DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Gurnoor Oberoi **UID: 22BCS15716**
Branch: BE-CSE::CS201 **Section/Group: 22BCS_FL_IOT-603/B**
Semester: 5th

- **DAY-6 [25-12-2024]**

1. Binary Tree Inorder Traversal *(Very Easy)*

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

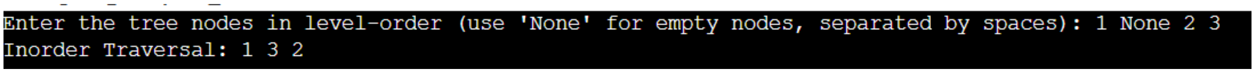
Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <sstream>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
TreeNode* buildTree(const vector<string>& values) {
    if (values.empty() || values[0] == "None") return nullptr;

    TreeNode* root = new TreeNode(stoi(values[0]));
    queue<TreeNode*> q;
    q.push(root);
    int i = 1;
    while (i < values.size()) {
        TreeNode* current = q.front();
        q.pop();
        if (i < values.size() && values[i] != "None") {
            current->left = new TreeNode(stoi(values[i]));
            q.push(current->left);
        }
        i++;
        if (i < values.size() && values[i] != "None") {
            current->right = new TreeNode(stoi(values[i]));
            q.push(current->right);
        }
        i++;
    }
    return root;
}
void inorderTraversal(TreeNode* root, vector<int>& result) {
    if (!root) return;

    inorderTraversal(root->left, result);
    result.push_back(root->val);
    inorderTraversal(root->right, result);
}
int main() {
    cout << "Enter the tree nodes in level-order (use 'None' for empty nodes, separated by spaces): ";
    string input;
    getline(cin, input);
    stringstream ss(input);
    string temp;
    vector<string> values;
    while (ss >> temp) {
        values.push_back(temp);
    }
    TreeNode* root = buildTree(values);
    vector<int> result;
    inorderTraversal(root, result);
    cout << "Inorder Traversal: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}
```

Output:



```
Enter the tree nodes in level-order (use 'None' for empty nodes, separated by spaces): 1 None 2 3
Inorder Traversal: 1 3 2
```

2. Same Tree *(Easy)*

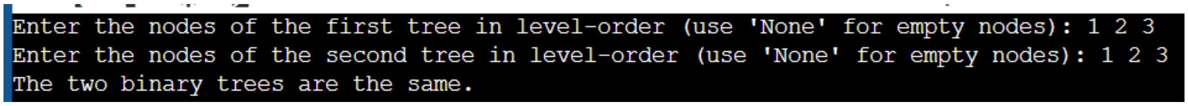
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <sstream>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
TreeNode* buildTree(const vector<string>& values) {
    if (values.empty() || values[0] == "None") return nullptr;

    TreeNode* root = new TreeNode(stoi(values[0]));
    queue<TreeNode*> q;
    q.push(root);
    int i = 1;
    while (i < values.size()) {
        TreeNode* current = q.front();
        q.pop();
        if (i < values.size() && values[i] != "None") {
            current->left = new TreeNode(stoi(values[i]));
            q.push(current->left);
        }
        i++;
        if (i < values.size() && values[i] != "None") {
            current->right = new TreeNode(stoi(values[i]));
            q.push(current->right);
        }
        i++;
    }
    return root;
}
bool isSameTree(TreeNode* p, TreeNode* q) {
    if (!p && !q) return true;
    if (!p || !q) return false;
    return (p->val == q->val) &&
           isSameTree(p->left, q->left) &&
           isSameTree(p->right, q->right);
}
int main() {
    cout << "Enter the nodes of the first tree in level-order (use 'None' for empty nodes): ";
    string input1;
    getline(cin, input1);
    stringstream ss1(input1);
    vector<string> values1;
    string temp;
    while (ss1 >> temp) {
        values1.push_back(temp);
    }
    cout << "Enter the nodes of the second tree in level-order (use 'None' for empty nodes): ";
    string input2;
    getline(cin, input2);
    stringstream ss2(input2);
    vector<string> values2;
    while (ss2 >> temp) {
        values2.push_back(temp);
    }
    TreeNode* tree1 = buildTree(values1);
    TreeNode* tree2 = buildTree(values2);
    if (isSameTree(tree1, tree2)) {
        cout << "The two binary trees are the same." << endl;
    } else {
        cout << "The two binary trees are not the same." << endl;
    }
    return 0;
}
```

Output:



```
Enter the nodes of the first tree in level-order (use 'None' for empty nodes): 1 2 3
Enter the nodes of the second tree in level-order (use 'None' for empty nodes): 1 2 3
The two binary trees are the same.
```

3. Construct Binary Tree from Preorder and Inorder Traversal *(Medium)*

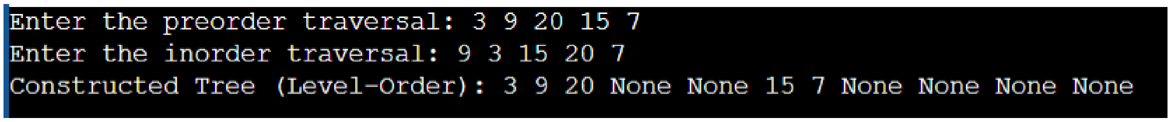
Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <sstream>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        unordered_map<int, int> inorderIndexMap;
        for (int i = 0; i < inorder.size(); i++) {
            inorderIndexMap[inorder[i]] = i;
        }
        int preorderIndex = 0;
        return build(preorder, inorder, preorderIndex, 0, inorder.size() - 1,
inorderIndexMap);
    }
private:
    TreeNode* build(vector<int>& preorder, vector<int>& inorder, int& preorderIndex,
int inorderStart, int inorderEnd, unordered_map<int, int>& inorderIndexMap) {
        if (inorderStart > inorderEnd) return nullptr;
        int rootVal = preorder[preorderIndex++];
        TreeNode* root = new TreeNode(rootVal);
        int inorderIndex = inorderIndexMap[rootVal];
        root->left = build(preorder, inorder, preorderIndex, inorderStart, inorderIndex - 1,
inorderIndexMap);
        root->right = build(preorder, inorder, preorderIndex, inorderIndex + 1, inorderEnd,
inorderIndexMap);
        return root;
    }
};
void printLevelOrder(TreeNode* root) {
    if (!root) {
        cout << "Tree is empty." << endl;
        return;
    }
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        if (current) {
            cout << current->val << " ";
            q.push(current->left);
            q.push(current->right);
        } else {
            cout << "None ";
        }
    }
    cout << endl;
}
int main() {
    cout << "Enter the preorder traversal: ";
    string preorderInput, inorderInput;
    getline(cin, preorderInput);
    cout << "Enter the inorder traversal: ";
    getline(cin, inorderInput);
    vector<int> preorder, inorder;
    stringstream pre(preorderInput), in(inorderInput);
    int val
    while (pre >> val) preorder.push_back(val);
    while (in >> val) inorder.push_back(val);
    Solution solution;
    TreeNode* root = solution.buildTree(preorder, inorder);
    cout << "Constructed Tree (Level-Order): ";
    printLevelOrder(root);

    return 0;
}
```

Output:



```
Enter the preorder traversal: 3 9 20 15 7
Enter the inorder traversal: 9 3 15 20 7
Constructed Tree (Level-Order): 3 9 20 None None 15 7 None None None None
```

4. Populating Next Right Pointers in Each Node *(Hard)*

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.

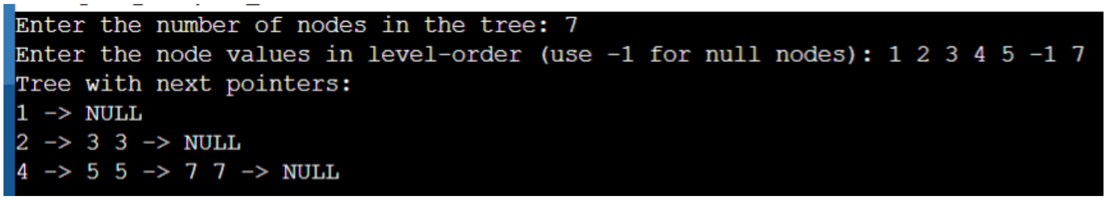
Initially, all next pointers are set to NULL.

Implementation/Code:

```
#include <iostream>
#include <queue>
using namespace std;
```

```
struct Node {
    int val;
    Node* left;
    Node* right;
    Node* next;
    Node() : val(0), left(nullptr), right(nullptr), next(nullptr) {}
    Node(int x) : val(x), left(nullptr), right(nullptr), next(nullptr) {}
    Node(int x, Node* left, Node* right, Node* next) : val(x), left(left), right(right), next(next) {}
};
class Solution {
public:
    Node* connect(Node* root) {
        if (!root) return nullptr;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            Node* prev = nullptr;
            for (int i = 0; i < size; i++) {
                Node* current = q.front();
                q.pop();
                if (prev) {
                    prev->next = current;
                }
                prev = current;
                if (current->left) q.push(current->left);
                if (current->right) q.push(current->right);
            }
            if (prev) {
                prev->next = nullptr;
            }
        }
        return root;
    }
    void printTree(Node* root) {
        Node* level = root;
        while (level) {
            Node* current = level;
            while (current) {
                cout << current->val << " -> ";
                if (current->next) {
                    cout << current->next->val << " ";
                } else {
                    cout << "NULL ";
                }
                current = current->next;
            }
            cout << endl;
            level = level->left;
        }
    }
};
Node* buildTree(vector<int>& nodes) {
    if (nodes.empty() || nodes[0] == -1) return nullptr;
    Node* root = new Node(nodes[0]);
    queue<Node*> q;
    q.push(root);
    int i = 1;
    while (!q.empty() && i < nodes.size()) {
        Node* current = q.front();
        q.pop();
        if (nodes[i] != -1) {
            current->left = new Node(nodes[i]);
            q.push(current->left);
        }
        i++;
        if (i < nodes.size() && nodes[i] != -1) {
            current->right = new Node(nodes[i]);
            q.push(current->right);
        }
        i++;
    }
    return root;
}
int main() {
    Solution solution;
    cout << "Enter the number of nodes in the tree: ";
    int n;
    cin >> n;
    cout << "Enter the node values in level-order (use -1 for null nodes): ";
    vector<int> nodes(n);
    for (int i = 0; i < n; ++i) {
        cin >> nodes[i];
    }
    Node* root = buildTree(nodes);
    solution.connect(root);
    cout << "Tree with next pointers:" << endl;
    solution.printTree(root);
    return 0;
}
```

Output:



```
Enter the number of nodes in the tree: 7
Enter the node values in level-order (use -1 for null nodes): 1 2 3 4 5 -1 7
Tree with next pointers:
1 -> NULL
2 -> 3 3 -> NULL
4 -> 5 5 -> 7 7 -> NULL
```

5. Count Paths That Can Form a Palindrome in a Tree *(Very Hard)*

You are given a tree (i.e. a connected, undirected graph that has no cycles) rooted at node 0 consisting of n nodes numbered from 0 to n - 1. The tree is represented by a 0-indexed array parent of size n, where parent[i] is the parent of node i. Since node 0 is the root, parent[0] == -1.

You are also given a string s of length n, where s[i] is the character assigned to the edge between i and parent[i]. s[0] can be ignored.

Return the number of pairs of nodes (u, v) such that u < v and the characters assigned to edges on the path from u to v can be rearranged to form a palindrome.

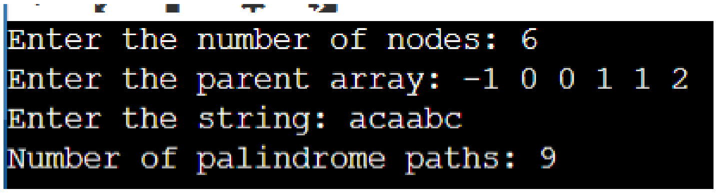
A string is a palindrome when it reads the same backwards as forwards.

Implementation/Code:

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

```
using namespace std;
class Solution {
public:
    long long countPalindromePaths(vector<int>& parent, string s) {
        int n = parent.size();
        for (int i = 1; i < n; ++i) {
            tree[parent[i]].push_back(i);
        }
        unordered_map<int, int> maskCount;
        maskCount[0] = 1;
        long long palindromePairs = 0;
        dfs(0, 0, tree, s, maskCount, palindromePairs);
        return palindromePairs;
    }
private:
    void dfs(int node, int currentMask, vector<vector<int>>& tree, string& s,
             unordered_map<int, int>& maskCount, long long& palindromePairs) {
        currentMask ^= (1 << (s[node] - 'a'));
        palindromePairs += maskCount[currentMask];
        for (int i = 0; i < 26; ++i) {
            int toggledMask = currentMask ^ (1 << i);
            palindromePairs += maskCount[toggledMask];
        }
        maskCount[currentMask]++;
        for (int child : tree[node]) {
            dfs(child, currentMask, tree, s, maskCount, palindromePairs);
        }
        maskCount[currentMask]--;
    }
};
int main() {
    int n;
    cout << "Enter the number of nodes: ";
    cin >> n;
    vector<int> parent(n);
    cout << "Enter the parent array: ";
    for (int i = 0; i < n; ++i) {
        cin >> parent[i];
    }
    string s;
    cout << "Enter the string: ";
    cin >> s;
    Solution solution;
    long long result = solution.countPalindromePaths(parent, s);
    cout << "Number of palindrome paths: " << result << endl;
    return 0;
}
```

Output:



```
Enter the number of nodes: 6
Enter the parent array: -1 0 0 1 1 2
Enter the string: acaabc
Number of palindrome paths: 9
```

6. Binary Tree Preorder Traversal *(Very Easy)*

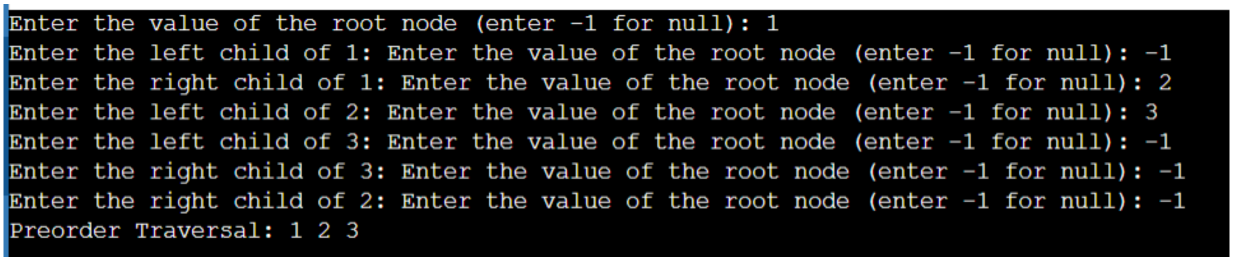
Given the root of a binary tree, return the preorder traversal of its nodes' values.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        if(root == nullptr) return {};
        stack<TreeNode *> s;
        vector<int> v;
        s.push(root);
        while(!s.empty()) {
            TreeNode* curr = s.top();
            s.pop();
            v.push_back(curr->val);
            if(curr->right != nullptr) s.push(curr->right);
            if(curr->left != nullptr) s.push(curr->left);
        }
        return v;
    }
};
TreeNode* createTree() {
    int val;
    cout << "Enter the value of the root node (enter -1 for null): ";
    cin >> val;
    if (val == -1) return nullptr;
    TreeNode* root = new TreeNode(val);
    cout << "Enter the left child of " << val << ": ";
    root->left = createTree();
    cout << "Enter the right child of " << val << ": ";
    root->right = createTree();

    return root;
}
int main() {
    Solution solution;
    TreeNode* root = createTree();
    vector<int> result = solution.preorderTraversal(root);
    cout << "Preorder Traversal: ";
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}
```

Output:



```
Enter the value of the root node (enter -1 for null): 1
Enter the left child of 1: Enter the value of the root node (enter -1 for null): -1
Enter the right child of 1: Enter the value of the root node (enter -1 for null): 2
Enter the left child of 2: Enter the value of the root node (enter -1 for null): 3
Enter the left child of 3: Enter the value of the root node (enter -1 for null): -1
Enter the right child of 3: Enter the value of the root node (enter -1 for null): -1
Enter the right child of 2: Enter the value of the root node (enter -1 for null): -1
Preorder Traversal: 1 2 3
```

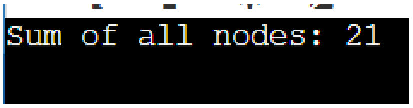
7. Print Odd Numbers up to N *(Very Easy)*

Given the root of a binary tree, you need to find the sum of all the node values in the binary tree.

Implementation/Code:

```
#include <iostream>
using namespace std;
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) {}
};
int sumOfAllNodes(TreeNode* root) {
    if (root == nullptr) return 0;
    return root->val + sumOfAllNodes(root->left) + sumOfAllNodes(root->right);
}
TreeNode* createSampleTree() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(6);
    return root;
}
int main() {
    TreeNode* root = createSampleTree();
    cout << "Sum of all nodes: " << sumOfAllNodes(root) << endl;
    return 0;
}
```

Output:



```
Sum of all nodes: 21
```

8. Inccert Binary Tree *(Easy)*

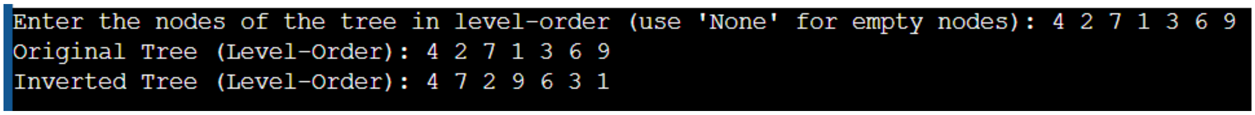
Given the root of a binary tree, invert the tree, and return its root.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <string>
#include <sstream>
using namespace std;
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) {}
};
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (root == NULL) return root;
        swap(root->left, root->right);
        invertTree(root->left);
        invertTree(root->right);
        return root;
    }
};
TreeNode* buildTree(const vector<string>& values) {
    if (values.empty() || values[0] == "None") return nullptr;
    TreeNode* root = new TreeNode(stoi(values[0]));
    queue<TreeNode*> q;
    q.push(root);
    int i = 1;
    while (i < values.size()) {
        TreeNode* current = q.front();
        q.pop();
        if (i < values.size() && values[i] != "None") {
            current->left = new TreeNode(stoi(values[i]));
            q.push(current->left);
        }
        i++;
        if (i < values.size() && values[i] != "None") {
            current->right = new TreeNode(stoi(values[i]));
            q.push(current->right);
        }
        i++;
    }
    return root;
}
void levelOrderTraversal(TreeNode* root) {
    if (!root) {
        cout << "Tree is empty." << endl;
        return;
    }
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        cout << current->val << " ";
        if (current->left) q.push(current->left);
        if (current->right) q.push(current->right);
    }
    cout << endl;
}
int main() {
    cout << "Enter the nodes of the tree in level-order (use 'None' for empty nodes): ";
    string input;
    getline(cin, input);
    stringstream ss(input);
    vector<string> values;
    string temp;
    while (ss >> temp) {
        values.push_back(temp);
    }
    TreeNode* root = buildTree(values);
    cout << "Original Tree (Level-Order): ";
    levelOrderTraversal(root);
    Solution solution;
    root = solution.invertTree(root);
    cout << "Inverted Tree (Level-Order): ";
    levelOrderTraversal(root);

    return 0;
}
```

Output:



```
Enter the nodes of the tree in level-order (use 'None' for empty nodes): 4 2 7 1 3 6 9
Original Tree (Level-Order): 4 2 7 1 3 6 9
Inverted Tree (Level-Order): 4 7 2 9 6 3 1
```

9. Path Sum *(Easy)*

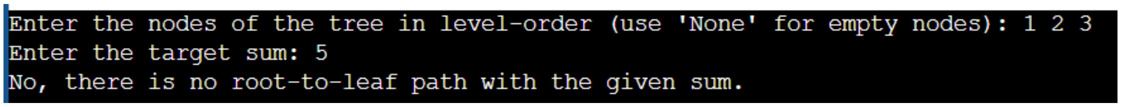
Given a binary tree and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found.

Implementation/Code:

```
#include <iostream>
#include <queue>
#include <vector>
#include <sstream>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if (!root) return false;
        targetSum -= root->val;
        if (!root->left && !root->right) {
            return targetSum == 0;
        }
        return hasPathSum(root->left, targetSum) || hasPathSum(root->right, targetSum);
    }
};
TreeNode* buildTree(const vector<string>& values) {
    if (values.empty() || values[0] == "None") return nullptr;
    TreeNode* root = new TreeNode(stoi(values[0]));
    queue<TreeNode*> q;
    q.push(root);
    int i = 1;
    while (i < values.size()) {
        TreeNode* current = q.front();
        q.pop();
        if (i < values.size() && values[i] != "None") {
            current->left = new TreeNode(stoi(values[i]));
            q.push(current->left);
        }
        i++;
        if (i < values.size() && values[i] != "None") {
            current->right = new TreeNode(stoi(values[i]));
            q.push(current->right);
        }
        i++;
    }
    return root;
}
int main() {
    cout << "Enter the nodes of the tree in level-order (use 'None' for empty nodes): ";
    string input;
    getline(cin, input);
    stringstream ss(input);
    vector<string> values;
    string temp;

    while (ss >> temp) {
        values.push_back(temp);
    }
    TreeNode* root = buildTree(values);
    cout << "Enter the target sum: ";
    int targetSum;
    cin >> targetSum;
    Solution solution;
    if (solution.hasPathSum(root, targetSum)) {
        cout << "Yes, there is a root-to-leaf path with the given sum." << endl;
    } else {
        cout << "No, there is no root-to-leaf path with the given sum." << endl;
    }
    return 0;
}
```

Output:



```
Enter the nodes of the tree in level-order (use 'None' for empty nodes): 1 2 3
Enter the target sum: 5
No, there is no root-to-leaf path with the given sum.
```

10. Construct Binary Tree from Inorder and Postorder Traversal*(Medium)*

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

Implementation/Code:

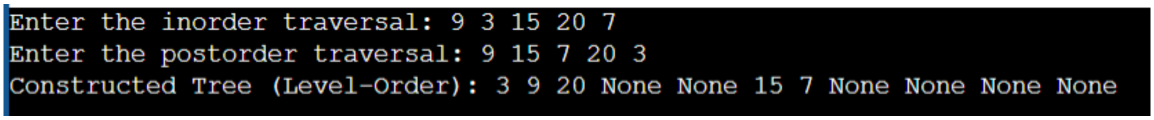
```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <sstream>
using namespace std;
```

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
class Solution {
public:
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        unordered_map<int, int> inorderIndexMap;
        for (int i = 0; i < inorder.size(); i++) {
            inorderIndexMap[inorder[i]] = i;
        }
        int postorderIndex = postorder.size() - 1;
        return build(inorder, postorder, postorderIndex, 0, inorder.size() - 1,
inorderIndexMap);
    }

private:
    TreeNode* build(vector<int>& inorder, vector<int>& postorder, int& postorderIndex,
int inorderStart, int inorderEnd, unordered_map<int, int>& inorderIndexMap) {
        if (inorderStart > inorderEnd) return nullptr;
        int rootVal = postorder[postorderIndex--];
        TreeNode* root = new TreeNode(rootVal);
        int inorderIndex = inorderIndexMap[rootVal];
        root->right = build(inorder, postorder, postorderIndex, inorderIndex + 1,
inorderEnd, inorderIndexMap);
        root->left = build(inorder, postorder, postorderIndex, inorderStart, inorderIndex - 1,
inorderIndexMap);

        return root;
    }
};
void printLevelOrder(TreeNode* root) {
    if (!root) {
        cout << "Tree is empty." << endl;
        return;
    }
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();
        if (current) {
            cout << current->val << " ";
            q.push(current->left);
            q.push(current->right);
        } else {
            cout << "None ";
        }
    }
    cout << endl;
}
int main() {
    cout << "Enter the inorder traversal: ";
    string inorderInput, postorderInput;
    getline(cin, inorderInput);
    cout << "Enter the postorder traversal: ";
    getline(cin, postorderInput);
    vector<int> inorder, postorder;
    stringstream in(inorderInput), post(postorderInput);
    int val;
    while (in >> val) inorder.push_back(val);
    while (post >> val) postorder.push_back(val);
    Solution solution;
    TreeNode* root = solution.buildTree(inorder, postorder);
    cout << "Constructed Tree (Level-Order): ";
    printLevelOrder(root);
    return 0;
}
```

Output:



```
Enter the inorder traversal: 9 3 15 20 7
Enter the postorder traversal: 9 15 7 20 3
Constructed Tree (Level-Order): 3 9 20 None None 15 7 None None None None
```

11. Lowest Common Ancestor of a Binary Tree *(Medium)*

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).

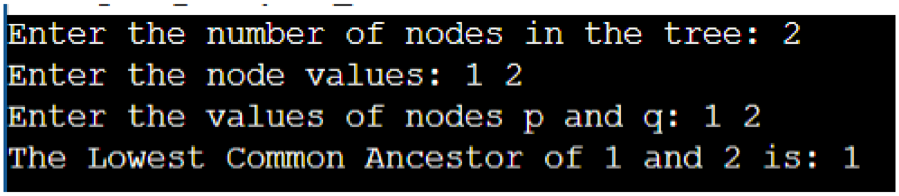
Implementation/Code:

```
#include <iostream>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root || root == p || root == q) {
            return root;
        }
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        if (left && right) {
            return root;
        }
        return left ? left : right;
    }
};
TreeNode* insertNode(TreeNode* root, int value) {
    if (!root) return new TreeNode(value);
    if (value < root->val) root->left = insertNode(root->left, value);
    else root->right = insertNode(root->right, value);
    return root;
}
TreeNode* findNode(TreeNode* root, int value) {
    if (!root || root->val == value) return root;
    if (value < root->val) return findNode(root->left, value);
    return findNode(root->right, value);
}
int main() {
    Solution solution;
    TreeNode* root = nullptr;
    cout << "Enter the number of nodes in the tree: ";
    int n;
    cin >> n;
    cout << "Enter the node values: ";
    for (int i = 0; i < n; i++) {
        int value;
        cin >> value;
        root = insertNode(root, value);
    }
    cout << "Enter the values of nodes p and q: ";
    int pVal, qVal;
    cin >> pVal >> qVal;
    TreeNode* p = findNode(root, pVal);
    TreeNode* q = findNode(root, qVal);
    if (!p || !q) {
        cout << "One or both nodes not found in the tree." << endl;
        return 0;
    }
    TreeNode* lca = solution.lowestCommonAncestor(root, p, q);
    if (lca) {
        cout << "The Lowest Common Ancestor of " << pVal << " and " << qVal << " is: "
<< lca->val << endl;
    } else {
        cout << "No common ancestor found." << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of nodes in the tree: 2
Enter the node values: 1 2
Enter the values of nodes p and q: 1 2
The Lowest Common Ancestor of 1 and 2 is: 1
```

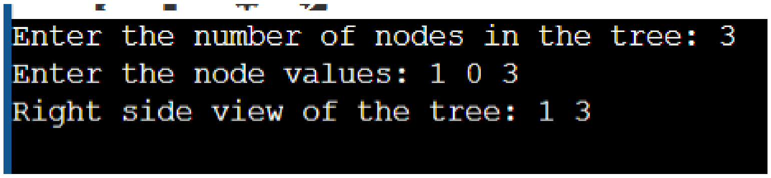
12. Binary Tree Right Side View *(Hard)*

Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        vector<int> result;
        if (!root) return result;
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int levelSize = q.size();
            int rightmostValue = 0;
            for (int i = 0; i < levelSize; i++) {
                TreeNode* current = q.front();
                q.pop();
                if (i == levelSize - 1) {
                    rightmostValue = current->val;
                }
                if (current->left) q.push(current->left);
                if (current->right) q.push(current->right);
            }
            result.push_back(rightmostValue);
        }
        return result;
    }
};
TreeNode* insertNode(TreeNode* root, int value) {
    if (!root) return new TreeNode(value);
    if (value < root->val) root->left = insertNode(root->left, value);
    else root->right = insertNode(root->right, value);
    return root;
}
int main() {
    Solution solution;
    TreeNode* root = nullptr;
    cout << "Enter the number of nodes in the tree: ";
    int n;
    cin >> n;
    cout << "Enter the node values: ";
    for (int i = 0; i < n; i++) {
        int value;
        cin >> value;
        root = insertNode(root, value);
    }
    vector<int> rightSide = solution.rightSideView(root);
    cout << "Right side view of the tree: ";
    for (int val : rightSide) {
        cout << val << " ";
    }
    cout << endl;
    return 0;
}
```

Output:



```
Enter the number of nodes in the tree: 3
Enter the node values: 1 0 3
Right side view of the tree: 1 3
```

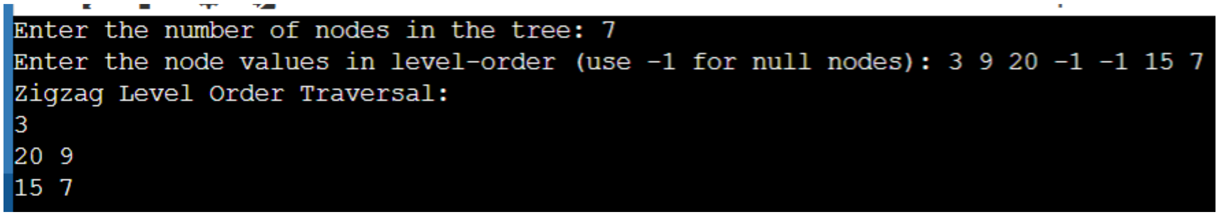
13. Binary Tree Zigzag Level Order Traversal *(Hard)*

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;
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) {}
};
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        queue<TreeNode*> q;
        if (root == NULL) {
            return ans;
        }
        q.push(root);
        while (!q.empty()) {
            int n = q.size();
            vector<int> level;
            for (int i = 0; i < n; ++i) {
                TreeNode* node = q.front();
                q.pop();
                if (node->left != NULL) {
                    q.push(node->left);
                }
                if (node->right != NULL) {
                    q.push(node->right);
                }
                level.push_back(node->val);
            }
            ans.push_back(level);
        }
        return ans;
    }
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans = levelOrder(root);
        int n = ans.size();
        for (int i = 0; i < n; ++i) {
            if (i % 2 == 1) {
                reverse(ans[i].begin(), ans[i].end());
            }
        }
        return ans;
    }
};
TreeNode* buildTree(vector<int>& nodes) {
    if (nodes.empty() || nodes[0] == -1) return nullptr;
    TreeNode* root = new TreeNode(nodes[0]);
    queue<TreeNode*> q;
    q.push(root);
    int i = 1;
    while (!q.empty() && i < nodes.size()) {
        TreeNode* current = q.front();
        q.pop();
        if (nodes[i] != -1) {
            current->left = new TreeNode(nodes[i]);
            q.push(current->left);
        }
        i++;
        if (i < nodes.size() && nodes[i] != -1) {
            current->right = new TreeNode(nodes[i]);
            q.push(current->right);
        }
        i++;
    }
    return root;
}
int main() {
    Solution solution;
    cout << "Enter the number of nodes in the tree: ";
    int n;
    cin >> n;
    cout << "Enter the node values in level-order (use -1 for null nodes): ";
    vector<int> nodes(n);
    for (int i = 0; i < n; ++i) {
        cin >> nodes[i];
    }
    TreeNode* root = buildTree(nodes);
    vector<vector<int>> zigzagOrder = solution.zigzagLevelOrder(root);
    cout << "Zigzag Level Order Traversal:" << endl;
    for (const auto& level : zigzagOrder) {
        for (int val : level) {
            cout << val << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Output:



```
Enter the number of nodes in the tree: 7
Enter the node values in level-order (use -1 for null nodes): 3 9 20 -1 -1 15 7
Zigzag Level Order Traversal:
3
20 9
15 7
```

14. Maximum Number of K-Divisible Components *(Very Hard)*

There is an undirected tree with n nodes labeled from 0 to n - 1. You are given the integer n and 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.
You are also given a 0-indexed integer array values of length n, where values[i] is the value associated with the ith node, and an integer k.

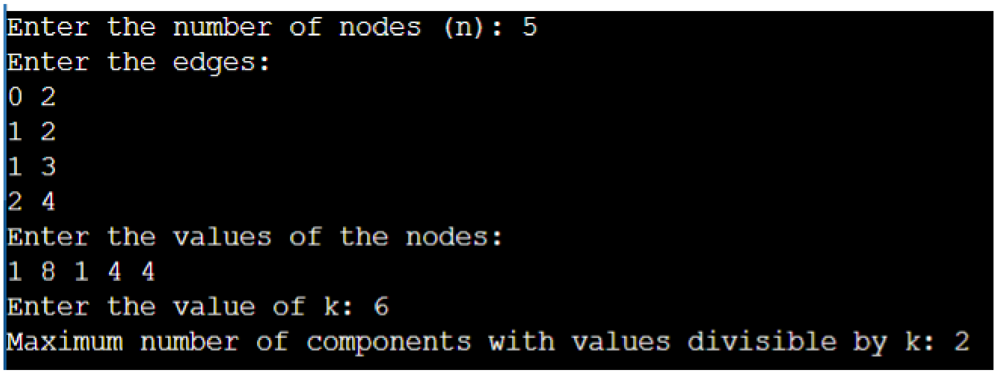
A valid split of the tree is obtained by removing any set of edges, possibly empty, from the tree such that the resulting components all have values that are divisible by k, where the value of a connected component is the sum of the values of its nodes.
Return the maximum number of components in any valid split.

Implementation/Code:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <functional>
using namespace std;
class Solution {
public:
    int maxComponents(int n, vector<vector<int>>& edges, vector<int>& values, int k) {
        vector<vector<int>> adj(n);
        for (const auto& edge : edges) {
            adj[edge[0]].push_back(edge[1]);
            adj[edge[1]].push_back(edge[0]);
        }
        int result = 0;
        function<int(int, int)> dfs = [&](int node, int parent) -> int {
            int subtreeSum = values[node];
            for (int neighbor : adj[node]) {
                if (neighbor != parent) {
                    subtreeSum += dfs(neighbor, node);
                }
            }
            if (subtreeSum % k == 0) {
                result++;
                return 0;
            }
            return subtreeSum;
        };
        dfs(0, -1);
        return result;
    }
};
int main() {
    int n, k;
    cout << "Enter the number of nodes (n): ";
    cin >> n;
    vector<vector<int>> edges(n - 1);
    cout << "Enter the edges: \n";
    for (int i = 0; i < n - 1; i++) {
        int u, v;
        cin >> u >> v;
        edges[i] = {u, v};
    }
    vector<int> values(n);
    cout << "Enter the values of the nodes: \n";
    for (int i = 0; i < n; i++) {
        cin >> values[i];
    }
    cout << "Enter the value of k: ";
    cin >> k;
    Solution solution;
    int result = solution.maxComponents(n, edges, values, k);
    cout << "Maximum number of components with values divisible by k: " << result <<
endl;

    return 0;
}
```

Output:



```
Enter the number of nodes (n): 5
Enter the edges:
0 2
1 2
1 3
2 4
Enter the values of the nodes:
1 8 1 4 4
Enter the value of k: 6
Maximum number of components with values divisible by k: 2
```

15. Count Number of Possible Root Nodes *(Very Hard)*

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.

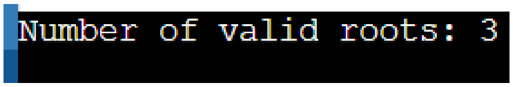
Implementation/Code:

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <functional>
using namespace std;
class Solution {
public:
   int rootCount(int n, vector<vector<int>>& edges, vector<vector<int>>& guesses, int
k) {
      vector<vector<int>> adj(n);
      for (const auto& edge : edges) {
         adj[edge[0]].push_back(edge[1]);
         adj[edge[1]].push_back(edge[0]);
      }
      unordered_set<string> guessSet;
      for (const auto& guess : guesses) {
         string g = to_string(guess[0]) + "-" + to_string(guess[1]);
         guessSet.insert(g);
      }
      int validRootCount = 0;
      for (int root = 0; root < n; root++) {
            int validGuesses = 0;
            vector<int> parent(n, -1);
            dfs(root, -1, adj, parent, guessSet, validGuesses);
            if (validGuesses >= k) {
                validRootCount++;
            }
        }
        return validRootCount;
    }
private:
    void dfs(int node, int parentNode, vector<vector<int>>& adj, vector<int>& parent,
             unordered_set<string>& guessSet, int& validGuesses) {
        parent[node] = parentNode;
        for (int neighbor : adj[node]) {
            if (neighbor != parentNode) {
                if (parent[node] != -1) {
                    string guess1 = to_string(node) + "-" + to_string(neighbor);
                    string guess2 = to_string(neighbor) + "-" + to_string(node);
                    if (guessSet.find(guess1) != guessSet.end() || guessSet.find(guess2) !=
guessSet.end()) {
                        validGuesses++;
                    }
                }

                dfs(neighbor, node, adj, parent, guessSet, validGuesses);
            }
        }
    }
};
int main() {
    Solution solution;
    int n = 5;
    vector<vector<int>> edges = {{0, 1}, {0, 2}, {1, 3}, {1, 4}};
    vector<vector<int>> guesses = {{0, 1}, {0, 2}, {1, 3}};
    int k = 2;
    int result = solution.rootCount(n, edges, guesses, k);
    cout << "Number of valid roots: " << result << endl;
    return 0;
```

```
}
```

Output:



```
Number of valid roots: 3
```