897. Increasing Order Search Tree

Easy Stack Tree Depth-First Search Binary Search Tree Binary Tree Leetcode Link

Problem Description

The problem statement presents a binary search tree (BST) and asks for a transformation of this BST into a new tree that is essentially a linked list in ascending order. In other words, the task is to rearrange the tree such that it maintains the in-order sequence of values, where every node only has a right child, effectively eliminating all the left children. The leftmost node of the original BST should become the new root of the so-called 'flattened' tree.

Intuition

The solution leverages the property of BSTs where an in-order traversal yields the nodes in ascending order. The concept here is to perform an in-order Depth-First Search (DFS) traversal, and as we visit each node, we rearrange the pointers to create the 'right-skewed' tree desired for the problem.

The intuition behind the approach can be broken down into steps:

1. Initialize a 'dummy' node as a precursor to the new root to make attachment easier.
2. Start the in-order DFS from the root of the original BST.
3. As the in-order DFS visits the nodes, adjust pointers such that each visited node becomes the right child of the previously visited node ('prev').
   - The current node's left child is set to `None`.
   - The current node's right child becomes the entry point for the next nodes that are yet to be visited in the DFS.
4. The 'prev' node, initialized as the dummy node, is updated in each step to the current node.
5. After the traversal, the right child of the dummy node (which was our starting point) now points to the new root of the in-order rearranged tree.
6. The leftmost node from the original tree is now acting as the root of this reordered straight line tree.

This approach ensures that every node, except for the first (new root), will only have a single right child, fulfilling the condition of having no left child and at most one right child.

Solution Approach

The solution implements an in-order traversal to process nodes of a binary search tree (BST) in the ascending order of their values. An in-order traversal visits nodes in the "left-root-right" sequence, which aligns with the BST property that left child nodes have smaller values than their parents, and right child nodes have larger values.

Here's a breakdown of the key parts of the solution:

- An inner function `dfs` (short for Depth-First Search) is defined, which recursively traverses the BST in an in-order fashion. It helps us visit each node in the required sequence to rearrange the tree.
- A 'dummy' node is initiated before the traversal begins. This node serves as a placeholder to easily point to the new tree's root after the rearrangement is complete.
- The `prev` variable points to the last processed node in the in-order sequence. Initially, it's set to the 'dummy' node to start the rearrangement process.
- During each call to `dfs`:
  1. The left subtree is processed, ensuring that all smaller values have already been visited before we rearrange the current node.
  2. The current node (`root`) is detached from its left child by setting it to `None`.
  3. The `right` attribute of the `prev` node is set to the current node, effectively 'flattening' the tree by making the current node the next element in the list.
  4. The `prev` variable is updated to the current node, preparing it to link to the next node in the traversal.
  5. The right subtree is processed, continuing the in-order traversal.
- Due to the `nonlocal` keyword usage, the `prev` variable retains its value across different `dfs` invocations.
- After completing the DFS, the `right` child of the 'dummy' node is returned. This now points to the leftmost node of the original BST, which is the new root of the restructured 'tree' (now resembling a linked list).
- The end result is that all nodes are rearranged into a straight line, each having no left child, and only one right child, reflecting an in-order traversal of the original BST.

In summary, the solution uses a recursive in-order DFS traversal, pointer manipulation, and a 'dummy' placeholder node to create a modified tree that satisfies the specific conditions. It efficiently flattens the BST into a single right-skewed path that represents the ascending order of the original tree's values.

Example Walkthrough

Let's illustrate the solution approach with an example. Consider the following BST:

```
    4
   / \
  2   6
 / \ / \
1  3 5  7
```

According to the in-order traversal, the nodes are visited in the order: 1, 2, 3, 4, 5, 6, 7. We will flatten this BST to a right-skewed tree.

Here's step by step how the algorithm will work on this BST:

1. Initialize a `dummy` node as a precursor to the new root to make the attachment of nodes easier.
   - Let's say `dummy.val` is 0 for illustration purposes.
2. Start in-order DFS with the `dummy` node's `prev` pointing to it.
3. Visit the leftmost node, which is 1. At this point:
   - Set `prev.right = node 1`, disconnect `node 1` from its left (none in this case).
   - Update `prev` to be `node 1`.
4. Visit node 2, the parent of node 1:
   - Since node 1 right is null, set `node 1.right = node 2` and disconnect `node 2` from its left (which was node 1).
   - Update `prev` to be `node 2`.
5. Proceed to node 3 by visiting the right subtree of node 2:
   - Set `node 2.right = node 3` and disconnect `node 3` from its left (none).
   - Update `prev` to be `node 3`.
6. Continue with node 4 (root of the original BST) similarly:
   - Set `node 3.right = node 4` and disconnect `node 4` from its left (which was node 2).
   - Update `prev` to be `node 4`.
7. Move to node 5 through the left part of the right subtree of node 4:
   - Set `node 4.right = node 5` and disconnect `node 5` from its left (none).
   - Update `prev` to be `node 5`.
8. Node 6 is processed next:
   - Set `node 5.right = node 6` and disconnect `node 6` from its left (which was node 5).
   - Update `prev` to be `node 6`.
9. Finally, visit node 7:
   - Set `node 6.right = node 7` and disconnect `node 7` from its left (none).
   - Update `prev` to be `node 7`.
10. At the end of this DFS process, the BST has been turned into a right-skewed list:

```
0 - 1 - 2 - 3 - 4 - 5 - 6 - 7
```

11. The `dummy` node's right child (0's right child) is node 1, so this is the new root of the flattened tree:

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

We've now flattened the original BST to a linked list in ascending order following the in-order sequence of the BST. Each node has no left child and only one right child.

Python Solution

```python
class TreeNode:
    # TreeNode class definition remains the same
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def increasingBST(self, root: TreeNode) -> TreeNode:
        # Helper function to perform in-order traversal
        def in_order_traversal(node):
            # Base case: if node is None, return
            if not node:
                return

            # Traverse the left subtree
            in_order_traversal(node.left)

            # Visit the current node:
            # Since we are rearranging the tree into a right-skewed tree,
            # we make the previously visited node's right point to the current node
            self.prev.right = node

            # Disconnect the current node's left child
            node.left = None

            # Update the previous node to be the current node
            self.prev = node

            # Traverse the right subtree
            in_order_traversal(node.right)

        # Dummy node to start the right-skewed tree
        dummy = TreeNode()
        # 'prev' initially points to the dummy node
        self.prev = dummy

        # Perform in-order traversal; this will rearrange the nodes
        in_order_traversal(root)

        # The right child of the dummy node is the root of the modified tree
        return dummy.right

```

Java Solution

```java
class Solution {
    // 'prev' will be used to keep track of the previous node in inorder traversal.
    private TreeNode previousNode;

    public TreeNode increasingBST(TreeNode root) {
        // 'dummyNode' will act as a placeholder to the beginning of the resulting increasing BST.
        TreeNode dummyNode = new TreeNode(0);
        previousNode = dummyNode;

        // Perform the 'inorder' traversal starting from the root.
        inorderTraversal(root);

        // Return the right child of the dummy node, which is the real root of the increasing BST.
        return dummyNode.right;
    }

    // The 'inorderTraversal' function recursively traverses the tree in an inorder fashion.
    private void inorderTraversal(TreeNode node) {
        // If the current node is 'null', we have reached the end and should return.
        if (node == null) {
            return;
        }

        // Recurse on the left subtree.
        inorderTraversal(node.left);

        // During the inorder traversal, we reassign the rights of the 'previousNode' to the current 'node'
        // and nullify the left child to adhere to increasing BST rules.
        previousNode.right = node;
        node.left = null;

        // Update 'previousNode' to the current node.
        previousNode = node;

        // Recurse on the right subtree.
        inorderTraversal(node.right);
    }
}

```

C++ Solution

```cpp
// Declaration for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor for a new empty TreeNode.
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    // Constructor for a TreeNode with a specific value.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor for a TreeNode with a value and specified left and right children.
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    TreeNode* previousNode; // Node to keep track of the previously processed node.

    // Takes a binary search tree and rearranges it so that it becomes a strictly increasing order tree
    // where each node only has a right child following an in-order traversal of the tree.
    TreeNode* increasingBST(TreeNode* root) {
        TreeNode* dummyNode = new TreeNode(0); // Create a dummy node that acts as the previous node of the first node in the in-or
        previousNode = dummyNode; // Initialize previousNode with the dummy node.

        inOrderTraversal(root); // Start the in-order traversal and rearrange the tree.

        // Return the right child of the dummy node, which is the new root of the rearranged tree.
        return dummyNode->right;
    }

    // Helper function that performs DFS in-order traversal of the binary tree.
    void inOrderTraversal(TreeNode* currentNode) {
        if (!currentNode) return; // If the current node is null, return.

        // Traverse the left subtree first (in-order).
        inOrderTraversal(currentNode->left);

        previousNode->right = currentNode; // Assign the current node to the right child of the previous node.
        currentNode->left = nullptr; // The current node should not have a left child in the rearranged tree.
        previousNode = currentNode; // Update the previousNode to be the current node after processing it.

        // Traverse the right subtree next (in-order).
        inOrderTraversal(currentNode->right);
    }
};

```

Typescript Solution

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

// Function to create a new TreeNode with a specific value.
function createTreeNode(val: number, left: TreeNode | null = null, right: TreeNode | null = null): TreeNode {
    return {
        val: val,
        left: left,
        right: right
    };
}

let previousNode: TreeNode | null; // Variable to keep track of the previously processed node.

// Function that takes a binary search tree and rearranges it so that it becomes a strictly
// increasing order tree where each node only has a right child following an in-order traversal
// of the tree.
function increasingBST(root: TreeNode): TreeNode {
    const dummyNode: TreeNode = createTreeNode(0); // Create a dummy node that will act as the "previous node" of the first node in t
    previousNode = dummyNode; // Initialize previousNode with the dummy node.

    // Start the in-order traversal and rearrange the tree.
    inOrderTraversal(root);

    // Return the right child of the dummy node, which is the new root of the rearranged tree.
    return dummyNode.right!;
}

// Helper function that performs a depth-first search (DFS) in-order traversal of the binary tree.
function inOrderTraversal(currentNode: TreeNode | null): void {
    if (currentNode === null) return; // If the current node is null, do nothing and return.

    // First, traverse the left subtree (in-order).
    inOrderTraversal(currentNode.left);

    // Assign the current node to the right child of the previous node.
    if (previousNode) previousNode.right = currentNode;

    // The current node should not have a left child in the rearranged tree.
    currentNode.left = null;

    // Update the previousNode to be the current node after processing it.
    previousNode = currentNode;

    // Next, traverse the right subtree (in-order).
    inOrderTraversal(currentNode.right);
}

```

Time and Space Complexity

The time complexity of the code is `O(n)` where `n` is the number of nodes in the binary tree. This is because the solution involves performing a depth-first search (dfs) traversal of the tree, which visits each node exactly once.

The space complexity of the code is also `O(n)` in the worst-case scenario, which occurs when the binary tree is completely unbalanced (e.g., every node only has a left child, forming a linear chain). In this case, the recursion stack depth would be `n`. For balanced trees, the space complexity would be `O(h)` where `h` is the height of the tree, since the depth of recursion would only be as deep as the height of the tree.