606. Construct String from Binary Tree

Easy Tree Depth-First Search String Binary Tree Leetcode Link

Problem Description

The problem provides a binary tree (where each node has at most two children) and asks you to transform it into a specific string format based on the rules of preorder traversal. This traversal visits the root, followed by the left subtree, and then the right subtree. As you navigate the tree, the goal is to create a string that represents the tree structure via parentheses. Each node's value must be captured in the string, and for non-leaf nodes, its children should be represented inside parentheses. Importantly, if a node has an empty right child, you still need to use a pair of parentheses to indicate the left child, but you are not required to include parentheses for an empty left child if there is also no right child. The challenge is to omit all unnecessary empty parentheses pairs to avoid redundancy in the resulting string.

Intuition

The solution is grounded in the idea of depth-first search (DFS), which explores as far as possible along each branch before backtracking. This concept is perfect for the preorder traversal required by the problem. To implement the DFS, you recursively visit nodes, starting from the root down to the leaves, while building the string representation as per the rules.

Here's an intuitive breakdown of the approach:

- If the current node is `None` (meaning you've reached a place without a child), return an empty string since you don't need to add anything to the string.
- If the current node is a leaf node (it has no left or right child), return the node's value as a string, since no parentheses are needed.
- If the node has a left child but no right child, you only need to consider the left subtree within parentheses immediately following the node's value. There's no need for parentheses for the nonexistent right child in this case.
- If the node has both left and right children, you must include both in the string with their respective parentheses following the node's value.

Recursive calls are made to cover the entire tree, and at each step, you construct the string according to which children the current node has. By following these recursive rules, you ensure that you only include necessary parentheses, thus preventing any empty pairs from appearing in the final output string.

Solution Approach

The solution uses a simple recursive algorithm, which is an application of the Depth-First Search (DFS) pattern. Here's how it works:

- A helper function `dfs` is defined that accepts a node of the tree as its parameter. This function is responsible for the recursive traversal and string construction.
- The base case of this recursion is when the `dfs` function encounters a `None` node. In binary trees, a `None` node signifies that you've reached beyond the leaf nodes (end of a branch). When this happens, an empty string is returned because there's nothing to add to the string representation from this node downwards.
- When the current node is a leaf (neither left nor right child exists), we simply return the string representation of the node's value because leaves do not require parentheses according to the problem rules.
- When a node has a left child but no right child, we include the left child's representation within parentheses following the node's value. However, the right side doesn't need any parentheses or representation because it's a `None` node.
- The most involved case is when both left and right children exist. In this scenario, we need to represent both children. Therefore, recursively the `dfs` function is called for both the left and the right child, and their representations are enclosed in parentheses with the format: `node value(left child representation)(right child representation)`. This ensures that the preorder traversal order is maintained.

Additionally, the given solution employs an inner function `dfs` and makes use of Python's string formatting to concatenate the values and parentheses cleanly.

Here's an implementation breakdown corresponding to the code sections:

```python
def tree2str(self, root: Optional[TreeNode]) -> str:
    def dfs(root):
        if root is None:  # Base case for NULL/None nodes.
            return ''
        if root.left is None and root.right is None:  # Leaf node case.
            return str(root.val)
        if root.right is None:  # Node with only left child case.
            return f'{root.val}({dfs(root.left)})'
        # Node with both left and right children case.
        return f'{root.val}({dfs(root.left)})({dfs(root.right)})'

    return dfs(root)  # Initial call to the dfs function with the tree root.
```

- At the start, `dfs(root)` is called with the root of the binary tree.
- The `if` conditions inside the `dfs` function handle the cases mentioned previously and are responsible for constructing the string as per the rules and format required by the problem statement.
- The resulting string is built up step-by-step through recursive calls until the entire tree has been traversed, thus providing the correct string representation.

Example Walkthrough

Let's consider a small binary tree example to illustrate the solution approach. The binary tree used in this example is as follows:

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

In this tree:

- The root node (1) has two children (2 and 3).
- The left child of the root (2) has a left child of its own (4) and no right child.
- The right child of the root (3) is a leaf node and has no children.
- The leaf node (4) also has no children.

Using the problem's rules and solution approach, we perform a preorder traversal to construct the string:

1. Start with the root (1). Since it's not `None` and not a leaf, the string representation starts with its value. So far we have the string `"1"`.
2. Next, visit the left child (2) of node (1). Again, as it is not a `None` or a leaf node and it has a left child but no right child, we include its left child wrapped in parentheses. So, we now have `"1(2"`.
3. We now go to the left child of node (2), which is node (4). This node is a leaf, so we simply return its value as a string, resulting in the substring `"4"`. Adding this to the previous string, we get `"1(2(4)"`.
4. Since node (2) has no right child, we do not need to include it, and we close the parentheses for node (2). Our string is now `"1(2(4))"`.
5. Next, we consider the right child of the root, which is node (3). It is a leaf node, so we just need its value for the string, wrapped in parentheses because its sibling node (2) has children. Our string becomes `"1(2(4))(3)"`.
6. Finally, since we have represented both children of the root (1), we close the representation of the root. The final string is `"1(2(4))(3)"`, with no unnecessary parentheses.

Using the given solution approach, the recursive function `dfs` would follow these steps internally to build the string representation smoothly. Here's the implementation of the tree structure in code and how the `dfs` function builds the string:

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

def tree2str(root: Optional[TreeNode]) -> str:
    def dfs(root):
        if root is None:
            return ''
        if root.left is None and root.right is None:  # Leaf node case.
            return str(root.val)
        if root.right is None:  # Node with only left child case.
            return f'{root.val}({dfs(root.left)})'
        # Node with both left and right children case.
        return f'{root.val}({dfs(root.left)})({dfs(root.right)})'

    return dfs(root)

# Example tree construction
node4 = TreeNode(4)
node2 = TreeNode(2, node4)
node3 = TreeNode(3)
root = TreeNode(1, node2, node3)
# Call to transform tree into string
print(tree2str(root))  # Output should be "1(2(4))(3)"
```

Following the recursive process described above using `dfs`, we obtain the expected string representation of the binary tree based on the problem's guidelines.

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 tree2str(self, root: TreeNode) -> str:
        # Helper function to perform depth-first search traversal of the tree.
        def dfs(node):
            # If current node is None, return an empty string.
            if node is None:
                return ''

            # Convert the node's value to a string if it's a leaf node.
            if node.left is None and node.right is None:
                return str(node.val)

            # If the right child is None, only include the left child.
            if node.right is None:
                return f'{node.val}({dfs(node.left)})'

            # When both children are present, include both in the string representation.
            return f'{node.val}({dfs(node.left)})({dfs(node.right)})'

        # The main function call which starts the DFS traversal from the root.
        return dfs(root)

```

Java Solution

```java
class Solution {

    // Converts a binary tree into a string representation following specific rules:
    // 1. Omit any children if both left and right child nodes are null
    // 2. Include only left child if right child is null
    // 3. Include both left and right children if they are not null
    public String tree2str(TreeNode root) {
        // Base case: if the current node is null, return an empty string
        if (root == null) {
            return "";
        }

        // Case when both left and right child nodes are null
        if (root.left == null && root.right == null) {
            return Integer.toString(root.val);
        }

        // Case when only the right child node is null
        if (root.right == null) {
            return root.val + "(" + tree2str(root.left) + ")";
        }

        // Case when both child nodes are not null
        // Note: The right child node is represented even when it might be null,
        //       because the left child node is not null, and its existence must be acknowledged
        return root.val + "(" + tree2str(root.left) + ")(" + tree2str(root.right) + ")";
    }

    // Definition for a binary tree node provided by the LeetCode environment
    public class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {}

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }
}

```

C++ Solution

```cpp
#include <string>  // Include the string library for string manipulation

// Definition for a binary tree node.
struct TreeNode {
    int val;                       // Value of the node
    TreeNode *left;                // Pointer to left child
    TreeNode *right;               // Pointer to right child
    // Constructors to initialize the node
    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 convert a binary tree to a string representation.
    std::string TreeToString(TreeNode* root) {
        // Base case: if current node is nullptr, return an empty string
        if (!root) return "";

        // Handle the case where the current node is a leaf node
        if (!root->left && !root->right) {
            return std::to_string(root->val);  // Simply return the string representation of the node value
        }

        // Handle the case where the current node has a left child but no right child
        if (!root->right) {
            // Return the string representation of current node value and left subtree
            return std::to_string(root->val) + "(" + TreeToString(root->left) + ")";
        }

        // If current node has both left and right children
        return std::to_string(root->val) +
               "(" + TreeToString(root->left) + ")" +
               "(" + TreeToString(root->right) + ")";
    }
};

```

Typescript Solution

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

/**
 * Serializes a binary tree to a string.
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @returns {string} The serialized tree string, according to the LeetCode problem requirement.
 */
function tree2str(root: TreeNode | null): string {
    // Return empty string for null nodes.
    if (root == null) {
        return '';
    }

    // Return the value as a string if the node is a leaf.
    if (root.left == null && root.right == null) {
        return `${root.val}`;
    }

    // Serialize the left subtree. If left is null, use an empty string.
    const leftStr = root.left ? `(${tree2str(root.left)})` : '()';

    // Serialize the right subtree only if it's not null.
    const rightStr = root.right ? `(${tree2str(root.right)})` : '';

    // Concatenate the string, omitting the right parenthesis if the right subtree is null.
    return `${root.val}${leftStr}${rightStr}`;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the binary tree. This is because the code performs a depth-first search (DFS) and visits each node exactly once when generating the string representation of the binary tree.

The space complexity of the code is also `O(n)` in the worst case when the tree is highly skewed (i.e., each node has only one child). This comes from the recursive call stack that could have a maximum depth equivalent to the number of nodes in the tree if the tree degenerates into a linked list. In a balanced tree, the space complexity would be `O(log n)` due to the height of the balanced tree being `log n`.