2445. Number of Nodes With Value One

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

Problem Description

In this problem, we are given a binary tree with `n` number of nodes labeled from `1` to `n`. The tree is connected, undirected, and, importantly, each node's parent is defined by the floor division of its label by 2, where the floor function returns the greatest integer less than or equal to its argument. For instance, `floor(3/2)` is `1`. This layout suggests that every node (except the root, labeled `1`) has exactly one parent, and node `1` is the root of the tree.

The objective is to process a series of queries. Each query corresponds to a node's label, and the query's operation is to flip the value of all nodes in the subtree rooted at the given node's label. Flipping a value means that if a node's value is `0`, it becomes `1`, and vice versa. Initially, all nodes have a value of `0`.

Our goal is to determine the total number of nodes with the value `1` after processing all queries. A query might be repeated, and if a node's value is flipped an odd number of times, it ends up with a value of `1`.

Intuition

To solve this problem, we take a stepwise approach. The total number of flips on a single node equals the number of times this node appeared in the queries. So, we first count the occurrences of each node in the queries using a frequency counter.

Next, we iterate over our counter, and for each node that has been queried an odd number of times (since an even number of flips would cancel out), we perform the flip operation on that node's subtree.

To flip a node's subtree iteratively, we use a depth-first search (DFS) strategy. Starting from the given node (which appears in queries), we recursively flip the current node's value and then perform the same operation on its left and right children, which are located at indices `2*i` and `2*i+1` respectively in a binary tree context. Given that it's a binary tree, each node could have up to two children, and this is fulfilled through bit manipulation: the left child is at index `i << 1`, and the right child is at index `i << 1 | 1`.

After processing all relevant queries, we sum up the values in the `tree` array to find the total count of nodes with value `1`.

This method works efficiently because we only perform subtree flips for nodes that were affected by an odd number of queries. We also avoid redundant work by skipping the count of nodes that would ultimately have a value of `0` after an even number of flips.

Solution Approach

The implementation of the solution aligns with the intuition described and makes use of a simple recursive depth-first search (DFS) algorithm along with bitwise manipulation techniques, as follows:

1. **Counter Collection**: We initiate by creating a `Counter` object for collecting the frequency of each node in the queries. The `Counter` is well-suited for this task as it efficiently tallies the counts with minimal manual bookkeeping.
2. **Tree Initialization**: A `tree` array is initialized with `n + 1` elements, all set to `0`. This extra element accounts for the 1-based node labeling. Each index i of the array will represent the value of the respective node labeled `i`.
3. **DFS Function**: A nested DFS function, `dfs(i)`, is defined to flip the subtree starting from node `i`. The function first checks if the node `i` is within bounds (i.e., does not exceed `n`). It then flips the value at `tree[i]` using `tree[i] ^= 1` (bitwise XOR operation), where flipping is done effectively by XOR-ing the current value with `1`. After that, recursive calls are made to flip the left and right children of the node `i` using `dfs(i << 1)` and `dfs(i << 1 | 1)` respectively. The left shift operator (`<<`) doubles the index, representing the left child, and the bitwise OR with `1` (`| 1`) obtains the right child's index.
4. **Processing Queries**: We iterate over the items in the `Counter`, checking each node label and associated count. If the count is odd (checked using `v & 1`), we call the `dfs(i)` function to flip the subtree rooted at that node.
5. **Summing the Values**: After processing all queries, we sum up the `tree` array, excluding the zero index (as it's not used), to get the total number of nodes that have a value of `1`.

This algorithm traverses the tree only once for every node that needs to be flipped, resulting in a time complexity proportional to the number of queries and the size of the subtrees that need to be flipped. Space complexity is linear to the size of the tree since we maintain an array of size `n + 1`.

Here is a breakdown of the main components in the code:

```python
cnt = Counter(queries)  # Using Counter to tally queries
tree = [0] * (n + 1)  # Tree array initialized

def dfs(i):  # Recursive DFS definition
    if i > n:  # Check bounds
        return
    tree[i] ^= 1  # Flip current value
    dfs(i << 1)  # Recur left
    dfs(i << 1 | 1)  # Recur right

for i, v in cnt.items():
    if v & 1:  # Is the count odd?
        dfs(i)  # Flip subtree if needed

return sum(tree)  # Sum the tree values for the result
```

The use of bitwise operations for index calculation and value flipping makes the implementation succinct and efficient.

Example Walkthrough

Let's consider a binary tree with 3 nodes. Now, here are the steps to illustrate the solution approach:

Step 1: Queries Counter Collection

Suppose we have queries with node labels `[2, 1, 2]`. Using a `Counter`, we count the occurrences: `{1: 1, 2: 2}`.

Step 2: Tree Initialization

We initialize our tree with `n + 1` elements all set to `0`, resulting in `tree = [0, 0, 0, 0]`.

Step 3: Define DFS Function

We define a `dfs` function that flips the value of a node and recursively does the same for its children.

Step 4: Processing Queries with DFS

According to the `Counter`, node `1` has been queried once, and node `2` has been queried twice. Queried node `2` is even, so flipping it twice will bring it back to `0`. Node `1` is odd, so we flip it and its subtree.

Flipping Node 1 and Its Subtree:

- Flip node `1`: `tree = [0, 1, 0, 0]`
- Flip its left child (`2*1 = 2`): `tree = [0, 1, 1, 0]`
- Flip its right child (`2*1 + 1 = 3`): `tree = [0, 1, 1, 1]`

Step 5: Summing the Values

Now, we sum up values from index `1` onwards to find the number of nodes with value `1`: `sum(tree[1:])`, which gives us `3`.

The result is `3`, meaning after processing all queries, there are 3 nodes with the value `1`.

This example illustrates how the given solution processes the nodes and their respective subtrees based on the queries, flip operation, and the use of DFS in an efficient manner.

Python Solution

```python
from collections import Counter

class Solution:
    def numberOfNodes(self, n: int, queries: List[int]) -> int:
        # Helper function to switch the state of the nodes in the tree.
        # It performs a depth-first search to toggle each node starting from index i.
        def toggle_nodes_dfs(index):
            # Base case: if the index is greater than n, return since it's not a valid node.
            if index > n:
                return
            # Toggle the current node's state.
            tree[index] ^= 1
            # Recursively toggle the state of the left child.
            toggle_nodes_dfs(index << 1)
            # Recursively toggle the state of the right child.
            toggle_nodes_dfs(index << 1 | 1)

        # Initialize the tree with 0s, each index represents a node, and the state is off (0).
        tree = [0] * (n + 1)
        # Count the occurrences of each query.
        query_counts = Counter(queries)
        # Iterate over items in query_counts to perform toggles for odd counts.
        for index, frequency in query_counts.items():
            # If the frequency of toggles is odd, we need to perform the toggle operations.
            if frequency & 1:
                toggle_nodes_dfs(index)

        # Sum up all the nodes' states to get the total number of nodes that are on (1).
        return sum(tree)

```

Java Solution

```java
class Solution {
    // The 'tree' array represents a binary tree where each node can have two states: 0 or 1.
    private int[] tree;

    // Method to calculate the number of nodes with state 1 after processing the queries.
    public int numberOfNodes(int n, int[] queries) {
        // Initialize the tree to have 'n + 1' nodes (indexing from 0 to n).
        tree = new int[n + 1];
        // Array to keep count of occurrences of each value in queries.
        int[] count = new int[n + 1];
        // Count the occurrences of each value in the queries.
        for (int value : queries) {
            ++count[value];
        }
        // Toggle the state of each node with an odd count.
        for (int i = 0; i < n + 1; ++i) {
            if (counter[i] % 2 == 1) {
                dfs(i); // Perform a depth-first search starting from node 'i'.
            }
        }
        // Calculate the answer by summing up the states of all nodes.
        int answer = 0;
        for (int i = 0; i < n + 1; ++i) {
            answer += tree[i];
        }
        return answer;
    }

    // Helper method to toggle the states of the current node and its children.
    private void dfs(int index) {
        // If the index is beyond the length of the tree array, end the recursion.
        if (index >= tree.length) {
            return;
        }
        // Toggle the current node's state.
        tree[index] ^= 1;
        // Recursively toggle the states of the left and right children nodes (if they exist).
        dfs(index << 1);       // Left child index is current index shifted one bit to the left.
        dfs(index << 1 | 1);   // Right child index is left child index with a bit-wise OR with 1.
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <numeric> // For std::accumulate
#include <functional> // For std::function

class Solution {
public:
    // Function to count the number of nodes set to '1' in a binary tree
    // after applying the queries.
    int numberOfNodes(int n, std::vector<int>& queries) {
        std::vector<int> tree(n + 1, 0); // Tree array to keep track of node status (0 or 1).
        std::vector<int> count(n + 1, 0); // Count array to record the number of times each node is queried.

        // Increment count for each query value.
        for (int value : queries) ++count[value];

        // Recursive function to toggle the status of nodes.
        std::function<void(int)> toggleNodes = [&](int index) {
            if (index > n) return; // Base case for recursion: index out of bounds.
            tree[index] ^= 1; // Toggle the node status using XOR operation.
            toggleNodes(index << 1); // Recurse for the left child.
            toggleNodes(index << 1 | 1); // Recurse for the right child.
        };

        // Apply the queries to the tree according to the count.
        for (int i = 0; i < n + 1; ++i) {
            // If the count is odd, toggle the node status.
            if (count[i] & 1) {
                toggleNodes(i);
            }
        }

        // Sum up the values in the tree array to count the nodes that are set to '1'.
        return std::accumulate(tree.begin(), tree.end(), 0);
    }
};

```

Typescript Solution

```typescript
// Function to count the number of nodes set to '1' in a binary tree
// after applying the queries.
function numberOfNodes(n: number, queries: number[]): number {
    // Tree array to keep track of node status (0 or 1).
    let tree: number[] = new Array(n + 1).fill(0);
    // Count array to record the number of times each node is queried.
    let count: number[] = new Array(n + 1).fill(0);

    // Increment count for each query value.
    for (let value of queries) count[value]++;

    // Recursive function to toggle the status of nodes.
    let toggleNodes = (index: number) => {
        if (index > n) return; // Base case for recursion: index out of bounds.
        tree[index] ^= 1; // Toggle the node status using XOR operation.
        toggleNodes(index << 1); // Recurse for the left child.
        toggleNodes(index << 1 | 1); // Recurse for the right child.
    };

    // Apply the queries to the tree according to the count.
    for (let i = 0; i <= n; i++) {
        // If the count is odd, toggle the node status.
        if (count[i] % 2 === 1) {
            toggleNodes(i);
        }
    }

    // Sum up the values in the tree array to count the nodes that are set to '1'.
    return tree.reduce((accumulator, currentValue) => accumulator + currentValue, 0);
}

export { numberOfNodes }; // Optionally, export the function to be used in other modules.

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is determined by the number of times the `dfs` function is called, which depends on the structure of the tree and how many times each node is toggled.

1. The tree has `n` nodes and in the worst-case scenario, for a complete binary tree, the `dfs` function would perform a full traversal to toggle nodes below it.
2. Each query potentially leads to a full traversal from the current node to all nodes below it.
3. If a node is toggled an even number of times, it ends up in its original state. Only nodes toggled an odd number of times will affect the final count.

Considering these points:

- The maximum height of the tree is `log2(n)` and the maximum number of nodes at the last level is `n/2` (in a complete binary tree).
- For each toggle operation, in the worst-case scenario, we may have to visit all nodes of the subtree rooted at the given node.
- The maximum number of operations for a single query is `O(n)` since in the worst case we may need to toggle all nodes in the tree.

Assuming `m` is the number of unique queries, the worst-case time complexity for odd-toggled nodes is `O(m * n)`, as every query can potentially result in traversing and toggling the entire tree.

Space Complexity

The space complexity depends on:

1. The `tree` array, which requires `O(n)` space to store the toggle state of each node.
2. The recursion stack for the `dfs` function, which in the worst-case scenario, may go up to `O(log n)` in depth for a balanced binary tree due to the binary tree's height.
3. The `Counter` object, which stores the counts of each query. In the worst case, if all queries are unique, it requires `O(m)` space.

Hence, the overall space complexity is `O(n + log n + m)`. However, since `m` can at most be `n` (if all nodes are queried once), and `log n` is lesser than `n`, the more dominant term is `O(n)`, which simplifies our space complexity to `O(n)`.