99. Recover Binary Search Tree Binary Tree Medium (Tree) Binary Search Tree Depth-First Search Leetcode Link

## Problem Description

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

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

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.

an anomaly:

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

- 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.

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.

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

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.

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

# 3. Using Non-Local Variables: We use non-local variables prev, first, and second to keep track of the previous node, the first out-

the current node.

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

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

violation of the BST properties.

second.val = second.val, first.val.

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.

and 3 appears after 4, which is not correct according to BST properties.

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

second anomaly. We assign second to the current node (3).

incorrect since 5 > 2, but prev is None, so we move on.

**Solution Approach** 

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

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,

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

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

• dfs(root.left): Recursively traverse the left subtree, which will visit all nodes smaller than the current node before it processes

• 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

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).

dfs(root.right): Recursively traverse the right subtree, which will visit all nodes greater than the current node.

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

prev = root: Update prev to the current node (root) as it is now the most recent node visited.

if root is None: return: If the current node is None, backtrack since it's the base case for the recursion.

we simply need to swap their values. This is done in the last line of the recoverTree method by the expression first.val,

N is the number of nodes in the tree, because each node is visited exactly once during the in-order traversal. Example Walkthrough

Using this approach, we ensure the issue is corrected without disrupting the tree's structure, and the time complexity is O(N), where

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:

We can see that the series is mostly ascending except for two points where the order is disrupted. Specifically, 5 appears before 4

1. Initialize prev, first, and second as None. 2. Perform in-order traversal starting at the root.

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

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

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.

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

Python Solution

class Solution:

10

13

14

20

21

22

24

25

26

31

32

33

34

35

36

37

44

45

46

47

48

49

50

# Definition for a binary tree node.

We start the depth-first search with these steps:

def recoverTree(self, root: Optional[TreeNode]) -> None:

It modifies the tree in-place without returning anything.

nonlocal previous, first\_swapped, second\_swapped

# This indicates that the nodes are swapped

if previous and previous.val > node.val:

first\_swapped = previous

if first\_swapped is None:

second\_swapped = node

previous = node

# Start the in-order traversal

inorder\_traversal(root)

private TreeNode previousNode;

private TreeNode firstSwappedNode;

private TreeNode secondSwappedNode;

# Traverse the left subtree

inorder\_traversal(node.left)

# Using 'nonlocal' to modify the outside scope variables

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

swapping the two nodes that have been mistakenly swapped.

This is how the given solution approach effectively corrects the binary search tree with a time complexity of O(N) by identifying and

- # class TreeNode: def \_\_init\_\_(self, val=0, left=None, right=None): self.val = val self.left = left self.right = right
- # Helper function to perform in-order traversal 15 def inorder\_traversal(node): 16 # Base case: if the current node is None, do nothing if node is None: return

# Check if the previous node has greater value than the current node

# If it's the first occurrence, update first\_swapped

# Swap the values of the two swapped nodes to recover the BST

// Class member variables to keep track of previous, first and second nodes.

# Either way, update second\_swapped to the current node

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

first\_swapped.val, second\_swapped.val = second\_swapped.val, first\_swapped.val

This function corrects a binary search tree (BST) where two nodes have been swapped by mistake.

```
38
39
               # Traverse the right subtree
                inorder_traversal(node.right)
40
41
42
           # Initialize the variables
            previous = first_swapped = second_swapped = None
43
```

Java Solution

class Solution {

class TreeNode {

int val;

C++ Solution

struct TreeNode {

int val;

13 class Solution {

public:

TreeNode \*left;

TreeNode \*right;

TreeNode left;

TreeNode() {}

TreeNode right;

TreeNode(int val) {

this.val = val;

this.val = val;

this.left = left;

this.right = right;

// Definition for a binary tree node.

void recoverTree(TreeNode\* root) {

TreeNode\* first = nullptr;

second = node;

TreeNode(int val, TreeNode left, TreeNode right) {

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

TreeNode() : val(0), left(nullptr), right(nullptr) {}

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

if (previous && previous->val > node->val) {

if (!first) first = previous;

std::swap(first->val, second->val);

TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

std::function<void(TreeNode\*)> inorderTraversal = [&](TreeNode\* node) {

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

if (!node) return; // If the node is null, return from the function

previous = node; // Update the previous pointer to the current node

// Swap the values of the first and second nodes to correct the tree

TreeNode\* previous = nullptr; // Pointer to keep track of the previous node during in-order traversal

TreeNode\* second = nullptr; // Pointer to the second node that is out of the expected order

// If this is the first occurrence of an out-of-order pair, store the first node

// In case of second occurrence or adjacent nodes being out of order, store the second node

// Function to perform in-order traversal and identify the two nodes that are out of order

// Check if the previous node's value is greater than the current node's value

// Pointer to the first node that is out of the expected order

62

65

66

67

68

70

71

72

73

74

75

76

78

77 }

10

12

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

41

42

43

44

46

45 };

11 };

```
6
       /**
        * 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.
 9
10
         * @param root The root of the binary tree that we are trying to recover.
       public void recoverTree(TreeNode root) {
13
           // Start in-order traversal to find the swapped nodes.
            inOrderTraversal(root);
15
16
17
           // Swap the values of the identified nodes to correct the tree.
            int temp = firstSwappedNode.val;
18
            firstSwappedNode.val = secondSwappedNode.val;
19
            secondSwappedNode.val = temp;
20
21
22
23
       /**
24
        * Performs an in-order traversal of the binary tree to identify the two nodes that are swapped.
25
        * It assumes that we are dealing with a binary search tree where an in-order traversal
26
        * would result in a sorted sequence of values.
27
28
         * @param node The current node being visited in the traversal.
29
       private void inOrderTraversal(TreeNode node) {
30
           // Base case: If the current node is null, return.
31
           if (node == null) {
33
                return;
34
35
36
           // Recursively traverse the left subtree.
37
            inOrderTraversal(node.left);
38
           // Process current node: Compare current node's value with previous node's value.
39
           if (previousNode != null && previousNode.val > node.val) {
40
               // If this condition is true, a swapped node is found.
41
               // If it's the first swapped node, assign previousNode to firstSwappedNode.
                if (firstSwappedNode == null) {
43
                    firstSwappedNode = previousNode;
44
45
               // Assign current node to secondSwappedNode.
46
                secondSwappedNode = node;
47
48
49
50
           // Update previous node to the current node before moving to the right subtree.
51
           previousNode = node;
           // Recursively traverse the right subtree.
54
            inOrderTraversal(node.right);
55
56 }
57
   /**
    * Definition for a binary tree node.
    */
```

### 35 inorderTraversal(node->right); // Traverse to the right child }; 36 37 38 inorderTraversal(root); // Start the in-order traversal from the root 39 40 if (first && second) {

```
Typescript Solution
   // Typing for a binary tree node
   interface TreeNode {
       val: number;
       left: TreeNode | null;
       right: TreeNode | null;
6 }
   /**
 8
    * 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.
12
    * @param {TreeNode | null} root - The root of the binary tree.
    * @returns {void} Doesn't return anything and modifies the root in-place.
15
    */
   function recoverTree(root: TreeNode | null): void {
        let previous: TreeNode | null = null;
       let firstSwappedNode: TreeNode | null = null;
18
       let secondSwappedNode: TreeNode | null = null;
19
20
21
       /**
22
        * Depth-first traversal of the tree to find the two nodes that need to be swapped.
23
24
        * @param {TreeNode | null} node - The current node to inspect in the DFS traversal.
25
        */
       function traverseAndFindSwappedNodes(node: TreeNode | null): void {
26
27
           if (!node) {
28
               return;
29
30
           // Traverse the left subtree
31
           traverseAndFindSwappedNodes(node.left);
32
           // 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) {
33
34
               // The first swapped node is the one with a greater value than the one that should have been after it
35
               if (firstSwappedNode === null) {
36
                   firstSwappedNode = previous; // Found first element that's out of order
37
               // The second node is the current one, or the one that should have gone before the previous one
38
               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
43
           traverseAndFindSwappedNodes(node.right);
44
45
46
       // Start the in-order DFS traversal from the root
47
       traverseAndFindSwappedNodes(root);
48
49
       if (firstSwappedNode && secondSwappedNode) {
50
           // Swap the values of the first and second swapped nodes to correct the tree
           [firstSwappedNode.val, secondSwappedNode.val] = [secondSwappedNode.val, firstSwappedNode.val];
52
53
54 }
55
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.

visit every node exactly once to compare the values and find the misplaced nodes.

**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

**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.