2583. Kth Largest Sum in a Binary Tree

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

Problem Description

This LeetCode problem requires finding the k^{th} largest level sum in a binary tree. A level sum is defined as the total value of all nodes at the same depth, where depth is measured by the distance from the root node. We are given the root of the binary tree and a positive integer k, and we need to return the k^{th} largest value among all level sums.

If the tree doesn't have k levels, we must return -1. This problem emphasizes that the notion of level is based on a node's distance from the root, meaning that all nodes at the same distance (or depth) from the root are on the same level.

Intuition

To solve this problem, we will perform a level-order traversal (also known as breadth-first search) to calculate the sum of each level in the tree. Since we traverse the tree level by level, we can easily compute the sum of the nodes at each depth. The level-order traversal uses a queue to keep track of the nodes as we progress.

Here's the step-by-step approach to arriving at the solution:

1. Initialize an empty list `arr` to store the sum of the values at each level and a queue `q` initialized with the root node of the tree.
2. While the queue is not empty, we proceed to the following:
   - Initialize a temporary sum `t` to 0 to accumulate the sum of the values for the current level.
   - For all the nodes in the queue (which are all on the same level):
     - Dequeue the node from the front of the queue.
     - Add the node's value to `t`.
     - If the node has a left child, enqueue it to the queue.
     - If the node has a right child, enqueue it to the queue.
   - After processing all the nodes at the current level, append the sum `t` to the list `arr`.
3. After the traversal is complete, we use the `nlargest` function from Python's heapq module to fetch the k highest values from the array. The last element in the list returned by `nlargest` will be the k^{th} largest value.
4. Finally, we check if the array contains fewer than k elements. If so, return -1 as there are not enough levels in the tree. Otherwise, we return the last element from the list obtained in the previous step, which is the k^{th} largest level sum.

Following this intuitive approach provides us with a simple and effective solution to identify the k^{th} largest level sum in the given binary tree.

Solution Approach

The solution follows a breadth-first search (BFS) approach, which is perfectly suited for level-by-level traversal in a binary tree. The key data structures and algorithms used are:

- A `deque` from the `collections` module: It functions as a double-ended queue allowing for efficient insertion and removal from both ends. In our BFS, it's used to enqueue and dequeue nodes.
- A list `arr` to keep the sum of values for each level.
- The `nlargest` function from Python's `heapq` module: Heaps are used to obtain the largest (or smallest) elements without sorting the entire list, and `nlargest` is specifically optimized to find the 'n' largest elements in a dataset.

Here's a detailed breakdown of the implementation:

1. Initialize a list `arr` which will hold the sums of node values at each level and a `deque` called `q` with the `root` as the only element. The `deque` structure allows easy access for enqueueing and dequeueing nodes as we traverse the tree level by level.

2. Initiate a `while` loop that continues as long as there are nodes in the queue `q`. This loop continues until every node has been visited.

3. Inside the loop, set a temporary cumulative sum `t` to zero before considering each level. This is crucial for accumulating the sum of node values at the current level.

4. Execute another loop over `q` for the number of nodes at the current level (determined by `len(q)`). Here are the key steps within this loop:

   - Dequeue (`popleft()`) the current node from `q` and add its value to the temporary sum `t`.
   - Check if the current node has a left child (`root.left`). If so, enqueue this child node to `q`.
   - Similarly, check for a right child (`root.right`) and enqueue it if present.

5. After processing all nodes at the current level, append the level sum `t` to the list `arr`.

6. Once the while loop is complete (no more nodes to process in the queue), check if we have at least `k` levels by comparing the length of `arr` with `k`.

7. If there aren't enough levels, we return `-1` as specified. Otherwise, we find the k^{th} largest value in the sums collected by calling `nlargest(k, arr)[-1]`. The `nlargest` function constructs a min-heap of the `k` largest elements from `arr` and returns them as a list. We are only interested in the k^{th} element, which is the last one in the list returned by `nlargest`.

This implementation ensures efficient computation of the solution, leveraging BFS for traversal and a min-heap to effectively get the k^{th} largest value without the need to sort the whole `arr`, which could be more expensive in terms of time complexity.

Example Walkthrough

Let's illustrate the solution approach using a simple example:

```
Consider the following binary tree:

         5
       /   \
      3     8
     / \   / \
    2   4 6   9

And let k = 2, meaning we want to find the 2nd largest level sum.
```

1. Initialize a list `arr = []`. Initialize a queue `q` and put the root node (value 5) in it.

2. Begin the level-order traversal:

   - Level 0 sum: Queue = [5]

     - Dequeue 5, add to temporary sum `t = 5`. Enqueue its left (3) and right (8) children.
     - Level sum = 5, so `arr = [5]`.
   - Level 1 sum: Queue = [3, 8]

     - Dequeue 3, add to `t = 3`. Enqueue its left (2) and right (4) children.
     - Dequeue 8, add to `t = 11`. Enqueue its left (6) and right (9) children.
     - Level sum = 11, `arr = [5, 11]`.
   - Level 2 sum: Queue = [2, 4, 6, 9]

     - Dequeue 2, add to `t = 2`.
     - Dequeue 4, add to `t = 6`.
     - Dequeue 6, add to `t = 12`.
     - Dequeue 9, add to `t = 21`.
     - Level sum = 21, `arr = [5, 11, 21]`.
3. After the traversal, `arr = [5, 11, 21]` now contains the level sums.

4. We use `nlargest` to fetch the 2 largest values from `arr`: `nlargest(2, arr)` gives `[21, 11]`.

5. Since `arr` contains more than `k` elements, we do not return -1.

6. Finally, we return the last element in the list returned by `nlargest` which is the 2nd largest value, `11`.

By following these steps, we efficiently solve the problem and find the 2nd largest level sum in our example binary tree.

Python Solution

```python
from collections import deque
import heapq

# 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 kthLargestLevelSum(self, root: Optional[TreeNode], k: int) -> int:
        # This list will contain the sum of node values at each level.
        level_sums = []
        # Queue for Breadth First Search (BFS).
        queue = deque([root])

        # Perform BFS to traverse the tree level by level.
        while queue:
            # Temporary variable to keep the sum of the current level's nodes.
            current_level_sum = 0
            # Iterate over all nodes at the current level.
            for _ in range(len(queue)):
                node = queue.popleft()
                current_level_sum += node.val
                # Add left child to the queue if it exists.
                if node.left:
                    queue.append(node.left)
                # Add right child to the queue if it exists.
                if node.right:
                    queue.append(node.right)
            # Append the sum of the current level to the list.
            level_sums.append(current_level_sum)

        # If there are fewer levels than k, return -1 as it's not possible to find the kth largest sum.
        if len(level_sums) < k:
            return -1
        # Use nlargest from heapq to find the kth largest element in the list of level sums.
        # nlargest(k, level_sums) will return the k largest elements in the list,
        # and [-1] will get the last element among them, which is the kth largest.
        return heapq.nlargest(k, level_sums)[-1]
```

Java Solution

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

    // Constructors to initialize tree nodes.
    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 {
    // Finds the kth largest level sum in a binary tree.
    public long kthLargestLevelSum(TreeNode root, int k) {
        // List to record the sum of values at each level.
        List<Long> levelSums = new ArrayList<>();

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

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

        // Perform level order traversal to calculate level sums.
        while (!queue.isEmpty()) {
            long levelSum = 0;

            // Iterate over all nodes at the current level.
            for (int count = queue.size(); count > 0; --count) {
                TreeNode currentNode = queue.pollFirst();

                // Accumulate the values of the nodes at this level.
                levelSum += currentNode.val;

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

            // Store the level sum.
            levelSums.add(levelSum);
        }

        // If there are fewer levels than k, return -1.
        if (levelSums.size() < k) {
            return -1;
        }

        // Sort the sums in descending order to find the kth largest sum.
        Collections.sort(levelSums, Collections.reverseOrder());

        // Return the kth element in the sorted list, adjusting the index as list is 0-based.
        return levelSums.get(k - 1);
    }
}
```

C++ Solution

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

// 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:
    // Function to find the kth largest sum of a level in a binary tree.
    long long kthLargestLevelSum(TreeNode* root, int k) {
        // Initialize a vector to store the sum of each level.
        std::vector<long long> levelSums;

        // Queue for level order traversal.
        std::queue<TreeNode*> queue;
        queue.push(root);

        // Perform level order traversal to calculate the sum of each level.
        while (!queue.empty()) {
            // Temporary variable to hold the sum of the current level.
            long long sumOfCurrentLevel = 0;

            // Go through all nodes at the current level.
            for (int levelSize = queue.size(); levelSize > 0; --levelSize) {
                // Get the next node in the queue.
                root = queue.front();
                queue.pop();

                // Add the node's value to the sum of the current level.
                sumOfCurrentLevel += root->val;

                // Add left and right children if they exist.
                if (root->left) {
                    queue.push(root->left);
                }
                if (root->right) {
                    queue.push(root->right);
                }
            }

            // Store the sum of the current level in the vector.
            levelSums.push_back(sumOfCurrentLevel);
        }

        // If there are fewer levels than k, return -1 as it's not possible to find the kth largest level sum.
        if (levelSums.size() < k) {
            return -1;
        }

        // Sort the vector in descending order to find the kth largest level sum.
        std::sort(levelSums.rbegin(), levelSums.rend());

        // Return the kth largest level sum.
        return levelSums[k - 1];
    }
};
```

Typescript Solution

```typescript
// Represents a node in a binary tree with a value, and left and right children
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 kth largest sum of values at each level of a binary tree.
 * @param {TreeNode | null} root - Root node of a binary tree.
 * @param {number} k - The kth largest element to find in the level sums.
 * @return {number} - The kth largest level sum, or -1 if there are less than k levels.
 */
function kthLargestLevelSum(root: TreeNode | null, k: number): number {
    // Array to store the sum of values at each level of the tree
    const levelSums: number[] = [];

    // Queue for breadth-first traversal, starting with the root node
    const queue: Array<TreeNode | null> = [root];

    // Traverse the tree by levels
    while (queue.length) {
        let levelSum = 0; // Sum of values at the current level
        const levelSize = queue.length;

        // Process each node at the current level
        for (let i = 0; i < levelSize; ++i) {
            const currentNode = queue.shift();
            if (currentNode) {
                // Add the value of the current node to the level sum
                levelSum += currentNode.val;
                // Add the left child to the queue if it exists
                if (currentNode.left) {
                    queue.push(currentNode.left);
                }
                // Add the right child to the queue if it exists
                if (currentNode.right) {
                    queue.push(currentNode.right);
                }
            }
        }

        // Store the sum of the current level
        levelSums.push(levelSum);
    }

    // If there are fewer levels than k, return -1
    if (levelSums.length < k) {
        return -1;
    }

    // Sort the sums in descending order to find the kth largest
    levelSums.sort((a, b) => b - a);

    // Return the kth largest level sum
    return levelSums[k - 1];
}
```

Time and Space Complexity

The time complexity of the given code is `O(N + klogN)`, where `N` is the number of nodes in the binary tree. The `O(N)` term comes from the BFS traversal of the tree, which processes each node once. The `O(klogN)` term comes from finding the k-th largest sum using the `nlargest` function from the heap module, which is typically implemented as a heap-based selection algorithm with that complexity.

The space complexity is `O(N)`, as the queue `q` can contain at most all the nodes of the last level in the worst case (considering a complete binary tree), and the array `arr` will also contain a sum for each level of the tree, which can be at most `N` if the tree is skewed (i.e., each level contains only one node).