Tree Data Structure-

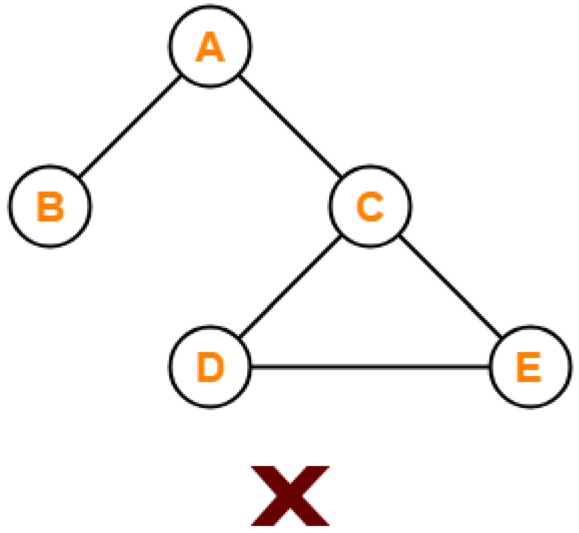
Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

OR

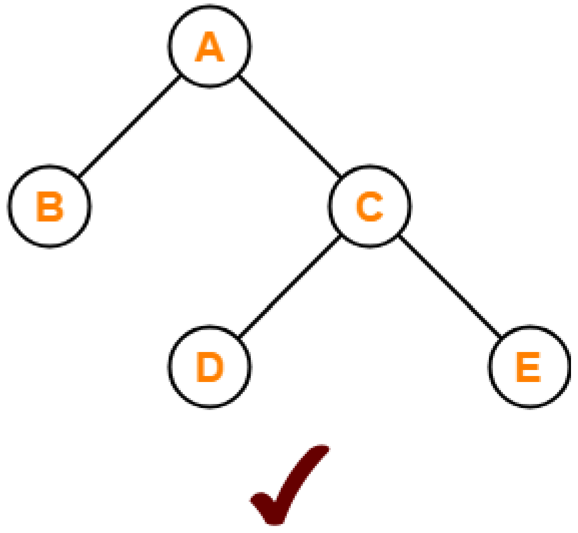
A tree is a connected graph without any circuits.

OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.



This graph is not a Tree



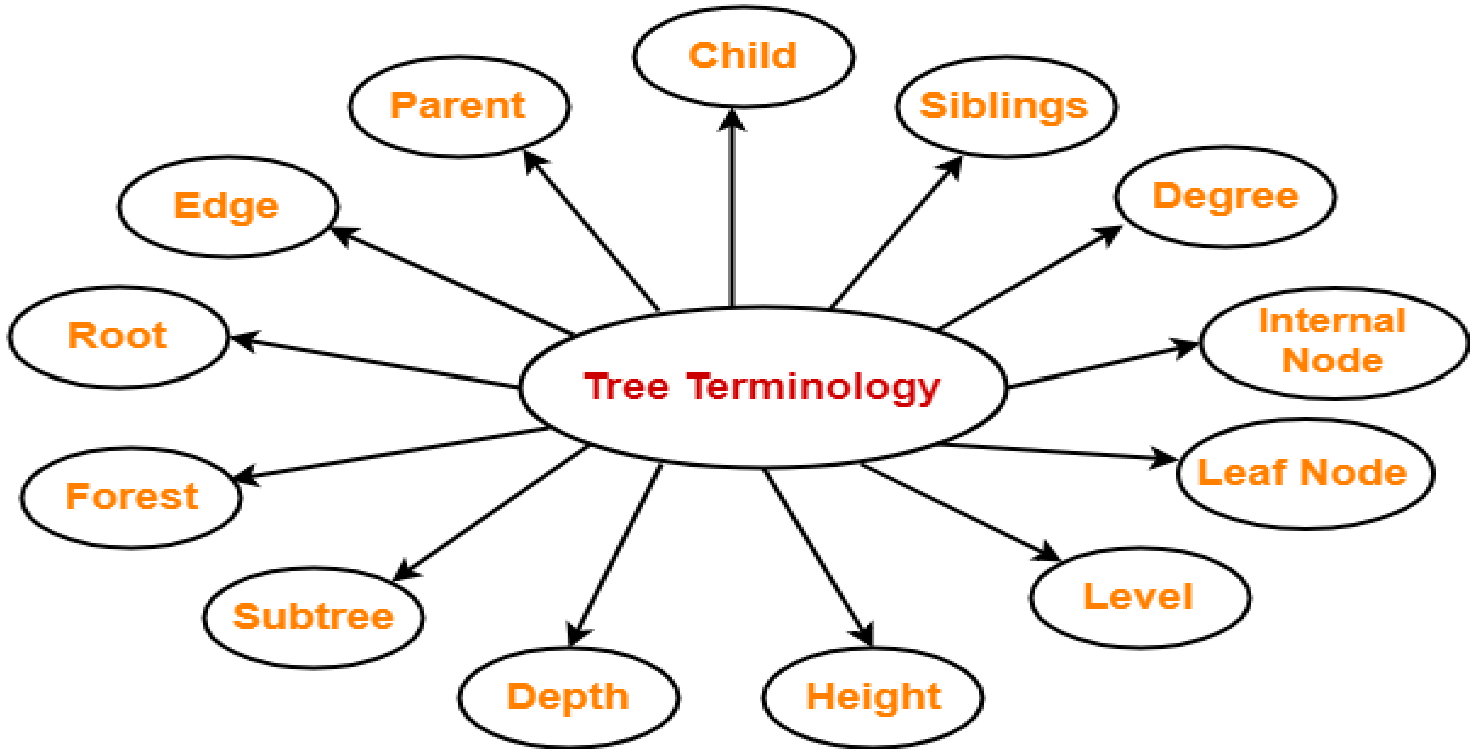
This graph is a Tree

Properties-

- The important properties of tree data structure are-
  - There is **one and only one path** between every pair of vertices in a tree.
  - A tree with n vertices has exactly **(n-1) edges.**
  - A graph is a tree if and only if it is minimally connected.
  - Any connected graph with n vertices and (n-1) edges is a tree.

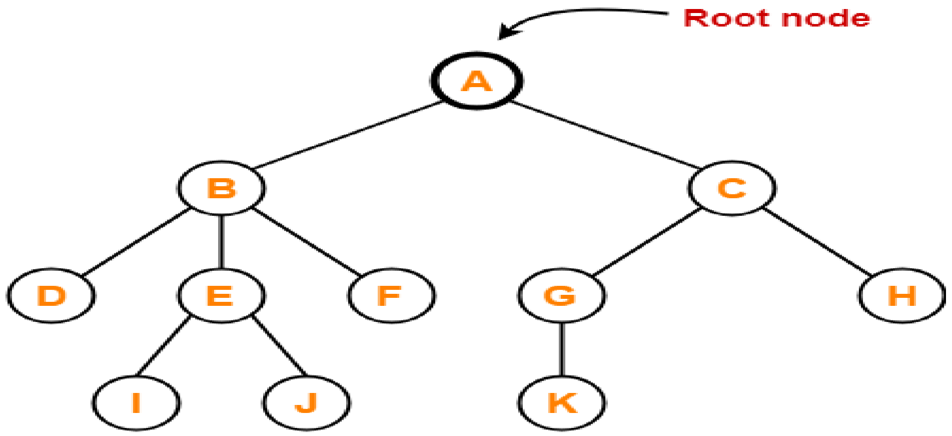
Tree Terminology-

The important terms related to tree data structure are-

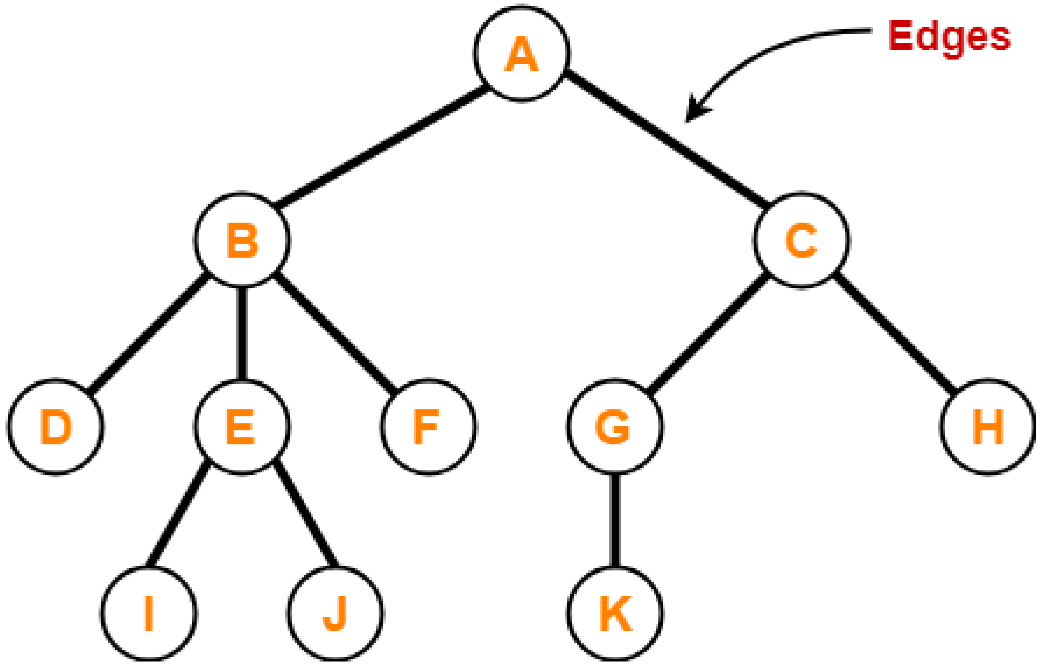


- **<u>1. Root-</u>**
  - The first node from where the tree originates is called as a **root node**.
  - In any tree, there must be only one root node.
  - We can never have multiple root nodes in a tree data structure.
-

Here, node A is the only root node.

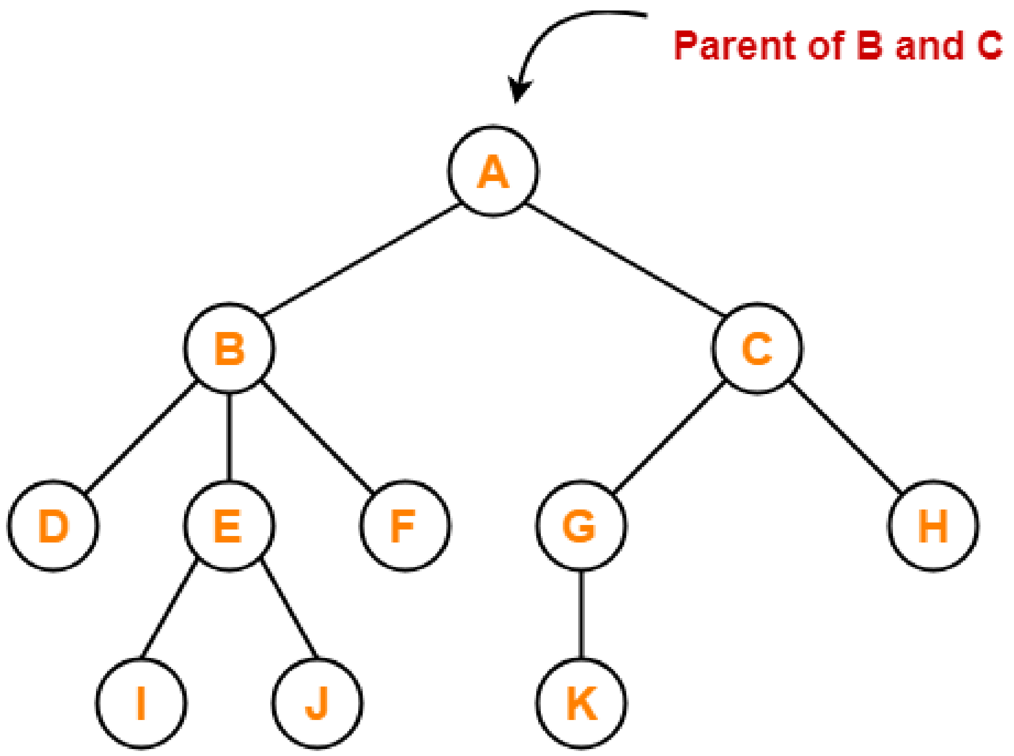


- **2. Edge-**
- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.
- **Example-**


Edges
A
B
C
D
E
F
G
H
I
J
K

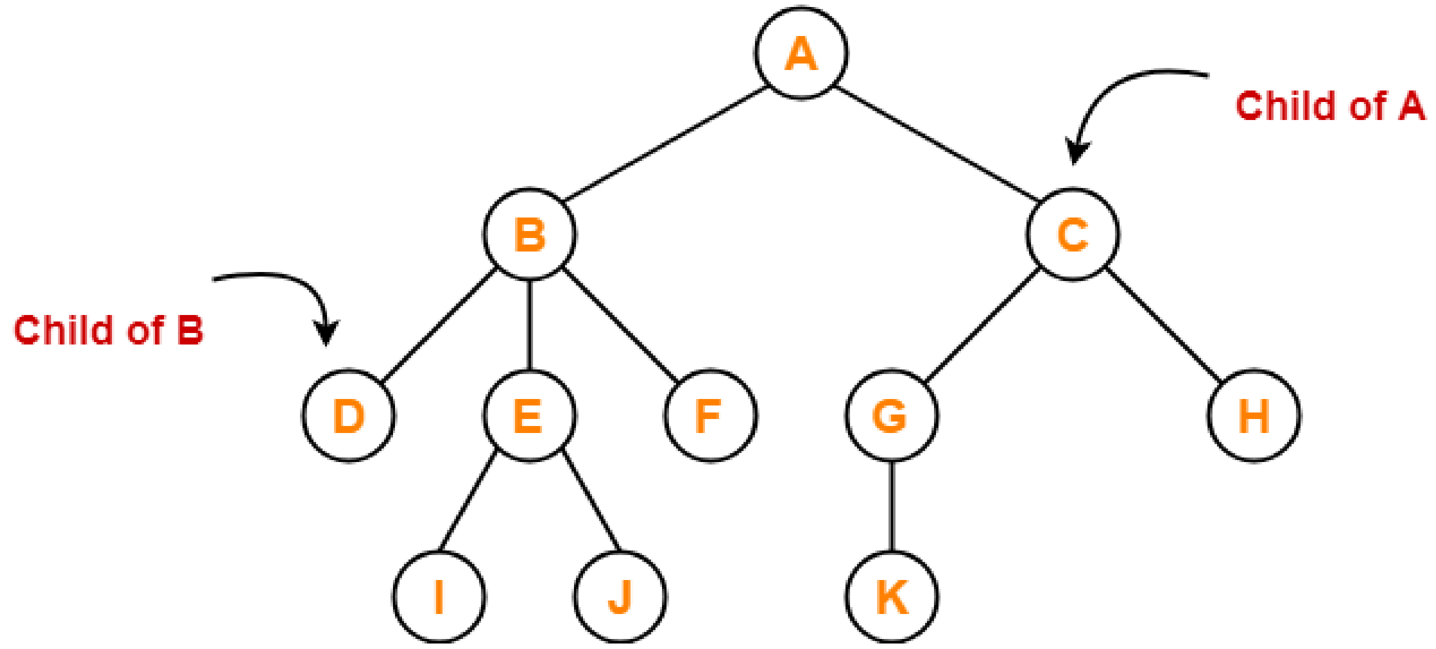
- **3. Parent-**
- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.
- **Example-**

Here,
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



4. Child-

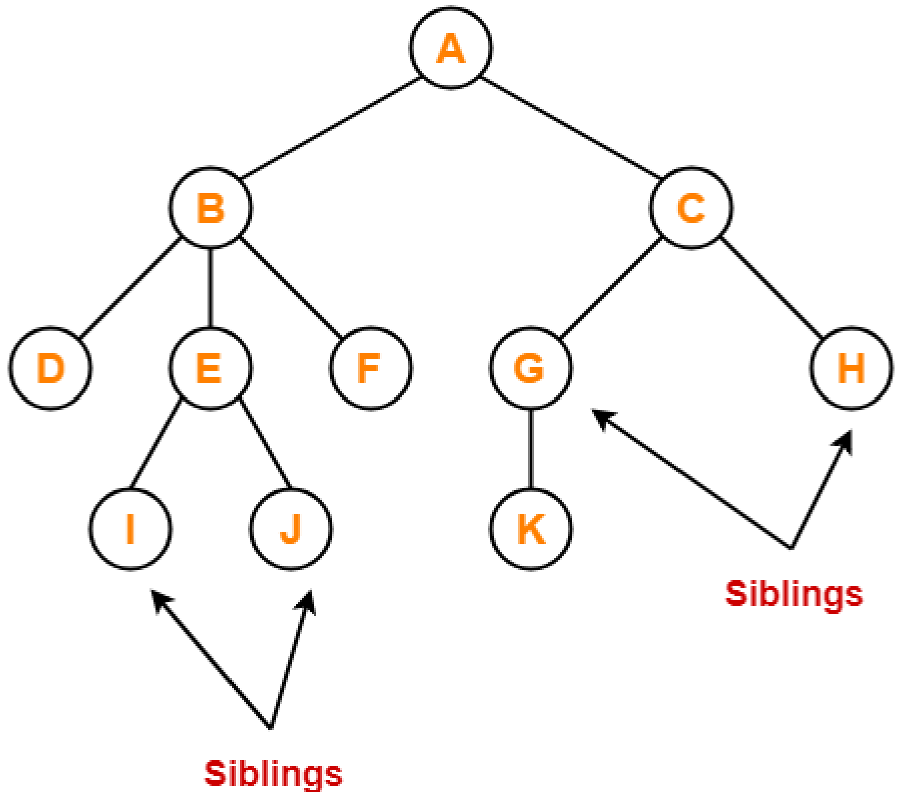
The node which is a descendant of some node is called as a **child node**.
All the nodes except root node are child nodes.



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

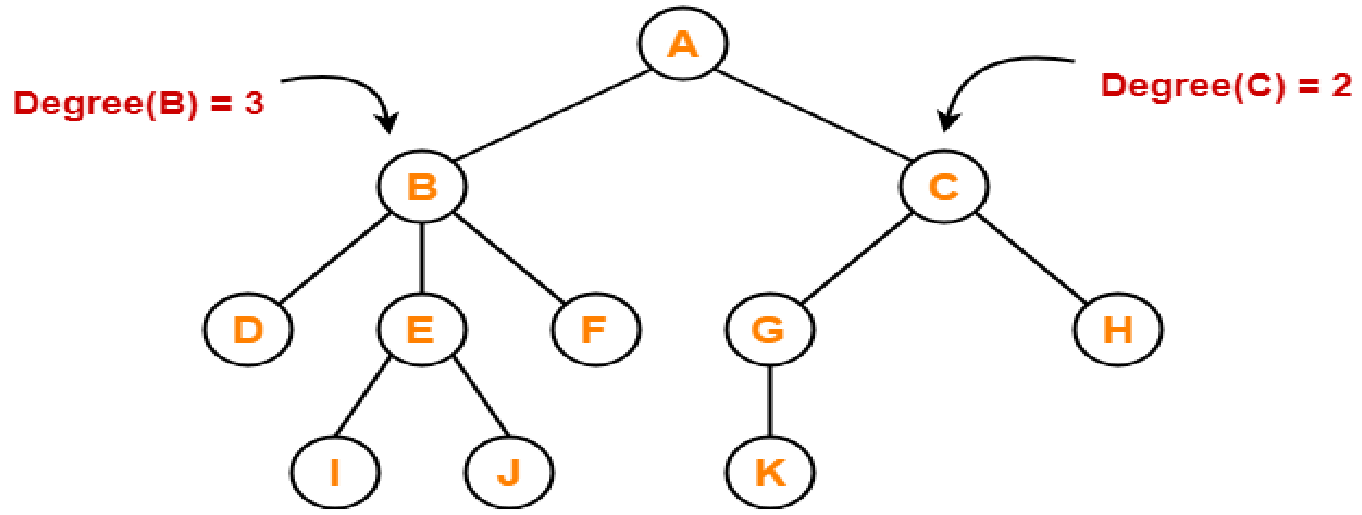


- Here,
  - Nodes B and C are siblings
  - Nodes D, E and F are siblings
  - Nodes G and H are siblings
  - Nodes I and J are siblings

6. Degree-

Degree of a node is the total number of children of that node.

Degree of a tree is the highest degree of a node among all the nodes in the tree.

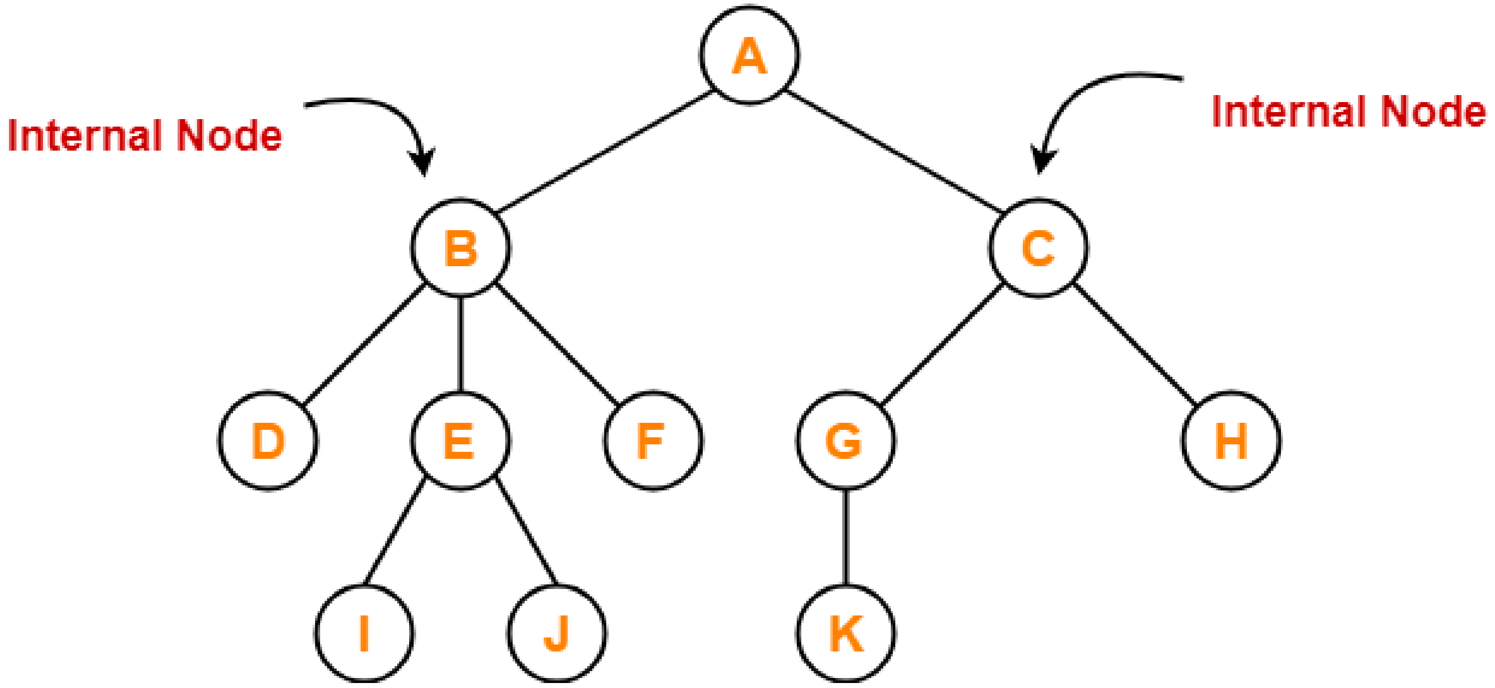


- Here,
- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

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.

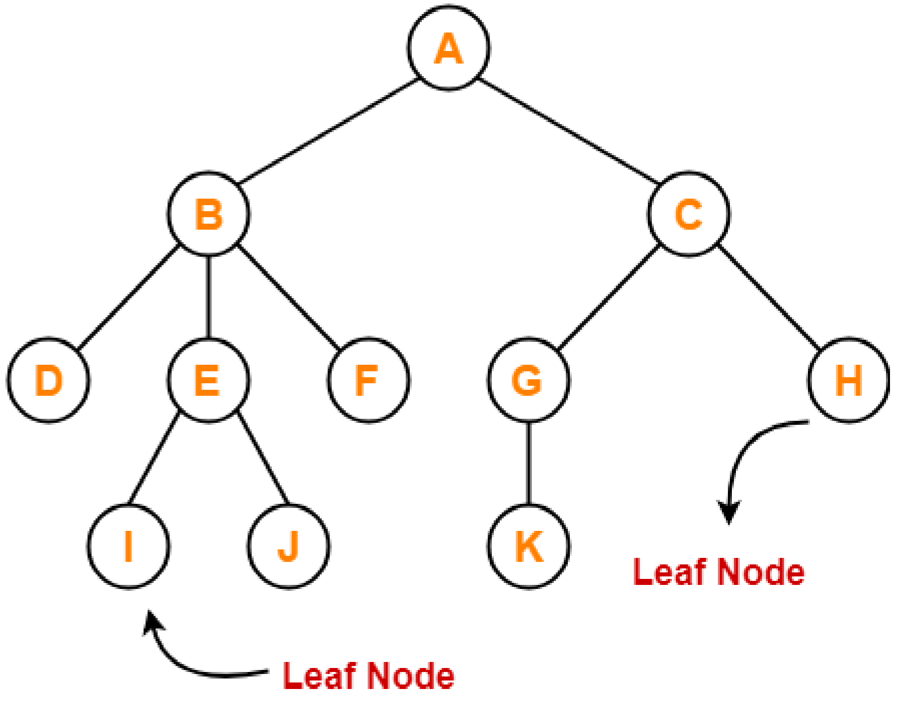


Leaf Node-

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

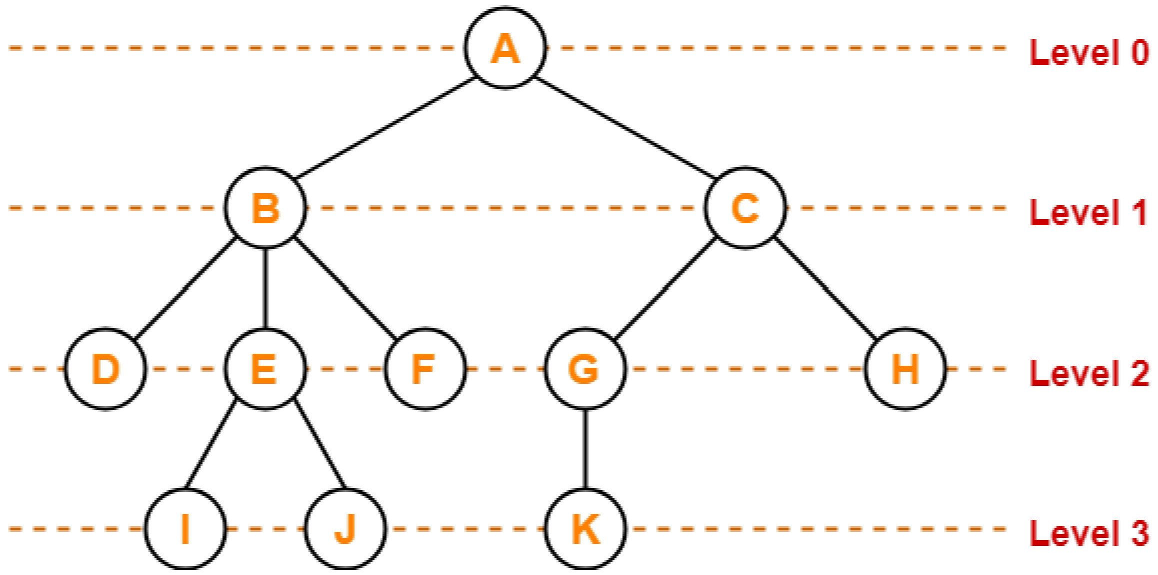
Leaf nodes are also called as **external nodes** or **terminal nodes**.

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



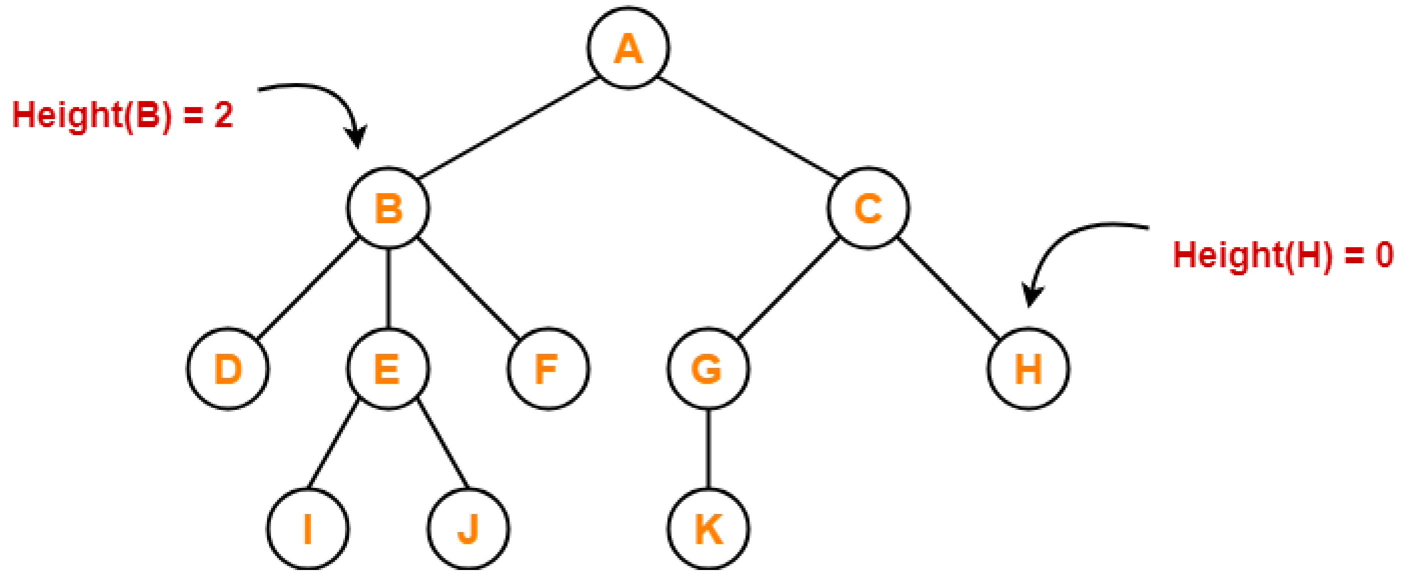
Level-

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



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

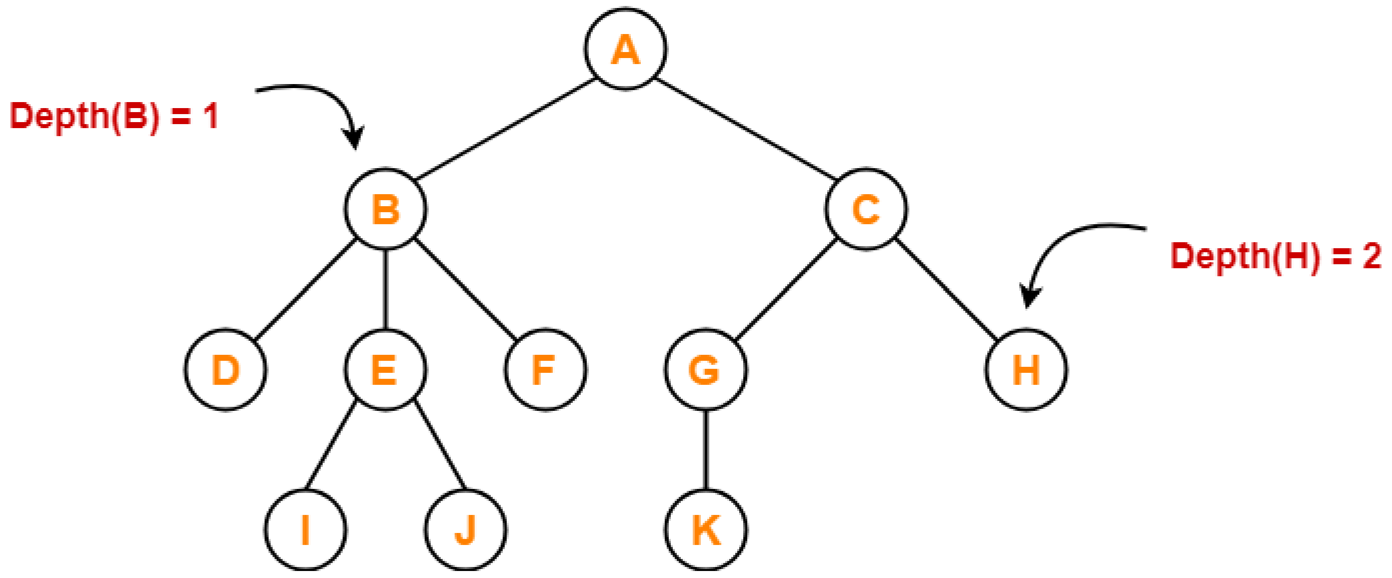
Depth-

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

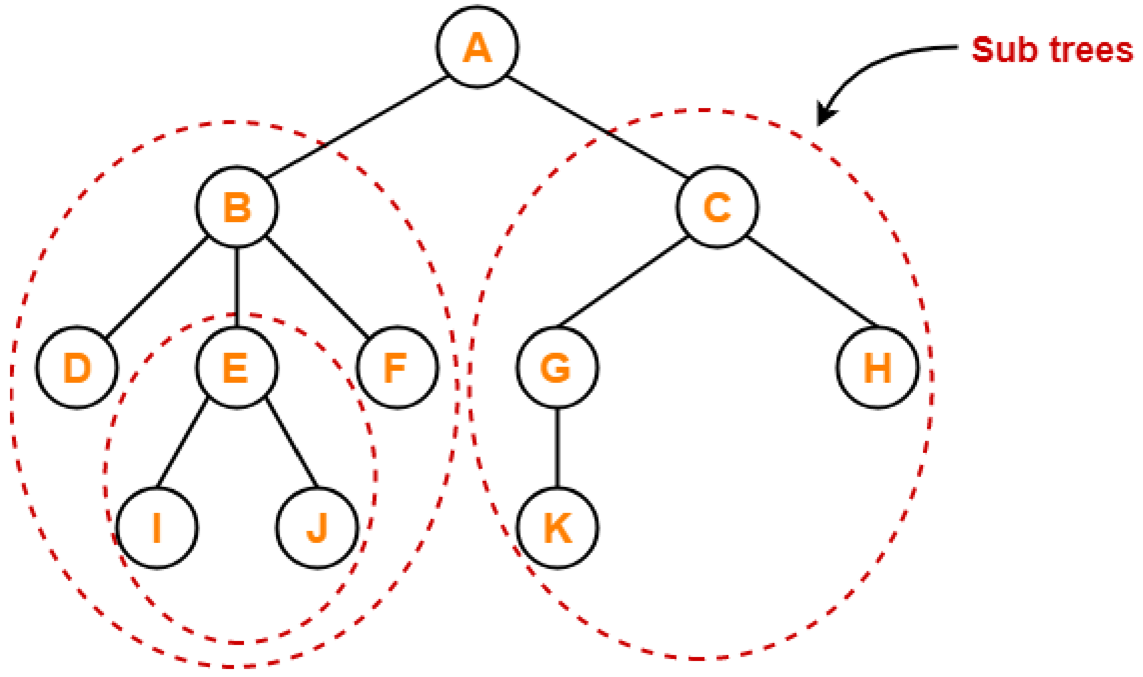
The terms “level” and “depth” are used interchangeably.



- Here,
- 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

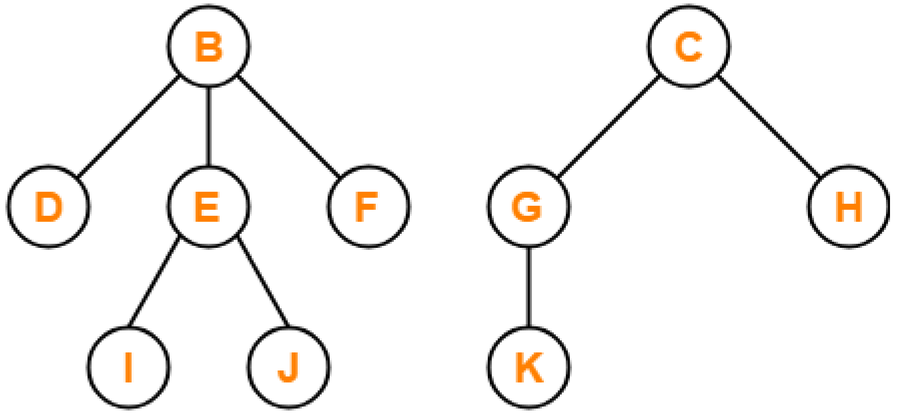
Subtree-

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



Forest-

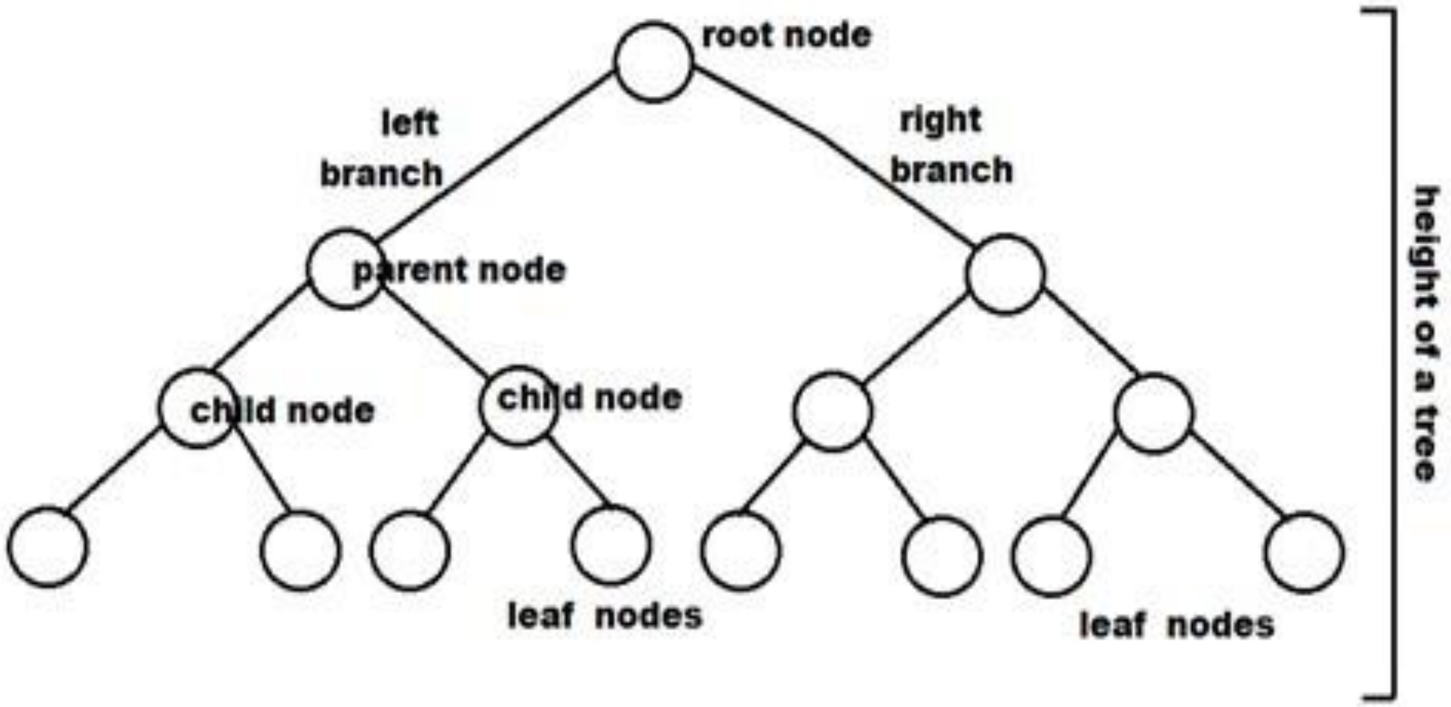
A forest is a set of disjoint trees.



Forest

Binary Tree

- A **binary tree** is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a **binary tree** has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.
- A parent node has two child nodes: the left child and right child. Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.


root node
left branch
right branch
parent node
child node
child node
leaf nodes
leaf nodes
height of a tree

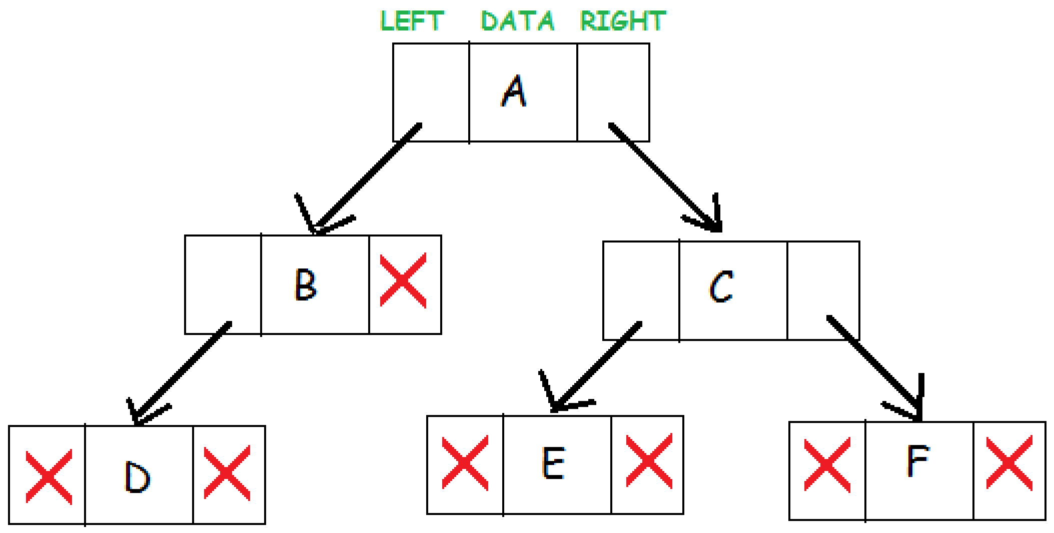
Terminologies associated with Binary Trees

- **Node:** It represents a termination point in a tree.
- **Root:** A tree’s topmost node.
- **Parent:** Each node (apart from the root) in a tree that has at least one sub-node of its own is called a parent node.
- **Child:** A node that straightway came from a parent node when moving away from the root is the child node.

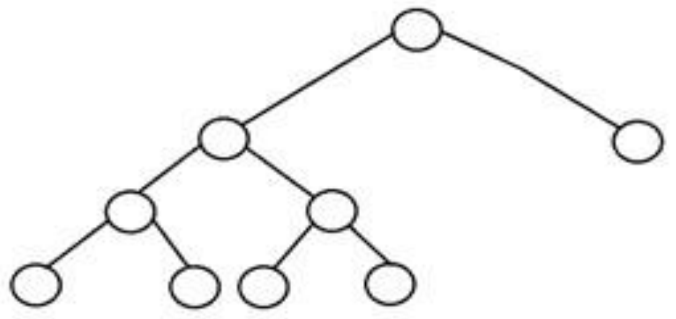
- **Leaf Node:** These are external nodes. They are the nodes that have no child.
- **Internal Node:** As the name suggests, these are inner nodes with at least one child.
- **Depth of a Tree:** The number of edges from the tree's node to the root is.
- **Height of a Tree:** It is the number of edges from the node to the deepest leaf. The tree height is also considered the root height.

- As you are now familiar with the terminologies associated with the binary tree and types of binary tree, it is time to understand the **binary tree components**.
- **Binary Tree Components**
- There are three **binary tree components**. Every **binary tree** node has these three components associated with it. It becomes an essential concept for programmers to understand these three **binary tree**

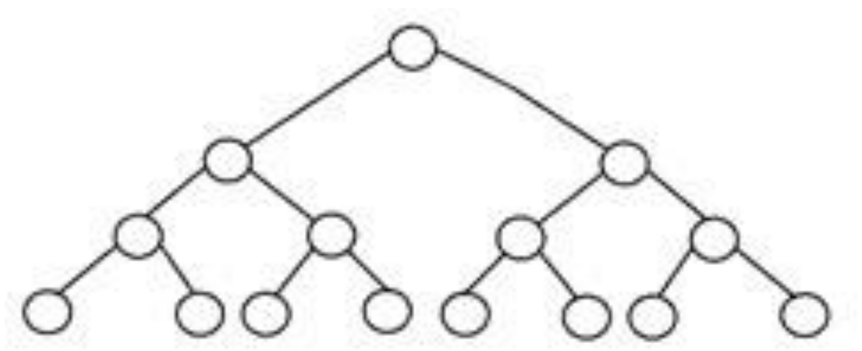
Data element
Pointer to left subtree
Pointer to right subtree

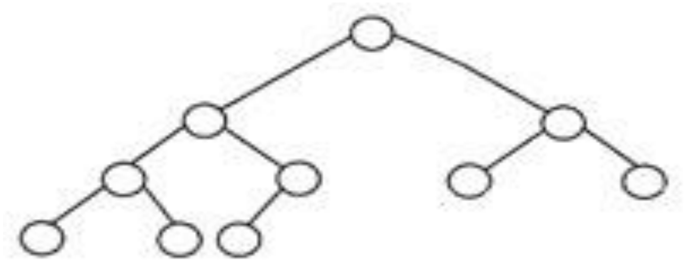


- These three **binary tree components** represent a node. **The data resides in the middle. The left pointer points to the child node, forming the left sub-tree. The right pointer points to the child node at its right, creating the right subtree.**
- **The structure of a full binary tree:** It is a special kind of a binary tree that **has either zero children or two children.** It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

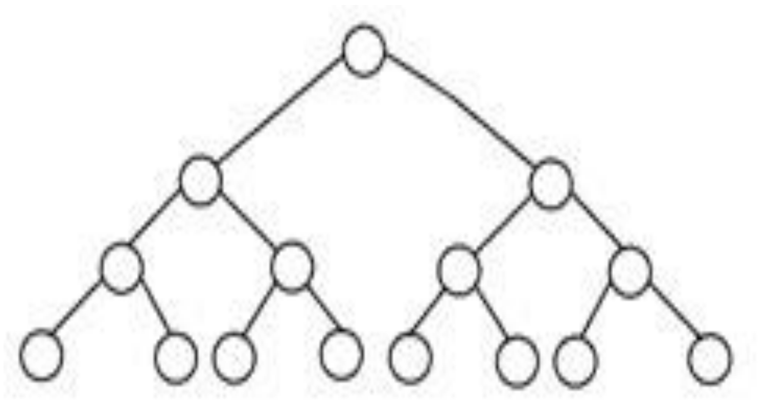


- The structure of a **complete binary tree**: A complete binary tree is another specific type of binary tree where **all the tree levels are filled entirely with nodes, except the lowest level of the tree.**



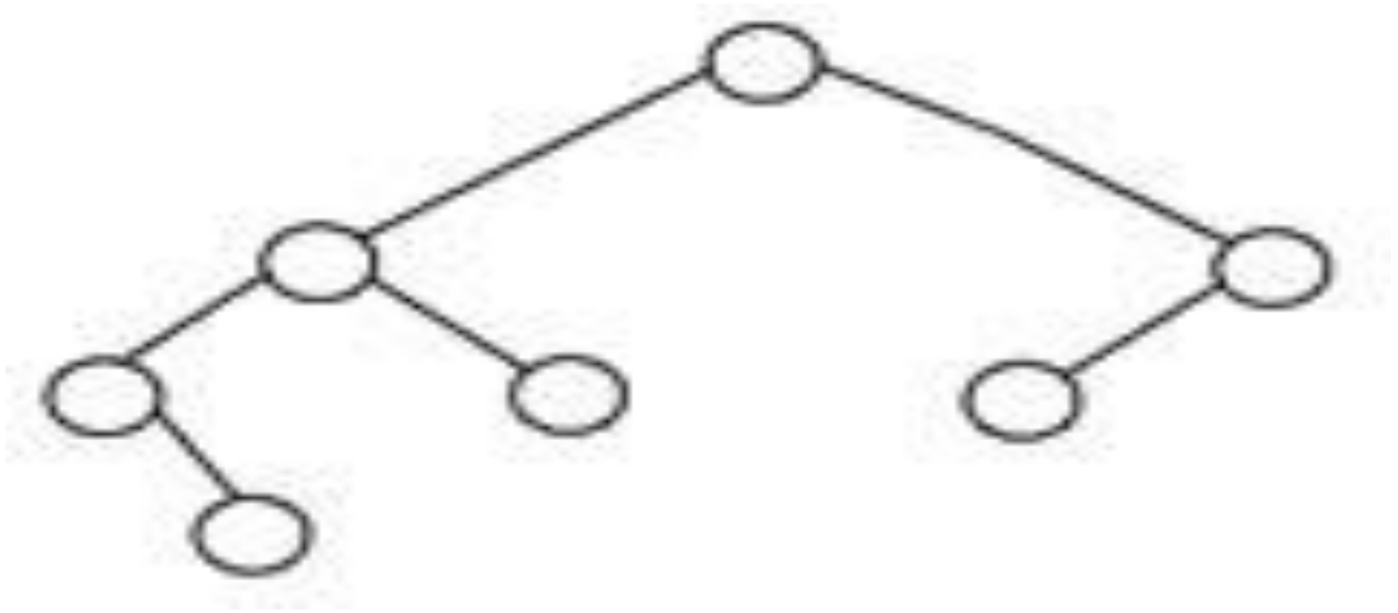


- The structure of a **perfect binary tree**: A binary tree is said to be ‘perfect’ if **all the internal nodes have strictly two children**, and every external or leaf node is at the same level or same depth within a tree.

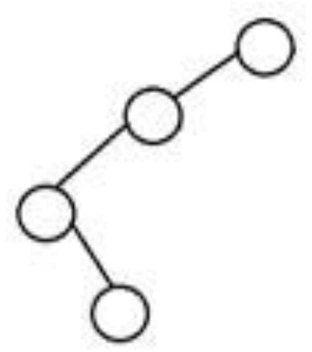


Balanced Binary Tree

A binary tree is said to be ‘balanced’ if the tree height is O(logN), where ‘N’ is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree. Here is an example of a balanced binary tree:



- **Degenerate Binary Tree**
- A binary tree is said to be a degenerate binary tree or **pathological binary tree if every internal node has only a single child**. Such trees are similar to a linked list performance-wise. Here is an example of a degenerate binary tree:

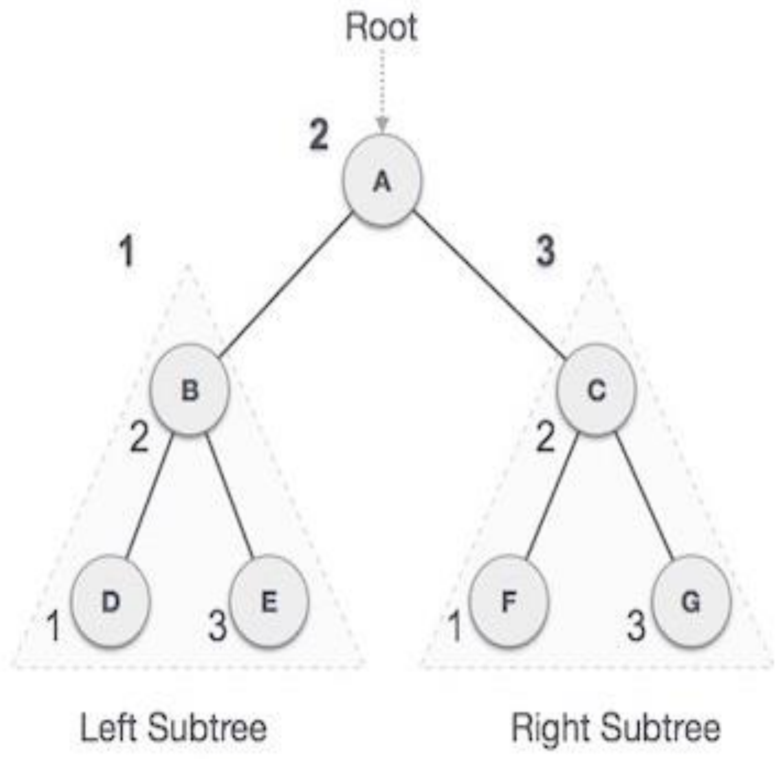


- **Benefits of a Binary Tree**
- The search operation in a binary tree is faster as compared to other trees
- Only two traversals are enough to provide the elements in sorted order
- It is easy to pick up the maximum and minimum elements
- Graph traversal also uses binary trees
- Converting different postfix and prefix expressions are possible using binary trees

1. <u>Tree Traversal</u>

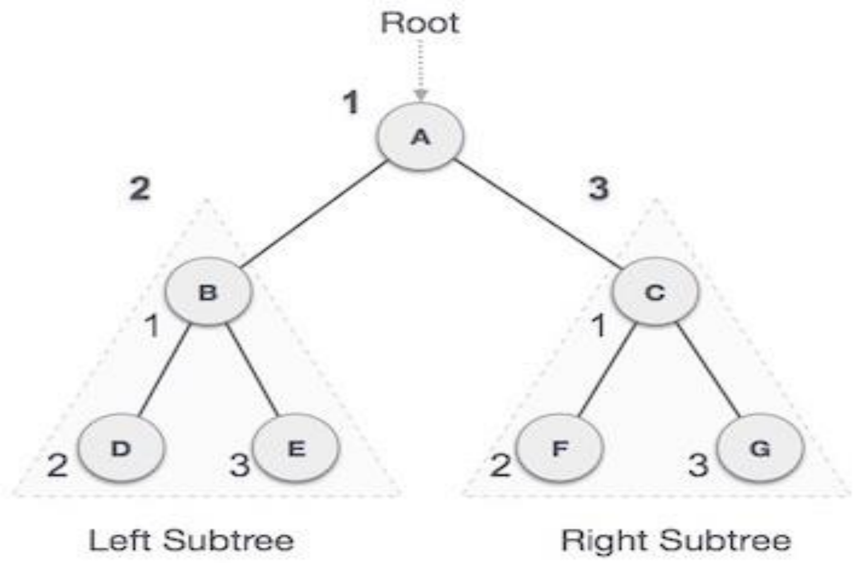
- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal
- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

- **In-order Traversal(L r R)**
- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.
- If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



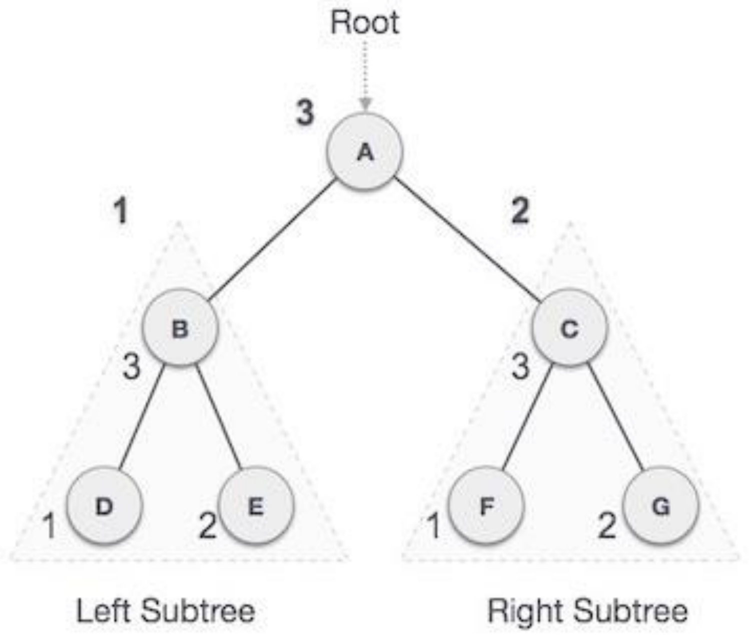
- We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −
  - $D \to B \to E \to A \to F \to C \to G$
- Algorithm
- Until all nodes are traversed −
- **Step 1** − Recursively traverse left subtree.
- **Step 2** − Visit root node.
- **Step 3** − Recursively traverse right subtree.

- **Pre-order Traversal(r L R)**
- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



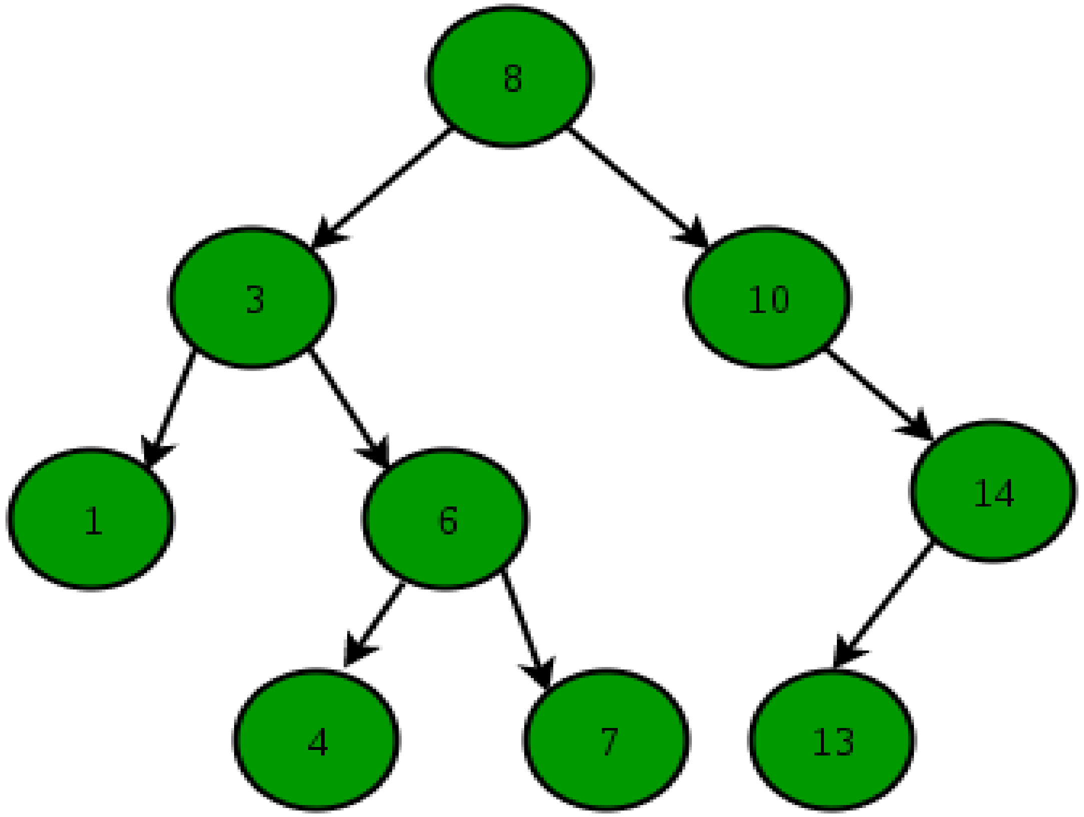
- We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −
  - $A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$
- Algorithm
- Until all nodes are traversed −
- **Step 1** − Visit root node.
- **Step 2** − Recursively traverse left subtree.
- **Step 3** − Recursively traverse right subtree.

- **Post-order Traversal(L R r)**
- In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

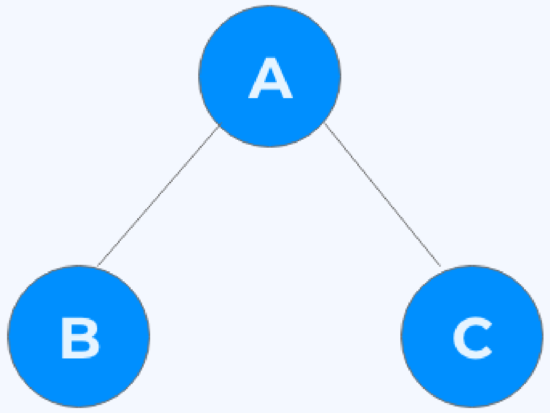


- We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −
  - $D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$
- Algorithm
- Until all nodes are traversed −
- **Step 1** − Recursively traverse left subtree.
- **Step 2** − Recursively traverse right subtree.
- **Step 3** − Visit root node.
-

- **Binary Search Tree**
- **Binary Search Tree** is a node-based binary tree data structure which has the following properties:
- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.


8
3
10
1
6
14
4
7
13

Inorder Postorder Preorder Example



PreOrder

ABC

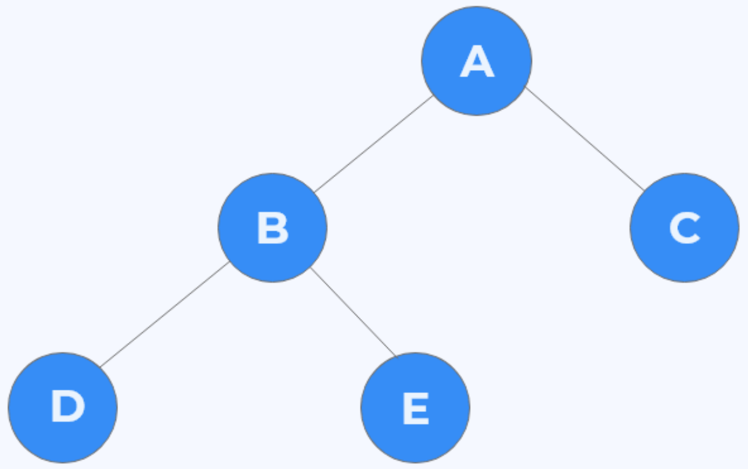
PostOrder

BCA

InOrder

ABC

Inorder Postorder Preorder Example 0



PreOrder:

A B D E C

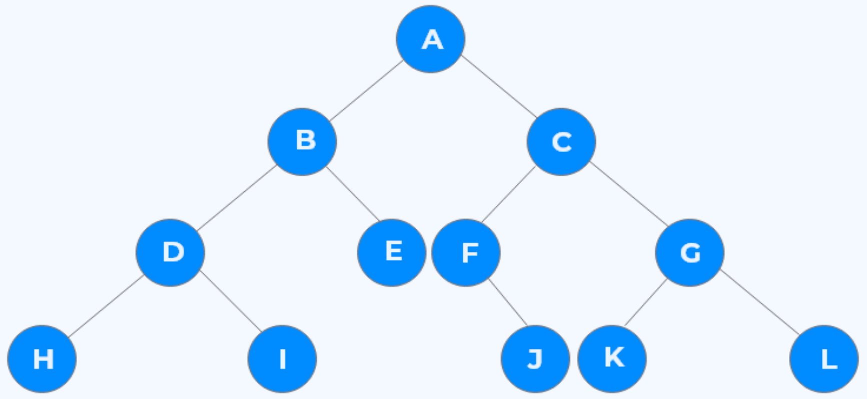
PostOrder:

D E B C A

Inorder:

D B E A C

Inorder Postorder Preorder Example 1



PreOrder:

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

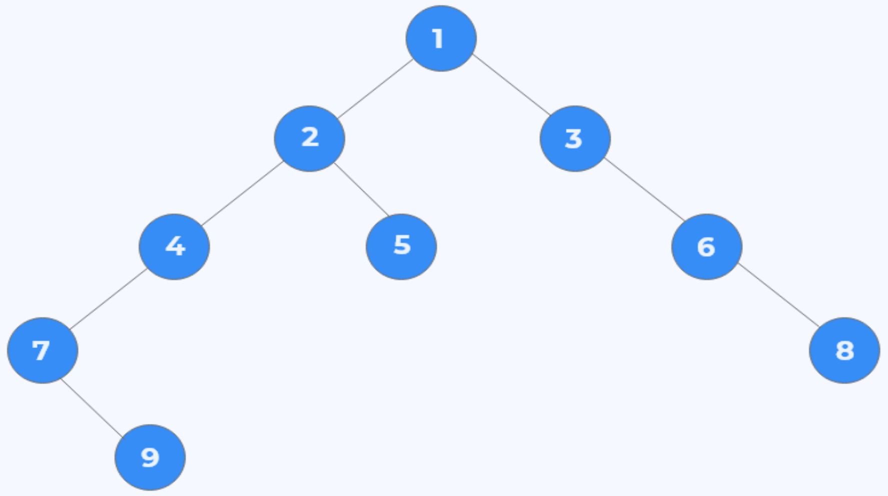
PostOrder:

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

Inorder:

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

Inorder Postorder Preorder Example 2



PreOrder:

1 2 4 7 9 5 3 6 8

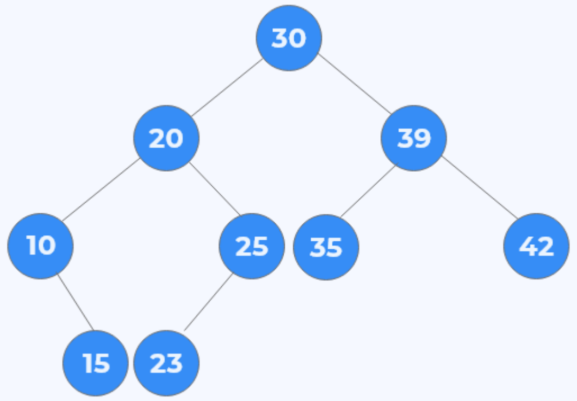
PostOrder:

9 7 4 5 2 8 6 3 1

Inorder:

7 9 4 2 5 1 3 6 8

Inorder Postorder Preorder Example 3



PreOrder:

30 20 10 15 25 23 39 35 42

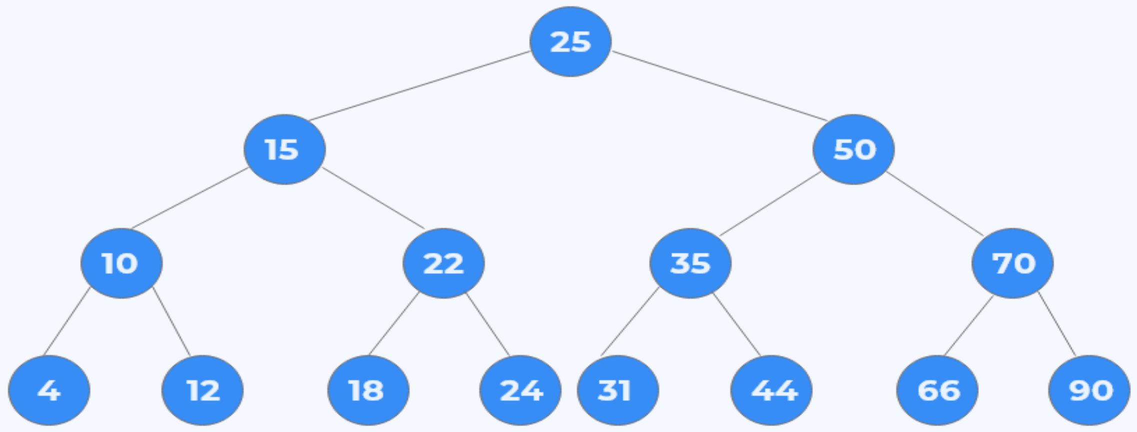
PostOrder:

15 10 23 25 20 35 42 39 30

Inorder:

10 15 20 23 25 30 35 39 42

Inorder Postorder Preorder Example 4



PreOrder:

25 15 10 4 12 22 18 24 50 35 31 44 70 66 90

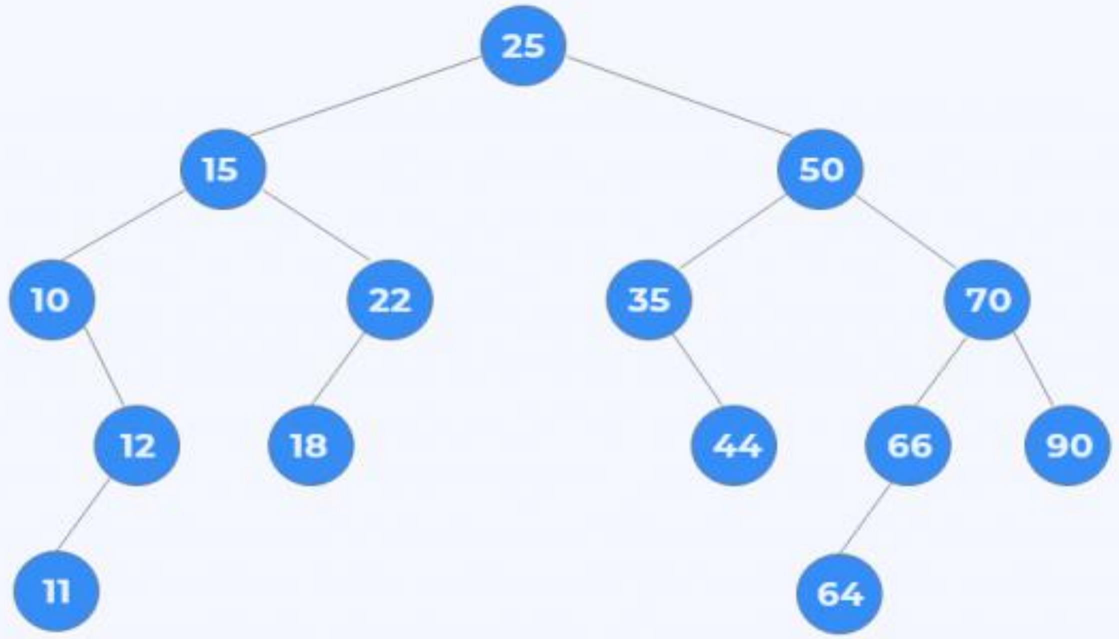
PostOrder:

4 12 10 18 24 22 15 31 44 35 66 90 70 50 25

Inorder:

4 10 12 15 18 22 24 25 31 35 44 50 66 70 90

Inorder Postorder Preorder Example 5



PreOrder:

25 15 10 12 11 22 18 50 35 44 70 66 64 90

PostOrder:

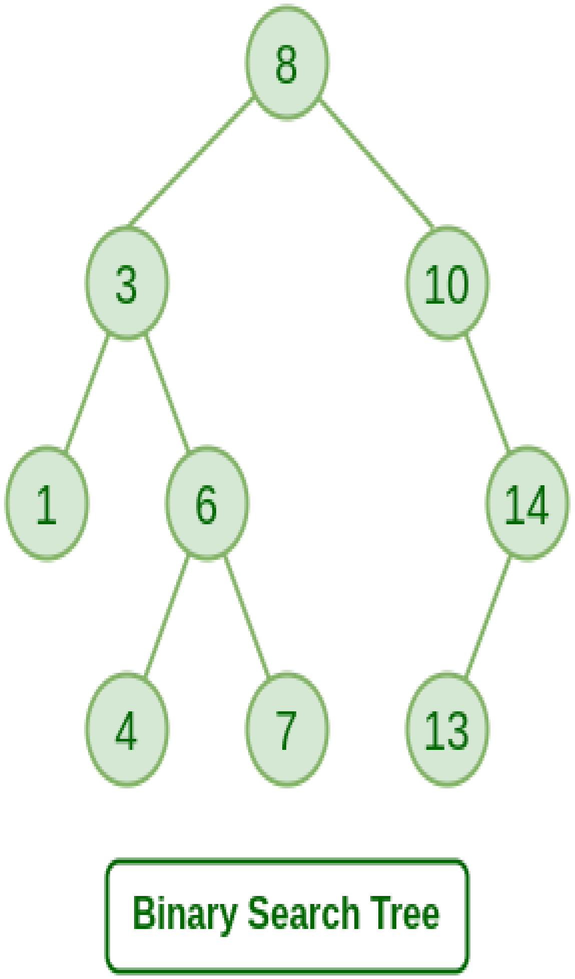
11 12 10 18 22 15 44 35 64 66 90 70 50 25

Inorder:

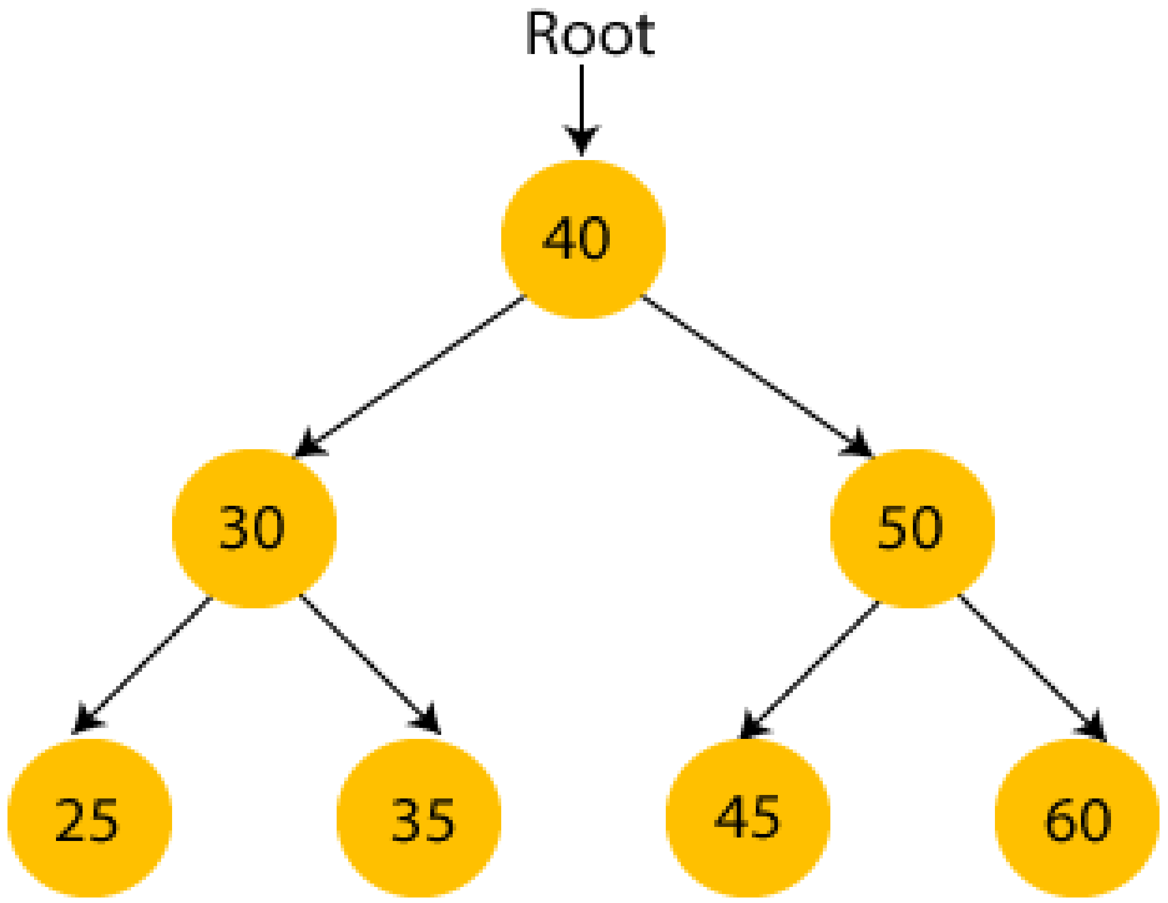
10 11 12 15 18 22 25 35 44 50 64 66 70 90

What is Binary Search Tree?

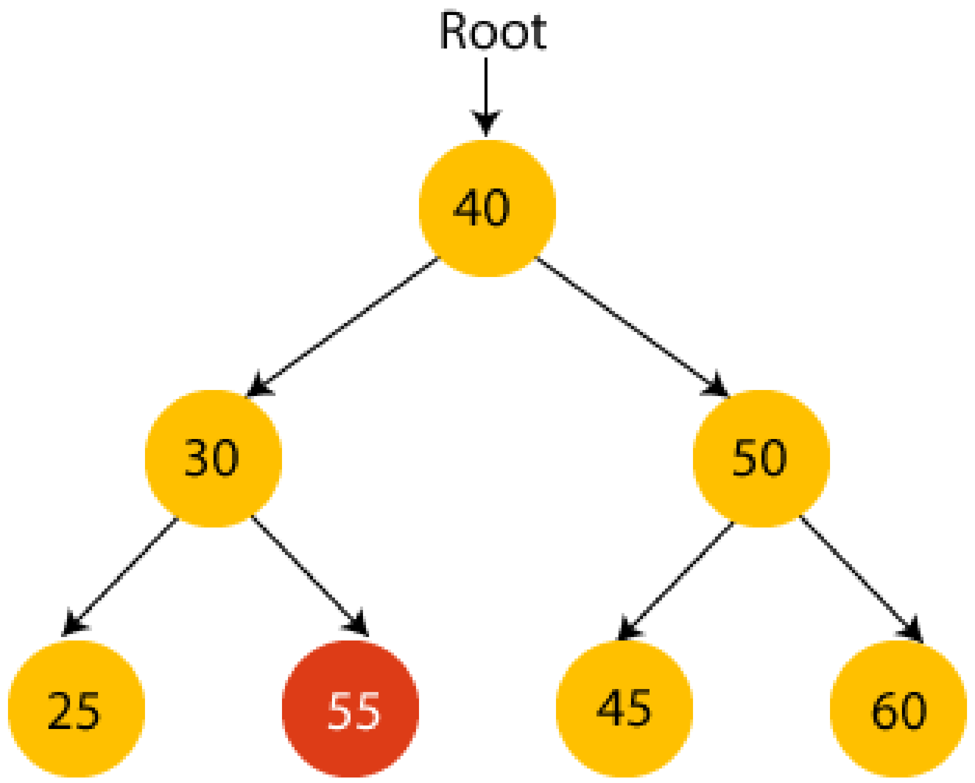
- **Binary Search Tree** is a node-based binary tree data structure which has the following properties:
- The left subtree of a node contains only nodes with keys **lesser than the node's key.**
- The right subtree of a node contains only nodes with keys **greater than the node's key.**
- The left and right subtree each must also be a binary search tree.


8
3
10
1
6
14
4
7
13
Binary Search Tree

What is a Binary Search tree?
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 subtrees of the root.
Let's understand the concept of Binary search tree with an example.


Root
40
30
50
25
35
45
60

In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree 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.
Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.

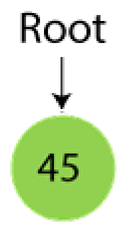

Root
40
30
50
25
55
45
60

In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not 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 subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.
- Example of creating a binary search tree
- Now, let's see the creation of binary search tree using an example.
- Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

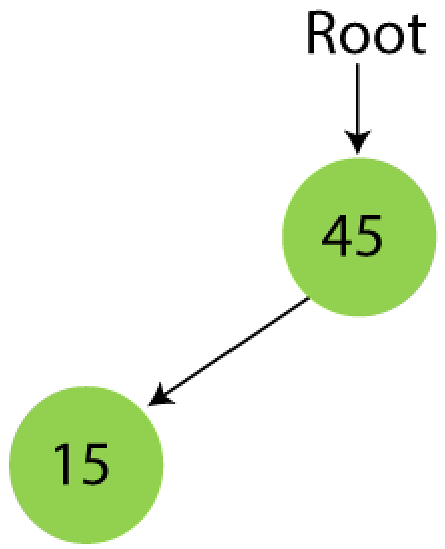
First, we have to insert **45** into the tree as the root of the tree.
Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

- **Step 1 - Insert 45.**



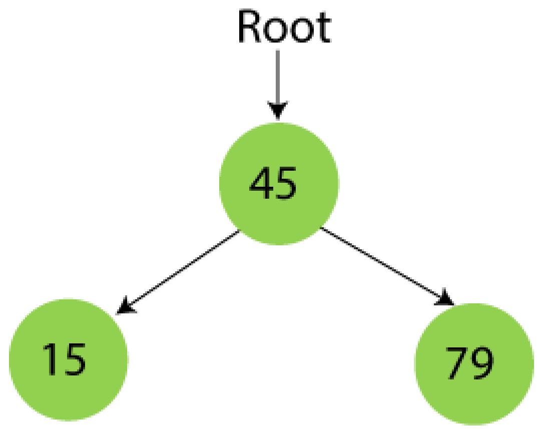
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



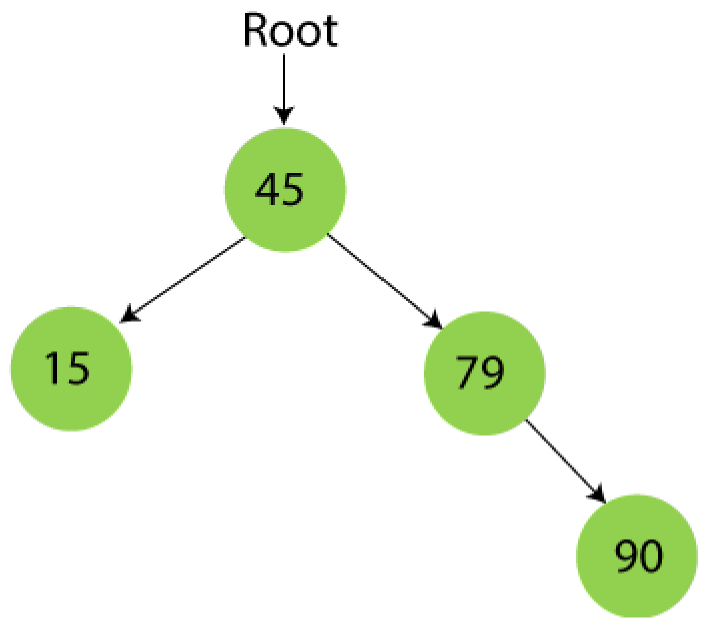
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



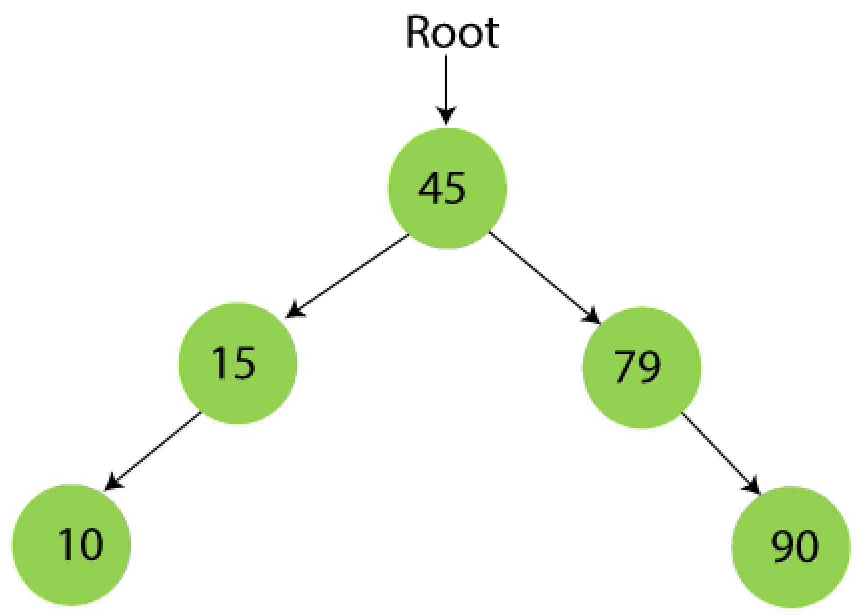
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



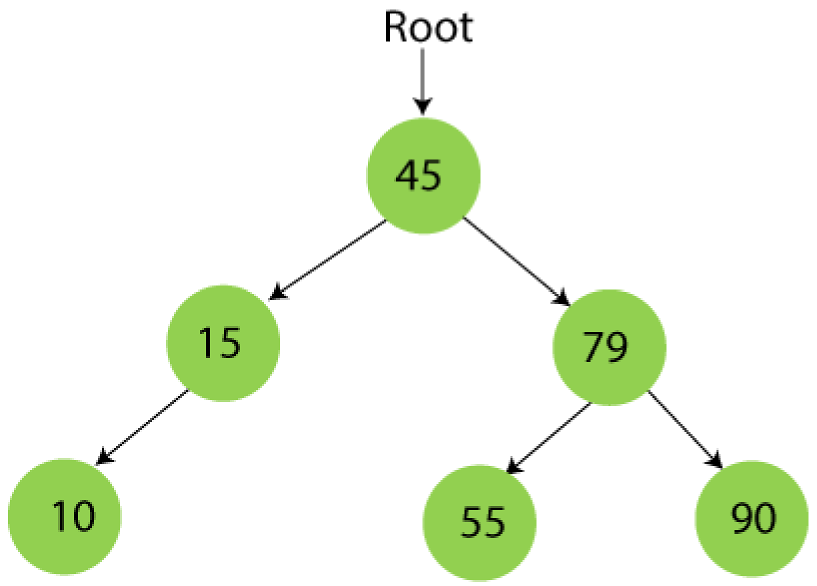
Step 5 - Insert 10:

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



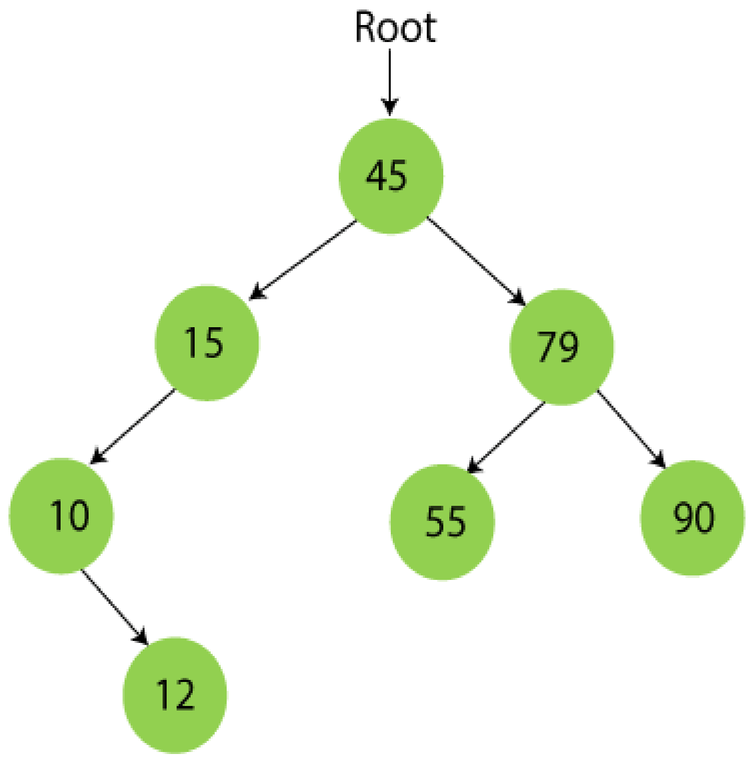
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



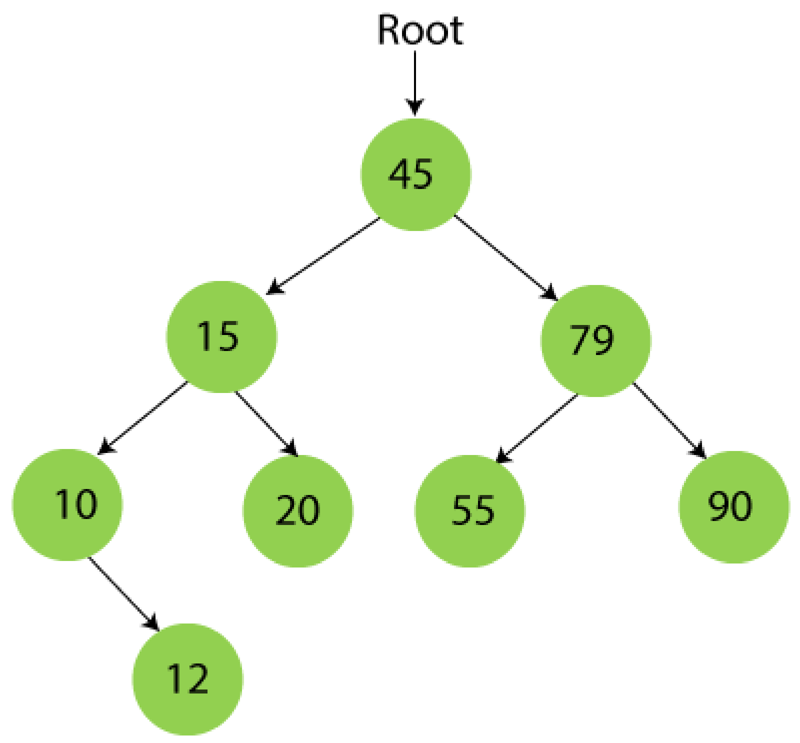
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



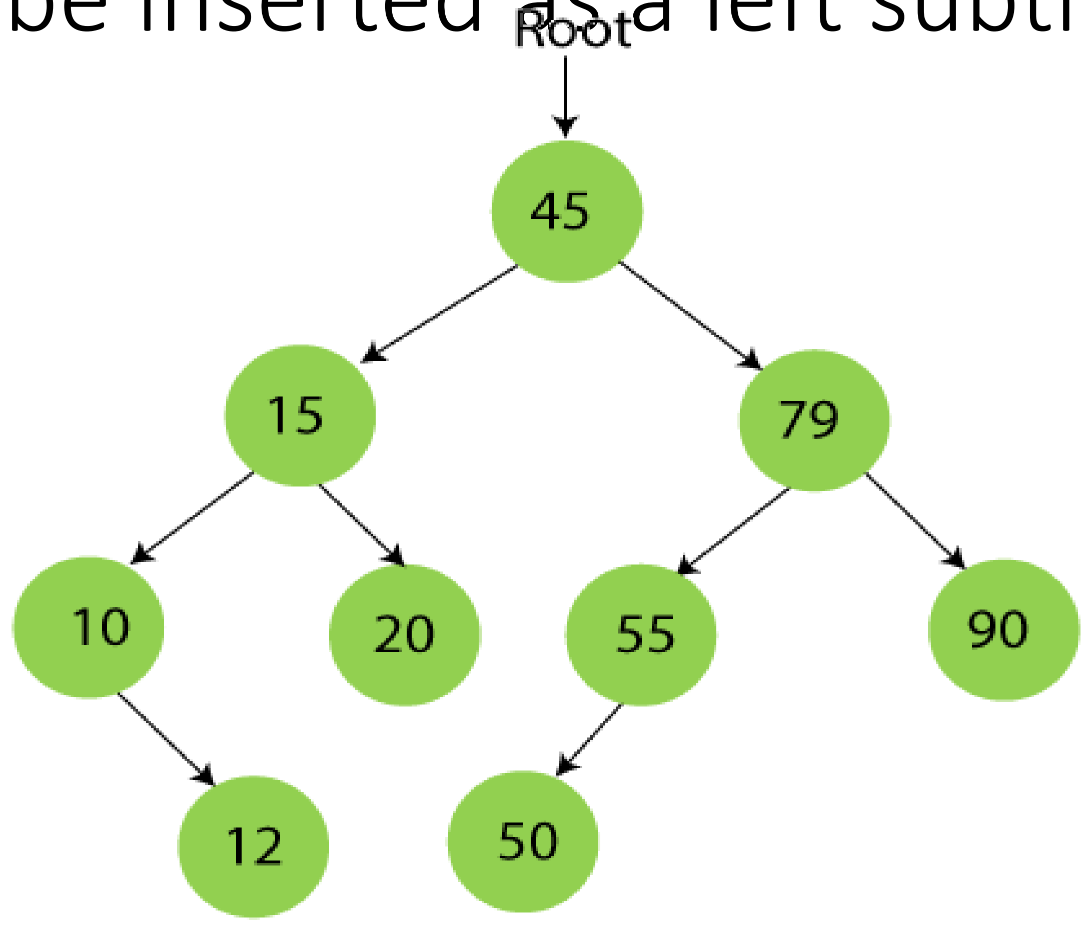
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55.
So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed. After that, let's move towards the operations that can be performed on Binary search tree.

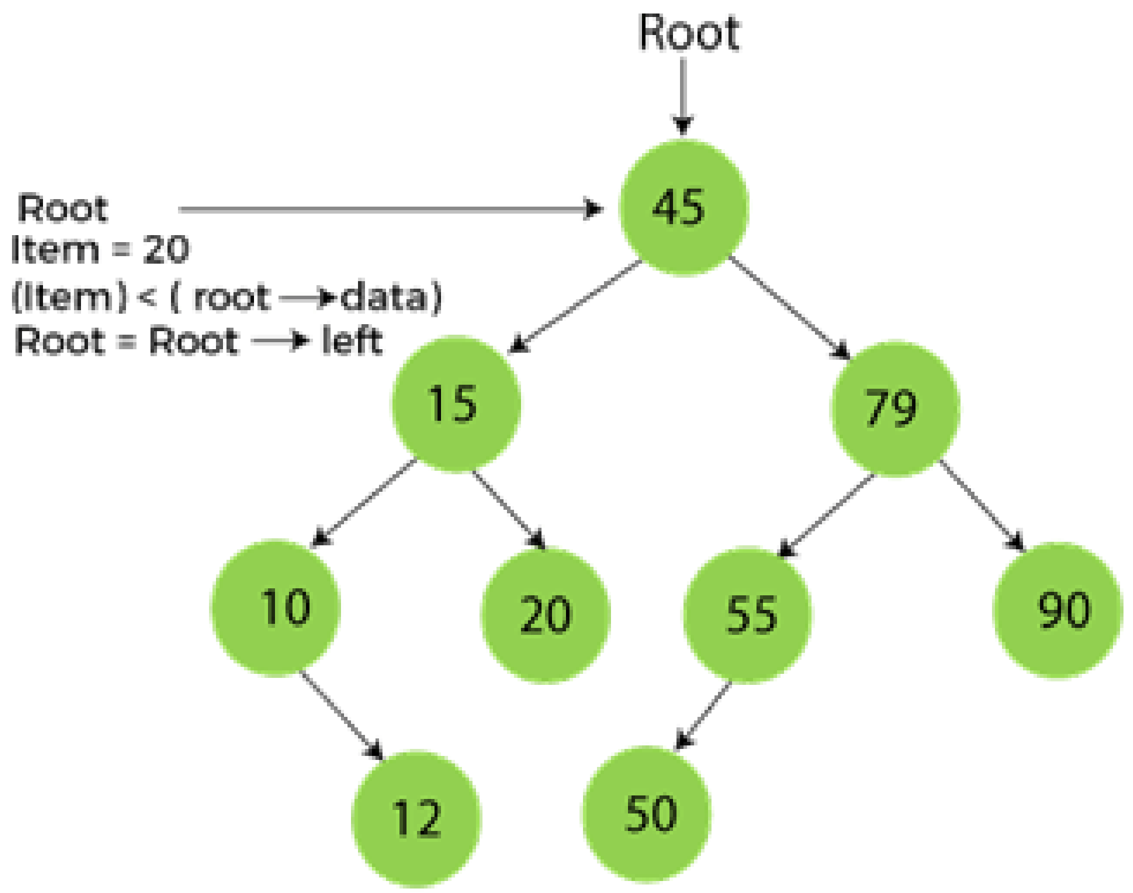
We can perform **insert, delete and search** operations on the binary search tree.

Let's understand how a search is performed on a binary search tree.

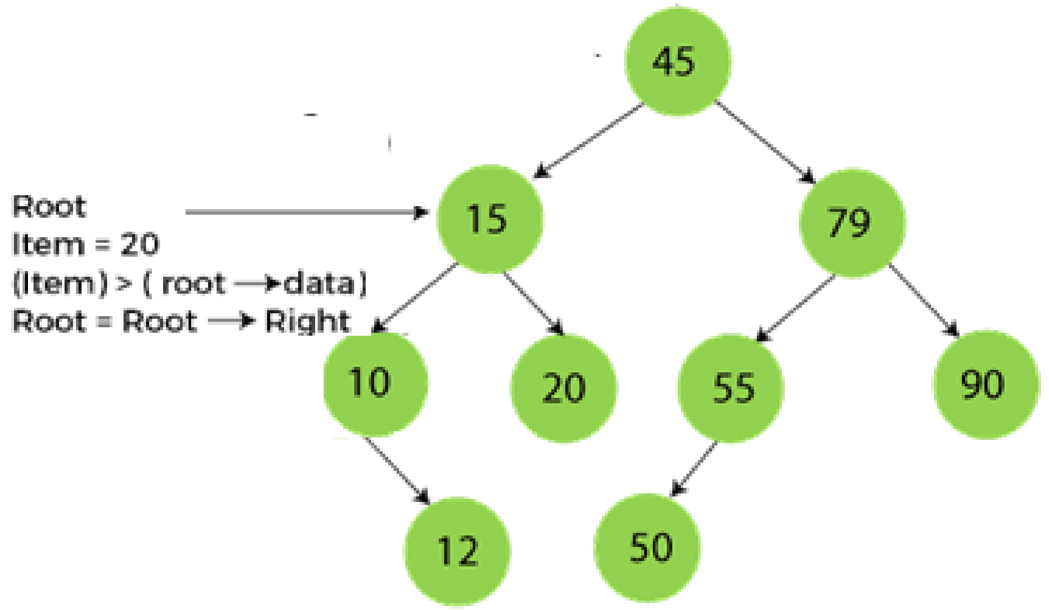
- Searching in Binary search tree
- Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as

First, compare the element to be searched with the root element of the tree.
If root is matched with the target element, then return the node's location.
If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
If it is larger than the root element, then move to the right subtree.
Repeat the above procedure recursively until the match is found.
If the element is not found or not present in the tree, then return NULL.
Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

Step1:

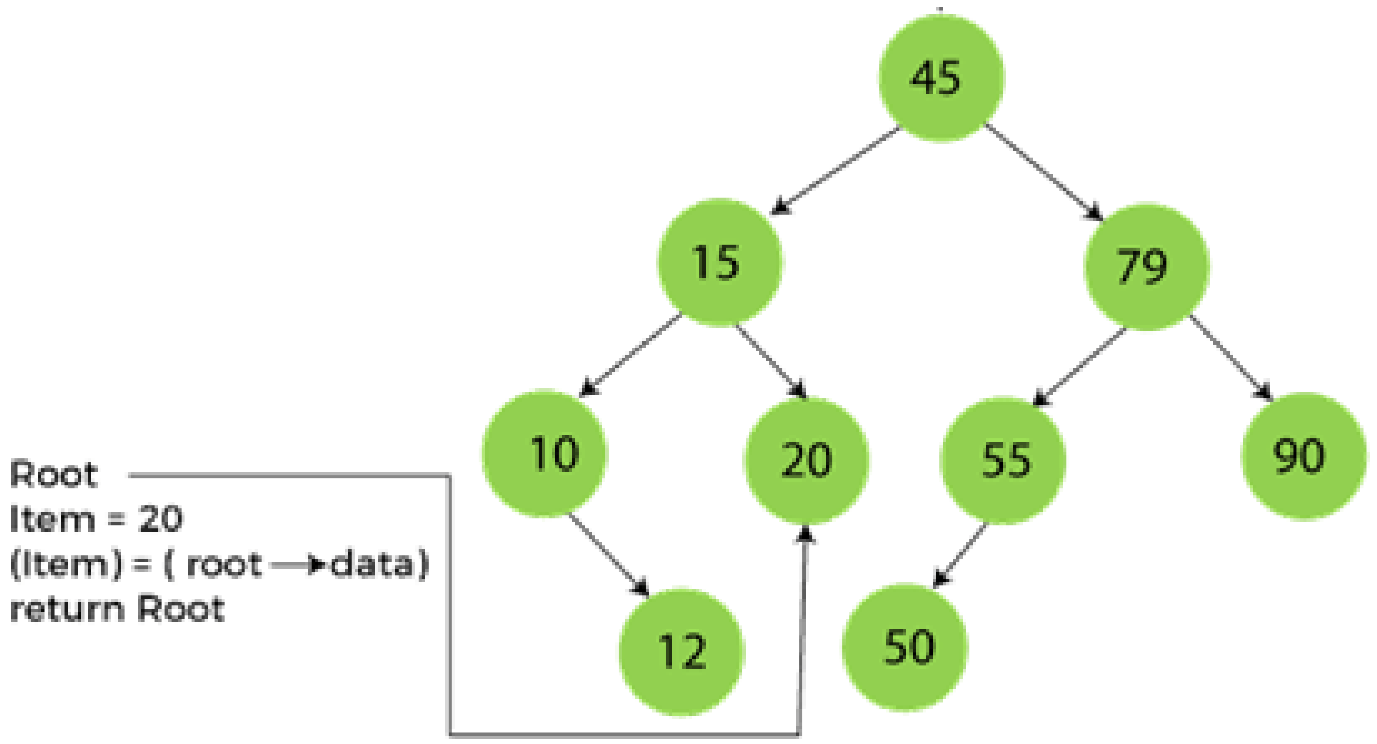

Root
Root
Item = 20
(Item) < (root → data)
Root = Root → left
45
15
79
10
20
55
90
12
50

Step2:



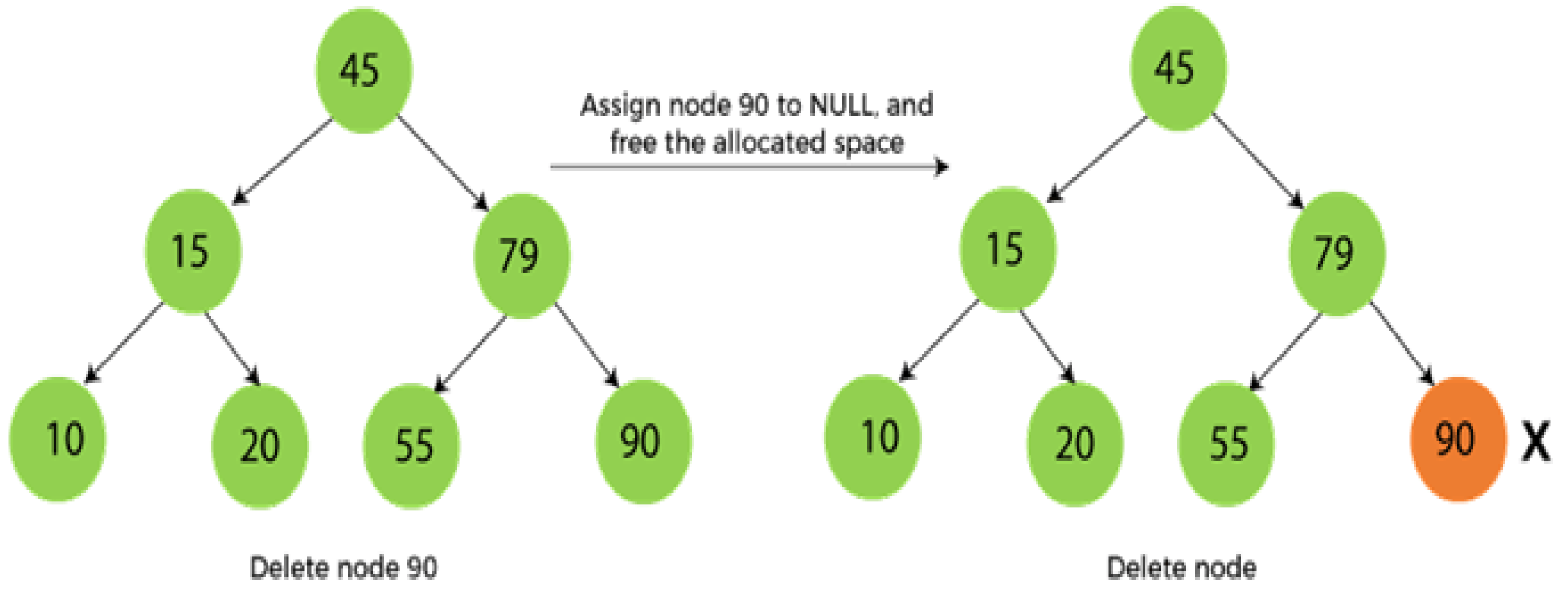
Root
Item = 20
(Item) > (root → data)
Root = Root → Right
45
15
79
10
20
55
90
12
50

Step3:

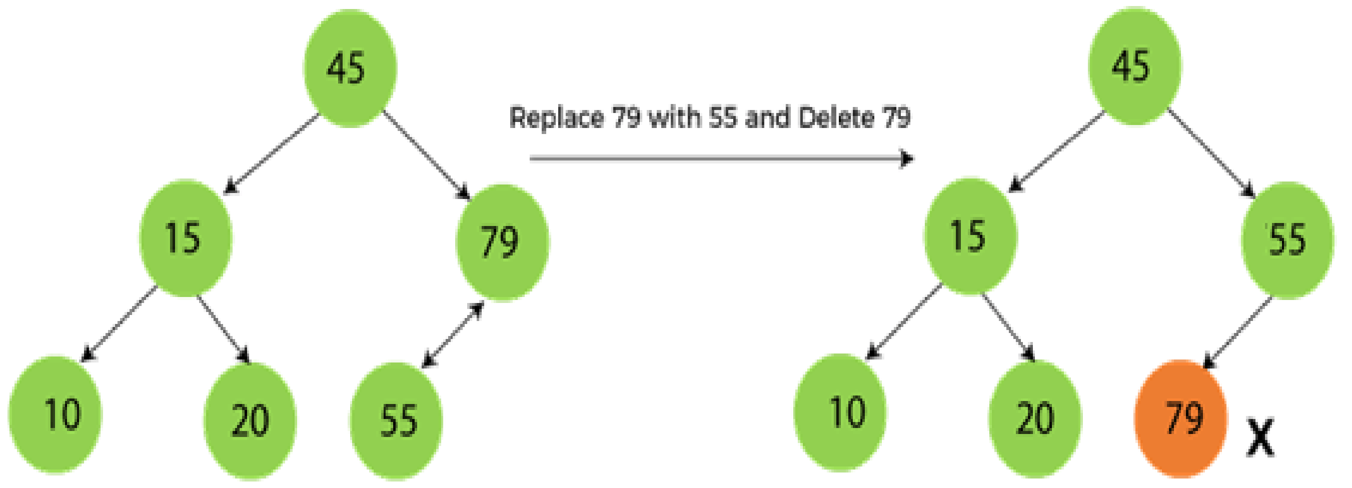

45
15
79
10
20
55
90
12
50
Root
Item = 20
(Item) = (root ⟶ data)
return Root

- Now let's understand how the deletion is performed on a binary search tree. We will also see an example to delete an element from the given tree.
- Deletion in Binary Search tree
- In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -
- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children
- We will understand the situations listed above in detail.

- **When the node to be deleted is the leaf node**
- It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.
- We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.


45
15
79
10
20
55
90
Delete node 90
Assign node 90 to NULL, and free the allocated space
45
15
79
10
20
55
90
X
Delete node

- **When the node to be deleted has only one child**
- In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.
- We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.
- So, the replaced node 79 will now be a leaf node that can be easily deleted.

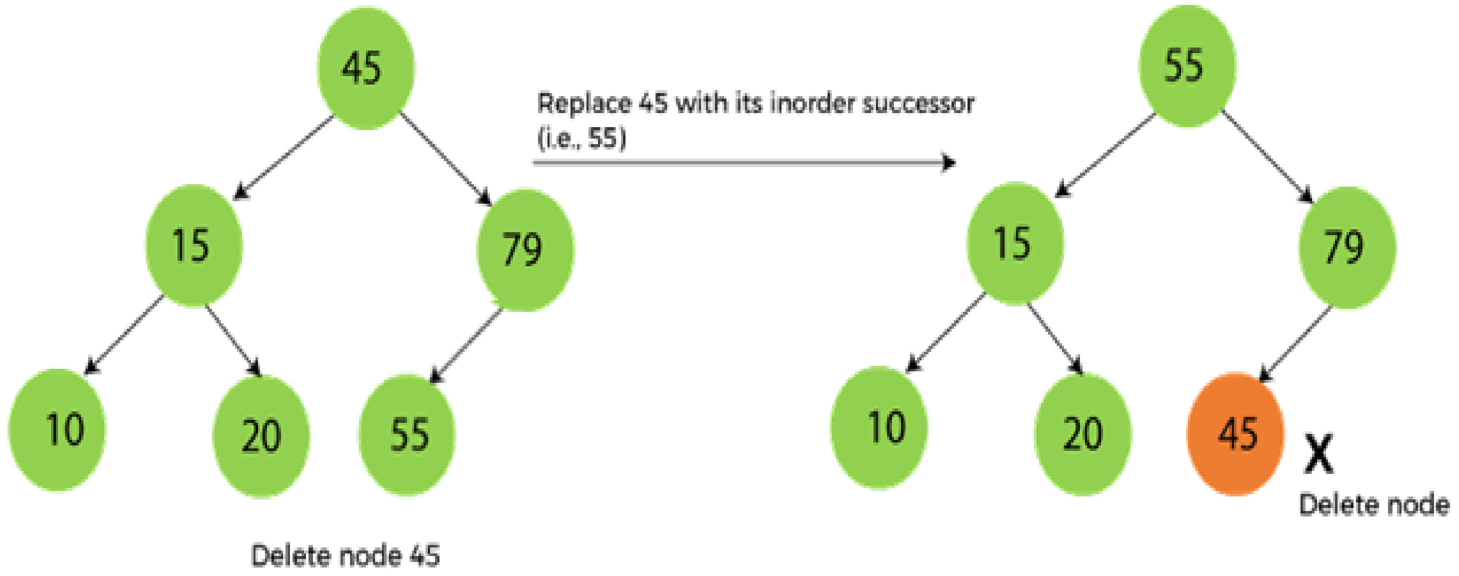


Delete node 79

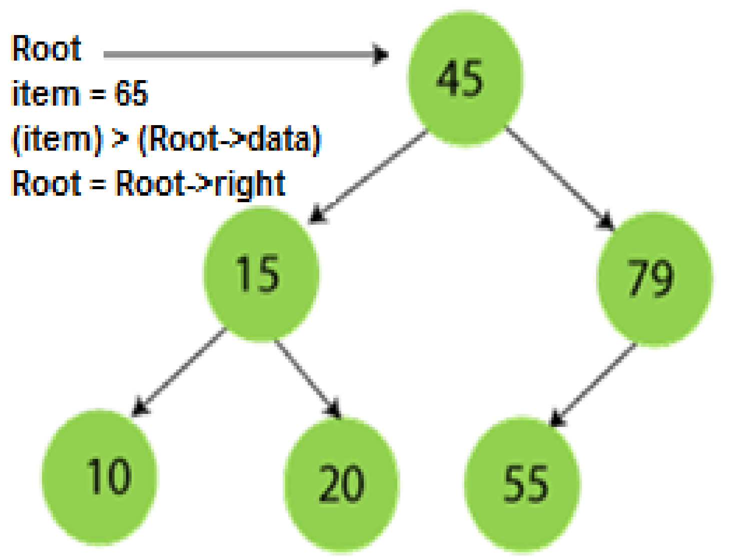
Delete node

- **When the node to be deleted has two children**
- This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -
- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

- The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.
- We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

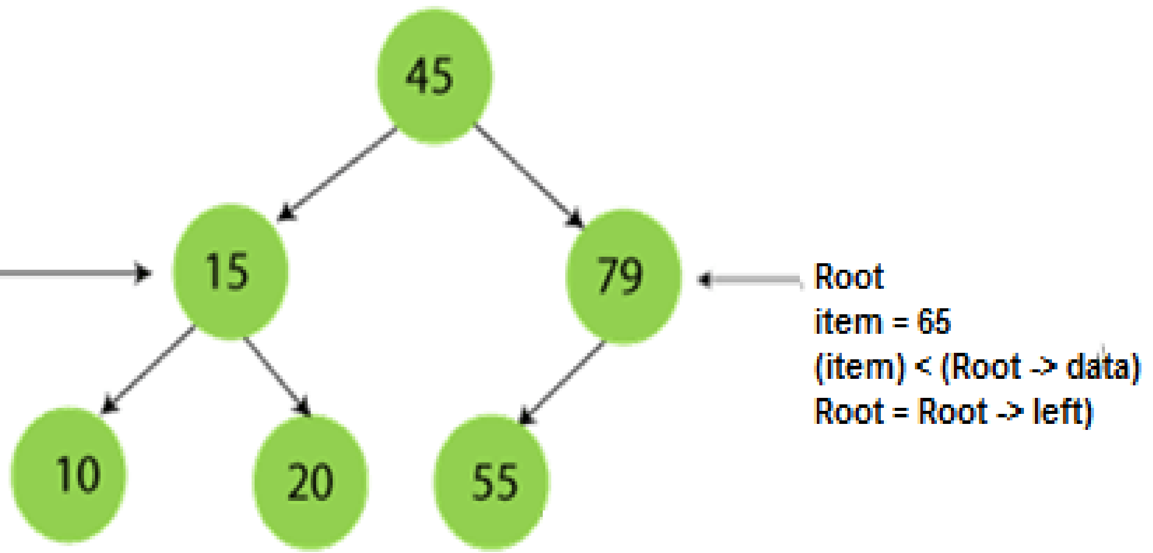

45
15
79
10
20
55
Delete node 45
Replace 45 with its inorder successor (i.e., 55)
55
15
79
10
20
45
X
Delete node

- Insertion in Binary Search tree
- A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.
- Now, let's see the process of inserting a node into BST using an example.



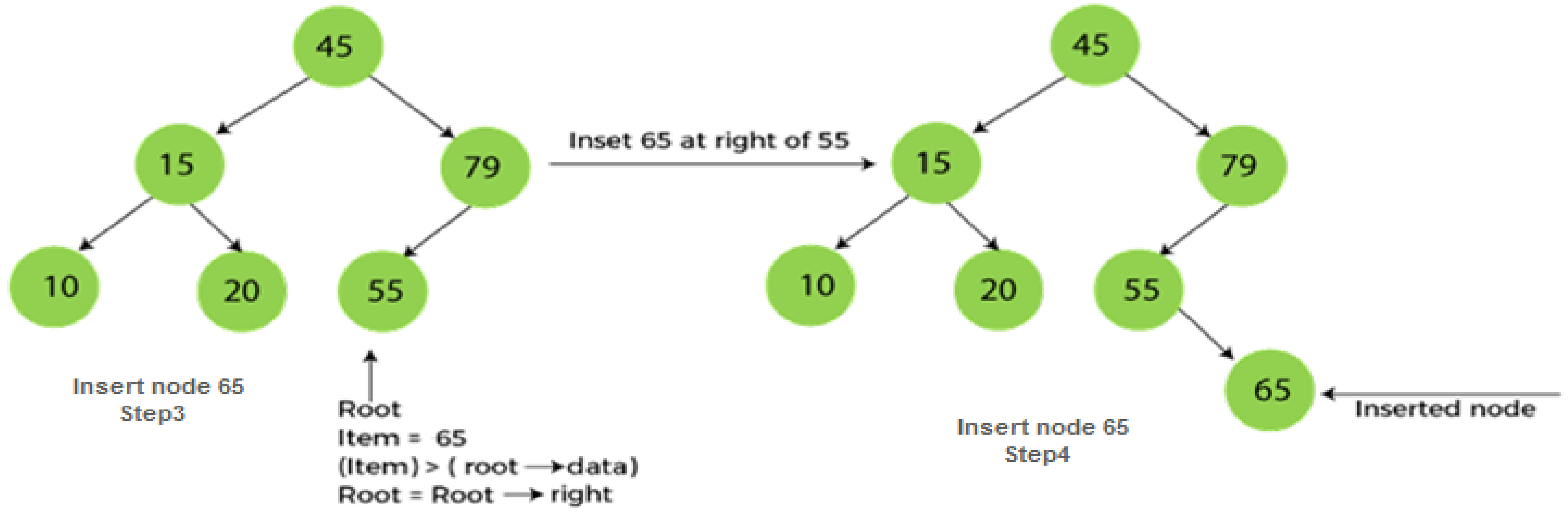
Insert node 65

Step1



Insert node 65

Step2

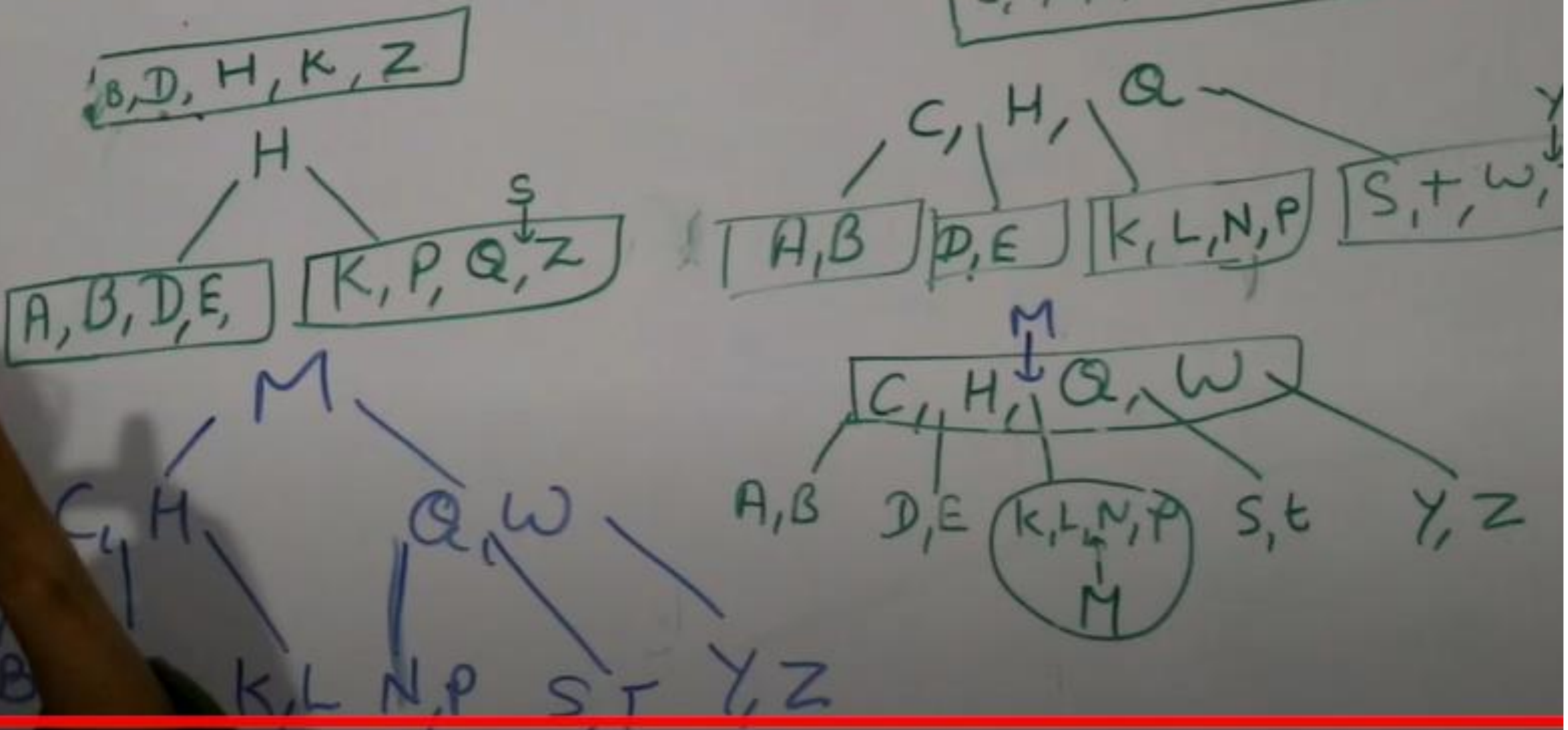

45
15
79
10
20
55
Insert node 65
Step3
Root
Item = 65
(Item) > (root ⟶data)
Root = Root ⟶ right
Inset 65 at right of 55
45
15
79
10
20
55
65
Inserted node
Insert node 65
Step4

Construct a B-tree of order $\boxed{5}$ with the following set of data :-

D, H, Z, K, B, P, Q, E, A, S, W, T, C, L, N, Y, M

m = 5

A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P,
S, T, U, V, W, X, Y, Z


B, D, H, K, Z
H
S
A, B, D, E,
K, P, Q, Z
M
C, H
Q, W
A, B
K, L
N, P
S, T
Y, Z
C, H, Q
Y
A, B
D, E
K, L, N, P
S, T, W,
M
C, H, Q, W
A, B
D, E
K, L, N, P
M
S, t
Y, Z