530. Minimum Absolute Difference in BST

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

Problem Description

This LeetCode problem asks us to find the minimum absolute difference between the values of any two different nodes in a Binary Search Tree (BST). A BST is a tree data structure where each node has at most two children, referred to as the left child and the right child. For any node in a BST, the value of all the nodes in the left subtree are less than the node's value, and the value of all the nodes in the right subtree are greater than the node's value.

The "minimum absolute difference" refers to the smallest difference in value between any pair of nodes in the tree. An important characteristic of a BST is that when it is traversed in-order (left node, current node, right node), it will yield the values in a sorted manner. This property will be crucial for finding the minimum absolute difference without having to compare every pair of nodes in the tree.

Intuition

To arrive at the solution, we leverage the in-order traversal property of the BST mentioned above. The idea is to perform an in-order traversal of the BST, which effectively means we will be processing the nodes in ascending order of their values.

1. During the traversal, we keep track of the value of the previously processed node.
2. At each current node, we calculate the absolute difference between the current node's value and the previously visited node's value.
3. We keep an ongoing count of the minimum absolute difference encountered so far.
4. By the end of the traversal, the minimum absolute difference will have been found.

The intuition behind this approach is based on the fact that the smallest difference between any two values in a sorted list is always between adjacent values. Since an in-order traversal of a BST gives us a sorted list of values, we only need to check adjacent nodes to determine the minimum absolute difference in the entire tree.

We use a helper function `dfs` (Depth-First Search) that performs the in-order traversal recursively. The `nonlocal` keyword is used to update the `ans` and `prev` variables defined in the outer scope of the `dfs` function.

Solution Approach

The provided Python solution makes use of recursion to implement the in-order traversal pattern for navigating the BST. Here's a step by step breakdown of how the implementation works:

1. A nested function named `dfs` (short for "Depth-First Search") is defined within the `getMinimumDifference` method. This `dfs` function is responsible for performing the in-order traversal of the BST.

2. The `dfs` function does nothing if it encounters a `None` node (the base case of the recursion), which means that it has reached a leaf node's child.

3. When the `dfs` function visits a node, it first recursively calls itself on the left child of the current node to ensure that the nodes are visited in ascending order.

4. After visiting the left child, the function then processes the current node by calculating the absolute difference between the current node's value and the previously visited node's value, and keeps track of this absolute difference if it's the smallest one seen so far. This is done using the `ans` variable, which is initialized with infinity (`inf`). `ans` represents the minimum absolute difference found during the traversal.

5. The `prev` variable holds the value of the previously visited node in the traversal. Initially, `prev` is also initialized with `inf`, and it is updated to the current node's value before moving to the right child.

6. Once the current node has been processed, the `dfs` function recursively calls itself on the right child to complete the in-order visitation pattern.

7. The `nonlocal` keyword is used for `ans` and `prev` to allow the nested `dfs` function to modify these variables that are defined in the enclosing `getMinimumDifference` method's scope.

8. The `getMinimumDifference` method initializes `ans` and `prev` with `inf` and begins the in-order traversal by calling the `dfs` function on the root node of the BST.

9. After the in-order traversal is complete, `ans` holds the minimum absolute difference between the values of any two nodes in the BST. This value is returned as the final result.

Through the use of in-order traversal, the algorithm ensures that each node is compared only with its immediate predecessor in terms of value, resulting in an efficient way to find the minimum absolute difference.

Here's the algorithm represented in pseudocode:

```
    def in_order_traversal(node):
        if node is not None:
            in_order_traversal(node.left)
            process(node)
            in_order_traversal(node.right)

    def process(node):
        update_minimum_difference(previous_node_value, node.value)
        previous_node_value = node.value
```

In the pseudocode, `process(node)` represents the steps of comparing the current node's value with the previous node's value and updating the minimum difference accordingly.

Example Walkthrough

Consider the following BST for this example:

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

Let's apply the in-order traversal to this BST and keep track of the previous node to calculate the differences and find the minimum absolute difference.

1. Start at the root (4), then traverse to the left child (2). Since 2 has a left child (1), continue traversing left until reaching a leaf node.

2. Process node 1: This is the first node, so there's no previous node to compare with. Set `prev` to 1.

3. Traverse up and to the right, now to node 2. Calculate the absolute difference with `prev`: $|2 - 1| = 1$, and since `prev` was set to 1 earlier, set `ans` to 1 (this is our first comparison, so it's also the smallest so far). Update `prev` to 2.

4. Process node 2: Node 2 is considered again, but the comparison has already been made with its left side. No left child traversal needed now.

5. Traverse to the right child of node 2, which is node 3. Calculate the absolute difference with `prev`: $|3 - 2| = 1$. Since `ans` is already 1 and $|3 - 2|$ is also 1, `ans` remains unchanged. Update `prev` to 3.

6. Move up to node 4, the root. Calculate the absolute difference with `prev`: $|4 - 3| = 1$. The `ans` is still 1, so there's no change. Update `prev` to 4.

7. Finally, move to the right child of the root, which is node 6. Calculate the absolute difference with `prev`: $|6 - 4| = 2$. Since `ans` is 1, and $|6 - 4|$ is greater than 1, there's no change to `ans`.

By the end of the traversal, `ans` holds the value of 1, which is the minimum absolute difference that was found between the values of adjacent nodes during the in-order traversal of the BST. Therefore, the minimum absolute difference between any two nodes in this BST is 1.

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 getMinimumDifference(self, root: TreeNode) -> int:
        # Initialize the minimum difference and previous node value
        self.min_diff = float('inf')
        self.prev_val = -float('inf')

        # Depth-First Search In-Order Traversal
        def in_order_traverse(node):
            if not node:
                return

            # Traverse the left subtree
            in_order_traverse(node.left)

            # Update the minimum difference
            self.min_diff = min(self.min_diff, node.val - self.prev_val)

            # Update the previous node value to the current node's value
            self.prev_val = node.val

            # Traverse the right subtree
            in_order_traverse(node.right)

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

        # Return the minimum absolute difference found between any two nodes' values
        return self.min_diff

```

Java Solution

```java
class Solution {
    private int minDifference;
    private int previousValue;
    private static final int INFINITY = Integer.MAX_VALUE; // Use a static final constant for infinity

    /**
     * Find the minimum absolute difference between values of any two nodes.
     *
     * @param root The root of the binary search tree.
     * @return The minimum absolute difference.
     */
    public int getMinimumDifference(TreeNode root) {
        minDifference = INFINITY; // Initialize minimum difference to the largest value possible
        previousValue = INFINITY; // Initialize previous value to the largest value possible for the start
        inOrderTraversal(root); // Perform in-order traversal to compare node values
        return minDifference; // Return the smallest difference found
    }

    /**
     * Perform in-order traversal on BST to find minimum absolute difference.
     *
     * @param node The current node being visited.
     */
    private void inOrderTraversal(TreeNode node) {
        if (node == null) {
            return; // Base case: if node is null, return to stop the traversal
        }
        inOrderTraversal(node.left); // Visit left subtree

        // Compute the minimum difference with the previous value (if not first node)
        if (previousValue != INFINITY) {
            minDifference = Math.min(minDifference, Math.abs(node.val - previousValue));
        }
        previousValue = node.val; // Update the previous value to the current node's value

        inOrderTraversal(node.right); // Visit right subtree
    }
}

/**
 * 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;
    }
}

```

C++ Solution

```cpp
#include <climits> // For using INT_MAX
#include <algorithm> // For using min function
#include <cstdlib> // For using abs function

// Definition for a binary tree node.
struct TreeNode {
    int val;         // Value of the node
    TreeNode *left;  // Pointer to left child
    TreeNode *right; // Pointer to right child
    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:
    const int INF = INT_MAX; // Define the maximum possible integer to represent infinity
    int minDifference;       // To store the minimum absolute difference found
    int previousValue;       // To store the last node value that was processed in in-order traversal

    // The function to initialize the problem and trigger the depth-first search
    int getMinimumDifference(TreeNode* root) {
        minDifference = INF; // Initialize the minimum difference to INF
        previousValue = INF; // Initialize the previous value as INF to handle the first node's case
        dfsInorderTraversal(root); // Start the DFS for in-order traversal
        return minDifference; // Return the final answer (minimum absolute difference)
    }

    // Recursive function to perform in-order traversal on a binary search tree
    void dfsInorderTraversal(TreeNode* node) {
        if (!node) return; // Base case: if the node is null, return

        // Traverse the left subtree
        dfsInorderTraversal(node->left);

        // Process the current node
        if (previousValue != INF) {
            // If the previous value is valid, update minDifference
            minDifference = std::min(minDifference, std::abs(previousValue - node->val));
        }
        previousValue = node->val; // Update the previous value with the current node's value

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

```

Typescript Solution

```typescript
// Import required functions from the standard library.
import { maxSafeInteger } from "util";
import { min, abs } from "Math";

// Definition for a binary tree node.
class TreeNode {
    val: number;            // Value of the node
    left: TreeNode | null;  // Pointer to left child
    right: TreeNode | null; // Pointer to right child
    constructor(val: number = 0, left: TreeNode | null = null, right: TreeNode | null = null) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// Initialize the maximum possible integer to represent infinity in TypeScript.
const INF: number = maxSafeInteger;
// Variables to store the minimum absolute difference and the last node value processed.
let minDifference: number = INF;
let previousValue: number = INF;

// Function to initialize the problem and trigger the depth-first search for in-order traversal.
function getMinimumDifference(root: TreeNode | null): number {
    minDifference = INF;
    previousValue = INF;
    dfsInorderTraversal(root);
    return minDifference;
}

// Recursive function to perform in-order traversal on a binary search tree.
function dfsInorderTraversal(node: TreeNode | null): void {
    if (!node) return; // If the node is null, exit the function.

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

    // Process the current node by updating the minDifference with the absolute difference
    // between the current node's value and the previous value if previousValue is valid.
    if (previousValue !== INF) {
        minDifference = min(minDifference, abs(previousValue - node.val));
    }
    // Update the previous value with the current node's value.
    previousValue = node.val;

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

```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the number of nodes in the binary tree. This time complexity arises because the depth-first search (DFS) in the function `dfs(root)` visits each node exactly once. The actions performed on each node—including updating the 'ans' and 'prev' variables—are all `O(1)` operations, so they do not add to the overall complexity beyond that imposed by the traversal.

The space complexity of the code is `O(h)`, where `h` is the height of the binary tree. This space complexity is due to the recursive call stack that will grow to the height of the tree in the worst case. For a balanced tree, this would be `O(log n)`, but for a skewed tree (e.g., a tree where each node only has a left or a right child), this could degrade to `O(n)`.