**Notes on Approach for LeetCode Problem 700: Search in a Binary Search Tree**

**Approach**

1. **Base Case Handling:**
    - If the root is NULL, return NULL as there are no nodes to search.
    - If the root's value matches val, return the root as it is the required subtree.
2. **Utilizing BST Property:**
    - Since a Binary Search Tree (BST) maintains the property where left subtree nodes have values smaller than the root, and right subtree nodes have values greater than the root, we can leverage this to optimize the search.
3. **Recursive Search:**
    - If val is smaller than root->val, search in the left subtree: searchBST(root->left, val).
    - If val is greater than root->val, search in the right subtree: searchBST(root->right, val).
    - This ensures that we only traverse the necessary part of the BST, making the search efficient.

**Code Implementation**

```
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(root == NULL || root->val == val) return root;
        else if(root->val > val) return searchBST(root->left, val);
        else return searchBST(root->right, val);
    }
};
```

**Time Complexity Analysis**

- **Best Case: O(1)** (If the root itself contains the value val, we return immediately.)
- **Average Case: O(log N)** (In a balanced BST, each recursive call halves the number of nodes we need to search.)
- **Worst Case: O(N)** (If the BST is skewed like a linked list, we may have to traverse all nodes.)

**Space Complexity Analysis**

- **Recursive Approach: O(H)**, where H is the height of the BST (log N for a balanced BST, N for a skewed BST due to recursion stack depth).

- **Iterative Approach (if used): O(1)** (No extra recursive stack space required, only a pointer variable needed.)

**Alternative Approach**

**Iterative Method:**

Instead of recursion, we can use a loop to reduce space complexity:

```
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        while(root != NULL && root->val != val) {
            root = (root->val > val) ? root->left : root->right;
        }
        return root;
    }
};
```

- **Time Complexity:** O(log N) in a balanced BST, O(N) in the worst case (skewed BST).
- **Space Complexity:** O(1) as no extra recursive calls are made.

**Key Takeaways:**

- Used the **Binary Search Tree property** to efficiently locate the node.
- The recursive approach is simple and intuitive but has an extra **O(H) space overhead** due to recursive calls.
- The iterative approach is more memory-efficient with **O(1) space complexity**.
- The overall **search operation is logarithmic O(log N) in a balanced BST** but degrades to **O(N) in an unbalanced BST**.

**Conclusion**

The recursive approach works well for moderate-sized BSTs but can cause stack overflow for very deep trees. If the BST is unbalanced, the iterative approach might be more space-efficient.