1261. Find Elements in a Contaminated Binary Tree

Medium · Tree · Depth-First Search · Breadth-First Search · Design · Hash Table · Binary Tree · Leetcode Link

Problem Description

The problem presents a unique type of binary tree structure where the value of each node can be determined by its parent node's value using specific rules. For a node with value 'x', if it has a left child, the left child's value would be `2 * x + 1`, and if it has a right child, the right child's value would be `2 * x + 2`. However, there is a twist: the binary tree is contaminated, meaning all node values are converted to `-1`. The `FindElements` class is designed to recover such a tree and perform searches for values after its recovery.

Intuition

The intuitive approach to solve this problem involves two phases: recovery and search. In the recovery phase, we need to reinstate each node's value based on the rules mentioned in the problem description. We start at the root node, which is set to 0, and traverse the tree using depth-first search (DFS). While traversing, we correct the value of each node according to its position (whether it's a left or right child) using the given formulas.

Once the values are recovered and set, we store them in a set during the traversal. This allows us to later check if a target value exists efficiently. The recovery process ensures that all values in the tree are unique, thus a set is the perfect data structure for quick lookup operations.

The search phase is quite simple. Since we've stored recovered values in a set, finding a target value is now a matter of checking its presence in the set, which is a constant time operation.

Solution Approach

To implement the solution, we utilize a depth-first search (DFS) algorithm starting from the root of the tree. DFS is a tree traversal technique that starts at the root node and explores as far as possible along each branch before backtracking. The algorithm has been slightly modified to recover and store the values of each node during traversal.

Here's a breakdown of how the algorithm works in the given solution:

1. **Initialization**:
   - We override the root value to `0` since it's given that `root.val == 0` when the tree is not contaminated.
   - A set named `self.vis` is initialized to keep track of all the recovered values in the tree. The set is chosen because it supports `O(1)` average time complexity for search operations.
2. **DFS Traversal and Recovery**:
   - We define a helper function `dfs`, which takes a node as an argument and performs the following operations:
     - It first adds the current node's value to the set `self.vis`.
     - If the current node has a left child, it calculates the left child's value using the formula `2 * node.val + 1` and then recursively calls `dfs` on the left child.
     - If the current node has a right child, the right child's value is set using `2 * node.val + 2`, and `dfs` is recursively called on the right child.
   - This DFS algorithm continues until all nodes are visited and their values are corrected and stored.
3. **Searching for a Target Value**:
   - The `find` function is straightforward. It simply checks if the target value is present in the `self.vis` set. If the target exists, it returns `true`; otherwise, it returns `false`.

The choice of using DFS for recovery ensures that all nodes are visited and corrected according to their intended values. The recovery process is one-time during the object's initialization, which makes subsequent search operations very efficient due to the constant lookup time in the set.

Example Walkthrough

Let's illustrate the solution approach with a simple example. Suppose we have a contaminated binary tree with all nodes having a value of `-1`, and the structure of the tree is as follows:

```
    -1
   /  \
 -1   -1
 /
-1
```

Upon initializing our `FindElements` class with this tree, we want to recover it. The recovery process will proceed as follows:

1. **Initial Recovery**:
   - We start with the root node and override its value to `0`.
2. **First Level of Recovery**:
   - We move to the left child of the root (initially `-1`). According to our formula, the left child's value should be `2 * 0 + 1 = 1`. So, our tree now looks like this:

```
     0
   /  \
  1   -1
 /
-1
```

- Next, we move to the right child of the root (initially `-1`). The right child's value should be `2 * 0 + 2 = 2`. Our tree becomes:

```
     0
   /  \
  1    2
 /
-1
```

3. **Second Level of Recovery**:
   - There's one more node left, which is the left child of node `1`. According to our formula, this node's value should be `2 * 1 + 1 = 3`. So, we recover this node, and the tree is fully recovered:

```
     0
   /  \
  1    2
 /
3
```

- During the recovery, we add all the values `0`, `1`, `2`, and `3` to the set `self.vis`.

Now the tree is recovered, and our set `self.vis` contains `{0, 1, 2, 3}`.

For the **search phase**, let's say we want to find the value `3`:

- We call the `find` function with `3` as the parameter.
- The `find` function checks whether `3` is in `self.vis`.
- Since `3` is present in the set, the function returns `true`.

Alternatively, if we were searching for a non-existent value like `5`, the `find` function would check for `5` in the set, not find it, and return `false`.

The above walkthrough provides a concrete example of how the described algorithm recovers a contaminated tree and facilitates fast searches for values post-recovery.

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 FindElements:
    def __init__(self, root: Optional[TreeNode]):
        # Helper function to recover the tree
        def recover_tree(node):
            # Add the current node's value to the visited set
            self.visited.add(node.val)
            # If the left child exists, set its value and recover the left subtree
            if node.left:
                node.left.val = node.val * 2 + 1
                recover_tree(node.left)
            # If the right child exists, set its value and recover the right subtree
            if node.right:
                node.right.val = node.val * 2 + 2
                recover_tree(node.right)

        # Initialize the root value to 0
        root.val = 0
        # Create a set to keep track of all values in the recovered tree
        self.visited = set()
        # Start the tree recovery process
        recover_tree(root)

    def find(self, target: int) -> bool:
        # Check if the target is in the visited set
        return target in self.visited


# Your FindElements object will be instantiated and called as such:
# obj = FindElements(root)
# param_1 = obj.find(target)

```

Java Solution

```java
import java.util.HashSet;
import java.util.Set;

// Reconstructed tree where every node's value equals twice the value of their
// parent (left child) or twice the value plus one (right child), as if it were
// a binary heap, starting with 0 as the root's value.
class FindElements {
    // Store the values of all the nodes after "recovering" the tree
    private Set<Integer> recoveredValues = new HashSet<>();

    // Constructor that starts the recovery process of the given tree
    public FindElements(TreeNode root) {
        if (root == null) {
            return;
        }
        root.val = 0; // The recovery process starts by setting the root's value to 0
        recoverTree(root);
    }

    // Helper method to recover the tree
    private void recoverTree(TreeNode node) {
        if (node == null) {
            return;
        }
        recoveredValues.add(node.val); // Add the current node's value to the set
        // Recursively recover the left subtree, if it exists, by setting the left child's value
        if (node.left != null) {
            node.left.val = 2 * node.val + 1;
            recoverTree(node.left);
        }
        // Recursively recover the right subtree, if it exists, by setting the right child's value
        if (node.right != null) {
            node.right.val = 2 * node.val + 2;
            recoverTree(node.right);
        }
    }

    // Check if a target value exists in the recovered tree
    public boolean find(int target) {
        return recoveredValues.contains(target);
    }
}

/**
 * This class can be used as shown below:
 * FindElements findElements = new FindElements(root);
 * boolean isFound = findElements.find(target);
 */

```

C++ Solution

```cpp
#include <functional>
#include <unordered_set>

// 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 FindElements {
public:
    // Constructor: Recovers a tree with values changed by the constructor.
    explicit FindElements(TreeNode* root) {
        // Start by setting the root value to 0, as per the problem statement.
        root->val = 0;
        // Depth-first search (DFS) to recover the tree.
        std::function<void(TreeNode*)> recoverTree = [&](TreeNode* node) {
            // Store the recovered value in the hash set.
            recoveredValues.insert(node->val);

            // If the left child exists, set its value and recover its subtree.
            if (node->left) {
                node->left->val = node->val * 2 + 1;
                recoverTree(node->left);
            }

            // If the right child exists, set its value and recover its subtree.
            if (node->right) {
                node->right->val = node->val * 2 + 2;
                recoverTree(node->right);
            }
        };

        // Start recovering the tree from the root.
        recoverTree(root);
    }

    // Checks if a value exists in the recovered tree.
    bool find(int target) {
        // Return true if the target value is in the hash set, false otherwise.
        return recoveredValues.count(target) > 0;
    }

private:
    std::unordered_set<int> recoveredValues; // Stores the recovered values in the tree.
};

// Usage example:
// TreeNode* root; // Assume root is a pointer to the TreeNode structure that represents the corrupted tree.
// FindElements* findElements = new FindElements(root);
// bool isFound = findElements->find(target); // Replace 'target' with the value you want to find.

```

Typescript Solution

```typescript
// Type 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;
  }
}

// Set to store the recovered values in the tree.
const recoveredValues = new Set<number>();

// Helper function to recover the tree.
function recoverTree(node: TreeNode): void {
  // Store the recovered value in the set.
  recoveredValues.add(node.val);

  // If the left child exists, set its value and recover its subtree.
  if (node.left) {
    node.left.val = node.val * 2 + 1;
    recoverTree(node.left);
  }

  // If the right child exists, set its value and recover its subtree.
  if (node.right) {
    node.right.val = node.val * 2 + 2;
    recoverTree(node.right);
  }
}

// Recovery function to initialize and restore the tree based on root.
function initializeRecovery(root: TreeNode): void {
  // Start by setting the root value to 0, as per the problem statement.
  root.val = 0;
  // Start recovering the tree from the root.
  recoverTree(root);
}

// Function to check if a value exists in the recovered tree.
function find(target: number): boolean {
  // Return true if the target value is in the set, false otherwise.
  return recoveredValues.has(target);
}
```

Here's how you can use the provided functions:

```typescript
let root = new TreeNode(); // Assume 'root' is an instance of TreeNode, representing the corrupted tree.
initializeRecovery(root); // Initialize recovery with 'root', which recovers the tree's values.
let isFound = find(target); // Replace 'target' with a numerical value you wish to find in the recovered tree.

```

Time and Space Complexity

Time Complexity

The `__init__` method of the `FindElements` class has a time complexity of `O(n)`, where `n` is the number of nodes in the tree. This is because it performs a Depth-First Search (DFS) on the tree, visiting each node exactly once to recover the original values assuming the tree was distorted by having every node's value changed to `-1`. During this traversal, each node's value is updated based on its parent's value, and the value is added to the `vis` (visited) set.

The `find` method has a time complexity of `O(1)`, as it is a simple lookup operation in a set to check for the presence of the `target` value.

Space Complexity

The space complexity of the `FindElements` class is `O(n)`, because it stores each node's value in a set `vis`. The size of this set is directly proportional to the number of nodes in the tree.

In summary, the DFS in the constructor (`__init__`) dominates the time complexity, making it `O(n)`, while the space complexity is also `O(n)` due to the storage required for the `vis` set.