1161. Maximum Level Sum of a Binary Tree

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

Problem Description

The problem provides us with the root of a binary tree and asks us to find the level with the maximum sum of node values. In a binary tree, each node has at most two children. The root of the tree is at level 1, its children are at level 2, and this pattern continues for subsequent levels. We need to determine the smallest level number x, which has the maximum sum compared to sums of nodes at other levels.

Intuition

To solve this problem, we can use breadth-first search (BFS), which is an algorithm that starts at the tree root and explores all nodes at the present depth level before moving on to nodes at the next depth level. This approach is suitable because it naturally processes nodes level by level. Here's how we can use BFS to solve the problem:

- Initialize a queue that will hold the nodes to be processed, and start with the root node.
- Initialize variables to keep track of the maximum sum (`mx`), the current level (`i`), and the level with the maximum sum (`ans`).
- For each level, we process all nodes on that level to calculate the sum of values at the current level (`s`).
- After processing all nodes at the current level, we compare the sum with `mx`. If the sum of the current level exceeds `mx`, we update `mx` to this sum and record the current level number as `ans` since it's the smallest level with this new maximum sum so far.
- Continue the process for all levels until all nodes have been processed.
- Return the level number `ans`, which corresponds to the smallest level with the maximal sum.

The provided solution implements this BFS approach efficiently.

Solution Approach

The Reference Solution Approach applies the breadth-first search (BFS) pattern using a queue to traverse the binary tree level by level. Here is the walk-through of the implementation steps:

1. Start by initializing a queue `q` with the root node of the tree. This queue will store the nodes to be processed for each level.
2. We track the maximum sum encountered so far with `mx`, set initially to negative infinity (`-inf`), since we want to be able to compare it with sums that can be potentially zero or positive.
3. The current level `i` is set to `0` at the beginning. It will be incremented as we start processing each level to keep track of the level number.
4. We enter a while loop that continues as long as there are nodes left in the queue `q`.
   - Increment the level `i` by one as we are beginning to process a new level.
   - Initialize a sum `s` for the current level which will be used to add up all node values at this level.
   - Use a for loop to process each node at the current level. The range is determined by the number of elements in the queue at the start of the level (`len(q)`).
     - Dequeue the front node of the queue and add its value to `s`.
     - If the current node has a left child, enqueue the left child.
     - Similarly, if the current node has a right child, enqueue the right child.
5. After the for loop ends, all nodes at the current level have been processed. We now compare the sum `s` of the current level to the max sum `mx`.
   - If `s` is greater than `mx`, we update `mx` to `s` and record the current level as `ans` because it's currently the level with the highest sum.
6. Once the queue is empty, meaning all levels have been processed, we exit the while loop.
7. Finally, return the variable `ans`, which holds the smallest level number with the greatest sum.

By using a queue alongside the BFS pattern, the solution ensures that the tree is traversed level by level, summing node values effectively and keeping track of the level with the greatest sum.

Example Walkthrough

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

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

In this tree, the root node has a value of 3, the root's left child has a value of 9, and the root's right child has a value of 20 which further has two children with values 15 and 7.

Here is how the BFS algorithm would process this tree to find the level with the maximum sum:

1. We start by initializing the queue `q` with the root node (value 3).
2. The maximum sum `mx` is initialized to negative infinity, and the current level `i` is set to 0.
3. Enter the while loop to begin processing since the queue is not empty.
4. Increment `i` to 1 since we are now on level 1, and initialize the sum `s` to 0.
5. Process all nodes at level 1:
   - There is only one node, which is the root node with value 3.
   - Add this value to `s`, so now `s = 3`.
   - Enqueue its children (nodes with values 9 and 20) to the queue.
6. Now, `s` is compared to `mx`. Since `s` (3) is greater than `mx` (-∞), update `mx` to 3 and mark the current level `ans` as 1.
7. Move to the next level (level 2):
   - Increment `i` to 2.
   - The sum `s` is reset to 0.
   - There are two nodes at this level, with values 9 and 20. Process each by dequeuing and adding their values to `s`.
   - `s` is now `9 + 20 = 29`.
   - Enqueue the children of these nodes (values 15 and 7) to the queue.
8. Now, `s` (29) is compared to `mx` (3). `s` is larger so we update `mx` to 29 and `ans` to 2 because this level has the new maximum sum.
9. Move to the next level (level 3):
   - Increment `i` to 3.
   - The sum `s` is reset to 0.
   - There are two nodes at this level, with values 15 and 7. Process each by dequeuing and adding their values to `s`.
   - `s` is now `15 + 7 = 22`.
10. Now, `s` (22) is compared to `mx` (29). Since `s` is less than `mx`, we do not update `mx` or `ans`.
11. Since the queue is now empty, we have processed all levels.
12. Return `ans`, which is 2, as level 2 has the maximum sum of 29.

Using the BFS approach outlined above, we traversed the given binary tree level by level, efficiently calculating the sum of node values at each level and identifying the level with the maximum sum.

Python Solution

```python
from collections import deque
from math import inf

# 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 maxLevelSum(self, root: Optional[TreeNode]) -> int:
        # Initialize the queue with the root node
        queue = deque([root])
        # Initialize max_sum to negative infinity to ensure any level's sum will be larger
        max_sum = -inf
        # Initialize level counter to 0
        level = 0
        # Initialize answer to store the level with the maximum sum
        answer = 0

        # Traverse the tree level by level using breadth-first search
        while queue:
            level += 1
            current_sum = 0
            # Process all the nodes at the current level
            for _ in range(len(queue)):
                # Pop the leftmost node from the queue
                node = queue.popleft()
                # Add the node's value to the current level's sum
                current_sum += node.val
                # 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)
            # Update max_sum and answer if the current level's sum is greater than max_sum
            if max_sum < current_sum:
                max_sum = current_sum
                answer = level

        # Return the level that had the maximum sum
        return answer
```

Java Solution

```java
class Solution {
    // Function to find the level of the binary tree with the maximum sum
    public int maxLevelSum(TreeNode root) {
        // Queue to store nodes of tree level-wise
        Deque<TreeNode> queue = new ArrayDeque<>();
        // Adding the root to the queue as the starting point
        queue.offer(root);

        // Variables to keep track of the maximum level sum and corresponding level
        int maxSum = Integer.MIN_VALUE; // Initialized to minimum value
        int level = 0;                  // Level counter
        int maxLevel = 0;               // Level with max sum

        // Loop through each level of the tree
        while (!queue.isEmpty()) {
            level++; // Increment the level counter
            int levelSum = 0; // Reset the level sum for the current level

            // Calculate the sum of all nodes at the current level
            for (int count = queue.size(); count > 0; count--) {
                TreeNode node = queue.pollFirst(); // Get the next node from the queue
                levelSum += node.val;              // Add the node's value to the level sum

                // Add child nodes to the queue for the next level
                if (node.left != null) {
                    queue.offer(node.left);
                }
                if (node.right != null) {
                    queue.offer(node.right);
                }
            }

            // Update max sum and corresponding level if the current level has a greater sum
            if (maxSum < levelSum) {
                maxSum = levelSum;
                maxLevel = level;
            }
        }

        // Return the level with the maximum sum
        return maxLevel;
    }
}

// Definition for a binary tree node.
class TreeNode {
    int val;         // Node's value
    TreeNode left;   // Reference to the left child node
    TreeNode right;  // Reference to the right child node

    // Constructors for TreeNode
    TreeNode() {}
    TreeNode(int val) { this.val = val; }
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}
```

C++ Solution

```cpp
#include <queue>
#include <climits>  // Include necessary headers

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    // Constructors
    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:
    // Function to find the level that has the maximum sum in a binary tree.
    int maxLevelSum(TreeNode* root) {
        // Using a queue to perform level order traversal
        std::queue<TreeNode*> nodeQueue{ {root} };

        int maxSum = INT_MIN;  // Initialize maxSum to the smallest possible integer
        int resultLevel = 0;   // The level with the max sum
        int currentLevel = 0;  // Current level during level order traversal

        // Continue while there are nodes in the queue
        while (!nodeQueue.empty()) {
            ++currentLevel;  // Increment level counter
            int levelSum = 0;  // Sum of nodes at the current level

            // Process all nodes of the current level
            for (int remaining = nodeQueue.size(); remaining > 0; --remaining) {
                TreeNode* currentNode = nodeQueue.front();  // Get the next node
                nodeQueue.pop();  // Remove the node from the queue
                levelSum += currentNode->val;  // Update the level's sum

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

            // Update the max sum and result level
            if (maxSum < levelSum) {
                maxSum = levelSum;
                resultLevel = currentLevel;
            }
        }

        // Return the level with the max sum
        return resultLevel;
    }
};
```

Typescript Solution

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

/**
 * Calculates the maximum level sum of a binary tree.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @returns {number} The level (1-indexed) with the maximum sum.
 */
function maxLevelSum(root: TreeNode | null): number {
    // Queue to hold nodes to visit using breadth-first search
    const queue: (TreeNode | null)[] = [root];
    // Variable to store the level with the maximum sum
    let maximumLevel = 1;
    // Variable to keep track of the current maximum sum
    let maximumSum = Number.NEGATIVE_INFINITY;
    // Height starts at 1, as levels are 1-indexed
    let height = 1;

    // Loop until the queue is empty
    while (queue.length !== 0) {
        // Get the number of nodes in the current level
        let levelSize = queue.length;
        // Initialize sum for the current level
        let levelSum = 0;

        // Process all nodes for the current level
        for (let i = 0; i < levelSize; i++) {
            // Remove the first node from the queue
            const node = queue.shift();
            if (node) {
                // Add the node's value to the level sum
                levelSum += node.val;
                // If the left child exists, add it to the queue
                if (node.left) queue.push(node.left);
                // If the right child exists, add it to the queue
                if (node.right) queue.push(node.right);
            }
        }

        // Check if the current level has the maximum sum so far
        if (levelSum > maximumSum) {
            // Update the maximum sum and the corresponding level
            maximumSum = levelSum;
            maximumLevel = height;
        }
        // Move to the next level
        height++;
    }

    // Return the level that has the maximum sum
    return maximumLevel;
}
```

Time and Space Complexity

The given code block is designed to find the level of a binary tree that has the maximum sum of values of its nodes.

Time complexity

The time complexity of the code can be determined by analyzing the operations performed for each node in the tree:

- The `while` loop will run once for each level of the tree.
- Inside this loop, a `for` loop iterates through all nodes at the current level. Since each node is visited exactly once during the BFS traversal, all the nodes in the tree will be accounted for.

Thus, if there are N nodes in the tree, the code has a time complexity of $O(N)$ as each node is processed once.

Space complexity

The space complexity is dependent on the storage used throughout the algorithm:

- The queue `q` holds the nodes at the current level being processed. In the worst case, this is equal to the maximum width of the tree, which occurs at the level with the most nodes.
- The maximum number of nodes at any level of a binary tree could be up to $N/2$ (in a complete binary tree).

Therefore, the space complexity is $O(N)$ in the worst case, because of the queue.