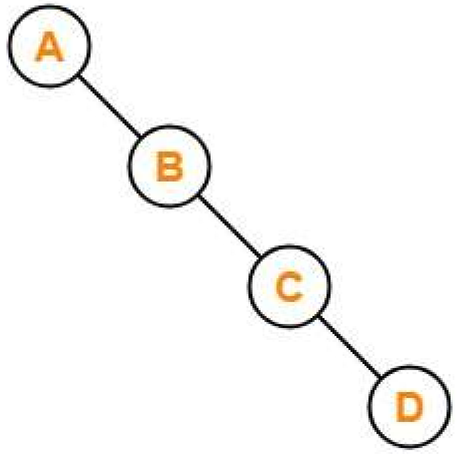
- Askewed binarytreeis a binarytreethatsatisfiesthefollowing2properties-
- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

- Askewed binary treeis a binary tree of n nodes such that its depth is (n-1).



Left Skewed Binary Tree

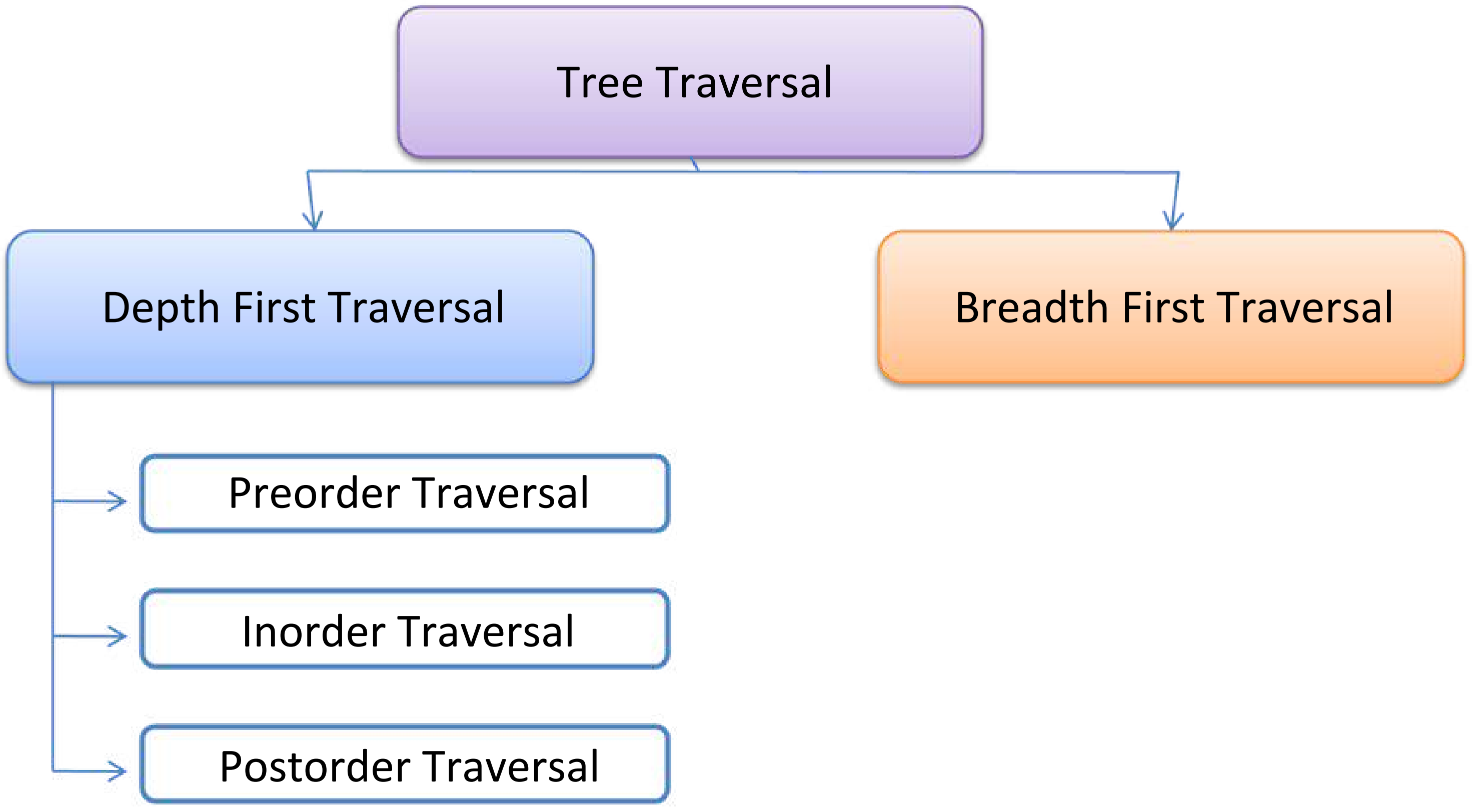


Right Skewed Binary Tree

## Tree Traversal

- In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.
- Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.

## Tree traversal techniques

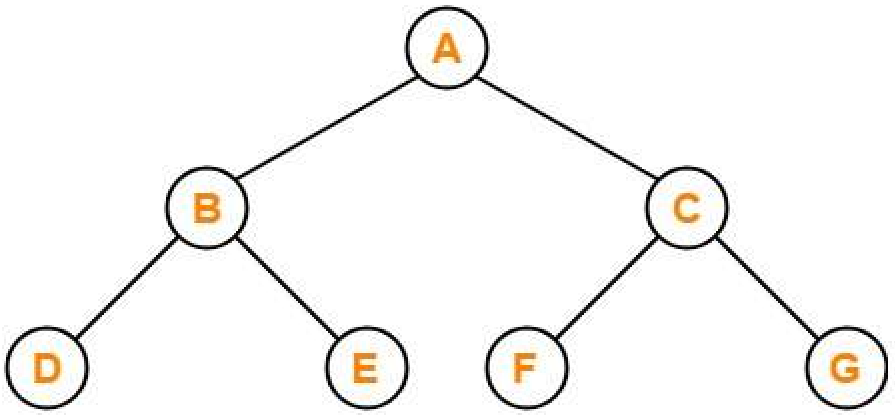


## Depth First Traversal

- Following three traversal techniques fall under Depth First Traversal-
  1. Preorder Traversal
  2. Inorder Traversal
  3. Postorder Traversal

## Preorder Traversal

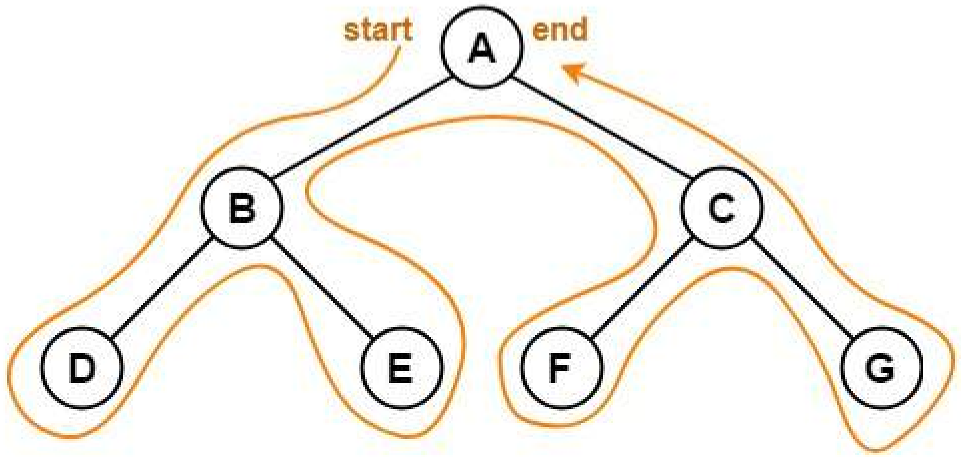
- Algorithm-
  - Visit the root
  - Traverse the left sub tree i.e. call Preorder(left sub tree)
  - Traverse the right sub tree i.e. call Preorder(right sub tree)



Preorder Traversal : A , B , D , E , C , F , G

## Preorder Traversal Shortcut

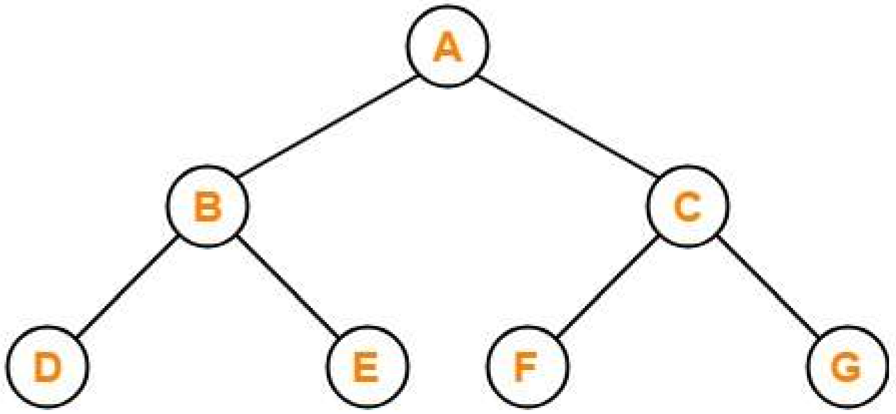
Traverse the entire tree starting from the root node keeping yourself to the left.



**Preorder Traversal : A , B , D , E , C , F , G**

# Inorder Traversal

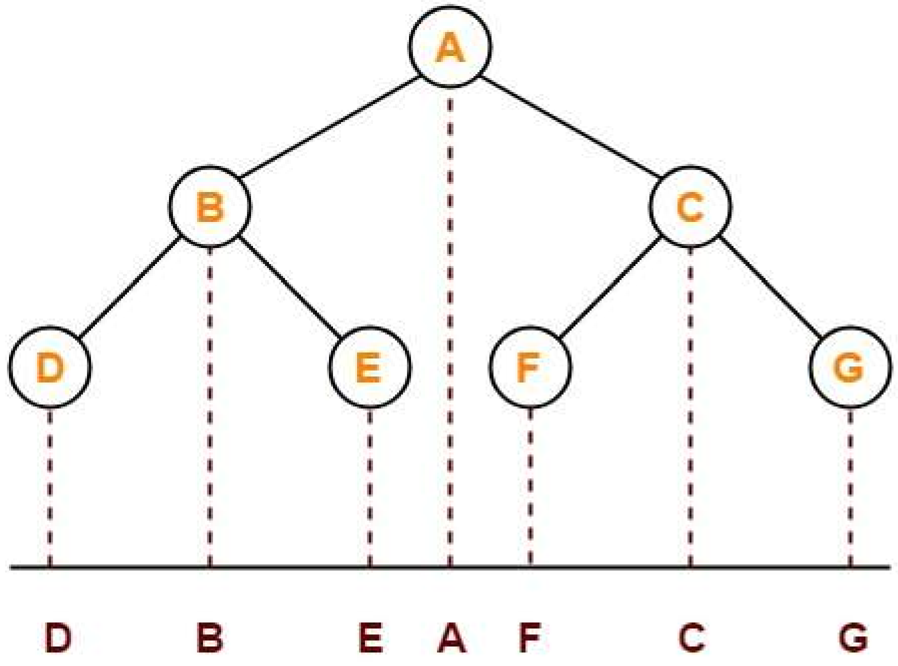
- Algorithm-
  - Traversethe left sub tree i.e. call Inorder(left sub tree)
  - Visitthe root
  - Traverse the right sub tree i.e. call Inorder(right sub tree)



Inorder Traversal : D , B , E , A , F , C , G

## Inorder Traversal Shortcut

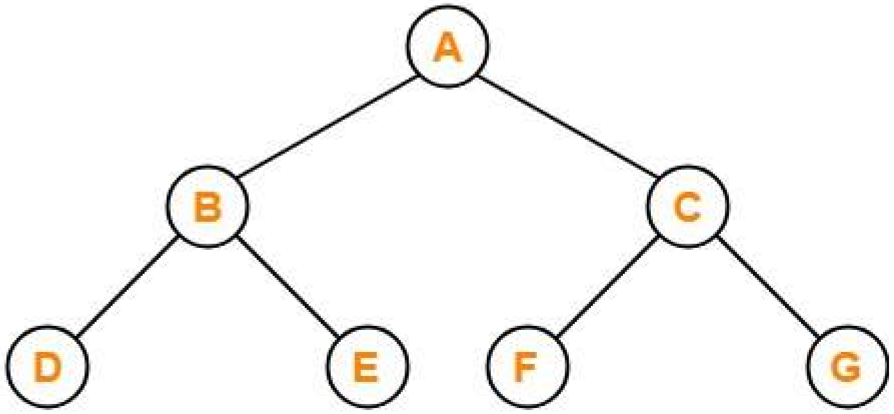
Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Inorder Traversal : D , B , E , A , F , C , G

# Postorder Traversal

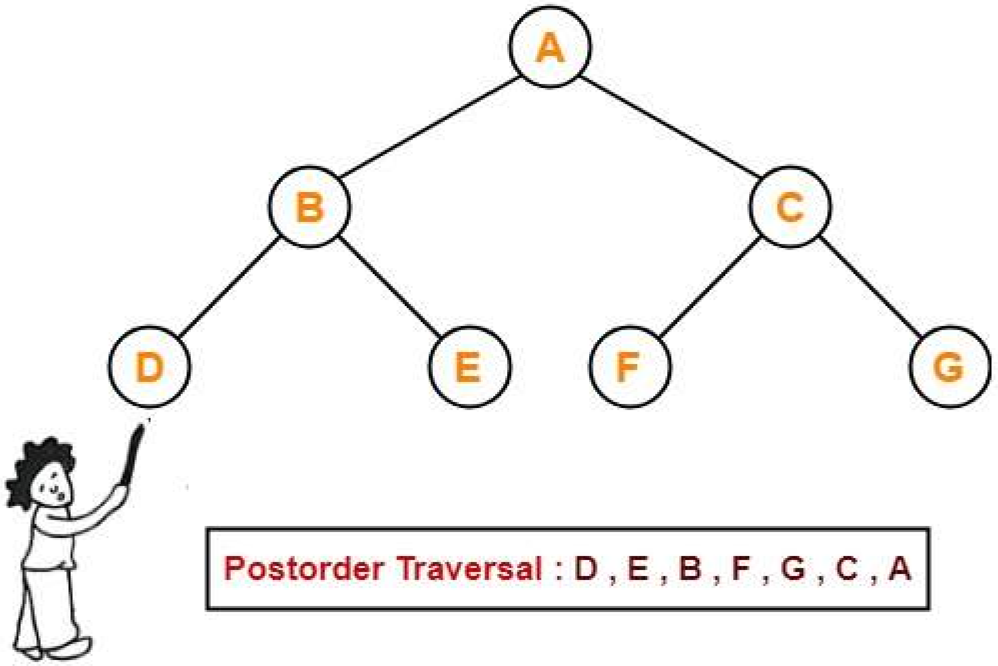
- Algorithm-
  - Traversethe left sub tree i.e. call Postorder(left sub tree)
  - Traverse therightsubtreei.e.callPostorder(rightsub tree)
  - Visit the root



Postorder Traversal : D , E , B , F , G , C , A

## Postorder Traversal Shortcut

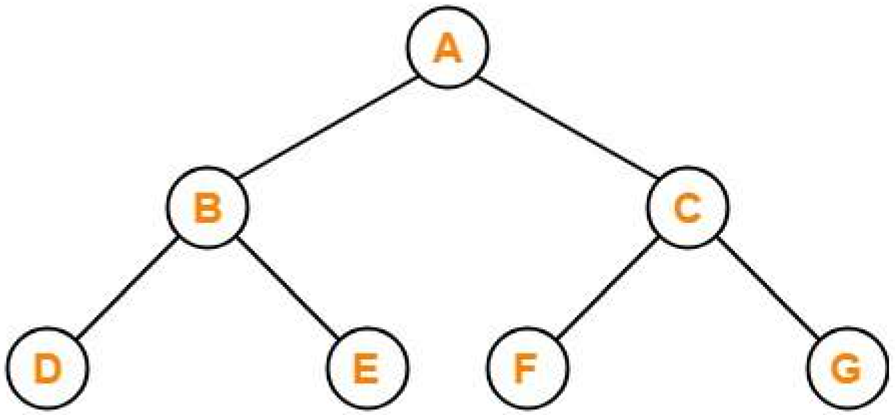
Pluck all the leftmost leaf nodes one by one.



Postorder Traversal : D , E , B , F , G , C , A

## Breadth First Search

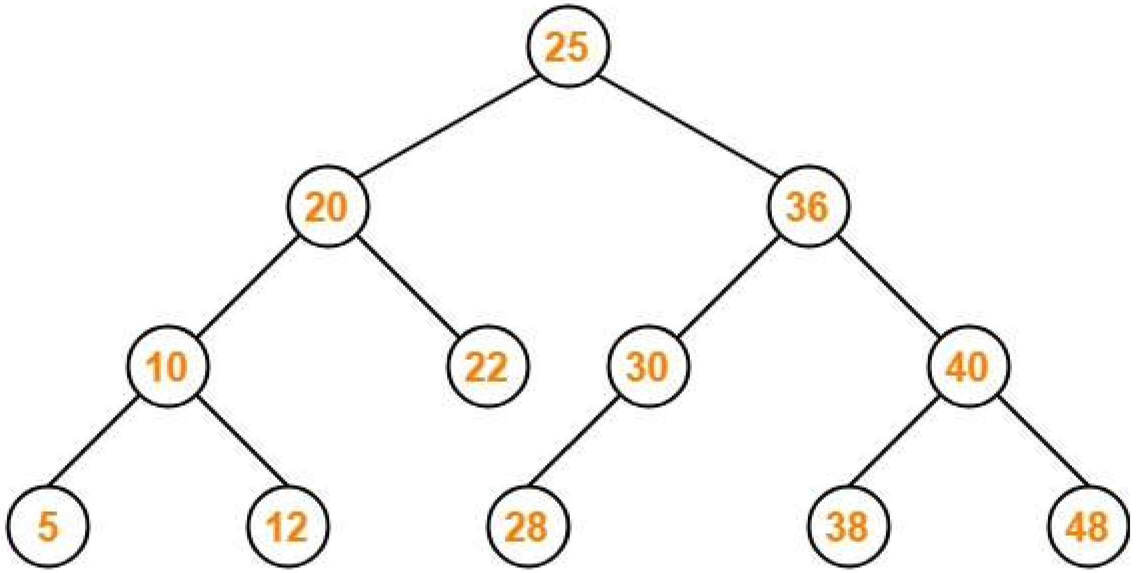
- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called asLevel Order Traversal.



Level Order Traversal : A , B , C , D , E , F , G

# Binary Search Tree construction

- In a binary search tree (BST), each node contains:
  - Only smaller values in its left sub tree
  - Only larger values in its right sub tree



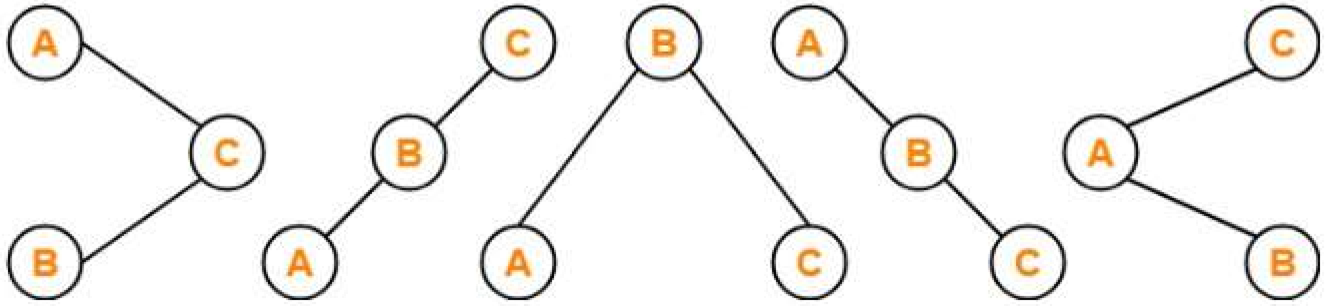
# BST construction

- Number of distinctbinarysearchtreespossiblewith3distinct Nodes

  =2×3C3/ 3+1

  =6C3/ 4

  = 5
- If three distinct Nodes are A, B and C, then 5 distinct binary search trees are:



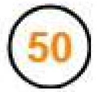
# Example

- Construct a Binary SearchTree(BST)forthefollowing sequence of numbers:

  50, 70, 60, 20, 90, 10, 40, 100

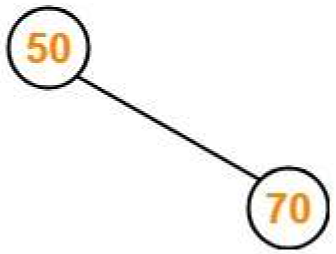
- When elements are given in a sequence,
  - Alwaysconsiderthe firstelement asthe root node.
  - Consider the given elements and insert them in the BST one by one.

**Insert 50-**



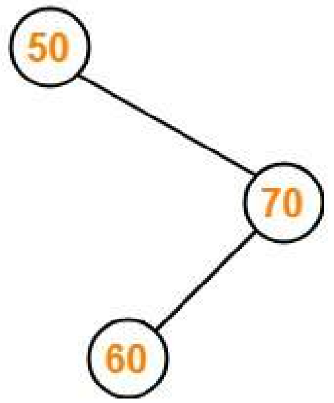
**Insert 70-**

- As 70 > 50, so insert 70 to the right of 50.



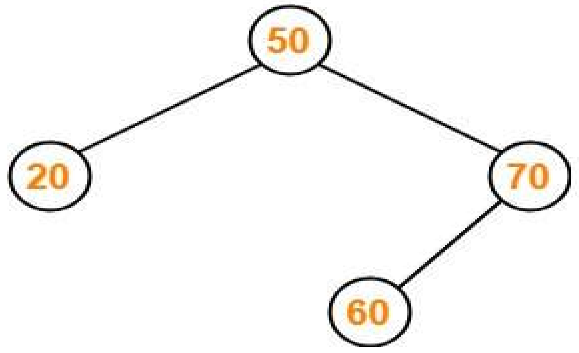
## Insert 60-

- As 60 > 50, so insert 60 to the right of 50.
- As 60 < 70, so insert 60 to the left of 70.



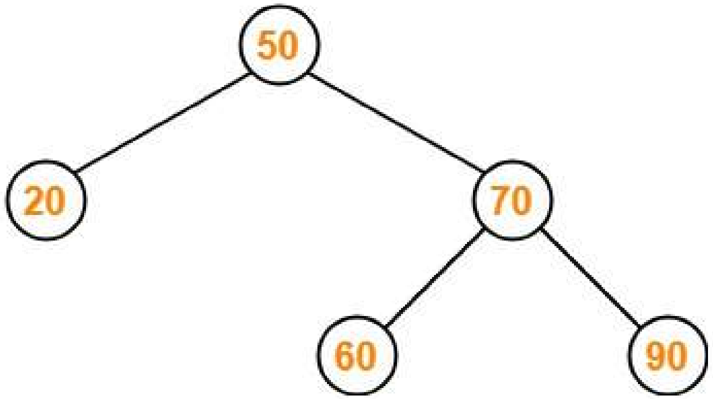
## Insert 20-

- As 20 < 50, so insert 20 to the left of 50.



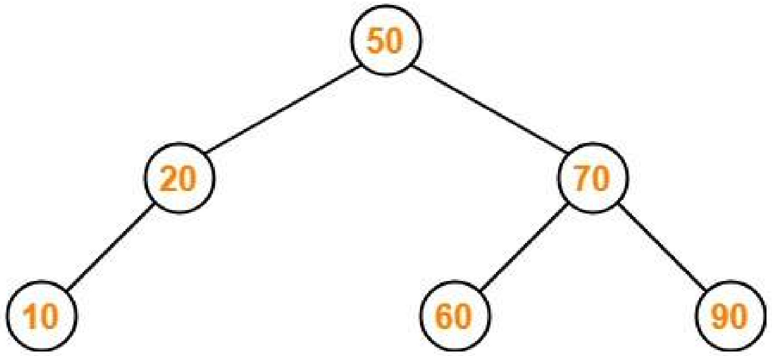
## Insert 90-

- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.



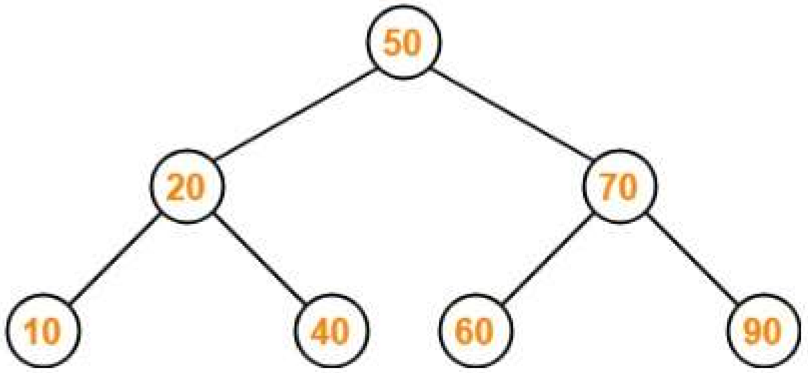
## Insert 10-

- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



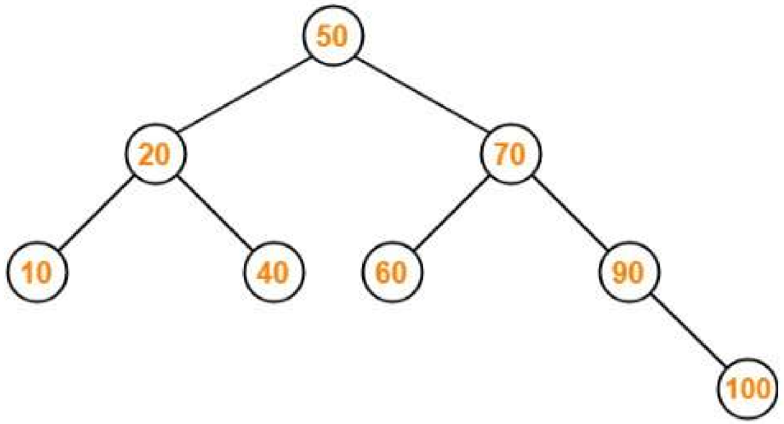
## Insert 40-

- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.



## Insert 100-

- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.



# Practice problem

- Problem-01:
- A binary search tree is generated by inserting in order of the following integers-

  50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24
- The number of nodes in the left subtreeand right subtreeof the root respectively is _____.
  - (4, 7)
  - (7, 4)
  - (8, 3)
  - (3, 8)

# Practice problem

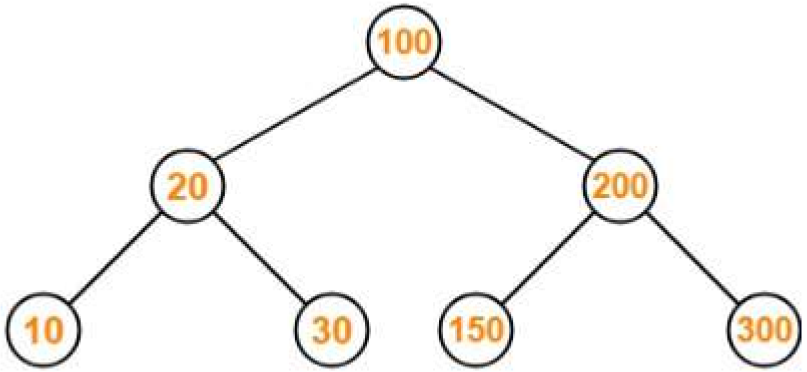
- Problem-02:
- Howmanydistinctbinary search trees can be constructed out of 4 distinct keys?
  - 5
  - 14
  - 24
  - 35

# C code –BST construction

```c
typedef struct BST
{
        int data;
        struct BST *left;
        struct BST *right;
}node;


node *create()
{
        node *temp;
        printf("nEnter data:");
        temp=(node*)malloc(sizeof(node));
        scanf("%d",&temp->data);
        temp->left=temp->right=NULL;
        return temp;
}

void insert(node *root,node *temp)
{
        if(temp->data<root->data)
        {
                if(root->left!=NULL)
                        insert(root->left,temp);
                else
                        root->left=temp;
        }

        if(temp->data>root->data)
        {
                if(root->right!=NULL)
                        insert(root->right,temp);
                else
                        root->right=temp;
        }
}
```

```c
int main()
{
        char ch;
        node *root=NULL,*temp;

        do
        {
                temp=create();
                if(root==NULL)
                        root=temp;
                else
                        insert(root,temp);

                printf("nDo you want to enter more(y/n)?");
                getchar();
                scanf("%c",&ch);
        }while(ch=='y'|ch=='Y');

        printf("nPreorder Traversal: ");
        preorder(root);
        return 0;
```

# BST Traversal



**Preorder Traversal-**

100 , 20 , 10 , 30 , 200 , 150 , 300

**Inorder Traversal-**

10 , 20 , 30 , 100 , 150 , 200 , 300

**Postorder Traversal-**

10 , 30 , 20 , 150 , 300 , 200 , 100

- Inordertraversal of a binary search tree always yields all the nodes in increasing order.

# C Code –BST Traversal

```
Void inorder(node*root)
{
   if(root!=NULL)
   {
         inorder(root->left);
         printf("%d ",root->data);
         inorder(root->right);
   }
}
```

## C Code –BST Traversal

```
void postorder(node*root)
{
    if(root!=NULL)
    {
        postorder(root->left);
        postorder(root->right);
        printf("%d ",root->data);
    }
}
```

## C Code –BST Traversal

```
void preorder(node*root)
{
    if(root!=NULL)
    {
        printf("%d ",root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

# Practice

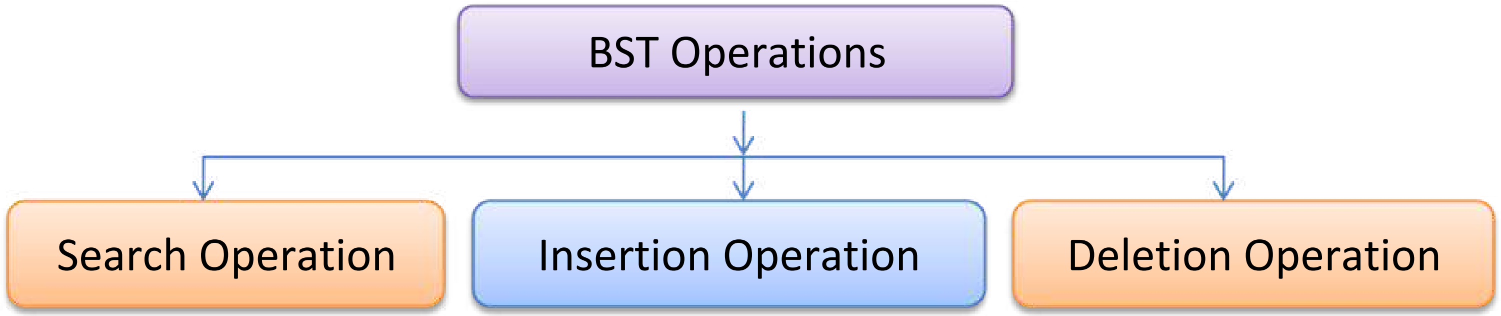
- Suppose the numbers 7 , 5 , 1 , 8 , 3 , 6 , 0 , 9 , 4 , 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers.
- What is the inordertraversal sequence of the resultant tree?
  A. 7 , 5 , 1 , 0 , 3 , 2 , 4 , 6 , 8 , 9
  B. 0 , 2 , 4 , 3 , 1 , 6 , 5 , 9 , 8 , 7
  C. 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9
  D. 9 , 8 , 6 , 4 , 2 , 3 , 0 , 1 , 5 , 7

## Problem-2

- The preordertraversal sequence of a binary search tree is-
  - 30 , 20 , 10 , 15 , 25 , 23 , 39 , 35 , 42
- Which one of the following is the postorder traversal sequence of the same tree?
  - A. 10 , 20 , 15 , 23 , 25 , 35 , 42 , 39 , 30
  - B. 15 , 10 , 25 , 23 , 20 , 42 , 35 , 39 , 30
  - C. 15 , 20 , 10 , 23 , 25 , 42 , 35 , 39 , 30
  - D. 15 , 10 , 23 , 25 , 20 , 35 , 42 , 39 , 30

# BST Operations

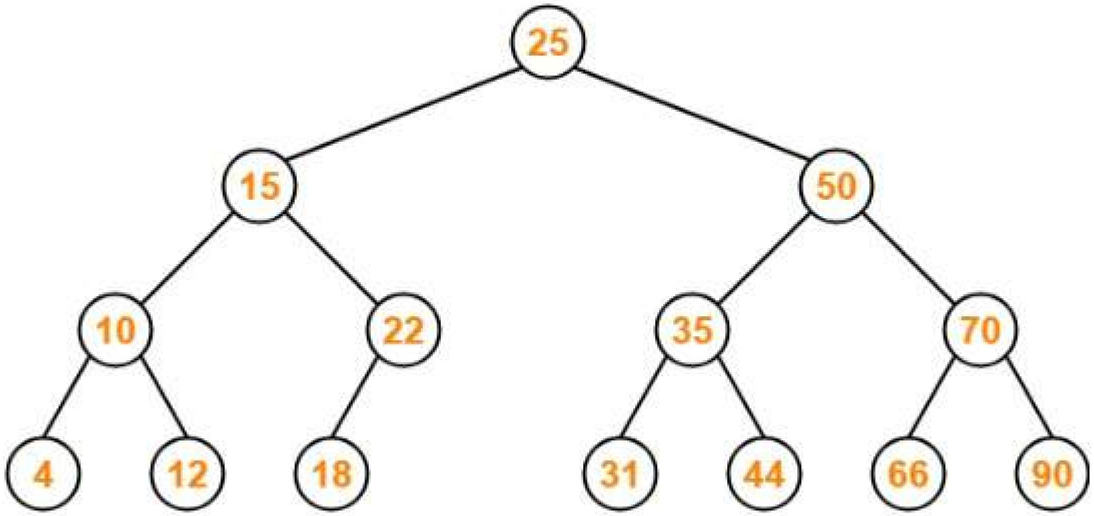
- Commonly performed binary search tree operations are:



## Search Operation

- Search Operation is performed to search a particular element in the Binary Search Tree.
- For searching a given key in the BST,
  - Compare the keywith the value ofrootnode.
  - If the key is present at the root node, then return the root node.
  - If the key is greater than the root node value, then recur for the root node’s right subtree.
  - If the key is smaller than the root node value, then recur for the root node’s left subtree.

Consider key = 45 has to be searched in the given BST-



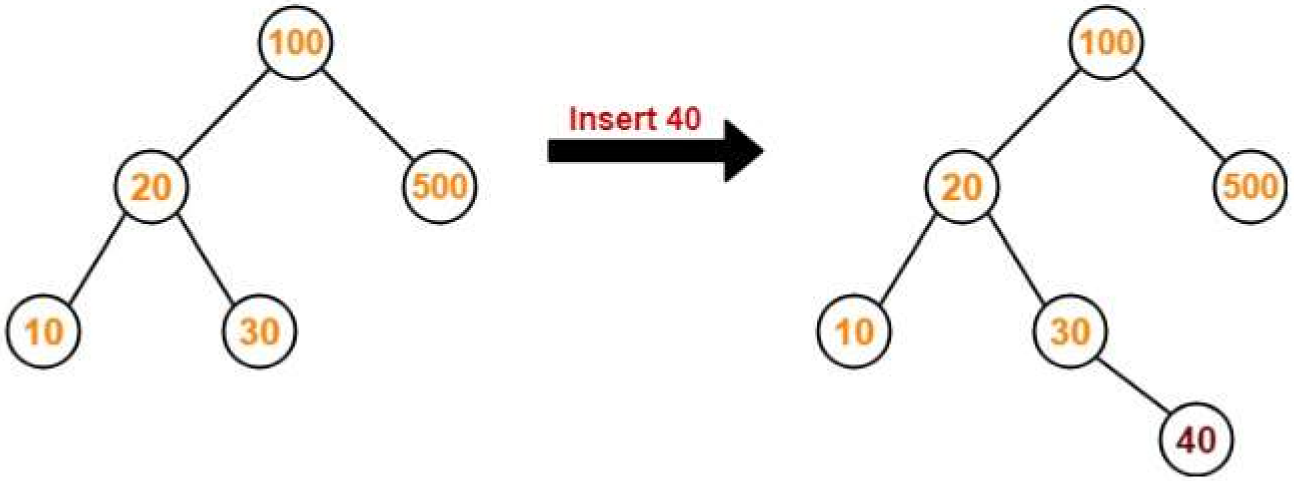
- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

# C Code –BST Search Operation

```
structnode *search(structnode*node,intkey){
 // Return NULL if the tree is empty
 if (node == NULL) return NULL;
 if (node->key == key) return node->data;
 if (key < node->key)
   search(node->left, key);
 else
   search(node->right, key);
}
```

## Insertion operation

- The insertion of a new key always takes place as the child of some leaf node.
- For finding out the suitable leaf node,
  - Search the keyto be inserted from the root node till some leaf node is reached.
  - Once a leaf node is reached, insert the key as child of that leaf node.



- We start searching for value 40 from the root node 100.
- As 40 < 100, so we search in 100's left subtree.
- As 40 > 20, so we search in 20's right subtree.
- As 40 > 30, so we add 40 to 30's right subtree.

# C Code –Insertion Operation

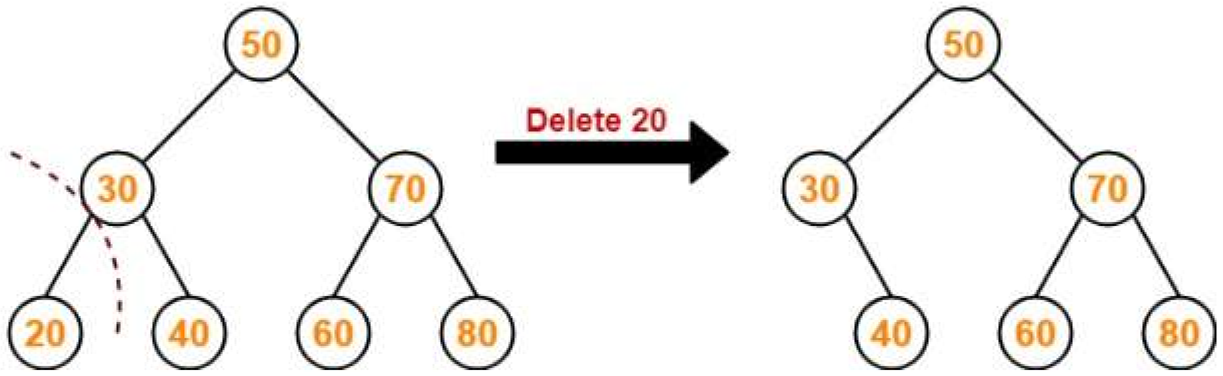
```
// Createa node
structnode *newNode(intitem) {
 structnode *temp = (structnode *)malloc(sizeof(structnode));
 temp->key = item;
 temp->left = temp->right = NULL;
 return temp;
}
// Insert a node
structnode *insert(structnode *node, intkey) {
 // Return a new node if the tree is empty
 if (node == NULL) return newNode(key);

 // Traverse to the right place and insert the node

 if (key < node->key)

  node->left = insert(node->left, key);
 else
  node->right = insert(node->right, key);

 return node;
}
```

## Deletion Operation

- Deletion Operation is performed to delete a particular element from the Binary Search Tree.
  - When it comes to deleting a node from the binary search tree, three cases are possible.

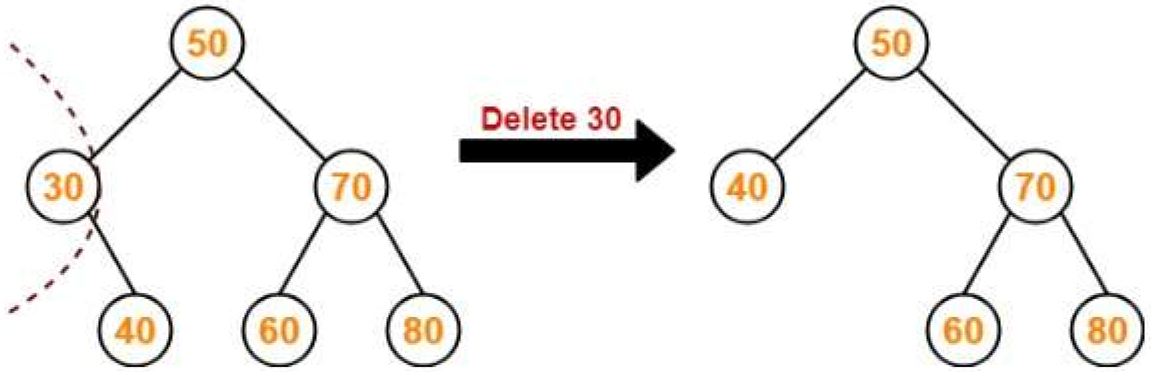
# Case-01: Deletion Of A Node Having NoChild (Leaf Node)

- Just remove/disconnecttheleafnode that is to deleted from the tree.



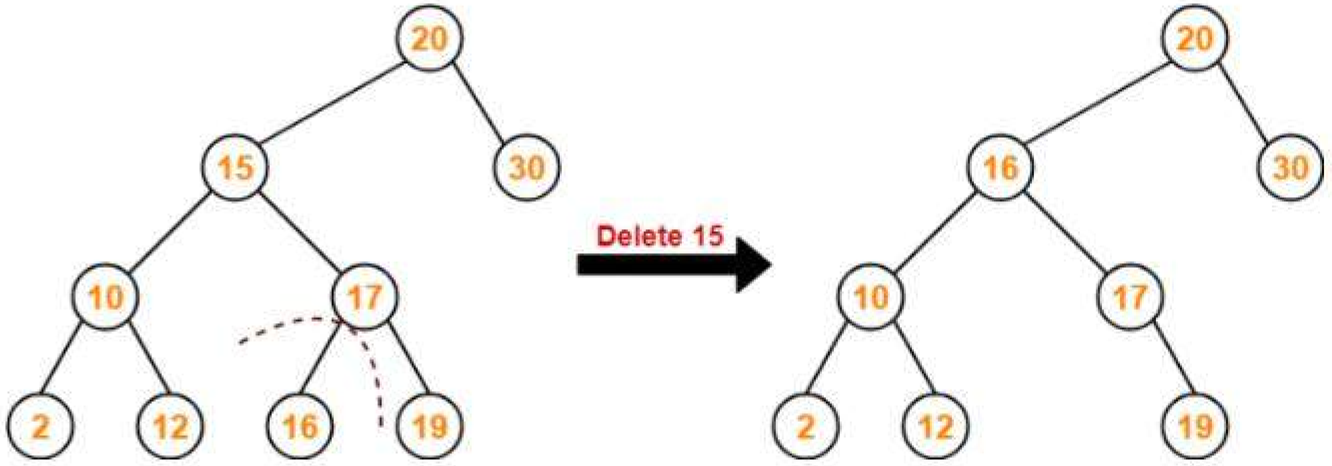
# Case-02: Deletion Of A Node Having Only One Child

- Consider thefollowingexamplewhere node with value = 30 is deleted from the BST.

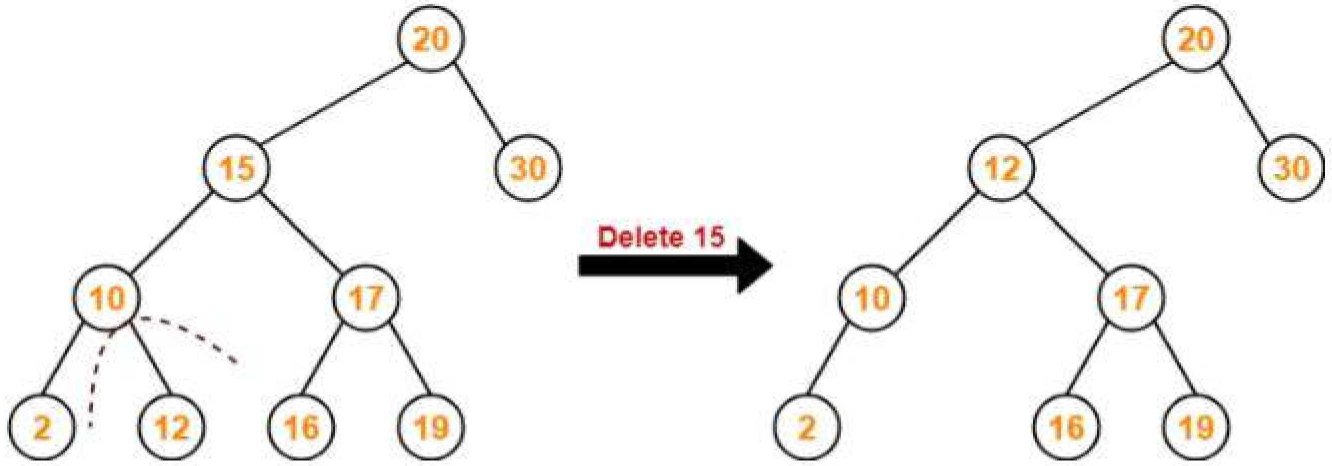


# Case-03: Deletion Of A Node Having Two Children

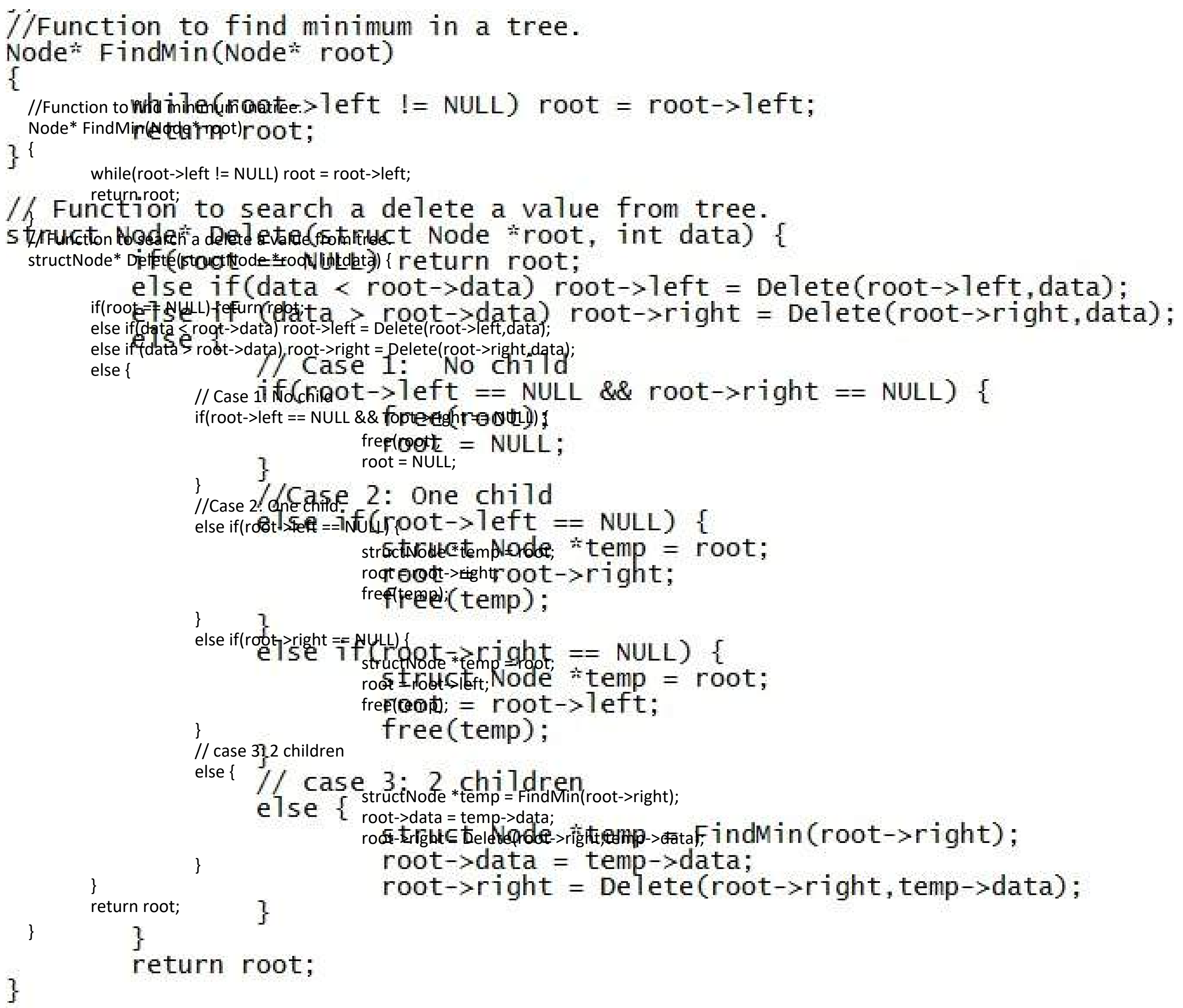
- Consider the followingexamplewhere node with value = 15 is deleted from the BST
- Method-1:
  - Visit to the rightsubtreeofthedeleting node.
  - Pluck the least value element called as inordersuccessor.
  - Replace the deleting element with its inordersuccessor.



- Method-2:
  - Visit to the left subtreeof the deleting node.
  - Pluck the greatest value element called as inordersuccessor.
  - Replace the deleting element with its inordersuccessor.



# C code –BST deletion



```c
//Function to find minimum in a tree.
Node* FindMin(Node* root)
{
    while(root->left != NULL) root = root->left;
    return root;
}

// Function to search a delete a value from tree.
struct Node* Delete(struct Node *root, int data) {
    if(root == NULL) return root;
    else if(data < root->data) root->left = Delete(root->left,data);
    else if (data > root->data) root->right = Delete(root->right,data);
    else {
        // Case 1:  No child
        if(root->left == NULL && root->right == NULL) {
            free(root);
            root = NULL;
        }
        //Case 2: One child
        else if(root->left == NULL) {
            struct Node *temp = root;
            root = root->right;
            free(temp);
        }
        else if(root->right == NULL) {
            struct Node *temp = root;
            root = root->left;
            free(temp);
        }
        // case 3: 2 children
        else {
            struct Node *temp = FindMin(root->right);
            root->data = temp->data;
            root->right = Delete(root->right,temp->data);
        }
    }
    return root;
}
```

```c
//Function to find minimum in a tree.
Node* FindMin(Node* root)
{
        while(root->left != NULL) root = root->left;
        return root;
}
//Function to search a delete a value from tree.
structNode* Delete(structNode *root, intdata) {
        if(root == NULL) return root;
        else if(data < root->data) root->left = Delete(root->left,data);
        else if (data > root->data) root->right = Delete(root->right,data);
        else {
                // Case 1: No child
                if(root->left == NULL && root->right == NULL) {
                        free(root);
                        root = NULL;
                }
                //Case 2: One child
                else if(root->left == NULL) {
                        structNode *temp = root;
                        root = root->right;
                        free(temp);
                }
                else if(root->right == NULL) {
                        structNode *temp = root;
                        root = root->left;
                        free(temp);
                }
                // case 3: 2 children
                else {
                        structNode *temp = FindMin(root->right);
                        root->data = temp->data;
                        root->right = Delete(root->right,temp->data);
                }
        }
        return root;
}
```

# AVL Tree Data Structure

An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

The difference between the heights of the left subtree and the right subtree for any node is known as the balance factor of the node.

The AVL tree is named after its inventors, Georgy Adelson-Velsky and Evgenii Landis, who published it in their 1962 paper "An algorithm for the organization of information".

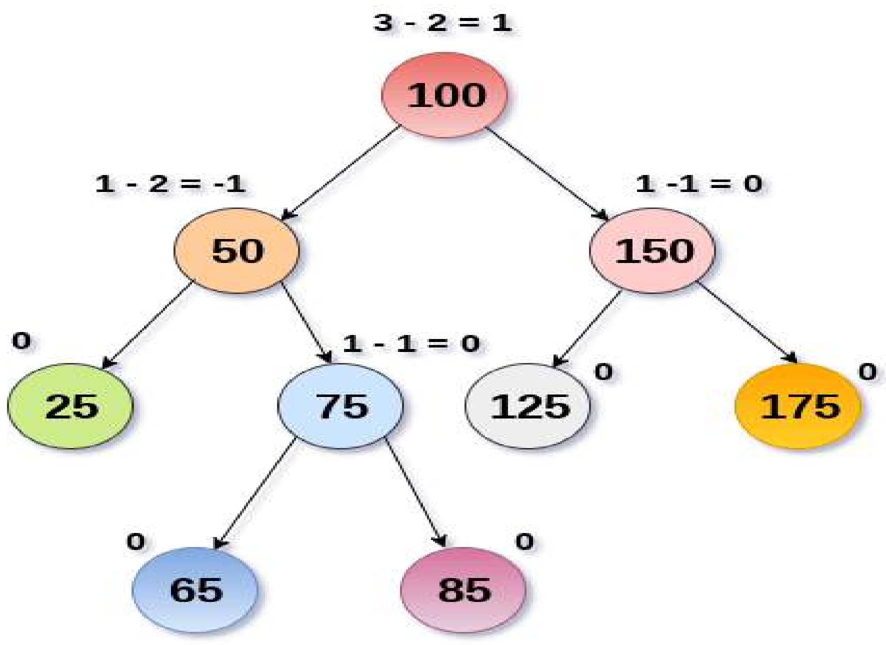
## Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

# AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

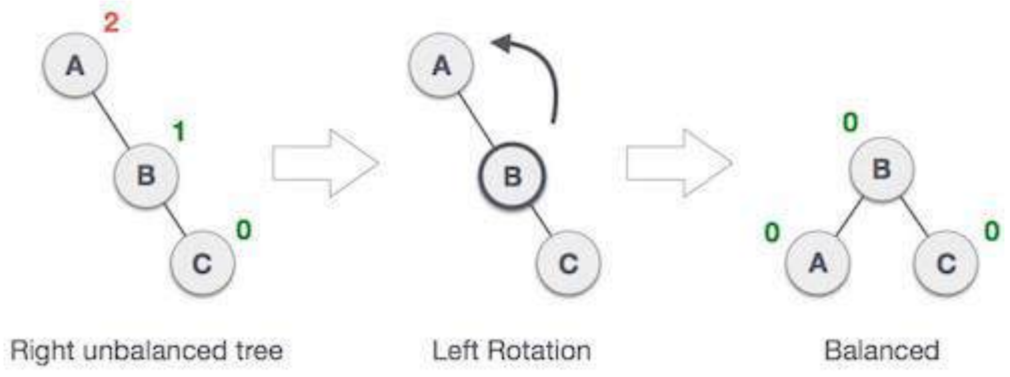
1. **L L rotation:** Inserted node is in the left subtree of left subtree of A
2. **R R rotation :** Inserted node is in the right subtree of right subtree of A
3. **L R rotation :** Inserted node is in the right subtree of left subtree of A
4. **R L rotation :** Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

## 1. RR Rotation

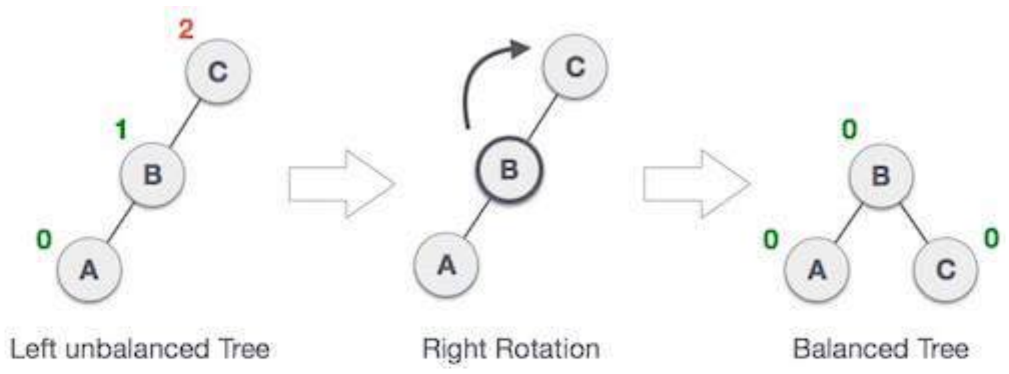
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

## 2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.

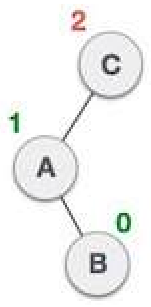
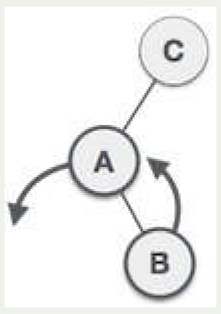


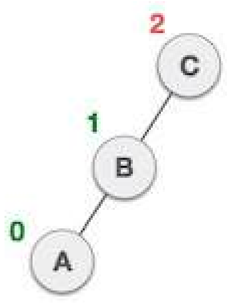
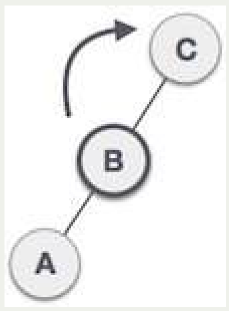
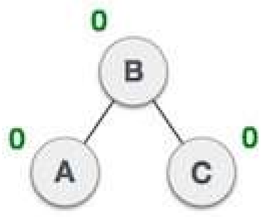
In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

## 3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

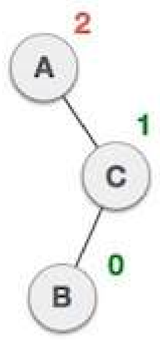
**Let us understand each and every step very clearly:**

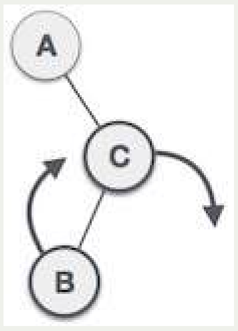
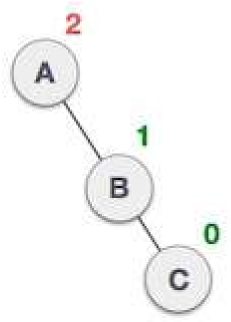
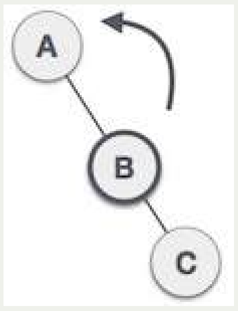
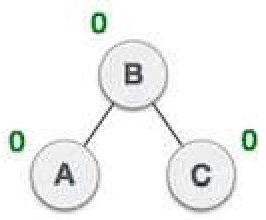
| State | Action |
|---|---|
|  | A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C |
|  | As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**. |

|   |   |
|---|---|
|  | After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C** |
|  | Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B |
|  | Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now. |

## 4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

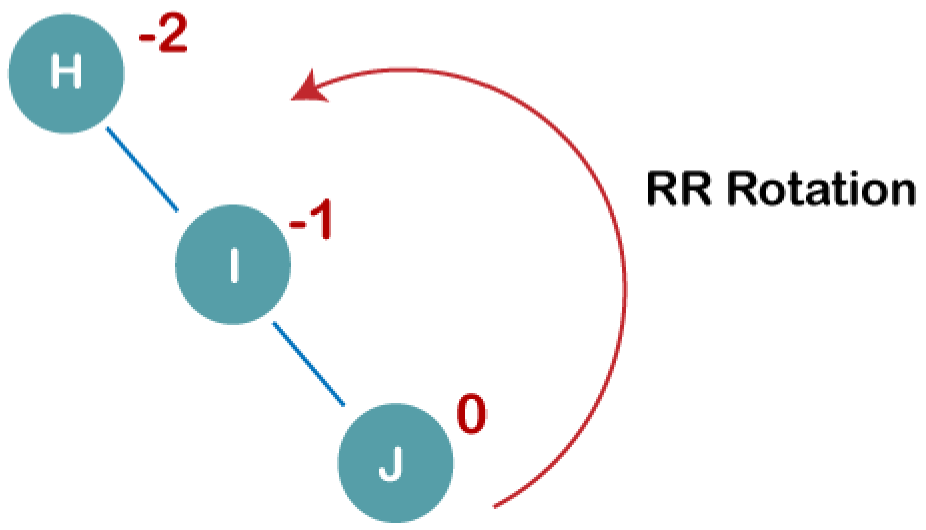
| **State** | **Action** |
|---|---|
|  | A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A |

| | |
|---|---|
|  | As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**. |
|  | After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A. |
|  | Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B. |
|  | Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now. |

## Q: Construct an AVL tree having the following elements

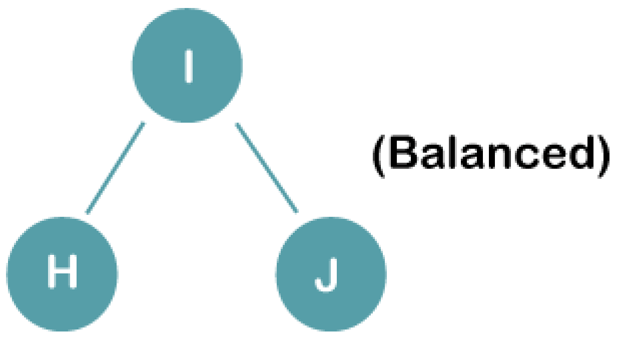
**H, I, J, B, A, E, C, F, D, G, K, L**

**1. Insert H, I, J**

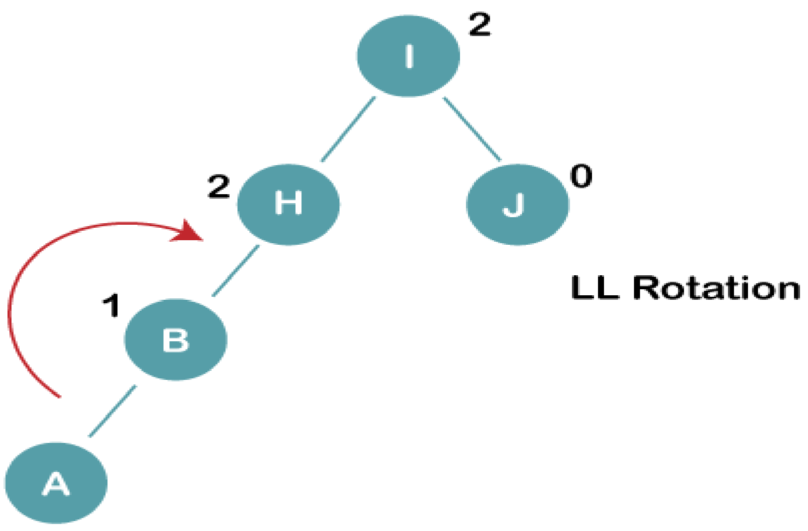


On inserting the above elements, especially in the case of H, the BST becomes unbalanced as the Balance Factor of H is -2. Since the BST is right-skewed, we will perform RR Rotation on node H.

**The resultant balance tree is:**

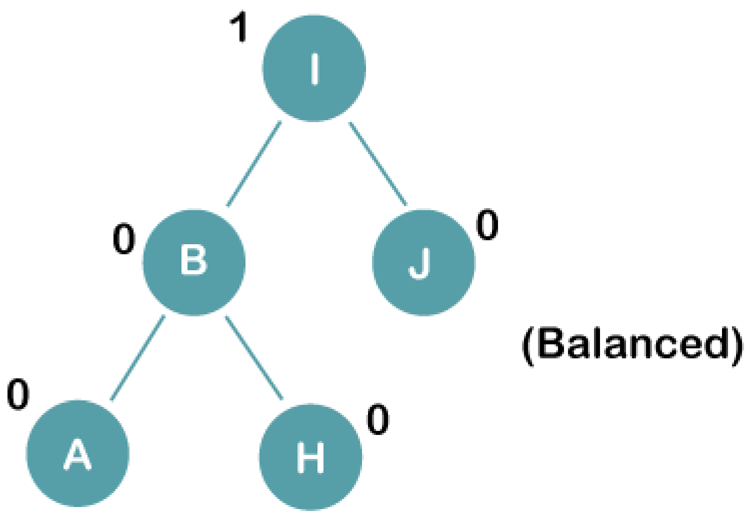


**2. Insert B, A**

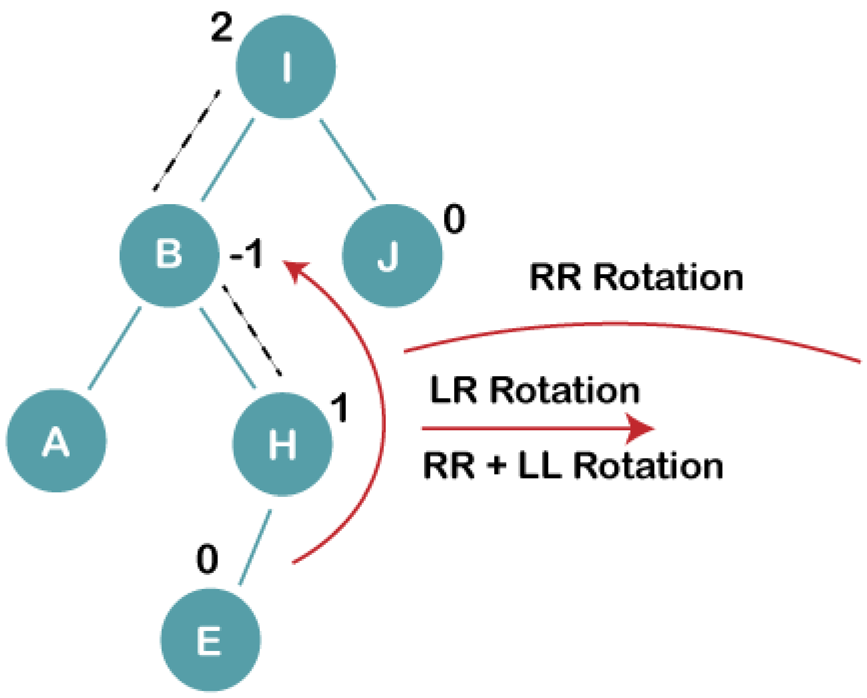


On inserting the above elements, especially in case of A, the BST becomes unbalanced as the Balance Factor of H and I is 2, we consider the first node from the last inserted node i.e. H. Since the BST from H is left-skewed, we will perform LL Rotation on node H.

**The resultant balance tree is:**



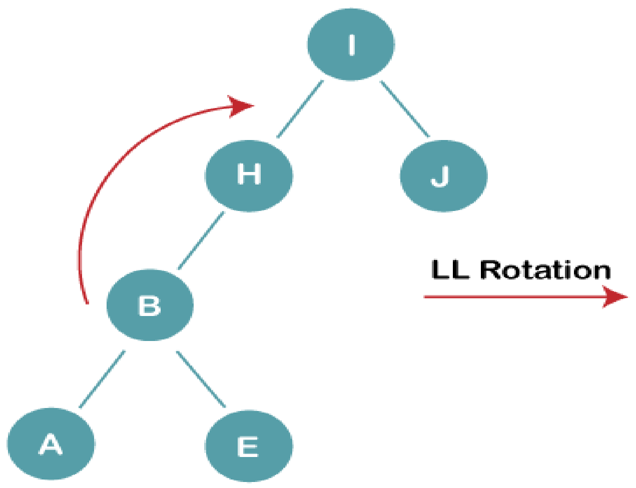
**3. Insert E**



On inserting E, BST becomes unbalanced as the Balance Factor of I is 2, since if we travel from E to I we find that it is inserted in the left subtree of right subtree of I, we will perform LR Rotation on node I. LR = RR + LL rotation

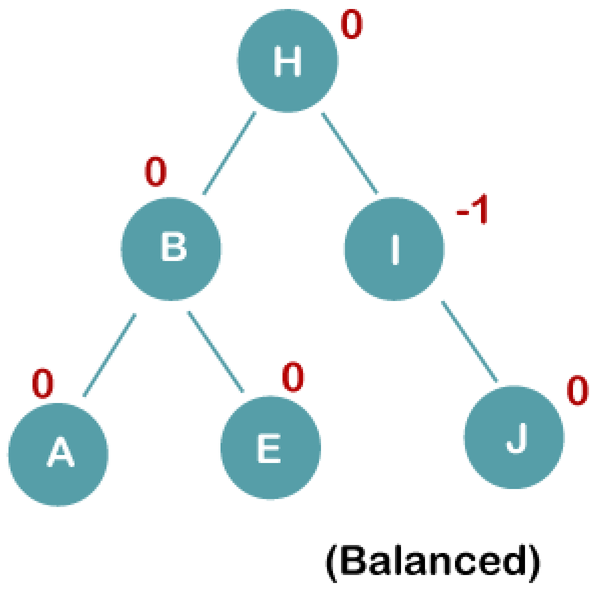
**3 a) We first perform RR rotation on node B**

**The resultant tree after RR rotation is:**

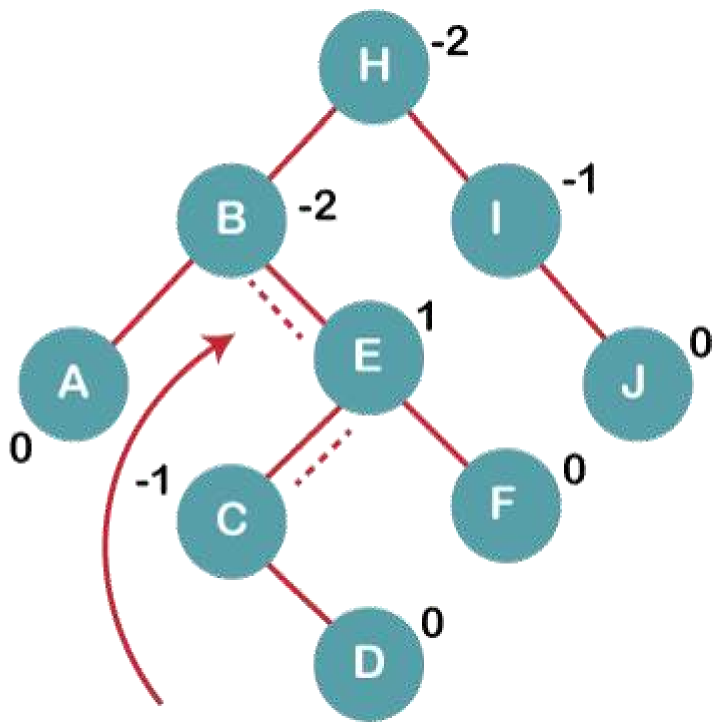


**3b) We first perform LL rotation on the node I**

**The resultant balanced tree after LL rotation is:**



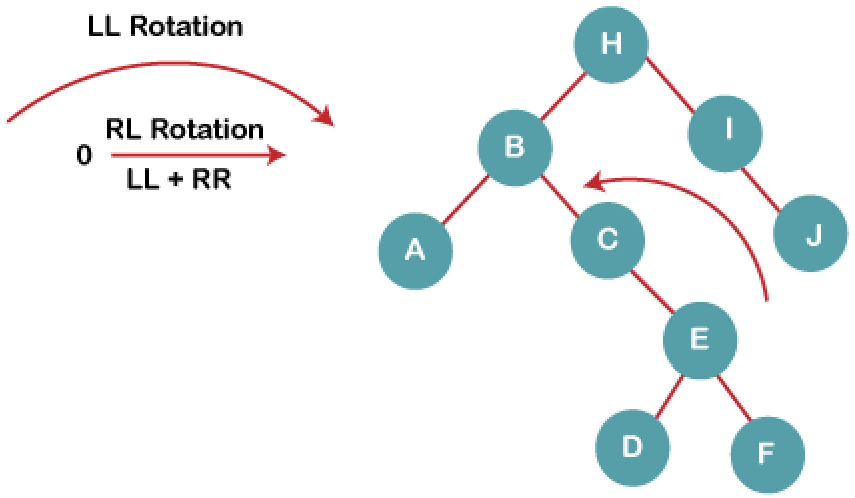
**4. Insert C, F, D**



On inserting C, F, D, BST becomes unbalanced as the Balance Factor of B and H is -2, since if we travel from D to B we find that it is inserted in the right subtree of left subtree of B, we will perform RL Rotation on node I. RL = LL + RR rotation.

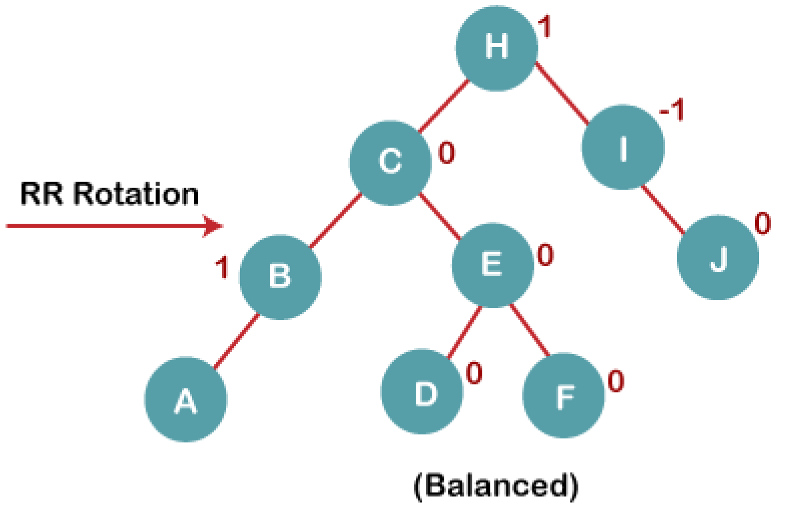
**4a) We first perform LL rotation on node E**

**The resultant tree after LL rotation is:**

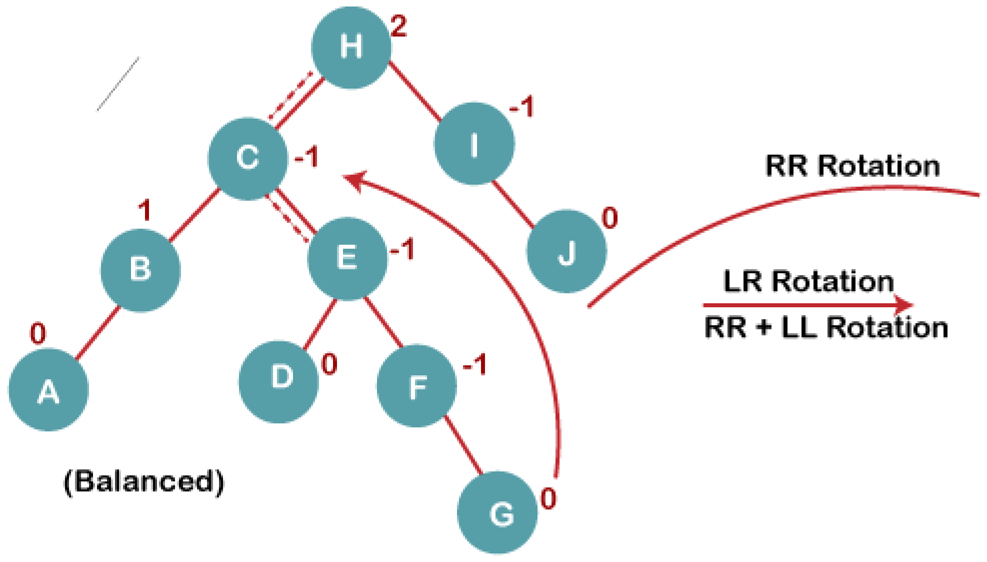


**4b) We then perform RR rotation on node B**

**The resultant balanced tree after RR rotation is:**



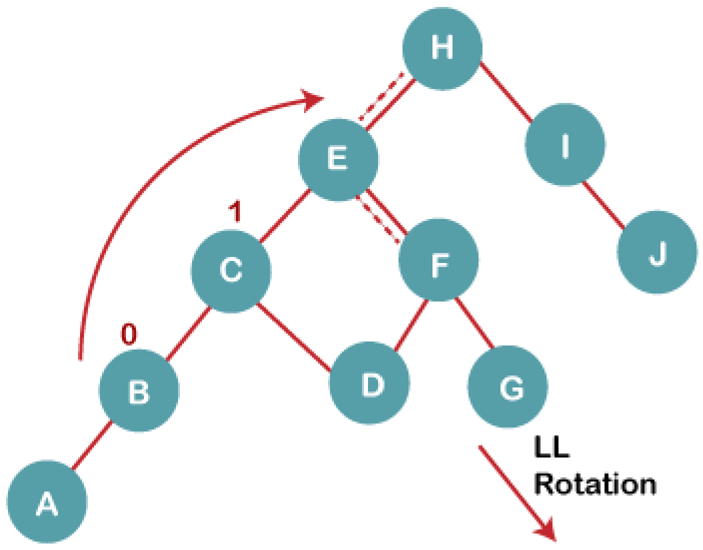
**5. Insert G**



On inserting G, BST become unbalanced as the Balance Factor of H is 2, since if we travel from G to H, we find that it is inserted in the left subtree of right subtree of H, we will perform LR Rotation on node I. LR = RR + LL rotation.

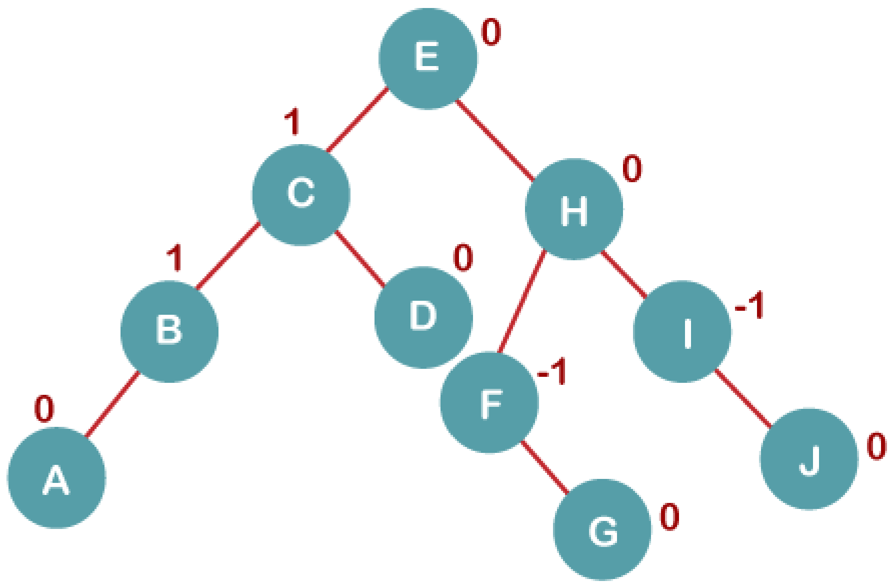
**5 a) We first perform RR rotation on node C**

**The resultant tree after RR rotation is:**



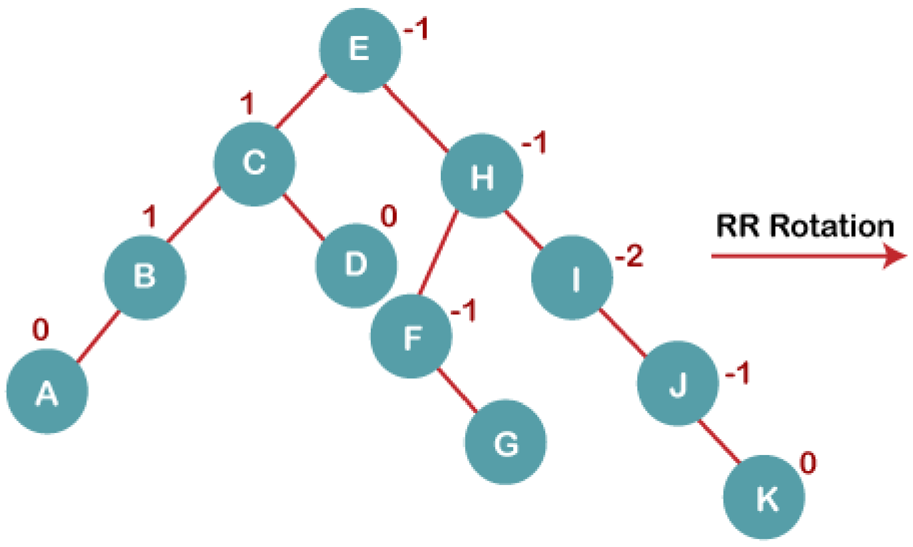
**5 b) We then perform LL rotation on node H**

**The resultant balanced tree after LL rotation is:**



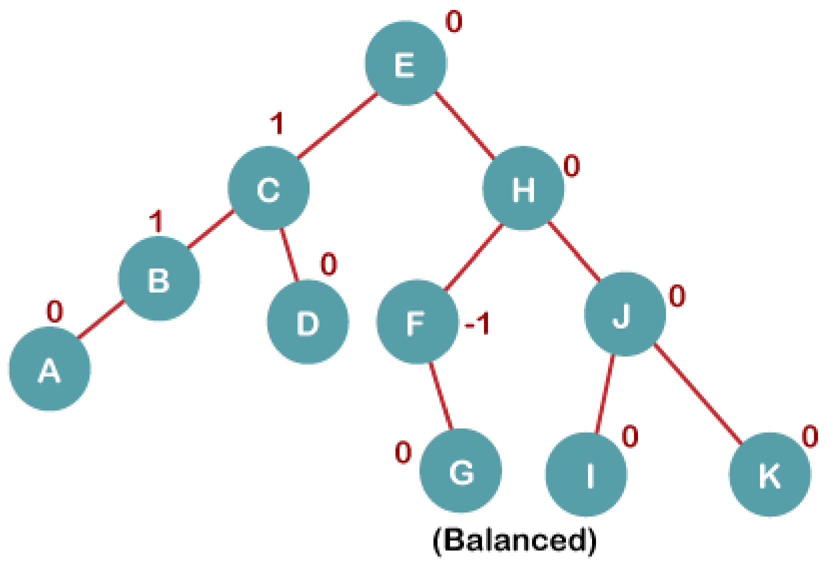
(Balanced)

**6. Insert K**



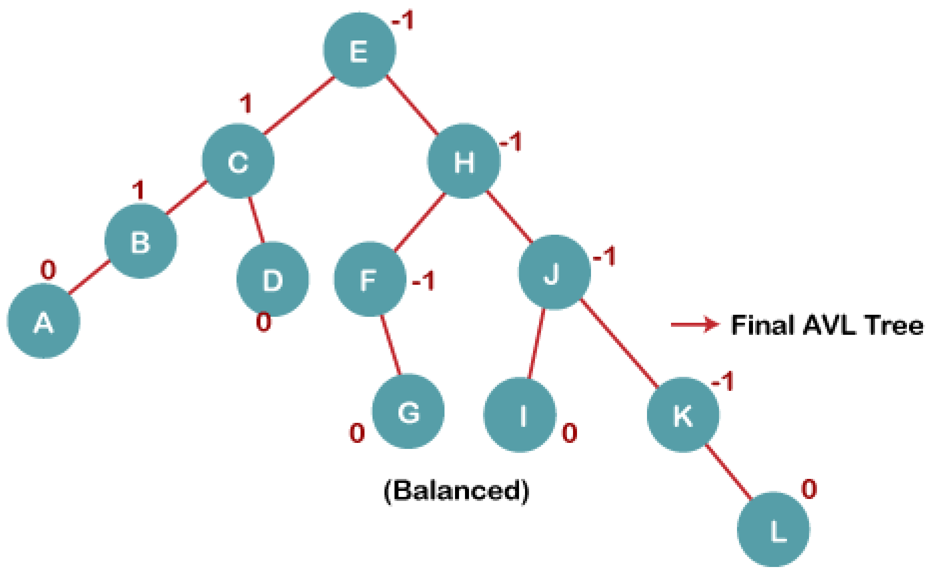
On inserting K, BST becomes unbalanced as the Balance Factor of I is -2. Since the BST is right-skewed from I to K, hence we will perform RR Rotation on the node I.

**The resultant balanced tree after RR rotation is:**



**7. Insert L**

On inserting the L tree is still balanced as the Balance Factor of each node is now either, -1, 0, +1. Hence the tree is a Balanced AVL tree


E -1
C 1
H -1
B 1
D 0
F -1
J -1
A 0
G 0
I 0
K -1
L 0
Final AVL Tree
(Balanced)