222. Count Complete Tree Nodes

Easy | Bit Manipulation | Tree | Binary Search | Binary Tree | Leetcode Link

Problem Description

The problem provides a complete binary tree and requires us to determine the total number of nodes in the tree. A complete binary tree is a tree where all levels are fully filled except possibly the last level, which is filled from left to right. Such a tree has a special property where the number of nodes at the last level is between 1 and (2^h), where (h) is the height of the last level.

Our task is to find a method to count the nodes of such a binary tree in less than (O(n)) time complexity, where (n) is the number of nodes in the tree.

Intuition

The intuition behind the solution is to leverage the properties of the complete binary tree, which allow us to use a divide and conquer strategy much more efficiently than checking each node. Because in a complete binary tree, either the left or the right subtree is a perfect binary tree, we can use this property to reduce the problem size at each step.

By calculating the depth (height) of the leftmost and rightmost paths separately, we can determine if the last level is completely filled or not.

1. If the left and right depths are the same, it means that the left subtree is a perfect binary tree with (2^{\text{depth}} - 1) nodes, and we only need to count the nodes in the right subtree recursively.
2. If the left and right depths are not the same, it means the last level of the tree is not full, and the right subtree is a perfect binary tree with (2^{\text{right depth}} - 1) nodes, and we count the nodes in the left subtree recursively.

By doing this recursive action, we exponentially decrease the number of nodes we have to visit, thus reducing the time complexity to less than (O(n)).

Solution Approach

The solution uses a recursive approach to solve the problem efficiently. The key is to identify whether the left or right subtree forms a perfect binary tree. To make this identification, we perform the following steps in the given Python code:

1. Define an inner function named `depth` that calculates the depth of a binary tree by traversing leftward until there are no more nodes. It incrementally increases a counter `d` on each iteration, which is returned as the tree's depth from the root to the deepest left leaf node.

2. Check if the given root node is `None` and return 0 because an empty tree has no nodes.

3. Calculate the depths of the left and right subtrees of the root using the `depth` function.

4. If the left and right depths are equal (`left == right`), according to the properties of complete binary trees, the left subtree is perfect and contains `2^left - 1` nodes. We then add 1 (the root node) to this number and recursively count the nodes in the right subtree.

5. If the left and right depths are not equal, the last level is not fully filled, so the perfect subtree is the right subtree with `2^right - 1` nodes. We again add 1 for the root node and recursively count the nodes in the left subtree.

In both cases, the expression `(1 << left)` or `(1 << right)` is used to calculate (2^\text{left}) or (2^\text{right}) respectively. This bit-shift operation is a fast way to compute powers of two.

By calling this recursive solution on appropriate subtrees and keeping track of the number of nodes, the function can calculate the total number of nodes in a complete binary tree efficiently with the time complexity better than (O(n)).

Overall, the implementation utilizes the divide and conquer principle by breaking down the problem into smaller subproblems. It also takes advantage of the characteristics of complete binary trees and utilizes recursion with depth calculation to minimize the nodes visited during the counting process.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Suppose we have the following complete binary tree:

```
       1
      / \
     2   3
    / \  /
   4  5 6
```

We want to find the total number of nodes in this tree efficiently.

1. First, we calculate the depth of the leftmost path. Starting from the root node (1) and moving left, we reach node (4) without any further left child nodes. Thus, the leftmost depth is 3.

2. Then, we calculate the depth of the rightmost path. Again, starting from the root node (1) and this time moving right, we reach node (3) which does not have a right child. Hence, the rightmost depth is also 3.

3. Since the leftmost depth (3) is equal to the rightmost depth (3), we conclude that the left subtree is a perfect binary tree and therefore has (2^{\text{depth}} - 1) nodes, which in this case is (2^3 - 1 = 7) nodes. However, because we only have 5 nodes in the left subtree (nodes 1, 2, 4, and 5), it shows that the tree is not full and the last level is not completely populated.

4. Now, since we established the leftmost and rightmost depths are same, we know that the left subtree is perfect (except for the last level). We count the nodes in this perfect subtree portion, which is (2^2 - 1) because the actual last level is not full, so we take one less depth, giving us (2^2 - 1 = 3) nodes (1, 2, and 5).

5. Next, we add 1 for the root node to the number we just counted, which gives us 4, representing the left subtree including the root.

6. Finally, we move on to the right subtree and recursively apply the same process. The right subtree has a depth of 2, so following the same logic, it has (2^2 - 1 = 3) nodes (nodes 3 and 6, plus the imaginary node at the depth 2 position).

7. By combining the counts of both subtrees and the root, we get (3 + 1 + 3 = 7). However, remember that the right subtree was not perfect and had only 2 nodes (3 and 6), we correct the count to (4 + 2 = 6).

Hence, the tree in our example has a total of 6 nodes. This method, as explained in the solution approach, is more efficient than traversing every node in the tree because it utilizes the properties of a complete binary tree to determine the count of nodes efficiently.

Python Solution

```python
class TreeNode:
    # Initializing the TreeNode structure as described
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def countNodes(self, root: Optional[TreeNode]) -> int:
        # Helper function to calculate the depth of the tree
        def calculate_depth(node):
            depth = 0
            while node:
                depth += 1
                # Move to the next left node to calculate full depth
                node = node.left
            return depth

        # Main function starts here
        # If the root is None, the tree is empty and has no nodes
        if root is None:
            return 0

        # Calculate the depth of the left and right subtrees
        left_depth = calculate_depth(root.left)
        right_depth = calculate_depth(root.right)

        # If the left and right subtrees have the same depth, it means the left subtree is complete
        # We can directly calculate the nodes in the left subtree using the formula (2^depth) - 1
        # and then recursively count the nodes in the right subtree.
        if left_depth == right_depth:
            # Left shift operation (1 << left_depth) calculates 2^left_depth
            # Add the count of nodes in the right subtree.
            return (1 << left_depth) + self.countNodes(root.right)

        # If the left and right subtree depths differ, the right subtree is complete
        # We calculate the nodes for the right subtree using the formula (2^depth) - 1
        # and recursively count the nodes in the left subtree.
        else:
            # Add the count of nodes in the left subtree.
            return (1 << right_depth) + self.countNodes(root.left)

# The TreeNode and Solution classes usage example (the actual nodes should be instantiated with their respective values)
# root = TreeNode(1)
# root.left = TreeNode(2)
# root.right = TreeNode(3)
# root.left.left = TreeNode(4)
# root.left.right = TreeNode(5)
# root.right.left = TreeNode(6)
# solution = Solution()
# number_of_nodes = solution.countNodes(root)
```

Java Solution

```java
// Custom definition for a binary tree node.
class TreeNode {
    int val; // holds the value of the node
    TreeNode left; // reference to the left child
    TreeNode right; // reference to the right child

    // Constructor to initialize the node with no children
    TreeNode() {}

    // Constructor to initialize the node with a specific value
    TreeNode(int val) { this.val = val; }

    // Constructor to initialize the node with a value and references to left and right children
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// Class containing a solution method to count the nodes of a binary tree.
class Solution {

    // Method that returns the count of nodes in a complete binary tree.
    public int countNodes(TreeNode root) {
        // Base case: if the tree is empty, return 0
        if (root == null) {
            return 0;
        }

        // Compute the depth of the left subtree
        int leftDepth = computeDepth(root.left);
        // Compute the depth of the right subtree
        int rightDepth = computeDepth(root.right);

        // Check if the left and right depths are equal
        if (leftDepth == rightDepth) {
            // If equal, the left subtree is complete and we add its node count to the recursive count of the right subtree
            return (1 << leftDepth) + countNodes(root.right);
        } else {
            // If not equal, the right subtree is complete and we add its node count to the recursive count of the left subtree
            return (1 << rightDepth) + countNodes(root.left);
        }
    }

    // Helper method that computes the depth of the tree (distance from the root to the deepest leaf node)
    private int computeDepth(TreeNode root) {
        int depth = 0;
        // Loop to travel down the left edge of the tree until a null is encountered
        for (; root != null; root = root.left) {
            depth++;
        }
        // Return the depth of the tree
        return depth;
    }
}
```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * Counts the number of nodes in a complete binary tree.
     *
     * @param root The root node of the binary tree.
     * @return The total number of nodes in the tree.
     */
    int countNodes(TreeNode* root) {
        if (!root) {
            // If the tree is empty, return a count of 0.
            return 0;
        }

        // Calculate the depth of the left and right subtrees.
        int leftDepth = calculateDepth(root->left);
        int rightDepth = calculateDepth(root->right);

        // If the depths are equal, it means the left subtree is complete and we can use the formula to calculate the number of nod
        if (leftDepth == rightDepth) {
            return (1 << leftDepth) + countNodes(root->right);
        }

        // If the depths are not equal, the right subtree must be complete, and we calculate the number of nodes in the left subtre
        return (1 << rightDepth) + countNodes(root->left);
    }

    /**
     * Calculates the depth of the tree (distance to the leaf) following the left child.
     *
     * @param node The node to measure the depth from.
     * @return The depth of the subtree.
     */
    int calculateDepth(TreeNode* node) {
        int depth = 0;
        while (node) {
            // Move to the left child and increment the depth count.
            node = node->left;
            ++depth;
        }
        return depth;
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node in TypeScript using an interface.
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

// Function to create a new TreeNode instance with default values if not provided.
function createTreeNode(
  value: number = 0,
  leftNode: TreeNode | null = null,
  rightNode: TreeNode | null = null
): TreeNode {
  return { val: value, left: leftNode, right: rightNode };
}

/**
 * Recursively counts the number of nodes in a complete binary tree.
 * @param root The root of the binary tree.
 * @return The total number of nodes in the binary tree.
 */
const countNodes = (root: TreeNode | null): number => {
  // Helper function to find the depth of the tree.
  const getDepth = (node: TreeNode | null): number => {
    let depth = 0;
    while (node !== null) {
      depth++;
      node = node.left; // Go to leftmost node to find the depth.
    }
    return depth;
  };

  // Base case: if the tree is empty, return 0.
  if (root === null) {
    return 0;
  }

  // Calculate the depth of the left and right subtrees.
  const leftDepth = getDepth(root.left);
  const rightDepth = getDepth(root.right);

  // Check if the left and right subtrees have the same depth.
  if (leftDepth === rightDepth) {
    // If depths are equal, left subtree is a perfect binary tree.
    // Calculate the number of nodes in the left subtree as 2^leftDepth - 1
    // Count the nodes in the right subtree recursively.
    return (1 << leftDepth) + countNodes(root.right);
  } else {
    // If depths are not equal, right subtree is a perfect binary tree.
    // Calculate the number of nodes in the right subtree as 2^rightDepth - 1
    // Count the nodes in the left subtree recursively.
    return (1 << rightDepth) + countNodes(root.left);
  }
};
```

Time and Space Complexity

The given Python code is designed to count the number of nodes in a complete binary tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

Time Complexity

The time complexity of the algorithm depends on the structure of the tree. Each recursive call to `countNodes` either moves to the right or to the left subtree and computes the height of the opposite subtree. The `depth` function takes time proportional to the height of the tree, which is `O(log n)` for a complete binary tree (where `n` is the number of nodes).

The recurrence relation for the worst-case time complexity can be approximated as `T(n) = T(n/2) + O(log n)` because at each step one of the subtrees is approximately half the size of the original tree and we spend `O(log n)` time to calculate the depth.

Solving this recurrence gives us a time complexity of `O(log^2 n)`. This is because for each level of recursion, which is logarithmic in the number of nodes, we compute the depth of a subtree, which is also logarithmic in the number of nodes.

Space Complexity

The space complexity of the code is primarily determined by the maximum size of the call stack since the function is recursive. In the worst case, the recursion goes `O(log n)` levels deep because the function will be called recursively on subtrees of decreasing size until the size becomes 1.

Therefore, the space complexity is `O(log n)`, due to the call stack of the recursive function calls.

These complexities are true assuming that the underlying Python implementation handles tail recursion properly.