671. Second Minimum Node In a Binary Tree

Easy Tree Depth-First Search Binary Tree Leetcode Link

Problem Description

The problem presents a special binary tree where every node has either two or zero child nodes (sub-nodes). The key property of this tree is that the value of a non-leaf node is the minimum of the values of its two children. The task is to find the second smallest value in the entire tree. If all the node values are the same or there is no second minimum value, the function should return -1.

Intuition

To find the second minimum value, we need to traverse the tree and keep track of the values we encounter. Since the smallest value in the tree is at the root (due to the property of the tree), we can ignore it in our search for the second minimum. We're interested in the smallest value that is larger than the root's value.

Here's the intuition for the provided solution approach:

1. Start a Depth-First Search (DFS) traversal from the root. Since the problem only needs us to find a value, DFS is preferred over Breadth-First Search (BFS) as it's simpler to implement and can short-circuit faster.
2. Each time we visit a node, we check if it has a value greater than the value of the root (since the root's value is the minimum of the entire tree).
3. If the current node's value is greater than the root's value, it is a candidate for being the second minimum value. If we haven't found a candidate before (`ans` is -1), we set our answer to the current node's value. If we have found candidates before, we update the answer to be the minimum of the current node's value and the current answer.
4. We keep updating the answer as we find new candidates during our traversal. Once the traversal is complete, `ans` will hold the second smallest value if it exists; otherwise, it remains as -1.
5. The nonlocal keyword is used to modify the `ans` and `v` variables that are defined outside the nested `dfs` function.

The reason we can't just compare all the node values directly is because of the special property of this kind of tree. The second smallest value must be the value of a node's child, where the node itself has the smallest value. So, we don't need to consider all nodes but can focus on leaf nodes and nodes one level above leaf nodes.

Solution Approach

The solution uses a recursion-based DFS (Depth-First Search) to traverse the tree. The goal is to maintain the second minimum value as we go through each node. Let's discuss the implementation details using the reference solution approach provided above:

1. **TreeNode Class**: Before diving into the algorithm, it's important to understand that the input tree is composed of instances of `TreeNode`, where each `TreeNode` object represents a node in the binary tree with a `val`, a pointer `left` to a left subtree, and a pointer `right` to a right subtree.
2. **Depth-First Search (DFS)**: We perform DFS starting from the root node. The DFS is realized through the function `dfs`.

   ```python
   def dfs(root):
       if root:
           dfs(root.left)
           dfs(root.right)
           nonlocal ans, v
           if root.val > v:
               ans = root.val if ans == -1 else min(ans, root.val)
   ```

   This algorithm recursively explores both sub-trees (`root.left` and `root.right`) before dealing with the current `root` node, which is characteristic of the post-order traversal pattern of DFS.
3. **Post-Order Traversal**: We use the post-order traversal here because we want to visit each child before dealing with the parent node. This allows us to find the candidate for the second minimum value from the leaf upwards, respecting the special property of this binary tree (the node's value is the minimum of its children).
4. **Tracking the Second Minimum**: We maintain two variables, `ans` and `v`.
   - `v` is assigned the value of the root and represents the smallest value in the tree.
   - `ans` is used to track the second smallest value, initialized as -1, which also represents the condition when there's no second minimum value.
5. **Local vs. Global Scope**: The use of the keyword `nonlocal` is significant. Since `ans` and `v` are modified within the `dfs` function, and Python functions create a new scope, we must declare these variables `nonlocal` to indicate that they are not local to `dfs`, allowing us to modify the outer scope variables `ans` and `v`.

By the end of DFS execution, `ans` will contain the second minimum value or -1 if there's no such value. Then, the main function `findSecondMinimumValue` just needs to return `ans`.

The given solution efficiently leverages recursion for tree traversal and runs in O(n) time where n is the number of nodes in the tree, since it visits each node exactly once. The space complexity is also O(n) due to the recursion call stack, which can go as deep as the height of the tree in the worst-case scenario (although for this special type of binary tree, that would mean a very unbalanced tree).

Example Walkthrough

Let's go through a small example using the solution approach. Consider a binary tree:

```
    2
   / \
  2   5
     / \
    5   7
```

We need to find the second minimum value in the special binary tree. In the given binary tree, the root node has the minimum value, which is `2`.

- We start our DFS traversal from the root node `(2)`. We set `v` as the value of the root node, which is `2`, and `ans` is set to `-1`.
- We visit the left child `(2)`. As its value is not greater than `v`, we continue our traversal.
- We then proceed to the right child `(5)`. Its value is greater than `v`, so `ans` is updated from `-1` to `5`.
- Next, we visit the left child of the right child, which again is `(5)`. Since `5` is already our current `ans`, and it's not less than the current `ans`, we don't update `ans`.
- The last node we visit is the right child of the right child `(7)`. The value of this node is greater than `v` and greater than our current `ans` of `5`, so we do not update `ans`.

After completing the DFS traversal, `ans` contains the value `5`, which is the second minimum value in the tree. If there were no value greater than `2` in the example tree, `ans` would have remained as `-1`. This corresponds to the situation where either all node values are the same or there's no second minimum value.

By performing DFS, we minimized unnecessary checks and efficiently found the second minimum value, if it existed. This approach ensures that every node is visited only once, leading to an O(n) time complexity, where `n` is the number of nodes in the tree. The space complexity is also O(n), which is attributed to the recursion call stack.

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 findSecondMinimumValue(self, root: Optional[TreeNode]) -> int:
        # Initialize variables for the final answer and the root value.
        # The root value is the minimum value by definition of the problem.
        self.second_minimum = -1
        self.root_value = root.val

        # Define a Depth-First Search (DFS) recursive function.
        def dfs(node):
            if node:
                # Recurse left sub-tree.
                dfs(node.left)
                # Recurse right sub-tree.
                dfs(node.right)
                # If the current node's value is greater than the root's value,
                # it's a candidate for the second minimum.
                if node.val > self.root_value:
                    # If the second minimum hasn't been found yet, use this value,
                    # else update it only if we find a smaller value.
                    self.second_minimum = min(self.second_minimum, node.val) if self.second_minimum != -1 else node.val

        # Perform DFS starting from the root.
        dfs(root)

        # Return the second minimum value found; -1 if it doesn't exist.
        return self.second_minimum

```

Java Solution

```java
// Definition for a binary tree node.
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;
    }
}

class Solution {
    private int secondMinValue = -1; // Variable to store the second minimum value

    public int findSecondMinimumValue(TreeNode root) {
        // Start depth-first search with the root node's value as the initial value
        depthFirstSearch(root, root.val);
        // After DFS, return the second minimum value found
        return secondMinValue;
    }

    // Helper method to perform a depth-first search on the tree
    private void depthFirstSearch(TreeNode node, int firstMinValue) {
        // If the current node is not null, proceed with DFS
        if (node != null) {
            // Recursively traverse the left subtree
            depthFirstSearch(node.left, firstMinValue);
            // Recursively traverse the right subtree
            depthFirstSearch(node.right, firstMinValue);
            // Check if node's value is greater than first minimum value
            // and update the second minimum value accordingly
            if (node.val > firstMinValue) {
                // If secondMinValue hasn't been updated yet, use the current node's value
                // Else, update it to be the minimum of the current node's value and the existing secondMinValue
                secondMinValue = secondMinValue == -1 ? node.val : Math.min(secondMinValue, node.val);
            }
        }
    }
}

```

C++ Solution

```cpp
#include <algorithm> // For using the min() function

// 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:
    int secondMinimumValue = -1; // initialized with -1 to indicate no second minimum found

    int findSecondMinimumValue(TreeNode* root) {
        // Start DFS with root value as reference
        depthFirstSearch(root, root->val);
        // Return the second minimum value found
        return secondMinimumValue;
    }

    // Perform a depth-first search to find the second minimum value
    void depthFirstSearch(TreeNode* node, int firstMinValue) {
        // Base condition: If node is nullptr, return immediately
        if (!node) return;

        // Recursively search left subtree
        depthFirstSearch(node->left, firstMinValue);
        // Recursively search right subtree
        depthFirstSearch(node->right, firstMinValue);

        // Check if the current node's value is greater than the first minimum value
        if (node->val > firstMinValue) {
            // Update second minimum value if it's either not set or found a smaller value
            secondMinimumValue = (secondMinimumValue == -1) ? node->val : std::min(secondMinimumValue, node->val);
        }
    }
};

```

Typescript Solution

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

/**
 * Finds the second minimum value in a special binary tree,
 * where each node in this tree has exactly two or zero sub-node.
 * If the second minimum value does not exist, return -1.
 *
 * @param {TreeNode | null} root - The root node of the binary tree.
 * @return {number} The second minimum value in the given binary tree.
 */
const findSecondMinimumValue = (root: TreeNode | null): number => {
    // Initialize the answer to -1, assuming the second minimum value might not exist.
    let answer: number = -1;

    // Store the value of the root, which is the minimum value in the tree.
    const rootValue: number = root.val;

    // Define a depth-first search function to traverse the tree.
    const dfs = (node: TreeNode | null): void => {
        // If the node is null, we've reached the end of a branch and should return.
        if (!node) {
            return;
        }

        // Continue the DFS on the left and right children.
        dfs(node.left);
        dfs(node.right);

        // If the node's value is greater than that of the root (minimum value)
        // and if this value is smaller than the current answer or if the answer
        // is still set to the initial -1, then we update the answer.
        if (node.val > rootValue) {
            if (answer === -1 || node.val < answer) {
                answer = node.val;
            }
        }
    };

    // Start the DFS traversal with the root node.
    dfs(root);

    // Return the answer, which is either the second minimum value or -1.
    return answer;
};

// This part of the code would be used to call the function and pass the tree root.
// Example usage:
// const root: TreeNode = { val: 2, left: { val: 2, left: null, right: null }, right: { val: 5, left: { val: 5, left: null, right: nu
// console.log(findSecondMinimumValue(root));

```

Time and Space Complexity

The provided code implements a Depth-First Search (DFS) to find the second minimum value in a binary tree. Here's an analysis of its time and space complexity:

Time Complexity

The time complexity of the DFS is `O(N)`, where `N` is the number of nodes in the binary tree. In the worst case, the algorithm visits every node exactly once to determine the second smallest value.

Space Complexity

The space complexity is `O(H)`, where `H` is the height of the tree. This space is required for the call stack during the recursion, which in the worst case is equivalent to the height of the tree. For a balanced tree, it would be `O(log N)`, but in the worst case of a skewed tree, it could be `O(N)`.