107. Binary Tree Level Order Traversal II

Medium Tree Breadth-First Search Binary Tree Leetcode Link

Problem Description

The LeetCode problem you're referring to asks for a bottom-up level order traversal of a binary tree. This means we should traverse the tree by levels, starting from the root, but instead of collecting the values from top to bottom, we need to return them from bottom to top. To visualize this, consider a tree with multiple levels; we would typically traverse it starting from the topmost level (the root), then move down one level at a time until we reach the leaves. In this problem, though, we're asked to report the level values starting from the leaves and work our way up to the root.

Intuition

The intuition behind the solution is that we can use a queue to perform a standard level order traversal from the root to the leaves. We would typically use a queue data structure to achieve this by enqueuing the root node, then continuously dequeuing a node while enqueueing its children until we have processed all levels of the tree.

However, to report the levels in reverse (from leaves to root), we can modify the standard level order traversal in the following way: as we traverse each level and collect values, we append the list of values to the beginning of our answer list (or alternatively, append to the end and then reverse the entire list at the end). By appending each new level to the front, the levels will be inverted when compared to a standard level order traversal, achieving the desired bottom-up effect.

To summarize, we're using a queue to manage the order of traversal and a list to collect the values. Each level's values are collected in a temporary list, which then gets added to our answer list. After completing the traversal, if we appended each level to the end, we simply reverse our answer list to achieve bottom-up order; if we appended each level to the front, no further action is required.

Solution Approach

The solution provided uses a breadth-first search (BFS) algorithm to traverse the binary tree by levels, starting from the root. The BFS algorithm is typically implemented with the help of a queue data structure to keep track of the nodes to visit.

Here's a breakdown of the implementation steps:

1. Initialize an empty list `ans` which will hold our level order values in reverse order (from bottom to top).
2. Check if the root is `None`. If it is, return an empty list as there are no levels to traverse.
3. Initialize a deque `q` and add the root node to it. A deque is used instead of a regular queue as it allows for efficient addition and removal of elements from both ends. This is important as we will be traversing the tree level by level, and at each level, we deque nodes and enqueue their children.
4. We then enter a while loop that will run as long as there are nodes left in the queue to process. Inside this loop:
   - A temporary list `t` is initialized to store the values of the nodes at the current level.
   - We loop through the nodes in the current level, which is determined by the current length of the queue. For each node in this level:
     - The node is dequeued using `q.popleft()`.
     - The value of the node is appended to the temporary list `t`.
     - If the node has a left child, it's enqueued.
     - Similarly, if the node has a right child, it's enqueued.
   - After collecting all values from the current level, the temporary list `t` is appended to the `ans` list. This ensures that `ans` contains list of values level by level.
5. Once the while loop completes (meaning all levels have been visited), the list `ans` is reversed using `ans[::-1]` since we've been appending each level's values at the end and we need the bottom-up order.
6. The reversed `ans` list is then returned. This list now represents the level order traversal of the binary tree from bottom to top.

Through the use of a deque and a list to collect our level values, coupled with the power of list comprehensions and the ease of reversing lists in Python, we've been able to construct a concise and effective solution to perform a bottom-up level order traversal of a binary tree.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider a binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

We want to perform a bottom-up level order traversal of this binary tree. To do so, we follow the solution approach:

1. **Initialization**: Begin with an empty list `ans` to hold the level order values from bottom to top. The binary tree's root is 3, and since it's not `None`, we continue.

2. **Set Up Queue**: Initialize a deque `q` and add the root node (the value 3) to it.

3. **Level Order Traversal**:

   - Start the while loop since the queue is not empty.
   - For the first iteration, the queue contains just the root node `[3]`.
   - Initialize a temporary list `t` for storing values of this level.

   First Level Iteration:

   - Dequeue the only element in the queue, which is 3.
   - Add 3 to the temporary list `t = [3]`.
   - Enqueue its children (9 and 20) to the queue, so `q = [9,20]`.
   - Append `t` to `ans` giving us `ans = [[3]]`.

   Second Level Iteration:

   - Now, for each element at the current level (2 nodes, 9 and 20), dequeue and process:
     - Dequeue 9 from the queue (no children to enqueue), `t` becomes [9].
     - Dequeue 20, `t` becomes [9,20].
     - Enqueue 20's children (15 and 7) to the queue, so `q = [15,7]`.
   - Append `t` to `ans` giving us `ans = [[3], [9, 20]]`.

   Third Level Iteration:

   - Finally, process the last level with nodes 15 and 7.
     - Dequeue 15 (no children), `t` becomes [15].
     - Dequeue 7 (no children), `t` becomes [15, 7].
   - `q` is now empty, terminating the loop.
   - Append `t` to `ans` giving us `ans = [[3], [9, 20], [15, 7]]`.
4. **Reversing the Result**: Since `ans` stores the levels from top to bottom, we reverse it to get the bottom-up order: `[[15, 7], [9, 20], [3]]`.

5. **Return Result**: Our final result, representing the level order traversal from bottom to top, is `[[15, 7], [9, 20], [3]]`.

This walkthrough demonstrates how the provided solution methodically performs a bottom-up level order traversal using BFS with a queue, a temporary list to collect each level's values, and then reverses the collected levels to achieve the desired result.

Python Solution

```python
from collections import deque

# Definition of 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 levelOrderBottom(self, root: Optional[TreeNode]) -> List[List[int]]:
        """
        Perform a level order traversal of the tree from the bottom up.

        Args:
        root (Optional[TreeNode]): The root of the binary tree.

        Returns:
        List[List[int]]: A list of lists with the values of the nodes at each level
                         from bottom to top.
        """
        # The final answer list.
        levels_reversed = []

        # Return an empty list if the root is None.
        if not root:
            return levels_reversed

        # Initialize a queue with the root for level order traversal.
        queue = deque([root])

        # Iterate while there are nodes in the queue.
        while queue:
            # Placeholder for values at the current level.
            level_values = []

            # Iterate over the nodes at the current level.
            for _ in range(len(queue)):
                current_node = queue.popleft()  # Get the next node from the queue.
                level_values.append(current_node.val)  # Add its value to the level list.

                # If there are left/right children, add them to the queue for the next level.
                if current_node.left:
                    queue.append(current_node.left)
                if current_node.right:
                    queue.append(current_node.right)

            # Append the current level's values to the beginning of the list.
            levels_reversed.append(level_values)

        # Return the level order values in reverse (i.e., from bottom to top).
        return levels_reversed[::-1]

```

Java Solution

```java
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        // Initialize a linked list to store the result. We use LinkedList for efficient insertions at the beginning.
        LinkedList<List<Integer>> result = new LinkedList<>();

        // If the root is null, return the empty result list.
        if (root == null) {
            return result;
        }

        // We use a queue to facilitate level order traversal. Deque interface provides the ability to add/remove elements from both
        Deque<TreeNode> queue = new LinkedList<>();

        // Offer the root node to the end of the queue as a starting point for the level order traversal.
        queue.offerLast(root);

        // Loop as long as there are nodes to process in the queue.
        while (!queue.isEmpty()) {
            // A temporary list to hold nodes values at the current level.
            List<Integer> currentLevel = new ArrayList<>();

            // Number of elements to process at the current level equals the current queue size.
            for (int i = queue.size(); i > 0; --i) {
                // Poll the node from the front of the queue.
                TreeNode node = queue.pollFirst();

                // Add its value to the current level list.
                currentLevel.add(node.val);

                // If the left child exists, offer it to the end of the queue.
                if (node.left != null) {
                    queue.offerLast(node.left);
                }

                // If the right child exists, offer it to the end of the queue.
                if (node.right != null) {
                    queue.offerLast(node.right);
                }
            }

            // At the end of each level, add the current level list to the front of the result list.
            result.addFirst(currentLevel);
        }

        // Return the resulting list of lists, which will be in bottom-up level order.
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <queue>
#include <algorithm>

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    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:
    // Function to perform a level-order traversal (bottom-up) of the tree.
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
        vector<vector<int>> result; // This will store the final level order traversal in reverse order.

        // If the root is nullptr, return the empty result vector.
        if (!root) return result;

        // Initialize a queue to hold the nodes of the tree.
        queue<TreeNode*> nodesQueue;
        nodesQueue.push(root);

        // Perform level-order traversal.
        while (!nodesQueue.empty()) {
            int levelSize = nodesQueue.size(); // Number of nodes at the current level.
            vector<int> currentLevel; // This will store node values at the current level.

            // Iterate over all nodes at the current level.
            for (int i = 0; i < levelSize; ++i) {
                TreeNode* currentNode = nodesQueue.front(); // Get the next node from the queue.
                nodesQueue.pop(); // Remove the node from the queue.

                currentLevel.push_back(currentNode->val); // Add the node's value to the current level's vector.

                // If the current node has a left child, add it to the queue for the next level.
                if (currentNode->left) nodesQueue.push(currentNode->left);

                // If the current node has a right child, add it to the queue for the next level.
                if (currentNode->right) nodesQueue.push(currentNode->right);
            }

            // After finishing the current level, add it to the result vector.
            result.push_back(currentLevel);
        }

        // Since we want to return the result in reverse order, we reverse the entire result vector.
        std::reverse(result.begin(), result.end());

        return result; // Return the final reversed level-order traversal.
    }
};

```

Typescript Solution

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

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

/**
 * Performs a level order traversal from bottom to top on a binary tree.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number[][]} - A list of node values for each level, starting from the bottom level.
 */
const levelOrderBottom = (root: TreeNode | null): number[][] => {
    const result: number[][] = [];
    if (!root) return result;

    // Queue to manage the tree nodes at each level
    const queue: TreeNode[] = [root];

    while (queue.length) {
        const currentLevel: number[] = [];

        // Loop through nodes at current level
        for (let i = queue.length; i > 0; --i) {
            const currentNode: TreeNode = queue.shift()!;  // the '!' asserts that we will not get null here
            currentLevel.push(currentNode.val);

            // Add child nodes to the queue for the next level
            if (currentNode.left) queue.push(currentNode.left);
            if (currentNode.right) queue.push(currentNode.right);
        }

        // Since we are dealing with bottom-up order, we prepend the current level to the result array
        result.unshift(currentLevel);
    }

    return result;
};

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(N)`, where `N` is the number of nodes in the binary tree. This is because the algorithm traverses every node exactly once. With a queue, each node's value is processed and its children are added to the queue if they exist. This operation occurs for every node in the tree exactly once.

Space Complexity

The space complexity of the code is also `O(N)`. In the worst-case scenario (when the tree is completely imbalanced), the queue could contain all nodes at the deepest level of the binary tree. In the case of a complete binary tree, this means the last level could have up to `N/2` nodes, which is still considered `O(N)` in terms of space complexity. Additionally, the `ans` list will contain `N` elements as well since it stores the value of each node exactly once, contributing to the space complexity.