Check if the Binary Tree is Balanced Binary Tree
Problem Statement: Given a Binary Tree, return true if it is a Balanced Binary Tree else return false. A Binary Tree is balanced if, for all nodes in the tree, the difference between left and right subtree height is not more than 1.

```
#include <iostream>
#include <algorithm>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to check if a binary tree is balanced
    bool isBalanced(Node* root) {
        // If the tree is empty, it's balanced
        if (root == nullptr) {
            return true;
        }

        // Calculate the height of left and right subtrees
        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);

        // Check if the absolute difference in heights
        // of left and right subtrees is <= 1
        if (abs(leftHeight - rightHeight) <= 1 &&
            isBalanced(root->left) &&
            isBalanced(root->right)) {
            return true;
        }

        // If any condition fails, the tree is unbalanced
        return false;
    }

    // Function to calculate the height of a subtree
    int getHeight(Node* root) {
        // Base case: if the current node is NULL,
        // return 0 (height of an empty tree)
        if (root == nullptr) {
            return 0;
        }

        // Recursively calculate the height
        // of left and right subtrees
        int leftHeight = getHeight(root->left);
        int rightHeight = getHeight(root->right);

        // Return the maximum height of left and right subtrees
        // plus 1 (to account for the current node)
        return max(leftHeight, rightHeight) + 1;
    }
};

// Main function
```

```
int main() {
    // Creating a sample binary tree
    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->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);

    // Creating an instance of the Solution class
    Solution solution;

    // Checking if the tree is balanced
    if (solution.isBalanced(root)) {
        cout << "The tree is balanced." << endl;
    } else {
        cout << "The tree is not balanced." << endl;
    }

    return 0;
}



#include <iostream>
#include <vector>
#include <stack>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {

public:
    // Function to check if a binary tree is balanced
    bool isBalanced(Node* root) {
        // Check if the tree's height difference
        // between subtrees is less than 2
        // If not, return false; otherwise, return true
        return dfsHeight(root) != -1;
    }

    // Recursive function to calculate
    // the height of the tree
    int dfsHeight(Node* root) {
        // Base case: if the current node is NULL,
        // return 0 (height of an empty tree)
        if (root == NULL) return 0;

        // Recursively calculate the
        // height of the left subtree
        int leftHeight = dfsHeight(root->left);

        // If the left subtree is unbalanced,
```

```
        // propagate the unbalance status
        if (leftHeight == -1)
            return -1;

        // Recursively calculate the
        // height of the right subtree
        int rightHeight = dfsHeight(root->right);

        // If the right subtree is unbalanced,
        // propagate the unbalance status
        if (rightHeight == -1)
            return -1;

        // Check if the difference in height between
        // left and right subtrees is greater than 1
        // If it's greater, the tree is unbalanced,
        // return -1 to propagate the unbalance status
        if (abs(leftHeight - rightHeight) > 1)
            return -1;

        // Return the maximum height of left and
        // right subtrees, adding 1 for the current node
        return max(leftHeight, rightHeight) + 1;
    }
};
```

```
// Main function
int main() {
    // Creating a sample binary tree
    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->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);

    // Creating an instance of the Solution class
    Solution solution;

    // Checking if the tree is balanced
    if (solution.isBalanced(root)) {
        cout << "The tree is balanced." << endl;
    } else {
        cout << "The tree is not balanced." << endl;
    }

    return 0;
}
```

Calculate the Diameter of a Binary Tree

Problem Statement: Given the root of the Binary Tree, return the length of its diameter. The Diameter of a Binary Tree is the longest distance between any two

nodes of that tree. This path may or may not pass through the root.

```
#include <iostream>
#include <algorithm>

using namespace std;

// Node structure for
// the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Global variable to
    // store the diameter
    int diameter = 0;

    // Function to calculate
    // the height of a subtree
    int calculateHeight(Node* node) {
        if (node == nullptr) {
            return 0;
        }

        // Recursively calculate the
        // height of left and right subtrees
        int leftHeight = calculateHeight(node->left);
        int rightHeight = calculateHeight(node->right);

        // Calculate the diameter at the current
        // node and update the global variable
        diameter = max(diameter, leftHeight + rightHeight);

        // Return the height
        // of the current subtree
        return 1 + max(leftHeight, rightHeight);
    }

    // Function to find the
    // diameter of a binary tree
    int diameterOfBinaryTree(Node* root) {
        // Start the recursive
        // traversal from the root
        calculateHeight(root);

        // Return the maximum diameter
        // found during traversal
        return diameter;
    }
};



// Main function
int main() {
    // Creating a sample binary tree
    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->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);

    // Creating an instance of the Solution class
    Solution solution;

    // Calculate the diameter of the binary tree
    int diameter = solution.diameterOfBinaryTree(root);

    cout << "The diameter of the binary tree is: " << diameter << endl;

    return 0;
}


#include <iostream>
#include <algorithm>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to find the
    // diameter of a binary tree
    int diameterOfBinaryTree(Node* root) {
        // Initialize the variable to
        // store the diameter of the tree
        int diameter = 0;
        // Call the height function to traverse
        // the tree and calculate diameter
        height(root, diameter);
        // Return the calculated diameter
        return diameter;
    }

private:
    // Function to calculate the height of
    // the tree and update the diameter
    int height(Node* node, int& diameter) {
        // Base case: If the node is null,
        // return 0 indicating the
        // height of an empty tree
        if (!node) {
            return 0;
        }

        // Recursively calculate the
        // height of left and right subtrees
        int lh = height(node->left, diameter);
        int rh = height(node->right, diameter);
```

```
        // Update the diameter with the maximum
        // of current diameter or sum of
        // left and right heights
        diameter = max(diameter, lh + rh);

        // Return the height of
        // the current node's subtree
        return 1 + max(lh, rh);
    }
};


// Main function
int main() {
    // Creating a sample binary tree
    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->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);

    // Creating an instance of the Solution class
    Solution solution;

    // Calculate the diameter of the binary tree
    int diameter = solution.diameterOfBinaryTree(root);

    cout << "The diameter of the binary tree is: " << diameter << endl;

    return 0;
}
```

Maximum Sum Path in Binary Tree
Problem Statement: Given a Binary Tree, determine the maximum sum achievable along any path within the tree. A path in a binary tree is defined as a sequence of nodes where each pair of adjacent nodes is connected by an edge. Nodes can only appear once in the sequence, and the path is not required to start from the root. Identify and compute the maximum sum possible along any path within the given binary tree.

```
#include <iostream>
#include <algorithm>
#include <climits>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
```

```
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Recursive function to find the maximum path sum
    // or a given subtree rooted at 'root'
    // The variable 'maxi' is a reference parameter
    // updated to store the maximum path sum encountered
    int findMaxPathSum(Node* root, int &maxi) {
        // Base case: If the current node is null, return 0
        if (root == nullptr) {
            return 0;
        }

        // Calculate the maximum path sum
        // for the left and right subtrees
        int leftMaxPath = max(0, findMaxPathSum(root->left, maxi));
        int rightMaxPath = max(0, findMaxPathSum(root->right, maxi));

        // Update the overall maximum
        // path sum including the current node
        maxi = max(maxi, leftMaxPath + rightMaxPath + root->data);

        // Return the maximum sum considering
        // only one branch (either left or right)
        // along with the current node
        return max(leftMaxPath, rightMaxPath) + root->data;
    }

    // Function to find the maximum
    // path sum for the entire binary tree
    int maxPathSum(Node* root) {
        // Initialize maxi to the
        // minimum possible integer value
        int maxi = INT_MIN;
         // Call the recursive function to
         // find the maximum path sum
        findMaxPathSum(root, maxi);
        // Return the final maximum path sum
        return maxi;
    }
};


int main() {
    // Creating a sample binary tree
    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->left->right->right = new Node(6);
    root->left->right->right->right = new Node(7);

    // Creating an instance of the Solution class
    Solution solution;

    // Finding and printing the maximum path sum
    int maxPathSum = solution.maxPathSum(root);
    cout << "Maximum Path Sum: " << maxPathSum << endl;


    return 0;
```

```
}
```

Output Maximum Path Sum: 24

Check if two trees are identical

Problem Statement: Given two Binary Trees, return if true if the two trees are identical, otherwise return false.

```
#include <iostream>
#include <algorithm>
#include <climits>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to check if two
    // binary trees are identical
    bool isIdentical(Node* node1, Node* node2){
        // If both nodes are NULL,
        // they are identical
        if(node1 == NULL && node2 == NULL){
            return true;
        }
        // If only one of the nodes is
        // NULL, they are not identical
        if( node1== NULL || node2==NULL){
            return false;
        }
        // Check if the current nodes
        // have the same data value
        // and recursively check their
        // left and right subtrees
        return ((node1->data == node2->data)
            && isIdentical(node1->left, node2->left)
                && isIdentical(node1->right, node2->right));
    }
};
```

```
int main() {
    // Node1
    Node* root1 = new Node(1);
    root1->left = new Node(2);
    root1->right = new Node(3);
    root1->left->left = new Node(4);

    // Node2
    Node* root2 = new Node(1);
    root2->left = new Node(2);
    root2->right = new Node(3);
    root2->left->left = new Node(4);

    Solution solution;

    if (solution.isIdentical(root1, root2)) {
        cout << "The binary trees are identical." << endl;
    } else {
        cout << "The binary trees are not identical." << endl;
    }

    return 0;
}
```

Zig Zag Traversal Of Binary Tree

Mark as Completed

84

Problem Statement: Given a Binary Tree, print the zigzag traversal of the Binary Tree. Zigzag traversal of a binary tree is a way of visiting the nodes of the tree in a zigzag pattern, alternating between left-to-right and right-to-left at each level.

```
#include <iostream>
#include <algorithm>
#include <climits>
#include <queue>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
```

```
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to perform zigzag level
    // order traversal of a binary tree
    vector<vector<int>> ZigZagLevelOrder(Node* root){
        // Vector to store the
        // result of zigzag traversal
        vector<vector<int>> result;

        // Check if the root is NULL,
        // return an empty result
        if(root == NULL){
            return result;
        }

        // Queue to perform
        // level order traversal
        queue<Node*> nodesQueue;
        nodesQueue.push(root);

        // Flag to determine the direction of
        // traversal (left to right or right to left)
        bool leftToRight = true;

        // Continue traversal until
        // the queue is empty
        while(!nodesQueue.empty()){
            // Get the number of nodes
            // at the current level
            int size = nodesQueue.size();

            // Vector to store the values
            // of nodes at the current level
            vector<int> row(size);

            // Traverse nodes at
            // the current level
            for(int i = 0; i < size; i++){
                // Get the front node
                // from the queue
                Node* node = nodesQueue.front();
                nodesQueue.pop();

                // Determine the index to insert the node's
                // value based on the traversal direction
                int index = leftToRight ? i : (size - 1 - i);

                // Insert the node's value at
                // the determined index
                row[index] = node->data;

                // Enqueue the left and right
                // children if they exist
                if(node->left){
                    nodesQueue.push(node->left);
                }
                if(node->right){
                    nodesQueue.push(node->right);
                }
            }
```

```
            // Switch the traversal
            // direction for the next level
            leftToRight = !leftToRight;

            // Add the current level's
            // values to the result vector
            result.push_back(row);
        }

        // Return the final result of
        // zigzag level order traversal
        return result;
    }
};



// Helper function to print the result
void printResult(const vector<vector<int>>& result) {
    for (const auto& row : result) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
}

int main() {
    // Creating a sample binary tree
    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->left = new Node(6);
    root->right->right = new Node(7);

    Solution solution;

    // Get the zigzag level order traversal
    vector<vector<int>> result = solution.ZigZagLevelOrder(root);

    // Print the result
    printResult(result);

    return 0;
}
```

Boundary Traversal of a Binary Tree

Mark as Completed

Problem Statement: Given a Binary Tree, perform the boundary traversal of the tree. The boundary traversal is the process of visiting the boundary nodes of the binary tree in the anticlockwise direction, starting from the root.

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

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to check
    // if a node is a leaf
    bool isLeaf(Node* root) {
        return !root->left && !root->right;
    }

    // Function to add the
    // left boundary of the tree
    void addLeftBoundary(Node* root, vector<int>& res) {
        Node* curr = root->left;
        while (curr) {
            // If the current node is not a leaf,
            // add its value to the result
            if (!isLeaf(curr)) {
                res.push_back(curr->data);
            }
            // Move to the left child if it exists,
            // otherwise move to the right child
            if (curr->left) {
                curr = curr->left;
            } else {
                curr = curr->right;
            }
        }
    }

    // Function to add the
    // right boundary of the tree
    void addRightBoundary(Node* root, vector<int>& res) {
        Node* curr = root->right;
        vector<int> temp;
        while (curr) {
            // If the current node is not a leaf,
            // add its value to a temporary vector
            if (!isLeaf(curr)) {
                temp.push_back(curr->data);
            }
            // Move to the right child if it exists,
            // otherwise move to the left child
            if (curr->right) {
                curr = curr->right;
            } else {
                curr = curr->left;
            }
```

```
        }
        // Reverse and add the values from
        // the temporary vector to the result
        for (int i = temp.size() - 1; i >= 0; --i) {
            res.push_back(temp[i]);
        }
    }

    // Function to add the
    // leaves of the tree
    void addLeaves(Node* root, vector<int>& res) {
        // If the current node is a
        // leaf, add its value to the result
        if (isLeaf(root)) {
            res.push_back(root->data);
            return;
        }
        // Recursively add leaves of
        // the left and right subtrees
        if (root->left) {
            addLeaves(root->left, res);
        }
        if (root->right) {
            addLeaves(root->right, res);
        }
    }

    // Main function to perform the
    // boundary traversal of the binary tree
    vector<int> printBoundary(Node* root) {
        vector<int> res;
        if (!root) {
            return res;
        }
        // If the root is not a leaf,
        // add its value to the result
        if (!isLeaf(root)) {
            res.push_back(root->data);
        }

        // Add the left boundary, leaves,
        // and right boundary in order
        addLeftBoundary(root, res);
        addLeaves(root, res);
        addRightBoundary(root, res);

        return res;
    }
};

// Helper function to
// print the result
void printResult(const vector<int>& result) {
    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;
}

int main() {
    // Creating a sample binary tree
    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->left = new Node(6);
    root->right->right = new Node(7);

    Solution solution;

    // Get the boundary traversal
    vector<int> result = solution.printBoundary(root);

    // Print the result
    cout << "Boundary Traversal: ";
    printResult(result);

    return 0;
}
```

Vertical Order Traversal of Binary Tree

Mark as Completed

119

Problem Statement: Given a Binary Tree, return the Vertical Order Traversal of it starting from the Leftmost level to the Rightmost level. If there are multiple nodes passing through a vertical line, then they should be printed as they appear in level order traversal of the tree.

```
#include <iostream>
#include <vector>
#include <set>
#include <queue>
#include <map>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to perform vertical order traversal
    // and return a 2D vector of node values
    vector<vector<int>> findVertical(Node* root){
        // Map to store nodes based on
        // vertical and level information
        map<int, map<int, multiset<int>>> nodes;
```

```
// Queue for BFS traversal, each
// element is a pair containing node
// and its vertical and level information
queue<pair<Node*, pair<int, int>>> todo;

// Push the root node with initial vertical
// and level values (0, 0)
todo.push({root, {0, 0}});

// BFS traversal
while(!todo.empty()){
    // Retrieve the node and its vertical
    // and level information from
    // the front of the queue
    auto p = todo.front();
    todo.pop();
    Node* temp = p.first;

    // Extract the vertical and level information
    // x -> vertical
    int x = p.second.first;
    // y -> level
    int y = p.second.second;

    // Insert the node value into the
    // corresponding vertical and level
    // in the map
    nodes[x][y].insert(temp->data);

    // Process left child
    if(temp->left){
        todo.push({
            temp->left,
            {
                // Move left in
                // terms of vertical
                x-1,
                // Move down in
                // terms of level
                y+1
            }
        });
    }

    // Process right child
    if(temp->right){
        todo.push({
            temp->right,
            {
                // Move right in
                // terms of vertical
                x+1,
                // Move down in
                // terms of level
                y+1
            }
        });
    }
}

// Prepare the final result vector
// by combining values from the map
vector<vector<int>> ans;
```

```
        for(auto p: nodes){
            vector<int> col;
            for(auto q: p.second){
                // Insert node values
                // into the column vector
                col.insert(col.end(), q.second.begin(), q.second.end());
            }
            // Add the column vector
            // to the final result
            ans.push_back(col);
        }
        return ans;
    }
};

// Helper function to
// print the result
void printResult(const vector<vector<int>>& result) {
    for(auto level: result){
        for(auto node: level){
            cout << node << " ";
        }
        cout << endl;
    }
    cout << endl;
}

int main() {
    // Creating a sample binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);

    Solution solution;

    // Get the Vertical traversal
    vector<vector<int>> verticalTraversal =
                    solution.findVertical(root);

    // Print the result
    cout << "Vertical Traversal: ";
    printResult(verticalTraversal);

    return 0;
}
```

Top view of a Binary Tree

Mark as Completed

Problem Statement: Given a Binary Tree, return its Top View. The Top View of a Binary Tree is the set of nodes visible when we see the tree from the top.

```
#include <iostream>
#include <vector>
#include <set>
#include <queue>
#include <map>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution{
public:
    // Function to return the
    // top view of the binary tree
    vector<int> topView(Node* root){
        // Vector to store the result
        vector<int> ans;

        // Check if the tree is empty
        if(root == NULL){
            return ans;
        }

        // Map to store the top view nodes
        // based on their vertical positions
        map<int, int> mpp;

        // Queue for BFS traversal, each element
        // is a pair containing node
        // and its vertical position
        queue<pair<Node*, int>> q;

        // Push the root node with its vertical
        // position (0) into the queue
        q.push({root, 0});

        // BFS traversal
        while(!q.empty()){
            // Retrieve the node and its vertical
            // position from the front of the queue
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;

            // If the vertical position is not already
            // in the map, add the node's data to the map
            if(mpp.find(line) == mpp.end()){
                mpp[line] = node->data;
            }
```

```
            // Process left child
            if(node->left != NULL){
                // Push the left child with a decreased
                // vertical position into the queue
                q.push({node->left, line - 1});
            }

            // Process right child
            if(node->right != NULL){
                // Push the right child with an increased
                // vertical position into the queue
                q.push({node->right, line + 1});
            }
        }

        // Transfer values from the
        // map to the result vector
        for(auto it : mpp){
            ans.push_back(it.second);
        }

        return ans;
    }
};

int main() {
    // Creating a sample binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);

    Solution solution;

    // Get the top view traversal
    vector<int> topView =
                    solution.topView(root);

    // Print the result
    cout << "Top View Traversal: "<< endl;
    for(auto node: topView){
        cout << node << " ";
    }

    return 0;
}
```

Bottom view of a Binary Tree

Mark as Completed

66

Problem Statement: Given a Binary Tree, return its Bottom View. The Bottom View of a Binary Tree is the set of nodes visible when we see the tree from the bottom.

```
#include <iostream>
#include <vector>
#include <set>
#include <queue>
#include <map>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution{
public:
    // Function to return the
    // bottom view of the binary tree
    vector<int> bottomView(Node* root){
        // Vector to store the result
        vector<int> ans;

        // Check if the tree is empty
        if(root == NULL){
            return ans;
        }

        // Map to store the bottom view nodes
        // based on their vertical positions
        map<int, int> mpp;

        // Queue for BFS traversal, each
        // element is a pair containing node
        // and its vertical position
        queue<pair<Node*, int>> q;

        // Push the root node with its vertical
        // position (0) into the queue
        q.push({root, 0});

        // BFS traversal
        while(!q.empty()){
            // Retrieve the node and its vertical
            // position from the front of the queue
            auto it = q.front();
            q.pop();
            Node* node = it.first;
            int line = it.second;

            // Update the map with the node's data
```

```
            // for the current vertical position
            mpp[line] = node->data;

            // Process left child
            if(node->left != NULL){
                // Push the left child with a decreased
                // vertical position into the queue
                q.push({node->left, line - 1});
            }

            // Process right child
            if(node->right != NULL){
                // Push the right child with an increased
                // vertical position into the queue
                q.push({node->right, line + 1});
            }
        }

        // Transfer values from the
        // map to the result vector
        for(auto it : mpp){
            ans.push_back(it.second);
        }

        return ans;
    }
};

int main() {
    // Creating a sample binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);

    Solution solution;

    // Get the Bottom View traversal
    vector<int> bottomView =
                    solution.bottomView(root);

    // Print the result
    cout << "Bottom View Traversal: "<< endl;
    for(auto node: bottomView){
        cout << node << " ";
    }

    return 0;
}
```

Right/Left view of binary tree

Mark as Completed

75

Problem Statement: Given a Binary Tree, return its right and left views.

The Right View of a Binary Tree is a list of nodes that can be seen when the tree is viewed from the right side. The Left View of a Binary Tree is a list of nodes that can be seen when the tree is viewed from the left side.

```
#include <iostream>
#include <vector>
#include <set>
#include <queue>
#include <map>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val),
        left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to return the
    // Right view of the binary tree
    vector<int> rightsideView(Node* root) {
        // Vector to store
        // the result
        vector<int> res;

        // Get the level order
        // traversal of the tree
        vector<vector<int>> levelTraversal = levelOrder(root);

        // Iterate through each level and
        // add the last element to the result
        for (auto level : levelTraversal) {
            res.push_back(level.back());
        }

        return res;
    }

    // Function to return the
    // Left view of the binary tree
    vector<int> leftsideView(Node* root) {
        // Vector to store the result
        vector<int> res;

        // Get the level order
        // traversal of the tree
        vector<vector<int>> levelTraversal = levelOrder(root);

        // Iterate through each level and
        // add the first element to the result
```

```
        for (auto level : levelTraversal) {
            res.push_back(level.front());
        }

        return res;
    }

private:
    // Function that returns the
    // level order traversal of a Binary tree
    vector<vector<int>> levelOrder(Node* root) {
        vector<vector<int>> ans;

        // Return an empty vector
        // if the tree is empty
        if (!root) {
            return ans;
        }

        // Use a queue to perform
        // level order traversal
        queue<Node*> q;
        q.push(root);

        while (!q.empty()) {
            int size = q.size();
            vector<int> level;

            // Process each node
            // in the current level
            for (int i = 0; i < size; i++) {
                Node* top = q.front();
                level.push_back(top->data);
                q.pop();

                // Enqueue the left
                // child if it exists
                if (top->left != NULL) {
                    q.push(top->left);
                }

                // Enqueue the right
                // child if it exists
                if (top->right != NULL) {
                    q.push(top->right);
                }
            }

            // Add the current
            // level to the result
            ans.push_back(level);
        }

        return ans;
    }
};




int main() {
    // Creating a sample binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
```

```
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);

    Solution solution;

        // Get the Right View traversal
    vector<int> rightView = solution.rightsideView(root);

    // Print the result for Right View
    cout << "Right View Traversal: ";
    for(auto node: rightView){
        cout << node << " ";
    }
    cout << endl;

    // Get the Left View traversal
    vector<int> leftView = solution.leftsideView(root);

    // Print the result for Left View
    cout << "Left View Traversal: ";
    for(auto node: leftView){
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

```
#include <iostream>
#include <vector>
#include <set>
#include <queue>
#include <map>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to return the Right view of the binary tree
    vector<int> rightsideView(Node* root){
        // Vector to store the result
        vector<int> res;
```

```
        // Call the recursive function
        // to populate the right-side view
        recursionRight(root, 0, res);

        return res;
    }

    // Function to return the Left view of the binary tree
    vector<int> leftsideView(Node* root){
        // Vector to store the result
        vector<int> res;

        // Call the recursive function
        // to populate the left-side view
        recursionLeft(root, 0, res);

        return res;
    }

private:
    // Recursive function to traverse the
    // binary tree and populate the left-side view
    void recursionLeft(Node* root, int level, vector<int>& res){
        // Check if the current node is NULL
        if(root == NULL){
            return;
        }

        // Check if the size of the result vector
        // is equal to the current level
        if(res.size() == level){
            // If equal, add the value of the
            // current node to the result vector
            res.push_back(root->data);
        }

        // Recursively call the function for the
        // left child with an increased level
        recursionLeft(root->left, level + 1, res);

        // Recursively call the function for the
        // right child with an increased level
        recursionLeft(root->right, level + 1, res);
    }

    // Recursive function to traverse the
    // binary tree and populate the right-side view
    void recursionRight(Node* root, int level, vector<int> &res){
        // Check if the current node is NULL
        if(root == NULL){
            return;
        }

        // Check if the size of the result vector
        // is equal to the current level
        if(res.size() == level){
            // If equal, add the value of the
            // current node to the result vector
            res.push_back(root->data);

            // Recursively call the function for the
            // right child with an increased level
            recursionRight(root->right, level + 1, res);
```

```
            // Recursively call the function for the
            // left child with an increased level
            recursionRight(root->left, level + 1, res);
        }
    }
};



int main() {
    // Creating a sample binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->left->left = new Node(4);
    root->left->right = new Node(10);
    root->left->left->right = new Node(5);
    root->left->left->right->right = new Node(6);
    root->right = new Node(3);
    root->right->right = new Node(10);
    root->right->left = new Node(9);

    Solution solution;

        // Get the Right View traversal
    vector<int> rightView = solution.rightsideView(root);

    // Print the result for Right View
    cout << "Right View Traversal: ";
    for(auto node: rightView){
        cout << node << " ";
    }
    cout << endl;

    // Get the Left View traversal
    vector<int> leftView = solution.leftsideView(root);

    // Print the result for Left View
    cout << "Left View Traversal: ";
    for(auto node: leftView){
        cout << node << " ";
    }
    cout << endl;

    return 0;
}
```

Check for Symmetrical Binary Tree

Mark as Completed

41

Problem Statement: Given a Binary Tree, determine whether the given tree is

symmetric or not. A Binary Tree would be Symmetric, when its mirror image is exactly the same as the original tree. If we were to draw a vertical line through the centre of the tree, the nodes on the left and right side would be mirror images of each other.

```
#include <iostream>
#include <vector>
#include <set>
#include <queue>
#include <map>

using namespace std;

// Node structure for the binary tree
struct Node {
    int data;
    Node* left;
    Node* right;
    // Constructor to initialize
    // the node with a value
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

class Solution {
private:
    // Function to check if
    // two subtrees are symmetric
    bool isSymmetricUtil(Node* root1, Node* root2) {
        // Check if either subtree is NULL
        if (root1 == NULL || root2 == NULL) {
            // If one subtree is NULL, the other
            // must also be NULL for symmetry
            return root1 == root2;
        }
        // Check if the data in the current nodes is equal
        // and recursively check for symmetry in subtrees
        return (root1->data == root2->data)
            && isSymmetricUtil(root1->left, root2->right)
            && isSymmetricUtil(root1->right, root2->left);
    }

public:
    // Public function to check if the
    // entire binary tree is symmetric
    bool isSymmetric(Node* root) {
        // Check if the tree is empty
        if (!root) {
            // An empty tree is
            // considered symmetric
            return true;
        }
        // Call the utility function
        // to check symmetry of subtrees
        return isSymmetricUtil(root->left, root->right);
    }
};


// Function to print the Inorder
// Traversal of the Binary Tree
void printInorder(Node* root){
    if(!root){
        return;
    }
    printInorder(root->left);
```

```
    cout << root->data << " ";
    printInorder(root->right);
}


int main() {
    // Creating a sample binary tree
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(2);
    root->left->left = new Node(3);
    root->right->right = new Node(3);
    root->left->right = new Node(4);
    root->right->left = new Node(4);

    Solution solution;

    cout << "Binary Tree (Inorder): ";
    printInorder(root);
    cout << endl;

    bool res;
    res = solution.isSymmetric(root);

    if(res){
        cout << "This Tree is Symmetrical" << endl;
    }
    else{
        cout << "This Tree is NOT Symmetrical" << endl;
    }

    return 0;
}
```

Print Root to Node Path in a Binary Tree

Mark as Completed

123

Problem Statement: Given a Binary Tree and a reference to a root belonging to it. Return the path from the root node to the given leaf node.

No two nodes in the tree have the same data value.
It is assured that the given node is present and a path always exists.

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

using namespace std;

// TreeNode structure
struct TreeNode {
```

```
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};


class Solution {
public:
    // Function to find the path from the
    // root to a given node with value 'x'
    bool getPath(TreeNode* root, vector<int>& arr, int x) {
        // Base case: If the current
        // node is null, return false
        if (!root) {
            return false;
        }

        // Add the current node's
        // value to the path vector
        arr.push_back(root->val);

        // If the current node's value is equal
        // to the target value 'x', return true
        if (root->val == x) {
            return true;
        }

        // Recursively search for the target value
        // 'x' in the left and right subtrees
        if (getPath(root->left, arr, x)
            || getPath(root->right, arr, x)) {
            return true;
        }

        // If the target value 'x' is not found
        // in the current path, backtrack
        arr.pop_back();
        return false;
    }

    // Function to find and return the path from
    // the root to a given node with value 'B'
    vector<int> solve(TreeNode* A, int B) {
        // Initialize an empty
        // vector to store the path
        vector<int> arr;

        // If the root node is null,
        // return the empty path vector
        if (A == NULL) {
            return arr;
        }

        // Call the getPath function to find
        // the path to the node with value 'B'
        getPath(A, arr, B);

        // Return the path vector
        return arr;
    }
};

int main() {
```

```
    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);

    Solution sol;

    int targetLeafValue = 7;

    vector<int> path = sol.solve(root, targetLeafValue);

    cout << "Path from root to leaf with value " <<
        targetLeafValue << ": ";
    for (int i = 0; i < path.size(); ++i) {
        cout << path[i];
        if (i < path.size() - 1) {
            cout << " -> ";
        }
    }

    return 0;
}
```

Lowest Common Ancestor for two given Nodes

Mark as Completed

130

Problem Statement: Given a binary tree, Find the Lowest Common Ancestor for two given Nodes (x,y).

Lowest Common Ancestor(LCA): The lowest common ancestor is defined between two nodes x and y as the lowest node in T that has both x and y as descendants (where we allow a node to be a descendant of itself.

Examples:

Consider the following Binary Tree

Example 1:

Input: x = 4 , y = 5

Output: 2

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

        //result
        if(left == NULL) {
            return right;
        }
        else if(right == NULL) {
            return left;
        }
        else { //both left and right are not null, we found our result
            return root;
        }
    }
};
```

Maximum Width of a Binary Tree

Mark as Completed

89

Problem Statement: Given a Binary Tree, return its maximum width. The maximum width of a Binary Tree is the maximum diameter among all its levels. The width or diameter of a level is the number of nodes between the leftmost and rightmost nodes.

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

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

```
class Solution {
public:
    // Function widthOfBinaryTree to find the
    // maximum width of the Binary Tree
    int widthOfBinaryTree(TreeNode* root) {
        // If the root is null,
        // the width is zero
        if (!root) {
            return 0;
        }

        // Initialize a variable 'ans'
        // to store the maximum width
        int ans = 0;

        // Create a queue to perform level-order
        // traversal, where each element is a pair
        // of TreeNode* and its position in the level
        queue<pair<TreeNode*, int>> q;
        // Push the root node and its
        // position (0) into the queue
        q.push({root, 0});

        // Perform level-order traversal
        while (!q.empty()) {
            // Get the number of
            // nodes at the current level
            int size = q.size();
            // Get the position of the front
            // node in the current level
            int mmin = q.front().second;

            // Store the first and last positions
            // of nodes in the current level
            int first, last;

            // Process each node
            // in the current level
            for (int i = 0; i < size; i++) {
                // Calculate current position relative
                // to the minimum position in the level
                int cur_id = q.front().second - mmin;
                // Get the current node
                TreeNode* node = q.front().first;
                // Pop the front node from the queue
                q.pop();

                // If this is the first node in the level,
                // update the 'first' variable
                if (i == 0) {
                    first = cur_id;
                }

                // If this is the last node in the level,
                // update the 'last' variable
                if (i == size - 1) {
                    last = cur_id;
                }

                // Enqueue the left child of the
                // current node with its position
                if (node->left) {
                    q.push({node->left, cur_id * 2 + 1});
                }
```

```
                // Enqueue the right child of the
                // current node with its position
                if (node->right) {
                    q.push({node->right, cur_id * 2 + 2});
                }
            }

            // Update the maximum width by calculating
            // the difference between the first and last
            // positions, and adding 1
            ans = max(ans, last - first + 1);
        }

        // Return the maximum
        // width of the binary tree
        return ans;
    }
};


int main() {
    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);

    Solution sol;

    int maxWidth = sol.widthOfBinaryTree(root);

    cout << "Maximum width of the binary tree is: "
                        << maxWidth << endl;

    return 0;
}
```

Check for Children Sum Property in a Binary Tree

Mark as Completed

91

Problem Statement: Given a Binary Tree, convert the value of its nodes to follow the Children Sum Property. The Children Sum Property in a binary tree states

that for every node, the sum of its children's values (if they exist) should be equal to the node's value. If a child is missing, it is considered as having a value of 0.

Note:

The node values can be increased by any positive integer any number of times, but decrementing any node value is not allowed.
A value for a NULL node can be assumed as 0.
We cannot change the structure of the given binary tree.
Examples

Example 1:

Input:Binary Tree: 2 35 10 2 3 5 2

Output: Binary Tree: 45 35 10 30 5 8 2

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

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};



class Solution {
public:
    // Function to change the values of the nodes
    // based on the sum of its children's values.
    void changeTree(TreeNode* root) {
        // Base case: If the current node
        // is NULL, return and do nothing.
        if (root == NULL) {
            return;
        }

        // Calculate the sum of the values of
        // the left and right children, if they exist.
        int child = 0;
        if (root->left) {
            child += root->left->val;
        }
        if (root->right) {
            child += root->right->val;
        }

        // Compare the sum of children with
        // the current node's value and update
        if (child >= root->val) {
            root->val = child;
        } else {
            // If the sum is smaller, update the
            // child with the current node's value.
            if (root->left) {
```

```
                root->left->val = root->val;
            } else if (root->right) {
                root->right->val = root->val;
            }
        }

        // Recursively call the function
        // on the left and right children.
        changeTree(root->left);
        changeTree(root->right);

        // Calculate the total sum of the
        // values of the left and right
        // children, if they exist.
        int tot = 0;
        if (root->left) {
            tot += root->left->val;
        }
        if (root->right) {
            tot += root->right->val;
        }

        // If either left or right child
        // exists, update the current node's
        // value with the total sum.
        if (root->left or root->right) {
            root->val = tot;
        }
    }
};


// Function to print the inorder
// traversal of the tree
void inorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }
    inorderTraversal(root->left);
    cout << root->val << " ";
    inorderTraversal(root->right);
}

int main() {
    // Create the binary tree
    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);

    Solution sol;

    // Print the inorder traversal
    // of tree before modification
    cout << "Binary Tree before modification: ";
    inorderTraversal(root);
    cout << endl;

    // Call the changeTree function
```

```
    // to modify the binary tree
    sol.changeTree(root);

    // Print the inorder traversal
    // after modification
    cout << "Binary Tree after Children Sum Property: " ;
    inorderTraversal(root);
    cout << endl;

    return 0;
}
```

Count Number of Nodes in a Binary Tree

Mark as Completed

31

Problem Statement: Given a Complete Binary Tree, count and return the number of nodes in the given tree. A Complete Binary Tree is a binary tree in which all levels are completely filled, except possibly for the last level, and all nodes are as left as possible.

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

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};


class Solution {
public:
    // Function to perform inorder
    // traversal and count nodes
    void inorder(TreeNode* root, int &count) {
        // Base case: If the current
        // node is NULL, return
        if (root == NULL) {
            return;
        }

        // Increment count
        // for the current node
        count++;

        // Recursively call inorder
```

```
        // on the left subtree
        inorder(root->left, count);

        // Recursively call inorder
        // on the right subtree
        inorder(root->right, count);
    }

    // Function to count nodes in the binary tree
    int countNodes(TreeNode* root) {
        // Base case: If the root is NULL,
        // the tree is empty, return 0
        if (root == NULL) {
            return 0;
        }

        // Initialize count variable to
        // store the number of nodes
        int count = 0;

        // Call the inorder traversal
        // function to count nodes
        inorder(root, count);

        // Return the final count of
        // nodes in the binary tree
        return count;
    }
};



int main() {
    // Create the binary tree
    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 sol;

    // Call the countNodes function
    int totalNodes = sol.countNodes(root);

    // Print the result
    cout << "Total number of nodes in the Complete Binary Tree: "
            << totalNodes << endl;



    return 0;
}



#include <iostream>
#include <unordered_map>
#include <vector>
#include <queue>

using namespace std;
```

```
// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};



class Solution {
public:
    // Function to count nodes
    // in a binary tree
    int countNodes(TreeNode* root) {
        // Base case: If the root
        // is NULL, there are no nodes
        if (root == NULL) {
            return 0;
        }

        // Find the left height and
        // right height of the tree
        int lh = findHeightLeft(root);
        int rh = findHeightRight(root);

        // Check if the last level
        // is completely filled
        if (lh == rh) {
            // If so, use the formula for
            // total nodes in a perfect
            // binary tree ie. 2^h - 1
            return (1 << lh) - 1;
        }

        // If the last level is not completely
        // filled, recursively count nodes in
        // left and right subtrees
        return 1 + countNodes(root->left) + countNodes(root->right);
    }

    // Function to find the left height of a tree
    int findHeightLeft(TreeNode* node) {
        int height = 0;
        // Traverse left child until
        // reaching the leftmost leaf
        while (node) {
            height++;
            node = node->left;
        }
        return height;
    }

    // Function to find the right height of a tree
    int findHeightRight(TreeNode* node) {
        int height = 0;
        // Traverse right child until
        // reaching the rightmost leaf
        while (node) {
            height++;
            node = node->right;
        }
        return height;
```

```
    }
};



int main() {
    // Create the binary tree
    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 sol;

    // Call the countNodes function
    int totalNodes = sol.countNodes(root);

    // Print the result
    cout << "Total number of nodes in the Complete Binary Tree: "
            << totalNodes << endl;



    return 0;
}
```

Construct A Binary Tree from Inorder and Preorder Traversal

Mark as Completed

61

Problem Statement: Given the Preorder and Inorder traversal of a Binary Tree, construct the Unique Binary Tree represented by them.

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

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

```
class Solution {
public:
    // Function to build a binary tree
    // from preorder and inorder traversals
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder){
        // Create a map to store indices
        // of elements in the inorder traversal
        map<int, int> inMap;

        // Populate the map with indices
        // of elements in the inorder traversal
        for(int i = 0; i < inorder.size(); i++){
            inMap[inorder[i]] = i;
        }

        // Call the private helper function
        // to recursively build the tree
        TreeNode* root = buildTree(preorder, 0, preorder.size()-1, inorder, 0,
inorder.size()-1, inMap);

        return root;
    }

private:
    // Recursive helper function to build the tree
    TreeNode* buildTree(vector<int>& preorder, int preStart, int preEnd,
            vector<int>& inorder, int inStart, int inEnd, map<int, int>& inMap){
                // Base case: If the start indices
                // exceed the end indices, return NULL
                if(preStart > preEnd || inStart > inEnd){
                    return NULL;
                }

                // Create a new TreeNode with value
                // at the current preorder index
                TreeNode* root = new TreeNode(preorder[preStart]);

                // Find the index of the current root
                // value in the inorder traversal
                int inRoot = inMap[root->val];

                // Calculate the number of
                // elements in the left subtree
                int numsLeft = inRoot - inStart;

                // Recursively build the left subtree
                root->left = buildTree(preorder, preStart + 1, preStart +
numsLeft,
                                inorder, inStart, inRoot - 1, inMap);

                // Recursively build the right subtree
                root->right = buildTree(preorder, preStart + numsLeft + 1,
preEnd,
                                inorder, inRoot + 1, inEnd, inMap);

                // Return the current root node
                return root;
            }
};

// Function to print the
// inorder traversal of a tree
void printInorder(TreeNode* root){
```

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

// Function to print the
// given vector
void printVector(vector<int>&vec){
    for(int i = 0; i < vec.size(); i++){
        cout << vec[i] << " ";
    }
    cout << endl;
}


int main() {
    vector<int> inorder = {9, 3, 15, 20, 7};
    vector<int> preorder = {3, 9, 20, 15, 7};

    cout << "Inorder Vector: ";
    printVector(inorder);

    cout << "Preorder Vector: ";
    printVector(preorder);

    Solution sol;

    TreeNode* root = sol.buildTree(preorder, inorder);

    cout << "Inorder of Unique Binary Tree Created: "<< endl;
    printInorder(root);
    cout << endl;


    return 0;
}
```

Construct Binary Tree from Inorder and PostOrder Traversal

Mark as Completed

27

Problem Statement: Given the Postorder and Inorder traversal of a Binary Tree, construct the Unique Binary Tree represented by them.

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

```
#include <queue>
#include <map>

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to build a binary tree
    // from inorder and postorder traversals
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if (inorder.size() != postorder.size()) {
            return NULL;
        }

        // Create a map to store the indices
        // of elements in the inorder traversal
        map<int, int> hm;
        for (int i = 0; i < inorder.size(); i++) {
            hm[inorder[i]] = i;
        }

        // Call the recursive function
        // to build the binary tree
        return buildTreePostIn(inorder, 0, inorder.size() - 1, postorder, 0,
            postorder.size() - 1, hm);
    }

    // Recursive function to build a binary
    // tree from inorder and postorder traversals
    TreeNode* buildTreePostIn(vector<int>& inorder, int is, int ie,
        vector<int>& postorder, int ps, int pe, map<int, int>& hm) {

        // Base case: If the subtree
        // is empty, return NULL
        if (ps > pe || is > ie) {
            return NULL;
        }

        // Create a new TreeNode
        // with the root value from postorder
        TreeNode* root = new TreeNode(postorder[pe]);

        // Find the index of the root
        // value in inorder traversal
        int inRoot = hm[postorder[pe]];

        // Number of nodes in the left subtree
        int numsLeft = inRoot - is;

        // Recursively build the
        // left and right subtrees
        root->left = buildTreePostIn(inorder, is, inRoot - 1, postorder,
            ps, ps + numsLeft - 1, hm);

        root->right = buildTreePostIn(inorder, inRoot + 1, ie, postorder,
            ps + numsLeft, pe - 1, hm);
```

```
        // Return the root of
        // the constructed subtree
        return root;
    }
};

// Function to print the
// inorder traversal of a tree
void printInorder(TreeNode* root) {
    if (!root) {
        return;
    }
    printInorder(root->left);
    cout << root->val << " ";
    printInorder(root->right);
}

// Function to print the given vector
void printVector(vector<int>& vec) {
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }
    cout << endl;
}

int main() {
    // Example input vectors
    vector<int> inorder = {40, 20, 50, 10, 60, 30};
    vector<int> postorder = {40, 50, 20, 60, 30, 10};

    // Display the input vectors
    cout << "Inorder Vector: ";
    printVector(inorder);

    cout << "Postorder Vector: ";
    printVector(postorder);

    Solution sol;

    // Build the binary tree and
    // print its inorder traversal
    TreeNode* root = sol.buildTree(inorder, postorder);

    cout << "Inorder of Unique Binary Tree Created: " << endl;
    printInorder(root);
    cout << endl;

    return 0;
}
```

Serialize And Deserialize a Binary Tree

Mark as Completed

46

Problem Statement: Given a Binary Tree, design an algorithm to serialise and deserialise it. There is no restriction on how the serialisation and deserialization takes place. But it needs to be ensured that the serialised binary tree can be deserialized to the original tree structure. Serialisation is the process of translating a data structure or object state into a format that can be stored or transmitted (for example, across a computer network) and reconstructed later. The opposite operation, that is, extracting a data structure from stored information, is deserialization.

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

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

class Solution {
public:
    // Encodes the tree into a single string
    string serialize(TreeNode* root) {
        // Check if the tree is empty
        if (!root) {
            return "";
        }

        // Initialize an empty string
        // to store the serialized data
        string s = "";
        // Use a queue for
        // level-order traversal
        queue<TreeNode*> q;
        // Start with the root node
        q.push(root);

        // Perform level-order traversal
        while (!q.empty()) {
            // Get the front node in the queue
            TreeNode* curNode = q.front();
            q.pop();

            // Check if the current node is
            // null and append "#" to the string
            if (curNode == nullptr) {
                s += "#,";
            } else {
                // Append the value of the
                // current node to the string
                s += to_string(curNode->val) + ",";
                // Push the left and right children
```

```
            // to the queue for further traversal
            q.push(curNode->left);
            q.push(curNode->right);
        }
    }

    // Return the
    // serialized string
    return s;
}

// Decode the encoded
// data to a tree
TreeNode* deserialize(string data) {
    // Check if the
    // serialized data is empty
    if (data.empty()) {
        return nullptr;
    }

    // Use a stringstream to
    // tokenize the serialized data
    stringstream s(data);
    string str;
    // Read the root value
    // from the serialized data
    getline(s, str, ',');
    TreeNode* root = new TreeNode(stoi(str));

    // Use a queue for
    // level-order traversal
    queue<TreeNode*> q;
    // Start with the root node
    q.push(root);

    // Perform level-order traversal
    // to reconstruct the tree
    while (!q.empty()) {
        // Get the front node in the queue
        TreeNode* node = q.front();
        q.pop();

        // Read the value of the left
        // child from the serialized data
        getline(s, str, ',');
        // If the value is not "#", create a new
        // left child and push it to the queue
        if (str != "#") {
            TreeNode* leftNode = new TreeNode(stoi(str));
            node->left = leftNode;
            q.push(leftNode);
        }

        // Read the value of the right child
        // from the serialized data
        getline(s, str, ',');
        // If the value is not "#", create a
        // new right child and push it to the queue
        if (str != "#") {
            TreeNode* rightNode = new TreeNode(stoi(str));
            node->right = rightNode;
            q.push(rightNode);
        }
    }
```

```
        // Return the reconstructed
        // root of the tree
        return root;
    }
};


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

int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->right->left = new TreeNode(4);
    root->right->right = new TreeNode(5);

    Solution solution;
    cout << "Orignal Tree: ";
    inorder(root);
    cout << endl;

    string serialized = solution.serialize(root);
    cout << "Serialized: " << serialized << endl;

    TreeNode* deserialized = solution.deserialize(serialized);
    cout << "Tree after deserialisation: ";
    inorder(deserialized);
    cout << endl;

    return 0;
}
```

Morris Preorder Traversal of a Binary Tree

Mark as Completed

52

Problem Statement: Given a Binary Tree, implement Morris Preorder Traversal and return the array containing its preorder sequence.

Morris Preorder Traversal is a tree traversal algorithm aiming to achieve a space complexity of O(1) without recursion or an external data structure. The

algorithm should efficiently visit each node in the binary tree in preorder sequence, printing or processing the node values as it traverses, without using a stack or recursion.

```
#include <iostream>
#include <sstream>
#include <unordered_map>
#include <vector>
#include <queue>
#include <map>

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to perform iterative Morris
    // preorder traversal of a binary tree
    vector<int> getPreorder(TreeNode* root) {
        // Vector to store the
        // preorder traversal result
        vector<int> preorder;

        // Pointer to the current node,
        // starting with the root
        TreeNode* cur = root;

        // Iterate until the
        // current node becomes NULL
        while (cur != NULL) {
            // If the current node
            // has no left child
            if (cur->left == NULL) {
                // Add the value of the
                // current node to the preorder vector
                preorder.push_back(cur->val);

                // Move to the right child
                cur = cur->right;
            } else {
                // If the current node has a left child
                // Create a pointer to traverse to the
                // rightmost node in the left subtree
                TreeNode* prev = cur->left;

                // Traverse to the rightmost node in the
                // left subtree or until we find a node
                // whose right child is not yet processed
                while (prev->right && prev->right != cur) {
                    prev = prev->right;
                }

                // If the right child of the
                // rightmost node is NULL
                if (prev->right == NULL) {
                    // Set the right child of the
                    // rightmost node to the current node
                    prev->right = cur;
```

```
                    // Move to the left child
                    cur = cur->left;
                } else {
                    // If the right child of the
                    // rightmost node is not NULL
                    // Reset the right child to NULL
                    prev->right = NULL;

                    // Add the value of the
                    // current node to the preorder vector
                    preorder.push_back(cur->val);

                    // Move to the right child
                    cur = cur->right;
                }
            }
        }

        // Return the resulting
        //preorder traversal vector
        return preorder;
    }
};


int main() {

    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->left->right->right = new TreeNode(6);

    Solution sol;

    vector<int> preorder = sol.getPreorder(root);

    cout << "Binary Tree Morris Preorder Traveral: ";
    for(int i = 0; i< preorder.size(); i++){
        cout << preorder[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Morris Inorder Traversal of a Binary tree

Mark as Completed

Problem Statement: Given a Binary Tree, implement Morris Inorder Traversal and return the array containing its inorder sequence.

Morris Inorder Traversal is a tree traversal algorithm aiming to achieve a space complexity of O(1) without recursion or an external data structure. The algorithm should efficiently visit each node in the binary tree in inorder sequence, printing or processing the node values as it traverses, without using a stack or recursion.

```
#include <iostream>
#include <sstream>
#include <unordered_map>
#include <vector>
#include <queue>
#include <map>

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to perform iterative Morris
    // inorder traversal of a binary tree
    vector<int> getInorder(TreeNode* root) {
        // Vector to store the
        // inorder traversal result
        vector<int> inorder;
        // Pointer to the current node,
        // starting from the root
        TreeNode* cur = root;

        // Loop until the current
        // node is not NULL
        while (cur != NULL) {
            // If the current node's
            // left child is NULL
            if (cur->left == NULL) {
                // Add the value of the current
                // node to the inorder vector
                inorder.push_back(cur->val);
                // Move to the right child
                cur = cur->right;
            } else {
                // If the left child is not NULL,
                // find the predecessor (rightmost node
                // in the left subtree)
                TreeNode* prev = cur->left;
                while (prev->right && prev->right != cur) {
                    prev = prev->right;
                }

                // If the predecessor's right child
                // is NULL, establish a temporary link
                // and move to the left child
                if (prev->right == NULL) {
                    prev->right = cur;
```

```
                    cur = cur->left;
                } else {
                    // If the predecessor's right child
                    // is already linked, remove the link,
                    // add current node to inorder vector,
                    // and move to the right child
                    prev->right = NULL;
                    inorder.push_back(cur->val);
                    cur = cur->right;
                }
            }
        }

        // Return the inorder
        // traversal result
        return inorder;
    }
};


int main() {

    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->left->right->right = new TreeNode(6);

    Solution sol;

    vector<int> inorder = sol.getInorder(root);

    cout << "Binary Tree Morris Inorder Traversal: ";
    for(int i = 0; i< inorder.size(); i++){
        cout << inorder[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Flatten Binary Tree to Linked List

Mark as Completed

61

Problem Statement: Given a Binary Tree, convert it to a Linked List where the linked list nodes follow the same order as the pre-order traversal of the binary tree.

Use the right pointer of the Binary Tree as the â ˜nextâ ™ pointer for the linked list and set the left pointer to null. Do this in place and do not create

```
extra nodes.
#include <iostream>
#include <sstream>
#include <unordered_map>
#include <vector>
#include <queue>
#include <map>
#include <stack>

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Initialize a global variable
    // 'prev' to keep track of the
    // previously processed node.
    TreeNode* prev = NULL;

    // Function to flatten a binary tree
    // to a right next Linked List structure
    void flatten(TreeNode* root) {
       // Base case: If the current
       // node is NULL, return.
       if(root==NULL){
           return;
       }

       // Recursive call to
       // flatten the right subtree
       flatten(root->right);

       // Recursive call to
       // flatten the left subtree
       flatten(root->left);

       // At this point, both left and right
       // subtrees are flattened, and 'prev'
       // is pointing to the rightmost node
       // in the flattened right subtree.

       // Set the right child of
       // the current node to 'prev'.
       root->right = prev;

       // Set the left child of the
       // current node to NULL.
       root->left = NULL;

       // Update 'prev' to the current
       // node for the next iteration.
       prev = root;
    }
};
```

```
// Print the preorder traversal of the
// Orignal Binary Tree
void printPreorder(TreeNode* root){
    if(!root){
        return;
    }
    cout << root->val << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}

// Print the Binary Tree along the
// Right Pointers after Flattening
void printFlattenTree(TreeNode* root){
    if(!root){
        return;
    }
    cout << root->val << " ";
    printFlattenTree(root->right);
}


int main() {

    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->left->right->right = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->left = new TreeNode(8);

    Solution sol;

    cout << "Binary Tree Preorder: ";
    printPreorder(root);
    cout << endl;

    sol.flatten(root);

    cout << "Binary Tree After Flatten: ";
    printFlattenTree(root);
    cout << endl;


    return 0;
}
```

```
#include <iostream>
#include <sstream>
#include <unordered_map>
#include <vector>
#include <queue>
#include <map>
#include <stack>

using namespace std;
```

```
// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Initialize a global variable
    // 'prev' to keep track of the
    // previously processed node.
    TreeNode* prev = NULL;

    // Function to flatten a binary tree
    // to a right next Linked List structure
    void flatten(TreeNode* root) {
        // Base case: If the current
        // node is NULL, return.
        if(root == NULL){
            return;
        }
        // Use a stack for
        // iterative traversal.
        stack<TreeNode*> st;
        // Push the root node
        // onto the stack.
        st.push(root);

        // Continue the loop until
        // the stack is empty.
        while (!st.empty()) {
            // Get the top node from the stack.
            TreeNode* cur = st.top();
            // Pop the top node.
            st.pop();

            if (cur->right != NULL) {
                 // Push the right child
                 // onto the stack.
                st.push(cur->right);
            }

            if (cur->left != NULL) {
                 // Push the left child
                 // onto the stack.
                st.push(cur->left);
            }

            if (!st.empty()) {
                // Connect the right child to
                // the next node in the stack.
                cur->right = st.top();
            }

            // Set the left child to NULL to
            // form a right-oriented linked list.
            cur->left = NULL;
        }
    }
};
```

```
// Print the preorder traversal of the
// Orignal Binary Tree
void printPreorder(TreeNode* root){
    if(!root){
        return;
    }
    cout << root->val << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}

// Print the Binary Tree along the
// Right Pointers after Flattening
void printFlattenTree(TreeNode* root){
    if(!root){
        return;
    }
    cout << root->val << " ";
    printFlattenTree(root->right);
}


int main() {

    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->left->right->right = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->left = new TreeNode(8);

    Solution sol;

    cout << "Binary Tree Preorder: ";
    printPreorder(root);
    cout << endl;

    sol.flatten(root);

    cout << "Binary Tree After Flatten: ";
    printFlattenTree(root);
    cout << endl;


    return 0;
}
```

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

```
#include <map>
#include <stack>

using namespace std;

// TreeNode structure
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Solution {
public:
    // Function to flatten a binary tree
    // to a right next Linked List structure
    void flatten(TreeNode* root) {
        // Initialize a pointer
        // 'curr' to the root of the tree
        TreeNode* curr = root;

        // Iterate until 'curr'
        // becomes NULL
        while (curr) {
            // Check if the current
            // node has a left child
            if (curr->left) {
                // If yes, find the rightmost
                // node in the left subtree
                TreeNode* pre = curr->left;
                while (pre->right) {
                    pre = pre->right;
                }

                // Connect the rightmost node in
                // the left subtree to the current
               //  node's right child
                pre->right = curr->right;

                // Move the entire left subtree to the
                // right child of the current node
                curr->right = curr->left;

                // Set the left child of
                // the current node to NULL
                curr->left = NULL;
            }

            // Move to the next node
            // on the right side
            curr = curr->right;
        }
    }
};



// Print the preorder traversal of the
// Orignal Binary Tree
void printPreorder(TreeNode* root){
    if(!root){
        return;
    }
```

```
    cout << root->val << " ";
    printPreorder(root->left);
    printPreorder(root->right);
}

// Print the Binary Tree along the
// Right Pointers after Flattening
void printFlattenTree(TreeNode* root){
    if(!root){
        return;
    }
    cout << root->val << " ";
    printFlattenTree(root->right);
}


int main() {

    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->left->right->right = new TreeNode(6);
    root->right->right = new TreeNode(7);
    root->right->left = new TreeNode(8);

    Solution sol;

    cout << "Binary Tree Preorder: ";
    printPreorder(root);
    cout << endl;

    sol.flatten(root);

    cout << "Binary Tree After Flatten: ";
    printFlattenTree(root);
    cout << endl;


    return 0;
}
```