515. Find Largest Value in Each Tree Row

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

Problem Description

In this problem, we are provided with the `root` of a binary tree. Our task is to find the largest value in each row of the tree and return them in an array. The binary tree is not necessarily a binary search tree, which means that the tree is not guaranteed to follow the ordering property of binary search trees. Each row in the array corresponds to a level of the tree, indexed starting from 0. The first element of the array corresponds to the root level, and subsequent elements correspond to the succeeding levels of the tree.

Intuition

To solve this problem, a common approach is to perform a level order traversal of the binary tree. A level order traversal means that we visit all the nodes at the current level before moving on to the nodes of the next level.

Here is how we can proceed:

1. Start with the root node by adding it to a queue. Since we only have the root, the largest value for the root level is the value of the root itself.
2. While the queue is not empty, we will perform the following steps for each level:
   - Initialize a variable to keep track of the maximum value for the current level. Initially, it can be set to negative infinity to ensure that any node value will be larger.
   - Get the number of nodes in the current level (the size of the queue).
   - Iterate through these nodes and for each node:
     - Update the maximum value if the current node's value is greater.
     - Add the left and right children of the current node to the queue if they exist.
3. After finishing the iteration for the current level, add the maximum value we found to the answer array.
4. Repeat the process for the next level.
5. Once the queue is empty, it means we have visited all levels and obtained the largest value for each level. The answer array is now complete, and we return it as the result.

This approach ensures that for each level, we are able to assess all the nodes, and identify the largest value, thereby constructing our array of the largest values for each level.

Solution Approach

The provided solution code implements the level order traversal approach to solve the problem. It uses the following algorithms, data structures, and patterns:

1. **Breadth-First Search (BFS):** The level order traversal is a type of BFS, in which nodes are visited level by level from top to bottom. This traversal is suitable for finding the largest value in each row since it processes all nodes at each level before moving on to the next one.
2. **Queue Data Structure:** The code utilizes a `deque` (double-ended queue) from Python's `collections` module to efficiently perform the BFS. A queue is ideal for BFS since it follows the First-In-First-Out (FIFO) principle, which ensures that nodes are processed in the order they are discovered.
3. **Looping through Levels:** The algorithm uses a `while` loop to keep processing nodes as long as the queue is not empty. Inside this loop, there is another `for` loop that iterates over each node of the current level, which is determined by the current size of the queue.
4. **Tracking Maximum Values:** The variable `t` is used to keep track of the maximum value for the current level. It's initially set to `-inf` (negative infinity) to ensure any node's value will be greater, and hence it can be updated as soon as a node with a value is encountered.
5. **Adding Children to Queue:** While processing each node, the code checks whether the left or right child exists. If they do, these children are added to the queue for future processing. This step is crucial to expand the traversal to subsequent levels.

Here's a step-by-step breakdown of the code implementation:

```python
# The TreeNode structure is predefined; each node has a value and links to left and right children.
class Solution:
    def largestValues(self, root: Optional[TreeNode]) -> List[int]:
        if root is None:  # If the tree is empty, there are no values to process, return an empty list.
            return []
        q = deque([root])  # Initialize the deque with the root node to start the level order traversal.
        ans = []  # This list will store the largest values for each level.

        # Continue the loop as long as there are nodes to process.
        while q:
            t = -inf  # Set the maximum value for current level to negative infinity.

            # Process each node of the current level.
            for _ in range(len(q)):  # 'for' loop runs for the number of nodes in the current level.
                node = q.popleft()  # Dequeue the front node of the queue.
                t = max(t, node.val)  # Update the maximum value if node's value is greater.

                # Enqueue children of the current node if they exist.
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)

            # After processing the current level, the largest value is appended to the answer.
            ans.append(t)

        # Return the array containing the largest values for each level.
        return ans
```

This implementation ensures that the traversal continues as long as there are nodes left to be processed. The largest value for each level is determined by comparing it with the default minimum value of `-inf`, and updating it with the node's value if it's greater. By the end of the traversal, the `ans` list contains the largest values for each level, which is the desired output of the problem.

Example Walkthrough

Let's consider a small binary tree as our example to illustrate the solution approach:

```
      3
     / \
    1   5
   /   / \
  0   4   7
```

Now, let's walk through the solution step-by-step, applying the level order traversal approach described above.

1. First, we initiate the traversal by adding the root node (which has the value 3) to a queue. Since this is the only node at level 0, the largest value for this level is 3.
2. Now, we begin iterating level by level. Our queue currently contains the root:

   ```
   Queue: [3]
   ```

3. We remove the root node from the queue and inspect its children. Since the root node has two children, with values 1 and 5, we add these two nodes to the queue. The largest value at this level (level 0) is still 3, so we store 3 in our answer array.

   ```
   Queue: [1, 5]
   Answer: [3]
   ```

4. For the next level (level 1), both nodes in the queue are processed. We start by setting the maximum value for this level to negative infinity:

   ```
   t = -inf
   ```

5. We dequeue the first node (with value 1) and compare it with our max value `t`. Since 1 is greater than `-inf`, `t` becomes 1. This node has a left child with value 0, which we enqueue:

   ```
   Queue: [5, 0]
   ```

6. The next node with value 5 is dequeued and compared with `t`. Since 5 is greater than 1, `t` becomes 5. This node has two children with values 4 and 7, which we enqueue:

   ```
   Queue: [0, 4, 7]
   ```

7. With all nodes at level 1 processed, we update our answer array with the maximum value found at this level, which is 5:

   ```
   Answer: [3, 5]
   ```

8. We repeat this process for the next level, iterating over the remaining nodes in the queue (0, 4, and 7). We'll find that 7 is the maximum value at this level (level 2).
9. Our process continues until the queue is empty, which means we've processed all the levels and found the maximum value for each.
10. Finally, we return the completed `ans` array as the result:

```
Answer: [3, 5, 7]
```

This example demonstrates how the approach processes each level of the binary tree and finds the largest value for each level, completing the task successfully.

Python Solution

```python
from collections import deque
from typing import List, Optional

# 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 largestValues(self, root: Optional[TreeNode]) -> List[int]:
        """
        Function to find the largest value in each level of a binary tree.

        :param root: Optional[TreeNode] - root of the binary tree
        :return: List[int] - a list containing the largest value of each level
        """
        # If the root is none, return an empty list as there are no levels in the tree.
        if root is None:
            return []

        # Initialize a queue for BFS traversal and a list to store the answers.
        queue = deque([root])
        largest_values = []

        # Traverse the tree level by level.
        while queue:
            # Initialize the largest element of the current level to negative infinity.
            max_value = float('-inf')
            # Iterate over all nodes at the current level.
            for _ in range(len(queue)):
                node = queue.popleft()  # Get the next node from the queue.
                max_value = max(max_value, node.val)  # Update the largest value found in this level.
                # If left child exists, append it to the queue for the next level.
                if node.left:
                    queue.append(node.left)
                # If right child exists, append it to the queue for the next level.
                if node.right:
                    queue.append(node.right)
            # After visiting all nodes at the current level, add the largest value to the results list.
            largest_values.append(max_value)

        # Return the list of largest values, one from each level.
        return largest_values

```

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;
    }
}

/**
 * Class containing the method to find the largest values in each tree row.
 */
class Solution {
    /**
     * Finds the largest values in each row of a binary tree.
     *
     * @param root The root of the binary tree.
     * @return A list of the largest values in each row of the tree.
     */
    public List<Integer> largestValues(TreeNode root) {
        // List to store the largest values in each row.
        List<Integer> largestValues = new ArrayList<>();

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

        // Queue to perform level order traversal
        Deque<TreeNode> queue = new ArrayDeque<>();

        // Start the traversal with the root node.
        queue.offer(root);

        // Loop until the queue is empty, indicating all levels have been visited.
        while (!queue.isEmpty()) {
            // Initialize the variable to store the max value for the current level.
            int maxInLevel = queue.peek().val;

            // Traverse all nodes at this level.
            for (int i = queue.size(); i > 0; --i) {
                // Get the next node from the queue.
                TreeNode node = queue.poll();

                // Update the max value with the max of current max and node's value.
                maxInLevel = Math.max(maxInLevel, node.val);

                // Add the left child to the queue if it exists.
                if (node.left != null) {
                    queue.offer(node.left);
                }

                // Add the right child to the queue if it exists.
                if (node.right != null) {
                    queue.offer(node.right);
                }
            }

            // After finishing the level, add the max value found to the result list.
            largestValues.add(maxInLevel);
        }

        // Return the list of maximum values found.
        return largestValues;
    }
}

```

C++ Solution

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

// 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:
    // Method to find the largest value in each level of a binary tree
    vector<int> largestValues(TreeNode* root) {
        // If the tree is empty, return an empty vector
        if (!root) return {};

        // Queue for holding nodes to process
        queue<TreeNode*> nodes_queue;
        nodes_queue.push(root);

        // Vector to store the largest values
        vector<int> largest_values;

        // Process nodes level by level
        while (!nodes_queue.empty()) {
            // Initialize the current level's largest value with the first node's value
            int current_level_largest = nodes_queue.front()->val;

            // Iterate through nodes in the current level
            for (int i = nodes_queue.size(); i > 0; --i) {
                // Get the next node in the queue
                TreeNode* current_node = nodes_queue.front();
                // Update the largest value if the current node is greater
                current_level_largest = max(current_level_largest, current_node->val);
                // Remove the processed node from the queue
                nodes_queue.pop();
                // Add child nodes of the current node to the queue for the next level
                if (current_node->left) nodes_queue.push(current_node->left);
                if (current_node->right) nodes_queue.push(current_node->right);
            }

            // Append the largest value for this level to the results
            largest_values.push_back(current_level_largest);
        }

        return largest_values;
    }
};

```

Typescript Solution

```typescript
// The TreeNode class represents a node in a binary tree with a numeric value.
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;
  }
}

/**
 * Finds the largest value in each level of a binary tree.
 * @param {TreeNode | null} root - The root of the binary tree.
 * @returns {number[]} An array containing the largest values from each level.
 */
function findLargestValues(root: TreeNode | null): number[] {
  const largestValues: number[] = [];
  const nodeQueue: TreeNode[] = [];

  if (root) nodeQueue.push(root);

  while (nodeQueue.length > 0) {
    const queueLength = nodeQueue.length;
    let levelMax = -Infinity;

    for (let i = 0; i < queueLength; i++) {
      const node = nodeQueue.shift(); // Retrieves and removes the head of the queue
      if (node) {
        levelMax = Math.max(levelMax, node.val);
        if (node.left) nodeQueue.push(node.left);
        if (node.right) nodeQueue.push(node.right);
      }
    }

    largestValues.push(levelMax);
  }

  return largestValues;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of nodes in the binary tree. This is because the code uses a breadth-first search (BFS) algorithm that visits each node exactly once to find the largest value in each level of the tree.

Space Complexity

The space complexity is also `O(n)`. This is the worst-case scenario where the tree is extremely unbalanced, and all nodes are on a single side of the tree, leading to all nodes being held in the queue at one point in time. For a balanced tree, however, the largest space consumption would occur at the level with the most nodes, which is approximately `n/2` nodes for a full binary tree at the last level, but it is still expressed as `O(n)` in Big O notation for simplicity as it remains proportional to the number of nodes.