Solution Review: Find kth maximum value in Binary Search Tree

This review provides a detailed analysis to solve the Find kth maximum value in Binary Search Tree challenge

We'll cover the following

Solution #1: Sorting the tree in order

#

main.py

BinarySearchTree.py

Node.py

```
from Node import Node
from BinarySearchTree import BinarySearchTree


def findKthMax(root, k):
    tree = []
    inOrderTraverse(root, tree)  # Get sorted tree lis
    if ((len(tree)-k) >= 0) and (k > 0):  # check if k
        return tree[-k]  # return the kth max value
    return None  # return none if no value found


def inOrderTraverse(node, tree):
    # Helper recursive function to traverse the tree inorder
    if node is not None:  # check if node exists
        inOrderTraverse(node.leftChild, tree)  # traverse left sub-tr
        if len(tree) is 0:
            # Append if empty tree
            tree.append(node.val)
        elif tree[-1] is not node.val:
            # Ensure not a duplicate
            tree.append(node.val)  # add current node value
        inOrderTraverse(node.rightChild, tree)  # traverse right sub-


BST = BinarySearchTree(6)
BST.insert(1)
BST.insert(133)
```

This is a quick and easy naive solution for this problem. In this solution, to find the **kth** *max* value, we first perform an In-Order Traversal on the tree to get a sorted array in *ascending* order. Before appending to the list, we cross-check with the last element for equality to avoid duplicates. Once, we have the sorted array, we can easily find the *kth* max. value by accessing the index **[length - k]**. To perform the in-order traversal on the tree, we use a helper recursive function which is a variation of the `inOrderPrint()` function that we studied in the in-Order Traversal.

Time Complexity#

The worst-case and the best-case complexity of this solution is $O(n)$ where n is the number of nodes in the tree. The reason is that no matter the value of k is, the function always traverses the entire tree!

Solution #2: Recursive Approach#

main.py

BinarySearchTree.py

Node.py

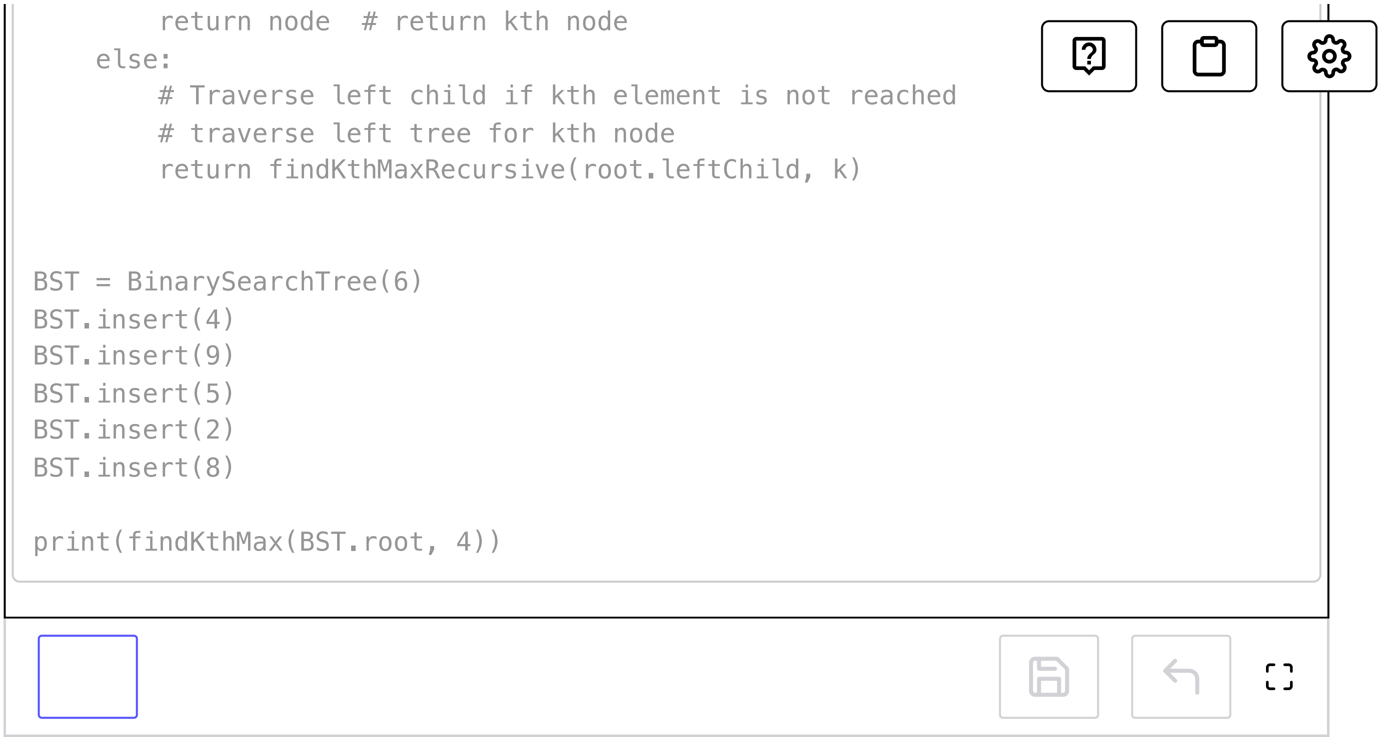
```
from Node import Node
from BinarySearchTree import BinarySearchTree


def findKthMax(root, k):
    if k < 1:
        return None
    node = findKthMaxRecursive(root, k)  # get the node at kth position
    if(node is not None):  # check if node received
        return node.val  # return kth node value
    return None  # return None if no node found


counter = 0  # global count variable
current_max = None

def findKthMaxRecursive(root, k):
    global counter  # use global counter to track k
    global current_max # track current max
    if(root is None):  # check if root exists
        return None

    # recurse to right for max node
    node = findKthMaxRecursive(root.rightChild, k)
    if(counter is not k) and (root.val is not current_max):
        # Increment counter if kth element is not found
        counter += 1
        current_max = root.val
        node = root
    elif current_max is None:
        # Increment counter if kth element is not found
        # and there is no current_max set
        counter += 1
        current_max = root.val
        node = root
    # Base condition reached as kth largest is found
    if(counter == k):
```



```
        return node  # return kth node
    else:
        # Traverse left child if kth element is not reached
        # traverse left tree for kth node
        return findKthMaxRecursive(root.leftChild, k)


BST = BinarySearchTree(6)
BST.insert(4)
BST.insert(9)
BST.insert(5)
BST.insert(2)
BST.insert(8)

print(findKthMax(BST.root, 4))
```

Explanation#

This approach is more efficient than the previous solution. In this solution, we have used a helper function called `findkthMaxRecursive()` and the function `findKthMax()` acts as a *wrapper* for this helper function. In the recursive function we first recursively traverse the tree in a **right to left** fashion because the maximum element is present in the *right-most* leaf node. We have also kept a global variable called `counter` which gets incremented after we have found the *maximum* element and `current_max` variable to track the previous value. The counter is incremented each time if `kth` maximum node is not found and the node value is not `current_max` to cater to duplicates. The **base condition** is reached when `k` becomes equal to the `counter`. This node is then returned to the wrapper function, and if the node is not `None`, then its value is returned.

Time Complexity#

The worst-case complexity of this solution is the same as the previous solution, i.e $O(n)$. But for the best-case scenario, when k = 1, the complexity of this solution will be $O(h)$ where h is the height of the tree re o average, this solution is more efficient than the previous one.

← Back

Challenge 2: Find kth maximum value i...

Next →

Challenge 3: Find Ancestors of a given...

☑ Completed

Report an Issue