Kth Smallest Element in a BST

🧠 **#ProblemOfTheDay**: "Kth Smallest Element in a BST"
Given a Binary Search Tree (BST) and a number **K**, find the **Kth smallest element**. The in-order traversal of a BST gives the elements in sorted order, so it's all about that order, baby! 🌳 #DataStructures #BinarySearchTree

---

👀 **#InOrderTraversal** FTW!
In a BST, the left subtree is smaller, the right subtree is larger. When you perform an **in-order traversal**, you get elements in increasing order. 😎 So, just traverse and count the nodes until you hit K! #AlgorithmTips #BST #TechExplained

---

💡 **Optimal Solution?**
Instead of traversing the whole tree, we stop as soon as we reach the Kth smallest element. No need to visit the remaining nodes! 🍃 #Efficiency #BinarySearchTree #SmartTraversal

---

🔍 **Algorithm Breakdown**:

1. Traverse the tree in-order (left -> root -> right).
2. Keep count of nodes you visit.
3. When count == K, return the current node's value. Boom, done! 🎯 #CodingChallenge #InOrderTraversal

---

🖥️ **Python Code**: Here's a clean solution in Python using a recursive approach:

python
Copier le code

```python
class Solution:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        self.count = 0
        self.result = None

        def inorder(node):
            if node:
                inorder(node.left)
                self.count += 1
                if self.count == k:
```

```
                self.result = node.val
                return
            inorder(node.right)

    inorder(root)
    return self.result
```

🔑 In this code:

- **count** tracks how many nodes we've visited.
- **result** stores the Kth smallest value when we find it. #CodeSnippet #Python

---

🔥 **Why does it work?**
In an in-order traversal of a BST, we process nodes in ascending order. By keeping track of our count, the Kth node we visit will be the Kth smallest. Simple but powerful! #TreeTraversal #BinarySearchTree #AlgorithmExplained

---

🔄 **Edge Cases?**

1. K = 1 → Just return the **smallest** element (leftmost).
2. K = n → Return the **largest** element (rightmost).
3. Empty tree? Handle gracefully with proper checks. 🚨 #EdgeCases #TestingYourCode

---

🚀 **Time Complexity**:

- Best case: **O(H)**, where H is the height of the tree (balanced tree).
- Worst case: **O(N)** if the tree is unbalanced (like a linked list).
- Space complexity: **O(H)** for the recursive call stack. #TimeComplexity #BigO

✨ **Pro Tip**: If you want to optimize further for large values of K, you can use a **reverse in-order traversal** for finding the **Kth largest** element. #Optimization #BST #CodingTips

---

🎉 **Conclusion**: The Kth smallest element in a BST is easier than it sounds. With in-order traversal, it's a walk in the park! 🌳 Now go ahead and try it on your own. #HappyCoding #BST #InterviewPrep

```
/**
 * 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:
    int kthSmallest(TreeNode* root, int k) {

    }
};
```

*******************solution **

```
# Method 1: recursive in-order traversal
def kthSmallest1(self, root, k):
    inorder = []
    def helper(root):
        if root:
            helper(root.left)
            inorder.append(root.val)
            helper(root.right)

    helper(root)
    return inorder[k-1]

# Method 2: iterative in-order traversal
def kthSmallest2(self, root, k):
    stack = []
    while root or stack:
        while root:
            stack.append(root)
            root = root.left
        root = stack.pop()
        k -= 1
        if k == 0: return root.val
        root = root.right
```