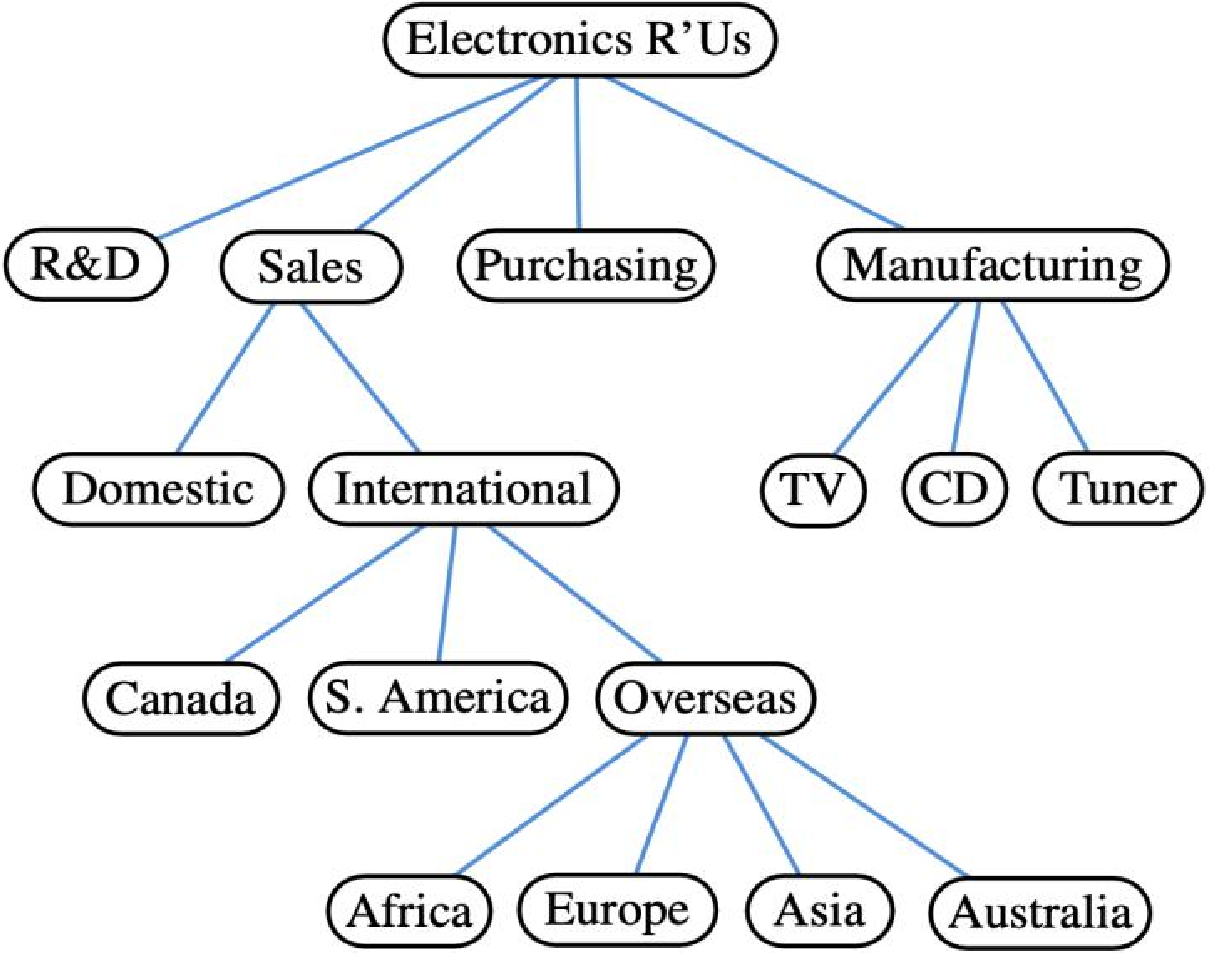
Algorithms
and
Data Structures Using Java

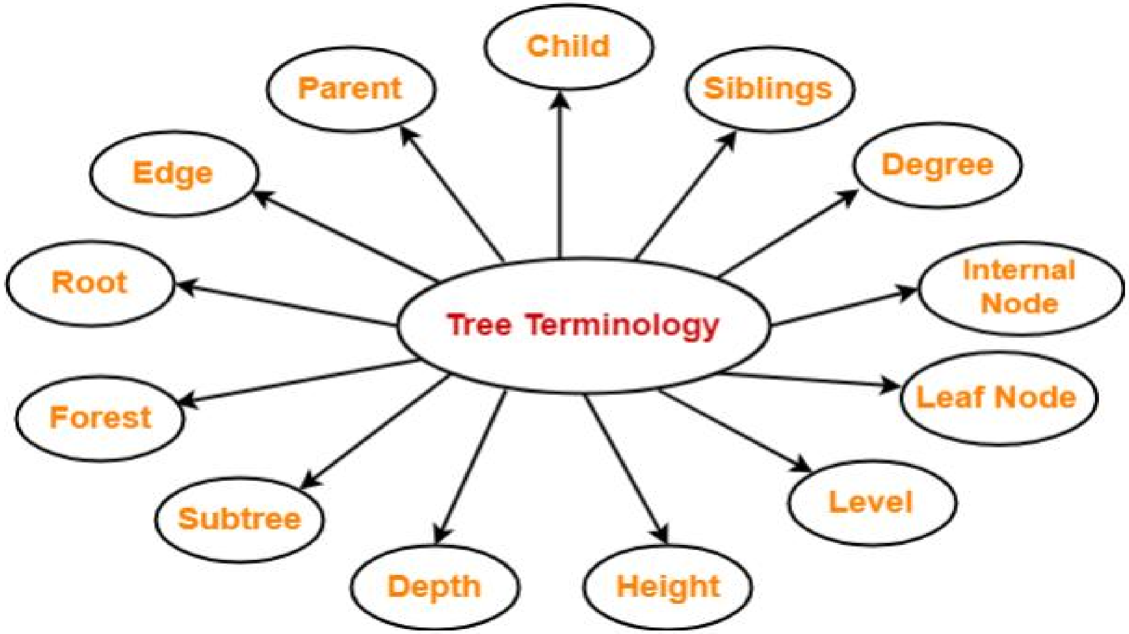
- Soumya

Tree

- Non-linear data structures do not follow a sequential path. They are organized hierarchically, meaning elements in these structures possess parent-child relationships.
- A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines.
- 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.
- With the exception of the top element, each element in a tree has a parent element and zero or more child elements.


Electronics R'Us
R&D
Sales
Purchasing
Manufacturing
Domestic
International
TV
CD
Tuner
Canada
S. America
Overseas
Africa
Europe
Asia
Australia

Tree Terminologies

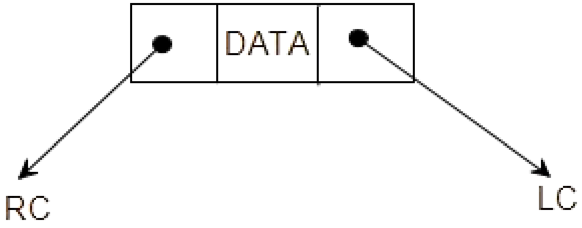


Node

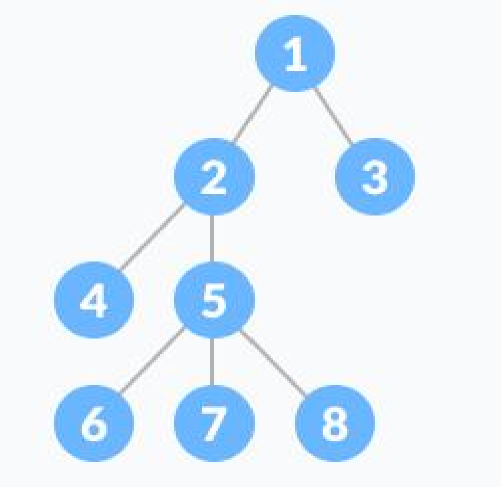
A node is a fundamental part of a tree that stores data and has connections (edges) to other nodes.

It consists of three parts.

- Data: The value stored in the node.
- Left Child: A reference to the left child node.
- Right Child: A reference to the right child node.

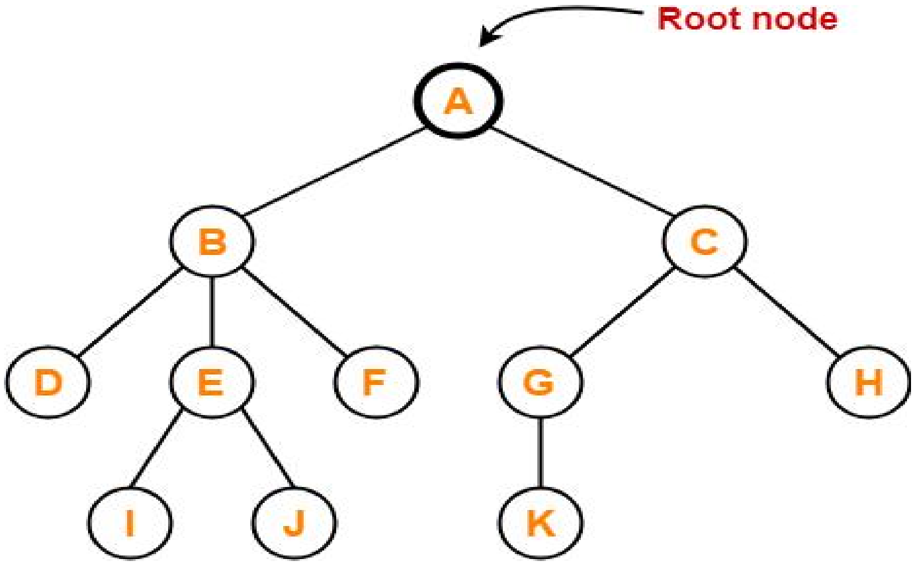


Structure of a node in a tree



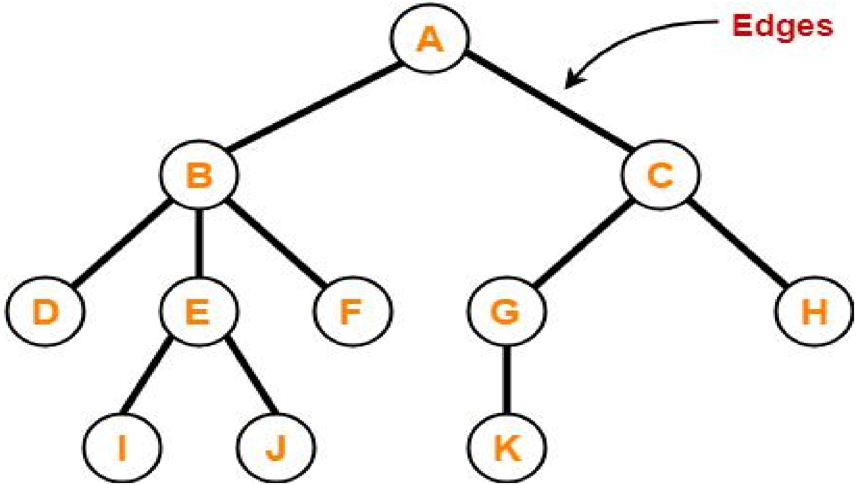
Root

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



Edge

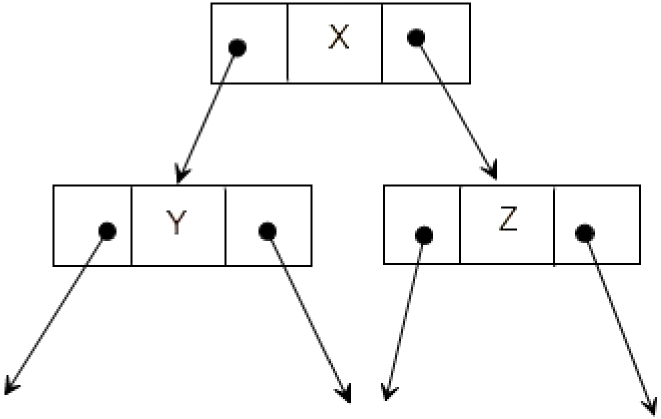
- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.



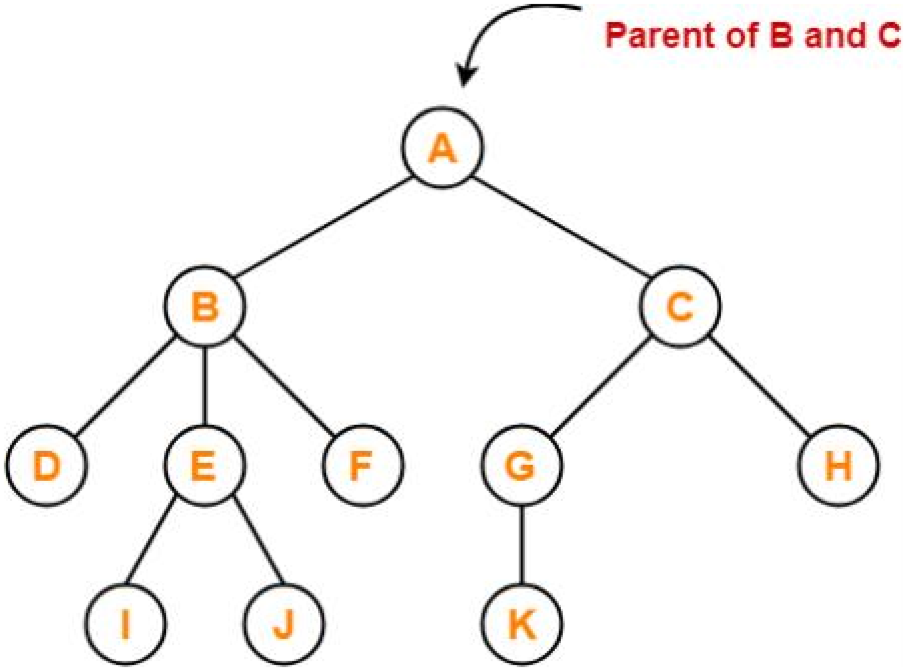
Parent

- The node which has a branch from it to any other node is called as a **parent node**.

- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



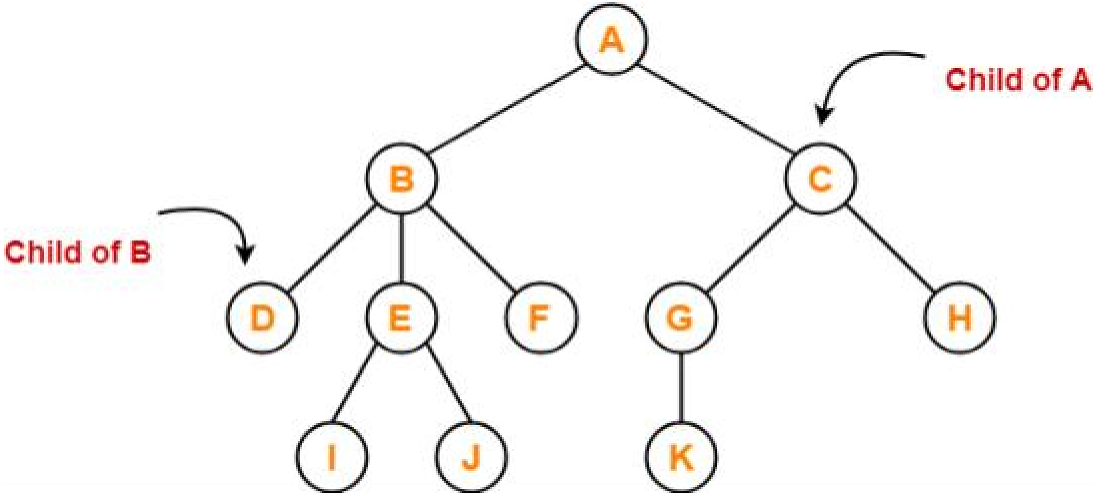
Parent, left child and right child of a node



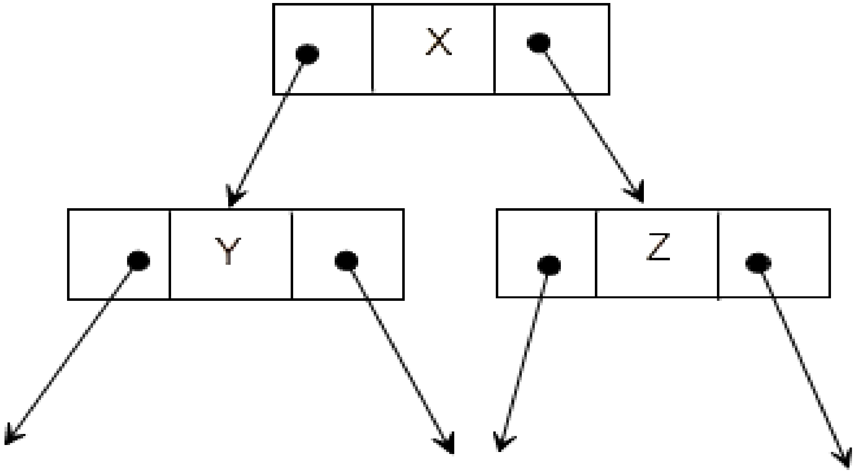
- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

Child

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



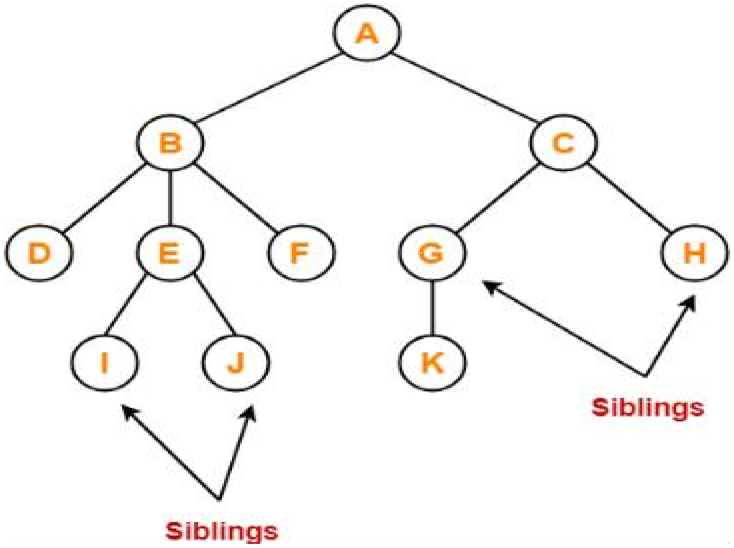
- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

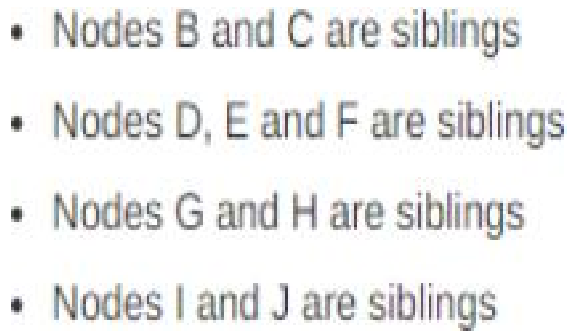


Parent, left child and right child of a node

Sibling

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



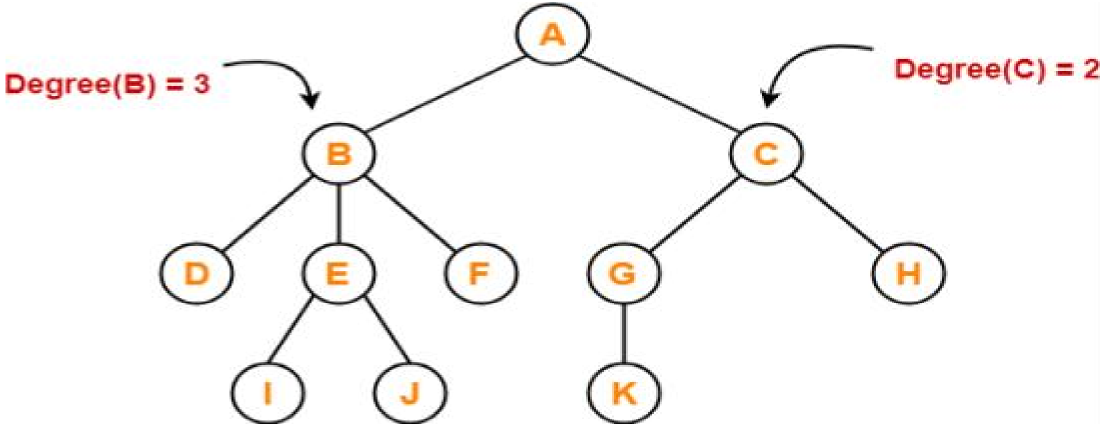


- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

Degree

- **Degree of a 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.

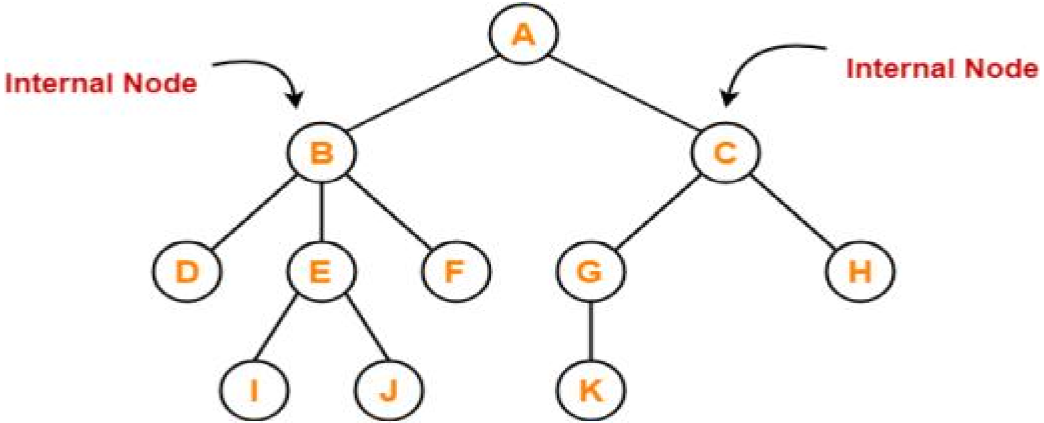


- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

Internal Node

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

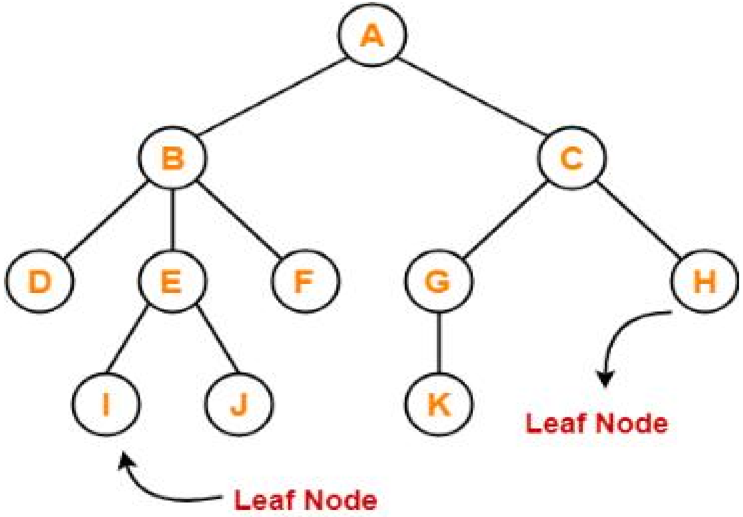
Every non-leaf node is an internal node.



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

Leaf Node

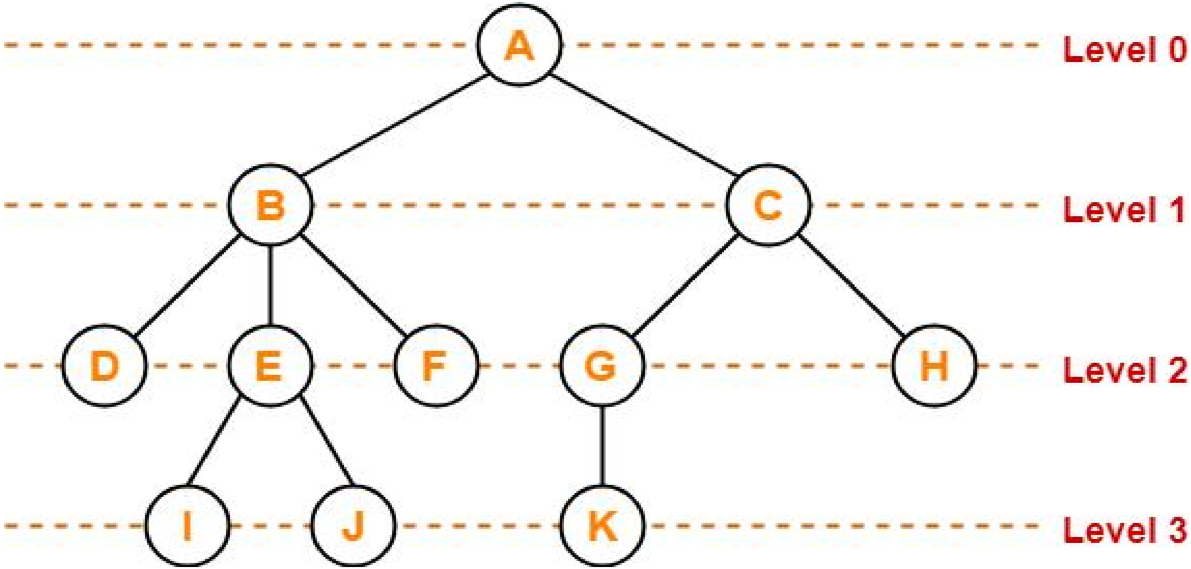
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



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

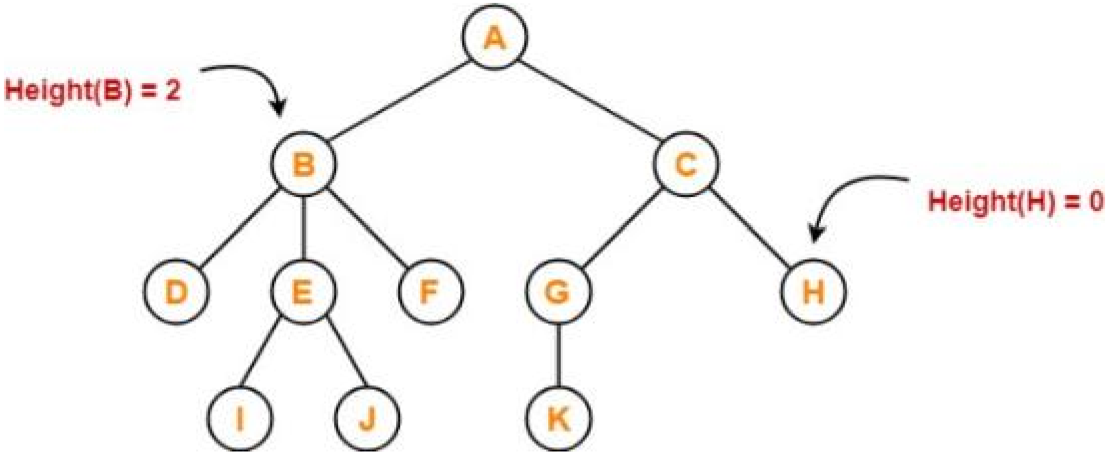
Level

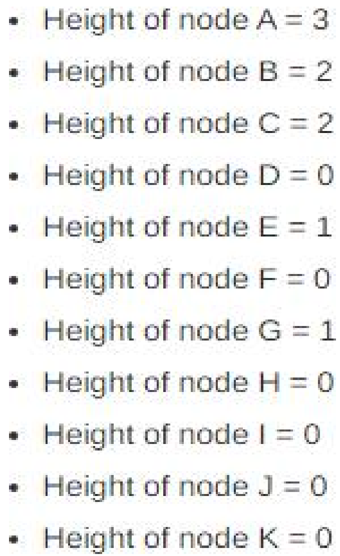
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



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

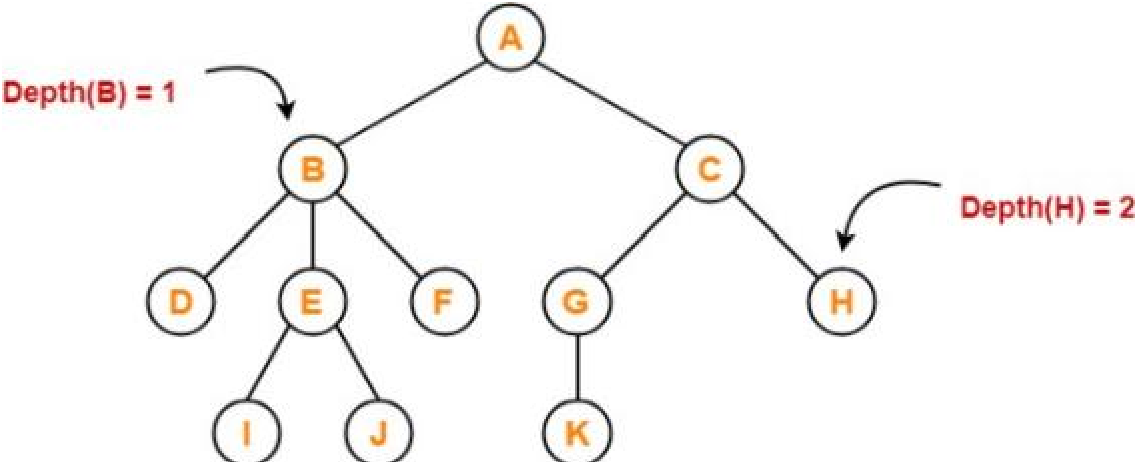


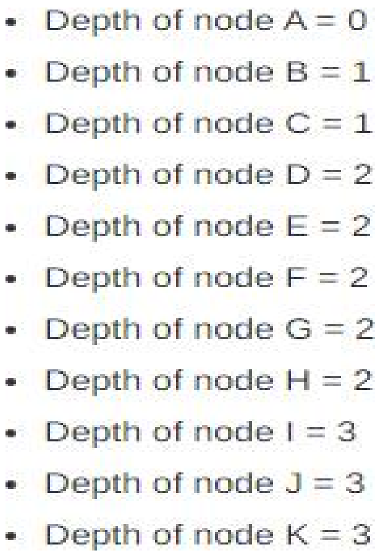


- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

Depth

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is 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.

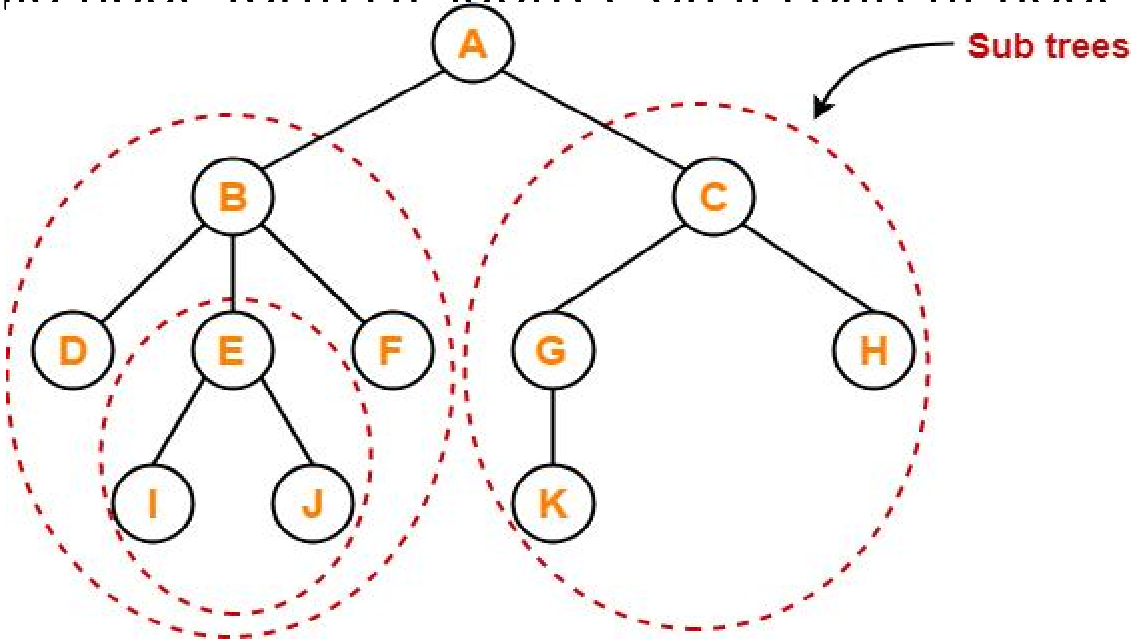




- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

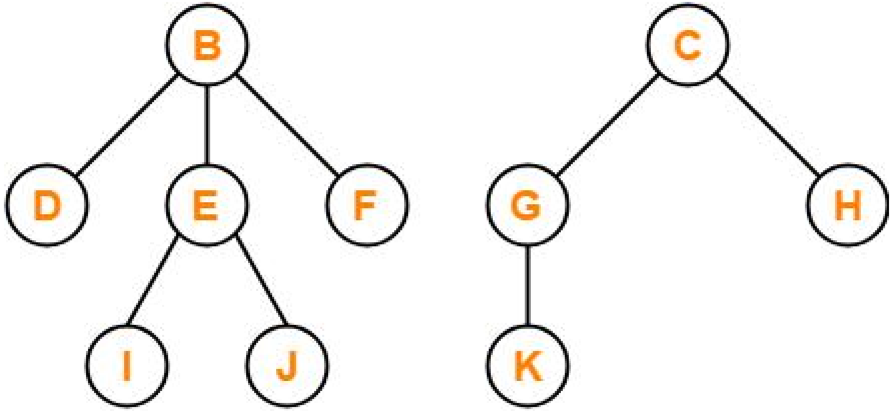
Subtree

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.



Forest

- A forest is a set of disjoint trees.



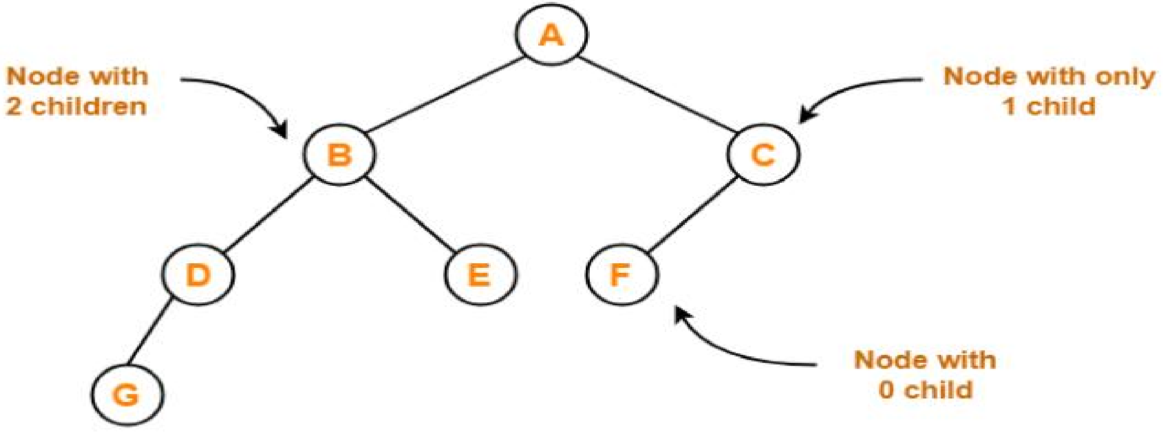
Forest

Types of Tree

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

Binary Tree

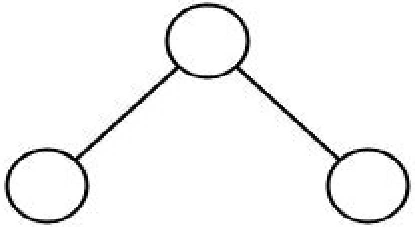
- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.



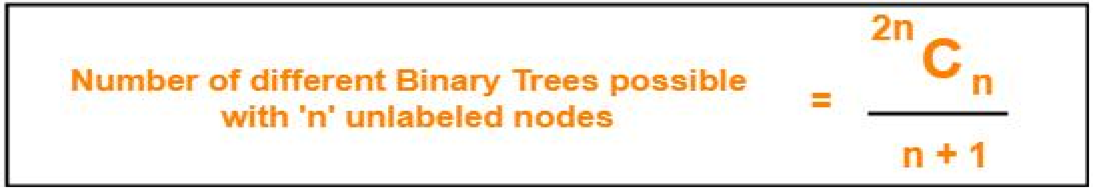
Binary Tree Example

Unlabeled Binary Tree

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



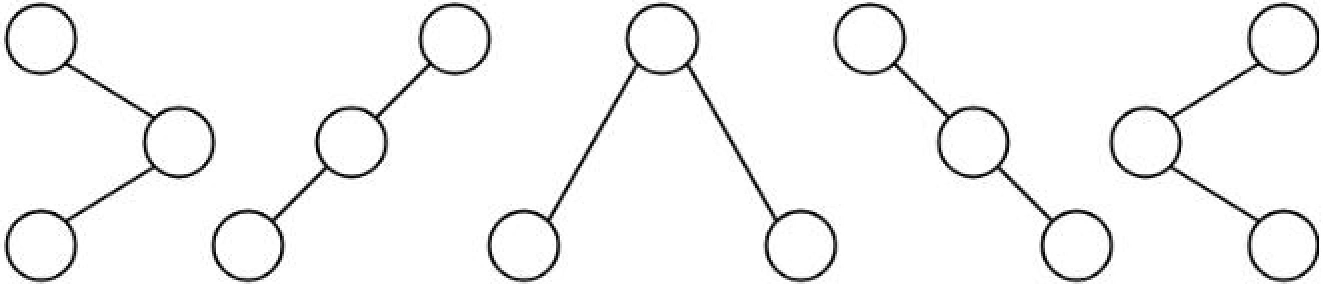
Unlabeled Binary Tree



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

Example

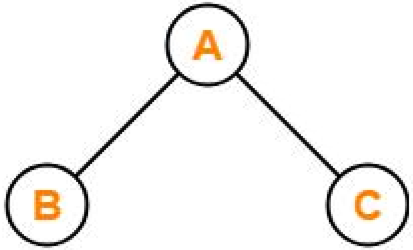
- Consider we want to draw all the binary trees possible
- Number of binary trees possible with 3 unlabeled nodes
- $= {}_{2 \times 3}C_{3} / (3 + 1)$
- $= {}_{6}C_{3} / 4$
- $= 5$



Binary Trees Possible With 3 Unlabeled Nodes

Labeled Binary Tree

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



Labeled Binary Tree

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

Example

- Consider we want to draw all the binary trees possible with 3 labeled nodes.

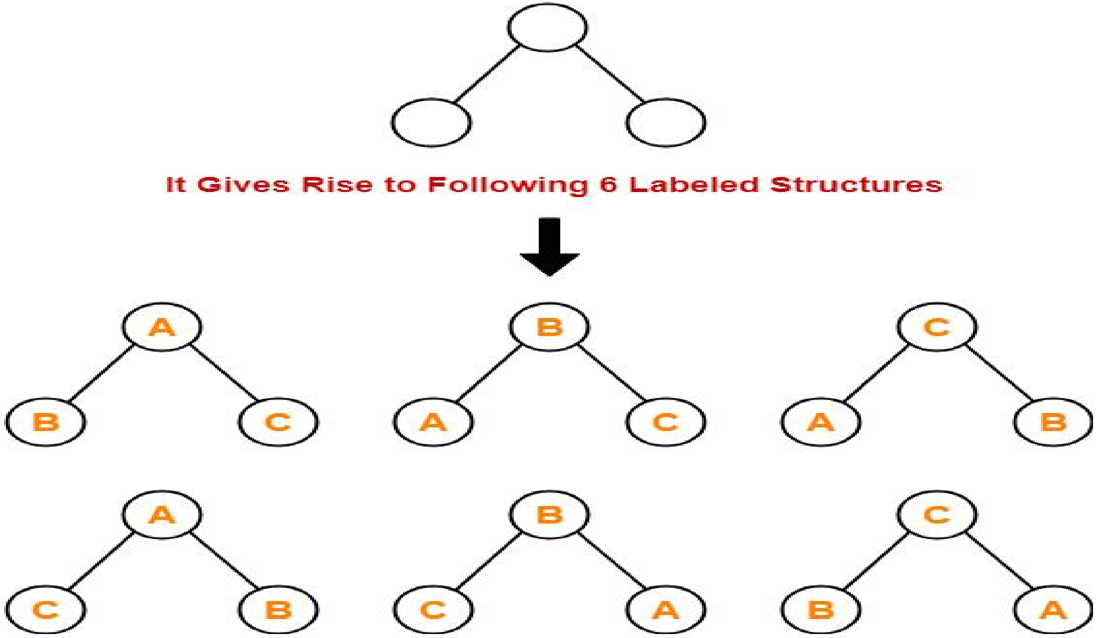
•Number of binary trees possible with 3 labeled nodes

- $= \{ {}_{2 \times 3}C_3 / (3 + 1) \} \times 3!$

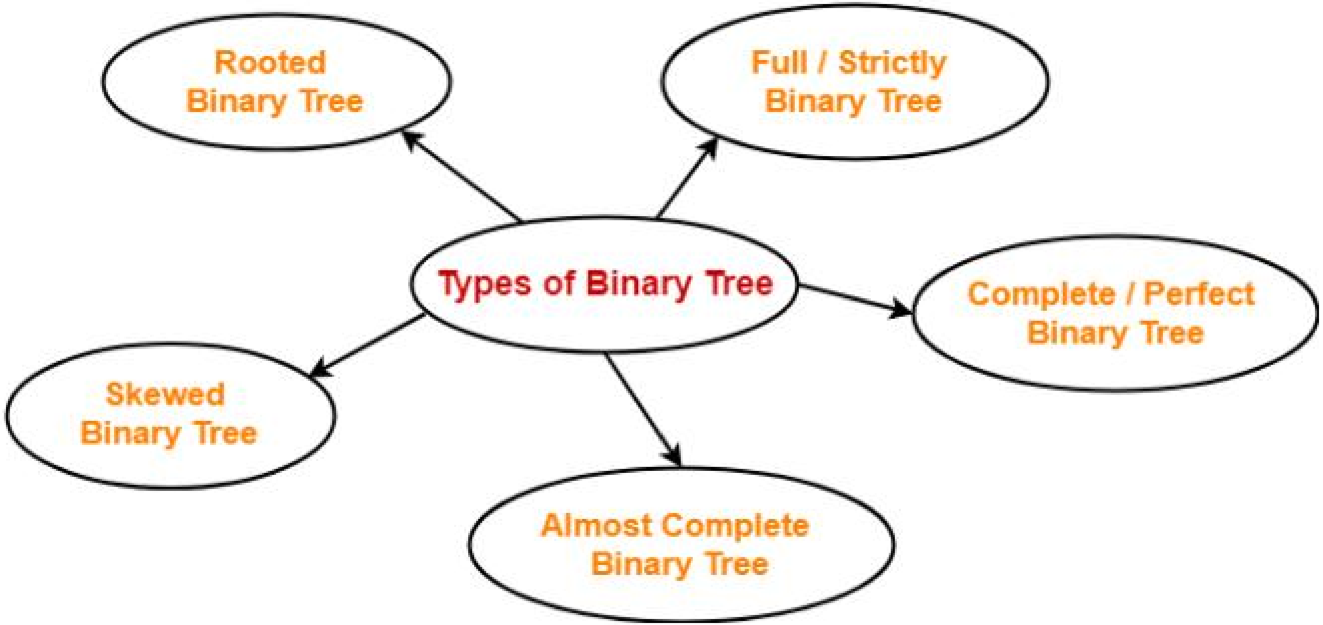
- $= \{ {}_{6}C_3 / 4 \} \times 6$

- $= 5 \times 6$

- $= 30$



Types of Binary Trees



Types of Binary Tree

Rooted Binary Tree

Full / Strictly Binary Tree

Complete / Perfect Binary Tree

Almost Complete Binary Tree

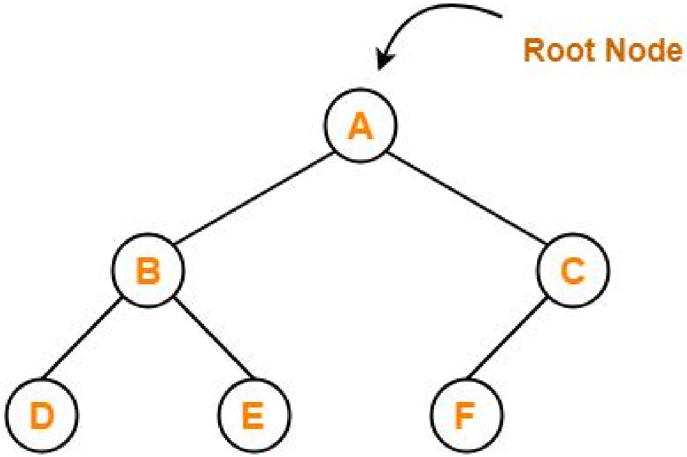
Skewed Binary Tree

Rooted Binary Tree

- A **rooted binary tree** is a binary tree that satisfies the following 2 properties:

– It has a root node.

– Each node has at most 2 children.

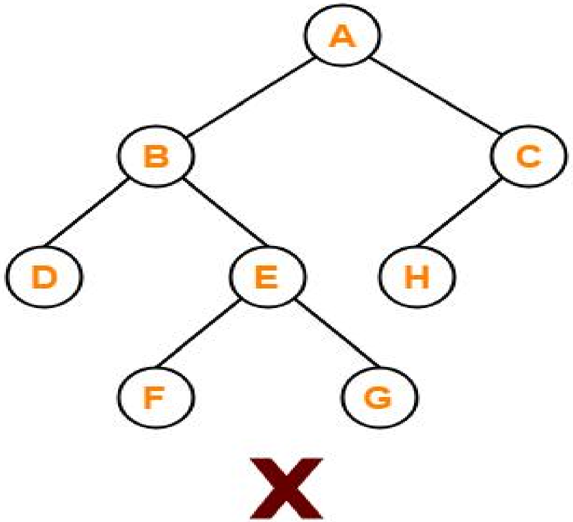


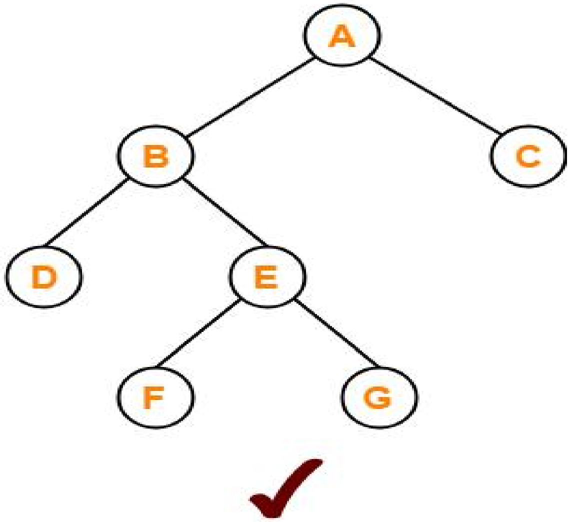
Rooted Binary Tree

Full/Strictly Binary Tree

- A binary tree in which every node has either 0 or 2 children is called as a **Full binary tree**.

- Full binary tree is also called as **Strictly binary tree**.



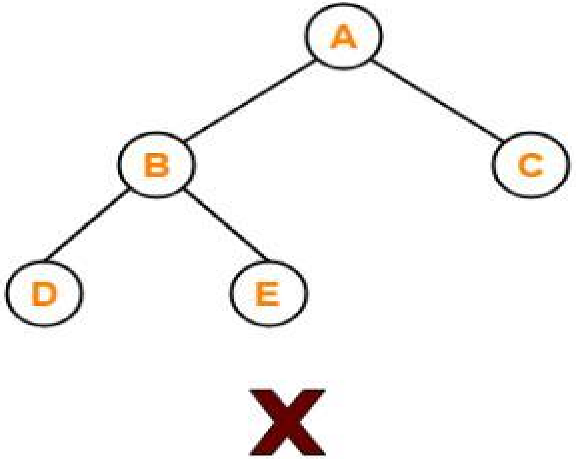


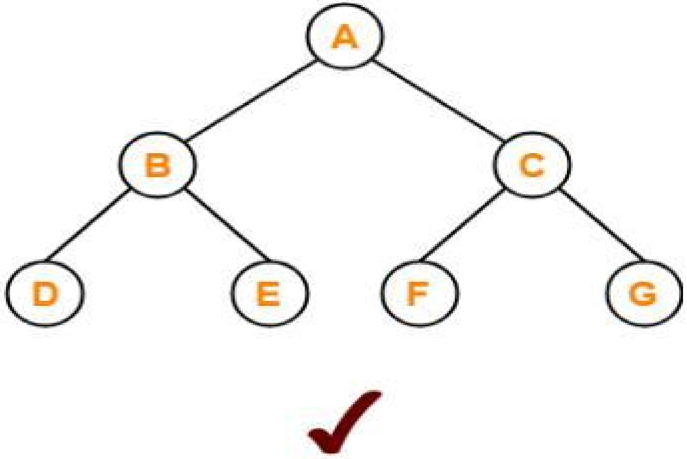
Complete /Perfect Binary Tree

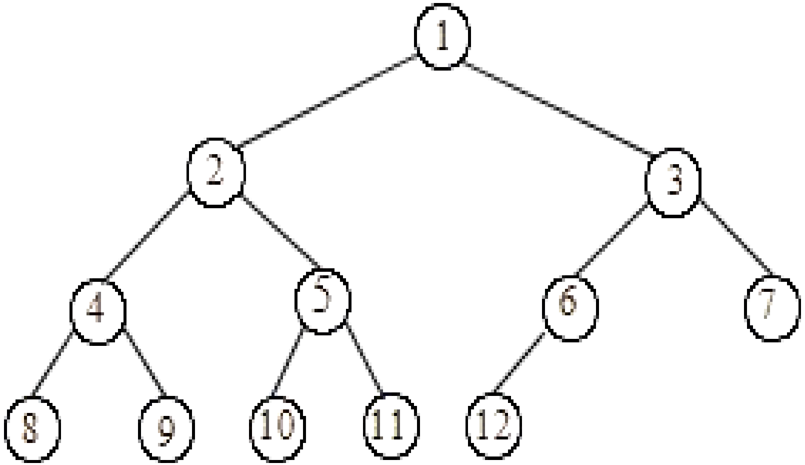
- A **complete binary tree** is a binary tree that satisfies the following 2 properties:

– Every internal node has exactly 2 children.

– All the leaf nodes are at the same level.







Level	Nodes
0	1
1	2
2	4
3	5

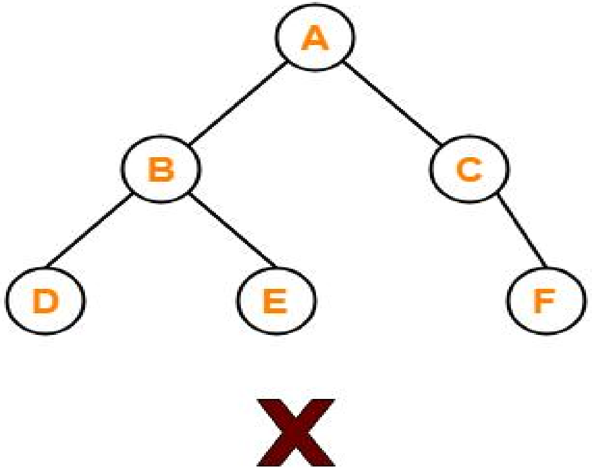
(b) A complete binary tree of height 4

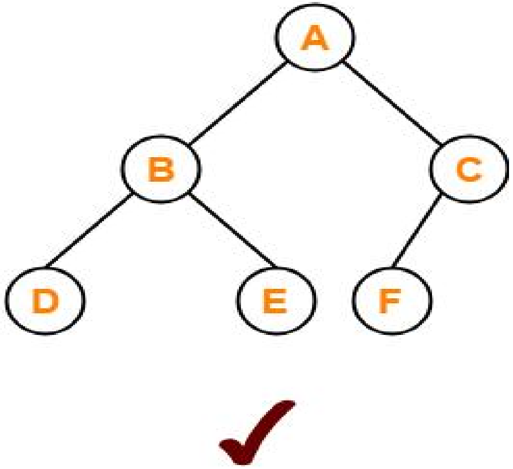
Almost Complete Binary Tree

- An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-

– All the levels are completely filled except possibly the last level.

– The last level must be strictly filled from left to right.





Skewed Binary Tree

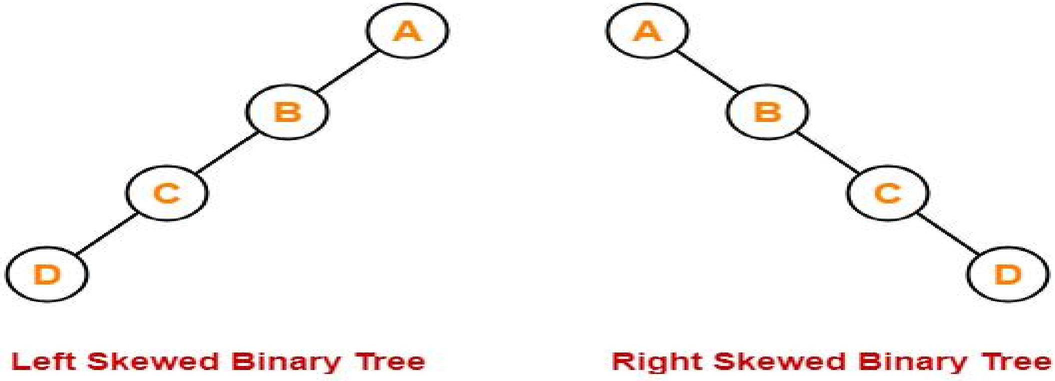
- A **skewed binary tree** is a binary tree that satisfies the following 2 properties-

- All the nodes except one node has one and only one child.

- The remaining node has no child.

OR

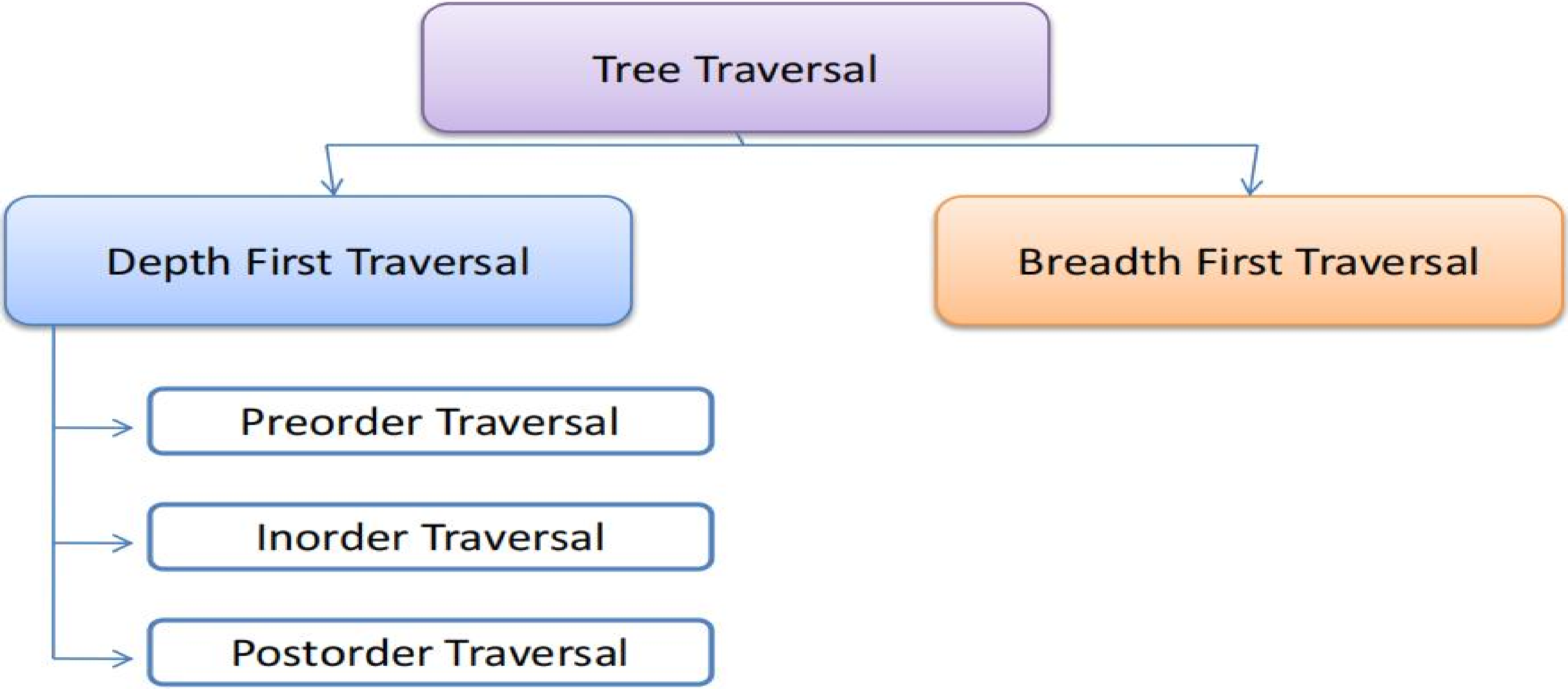
- A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).



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



Tree Traversal

Depth First Traversal

Breadth First Traversal

Preorder Traversal

Inorder Traversal

Postorder Traversal

Depth First Traversal

- Following three traversal techniques fall under

Depth First Traversal-

1. Preorder Traversal

2. Inorder Traversal

3. Postorder Traversal

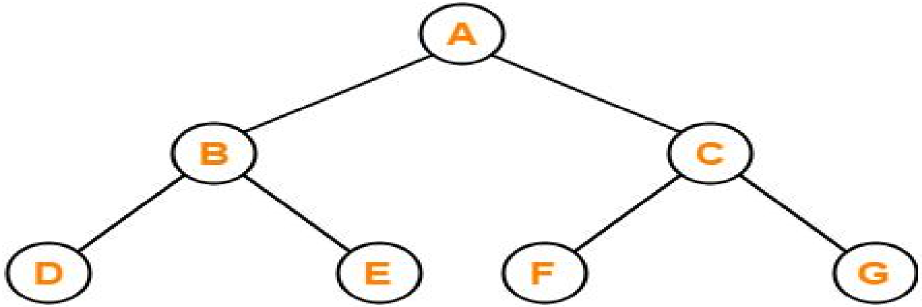
Preorder Traversal

- **Algorithm-**

– Visit the root

– Traverse the left sub tree i.e. call Preorder (left subtree)

– Traverse the right subtree i.e. call Preorder (right subtree)



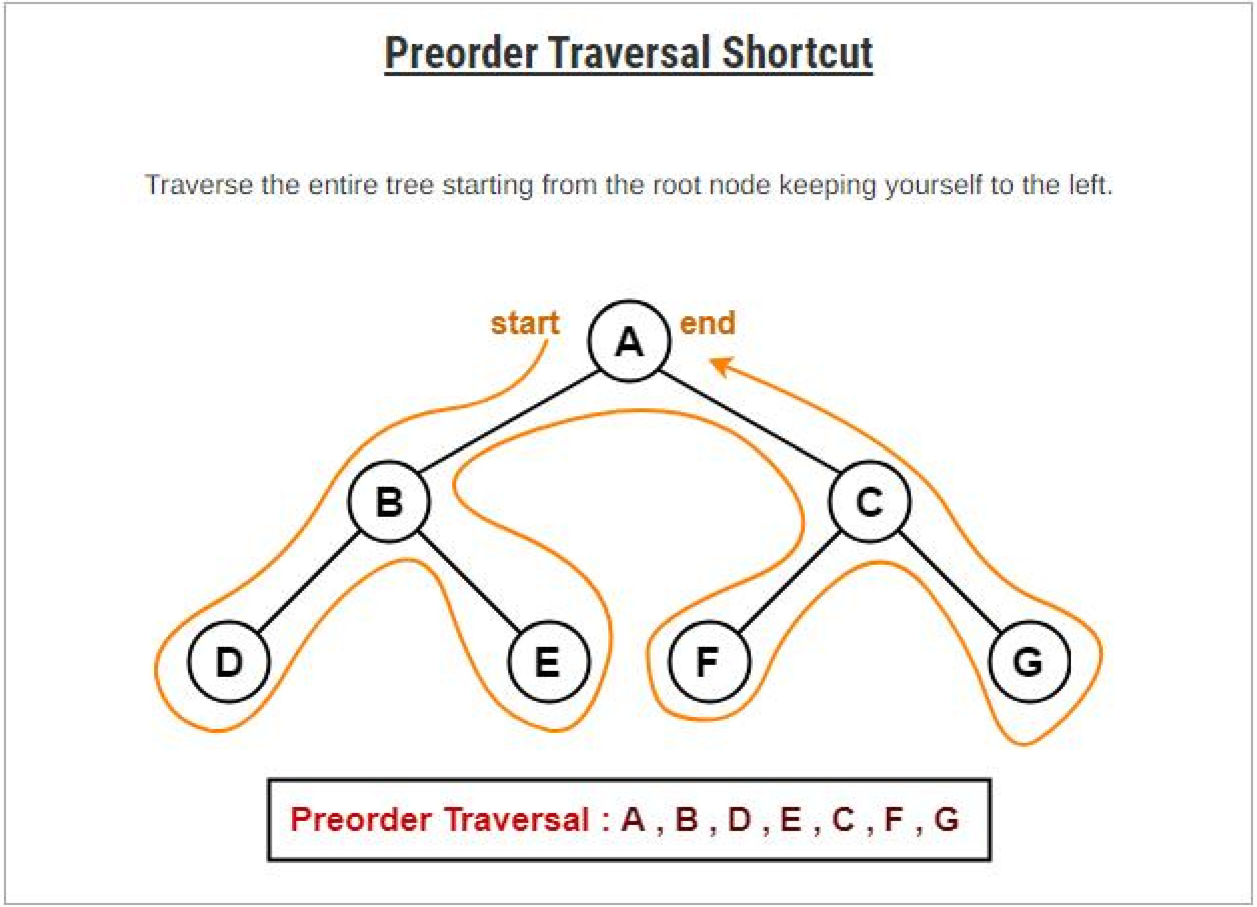
Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal

The applications of preorder traversal include -

– It is used to create a copy of the tree.

– It can also be used to get the prefix expression of an expression tree.



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

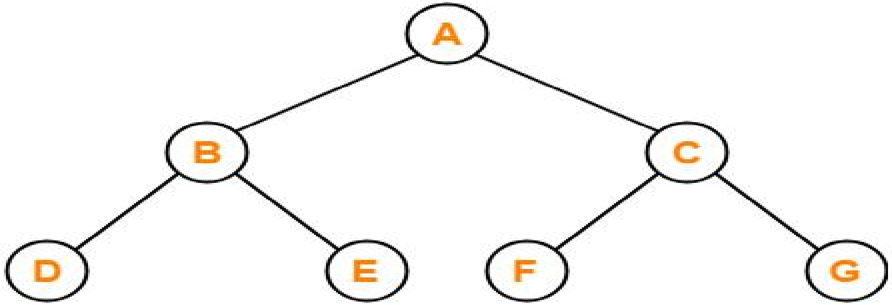
Inorder Traversal

- **Algorithm-**

– Traverse the left sub tree i.e. call Inorder (left subtree)

– Visit the root

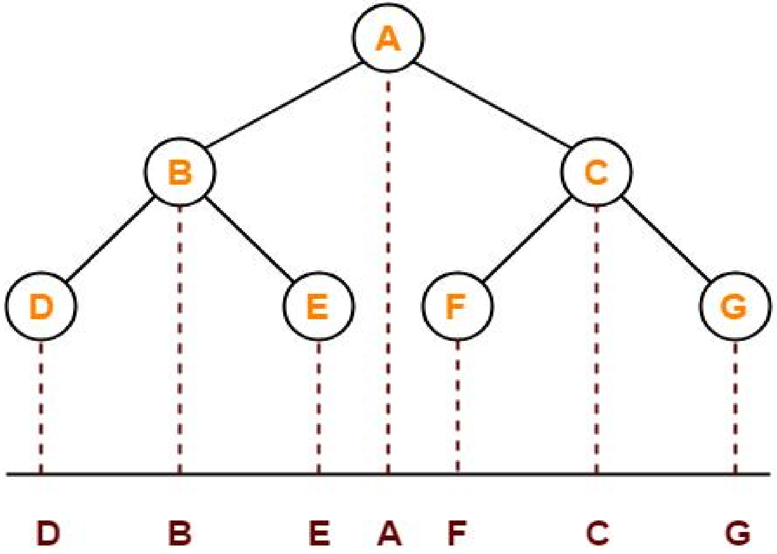
– Traverse the right subtree i.e. call Inorder (right subtree)



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

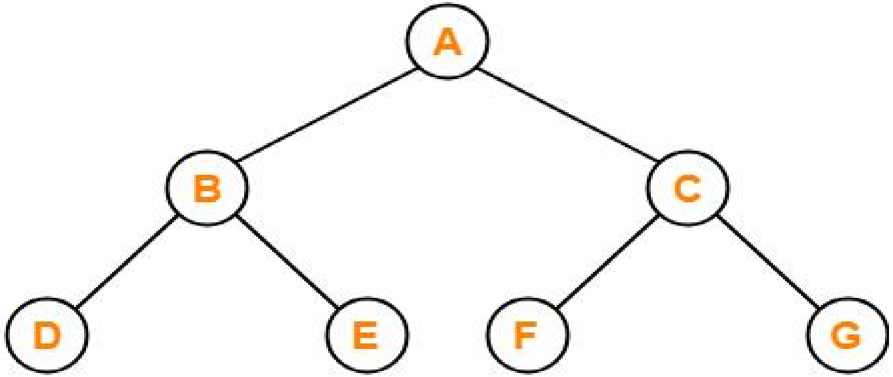
Postorder Traversal

- **Algorithm-**

– Traverse the left sub tree i.e. call Postorder (left sub tree)

– Traverse the right sub tree i.e. call Postorder (right sub tree)

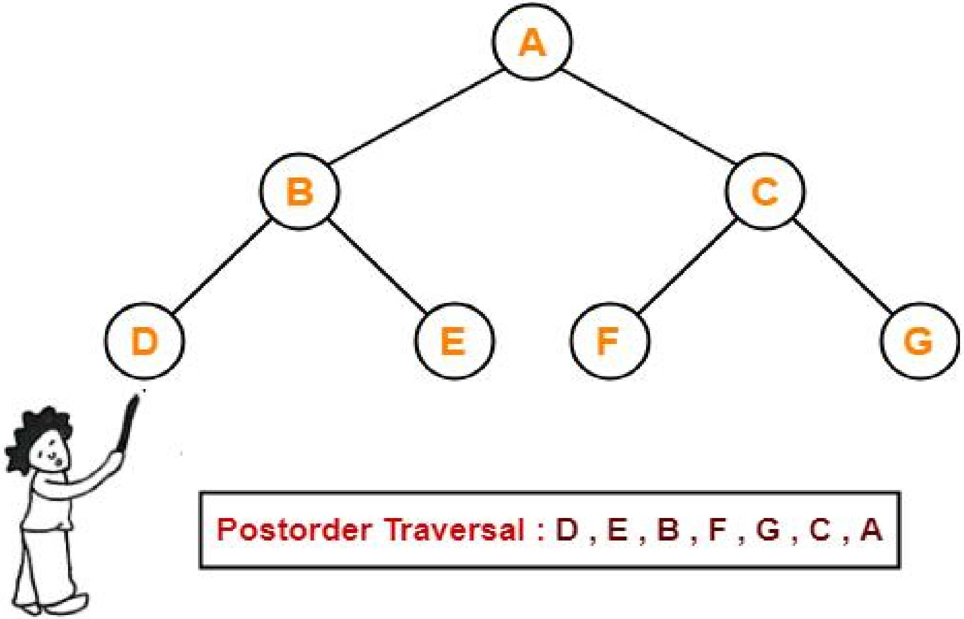
– Visit the root



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

Complexities of Tree Traversal

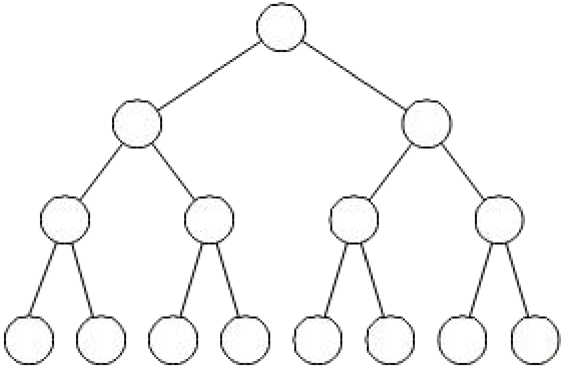
- The time complexity of tree traversal techniques discussed above is O(n), where 'n' is the size of binary tree.

- Whereas the space complexity of tree traversal techniques discussed above is O(1) if we do not consider the stack size for function calls.

- Otherwise, the space complexity of these techniques is O(h), where 'h' is the tree's height.

Binary Trees:Properties

- Property: Maximum number of nodes in a level of a binary tree
- In any binary tree, maximum number of nodes on level l is 2l, where l ≥ 0.

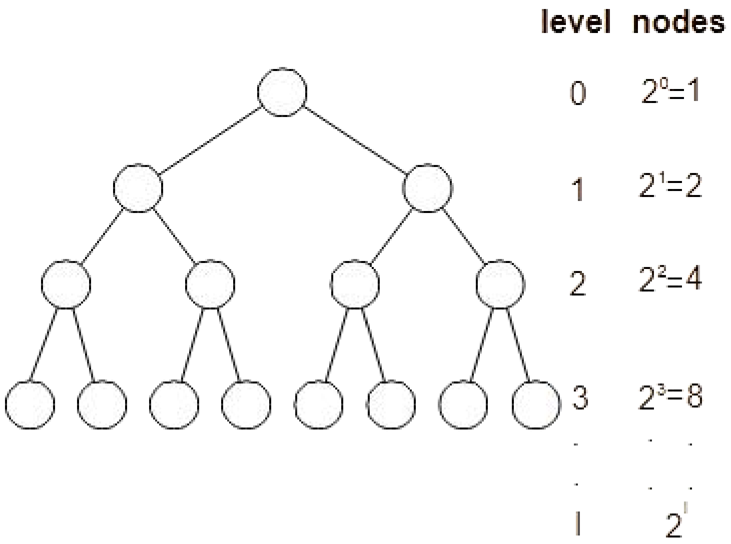


level	nodes
0	$2^0=1$
1	$2^1=2$
2	$2^2=4$
3	$2^3=8$
.	. .
.	. .
l	2^l

Binary Trees:Properties

Property: Maximum number of nodes in a binary tree.

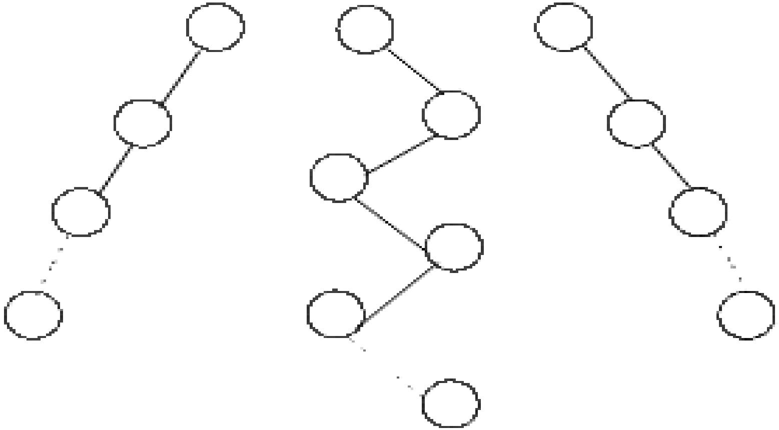
Maximum number of nodes possible in a binary tree of height h is 2h – 1.



Binary Trees:Properties

Property: Minimum number of nodes in a binary tree.

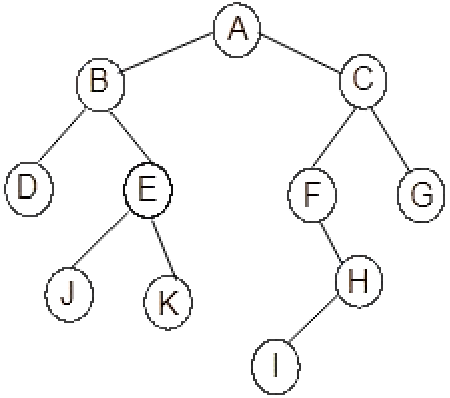
Minimum number of nodes possible in a binary tree of height h is h.

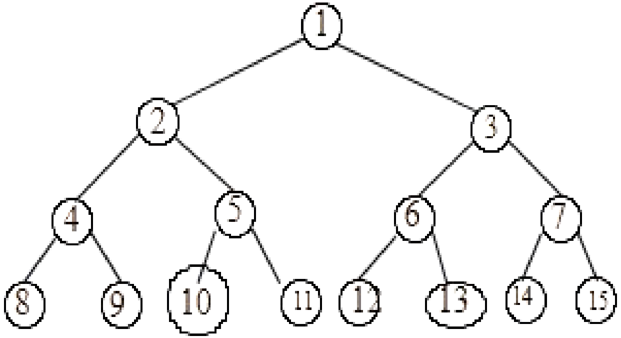


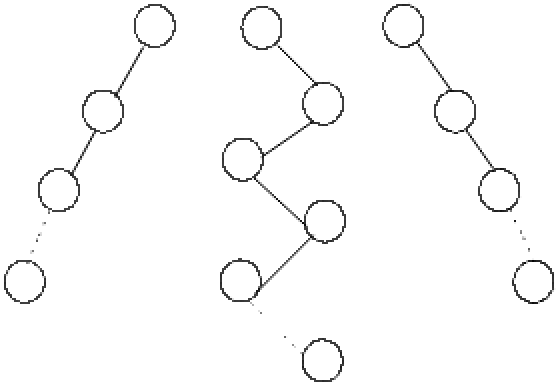
Binary Trees:Properties

Property: Number of nodes and edges in a binary tree

For any non-empty binary tree, if n is the number of nodes and e is the number of edges, then n = e + 1.

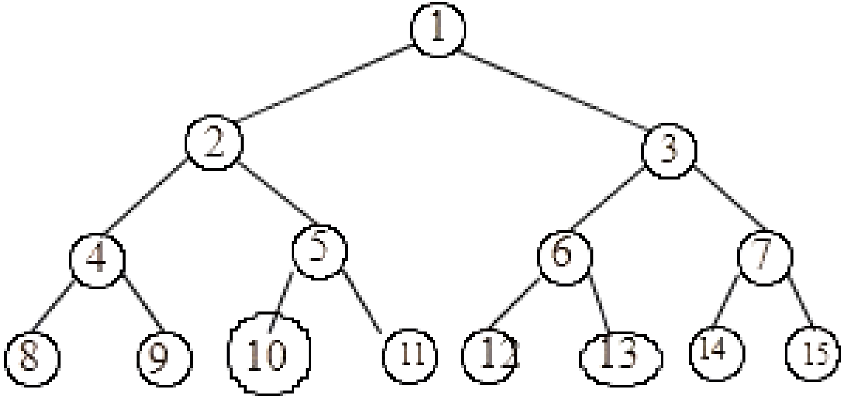






Binary Trees:Properties

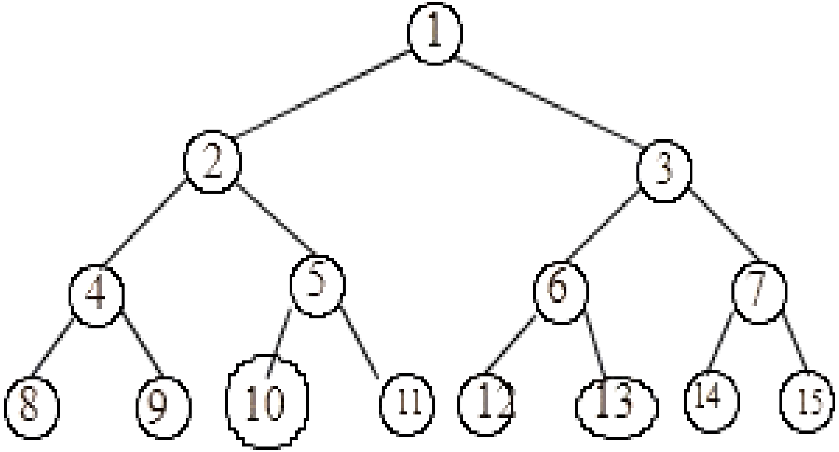
- Number of leaf nodes and non-leaf nodes in a binary tree
- For any non-empty binary tree T, if n0 is the number of leaf nodes (degree = 0) and n2 is the number of internal node (degree = 2), then n0 = n2 + 1.



Binary Trees:Properties

Height of a complete binary tree

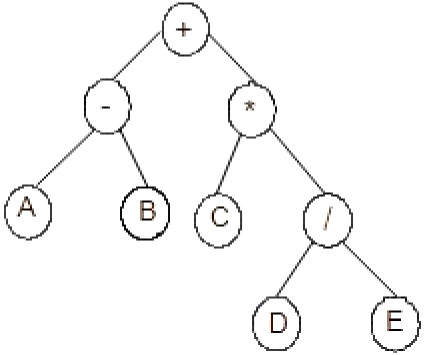
Height of a complete binary tree with n number of nodes is $\lceil log_2(n+1) \rceil$



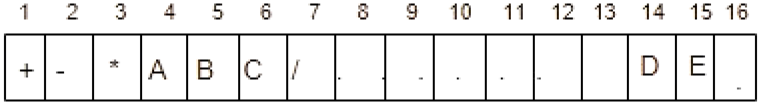
Representation of Binary Trees

- ➢ Linear representation
  - ○ Using array
- ➢ Linked representation
  - ○ Using linked list structure

Using array

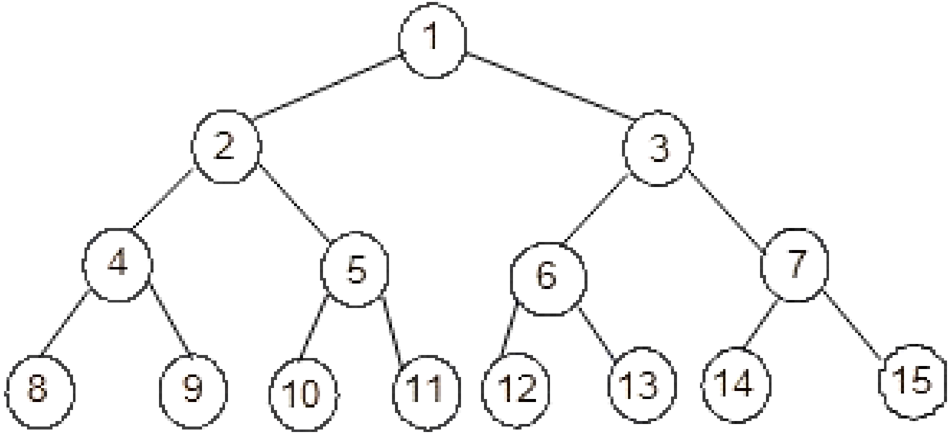


(a) A blnary tree



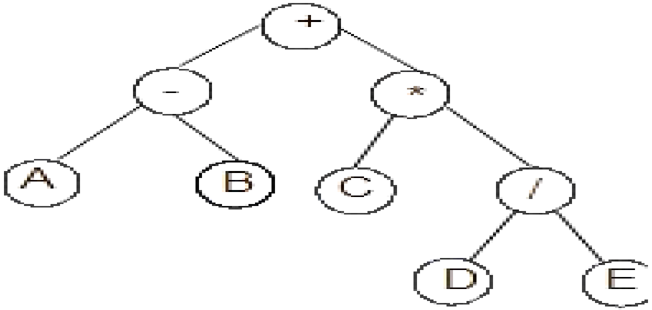
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
+	-	*	A	B	C	/	.	.	.	.	.		D	E	.

(b) Array representatlon of the blnary tree

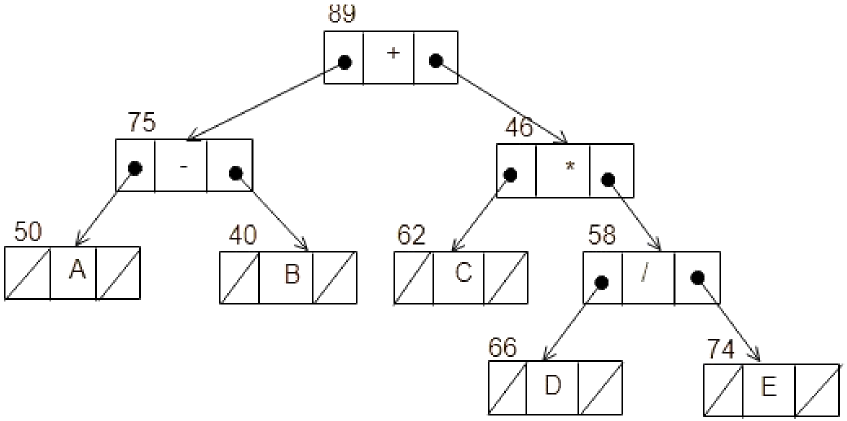


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Linked List



(a) A binary tree



(c) Logical view of the linked representation of a binary tree

Binary Tree :Implementation

```
// creating a node that holds the data, address of the left child, and the address of the
right child

class Node {

 int key;

 Node left, right;

 //setting data in the node

 public Node(int item) {

   key = item;

   //setting left and right child equal to NULL

   left = right = null;

 }
```

```
class BinaryTree {
 Node root;
  //inserting data into the binary tree
 BinaryTree(int key) {
   root = new Node(key);
 }
 //set root NULL when the binary tree is created for the first time
 BinaryTree() {
   root = null;
 }
}
```

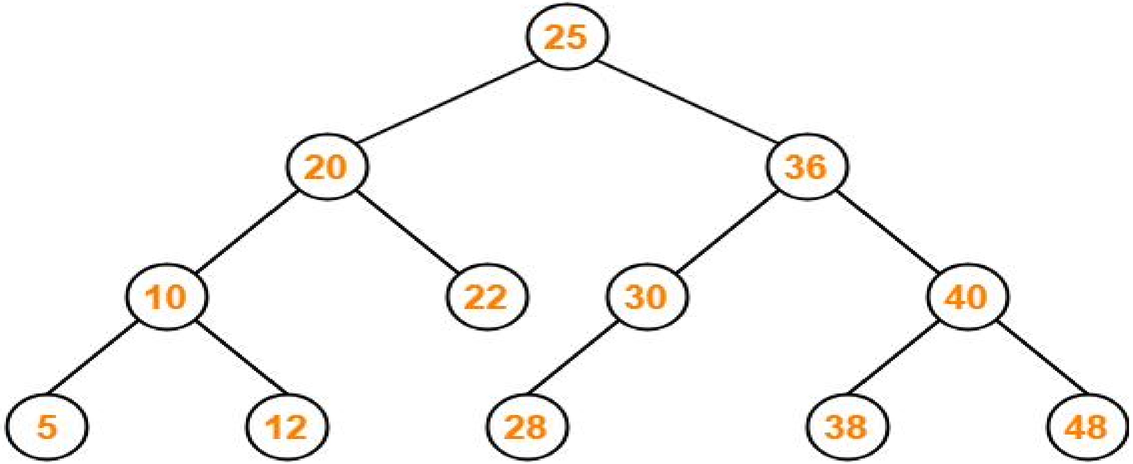
```
public static void main(String[] args) {
//creating a new instance of Binary Tree
   BinaryTree tree = new BinaryTree();
  //inserting into the binary tree
   tree.root = new Node(10);
   tree.root.left = new Node(20);
   tree.root.right = new Node(30);
 }
```

Binary search trees

- 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



Binary search tree

- ❑ Binary search tree is a non-linear data structure in which one node is connected to n number of nodes. It is a node- based data structure.
- ❑ A node can be represented in a binary search tree with three fields, i.e., data part, left-child, and right-child. A node can be connected to the utmost two child nodes in a binary search tree, so the node contains two pointers (left child and right child pointer).
- ❑ Every node in the left subtree must contain a value less than the value of the root node, and the value of each node in the right subtree must be bigger than the value of the root node.

Example

- Construct a Binary Search Tree (BST) for the following sequence of numbers:

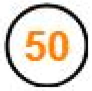
50, 70, 60, 20, 90, 10, 40, 100

- When elements are given in a sequence,

– Always consider the first element as the root node.

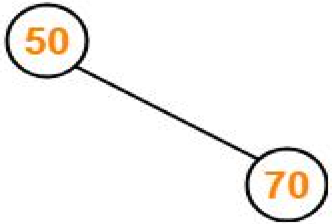
– Consider the given elements and insert them in the BST one by one.

Insert 50-



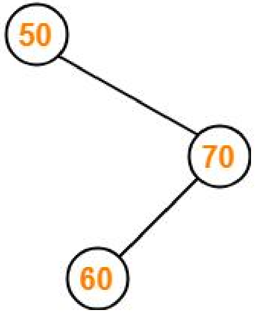
Insert 70-

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



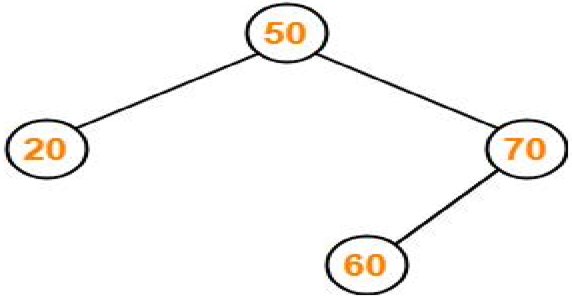
Insert 60-

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



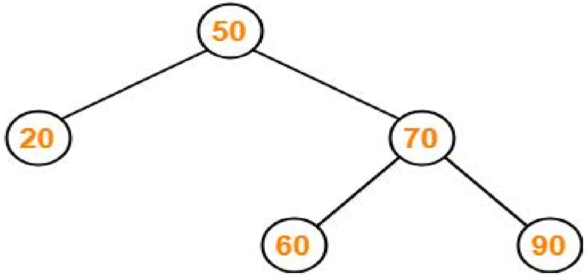
Insert 20-

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



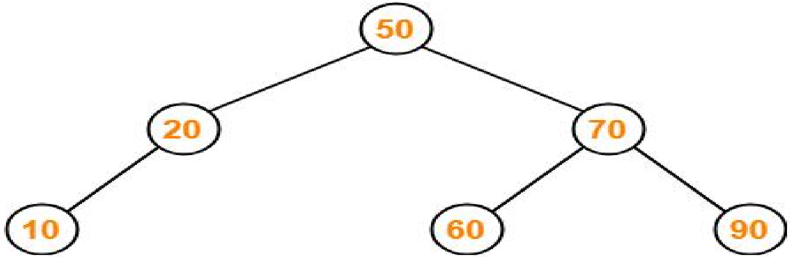
Insert 90-

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



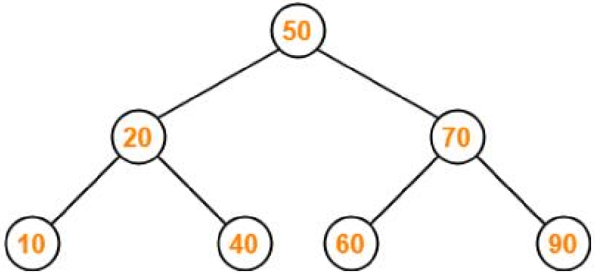
Insert 10-

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



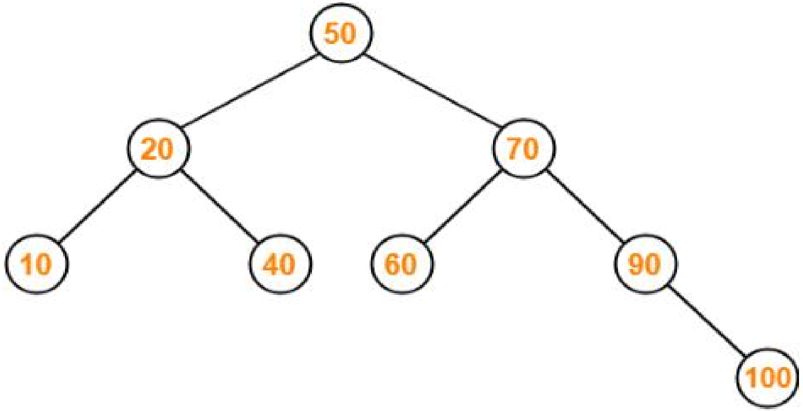
Insert 40-

- 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 subtree and right subtree of the root respectively is _____.

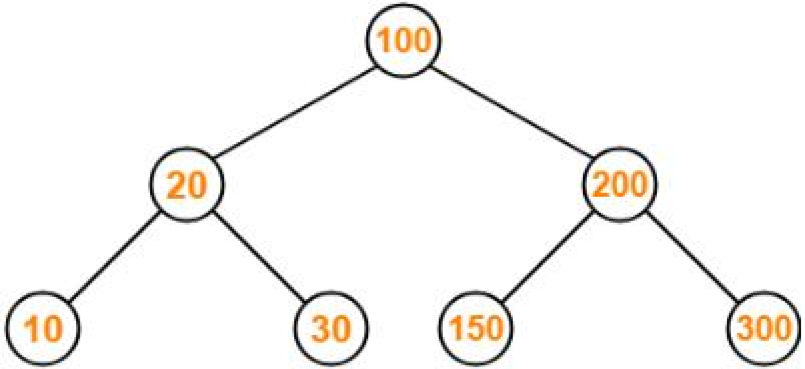
– (4, 7)

– (7, 4)

– (8, 3)

– (3, 8)

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

Use Binary Search Trees

Efficient Search Operations: Due to their structure, BSTs allow for efficient search operations, similar to those in sorted arrays but withfaster insertion and deletion capabilities.

Sorted Order Retrieval: BSTs can retrieve the elements in sorted order using in-order traversal. This feature is beneficial for algorithms that require sorted elements without needing to sort an entire array or list.

Flexibility in Size: Unlike static data structures (like arrays), BSTs are dynamic and can easily expand or shrink in size, accommodating new elements or removing existing ones.

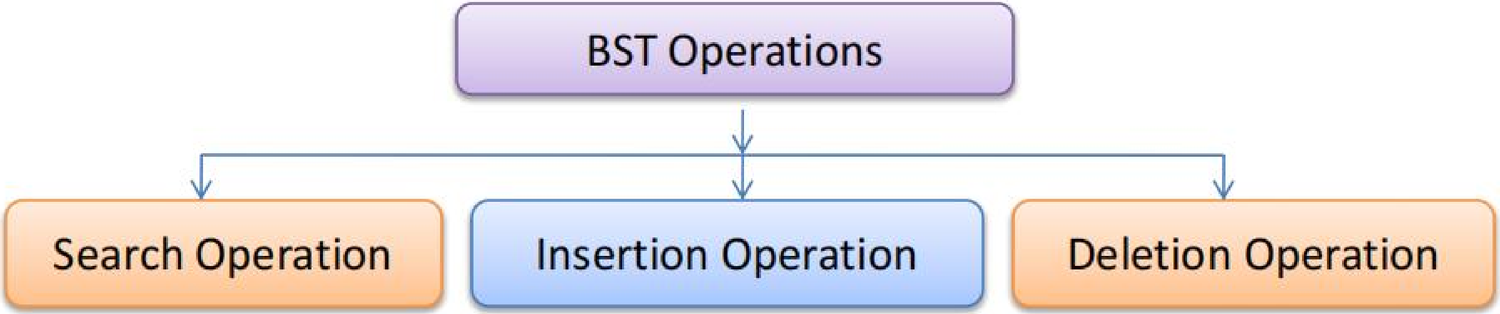
Balanced BSTs: Certain types of BSTs (like AVL trees or Red-Black trees) self-balance, ensuring that the tree remains proportioned and that operations like insertion, deletion, and search take logarithmic time.

Applications

- ➢ BSTs are instrumental in various applications, including:
- ➢ Database indexing, where quick search, insertion, and deletion are crucial.
- ➢ Dynamic sorting of datasets that frequently change.
- ➢ Implementing associative arrays (maps) efficiently.

BST Operations

- Commonly performed binary search tree operations are:



BST Operations

Search Operation

Insertion Operation

Deletion Operation

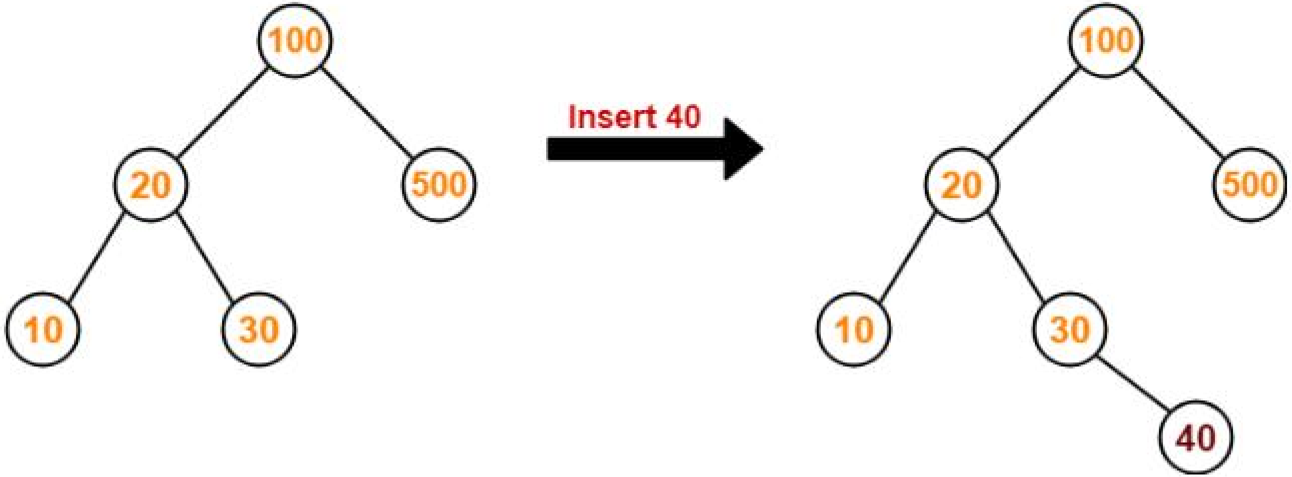
Insertion operation

- 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 key to 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.

1. if node == null, create a new node with the value of the key field equal to K. We return this newly created node directly from here.
2. if K <= node.key, it means K must be inserted in the left subtree of the current node. We repeat(recur) the process from step 1 for the left subtree.
3. else K > node.key, which means K must be inserted in the right subtree of the current node. We repeat(recur) the process from step 1 for the right subtree.
4. Return the current node.

Let’s say we are trying to insert the following integers in the Binary Search Tree.

10 15 5 8 18 12 10

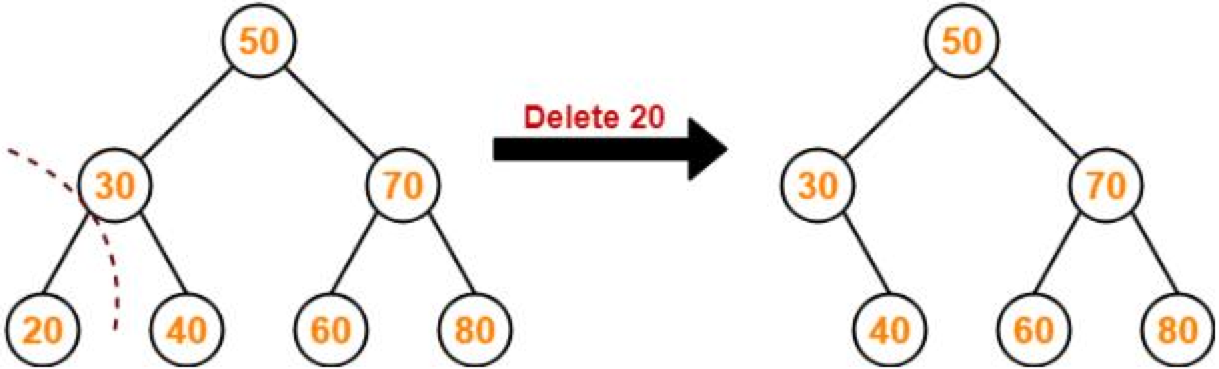
Deletion Operation

- 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 No Child (Leaf Node)

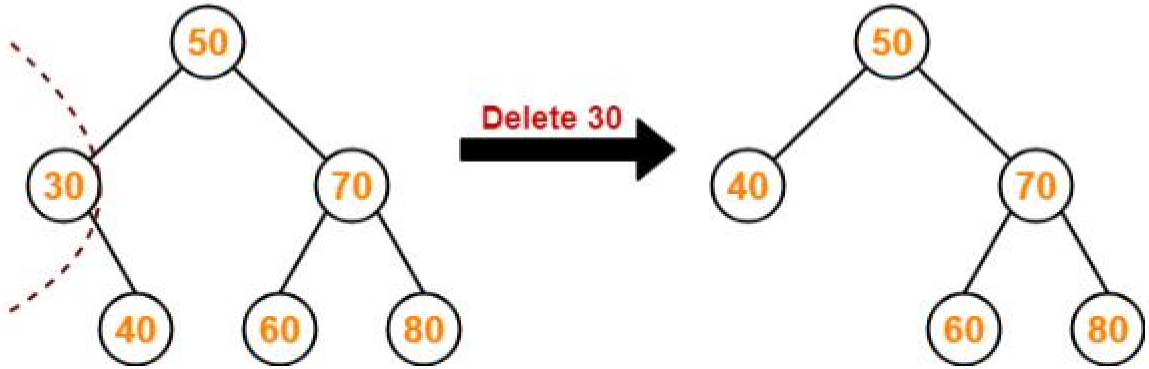
- Just remove / disconnect the leaf node that is

to deleted from the tree.



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

- Consider the following example where node with value = 30 is deleted from the BST.



Case-03: Deletion Of A Node Having Two Children

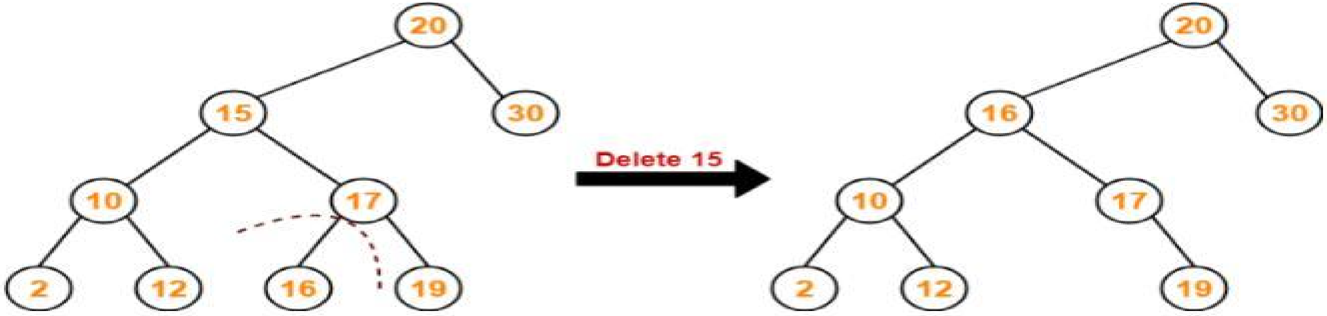
- Consider the following example where node with value = 15 is deleted from the BST

- Method-1:

– Visit to the right subtree of the deleting node.

– Pluck the least value element called as inorder successor.

– Replace the deleting element with its inorder successor

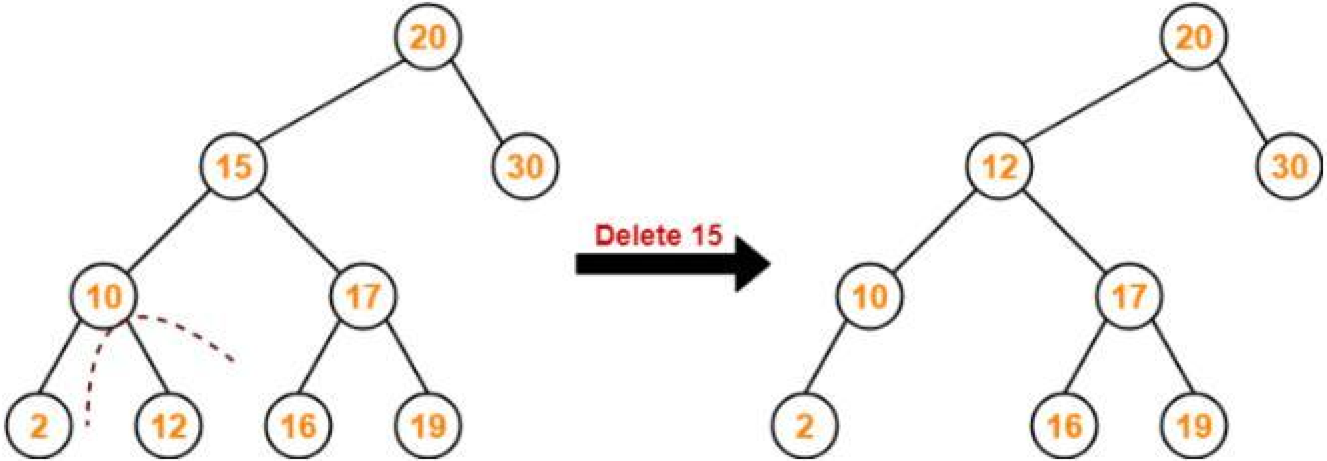


Method-2:

– Visit to the left subtree of the deleting node.

– Pluck the greatest value element called as inorder successor.

– Replace the deleting element with its inorder successor.



Search Operation

- Search Operation is performed to search a particular element in the Binary Search Tree.

- For searching a given key in the BST,

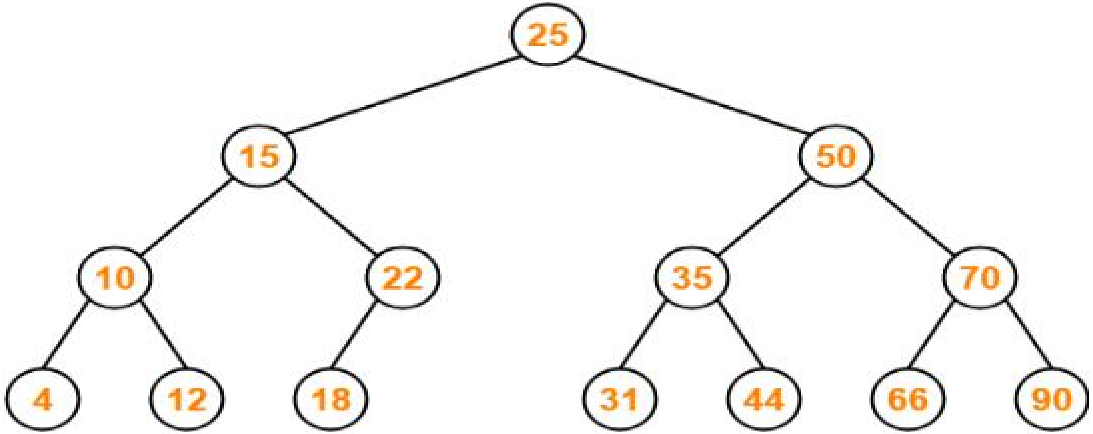
– Compare the key with the value of root node.

– 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-

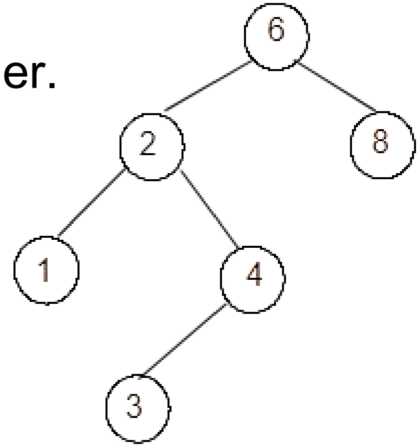


- We start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present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

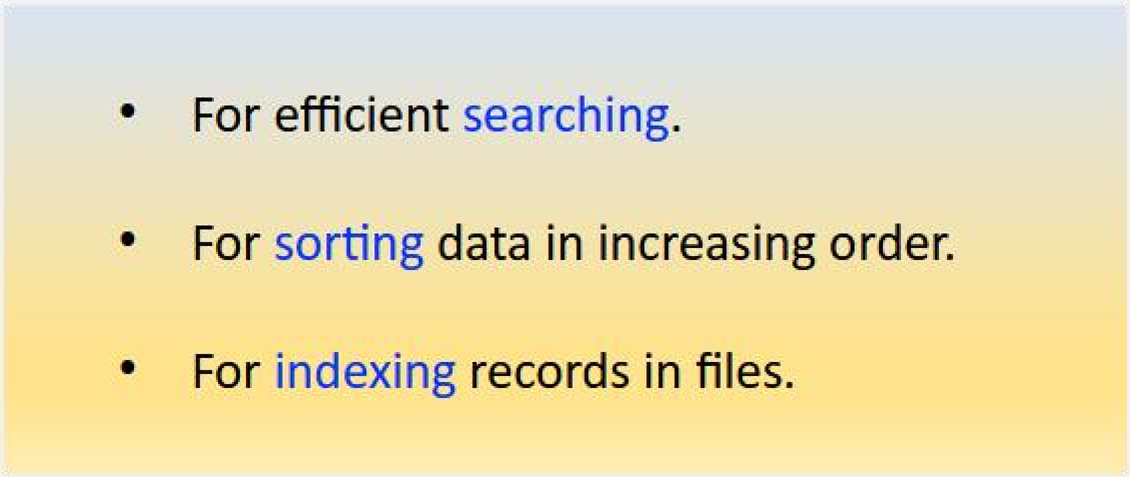
Traversal

- All the traversal operations for binary tree are applicable to binary search trees without any alteration.
- It can be verified that inorder traversal on a binary search tree will give the sorted order of data in ascending order.
- If we require to sort a set of data, a binary search tree can be built with those data and then inorder traversal can be applied.
- This method of sorting is known as binary sort and this is why binary search tree is also termed as binary sorted tree.
- This sorting method is considered as one of the efficient sorting method.

- Inorder traversal on a BST gives the data in ascending order.
- The minimum value is at the left-most node.
- The maximum value is at the right-most node.



Applications of BST



- For efficient searching.
- For sorting data in increasing order.
- For indexing records in files.

Time complexity of BST

- Sorting
  - Complexity ≈ Building a binary search tree

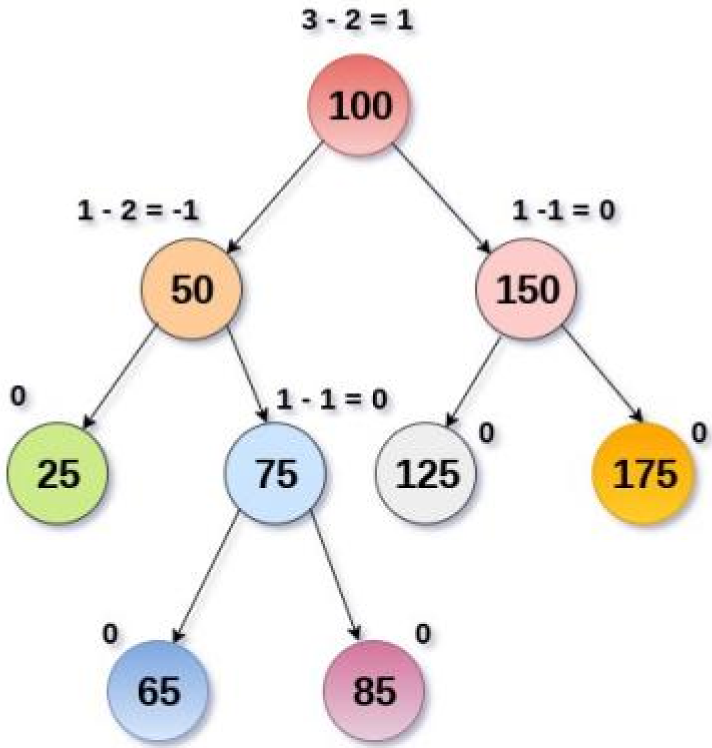
    ≈ O(nlog2n)

- Searching
  - Best case: O(1)
  - Worst case: O(n)
  - Average case: O(log2n)

AVL tree

It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree. AVL tree satisfies the property of the binary tree as well as of the binary search tree.

- It is a self-balancing binary search tree that was invented by Adelson Velsky Lindas. Here, self- balancing means that balancing the heights of left subtree and right subtree.
- This balancing is measured in terms of the balancing factor.



AVL Tree

AVL tree

- We can consider a tree as an AVL tree if the tree obeys the binary search tree as well as a balancing factor.
- The balancing factor can be defined as the difference between the height of the left subtree and the height of the right subtree.
  - **Balance Factor:** *Balance Factor (node) = Height (left subtree) - Height (right subtree)*
- The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should have the value of the balancing factor either as 0, -1, or 1.

Creating the Node

```
class Node {
int value, height;
Node left, right;
public Node(int v) {
value = v;
}
}
```

Creating the Tree

```
class AVLTree {
private Node root;
public AVLTree() {
root = null;
}
}
```

```
public void insert(int value) {
root = insert(root, value);
}
private Node insert(Node node, int value) {
if (node == null)
return (new Node(value));
if (value < node.value)
node.left = insert(node.left, value);
else if (value > node.value)
node.right = insert(node.right, value);
return node;
}
}
```

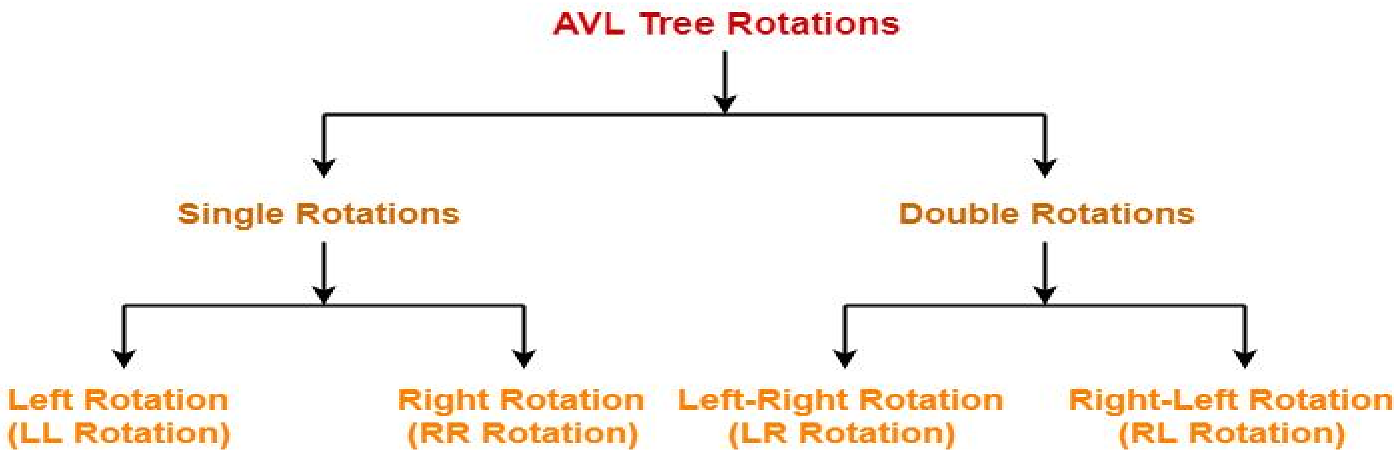
```
public void preOrder() {
preOrder(root);
}
private void preOrder(Node node) {
if (node != null) {
System.out.print(node.value + " ");
preOrder(node.left);
preOrder(node.right);
}
}
}
public class AVLImplementation {
public static void main(String[] args) {
AVLTree tree = new AVLTree();
// Insert nodes
tree.insert(10);
tree.insert(20);
tree.insert(30);
tree.insert(40);
tree.insert(50);
tree.insert(25);
// Print tree preorder
tree.preOrder();
}
}
```

```
 output:0 20 30 25 40 50
```

AVL tree

Rotation is the process of moving the nodes to make tree balanced.

There are 4 kinds of rotations possible in AVL Trees-



AVL Tree Rotations

Single Rotations

Double Rotations

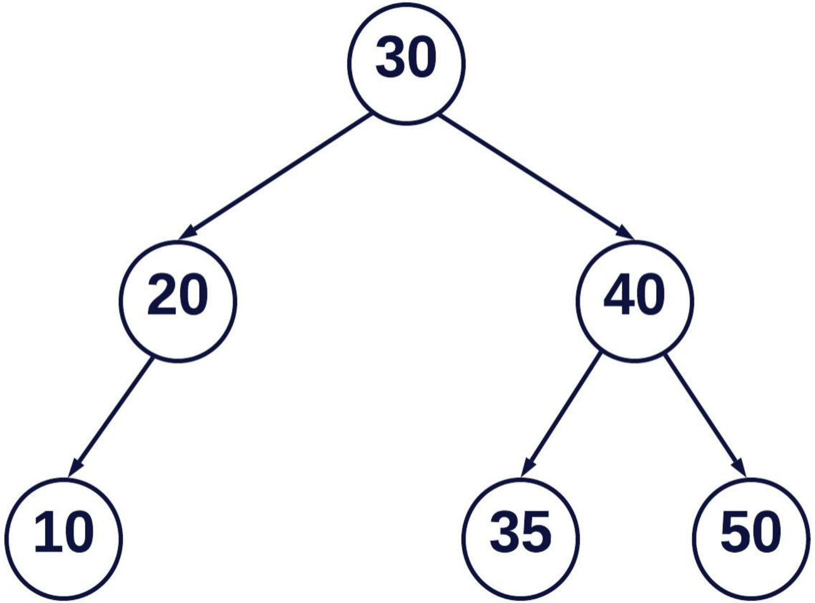
Left Rotation (LL Rotation)

Right Rotation (RR Rotation)

Left-Right Rotation (LR Rotation)

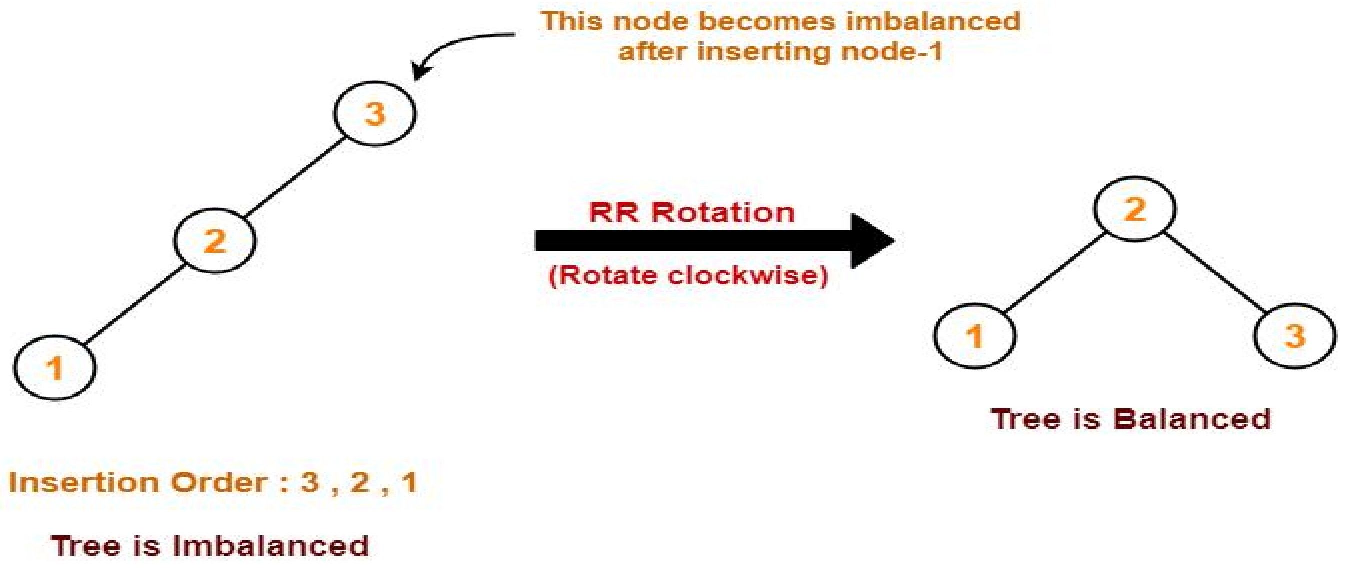
Right-Left Rotation (RL Rotation)

AVL tree



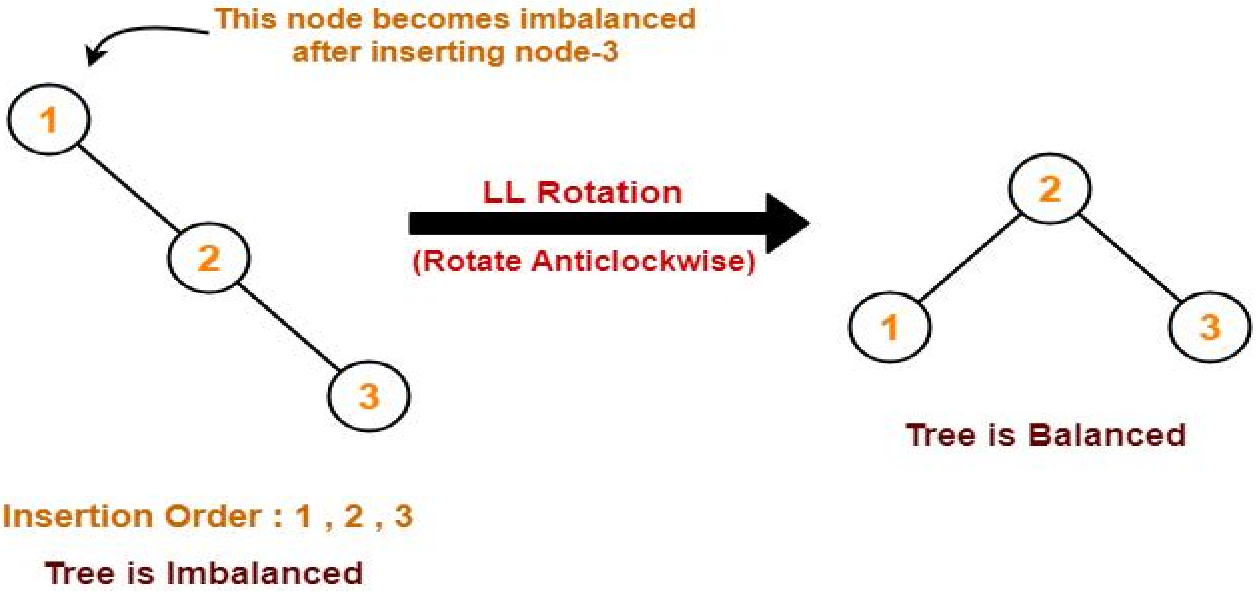
AVL tree

RR Rotation (Right Rotation): This rotation is applied when the left subtree of the left child of a node becomes longer than the right subtree. The unbalanced node is rotated to the right.



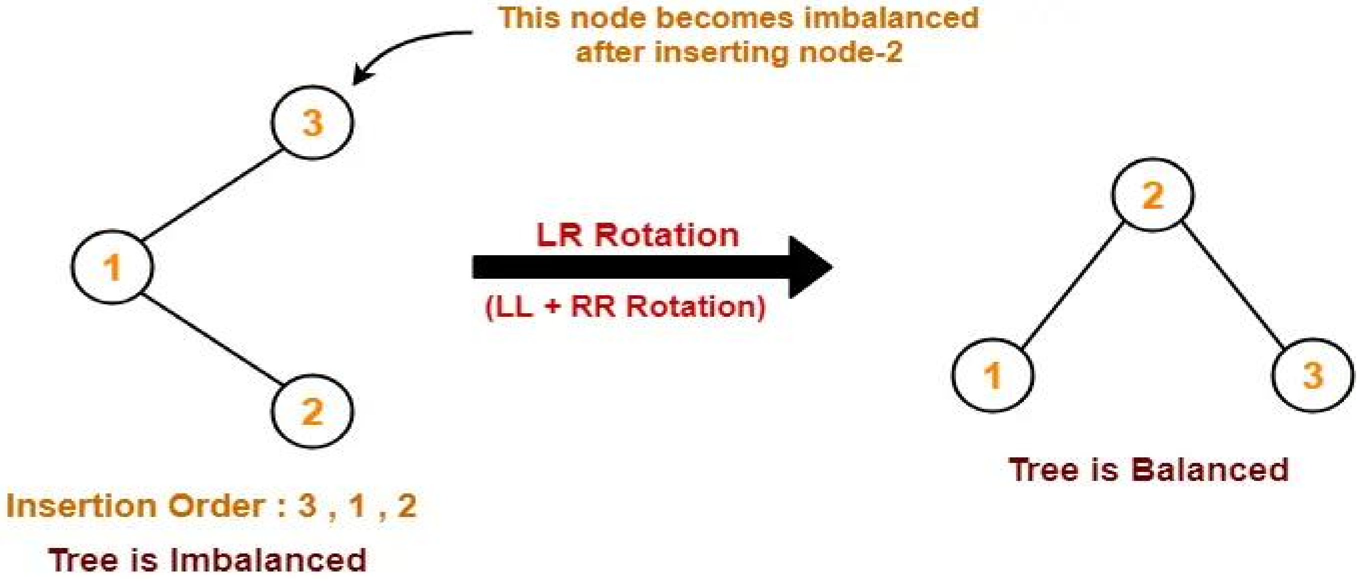
AVL tree

LL Rotation (Left Rotation): This rotation is applied when the right subtree of the right child of a node becomes longer than the left subtree. The unbalanced node is rotated to the left.



AVL tree

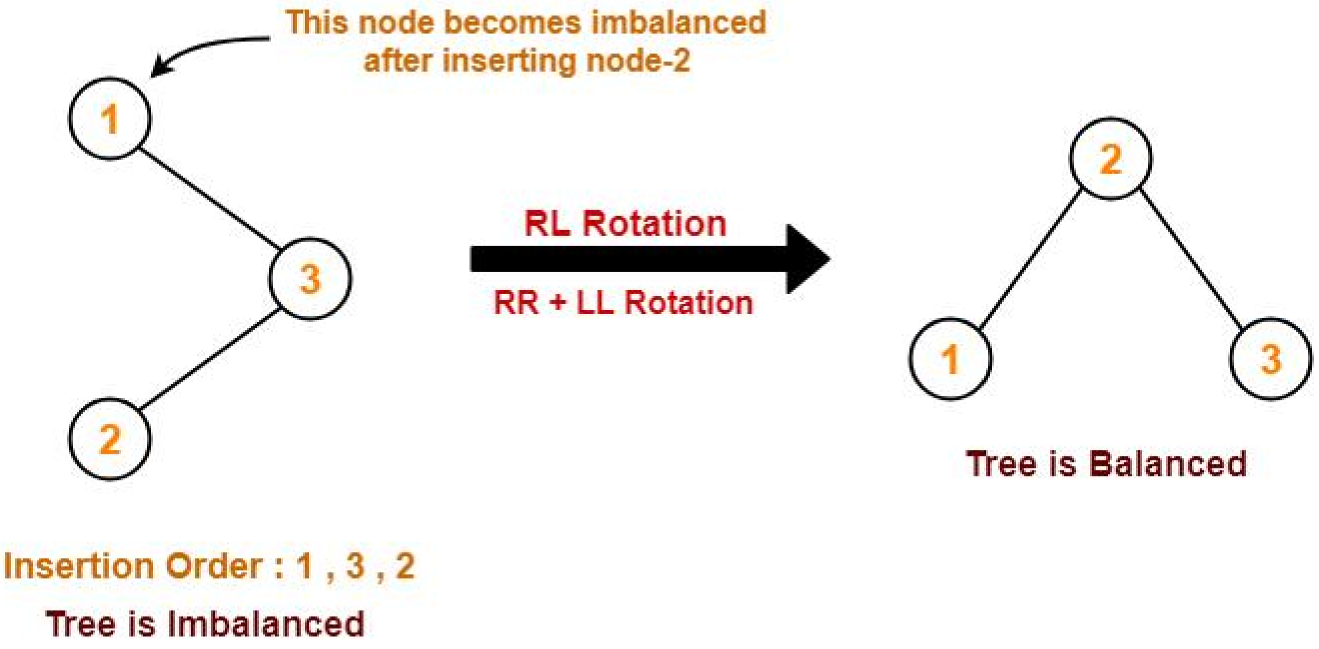
LR Rotation (Left-Right Rotation): This rotation is a combination of a left rotation followed by a right rotation. It’s applied when the right subtree of the left child of a node becomes longer than its left subtree.



This node becomes imbalanced after inserting node-2
3
1
2
LR Rotation
(LL + RR Rotation)
2
1
3
Insertion Order : 3 , 1 , 2
Tree is Imbalanced
Tree is Balanced

AVL tree

RL Rotation (Right-Left Rotation): This rotation is a combination of a right rotation followed by a left rotation. It’s applied when the left subtree of the right child of a node becomes longer than its right subtree.


This node becomes imbalanced after inserting node-2
1
3
2
RL Rotation
RR + LL Rotation
2
1
3
Insertion Order : 1 , 3 , 2
Tree is Imbalanced
Tree is Balanced