501. Find Mode in Binary Search Tree

Easy Tree Depth-First Search Binary Search Tree Binary Tree Leetcode Link

Problem Description

The problem is about finding the most frequently occurring elements in a Binary Search Tree (BST). The challenge is that a BST may contain duplicates and the goal is to identify the mode(s), which are the values that appear most frequently. As a reminder, a BST has the property where each node's left child has a value less than or equal to its own value, while the right child has a value greater than or equal to its own value. This property must hold true for every node's left and right subtrees as well.

The output should be a list of the modes. If there's more than one mode, they can be returned in any order. What makes this interesting is that we must traverse the tree and keep track of the frequency of each element without knowing beforehand which values appear in the BST.

Intuition

To solve this problem, one effective approach is to perform an in-order traversal of the BST. An in-order traversal is a type of depth-first traversal that visits the left subtree, then the root node, and finally the right subtree. This traversal order guarantees that the values are accessed in a sorted manner because of the BST property.

As we perform the traversal, we can track the current value, its count, and compare it with the maximum frequency (or mode) encountered so far. There are a few edge cases to consider:

- If the current value is the same as the previous value we saw, we increment the count for this value.
- If it's a new value (not the same as the previous one), we reset the count to 1.
- If the count for the current value is greater than the current maximum frequency, we update the maximum frequency and reset the list of modes to now only include this new value.
- If the count matches the current maximum frequency, we add the current value to the list of modes.

A nonlocal variable is used to maintain state between recursive calls. Variables like `mx` (short for 'maximum'), `prev` (short for 'previous value'), `ans` (short for 'answer'), and `cnt` (short for 'count') are updated as the in-order traversal proceeds. After the traversal is complete, the `ans` list will contain the mode(s) of the BST.

This approach efficiently computes the mode(s) using the BST's inherent properties, all without needing additional data structures to track frequency of all elements, thus saving space.

Solution Approach

The solution uses a Depth-First Search (DFS) in-order traversal algorithm to walk through the tree. Since it's a BST, an in-order traversal will access the nodes in ascending order. This property is critical because it allows us to track the mode(s) without the need for additional data structures like a hash table to count the frequencies of each value.

In the Python solution, a nested function `dfs` is defined, capturing the variables `mx`, `prev`, `ans`, and `cnt` from the enclosing `findMode` function scope using the `nonlocal` keyword. These captured variables serve as state throughout the traversal:

- `mx`: This keeps track of the maximum frequency of any value encountered so far.
- `prev`: This holds the previous value processed in the in-order traversal. It's used to compare with the current node's value to determine if the count should be increased or reset.
- `cnt`: This records the current count of consecutive nodes with the same value.
- `ans`: This list stores the mode(s) — the values with the highest frequency seen so far.

Here is a step-by-step explanation of the function `dfs` which is called with the `root` of the BST:

- The base case checks if the current node `root` is `None`, and if it is, the function returns because there's nothing to process.
- Recursive calls are made to visit the left subtree: `dfs(root.left)`.
- When control returns from the left subtree, the current node is processed. If `prev` is the same as `root.val`, `cnt` is incremented. Otherwise, `cnt` is set to 1 since we're seeing this value for the first time.
- We then check if `cnt` is greater than `mx`. If it is, this is the new maximum frequency, and so `mx` is updated to `cnt`, and `ans` is set to a list containing just `root.val`.
- If `cnt` equals `mx`, we append `root.val` to `ans` because it's a mode with the same frequency as the current maximum.
- `prev` is updated to the current node's value: `prev = root.val`.
- Recursive calls are made to visit the right subtree: `dfs(root.right)`.

After the `dfs` function has completed the in-order traversal, the `ans` list will have the mode(s). The Solution class's `findMode` function then returns `ans`. The result is that all modes in the BST are found in a single pass through the tree, which is an efficient use of time and space.

Example Walkthrough

Let's illustrate the solution with a small example. Suppose our BST looks like this:

```
  2
 / \
1   2
```

The BST contains duplicates, and we want to find the mode. In this case, the mode is `2`, since it occurs more frequently than any other number.

Now, let's walk through how the solution processes this BST:

1. We start with an in-order traversal of the BST, visiting the left child first, then the current node, and finally the right child.
2. Begin the traversal. The `dfs` function runs:
   - The `dfs` function visits the leftmost node which is `1`.
   - Since this is our first visit, we compare `prev` and `root.val`. They are different (`prev` is `None` at the start, and `root.val` is `1`), so we set `cnt` to `1`.
   - As this is the first value we've seen, `mx` is also set to `1`, and `ans` is updated to `[1]` because `cnt` equals `mx` at this point.
   - We update `prev` to `1` now that we have processed this node.
3. Up next, we backtrack and visit the root node which has a value `2`. Now, `prev` is `1`:
   - `prev` (1) does not match `root.val` (2), so we set `cnt` to `1`.
   - Currently, `cnt` does not exceed `mx`, so `ans` remains the same.
   - We update `prev` to `2`.
4. Finally, we visit the right child which also has a value of `2`:
   - This time, `prev` matches `root.val`, so we increment `cnt` to `2`.
   - Now `cnt` (2) is greater than `mx` (1), so we have a new mode. Therefore, we update `mx` to `2`, and reset `ans` to `[2]`, the new mode.
5. The traversal is now complete, and since there are no more nodes to visit, `ans` contains the mode(s).

In this example, the output is `[2]`, which is the correct mode of the given BST.

This walkthrough demonstrates the effectiveness of the algorithm as it leverages the BST property to perform an in-order traversal and identify the most frequently occurring element with the need for any additional data structures for frequency counting, making it space efficient.

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 findMode(self, root: TreeNode) -> List[int]:
        # Helper function to perform in-order traversal
        def in_order_traversal(node):
            if node is None:
                return
            nonlocal max_count, previous_value, modes, current_count
            # Traverse the left subtree
            in_order_traversal(node.left)

            # Update the count for the current value
            # If the current value matches the previous value, increment the count;
            # Otherwise, reset the count to 1 for the new value
            current_count = current_count + 1 if previous_value == node.val else 1

            # If the current count exceeds the max_count found so far,
            # update max_count and reset modes with the current value
            if current_count > max_count:
                modes = [node.val]
                max_count = current_count
            # If the current count matches the max_count,
            # add the current value to the list of modes
            elif current_count == max_count:
                modes.append(node.val)

            # Update the previous_value to the current node's value
            previous_value = node.val

            # Traverse the right subtree
            in_order_traversal(node.right)

        # Initialize variables
        previous_value = None  # Stores the previously seen value
        max_count = current_count = 0  # max_count is the maximum frequency of any value,
                                       # current_count is the frequency of the current node value
        modes = []  # Stores the modes

        # Perform the in-order traversal starting from the root
        in_order_traversal(root)

        # Return the list of modes
        return modes

```

Java Solution

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

// 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 maxFrequency;
    private int currentCount;
    private TreeNode previousNode;
    private List<Integer> modes;

    // Finds the mode(s) in a binary search tree.
    public int[] findMode(TreeNode root) {
        modes = new ArrayList<>();
        inorderTraversal(root);
        int[] result = new int[modes.size()];
        for (int i = 0; i < modes.size(); ++i) {
            result[i] = modes.get(i);
        }
        return result;
    }

    // Helper function to perform inorder traversal of the binary tree.
    private void inorderTraversal(TreeNode node) {
        if (node == null) {
            return;
        }

        // Traverse the left subtree
        inorderTraversal(node.left);

        // Update the current count of the current value
        // If it's the same as the previous node's value, increment the count; otherwise, reset it to 1
        currentCount = (previousNode != null && previousNode.val == node.val) ? currentCount + 1 : 1;

        // If current count is greater than max frequency, update max frequency and reset modes list
        if (currentCount > maxFrequency) {
            maxFrequency = currentCount;
            modes = new ArrayList<>(Arrays.asList(node.val));
        }
        // If current count is equal to max frequency, add the current value to modes list
        else if (currentCount == maxFrequency) {
            modes.add(node.val);
        }

        // Update previousNode to the current node before traversing right subtree
        previousNode = node;

        // Traverse the right subtree
        inorderTraversal(node.right);
    }
}

```

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:
    TreeNode* previous;
    int maxFrequency, currentCount;
    vector<int> modes;

    /**
     * Finds the mode(s) in a binary search tree (BST), which are the values with the highest frequency of occurrence.
     *
     * @param root The root node of the BST.
     * @return A vector containing the mode(s) of the BST.
     */
    vector<int> findMode(TreeNode* root) {
        previous = nullptr;
        maxFrequency = 0;
        currentCount = 0;
        modes.clear();

        inOrderTraversal(root);

        return modes;
    }

    /**
     * In-order depth-first search (DFS) traversal of the BST to find the mode(s).
     *
     * @param node The current node being visited.
     */
    void inOrderTraversal(TreeNode* node) {
        if (!node) return; // Base case: if the current node is null, do nothing.

        // Traverse the left subtree.
        inOrderTraversal(node->left);

        // Process the current node.
        if (previous != nullptr && previous->val == node->val) {
            // If the value of the current node is the same as the previous node's value, increment the count.
            currentCount++;
        } else {
            // Otherwise, reset the count to 1.
            currentCount = 1;
        }

        if (currentCount > maxFrequency) {
            // If the current count exceeds the max frequency found so far,
            // clear the current modes and add the new mode.
            modes.clear();
            modes.push_back(node->val);
            maxFrequency = currentCount;
        } else if (currentCount == maxFrequency) {
            // If the current count equals the max frequency, add this node's value to the modes.
            modes.push_back(node->val);
        }

        // Update the previous node to the current one.
        previous = node;

        // Traverse the right subtree.
        inOrderTraversal(node->right);
    }
};

```

Typescript Solution

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

    constructor(val: number, left?: TreeNode, right?: TreeNode) {
        this.val = val === undefined ? 0 : val;
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}

let previous: TreeNode | null = null;
let maxFrequency: number = 0;
let currentCount: number = 0;
let modes: number[] = [];

/**
 * Finds the modes in a binary search tree (BST), which are the values with the highest frequency of occurrence.
 * @param root The root node of the BST.
 * @returns An array containing the modes of the BST.
 */
function findMode(root: TreeNode | null): number[] {
    previous = null;
    maxFrequency = 0;
    currentCount = 0;
    modes = [];

    inOrderTraversal(root);

    return modes;
}

/**
 * Performs an in-order traversal of the BST to find the modes.
 * @param node The current node in the traversal.
 */
function inOrderTraversal(node: TreeNode | null): void {
    if (!node) return; // Base case: If the current node is null, return.

    // Traverse the left subtree.
    inOrderTraversal(node.left);

    // Process the current node.
    if (previous && node.val === previous.val) {
        // If the current node value is the same as the previous node's value, increment the count.
        currentCount++;
    } else {
        // Otherwise, reset the count to 1.
        currentCount = 1;
    }

    if (currentCount > maxFrequency) {
        // If the current count is greater than the max frequency found so far,
        // clear the modes array and add the new mode.
        modes = [node.val];
        maxFrequency = currentCount;
    } else if (currentCount === maxFrequency) {
        // If the current count is equal to the max frequency, add this node's value to the modes.
        modes.push(node.val);
    }

    // Update the previous node to the current node for the next iteration.
    previous = node;

    // Traverse the right subtree.
    inOrderTraversal(node.right);
}

```

Time and Space Complexity

The time complexity of the provided code is `O(N)`, where `N` is the number of nodes in the binary tree. This is because the in-order traversal performed by the `dfs` function visits each node exactly once.

The space complexity of the code is `O(H)`, where `H` is the height of the binary tree, due to the call stack during the recursive traversals. In the worst-case scenario of a skewed tree, the height `H` can be `N`, leading to a space complexity of `O(N)`. Additionally, the space required to store the modes in `ans` does not exceed `O(N)` (in the case where all elements are the same and hence the mode list contains all elements), but it does not affect the worst-case space complexity which is dominated by the recursive stack.