1430. Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree

Medium Tree Depth-First Search Breadth-First Search Binary Tree Leetcode Link

Problem Description

In this problem, we are given a binary tree where each root-to-leaf path represents a sequence of numbers corresponding to the values of the nodes along the path. The task is to determine if a given sequence, represented by an array `arr`, matches any of these root-to-leaf sequences in the binary tree.

The sequence is considered valid if it corresponds exactly to the sequence of node values from the root node to a leaf node. This means that each value in the given array `arr` must match the value of the corresponding node in the tree as we traverse from the root to a leaf, and the sequence should end at a leaf node.

A leaf node is defined as a node with no children, implying that it doesn't have a left or right child node. In simple terms, we need to check if there's a path in the given binary tree such that the concatenation of the node values along the path is exactly the same as the given sequence `arr`.

Intuition

The solution to this problem is based on Depth-First Search (DFS), which is a fundamental traversal algorithm that explores as far as possible along each branch before backtracking. The idea is to traverse the tree from the root, comparing the value at each node with the corresponding element in the given sequence `arr`.

Here's the thinking process for arriving at the DFS solution:

- We start the DFS from the root of the tree.
- At any given node, we check if the node's value matches the corresponding element in the sequence. If not, we return `False` because this path can't possibly match the sequence.
- We also keep track of the index `u` within the sequence `arr` that we're currently checking. We increment this index as we move down the tree to ensure we compare the correct node value with the correct sequence element.
- If the current node's value matches the current sequence element and we're at the last element in the sequence (i.e., `u == len(arr) - 1`), we check if the current node is also a leaf (it has no children). If it is a leaf, the sequence is valid and we return `True`. If it's not a leaf, the sequence is invalid because it hasn't ended at a leaf node.
- If the current node's value matches the current sequence element but we're not yet at the end of the sequence, we recursively perform DFS on both the left and right children, incrementing the index `u`.
- The invocation of DFS on a child node returns `True` if that subtree contains a path corresponding to the remaining portion of the sequence.
- If DFS on both the left and right children returns `False`, this means that there is no valid continuation of the sequence in this part of the tree, and we return `False`.
- The initial call to DFS starts with the root node and the first index of the sequence `arr`.

By applying this approach, we incrementally check each root-to-leaf path against the sequence until we either find a valid path that matches the sequence or exhaust all paths and conclude that no valid sequence exists in the binary tree.

Solution Approach

The implementation of the solution utilizes Depth-First Search (DFS), a classic algorithm often used to explore all paths in a tree or graph until a condition is met or all paths have been explored.

Below are the details of the DFS implementation:

- We start by defining a recursive function `dfs` within the method `isValidSequence`. This function takes two arguments: `root`, representing the current node in the tree, and `u`, which is the index of the current element in the sequence array `arr`.

- Within the `dfs` function, we first check if the current node `root` is `None` (indicating we've reached a non-existent node beyond the leaf of a path) or if the value of `root.val` does not match the sequence value `arr[u]`. In either case, we return `False` because it means we cannot form the desired sequence along this path.

- The next crucial check determines if we have reached the end of our sequence with `u == len(arr) - 1`. If the current node is at this index of the sequence array, we must also verify if it is a leaf. The condition `root.left is None and root.right is None` confirms whether the current node has no children. If both conditions hold, we have found a valid sequence and return `True`.

- If we are not at the end of the sequence, we continue our DFS on both the left and right subtrees by calling `dfs(root.left, u + 1)` and `dfs(root.right, u + 1)`. Here, we increment the index `u` by `1` indicating that we're moving to the next element in the sequence as we go one level down the tree.

- Finally, we use an `or` operator between the calls to `dfs` on the left and right children because a valid sequence can exist in either subtree. If either subtree call returns `True`, it means we have a match, and so we return `True` for this path.

Here is the implementation of the recursive `dfs` function encapsulated in the `Solution` class and its method `isValidSequence`:

```python
class Solution:
    def isValidSequence(self, root: TreeNode, arr: List[int]) -> bool:
        def dfs(root, u):
            if root is None or root.val != arr[u]:
                return False
            if u == len(arr) - 1:
                return root.left is None and root.right is None
            return dfs(root.left, u + 1) or dfs(root.right, u + 1)

        return dfs(root, 0)
```

- The solution overall begins with the call to `dfs(root, 0)` where `root` is the tree's root node and `0` is the starting index of the sequence `arr`.

This approach effectively performs a depth-first traversal of the tree, comparing each node's value with the sequence value at the corresponding index. It systematically explores all viable paths down the tree to determine if a valid sequence matching the given array exists.

Example Walkthrough

To illustrate the solution approach, let's consider a small binary tree example and a given sequence to match:

```
Tree structure:
     1
    / \
   0   2
  / \
 3   4

Sequence to match: [1, 0, 3]
```

We want to determine if this sequence can be formed by traversing from the root to a leaf. Now let's perform a walk through using the `dfs` function step by step.

1. We start at the root node with the value `1`. Since `arr[0]` is `1`, the first element of the sequence matches the root's value. We proceed to call the `dfs` function recursively on both children with the next index (`u + 1`), which is `1`.

2. First, we consider the left child of the root with the value `0`. At this point, `arr[1]` is `0`, which matches the current node's value. We again proceed with the `dfs` function recursively on this node's children.

3. For the left child of the node `0`, we have the leaf node with the value `3`. The sequence index we're looking for is `u + 1 = 2`, which gives us `arr[2] = 3`, and it matches the leaf node's value. We check if this is the end of the sequence (since `u == len(arr) - 1`) and if the current node is a leaf (no children), which is true.

4. Considering the conditions are satisfied, the sequence `[1, 0, 3]` is found to be valid, and the `dfs` function returns `True`.

Since the `dfs` function has found a valid path that matches the given sequence, the entire `isValidSequence` method would return `True`. This confirms that the sequence `[1, 0, 3]` is indeed a root-to-leaf path in the given binary tree.

Python Solution

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

class Solution:
    def isValidSequence(self, root: TreeNode, sequence: List[int]) -> bool:
        # Function to perform depth-first search on the tree
        def dfs(node, index):
            # Ensure that the current node exists and the value matches the sequence at the current index
            if not node or node.val != sequence[index]:
                return False

            # If we are at the last index, check if it's a leaf node
            if index == len(sequence) - 1:
                return node.left is None and node.right is None

            # Otherwise, move to the left and right children and increment the index
            return dfs(node.left, index + 1) or dfs(node.right, index + 1)

        # Begin depth-first search from the root node starting at index 0 of the sequence
        return dfs(root, 0)

```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int value; // Node's value
    TreeNode left; // Reference to the left child
    TreeNode right; // Reference to the right child

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

class Solution {
    private int[] sequence; // Array to hold the sequence to validate against the tree nodes

    // Entry method to start the process of validating sequence in the tree
    public boolean isValidSequence(TreeNode root, int[] sequence) {
        this.sequence = sequence; // Assign the sequence to the instance variable
        return isPathExist(root, 0); // Begin depth-first search from the root of the tree
    }

    // Helper method for depth-first search to validate the sequence
    private boolean isPathExist(TreeNode node, int index) {
        // If the current node is null or the value does not match the sequence, return false.
        if (node == null || node.value != sequence[index]) {
            return false;
        }
        // Check if this node is a leaf and it's the last element in the sequence
        if (index == sequence.length - 1) {
            return node.left == null && node.right == null;
        }
        // Move to the next index in the sequence and search in both left and right subtrees
        return isPathExist(node.left, index + 1) || isPathExist(node.right, index + 1);
    }
}

```

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 determine if a given sequence is a valid path from root to a leaf in the binary tree
    bool isValidSequence(TreeNode* root, vector<int>& sequence) {
        // Define a lambda function for depth-first search
        function<bool(TreeNode*, int)> depthFirstSearch = [&](TreeNode* node, int index) -> bool {
            // Return false if current node is null or value does not match the sequence at current index
            if (!node || node->val != sequence[index]) return false;

            // If we are at the last element of the sequence, we should also be at a leaf node
            if (index == sequence.size() - 1) return !(node->left) && !(node->right);

            // Continue the search in the left or right child, incrementing the sequence index
            return depthFirstSearch(node->left, index + 1) || depthFirstSearch(node->right, index + 1);
        };

        // Initial call to depthFirstSearch starting with the root node and the first element of the sequence
        return depthFirstSearch(root, 0);
    }
};

```

Typescript Solution

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

// Function to create a new TreeNode with default values
function createTreeNode(value: number = 0, left: TreeNode | null = null, right: TreeNode | null = null): TreeNode {
  return {
    val: value,
    left: left,
    right: right
  };
}

// Type for the depth-first search function
type DepthFirstSearchFunction = (node: TreeNode | null, index: number) => boolean;

// Function to determine if a given sequence is a valid path from root to a leaf in the binary tree
function isValidSequence(root: TreeNode | null, sequence: number[]): boolean {
  // Define a depth-first search (DFS) function
  const depthFirstSearch: DepthFirstSearchFunction = (node, index) => {
    // Return false if current node is null or the value does not match the sequence at the current index
    if (!node || node.val !== sequence[index]) {
      return false;
    }

    // If we are at the last element of the sequence, we should also be at a leaf node
    if (index === sequence.length - 1) {
      return !node.left && !node.right;
    }

    // Continue the search in the left or right child, incrementing the sequence index
    return depthFirstSearch(node.left, index + 1) || depthFirstSearch(node.right, index + 1);
  };

  // Initial call to DFS starting with the root node and the first element of the sequence
  return depthFirstSearch(root, 0);
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(N)`, where `N` is the number of nodes in the binary tree. This is because in the worst-case scenario, we will have to visit every node to check if it's a part of the valid sequence. The `dfs` function is called for every node, but never more than once for a node, so we visit each node a maximum of one time.

Space Complexity

The space complexity of the code is `O(H)`, where `H` is the height of the binary tree. This is due to the recursive nature of the depth-first search algorithm, which will use stack space for each recursive call. In the worst case (completely unbalanced tree), this could be `O(N)`, if the tree takes the form of a linked list (essentially, each node only has one child). However, in the best case (completely balanced tree), the height of the tree is `log(N)`, and thus the space complexity would be `O(log(N))`.