Recover Binary Search Tree

Problem Statement

You are given the root of a binary search tree (BST), where the values of exactly two nodes of the tree were swapped by mistake. Recover the tree without changing its structure.

Example 1

Input: root = [1, 3, null, null, 2]

Output: root = [3, 1, null, null, 2]

Explanation: 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the BST valid.

Example 2

Input: root = [3, 1, 4, null, null, 2]

Output: root = [2, 1, 4, null, null, 3]

Explanation: 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 makes the BST valid.

Constraints

- The number of nodes in the tree is in the range [2, 1000].
- (-2^{31} leq text{Node.val} leq 2^{31} 1)

Follow-up

A solution using (O(n)) space is pretty straightforward. Could you devise a constant (O(1)) space solution?

Approach

The goal is to recover the BST by identifying and swapping the two incorrect nodes without changing the structure of the tree. This can be achieved through an in-order traversal, which visits nodes in a non-decreasing order for a valid BST.

Steps

1. **In-order Traversal:** Perform an in-order traversal of the tree while maintaining pointers to identify the two swapped nodes.
2. **Anomaly Detection:** During the traversal, check if the current node's value is less than the previous node's value. This indicates an anomaly.
3. **Identify Nodes to Swap:**
   - *First Anomaly:* Mark the previous node as the first element.
   - *Second Anomaly:* Mark the current node as the second element.
4. **Swap the Values:** Swap the values of the two identified nodes to correct the BST.

Explanation

1. **Initialization:** The function initializes three pointers: first_element, second_element, and prev_element. The prev_element is set to negative infinity to ensure proper comparison during the traversal.

2. **In-order Traversal Function:** *This nested function performs the in-order traversal recursively. It:*

- Traverses the left subtree.
- Checks for anomalies by comparing the current node's value with the prev_element's value.
- Updates the prev_element to the current node.
- Traverses the right subtree.

3. **Identifying Swapped Nodes:** *During the traversal, if an anomaly is detected:*

- If it's the first anomaly, both first_element and second_element are marked.
- If a second anomaly is detected, only second_element is updated.

4. **Swapping Values:** After the traversal, if the two swapped nodes are identified, their values are swapped to restore the BST.

This solution ensures the BST is recovered by modifying the node values in-place and operates with (O(1)) additional space, excluding the recursive stack space used during traversal.