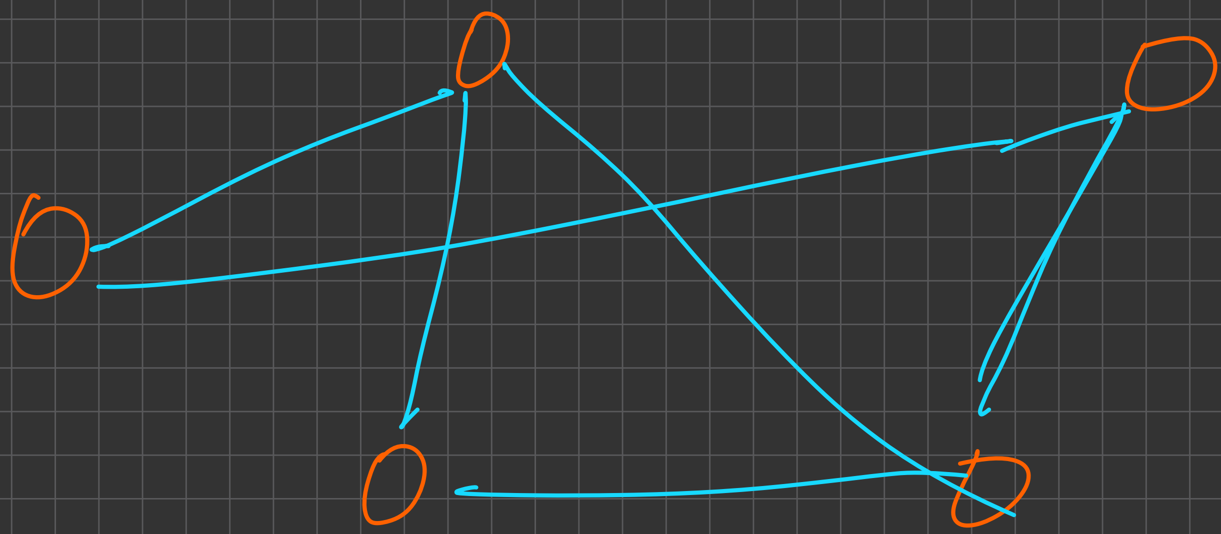
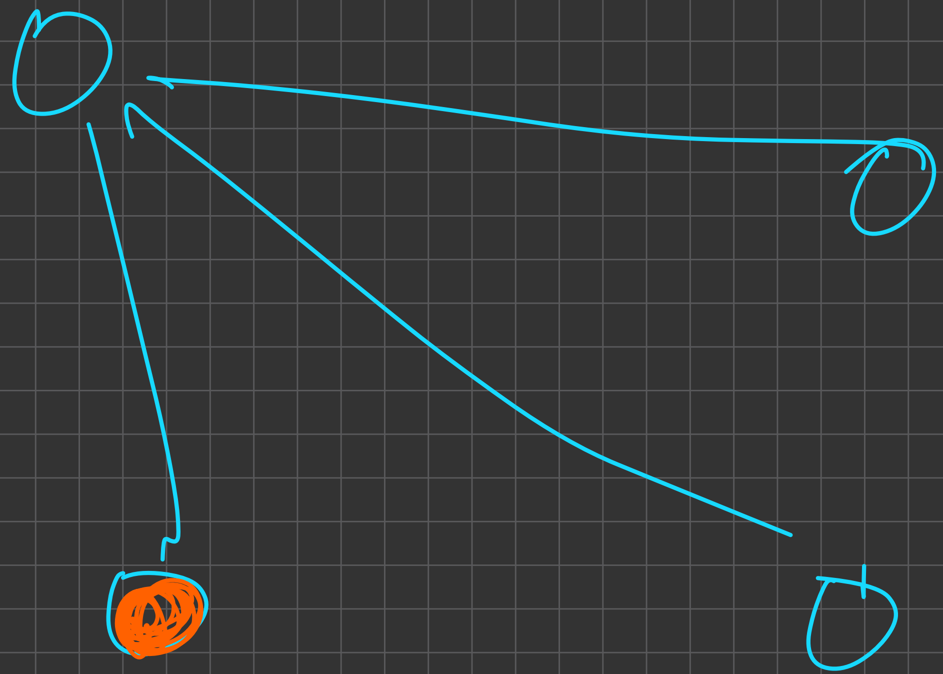
Trees 1

Graph

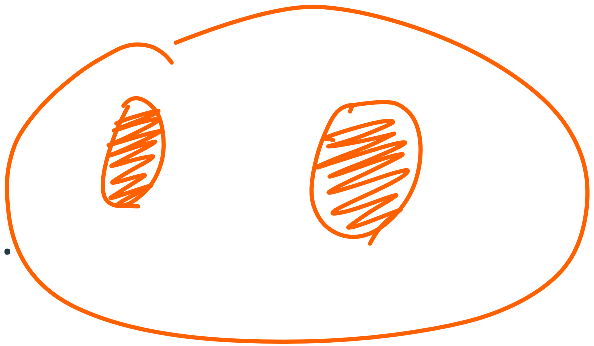


Tree → a graph which is connected and has no cycles



Connected

What is a Tree



A connected graph of N nodes without any cycles.

What is a graph?

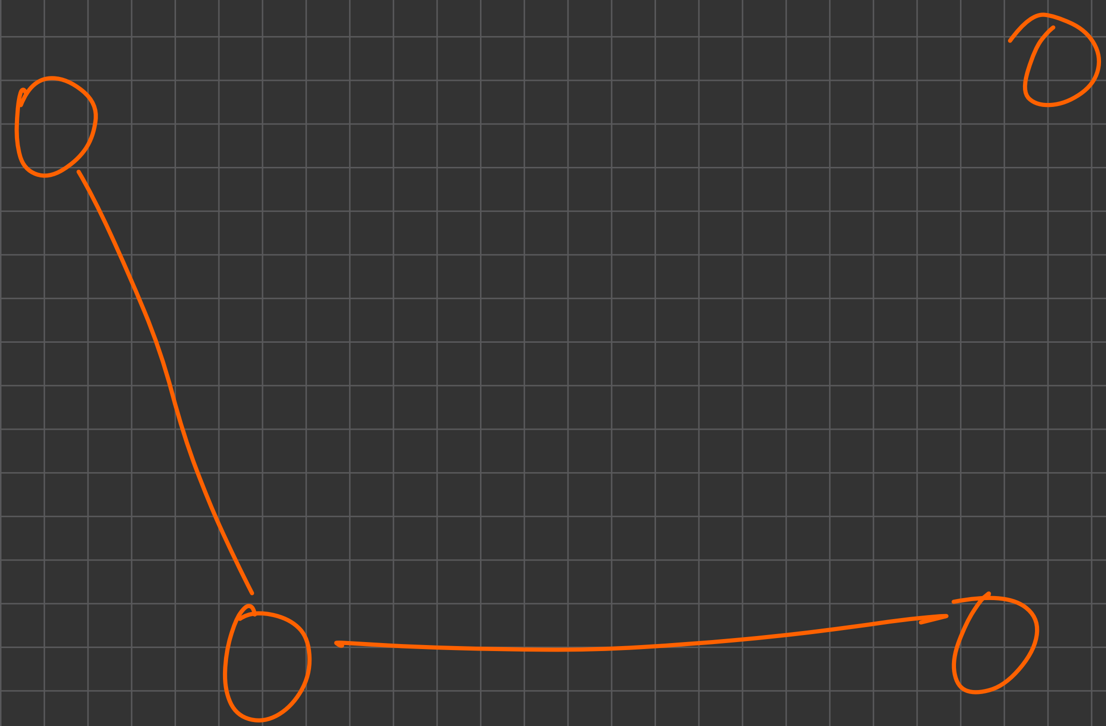
Imagine it like the Earth

Contains a bunch of countries connected via roads

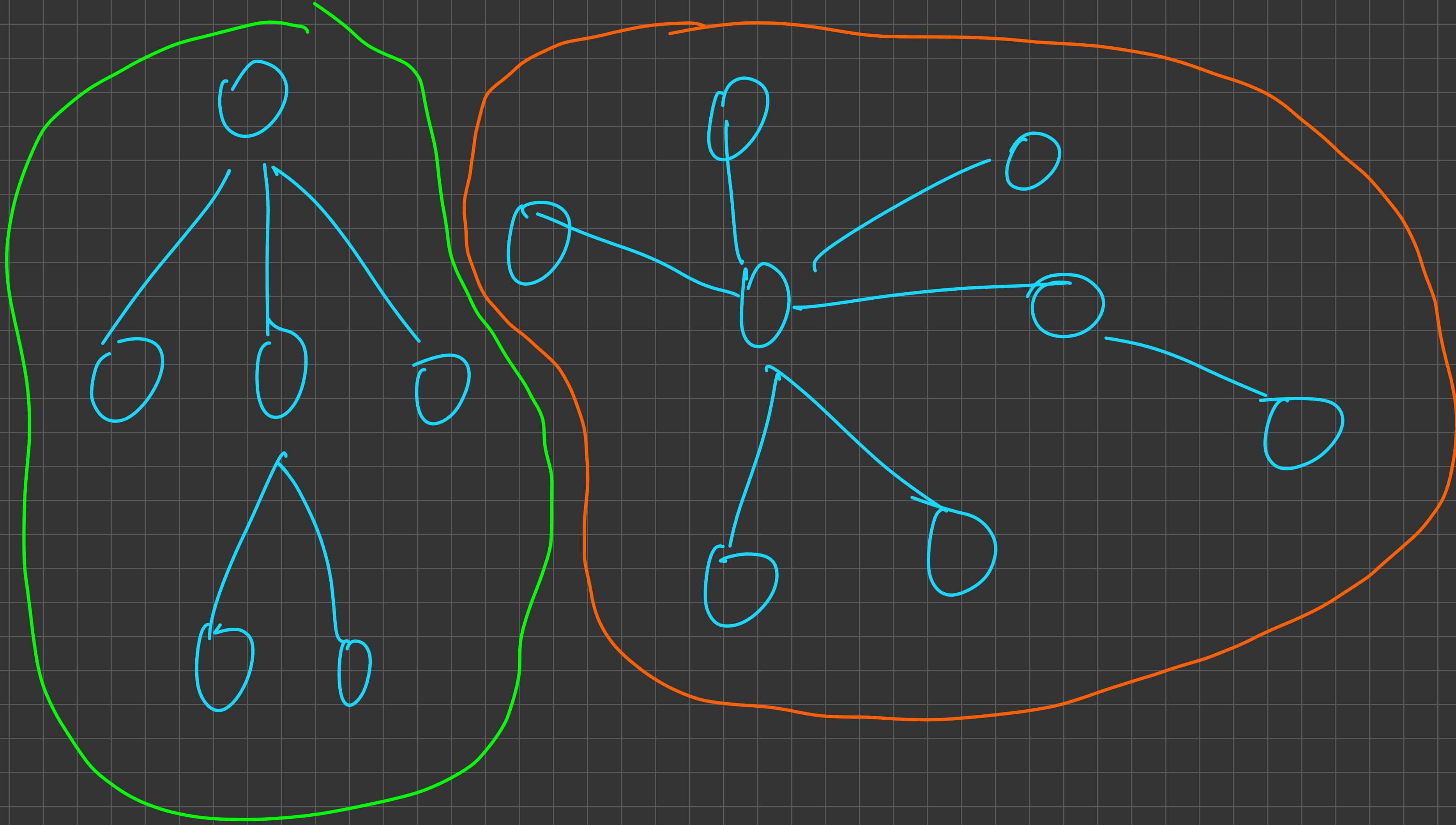
A continent is a group of countries directly or indirectly connected to each other

Some countries might be in different continents -> disconnected

A tree is one such continent with a unique path b/w any 2 countries



Tree → a component



Tree ⟶ connected and acyclic

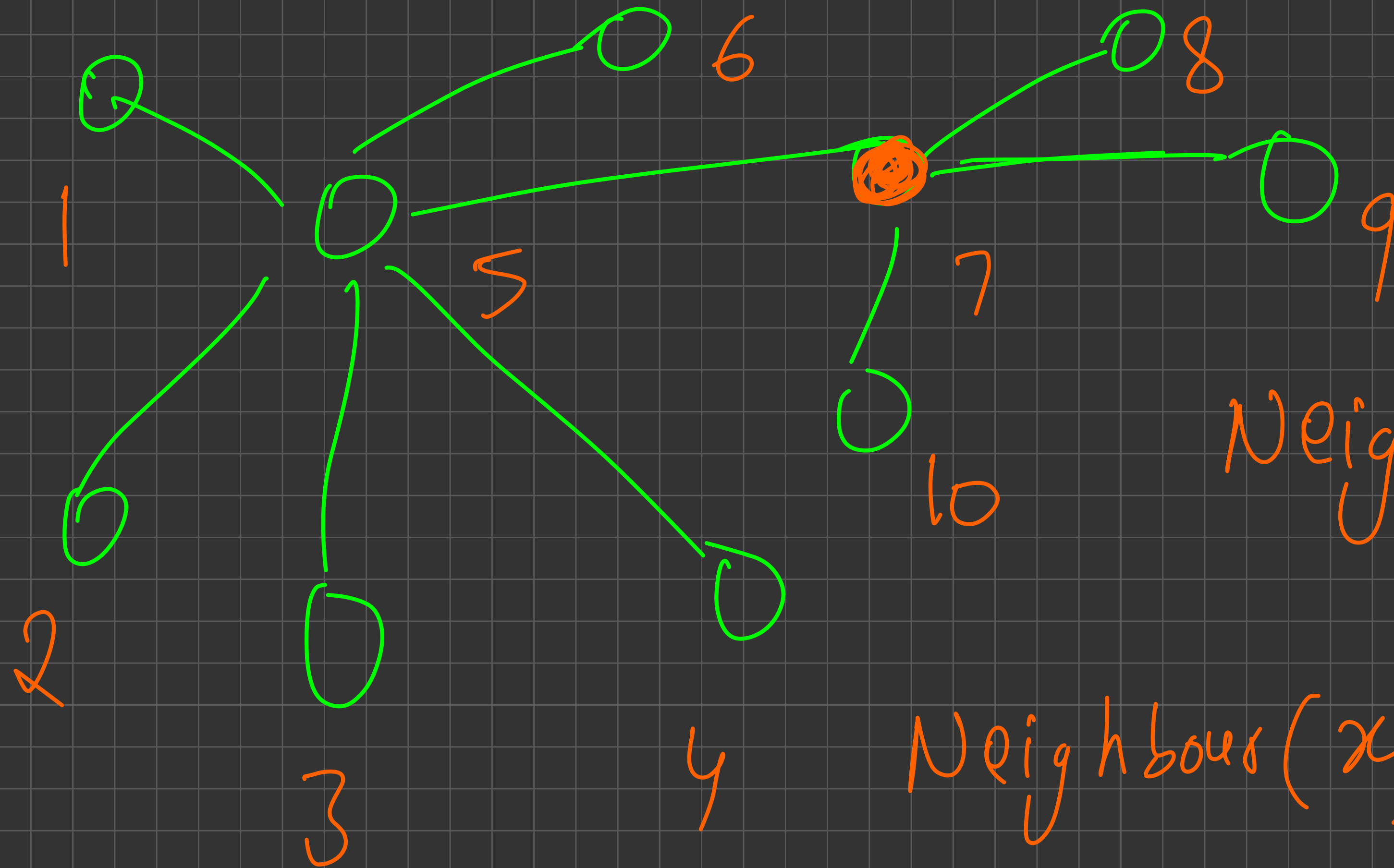
What is a Tree

Differentiate b/w a Tree and a Graph from examples

- One Note Illustrations

Some Common Terms

- Neighbour
- Degree
- Leaf and non-leaf Nodes (aka Internal Nodes)
- Diameter (can be non-unique)



Neighbour

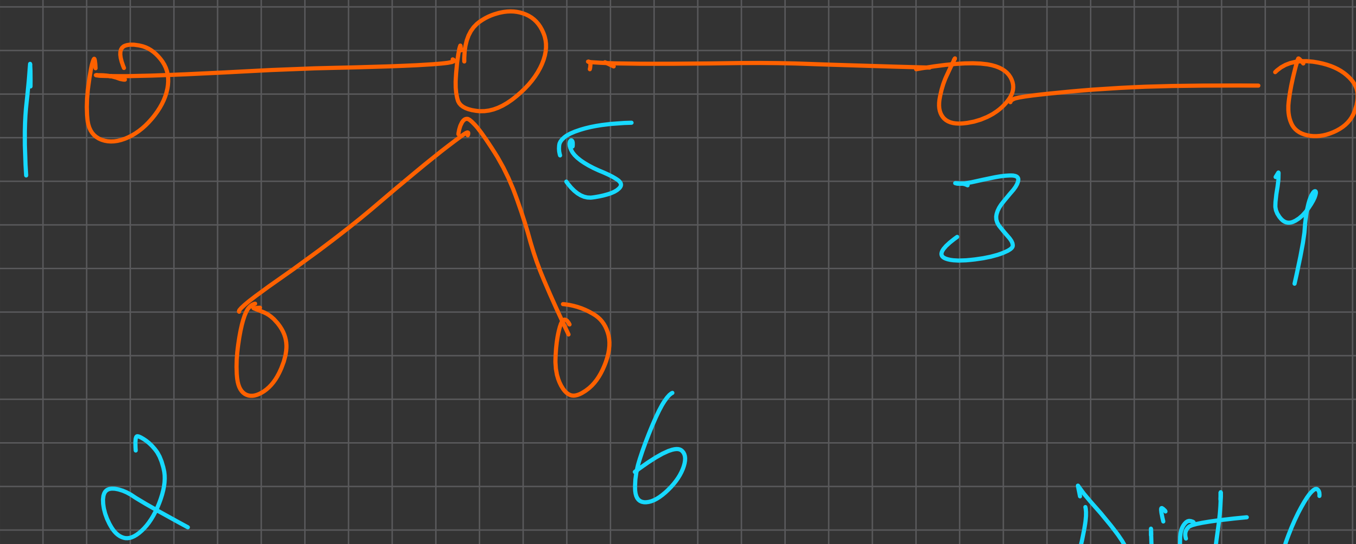
Neighbour(x) = All nodes that can be reached with just **1 edge**

$$\text{Degree}(x) = |\text{Neighbours}(x)|$$

Leaf node = any node with degree = 1

Diameter of Tree = maximum distance b/w any 2 nodes

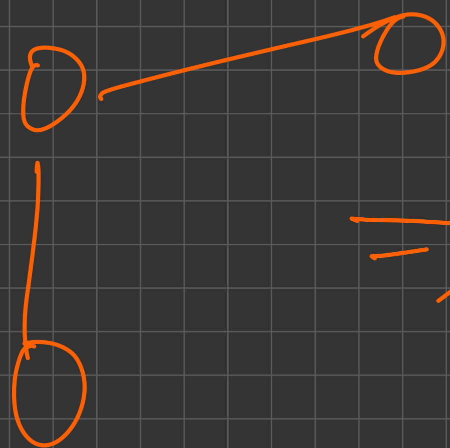
= # of edges



$$\mathrm{Dist}(1, 4) = 3$$

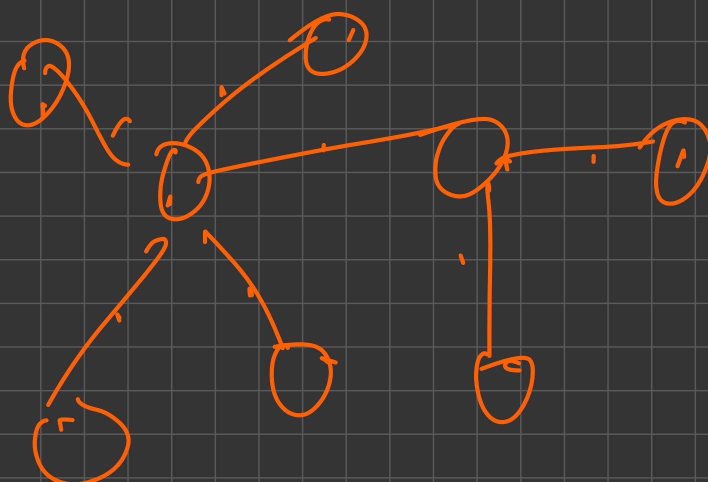
Properties 1

- Number of Edges in a Tree for N nodes
  - N - 1
- Number of paths between 2 nodes
  - 1
- Sum of Degree of all nodes
  - 2 * (N - 1)
- Can there be less than 2 leaf nodes in a Tree
  - No, except for the case when there is just one node in the entire tree



$\Rightarrow$ $N = 3$

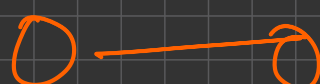
Edges $= 2$

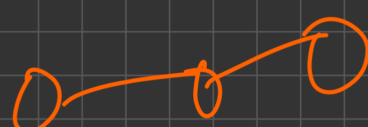


$\Rightarrow$ $N = 8$

Edges $= 7$

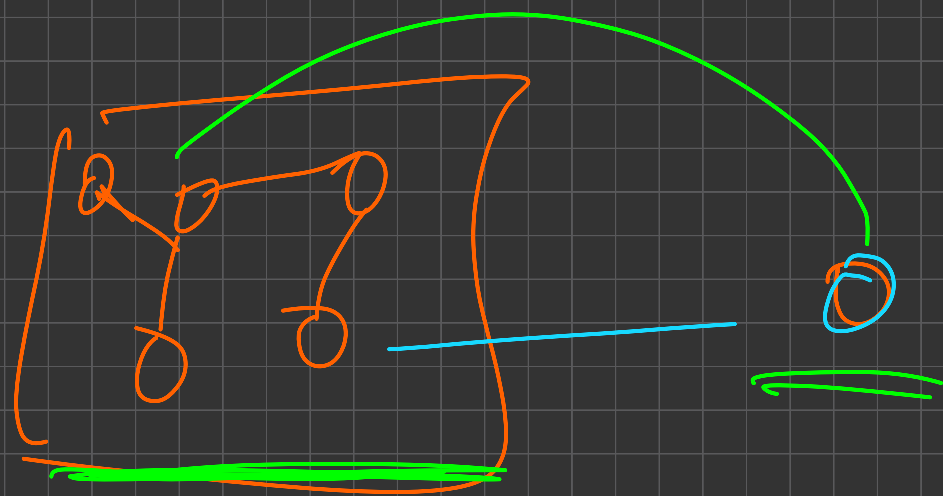
$N = 1$ $\rightarrow$  Edges $= 0$

$N = 2$ $\rightarrow$  Edges $= 1$

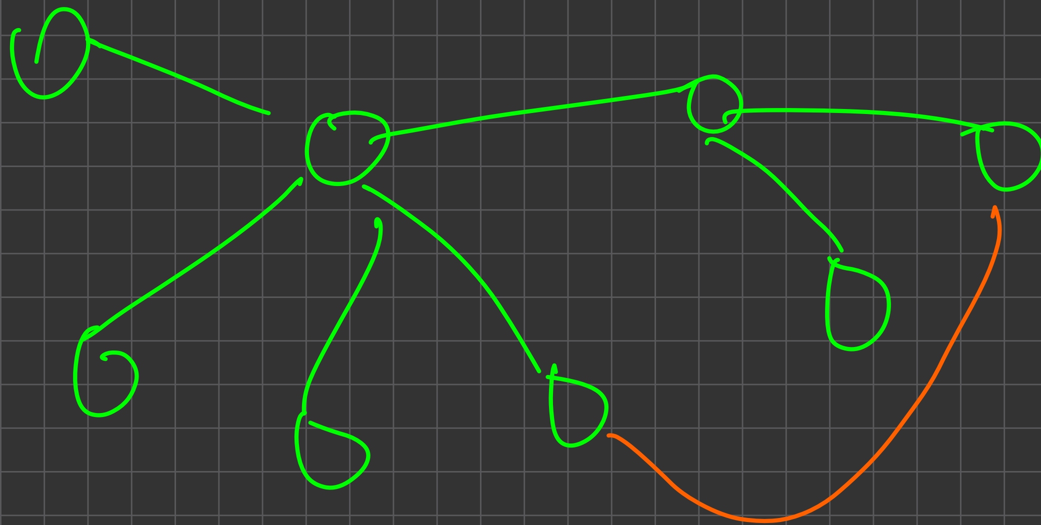
$N = 3$ $\rightarrow$  Edges $= 2$

Assume that a Tree of size k requires $k-1$ edges to be connected

$k+1$



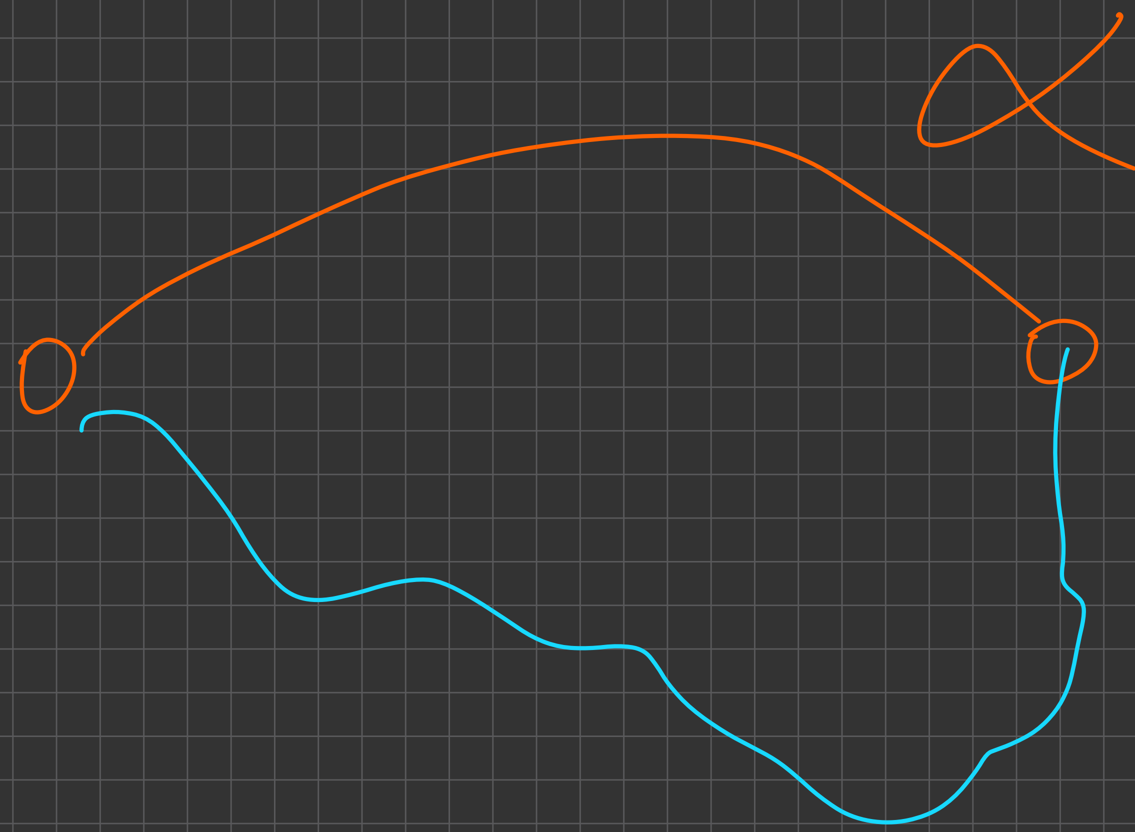
k nodes



You require $N-1$ edges to keep a tree connected and adding even 1 more edge makes a cycle

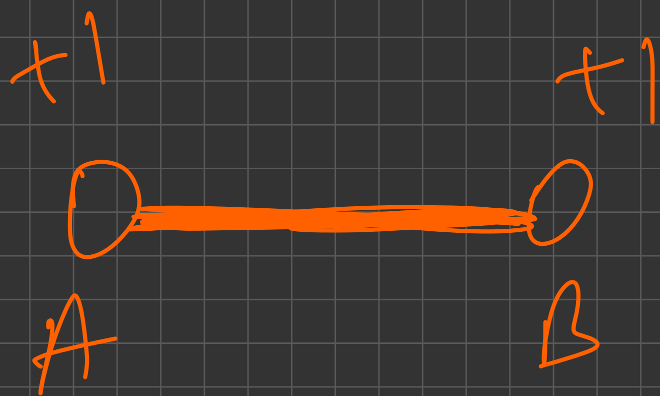
$\boxed{\text{exact} = \underline{N-1} \text{ edges}}$

Tree



Sum of degree of all nodes $= \boxed{2\cdot(N-1)}$

No. of edges $= n-1$



+2

2 3 4 1 1 1

Can there be < 2 leaf nodes in a tree with at least 2 nodes

0

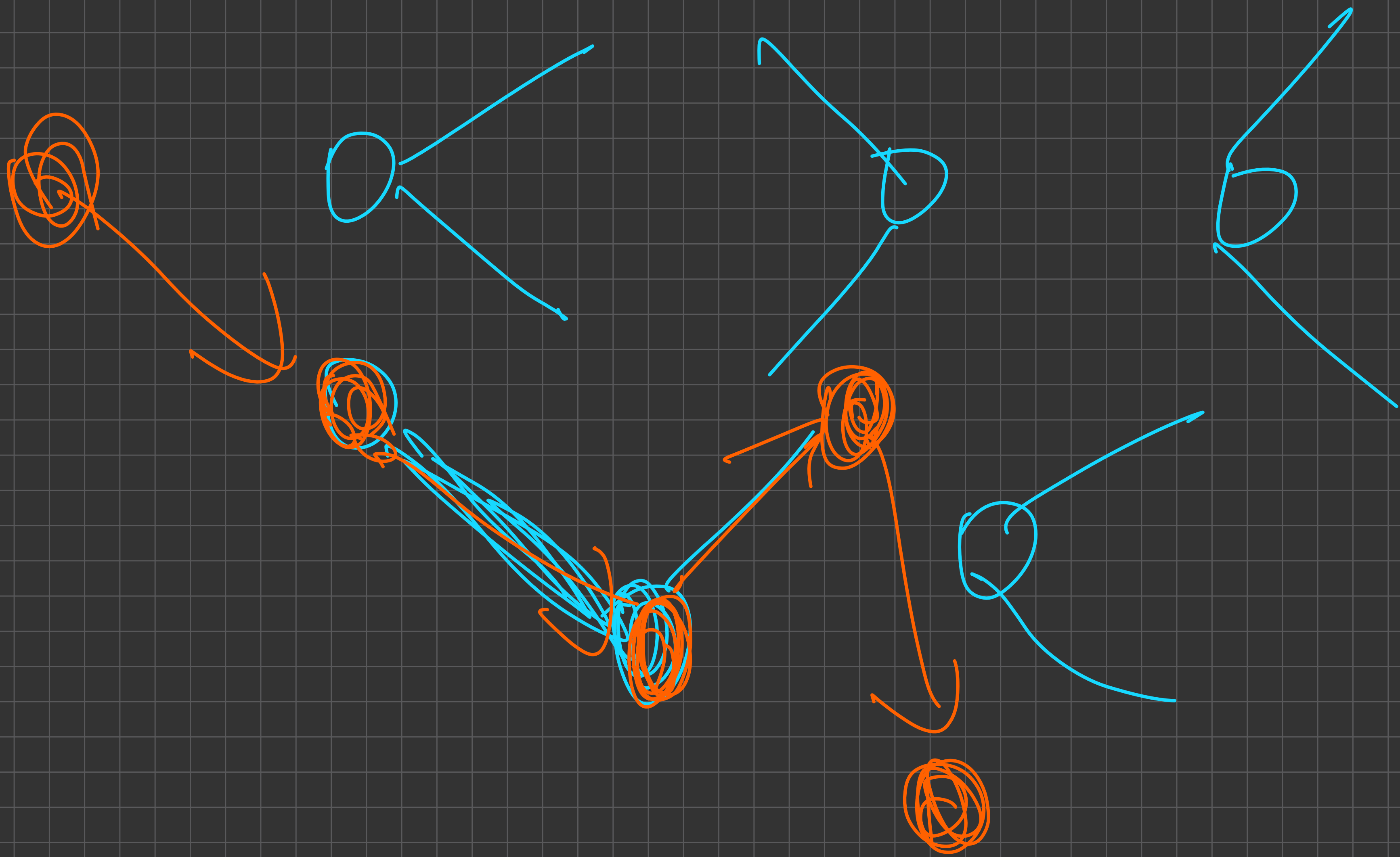
No

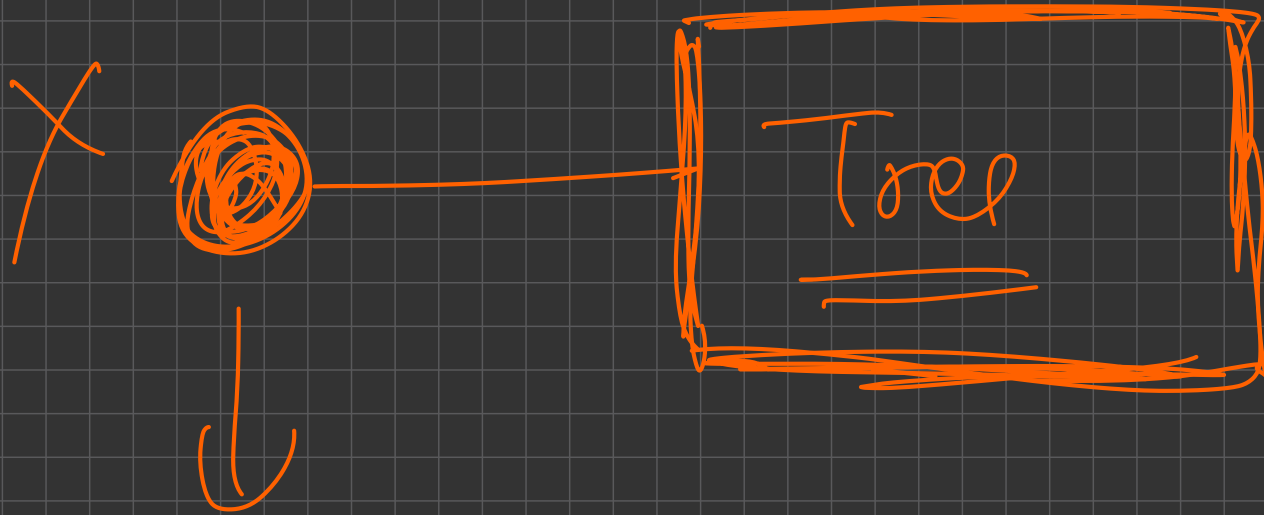
Can we have just 1 leaf node

✗

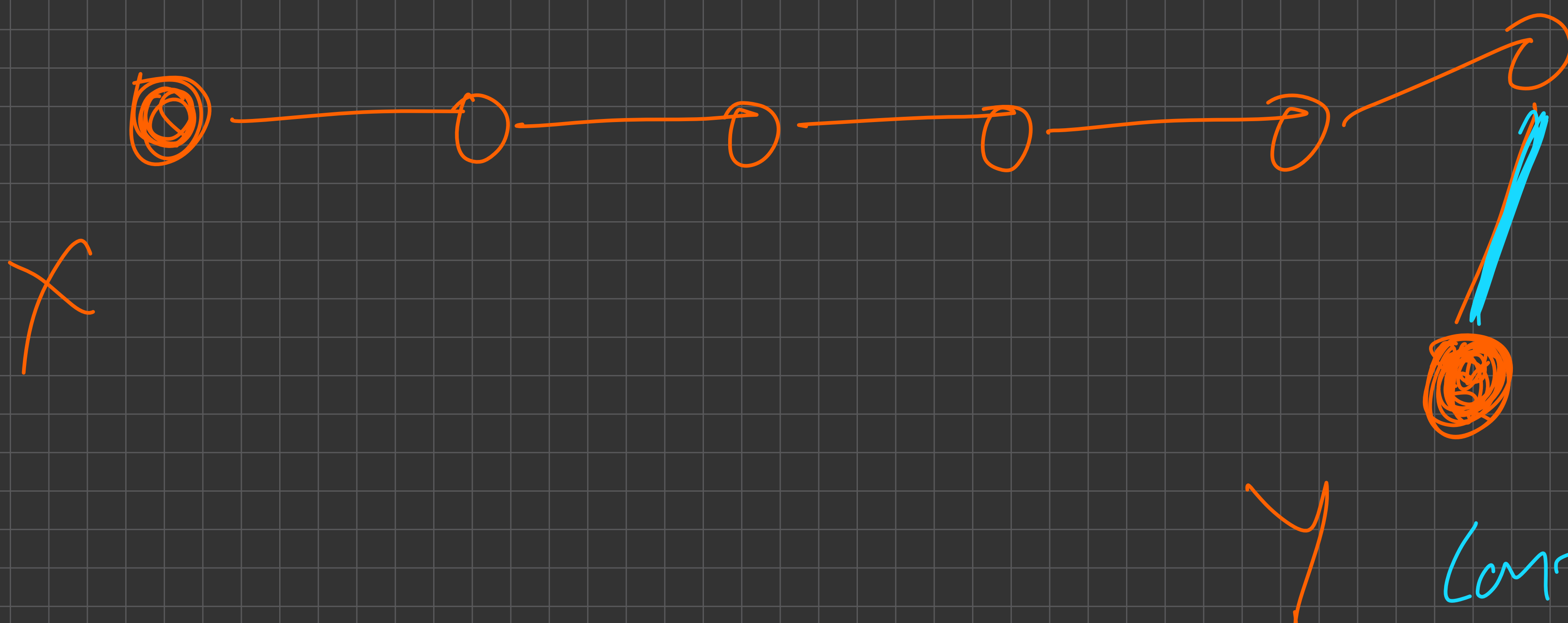
Can we have 0 leaf nodes

✗





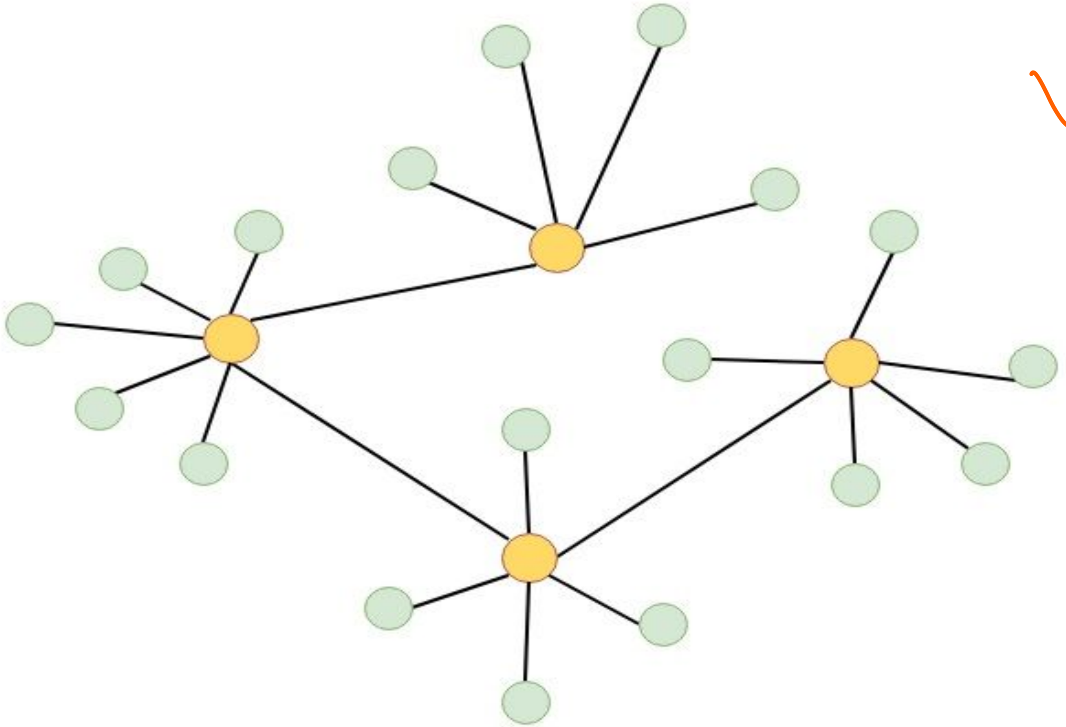
no leaf nodes



Contradiction

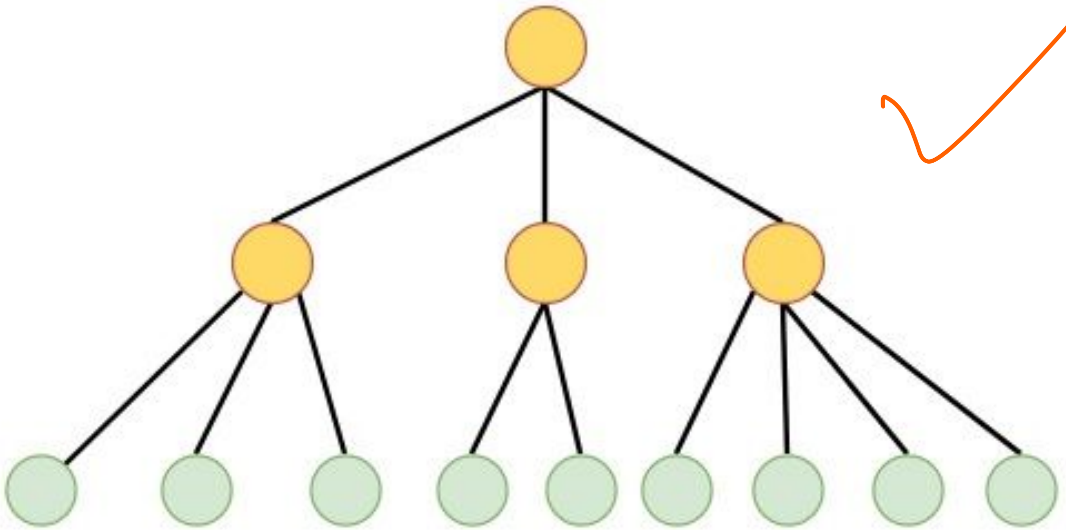
Rooted and Unrooted Trees

Unrooted



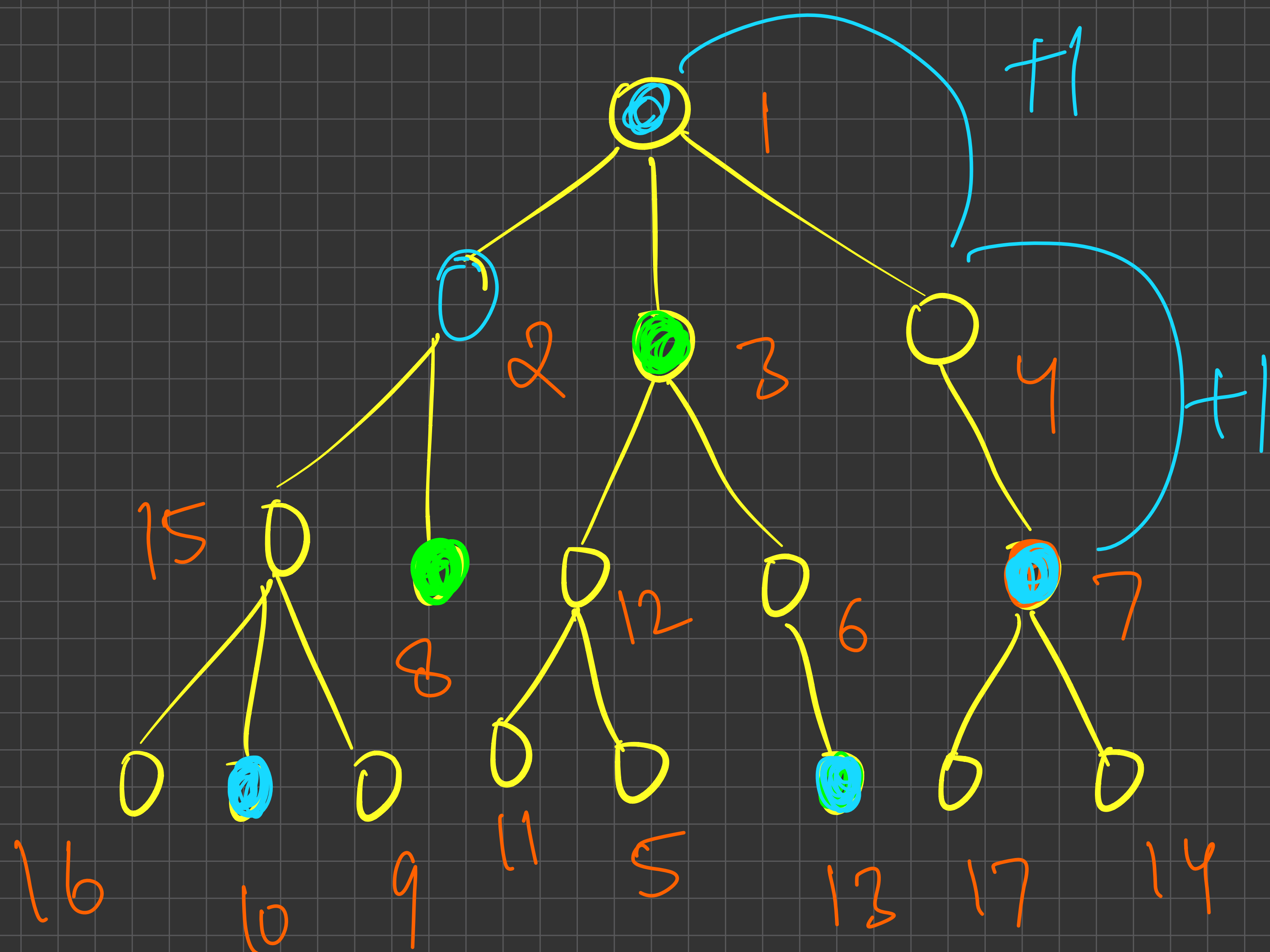
Rooted and Unrooted Trees

Rooted



Some more terms

- Root
- Parent
- Child
- Ancestor
- Descendant
- Level of Node
- Subtree
- Subtree Size = no. of nodes in subtree
- Height of Tree
- Lowest Common Ancestor


+1
+1
1
2
3
4
15
8
12
6
7
16
10
9
11
5
13
17
14

Ancestor of x

$= \{$ parent(x) $\cup$ Ancestors of parent(x) $\}$

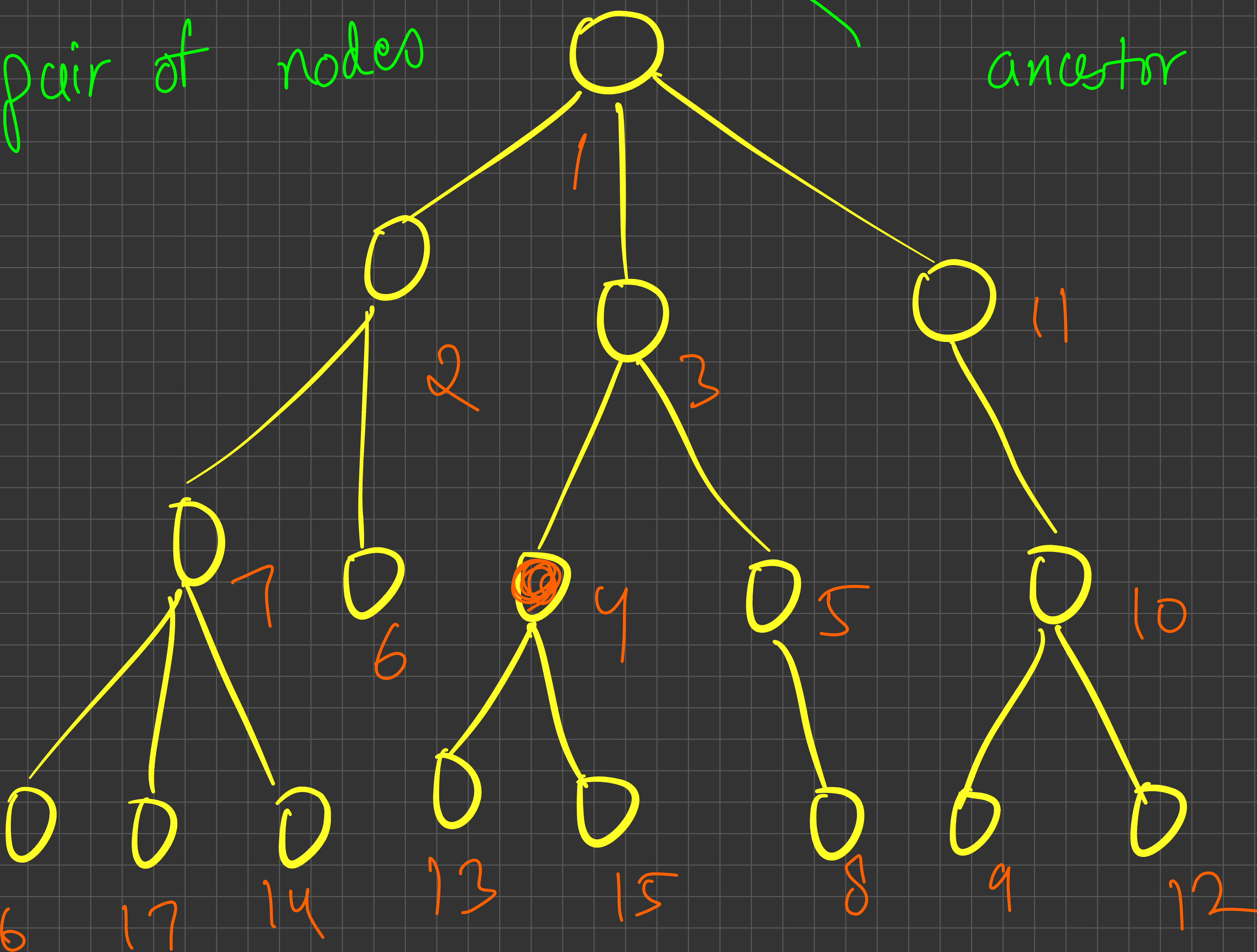
Descendents of x

$= \{$ children(x) $\cup$ descendents of every child of x $\}$

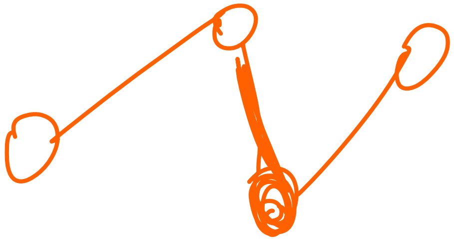
Level of node = distance of node from root

Height of Tree = maximum level of a node defined for a pair of nodes

lowest common ancestor



Properties 2



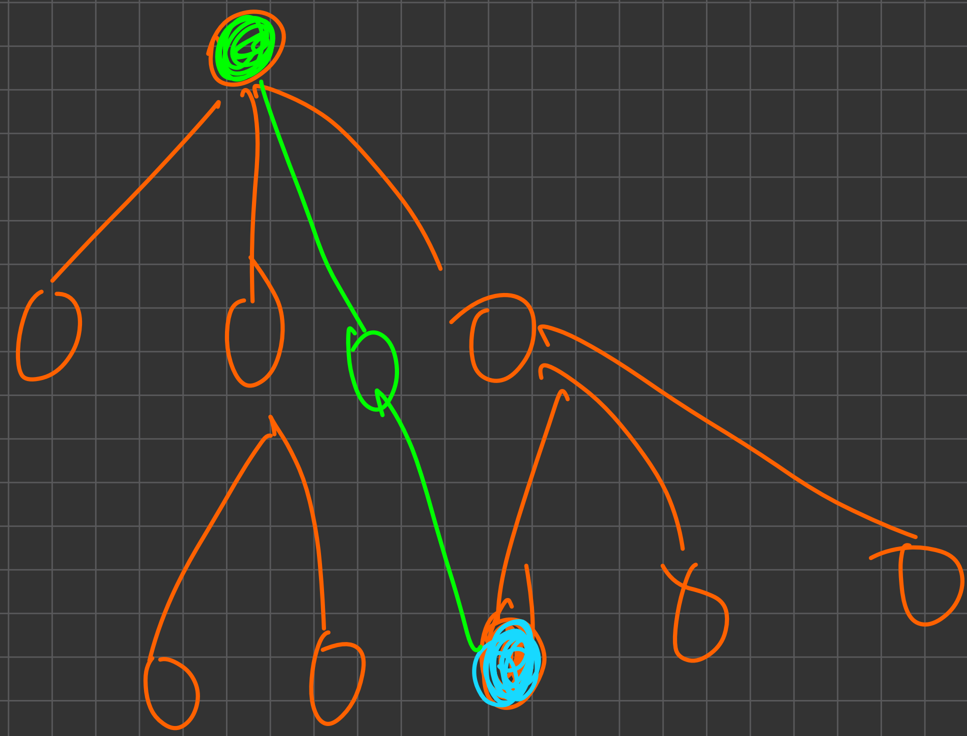
- Can there be more than 2 parents of a single node in a rooted tree
    - No
- Path property. What are the only 3 types of paths possible?
    - 1. **Go Up, Come Down** 2. **Go Up** 3. **Come Down**
- How to color a Tree with just 2 colors such that no two neighbours have the same color
    - Just root the tree and color level wise

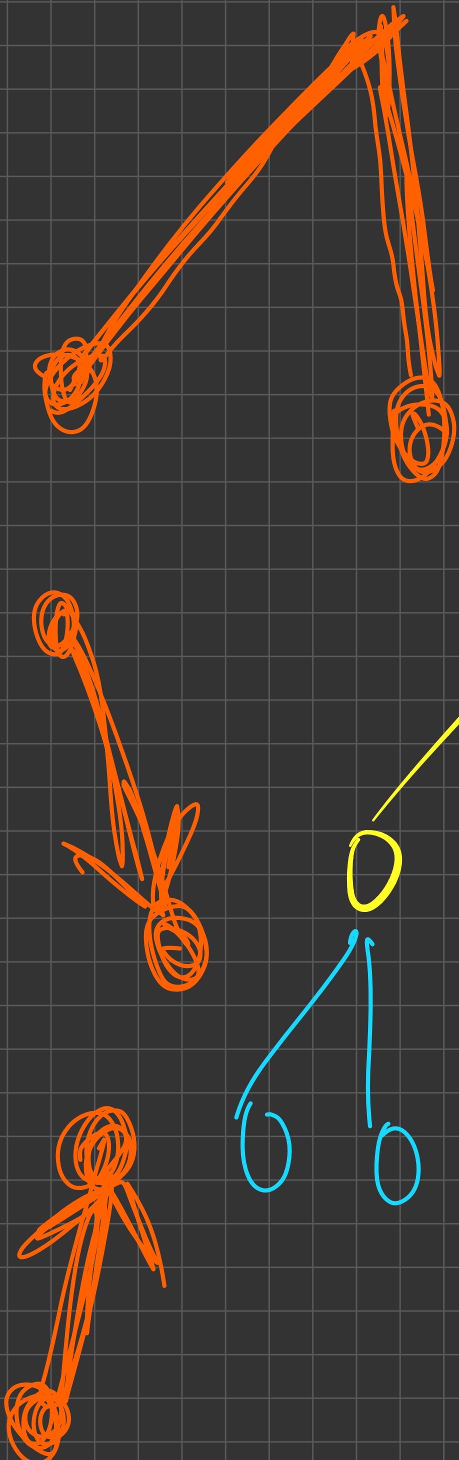
Binary lifting, euler tour

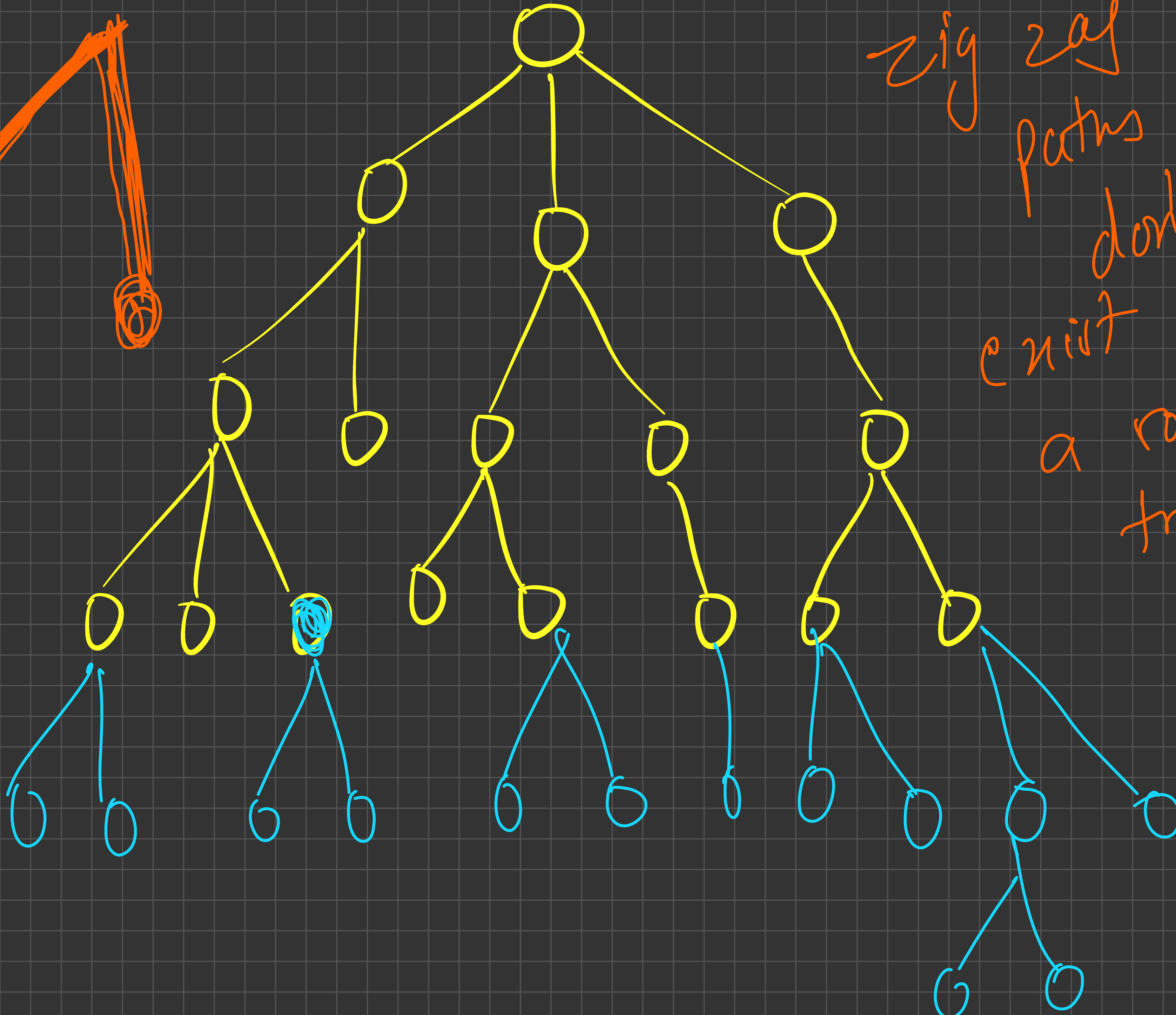
Bonus Tip:

Most Codeforces problems less than 1800 rated on Codeforces can be easily solved if you just remember the basic properties and learn to apply them.

Dp on trees





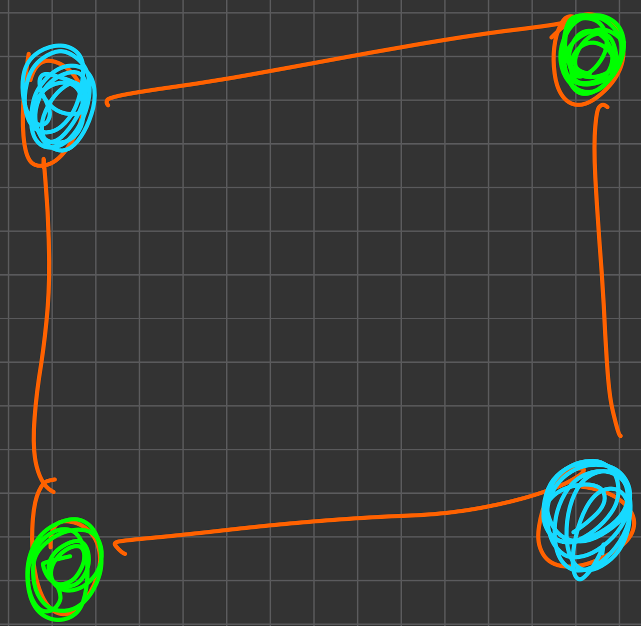


zig zag paths don't exist in a rooted tree

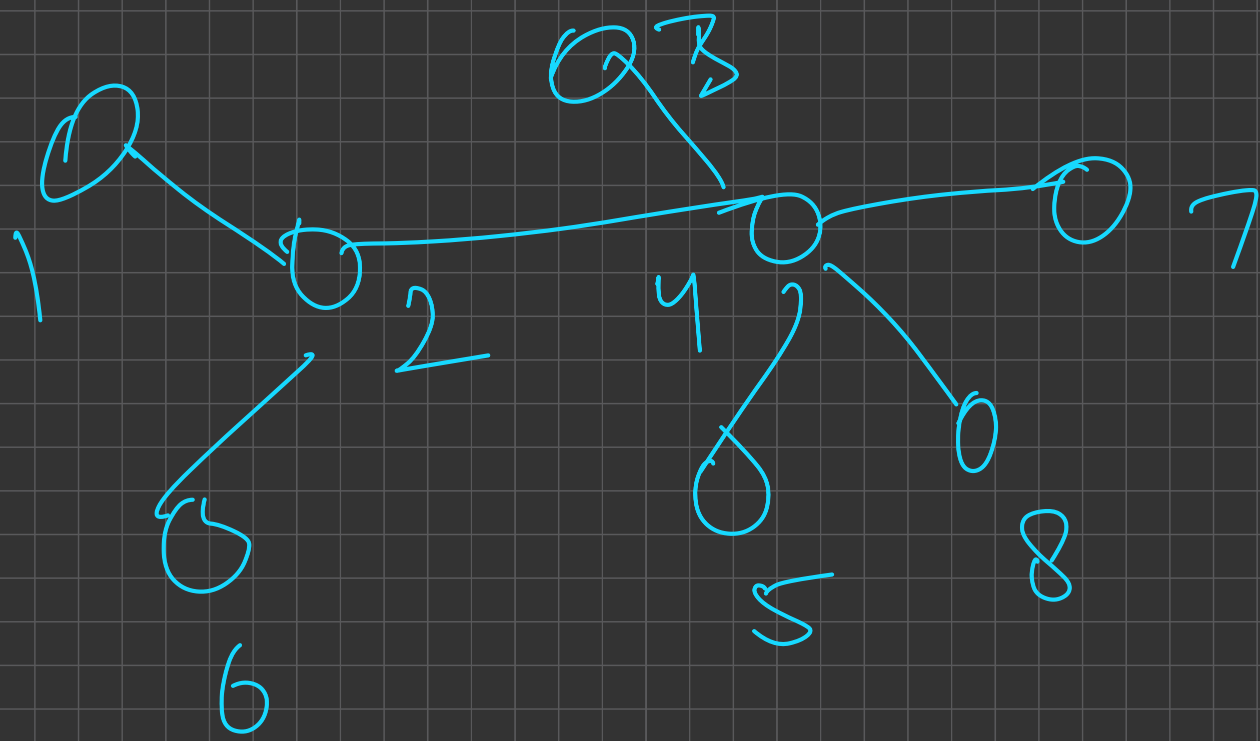
Bipartite Colouring

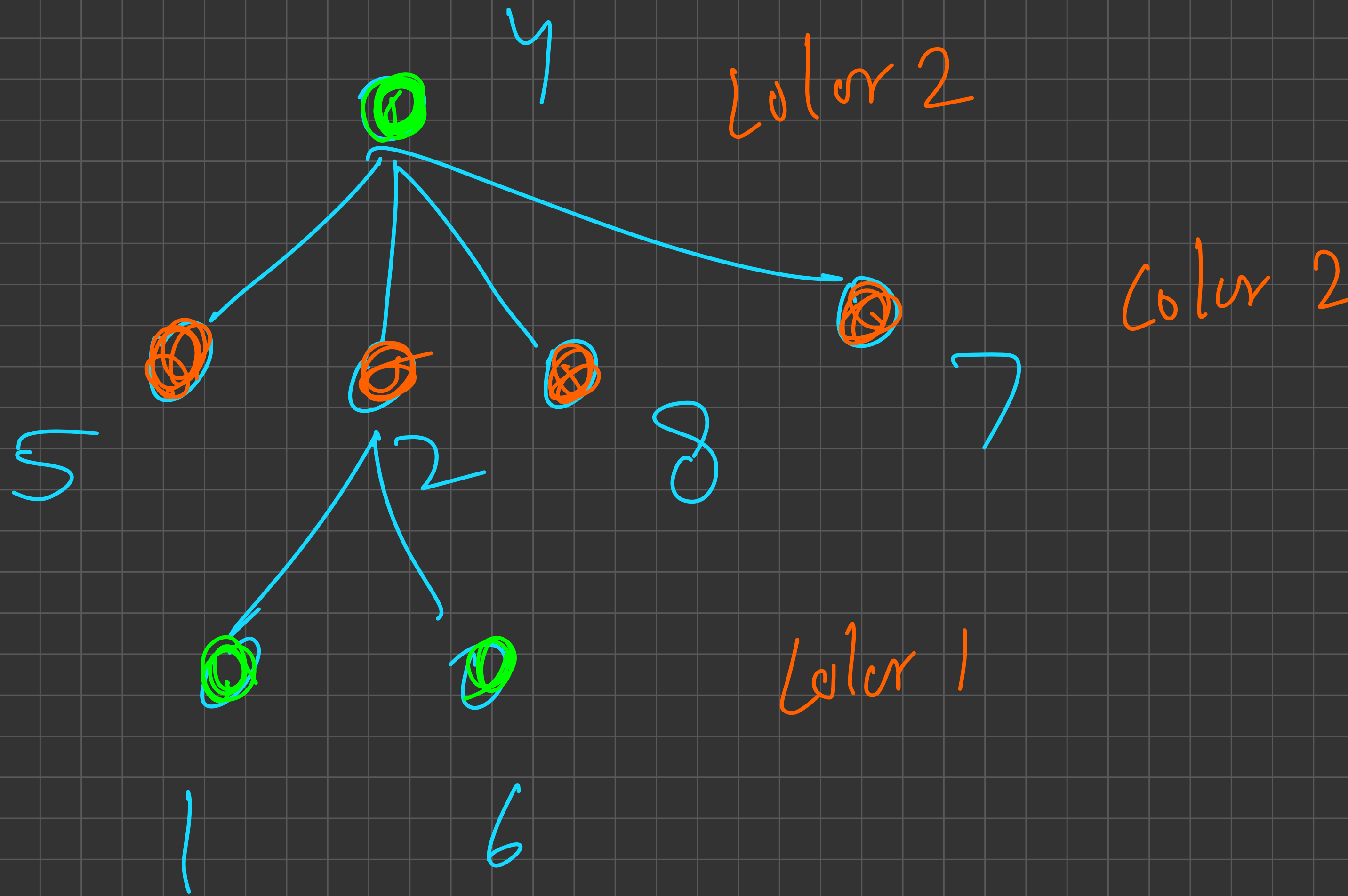
Given a graph color the nodes in a way so that all the neighbours of a node have a different color from the node

Coloring in 2 colours



bipartite colouring





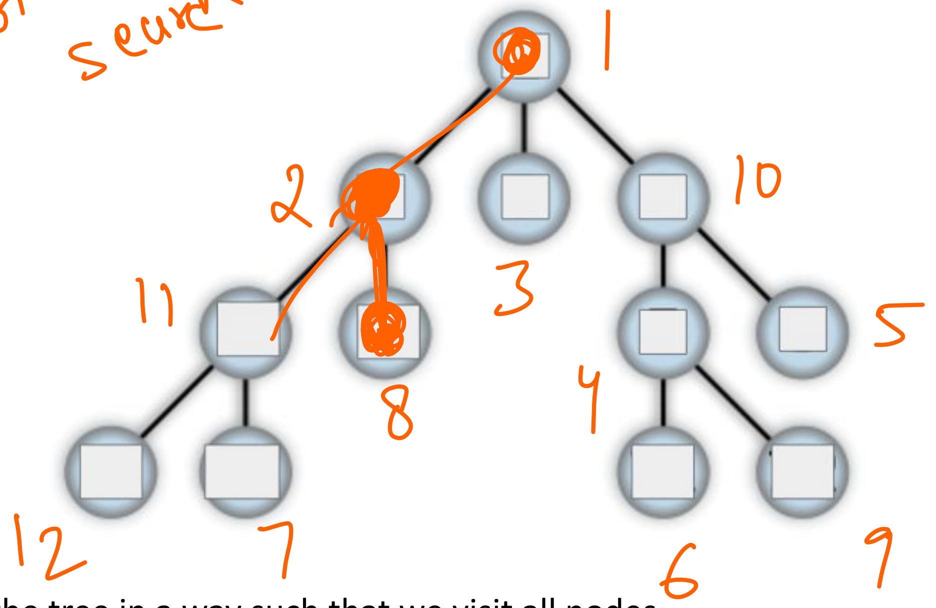
4
Color 2
Color 2
5
2
8
7
Color 1
1
6

Traversals in a Tree

Depth first search

Search for node 6

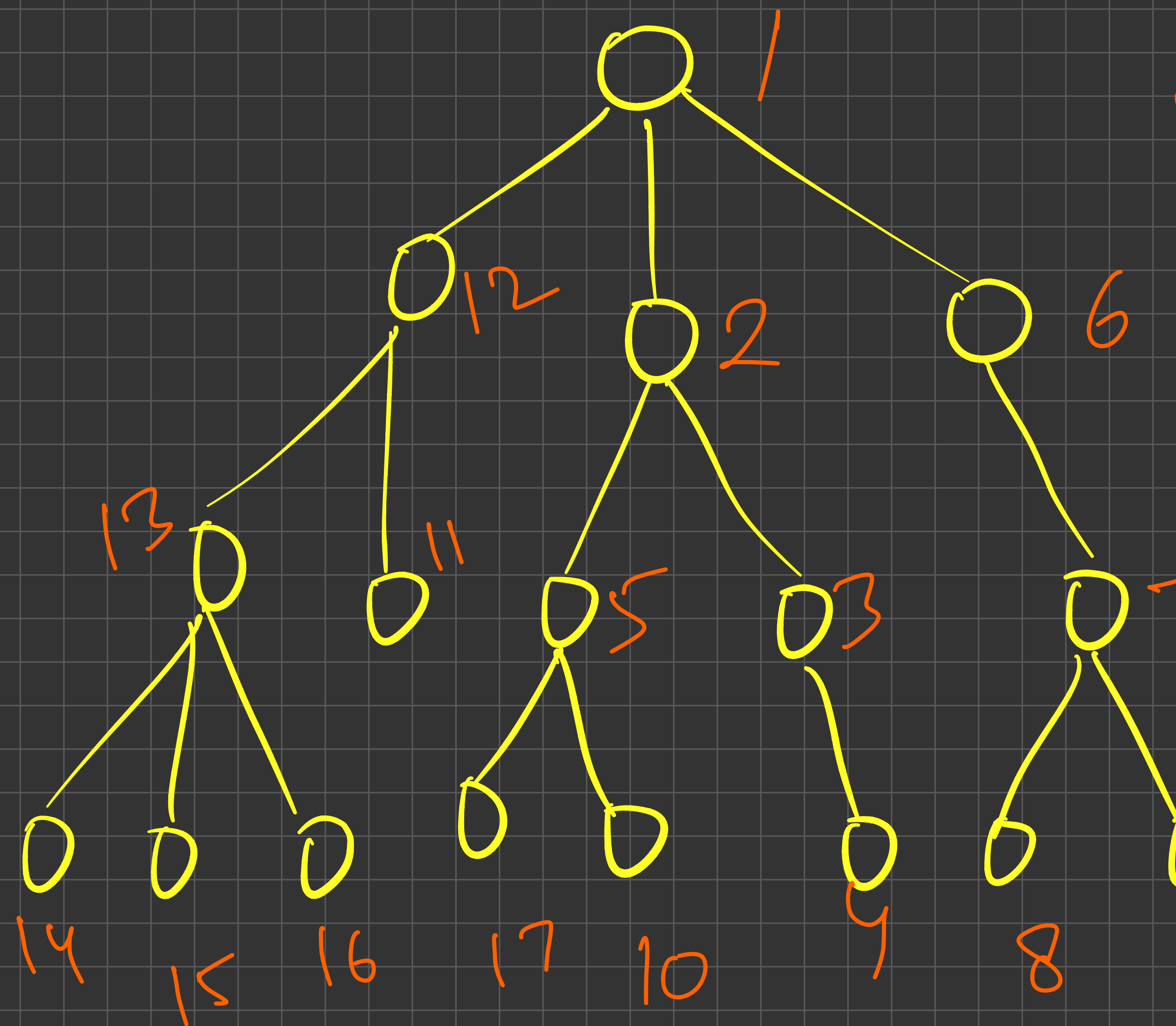
DFS traversal



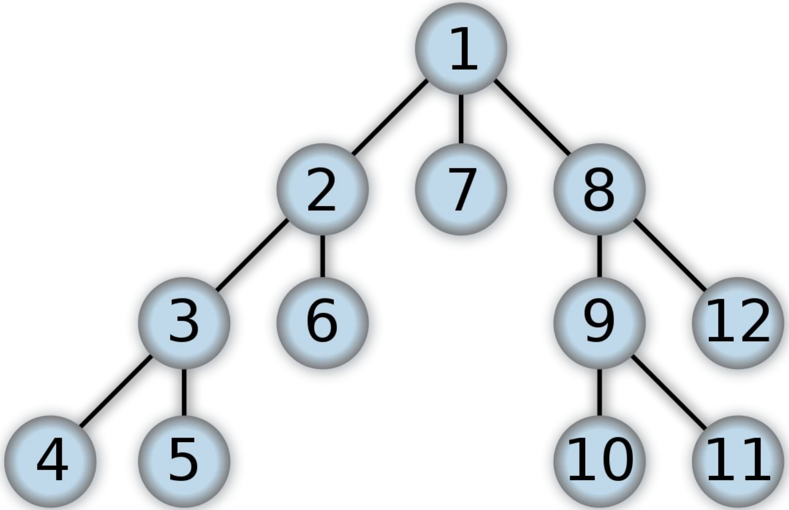
How to traverse the tree in a way such that we visit all nodes

$1 \to \boxed{2 \quad 12 \quad 6}$

$2 \to \boxed{1 \quad 5 \quad 3}$

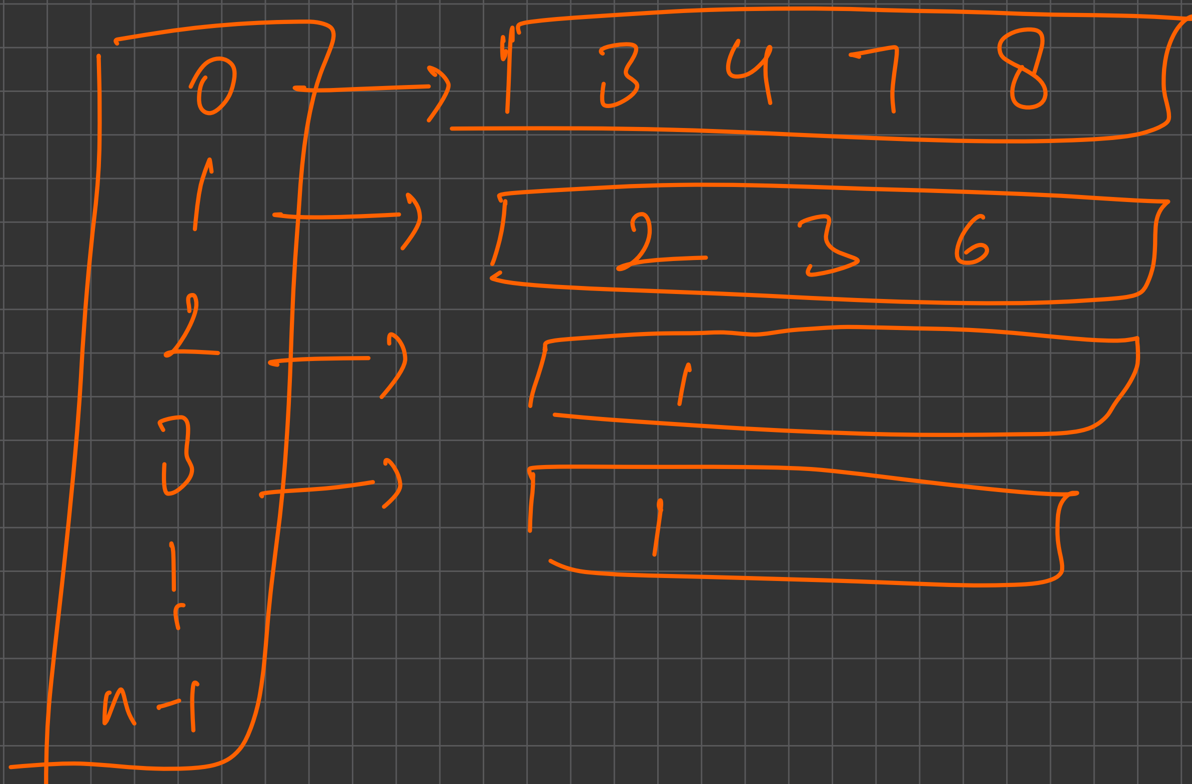


DFS Traversal in a Tree

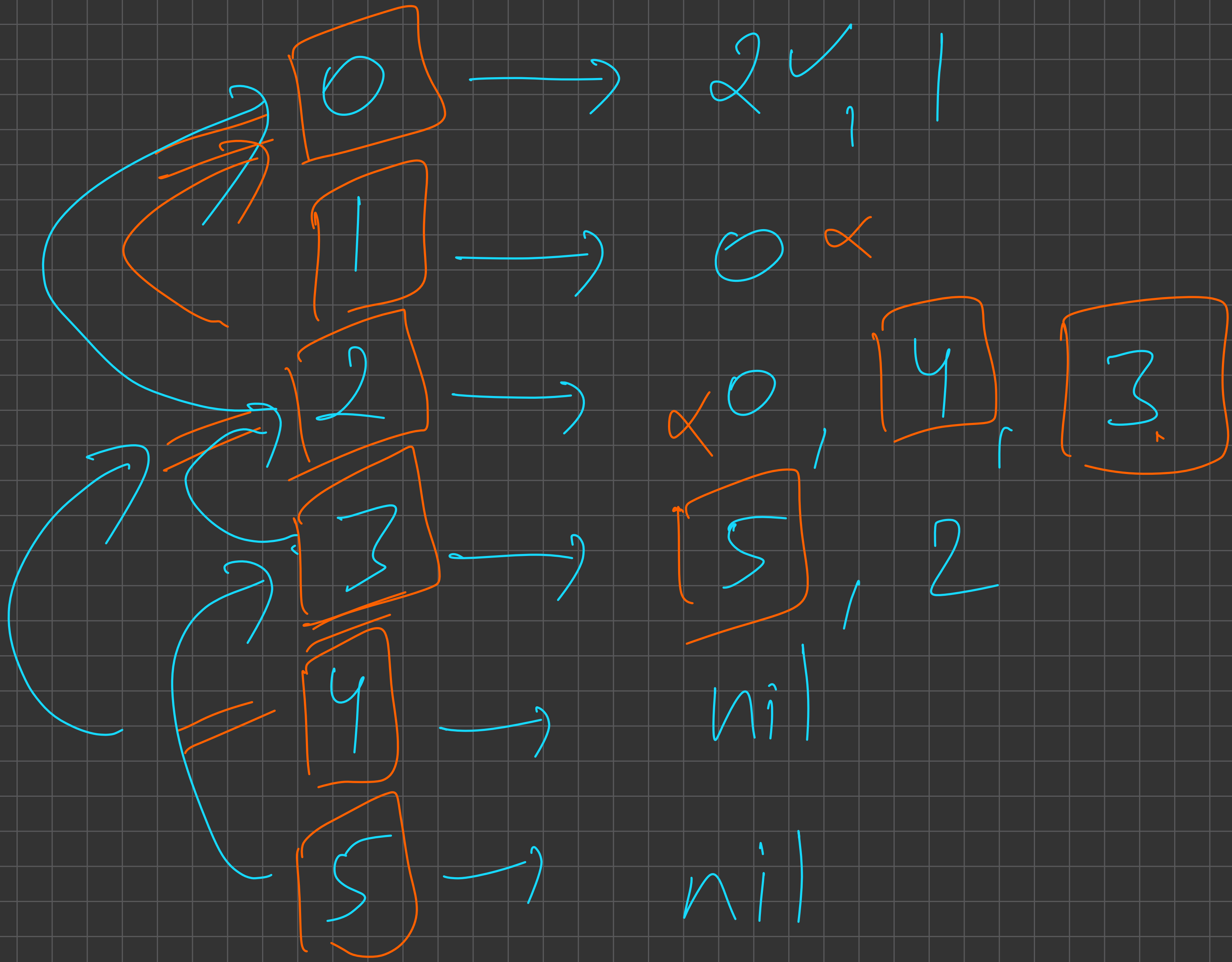


Nodes are numbered in the order in which they are visited

Tree $\Rightarrow$ n nodes

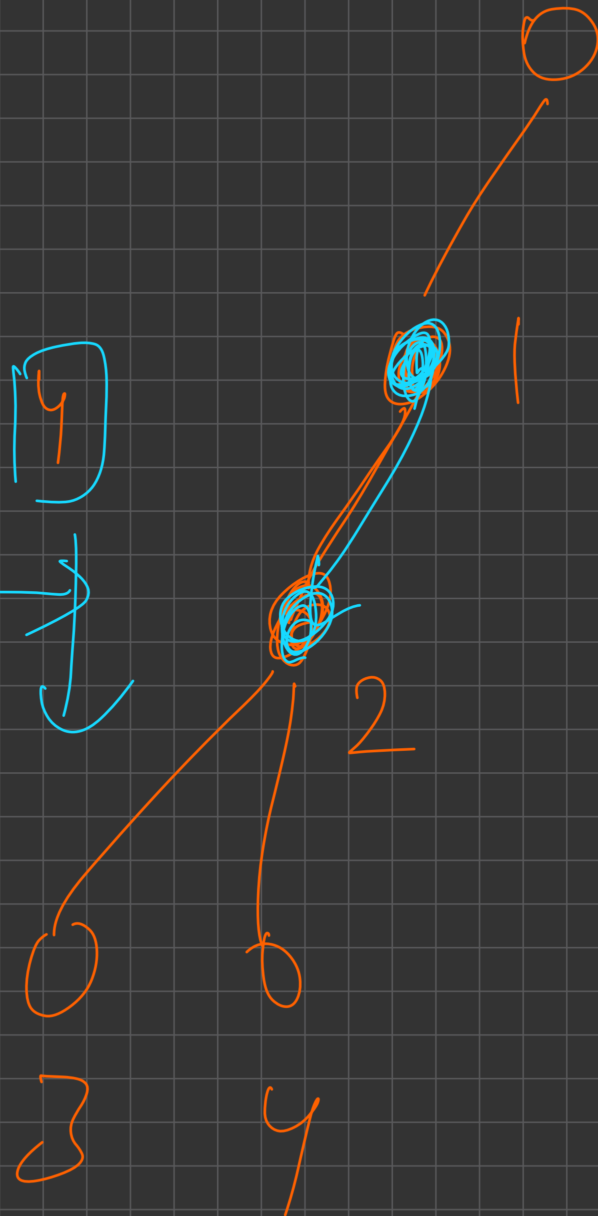


Adjacency list



0 → 2, 1
1 → 0
2 → 0, 4, 3
3 → 5, 2
4 → nil
5 → nil

$2 \rightarrow 1, \boxed{3}, \boxed{4}$



explored

DFS Traversal in a Tree

Implementation:

```
void dfs(int currentNode, vector<vector<int>>& adj, int parent, vector<int>& ans){
    ans.push_back(currentNode);
    v[illegible]
    for(int neighbour : adj[illegible]{
        if(neighbour != parent)
            dfs(neighbour, adj, currentNode, ans);
    }
}
void solve(){
    int n;
    vector<vector<int>> adj(n);
    for(int i = 0; i < n - 1; i++){
        int u, v;
        cin >> u >> v;
        u--, v--;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int root = 0;
    vector<int> dfs_traversal;
    dfs(0, adj, -1, dfs_traversal);
}
```

[currentNode] {

for every edge we are storing 2 integer

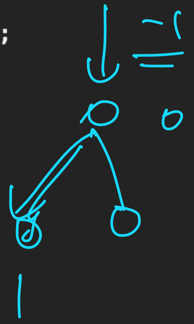
u, v [2·E]

= 2(n-1)

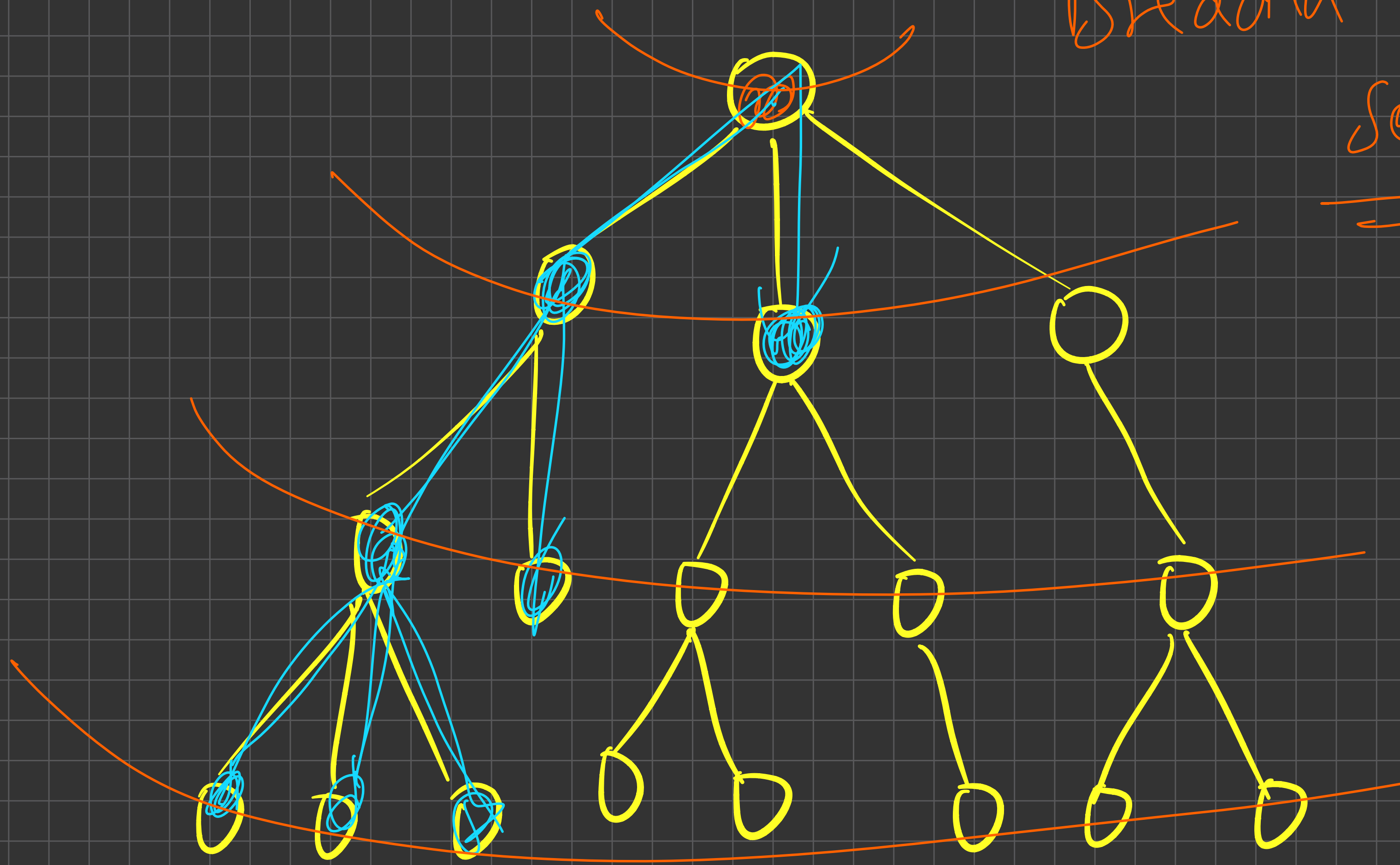
= O(n)

1 10 12

-1



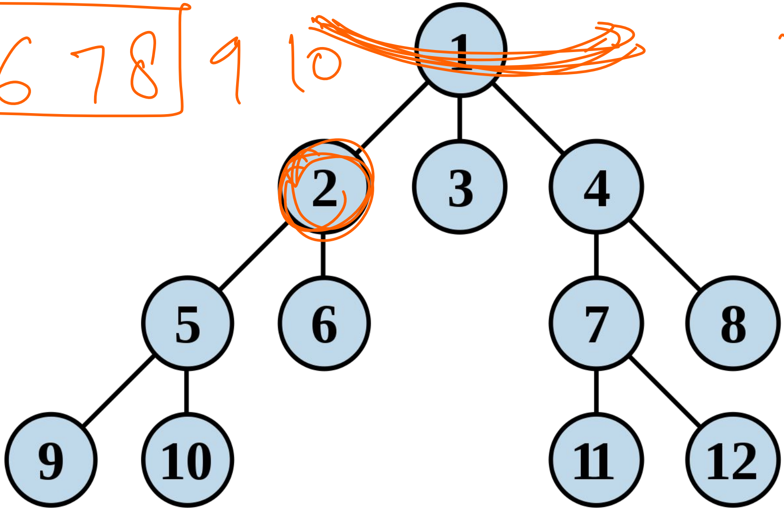
Time Complexity: O(N)



Breadth first search

BFS Traversal in a Tree

6 7 8 9 10



visited array

0, 5, 6

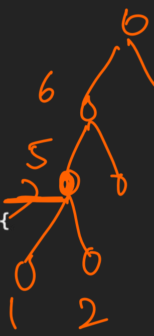
Nodes are numbered in the order in which they are visited

BFS Traversal in a Tree

Implementation:

```
void solve(){
    int n;
    vector<vector<int>> adj(n);
    for(int i = 0; i < n - 1; i++){
        int u, v;
        cin >> u >> v;
        u--, v--;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    int root = 0;
    vector<int> bfs_traversal;
    queue<int> qu;
    vector<bool> visited(n, false);
    qu.push(root);
    visited[root] = true;
    while(!(qu.empty())){
        int currentNode = qu.front();
        qu.pop();
        bfs_traversal.push_back(currentNode);
        for(int neighbour : adj[currentNode]){
            if(!visited[neighbour]){
                visited[neighbour] = true;
                qu.push(neighbour);
            }
        }
    }
}
```

adj. list

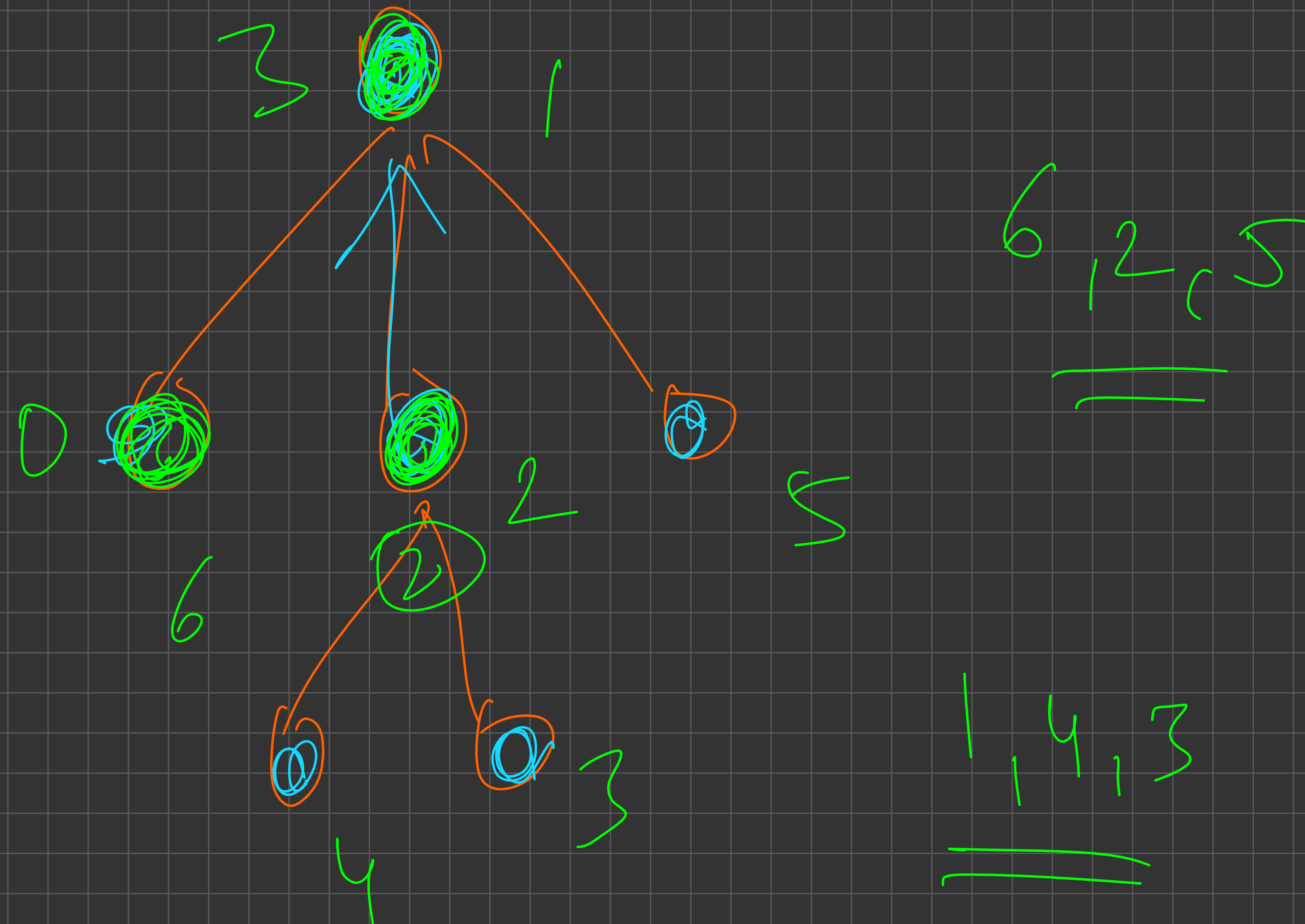


6, 1, 2

Time Complexity: O(N)

Problems on Traversals

- Level of each node
- Storing the parent of each node
- Finding the number of children of each node
- Finding the subtree size of each node, number of leaf nodes
- Finding the diameter



for root |children| = size of adjacency list

o/w children = size of adj list − 1