1382. Balance a Binary Search Tree

Problem Description

The problem presents us with a binary search tree (BST) and asks us to transform it into a balanced binary search tree (BBST). A BST is considered balanced if, for every node in the tree, the depth difference between its left and right subtrees is no more than 1. The solution must maintain the original BST's node values and is not required to be unique—any correctly balanced BST with the original values is considered a valid answer.

Intuition

To balance a BST, we want to ensure that we minimize the depth differences between the left and right subtrees of all the nodes. A well-known approach to creating a balanced BST is to first convert the BST into a sorted array and then to rebuild the BST from this array.

The intuition behind this is that a sorted array will have the smaller values towards the beginning and larger values towards the end. By consistently selecting the middle element of the array (or subarray) to be the root node of the (sub)tree, we can ensure that the number of nodes on the left and right of any node is as equal as possible, thus creating a balanced tree.

The provided solution implements these two key phases with the help of two helper functions: `dfs` for in-order traversal and `build` for constructing the balanced BST from the sorted values array.

Solution Approach

The implementation of the solution is done in two parts as per the problem requirement: first, the BST is converted to a sorted list, and then the list is used to construct a balanced BST.

Conversion of BST to Sorted List

The conversion of the BST to a sorted list is done using an in-order traversal, which is a depth-first search (DFS). Before the traversal, an empty list `vals` is initialized. The `dfs` function is defined to perform the traversal:

1. If the current node (`root`) is `None`, the function returns, as it means we've reached a leaf node's child.
2. It recursively calls itself on the left child (`dfs(root.left)`), processing all left descendants first (which are smaller in a BST).
3. The value of the current node is appended to the list `vals` (`vals.append(root.val)`).
4. It recursively calls itself on the right child (`dfs(root.right)`), processing all right descendants afterward (which are larger in a BST).

After defining the `dfs` function, it is called initially with the root of the BST. This will result in `vals` containing all the node values in sorted order since BSTs inherently maintain an in-order sequence of ascending values.

Building the Balanced BST

The balanced BST is constructed using the `build` function which follows a divide and conquer strategy:

1. The function takes two indices, `i` and `j`, which represent the start and end of the current subarray within `vals` we're using to build the subtrees.
2. The base case checks if `i` is greater than `j`. If so, it returns `None`, as this means there are no more elements to create nodes from in this subarray.
3. The middle index `mid` is calculated using `(i + j) >> 1`, which is a bitwise operation equivalent to dividing the sum by 2 but is faster to compute.
4. A new TreeNode is created using `vals[mid]` as the root value.
5. The left subtree of the root is built by recursively calling `build(i, mid - 1)`, effectively using the left half of the current subarray.
6. The right subtree of the root is built by recursively calling `build(mid + 1, j)`, using the right half of the current subarray.

The `build` function initializes the process by being called with the start `i` and end `len(vals) - 1` indices, essentially representing the entirety of the sorted list.

Finally, the `balanceBST` function returns the root node of the newly constructed balanced BST, which results from calling `build(0, len(vals) - 1)`.

Throughout the solution, we make use of recursion and divide-and-conquer techniques to arrive at a balanced BST that meets the depth difference requirement for all nodes.

Example Walkthrough

Let's consider a binary search tree (BST) that is skewed to the right as our example:

```
1
 \
  2
   \
    3
     \
      4
```

This BST is not balanced because the right subtree of each node has a height that is more than 1 greater than the left subtree (which is non-existent). Our objective is to balance this BST according to the solution approach described above.

Conversion of BST to Sorted List

Following the in-order traversal, we obtain the values from the tree in sorted order. We start with an empty list `vals`.

1. Begin at the root (`1`). Left child is `None`, so move to step 3.
2. Append `1` to `vals` (thus `vals = [1]`).
3. Move to the right child, repeat steps for node `2`, and so on.

After traversing all the nodes, the `vals` list becomes:

```
vals = [1, 2, 3, 4]
```

This list represents the in-order traversal of the given BST and is sorted.

Building the Balanced BST

Now, we use the sorted list `vals` to construct a balanced BST.

1. Call `build(0, 3)` since `vals` has four elements, indices ranging from `0` to `3`.
2. Calculate `mid = (0 + 3) >> 1`, resulting in `mid = 1`.
3. Create a new TreeNode with `vals[1]` (which is `2`), making it the root.
4. For the left subtree, we call `build(0, 0)`:
   - Calculate `mid = (0 + 0) >> 1`, resulting in `mid = 0`.
   - Create a new TreeNode with `vals[0]` (which is `1`).
   - No more elements to the left or right, so the left subtree is just the node with `1`.
5. For the right subtree, we call `build(2, 3)`:
   - Calculate `mid = (2 + 3) >> 1`, resulting in `mid = 2`.
   - Create a new TreeNode with `vals[2]` (which is `3`).
   - For the left subtree of `3`, there's no element, so return `None`.
   - For the right subtree of `3`, call `build(3, 3)`:
     - Only one element left, so create a new TreeNode with `vals[3]` (which is `4`).
     - No more elements to the left or right.

After completing these steps, we have successfully constructed the following balanced BST:

```
    2
   / \
  1   3
       \
        4
```

This resulting tree is balanced as the depth difference between the left and right subtrees for all nodes is no more than 1. It maintains all the original values from the input BST and is a valid solution according to the problem description.

Python Solution

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def balanceBST(self, root: TreeNode) -> TreeNode:
        # Helper function to perform inorder traversal and collect values in a list.
        def inorder_traversal(node):
            if node is None:  # Base case: if node is None, return.
                return
            inorder_traversal(node.left)  # Recurse on left subtree.
            node_values.append(node.val)  # Append the value of the current node.
            inorder_traversal(node.right)  # Recurse on right subtree.

        # Helper function to build a balanced BST given a sorted value list.
        def build_balanced_bst(start, end):
            if start > end:  # If starting index is greater than ending, subtree is empty.
                return None

            mid = (start + end) // 2  # Compute the middle index.
            node = TreeNode(node_values[mid])  # Create a node with the middle value.
            # Recursively build left and right subtrees using the split lists.
            node.left = build_balanced_bst(start, mid - 1)
            node.right = build_balanced_bst(mid + 1, end)
            return node  # Return the newly created node.

        node_values = []  # Initialize an empty list to store the tree node values.
        inorder_traversal(root)  # Fill the node_values list with values from the BST.
        # Build and return a balanced BST using the list of values.
        return build_balanced_bst(0, len(node_values) - 1)
```

Java Solution

```java
class Solution {
    // Class member to hold the tree values in sorted order.
    private List<Integer> treeValues;

    // Method to turn a binary search tree into a balanced binary search tree.
    public TreeNode balanceBST(TreeNode root) {
        treeValues = new ArrayList<>();
        // Populate the class member list with the values in sorted order.
        inOrderTraversal(root);
        // Reconstruct the tree in a balanced manner.
        return buildBalancedTree(0, treeValues.size() - 1);
    }

    // Helper method to perform an in-order traversal of the tree and store the values.
    private void inOrderTraversal(TreeNode node) {
        if (node == null) {
            // Base case: if the node is null, do nothing.
            return;
        }
        // Traverse the left subtree first.
        inOrderTraversal(node.left);
        // Store the value of the current node.
        treeValues.add(node.val);
        // Traverse the right subtree next.
        inOrderTraversal(node.right);
    }

    // Helper method to build a balanced binary search tree using the stored values.
    private TreeNode buildBalancedTree(int start, int end) {
        // Base case: if start index is greater than end index, subtree is null.
        if (start > end) {
            return null;
        }
        // Find the mid point to make the current root.
        int mid = start + (end - start) / 2;
        // Create a new TreeNode with the mid value.
        TreeNode root = new TreeNode(treeValues.get(mid));
        // Recursively build the left subtree.
        root.left = buildBalancedTree(start, mid - 1);
        // Recursively build the right subtree.
        root.right = buildBalancedTree(mid + 1, end);
        // Return the root node.
        return root;
    }
}
```

C++ Solution

```cpp
// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    vector<int> nodeValues; // Vector to store the values of nodes in sorted order.

    // Main function to create a balanced BST from the unbalanced BST 'root'.
    TreeNode* balanceBST(TreeNode* root) {
        // Perform in-order traversal to get the values in the sorted order.
        inOrderTraversal(root);
        // Rebuild the tree using the sorted values from 'nodeValues'.
        return rebuildTree(0, nodeValues.size() - 1);
    }

    // Helper function to perform an in-order traversal of a BST.
    void inOrderTraversal(TreeNode* node) {
        if (!node) return; // If the node is null, do nothing.
        inOrderTraversal(node->left); // Traverse the left subtree.
        nodeValues.push_back(node->val); // Visit the node and add its value to 'nodeValues'.
        inOrderTraversal(node->right); // Traverse the right subtree.
    }

    // Helper function to build a balanced BST from the sorted values.
    TreeNode* rebuildTree(int start, int end) {
        if (start > end) return nullptr; // Base case: no nodes to construct the tree.

        // Calculate the middle index and use it as the root to balance the tree.
        int mid = (start + end) / 2;
        TreeNode* newNode = new TreeNode(nodeValues[mid]); // Create a new node with the mid value.

        // Recursively construct the left and right subtrees and link them to the new node.
        newNode->left = rebuildTree(start, mid - 1);
        newNode->right = rebuildTree(mid + 1, end);

        return newNode; // Return the new subtree root.
    }
};
```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// Array to store the values of nodes in sorted order
const nodeValues: number[] = [];

// Main function to create a balanced BST from the unbalanced BST 'root'
function balanceBST(root: TreeNode | null): TreeNode | null {
    if (root === null) return null; // Guard clause for null input

    // Perform in-order traversal to get the values in sorted order
    inOrderTraversal(root);
    // Rebuild the tree using the sorted values from 'nodeValues'
    return rebuildTree(0, nodeValues.length - 1);
}

// Helper function to perform an in-order traversal of a BST
function inOrderTraversal(node: TreeNode | null): void {
    if (node === null) return; // Base case: if the node is null, do nothing
    inOrderTraversal(node.left); // Traverse the left subtree
    nodeValues.push(node.val); // Visit the node and add its value to 'nodeValues'
    inOrderTraversal(node.right); // Traverse the right subtree
}

// Helper function to build a balanced BST from the sorted values
function rebuildTree(start: number, end: number): TreeNode | null {
    if (start > end) return null; // Base case: no nodes to construct the tree

    // Calculate the middle index and use it as the root to balance the tree
    const mid = start + Math.floor((end - start) / 2);
    const newNode = new TreeNode(nodeValues[mid]); // Create a new node with the mid value

    // Recursively construct the left and right subtrees and link them to the new node
    newNode.left = rebuildTree(start, mid - 1);
    newNode.right = rebuildTree(mid + 1, end);

    return newNode; // Return the new subtree root
}
```

Time and Space Complexity

The given code consists of a depth-first search (DFS) traversal (`dfs` function) to collect the values of the nodes in an inorder fashion and a recursive function (`build` function) to construct a balanced binary search tree (BST) from the sorted list of values.

Time Complexity

1. **DFS Traversal (`dfs` function):** The DFS traversal visits each node exactly once. With n being the number of nodes in the tree, the time complexity for the DFS traversal is $O(n)$.
2. **Constructing Balanced BST (`build` function):** The build function is called once for each element in the sorted list of values. Similar to binary search, at each recursive call, it splits the list into two halves until the base case is reached (when `i > j`). Therefore, constructing the balanced BST would also take $O(n)$ time since each node is visited once during the construction.

Adding both complexities together, the overall time complexity of the algorithm is $O(n)$.

Space Complexity

1. **DFS Traversal:** The auxiliary space is used to store the inorder traversal of the BST in the `vals` list, which will contain n elements, thus requiring $O(n)$ space.
2. **Constructing Balanced BST:** The recursive `build` function will use additional space on the call stack. In the worst case, there will be $O(\log n)$ recursive calls on the stack at the same time, given that the newly constructed BST is balanced (hence the maximum height of the call stack corresponds to the height of a balanced BST).

Overall, considering both the space used by the list `vals` and the recursion call stack, the space complexity of the algorithm is $O(n)$ for the list and $O(\log n)$ for the call stack, leading to $O(n)$ space complexity when combined.