# 513. Find Bottom Left Tree Value

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

## Problem Description

The problem gives us a binary tree and asks us to find the leftmost value in the tree's last row. In simple terms, we need to go to the very bottom row of the tree and return the value of the node that is furthest to the left.

## Intuition

To find the leftmost value in the last row, we can perform a breadth-first search (BFS) traversal of the tree. In BFS, we start with the root and explore all the nodes at the current depth level before moving on to the nodes at the next level. We can use a queue data structure to keep track of the nodes at the current level.

The BFS approach is useful here because it naturally visits levels from top to bottom, and for each level, from left to right. So, the last value we encounter at each level would be the leftmost value. When the queue no longer has nodes to process, the last node we looked at would be the leftmost node of the bottom row.

The intuition for this approach comes from understanding that the level order traversal will encounter all nodes level by level. We don't need to preserve the entire level; we only keep the last node processed (the leftmost of that level). We update this value as we proceed to the next level.

## Solution Approach

The solution to this problem uses a simple Breadth-First Search (BFS) algorithm. BFS is usually implemented using a queue data structure, which allows us to process nodes in a "first-in, first-out" (FIFO) order. In this solution, a `deque` from the `collections` module is used because it enables efficient append and pop operations from both ends of the queue.

Here is the step-by-step approach used in the solution:

1. Initialize a queue (in this case, `q`) with the root node of the binary tree. This queue will hold the nodes to be processed.
2. Initialize a variable (`ans`) to keep track of the leftmost value.
3. While the queue is not empty, perform the following steps:
   - Update `ans` with the value of the first node in the queue (`q[0].val`), as this is guaranteed to be the leftmost node of the current level.
   - Loop over the nodes at the current level, which is the current size of the queue.
     - Remove the node from the front of the queue using `popleft()`.
     - If the node has a left child, append it to the queue. This ensures that the left child is processed before the right child.
     - If the node has a right child, append it to the queue.
4. After the last level has been processed, and the queue is empty, `ans` will hold the value of the leftmost node in the last row of the tree.
5. Return the value stored in `ans`.

The BFS process ensures that we traverse the tree level by level, and by always taking the first element in the queue, we are guaranteed to process the leftmost node of each level. When we are at the last level, the first node in the queue will be the leftmost node of the bottommost level, which is what the problem asks us to return.

### Example Walkthrough

Let's walk through an example to illustrate this solution approach using a simple binary tree.

Suppose our binary tree looks like this:

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

We want to find the leftmost value in the tree's last row. According to the tree above, the last row is the row with the nodes 4, 7, and 6, and the leftmost value is 4.

Let's apply the BFS algorithm step-by-step:

1. Initialize the queue with the root node (which is 1 in this case). Our queue (q) looks like this: [1]
2. Initialize the `ans` variable to keep track of the leftmost value. Currently, `ans` is not set.
3. The queue is not empty, so we start our while loop.
   - Update `ans` with the value of the first node in the queue, which is 1 (`q[0].val`).
   - There is only one node at this level, so we process it.
     - We pop the node 1 using `popleft()`, leaving the queue empty.
     - Since node 1 has a left child (node 2), we append it to the queue. The queue now contains [2].
     - Since node 1 also has a right child (node 3), we append it as well. The queue now contains [2, 3].
4. We proceed with the next iteration of the while loop, since the queue is not empty. The queue currently has [2, 3].
   - Update `ans` with the value of the first node in the queue, which is now 2.
   - There are two nodes at this level (2 and 3). We process these two nodes.
     - Pop node 2 from the queue and check its children. It has one left child, node 4, which we append to the queue. Now the queue is [3, 4].
     - Pop node 3, append its left child (node 5) and its right child (node 6) to the queue. The queue becomes [4, 5, 6].
5. After processing these nodes, we have the following snapshot of our queue, which represents the next level: [4, 5, 6].
   - Update `ans` to the first node's value in the queue, which is now 4.
   - The current level has three nodes. We need to process each:
     - Pop node 4 from the queue; as it has no children, we do not append anything to the queue. The queue becomes [5, 6].
     - Pop node 5, and since it has a left child (node 7), we append node 7 to the queue. We don't append anything for the right child as it doesn't exist. The queue is now [6, 7].
     - Pop node 6 from the queue; as it has no children, we do not append anything to the queue. The queue becomes [7].
6. One final iteration shows us that we are at the last level. Update `ans` to 7, and process the only node at this level:
   - Pop node 7 from the queue; it has no children, so the queue is now empty.
7. The queue is empty, and the while loop exits. The `ans` value, which is 7, is our final result. It represents the leftmost value of the last row in the binary tree.
8. We return the value stored in `ans`, which is 7, as the leftmost value in the tree's last row.

## 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 findBottomLeftValue(self, root: TreeNode) -> int:
        # Initialize a queue with the root node.
        node_queue = deque([root])

        # This will hold the leftmost value as the tree is traversed level by level.
        bottom_left_value = 0

        # Perform a level order traversal on the tree.
        while node_queue:
            # At new level's beginning, the first node is the leftmost node.
            bottom_left_value = node_queue[0].val

            # Iterate through nodes at the current level.
            for _ in range(len(node_queue)):
                # Pop the node from the front of the queue.
                node = node_queue.popleft()

                # If the left child exists, add it to the queue.
                if node.left:
                    node_queue.append(node.left)
                # If the right child exists, add it to the queue.
                if node.right:
                    node_queue.append(node.right)

        # Return the bottom left value found during traversal.
        return bottom_left_value

```

## Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode() {}

    // Constructor for creating a new node with a given value.
    TreeNode(int val) {
        this.val = val;
    }

    // Constructor for creating a new node with a given value and left & right children.
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    /**
     * Finds the value of the bottom-leftmost node in a binary tree using level order traversal.
     *
     * @param root The root node of the binary tree.
     * @return The value of the bottom-leftmost node.
     */
    public int findBottomLeftValue(TreeNode root) {
        // Initialize a queue to hold tree nodes in level order.
        Queue<TreeNode> queue = new ArrayDeque<>();

        // Begin with the root node.
        queue.offer(root);

        // This will hold the most recent leftmost value found at each level.
        int bottomLeftValue = 0;

        // Traverse the tree level by level.
        while (!queue.isEmpty()) {
            // Update the bottomLeftValue with the value of the first node in this level.
            bottomLeftValue = queue.peek().val;

            // Process each node in the current level and enqueue their children.
            for (int i = queue.size(); i > 0; --i) {
                TreeNode node = queue.poll();

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

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

        // Return the bottom-leftmost value found.
        return bottomLeftValue;
    }
}

```

## 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:
    // Finds the leftmost bottom value in a binary tree.
    int findBottomLeftValue(TreeNode* root) {
        // Initialize a queue to perform level order traversal
        std::queue<TreeNode*> queue{{root}};

        // Variable to store the leftmost value as we traverse
        int bottomLeftValue = 0;

        // Loop until the queue is empty
        while (!queue.empty()) {
            // Get the value of the current front node as it could be the leftmost node of this level
            bottomLeftValue = queue.front()->val;

            // Iterate over all the nodes at the current level
            for (int i = static_cast<int>(queue.size()); i > 0; --i) {
                TreeNode* currentNode = queue.front();
                queue.pop();

                // If the left child exists, add it to the queue for the next level
                if (currentNode->left) queue.push(currentNode->left);
                // If the right child exists, add it to the queue for the next level
                if (currentNode->right) queue.push(currentNode->right);
            }
        }

        // After traversing the whole tree, bottomLeftValue will contain the leftmost value of the bottom level
        return bottomLeftValue;
    }
};

```

## Typescript Solution

```typescript
// Function to find the bottom-left value of a binary tree.
function findBottomLeftValue(root: TreeNode | null): number {
    let bottomLeftValue = 0; // Initialize a variable to store the bottom-left value.

    // Initialize a queue for level-order traversal starting with the root node.
    const queue: Array<TreeNode | null> = [root];

    // Execute while there are nodes to process in the queue.
    while (queue.length > 0) {
        // The first node's value of each level is the potential bottom-left value.
        bottomLeftValue = queue[0].val;

        // Traverse the current level.
        for (let i = queue.length; i > 0; --i) {
            // Remove the node from the front of the queue.
            const currentNode: TreeNode | null | undefined = queue.shift();

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

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

    // After the traversal, bottomLeftValue will contain the leftmost value of the bottom-most level.
    return bottomLeftValue;
}

```

## Time and Space Complexity

### Time Complexity

The time complexity of the code is `O(N)`, where `N` is the number of nodes in the tree. This is because the code performs a breadth-first search (BFS) of the tree, visiting each node exactly once.

### Space Complexity

The space complexity is also `O(N)`. In the worst case, the queue could have all nodes at the last level of a complete binary tree. In a complete binary tree, the number of nodes at the last level is approximately `N/2`. Since `N/2` is still in the order of `N`, the space complexity is `O(N)`.