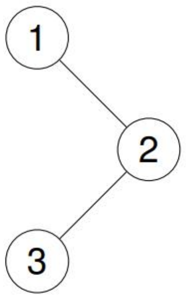
Name:- Kuldeep
UID:- 22BCS10071
Section:- KPIT_901/A
Day- 6

Very Easy:

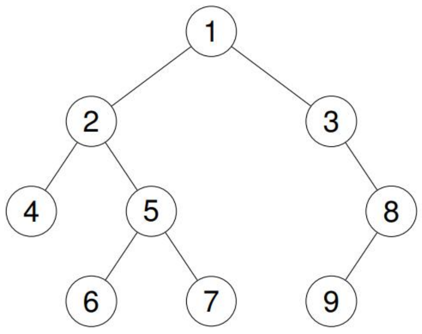
1. Binary Tree Inorder Traversal

Given the root of a binary tree, return the inorder traversal of its nodes' values.

Example 1:
Input: root = [1,null,2,3]
Output: [1,3,2]
Explanation:



Example 2:
Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]
Output: [4,2,6,5,7,1,3,9,8]
Explanation:



Constraints:
The number of nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100

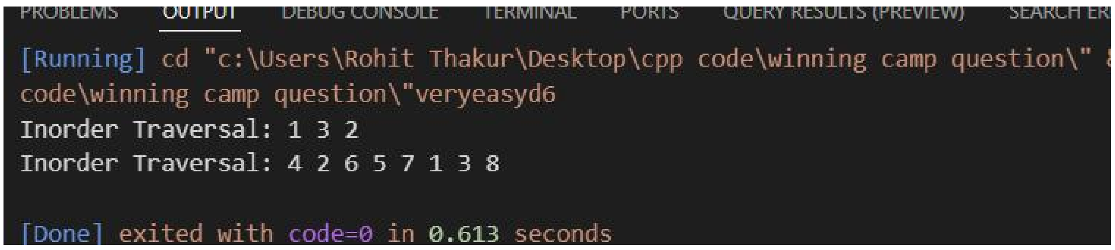
Reference: https://leetcode.com/problems/binary-tree-inorder-traversal/

Code:-

```
#include <iostream>
#include <vector>

using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
};

// Helper function for inorder traversal.
void inorderTraversalHelper(TreeNode* root, vector<int>& result) {
    if (root == nullptr) {
        return;
    }
    // Traverse the left subtree
    inorderTraversalHelper(root->left, result);

    // Visit the current node
    result.push_back(root->val);

    // Traverse the right subtree
    inorderTraversalHelper(root->right, result);
}

// Main function to return the inorder traversal of a binary tree.
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    inorderTraversalHelper(root, result);
    return result;
}

// Helper function to build a binary tree from a vector (example usage).
TreeNode* buildTree(const vector<int>& nodes, int index) {
    if (index >= nodes.size() || nodes[index] == -1) { // -1 represents null.
        return nullptr;
    }
    TreeNode* root = new TreeNode(nodes[index]);
    root->left = buildTree(nodes, 2 * index + 1);
    root->right = buildTree(nodes, 2 * index + 2);
    return root;
}
```

```
// Example usage.
int main() {
    // Example 1
    vector<int> treeNodes1 = {1, -1, 2, -1, -1, 3}; // Represents [1, null, 2, 3]
    TreeNode* root1 = buildTree(treeNodes1, 0);
    vector<int> result1 = inorderTraversal(root1);
    cout << "Inorder Traversal: ";
    for (int val : result1) {
        cout << val << " ";
    }
    cout << endl;

    // Example 2
    vector<int> treeNodes2 = {1, 2, 3, 4, 5, -1, 8, -1, -1, 6, 7, 9}; // Represents [1, 2, 3,
4, 5, null, 8, null, null, 6, 7, 9]
    TreeNode* root2 = buildTree(treeNodes2, 0);
    vector<int> result2 = inorderTraversal(root2);
    cout << "Inorder Traversal: ";
    for (int val : result2) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

Output:-



```
[Running] cd "c:\Users\Rohit Thakur\Desktop\cpp code\winning camp question\" &
code\winning camp question\"veryeasyd6
Inorder Traversal: 1 3 2
Inorder Traversal: 4 2 6 5 7 1 3 8

[Done] exited with code=0 in 0.613 seconds
```

Easy:

1. Same Tree

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:
Input: p = [1,2,3], q = [1,2,3]
Output: true

Example 2:

Input: p = [1,2], q = [1,null,2]
Output: false

Constraints:
The number of nodes in both trees is in the range [0, 100].
-104 <= Node.val <= 104

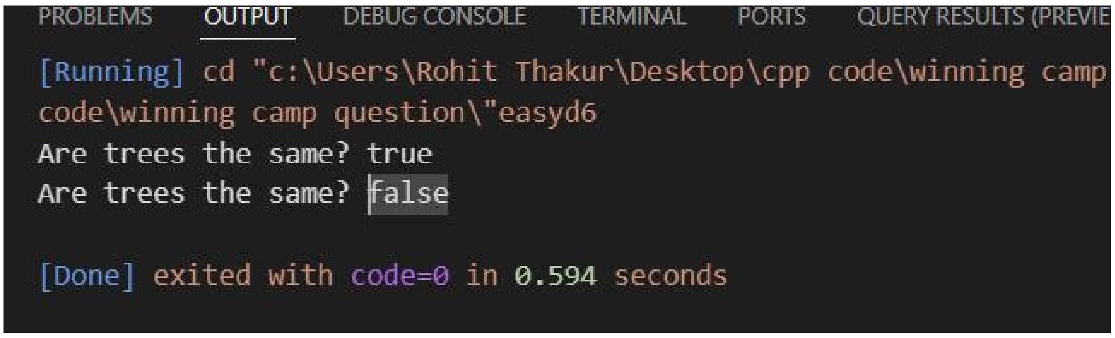
Reference: https://leetcode.com/problems/same-tree/description/?envType=study-plan-v2&envId=top-interview-150

Code:-

```
#include <iostream>
#include <vector>

using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
};

// Function to check if two binary trees are the same.
bool isSameTree(TreeNode* p, TreeNode* q) {
    // Base cases
    if (!p && !q) return true; // Both nodes are null
    if (!p || !q) return false; // One node is null and the other isn't
    if (p->val != q->val) return false; // Nodes have different values

    // Recursively check left and right subtrees
    return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
}

// Helper function to build a binary tree from a vector (example usage).
TreeNode* buildTree(const vector<int>& nodes, int index) {
    if (index >= nodes.size() || nodes[index] == -1) { // -1 represents null.
        return nullptr;
    }
    TreeNode* root = new TreeNode(nodes[index]);
    root->left = buildTree(nodes, 2 * index + 1);
    root->right = buildTree(nodes, 2 * index + 2);
    return root;
}
```

```
// Example usage.
int main() {
    // Example 1
    vector<int> treeNodesP1 = {1, 2, 3};
    vector<int> treeNodesQ1 = {1, 2, 3};
    TreeNode* p1 = buildTree(treeNodesP1, 0);
    TreeNode* q1 = buildTree(treeNodesQ1, 0);
    cout << "Are trees the same? " << (isSameTree(p1, q1) ? "true" : "false") << endl;

    // Example 2
    vector<int> treeNodesP2 = {1, 2};
    vector<int> treeNodesQ2 = {1, -1, 2}; // -1 represents null.
    TreeNode* p2 = buildTree(treeNodesP2, 0);
    TreeNode* q2 = buildTree(treeNodesQ2, 0);
    cout << "Are trees the same? " << (isSameTree(p2, q2) ? "true" : "false") << endl;

    return 0;
}
```

Output:-



```
[Running] cd "c:\Users\Rohit Thakur\Desktop\cpp code\winning camp
code\winning camp question\"easyd6
Are trees the same? true
Are trees the same? false

[Done] exited with code=0 in 0.594 seconds
```

Medium:

1. Construct Binary Tree from Preorder and Inorder Traversal

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Example 1:
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]

Example 2:
Input: preorder = [-1], inorder = [-1]
Output: [-1]

Constraints:

1 <= preorder.length <= 3000
inorder.length == preorder.length
-3000 <= preorder[i], inorder[i] <= 3000
preorder and inorder consist of unique values.
Each value of inorder also appears in preorder.
preorder is guaranteed to be the preorder traversal of the tree.
inorder is guaranteed to be the inorder traversal of the tree.

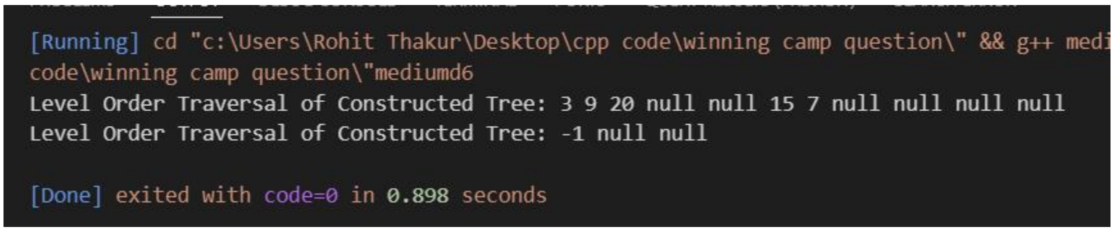
Reference: https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/description/?envType=study-plan-v2&envId=top-interview-150

Code:-

```
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
};

// Helper function to build the tree recursively.
TreeNode* buildTreeHelper(vector<int>& preorder, vector<int>& inorder, int&
preorderIndex, int inorderStart, int inorderEnd, unordered_map<int, int>&
inorderMap) {
    if (inorderStart > inorderEnd) {
        return nullptr;
    }

    // Get the current root value from preorder and increment preorder index
    int rootVal = preorder[preorderIndex++];
    TreeNode* root = new TreeNode(rootVal);

    // Get the root index in inorder traversal
    int rootIndex = inorderMap[rootVal];

    // Build the left and right subtrees
    root->left = buildTreeHelper(preorder, inorder, preorderIndex, inorderStart,
rootIndex - 1, inorderMap);
    root->right = buildTreeHelper(preorder, inorder, preorderIndex, rootIndex + 1,
inorderEnd, inorderMap);
```

```
    return root;
}

// Main function to construct the binary tree.
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    unordered_map<int, int> inorderMap;

    // Create a map to store indices of values in inorder traversal
    for (int i = 0; i < inorder.size(); i++) {
        inorderMap[inorder[i]] = i;
    }

    int preorderIndex = 0;
    return buildTreeHelper(preorder, inorder, preorderIndex, 0, inorder.size() - 1,
inorderMap);
}

// Helper function to print the tree in level order for visualization.
void printLevelOrder(TreeNode* root) {
    if (!root) return;

    vector<TreeNode*> queue = {root};
    while (!queue.empty()) {
        TreeNode* node = queue.front();
        queue.erase(queue.begin());
        if (node) {
            cout << node->val << " ";
            queue.push_back(node->left);
            queue.push_back(node->right);
        } else {
            cout << "null ";
        }
    }
    cout << endl;
}

// Example usage.
int main() {
    // Example 1
    vector<int> preorder1 = {3, 9, 20, 15, 7};
    vector<int> inorder1 = {9, 3, 15, 20, 7};
    TreeNode* root1 = buildTree(preorder1, inorder1);
    cout << "Level Order Traversal of Constructed Tree: ";
    printLevelOrder(root1);

    // Example 2
    vector<int> preorder2 = {-1};
    vector<int> inorder2 = {-1};
    TreeNode* root2 = buildTree(preorder2, inorder2);
    cout << "Level Order Traversal of Constructed Tree: ";
    printLevelOrder(root2);

    return 0;
}
```

Output:-



```
[Running] cd "c:\Users\Rohit Thakur\Desktop\cpp code\winning camp question\" && g++ medi
code\winning camp question\"mediumd6
Level Order Traversal of Constructed Tree: 3 9 20 null null 15 7 null null null null
Level Order Traversal of Constructed Tree: -1 null null

[Done] exited with code=0 in 0.898 seconds
```

Hard :

1. Populating Next Right Pointers in Each Node

Given a binary tree

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Example 1:

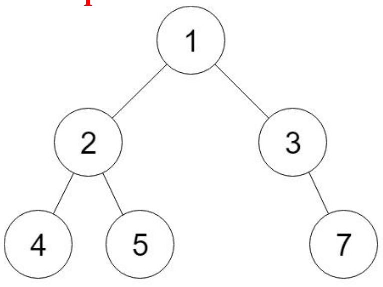


Figure A

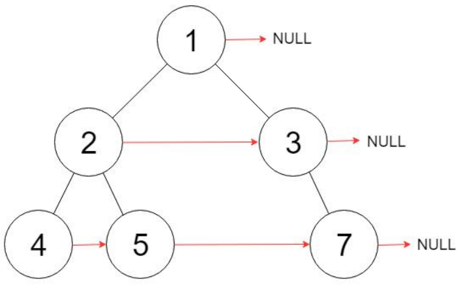


Figure B

Input: root = [1,2,3,4,5,null,7]
Output: [1,#,2,3,#,4,5,7,#]
Explanation: Given the above binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

Example 2:

Input: root = []Output: []

Constraints:
The number of nodes in the tree is in the range [0, 6000].
-100 <= Node.val <= 100

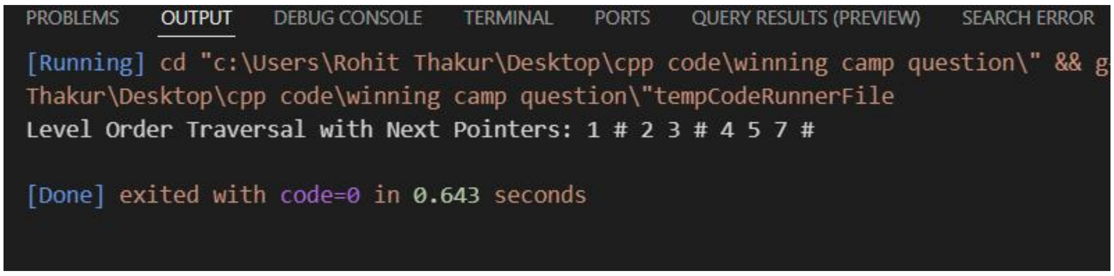
Follow-up:
You may only use constant extra space.
The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

Reference: http://leetcode.com/problems/populating-next-right-pointers-in-each-node

Code:-

```
#include <iostream>
#include <queue>

using namespace std;

// Definition for a binary tree node with a `next` pointer.
struct Node {
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(nullptr), right(nullptr), next(nullptr) {}
    Node(int _val) : val(_val), left(nullptr), right(nullptr), next(nullptr) {}
    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};

// Function to connect next pointers in the tree.
Node* connect(Node* root) {
    if (!root) return nullptr;

    Node* current = root;

    // Outer loop to traverse levels
    while (current) {
        Node* dummy = new Node(0); // Dummy node for the next level
        Node* temp = dummy;

        // Inner loop to traverse nodes within the current level
        while (current) {
            if (current->left) {
                temp->next = current->left;
                temp = temp->next;
            }
            if (current->right) {
                temp->next = current->right;
                temp = temp->next;
            }
            current = current->next; // Move to the next node in the current level
        }

        current = dummy->next; // Move to the next level
        delete dummy; // Clean up the dummy node
    }

    return root;
}

// Helper function to print tree in level order using next pointers.
void printLevelOrder(Node* root) {
    Node* levelStart = root;

    while (levelStart) {
        Node* current = levelStart;
        levelStart = nullptr;

        while (current) {
            cout << current->val << " ";
            if (!levelStart) {
                if (current->left) levelStart = current->left;
                else if (current->right) levelStart = current->right;
            }
            current = current->next;
        }
        cout << "# "; // End of level
    }
    cout << endl;
}

// Example usage.
int main() {
    // Construct the tree from the example: [1,2,3,4,5,null,7]
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->right = new Node(7);

    // Connect the next pointers
    root = connect(root);

    // Print the tree level order traversal using next pointers
    cout << "Level Order Traversal with Next Pointers: ";
    printLevelOrder(root);

    return 0;
}
```

Output:-



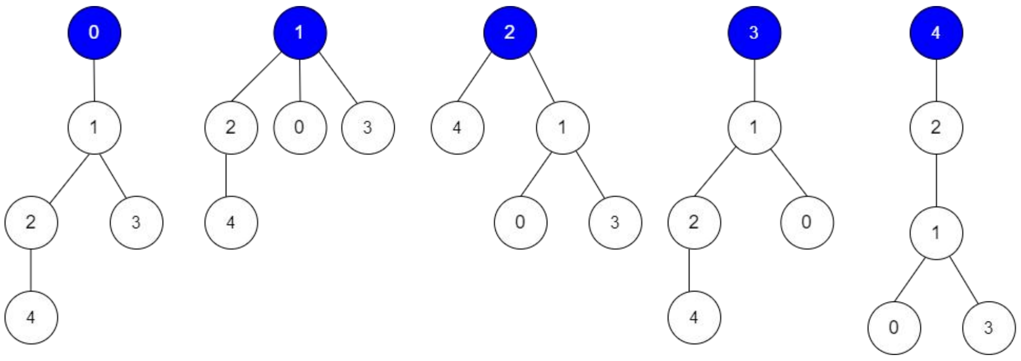
```
[Running] cd "c:\Users\Rohit Thakur\Desktop\cpp code\winning camp question\" && g
Thakur\Desktop\cpp code\winning camp question\"tempCodeRunnerFile
Level Order Traversal with Next Pointers: 1 # 2 3 # 4 5 7 #

[Done] exited with code=0 in 0.643 seconds
```

Very Hard :

1. Count Number of Possible Root Nodes

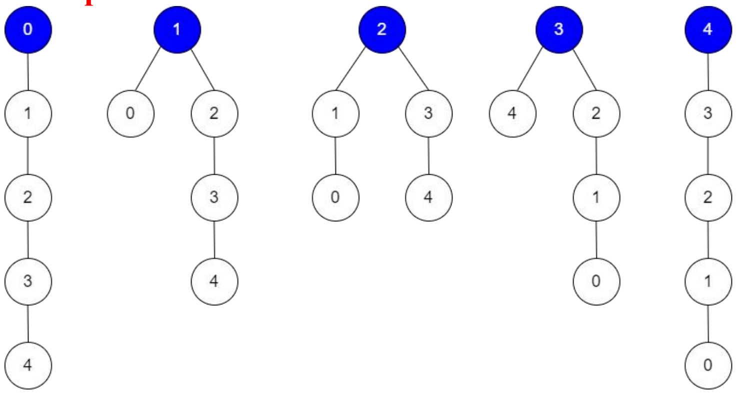
Alice has an undirected tree with n nodes labeled from 0 to n - 1. The tree is represented as a 2D integer array edges of length n - 1 where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the tree.
Alice wants Bob to find the root of the tree. She allows Bob to make several guesses about her tree. In one guess, he does the following:
Chooses two distinct integers u and v such that there exists an edge [u, v] in the tree.
He tells Alice that u is the parent of v in the tree.
Bob's guesses are represented by a 2D integer array guesses where guesses[j] = [uj, vj] indicates Bob guessed uj to be the parent of vj.
Alice being lazy, does not reply to each of Bob's guesses, but just says that at least k of his guesses are true.
Given the 2D integer arrays edges, guesses and the integer k, return the number of possible nodes that can be the root of Alice's tree. If there is no such tree, return 0.

Example 1:



Input: edges = [[0,1],[1,2],[1,3],[4,2]], guesses = [[1,3],[0,1],[1,0],[2,4]], k = 3Output: 3Explanation:
Root = 0, correct guesses = [1,3], [0,1], [2,4]
Root = 1, correct guesses = [1,3], [1,0], [2,4]
Root = 2, correct guesses = [1,3], [1,0], [2,4]
Root = 3, correct guesses = [1,0], [2,4]
Root = 4, correct guesses = [1,3], [1,0]
Considering 0, 1, or 2 as root node leads to 3 correct guesses.

Example 2:



Input: edges = [[0,1],[1,2],[2,3],[3,4]], guesses = [[1,0],[3,4],[2,1],[3,2]], k = 1Output: 5Explanation:
Root = 0, correct guesses = [3,4]
Root = 1, correct guesses = [1,0], [3,4]
Root = 2, correct guesses = [1,0], [2,1], [3,4]
Root = 3, correct guesses = [1,0], [2,1], [3,2], [3,4]
Root = 4, correct guesses = [1,0], [2,1], [3,2]
Considering any node as root will give at least 1 correct guess.

Constraints:
edges.length == n - 1
2 <= n <= 105
1 <= guesses.length <= 105
0 <= ai, bi, uj, vj <= n - 1
ai != bi
uj != vj
edges represents a valid tree.
guesses[j] is an edge of the tree.
guesses is unique.
0 <= k <= guesses.length

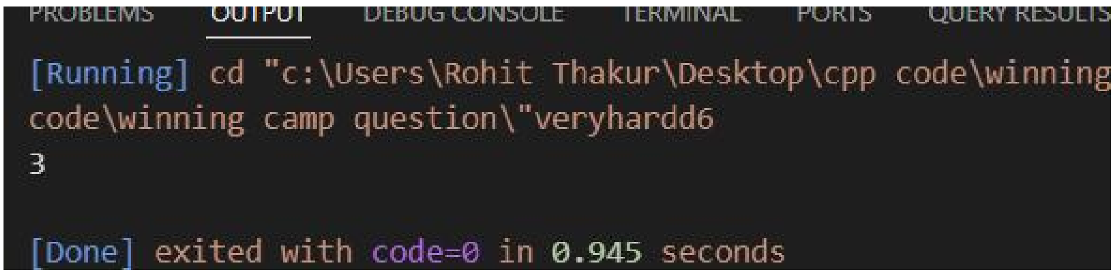
Reference: https://leetcode.com/problems/count-number-of-possible-root-nodes/description/?envType=problem-list-v2&envId=tree

Code:-

```
#include <iostream>
#include <vector>
```

```
#include <unordered_map>
#include <functional>

using namespace std;

class Solution {
public:
    int countValidRoots(int n, vector<vector<int>>& edges, vector<vector<int>>&
guesses, int k) {
        // Step 1: Build the adjacency list (tree structure)
        vector<vector<int>> tree(n);
        for (const auto& edge : edges) {
            tree[edge[0]].push_back(edge[1]);
            tree[edge[1]].push_back(edge[0]);
        }

        // Step 2: Prepare for guess processing
        unordered_map<int, vector<int>> guess_from_parent, guess_to_child;

        // Store guesses where parent-child relationship is guessed
        for (const auto& guess : guesses) {
            guess_from_parent[guess[0]].push_back(guess[1]);
            guess_to_child[guess[1]].push_back(guess[0]);
        }

        int valid_root_count = 0;

        // Step 3: Check each node as the root and calculate valid guesses
        function<int(int, int)> dfs = [&](int node, int parent) -> int {
            int valid_guesses = 0;

            // Check guesses for the current node as the parent
            for (int child : guess_from_parent[node]) {
                if (child != parent) {
                    valid_guesses++;
                }
            }

            // Traverse the tree and calculate guesses recursively
            for (int child : tree[node]) {
                if (child != parent) {
                    valid_guesses += dfs(child, node);
                }
            }

            return valid_guesses;
        };

        // Try every node as the root
        for (int i = 0; i < n; ++i) {
            int valid_guesses = dfs(i, -1);  // DFS from node i with no parent (-1)
            if (valid_guesses >= k) {
                valid_root_count++;
            }
        }

        return valid_root_count;
    }
};

int main() {
    Solution sol;

    // Test case 1
    vector<vector<int>> edges1 = {{0, 1}, {1, 2}, {1, 3}, {4, 2}};
    vector<vector<int>> guesses1 = {{1, 3}, {0, 1}, {1, 0}, {2, 4}};
    int k1 = 3;
    cout << sol.countValidRoots(5, edges1, guesses1, k1) << endl; // Output: 3

    return 0;
}
```

Output:-



```
[Running] cd "c:\Users\Rohit Thakur\Desktop\cpp code\winning
code\winning camp question\"veryhardd6
3

[Done] exited with code=0 in 0.945 seconds
```