K’th Largest Element in BST when modification to BST is not allowed

```
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.key = data
        self.left = None
        self.right = None

# A function to find k'th largest
# element in a given tree.
def kthLargestUtil(root, k, c):

    # Base cases, the second condition
    # is important to avoid unnecessary
    # recursive calls
    if root == None or c[0] >= k:
        return

    # Follow reverse inorder traversal
    # so that the largest element is
    # visited first
    kthLargestUtil(root.right, k, c)

    # Increment count of visited nodes
    c[0] += 1

    # If c becomes k now, then this is
    # the k'th largest
    if c[0] == k:
        print("K'th largest element is",
                                root.key)
        return

    # Recur for left subtree
    kthLargestUtil(root.left, k, c)

# Function to find k'th largest element
```

```
def kthLargest(root, k):

    # Initialize count of nodes
    # visited as 0
    c = [0]

    # Note that c is passed by reference
    kthLargestUtil(root, k, c)

# A utility function to insert a new
# node with given key in BST */
def insert(node, key):

    # If the tree is empty,
    # return a new node
    if node == None:
        return Node(key)

    # Otherwise, recur down the tree
    if key < node.key:
        node.left = insert(node.left, key)
    elif key > node.key:
        node.right = insert(node.right, key)

    # return the (unchanged) node pointer
    return node
```

This is reverse Inorder Traversal.

Method 2: Augmented Tree Data Structure (O(h) Time Complexity and O(h) auxiliary space)

The idea is to maintain the rank of each node. We can keep track of elements in the left subtree of every node while building the tree. Since we need the K-th smallest element, we can maintain the number of elements of the left subtree in every node.
Assume that the root is having 'lCount' nodes in its left subtree. If K = lCount + 1, root is K-th node. If K < lCount + 1, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > lCount + 1, we continue our search in the right subtree for the (K – lCount – 1)-th smallest element. Note that we need the count of elements in the left subtree only.

```
class newNode:

    def __init__(self, x):
```

```
        self.data = x
        self.left = None
        self.right = None
        self.lCount = 0

# Recursive function to insert
# an key into BST
def insert(root, x):

    if (root == None):
        return newNode(x)

    # If a node is inserted in left subtree,
    # then lCount of this node is increased.
    # For simplicity, we are assuming that
    # all keys (tried to be inserted) are
    # distinct.
    if (x < root.data):
        root.left = insert(root.left, x)
        root.lCount += 1

    elif (x > root.data):
        root.right = insert(root.right, x);

    return root

# Function to find k'th largest element
# in BST. Here count denotes the number
# of nodes processed so far
def kthSmallest(root, k):

    # Base case
    if (root == None):
        return None

    count = root.lCount + 1

    if (count == k):
        return root

    if (count > k):
        return kthSmallest(root.left, k)
```

```
    # Else search in right subtree
    return kthSmallest(root.right, k - count)

# Driver code
if __name__ == '__main__':

    root = None
    keys = [ 20, 8, 22, 4, 12, 10, 14 ]

    for x in keys:
        root = insert(root, x)

    k = 4
    res = kthSmallest(root, k)

    if (res == None):
        print("There are less than k nodes in the BST")
    else:
        print("K-th Smallest Element is", res.data)
```