Problem: Design a class to find the **k**th largest element in a stream.

We have mentioned the basic idea of the algorithm using the BST in the previous article. Here we provide java and C++ codes for your reference.

|  |
| --- |
| class KthLargest {  // insert a node into the BST  private Node insertNode(Node root, int num) {  if (root == null) {  return new Node(num, 1);  }  if (root.val < num) {  root.right = insertNode(root.right, num);  } else {  root.left = insertNode(root.left, num);  }  root.cnt++;  return root;  }  private int searchKth(Node root, int k) {  // m = the size of right subtree  int m = root.right != null ? root.right.cnt : 0;  // root is the m+1 largest node in the BST  if (k == m + 1) {  return root.val;  }  if (k <= m) {  // find kth largest in the right subtree  return searchKth(root.right, k);  } else {  // find (k-m-1)th largest in the left subtree  return searchKth(root.left, k - m - 1);  }  }    private Node root;  private int m\_k;  public KthLargest(int k, int[] nums) {  root = null;  for (int i = 0; i < nums.length; ++i) {  root = insertNode(root, nums[i]);  }  m\_k = k;  }    public int add(int val) {  root = insertNode(root, val);  return searchKth(root, m\_k);  }  }  class Node { // the structure for the tree node  Node left;  Node right;  int val;  int cnt; // the size of the subtree rooted at the node  public Node (int v, int c) {  left = null;  right = null;  val = v;  cnt = c;  }  }  /\*\*  \* Your KthLargest object will be instantiated and called as such:  \* KthLargest obj = new KthLargest(k, nums);  \* int param\_1 = obj.add(val);  \*/ |

By using a BST, the time complexity for insertion and search are both O(h). The time complexity of performing all the operations will be O(N\*h). That is, O(N^2) in the worst case and O(NlogN) ideally.

If the BST is well organized, you can always keep h = logN, where h is the height of the tree and N is the total number of nodes. In that case, you can reduce the time complexity of search, insertion and deletion to O(logN) which is really a considerable solution in many cases. That's the idea of the height-balanced BST. We will introduce height-balanced BST in later chapters.

As we mentioned before, this algorithm is suitable if we have to handle both insertion and search operations.