99. Recover Binary Search Tree

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

Problem Description

In this problem, we have a binary search tree (BST) that has had the values of exactly two of its nodes swapped by mistake. Our task is to recover this BST, which means we need to find these two nodes and swap their values back to their correct positions. It is important to note that we should achieve this without altering the structure of the tree; only the values of the nodes should be swapped back.

Intuition

To solve this problem, we should first understand what a binary search tree is. A BST is a tree structure where the left child's value of a node is less than the node's value, and the right child's value is greater than the node's value. This property must be true for all nodes in the BST.

Given this property, if two nodes' values are swapped, it violates the tree's ordering rule. Specifically, there will be a pair of consecutive nodes in an in-order traversal of the BST where the first node's value is greater than the second node's value, which should not happen in a correctly ordered BST.

The solution approach uses an in-order traversal, which visits the nodes of the BST in ascending order of their values. During the traversal, we can find the two nodes that are out of order. The `dfs` function recursively traverses the tree in-order and uses three non-local variables `prev`, `first`, and `second`. These help to track the previous node visited and the two nodes that are out of order.

- `prev` keeps track of the previous node in the in-order traversal.
- `first` will be assigned to the first node that appears in the wrong order.
- `second` will be updated to the subsequent node that appears in the wrong order.

As we traverse the tree, whenever we find a node whose value is less than the value of the `prev` node, we know we've encountered an anomaly:

- If it's the first anomaly, we assign `prev` to `first`.
- If it's the second anomaly, we assign the current node to `second`.

After the traversal, `first` and `second` will be the two nodes that need their values swapped. The last line of the solution swaps the values of these two nodes.

While there could be more than one pair of nodes which appear to be out of order due to the swap, it's guaranteed that swapping `first` and `second` will correct the BST because the problem states that exactly two nodes have been swapped. Thus the solution correctly recovers the BST by swapping the values of `first` and `second`.

Solution Approach

The implementation of the solution utilizes a depth-first search algorithm, which is a standard approach to traverse and search all the nodes in a tree. This search pattern is essential for checking each node and its value relative to the BST's ordering properties while maintaining a manageable complexity. Here's a step-by-step explanation of how the implemented code works:

1. **In-Order Traversal**: The core of the solution relies on an in-order traversal. In a BST, an in-order traversal visits nodes in a sorted order. If two nodes are swapped, the order will be disrupted, and we can identify those nodes during this traversal.

2. **Recursive Depth-First Search (DFS)**: We define the `dfs` function, which is a recursive function that performs the in-order traversal of the tree. It visits the left child, processes the current node, and then visits the right child.

3. **Using Non-Local Variables**: We use non-local variables `prev`, `first`, and `second` to keep track of the previous node, the first out-of-order node, and the second out-of-order node, respectively. Non-local variables are needed because they maintain their values between recursive calls.

4. **Identifying Out-of-Order Nodes**: As we perform the in-order traversal, whenever we encounter a node that has a value less than the `prev` node, this indicates an anomaly in the ordering. The `first` anomaly is when we set the `prev` as the `first`, and the next anomaly detected involves setting `second` as the current node with the lesser value.

5. **Swapping Values**: After the in-order traversal, we have isolated the two nodes that were incorrectly swapped. To fix the BST, we simply need to swap their values. This is done in the last line of the `recoverTree` method by the expression `first.val, second.val = second.val, first.val`.

Here is what the code does, broken down by each line:

- `prev = first = second = None`: Initialize variables to `None`. `prev` will track the most recent node during in-order traversal, while `first` and `second` will track the two nodes that need to be swapped.
- `dfs(root)`: Start the DFS in-order traversal from the root of the tree.
- `if root is None: return`: If the current node is `None`, backtrack since it's the base case for the recursion.
- `dfs(root.left)`: Recursively traverse the left subtree, which will visit all nodes smaller than the current node before it processes the current node.
- `if prev and prev.val > root.val`: Check if the current node (`root`) has a value less than that of `prev`, which would indicate a violation of the BST properties.
- `if first is None: first = prev`: If `first` is not set, it means this is the first anomaly found, and we set `first` to `prev`.
- `second = root`: Whether the current anomaly is the first or subsequent, update `second` to the current node (`root`).
- `prev = root`: Update `prev` to the current node (`root`) as it is now the most recent node visited.
- `dfs(root.right)`: Recursively traverse the right subtree, which will visit all nodes greater than the current node.

Using this approach, we ensure the issue is corrected without disrupting the tree's structure, and the time complexity is O(N), where N is the number of nodes in the tree, because each node is visited exactly once during the in-order traversal.

Example Walkthrough

Let's assume we have a binary search tree with the following in-order traversal before any nodes have been swapped:

```
1, 2, 3, 4, 5, 6, 7
```

Now, let's say the values at nodes 3 and 5 have been swapped by mistake. The in-order traversal of the BST will now look like this:

```
1, 2, 5, 4, 3, 6, 7
```

We can see that the series is mostly ascending except for two points where the order is disrupted. Specifically, 5 appears before 4 and 3 appears after 4, which is not correct according to BST properties.

We start the depth-first search with these steps:

1. Initialize `prev`, `first`, and `second` as `None`.

2. Perform in-order traversal starting at the root.

3. Traverse to the left-most node, updating `prev` as we go along.

4. Once we reach the node with value 2, we traverse up and then right, encountering 5. This is where we first notice the order is incorrect since 5 > 2, but `prev` is `None`, so we move on.

5. Next, we traverse to 4. Here we find `prev.val` (which is 5) > `root.val` (which is 4). This is our first anomaly, so we set `first` to `prev` (5) and leave `second` as `None` for now.

6. We continue our traversal. Now `prev` is set to 4. When we get to 3, we see that `prev.val` (4) > `root.val` (3). We've found our second anomaly. We assign `second` to the current node (3).

7. We finish the traversal by visiting the remaining nodes (6 and 7), but no further anomalies are found.

Now, we have our two nodes that were swapped: `first` (with value 5) and `second` (with value 3). The final step is to swap back their values. After swapping, the `first` node will hold the value 3 and the `second` node will hold the value 5.

By following these steps, we have successfully recovered the original BST:

```
1, 2, 3, 4, 5, 6, 7
```

This is how the given solution approach effectively corrects the binary search tree with a time complexity of O(N) by identifying and swapping the two nodes that have been mistakenly swapped.

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 recoverTree(self, root: Optional[TreeNode]) -> None:
        """
        This function corrects a binary search tree (BST) where two nodes have been swapped by mistake.
        It modifies the tree in-place without returning anything.
        """

        # Helper function to perform in-order traversal
        def inorder_traversal(node):
            # Base case: if the current node is None, do nothing
            if node is None:
                return

            # Using 'nonlocal' to modify the outside scope variables
            nonlocal previous, first_swapped, second_swapped

            # Traverse the left subtree
            inorder_traversal(node.left)

            # Check if the previous node has greater value than the current node
            # This indicates that the nodes are swapped
            if previous and previous.val > node.val:
                # If it's the first occurrence, update first_swapped
                if first_swapped is None:
                    first_swapped = previous
                # Either way, update second_swapped to the current node
                second_swapped = node

            # Update previous to the current node before moving to the right subtree
            previous = node

            # Traverse the right subtree
            inorder_traversal(node.right)

        # Initialize the variables
        previous = first_swapped = second_swapped = None

        # Start the in-order traversal
        inorder_traversal(root)

        # Swap the values of the two swapped nodes to recover the BST
        first_swapped.val, second_swapped.val = second_swapped.val, first_swapped.val

```

Java Solution

```java
class Solution {
    // Class member variables to keep track of previous, first and second nodes.
    private TreeNode previousNode;
    private TreeNode firstSwappedNode;
    private TreeNode secondSwappedNode;

    /**
     * Initiates the recovery process of the binary search tree by calling the depth-first search method
     * and then swapping the values of the two nodes that were identified as incorrectly placed.
     *
     * @param root The root of the binary tree that we are trying to recover.
     */
    public void recoverTree(TreeNode root) {
        // Start in-order traversal to find the swapped nodes.
        inOrderTraversal(root);

        // Swap the values of the identified nodes to correct the tree.
        int temp = firstSwappedNode.val;
        firstSwappedNode.val = secondSwappedNode.val;
        secondSwappedNode.val = temp;
    }

    /**
     * Performs an in-order traversal of the binary tree to identify the two nodes that are swapped.
     * It assumes that we are dealing with a binary search tree where an in-order traversal
     * would result in a sorted sequence of values.
     *
     * @param node The current node being visited in the traversal.
     */
    private void inOrderTraversal(TreeNode node) {
        // Base case: If the current node is null, return.
        if (node == null) {
            return;
        }

        // Recursively traverse the left subtree.
        inOrderTraversal(node.left);

        // Process current node: Compare current node's value with previous node's value.
        if (previousNode != null && previousNode.val > node.val) {
            // If this condition is true, a swapped node is found.
            // If it's the first swapped node, assign previousNode to firstSwappedNode.
            if (firstSwappedNode == null) {
                firstSwappedNode = previousNode;
            }
            // Assign current node to secondSwappedNode.
            secondSwappedNode = node;
        }

        // Update previous node to the current node before moving to the right subtree.
        previousNode = node;

        // Recursively traverse the right subtree.
        inOrderTraversal(node.right);
    }
}

/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode() {}

    TreeNode(int val) {
        this.val = val;
    }

    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

```

C++ Solution

```cpp
#include <functional> // Include the functional header for std::function

// 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:
    void recoverTree(TreeNode* root) {
        TreeNode* previous = nullptr; // Pointer to keep track of the previous node during in-order traversal
        TreeNode* first = nullptr;    // Pointer to the first node that is out of the expected order
        TreeNode* second = nullptr;   // Pointer to the second node that is out of the expected order

        // Function to perform in-order traversal and identify the two nodes that are out of order
        std::function<void(TreeNode*)> inorderTraversal = [&](TreeNode* node) {
            if (!node) return; // If the node is null, return from the function

            inorderTraversal(node->left); // Traverse to the left child

            // Check if the previous node's value is greater than the current node's value
            if (previous && previous->val > node->val) {
                // If this is the first occurrence of an out-of-order pair, store the first node
                if (!first) first = previous;
                // In case of second occurrence or adjacent nodes being out of order, store the second node
                second = node;
            }
            previous = node; // Update the previous pointer to the current node

            inorderTraversal(node->right); // Traverse to the right child
        };

        inorderTraversal(root); // Start the in-order traversal from the root

        if (first && second) {
            // Swap the values of the first and second nodes to correct the tree
            std::swap(first->val, second->val);
        }
    }
};

```

Typescript Solution

```typescript
// Typing for a binary tree node
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

/**
 * Function to recover a binary search tree. Two of the nodes of the tree are swapped by mistake.
 * To correct this, we need to swap them back to their original position without changing the tree structure.
 * This function modifies the tree in-place.
 *
 * @param {TreeNode | null} root - The root of the binary tree.
 * @returns {void} Doesn't return anything and modifies the root in-place.
 */
function recoverTree(root: TreeNode | null): void {
    let previous: TreeNode | null = null;
    let firstSwappedNode: TreeNode | null = null;
    let secondSwappedNode: TreeNode | null = null;

    /**
     * Depth-first traversal of the tree to find the two nodes that need to be swapped.
     *
     * @param {TreeNode | null} node - The current node to inspect in the DFS traversal.
     */
    function traverseAndFindSwappedNodes(node: TreeNode | null): void {
        if (!node) {
            return;
        }
        // Traverse the left subtree
        traverseAndFindSwappedNodes(node.left);
        // If the previous node's value is greater than the current node's value, we have found a swapped node
        if (previous && previous.val > node.val) {
            // The first swapped node is the one with a greater value than the one that should have been after it
            if (firstSwappedNode === null) {
                firstSwappedNode = previous; // Found first element that's out of order
            }
            // The second node is the current one, or the one that should have gone before the previous one
            secondSwappedNode = node; // Found next element that's out of order
        }
        // Mark this node as the previous node for comparison in the next iteration
        previous = node;
        // Traverse the right subtree
        traverseAndFindSwappedNodes(node.right);
    }

    // Start the in-order DFS traversal from the root
    traverseAndFindSwappedNodes(root);

    if (firstSwappedNode && secondSwappedNode) {
        // Swap the values of the first and second swapped nodes to correct the tree
        [firstSwappedNode.val, secondSwappedNode.val] = [secondSwappedNode.val, firstSwappedNode.val];
    }
}

```

Time and Space Complexity

The provided code aims to correct a binary search tree (BST) where exactly two nodes have been swapped by mistake. To identify and swap them back, an in-order depth-first search (DFS) is employed.

Time Complexity

The time complexity of the DFS in a BST is `O(N)`, where `N` is the total number of nodes in the tree. This is because the algorithm must visit every node exactly once to compare the values and find the misplaced nodes.

Space Complexity

The space complexity of the algorithm is `O(H)`, where `H` is the height of the tree. This space complexity arises from the maximum size of the call stack when the recursive DFS reaches the deepest level of the tree. For a balanced tree, the height `H` would be `log(N)`, thus giving a best case of `O(log(N))`.

However, in the worst case scenario where the tree becomes skewed, resembling a linked list, the height `H` becomes `N`, thereby degrading the space complexity to `O(H)` which is `O(N)` in the worst case.