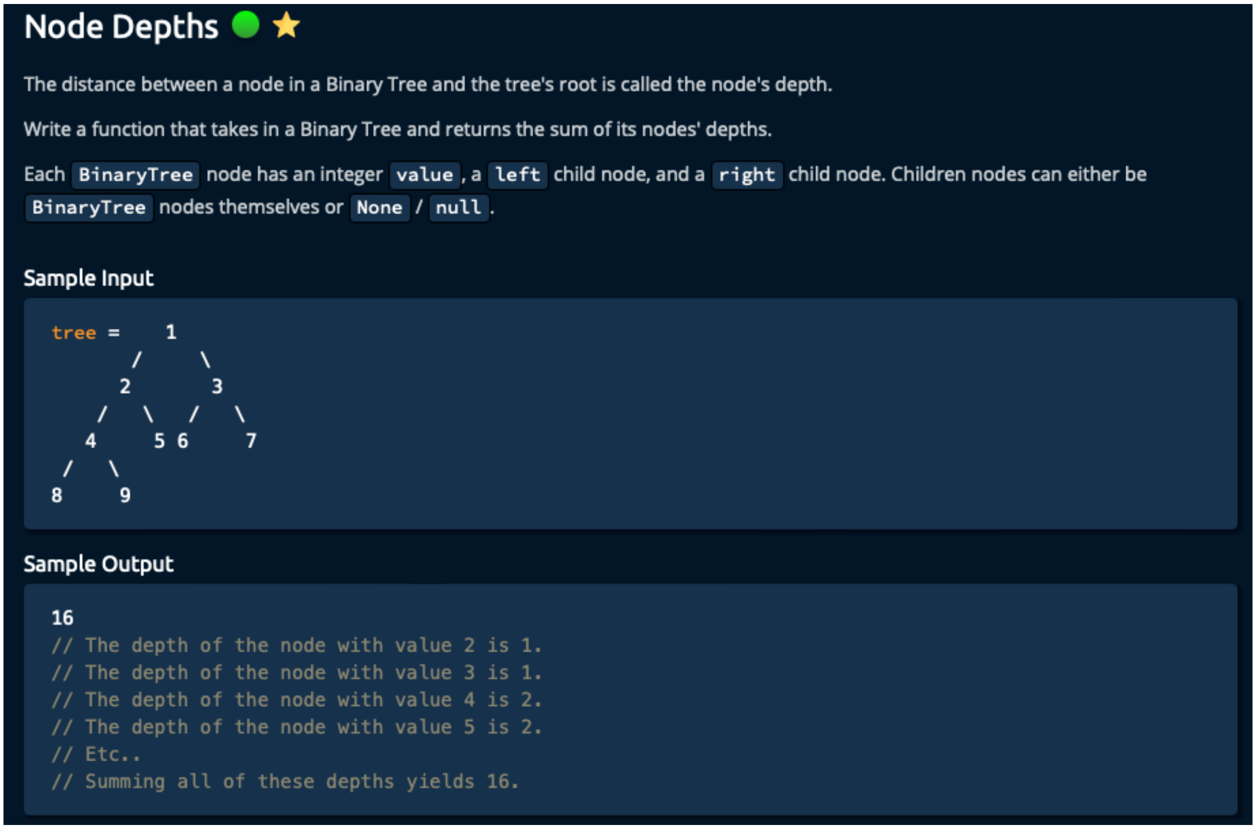
Code Challenge #7 Node Depths (Easy)



Node Depths

The distance between a node in a Binary Tree and the tree's root is called the node's depth.

Write a function that takes in a Binary Tree and returns the sum of its nodes' depths.

Each `BinaryTree` node has an integer `value`, a `left` child node, and a `right` child node. Children nodes can either be `BinaryTree` nodes themselves or `None` / `null`.

Sample Input

```
tree =    1
       /     \
      2       3
    /   \   /   \
   4     5 6     7
 /   \
8     9
```

Sample Output

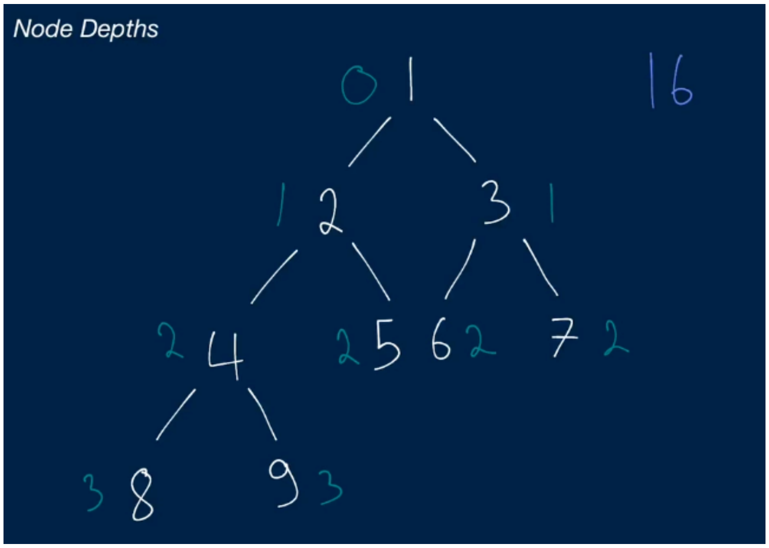
```
16
// The depth of the node with value 2 is 1.
// The depth of the node with value 3 is 1.
// The depth of the node with value 4 is 2.
// The depth of the node with value 5 is 2.
// Etc..
// Summing all of these depths yields 16.
```

Solution #1

```
function nodeDepths(root) {
 let sumOfDepths = 0
  const stack = [{node: root, depth: 0}];
 while (stack.length > 0) {
        const {node, depth} = stack.pop();

        if (node === null) continue;
        sumOfDepths += depth;
        stack.push({node: node.left, depth: depth + 1});
        stack.push({node: node.right, depth: depth + 1})
        }
        return sumOfDepths;
 }

 // This is the class of the input binary tree.
 class BinaryTree {
   constructor(value) {
     this.value = value;
     this.left = null;
     this.right = null;
   }
 }

```

Explanation

This code challenge requires you to add the total number of nodes based on depth of a binary tree. For example, the depth of this binary tree is 16. The depth of a binary tree is the number of levels a node is from the root. The first node is the root node which has a depth of 0 (it is 0 depth from the root since it is the root itself). The second level (depth of 1) contains two items, so we say it is 2 times 1 which is 2. The third level (depth of 2) contains 4 times, so we say it is 4 times 2 which is 8. The fourth level (depth of 3) contains 2 items so we it is 2 times 3 which is 6.



In order to solve this iteratively we will first create a let variable called sumOfDepths = 0. We will then create a const called stack which is an array which initially contains an object where the key value pairs are node: root and depth: 0. We will then create a while loop that runs as long as the stack.length is greater than 0 aka there are items in the stack. Then we will create a const with variables called node and depth using object destructuring and using stack.pop which removes the last item from the stack. If the node === null aka no children node we will continue. We will add depth to the SumOfDepths using +=. We will then push onto stack an object {node: node.left, depth: depth + 1}) . Followed by pushing onto the stack {node: node.right, depth: depth + 1}). We finally return the sumOfDepths. The code runs in O(n) time.

Solution #2

```
function nodeDepths(root, depth = 0) {
  if (root === null) return 0;
  return  depth + nodeDepths(root.left, depth + 1) + nodeDepths(root.right, depth + 1)

}

// This is the class of the input binary tree.
class BinaryTree {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }
}
```

The recursive solution is much simpler. It starts off with two arguments which are root and depth. First we check to see if the root is equal to null. If it is we return 0. If it isn't we return depth plus recursive call on root.left, depth + 1 plus recursive call on root.right, depth + 1. The code runs in O(n) times.