426. Convert Binary Search Tree to Sorted Doubly Linked List

Problem Description

The task is to convert a Binary Search Tree (BST) into a sorted Circular Doubly-Linked List, utilizing an in-place transformation. A BST is a tree data structure where each node has at most two children, for which the left child is less than the current node, and the right child is greater. A Circular Doubly-Linked List is a series of elements in which each element points to both its previous and next element, and the list is connected end-to-end.

The conversion should maintain the following conditions:

- Each node's left pointer should reference its predecessor in the list.
- Each node's right pointer should reference its successor in the list.
- The list should be circular, with the last element pointing to the first as its successor, and the first element pointing to the last as its predecessor.

The resultant structure needs to be returned with a pointer to its smallest element, effectively the head of the list.

Intuition

The solution to transforming a BST into a sorted doubly-linked list lies in the properties of the BST. The in-order traversal of a BST yields the nodes in ascending order, which is exactly what we want for our sorted doubly-linked list.

By performing an in-order traversal, we can visit each node in the BST in sorted order, and then adjust their left and right pointers on the fly to link them as if they were in a doubly-linked list. The following steps illustrate the approach:

1. Recursively perform an in-order traversal (left node, current node, right node) of the BST.
2. As we visit each node during the traversal, we connect it with the previously visited node (`prev`) to simulate the linked list's structure:
   - Make `prev.right` point to the current node (`root`), and make the current node's left point to `prev`. This process connects the nodes in a doubly-linked fashion.
3. For the first node, which does not have a `prev`, we set it as the `head` of our linked list.
4. After we have visited all the nodes, we connect the last visited node with the `head` to make the list circular.

By carefully updating the pointers as we traverse the BST, we manage to rearrange the original tree structure into a sorted circular doubly-linked list.

Solution Approach

The implementation of the solution follows the intuition discussed and is executed in several steps, utilizing a depth-first search (DFS) in-order traversal.

Here's a step-by-step breakdown:

1. **Initialization**: Before starting the DFS, we declare two variables `head` and `prev` as `None`. The `head` will eventually point to the smallest element of the BST, which will become the head of the doubly-linked list. The `prev` is used to keep track of the last processed node in the in-order traversal.
2. **Using DFS for In-order Traversal**: The function `dfs` is defined nested within the `treeToDoublyList` function. It takes a single argument `root`, which initially is the root of the BST.
3. **Recursive Traversal**: The `dfs` function is designed to be called recursively, going left (`dfs(root.left)`), processing the current node, and then going right (`dfs(root.right)`). This is the essence of in-order traversal.
4. **Connecting Nodes**: Upon visiting each node `root` during the traversal, we execute logic to redefine its left and right pointers:
   - If `prev` is not `None`, we set `root.left` to `prev` and `prev.right` to `root`, creating reciprocal links between `prev` (the predecessor) and `root` (the current node).
   - If `prev` is `None`, it means we are processing the very first node (smallest element) of the BST which should become the `head` of the doubly-linked list.
5. **Update Previous Node**: After linking the current node to its predecessor, we update `prev` to point to the current node before proceeding with the traversal to the right.
6. **Closing the Circular List**: Once the DFS is done processing all nodes, the `prev` will be pointing to the last (largest) element in the BST. At this point, we connect the `prev.right` to `head` and `head.left` to `prev` to close the loop and make the doubly-linked list circular.

In summary, the complexity of the algorithm lies in the recursive traversal of the BST and the efficient updating of the node's pointers to transform the BST structure into a doubly-linked list format.

Example Walkthrough

Let's consider a simple BST for this example:

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

In this BST, the traversal order for converting it into a sorted doubly-linked list using in-order traversal would be: 1 → 2 → 3 → 4 → 5.

Let's walk through the transformation process:

Initialization:

- `head` initially `None`, will point to the node with the smallest value (in this example, it will be `1`).
- `prev` initially `None`, will keep track of the last node processed to link it with the current node.

Using DFS for In-order Traversal:

- Define and begin the recursive in-order traversal with the root (the node with the value `4`).

Recursive Traversal:

- Process left subtree (`1`). This takes us further to the left to node `1`, which is the start of the list.
- At node `1`, there is no `prev`, so set `head` to `1`. Since there's no previous node, we can't link it yet. `prev` becomes `1`.
- Return to node `2` and link it with `prev` (which is `1`). Set `1.right` to `2` and `2.left` to `1`.
- There is no left subtree to the node `3`, so we just set `prev.right` which is `2.right` to `3` and `3.left` to `2`. `prev` is updated to `3`.
- Return to `4`, link `3.right` to `4` and `4.left` to `3`. `prev` set to `4`.
- Finally, process right subtree (`5`). Since there's no left child for the node with `5`, we directly link `4.right` to `5` and `5.left` to `4`. `prev` is updated to `5`.

Closing the Circular List:

- Post traversal, `prev` is pointing to `5`.
- Close the circular list by linking `5.right` to `head` (which is pointing to `1`) and `1.left` to `5`.

The result is a circular doubly-linked list with the following connections (illustrated as pointers):

```
1 <==> 2 <==> 3 <==> 4 <==> 5
|                           |
|___________________________|
```

The circular list starts at `1` and ends at `5`, with `5` pointing back to `1` and `1` pointing back to `5`.

Python Solution

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

class Solution:
    def treeToDoublyList(self, root: 'Optional[Node]') -> 'Optional[Node]':
        # Helper function to perform the in-order traversal of the tree.
        def dfs_inorder_traversal(node, prev):
            # Base case: if the node is None, return to the previous call.
            if not node:
                return prev

            # Recursive case: traverse the left subtree.
            dfs_inorder_traversal(node.left)

            # Process the current node.
            nonlocal previous_head
            if previous:
                # Link the previous node with the current node from the left.
                previous.right = node
                # Link the current node with the previous node from the right.
                node.left = previous
            else:
                # If this node is the leftmost node, it will be the head of the doubly linked list.
                head = node
            # Mark the current node as the previous one before the next call.
            previous = node

            # Recursive case: traverse the right subtree.
            dfs_inorder_traversal(node.right)

        # If the input tree is empty, return None.
        if not root:
            return None

        # Initialize the head and previous pointer used during the in-order traversal.
        head = previous = None
        # Perform the in-order traversal to transform the tree to a doubly linked list.
        dfs_inorder_traversal(root)
        # Connect the last node visited (previous) with the head of the list to make it circular.
        previous.right = head
        head.left = previous

        # Return the head of the doubly linked list.
        return head
```

Java Solution

```java
class Solution {
    // This 'previous' node will help in keeping track of the previously processed node in the inorder traversal
    private Node previous;

    // The 'head' node will serve as the head of the doubly linked list
    private Node head;

    // Convert a binary search tree to a sorted, circular, doubly-linked list
    public Node treeToDoublyList(Node root) {
        if (root == null) {
            // Return null if the tree is empty, there is nothing to process or convert
            return null;
        }

        // Initialize 'previous' and 'head' as null before the depth-first search
        previous = null;
        head = null;

        // Start the inorder traversal from the root to convert the BST into a sorted list
        inorderTraversal(root);

        // After the traversal, connect the last node with the 'head' to make it circular
        head.left = previous;
        previous.right = head;

        return head; // Return the head of the doubly linked list
    }

    // Inorder traversal of the binary search tree
    private void inorderTraversal(Node node) {
        if (node == null) {
            return; // Base case for the recursive function, stop if the current node is null
        }

        // Recursively process the left subtree
        inorderTraversal(node.left);

        // In the inorder traversal, 'previous' will be null only for the leftmost node
        if (previous != null) {
            // Connect the previous node's right to the current node
            previous.right = node;
        } else {
            // Connect the current node's left to the previous node
            node.left = previous;
        }
        // If 'previous' is null, it means we are at the leftmost node which is the 'head' of the list
        head = node;

        // Update 'previous' to be the current node before moving to the right subtree
        previous = node;

        // Recursively process the right subtree
        inorderTraversal(node.right);
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    Node* prevNode;
    Node* listHead;

    // Main function to convert a BST to a sorted circular doubly-linked list.
    Node* treeToDoublyList(Node* root) {
        if (!root) return nullptr;  // If the tree is empty, return nullptr.

        // Reset the prevNode and listHead pointers before starting the DFS.
        prevNode = nullptr;
        listHead = nullptr;

        // Perform a depth-first search to traverse the tree in order.
        DFSInOrder(root);

        // After the DFS is done, connect the head and tail to make it circular.
        prevNode->right = listHead;
        listHead->left = prevNode;

        return listHead;  // Return the head of the doubly linked list.
    }

    // Helper DFS function to perform in-order traversal.
    void DFSInOrder(Node* currentNode) {
        if (!currentNode) return;  // Base case: if the current node is null, just return.

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

        // When processing the current node:
        if (prevNode) {
            // If prevNode is not null, link it with the current node.
            prevNode->right = currentNode;
            currentNode->left = prevNode;
        } else {
            // If this is the leftmost node, it will be the head of the doubly linked list.
            listHead = currentNode;
        }

        // Move prevNode to the current node before traversing the right subtree.
        prevNode = currentNode;

        // Traverse the right subtree.
        DFSInOrder(currentNode->right);
    }
};
```

Typescript Solution

```typescript
// Definition for a Node.
class Node {
    val: number
    left: Node | null
    right: Node | null
    constructor(val: number, left: Node | null = null, right: Node | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

/**
 * Converts a binary search tree to a sorted circular doubly-linked list.
 * @param {Node | null} root - The root node of the binary search tree.
 * @return {Node | null}
 */
function treeToDoublyList(root: Node | null): Node | null {
    if (!root) return root;

    let previous: Node | null = null;
    let head: Node | null = null;

    /**
     * Depth-first search (In-order traversal) to iterate over the tree and create the doubly linked list.
     * @param {Node | null} node - The current node being visited.
     */
    function inOrderTraversal(node: Node | null): void {
        if (!node) return;

        // Traverse the left subtree
        inOrderTraversal(node.left);

        // Link the current node with the previous node
        if (previous) {
            previous.right = node;
            node.left = previous;
        } else {
            // Set the head if this is the leftmost node
            head = node;
        }

        // Move the 'previous' pointer to the current node
        previous = node;

        // Traverse the right subtree
        inOrderTraversal(node.right);
    }

    // Start the in-order traversal
    inOrderTraversal(root);

    // Connect the head and tail to make the list circular
    head.left = previous;
    previous.right = head;

    return head;
};
```

Time and Space Complexity

The given Python code is designed to convert a binary search tree into a sorted, circular doubly-linked list in-place. We analyze the time complexity and space complexity of the provided code as follows:

Time Complexity:

The time complexity of the code is determined by the in-order traversal of the binary search tree.

- Each node in the tree is visited exactly once during the traversal.
- The operations performed for each node are constant time operations, including updating the previous (`prev`) and current node's pointers.

Therefore, if there are `n` nodes in the tree, the in-order traversal will take `O(n)` time. As a result, the overall **time complexity** of the code is `O(n)`.

Space Complexity:

The space complexity of the code is determined by:

- The implicit stack space used during the recursive in-order traversal (due to `dfs` calls).
- No additional data structures are used that are proportional to the size of the input.

Hence, the maximum space is taken by the recursion stack, which in the worst case (a completely unbalanced tree) could have `n` recursive calls on the stack. However, in the best case (a completely balanced tree), the height would be `log(n)`, and therefore the recursion stack would be `O(log(n))`.

In accordance with this, the **space complexity** of the code is:

- Worst case (skewed tree): `O(n)`
- Average case (balanced tree): `O(log(n))`