2792. Count Nodes That Are Great Enough

Medium

Leetcode Link

Problem Description

In this problem, we are given a binary tree where each node has an integer value. Alongside, we are given an integer `k`. The task is to find and count the nodes in the binary tree that are considered "great enough". A node is "great enough" if it meets the following criteria:

- The subtree rooted with this node has at least `k` nodes.
- The value of this node is greater than the value of at least `k` nodes in its subtree. We must explore the tree and count how many nodes satisfy these conditions.

Intuition

The intuition behind the solution is to perform a depth-first search (DFS) starting from the root node, to explore the entire tree. While exploring, we need to keep track of the nodes and their values in a way that enables us to check if a node is "great enough" according to the given conditions.

To check if a node is "great enough", we accumulate the greatest values we encounter in its subtree, including the node itself. We use a min-heap data structure, which allows us to efficiently keep track of the largest elements (since in Python's `heapq` the smallest elements come first, we push negatives of the values so that when we pop, we pop the smallest of the largest values).

Here's the key to our approach:

- Perform DFS from the root.
- For each node, merge the heaps of its left and right children to congregate their greatest values.
- Ensure our heap does not exceed `k` elements.
- Once merged, we compare the negative of the top of our min-heap (which is the `k`th largest value) with current node's value to determine if our node is "great enough".
- If it is, increment our counter `ans`.
- Finally, push the negative of the current node's value onto our heap (to include the current node in the subtree for future parent nodes) before returning the heap to its parent call.

The main advantages of this approach are:

- Space efficiency, as we carry a single heap up the tree while we are gathering subsets of values.
- Time efficiency, since heaps provide O(log k) pushing and popping, making the overall complexity better compared to sorting the values or other brute force methods that might seem tempting at first glance.

Solution Approach

The provided solution uses a Depth-First Search (DFS) algorithm in combination with a min-heap to solve the problem. Here is a step-by-step explanation of the implementation:

- A DFS `dfs` function is defined, which will be called recursively to explore the tree nodes. At each node, this function will return a min-heap containing the largest values (negated) from its subtree, with size at most `k`.

- Within this `dfs` function, when the base case is encountered (i.e., when a `None` node is encountered), an empty heap is returned, indicating that a leaf has been reached.

- When at a non-leaf node, DFS is called on both the `left` and `right` children, and their returned heaps are merged. It's important to note that we're using the `push` helper function to maintain the heap's size at no more than `k`.

- The `push` helper function works as follows: It pushes an element onto the heap and if the heap size is greater than `k` after the push, it pops the smallest element (which is represented as the largest negative value). This ensures that only the `k` largest (negated) elements remain.

- Back in the `dfs` function, after merging the heaps from the left and right subtrees, we check if the current node's value is "great enough". This check is done by ensuring that the heap size is exactly `k` and the top element of the heap (i.e., the smallest of the `k` largest negated values) is less than the negated value of the current node. If these conditions are met, it means the current node is greater than at least `k` nodes in its subtree, and we increment our answer `ans`.

- Finally, the current node's value (negated) is pushed onto the heap, and the heap is returned to be merged in the parent node's DFS call.

- After the DFS traversal completes, the counter `ans` holds the number of "great enough" nodes, and this value is returned as the solution to the problem.

Here is the pseudocode for the DFS and push function:

```
def dfs(node):
    if node is None:
        return new empty heap
    left_heap = dfs(node.left)
    right_heap = dfs(node.right)
    # Merge heaps
    for value in right_heap:
        push(left_heap, value)
    # Check if the current node is "great enough"
    if heap size is k and (-top of heap < node's value):
        ans += 1
    push(left_heap, -node's value)
    return left_heap

def push(heap, value):
    add value to heap
    if size of heap > k:
        remove the smallest element from heap
```

Notice that the `dfs` function's merge step iterates over the right heap and pushes its elements into the left heap. This merging avoids creating a new heap at each node, which saves memory and potentially decreases runtime due to fewer allocations.

Example Walkthrough

Let's use a small binary tree example and `k = 2` to illustrate the solution approach.

Imagine a binary tree like this:

```
      5
     / \
    3   8
   / \   \
  1   4   10
```

In this tree, we want to count the number of "great enough" nodes, i.e., nodes with at least `k` nodes in their subtree and a value greater than at least `k` nodes in their subtree.

1. Begin DFS at the root node (5). Since it's not `None`, we proceed.
2. Perform DFS on the left child (3).
   - Left child (3) is not `None`, we perform DFS on its children.
     - Left child (1) has no children (both `None`), so it returns an empty heap.
     - Right child (4) has no children (both `None`), so it returns an empty heap.
   - Merge returned heaps from 1 and 4 and push negated values: `left_heap` contains `[-4, -3]`.
   - Now, `left_heap` has 2 elements (size is `k`) and `top of heap` is -4, which is smaller than `-3`. So, node 3 is "great enough".
   - Increment `ans` to 1 and push `-3` into the `left_heap` before returning it.
3. Perform DFS on the right child (8).
   - Right child (8) is not `None`, perform DFS on its left and right children.
     - It has no left child, so returns an empty heap.
     - Right child (10) has no children (both `None`), so it returns an empty heap.
   - Merge returned heaps and push negated values: `right_heap` contains `[-10]`.
   - Heap size is less than `k`, so we cannot say node 8 is "great enough" yet.
   - Push `-8` into the `right_heap` before returning it.
4. Back to the root now, `left_heap` contains `[-4, -3]` and `right_heap` contains `[-10, -8]`. Merge them.
   - After merging, the combined heap will have `[-10, -8, -4, -3]`. We need to maintain only the largest `k` values, so we pop off -3 to get `left_heap` as `[-10, -8, -4]`.
   - The `top of heap` is now -4, which is smaller than `-5`. Thus, the root node (5) is "great enough".
   - Increment `ans` to 2. Push `-5` into `left_heap` and return `left_heap`.
5. After DFS traversal, the `ans` is 2, representing the number of "great enough" nodes (nodes 3 and 5).

Thus, the output for our binary tree example with `k = 2` is 2, meaning there are 2 nodes that are "great enough" by those criteria.

Python Solution

```python
# 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 countGreatEnoughNodes(self, root: Optional[TreeNode], k: int) -> int:
        from heapq import heappush, heappop

        # Helper method to maintain a heap of size 'k'
        def push_to_heap(heap, val):
            # Add 'val' to 'heap' and ensure the heap size does not exceed 'k'
            heappush(heap, val)
            # If the heap size is > k, pop the smallest element
            if len(heap) > k:
                heappop(heap)

        # DFS traversal to count the great enough nodes
        def dfs(node):
            if node is None:
                # If the node is None, return an empty heap
                return []
            # Recursively traverse left and right subtrees
            left_heap = dfs(node.left)
            right_heap = dfs(node.right)
            # Combine heaps from children and maintain size 'k'
            for val in right_heap:
                push_to_heap(left_heap, val)
            # Check if current node's value is greater than k-th value (if heap size is k)
            if len(left_heap) == k and -left_heap[0] < node.val:
                # If condition satisfies, increment the result
                nonlocal count
                count += 1
            # Add current node's value to the heap (negated to convert to max-heap)
            push_to_heap(left_heap, -node.val)
            # Return the updated heap
            return left_heap

        # Initialize count of great enough nodes as 0
        count = 0
        # Start DFS from the root
        dfs(root)
        # Return the final count
        return count
```

Java Solution

```java
class Solution {
    private int greatEnoughCount; // Counter to keep track of nodes meeting the condition
    private int threshold; // The value k used as a threshold to compare node values

    /**
     * Counts the number of nodes in a binary tree where the node's value
     * is greater than the k-th largest value of all nodes from the root to that node.
     *
     * @param root Root of the binary tree.
     * @param k The k-th largest value to be considered.
     * @return The count of nodes meeting the mentioned criterion.
     */
    public int countGreatEnoughNodes(TreeNode root, int k) {
        this.threshold = k;
        this.greatEnoughCount = 0;
        depthFirstSearch(root);
        return greatEnoughCount;
    }

    /**
     * Performs a depth-first search on the binary tree, examining each node
     * and updating the greatEnoughCount when conditions are met using priority queue
     * for finding the k-th largest value from the root to the current node.
     *
     * @param node The current node being examined.
     * @return A priority queue containing the largest values encountered up to the current node,
     *         with size maintained at most k.
     */
    private PriorityQueue<Integer> depthFirstSearch(TreeNode node) {
        // If we reach a null node, return a priority queue that orders elements in reverse order
        if (node == null) {
            return new PriorityQueue<>(Comparator.reverseOrder());
        }

        // Recursively traverse left and right subtrees
        PriorityQueue<Integer> leftQueue = depthFirstSearch(node.left);
        PriorityQueue<Integer> rightQueue = depthFirstSearch(node.right);

        // Merge the right queue into the left queue
        for (int value : rightQueue) {
            leftQueue.offer(value);
            // If the size of the queue exceeds k, remove the smallest element
            if (leftQueue.size() > threshold) {
                leftQueue.poll();
            }
        }

        // If the k-th largest value is defined and is less than the current node's value
        if (leftQueue.size() == threshold && leftQueue.peek() < node.val) {
            greatEnoughCount++; // Increment our count
        }

        // Add the current node's value to the queue
        leftQueue.offer(node.val);
        // Ensure the size of the queue doesn't exceed k
        if (leftQueue.size() > threshold) {
            leftQueue.poll();
        }

        return leftQueue; // Return the priority queue holding the largest k values up to and including this node
    }
}
```

C++ Solution

```cpp
#include <functional>
#include <queue>

// 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:
    // This function returns the number of nodes in a binary tree 'root'
    // whose values are greater than all of the last 'k' visited nodes
    // in a depth-first search order.
    int countGreatEnoughNodes(TreeNode* root, int k) {
        int answer = 0;

        // Define a lambda function that uses Depth-First Search (DFS) to iterate
        // through the tree and keep track of the largest 'k' values found so far.
        std::function<std::priority_queue<int>(TreeNode*)> dfs = [&](TreeNode* node) -> std::priority_queue<int> {
            // If the node is null, return an empty priority queue.
            if (!node) {
                return std::priority_queue<int>();
            }

            // Recursively apply DFS to the left and right subtrees.
            auto leftMaxes = dfs(node->left);
            auto rightMaxes = dfs(node->right);

            // Combine the max values from left and right subtrees.
            while (!rightMaxes.empty()) {
                leftMaxes.push(rightMaxes.top());
                rightMaxes.pop();

                // Maintain the size of the priority queue as 'k'.
                if (leftMaxes.size() > k) {
                    leftMaxes.pop();
                }
            }

            // Check if the current node's value is greater than the smallest
            // value in the max 'k' values seen so far (means it's greater than
            // all 'k' values because all others are larger).
            if (leftMaxes.size() == k && leftMaxes.top() < node->val) {
                answer++;
            }

            // Add the current node's value to the collection of max values.
            leftMaxes.push(node->val);

            // Ensure the size of the priority queue does not exceed 'k'.
            if (leftMaxes.size() > k) {
                leftMaxes.pop();
            }

            return leftMaxes;
        };

        // Start the DFS from the root of the tree.
        dfs(root);

        // Return the final count of nodes that meet the condition.
        return answer;
    }
};
```

Typescript Solution

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

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// This function performs Depth-First Search on the binary tree to find 'k' largest values.
const dfs = (node: TreeNode | null, k: number, answer: { count: number }): number[] => {
    if (node === null) {
        return [];
    }

    // Recursively traverse left and right children.
    const leftMaxes = dfs(node.left, k, answer);
    const rightMaxes = dfs(node.right, k, answer);

    // Flatten left and right max values and sort them in descending order.
    const maxes = leftMaxes.concat(rightMaxes).sort((a, b) => b - a).slice(0, k);

    // If the current node's value is greater than all 'k' values seen so far, increment the count.
    if (maxes.length === k && maxes[maxes.length - 1] < node.val) {
        answer.count++;
    }

    // Add the current node's value to the list of maxes and return.
    maxes.push(node.val);
    return maxes.sort((a, b) => b - a).slice(0, k);
};

// This function returns the number of nodes in a binary tree 'root'
// whose values are greater than all of the last 'k' visited nodes
// in a depth-first search order.
const countGreatEnoughNodes = (root: TreeNode | null, k: number): number => {
    // Initialize count as a reference object to keep track during recursive calls.
    let answer = { count: 0 };

    // Start the DFS from the root of the tree.
    dfs(root, k, answer);

    // Return the final count of nodes that meet the condition.
    return answer.count;
};

// Example usage:
const rootNode = new TreeNode(2, new TreeNode(1), new TreeNode(3));
const result = countGreatEnoughNodes(rootNode, 2);
console.log(result);  // Output should be based on the tree structure and 'k' value.
```

Time and Space Complexity

The time complexity of the given `countGreatEnoughNodes` function is `O(n * log(n))`, where `n` is the number of nodes in the binary tree. The function performs a depth-first search (DFS) on the tree, visiting each node once (`O(n)`). At each node, the `push` function potentially performs a heap push and pop operation in the priority queue. The length of the priority queue is capped at `k`, so each heap operation is `O(log(k))`. Since `k` can, at maximum, be as large as `n` (the total number of nodes), in the worst-case scenario, this becomes `O(log(n))`. When multiplied by the total number of nodes visited, we obtain `O(n * log(n))`.

The space complexity of the function is `O(n)`. This is because the recursive DFS function call stack could potentially store every node in a path from the root to a leaf in the case of a skewed binary tree. In addition, the priority queue `l` also stores up to `k` elements, but since `k` is at most `n`, it does not exceed `O(n)` space. Thus, the overall space complexity is determined by the maximum between the height of the tree and the number length of the priority queue, which is `O(n)`.