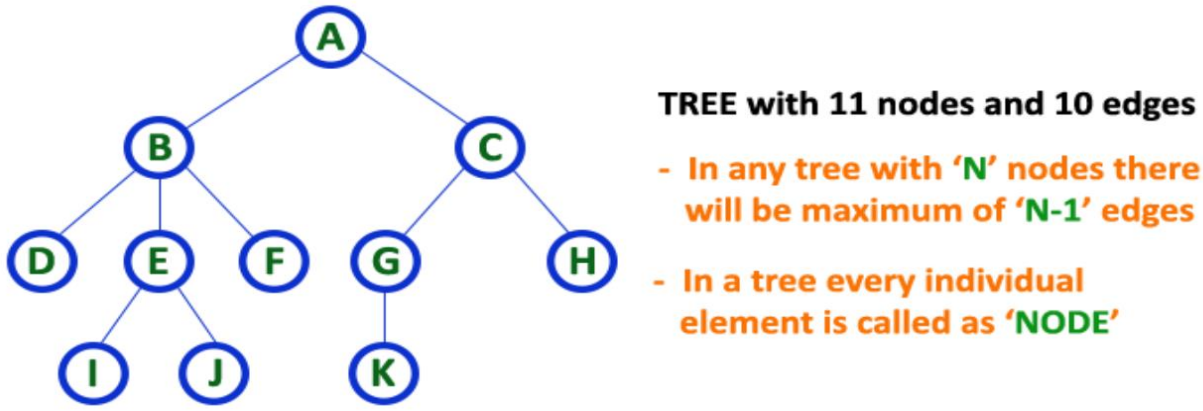
THE TREES DATA STRUCTURE

11.1 The Tree Data Structure

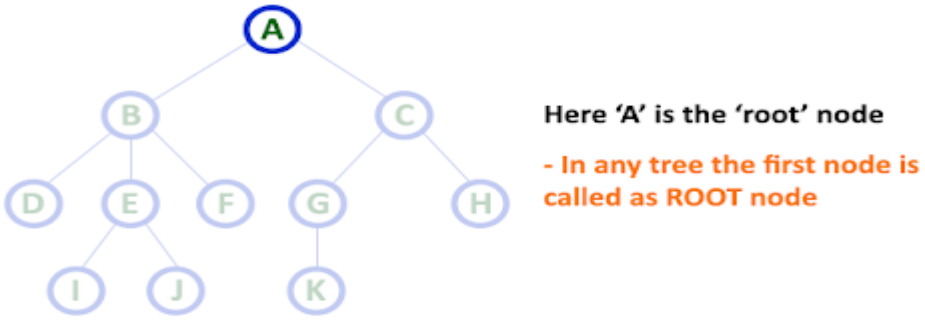
- Tree is a non-linear data structure which organizes data in hierarchical structure. In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.
- Trees usually consist of a single root node and one or more sub-trees attached to it. It is an acyclic connected graph, without any circuits. Trees do not contain self-loops, nor do they possess parallel edges. The edges of Trees are called branches.
- Each element of a Tree is called a node. While the nodes without any children are called **leaf nodes**.



11.3 Tree Terminologies

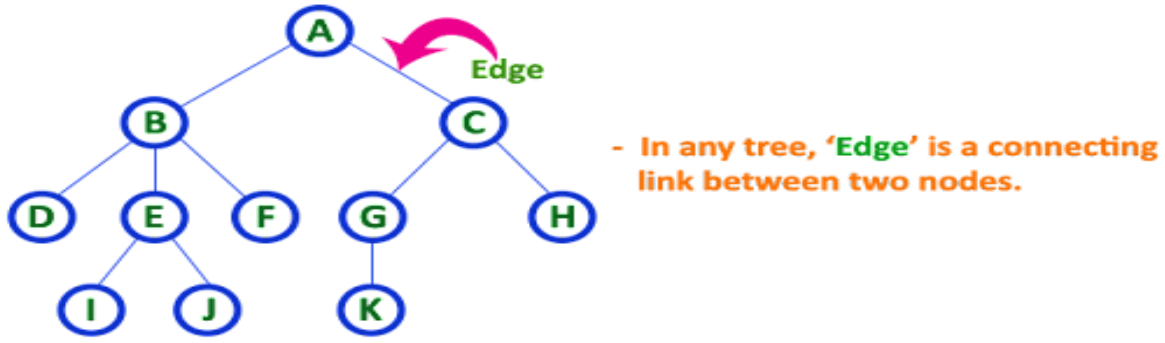
1. Root

In a tree data structure, the first node is called as Root Node. Every tree must have a root node. The root node is the origin of the tree data structure. In any tree, there must be only one root node. There can't be multiple root nodes in a tree.



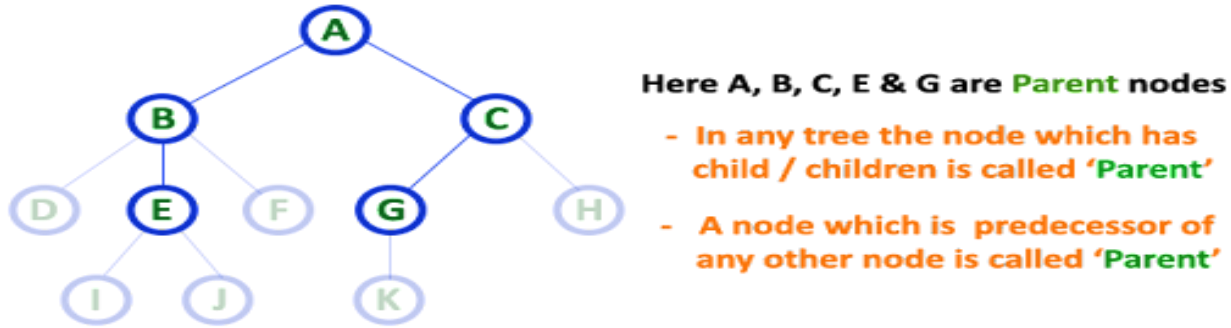
2. Edge

In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges



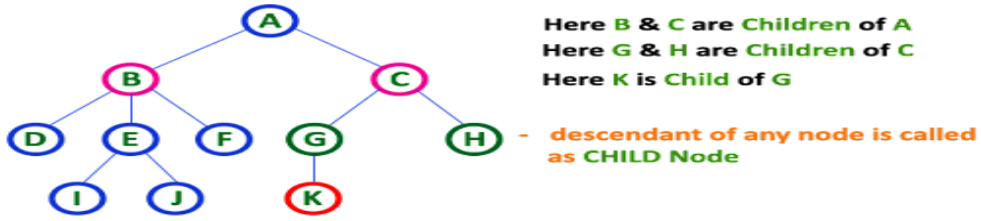
3. Parent

In a tree data structure, the node which is a predecessor of any node is called as PARENT NODE. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "The node which has child / children".



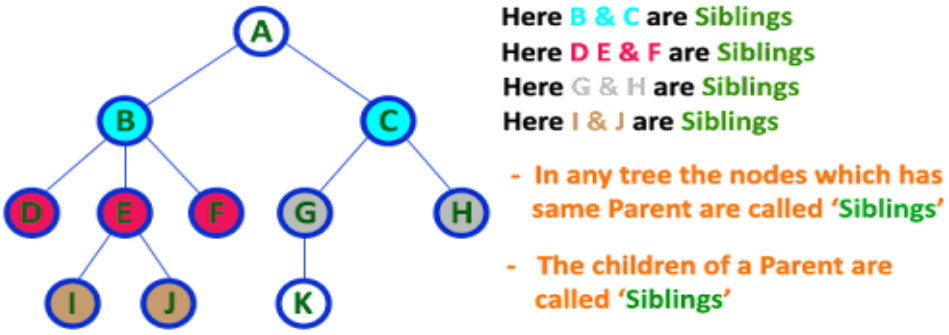
4. Child

In a tree data structure, the node which is descendant of any node is called as CHILD Node. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



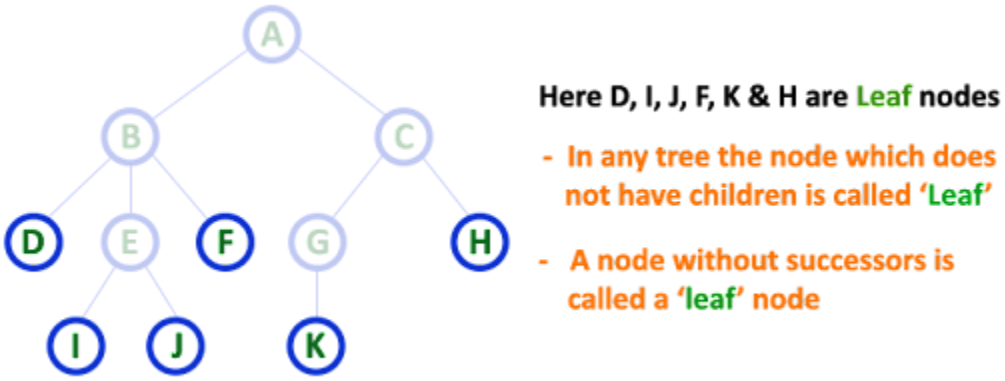
5. Siblings

In a tree data structure, nodes which belong to same Parent are called as SIBLINGS. In simple words, the nodes with the same parent are called Sibling nodes.



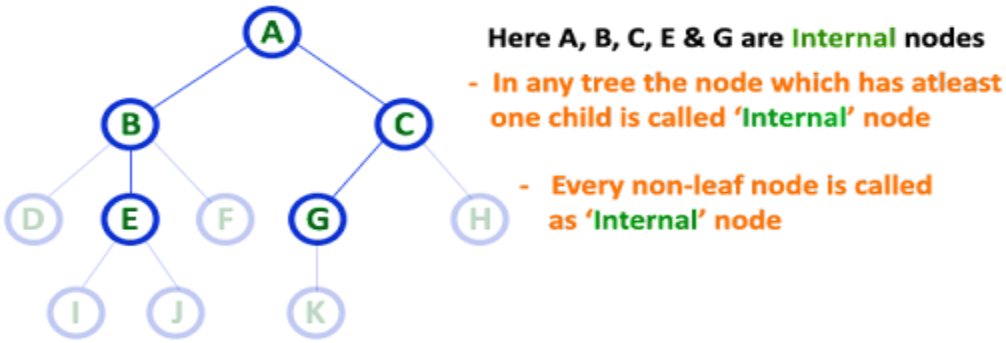
6. Leaf

In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



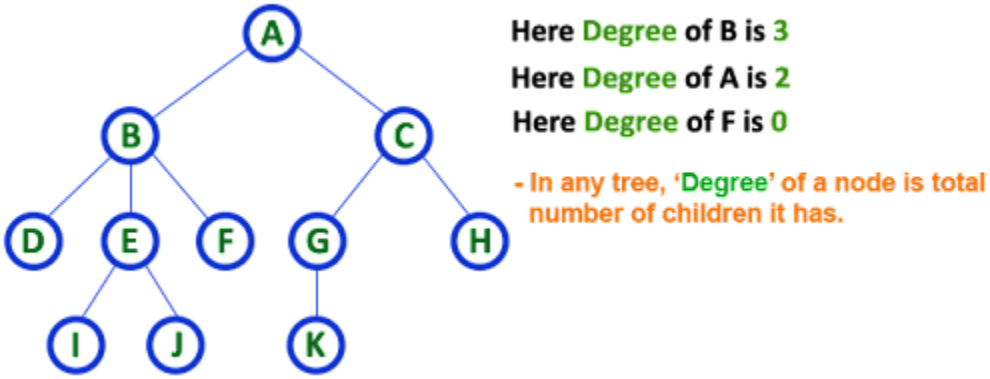
7. Internal Nodes

In a tree data, structure, the node which has at least one child is called as INTERNAL Node. In simple words, an internal node is a node with at least one child. In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non- Terminal' nodes.



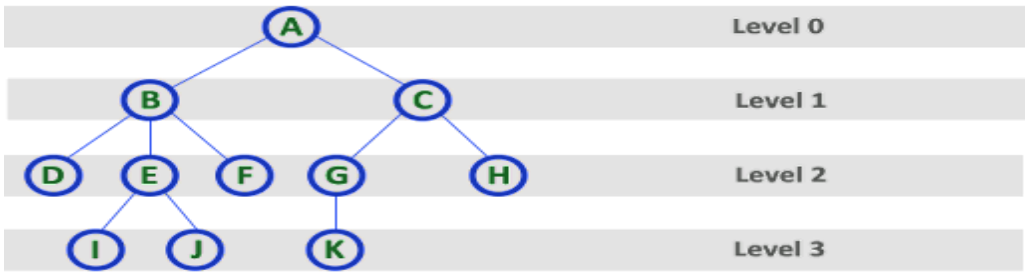
8. Degree

In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'.



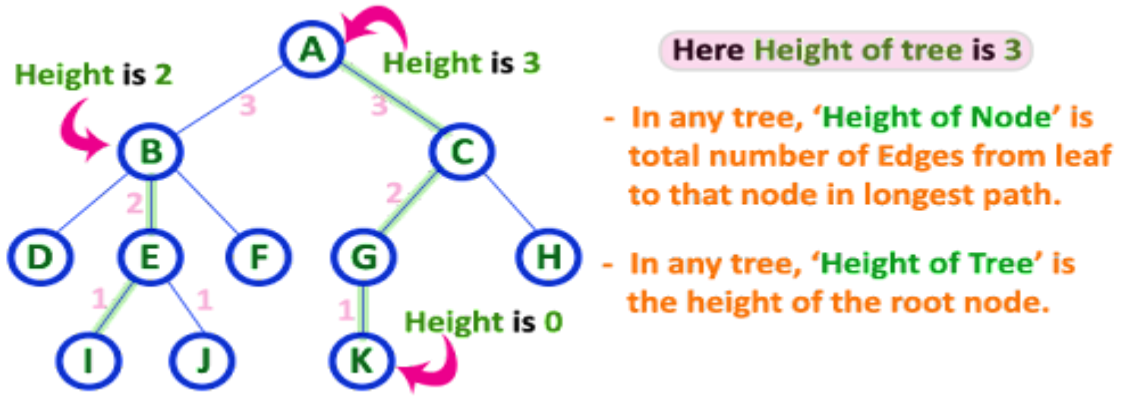
9. Level

In a tree data structure, the root node is said to be at Level O and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



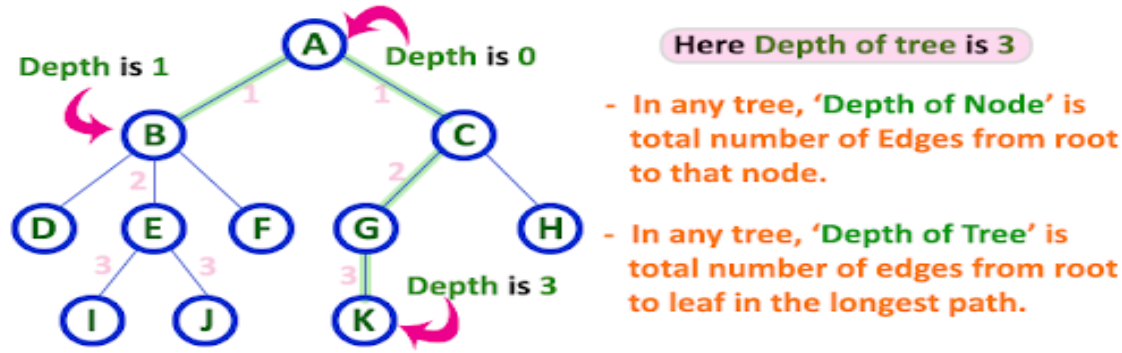
10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



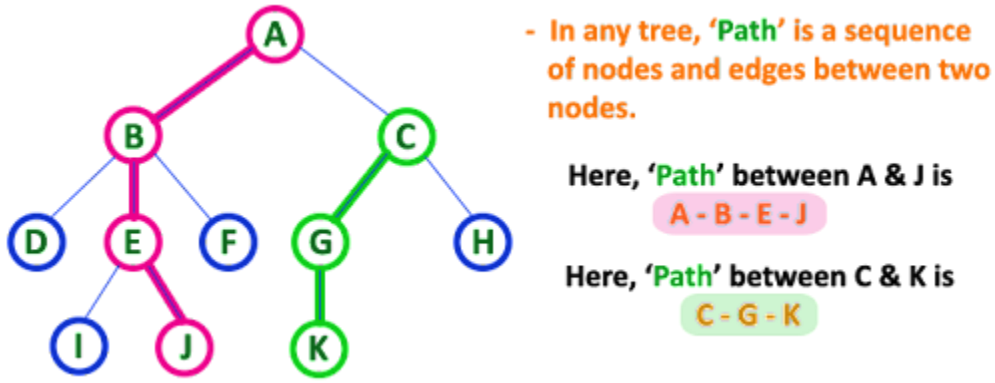
11. Depth

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be Depth of the tree. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.



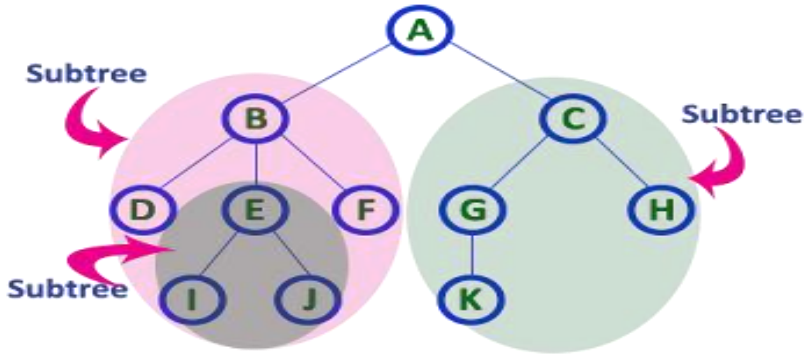
12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between those two Nodes. Length of a Path is total number of nodes in that path. In below example the path AB-E-J has length 4.



13. Sub Tree

In a tree data structure, each child from a node forms a sub tree recursively. Every child node will form a sub tree on its parent node.



11.4 Tree Node Representation

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

1. Array Representation

2. Linked List Representation

11.4.1 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. Consider the above example of a binary tree and it is represented as follows:

A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

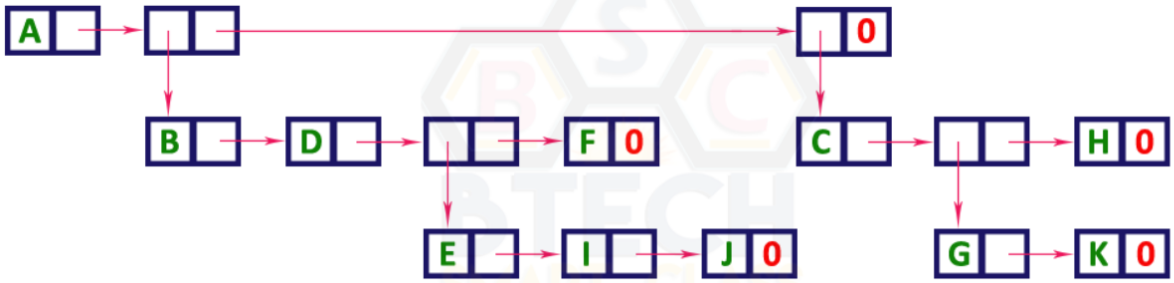
To represent a binary tree of depth 'n' using array representat

ion, we need one dimensional array with a maximum size of 2n + 1.

11.4.2 Linked List Representation of Binary Tree

1. Using Single Linked List

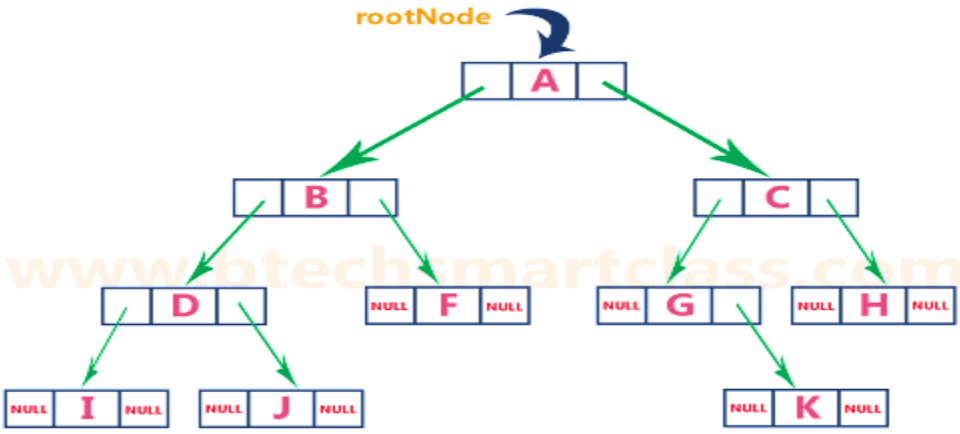
In this representation, we use two types of fields one for representing the node with data called 'data field' and another for representing only references called 'reference field. We start with a data field' from the root node in the tree. Then it is linked to an internal node through a 'reference field' which is further linked to any other node directly. This process repeats for all the nodes in the tree. The above example tree can be represented using List representation as follows:



2. Using Double Linked List

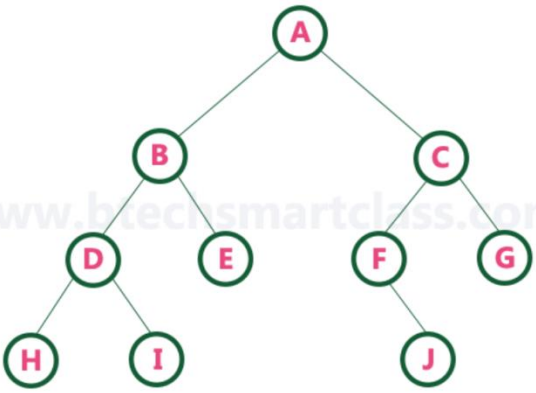
In this representation, we use a list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field storesthe address of the right sibling node. Graphical representation of that node is as shown in figure.

In this representation, every node's data field stores the actual value of that node. If that node has a left child, then left reference field stores the address of that left child node otherwise stores NULL. If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL. The above example tree can be represented using Left Child Right Sibling representation as follows.



11.5 Binary Trees

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have 0, 1 or 2 children. Let's understand the binary tree through an example.



11.5.1 Properties of Binary Tree

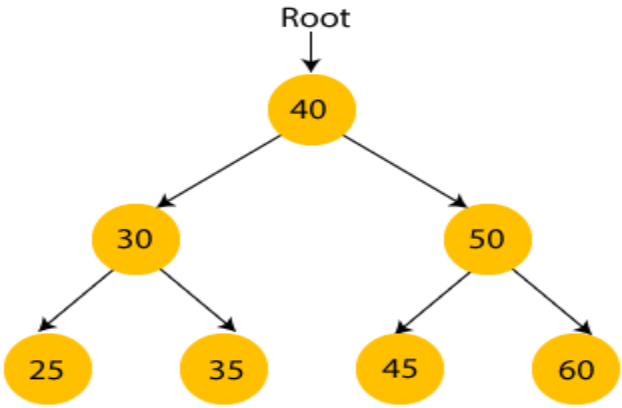
1. A binary tree can have a maximum of nodes at level if the level of the root is zero.
2. When each node of a binary tree has one or two children, the number of leaf nodes (nodes with no children) is one more than the number of nodes that have two children.
3. There exists a maximum of nodes in a binary tree if its height is, and the height of a leaf node is one.
4. If there exist leaf nodes in a binary tree, then it has at least levels.
5. A binary tree of nodes has minimum number of levels or minimum height.
6. A binary tree of nodes has null references.

There are four types of Binary tree:

1. Full/ proper/ strict Binary tree
2. Complete Binary tree
3. Perfect Binary tree
4. Degenerate Binary tree
5. Balanced Binary tree

11.6 Binary Search Trees

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right sub trees of the root.



In the above figure, we can observe that the root node is 40, and all the nodes of the left sub tree are smaller than the root node, and all the nodes of the right sub tree are greater than the root node. Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which sub tree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

11.6.1 Properties of Binary Search Tree

- Every Binary Search Tree is a binary tree
- Every left child will hold lesser value than root
- Every right child will hold greater value than root
- Ideal binary search tree will not hold same value twice.

11.7 Properties of Tree

- **Recursive data structure:** The tree is also known as a recursive data structure. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a root node. The root node of the tree contains a link to all the roots of its subtrees. The left subtree can be further split into subtrees. Recursion means reducing some- thing in a self-similar manner. So, this recursive property of the tree data structure is implemented in various applications.
- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

11.7 Implementation of Tree Operations

To initialize a binary search tree, we create a Binary Search tree class having three components a data value, reference to the left child node, and reference to the right child node. Below linked list is used to implement the various operations of a binary search tree.

```
Class BSTNode:
        def __init__ (self, data):
                self.data = data
                self.leftChild = None
                self.rightChild = None
newTree = BSTNode(None)
```

11.8 Insertion

The process of insertion in a binary search tree is ordered. This means that while inserting a node in a binary search tree, locate the right place where the node must be inserted according to the value it holds. There could be following three situations while inserting a node:

- If the binary tree is empty, insert the new node and set it as the root of the binary search tree.
- If the value of the new node is less than the value of the parent, move to the left sub tree and continue the process of comparison until the correct position is found.

- If the value of the new node is greater than the value of the parent, move to the right sub tree and continue the process of comparison until the correct position is found.

```
#Function to insert nodes in Binary Search Tree

def insertNode(root_node, node_value):
        if root_node.data == None:
                root_node.data = node_value
        elif node_value<= root_node.data:
                if root_node.leftChild is None:
                        root_node.leftChild= BSTNode(node_value)
                else:
                        insertNode(root_node.leftChild, node_value)
        else:
                if root node.rightChild is None:
                        root_node.rightChild = BSTNode(node_value)
                else:
                        insertNode(root_node.rightChild, node_value)
        return "The node has been successfully inserted."
#Initializing the Binary Search Tree
NewTree=BSTNode(None)
insertNode(newTree, 70)
insertNode(newTree, 50)
insertNode(newTree, 90)
insertNode(newTree, 30)
insertNode(newTree, 80)
insertNode(newTree, 100)
insertNode(newTree, 20)
insertNode(new Tree, 40)
print(insertNode(newTree, 60))
```

Output:

The node has been successfully inserted.

11.10 Deletion

While deletion in a binary search tree, we can encounter any of the following three cases:

- If the node to be deleted is a leaf node then simply remove the node.
- If the node to be deleted has a child node then copy the child to the node and delete the child.
- If the node to be deleted has two children then find the minimum value in the right subtree of current node using the minValueNode function and replace it with the current node to be deleted.

```
#Deleting a node from a Binary Search Tree

def minValueNode(bstNode):
        current = bstNode
        while (current.leftChild is not None):
                current = current.leftChild
        return current
def deleteNode(root_node, node_value):
        if root node is None:
                return root_node
        if node_value<root_node.data:
                root_node.leftChild = deleteNode(root_node.leftChild, node_value)
        elif node_value>root_node.data:
                root_node.rightChild = deleteNode(root_node. rightChild, node_value)
        else:
                if root_node.leftChild is None:
                        temp = root_node.rightChild
                        root_node = None
                        return temp
                if root_node.rightChild is None:
                        temp = root_node.leftChild
                        root_node = None
                        return temp
                temp = minValueNode(root_node.rightChild)
                root_node.data = temp.data
                root_node.rightChild = deleteNode(root_node.rightChild, temp.data)
        return root_node
#Initializing the Binary Search Tree
newTreeBSTNode(None)
insertNode(newTree, 70)
insertNode(newTree, 50)
insertNode(new Tree, 90)
insertNode(newTree, 30)
insertNode(new Tree, 80)
insertNode(newTree, 100)
insertNode(newTree, 20)
insertNode(newTree, 40)
deleteNode(newTree, 50)
inOrder Traversal(newTree)
```

11.11 Search

In a Binary Search Tree, the left child of a node has a value lesser than that node and the right child has a greater value. Thus, while searching, compare the value to be searched with that of the node that is visited. If the value is less than that of the node, traverse to the left subtree. Otherwise, traverse to the right subtree. This optimizes the time utilization as the tree is divided at each level during the search.

Implementation of Search in Binary Search Tree

To implement searching a node in a binary search tree, create a function by name searchNode that takes two arguments -the root node of the tree and the value to be searched. The recursive function compares the value of the current node with the value to be searched. Further, the recursive function is called for the left child or the right child of the current node depending on whether the value to be searched is lesser or greater than the current node.

```
#Searching a node in a Binary Search Tree

def searchNode(root_node, node_value):
        if root_node.data = = node_value:
                print("The element has been found.")
        elif node_value<root_node.data:
                if root_node.leftChild.data == node_value:
                        print("The element has been found.")
                else:
                        searchNode(root_node.leftChild, node_value)
        else:
                if root_node.rightChild.data == node_value:
                        print("The element has been found.")
                else:
                        searchNode(root_node.rightChild, node_value)
#Initializing the Binary Search Tree
newTree=BSTNode(None)
insertNode(new Tree, 70)
insertNode(newTree, 50)
insertNode(newTree, 90)
insertNode(new Tree, 30)
insertNode(newTree, 80)
insertNode(new Tree, 100)
insertNode(newTree, 20)
insertNode(newTree, 40)
searchNode(new Tree, 90)
```

Output

The element has been found.

11.12 Tree Traversals

Traversal is the process of visiting every node in the tree, exactly once, to print values, search nodes, etc. For a Binary Search Tree, traversal can be implemented in any of the following ways:

Depth First Traversals:

1. Inorder (Left, Root, Right)

2. Preorder (Root, Left, Right)

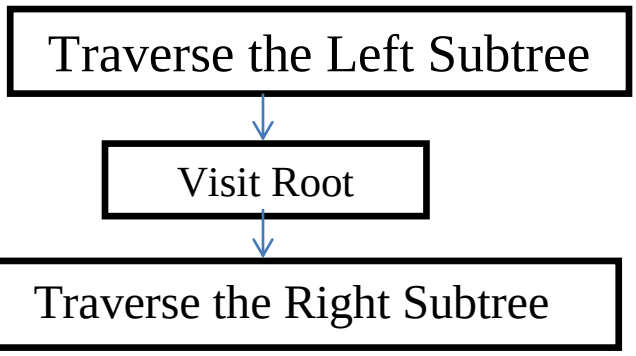
3. Postorder (Left, Right, Root)

Breadth-First Traversal:

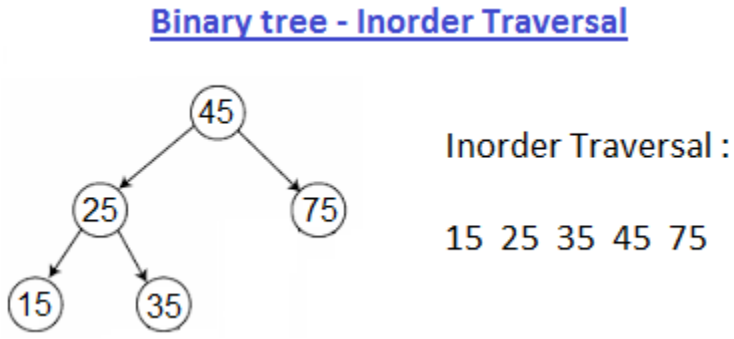
1. Level. Order Traversal

11.12.1 InOrder Traversal in Binary Search Tree

In-order traversal means to visit the left branch, then the current node, and finally, the right branch.



Example:



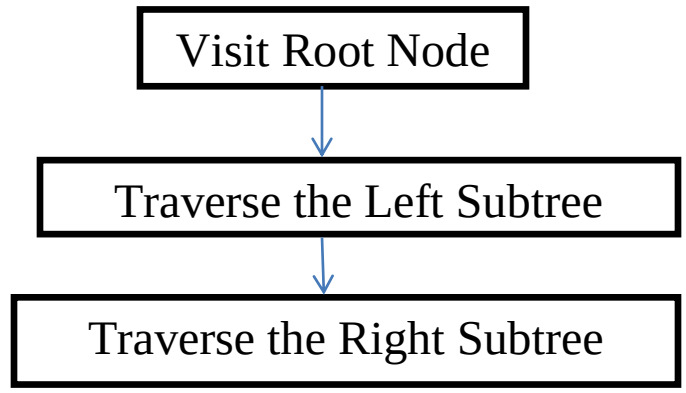
Implementation of InOrder Traversal in Binary Search Tree For implementing in-order traversal in a binary search tree, the below code makes use of a recursive function inOrder Traversal. Initially, the root node is passed as an argument to this function.

If the root node is not empty, a recursive call to the left child is made, followed by displaying the content of the current node, and then a recursive call to the right child is made. The function inOrderTraversal terminates a recursion when the encountered node is found to be empty.

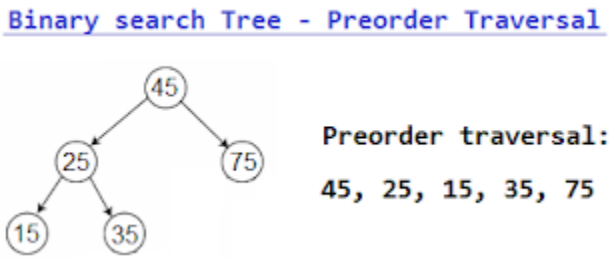
```
#Function to implement InOrder Traversal

def inOrder Traversal(root_node):
        if not root_node:
                return
        inOrder Traversal(root_node.leftChild)
        print(root_node.data)
        inOrder Traversal(root_node.rightChild)
```

11.12.2 PreOrder Traversal in Binary Search Tree

Pre-order traversal visits the current node before its child nodes. In a pre-order traversal, the root is always the first node visited.



Example:



Implementation of PreOrder Traversal in Binary Search Tree

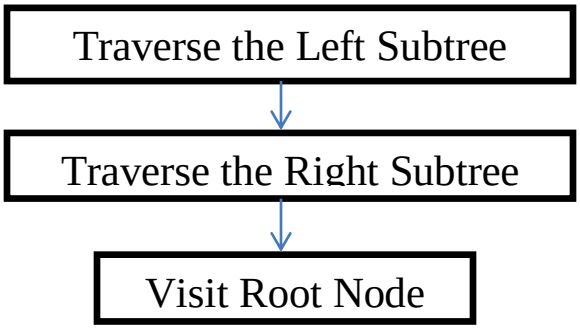
For implementing pre-order traversal in a binary search tree, the below code makes use of a recursive function preOrder Traversal. Initially, the root node is passed as an argument to this function.

If the root node is not empty, its content is displayed followed by calling the same recursive function with the left and right of the current node. The function preOrder Traversal terminates a recursion when the encountered node is found to be empty.

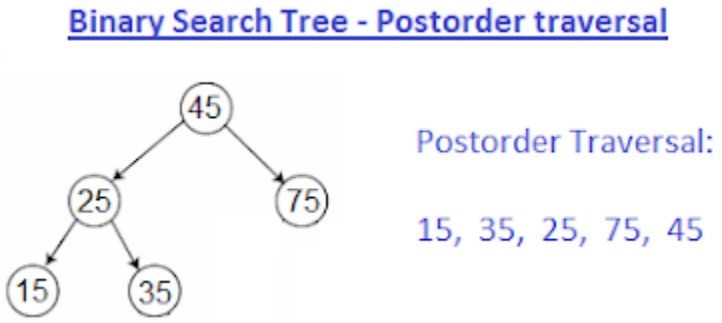
```
#Function to implement PreOrder Traversal

def preOrder Traversal(root_node):
        if not root_node:
                return
        print(root_node.data)
        preOrder Traversal (root_node.leftChild)
        preOrder Traversal(root_node.rightChild)
```

11.12.3 PostOrder Traversal in Binary Search Tree

Post-order traversal means visiting the left branch, then the current node, and finally, the right branch. In a post-order traversal, the root is always the last node visited.



Example:



Implementation of PostOrder Traversal in Binary Search

For implementing Post-order traversal in a binary search tree, the below code makes use of a recursive function postOrder Traversal. Initially, the root node is passed as an argument to this function.

If the root node is not empty, a recursive call to the left child is made, followed by a recursive call to the right child, and then a displaying the content of the current node. The function postOrder Traversal terminates a recursion when the encountered node is found to be empty.

```
#Function to implement PostOrder Traversal

def postOrder Traversal(root_node):
        if not root_node:
```

```
        return
    postOrder Traversal (root_node.leftChild)
    postOrder Traversal(root_node.rightChild)
    print(root_node.data)
```