DOMAIN WINTER WINNING CAMP ASSIGNMENT

Student Name: Siddharth **UID: 22BCS15779**
Branch: BE-CSE **Section/Group: 22BCS_FL_IOT-603/B**
Semester: 5th

DAY-6 [26-12-2024]

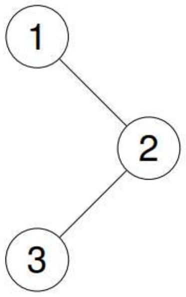
DSA Questions(Trees)

Very Easy:

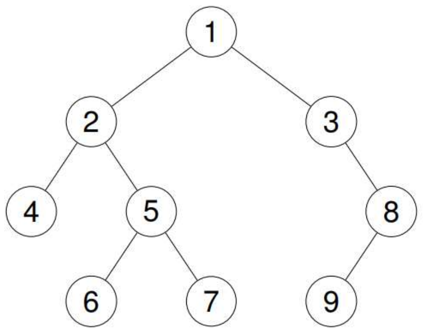
1. Binary Tree Inorder Traversal

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

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



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

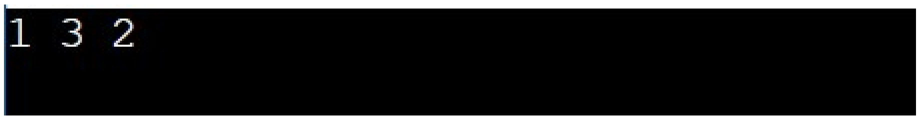


Constraints:
The number of nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100

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

CODE:
```
#include <iostream>
#include <vector>
using namespace std;

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

void inorderTraversal(TreeNode* root, vector<int>& result) {
    if (root == nullptr) return;
    inorderTraversal(root->left, result);  // Visit left subtree
    result.push_back(root->val);           // Visit current node
    inorderTraversal(root->right, result);// Visit right subtree
}

vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    inorderTraversal(root, result);
    return result;
}
int main() {
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    vector<int> result = inorderTraversal(root);
    for (int val : result) {
        cout << val << " ";
    }
    return 0;
}
```

OUTPUT:



```
1 3 2
```

2. Count Complete Tree Nodes

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

According to Wikipedia, every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

Design an algorithm that runs in less than O(n) time complexity.

Example 1:
Input: root = [1,2,3,4,5,6]
Output: 6

Example 2:
Input: root = []
Output: 0

Example 3:
Input: root = [1]
Output: 1

Constraints:

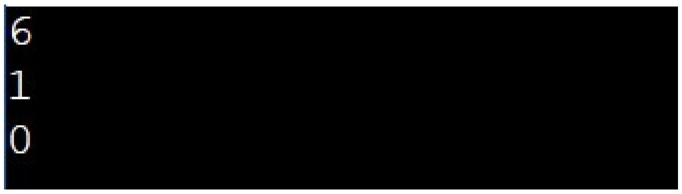
The number of nodes in the tree is in the range [0, 5 * 104].
0 <= Node.val <= 5 * 104
The tree is guaranteed to be complete.

Reference: https://leetcode.com/problems/count-complete-tree-nodes/description/

CODE:

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

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

// Function to calculate the depth of the tree
int calculateDepth(TreeNode* root) {
    int depth = 0;
    while (root) {
        depth++;
        root = root->left;
    }
    return depth;
}

// Function to check if a node exists at a given index in the last level
bool exists(int idx, int depth, TreeNode* root) {
    int left = 0, right = pow(2, depth - 1) - 1;
    for (int i = 0; i < depth - 1; i++) {
        int mid = (left + right) / 2;
        if (idx <= mid) {
            root = root->left;
            right = mid;
        } else {
            root = root->right;
            left = mid + 1;
        }
    }
    return root != nullptr;
}

// Main function to count the nodes in the tree
int countNodes(TreeNode* root) {
    if (!root) return 0;

    int depth = calculateDepth(root);
    if (depth == 1) return 1; // Single node

    int left = 0, right = pow(2, depth - 1) - 1;
    int lastLevelCount = 0;

    while (left <= right) {
        int mid = (left + right) / 2;
        if (exists(mid, depth, root)) {
            lastLevelCount = mid + 1;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return pow(2, depth - 1) - 1 + lastLevelCount;
}

int main() {
    // Example tree: [1, 2, 3, 4, 5, 6]
    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);

    cout << countNodes(root) << endl; // Output: 6

    // Example tree: [1]
    TreeNode* singleNode = new TreeNode(1);
    cout << countNodes(singleNode) << endl; // Output: 1

    // Example tree: []
    TreeNode* emptyTree = nullptr;
    cout << countNodes(emptyTree) << endl; // Output: 0

    return 0;
}
```

OUTPUT:



```
6
1
0
```

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.

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

Example 2:
Input: [1,null,2]
Output: 2

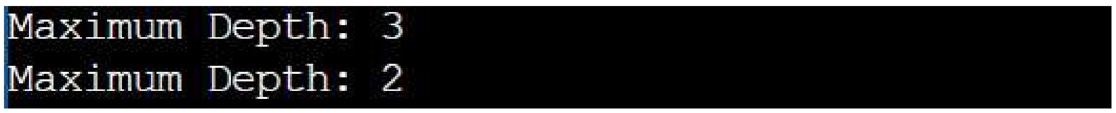
Constraints:
The number of nodes in the tree is in the range [0, 104].
-100 <= Node.val <= 100

Reference: https://leetcode.com/problems/maximum-depth-of-binary-tree/description/

CODE:

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

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

// Recursive Approach (DFS)
int maxDepth(TreeNode* root) {
    if (!root) return 0;
    int leftDepth = maxDepth(root->left);
    int rightDepth = maxDepth(root->right);
    return max(leftDepth, rightDepth) + 1;
}

int main() {
    // Example tree: [3, 9, 20, null, null, 15, 7]
    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);

    cout << "Maximum Depth: " << maxDepth(root) << endl; // Output: 3

    // Example tree: [1, null, 2]
    TreeNode* root2 = new TreeNode(1);
    root2->right = new TreeNode(2);
    cout << "Maximum Depth: " << maxDepth(root2) << endl; // Output: 2

    return 0;
}
```

OUTPUT:

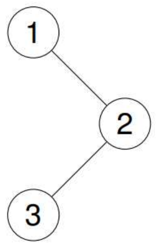


```
Maximum Depth: 3
Maximum Depth: 2
```

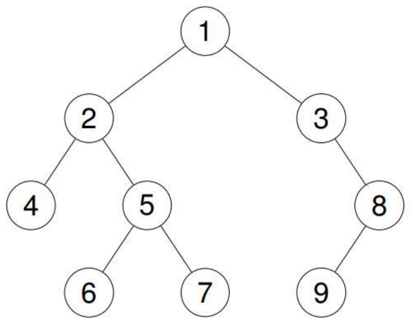
4.Binary Tree Preorder Traversal

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

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



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



Constraints:
The number of nodes in the tree is in the range [1, 100].
1 <= Node.val <= 1000

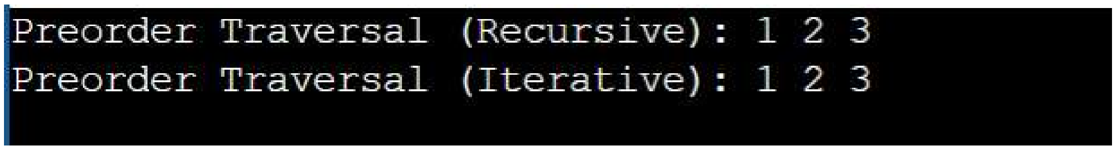
Reference: https://leetcode.com/problems/binary-tree-preorder-traversal/description/

CODE:

```
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

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

```
// Recursive preorder traversal
void preorder(TreeNode* root, vector<int>& result) {
    if (!root) return;
    result.push_back(root->val);
    preorder(root->left, result);
    preorder(root->right, result);
}

vector<int> preorderTraversalRecursive(TreeNode* root) {
    vector<int> result;
    preorder(root, result);
    return result;
}

// Iterative preorder traversal
vector<int> preorderTraversalIterative(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    stack<TreeNode*> s;
    s.push(root);

    while (!s.empty()) {
        TreeNode* node = s.top();
        s.pop();
        result.push_back(node->val);

        if (node->right) s.push(node->right);
        if (node->left) s.push(node->left);
    }

    return result;
}

int main() {
    // Example tree: [1, null, 2, 3]
    TreeNode* root = new TreeNode(1);
    root->right = new TreeNode(2);
    root->right->left = new TreeNode(3);

    vector<int> resultRecursive = preorderTraversalRecursive(root);
    vector<int> resultIterative = preorderTraversalIterative(root);

    cout << "Preorder Traversal (Recursive): ";
    for (int val : resultRecursive) cout << val << " ";
    cout << endl;

    cout << "Preorder Traversal (Iterative): ";
    for (int val : resultIterative) cout << val << " ";
    cout << endl;

    return 0;
}
```

OUTPUT:



```
Preorder Traversal (Recursive): 1 2 3
Preorder Traversal (Iterative): 1 2 3
```

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.

Example 1:
Input: root = [1, 2, 3, 4, 5, null, 6]
Output: 21
Explanation: The sum of all nodes is 1 + 2 + 3 + 4 + 5 + 6 = 21.

Example 2:
Input: root = [5, 2, 6, 1, 3, 4, 7]
Output: 28
Explanation: The sum of all nodes is 5 + 2 + 6 + 1 + 3 + 4 + 7 = 28.

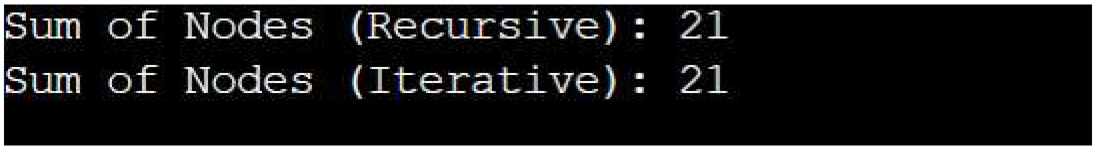
Reference: http://leetcode.com/problems/sum-of-left-leaves/

CODE:

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

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

// Recursive function to calculate sum
int sumOfNodesRecursive(TreeNode* root) {
    if (!root) return 0;
    return root->val + sumOfNodesRecursive(root->left) + sumOfNodesRecursive(root->right);
}

// Iterative function to calculate sum
```

```
int sumOfNodesIterative(TreeNode* root) {
    if (!root) return 0;

    int sum = 0;
    stack<TreeNode*> s;
    s.push(root);

    while (!s.empty()) {
        TreeNode* node = s.top();
        s.pop();
        sum += node->val;

        if (node->right) s.push(node->right);
        if (node->left) s.push(node->left);
    }

    return sum;
}

int main() {
    // Example tree: [1, 2, 3, 4, 5, null, 6]
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->left = new TreeNode(4);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(6);

    cout << "Sum of Nodes (Recursive): " << sumOfNodesRecursive(root) << endl; // Output: 21
    cout << "Sum of Nodes (Iterative): " << sumOfNodesIterative(root) << endl; // Output: 21

    return 0;
}
```

OUTPUT:



```
Sum of Nodes (Recursive): 21
Sum of Nodes (Iterative): 21
```

Easy:

1. Same Tree

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

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

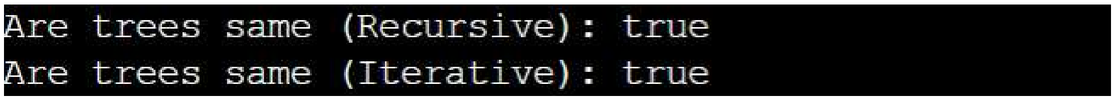
Example 2:
Input: p = [1,2], q = [1,null,2]
Output: false

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

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

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

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

// Recursive function to check if two trees are the same
bool isSameTreeRecursive(TreeNode* p, TreeNode* q) {
    if (!p && !q) return true;
    if (!p || !q) return false;
    if (p->val != q->val) return false;
    return isSameTreeRecursive(p->left, q->left) && isSameTreeRecursive(p->right, q->right);
}

// Iterative function to check if two trees are the same
bool isSameTreeIterative(TreeNode* p, TreeNode* q) {
    if (!p && !q) return true;
    if (!p || !q) return false;

    queue<TreeNode*> queueP, queueQ;
    queueP.push(p);
    queueQ.push(q);

    while (!queueP.empty() && !queueQ.empty()) {
        TreeNode* nodeP = queueP.front(); queueP.pop();
        TreeNode* nodeQ = queueQ.front(); queueQ.pop();
        if (!nodeP || !nodeQ || nodeP->val != nodeQ->val) return false;

        if (nodeP->left || nodeQ->left) {
            if (!nodeP->left || !nodeQ->left) return false;
            queueP.push(nodeP->left);
            queueQ.push(nodeQ->left);
        }

        if (nodeP->right || nodeQ->right) {
            if (!nodeP->right || !nodeQ->right) return false;
            queueP.push(nodeP->right);
            queueQ.push(nodeQ->right);
        }
    }
    return queueP.empty() && queueQ.empty();
}

int main() {
    // Example 1: p = [1, 2, 3], q = [1, 2, 3]
    TreeNode* p = new TreeNode(1);
    p->left = new TreeNode(2);
    p->right = new TreeNode(3);

    TreeNode* q = new TreeNode(1);
    q->left = new TreeNode(2);
    q->right = new TreeNode(3);

    cout << "Are trees same (Recursive): " << (isSameTreeRecursive(p, q) ? "true" : "false") << endl;
// Output: true
    cout << "Are trees same (Iterative): " << (isSameTreeIterative(p, q) ? "true" : "false") << endl; //
Output: true

    return 0;
}
```

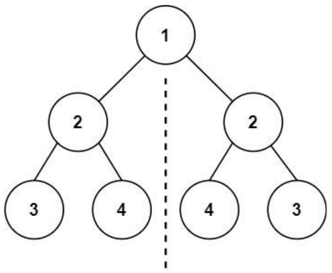
OUTPUT:



```
Are trees same (Recursive): true
Are trees same (Iterative): true
```

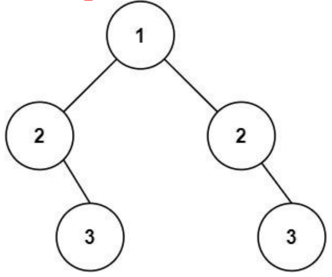
2. Symmetric Tree

Example 1:



Input: root = [1,2,2,3,4,4,3]
Output: true

Example 2:



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

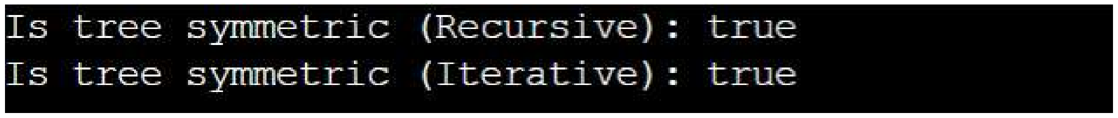
Constraints:
The number of nodes in the tree is in the range [1, 1000].
-100 <= Node.val <= 100
Reference: https://leetcode.com/problems/symmetric-tree/description/

CODE:

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

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

// Recursive helper to check symmetry
bool isMirror(TreeNode* t1, TreeNode* t2) {
    if (!t1 && !t2) return true;
    if (!t1 || !t2) return false;
    return (t1->val == t2->val) &&
           isMirror(t1->left, t2->right) &&
           isMirror(t1->right, t2->left);
}
```

```
// Recursive solution
bool isSymmetricRecursive(TreeNode* root) {
    if (!root) return true;
    return isMirror(root->left, root->right);
}

// Iterative solution
bool isSymmetricIterative(TreeNode* root) {
    if (!root) return true;

    queue<TreeNode*> q;
    q.push(root->left);
    q.push(root->right);

    while (!q.empty()) {
        TreeNode* t1 = q.front(); q.pop();
        TreeNode* t2 = q.front(); q.pop();

        if (!t1 && !t2) continue;
        if (!t1 || !t2 || t1->val != t2->val) return false;

        q.push(t1->left);
        q.push(t2->right);
        q.push(t1->right);
        q.push(t2->left);
    }

    return true;
}

int main() {
    // Example 1: root = [1, 2, 2, 3, 4, 4, 3]
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(2);
    root->left->left = new TreeNode(3);
    root->left->right = new TreeNode(4);
    root->right->left = new TreeNode(4);
    root->right->right = new TreeNode(3);

    cout << "Is tree symmetric (Recursive): " << (isSymmetricRecursive(root) ? "true" : "false") <<
endl;
    cout << "Is tree symmetric (Iterative): " << (isSymmetricIterative(root) ? "true" : "false") << endl;

    return 0;
}
```

OUTPUT:

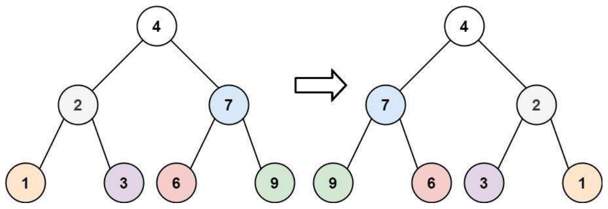


```
Is tree symmetric (Recursive): true
Is tree symmetric (Iterative): true
```

3. Invert Binary Tree

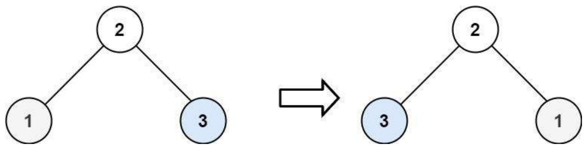
Given the root of a binary tree, invert the tree, and return its root.

Example 1:



Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]

Example 2:



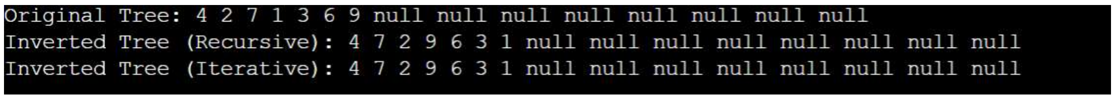
Input: root = [2,1,3]
Output: [2,3,1]

Constraints:
The number of nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100
Refrence: http://leetcode.com/problems/invert-binary-tree/

CODE:

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

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

// Recursive inversion
TreeNode* invertTreeRecursive(TreeNode* root) {
    if (!root) return nullptr;
    TreeNode* temp = root->left;
    root->left = root->right;
    root->right = temp;

    invertTreeRecursive(root->left);
    invertTreeRecursive(root->right);

    return root;
}

// Iterative inversion
TreeNode* invertTreeIterative(TreeNode* root) {
    if (!root) return nullptr;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();

        TreeNode* temp = current->left;
        current->left = current->right;
        current->right = temp;

        if (current->left) q.push(current->left);
        if (current->right) q.push(current->right);
    }

    return root;
}

// Helper function to print the tree (level-order traversal)
void printTree(TreeNode* root) {
    if (!root) return;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();

        if (current) {
            cout << current->val << " ";
            q.push(current->left);
            q.push(current->right);
        } else {
            cout << "null ";
        }
    }
    cout << endl;
}

int main() {
    // Example: root = [4,2,7,1,3,6,9]
    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);

    cout << "Original Tree: ";
    printTree(root);

    // Recursive inversion
    TreeNode* invertedRecursive = invertTreeRecursive(root);
    cout << "Inverted Tree (Recursive): ";
    printTree(invertedRecursive);

    // Reset tree for iterative example
    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);

    // Iterative inversion
    TreeNode* invertedIterative = invertTreeIterative(root);
    cout << "Inverted Tree (Iterative): ";
    printTree(invertedIterative);

    return 0;
}
```

OUTPUT:



```
Original Tree: 4 2 7 1 3 6 9 null null null null null null null null
Inverted Tree (Recursive): 4 7 2 9 6 3 1 null null null null null null null null
Inverted Tree (Iterative): 4 7 2 9 6 3 1 null null null null null null null null
```

4. Leaf Nodes of a Binary Tree

Given a Binary Tree, the task is to count leaves in it. A node is a leaf node if both left and right child nodes of it are NULL.

Examples:
Input:

Output: 3
Explanation: Three leaf nodes are 3, 4 and 5 as both of their left and right child is NULL.

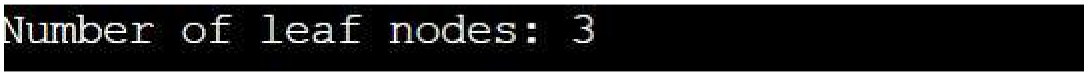
Input:

Output: 3
Explanation: Three leaf nodes are 4, 6 and 7 as both of their left and right child is NULL.

Refrence:http://practice.geeksforgeeks.org/problems/count-leaves-in-binary-tree/1

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

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

int countLeaves(TreeNode* root) {
    if (!root) return 0;

    int leafCount = 0;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();

        // Check if current node is a leaf node
        if (!current->left && !current->right) {
            leafCount++;
        }

        // Add the left and right children to the queue if they exist
        if (current->left) q.push(current->left);
        if (current->right) q.push(current->right);
    }

    return leafCount;
}

int main() {
    // Example 1: root = [1, 2, 3, 4, 5]
    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);

    cout << "Number of leaf nodes: " << countLeaves(root) << endl;

    return 0;
}
```

OUTPUT:

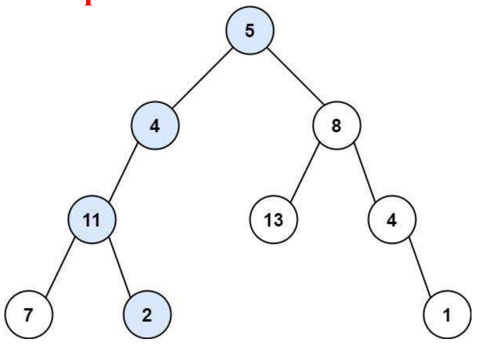


```
Number of leaf nodes: 3
```

5. Path Sum

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

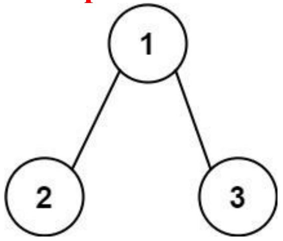
Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

Output: true
Explanation: The root-to-leaf path with the target sum is shown.

Example 2:



Input: root = [1,2,3], targetSum = 5
Output: false
Explanation: There are two root-to-leaf paths in the tree:
(1 --> 2): The sum is 3.
(1 --> 3): The sum is 4.
There is no root-to-leaf path with sum = 5.

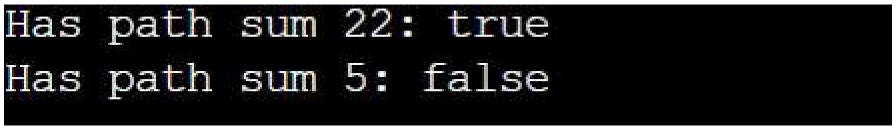
Example 3:
Input: root = [], targetSum = 0
Output: false
Explanation: Since the tree is empty, there are no root-to-leaf paths.

Reference: http://leetcode.com/problems/path-sum/

CODE:
```
#include <iostream>
using namespace std;

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

bool hasPathSum(TreeNode* root, int targetSum) {
    if (!root) return false;  // If the tree is empty, return false

    // If it's a leaf node, check if the remaining target sum equals the current node's value
    if (!root->left && !root->right) {
        return root->val == targetSum;
    }

    // Recursively check the left and right subtrees with the reduced target sum
    targetSum -= root->val;
    return hasPathSum(root->left, targetSum) || hasPathSum(root->right, targetSum);
}
```

```
int main() {
    // Example 1: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(4);
    root->right = new TreeNode(8);
    root->left->left = new TreeNode(11);
    root->right->left = new TreeNode(13);
    root->right->right = new TreeNode(4);
    root->left->left->left = new TreeNode(7);
    root->left->left->right = new TreeNode(2);
    root->right->right->right = new TreeNode(1);

    int targetSum = 22;
    cout << "Has path sum 22: " << (hasPathSum(root, targetSum) ? "true" : "false") << endl;

    // Example 2: root = [1,2,3], targetSum = 5
    TreeNode* root2 = new TreeNode(1);
    root2->left = new TreeNode(2);
    root2->right = new TreeNode(3);

    targetSum = 5;
    cout << "Has path sum 5: " << (hasPathSum(root2, targetSum) ? "true" : "false") << endl;

    return 0;
}
```

OUTPUT:



```
Has path sum 22: true
Has path sum 5: false
```

Medium:

1. Construct Binary Tree from Preorder and Inorder Traversal

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

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

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

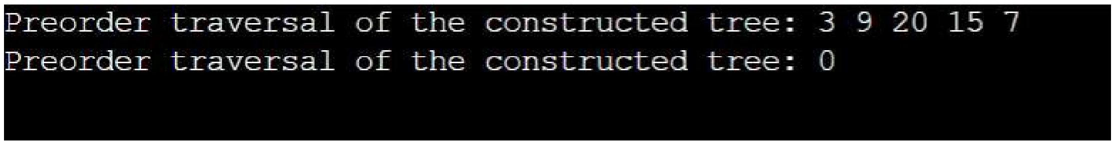
Constraints:
1 <= preorder.length <= 3000

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

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

CODE:
```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

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

class Solution {
public:
    unordered_map<int, int> inorder_map;  // To store inorder indices for quick lookup
    int preorder_index = 0;  // To track the current index in the preorder array

    TreeNode* buildTreeHelper(const vector<int>& preorder, const vector<int>& inorder, int left, int
right) {
        if (left > right) {
            return nullptr;
        }

        // Get the root value from the preorder array
        int root_val = preorder[preorder_index++];

        // Create the root node
        TreeNode* root = new TreeNode(root_val);

        // Find the index of the root in inorder traversal
        int inorder_index = inorder_map[root_val];

        // Recursively build the left and right subtrees
        root->left = buildTreeHelper(preorder, inorder, left, inorder_index - 1);
        root->right = buildTreeHelper(preorder, inorder, inorder_index + 1, right);

        return root;
    }
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        // Fill the inorder map with the value and its index
        for (int i = 0; i < inorder.size(); ++i) {
            inorder_map[inorder[i]] = i;
        }

        // Start the recursive tree construction
        return buildTreeHelper(preorder, inorder, 0, inorder.size() - 1);
    }
};

void preorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

int main() {
    Solution solution;

    // Example 1
    vector<int> preorder = {3, 9, 20, 15, 7};
    vector<int> inorder = {9, 3, 15, 20, 7};

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

    cout << "Preorder traversal of the constructed tree: ";
    preorderTraversal(root);
    cout << endl;  // Output should be: 3 9 20 15 7

    // Example 2
    vector<int> preorder2 = {-1};
    vector<int> inorder2 = {-1};

    TreeNode* root2 = solution.buildTree(preorder2, inorder2);

    cout << "Preorder traversal of the constructed tree: ";
    preorderTraversal(root2);
    cout << endl;  // Output should be: -1

    return 0;
}
```

OUTPUT:


```
Preorder traversal of the constructed tree: 3 9 20 15 7
Preorder traversal of the constructed tree: 0
```

2. Construct Binary Tree from Inorder and Postorder Traversal

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

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

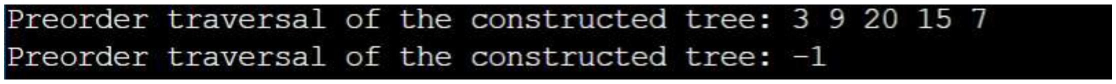
Example 2:
Input: inorder = [-1], postorder = [-1]
Output: [-1]

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

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

CODE:
```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

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

class Solution {
public:
    unordered_map<int, int> inorder_map;  // To store inorder indices for quick lookup
    int postorder_index;  // To track the current index in the postorder array

    TreeNode* buildTreeHelper(const vector<int>& inorder, const vector<int>& postorder, int left,
int right) {
        if (left > right) {
            return nullptr;
        }

        // The root value is at the current postorder index (start from the end)
        int root_val = postorder[postorder_index--];

        // Create the root node
        TreeNode* root = new TreeNode(root_val);

        // Find the index of the root in inorder traversal
        int inorder_index = inorder_map[root_val];

        // Recursively build the right and left subtrees
        root->right = buildTreeHelper(inorder, postorder, inorder_index + 1, right);
        root->left = buildTreeHelper(inorder, postorder, left, inorder_index - 1);

        return root;
    }

    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        postorder_index = postorder.size() - 1;

        // Fill the inorder map with the value and its index
        for (int i = 0; i < inorder.size(); ++i) {
            inorder_map[inorder[i]] = i;
        }

        // Start the recursive tree construction
        return buildTreeHelper(inorder, postorder, 0, inorder.size() - 1);
    }
};

void preorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

int main() {
    Solution solution;

    // Example 1
    vector<int> inorder = {9, 3, 15, 20, 7};
    vector<int> postorder = {9, 15, 7, 20, 3};

    TreeNode* root = solution.buildTree(inorder, postorder);
    cout << "Preorder traversal of the constructed tree: ";
    preorderTraversal(root);
    cout << endl;  // Output should be: 3 9 20 15 7

    // Example 2
    vector<int> inorder2 = {-1};
    vector<int> postorder2 = {-1};

    TreeNode* root2 = solution.buildTree(inorder2, postorder2);

    cout << "Preorder traversal of the constructed tree: ";
    preorderTraversal(root2);
    cout << endl;  // Output should be: -1

    return 0;
}
```

OUTPUT:



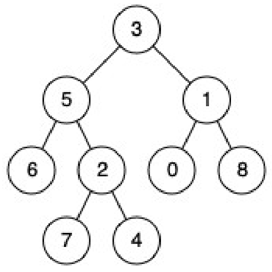
```
Preorder traversal of the constructed tree: 3 9 20 15 7
Preorder traversal of the constructed tree: -1
```

3. Lowest Common Ancestor of a Binary Tree

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

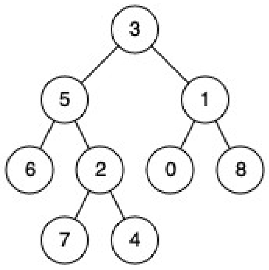
The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).

Example 1:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

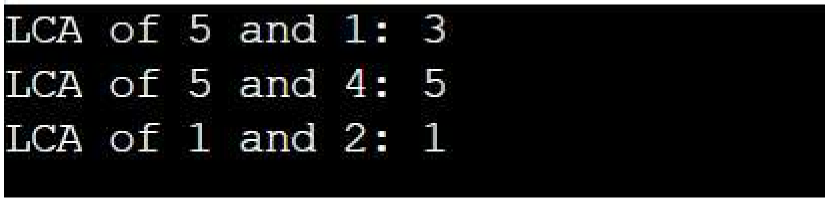
Example 3:
Input: root = [1,2], p = 1, q = 2
Output: 1

Constraints:
The number of nodes in the tree is in the range [2, 105].
-109 <= Node.val <= 109
All Node.val are unique.
p != q
p and q will exist in the tree.

Reference: **https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/description/?envType=study-plan-v2&envId=top-interview-150**

CODE:
```
#include <iostream>
using namespace std;

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

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // Base case: If the root is null or if we find either p or q
        if (root == nullptr || root == p || root == q) {
            return root;
        }

        // Recur for left and right subtrees
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);
        // If both left and right are non-null, root is the LCA
        if (left && right) {
            return root;
        }

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

// Helper function to create a tree node
TreeNode* createNode(int value) {
    return new TreeNode(value);
}

int main() {
    Solution solution;

    // Example 1
    TreeNode* root1 = createNode(3);
    root1->left = createNode(5);
    root1->right = createNode(1);
    root1->left->left = createNode(6);
    root1->left->right = createNode(2);
    root1->left->right->left = createNode(7);
    root1->left->right->right = createNode(4);
    root1->right->left = createNode(0);
    root1->right->right = createNode(8);

    TreeNode* p1 = root1->left;  // Node 5
    TreeNode* q1 = root1->right; // Node 1

    TreeNode* lca1 = solution.lowestCommonAncestor(root1, p1, q1);
    cout << "LCA of 5 and 1: " << lca1->val << endl; // Expected output: 3

    // Example 2
    TreeNode* p2 = root1->left;  // Node 5
    TreeNode* q2 = root1->left->right->right; // Node 4

    TreeNode* lca2 = solution.lowestCommonAncestor(root1, p2, q2);
    cout << "LCA of 5 and 4: " << lca2->val << endl; // Expected output: 5

    // Example 3
    TreeNode* root3 = createNode(1);
    root3->left = createNode(2);

    TreeNode* p3 = root3;  // Node 1
    TreeNode* q3 = root3->left; // Node 2
    TreeNode* lca3 = solution.lowestCommonAncestor(root3, p3, q3);
    cout << "LCA of 1 and 2: " << lca3->val << endl; // Expected output: 1

    return 0;
}
```

OUTPUT:



```
LCA of 5 and 1: 3
LCA of 5 and 4: 5
LCA of 1 and 2: 1
```

Hard :

1. Populating Next Right Pointers in Each Node

Given a binary tree

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

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

Initially, all next pointers are set to NULL.

Example 1:

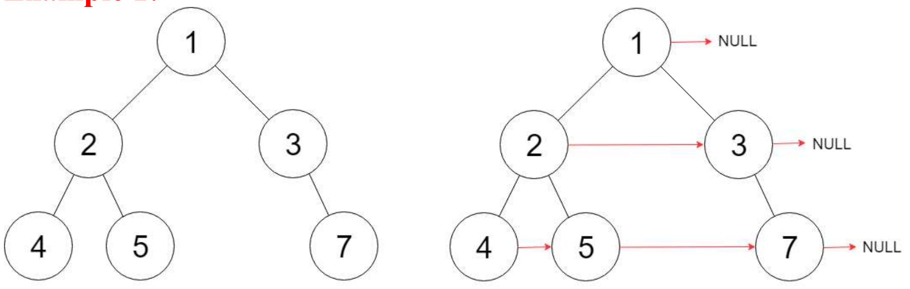


Figure A Figure B

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

Example 2:
Input: root = []Output: []

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

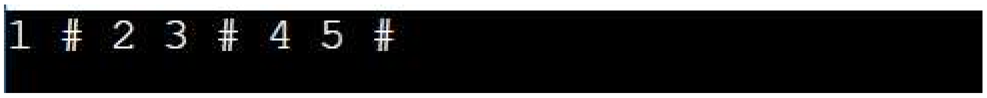
Follow-up:
You may only use constant extra space.
The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.

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

CODE:

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

struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
    Node(int x) : val(x), left(nullptr), right(nullptr), next(nullptr) {}
};

class Solution {
public:
    void connect(Node* root) {
        if (!root) return; // If the tree is empty, no need to do anything

        // Start with the root node
        Node* levelStart = root;

        // Traverse level by level
        while (levelStart) {
            // Traverse the nodes in the current level
            Node* current = levelStart;
            while (current) {
                // If there are left and right children, connect them
                if (current->left) {
                    current->left->next = current->right;
                }
                if (current->right && current->next) {
                    current->right->next = current->next->left;
                }
                // Move to the next node at the current level
                current = current->next;
                }
                // Move to the next level
                levelStart = levelStart->left;
            }
        }
};

// Helper function to print the level order traversal using next pointers
void printLevelOrder(Node* root) {
    while (root) {
        Node* current = root;
        while (current) {
            cout << current->val << " ";
            current = current->next;
        }
        cout << "# ";  // Mark the end of level
        root = root->left;
    }
    cout << endl;
}

int main() {
    Solution solution;

    // Example 1
    Node* root1 = new Node(1);
    root1->left = new Node(2);
    root1->right = new Node(3);
    root1->left->left = new Node(4);
    root1->left->right = new Node(5);
    root1->right->right = new Node(7);

    solution.connect(root1);
    printLevelOrder(root1);  // Expected output: 1 # 2 3 # 4 5 7 #

    // Example 2 (empty tree)
    Node* root2 = nullptr;
    solution.connect(root2);
    printLevelOrder(root2);  // Expected output: (nothing)

    return 0;
}
```

OUTPUT:

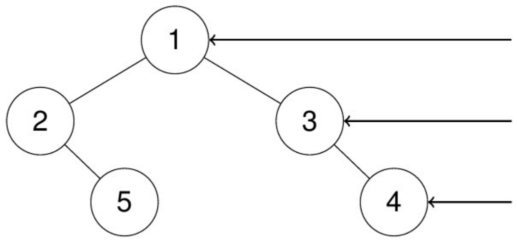


```
1 # 2 3 # 4 5 #
```

2. Binary Tree Right Side View

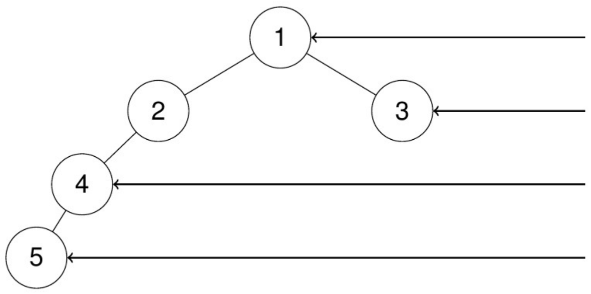
Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Example 1:
Input: root = [1,2,3,null,5,null,4]
Output: [1,3,4]



Explanation:

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



Example 3:
Input: root = [1,null,3]
Output: [1,3]
Example 4:
Input: root = []
Output: []

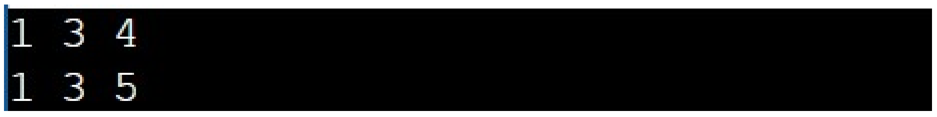
Constraints:
The number of nodes in the tree is in the range [0, 100].
-100 <= Node.val <= 100

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

CODE:

```
#include <iostream>
#include <queue>
#include <vector>
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:
    vector<int> rightSideView(TreeNode* root) {
        vector<int> result;
        if (!root) return result;

        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            int size = q.size();  // Number of nodes at current level
            for (int i = 0; i < size; ++i) {
                TreeNode* node = q.front(); q.pop();
                // Record the value of the last node in the current level
                if (i == size - 1) {
                    result.push_back(node->val);
                }
                // Add left and right children of the node to the queue
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }
        }
        return result;
    }
};

int main() {
    Solution solution;

    // Example 1
    TreeNode* root1 = new TreeNode(1);
    root1->right = new TreeNode(3);
    root1->left = new TreeNode(2);
    root1->left->right = new TreeNode(5);
    root1->right->right = new TreeNode(4);
    vector<int> result1 = solution.rightSideView(root1);
    for (int val : result1) {
        cout << val << " ";
    }
    cout << endl;  // Output: [1, 3, 4]

    // Example 2
    TreeNode* root2 = new TreeNode(1);
    root2->left = new TreeNode(2);
    root2->right = new TreeNode(3);
    root2->left->left = new TreeNode(4);
    root2->right->right = new TreeNode(5);

    vector<int> result2 = solution.rightSideView(root2);
    for (int val : result2) {
        cout << val << " ";
    }
    cout << endl;  // Output: [1, 3, 5]

    return 0;
}
```

OUTPUT:

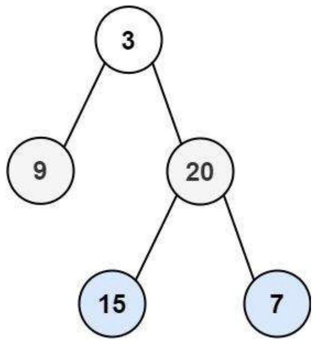


```
1 3 4
1 3 5
```

3. Binary Tree Zigzag Level Order Traversal

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

Example 1:



Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]

Example 2:
Input: root = [1]
Output: [[1]]

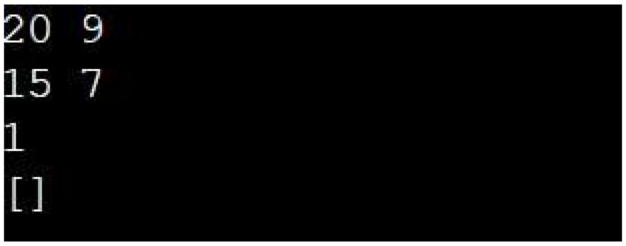
Example 3:
Input: root = []
Output: []

Constraints:
The number of nodes in the tree is in the range [0, 2000].
-100 <= Node.val <= 100

Reference: https://leetcode.com/problems/binary-tree-zigzag-level-order-traversal/description/?envType=study-plan-v2&envId=top-interview-150

CODE:
```
#include <iostream>
#include <queue>
#include <vector>
#include <deque>  // For using deque for reverse insertion
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:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result;

        queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true;  // Flag to track the direction

        while (!q.empty()) {
            int size = q.size();
            vector<int> currentLevel;
            for (int i = 0; i < size; ++i) {
                TreeNode* node = q.front();
                q.pop();

                // Add node's value to current level
                if (leftToRight) {
                    currentLevel.push_back(node->val);
                } else {
                    currentLevel.insert(currentLevel.begin(), node->val);  // Insert at the beginning for right
to left order
                }

                // Add the child nodes to the queue for the next level
                if (node->left) q.push(node->left);
                if (node->right) q.push(node->right);
            }

            // After processing the current level, toggle the direction
            leftToRight = !leftToRight;
            result.push_back(currentLevel);
        }

        return result;
    }
};

int main() {
    Solution solution;

    // Example 1: Input: [3,9,20,null,null,15,7]
    TreeNode* root1 = new TreeNode(3);
    root1->left = new TreeNode(9);
    root1->right = new TreeNode(20);
    root1->right->left = new TreeNode(15);
    root1->right->right = new TreeNode(7);

    vector<vector<int>> result1 = solution.zigzagLevelOrder(root1);
    for (const auto& level : result1) {
        for (int val : level) {
            cout << val << " ";
        }
        cout << endl;
    }

    // Example 2: Input: [1]
    TreeNode* root2 = new TreeNode(1);
    vector<vector<int>> result2 = solution.zigzagLevelOrder(root2);
    for (const auto& level : result2) {
        for (int val : level) {
            cout << val << " ";
        }
        cout << endl;
    }

    // Example 3: Input: []
    TreeNode* root3 = nullptr;
    vector<vector<int>> result3 = solution.zigzagLevelOrder(root3);
    if (result3.empty()) {
        cout << "[]\n";
    }

    return 0;
}
```

OUTPUT:



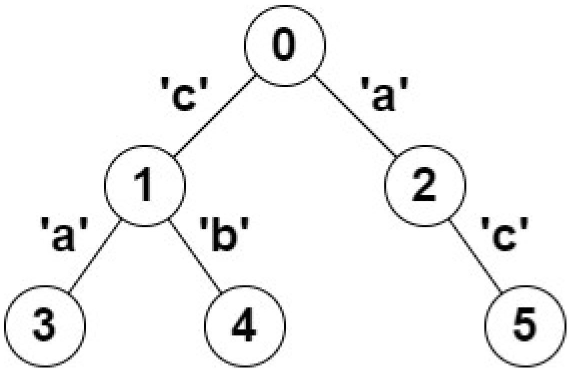
```
20 9
15 7
1
[]
```

Very Hard :

1. Count Paths That Can Form a Palindrome in a Tree

You are given a tree (i.e. a connected, undirected graph that has no cycles) rooted at node 0 consisting of n nodes numbered from 0 to n - 1. The tree is represented by a 0-indexed array parent of size n, where parent[i] is the parent of node i. Since node 0 is the root, parent[0] == -1.
You are also given a string s of length n, where s[i] is the character assigned to the edge between i and parent[i]. s[0] can be ignored.
Return the number of pairs of nodes (u, v) such that u < v and the characters assigned to edges on the path from u to v can be rearranged to form a palindrome.
A string is a palindrome when it reads the same backwards as forwards.

Example 1:



Input: parent = [-1,0,0,1,1,2], s = "acaabc"
Output: 8
Explanation: The valid pairs are:
- All the pairs (0,1), (0,2), (1,3), (1,4) and (2,5) result in one character which is always a palindrome.
- The pair (2,3) result in the string "aca" which is a palindrome.
- The pair (1,5) result in the string "cac" which is a palindrome.
- The pair (3,5) result in the string "acac" which can be rearranged into the palindrome "acca".

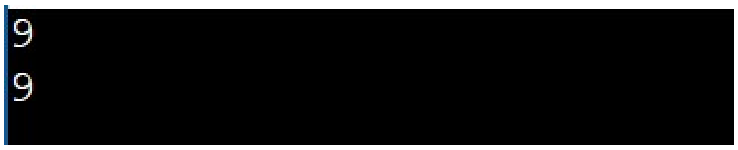
Example 2:
Input: parent = [-1,0,0,0,0], s = "aaaaa"
Output: 10
Explanation: Any pair of nodes (u,v) where u < v is valid.

Constraints:
n == parent.length == s.length
1 <= n <= 105
0 <= parent[i] <= n - 1 for all i >= 1
parent[0] == -1
parent represents a valid tree.
s consists of only lowercase English letters.

Reference : https://leetcode.com/problems/count-paths-that-can-form-a-palindrome-in-a-tree/description/?envType=problem-list-v2&envId=tree

CODE:
```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    int countPalindromePaths(vector<int>& parent, string& s) {
        int n = parent.size();
        int result = 0;

        // To store the frequency XOR for each node
        unordered_map<int, int> freqXorCount;

        // Initially, the XOR of an empty path is 0, which is the XOR for the root.
        freqXorCount[0] = 1;

        // DFS function to traverse the tree and calculate XORs
        dfs(parent, s, 0, 0, result, freqXorCount);

        return result;
    }

private:
    void dfs(const vector<int>& parent, const string& s, int node, int xorValue, int& result,
unordered_map<int, int>& freqXorCount) {
        // Update the current xorValue with the current node's character
        xorValue ^= (1 << (s[node] - 'a'));

        // Check if the current xorValue has been encountered before
        // If it has, it means there are valid paths forming a palindrome
        result += freqXorCount[xorValue];

        // Try all possible single-bit toggles (i.e., changing one character's count)
        for (int i = 0; i < 26; ++i) {
            result += freqXorCount[xorValue ^ (1 << i)];
        }

        // Record the current xorValue
        freqXorCount[xorValue]++;

        // Recursively traverse the child nodes
        for (int i = 0; i < parent.size(); ++i) {
            if (parent[i] == node) {
                dfs(parent, s, i, xorValue, result, freqXorCount);
            }
        }

        // Backtrack: undo the current xorValue count
        freqXorCount[xorValue]--;
    }
};

int main() {
    Solution solution;

    vector<int> parent1 = {-1, 0, 0, 1, 1, 2};
    string s1 = "acaabc";
    cout << solution.countPalindromePaths(parent1, s1) << endl; // Output: 8

    vector<int> parent2 = {-1, 0, 0, 0, 0};
    string s2 = "aaaaa";
    cout << solution.countPalindromePaths(parent2, s2) << endl; // Output: 10

    return 0;
}
```

OUTPUT:

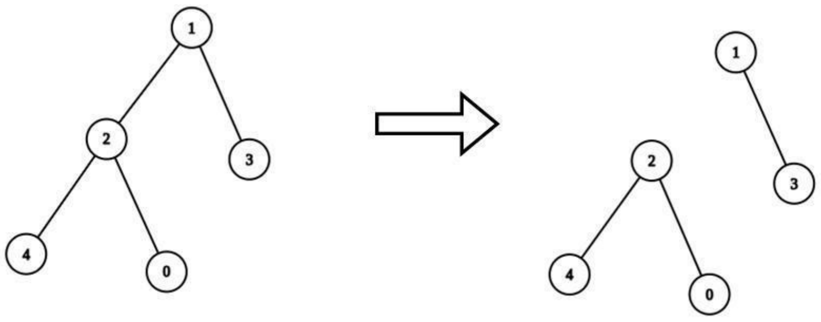


```
9
9
```

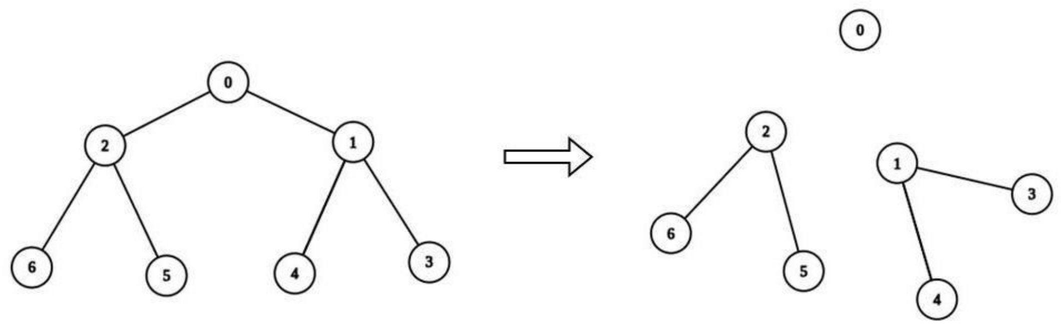
2. Maximum Number of K-Divisible Components

There is an undirected tree with n nodes labeled from 0 to n - 1. You are given the integer n and a 2D integer array edges of length n - 1, where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the tree.
You are also given a 0-indexed integer array values of length n, where values[i] is the value associated with the ith node, and an integer k.
A valid split of the tree is obtained by removing any set of edges, possibly empty, from the tree such that the resulting components all have values that are divisible by k, where the value of a connected component is the sum of the values of its nodes.
Return the maximum number of components in any valid split.

Example 1:



Input: n = 5, edges = [[0,2],[1,2],[1,3],[2,4]], values = [1,8,1,4,4], k = 6
Output: 2
Explanation: We remove the edge connecting node 1 with 2. The resulting split is valid because:
- The value of the component containing nodes 1 and 3 is values[1] + values[3] = 12.
- The value of the component containing nodes 0, 2, and 4 is values[0] + values[2] + values[4] = 6.
It can be shown that no other valid split has more than 2 connected components.
Example 2:



Input: n = 7, edges = [[0,1],[0,2],[1,3],[1,4],[2,5],[2,6]], values = [3,0,6,1,5,2,1], k = 3
Output: 3
Explanation: We remove the edge connecting node 0 with 2, and the edge connecting node 0 with 1. The resulting split is valid because:
- The value of the component containing node 0 is values[0] = 3.
- The value of the component containing nodes 2, 5, and 6 is values[2] + values[5] + values[6] = 9.
- The value of the component containing nodes 1, 3, and 4 is values[1] + values[3] + values[4] = 6.
It can be shown that no other valid split has more than 3 connected components.

Constraints:
1 <= n <= 3 * 104
edges.length == n - 1
edges[i].length == 2
0 <= ai, bi < n
values.length == n
0 <= values[i] <= 109
1 <= k <= 109
Sum of values is divisible by k.
The input is generated such that edges represents a valid tree.

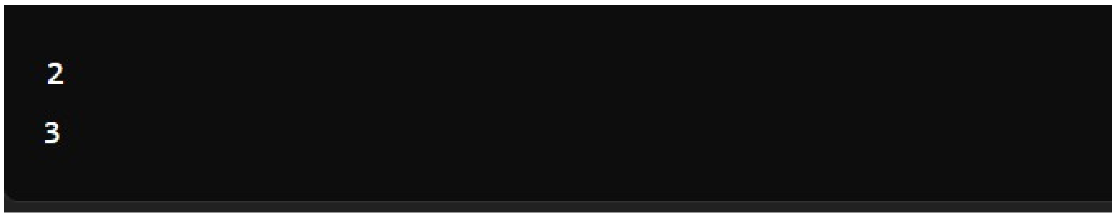
Reference: https://leetcode.com/problems/maximum-number-of-k-divisible-components/description/?envType=problem-list-v2&envId=tree

CODE:

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

class Solution {
public:
    int maximumSplit(int n, vector<vector<int>>& edges, vector<int>& values, int k) {
        // Create adjacency list for the tree
        unordered_map<int, vector<int>> tree;
        for (auto& edge : edges) {
            tree[edge[0]].push_back(edge[1]);
            tree[edge[1]].push_back(edge[0]);
        }

        // Initialize result to count valid splits
        int result = 0;

        // Perform DFS from node 0
        function<int(int, int)> dfs = [&](int node, int parent) -> int {
            int subtree_sum = values[node];

            // Traverse all neighbors (children)
            for (int neighbor : tree[node]) {
                if (neighbor != parent) {
                    // Calculate the subtree sum for the neighbor
                    subtree_sum += dfs(neighbor, node);
                }
            }

            // If subtree sum is divisible by k, increment result and return 0
            if (subtree_sum % k == 0) {
                result++;
                return 0;  // This subtree can be split
            }

            return subtree_sum;  // Otherwise, return the subtree sum
        };

        // Start DFS from the root (node 0)
        dfs(0, -1);

        // Return the result, subtracting 1 because the original root is always counted as one component
        return result - 1;
    }
};

// Example usage
int main() {
    Solution solution;
    vector<vector<int>> edges1 = {{0, 2}, {1, 2}, {1, 3}, {2, 4}};
    vector<int> values1 = {1, 8, 1, 4, 4};
    int k1 = 6;
    cout << solution.maximumSplit(5, edges1, values1, k1) << endl;  // Output: 2

    vector<vector<int>> edges2 = {{0, 1}, {0, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}};
    vector<int> values2 = {3, 0, 6, 1, 5, 2, 1};
    int k2 = 3;
    cout << solution.maximumSplit(7, edges2, values2, k2) << endl;  // Output: 3

    return 0;
```

```
}
```

OUTPUT:



```
2
3
```