114. Flatten Binary Tree to Linked List

Medium | Stack | Tree | Depth-First Search | Linked List | Binary Tree | Leetcode Link

Problem Description

In this problem, we're asked to transform a given binary tree into a “flattened” linked list. The conditions for the flattened linked list are as follows:

- We need to use the existing `TreeNode` class to represent the list nodes.
- Each `right` child pointer of the `TreeNode` should point to the next node in the linked list.
- The `left` child pointer of each `TreeNode` should be set to `null`, effectively removing the left children from all nodes.
- The order of the nodes in the linked list must reflect the order of nodes visited in a pre-order traversal (visit the current node, then traverse left, then traverse right) of the binary tree.

The problem constraints are such that the tree needs to be modified in place, meaning no additional data structures should be created and the transformation has to use the existing tree nodes.

Intuition

To solve the problem, we need to understand that a pre-order traversal follows the root-left-right order. This guides us to the solution where we can repeatedly move the left subtree of a node so that it becomes the right subtree, while also ensuring that the original right subtree is not lost but appended after the moved left subtree.

Here's the intuition behind the solution step by step:

1. We start at the root of the binary tree and check if there's a left child.
2. If a left child is present, we find the right-most node of the left subtree (this will be the pre node mentioned in the code). This is where we will link the existing right subtree after we've moved the left subtree.
3. We then attach the current right subtree to the right-most node of the left subtree.
4. Now, we can safely move the left subtree to replace the right subtree (since we've already preserved the original right subtree).
5. We set the left child of the current node to `null`, effectively flattening this part of the tree.
6. We then move on to the next right child and repeat this process.
7. We continue this operation until we have visited all nodes and flattened the entire tree.

This in-place transformation is accomplished without the use of additional space (i.e., no extra list or tree is created), which is a requirement of this problem.

Solution Approach

The solution implementation provided is an excellent example of in-place tree manipulation, which cleverly modifies the tree structure without the need for additional data structures. Here are the key elements of the implementation, expanded to clarify the process:

1. **Loop Structure**: The solution utilizes a `while` loop to iterate through the nodes of the tree. This loop continues until there are no more nodes to process, which is when `root` becomes `None`.

2. **Left Subtree Processing**: Inside the loop, the first step is to check if the current node (`root`) has a left child. If not, the tree is already flat from the current node, so the algorithm simply moves to the `root.right`.

3. **Finding Right-Most Node**: When a left child exists, we then need to find the right-most node of the left subtree. This part of the solution uses another `while` loop to traverse down to the bottom right of the subtree (`while pre.right:`). The right-most node is where the original right subtree will be appended after the left subtree is moved to the right.

4. **Re-linking Subtrees**: After finding the right-most node, the next step in the algorithm is to set its `right` child to the current `root.right`, which preserves the original right subtree (`pre.right = root.right`).

5. **Flattening**: Now that the right subtree has been saved, the entire left subtree is moved to the right side (`root.right = root.left`). This move aligns with the requirement that the `right` child pointer points to the next node in the list.

6. **Eliminating Left Children**: To comply with the flattened structure, the left child of the current node is set to `None` (`root.left = None`).

7. **Advancing to the Next Node**: Finally, with the left child processed and moved, the algorithm progresses to the next node by moving `root` to `root.right`.

By iteratively applying these steps to each node in the tree, starting from the root and advancing rightward, the binary tree is gradually transformed into a "linked list" that satisfies all given requirements.

The algorithm's foundation relies on tree traversal and careful reassignment of pointers, and the procedure harnesses the feature of the pre-order traversal pattern. Due to its in-place nature, the algorithm avoids using additional memory beyond a few pointers, and the entire tree is flattened with a space complexity of O(1) and time complexity of O(n), where n is the number of nodes in the binary tree.

Example Walkthrough

Let's consider a simple binary tree and walk through the process of flattening it into a linked list using the solution approach described above.

Suppose we have the following binary tree:

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

Now, let's flatten this tree step-by-step:

1. **Start at Root**: We begin with the root node `1`. It has a left child, which means we will need to find the right-most node of the left subtree.

2. **Find Right-most Node of Left Subtree**: The left child of `1` is `2`, and it has a right child `4`. We traverse to the right-most node of this left subtree, which is `4`.

3. **Re-linking Subtrees**: We attach the right subtree of `1` (which starts with node `5`) to the right pointer of node `4`. Now node `4`'s right pointer points to node `5`.

```
    1        4
   /          \
  2     =>     5
 / \            \
3   4            6
```

4. **Flattening**: We move the entire left subtree of `1` to the right. Now, the right child of `1` is `2`, and the tree looks like this:

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

5. **Eliminate Left Children**: We set the left child of node `1` to `null`.

6. **Advancing to Next Node**: We now move to the right child of the root, which is `2`. We repeat the process. Node `2` has a left child `3`, but since `3` has no right child, it is already in the right place. We set the left child of node `2` to `null`.

7. **Continue Flattening**: Moving to node `4`, we find no left child, so we continue to node `5`. Node `5` has no left child but has a right child `6`, which is already in the right place.

The tree is now a linked list:

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

We've completed the in-place transformation. Each step of the process either involved relinking or moving to the right child, and at no point did we use additional space. The original tree structure is efficiently re-purposed into a linked list following the pre-order traversal order.

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 flatten(self, root: Optional[TreeNode]) -> None:
        """
        Flattens a binary tree to a linked list in place using the right child pointers.
        The preorder traversal should be followed to flatten the tree.

        :param root: The root of the binary tree.
        :return: None, the tree is modified in place.
        """
        # Iterate through each node of the binary tree
        while root:
            # If there is a left child, we need to rewire the connections according to
            # the "flattened" preorder traversal
            if root.left:
                # Find the rightmost node of the left subtree
                predecessor = root.left
                while predecessor.right:
                    predecessor = predecessor.right
                # Connect the predecessor's right pointer to the current node's right child
                predecessor.right = root.right
                # The left child becomes the right child and the left child is set to None
                root.right = root.left
                root.left = None
            # Move to the next right node (this could be the former left child of the current node)
            root = root.right

        # The tree has now been converted to a linked list following the right child pointers.

```

Java Solution

```java
class Solution {

    // The method to flatten a binary tree to a linked list in-place.
    public void flatten(TreeNode root) {
        // Iterate until all the nodes are processed.
        while (root != null) {
            // If the current node has a left child, we need to process it.
            if (root.left != null) {
                // Find the rightmost node of the left subtree.
                TreeNode rightmost = root.left;
                while (rightmost.right != null) {
                    rightmost = rightmost.right;
                }
                // Make the right of the rightmost node point to the root's right node.
                rightmost.right = root.right;
                // Move the left subtree to the right.
                root.right = root.left;
                // After moving the left subtree, set the left child to null.
                root.left = null;
            }
            // Move on to the right child of the current node.
            root = root.right;
        }
    }
}

/**
 * Definition for a binary tree node.
 * This is provided for completeness. Assume the TreeNode class is predefined elsewhere.
 */
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    // Constructor with no arguments initializes an empty node.
    TreeNode() {}
    // Constructor with a value initializes a node with the given value.
    TreeNode(int val) { this.val = val; }
    // Constructor with a value and two subtrees initializes a node with those values.
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

```

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:
    // Function to flatten a binary tree into a linked list in-place.
    void flatten(TreeNode* root) {
        // Continue flattening the tree until we have processed all nodes.
        while (root) {
            // Check if the current node has a left child.
            if (root->left) {
                // Find the rightmost node in the left subtree.
                TreeNode* rightMost = root->left;
                while (rightMost->right) {
                    rightMost = rightMost->right;
                }

                // Attach the right subtree of the current node to the rightmost node.
                rightMost->right = root->right;

                // Move the left subtree under the current node to the right.
                root->right = root->left;

                // Set the left child of the current node to nullptr.
                root->left = nullptr;
            }

            // Move on to the right child of the current node.
            root = root->right;
        }
    }
};

```

Typescript Solution

```typescript
// Definition for a binary tree node.
class TreeNode {
    val: number
    left: TreeNode | null
    right: TreeNode | null
    constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
        this.val = (val === undefined ? 0 : val);
        this.left = (left === undefined ? null : left);
        this.right = (right === undefined ? null : right);
    }
}

/**
 * Flattens a binary tree to a linked list in-place.
 * @param {TreeNode | null} root - The root of the binary tree.
 */
function flatten(root: TreeNode | null): void {
    // Iterate through the binary tree nodes.
    while (root !== null) {
        // If the current node has a left child...
        if (root.left !== null) {
            // Find the rightmost node of the left subtree.
            let predecessor = root.left;
            while (predecessor.right !== null) {
                predecessor = predecessor.right;
            }
            // Connect right subtree of the current node to the rightmost node of the left subtree.
            predecessor.right = root.right;

            // Move the left subtree to the right side and nullify the left pointer.
            root.right = root.left;
            root.left = null;
        }
        // Move to the next right node.
        root = root.right;
    }
}

```

Time and Space Complexity

The above Python code implements a method to flatten a binary tree in-place along its pre-order traversal.

Time Complexity:

The time complexity of the algorithm is `O(N)`, where `N` is the number of nodes in the tree. This is because each node is visited once. The inner while loop that finds the rightmost node of the left subtree does not add to the overall asymptotic complexity since it operates on nodes only once throughout the entire run of the algorithm. Each edge in the tree will be traversed at most twice: once when visiting the left child and once when attaching the left subtree to the right.

Space Complexity:

The space complexity of the algorithm is `O(1)` if we don't consider the recursive stack space used by the system for managing function calls. This is because no extra data structures are used; the tree is modified in place.

However, if we were to consider the recursive stack space (implicit stack space due to function calls), the space complexity would be `O(H)`, where `H` is the height of the binary tree because the space used at any time is proportional to the depth of the recursion tree, which, in the worst case, is equal to the height of the binary tree. In a balanced tree, this would be `O(log N)`, but in a skewed tree, it can be as bad as `O(N)` in the case of a degenerate tree (when the tree is like a linked list).