SNS COLLEGE OF TECHNOLOGY
(An Autonomous Institution)
COIMBATORE- 641 035

Department of Computer Science and Engineering

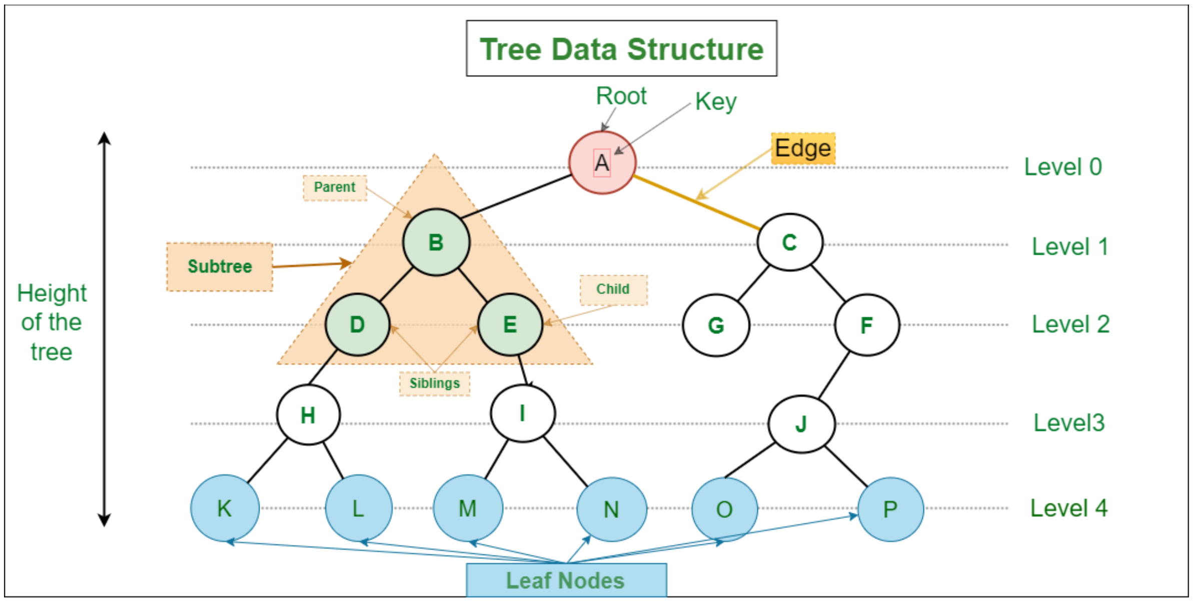
23ITT101 - PROGRAMMING IN C AND DATA STRUCTURES

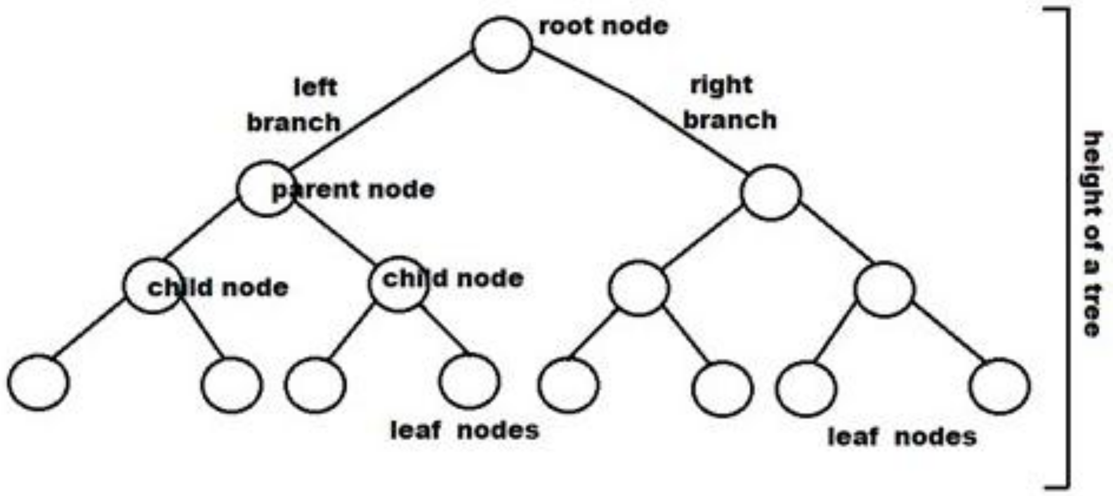
UNIT V TREES

Introduction to Trees

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.
The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

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.



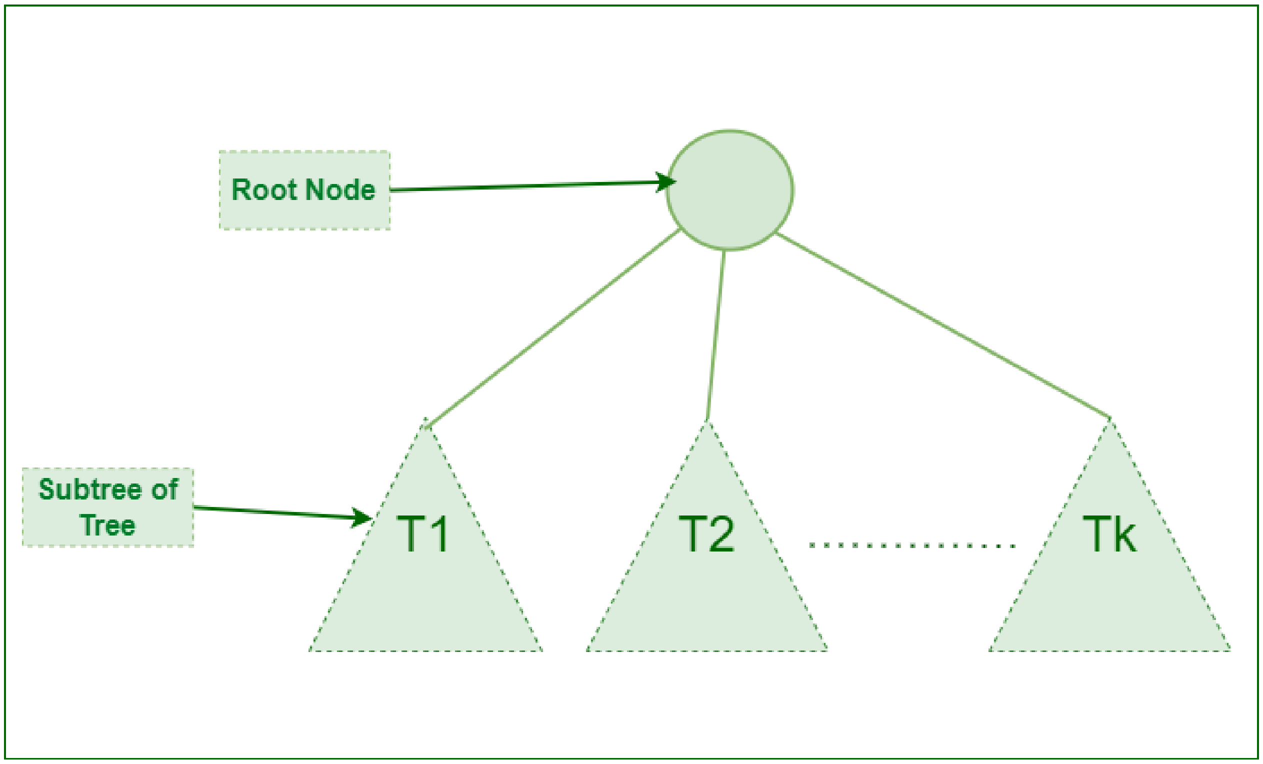


Basic Terminologies In Tree Data Structure:

- Parent Node: The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- Child Node: The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- Root Node: The topmost node of a tree or the node which does not have any parent node is called the root node. {A} 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. {K, L, M, N, O, P} 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. {A,B} are the ancestor nodes of the node {E}
- Descendant: Any successor node on the path from the leaf node to that node. {E,I} are the descendants of the node {B}.
- Sibling: Children of the same parent node are called siblings. {D,E} 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.

Representation of Tree Data Structure:

A tree consists of a root, and zero or more subtrees T_1, T_2, … , T_k such that there is an edge from the root of the tree to the root of each subtree.

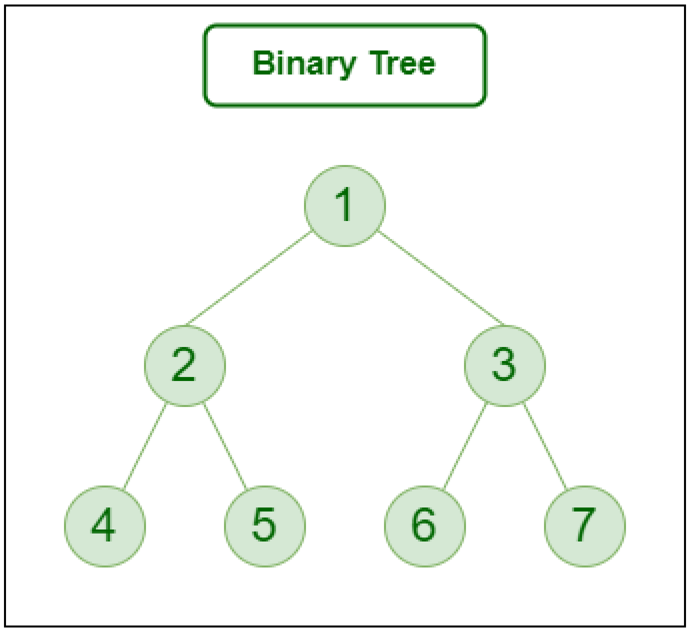


Representation of Tree Data Structure

Representation of a Node in Tree Data Structure:

```
struct Node
{
  int data;
  struct Node *first_child;
  struct Node *second_child;
  struct Node *third_child;
  .
  .
  .
  struct Node *nth_child;
};
```

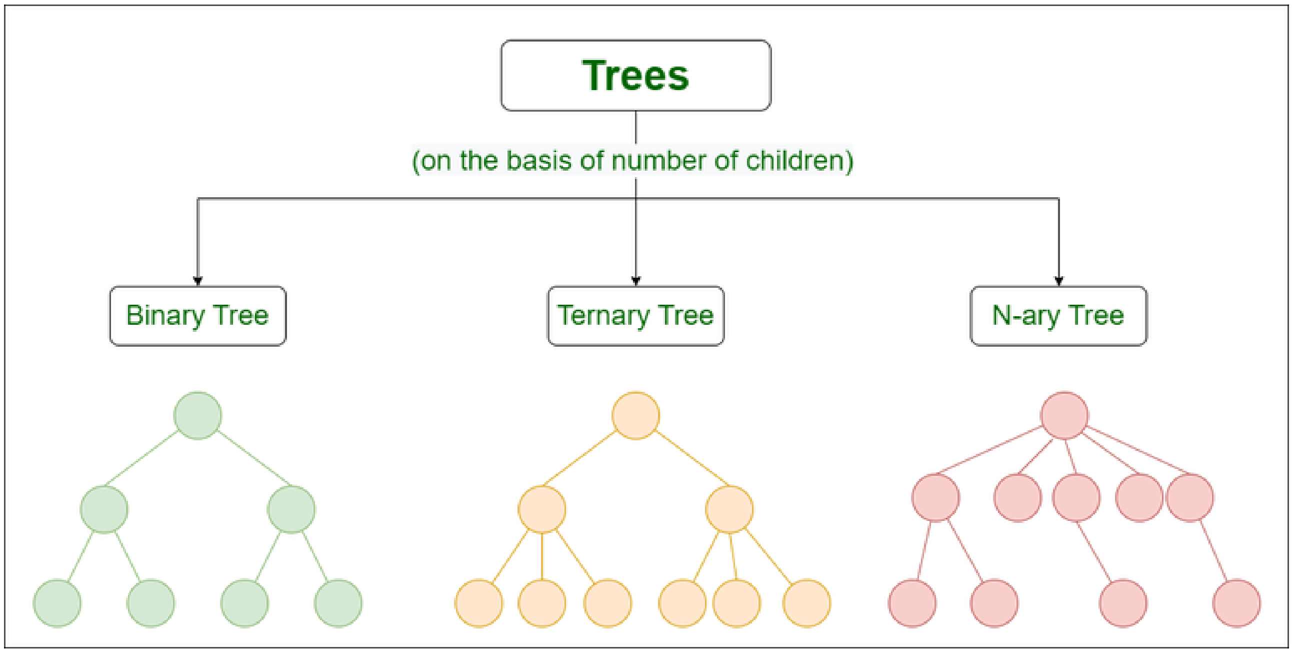
Example of Tree data structure



Here,

- Node 1 is the root node
- 1 is the parent of 2 and 3
- 2 and 3 are the siblings
- 4, 5, 6, and 7 are the leaf nodes
- 1 and 2 are the ancestors of 5

Types of Tree data structures:



- Binary tree: In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
- Ternary Tree: A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as "left", "mid" and "right".
- N-ary Tree or Generic Tree: Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

Basic Operation Of Tree Data Structure:

- Create – create a tree in the data structure.
- Insert − Inserts data in a tree.
- Search − Searches specific data in a tree to check whether it is present or not.
- Traversal:
    - Preorder Traversal – perform Traveling a tree in a pre-order manner in the 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.

```
// 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

2: 2

3: 0

4: 1

5: 0

6: 0

7: 0
```

Why Tree is considered a non-linear data structure?

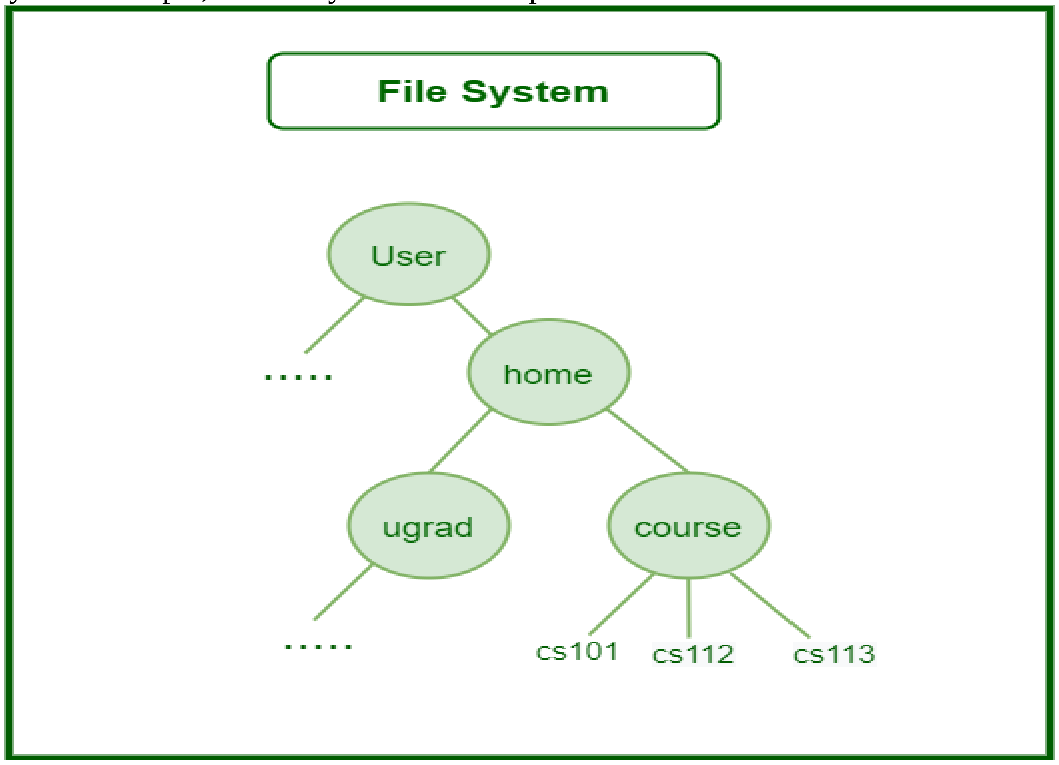
The data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

Properties of Tree Data Structure:

- 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.

Need for Tree Data Structure

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:



File System

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).

3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Application of Tree Data Structure:

- File System: This allows for efficient navigation and organization of files.
- Data Compression: Huffman coding is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
- Compiler Design: In compiler design, a syntax tree is used to represent the structure of a program.
- Database Indexing: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.

Advantages of Tree Data Structure:

- Tree offer Efficient Searching Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.
- Trees provide a hierarchical representation of data, making it easy to organize and navigate large amounts of information.
- The recursive nature of trees makes them easy to traverse and manipulate using recursive algorithms.

Disadvantages of Tree Data Structure:

- Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to inefficient search times.
- Trees demand more memory space requirements than some other data structures like arrays and linked lists, especially if the tree is very large.
- The implementation and manipulation of trees can be complex and require a good understanding of the algorithms.