230. Kth Smallest Element in a BST

Problem Description

The problem asks for the kth smallest value in a Binary Search Tree (BST). A BST is a tree data structure where each node has at most two children, known as the left child and the right child. In a BST, the left child's value is less than its parent's value, and the right child's value is greater than its parent's value. This property of BST provides an in-order sequence of nodes, which is an array of node values sorted in ascending order. The values in the nodes are unique, allowing us to talk about the kth smallest value. Since the value k is 1-indexed, the first smallest value corresponds to k=1, the second smallest to k=2, and so on.

Intuition

The intuition behind the solution is hinged on the in-order traversal of a BST, which yields the nodes in ascending order. An in-order traversal involves visiting the left subtree, the node itself, and then the right subtree. When this is done for all nodes, we end up visiting nodes in increasing order due to the properties of the BST.

The stated problem is a classic case for using this traversal strategy. However, instead of doing a complete in-order traversal and then finding the kth smallest element, we aim to optimize the process by stopping as soon as we've found the kth smallest element.

The solution uses an iterative approach with a stack to simulate the in-order traversal rather than doing it recursively. We do this by traversing to the leftmost node while pushing all the nodes on the path onto the stack. Once we reach the leftmost node, we start popping nodes from the stack.

Each pop operation corresponds to visiting a node in the in-order sequence. We decrement the value of k each time a node is popped. When k becomes zero, it means we have reached the kth smallest element, and we can stop the traversal and return the value of the current node.

The benefit of this approach is that we only visit k nodes, making the time complexity proportional to k, which could be substantially less than the total number of nodes in the BST.

Solution Approach

The solution provided uses an iterative approach to perform the in-order traversal of the binary search tree. Here is how the algorithm is implemented:

1. A stack, denoted by `stk`, is initialized. It's a fundamental data structure used here to simulate the recursive nature of in-order traversal without actually using recursion. The stack keeps track of nodes along the path that we've yet to visit.
2. The main loop of the algorithm continues as long as there are nodes to be processed, which is indicated by the conditions `while root or stk`.
3. Within the loop, if the current node `root` is not `None`, the algorithm pushes this node onto the stack using `stk.append(root)`, and then moves left down the tree by setting `root = root.left`. This is essentially diving into the leftmost node, adhering to the in-order traversal pattern.
4. If the current node `root` is `None`, which means there is no left child, the algorithm then pops a node from the stack with `root = stk.pop()`. Popping from the stack is the equivalent of "visiting" a node in the in-order sequence since it has no left children or all its left children have been visited.
5. After popping a node, `k` is decremented by 1 to indicate that we have found one more element in the in-order traversal.
6. When `k` becomes 0, the loop has reached the kth smallest node, and the value of this node is returned with `return root.val`. At this point, the traversal stops and does not continue to process the remaining nodes.
7. If `k` has not reached 0, it means the kth smallest element has not been found yet, and the algorithm needs to move to the right child of the current node by setting `root = root.right`. And then the loop will continue to the next iteration to check if there's a left subtree to process from this new node.

The implementation uses a stack to store nodes and a while loop to maintain state, which avoids the overhead of function calls in a recursion, especially if the tree is skewed or the `k` value is large. With the use of this stack and loop-based approach, the algorithm efficiently realizes the in-order traversal and effectively returns the kth smallest element with the desired complexity.

Example Walkthrough

Let us consider the following binary search tree (BST) for an example walkthrough:

```
    5
   / \
  3   6
 / \
2   4
/
1
```

Suppose we want to find the kth smallest element with `k = 3`. The in-order traversal of this BST will give us the nodes in ascending order: `1, 2, 3, 4, 5, 6`.

1. We initialize an empty stack `stk`.
2. Starting with `root` as the root of the BST (node with value 5), we follow the left children by moving down to nodes with values `3`, then `2`, and finally to `1`. Each node is pushed onto the stack `stk` before moving to its left child.

   Stack `stk` status: [5, 3, 2, 1] (top to bottom).
3. Since node 1 has no left child, we start popping from the stack.
4. Node 1 is popped; `k` is decremented to 2.

   Stack `stk` status: [5, 3, 2].
5. Since node 1 has no right child, we continue popping.
6. Node 2 is popped; `k` is decremented to 1.

   Stack `stk` status: [5, 3].
7. Now, the current root becomes 3 (after popping 2) and hence, we check its right child. Since it has one, we push it onto the stack and set `root` to node 3's right child.

   Stack `stk` status: [5, 3, 4].
8. Node 4 has no left child, so we pop it from the stack.
9. Node 4 is popped; `k` is decremented to 0.

   Stack `stk` status: [5, 3].
10. When `k` is 0, we've found our kth smallest element, which is the value of node 4.

Thus, the value of the 3rd smallest element in the BST is 4. The algorithm successfully finds the kth smallest element using the iterative in-order traversal, with the help of the stack data structure, and stops as soon as it finds the required element without traversing the whole tree.

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 kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        # Initialize an empty list to be used as a stack
        stack = []

        # Continue the loop as long as there are nodes to be processed
        # either in the stack or in the tree starting with the root.
        while root or stack:
            # Go to the leftmost node
            if root:
                # Push the current node onto the stack
                stack.append(root)
                # Move to the left child of the current node
                root = root.left
            else:
                # If there is no left child, pop the node from the stack,
                # this node is the next in the in-order traversal
                root = stack.pop()
                # Decrement k, as we have found one more small element
                k -= 1
                # If k is 0, we have found the kth smallest element
                if k == 0:
                    # Return the value of kth smallest node
                    return root.val
                # Move to the right child of the current node
                root = root.right

        # The function should always return before reaching this point
```

Java Solution

```java
class Solution {
    // Function to find the k-th smallest element in a BST
    public int kthSmallest(TreeNode root, int k) {
        // Stack to simulate the in-order traversal iteratively
        Deque<TreeNode> stack = new ArrayDeque<>();

        // Continue until we have no unfinished nodes to process
        while (root != null || !stack.isEmpty()) {
            // Traverse the left side of the current subtree
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            // Process leftmost node
            root = stack.pop();
            // Decrement k and see if it becomes 0, it means we found our k-th smallest
            if (--k == 0) {
                return root.val;
            }
            // Move to right subtree to continue the process
            root = root.right;
        }
        // If k is not valid, return 0 as a default value
        return 0;
    }
}
```

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:
    /**
     * Returns the k-th smallest element in a BST.
     * This is achieved by performing an in-order traversal iteratively.
     * In-order traversal of a BST yields the nodes in increasing order.
     *
     * @param root A pointer to the root node of the BST.
     * @param k The index (1-indexed) of the smallest element to be found.
     * @return The value of the k-th smallest node.
     */
    int kthSmallest(TreeNode* root, int k) {
        // Initialize an empty stack that will be used to perform the in-order traversal.
        stack<TreeNode*> stk;

        // Continue traversing the tree until we have visited all nodes.
        while (root || !stk.empty()) {
            // Go left as long as there is a left child.
            if (root) {
                stk.push(root);  // Push the current node onto the stack before moving to left child.
                root = root->left;  // Move left.
            } else {
                // Process the nodes that do not have a left child anymore.
                root = stk.top();
                stk.pop();  // Remove the node from the stack.
                if (--k == 0) {
                    // If we have reached the k-th smallest, return its value.
                    return root->val;
                }
                // Move to the right child, which will be processed after all its left children.
                root = root->right;
            }
        }
        // If we are here, it means the k-th smallest element could not be found, return 0.
        // Although in valid BSTs with at least k nodes, the function will never reach here.
        return 0;
    }
};
```

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;
    }
}

/**
 * Finds the k-th smallest element in a BST (Binary Search Tree).
 * @param {TreeNode | null} root - The root of the BST.
 * @param {number} k - The k-th position to find the smallest element.
 * @returns {number} - The k-th smallest element in the BST.
 */
function kthSmallest(root: TreeNode | null, k: number): number {
    // Decrement k-th position, initially the number of elements to skip
    let elementsToSkip: number = k;

    /**
     * An in-order traversal of the BST that keeps track of the traversal sequence
     * and searches for the k-th smallest element.
     * @param {TreeNode | null} node - The current node in the traversal.
     * @returns {number} - The value of the k-th smallest element or -1 if not found.
     */
    function inOrderTraversal(node: TreeNode | null): number {
        // Base case: if the current node is null, return -1 to indicate no further nodes in this path
        if (node === null) {
            return -1;
        }

        // Recurse on the left subtree
        const leftValue = inOrderTraversal(node.left);
        // If a valid non-negative value is found from the left subtree, return it as the result
        if (leftValue !== -1) {
            return leftValue;
        }

        // Decrement elementsToSkip to move towards the k-th smallest element
        elementsToSkip--;
        // If elementsToSkip becomes 0, current node's value is the k-th smallest, return it
        if (elementsToSkip === 0) {
            return node.val;
        }

        // Recurse on the right subtree
        return inOrderTraversal(node.right);
    }

    // Perform in-order traversal and return value of the k-th smallest element
    return inOrderTraversal(root);
}

// Note: This algorithm assumes the BST property where for every node, all values in its left subtree
// are smaller than its value, and all values in the right subtree are greater. The in-order traversal
// then guarantees that the elements are visited in ascending order.
```

Time and Space Complexity

The code implements an in-order traversal of a binary search tree to find the kth smallest element. Let's analyze the time complexity and space complexity:

Time Complexity

The time complexity of the code is $O(H + k)$, where H is the height of the tree, and k is the input parameter. This is because in the worst case, the code performs the in-order traversal to the left-most node (which could be a height of H), and then continues to traverse the tree until the kth smallest element is found. However, since this is a binary search tree, in the average case where the tree is balanced, the height H can be considered as $\log(N)$ and thus the average time complexity would be $O(\log(N) + k)$.

Space Complexity

The space complexity is determined by the size of the stack `stk` which, in the worst case, will hold all the nodes of one branch of the tree at the same time. In the worst case, this can be $O(H)$ - which is the height of the tree if the tree is skewed. However, in an average case where the tree is reasonably balanced, the height H would be $\log(N)$ and hence we can consider the space complexity as $O(\log(N))$.