Binary Tree (Array implementation)

Given an array that represents a tree in such a way that array indexes are values in tree nodes and array values

give the parent node of that particular index (or node). The value of the root node index would always be -1

as there is no parent for root.

Input: parent[] = {1, 5, 5, 2, 2, -1, 3}

Output: root of below tree

5

/ \

1 2

/ / \

0 3 4

/

6

Explanation:

Index of -1 is 5. So 5 is root.

5 is present at indexes 1 and 2. So 1 and 2 are

children of 5.

1 is present at index 0, so 0 is child of 1.

2 is present at indexes 3 and 4. So 3 and 4 are

children of 2.

3 is present at index 6, so 6 is child of 3.

Input: parent[] = {-1, 0, 0, 1, 1, 3, 5};

Output: root of below tree

0

/ \

1 2

/ \

3 4

/

5

/

6

Expected time complexity is O(n) where n is number of elements in given array.

A Simple Solution to recursively construct by first searching the current root, then recurring for the found

indexes (there can be at most two indexes) and making them left and right subtrees of root. This solution

takes O(n2) as we have to linearly search for every node.

An Efficient Solution can solve the above problem in O(n) time. The idea is to use extra space. An array

created[0..n-1] is used to keep track of created nodes.

createTree(parent[], n)

1. Create an array of pointers say created[0..n-1]. The value of created[i] is NULL if node for index i is

not created, else value is pointer to the created node.

2. Do following for every index i of given array

createNode(parent, i, created)

createNode(parent[], i, created[])

1. If created[i] is not NULL, then node is already created. So return.

2. Create a new node with value ‘i’.

3. If parent[i] is -1 (i is root), make created node as root and return.

4. Check if parent of ‘i’ is created (We can check this by checking if created[parent[i]] is NULL or not.

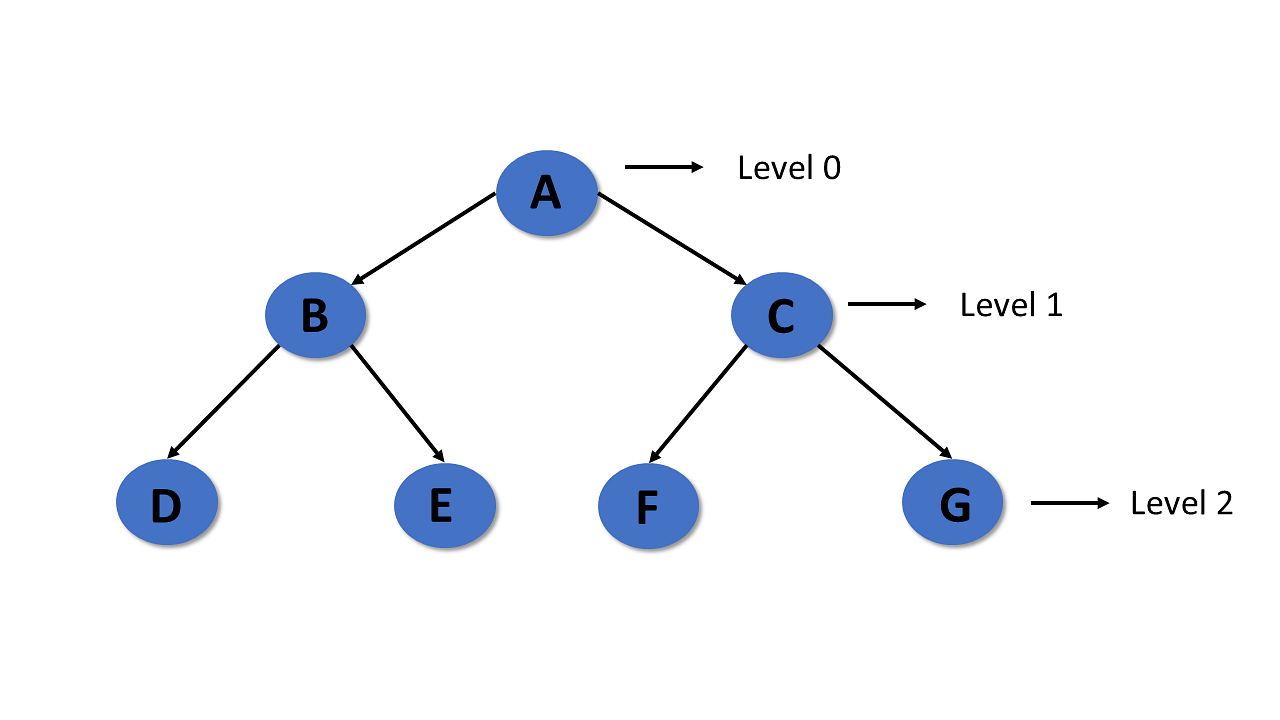
5. If parent is not created, recur for parent and create the parent first.

6. Let the pointer to parent be p. If p->left is NULL, then make the new node as left child. Else make

the new node as right child of parent.

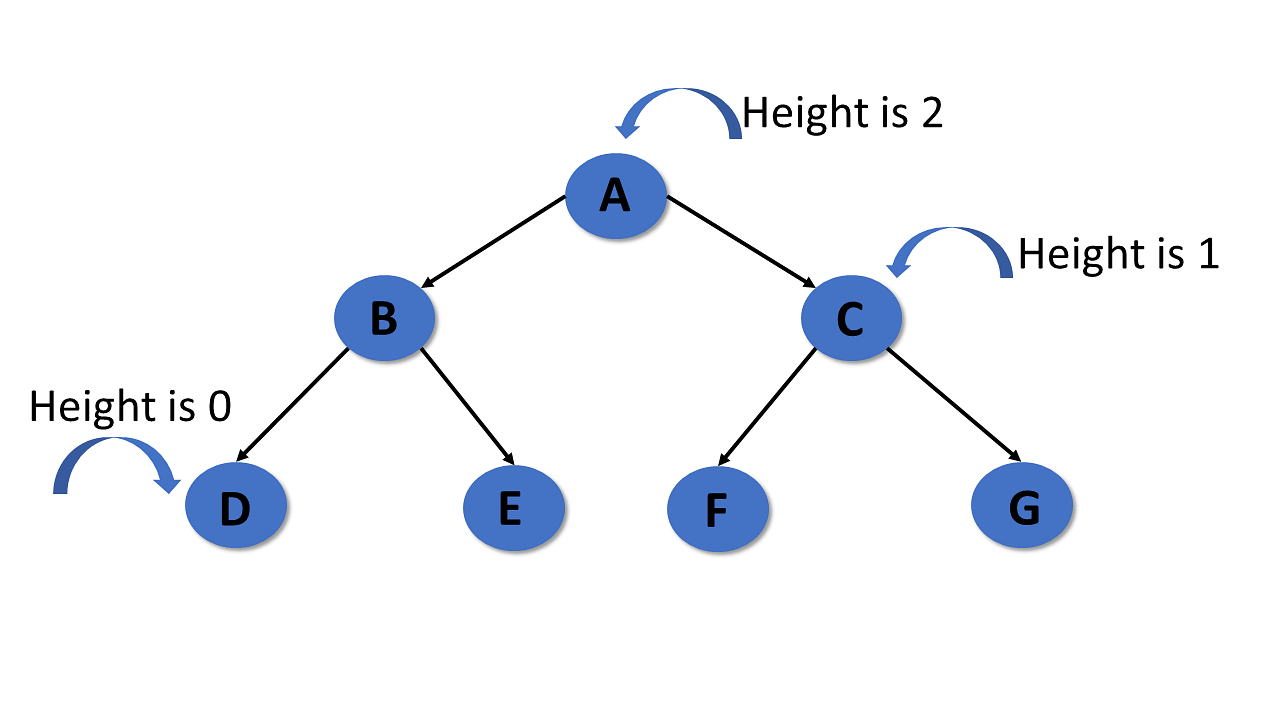
Level

In tree data structures, the root node is said to be at level 0, and the root node's children are at level 1, and the children of that node at level 1 will be level 2, and so on.



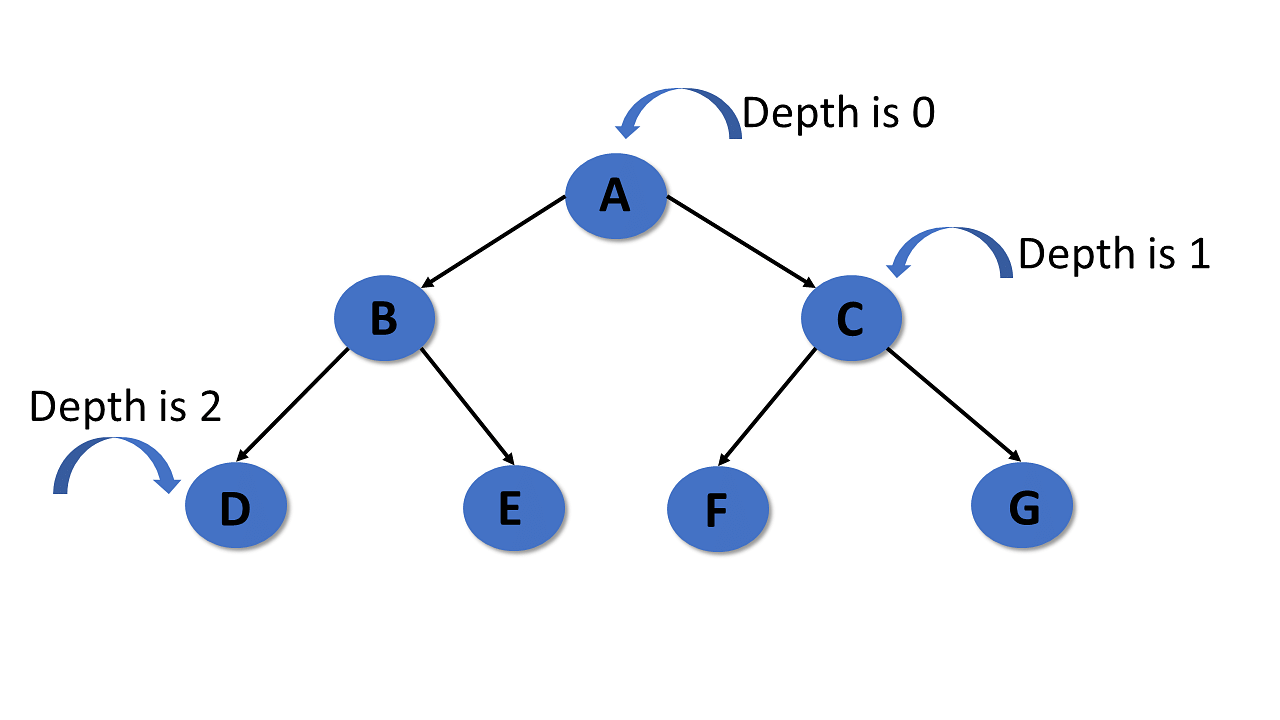
Height

* In a tree data structure, the number of edges from the leaf node to the particular node in the longest path is known as the height of that node.
* In the tree, the height of the root node is called "Height of Tree".
* The tree height of all leaf nodes is 0.



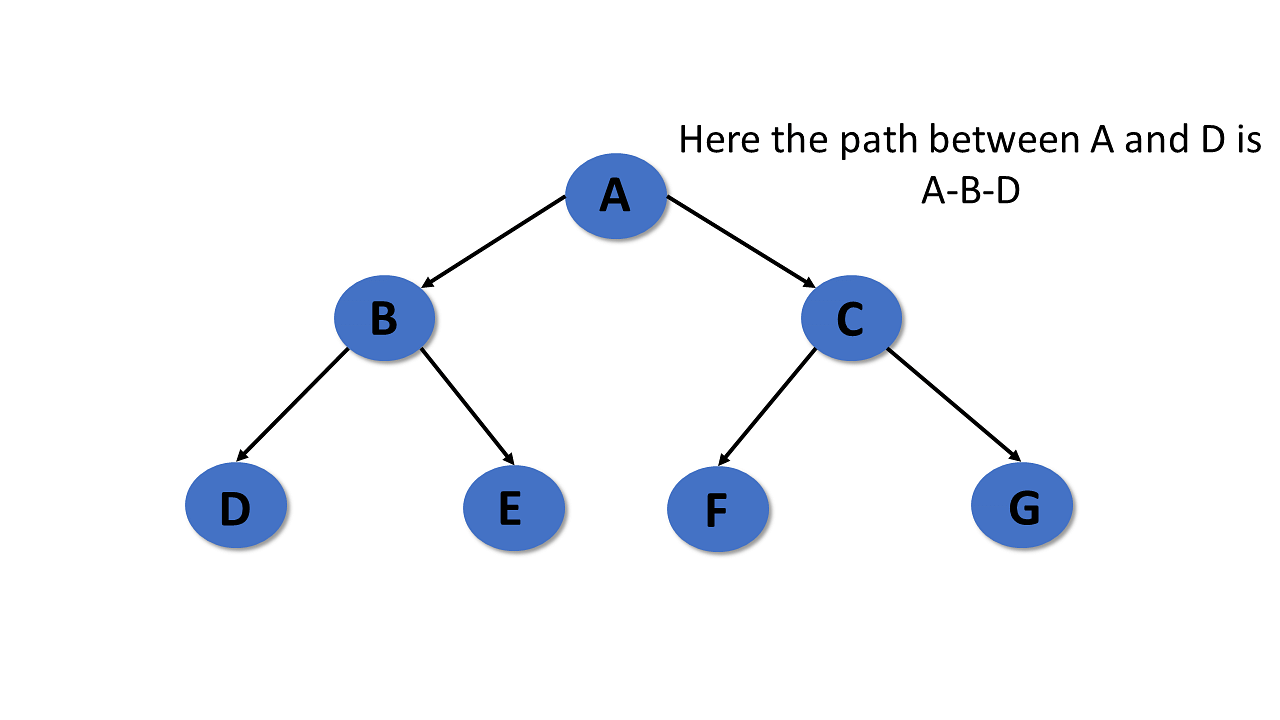
Depth

* In a tree, many edges from the root node to the particular node are called the depth of the tree.
* In the tree, the total number of edges from the root node to the leaf node in the longest path is known as "Depth of Tree".
* In the tree data structures, the depth of the root node is 0.



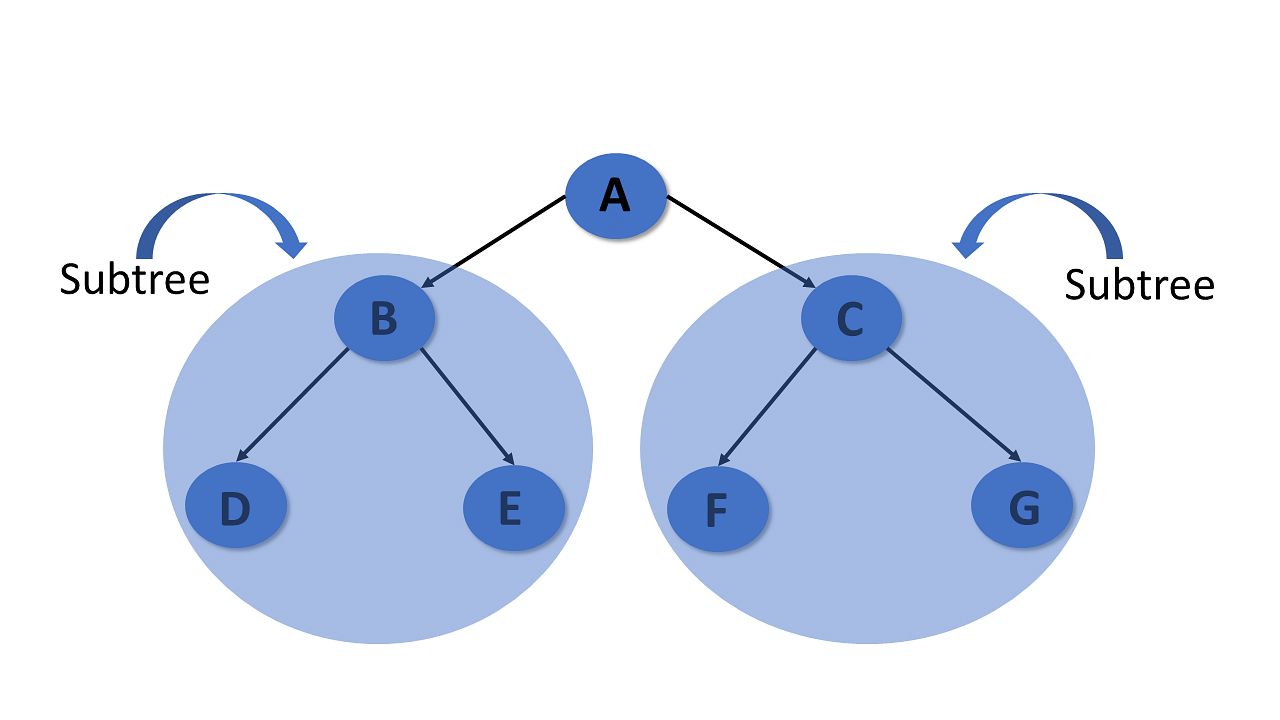
Path

* In the tree in data structures, the sequence of nodes and edges from one node to another node is called the path between those two nodes.
* The length of a path is the total number of nodes in a path.zx



Subtree

In the tree in data structures, each child from a node shapes a sub-tree recursively and every child in the tree will form a sub-tree on its parent node.



Types of Tree in Data Structures

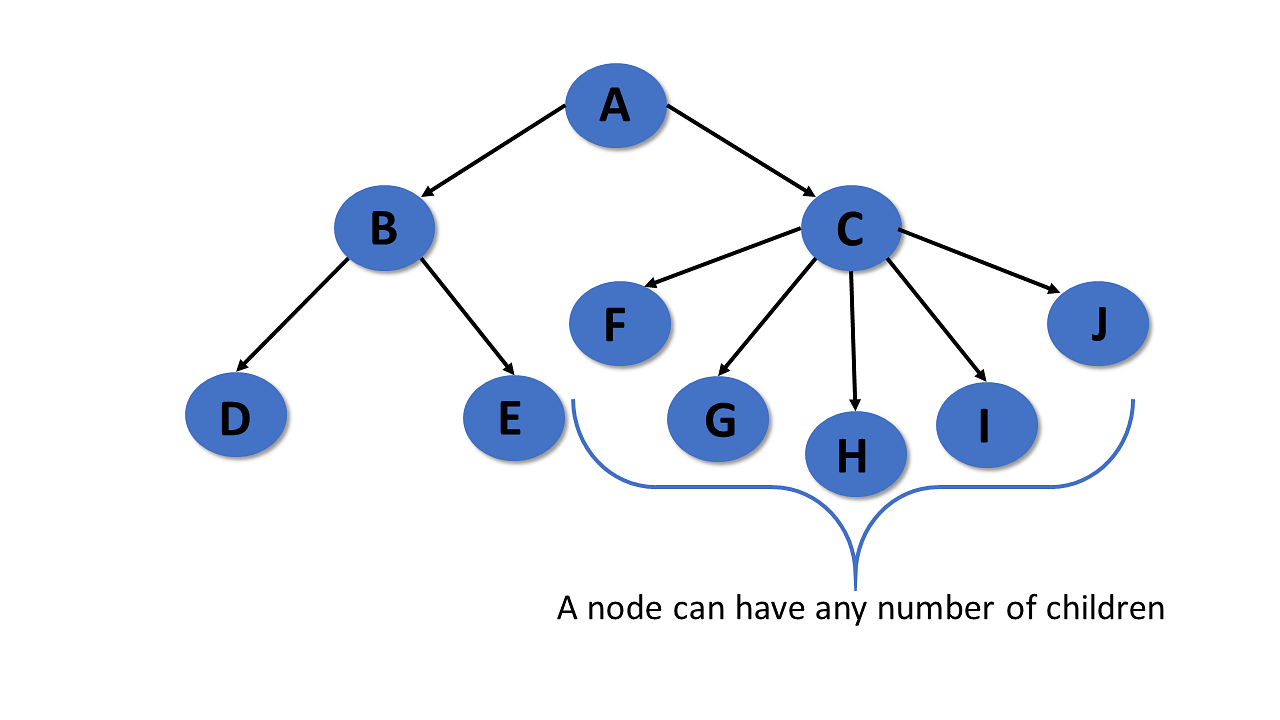
Here are the different kinds of tree in data structures:

General Tree

The general tree is the type of tree where there are no constraints on the hierarchical structure.

Properties

* The general tree follows all properties of the tree data structure.
* A node can have any number of nodes.

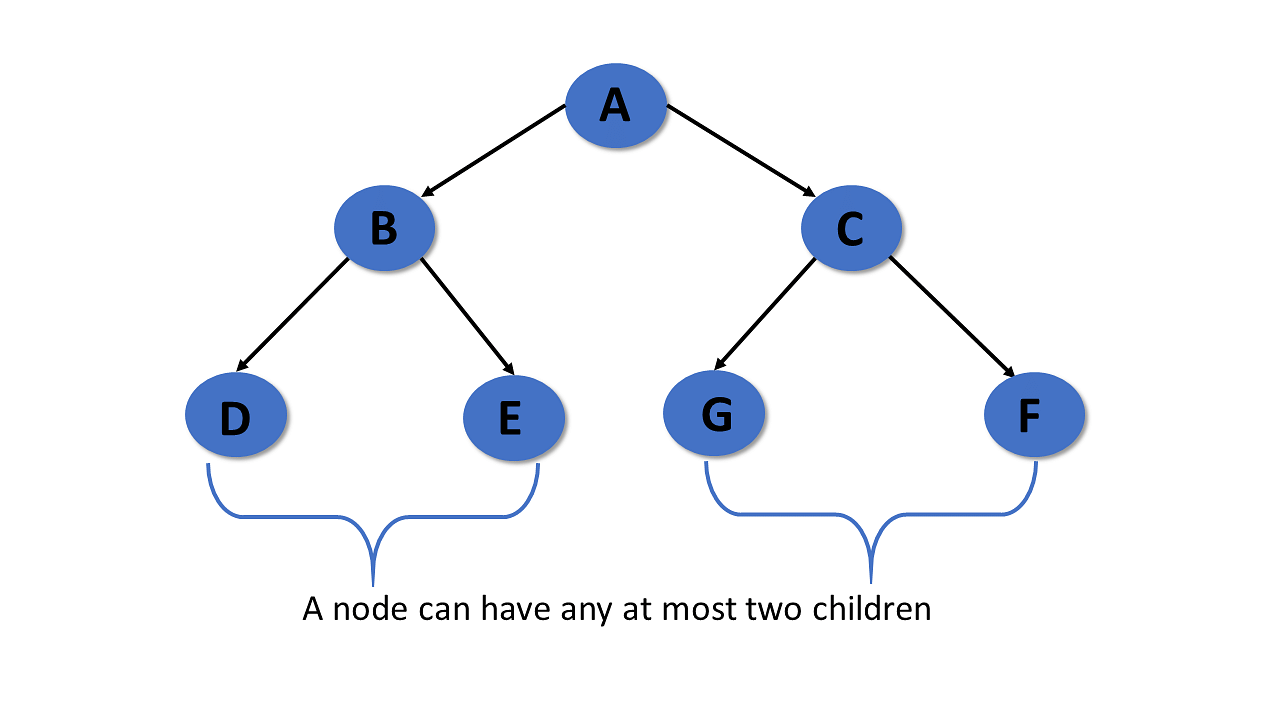


Binary Tree

A binary tree has the following properties:

Properties

* Follows all properties of the tree data structure.
* Binary trees can have at most two child nodes.
* These two children are called the left child and the right child.



Tree Traversal

Traversal of the tree in data structures is a process of visiting each node and prints their value. There are three ways to traverse tree data structure.

* Pre-order Traversal
* In-Order Traversal
* Post-order Traversal

In-Order Traversal

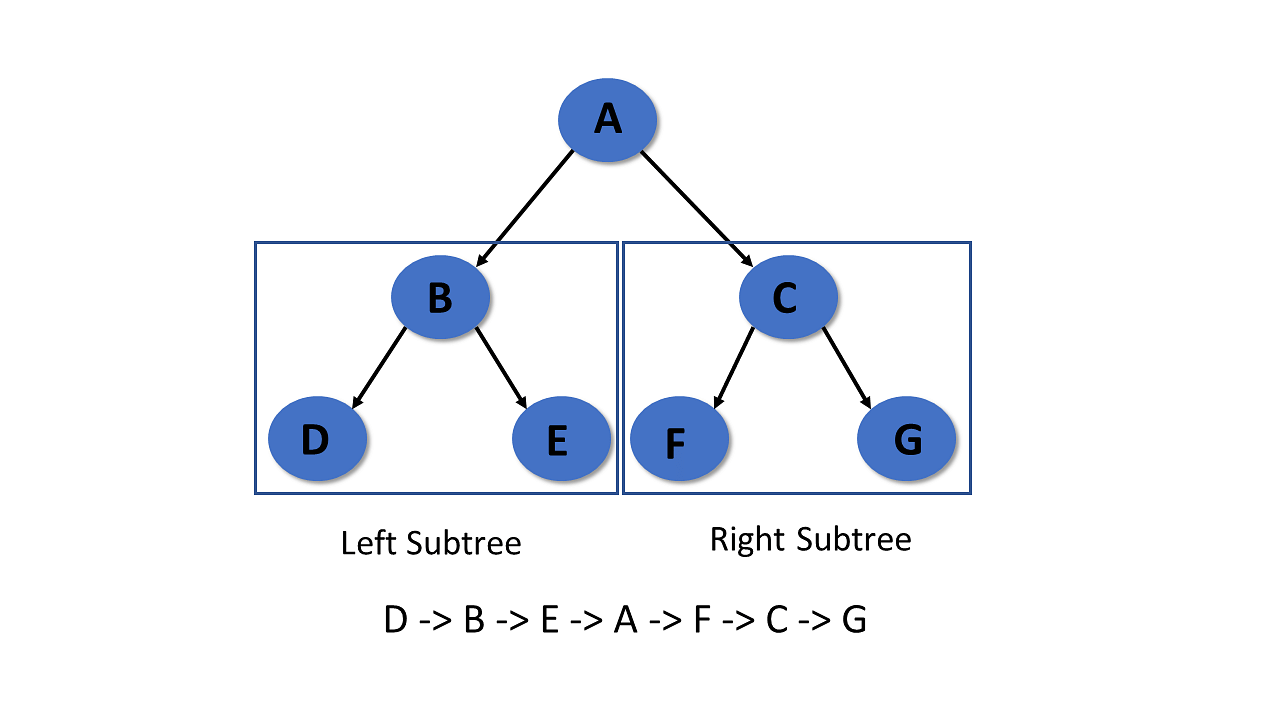
In the in-order traversal, the left subtree is visited first, then the root, and later the right subtree.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree



Pre-Order Traversal

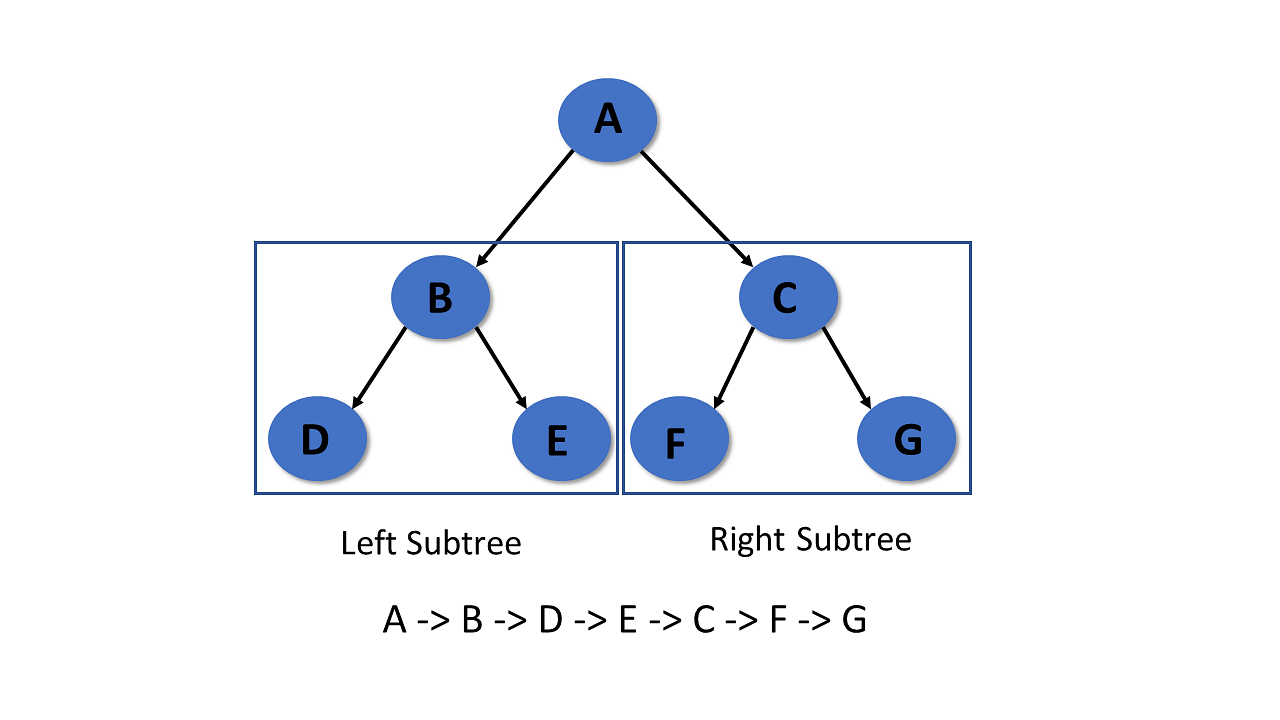
In pre-order traversal, it visits the root node first, then the left subtree, and lastly right subtree.

Algorithm:

Step 1- Visit root node

Step 2- Recursively traverse the left subtree

Step 3- Recursively traverse right subtree



Post-Order Traversal

It visits the left subtree first in post-order traversal, then the right subtree, and finally the root node.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Visit root node

Step 3- Recursively traverse right subtree

