DATA STRUCTURES AND ALGORITHMS

DR SAMABIA TEHSIN

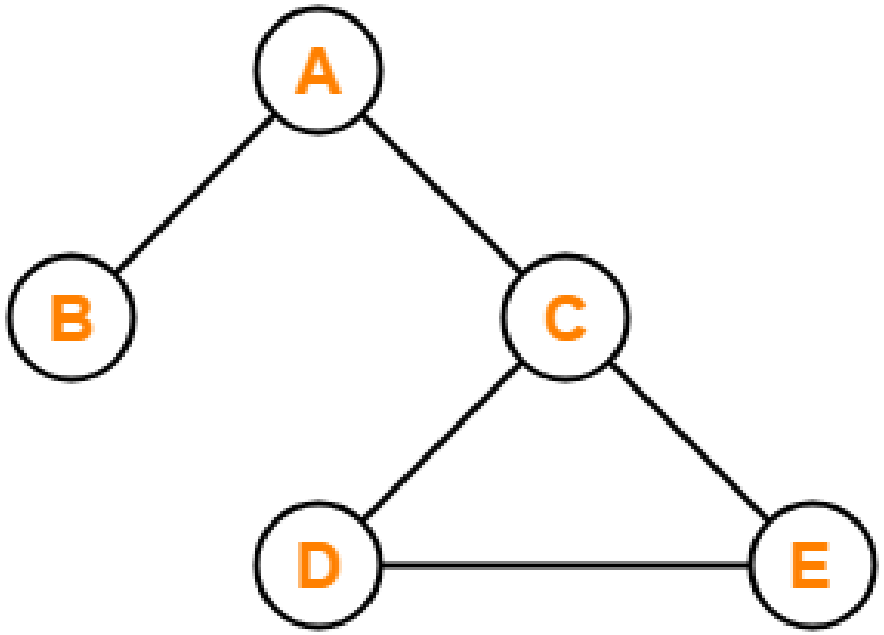
BS (AI)

Tree

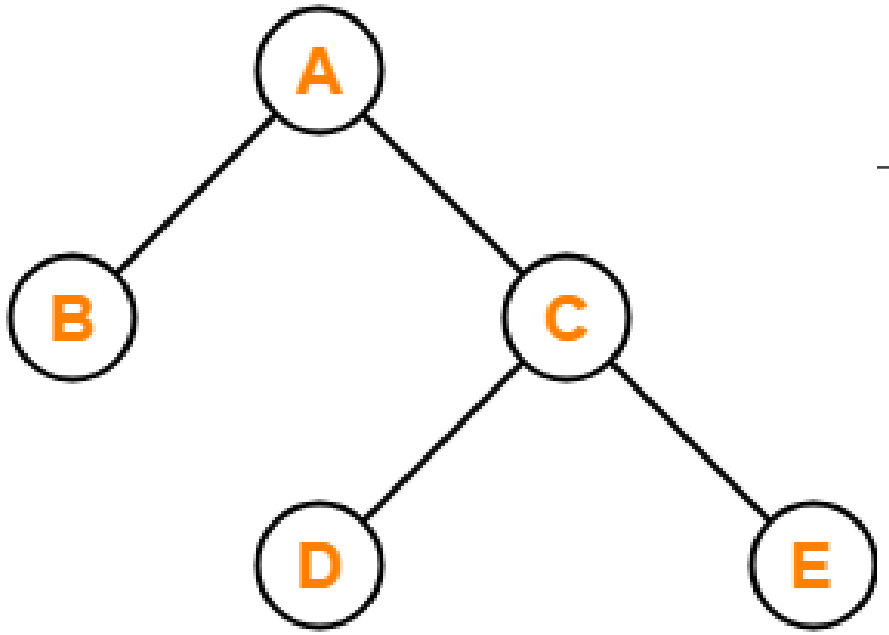
Tree is a non-linear data structure which organizes data in a hierarchical structure, and this is a recursive definition.

A tree is a connected graph without any circuits.

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



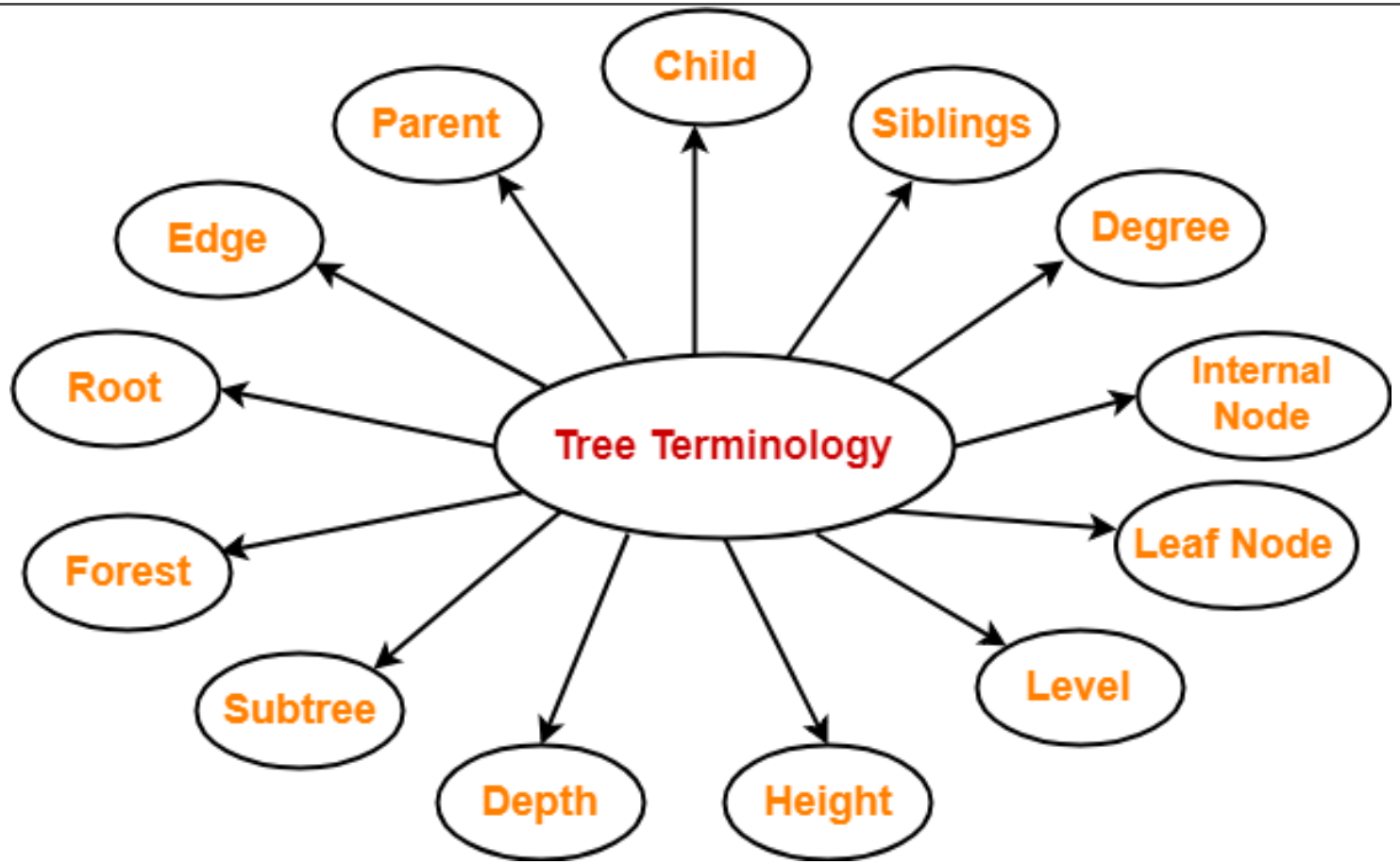
This graph is a Tree

Properties

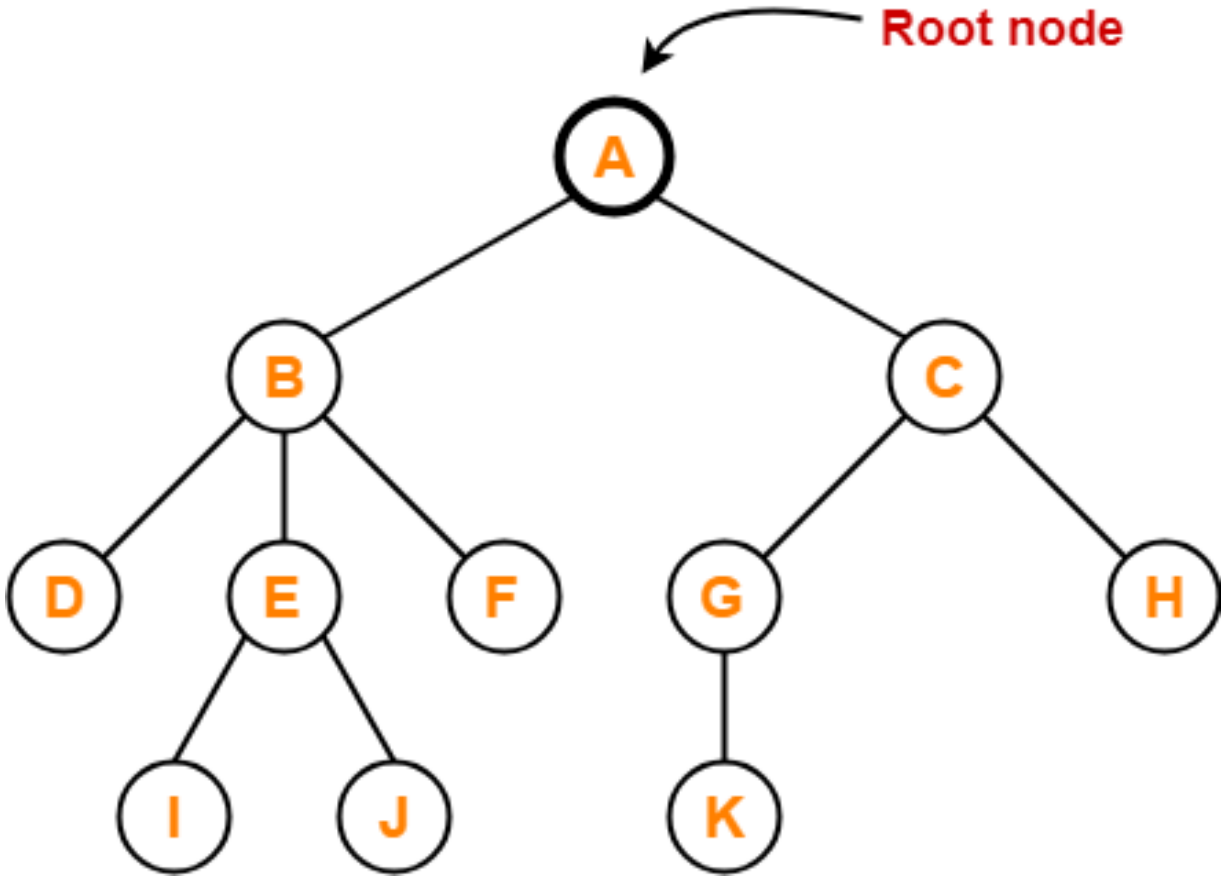
The important properties of tree data structure are-

- 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.

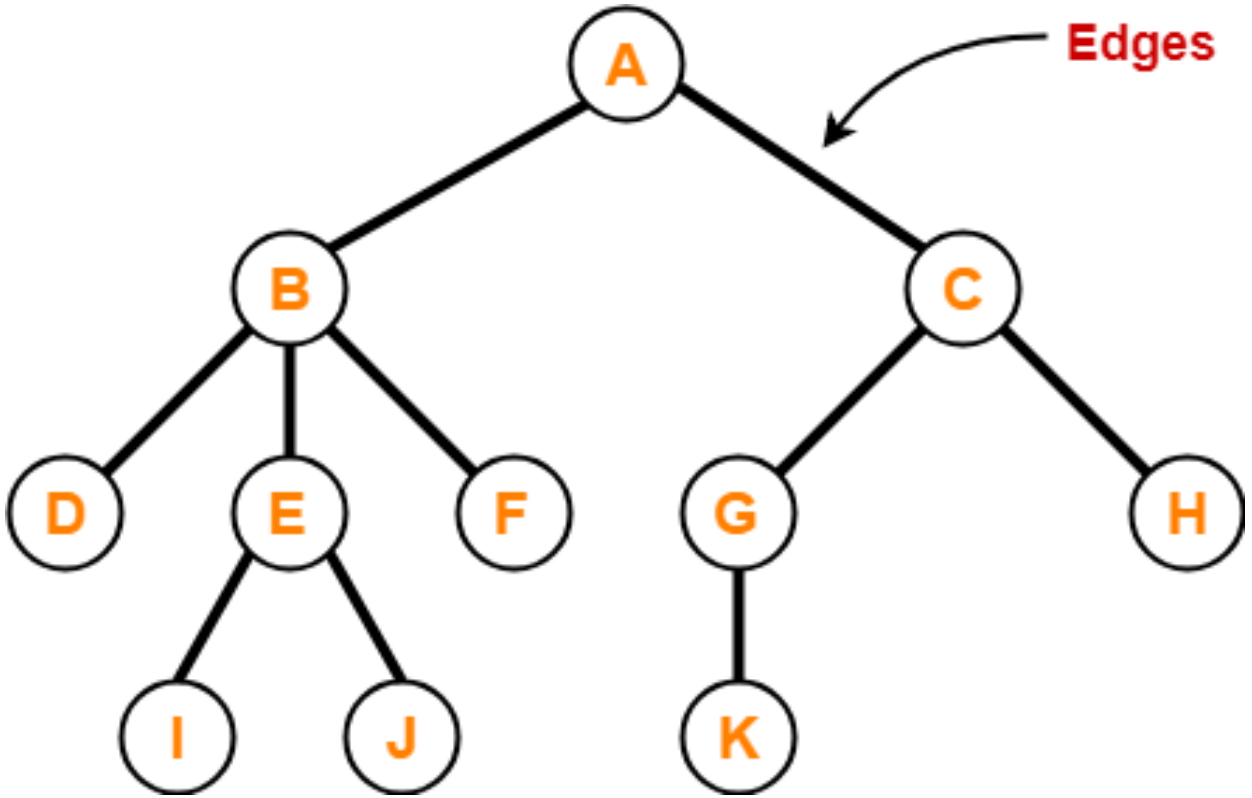
Tree terminology



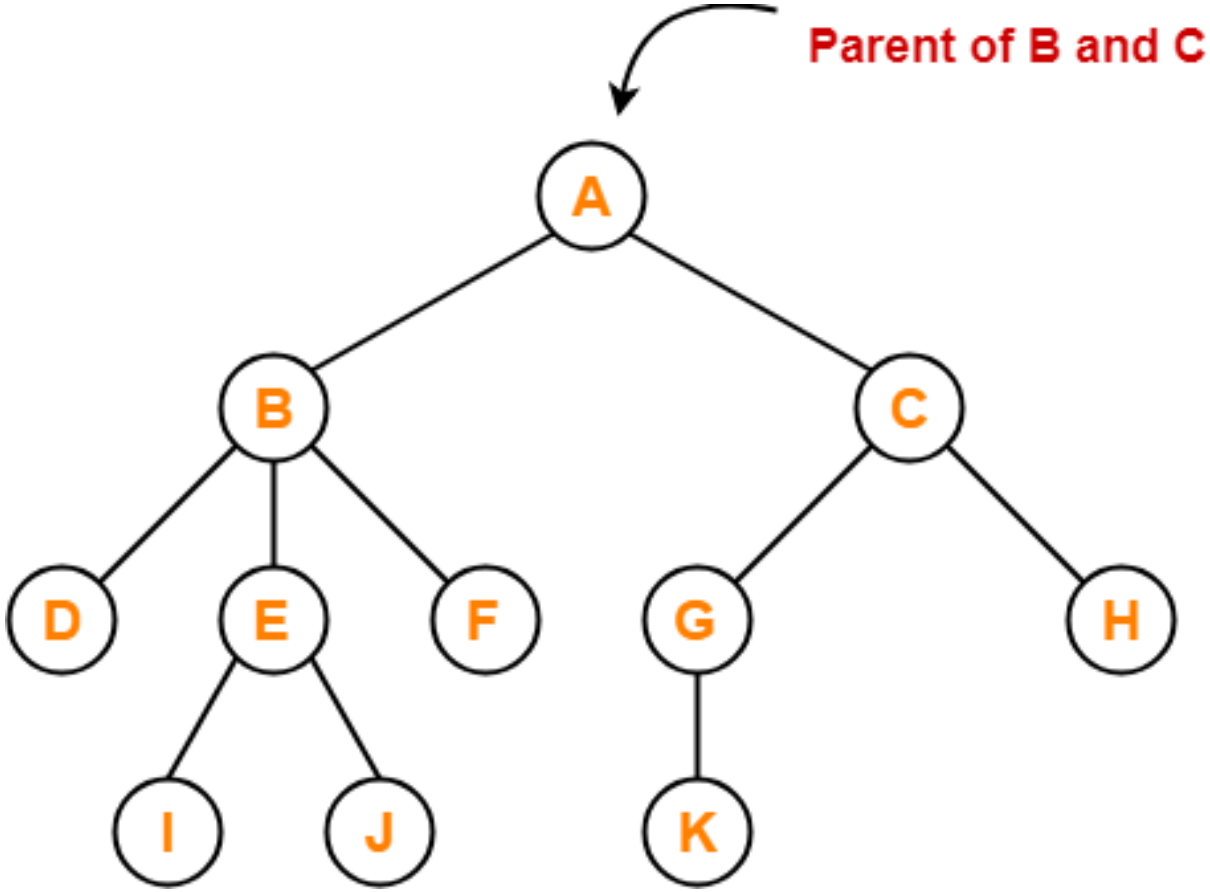
Tree Terminology



Tree Terminology

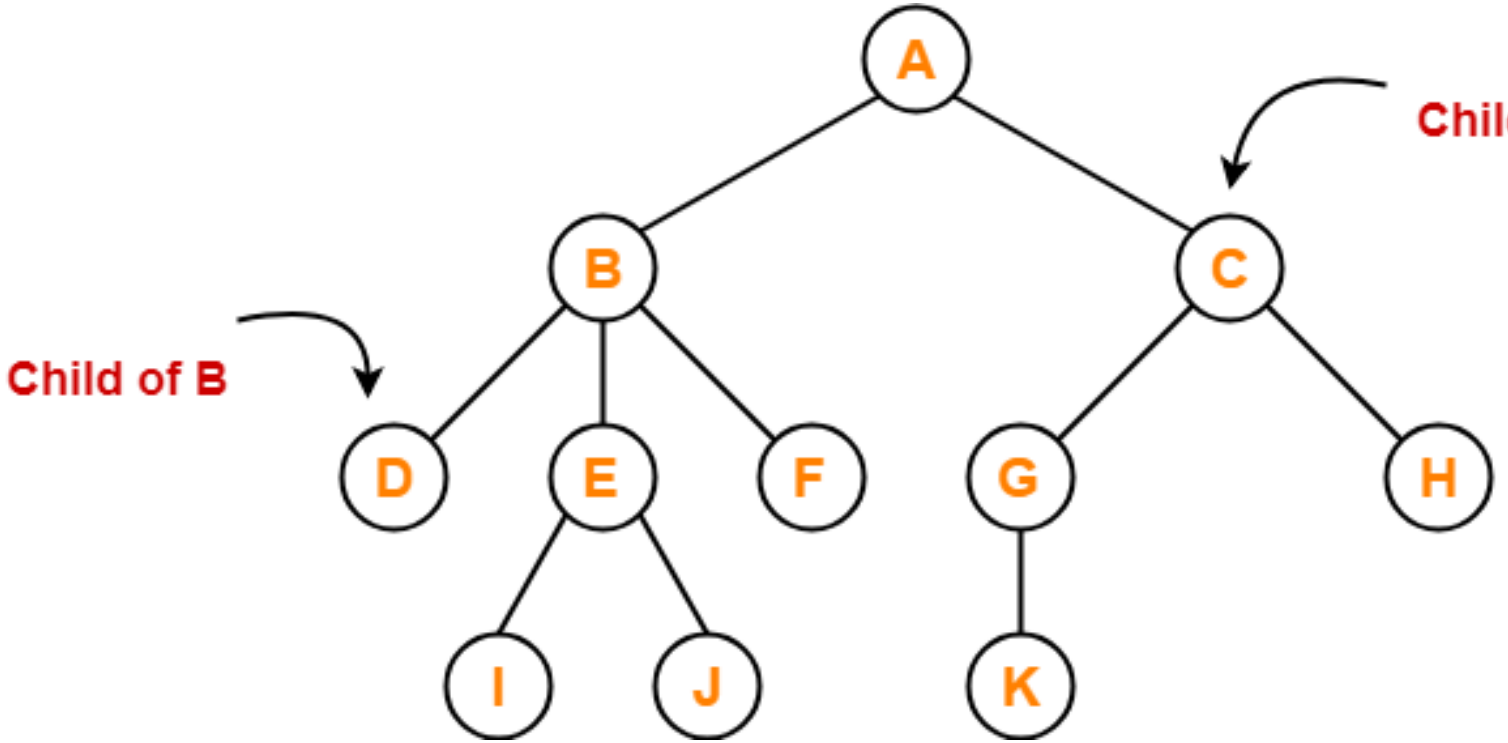


Tree Terminology



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

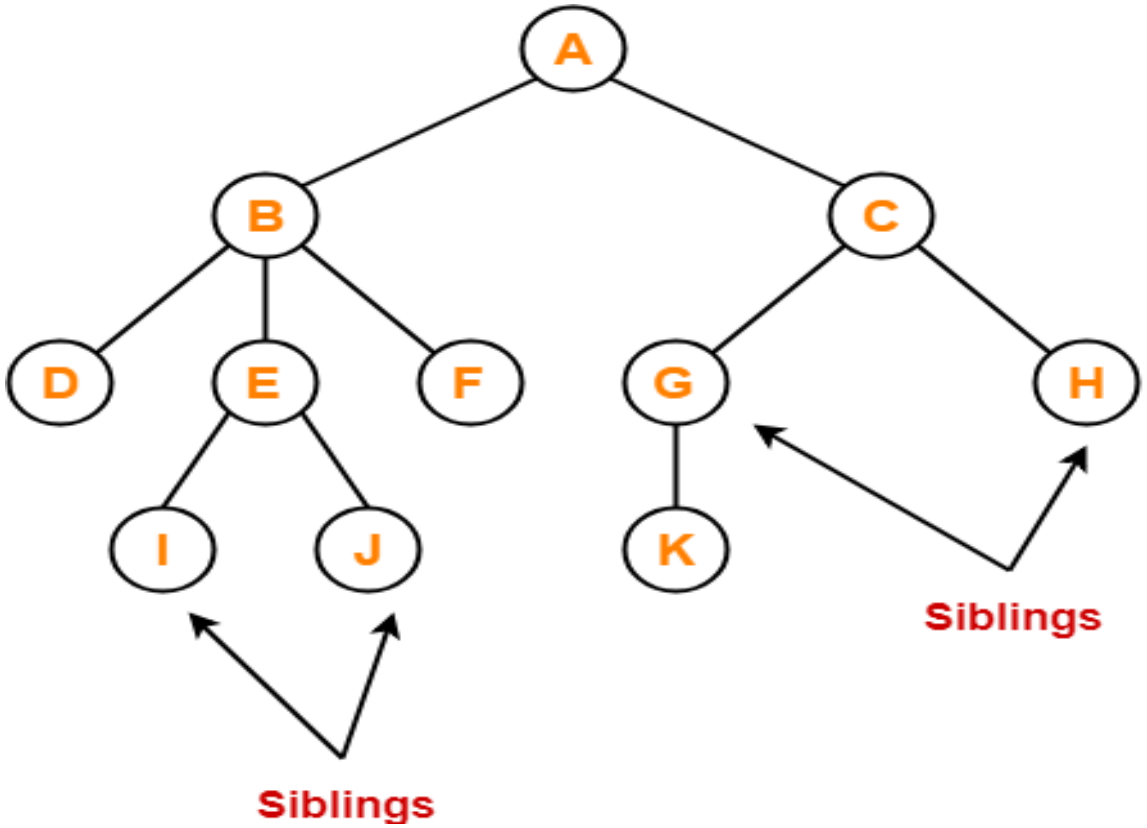
Tree Terminology



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

Tree Terminology

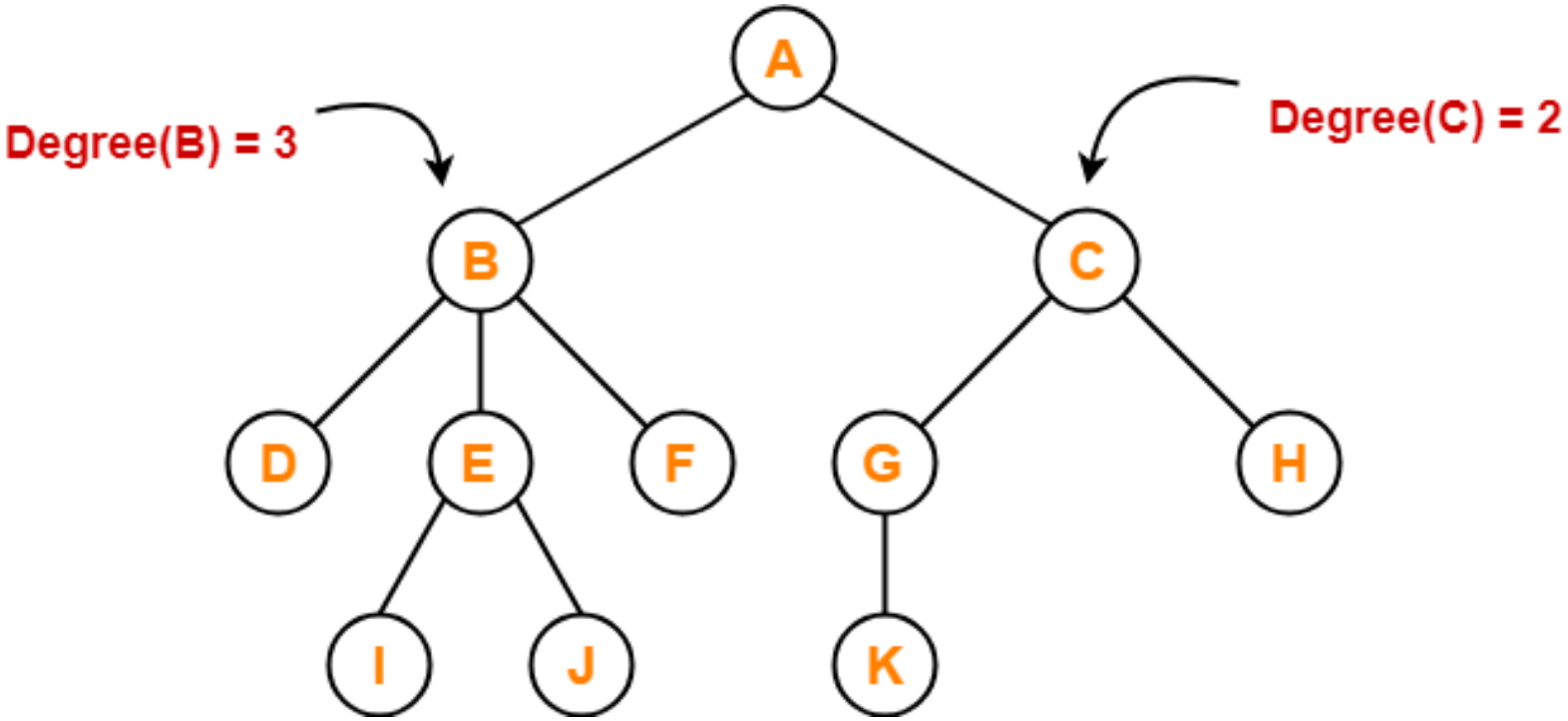
Nodes which belong to the same parent are called as **siblings**.



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

Tree Terminology

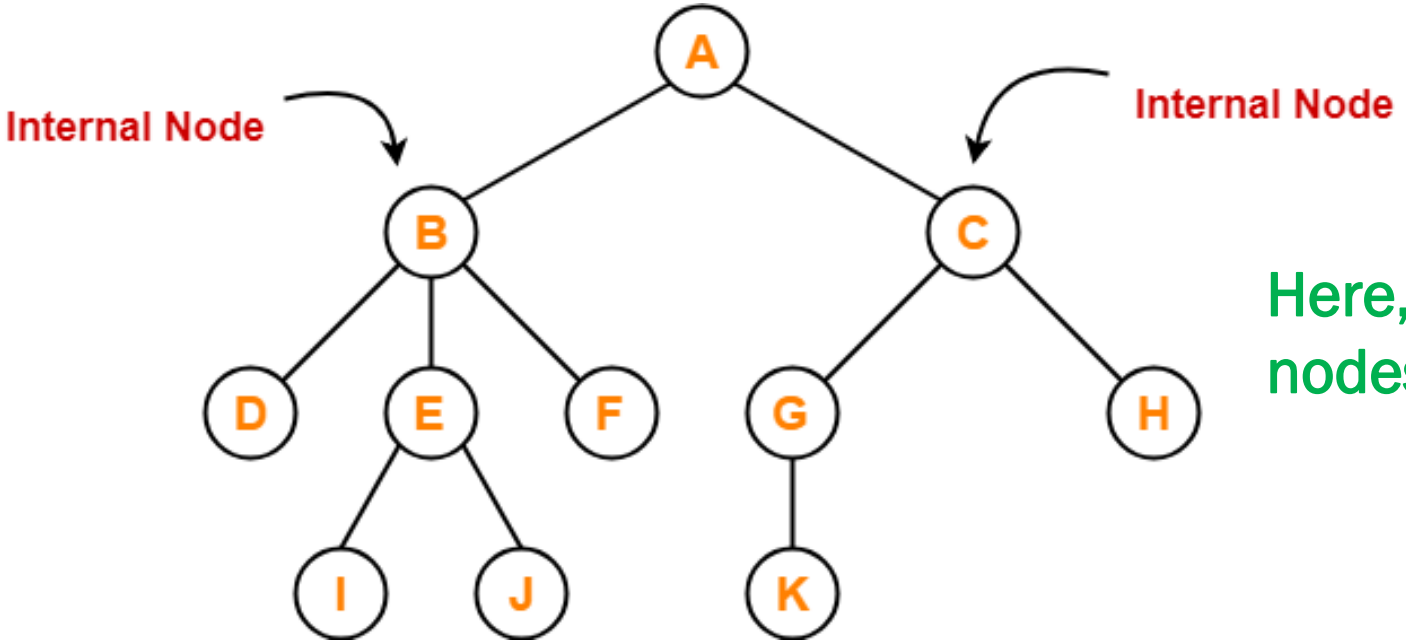
- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.



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

Tree Terminology

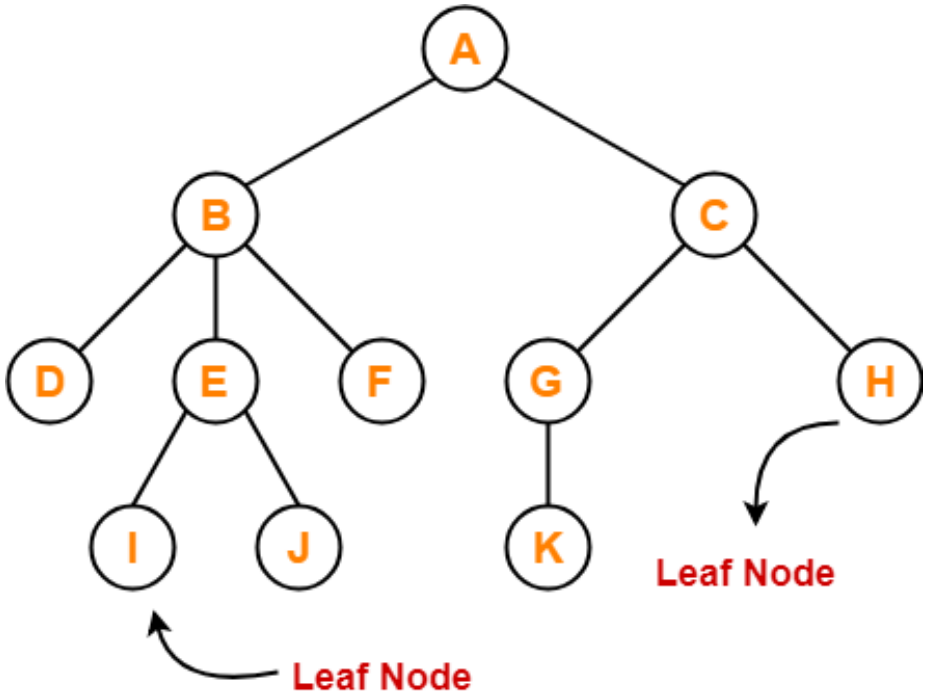
- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



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

Tree Terminology

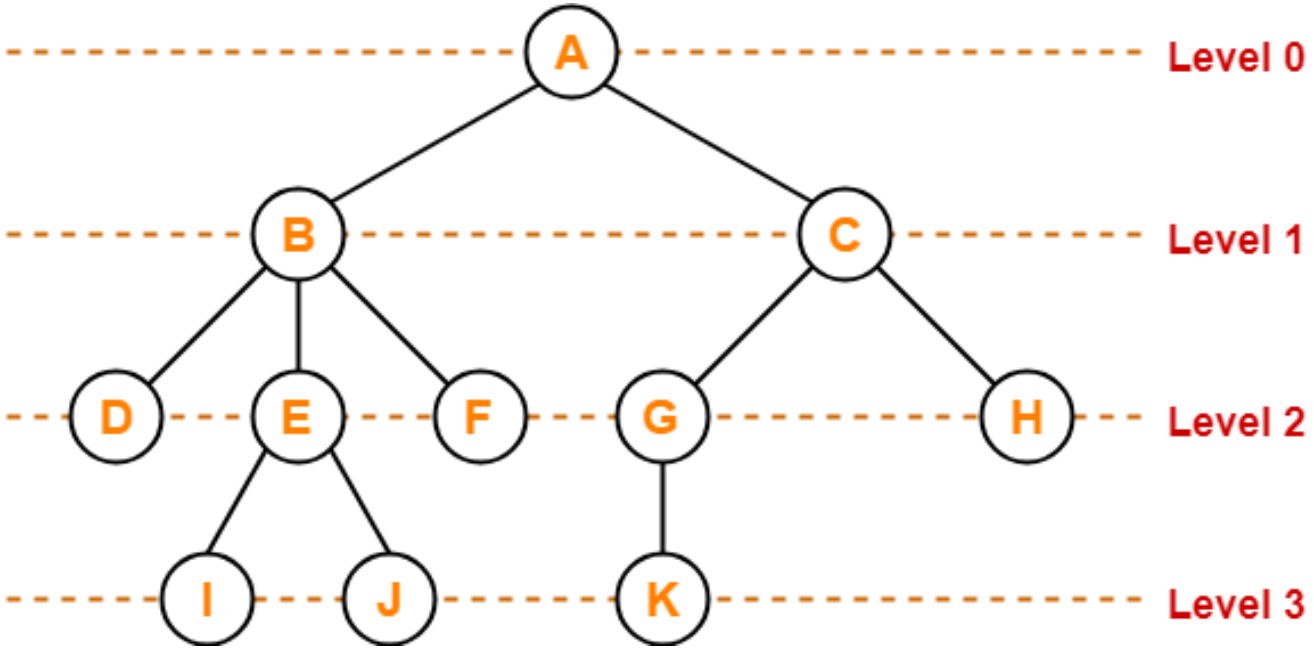
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



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

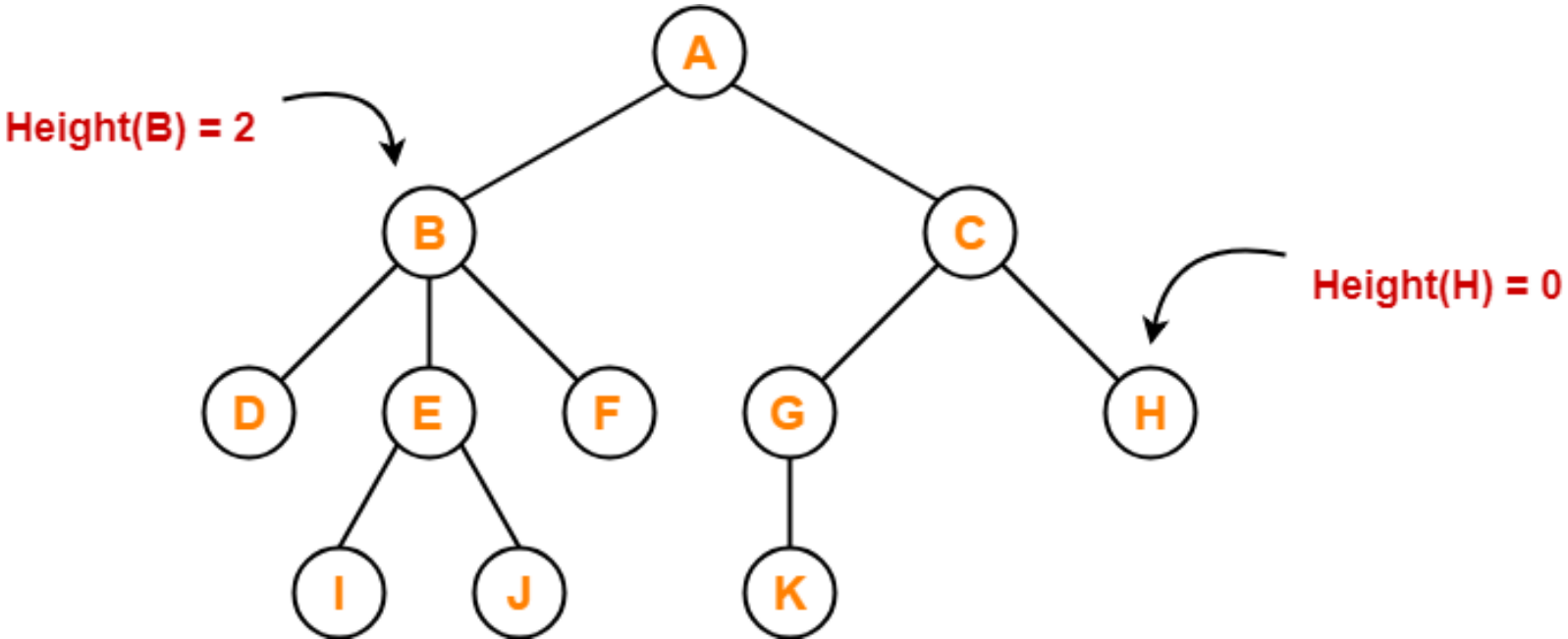
Tree Terminology

- 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.



Tree Terminology

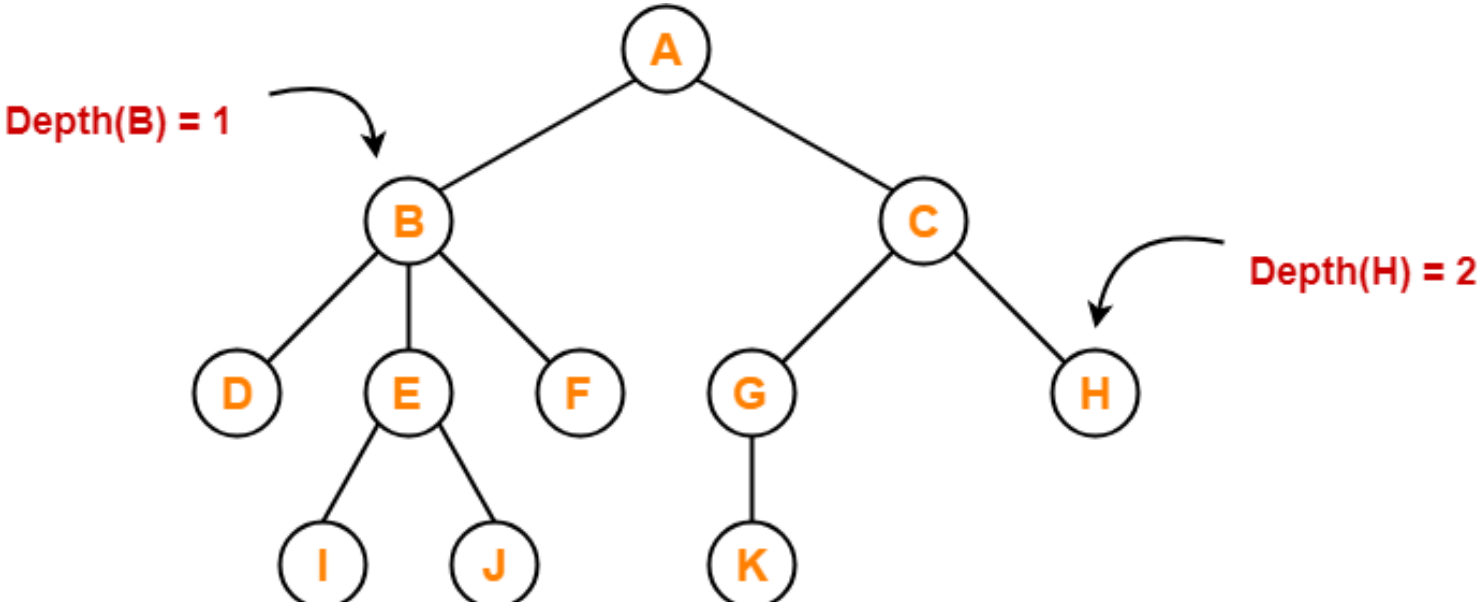
- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0



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

Tree Terminology

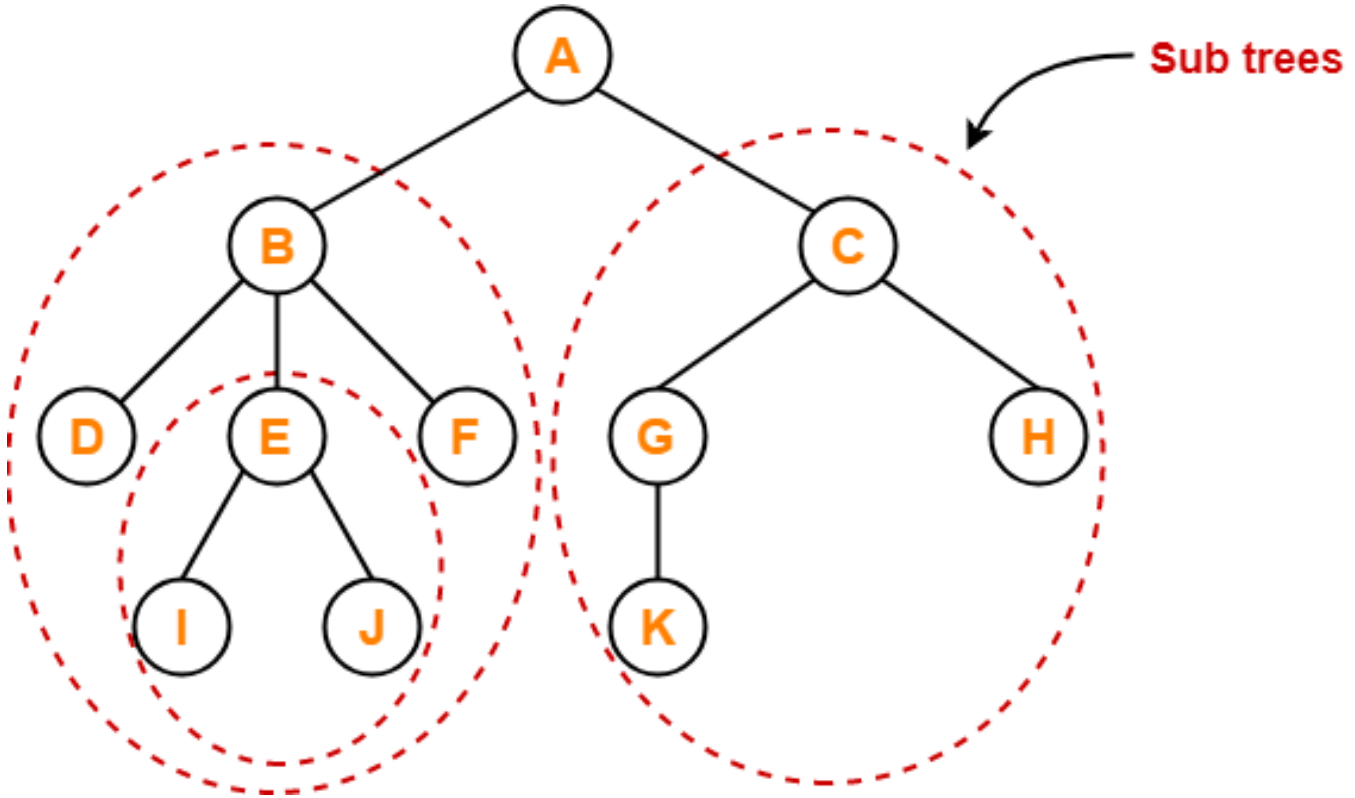
- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.



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

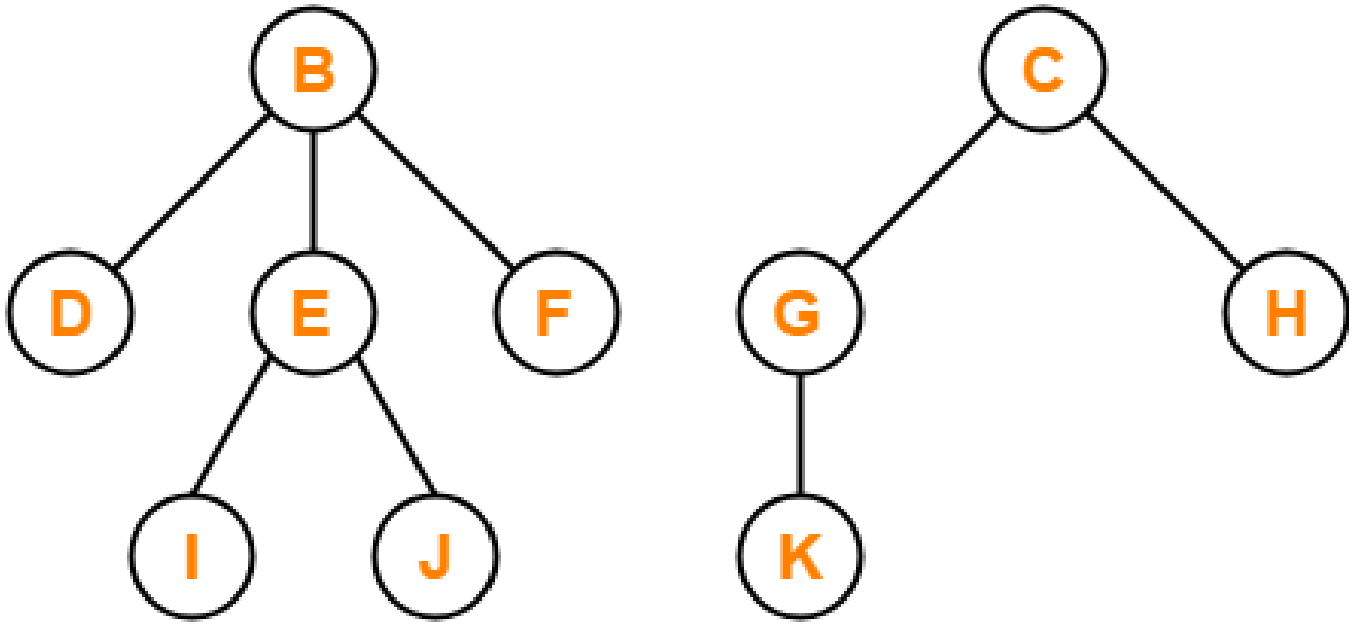
Tree Terminology

Sub Tree



Tree Terminology

A forest is a set of disjoint trees.



Forest

Applications

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

Types of tree

Binary Tree

Binary Search Tree

AVL Tree

B-Tree

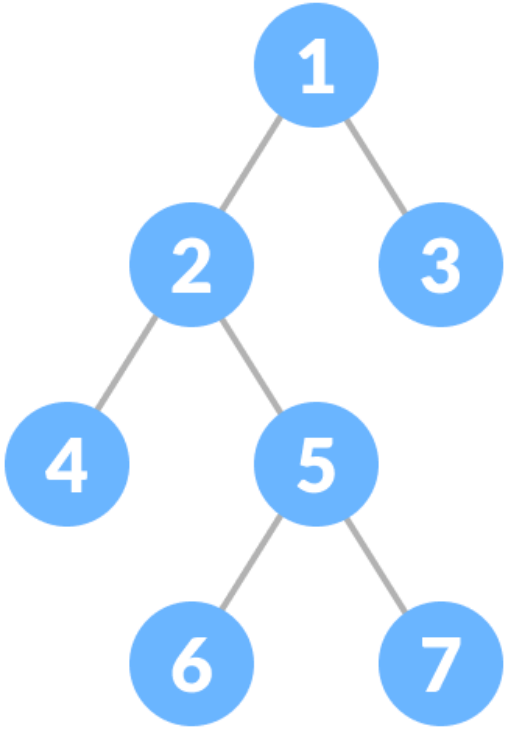
Binary Tree

A binary tree is a tree data structure in which each parent node can have at most two children.

Types of Binary Tree

Full Binary Tree

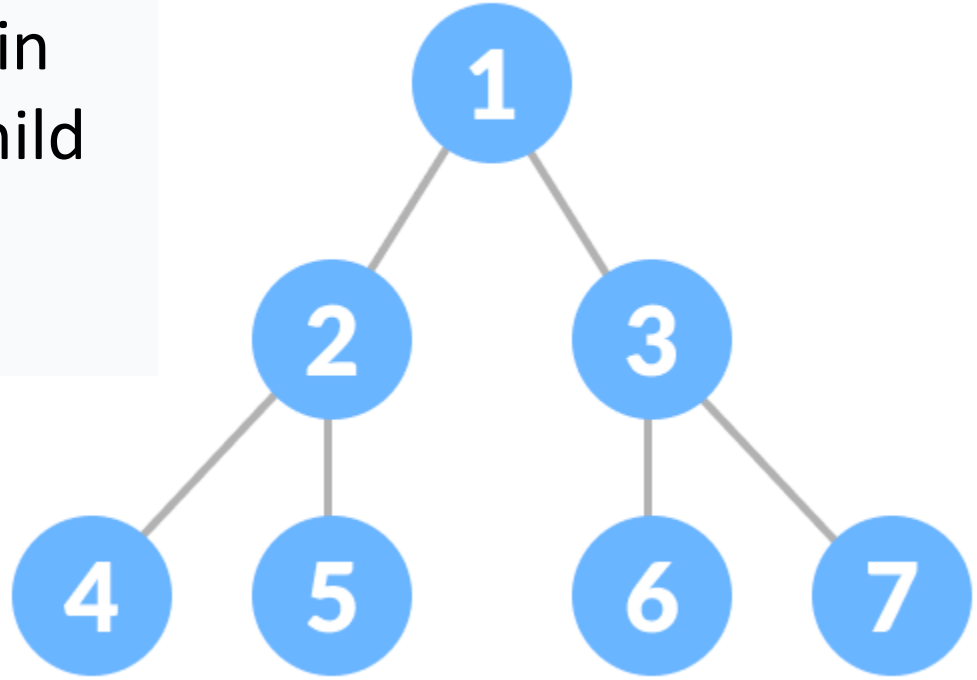
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Types of Binary Tree

2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Types of Binary Tree

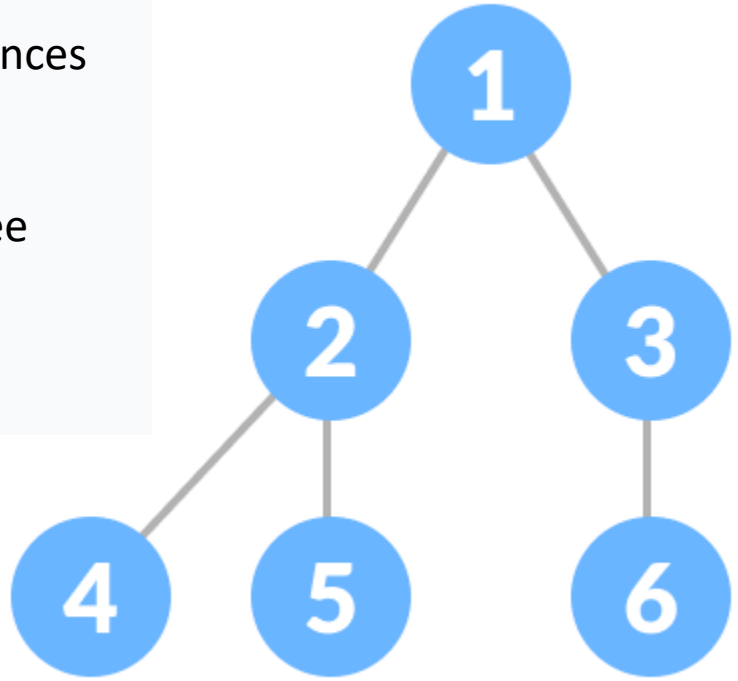
3. Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

1.Every level must be completely filled

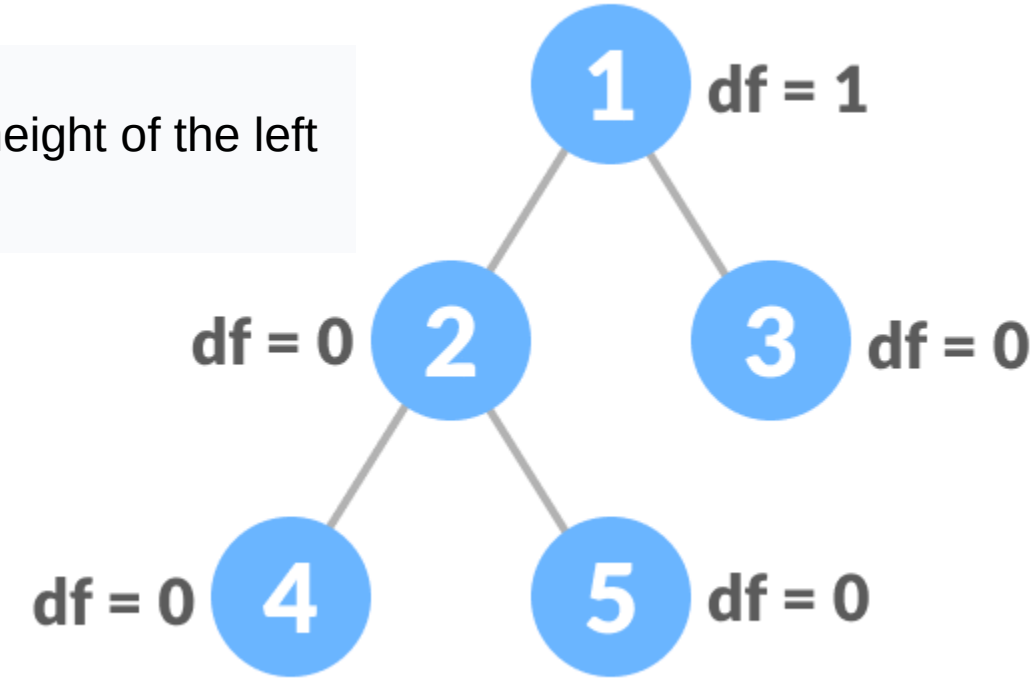
2.All the leaf elements must lean towards the left.

3.The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



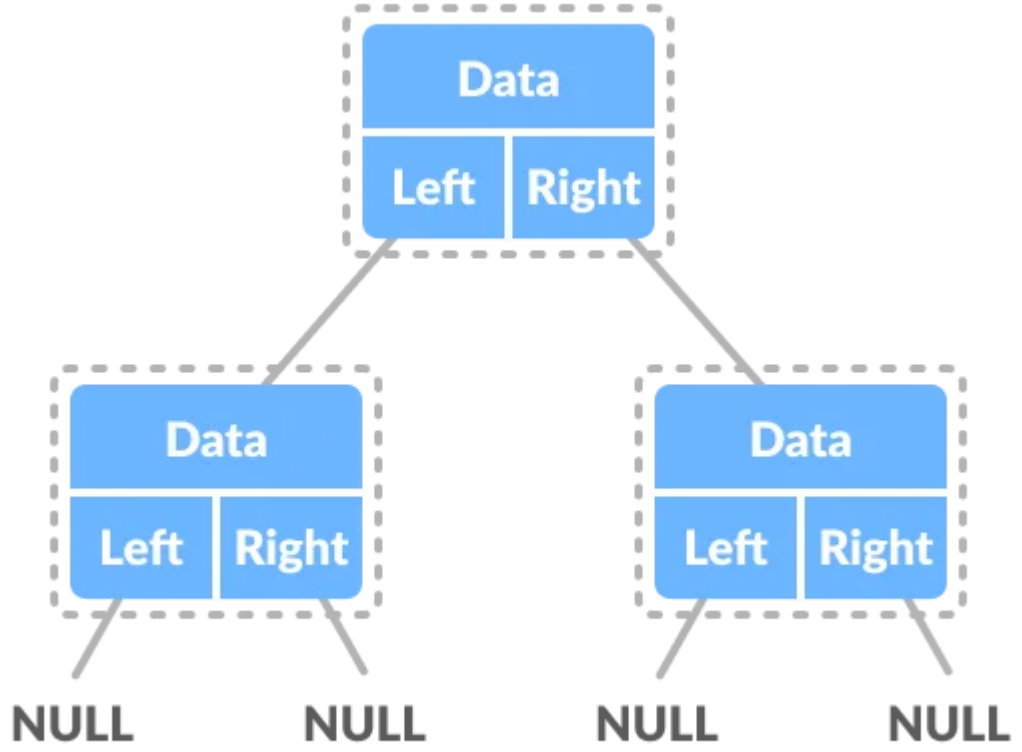
Types of Binary Tree

Balanced Binary Tree
It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Binary Tree Representation

```
struct Bnode {
        int data;
        struct  node *left;
        struct  node *right; };
```



```
// New node creation
struct Bnode *newNode(int data) {
          struct Bnode *node =new Bnode;
          node->data = data;
          node->left = NULL;
          node->right = NULL;
          return (node); }
```

```
int main() {
        struct node *root = newNode(1);
        root->left = newNode(2);
        root->right = newNode(3);
        root->left->left = newNode(4); }
```

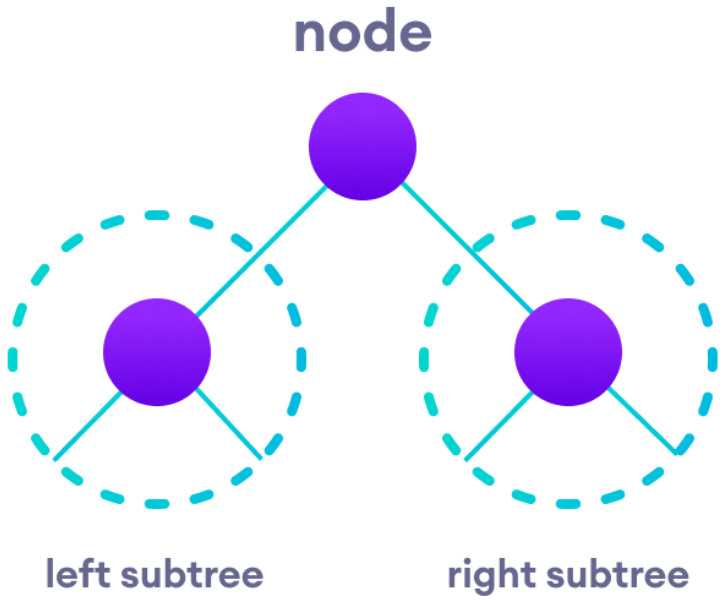
Tree Traversal

Tree traversal

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.

```
struct node {
    int data;
    struct node* left;
    struct node* right;
}
```



Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.

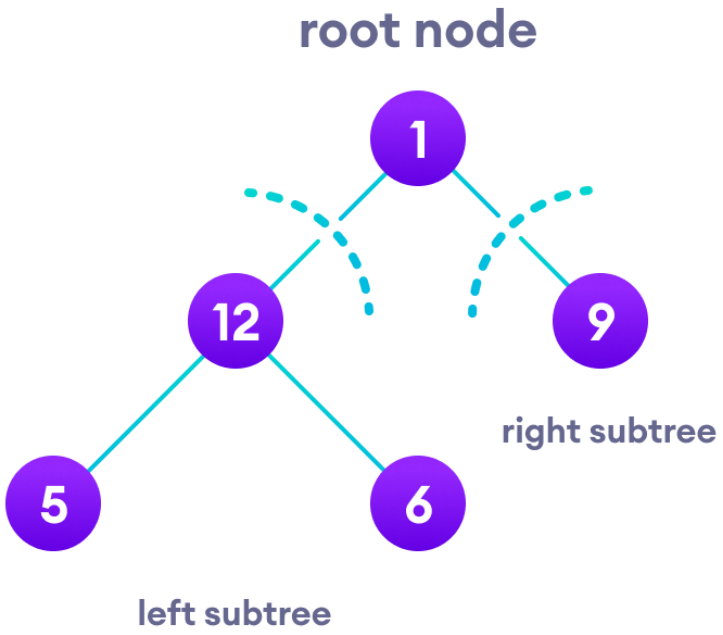
Depending on the order in which we do this, there can be three types of traversal

Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

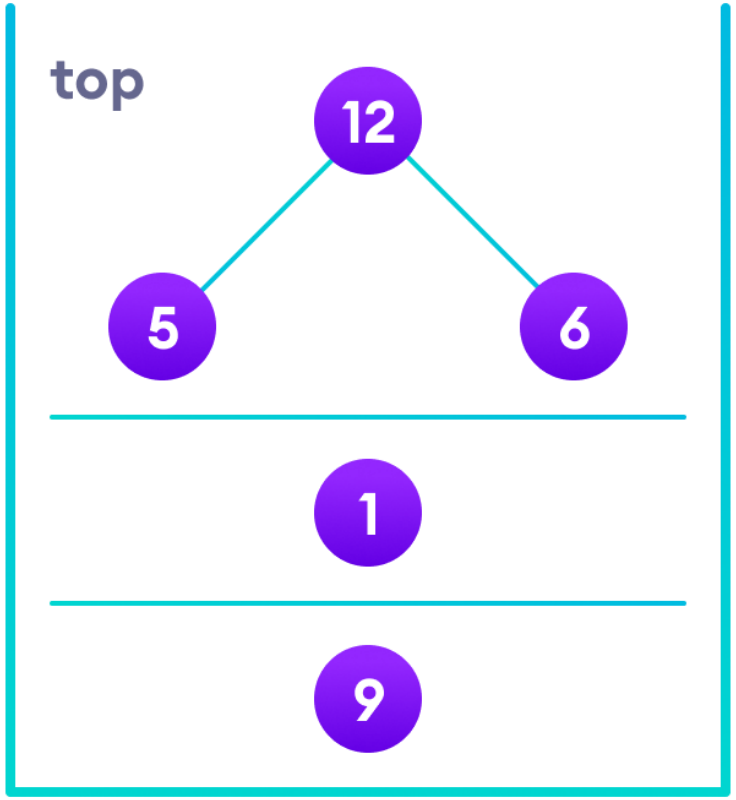
```
inorder(root->left)
display(root->data)
 inorder(root->right)
```

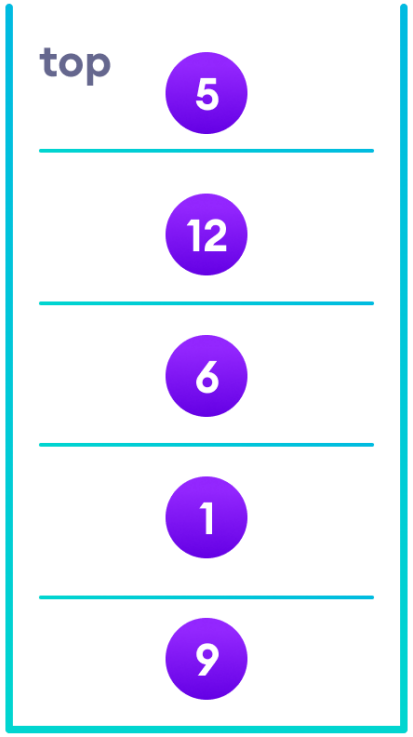
Inorder traversal



We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.

Let's put all this in a stack so that we remember.




top
5
12
6
1
9

Preorder traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

```
display(root->data)
preorder(root->left)
preorder(root->right)
```

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

Code for tree traversal

```
#include <iostream>
using namespace std;

struct Node {
  int data;
  struct Node *left, *right;
  Node(int data) {
    this->data = data;
    left = right = NULL;
  }
};

// Preorder traversal
void preorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  cout << node->data << "->";
  preorderTraversal(node->left);
  preorderTraversal(node->right);
}
```

Code for tree traversal

```
// Postorder traversal
void postorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  postorderTraversal(node->left);
  postorderTraversal(node->right);
  cout << node->data << "->";
}

// Inorder traversal
void inorderTraversal(struct Node* node) {
  if (node == NULL)
    return;

  inorderTraversal(node->left);
  cout << node->data << "->";
  inorderTraversal(node->right);
}
```

Code for tree traversal

```
int main() {
  struct Node* root = new Node(1);
  root->left = new Node(12);
  root->right = new Node(9);
  root->left->left = new Node(5);
  root->left->right = new Node(6);

  cout << "Inorder traversal ";
  inorderTraversal(root);

  cout << "\nPreorder traversal ";
  preorderTraversal(root);

  cout << "\nPostorder traversal ";
  postorderTraversal(root);
```

Credits and Acknowledgements

https://www.gatevidyalay.com

https://www.programiz.com/