272. Closest Binary Search Tree Value II

Hard Stack Tree Depth-First Search Binary Search Tree Two Pointers Binary Tree Heap (Priority Queue) Leetcode Link

Problem Description

The problem provides us with a binary search tree (BST) and a target value `target`. Our goal is to find `k` values in the BST that are closest to the target value. The values can be returned in any order. The problem guarantees that there is a unique set of `k` values in the tree that are closest to the target.

Intuition

To resolve this problem, we use an in-order traversal of the BST. The reason for this choice is that an in-order traversal of a BST yields the values in sorted order. As we traverse the tree, we compare the values of the nodes with the target value to determine how close they are to the target.

We use a deque (double-ended queue) to keep track of the closest values found so far. The deque maintains the `k` closest values in sorted order due to the nature of in-order traversal. Here's a step-by-step outline of the intuition:

- Perform an in-order traversal (left-root-right) because the BST's property guarantees sorted values.
- Use a deque of size `k` to maintain the closest values to the target. We start by adding values to the deque until it is full.
- Once the deque has `k` elements, we compare the current value (as we continue in-order traversal) with the first element in the deque (the element that is the farthest from the target among those in the deque). If the current value is closer, we remove the first element and add the current value to the deque.
- If we find a value that is not closer than the first element in the deque, we can stop the traversal. Since the values are sorted, all subsequent values will be even farther from the target.
- After completing the traversal, we return the contents of the deque as our result.

This approach efficiently finds the k closest values by leveraging the sorted nature of the BST and by keeping our list of candidates to a fixed size (k).

Solution Approach

The solution uses a Depth-First Search (DFS) in-order traversal strategy to explore the BST and a deque data structure to keep track of the `k` closest values. Let's look at how the provided code implements this:

1. A helper function `dfs(root)` is defined for traversal purposes. It's a recursive function that takes the current node as its argument and explores it in an in-order fashion (left-root-right).

2. When the `dfs` function is called with the BST `root`, it first checks if the current node is `None`, meaning it's reached the end of a path in the tree, and in that case, it returns without doing anything further.

3. The function then proceeds to recursively call itself to explore the left subtree: `dfs(root.left)`.

4. After exploring the left sub-tree, the function checks if the deque named `q` is already full (i.e., if it already contains `k` elements):

    - If `q` is not full, it appends the current node's value to `q`.
    - If `q` is full, the function compares the absolute difference between the target and the current value (`abs(root.val - target)`) with the absolute difference between the target and the value at the front of the deque (`abs(q[0] - target)`):
        - If the current value is closer to the target than the value at the front of `q`, we `popleft` from `q` to remove the farthest value, and append the current value (`q.append(root.val)`).
        - If the current value is not closer, the traversal is halted as further right nodes will be even farther from the target.

5. The DFS continues to the right subtree: `dfs(root.right)`, as long as the closest k values are not yet finalized (meaning it returns early if a value is farther from the target than the first value in `q`).

6. A deque object `q` is created outside of the `dfs` function and is passed by reference into it. A deque is used because it allows efficient addition and removal of elements from the start of the queue (`popleft`), which is required when we find a closer value and we need to remove the least close value in it.

7. After the DFS traversal is complete, the function `closestKValues` returns the current content of the deque `q` converted to a list with `list(q)`, which contains the `k` values closest to the target.

By utilizing a modified in-order traversal that stops early when appropriate, and a deque to keep a running set of the closest values, the solution efficiently finds the `k` values in the BST that are closest to the target value.

Example Walkthrough

Let's walk through a small example using the solution approach outlined above to demonstrate how we can find the `k` values closest to a `target` within a Binary Search Tree (BST):

Example BST

Consider the following BST, where `k = 3` and `target = 5`:

```
     4
    / \
   2   6
  / \   \
 1   3   7
```

Our goal is to find the 3 values closest to 5.

Step-by-Step Traversal and Deque Operations

1. Begin with an empty deque `q` and start the in-order DFS traversal from the root (4).

2. Visit the left child, 2.

    - Call dfs(2), visit left child (1), and call dfs(1). Since 1 has no left child, append 1 to the deque.
        - `q = [1]`
    - Return to 2, visit right child (3), and call dfs(3). Since 3 has no children, append 3 to deque.
        - `q = [1, 3]`
    - With `q` still not full, append 2’s value to `q`.
        - `q = [1, 3, 2]`

3. Return to the root, 4. Now `q` is full.

    - Compare 4 with the front of `q` (1). Since `abs(4 - 5) < abs(1 - 5)`, pop from the left and append 4.
        - `q = [3, 2, 4]`

4. Visit the right child, 6.

    - Call dfs(6), and compare 6 with the front of `q` (3).
        - Since `abs(6 - 5) < abs(3 - 5)`, we remove the front value and add 6.
            - `q = [2, 4, 6]`
        - Visit the right child of 6, which is 7, and compare with the front of `q` (2).
        - Since `abs(7 - 5) > abs(2 - 5)`, we do not need to visit any more nodes because all subsequent nodes will be farther away.

5. Conversion to a list and return:

    - Convert the deque to a list, resulting in the k values closest to the target: `[2, 4, 6]`

By following this walkthrough, we have efficiently located the `k` closest values to the target in the BST.

Python Solution

```python
from collections import deque

# 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 closestKValues(self, root: TreeNode, target: float, k: int) -> List[int]:
        # Perform in-order depth-first search to traverse the tree.
        def in_order_dfs(node):
            if node is None:
                return

            # Recurse on the left child.
            in_order_dfs(node.left)

            # Process the current node.
            # If we have fewer than k values, add current node's value.
            if len(closest_values) < k:
                closest_values.append(node.val)
            else:
                # Once we have k values, check if current node is closer to target
                # than the first value in the deque. If not, no need to proceed further.
                if abs(node.val - target) >= abs(closest_values[0] - target):
                    return

                # If the current node is closer, pop the first value and append the current value.
                closest_values.popleft()
                closest_values.append(node.val)

            # Recurse on the right child.
            in_order_dfs(node.right)

        # This deque will store the closest k values encountered so far.
        closest_values = deque()

        # Start the in-order traversal of the tree.
        in_order_dfs(root)

        # Return the k closest values as a list.
        return list(closest_values)

```

Java Solution

```java
class Solution {
    // Define a list to hold the closest k values.
    private List<Integer> closestValues;

    // Define a variable to hold the target value for comparison.
    private double targetValue;

    // Define a variable to hold the number of closest values required.
    private int numOfClosestValues;

    // Public method to call to find the k closest values to a target in a binary search tree.
    public List<Integer> closestKValues(TreeNode root, double target, int k) {
        closestValues = new LinkedList<>();
        targetValue = target;
        numOfClosestValues = k;
        inOrderTraversal(root);
        return closestValues;
    }

    // Helper method to perform in-order traversal of the binary tree.
    private void inOrderTraversal(TreeNode node) {
        // Base case: if the node is null, return immediately.
        if (node == null) {
            return;
        }

        // Recursive call on the left subtree.
        inOrderTraversal(node.left);

        // If the current size of the closestValues list is less than k,
        // add the current node's value to the list.
        if (closestValues.size() < numOfClosestValues) {
            closestValues.add(node.val);
        } else {
            // If adding the current node's value to the list does not bring it closer to the target,
            // stop the traversal since nodes farther to the right will be even less close.
            if (Math.abs(node.val - targetValue) >= Math.abs(closestValues.get(0) - targetValue)) {
                return;
            }

            // Remove the first/oldest element in the list
            closestValues.remove(0);

            // Add the current node's value to the list
            closestValues.add(node.val);
        }

        // Recursive call on 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:
    std::queue<int> closeValuesQueue; // Queue to keep track of the k closest values
    double targetValue;               // Target value to compare against
    int kValues;                      // Number of closest values to find

    // Function to find k values in the BST closest to the target value
    std::vector<int> closestKValues(TreeNode* root, double target, int k) {
        this->targetValue = target;
        this->kValues = k;
        traverseInOrder(root);

        // Extract values from queue and store them in an answer vector
        std::vector<int> closestValues;
        while (!closeValuesQueue.empty()) {
            closestValues.push_back(closeValuesQueue.front());
            closeValuesQueue.pop();
        }
        return closestValues;
    }

    // In-order traversal of the BST
    void traverseInOrder(TreeNode* node) {
        if (!node) return; // Base case: node is null

        // Traverse left subtree
        traverseInOrder(node->left);

        // Check if the number of elements in the queue is less than k
        if (closeValuesQueue.size() < kValues)
            closeValuesQueue.push(node->val);
        else {
            // Check if the current value is closer to the target than the front of the queue
            if (std::abs(node->val - targetValue) >= std::abs(closeValuesQueue.front() - targetValue))
                return; // If not, we don't need to continue as the right subtree will have even larger values
            closeValuesQueue.pop(); // Remove the furthest value
            closeValuesQueue.push(node->val); // Add the current, closer value
        }

        // Traverse right subtree
        traverseInOrder(node->right);
    }
};

```

Typescript Solution

```typescript
// Tree node structure
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

let closeValuesQueue: number[] = []; // Queue to keep track of the k closest values
let targetValue: number = 0;         // Target value to compare against
let kValues: number = 0;             // Number of closest values to find

// Function to find k values in the BST closest to the target value
function closestKValues(root: TreeNode | null, target: number, k: number): number[] {
    targetValue = target;
    kValues = k;
    closeValuesQueue = []; // initialize the queue to be empty
    traverseInOrder(root);

    // The queue is already a list of closest values when using TypeScript arrays
    return closeValuesQueue;
}

// In-order traversal of the BST
function traverseInOrder(node: TreeNode | null): void {
    if (!node) return; // Base case: node is null

    // Traverse left subtree
    traverseInOrder(node.left);

    // Check if the number of elements in the queue is less than k
    if (closeValuesQueue.length < kValues) {
        closeValuesQueue.push(node.val);
    } else {
        // Check if the current value is closer to the target than the first element of the queue
        if (Math.abs(node.val - targetValue) < Math.abs(closeValuesQueue[0] - targetValue)) {
            closeValuesQueue.shift(); // Remove the furthest value
            closeValuesQueue.push(node.val); // Add the current, closer value
        } else {
            // If not closer, we can break here because the right subtree will not have closer values
            return;
        }
    }

    // After processing current node, ensure queue is sorted by closest to the target value
    closeValuesQueue.sort((a, b) => Math.abs(a - targetValue) - Math.abs(b - targetValue));

    // Traverse right subtree
    traverseInOrder(node.right);
}

```

Time and Space Complexity

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

The space complexity is `O(H + k)` where `H` denotes the height of the binary tree, which is the space required for the call stack during the recursive traversal, and `k` is the space for storing closest values in the queue. In the worst case, the height of the tree can be `O(N)` when the tree is skewed, leading to the worst-case space complexity of `O(N + k)`. In a balanced tree, however, the height `H` is `O(log N)`, leading to a more typical space complexity of `O(log N + k)`.