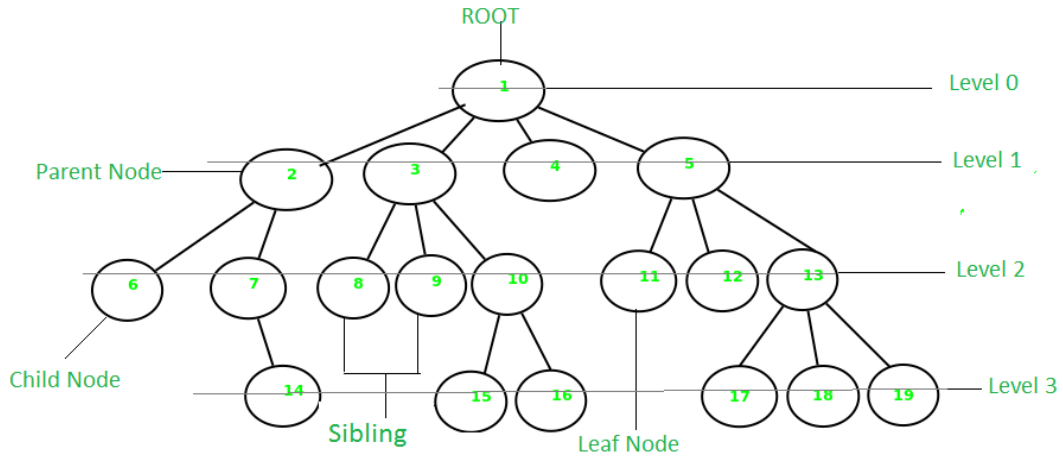
Trees:

Basic Tree Terminologies:

What is a Tree data structure?

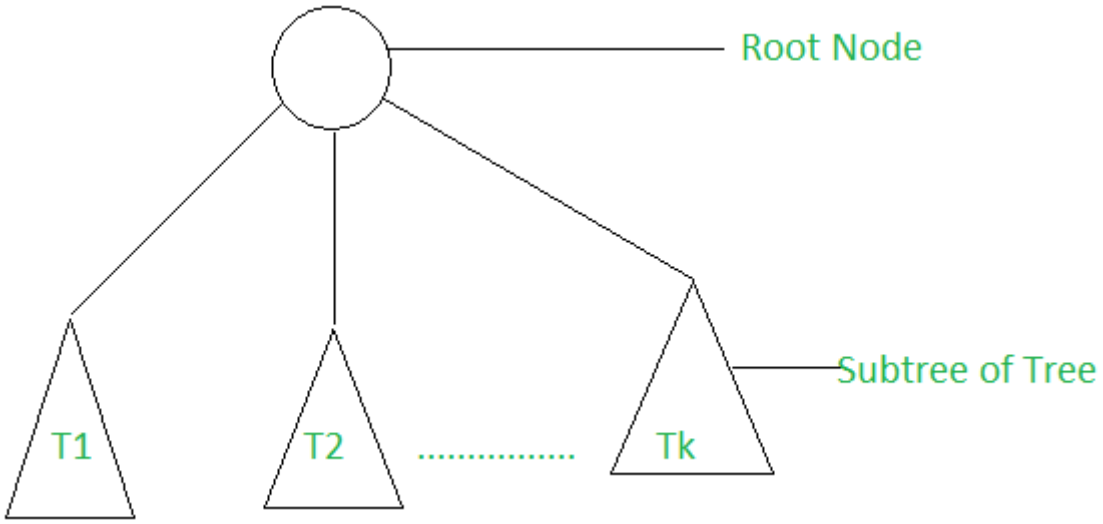
A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



Recursive Definition:

A tree consists of a root, and zero or more subtrees T1, T2, … , Tk such that there is an edge from the root of the tree to the root of each subtree.



Why Tree is considered a non-linear data structure?

The data in a tree are not stored in a sequential manner i.e, they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {**2}** is the parent node of {**6, 7}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {**6, 7}** are the child nodes of {**2}**.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {**1}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {**6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19}** are the leaf nodes of the tree.

- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {**1, 2}** are the ancestor nodes of the node {**7}**
- **Descendant:** Any successor node on the path from the leaf node to that node. {**7, 14}** are the descendants of the node. {**2}**.
- **Sibling:** Children of the same parent node are called siblings. {**8, 9, 10}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.

Properties of a Tree:

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.

- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

Some more properties are:

- Traversing in a tree is done by depth first search and breadth first search algorithm.
- It has no loop and no circuit
- It has no self-loop
- Its hierarchical model.

Syntax:

```
struct Node

{

   int data;

   struct Node *left_child;

   struct Node *right_child;

};
```

Basic Operation Of Tree:

Create *– create a tree in data structure.*

Insert *− Inserts data in a tree.*

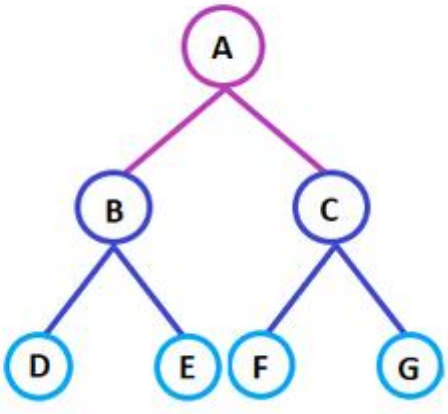
Search *− Searches specific data in a tree to check it is present or not.*

Preorder Traversal *– perform Traveling a tree in a pre-order manner in data structure .*

In order Traversal *– perform Traveling a tree in an in-order manner.*

Post order Traversal *–perform Traveling a tree in a post-order manner.*

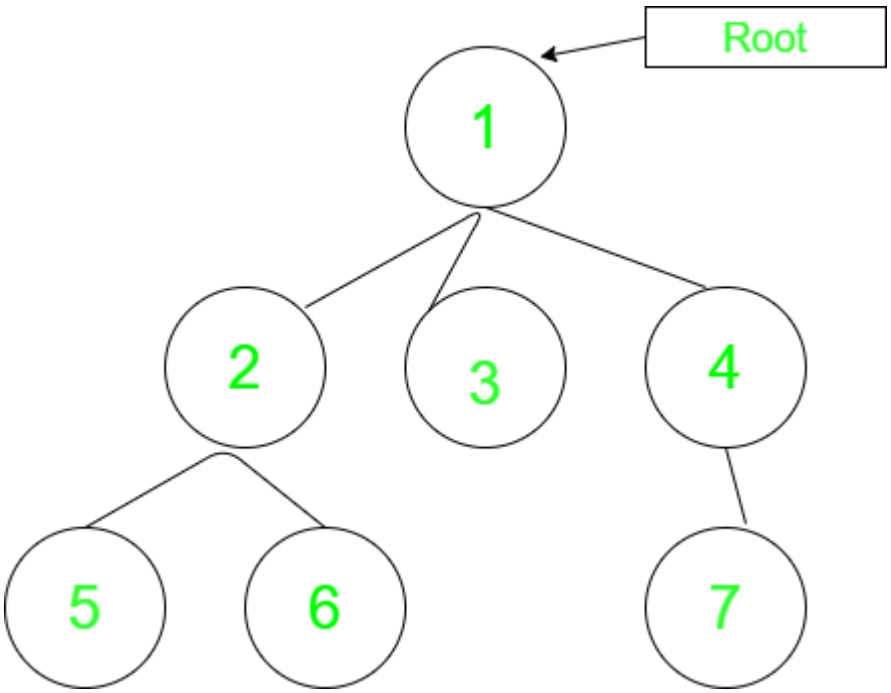
Example of Tree data structure



Here,
Node A is the root node
B is the parent of D and E
D and E are the siblings

D, E, F and G are the leaf nodes
A and B are the ancestors of E

Few examples on Tree Data Structure: A code to demonstrate few of the above terminologies has been described below:



```
// C++ program to demonstrate some of the above
// terminologies
#include <bits/stdc++.h>
using namespace std;
// Function to add an edge between vertices x and y
void addEdge(int x, int y, vector<vector<int> >& adj)
{
    adj[x].push_back(y);
    adj[y].push_back(x);
}
// Function to print the parent of each node
void printParents(int node, vector<vector<int> >& adj,
                  int parent)
{
    // current node is Root, thus, has no parent
    if (parent == 0)
        cout << node << "->Root" << endl;
    else
        cout << node << "->" << parent << endl;
    // Using DFS
```

```
    for (auto cur : adj[node])
        if (cur != parent)
            printParents(cur, adj, node);
}
// Function to print the children of each node
void printChildren(int Root, vector<vector<int> >& adj)
{
    // Queue for the BFS
    queue<int> q;
    // pushing the root
    q.push(Root);
    // visit array to keep track of nodes that have been
    // visited
    int vis[adj.size()] = { 0 };
    // BFS
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        vis[node] = 1;
        cout << node << "-> ";
```

```
        for (auto cur : adj[node])
            if (vis[cur] == 0) {
                cout << cur << " ";
                q.push(cur);
            }
        cout << endl;
    }
}
// Function to print the leaf nodes
void printLeafNodes(int Root, vector<vector<int> >& adj)
{
    // Leaf nodes have only one edge and are not the root
    for (int i = 1; i < adj.size(); i++)
        if (adj[i].size() == 1 && i != Root)
            cout << i << " ";
    cout << endl;
}
// Function to print the degrees of each node
void printDegrees(int Root, vector<vector<int> >& adj)
{
```

```
    for (int i = 1; i < adj.size(); i++) {
        cout << i << ": ";
        // Root has no parent, thus, its degree is equal to
        // the edges it is connected to
        if (i == Root)
            cout << adj[i].size() << endl;
        else
            cout << adj[i].size() - 1 << endl;
    }
}
// Driver code
int main()
{
    // Number of nodes
    int N = 7, Root = 1;
    // Adjacency list to store the tree
    vector<vector<int> > adj(N + 1, vector<int>());
    // Creating the tree
    addEdge(1, 2, adj);
    addEdge(1, 3, adj);
```

```
addEdge(1, 4, adj);
addEdge(2, 5, adj);
addEdge(2, 6, adj);
addEdge(4, 7, adj);
// Printing the parents of each node
cout << "The parents of each node are:" << endl;
printParents(Root, adj, 0);

// Printing the children of each node
cout << "The children of each node are:" << endl;
printChildren(Root, adj);

// Printing the leaf nodes in the tree
cout << "The leaf nodes of the tree are:" << endl;
printLeafNodes(Root, adj);

// Printing the degrees of each node
cout << "The degrees of each node are:" << endl;
printDegrees(Root, adj);
```

```
    return 0;
}
```

Output

The parents of each node are:

1->Root

2->1

5->2

6->2

3->1

4->1

7->4

The children of each node are:

1-> 2 3 4

2-> 5 6

3->

4-> 7

5->

6->

7->

The leaf nodes of the tree are:

3 5 6 7

The degrees of each node are:

1: 3