98. Validate Binary Search Tree

Medium Tree Depth-First Search Binary Search Tree Binary Tree

Leetcode Link

Problem Description

The problem asks to verify whether a given binary tree is a valid binary search tree (BST). By definition, a valid BST is characterized by the following properties:

- Every node's left subtree contains only nodes with keys that are less than the node's own key.
- Every node's right subtree contains only nodes with keys that are greater than the node's own key.
- Both the left and right subtrees must also themselves be valid BSTs.

The objective is to use these criteria to check every node in the tree and ensure that the structure adheres to the rules of a BST.

Intuition

The solution uses the concept of in-order traversal. In a BST, an in-order traversal yields the nodes' values in ascending order. The approach involves a Depth-First Search (DFS) where we traverse the tree in an in-order manner (left, node, right), keeping track of the value of the previously visited node (`prev`). During the traversal, we ensure that each subsequent node has a greater value than the previous node.

1. A recursive function `dfs` is defined, which will do an in-order traversal of the tree.
2. If we encounter a `None` (indicative of reaching the end of a path), we return `True`, because an empty tree is technically a valid BST.
3. As we traverse the left subtree, we check for violations of the BST properties.
4. Before visiting the current node, we ensure that we've finished validating the left subtree. If the left subtree is invalid, the function returns `False`.
5. We then check the current node's value against `prev` (the previous node's value, initialized to negative infinity). The value must be strictly greater to satisfy BST conditions.
6. Update `prev` to the current node's value.
7. Proceed to validate the right subtree.
8. If every recursive call returns `True`, the entire tree is a valid BST, and thus the function returns `True`.

This approach ensures that we confirm the BST property where each node's value must be greater than all values in its left subtree and less than all values in its right subtree.

Solution Approach

The provided Python code implements the DFS in-order traversal to check the validity of the BST, and it uses the `TreeNode` data structure, which is a standard representation of a node in a binary tree, consisting of a value `val` and pointers to `left` and `right` child nodes.

Here is the step-by-step breakdown of the solution approach:

1. **In-order traversal (DFS):** We recursively traverse the binary tree using in-order traversal, where we first visit the left subtree, then the node, and finally the right subtree.
2. **Global variable:** A global variable `prev` (initialized to negative infinity) is used to keep track of the value of the last visited node in the traversal.
3. **Checking BST properties:**
   - When visiting a node, we first call the `dfs` function on its left child. If this function call returns `False`, it means the left subtree contains a violation of the BST rules, and thus, we return `False` immediately to stop checking further.
   - After checking the left subtree, we examine the current node by comparing its value with `prev`. If the current node's value is less than or equal to `prev`, this means the in-order traversal sequence is broken, and hence, the tree is not a valid BST. We return `False` in this case.
   - The `prev` variable is then updated to the current node's value, indicating that this node has been processed, and we move to the right subtree.
4. **Recursive base case:** For a `None` node, which signifies the end of a branch, the `dfs` function returns `True`, as an empty tree or subtree is valid by definition.
5. **Final validation:** After the entire tree has been traversed, if no violations were found, the initial call to `dfs(root)` will ultimately return `True`, confirming the tree is a valid BST.

The key algorithmic pattern used here is recursion, combined with the properties of in-order traversal for a BST. The recursive function `dfs` combines the logic for traversal and validation, effectively checking the BST properties as it moves through the tree.

Example Walkthrough

Let's consider a small binary tree with the following structure to illustrate the solution approach:

```
  2
 / \
1   3
```

Our target is to walk through the solution approach provided above and confirm whether this tree is a valid BST.

1. **Step 1 - In-order traversal (DFS):** We begin the depth-first search (DFS) with an in-order traversal from the root node which is 2. The in-order traversal dictates that we visit the left subtree first, then the root node, and finally the right subtree.
2. **Step 2 - Global variable:** Initially, the `prev` variable is set to negative infinity. It will help us to keep track of the last visited node's value.
3. **Step 3 - Checking BST properties:**
   - We start with the left child (node 1). Since node 1 has no children, we compare it to `prev`, which is negative infinity. Node 1 is greater, so we update `prev` to 1, and then we return back to node 2.
   - Now, we are at node 2 and compare its value with `prev`, which is now 1. The value of node 2 is greater; therefore, it satisfies the BST condition. We update `prev` to 2 and proceed to the right subtree.
   - In the right subtree, we have node 3. We call the `dfs` function on node 3. As it has no children, we compare it to `prev` (which is now 2). Node 3 is greater than 2, so we update `prev` to 3.
4. **Step 4 - Recursive base case:** Since we reached the leaf nodes without encountering any `None` nodes, we confirm that all subtrees are valid BSTs as well.
5. **Step 5 - Final validation:** After visiting all the nodes following the in-order traversal, and none of the recursive `dfs` calls returned `False`, the whole tree is thus confirmed to be a valid BST. The final call to `dfs(root)` returns `True`.

By following the in-order traversal method and checking each node against the previous node's value, we have verified that the given tree meets all the properties of a valid BST:

- Each node's left subtree contains only nodes with values less than the node's own value.
- Each node's right subtree contains only nodes with values greater than the node's own value.
- All the subtrees are valid BSTs on their own.

In summary, the example binary tree with nodes 1, 2, and 3 is indeed a valid binary search tree.

Python Solution

```python
class TreeNode:
    # A binary tree node has a value, and references to left and right child nodes.
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        # Helper function to perform an in-order traversal of the tree.
        def dfs(node):
            nonlocal last_value_visited  # To keep track of the last value visited in in-order traversal.

            # If the current node is None, it's a leaf, so return True.
            if node is None:
                return True

            # Traverse the left subtree.
            if not in_order_traversal(node.left):
                return False

            # Check if the previous value in the in-order traversal is greater than or equal to the current node value,
            # which would violate the BST property.
            if last_value_visited >= node.val:
                return False

            # Update the last value visited with the current node's value.
            last_value_visited = node.val

            # Traverse the right subtree.
            if not in_order_traversal(node.right):
                return False

            # No violations found, return True.
            return True

        # We initialize the last visited value with negative infinity (smallest possible value)
        # to make sure the first comparison is always True.
        last_value_visited = float('-inf')

        # Start the in-order traversal at the tree with the root node.
        return in_order_traversal(root)
```

Java Solution

```java
/**
 * Definition for a binary tree node.
 * This class represents a node of a binary tree which includes value, pointer to left child
 * and pointer to right child.
 */
class TreeNode {
    int val; // node's value
    TreeNode left; // pointer to left child
    TreeNode right; // pointer to right child

    // Default constructor.
    TreeNode() {}

    // Constructor with node value.
    TreeNode(int val) {
        this.val = val;
    }

    // Constructor with node value, left child, and right child.
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * This class contains method to validate if a binary tree is a binary search tree (BST).
 */
class Solution {
    private Integer previousValue; // variable to store the previously visited node's value

    /**
     * Validates if the given binary tree is a valid binary search tree (BST).
     *
     * @param root The root of the binary tree to check.
     * @return true if the given tree is a BST, false otherwise.
     */
    public boolean isValidBST(TreeNode root) {
        previousValue = null; // Initialize previousValue as null before starting traversal
        return inOrderTraversal(root);
    }

    /**
     * Performs an in-order depth-first traversal on the binary tree to validate BST property.
     * In a BST, for every node, all nodes in its left subtree must be smaller than the values of all nodes in its left subtree
     * and all nodes in the right subtree must be greater than the node's value.
     *
     * @param node The current node being visited in the traversal.
     * @return true if the subtree rooted at 'node' satisfies BST properties, false otherwise.
     */
    private boolean inOrderTraversal(TreeNode node) {
        if (node == null) {
            return true; // Base case: An empty tree is a valid BST.
        }

        // Traverse the left subtree. If it's not a valid BST, return false immediately.
        if (!inOrderTraversal(node.left)) {
            return false;
        }

        // Check the current node value with the previous node's value.
        // As it's an in-order traversal, previousValue should be less than the current node's value.
        if (previousValue != null && previousValue >= node.val) {
            return false; // The BST property is violated.
        }

        // Update previousValue with the current node's value.
        previousValue = node.val; // Update previousValue with the current node's value.
        // Traverse the right subtree. If it's not a valid BST, return false immediately.
        if (!inOrderTraversal(node.right)) {
            return false;
        }

        return true; // All checks passed, it's a valid BST.
    }
}
```

C++ Solution

```cpp
#include <climits>

/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor to initialize a node with a given value,
    // left and right pointers set to nullptr by default
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor to create a node with given value, left, and right children
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
private:
    // Pointer to store the last visited node in the binary search tree
    // This is used to compare with the current node's value.
    TreeNode *lastVisited;

    // Helper function to perform an in-order traversal of the tree
    // and check if it is a valid binary search tree.
    bool inorderTraversal(TreeNode *node) {
        // Base case: If the current node is null, then it's valid
        if (!node) return true;

        // Recursively traverse the left subtree.
        // If the left subtree is not a valid BST, the entire tree is not.
        if (!inorderTraversal(node->left)) return false;

        // If the last visited node is not null and the value of the last visited node
        // is greater than or equal to the current node's value, then it's not a valid BST.
        if (lastVisited && lastVisited->val >= node->val) return false;

        // Update the last visited node to the current node
        lastVisited = node;

        // Recursively traverse the right subtree.
        // If the right subtree is not a valid BST, the entire tree is not.
        if (!inorderTraversal(node->right)) return false;

        // If both subtrees are valid, return true
        return true;
    }

public:
    // Function to check whether a binary tree is a valid binary search tree.
    bool isValidBST(TreeNode* root) {
        // Initialize the last visited node as null before starting the traversal
        lastVisited = nullptr;

        // Call the helper function to check if the tree is a valid BST
        return inorderTraversal(root);
    }
};
```

Typescript Solution

```typescript
// Define the structure of a TreeNode with TypeScript interface
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Validates if the provided tree is a binary search tree.
 * @param {TreeNode | null} root - The root node of the binary tree to validate.
 * @return {boolean} - True if the tree is a valid BST; otherwise, false.
 */
const isValidBST = (root: TreeNode | null): boolean => {
  // Define the previous node to keep track of during in-order traversal
  let previous: TreeNode | null = null;

  /**
   * Performs in-order traversal to check each node's value strictly increasing.
   * @param {TreeNode | null} node - The current node in the traversal.
   * @return {boolean} - True if the subtree is valid; otherwise, false.
   */
  const inorderTraversal = (node: TreeNode | null): boolean => {
    if (node === null) {
      return true;
    }

    // Traverse the left subtree
    if (!inorderTraversal(node.left)) {
      return false;
    }

    // Check if the current node's value is greater than the previous node's value
    if (previous && node.val <= previous.val) {
      return false;
    }

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

    // Traverse the right subtree
    return inorderTraversal(node.right);
  };

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

// Sample usage (in TypeScript you would normally type the input, here it is implicit for brevity)
// let tree: TreeNode = {
//   val: 2,
//   left: { val: 1, left: null, right: null },
//   right: { val: 3, left: null, right: null }
// };
// console.log(isValidBST(tree)); // Outputs: true
```

Time and Space Complexity

The given Python code defines a method `isValidBST` to determine if a binary tree is a valid binary search tree. It uses depth-first search (DFS) to traverse the tree.

Time Complexity:

The time complexity of this algorithm is $O(n)$, where n is the number of nodes in the binary tree. This is because the DFS visits each node exactly once to compare its value with the previously visited node's value.

Space Complexity:

The space complexity of this code is $O(h)$, where h is the height of the tree. This is determined by the maximum number of function calls on the call stack at any one time, which occurs when the algorithm is traversing down one side of the tree. In the worst case of a completely unbalanced tree (like a linked list), the space complexity would be $O(n)$. For a balanced tree, it would be $O(\log n)$.