1506. Find Root of N-Ary Tree

Medium | Bit Manipulation | Tree | Depth-First Search | Hash Table | Leetcode Link

Problem Description

In this problem, we are given all the nodes of an N-ary tree in the form of an array of `Node` objects. Each `Node` has a unique value, and the goal is to determine the root node of the given N-ary tree.

An N-ary tree is a tree in which a node can have any number of children (0 or more). The array provided does not directly show the structure of the tree or the parent-child relationships. Instead, we need to infer which one of these nodes is the root based on the details given.

To aid in this, we have a way to serialize the tree through its level order traversal: each group of children is followed by a `null` value to indicate the end of that particular level of children. The given example uses serialization to show the structure of an N-ary tree.

Our task is to implement a function, `findRoot`, which will receive an unsorted array of `Node` objects and must return the root of the N-ary tree. A key aspect of the problem is that the underlying tree structure is not provided explicitly, and we have to determine the root node without reconstructing the tree.

Intuition

The intuition behind the provided solution involves using a property of XOR (exclusive or) operation, which is that `x ^ x = 0` for any integer `x`, and `x ^ 0 = x`. In essence, if you XOR an even number of the same number, you get `0`, but if you have an odd number of a number (which would be the case with the root value because it is not someone's child), you'd be left with that number.

To find the root of the tree, the solution exploits the fact that the root is the only node that does not appear as a child of any other node. By XOR'ing all the values of the nodes and their children, every node except the root will cancel out because they will appear twice—once as a node value and once as a child value.

The algorithm iterates through all nodes, XOR'ing the node's value and XOR'ing the values of its children. Since all nodes except the root will appear twice and XOR'ing a number with itself yields 0, the final result of the XOR operations will be the value of the root node, because it's the only one that doesn't get canceled out.

At the end of the XOR operation, we are left with a value that corresponds to the root node's value. We can then easily find the root node by comparing the value obtained from our XOR operation with the values of each node in the array and returning the node that matches.

Solution Approach

The solution implements an efficient approach to find the root of an N-ary tree without rebuilding the tree structure. The primary algorithmic idea uses bitwise XOR (exclusive or) to identify the root node. The XOR operation has a unique property where XORing a number with itself results in 0, and XORing a number with 0 gives the number back. Furthermore, XOR is commutative and associative, which means the order of operations does not matter.

Algorithm:

1. Initialize an accumulator variable `x` to 0. This variable will be used to collect the result of consecutive XOR operations.
2. Iterate over each node in the tree array. a. XOR the node's value with the accumulator `x`. b. Iterate over each child of the current node and XOR the child's value with `x`.
3. After the completion of the iterations, `x` will hold the value of the root node. This occurs because the root is the only node not XOR'd twice (once as a node and once as a child).
4. Iterate through the tree array again and find the node with the value equal to `x`. This is the root node.

The implementation uses two `for` loops:

- The first is to apply the above-described XOR operations.
- The second is to find and return the node whose value matches the result of the XOR operations.

Data Structures:

- The primary data structure used for storing nodes is the given array `tree`, which holds `Node` objects.
- A single integer `x` is used to accumulate the XOR results.

Patterns:

- Bit manipulation via XOR is the main pattern that allows us to avoid reconstructing the entire tree.

The provided solution utilizes minimal additional space and performs the task in linear time relative to the number of nodes, making it an efficient approach for this problem.

Reference Solution Code:

```python
class Solution:
    def findRoot(self, tree: List['Node']) -> 'Node':
        x = 0
        for node in tree:
            # XOR the node's value
            x ^= node.val
            # XOR the values of children
            for child in node.children:
                x ^= child.val

        # Identify the node with the accumulated XOR value (the root)
        return next(node for node in tree if node.val == x)
```

The `next` function in the last line is a Python built-in that returns the next item from the iterator, in this case, the node for which `node.val == x`.

As we can see from the code, the solution is concise, leveraging the XOR operation to cleverly discern the root node from a pool of nodes without explicit parent-child linkage information.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we are given an array of `Node` objects representing an N-ary tree:

```
Nodes: [1, 2, 3, 4, 5, 6]
Edges: 1 -> [3, 5, 6], 2 -> [4], 3 -> [2]
(Here, "1 -> [3, 5, 6]" means that the node with value 1 has children with values 3, 5, and 6, and similarly for the others.)
```

Though the problem doesn't provide the tree structure, this is added here for illustrative purposes.

So our `Node` objects array (unsorted) could look something like this:

```
tree = [
    Node(1, [Node(3, [Node(2, [Node(4)])]), Node(5), Node(6)]),
    Node(3, [Node(2, [Node(4)])]),
    Node(5),
    Node(6),
    Node(2, [Node(4)]),
    Node(4)
]
```

Let's walk through the XOR operation as described:

Initial `x`: 0

- Processing Node(1): `x = x ^ 1 (x becomes 1)` Child Node(3): `x = x ^ 3 (x becomes 2)` Child Node(5): `x = x ^ 5 (x becomes 7)` Child Node(6): `x = x ^ 6 (x becomes 1)`
- Processing Node(3): `x = x ^ 3 (x becomes 2)` Child Node(2): `x = x ^ 2 (x becomes 0)` Child Node(4): `x = x ^ 4 (x becomes 4)`
- Processing Node(5): `x = x ^ 5 (x becomes 1)`
- Processing Node(6): `x = x ^ 6 (x becomes 7)`
- Processing Node(2): `x = x ^ 2 (x becomes 5)` Child Node(4): `x = x ^ 4 (x becomes 1)`
- Processing Node(4): `x = x ^ 4 (x becomes 5)`

Notice how every node that is not the root appeared twice in the XOR operation due to being a child of some other node, canceling itself out and leaving us with `x = 1` at the end, which is the value of the root.

Now, we can go through the array of nodes again to find the node with value `1`, which is our root node.

Using this approach, the algorithm identifies the root node without reconstructing the tree's structure. It is efficient because each node and its children are processed exactly once, with the XOR operation ensuring that only the root node's value remains at the end.

Python Solution

```python
# The given code is already written in Python 3 syntax, but I will revise it to
# include clearer variable names and add comments to enhance readability.

# Definition for a Node in a tree.
class Node:
    def __init__(self, value=None, children=None):
        self.value = value  # The value contained in the node
        self.children = children if children is not None else []  # Child nodes


# Class to encapsulate the solution methods.
class Solution:
    # Method to find the root of a tree where all nodes are present in an array.
    # The tree has no cycles and each child has exactly one parent, so the root
    # has no parent.
    def findRoot(self, all_nodes: List['Node']) -> 'Node':
        # Initialize an integer to use it for XOR operation.
        # XOR is used because it cancels out when applied to a pair of the same numbers.
        xor_sum = 0

        # Loop over each node and its children in the list of all nodes.
        for node in all_nodes:
            # XOR the node's value with the xor_sum. If it's the root's value,
            # it will appear only once and stay in the xor_sum as all other non-root
            # nodes will be cancelled out with their children counterpart.
            xor_sum ^= node.value
            # Loop over the children of the current node.
            for child in node.children:
                # XOR each child's value as well, cancelling out their values.
                xor_sum ^= child.value

        # After the above operation, the xor_sum will contain the value of the root node only.
        # Loop over the nodes once more to find the node with the same value as the xor_sum.
        return next(node for node in all_nodes if node.value == xor_sum)

```

Java Solution

```java
import java.util.List;
import java.util.ArrayList;

// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;

    public Node() {
        children = new ArrayList<>();
    }

    public Node(int value) {
        val = value;
        children = new ArrayList<>();
    }

    public Node(int value, ArrayList<Node> childrenList) {
        val = value;
        children = childrenList;
    }
}

class Solution {

    // Function to find the root of the given N-ary tree.
    public Node findRoot(List<Node> tree) {
        // Initialize a variable that will be used for xor operation
        int xorSum = 0;

        // Iterate through each node of the tree
        for (Node node : tree) {
            // Xor the current node value
            xorSum ^= node.val;

            // Iterate through the children of the current node
            for (Node child : node.children) {
                // Xor the value of each child
                xorSum ^= child.val;
            }
        }

        // After the above loops, xorSum will have the value of the root node (as it's only counted once)

        // Now, search for the node with the value equal to xorSum (which is the root)
        for (Node potentialRoot : tree) {
            if (potentialRoot.val == xorSum) {
                // If the value matches, we have found the root
                return potentialRoot;
            }
        }

        // The code should never reach this point, as the root must be in the tree,
        // we do not need to handle the case where the root isn't found
        return null; // This return added just to satisfy the function's return type contract
    }
}

```

C++ Solution

```cpp
#include <vector>

// Forward declaration for Node.
class Node{
public:
    int val; // Node's value
    std::vector<Node*> children; // Children of the node

    // Constructor for a node without children.
    Node() {}

    // Constructor for a node with a given value.
    Node(int _val) {
        val = _val;
    }

    // Constructor for a node with a given value and a list of children.
    Node(int _val, std::vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};

class Solution {
public:
    Node* findRoot(std::vector<Node*> tree) {
        int xorSum = 0; // Initialize the XOR accumulator.

        // Calculate the XOR of all the node values and their children's values.
        for (Node* node : tree) {
            // XOR with the current node's value.
            xorSum ^= node->val;
            // XOR with each of the node's children values.
            for (Node* child : node->children) {
                xorSum ^= child->val;
            }
        }

        // Find the node whose value is equal to the xorSum result,
        // this is the root node because its value will only be counted once (all other nodes will be counted twice).
        for (Node* node : tree) {
            if (node->val == xorSum) {
                return node; // Return the root node.
            }
        }

        // Note: The infinite loop in the original code is removed as the return within the loop
        // will always exit the method once the root node is found or the loop ends.

        // In case no root is found (which shouldn't happen), return nullptr.
        return nullptr;
    }
};

```

Typescript Solution

```typescript
interface INode {
    val: number;
    children: INode[];
}

/**
 * Find the root of a tree based on the property that the root's value
 * will appear an odd number of times when XOR'ing all values together.
 * @param {INode[]} tree - An array of nodes representing a tree.
 * @return {INode | null} - The root node or null if not found.
 */
function findRoot(tree: INode[]): INode | null {
    let xorSum = 0; // This will hold the XOR sum of all node values.

    // Iterate over all nodes in the tree.
    for (const node of tree) {
        // XOR the current node's value.
        xorSum ^= node.val;

        // XOR the values of the current node's children.
        for (const child of node.children) {
            xorSum ^= child.val;
        }
    }

    // Find and return the node whose value matches the xorSum.
    // This is the root node, as its value will only be XOR'd once.
    return tree.find(node => node.val === xorSum) || null;
}

```

Time and Space Complexity

The provided code calculates the root of an N-ary tree. It uses bitwise XOR to identify the root node, under the assumption that each value in the tree is unique.

- **Time Complexity**: The time complexity of the code is `O(N)`, where `N` is the total number of nodes in the tree. This is because the code iterates through all nodes exactly once and iterates once more through all the children of each node.
- **Space Complexity**: The space complexity of the function is `O(1)`, not considering the space taken by the input itself. This is because the code only uses a variable `x` to keep track of the XOR operation and does not use any additional data structures that grow with the size of the input.