1367. Linked List in Binary Tree

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

Problem Description

The LeetCode problem presents a scenario where we have two data structures: a binary tree and a linked list. We are asked to determine if the linked list is represented by any downward path in the binary tree. A downward path in the binary tree is defined as a path that starts at any node and extends to subsequent child nodes, going downwards. The path does not need to start at the root of the binary tree. The problem requires us to create a function returning a boolean value: `True` if the linked list can be found as a downward path in the binary tree, and `False` if it cannot.

Intuition

To solve this problem, we need to traverse the binary tree and simultaneously compare the values with the nodes in the linked list. As the linked list must be followed in order, we cannot skip nodes in the linked list or in the binary tree's path. The nature of tree traversal and the need to compare it with the linked list suggest a depth-first search (DFS) approach, which allows us to explore a complete path from a node downward before moving to a sibling node.

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

1. At each node in the binary tree, we initiate a DFS to check if starting from this node, we can find a path matching the linked list.
2. In the DFS, we compare nodes of the binary tree with the linked list in order:
   - If the current binary tree node is `None`, it means we've reached the bottom of a path without a mismatch; hence, if we also reached the end of the linked list (`head is None`), we've found a matching path.
   - If the binary tree node's value does not match the current list node's value, the current path is immediately invalid.
   - If the values match, we proceed with the next node in both the linked list and continue exploring both subtrees (left and right) of the current binary tree node.
3. If a full linked list traversal is matched by a downward path in the binary tree (DFS returns `True`), we've found a corresponding path, and we return `True`.
4. If not, we continue checking from the left and right children of the current binary tree node, because the path could start from either subtree.

The proposed solution is recursive in nature. It makes use of a helper function `dfs` to handle the downward path checking and calls `dfs` repeatedly for each node in the binary tree until a match is found or the entire tree is traversed without finding a corresponding path.

Solution Approach

The provided solution is a recursive algorithm that uses depth-first search (DFS) to solve the problem. Let's break down the important parts of the solution and explain the mechanics step by step:

1. **Definition of DFS Helper Function**: The `dfs` function is a tailored depth-first search that takes two parameters—the current node of the linked list (`head`) and the current node of the binary tree (`root`). Its purpose is to check if starting from the current binary tree node, a downward path exists that corresponds to the linked list.

2. **Base Cases in DFS**:

   - If the current `head` of the linked list is `None`, this means the linked list has been completely traversed, and thus, a matching path in the tree has been found. The function returns `True` in this case.
   - If the current `root` of the binary tree is `None` or the value of the `root` does not match the value of `head`, the path being checked does not match the linked list, and the function returns `False`.
3. **Recursive Step in DFS**: If the current `head` and `root` values match, the algorithm proceeds by checking both the left and right children of the current binary tree node with the next node in the linked list. The dfs function is called recursively for both `root.left` and `root.right` with `head.next`. This recursion propagates downwards through the binary tree, building the downward path. Additionally, a logical `OR` is used between the recursive calls to `root.left` and `root.right`, meaning if either subtree contains a matching path, the function will return `True`.

4. **Primary Function Flow**:

   - If the `root` of the binary tree is `None`, then there cannot be any path that matches the linked list; therefore, it returns `False`.
   - The primary function, `isSubPath`, initializes the process by calling the `dfs` function with the `head` of the linked list and `root` of the binary tree.
   - It also calls itself recursively for both `root.left` and `root.right` in a similar logical `OR` pattern. This branching out ensures that the algorithm checks all possible starting points in the binary tree for the linked list sequence.

The use of recursion for both the DFS and the overall traversal of the binary tree nodes enables the algorithm to comprehensively search for the linked list pattern within all downward paths of the tree. This approach is efficient in finding the solution, but it may not be optimal in terms of time complexity due to the multiple recursive calls and the potential for repeating work on overlapping subtrees. However, it is a clear and concise way to solve this particular problem.

Here is the critical section of the code implemented in Python, highlighting the recursive nature of the solution:

```python
class Solution:
    def isSubPath(self, head: Optional[ListNode], root: Optional[TreeNode]) -> bool:
        def dfs(head, root):
            if head is None:
                return True
            if root is None or root.val != head.val:
                return False
            return dfs(head.next, root.left) or dfs(head.next, root.right)

        if root is None:
            return False
        return (
            dfs(head, root)
            or self.isSubPath(head, root.left)
            or self.isSubPath(head, root.right)
        )
```

In summary, the solution leverages recursive DFS to traverse the binary tree and matches each path with the linked list from its starting node to the end of the list to determine if a corresponding downward path exists.

Example Walkthrough

Let's illustrate the solution approach with a simple example.

Suppose we have the following binary tree:

```
    1
   / \
  4   4
 /     \
2       6
       /
      2
```

And the given linked list is `4 -> 2`.

Now, let's walk through the solution:

Step 1: Start with the root node of the binary tree

We start with the root, which is node 1, and we compare it to the head of the linked list, which is 4. They do not match, so we proceed to check the left and right children of node 1.

Step 2: Move to the left child of the root node

The left child is node 4, which matches the head of the linked list. Now we invoke the DFS helper function to check for a downward path.

DFS Call for Node 4 (Left Child of Root)

1. We check node 4 of the binary tree against the head of the linked list, which is also 4, and there's a match.
2. We move to the next element in the linked list (which is 2) and to the left and right children (which are node 2 and `None`, respectively).
3. Since the left child (node 2) matches the next list element, we continue the DFS call with the next element of the linked list (`None`, since we've reached the end) and the left child of node 2 (which is `None`).

At this point, the linked list has been completely matched with a downward path in the binary tree, and the DFS function returns `True`.

The `isSubPath` function would return `True`, signalling that the linked list 4 → 2 is indeed a downward path in the binary tree.

If No Match Was Found

If the first DFS call did not return `True`, we would continue with other children of the binary tree nodes. For instance, we would also check the right child (node 4) of the root, which again matches the head of the linked list and perform DFS to explore its downward paths.

In this case, although there's another node 4, its children nodes do not continue the sequence with node 2, so the downward path that corresponds to the linked list does not exist on this side of the binary tree.

Following this approach recursively, the solution checks all potential starting nodes in the binary tree and their respective downward paths until a match is found or all possibilities are exhausted.

Python Solution

```python
class ListNode:
    def __init__(self, value=0, next_node=None):
        self.value = value
        self.next_node = next_node

class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class Solution:
    def isSubPath(self, head: Optional[ListNode], root: Optional[TreeNode]) -> bool:
        # Helper function to perform DFS on the binary tree
        def dfs(linked_list_node, tree_node):
            # If linked list is fully traversed, a subpath exists
            if linked_list_node is None:
                return True
            # If tree node is None or values don't match, return False
            if tree_node is None or tree_node.value != linked_list_node.value:
                return False
            # Move to next of linked list and left or right child of the tree node
            return dfs(linked_list_node.next_node, tree_node.left) or dfs(linked_list_node.next_node, tree_node.right)

        # Base case for when the binary tree is empty
        if root is None:
            return False
        # Starting from this root, check for subpath or proceed to its left/right child and repeat
        return (
            dfs(head, root)
            or self.isSubPath(head, root.left)
            or self.isSubPath(head, root.right)
        )

# Note: Optional is not imported in this snippet. To use it, add: from typing import Optional
```

Java Solution

```java
// Definition for singly-linked list.
class ListNode {
    int val;
    ListNode next; // Reference to the next node in the list

    ListNode() {}

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

    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left; // Reference to the left child node
    TreeNode right; // Reference to the right child node

    TreeNode() {}

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

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

class Solution {
    // Checks if there's a subpath in a binary tree that matches the values in a linked list.
    public boolean isSubPath(ListNode head, TreeNode root) {
        // If the binary tree is empty, there can't be a subpath.
        if (root == null) {
            return false;
        }

        // Check the current path, or traverse left and right subtree to find the subpath.
        return dfs(head, root) || isSubPath(head, root.left) || isSubPath(head, root.right);
    }

    // Helper method using DFS to match the linked list with the path in the binary tree.
    private boolean dfs(ListNode head, TreeNode root) {
        // If we've successfully reached the end of the linked list, the subpath is found.
        if (head == null) {
            return true;
        }

        // If the binary tree node is null or values do not match, this path isn't valid.
        if (root == null || head.val != root.val) {
            return false;
        }

        // Continue onto the left or right subtree to find the matching subpath.
        return dfs(head.next, root.left) || dfs(head.next, root.right);
    }
}
```

C++ Solution

```cpp
/**
 * Definition for singly-linked list.
 */
struct ListNode {
    int val;
    ListNode *next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode *next) : val(x), next(next) {}
};

/**
 * 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:
    // Checks if the linked list is a subpath of the binary tree
    bool isSubPath(ListNode* head, TreeNode* root) {
        // If the tree node is null, the list cannot be a subpath
        if (!root) {
            return false;
        }
        // Check if the list can start from the current tree node or any subtree
        return depthFirstSearch(head, root) || isSubPath(head, root->left) || isSubPath(head, root->right);
    }

    // Helper function to perform depth-first search starting from a tree node
    bool depthFirstSearch(ListNode* head, TreeNode* node) {
        // If list node is null, we've reached the end of the list successfully
        if (!head) {
            return true;
        }
        // If tree node is null or values don't match, it's not a match
        if (!node || head->val != node->val) {
            return false;
        }
        // Continue searching the rest of the list in the left and right subtrees
        return depthFirstSearch(head->next, node->left) || depthFirstSearch(head->next, node->right);
    }
};
```

Typescript Solution

```typescript
// This TypeScript code defines two utility functions to determine whether
// a linked list is a subpath of a binary tree.

/**
 * This function performs a deep-first search to check if the
 * current linked list starting at 'head' is a subpath of the binary tree rooted at 'node'.
 * @param {ListNode | null} head - The current node in the linked list.
 * @param {TreeNode | null} node - The current node in the binary tree.
 * @returns {boolean} - Returns true if the list is a subpath from the current tree node, false otherwise.
 */
const isSubPathFromNode = (head: ListNode | null, node: TreeNode | null): boolean => {
    // If the linked list is exhausted, it means we've found a subpath.
    if (head === null) {
        return true;
    }
    // If the binary tree node is null or the values do not match,
    // the current path does not match the linked list.
    if (node === null || head.val !== node.val) {
        return false;
    }
    // Continue the search deeply in both left and right directions of the tree.
    return isSubPathFromNode(head.next, node.left) || isSubPathFromNode(head.next, node.right);
};

/**
 * This function checks if the linked list starting at 'head' is a subpath of the binary tree rooted at 'root'.
 * It does so by traversing the tree nodes and, for each node, attempting to match the linked list from that starting point.
 * @param {ListNode | null} head - The head of the linked list.
 * @param {TreeNode | null} root - The root of the binary tree.
 * @returns {boolean} - Returns true if the linked list is a subpath of the tree, false otherwise.
 */
const isSubPathOfTree = (head: ListNode | null, root: TreeNode | null): boolean => {
    // If the root of the tree is null, the linked list cannot be a subpath.
    if (root === null) {
        return false;
    }
    // Check if the linked list is a subpath from the current root node or any of its subtrees.
    return isSubPathFromNode(head, root) || isSubPathOfTree(head, root.left) || isSubPathOfTree(head, root.right);
};

// Note that the two methods 'isSubPathFromNode' and 'isSubPathOfTree' are to be used internally,
// and 'isSubPathOfTree' is the entry point function equivalent to the 'isSubPath' in the original code.
// The 'isSubPath' name was changed only because the naming convention generally favors descriptive names.
```

Time and Space Complexity

The given code defines a function that checks whether a linked list is a subpath of a binary tree. The complexity is calculated based on two main operations: the `dfs` function that performs a deep search to compare a path in the tree with the linked list, and the recursive call `isSubPath` to move down the binary tree.

Time Complexity:

The time complexity of the `dfs` function is $O(N)$ where `N` is the number of nodes in the tree. This is because, in the worst case, it might have to compare every node of the tree with the head of the linked list. Additionally, `isSubPath` is called for each node of the tree. Therefore, in the worst case, the time complexity becomes $O(N * L)$ where `N` is the number of nodes in the binary tree and `L` is the length of the linked list.

Space Complexity:

The space complexity is determined by the maximum depth of the recursive call stack, which would also be proportional to the height of the tree in the worst case. Thus, the space complexity is $O(H)$ where `H` is the height of the binary tree. For a skewed tree (one that resembles a linked list), the height of the tree can be `N`, making the space complexity $O(N)$ in the worst case.