783. Minimum Distance Between BST Nodes

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

Problem Description

The task is to find the minimum difference (also known as the minimum absolute difference) between the values of any two different nodes in a Binary Search Tree (BST). Remember, a BST has the property that all left descendants of a node have values less than the node's value, and all right descendants have values greater than the node's value. This property can be utilized to find the minimum difference in an efficient way.

Intuition

To solve this problem, we need to consider the properties of a BST and how they can help in finding the minimum difference. In a BST, an in-order traversal results in a sorted sequence of values. The smallest difference between any two nodes is likely to be between two adjacent nodes in this sorted order. So, the solution involves performing an in-order traversal of the tree.

The solution uses a recursive depth-first search (DFS) strategy to perform the in-order traversal. During the traversal:

1. Visit the left subtree.
2. Process the current node:
   - Compare the current node's value with the value of the previous node (which is the closest smaller value since we are doing in-order traversal).
   - Update the minimum difference (`ans`) if the current difference is less than the previously recorded minimum difference.
3. Visit the right subtree.

The `nonlocal` keyword is used for the variables `ans` and `prev` inside the `dfs` helper function to update their values across recursive calls. This is necessary because `ans` keeps track of the current minimum difference we have found, and `prev` keeps track of the last node's value we visited.

Initially, we set `ans` to infinity (`inf` in Python) to ensure any valid difference found will be smaller and thus update `ans`. The `prev` variable is initialized to infinity as well since we have not encountered any nodes before the traversal. The algorithm starts by calling the `dfs` helper function with the `root` of the BST, which then recursively performs the in-order traversal and updates `ans` with the minimum difference that it finds.

By the time we are done with the in-order traversal, `ans` holds the minimum difference between any two nodes in the BST, and we return it as the result.

Solution Approach

The implementation of the solution is based on a Depth-First Search (DFS) algorithm performing an in-order traversal of the Binary Search Tree (BST). A traversal is in-order if it visits the left child, then the node itself, and then the right child recursively. This approach leverages the BST's property that an in-order traversal results in a sorted sequence of node values.

Here are the steps in the algorithm, along with how each is implemented in the solution:

1. **Depth-First Search (DFS):**
   - A recursive method `dfs(root)` is defined, where `root` is the current node in the traversal. If `root` is `None`, the function returns immediately, as there's nothing to process (base case for the recursion).
   - The `dfs` function is called with the left child of the current node, `dfs(root.left)`, to traverse the left subtree first.
2. **Processing Current Node:**
   - Once we are at the leftmost node, which has no left child, the processing of nodes begins.
   - The current node's value is compared with the last node's value stored in the variable `prev` (initialized at infinity to start with). The difference is calculated as `abs(prev - root.val)`.
3. **Updating Minimum Difference:**
   - A `nonlocal` variable `ans` is used to keep track of the minimum difference encountered throughout the traversal. If the current difference is less than `ans`, we update `ans` with this new minimum.
   - The variable `prev` is updated with the current node's value, `root.val`, to ensure it's always set to the value of the last node visited.
4. **Recursive Right Subtree Traversal:**
   - After processing the current node, the function calls itself with the right child of the current node, `dfs(root.right)`, thus exploring the right subtree.
5. **Result:**
   - Once the entire tree has been traversed, the `ans` variable will hold the smallest difference found between any two nodes.

The use of the `nonlocal` keyword is key here, as `ans` and `prev` are updated by the recursive calls within the `dfs` function, and their values need to be retained across those calls.

A Python `class` named `Solution` contains the method `minDiffInBST` which initiates the DFS with the first call to `dfs(root)`. The `inf` value is imported from Python's `math` module and represents infinity, used for comparison purposes to ensure the actual difference always replaces the initial value.

The `ans` is initialized with `inf` so that the first computed difference (which will be finite) will become the new minimum. The `prev` variable is also initialized outside the `dfs` function to `inf` in order to manage the case when the first node (the smallest value in the BST) has no predecessor.

Data Structure: The recursive stack is the implicit data structure used here for the DFS implementation. No additional data structures are needed since the problem can be solved with the traversal alone.

Considering the algorithm and pattern applied, the solution efficiently finds and returns the minimum absolute difference between any two nodes in the BST by leveraging its properties and an in-order DFS.

Example Walkthrough

Let's say we have a Binary Search Tree (BST) with the following structure:

```
    4
   / \
  2   6
 / \
1   3
```

1. We initiate a Depth-First Search (DFS) from the root of the BST, which is the node with value `4`.

2. The DFS will first go to the left subtree (value `2`). Since `2` also has a left child, the DFS will go further left, reaching all the way to node `1`, which is the leftmost node and has no children.

3. Node `1` is processed, and since it's the first node, there's no previous node to compare with, so `prev` is set to this node's value (`1`). The `ans` is still infinity at this point.

4. The DFS backtracks to node `2`, and `2` is compared with the previous node `1`. The absolute difference is `2 - 1 = 1`, which is less than infinity, so `ans` is updated to `1`. `Prev` is now updated to `2`.

5. The DFS then moves to the right child of node `2`, which is node `3`. The difference between `3` and the previous node `2` is `3 - 2 = 1`. Since `ans` is already `1`, it does not get updated.

6. DFS backtracks to the root node `4`, but we've already processed its left subtree. Now, we compare `4` with the previous node `3`. The absolute difference is `4 - 3 = 1`, which is the same as `ans`, so no update to `ans` is made. `Prev` is updated to `4`.

7. The DFS proceeds to the right child of root, node `6`. We calculate the difference between the previous node `4` and `6`, giving us `6 - 4 = 2`, which is greater than `ans`. Thus, no update is made.

8. The traversal ends as there are no more nodes to visit. Our minimum difference `ans` is `1`, which is the minimum absolute difference between any two nodes in this BST.

The process concludes by returning `1` as the minimum difference between the values of any two different nodes in the BST.

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 minDiffInBST(self, root: TreeNode) -> int:
        # Helper function for depth-first search traversal of the binary tree
        def in_order_traversal(node):
            if node is None:
                return

            # Traverse the left subtree
            in_order_traversal(node.left)

            # Process the current node
            # Update the smallest difference and the previous value
            nonlocal min_difference, previous_value
            min_difference = min(min_difference, node.val - previous_value)
            previous_value = node.val

            # Traverse the right subtree
            in_order_traversal(node.right)

        # Initialize the minimum difference as infinity
        # The prev variable is used to keep track of the previous node's value
        # Because the initial "previous value" is not defined, we use infinity
        min_difference, previous_value = float('inf'), float('-inf')

        # Call the helper function to start in-order traversal from the root
        in_order_traversal(root)

        # Return the minimum difference found
        return min_difference

```

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 {
    // Variable to store the minimum difference
    private int minDifference;
    // Variable to keep track of the previously visited node's value in in-order traversal
    private int prevValue;
    // A large value to initialize minDifference and prevValue. It signifies "infinity".
    private final int INF = Integer.MAX_VALUE;

    // Public method to find the minimum difference between values of any two nodes in a BST
    public int minDiffInBST(TreeNode root) {
        // Initialize minDifference to "infinity" as we look for the minimum value
        minDifference = INF;
        // Initialize prevValue to "infinity" to handle the edge case of the first node
        prevValue = INF;
        // Call the helper method to do a depth-first search starting from the root
        dfs(root);
        // Return the minimum difference found
        return minDifference;
    }

    // Helper method to perform in-order traversal and find the minimum difference
    private void dfs(TreeNode node) {
        // Base case: if the node is null, just return
        if (node == null) {
            return;
        }
        // Recursively call dfs for the left subtree to visit nodes in in-order
        dfs(node.left);
        // Update the minimum difference with the smallest difference found so far
        // between the current node's value and the previous node's value
        minDifference = Math.min(minDifference, Math.abs(node.val - prevValue));
        // Update the prevValue to the current node's value for subsequent iterations
        prevValue = node.val;
        // Recursively call dfs for the right subtree to continue in-order traversal
        dfs(node.right);
    }
}

```

C++ Solution

```cpp
#include <algorithm>  // Include algorithm header for std::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:
    // Initialize the maximum possible value for an integer as infinity.
    // This will be used to calculate the minimum difference.
    static const int INF = INT_MAX;

    // Variable to hold the current minimum difference found in the BST.
    int minDifference;

    // Variable to keep track of the previous node's value during in-order traversal.
    int previousValue;

    // Constructor initializes member variables.
    Solution() : minDifference(INF), previousValue(-INF) {}

    // Function to find the minimum difference between any two nodes in the BST.
    int minDiffInBST(TreeNode* root) {
        // Reset the minDifference and previousValue before reusing the solution instance.
        minDifference = INF;
        previousValue = -INF;
        // Start the depth-first search (in-order traversal) of the tree.
        inOrderTraversal(root);
        // After the traversal, minDifference will hold the minimum difference.
        return minDifference;
    }

    // Helper function to perform in-order traversal on the BST and
    // compute the minimum difference.
    void inOrderTraversal(TreeNode* node) {
        // If the node is null, return immediately.
        if (!node) return;

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

        // If previousValue is not set to the initial value (which is -INF here),
        // update the minDifference with the minimum of current minDifference and
        // the difference between the current node's value and previousValue.
        if (previousValue != -INF) {
            minDifference = std::min(minDifference, abs(node->val - previousValue));
        }
        // Update previousValue to hold the current node's value for the next comparison.
        previousValue = node->val;

        // 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 = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}

// The maximum possible value for a number in JavaScript.
const INFINITY = Number.MAX_SAFE_INTEGER;

// Variable to hold the current minimum difference found in the BST.
let minDifference: number;

// Variable to keep track of the previous node's value during in-order traversal.
let previousValue: number;

// Function to find the minimum difference between any two nodes in the BST.
// Note that we don't have to reset minDifference and previousValue here
// because they're not part of a class instance anymore.
function minDiffInBST(root: TreeNode | null): number {
  // Initialize minDifference and previousValue for a new computation.
  minDifference = INFINITY;
  previousValue = -INFINITY;

  // Start the depth-first search (in-order traversal) of the tree.
  inOrderTraversal(root);

  // After the traversal, minDifference will hold the minimum difference.
  return minDifference;
}

// Helper function to perform in-order traversal on the BST and
// compute the minimum difference.
function inOrderTraversal(node: TreeNode | null) {
  // If the node is null, return immediately.
  if (node === null) return;

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

  // If previousValue is not set to the initial value (which is -INFINITY here),
  // update the minDifference with the minimum of current minDifference and
  // the difference between the current node's value and previousValue.
  if (previousValue !== -INFINITY) {
    minDifference = Math.min(minDifference, Math.abs(node.val - previousValue));
  }

  // Update previousValue to hold the current node's value for the next comparison.
  previousValue = node.val;

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

```

Time and Space Complexity

The given code performs an in-order traversal of a binary search tree (BST) to find the minimum difference between any two nodes. Here's the analysis of its complexity:

- **Time Complexity:** The time complexity of the function is `O(n)`, where `n` is the number of nodes in the BST. This is because the in-order traversal visits each node exactly once.

- **Space Complexity:** The space complexity of the recursive in-order traversal is `O(h)`, where `h` is the height of the BST. This accounts for the call stack during the recursive calls. In the worst case of a skewed BST, the height `h` can become `n`, which would make the space complexity `O(n)`. In a balanced BST, however, the space complexity would be `O(log n)`.