701. Insert into a Binary Search Tree

Medium Tree Binary Search Tree Binary Tree Leetcode Link

Problem Description

In this problem, we are dealing with the operation of inserting a new node into a binary search tree (BST). A BST is a special kind of binary tree where the value of each node must be greater than all values stored in its left subtree and less than the values stored in its right subtree.

You are provided with two pieces of information:

1. The `root` node of the BST, which is the starting point from which we can traverse the entire tree.
2. A `value` that you need to insert into the BST.

The task is to insert the `value` into the BST by finding the correct position without violating the BST properties. After the insertion, the function must return the `root` node of the modified BST.

It is important to note that since the `value` does not already exist in the BST, there won't be any duplicate values after insertion. Additionally, there can be several valid insertion points that could maintain the BST properties. The function can return any valid final BST.

Intuition

The solution approach uses Depth-First Search (DFS), which is an algorithm for traversing or searching tree or graph data structures. The DFS approach is recursive, as it applies the same logic at each node starting from the `root`.

Here's how we arrive at the solution approach:

1. Begin at the `root` and compare the `value` with the `root.val`.
2. If the `value` is greater than `root.val`, the value must be inserted somewhere in the right subtree of the current node because of the BST property (all right subtree values should be greater). Thus, we perform DFS on the right child of the current node.
3. If the `value` is less than `root.val`, the value must be inserted in the left subtree. We then perform DFS on the left child of the current node.
4. If at any point, we find that the current node (`root`) is `None`, it means we have reached an external point in the BST where the new node can be attached. So, we create a new node with the `value` and return it.
5. As the recursion unwinds, each call updates its left or right child pointer to the newly created node if a new node was created below it.
6. When the top of the recursion stack is reached (the original `root`), the entire BST with the new value inserted is returned.

The recursive nature of this approach handles all possible cases for insertion and ensures the maintenance of the BST properties post-insertion.

Solution Approach

The given problem is addressed by implementing a recursive function to perform a depth-first search (DFS) on the tree. Here is a step-by-step walk-through of `dfs` function implementation, considering the Reference Solution Approach provided:

1. The `dfs` function takes the current `root` node as its argument.
2. If the current `root` node is `None`, it implies that the value should be inserted at this point. Therefore, a new `TreeNode` with the insert `val` is returned.
3. If the current `root` node is not `None`, we compare the insert `val` with the current node's value (`root.val`) to decide the direction of traversal:
   - If `val` is greater than `root.val`, we recursively call `dfs` on the right child of the current node and update `root.right` with the result of that call.
   - If `val` is less than `root.val`, we recursively call `dfs` on the left child of the current node and update `root.left` with the result of that call.
4. Each recursive call to `dfs` follows the same logic until a position to insert the new value is found (when `root` is `None`).
5. Once the base case is reached and the new node is created, the recursion begins to unwind. Every recursive call returns the current state of the `root` node, either unchanged (if no insertion happened in the respective subtree) or updated (if the new node was appended to the subtree).
6. Eventually, the function unwinds to the first recursive call, which returns the modified `root` of the entire tree with the new value inserted while maintaining the BST properties.

Algorithms and data structures utilized in this approach:

- **Recursion**: A function that calls itself, simplifying complex problems by breaking them down into self-similar sub-problems.
- **Binary Search Tree (BST)**: A binary tree with the property that for any given node, values in its left subtree are smaller, and values in its right subtree are larger.
- **DFS**: A traversal algorithm that starts at the root node and explores as far as possible along each branch before backtracking.

The recursive implementation efficiently uses the natural structure of a BST to locate the proper insertion point, ensuring a valid BST at every step of the recursion.

Example Walkthrough

Let's walk through an example to illustrate the solution approach. Suppose we have the following binary search tree and we want to insert the `value` **4** into this tree:

```
    3
   / \
  1   5
```

And, here are the steps we would take following the solution approach:

1. Since the `root` value (3) is less than our `value` (4), we look to the right subtree of the `root`. The `root` has a right child with the value 5.

2. Now considering node with value 5 as the new `root`, we compare our `value` (4) to 5. Since 4 is less than 5, we need to explore the left subtree of this node.

3. The left child of node 5 is `None`, which means this is the position where we should insert our new `value` (4). So, we create a new node with `value` 4.

4. We then update the left child of node 5 to point to this new node.

5. As the recursion unwinds, we return to the original root, but now its right subtree includes our new value, resulting in the following BST:

```
    3
   / \
  1   5
     /
    4
```

Through this recursive DFS approach, we have inserted the `value` 4 into the BST while maintaining the BST properties. The resulting tree is a valid binary search tree with the new node correctly placed.

Python Solution

```python
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class Solution:
    def insertIntoBST(self, root: TreeNode, value: int) -> TreeNode:
        # Helper function to traverse the tree and insert the node
        def insert_helper(node):
            # If we've reached a null node, insert the new value here
            if node is None:
                return TreeNode(value)

            # Decide to proceed left or right depending on the value
            if node.value < value:
                # Value is greater, go to the right subtree
                node.right = insert_helper(node.right)
            else:
                # Value is smaller or equal, go to the left subtree
                node.left = insert_helper(node.left)

            # Return the node with its updated children
            return node

        # Call helper function to start insertion from root
        return insert_helper(root)

```

Java Solution

```java
// Class for the binary tree node structure
class TreeNode {
    int value;
    TreeNode left;
    TreeNode right;

    // Constructor to create a node without children
    TreeNode(int value) {
        this.value = value;
    }

    // Constructor to create a node with specified left and right children
    TreeNode(int value, TreeNode left, TreeNode right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
}

class Solution {

    // Inserts a value into a Binary Search Tree (BST) and returns the updated tree root.
    public TreeNode insertIntoBST(TreeNode root, int value) {
        // If the current node is null, we've found the position for the new value.
        if (root == null) {
            return new TreeNode(value);
        }

        // If the value is greater than the current node's value, insert into the right subtree.
        if (root.value < value) {
            root.right = insertIntoBST(root.right, value);
        }
        // If the value is less than or equal to the current node's value, insert into the left subtree.
        else {
            root.left = insertIntoBST(root.left, value);
        }

        // Return the node itself after performing the insertion to maintain tree connections.
        return root;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int value;
    TreeNode *leftChild;
    TreeNode *rightChild;

    // Constructor with no arguments initializes node with value 0 and null children.
    TreeNode() : value(0), leftChild(nullptr), rightChild(nullptr) {}

    // Constructor with value argument sets the node value and initializes children to null.
    TreeNode(int x) : value(x), leftChild(nullptr), rightChild(nullptr) {}

    // Constructor with value and children arguments initializes node with given values.
    TreeNode(int x, TreeNode *left, TreeNode *right) : value(x), leftChild(left), rightChild(right) {}
};

class Solution {
public:
    /**
     * Inserts a value into a binary search tree (BST) and returns the root node of the BST.
     *
     * @param root Pointer to the root node of the tree where value is to be inserted.
     * @param val The value to be inserted into the BST.
     * @return Returns the root of the BST after insertion of the value.
     */
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        // If root is null, the new value should be placed here, so create and return a new node.
        if (!root) {
            return new TreeNode(val);
        }

        // If the value to insert is greater than the root's value, insert into the right subtree.
        if (root->value < val) {
            root->rightChild = insertIntoBST(root->rightChild, val);
        }
        // Otherwise, the value is less than or equal to the root's value, insert into the left subtree.
        else {
            root->leftChild = insertIntoBST(root->leftChild, val);
        }

        // Return the unchanged root pointer.
        return root;
    }
};

```

Typescript Solution

```typescript
// Node structure for the binary search tree.
interface TreeNode {
  value: number;
  leftChild: TreeNode | null;
  rightChild: TreeNode | null;
}

// Function to create a new tree node with a value, and with left and right children initialized to null.
function createTreeNode(value: number, leftChild: TreeNode | null = null, rightChild: TreeNode | null = null): TreeNode {
  return {
    value: value,
    leftChild: leftChild,
    rightChild: rightChild
  };
}

/**
 * Inserts a value into a binary search tree (BST) and returns the root node of the BST.
 *
 * @param root - The root node of the tree where the value is to be inserted.
 * @param value - The value to be inserted into the BST.
 * @return The root of the BST after the insertion of the value.
 */
function insertIntoBST(root: TreeNode | null, value: number): TreeNode | null {
  // If the root is null, create a new node with the value to be inserted and return it.
  if (root === null) {
    return createTreeNode(value);
  }

  // If the value to insert is greater than the root's value, recursively insert into the right subtree.
  if (value > root.value) {
    root.rightChild = insertIntoBST(root.rightChild, value);
  } else { // If the value is less than or equal to the root's value, recursively insert into the left subtree.
    root.leftChild = insertIntoBST(root.leftChild, value);
  }

  // Return the unchanged root node.
  return root;
}

```

Time and Space Complexity

Time Complexity

The time complexity of this code is `O(H)`, where `H` is the height of the binary search tree (BST). This is because in the worst-case scenario, you may need to traverse from the root to the leaf node to find the correct spot for insertion, which takes a time proportional to the height of the tree.

Space Complexity

The space complexity is also `O(H)` because the code uses recursive calls to insert the new node. The maximum depth of the recursive call stack would also be the height of the tree in the worst-case scenario, which makes the space complexity proportional to the height of the tree. For a balanced BST, this would be `O(log N)`, where `N` is the number of nodes in the BST. However, for a skewed BST (resembling a linked list), this would be `O(N)`.