270. Closest Binary Search Tree Value

Problem Description

In this problem, you're given the root of a binary search tree (BST) and a target value. You need to find the value in the BST that is closest to the target value. It's important to understand that a BST is a special type of binary tree where for each node, the left child has a value less than the node's value, and the right child has a value greater than the node's value. This property of BSTs allows efficient searching, similar to binary search in a sorted array.

When looking for the value closest to the target, if there are multiple such values that are equally close, you should return the smaller one. Your task is to implement an algorithm that will traverse the BST and return the value that meets these conditions.

Intuition

To solve this problem, we can utilize the properties of the BST to guide our search similar to a binary search algorithm. The intuition here is that, when searching in a BST, you can move left or right from a node depending on how the current node's value compares to the target value. If the current node's value is greater than the target, then the closest value can either be the current node's value or in the left subtree. If the current node's value is less than the target, then the closest value can either be the current node's value or in the right subtree.

We use the absolute difference between the current node's value and the target value to judge closeness. As we traverse the BST, we keep updating the closest value found so far. We start with the root node's value as the initial closest value and an infinite difference to ensure any actual number found is nearer.

The key elements for our approach here:

- We will traverse the BST in a while loop, choosing left or right child nodes depending on how they compare to the target value.
- In each step of the traversal, we will keep track of the closest value and the smallest difference seen so far.
- We only update the closest value if the current node offers a closer or equal but smaller value compared to what we have seen so far.
- Since we are doing a binary search-like traversal, the time complexity is O(h) where h is the height of the BST. In the worst case, this can be O(n) if the tree is skewed, but it's generally more efficient than a full traversal.

By following this approach, we are able to leverage the BST property to efficiently find the value closest to the given target.

Solution Approach

The implementation follows a binary search approach which, in the context of BST, translates into a traversal where the decision to move left or right is based on how the current node's value compares to the target value.

Algorithm

1. Initialize two variables, `ans` to store the current closest value and `mi` to store the minimum difference, which is initially set to infinity (`inf`).
2. Begin a while loop that continues as long as there is a valid node (while `root` is not `None`).
3. For every node visited, calculate the absolute difference `t` between the node's value (`root.val`) and the target.
4. If this difference is less than the current minimum difference (`mi`), update `mi` to this new difference, and set `ans` to the current node's value (`root.val`).
5. If there's a tie in differences (meaning the current difference `t` is exactly equal to the `mi`), and the current node's value is smaller than the `ans`, update `ans` to the current node's value. This ensures that in case of multiple closest values, the smallest one is chosen.
6. Decide the direction of traversal:
   - If the current node's value is greater than the target, move to the left child (`root = root.left`), since a smaller value closer to the target might be found there.
   - If the current node's value is less than or equal to the target, move to the right child (`root = root.right`), as a value closer to the target (or a larger value that is equaly close) might be found.
7. Once the while loop exits (because `root` becomes `None`), return `ans`, which will hold the closest value found during the traversal.

Data Structures

The primary data structure we need is the BST, represented by the `TreeNode` class, which contains `val` (the value of the node), `left` (left subtree), and `right` (right subtree). Variables `ans` and `mi` are integers that help us keep track of the closest value and the minimum difference found during the traversal.

Patterns Used

The main pattern used here is **Binary Search**. In a sorted array, binary search compares the target with the middle element and decides to continue searching in the left or right halve, effectively cutting down the search space by half in each step. Similarly, in a BST, each decision to move left or right at a node reduces the potential search space considering the ordered nature of BST, where left subtree contains values smaller than the node and the right subtree contains values larger than the node.

This implementation allows us to find the closest value to the target with the efficiency of binary search, thus utilizing the BST's ordered structure to its full advantage.

Code Implementation

The solution definition in Python using the described algorithm is as follows:

```
class Solution:
    def closestValue(self, root: Optional[TreeNode], target: float) -> int:
        ans = mi = root.val, inf
        while root:
            t = abs(root.val - target)
            if t < mi or (t == mi and root.val < ans):
                mi = t
                ans = root.val
            if root.val > target:
                root = root.left
            else:
                root = root.right
        return ans
```

In this implementation, `Optional[TreeNode]` indicates that `root` may be either an instance of `TreeNode` or `None`.

By adhering to this approach, we leverage the given structure of the BST to efficiently locate the closest value to the target. The algorithm is optimized to run with a time complexity that is dependent on the height of the tree.

Example Walkthrough

Let's illustrate the solution approach with a small example. Consider the following BST and a target value of 4.5:

```
    4
   / \
  2   5
 / \
1   3
```

We will walk through the algorithm step-by-step with this BST:

1. **Initialize**:
   - `ans = 4` (First node value)
   - `mi = inf` (As we haven't calculated any differences yet)
2. **First iteration (Node = 4)**:
   - Calculate the absolute difference `t` between `root.val` (4) and the target (4.5), which is `t = 0.5`.
   - Since `t < mi`, update `mi = 0.5` and `ans = 4`.
   - Node value 4 is less than the target, so move right to find a potentially closer value.
3. **Second Iteration (Node = 5)**:
   - Now `t` is `0.5` (`5 - 4.5`).
   - `t` is equal to the current `mi`, and since we prefer smaller values, we do not update `ans`, keeping `ans = 4`.
   - Node value 5 is greater than the target, so move left to find a potentially closer, smaller value.
4. **Third iteration (Node = 2)**:
   - Now, the tree node is `None` because there is no left child of node 5, so we stop.

As there are no more nodes to visit, we return the `ans`. In this case, `ans = 4`, which is the closest value to the target 4.5 in the BST given. If there had been nodes on the left side of the current node 5, the algorithm would have continued evaluating those nodes.

This walkthrough demonstrates how the decision to move left or right at each node effectively narrows the search utilizing the BST property and binary search principle. The algorithm ensures we do not traverse the entire tree but only the path that could potentially contain the closest value. Given its reliance on tree height, it offers a time-efficient solution.

Python Solution

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def closestValue(self, root: TreeNode, target: float) -> int:
        # Initialize the closest_value with the root's value
        closest_value = root.val

        # Initialize the minimum difference found
        minimum_difference = float('inf')

        # Iterate over the nodes of the binary search tree
        while root:
            # Calculate the current difference between node's value and the target
            current_difference = abs(root.val - target)

            # If the current difference is smaller or equal but with a lesser value, update the closest_value
            if current_difference < minimum_difference or (current_difference == minimum_difference and root.val < closest_value):
                minimum_difference = current_difference
                closest_value = root.val

            # Move left if the target is smaller than the current node's value
            if root.val > target:
                root = root.left
            # Otherwise, move right
            else:
                root = root.right

        # Once we find the closest value, return it
        return closest_value
```

Java Solution

```
class Solution {
    public int closestValue(TreeNode root, double target) {
        int closestValue = root.val; // Initialize closest value to root's value
        double minDifference = Double.MAX_VALUE; // Initialize minimum difference to maximum possible value

        // Traverse the tree starting from the root
        while (root != null) {
            // Current difference between node's value and the target value
            double currentDifference = Math.abs(root.val - target);

            // If current difference is smaller than previous minDifference
            // or current difference is equal but value is smaller than previously stored value (closer to target)
            if (currentDifference < minDifference || (currentDifference == minDifference && root.val < closestValue)) {
                minDifference = currentDifference; // Update the smallest difference
                closestValue = root.val; // Update the closest value
            }

            // If the current node's value is greater than the target,
            // the closer value must be in the left subtree
            if (root.val > target) {
                root = root.left;
            }
            // Otherwise, it must be in the right subtree
            else {
                root = root.right;
            }
        }
        // After traversing the tree, return the value that is closest to the target
        return closestValue;
    }
}
```

C++ Solution

```
// Define a structure for the Binary Tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    // Constructor for creating a node without children.
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    // Constructor for creating a leaf node with a specific value.
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    // Constructor for creating a node with a value and left and right children.
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

// Solution class contains methods to solve the problem.
class Solution {
public:
    // Method that finds the value in the binary tree which is closest to the target value.
    int closestValue(TreeNode* root, double target) {
        // Initialize the closest value reference as the root value.
        int closestValueRef = root->val;
        // Initialize minimum difference seen so far to maximum possible integer value.
        double minimumDiff = INT_MAX;

        // Iterate down the tree until we reach a leaf (NULL) node.
        while (root) {
            // Calculate current difference between node value and target.
            double currentDiff = abs(root->val - target);

            // Check if the current difference is less than the minimum difference found so far.
            // If the differences are equal, we prioritize smaller values per the problem's constraint.
            if (currentDiff < minimumDiff || (currentDiff == minimumDiff && root->val < closestValueRef)) {
                minimumDiff = currentDiff;  // Update the minimum difference.
                closestValueRef = root->val;  // Update the closest value reference.
            }

            // Decide to go left or right in the tree depending on the comparison of the current node's value and target.
            if (root->val > target) {  // Target is smaller, go left.
                root = root->left;
            } else {
                root = root->right;  // Target is larger, go right.
            }
        }

        // Return the closest value found.
        return closestValueRef;
    }
};
```

Typescript Solution

```
// Definition for a binary tree node using TypeScript syntax with explicit types.
interface TreeNode {
  val: number;
  left: TreeNode | null;
  right: TreeNode | null;
}

/**
 * Finds the value in a binary search tree that is closest to the target value.
 * @param {TreeNode | null} root - The root of the binary search tree.
 * @param {number} target - The target value to find the closest node to.
 * @returns {number} - The value of the closest node.
 */
function closestValue(root: TreeNode | null, target: number): number {
  // Set an initial answer to the root's value if the root exists, otherwise 0.
  let closestValue: number = root ? root.val : 0;
  // Initialize a variable to store the smallest difference encountered.
  let minDifference: number = Number.MAX_VALUE;

  // Traverse the tree starting from the root.
  while (root) {
    // Compute the absolute difference between the current node's value and the target.
    const currentDifference: number = Math.abs(root.val - target);
    // If the current difference is smaller than any previously encountered,
    // or if it's the same but the value is less than our current answer, update the answer.
    if (currentDifference < minDifference || (currentDifference === minDifference && root.val < closestValue)) {
      minDifference = currentDifference;
      closestValue = root.val;
    }
    // Decide to move left or right in the tree based on how the current node's value compares to the target.
    if (root.val > target) {
      root = root.left; // Move left if the current value is greater than the target.
    } else {
      root = root.right; // Move right if the current value is less than or equal to the target.
    }
  }
  // Return the value of the closest node after traversing the tree.
  return closestValue;
}
```

Time and Space Complexity

Time Complexity

The time complexity of this function depends on the height of the binary tree. In the worst-case scenario, the tree is skewed (i.e., each node has only one child), resulting in a height of `n`, where `n` is the number of nodes in the tree. In that case, we would have to visit every node, making the time complexity `O(n)`.

In the best-case scenario, the binary tree is balanced, and the height of the tree would be `log(n)`. Since in a balanced tree, with each comparison of the node's value to the target, we are effectively eliminating half of the potential nodes in which the closest value could reside, we end up with a time complexity of `O(log(n))`.

Therefore, the overall time complexity can be expressed as `O(h)`, where `h` denotes the height of the tree, which is between `O(log(n))` and `O(n)`.

Space Complexity

The space complexity of the given function is `O(1)`. This is because the function only uses a constant amount of extra space for a few variables (`ans`, `mi`, and `t`) regardless of the input size. It is an iterative solution that does not incur any additional stack space usage that would be associated with recursion. Therefore, no matter how large the binary tree is, the space used by the function remains the same.