653. Two Sum IV - Input is a BST

Easy Tree Depth-First Search Breadth-First Search Binary Search Tree Hash Table Two Pointers Binary Tree

Leetcode Link

Problem Description

The problem presents a Binary Search Tree (BST) and an integer `k`. The goal is to determine if there are any two distinct elements within the BST that add up to `k`. The BST is a tree structure where the value of each node is greater than all the values in its left subtree and less than all the values in its right subtree. This special property will play a key role in solving the problem efficiently. The solution should return `true` if such a pair exists or `false` otherwise.

Intuition

The intuition behind the solution comes from the properties of a BST. In a sorted array, a common approach to find if two numbers sum up to `k` is to use two pointers, one starting from the beginning and one from the end, moving towards each other until they find a pair that adds up to `k` or until they meet. We can use a similar approach with a BST using an inorder traversal, which will give us the elements in sorted order.

However, instead of using two pointers, we can use a HashSet (`vis` in the provided code) to store the values we've seen so far as we traverse the tree. While traversing, for each node with value `val`, we check if `k - val` is in our HashSet. If it is, we've found a pair that adds up to `k` and we can return `true`. If not, we add `val` to the HashSet and keep traversing. This approach leverages the BST property to eliminate the necessity of having the sorted list beforehand, and the HashSet to check for the complement in `O(1)` average time complexity, leading to an efficient overall solution.

The provided solution uses Depth-First Search (DFS) to traverse the BST. The recursive `dfs` function checks if the current node is `None` and, if not, proceeds to do the following:

1. Check if the complement of the current node's value (i.e., `k - root.val`) exists in the visited HashSet.
2. If the complement exists, return `true` as we found a valid pair.
3. If the complement does not exist, add the current node's value to the HashSet.
4. Recursively apply the same logic to the left and right children of the current node, returning `true` if either call finds a valid pair.

This solution effectively traverses the BST once, resulting in a time complexity of `O(n)` where `n` is the number of nodes in the BST, and a space complexity of `O(n)` for the storage in the HashSet.

Solution Approach

The solution implements a Depth-First Search (DFS) to traverse through the nodes of the Binary Search Tree. The DFS pattern is chosen because it allows us to systematically visit and check every node in the BST for the conditions stipulated in the problem. Let's walk through the implementation step by step:

- A helper function named `dfs` is defined, which is recursively called on each node in the tree. The purpose of this function is to both perform the search and to propagate the result of the search back up the call stack.
- Inside the `dfs` function, we first check if the current node, referred to as `root`, is `None`. If it is, it means we have reached the end of a branch, and we return `False` because we cannot find a pair in a nonexistent (null) sub-tree.

```
if root is None:
    return False
```

- Next, we check if the difference between `k` and the value of the current node `root.val` is in the set `vis`. If the difference is found, that means there is a value we have already visited in the tree that pairs with the current node's value to sum up to `k`. We then return `True` as we have found a valid pair.

```
if k - root.val in vis:
    return True
```

- If we haven't found a valid pair yet, we add the value of the current node to the `vis` set to keep a record of it. This is because it could potentially be a complement to a node we visit later in the traversal.

```
vis.add(root.val)
```

- Finally, we need to continue the traversal on both the left and right subtrees. We recursively call the `dfs` function on `root.left` and `root.right`. We leverage the `or` operator to propagate a `True` result back up if either of the recursive calls finds a valid pair.

```
return dfs(root.left) or dfs(root.right)
```

- The `vis` set is a crucial data structure utilized in this approach. Its main purpose is to keep track of all the values we have seen in the BST during the traversal. The set is chosen because it provides an average-time complexity of `O(1)` for both insertions and lookups, making our checks for the complement efficient.
- The `dfs` function is then called from within the `findTarget` function, which initializes the `vis` set and starts the process. The search is kicked off at the root of the tree.

```
vis = set()
return dfs(root)
```

By using the DFS traversal coupled with a set to record visited values, we efficiently search the BST for a pair of values that sum up to `k`. This approach effectively uses properties of the data structure (BST), an algorithm (DFS), and a pattern (using a set to track complement values) to solve the problem in `O(n)` time complexity, where `n` is the number of nodes in the tree, while maintaining `O(n)` space complexity for the set.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Assume we have a Binary Search Tree (BST) with the following structure:

```
    5
   / \
  3   6
 / \
2   4
```

And let's say we want to find if there are any two numbers within this BST that add up to `k = 9`.

Now, we will walk through the solution using the Depth-First Search (DFS) traversal and HashSet as described:

1. We start the DFS with the root of the BST, which is the node with value `5`. We initialize an empty HashSet `vis` to keep track of the values we've encountered.
2. Since root is not `None`, we check if `k - root.val` (which is `9 - 5 = 4`) is present in `vis`. It is not, so we proceed to add `5` to `vis`.
3. Now we apply DFS to the left child of `5`, which has the value `3`.
4. Again, we check if `k - 3 = 6` is in `vis`. It is not, so we add `3` to `vis`.
5. Next, we traverse to the left child of `3`, which has the value `2`. We check if `k - 2 = 7` is in `vis`. It is not, we add `2` to `vis`.
6. Since the node with value `2` has no children, we backtrack and apply DFS to the right child of `3`, which is `4`. Checking if `k - 4 = 5` is in `vis` and we find that it is, meaning we have discovered a valid pair (`4` and `5`) which adds up to `k = 9`.
7. Since we have found a valid pair, the function would return `True` immediately without continuing to traverse the rest of the tree.

Using this method, our search was efficient since we leveraged the BST's structure and avoided checking every possible combination of nodes. The HashSet allowed us to quickly assess whether the complement to the current node's value had already been visited. By using a recursive DFS approach, we were able to keep the implementation clear and systematic.

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  # Reference to the left child node
        self.right = right  # Reference to the right child node

class Solution:
    def findTarget(self, root: Optional[TreeNode], k: int) -> bool:
        """
        Determines if there are two elements in the BST such that their sum is equal to a given target.

        :param root: Root node of the binary search tree
        :param k: Target sum we want to find
        :return: True if such a pair is found, False otherwise
        """

        # Helper function to perform a depth-first search
        def depth_first_search(node):
            # Base case: if the node is None, return False
            if node is None:
                return False
            # If the complement of the current node's value (k - node.val) is in the visited set,
            # we've found a pair that sums up to k.
            if k - node.val in visited:
                return True
            # Add the current node's value to the visited set
            visited.add(node.val)
            # Recursively search the left and right subtrees
            return depth_first_search(node.left) or depth_first_search(node.right)

        visited = set()  # Initialize an empty set to store visited node values
        return depth_first_search(root)  # Begin DFS with the root node
```

Java Solution

```java
// Definition for a binary tree node.
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode() {} // Default constructor
    TreeNode(int val) { this.val = val; } // Constructor with node value
    TreeNode(int val, TreeNode left, TreeNode right) { // Constructor with node value and children
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    // A HashSet to keep track of the values we've visited
    private Set<Integer> visited = new HashSet<>();
    private int targetSum;

    // Public method to find if the tree has two elements such that their sum equals k
    public boolean findTarget(TreeNode root, int k) {
        this.targetSum = k;
        return dfs(root);
    }

    // Helper method that uses depth-first search to find if the property is satisfied
    private boolean dfs(TreeNode node) {
        if (node == null) {
            return false; // Base case: if the node is null, return false
        }
        if (visited.contains(targetSum - node.val)) {
            // If the complementary value is in 'visited', we found two numbers that add up to 'k'
            return true;
        }
        visited.add(node.val); // Add current node's value to the set
        // Recurse on left and right subtrees; return true if any subtree returns true
        return dfs(node.left) || dfs(node.right);
    }
}
```

C++ Solution

```cpp
#include <unordered_set>
#include <functional>

// 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 check if there exists two elements in the BST such that their sum is equal to the given value 'k'.
    bool findTarget(TreeNode* root, int k) {
        // Using an unordered set to store the values we've visited.
        std::unordered_set<int> visited;

        // Lambda function to perform Depth-First Search (DFS) on the binary tree.
        std::function<bool(TreeNode*)> dfs = [&](TreeNode* node) {
            // Base case: if the current node is null, we return false as we haven't found the pair.
            if (!node) {
                return false;
            }

            // If the complement of the current node's value (k - node's value) has already been visited, we've found a pair.
            if (visited.count(k - node->val)) {
                return true;
            }

            // Insert the current node's value into the visited set.
            visited.insert(node->val);
            // Continue the search on the left and right subtree.
            return dfs(node->left) || dfs(node->right);
        };

        // Start DFS from the root of the binary tree.
        return dfs(root);
    }
};
```

Typescript Solution

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

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

/**
 * Find if there exists any pair of nodes such that their sum equals k.
 *
 * @param {TreeNode | null} root - The root of the binary tree.
 * @param {number} k - The sum that we need to find.
 * @returns {boolean} - True if such a pair is found, else false.
 */
function findTarget(root: TreeNode | null, k: number): boolean {
    // Function to perform the depth-first search
    const depthFirstSearch = (node: TreeNode | null): boolean => {
        // If the node is null, return false.
        if (!node) {
            return false;
        }
        // If the complement of current node's value is seen before, return true.
        if (visitedValues.has(k - node.val)) {
            return true;
        }
        // Add the current node's value into the set.
        visitedValues.add(node.val);
        // Recur for left and right children.
        return depthFirstSearch(node.left) || depthFirstSearch(node.right);
    };

    // Set to store visited nodes values
    const visitedValues = new Set<number>();

    // Initiating the depth-first search
    return depthFirstSearch(root);
}
```

Time and Space Complexity

The given Python code is aimed at determining whether there are two nodes in a binary tree that add up to a given sum `k`. To achieve this, it performs a Depth-First Search (DFS) and uses a set `vis` to record the values it has encountered.

Here's the computational complexity breakdown:

Time complexity

The DFS traverses each node exactly once, irrespective of the tree structure, because we are looking at each value only once and checking if its complementary `(k - root.val)` is in the `vis` set.

Therefore, the time complexity is `O(n)`, where `n` is the number of nodes in the binary tree.

Space complexity

The space complexity consists of two main components:

1. The space used by the `vis` set: In the worst case, it might store all the values contained in the tree, so it would be `O(n)`.
2. The space used by the call stack during the DFS: In the worst-case scenario of a completely unbalanced tree (where each node has only one child), the call stack would grow as large as the height of the tree, which would be `O(n)`.

Considering both components, the overall space complexity of the algorithm is `O(n)`, with `n` being the number of nodes in the tree.