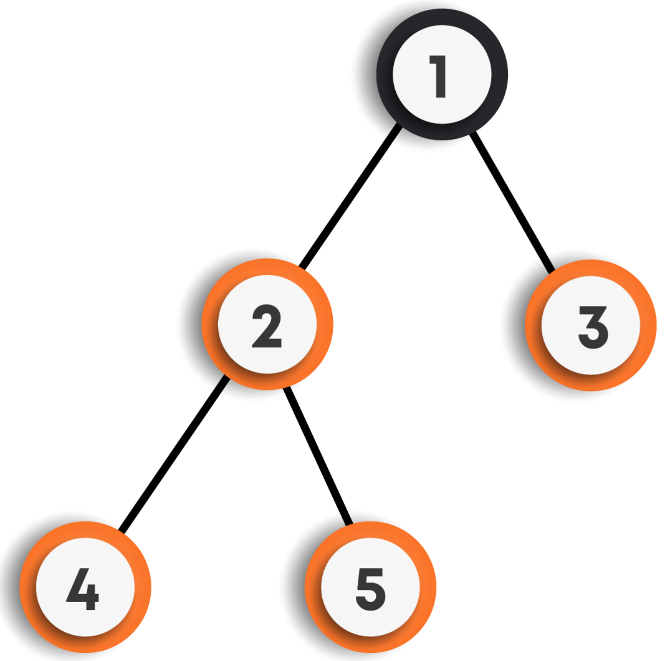
Tree - 1

Introduction

There are various systems around us that are hierarchical in nature; military, government organizations, corporations, all have a hierarchical form of The command chain or 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 trees, depending on the rule/property they follow. The simplest variant of the tree is called a **general tree (or N-ary tree)**. As in the 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.

Trees are non-linear hierarchical data structures. It is a collection of nodes connected to each other by means of **edges** that are either directed or undirected. One of the nodes is designated as **Root nodes** and the remaining nodes are called child nodes or the **leaf nodes**(nodes with no child nodes). In general, each node can have as many children but only one parent node.



In the figure above there are 5 nodes **1, 2, 3, 4, 5**.

Node **1** is a **Root node.**

Nodes **3, 4, 5** are **leaf nodes.**

Node **2** is a **parent node** of **4&5.**

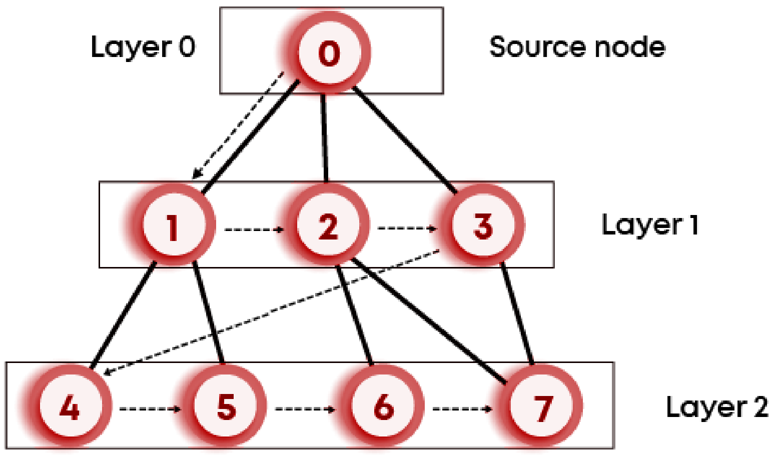
Tree traversal techniques

BFS Traversal:

Breadth-first search(BFS) is an algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes (which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

Algorithm:

- We first move horizontally and visit all the nodes of the current layer.
- Then move to the next layer.

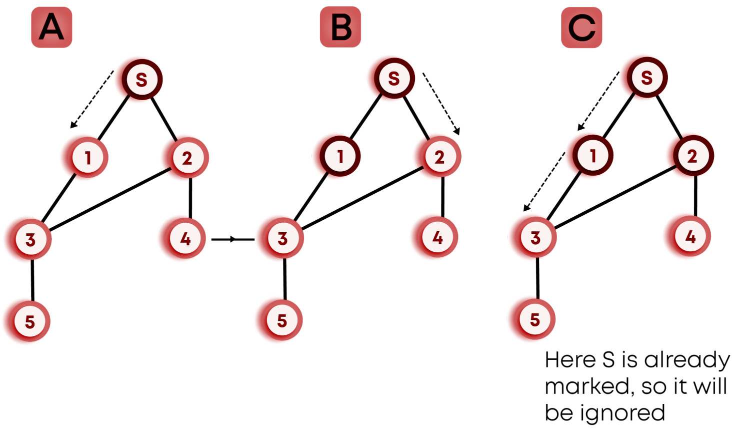


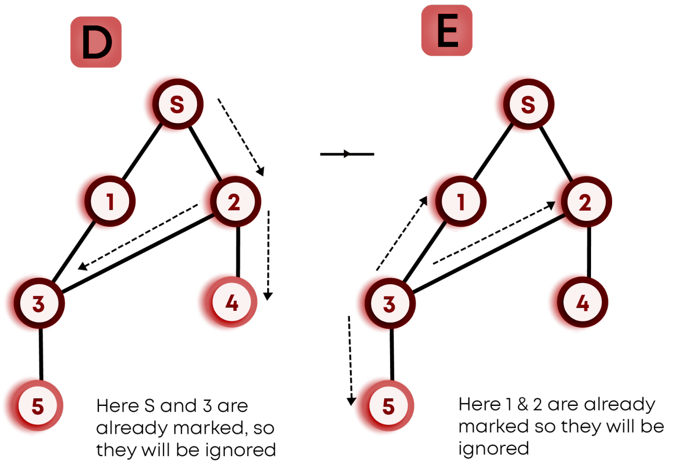
This is an iterative approach. We will use the queue data structure to store the child nodes of the current node and then pop out the current node. This process will continue until we have covered all the nodes. Remember to put only those nodes in the queue which have not been visited.

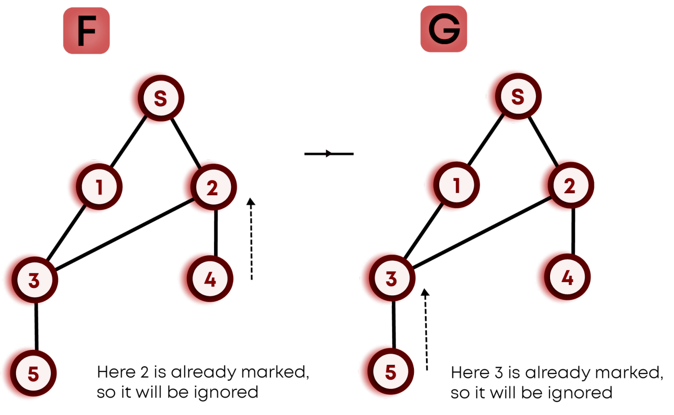
Code:

```
void bfs(int root){
        queue<int> q;
        q.push(root);
        vector<bool> visited(N, false);

        visited[root] = true;

        cout<<"Running BFS---\n";
        while(!q.empty()){
               int u = q.front();
               cout<<u<<" ";
               q.pop();

               for(int v: tree[u]){
                      if(!visited[v]){
                             visited[v] = true;
                             q.push(v);
                      }
               }
        }
        cout<<endl;

}
```

Consider the dry run over the example graph below for a better understanding of the same:




D
E
S
1
2
3
4
5
Here S and 3 are already marked, so they will be ignored
Here 1 & 2 are already marked so they will be ignored


F
G
S
1
2
3
4
5
Here 2 is already marked, so it will be ignored
Here 3 is already marked, so it will be ignored

DFS Traversal:

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Algorithm:

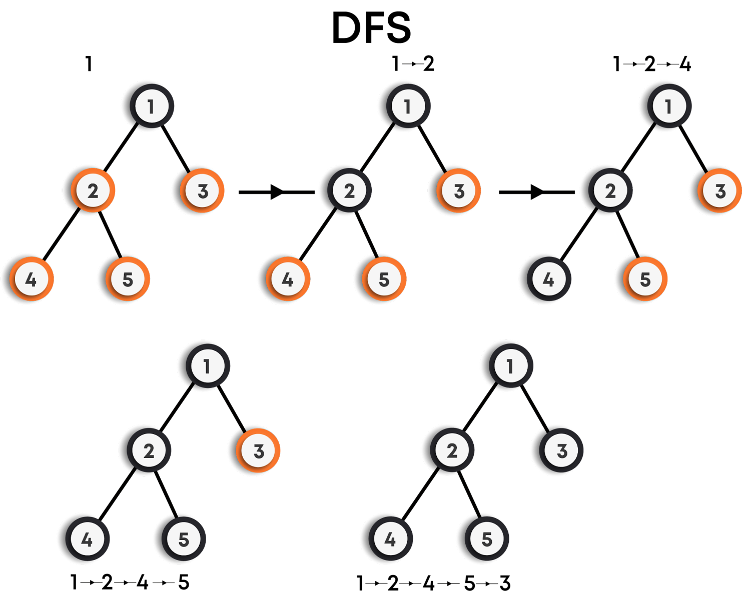
- Create a recursive function that takes the index of the node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.

Code:

```
void dfs(int s, int p){
      cout<<s<<" ";
      for(int v: tree[s]){
            if(v != p){
                  dfs(v, s);
            }
      }
}
```

Time complexity: **O(V + E)**, where **V** is the number of vertices and **E** is the number of edges in the graph.

Space Complexity: O(V), since an extra visited array of size **V** is required.



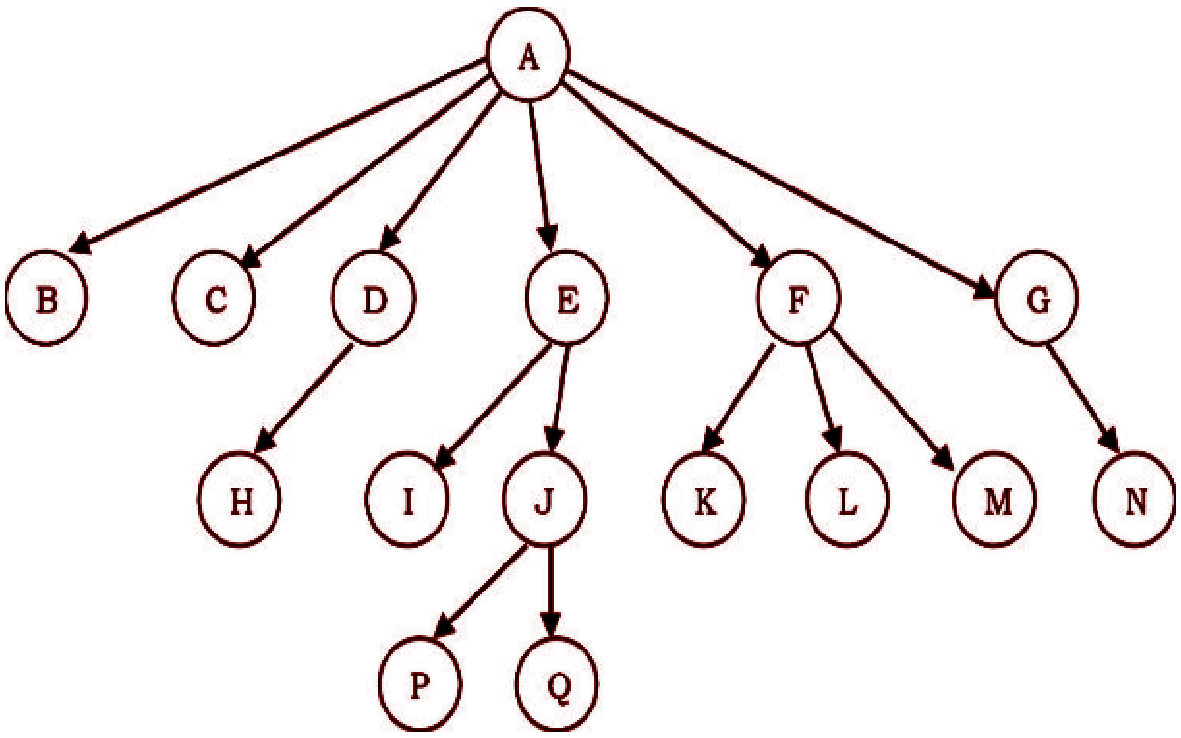
Tree Diameter

Problem Statement: You are given a tree with **N** vertex and **N - 1** edge and you are supposed to find the diameter of the tree. The diameter of a tree is defined as the maximum distance between any pair of leaves of a tree.

Explanation:

Example: Let's take an example. In the following tree, we have **16 nodes (A-Q)**. The diameter of a given tree is **5** as this is the maximum distance between any pair of nodes. There exist multiple pairs of nodes having distance **5** one of them is (**P-H**).

Traversal would be like (P->J->E->A->D->H).



Algorithm:

- For every node, Calculate the maximum height of a leaf node in the subtree of this node
- Let's denote this with **max_ht[u]**. Where **u** is the node for which we are doing the calculation.
- Now, we find the maximum and the second value of **max_ht[v],** where **v** is the child node of **u.**
- The diameter is equal to the sum of these two maximums.

Code:

```cpp
#include<bits/stdc++.h>
using namespace std;

const int N = 1e4+5;
vector<int> tree[N];
int height[N];
int max_ht[N];

int ans = 0;

void dfs(int s, int p, int ht=0){
    height[s] = ht;
    max_ht[s] = ht;
    int maxim = 0, secondmaxim = 0;

    for(int v: tree[s]){
        if(v != p){
            dfs(v, s, ht+1);
            max_ht[s] = max(max_ht[s], max_ht[v]);
            if(max_ht[v] > maxim){
                secondmaxim = maxim;
                maxim = max_ht[v];
            } else if(max_ht[v] > secondmaxim){
                secondmaxim = max_ht[v];
            }
        }
    }

    int max_distance = max(0, max(maxim - ht, maxim + secondmaxim - 2*ht));
    ans = max(ans, max_distance);
}

int main(){
    int t;
    cin>>t;
    while(t--){
        int n;
        cin>>n;

        for(int i=0; i<=n; i++){
            tree[i].clear();
            height[i] = max_ht[i] = 0;
        }

        for(int i=0; i<n-1; i++){
            int u, v;
            cin>>u>>v;
            tree[u].push_back(v);
            tree[v].push_back(u);
        }

        ans = 0;
        dfs(1, 0);

        cout<<ans<<"\n";

    }
    return 0;
}
```

XOR Query:

Problem Statement: You are given a tree(root 0) with **N** vertex having **N - 1** edge. You are also given an array ‘**QUERY’** of size **‘Q’**, where each query consists of an integer that denotes a node. You have to print the xor of all the values of nodes in the subtree of the given node.

Note: Here XOR denotes the bitwise operator (^).

Algorithm:

- Create a graph **GRAPH** using the **EDGES** array in the form of an adjacency list. (An array of lists in which 'arr[i]' represents the lists of vertices adjacent to the 'ith' vertex).
- Create an array **RES** to store the result of queries.
- Run a loop from 0 to **Q** iterator 'i' to traverse the queries.
- Call the **DFS** function to compute the **XOR** of the subtree and store the value returned by the function in the **RES**.
- Return **RES**.

DFS('GRAPH', 'CURR', 'PREV') (where 'CURR' is the current vertex and 'PREV' is the previous vertex traversed).

- Create a variable **VAL** to store the **XOR** value and initialize it with **CURR**.
- Traverse the adjacent vertices of the **CURR** vertex.
- Check if **V** is not equal to **PREV**.
- Recursively call the **DFS** function to traverse the tree and update the **VAL** by taking the **XOR** of it by the value returned by the function.
- Return **VAL**.

Code:

```
int dfs(vector<vector<int>> &graph, int curr, int prev) {
    int val = curr;
    for(int i = 0; i < graph[curr].size(); ++i) {
        if(graph[curr][i] != prev) {
            val ^= dfs(graph, graph[curr][i], curr);
        }
    }

    return val;
}
```

```
vector<int> XORquery(int n, vector<vector<int>> &edges, int q,
vector<int> &query) {

    vector<vector<int>> graph(n);
    for(int i = 0; i < n - 1; ++i) {
        graph[edges[i][0]].push_back(edges[i][1]);
    }
    vector<int> res;
    for(int i = 0; i < q; ++i) {
        res.push_back(dfs(graph, query[i], -1));
    }
```

Time Complexity: O(N * Q), where **N** is the number of vertices and **Q** is the number of queries.

We run a loop to answer all the queries which take **O(Q)** time and in the worst case for each query we traverse all the nodes which take **O(N)** time. Therefore, the overall time complexity will be **O(N * Q)**.

Space Complexity: O(N), where **N** is the number of vertices.

For each vertex, we store its adjacent vertices; therefore, storing them in the form of a graph will require **O(N)** space. The recursive stack of the HELPER function will contain all the neighbors of a vertex, which at most can be **N** vertices. Therefore, the overall space complexity will be **O(N)**.