Trees

Introduction

There are various systems around us which are hierarchical in nature; military, government organisations, corporations, they all have a hierarchical form of command chain. There are various systems around us which are hierarchical in nature; military, government organisations. Consider a company with a president and some number of vice presidents who report to the president. Each vice president has some number of direct subordinates, those subordinates further have people reporting to them and so on. If we wanted to model this company with a data structure, it would be natural to think of the president as the head of all, the vice presidents at level 1, and their subordinates at lower levels as we go down the organizational hierarchy.

The examples of hierarchical models that we discussed above cannot be represented/stored using a linear data structure. For this very scenario, a data structure called a tree comes to our rescue. Now there are various types of tree, depending on the rule/property they follow. The simplest variant of tree is called a **general tree** (or N-ary tree). As in above example, the number of vice presidents is likely to be more than zero, we need to use a data structure that represents the data in the form of a hierarchy.

In this module we will examine general tree terminology and define a basic ADT for general trees.

Trees are non-linear hierarchical data structures. It is a collection of nodes connected to each other by means of “edges” which are either directed or undirected. One of the nodes is designated as “Root node” and the remaining nodes are called child nodes or the leaf nodes(nodes with no chid nodes).

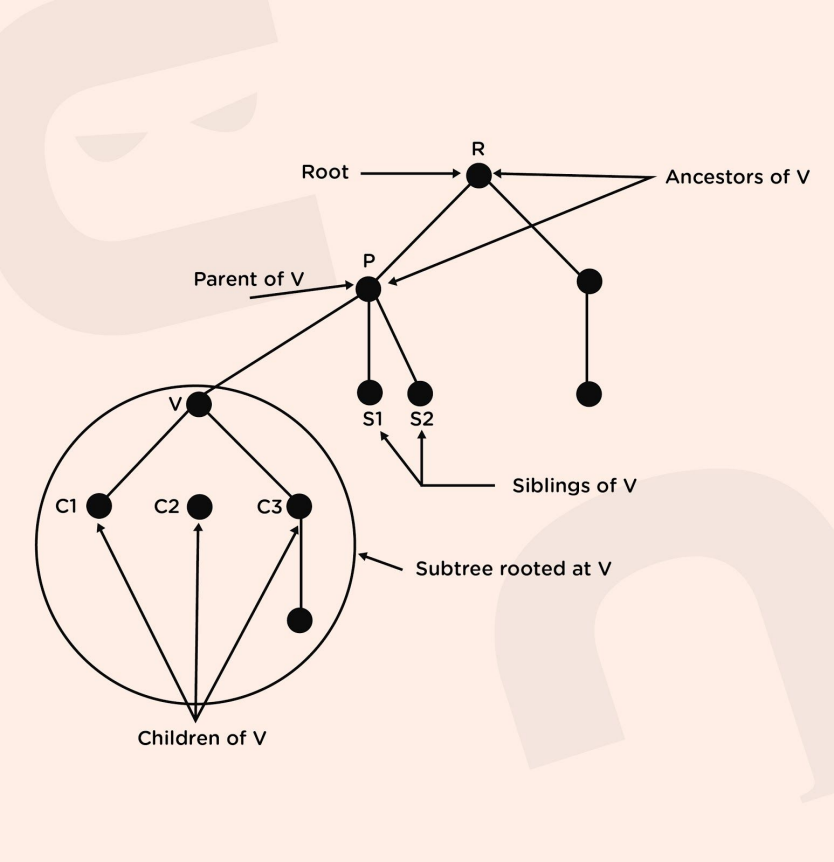
In general, each node can have as many children but only one parent node.

More formally, a tree T is a finite set of one or more nodes such that there is one designated node R, called the root of the tree. If the set (T - {R}) is not empty, these nodes are partitioned into n > 0 disjoint sets T_0, T_1, T_2,..., T_{n-1}, each of which is a tree, and whose roots R_1, R_2,..., R_n, respectively, are children of R. The subsets T_i (($0 \leq i < n$) are said to be ***subtrees*** of T. These subtrees, as ordered in that T_i set, are said to come before T_j if $i < j$. By convention, the subtrees are arranged from left to right with subtree T_0 called the leftmost child of R.

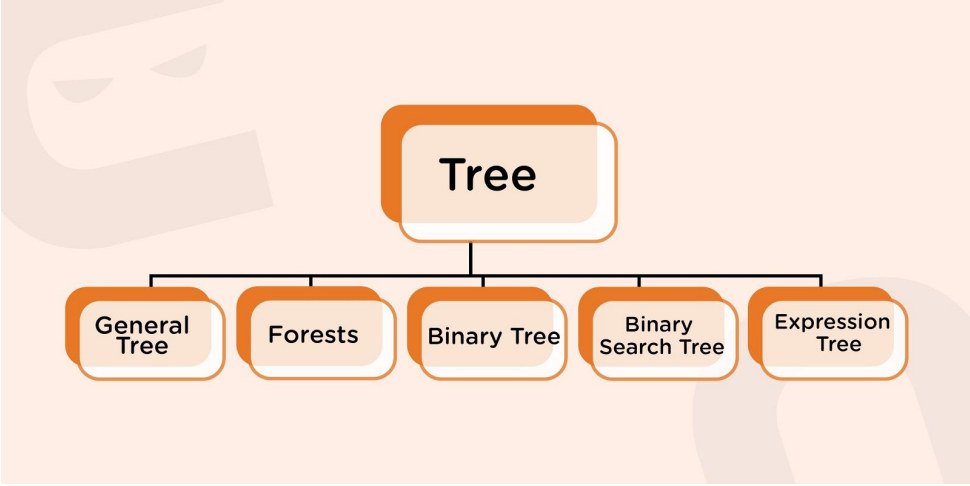
Let us go through some basic terminology associated with trees:

- **Root node:** This is the topmost node in the tree hierarchy.
- **Leaf node:** These are the bottommost nodes in a tree hierarchy. The leaf nodes do not have any child nodes. They are also known as external nodes.
- **Subtree:** Subtree represents various descendants of a node when the root is not null. A tree usually consists of a root node and one or more subtrees.
  Parent node: Any node except the root node that has a child node and an edge upward towards the parent.
- **Ancestor Node:** It is any predecessor node on a path from the root to that node. Note that the root does not have any ancestors.
- **Key:** It represents the data stored in a node.
- **Level:** Represents the generation of a node. A root node is always at level 1. Child nodes of the root are at level 2, grandchildren of the root are at level 3 and so on. In general, each node is at a level higher than its parent.
- **Path:** The path is a sequence of consecutive edges.
- **Degree:** Degree of a node indicates the number of children that a node has.

Below is provided the pictorial representation of above:



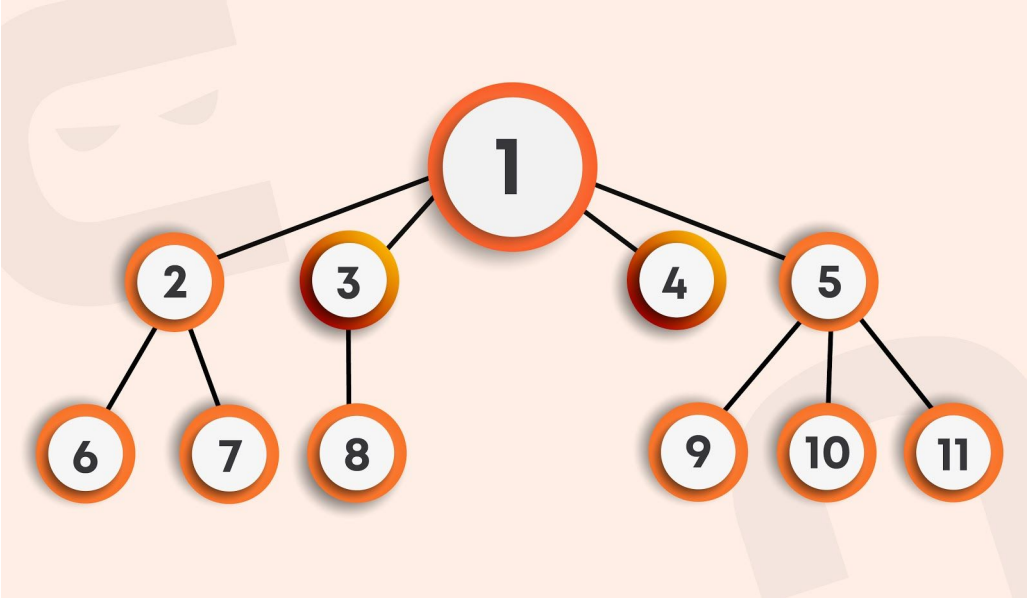
Types of Trees



In this course, we will discuss only General trees, Binary trees and Binary Search trees.

Tree node class

Before discussing general tree implementations, we should first make precise what operations such implementations must support. Firstly, any implementation must be able to initialize a tree. Given a tree, we need access to the root of that tree. There must be some way to access the children of a node. In the case of the ADT for binary tree nodes, this was done by providing member functions that give explicit access to the left and right child pointers. Unfortunately, because we do not know in advance how many children a given node will have in the general tree, we cannot give explicit functions to access each child. An alternative must be found that works for an unknown number of children.



One choice would be to provide a function that takes as its parameter the index for the desired child. That combined with a function that returns the number of children for a given node would support the ability to access any node or process all children of a node. Unfortunately, this view of access tends to bias the choice for

node implementations in favor of an array-based approach, because these functions favor random access to a list of children. In practice, an implementation based on a linked list is often preferred.

An alternative is to provide access to the first (or leftmost) child of a node, and to provide access to the next (or right) sibling of a node.

Here are the class declarations for general trees and their nodes. Based on these two access functions, the children of a node can be traversed like a list. Trying to find the next sibling of the rightmost sibling would return null.

Kindly refer below for code- (we will name this file as **TreeNode.h** to use it further in our program)

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

template <typename T>
class TreeNode {
        public:
        T data;                                 // To store data
        vector<TreeNode<T>*> children;          // To store children for each node

        TreeNode(T data) {                      // Constructor to initialize data
                this->data = data;
        }
};
```

Taking input and print Recursive

Kindly follow the comments in the code to understand it better...

```
#include <iostream>
#include "TreeNode.h"                   // TreeNode.h file included as told above
using namespace std;
```

```
TreeNode<int>* takeInput() {   // Function that returns root node after taking input
        int rootData;                // To store root data
        cout << "Enter data" << endl;
        cin >> rootData;
        TreeNode<int>* root = new TreeNode<int>(rootData);
        // Dynamically created a root node and initialized with constructor

        int n;                    // To store number of children of the node
        cout << "Enter num of children of " << rootData << endl;
        cin >> n;
        for (int i = 0; i < n; i++) {
                TreeNode<int>* child = takeInput();        // Input taken recursively for
                                                           // each child node of the current node
                root->children.push_back(child);      // Each child node is inserted into
                                                           // the list of children nodes'
        }
        return root;
}

void printTree(TreeNode<int>* root) {    // Function to print the tree that takes the
                                         //  root node as its argument

        if (root == NULL) {                    // Base case
                return;
        }

        cout << root->data << ":";
        for (int i = 0; i < root->children.size(); i++) {    // Traversing over the vector of
                                                             // its child nodes and printing each of it
                cout << root->children[i]->data << ",";
        }
        cout << endl;
        for (int i = 0; i < root->children.size() ; i++) {      // Now recursively calling print
                printTree(root->children[i]);           // function over each child
        }
}

int main() {
        TreeNode<int>* root = takeInput();
        printTree(root);
      // TODO : Delete tree
}
```

We learnt that the memory that is created dynamically also needs to be deleted. The same will be followed here also. We suggest you implement it yourself as an exercise.

HINT: Idea will be recursively traversing over each node of the tree and first delete each child and then delete the root node.

Take input level-wise

For taking input level-wise, we will use **queue data structure.** Follow the comments in the code below:

```
TreeNode<int>* takeInputLevelWise() {           // Function to take level-wise input
        int rootData;
        cout << "Enter root data" << endl;
        cin >> rootData;
        TreeNode<int>* root = new TreeNode<int>(rootData);

        queue<TreeNode<int>*> pendingNodes; // Queue declared of type TreeNode
        pendingNodes.push(root);                // Root data pushed into queue at first
        while (pendingNodes.size() != 0) {      // Runs until the queue is not empty
                TreeNode<int>* front = pendingNodes.front();  // stores front of queue
                pendingNodes.pop();         // deleted that front node stored previously
                cout << "Enter num of children of " << front->data << endl;
                int numChild;
                cin >> numChild;    // get the number of child nodes
                for (int i = 0; i < numChild; i++) {  // iterated over each child node to
                                                      //  input it
                        int childData;
                        cout << "Enter " << i << "th child of " << front->data << endl;
                        cin >> childData;
                        TreeNode<int>* child = new TreeNode<int>(childData);
                        front->children.push_back(child);   // Each child node is pushed
//into the queue as well as the list of child nodes as it is taken input so that next
// time we can take its children as input while we kept moving in the level-wise
// fashion
                        pendingNodes.push(child);
                }
```

```
        }
        return root;                // Finally returns the root node
}
```

Similarly, we can also print the child nodes using a queue itself. Now, try doing the same yourselves and for solution refer to the solution tab of the respective question.

Count total nodes in a tree

To count the total number of nodes in the tree, we will just traverse the tree recursively starting from the root node until we reach the leaf node by iterating over the vector of child nodes. As the size of the child nodes vector becomes 0, we will simply return. Kindly check the code below:

```
int numNodes(TreeNode<int>* root) {
        if(root == NULL) {                          // Edge case
                return 0;
        }
        int ans = 1;                                // To store total count
        for (int i = 0; i < root->children.size(); i++) {      // iterating over children vector
                ans += numNodes(root->children[i]);   // recursively storing the count
                                                      // of children's children nodes.
        }
        return ans;                 // ultimately returning the final answer
}
```

Height of the tree

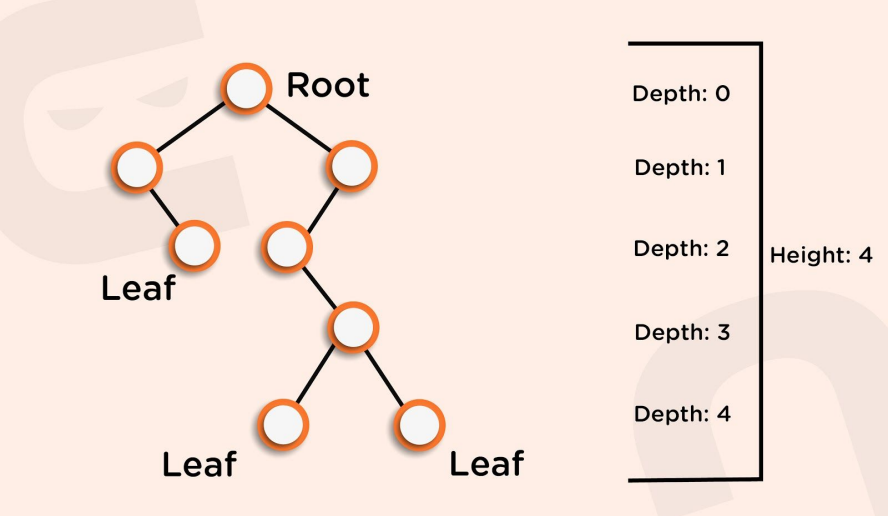
Height of a tree is defined as the length of the path from the tree's root node to any of its leaf nodes. Just think what should be the height of a tree with just one node? Well, there are a couple of conventions; we can define the height of a tree with just one node to be either 1 or zero. We will be following the convention where the height of a NULL tree is zero and that with only one node is one. This has been left

as an exercise for you, if need be you may follow the code provided in the solution tab of the topic corresponding to the same question.

Approach: Consider the height of the root node as 1 instead of 0. Now, traverse each child of the root node and recursively traverse over each one of them also and the one with the maximum height is added to the final answer along by adding 1 (this 1 is for the current node itself).

Depth of a node

Depth of a node is defined as it's distance from the root node. For example, the depth of the root node is 0, depth of a node directly connected to root node is 1 and so on. Now we will write the code to find the same... (Below is the pictorial representation of the depth of a node)



If you observe carefully, then the depth of the node is just equal to the level in which it resides. We have already figured out how to calculate the level of any node,using a similar approach we will find the depth of the node as well. Suppose, we want to find all the nodes at level 3, then from the root node we will tell its children to find the node that is at level 3 - 1 = 2, and similarly keep this up

recursively until we reach the depth = 0. Look at the code below for better understanding...

```
void printAtLevelK(TreeNode<int>* root, int k) {
        if(root == NULL) {                                  // Edge case
                return;
        }

        if(k == 0) {                                        // Base case: when the depth is 0
                cout << root->data << endl;
                return;
        }

        for(int i = 0; i < root->children.size(); i++) {    // Iterating over each child and
                printAtLevelK(root->children[i], k - 1);    // recursively calling with with 1
                                                            // depth less
        }
}
```

Count Leaf nodes

To count the number of leaves, we can simply traverse the nodes recursively until we reach the leaf nodes (the size of the children vector becomes zero). Following recursion, this is very similar to finding the height of the tree. Try to code it yourself and for the solution refer to the solution tab of the same.

Traversals

Traversing the tree is the manner in which we move on the tree in order to access all its nodes. There are generally 4 types of traversals in a tree:

- Level order traversal
- Preorder traversal
- Inorder traversal

- Postorder traversal

We have already discussed level order traversal. Now let's discuss the other traversals.

In Preorder traversal, we visit the current node first(starting with root) and then traverse the left sub-tree. After covering all nodes there, we will move towards the right subtree and visit in a similar manner. Refer the code below:

```
void preorder(TreeNode<int>* root) {
        if(root == NULL) {
            return;
        }
        cout << root->data << endl;
        for(int i = 0; i < root->children.size(); i++) {
            preorder(root->children[i].size());
        }
}
```

In postorder traversal, we visit all the child nodes first (from left to right order) and then we visit the current node. You will be coding this yourself (very similar to preorder traversal) and for solution, refer the solution tab in the corresponding questions.

We will study inorder traversal in further sections...

Destructor

Now, let's check the code to delete the tree that we left earlier. We will first-of-all delete the child nodes and then delete their root nodes and ultimately the main root node of the tree. If we simply delete the root node, then we will lose the references to its child nodes and hence only the root node will be deleted.

Kindly, refer to the code below for your reference:

```
void deleteTree(TreeNode<int> *root) {
        for(int i = 0; i < root->children.size(); i++) {
                deleteTree(root->children[i]);
        }
        delete root;
}
```

We will call this function in the main() and the tree will be deleted. But to make the code robust and easy to understand, we will be using destructors. We just want to call **delete root;** in the main() and it should delete the complete tree. Let's check the destructor part below:

```
~TreeNode() {
        for(int i = 0; i < children.size(); i++) {      // We will call delete on all its
                delete children[i];                     // children which will invoke
        } // corresponding destructor and ultimately delete the root node itself.
}
```

This destructor works in the same way as the recursive functions but is a better choice as code looks clean and easy-to interpret.