Binary Tree Worksheet

# Q1. Problem Statement

Find the height of a binary tree (by defining a recursive function getHeight()) where the height of a binary tree is defined as the number of edges in the longest path from the root node to a leaf node. In other words, it’s the maximum depth of the tree. If a tree has only a root node, its height is 0. A tree node is defined as follows:

**struct** TreeNode

*{*

**int** val ; TreeNode ∗ l e f t ; TreeNode ∗ r i g h t ;

TreeNode ( **int** x) : val ( x ) , l e f t ( n u l l p t r ) , r i g h t ( n u l l p t r ) *{}*

*}* ;

## Example test case

TreeNode∗ root = **new** TreeNode ( 1 ) ; root −*>* l e f t = **new** TreeNode ( 2 ) ;

root −*>*r i g h t = **new** TreeNode ( 3 ) ;

root −*>*l e f t −*>* l e f t = **new** TreeNode ( 4 ) ; root −*>*l e f t −*>*r i g h t = **new** TreeNode ( 5 ) ; root −*>*r ight −*>* l e f t = **new** TreeNode ( 6 ) ;

root −*>*l e f t −*>*l e f t −*>* l e f t = **new** TreeNode ( 7 ) ;

**int** he ight = get Height ( root ) ; *// returns 3*

int getHeight(TreeNode\* root){

//base case: if the node is null, return -1

if(root == nullptr){

return -1;

}

//recursively find the height of the left and right subtrees

int leftHeight = getHeight(root->left);

int rightHeight = getHeight(root->right);

//return the maximum of the two heights

return max(leftHeight, rightHeight) + 1;

}

## Questions

1. Assume that you have a recursive function getHeight(TreeNode\* root) that gets the height of the left and right subtree and calculates the maximum height. What should be the base case for such a recursive approach; i, ii, or iii? Justify your answer.

i ) **i f** ( root == n u l l p t r )

*{*

**return** 0 ;

*}*

i i ) **i f** ( root == n u l l p t r )

*{*

**return** −1;

*}*

When using edges to define height, a single-node tree has a height of 0 because there are no edges below the root. Therefore, an empty tree logically has a height of -1, representing “no edges at all”.

i i i ) **i f** ( root == n u l l p t r )

*{*

**return** 1 ;

*}*

1. How would you get the height of the left and the right subtree? Write the code in C++.

code is shown above

1. How would you choose the maximum between two subtree heights?

You would compare the height of both subtrees and return the larger height.

1. Do you need to maintain a separate variable for calculating the height? Explain.

A separate variable for calculating the height is not necessary because the function relies on recursion to navigate down to the leaf nodes of the tree. Each time we call getHeight on a subtree, we get back a height value that reflects the maximum depth from that node down to its leaf nodes. By using the return values directly from the recursive calls, the function efficiently calculates the height without needing an additional variable to keep track of the height as it goes. Each call inherently computes its height based on the heights of its children.

1. What is the time and space complexity for this approach? Justify your answer. Imagine you have a total of *n* nodes and the tree’s height is *h*.

The time complexity is O(n), since each node is visited exactly once. The space complexity is O(log n) or O(h) because the maximum depth of the recursion is equal to the height of the tree. Since we are working with a binary tree, the tree is balanced which means the height is logn.

# Q2. Problem Statement

You are given the root of a full binary tree with the following properties:

* Leaf nodes have either the value 0 or 1, where 0 represents False and 1 represents True.
* Non-leaf nodes have either the value 2 or 3, where 2 represents the boolean OR and 3 represents the boolean AND.

The evaluation of a node is as follows:

* If the node is a leaf node, the evaluation is the value of the node, i.e. True or False.
* Otherwise, evaluate the node’s two children and apply the boolean oper- ation of its value with the children’s evaluations.

Return the boolean result of evaluating the root node using a recursive func- tion evaluateTree().

A tree node is defined as follows:

**struct** TreeNode

*{*

**int** val ; TreeNode ∗ l e f t ; TreeNode ∗ r i g h t ;

TreeNode ( **int** x) : val ( x ) , l e f t ( n u l l p t r ) , r i g h t ( n u l l p t r ) *{}*

*}* ;

## Example test case

TreeNode∗ root = **new** TreeNode ( 2 ) ; *// OR*

root −*>* l e f t = **new** TreeNode ( 1 ) ; *// True*

root −*>*r i g h t = **new** TreeNode ( 3 ) ; *// AND*

root −*>*r ight −*>* l e f t = **new** TreeNode ( 0 ) ; *// False*

root −*>*r ight −*>*r i g h t = **new** TreeNode ( 1 ) ;

**bool** r e s u l t = evaluate Tree ( root ) ; *// returns True*

## Questions

1. Assume the recursive function evaluateTree(TreeNode\* root) evaluates its left and right subtrees recursively before applying any boolean opera- tion OR or AND according to the value of the node. What should be the base case for such a recursive function? Write the code in C++. Hint: For the base case, we return the value of a leaf node. What can be the value of a leaf node?

bool evaluateTree(TreeNode\* root){

//base case: if the node is a leaf node, return its value as a boolean

if(root->val == 0){

return false;

} else if(root->val == 1){

return true;

}

//recursively evaluate left and right subtrees

bool leftEval = evaluateTree(root->left);

bool rightEval = evaluateTree(root->right);

//apply the boolean operation based on the current node's value

if(root->val == 2){

//OR operation

return leftEval || rightEval;

} else if(root->val == 3){

//AND operation

return leftEval && rightEval;

}

//this line should not be reachd if the tree is defined correctly

return false;

};

1. Which traversal resembles the fact that ”we evaluate a node’s left and right subtrees recursively before applying any boolean operation OR or AND according to the value of the node.” Explain.

The traversal that resembles evaluating anode’s left and right sutrees recursively before applying any Boolean operation is post-order traversal. In post-order traversal, we visit the left subtree first, the the right subtree, and finally the node itself. This is similar to evaluating the left and right children before applying the Boolean operation at the current node.

1. How do we know whether to apply OR or AND according to the value of a node? Get the result of evaluating subtrees recursively and apply the boolean operations in C++.

To determine whether to apply OR or AND based on the value of a node, we check the val attribute of the TreeNode. The value will indicate the operation to perform: if val is 2, we apply the OR operation on the results of the left and right subree evaluations, and if val is 3, we apply the AND operation.

1. Do you need to maintain a separate boolean variable for evaluating the tree? Explain.

You do not need to maintain a separate Boolean variable to evaluate the tree. The evaluation of the tree can be directly derived from the recursive calls to evaluateTree. Each function call returns a Boolean result based on the evaluation of its subtrees. Thus, the Boolean results can be computed and returned immediately based on the operation defined at the node.

1. What is the time and space complexity for this approach? Justify your answer. Imagine you have a total of *n* nodes and the tree’s height is *h*.

The time complexity is O(n), where n is number of nodes. This is because we visit each node exactly once to evaluate it. The space complexity is O(h), where h is the height of the tree. This space is used for the recursion stack to due to the recursive calls.

# Q3. Problem Statement

Given the root of a binary tree, return the sum of every tree node’s tilt using the findTilt() function.

The tilt of a tree node is the absolute difference between the sum of all left subtree node values and all right subtree node values. If a node does not have a left child, then the sum of the left subtree node values is treated as 0. The rule is similar if the node does not have a right child.

A tree node is defined as follows:

**struct** TreeNode

*{*

**int** val ; TreeNode ∗ l e f t ; TreeNode ∗ r i g h t ;

TreeNode ( **int** x) : val ( x ) , l e f t ( n u l l p t r ) , r i g h t ( n u l l p t r ) *{}*

*}* ;

## Example test cases

### 1)



1

2

3

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

### Explanation:

Tilt of node 2: 0 0 = 0 (no children)

*| − |*

Tilt of node 3: 0 0 = 0 (no children)

*| − |*

Tilt of node 1: 2 3 = 1 (left subtree is just left child, so the sum is 2; right subtree is just right child, so the sum is 3)

*| − |*

Sum of every tilt: 0 + 0 + 1 = 1

### 2)



4

2

9

3

5

7

Input: root = [4,2,9,3,5,null,7] Output: 15

### Explanation:

Tilt of node 3: 0 0 = 0 (no children)

*| − |*

Tilt of node 5: 0 0 = 0 (no children)

*| − |*

Tilt of node 7: 0 0 = 0 (no children)

*| − |*

Tilt of node 2: 3 5 = 2 (left subtree is just left child, so sum is 3; right subtree is just right child, so sum is 5)

*| − |*

Tilt of node 9: 0 7 = 7 (no left child, so sum is 0; right subtree is just right child, so sum is 7)

*| − |*

Tilt of node 4: (3 + 5 + 2) (9 + 7) = 10 16 = 6 (left subtree values are 3,

*| − | | − |*

5, and 2, which sums to 10; right subtree values are 9 and 7, which sums to 16)

Sum of every tilt: 0 + 0 + 0 + 2 + 7 + 6 = 15

## Questions

1. Assume we call a recursive function order that takes the root of the tree as the only input argument i.e. order(root). The order() function
   1. recursively calculates the sum of left and right subtrees and then ii) calculates the tilt of the current node. What should be the base case for such a recursive function?

class Solution {

public:

int totalTilt;

Solution() : totalTilt(0) {} //constructor to initialize totalTilt

int order(TreeNode\* root){

if(root == nullptr){

return 0;

}

//calculate the sums of the left and right subtrees

int leftSum = order(root->left);

int rightSum = order(root->right);

//calculate the tilt of the current node

int tilt = abs(leftSum - rightSum);

//update the total tilt

totalTilt += tilt;

//return the sum of values under this node

return leftSum + rightSum + root->val;

}

int findTilt(TreeNode\* root){

totalTilt = 0; //reset totalTilt

order(root); //start the recursion

return totalTilt;

}

};

1. Do you need to maintain a separate int variable for calculating the tilt of the tree? Explain.

You do not need a separate variable for calculating the total tilt of the tree. Instead, you can use a member variable within the class that contains the findTilt function to accumulate the total tilt as you compute it recursively. This way, you can track the total tilt while traversing the tree without the need for separate variables in each recursive call.

1. We can recursively calculate the sum of left and right subtrees in the following way:

**int** left Sum = order ( root −*>* l e f t ) ;

**int** rightSum = order ( root −*>*r i g h t ) ;

How can we calculate the tilt of the current node given the code above? Write in C++. Assume you can use std::abs to calculate the absolute value of an integer.

Given that we can calculate the sums of the left and right subtrees as leftSum and rightSum, the tilt of the current node can be calculated using the absolute difference between these two sums.

1. What is the time and space complexity for this approach? Justify your answer. Imagine you have a total of *n* nodes and the tree’s height is *h*.

The time complexity is O(n) because each node is visited exactly once during the traversal to compute its left and right subtree sums and its tilt. The space complexity is O(h) because this space is used for the recursion stack during the recursive calls.

# Additional Problems

1. Check if a Binary Tree is Balanced.
2. Find the Maximum Depth of a Binary Tree
3. How to add a new node to a binary tree given its parents’ position? ( see binary tree.cpp)
4. How do you remove a node from binary tree given the node is not an external node ( see binary tree.cpp)