Name: Lakshita UID: 22BCS15284

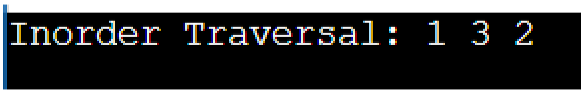
Day: 6 Date: 27-12-24

1. Binary Tree Inorder Traversal

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

```
#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) {}
};

class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        inorderHelper(root, result);
        return result;
    }

private:
    void inorderHelper(TreeNode* node, vector<int>& result) {
        if (!node) return;
        inorderHelper(node->left, result);  // Visit left subtree
        result.push_back(node->val);        // Visit root node
        inorderHelper(node->right, result); // Visit right subtree
    }
};

int main() {
    // Example usage:
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    Solution solution;
    vector<int> inorder = solution.inorderTraversal(root);
```

```
    cout << "Inorder Traversal: ";
    for (int val : inorder) {
        cout << val << " ";
    }

    return 0;
}
```



2. Count Complete Tree Nodes

Given the root of a complete binary tree, return the number of the nodes in the tree.

```
#include <iostream>
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) {}
};

class Solution {
public:
    int countNodes(TreeNode* root) {
        if (!root) return 0;

        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);

        if (leftHeight == rightHeight) {
            // Left subtree is a perfect binary tree
            return (1 << leftHeight) + countNodes(root->right);
        } else {
            // Right subtree is a perfect binary tree
            return (1 << rightHeight) + countNodes(root->left);
        }
    }

private:
    int getHeight(TreeNode* node) {
```

```
        int height = 0;
        while (node) {
            height++;
            node = node->left; // Move down the leftmost path
        }
        return height;
    }
};

int main() {
    // Example usage:
    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->left = new TreeNode(6);

    Solution solution;
    int nodeCount = solution.countNodes(root);

    cout << "Total nodes in the tree: " << nodeCount << endl;

    return 0;
}
```

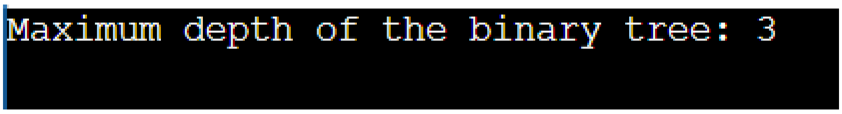
```
Total nodes in the tree: 6
```

3. Binary Tree - Find Maximum Depth

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

```
#include <iostream>
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) {}
};

class Solution {
public:
```

```
    int maxDepth(TreeNode* root) {
        if (!root) return 0; // Base case: empty tree has depth 0

        // Recursive case: find depth of left and right subtrees
        int leftDepth = maxDepth(root->left);
        int rightDepth = maxDepth(root->right);

        // Return the larger depth + 1 for the current node
        return max(leftDepth, rightDepth) + 1;
    }
};

int main() {
    // Example usage:
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(9);
    root->right = new TreeNode(20);
    root->right->left = new TreeNode(15);
    root->right->right = new TreeNode(7);

    Solution solution;
    int depth = solution.maxDepth(root);

    cout << "Maximum depth of the binary tree: " << depth << endl;

    return 0;
}
```

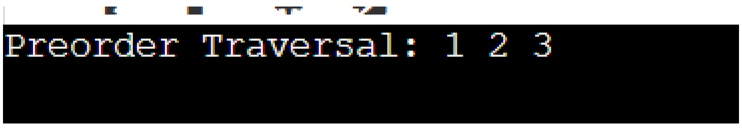


4. Binary Tree Preorder Traversal

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

```
#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) {}
```

```
};

class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> result;
        preorderHelper(root, result);
        return result;
    }

private:
    void preorderHelper(TreeNode* node, vector<int>& result) {
        if (!node) return;
        result.push_back(node->val);       // Visit root node
        preorderHelper(node->left, result);  // Visit left subtree
        preorderHelper(node->right, result); // Visit right subtree
    }
};

int main() {
    // Example usage:
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    Solution solution;
    vector<int> preorder = solution.preorderTraversal(root);

    cout << "Preorder Traversal: ";
    for (int val : preorder) {
        cout << val << " ";
    }

    return 0;
}
```

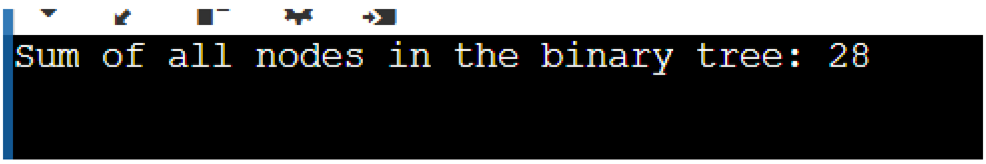


5. Binary Tree - Sum of All Nodes

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

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

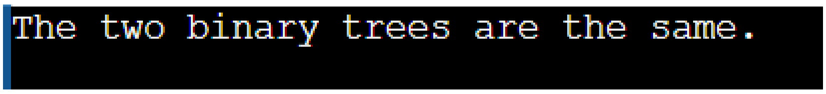
```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
};
class Solution {
public:
    int sumOfNodes(TreeNode* root) {
        if (!root) return 0; // Base case: empty tree has a sum of 0
        int sum = 0;
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            TreeNode* node = q.front();
            q.pop();
            sum += node->val; // Add current node value to sum
            // Push left and right children to the queue if they exist
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
        return sum;
    }};
int main() {
    // Example usage:
    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->left = new TreeNode(6);
    root->right->right = new TreeNode(7);

    Solution solution;
    int sum = solution.sumOfNodes(root);

    cout << "Sum of all nodes in the binary tree: " << sum << endl;

    return 0;
}
```



6. Same Tree

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

```
#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) {}
};
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        // Base cases:
        if (!p && !q) return true;  // Both trees are empty
        if (!p || !q) return false; // One tree is empty
        return (p->val == q->val) && isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }};
int main() {
    // Example usage:
    TreeNode* root1 = new TreeNode(1);
    root1->left = new TreeNode(2);
    root1->right = new TreeNode(3);
    TreeNode* root2 = new TreeNode(1);
    root2->left = new TreeNode(2);
    root2->right = new TreeNode(3);
    Solution solution;
    if (solution.isSameTree(root1, root2)) {
        cout << "The two binary trees are the same." << endl;
    } else {
        cout << "The two binary trees are not the same." << endl;
    }
    return 0;
}
```



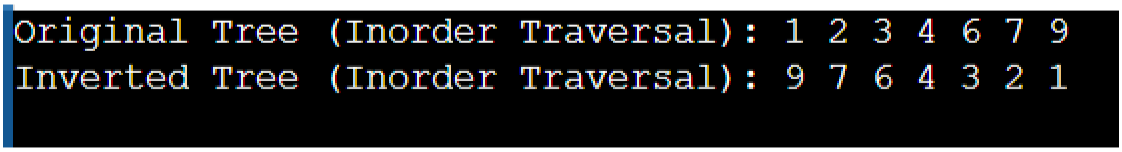
7. Invert Binary Tree

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

```
#include <iostream>
```

```
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) {}
};

class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (!root) return nullptr; // Base case: empty tree

        // Swap the left and right subtrees
        TreeNode* temp = root->left;
        root->left = root->right;
        root->right = temp;

        // Recursively invert the left and right subtrees
        invertTree(root->left);
        invertTree(root->right);

        return root;
    }
};

void printTree(TreeNode* root) {
    if (!root) return;
    printTree(root->left);
    cout << root->val << " ";
    printTree(root->right);
}

int main() {
    // Example usage:
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(7);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(9);

    Solution solution;
```

```
    cout << "Original Tree (Inorder Traversal): ";
    printTree(root);
    cout << endl;

    TreeNode* invertedRoot = solution.invertTree(root);

    cout << "Inverted Tree (Inorder Traversal): ";
    printTree(invertedRoot);
    cout << endl;

    return 0;
}
```



8. 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.

```
#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) {}
};

class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        unordered_map<int, int> inorderMap;
        for (int i = 0; i < inorder.size(); ++i) {
            inorderMap[inorder[i]] = i;
        }
        return buildTreeHelper(preorder, 0, preorder.size() - 1, inorder, 0, inorder.size() - 1,
inorderMap);
    }
```

```
private:
    TreeNode* buildTreeHelper(vector<int>& preorder, int preStart, int preEnd,
                              vector<int>& inorder, int inStart, int inEnd,
                              unordered_map<int, int>& inorderMap) {
        if (preStart > preEnd || inStart > inEnd) return nullptr;

        // The first element of preorder is the root node
        int rootVal = preorder[preStart];
        TreeNode* root = new TreeNode(rootVal);

        // Find the index of the root in the inorder array
        int inRootIndex = inorderMap[rootVal];
        int leftSubtreeSize = inRootIndex - inStart;

        // Recursively build the left and right subtrees
        root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
                                     inorder, inStart, inRootIndex - 1, inorderMap);
        root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
                                      inorder, inRootIndex + 1, inEnd, inorderMap);

        return root;
    }
};
void printInorder(TreeNode* root) {
    if (!root) return;
    printInorder(root->left);
    cout << root->val << " ";
    printInorder(root->right);
}

int main() {
    // Example usage:
    vector<int> preorder = {3, 9, 20, 15, 7};
    vector<int> inorder = {9, 3, 15, 20, 7};
    Solution solution;
    TreeNode* root = solution.buildTree(preorder, inorder);
    cout << "Inorder Traversal of Constructed Tree: ";
    printInorder(root);
    cout << endl;
    return 0;
}
```

```
Inorder Traversal of Constructed Tree: 9 3 15 20 7
```

9. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

```
#include <iostream>
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) {}
};

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root) return nullptr; // Base case: empty tree
        if (root == p || root == q) return root; // If root is one of the nodes, return it

        // Recursively search for p and q in left and right subtrees
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        // If p and q are found in different subtrees, root is the LCA
        if (left && right) return root;

        // Otherwise, return the non-null child (either left or right)
        return left ? left : right;
    }
};

int main() {
    // Example usage:
    TreeNode* root = new TreeNode(3);
    root->left = new TreeNode(5);
    root->right = new TreeNode(1);
    root->left->left = new TreeNode(6);
    root->left->right = new TreeNode(2);
    root->right->left = new TreeNode(0);
    root->right->right = new TreeNode(8);
    root->left->right->left = new TreeNode(7);
    root->left->right->right = new TreeNode(4);

    TreeNode* p = root->left;      // Node with value 5
    TreeNode* q = root->left->right->right; // Node with value 4
```

```
    Solution solution;
    TreeNode* lca = solution.lowestCommonAncestor(root, p, q);

    cout << "Lowest Common Ancestor of " << p->val << " and " << q->val << " is: " << lca->val <<
endl;

    return 0;
}
```

```
Lowest Common Ancestor of 5 and 4 is: 5
```