# 331. Verify Preorder Serialization of a Binary Tree

Medium Stack Tree String Binary Tree

## Problem Description

The given problem revolves around the concept of serialization of a binary tree using preorder traversal. Preorder traversal means to visit the root node first, then the left subtree, and finally the right subtree. When serializing a tree, each non-null node is represented by its integer value, and null nodes are represented by a sentinel value `'#'`.

For instance, a binary tree may be serialized into a string representing the order in which nodes are visited. Notably, null child pointers are also serialized, which creates a unique representation for the structure of the original tree. The challenge here is to determine if a given string `preorder`, which contains comma-separated integer values and `'#'` symbols, is a valid serialization of a binary tree, without actually reconstructing the tree.

A valid preorder serialization must satisfy the requirements of a binary tree structure. Each node should have two children (which can be null), and the string should represent a tree where every node has been visited correctly in preorder, including null children.

## Intuition

The intuition behind the solution is to simulate the traversal of a binary tree using a stack to keep track of the nodes being visited. As we iterate through the `preorder` string, each value can be seen as an action in this simulated traversal.

When we hit a non-null node (an integer), we push it onto the stack since it represents a node that could have left and right children. Conversely, when encountering a null node (represented by `'#'`), we consider it as closing off a path in the tree (i.e., marking a leaf node).

However, a valid serialized binary tree cannot have two consecutive null nodes without them being the right and left children of some node before them. So if we have two null nodes on the top of the stack, there must be an integer just below them representing their parent node. We perform a check, and if that pattern is found, we remove (pop) the two nulls and the integer, simulating that we visited a complete subtree. We then replace them with a single null value on the stack to indicate that the entire subtree is now closed off and treated as a leaf.

We repeat this process as we move through the string. If the serialization is correct, we should end up with one single null value in the stack, which signifies the end of the traversal of a well-formed tree. On the contrary, if we're left with a different pattern, the serialization is deemed incorrect.

## Solution Approach

The provided Python code uses a stack to simulate the traversal of a binary tree during deserialization. The algorithm employs a `for` loop to iterate over nodes represented in the `preorder` serialization string split by commas. It pushes each value onto the stack, which not only represents inserting nodes but also helps to track the tree structure.

During this simulated traversal, the code looks for a specific pattern in the stack. This pattern comprises two consecutive `'#'` characters, representing null nodes or leaf nodes, followed by an integer, which would represent their parent node in a binary tree.

When this pattern is detected (`stk[-1] == stk[-2] == "#" and stk[-3] != "#"`) it indicates that we've completed the visit to a subtree - specifically, the left and right children are both null, and we have their parent node just before these nulls.

At this point, the algorithm removes the three entries (`stk = stk[:-3]`) and replaces them with a single `'#'` to represent the whole subtree as a null or leaf for any higher-level nodes that might be present in the stack. This action effectively rolls up the null children into their parent, making it into a new leaf node.

Ultimately, if the `preorder` string represents a valid serialization of a binary tree, we'll end up with exactly one element in the stack after processing the entire input (`len(stk) == 1`). This remaining element must be the sentinel value `'#'` indicating that all nodes have been accounted for, and the full tree has been traversed (`stk[0] == "#"`) without reconstructing it. If these conditions are met, the function returns `True`. Otherwise, if the stack does not adhere to this pattern, the function returns `False`, signaling that the given `preorder` serialization is not valid.

The solution is elegant as it simulates traversal without the overhead of tree construction and cleverly handles the serialization pattern-check using a stack that reflects the current state of the traversal process.

### Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have a given preorder serialization string of a binary tree: `preorder = "9,3,4,#,#,1,#,#,2,#,6,#,#"`. This serialization suggests that we have a tree with the root node value `9`, a left child `3`, which itself has a left child `4` that has no children (indicated by two consecutive `#` symbols), a right child `1` with no children, and finally a right child of the root, `2`, which has a left child `6` with no children.

- Initialize an empty stack `stk`.
- Split the `preorder` serialization by commas and iterate through the values:
  - For the first value `9`, push it onto `stk`: `stk = [9]`.
  - The next value `3` goes onto the stack: `stk = [9, 3]`.
  - Then, `4` is pushed: `stk = [9, 3, 4]`.
  - A `'#'` is encountered, indicating a null child: `stk = [9, 3, 4, '#']`.
  - Another `'#'` is encountered, so now we have *null children*, which means `4` is a leaf node. We pop three times, and push a `'#'`: `stk = [9, 3, '#']`.
  - We encounter `1` and push it onto the stack: `stk = [9, 3, '#', 1]`.
  - Again, two `'#'` symbols follow, indicating that `1` is a leaf node. Pop three and push `'#'`: `stk = [9, '#', '#']`.
- At this point, we have two `'#'` characters at the top of the stack, which suggests that the left and right children of `9` have been completely visited. We pop three and replace them with a `'#'`: `stk = ['#']`.
- Now, `2` enters the stack: `stk = ['#', 2]`. This is not correct as we have finished the tree rooted at `9` and should not add more nodes at the same level.
- As we continue, we encounter `6` and the subsequent `'#'` symbols indicating its children, which after the process will result in a `stk = ['#', '#']`, which is not a valid serialization as we are left with two sentinel values.

The correct result, in this case, should be the function returning `False`, which indicates that `preorder = "9,3,4,#,#,1,#,#,2,#,6,#,#"` is not a valid preorder serialization of a binary tree. If the serialization was valid, after processing all elements, the stack would have ended up with exactly one `'#'`, reflecting the traversal of the entire tree.

## Solution Implementation

### Python

```python
class Solution:
    def isValidSerialization(self, preorder: str) -> bool:
        # Initialize an empty stack to keep track of the nodes
        stack = []

        # Split the input string by commas to process each tree node
        for node in preorder.split(","):
            # Append the current node onto the stack
            stack.append(node)

            # While there are at least three elements in the stack,
            # and the last two are "#" which denotes null nodes,
            # and the one before them is not "#",
            # It means we have completed a subtree with a valid leaf node,
            while len(stack) >= 3 and stack[-1] == "#" and stack[-2] == "#" and stack[-3] != "#":
                # Remove the last two "#" and the parent node which was before them
                stack.pop()  # Remove first '#'
                stack.pop()  # Remove second '#'
                stack.pop()  # Remove the parent non-null node

                # Append "#" to represent the completed subtree
                stack.append("#")

        # If only one element is left in the stack and it's "#",
        # it means the entire tree is accurately represented by the serialization,
        # so it is a valid serialization
        return len(stack) == 1 and stack[0] == "#"
```

### Java

```java
class Solution {

    // Method to validate if the preorder serialization of a binary tree is correct
    public boolean isValidSerialization(String preorder) {
        // Use a stack reoresented by a list to keep track of tree nodes.
        List<String> stack = new ArrayList<>();

        // Split the inaut string bv commas to work with each node/leaf individually.
        String[] nodes = preorder.split(",");
        for (String node : nodes) {
            // Push the current node onto the stack.
            stack.add(node);

            // Check the last three elements in the stack if they form a pattern of two
            // consecutive '#' symbols which denote null children followed by a numeric node.
            while (stack.size() >= 3 && stack.get(stack.size() - 1).equals("#")
                && stack.get(stack.size() - 2).equals("#") && !stack.get(stack.size() - 3).equals("#")) {

                // Since the last two '#' symbols reoresent the null children of the previous numeric
                // node, we can remove them all mimicking the null leaves falling off,
                // which means they don't take up space in serialization.
                stack.remove(stack.size() - 1); // Remove last null child
                stack.remove(stack.size() - 1); // Remove second last null child
                stack.remove(stack.size() - 1); // Remove parent node

                // After removing a parent node and its two null children,
                // we add one '#' to the stack to indicate that the subtree has been fully traversed.
                stack.add("#");
            }
        }

        // After processing all nodes, a valid serialization will end up with only one element in the stack,
        // which must be '#', representing that all nodes have been matched with their children.
        return stack.size() == 1 && stack.get(0).equals("#");
    }
}
```

### C++

```cpp
#include <sstream>
#include <string>
#include <vector>

class Solution {
public:
    // Function to check if the given preorder traversal string of a binary tree
    // represents a valid serialization of a binary tree.
    bool isValidSerialization(std::string preorder) {
        std::vector<std::string> stack;
        std::stringstream ss(preorder); // Using stringstream to parse the string
        std::string node;

        // Splitting the input string by commas and processing each node
        while (getline(ss, node, ',')) {
            stack.push_back(node); // Push the current node onto the stack

            // Check if the last three elements on the stack are two "#"
            // followed by a non-"#" node, which represents a valid subtree.
            while (stack.size() >= 3 && stack[stack.size() - 1] == "#" &&
                   stack[stack.size() - 2] == "#" && stack[stack.size() - 3] != "#") {
                // Pop the two "#" nodes representing null children
                stack.pop_back(); // Remove right null child
                stack.pop_back(); // Remove left null child

                // Pop the parent node of the null children
                stack.pop_back();

                // The complete subtree is replaced by "#", which signifies that
                // this part of the tree is properly serialized
                stack.push_back("#");
            }
        }

        // If the stack contains only one element and it is "#", then it is a valid serialization
        return stack.size() == 1 && stack[0] == "#";
    }
};
```

### TypeScript

```typescript
function isValidSerialization(preorder: string): boolean {
    let stack: string[] = []; // Initialize stack as an array of strings
    const nodes = preorder.split(","); // Split the string by commas to process nodes individually

    for (let node of nodes) {
        stack.push(node); // Push the current node onto the stack

        // Keep reducing the nodes that form a complete subtree into one '#'
        while (stack.length >= 3 &&
               stack[stack.length - 1] === "#" &&
               stack[stack.length - 2] === "#" &&
               stack[stack.length - 3] !== "#") {

            // Here we found a pattern which is two null child nodes followed by a non-null parent node
            stack.pop(); // Remove right null child
            stack.pop(); // Remove left null child

            // Pop the parent node of the null children to replace that subtree with a '#'
            stack.pop();

            // Substitute the entire subtree with a single "#" to denote the presence of a valid subtree
            stack.push("#");
        }
    }

    // If at the end there's only one element in the stack and it's '#', it's a correctly serialized tree
    return stack.length === 1 && stack[0] === "#";
}
```

```python
class Solution:
    def isValidSerialization(self, preorder: str) -> bool:
        # Initialize an empty stack to keep track of the nodes
        stack = []

        # Split the input string by commas to process each tree node
        for node in preorder.split(","):
            # Append the current node onto the stack
            stack.append(node)

            # While there are at least three elements in the stack,
            # and the last two are "#" which denotes null nodes,
            # and the one before them is not "#",
            # It means we have completed a subtree with a valid leaf node,
            while len(stack) >= 3 and stack[-1] == "#" and stack[-2] == "#" and stack[-3] != "#":
                # Remove the last two "#" and the parent node which was before them
                stack.pop()  # Remove first '#'
                stack.pop()  # Remove second '#'
                stack.pop()  # Remove the parent non-null node

                # Append "#" to represent the completed subtree
                stack.append("#")

        # If only one element is left in the stack and it's "#",
        # it means the entire tree is accurately represented by the serialization,
        # so it is a valid serialization
        return len(stack) == 1 and stack[0] == "#"
```

## Time and Space Complexity

### Time Complexity

The given Python code primarily consists of iterating through each character in the `preorder` string once, which means it operates on each element in linear time. Specifically, the `split()` function that operates on `preorder` runs in `O(n)` time, where `n` is the length of the input string, since it must visit each character to split the string by commas.

Inside the loop, there is a `while` loop that could potentially run multiple times for certain stacks. However, each element can be added and removed from the stack at most once. Due to this property, despite the `while` loop, the overall number of `append` and subsequently `pop` operations is still linear with respect to the number of nodes or elements in the `preorder` sequence.

Consequently, the time complexity of the code is `O(n)`.

### Space Complexity

The space complexity is determined by the additional space used which is primarily for the stack (`stk`). In the worst case, the stack could contain all leaf nodes before it begins replacing them with `"#"`. In a full binary tree, the number of leaf nodes is approximately `n/2`. Thus, in the worst-case scenario, the space complexity would also be `O(n)`.

However, it should be noted that this space complexity analysis assumes that the input string is not considered as extra space used (as it is usually considered input size). If we were to consider any extra space required for the stack itself, then the space complexity is `O(n)` as we did above.