102. Binary Tree Level Order Traversal

Medium Tree Breadth-First Search Binary Tree Leetcode Link

Problem Description

The given problem is about performing a level order traversal on a binary tree. In a level order traversal, we visit all the nodes of a tree level by level, starting from the root. We start at the root level, then move to the nodes on the next level, and continue this process until all levels are covered. The goal is to return the values of the nodes in an array of arrays, where each inner array represents a level in the tree and contains the values of the nodes at that level, ordered from left to right.

Intuition

To solve this problem, we use an algorithm known as Breadth-First Search (BFS). BFS is a traversal technique that explores the neighbor nodes before moving to the next level. This characteristic of BFS makes it perfectly suited for level order traversal.

To implement BFS, we use a queue data structure. The queue helps us process nodes in the order they were added, ensuring that we visit nodes level by level. Here's the step-by-step intuition behind the solution:

1. If the root is `None`, the binary tree is empty, and we have nothing to traverse. Therefore, we return an empty list.
2. Initialize an empty list `ans` to store the level order traversal result, and a queue `q` (implemented as a `deque` to allow for efficient pop and append operations) with the root node as the starting point.
3. We want to process each level of the tree one at a time. We do this by iterating while there are still nodes in our queue `q`.
4. For each level, we initialize a temporary list `t` to store the values of the nodes at this level.
5. Before we start processing the nodes in the current level, we note how many nodes there are in this level which is the current size of our queue `q`.
6. We then dequeue each node in this level from `q`, one by one, and add their values to our temporary list `t`. For each node we process, we check if they have a left or right child. If they do, we enqueue those children to `q`. This prepares our queue for the next level.
7. Once a level is processed, we append our temporary list `t` to `ans`.
8. Continue the process until there are no more nodes left in the queue, meaning we have visited all levels.
9. Return `ans`, which contains the level order traversal result. Each inner list within `ans` represents node values at a respective level of the tree.

The breadth-first nature of the queue ensures that we visit all nodes at a given depth before moving on to nodes at the next depth, fitting the needs of a level order traversal perfectly.

Solution Approach

The implementation of the solution primarily involves utilizing the Breadth-First Search (BFS) algorithm with the help of a queue data structure to traverse the tree. The code involves the following steps:

1. **Initialization**: We first define a class `Solution` with the method `levelOrder`, which takes the `root` of the binary tree as an input. We initialize an empty list `ans` to hold our results. If the `root` is `None`, we immediately return the empty list as there are no nodes to traverse.
2. **Queue Setup**: We initialize a queue `q` (typically implemented as a double-ended queue `deque` in Python for efficient addition and removal of elements). We add the `root` to the queue to serve as our starting point for the level order traversal.
3. **Traversal Loop**: We then enter a while loop, which will run until the queue is empty. This loop is responsible for processing all levels in the tree.
4. **Level Processing**: Inside the loop, we start by creating a temporary list `t` to store node values for the current level. Since queue `q` currently holds all nodes at the current level, we use a for loop to process the number of nodes equal to the current size of the queue. For each node:
   - We dequeue the node from `q` using `popleft`.
   - We add the node's value to the temporary list `t`.
   - If the node has a left child, we add it to the queue.
   - Similarly, if the node has a right child, we add it to the queue as well.

   After we process all nodes at the current level, their children are in the queue, ready to be processed for the next level.
5. **Appending to Result**: Once we finish processing all nodes at one level, we append temporary list `t`, which contains the values of these nodes, to our main result list `ans`.
6. **Completion**: After the while loop exits (the queue is now empty since all nodes have been visited), we return `ans`. The list `ans` now holds the node values for each level of the binary tree, arranged from top to bottom and left to right, as required by the problem.

The `BFS` approach ensures we process all nodes at one level before moving onto the next, aligning with the level order traversal definition. By taking advantage of the queue to keep track of nodes to visit and maintaining a list for each level's node values, the solution effectively combines data structure utilization with traversal strategy to solve the problem efficiently.

Example Walkthrough

Let's illustrate the solution approach with a small binary tree example.

Consider the following binary tree:

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

Here's how the level order traversal would process this tree:

1. **Initialization**: We create an instance of `Solution` and call the method `levelOrder` with the root of the tree (node with value 1). We also initialize an empty list `ans` to store the result of the traversal.
2. **Queue Setup**: We initialize a queue `q` and add the root node with value 1 to it.
3. **Traversal Loop**: The queue is not empty (it contains the root), so we start the while loop.
4. **Level Processing** - First Iteration
   - Create a temporary list `t` to store node values for this level.
   - The current queue size is 1, so the loop runs for one iteration.
   - We dequeue the root node (value 1) from the queue and add it to list `t` (which is now [1]).
   - The root node has two children, nodes with values 2 and 3, which we add to the queue.
5. **Appending to Result**: We've finished processing the first level and append `t` to `ans` (which is now [[1]]).
6. **Level Processing** - Second Iteration
   - Since the queue has two elements (values 2 and 3), we process two nodes this round.
   - Dequeue node with value 2, add it to temporary list `t`, and enqueue its child (value 4 to queue).
   - Dequeue node with value 3, add it to `t`, and enqueue its children (values 5 and 6 to queue).
   - At the end of this iteration, `t` has [2, 3], and the queue has nodes 4, 5, and 6.
7. **Appending to Result**: We've finished processing the second level and append `t` to `ans` (which is now [[1], [2, 3]]).
8. **Level Processing** - Third Iteration
   - This time queue size is 3, and we process nodes 4, 5, and 6 similarly.
   - We end up with `t` having [4, 5, 6], and the queue is now empty.
9. **Appending to Result**: The third level is processed, and we append `t` to `ans` (which is now [[1], [2, 3], [4, 5, 6]]).
10. **Completion**: The queue is empty, so the loop ends and we return `ans`. The final result is [[1], [2, 3], [4, 5, 6]], which correctly represents the level order traversal of the tree.

The traversal has visited all nodes level by level and has constructed a list of node values for each level, fitting the problem requirements perfectly.

Python Solution

```python
from collections import deque

# 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 levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        # Initialize a list to hold the level order traversal result
        result = []

        # If the tree is empty, return the empty list
        if root is None:
            return result

        # Use a queue to keep track of nodes to visit, starting with the root node
        queue = deque([root])

        # Loop until the queue is empty
        while queue:
            # Temporary list to store the values of nodes at the current level
            level_values = []

            # Iterate over nodes at the current level
            for _ in range(len(queue)):
                # Pop the node from the left side of the queue
                current_node = queue.popleft()
                # Append the node's value to the temporary list
                level_values.append(current_node.val)

                # If the node has a left child, add it to the queue for the next level
                if current_node.left:
                    queue.append(current_node.left)
                # If the node has a right child, add it to the queue for the next level
                if current_node.right:
                    queue.append(current_node.right)

            # Append the list of values for this level to the result list
            result.append(level_values)

        # Return the result list containing the level order traversal
        return result

```

Java Solution

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Deque;
import java.util.ArrayDeque;

// 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;
    }
}

class Solution {
    // Method to perform a level order traversal of a binary tree.
    public List<List<Integer>> levelOrder(TreeNode root) {
        // Create a list to hold the result.
        List<List<Integer>> result = new ArrayList<>();

        // Return an empty list if the tree is empty.
        if (root == null) {
            return result;
        }

        // Create a queue to hold nodes at each level.
        Deque<TreeNode> queue = new ArrayDeque<>();

        // Start the level order traversal from the root.
        queue.offer(root);

        // While there are nodes to process
        while (!queue.isEmpty()) {
            // Temporary list to store the values of nodes at the current level.
            List<Integer> level = new ArrayList<>();

            // Process all nodes at the current level.
            int levelLength = queue.size();
            for (int i = 0; i < levelLength; ++i) {
                // Retrieve and remove the head of the queue.
                TreeNode currentNode = queue.poll();

                // Add the node's value to the temporary list.
                level.add(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);
                }
            }

            // Add the temporary list to the result list.
            result.add(level);
        }

        // Return the list of levels.
        return result;
    }
}

```

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:
    /**
     * Performs a level order traversal of a binary tree and returns the values of nodes at each level
     * as a 2D vector.
     *
     * @param root The root node of the binary tree.
     * @return A 2D vector where each inner vector contains the values of nodes at the corresponding level.
     */
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> levels;  // Stores the values of nodes at each level
        if (!root) return levels;   // If the tree is empty, return an empty vector

        // Initialize a queue to assist in BFS traversal
        queue<TreeNode*> nodesQueue;
        nodesQueue.push(root);

        while (!nodesQueue.empty()) {
            int levelSize = nodesQueue.size(); // Number of nodes at the current level
            vector<int> currentLevel;          // Vector to store values of node at current level

            while (levelSize > 0) {
                TreeNode* currentNode = nodesQueue.front(); // Take the front node in the queue
                nodesQueue.pop();                           // Remove that node from the queue

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

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

                levelSize--; // Decrement the counter for nodes remaining at current level
            }

            // After processing all nodes at the current level, add the current level vector to the results
            levels.push_back(currentLevel);
        }

        return levels; // Return the 2D vector with all the levels' values
    }
};

```

Typescript Solution

```typescript
// Function to perform level order traversal on a binary tree and return the values in level-wise order.
function levelOrder(root: TreeNode | null): number[][] {
    // Initialize the result array.
    const result: number[][] = [];

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

    // Initialize a queue to hold nodes at each level.
    const queue: TreeNode[] = [root];

    // Iterate until the queue is empty.
    while (queue.length !== 0) {
        // Get the number of nodes at the current level.
        const levelLength = queue.length;

        // Process all nodes at the current level.
        const currentLevelValues: number[] = [];
        for (let i = 0; i < levelLength; i++) {
            // Shift the first node from the queue.
            const currentNode = queue.shift();

            // Proceed if the currentNode is not null.
            if (currentNode) {
                // Add the value to the current level's result.
                currentLevelValues.push(currentNode.val);

                // Add left and right children if they exist.
                if (currentNode.left) queue.push(currentNode.left);
                if (currentNode.right) queue.push(currentNode.right);
            }
        }

        // Add the current level's values to the overall result array.
        result.push(currentLevelValues);
    }

    // Return the result array containing the level order traversal.
    return result;
}

```

Time and Space Complexity

The given Python code implements a level order traversal of a binary tree, where `root` is the root of the binary tree.

Time Complexity:

The time complexity of this algorithm is `O(n)`, where `n` is the total number of nodes in the binary tree. This is because each node in the tree is visited exactly once during the traversal.

Space Complexity:

The space complexity of the algorithm can be analyzed in terms of two components: the space used by the data structure `q` (which holds the nodes at the current level), and the space for the `ans` list.

The worst-case space complexity is `O(w)`, where `w` is the maximum width of the tree. This worst case occurs when the binary tree is a complete binary tree or has the most nodes at one level. Space is needed to store all nodes of the widest level in the data structure `q` at one time.

Therefore, the overall space complexity of the algorithm is `O(w)` if we only consider the space taken by the queue. If we add the space taken by the output list `ans`, the total space complexity remains `O(n)` since `ans` will eventually contain all nodes of the tree.