1302. Deepest Leaves Sum

Medium Tree Depth-First Search Breadth-First Search Binary Tree Leetcode Link

Problem Description

The problem presents us with a binary tree, where each node contains an integer value. Our task is to find the sum of the values that reside in the deepest leaves of the tree. To clarify, the deepest leaves are the nodes at the bottom most level of the tree, which have no children.

Intuition

The intuition behind the solution is to traverse the entire tree to find the deepest leaves and accumulate their values. We can approach this problem using either Depth-First Search (DFS) or Breadth-First Search (BFS).

DFS would involve a recursive function that keeps track of the current level and compares it against the maximum level found so far. If the node being visited is a leaf and is at the deepest level seen so far, its value is added to the running sum.

However, the provided solution utilizes BFS, which is a suitable approach for this type of problem because BFS explores the tree level by level. We use a queue to keep track of nodes to visit and proceed through each level of the tree one by one. As we traverse and exhaust all nodes on a particular level, we can be confident that eventually, we will reach the deepest level. By summing all values at that final level, we get the sum of all the deepest leaves.

We can implement BFS in Python using a queue (in this case, a deque for efficiency reasons). We initiate the queue with the root node, and in each iteration, we process all nodes at the current level (queue size determines the level width). For each node, we add its value to a sum accumulator and extend the queue with its children, if any. When the loop ends, the sum accumulator contains the sum of the values at the deepest level, which is the final answer we want to return.

Solution Approach

The solution uses the Breadth-First Search (BFS) pattern to solve the problem. BFS is ideal for this problem since it processes nodes level by level. In this approach, the algorithm makes use of a `deque`, which is a double-ended queue that allows for efficient addition and removal of elements from both ends.

Here's a step-by-step breakdown of the algorithm incorporated in the solution:

1. Initialize a `deque` with the root of the binary tree.
2. Initiate a loop that runs as long as there are nodes present in the `deque`.
3. Reset the sum accumulator `ans` to 0 at the start of each level.
4. Determine the number of nodes present at the current level by checking the length of the `deque`.
5. For each node of the current level, pop it from the left side of the `deque`.
6. Add the value of the current node to the accumulator `ans`.
7. Check if the current node has a left child, if so, append it to the right side of the `deque`.
8. Check if the current node has a right child, if so, append it to the right side of the `deque`.
9. At the end of the level, all node values will have been added to `ans`, we then proceed to the next level (if any).
10. Once the loop exits, it means we have visited all levels. The last accumulated sum stored in `ans` is the sum of all the values of the deepest leaves in the binary tree.

In terms of algorithmic efficiency, at each level `n`, all `n` nodes are processed, and each node is added to the queue once. This results in an `O(N)` time complexity, where `N` is the number of nodes in the binary tree. The space complexity is `O(N)` in the worst case, when the tree is complete, and the bottom level is full (half of the nodes are on the last level).

Example Walkthrough

Let's take an example binary tree to walk through the BFS solution approach.

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

Now we apply the BFS algorithm step-by-step:

1. We initialize a `deque` with the root of the binary tree. In this case, the root is the node with value `1`.

2. There's a node in the `deque` (the root), so we begin the loop.

3. Reset the sum accumulator `ans` to 0. This holds the sum of values at the current level.

4. Determine the number of nodes present at the current level. Initially, this is 1 since we start with just the root.

5. We pop the node with value `1` from the left side of the deque and add its value to `ans`. `ans` now becomes `1`.

6. We check for the left and right children of node `1`. Both exist, so we append them to the right side of the deque. The deque now holds nodes `2` and `3`.

7. Moving to the second level, we reset `ans` to 0. The deque has 2 nodes, so we will process both nodes `2` and `3`.

8. For both nodes, we remove them from the deque, add their values to `ans`, and add their children to the deque. After processing, `ans` is `2 + 3 = 5`, and the deque now holds nodes `4`, `5`, and `6`.

9. Continuing to the third level, we reset `ans` to 0 again. The deque has 3 nodes, so it represents the third level.

10. We process nodes `4`, `5`, and `6` as before. We add their values to `ans`, and add their children to the deque. After this level, `ans` is `4 + 5 + 6 = 15` and the deque holds `7`, `8`, and `9`.

11. At the fourth level, we process the final nodes `7`, `8`, and `9`, add their values to `ans`, which is now `7 + 8 + 9 = 24`, and since there are no more children, the deque becomes empty.

12. As the deque is now empty, the loop exits. At this point, `ans` holds the sum of all the deepest leaf nodes, which is `24`. This sum is the final answer, and it is the sum of the values of the deepest leaves in our example binary tree: `7`, `8`, and `9`.

By following these steps, applying BFS allowed us to efficiently find the sum of the deepest leaves in the binary tree for our example. The algorithm progresses level by level, and the running sum at the last level gives us our answer.

Python Solution

```python
from collections import deque  # Import deque for queue operations

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def deepestLeavesSum(self, root: Optional[TreeNode]) -> int:
        # Initialize a queue with the root node
        queue = deque([root])

        while queue:  # Continue until the queue is empty
            level_sum = 0  # Sum of values at the current level
            level_count = len(queue)  # Number of nodes at the current level

            # Iterate over all nodes at the current level
            for _ in range(level_count):
                node = queue.popleft()  # Remove the node from the queue
                level_sum += node.val  # Add the node's value to the level sum

                # If the node has a left child, add it to the queue
                if node.left:
                    queue.append(node.left)

                # If the node has a right child, add it to the queue
                if node.right:
                    queue.append(node.right)

        # After the loop, level_sum contains the sum of values of the deepest leaves
        return level_sum

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * Solution class to find the sum of the deepest leaves in a binary tree.
 */
class Solution {
    /**
     * Computes the sum of the deepest leaves in the binary tree.
     *
     * @param root the root node of the binary tree.
     * @return the sum of the deepest leaves.
     */
    public int deepestLeavesSum(TreeNode root) {
        // Initialize a queue to perform level-order traversal
        Deque<TreeNode> queue = new ArrayDeque<>();
        queue.offer(root);

        // Variable to store the sum of the values of the deepest leaves
        int sum = 0;

        // Perform level-order traversal of the tree
        while (!queue.isEmpty()) {
            sum = 0; // Reset sum for the current level
            // Process all nodes at the current level
            for (int size = queue.size(); size > 0; --size) {
                TreeNode currentNode = queue.pollFirst();
                // Add the value of the current node to the sum
                sum += currentNode.val;
                // If the left child exists, add it to the queue for the next level
                if (currentNode.left != null) {
                    queue.offer(currentNode.left);
                }
                // If the right child exists, add it to the queue for the next level
                if (currentNode.right != null) {
                    queue.offer(currentNode.right);
                }
            }
            // The loop goes back to process the next level, if any
        }
        // After the loop, sum contains the sum of the deepest leaves
        return sum;
    }
}

```

C++ Solution

```cpp
#include <queue>

// Definition for a binary tree node.
struct TreeNode {
    int val;            // Value of the node
    TreeNode *left;     // Pointer to the left child
    TreeNode *right;    // Pointer to the right child

    // Constructor to initialize the node with no children
    TreeNode() : val(0), left(nullptr), right(nullptr) {}

    // Constructor to initialize the node with a given value and no children
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

    // Constructor to initialize the node with a value and left and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // Function that computes the sum of values of the deepest leaves in a binary tree
    int deepestLeavesSum(TreeNode* root) {
        int sumAtCurrentDepth = 0; // This will hold the sum of the nodes at the current depth
        // Queue to hold the nodes at the current level for breadth-first traversal
        std::queue<TreeNode*> nodeQueue;
        nodeQueue.push(root);

        // Perform level order traversal (breadth-first traversal) of the tree
        while (!nodeQueue.empty()) {
            sumAtCurrentDepth = 0; // Reset the sum at the start of each level
            // Process all nodes at the current level
            for (int i = nodeQueue.size(); i > 0; --i) {
                TreeNode* currentNode = nodeQueue.front(); // Get the front node in the queue
                nodeQueue.pop(); // Remove the node from the queue

                // Add the node's value to the sum of the current depth/level
                sumAtCurrentDepth += currentNode->val;

                // If there is a left child, add it to the queue for the next level
                if (currentNode->left) nodeQueue.push(currentNode->left);
                // If there is a right child, add it to the queue for the next level
                if (currentNode->right) nodeQueue.push(currentNode->right);
            }
        }
        // Return the sum of the deepest leaves (sum of the last level processed)
        return sumAtCurrentDepth;
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = (val === undefined ? 0 : val);
        this.left = (left === undefined ? null : left);
        this.right = (right === undefined ? null : right);
    }
}

/**
 * Calculate the sum of the deepest leaves in a binary tree.
 * @param {TreeNode | null} root - The root of the binary tree.
 * @return {number} The sum of the deepest leaves' values.
 */
function deepestLeavesSum(root: TreeNode | null): number {
    // If the root is null, return 0 as there are no nodes.
    if (root === null) {
        return 0;
    }

    // Initialize a queue to perform a level-order traversal of the tree.
    const queue: TreeNode[] = [root];
    let currentLevelSum = 0; // This will hold the sum of the values of the current level nodes.

    // Perform a level-order traversal to find the deepest leaves.
    while (queue.length !== 0) {
        const numberOfNodesAtCurrentLevel = queue.length;
        let sum = 0; // Initialize sum for the current level.

        // Process all nodes at the current level.
        for (let i = 0; i < numberOfNodesAtCurrentLevel; i++) {
            // Get the next node from the queue.
            const node = queue.shift();
            if (node) {
                // Accumulate the values of nodes at this level.
                sum += node.val;

                // If the node has a left child, add it to the queue for the next level.
                if (node.left) {
                    queue.push(node.left);
                }

                // If the node has a right child, add it to the queue for the next level.
                if (node.right) {
                    queue.push(node.right);
                }
            }
        }

        // After processing all nodes at the current level, the sum becomes
        // the result as it represents the sum of the deepest leaves seen so far.
        currentLevelSum = sum;
    }

    // Return the sum of the values of the deepest leaves.
    return currentLevelSum;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(N)`, where `N` is the total number of nodes in the binary tree. This is because the algorithm uses a breadth-first search (BFS) approach to traverse each node in the tree exactly once to calculate the sum of the deepest leaves.

Space Complexity

The space complexity of the code is `O(N)`, where `N` is the maximum width of the tree or the maximum number of nodes at any level of the tree. In the worst-case scenario (e.g., a complete binary tree), this is essentially the number of nodes at the last level. The space complexity comes from the queue used to store nodes at each level while performing BFS.