703. Kth Largest Element in a Stream Heap (Priority Queue) Binary Tree Data Stream Tree Design Binary Search Tree Easy

## **Problem Description**

Importantly, when we refer to the kth largest element, we mean according to sorted order, not that the element is distinct from the others. The numbers can be repeated in the stream. The class should be able to handle two types of actions:

In this problem, we are asked to design a KthLargest class that can determine the kth largest element in a stream of numbers.

Leetcode Link

- 1. Initialization (\_\_init\_\_): When an instance of the class is created, it should be initialized with an integer k and an array of integers nums. The k represents the position of the largest element we are interested in, and nums is the initial stream of numbers.
- 2. Adding New Elements (add): The class should provide a method to add a new integer val to the stream. After adding the new integer, this method should return the current kth largest element from the stream.

### The reason behind choosing a Min Heap is its useful property: the smallest element is always at the root and can be accessed in constant time. If we maintain a Min Heap of exactly k largest elements from the current stream, the root of the heap (the smallest element in the heap) is our desired kth largest element.

Intuition

element and then pick the kth largest. However, sorting every time we insert a new element leads to a less efficient solution. To solve the problem more optimally, we can use a data structure that does the hard work for us - a Min Heap.

The straightforward approach to find the k<sup>th</sup> largest element would be to sort the stream of numbers each time we insert a new

1. Initialize a Min Heap (self.q) and a variable self.size to store k. 2. Insert the initial stream of numbers to the Min Heap using the add method. During each insert, we push the new value into the heap. ∘ If the heap size exceeds k, it means we have more than k elements, so we can remove the smallest one (the root).

The k<sup>th</sup> largest element will then be the new root of the heap.

heappush(self.q, val) is used to add the new value val to the Min Heap.

This keeps the heap size at max k elements at all times.

Here's how we construct our solution using this intuition:

- 3. The add method maintains the Min Heap invariant and ensures it always contains the k largest elements after each insertion. 4. Finally, we can return the kth largest element by looking at the root of the Min Heap.
- Using this approach, we avoid the need to sort the entire stream every time, thereby improving the efficiency of finding the kth largest element in the stream. Solution Approach
- To implement the solution, we employ Python's heapq module, which provides an implementation of the Min Heap data structure. Here's a step-by-step breakdown of the implementation:

k is stored in self.size which represents the position of the kth largest element we are interested in. self.q is initialized as an empty list, which will be used as the Min Heap. 2. The nums list is iterated over and each number is added to the Min Heap using the add method.

3. The add method handles inserting a new element into the Min Heap and ensures the Min Heap contains only the k largest

1. When an instance of KthLargest class is created, the \_\_init\_\_ method is called with two parameters: k and nums.

# We check if the size of the heap exceeds k by comparing len(self.q) to self.size.

also in the heap.

By following this approach, we ensure that:

array each time a new element is added.

Let us consider an example to illustrate the solution approach.

self.size to 3 and self.q as an empty Min Heap.

Add 8: self.q becomes [4, 5, 8].

be removed if the heap's size exceeds k.

elements and always be able to find the 3rd largest element in the current stream.

Here is a step-by-step walkthrough of how the KthLargest class processes the stream of numbers:

Add 2: self.q remains [4, 5, 8] because 2 is not among the 3 largest elements.

Example Walkthrough

elements.

 If the heap size is greater than k, we remove the smallest element (root of the Min Heap) using heappop(self.q) to ensure that only the k largest elements remain.

• The root element, which can be accessed with self.q[0], is the kth largest element because all elements bigger than it are

The add operation has a time complexity of O(log k) since it involves heap operations which work in logarithmic time relative to

This provides a balanced and efficient way to continuously find the k<sup>th</sup> largest element in the stream without having to sort the entire

Suppose k is 3, and the initial array of numbers nums is [4, 5, 8, 2]. We want to design a KthLargest class where we can add

1. We create an instance of KthLargest by passing in our values k=3 and nums=[4,5,8,2]. The \_\_init\_\_ method initializes

the number of elements in the heap. The space complexity of the solution is O(k) because the Min Heap holds at most k elements at any given time.

most k largest elements. Initially, the Min Heap (self.q) is empty, and we proceed as follows: Add 4: self.q becomes [4]. Add 5: self.q becomes [4, 5].

2. We add each element of nums into the Min Heap using the add method, ensuring that after every addition, the Min Heap holds at

#### 3. If we call the add method to insert a new element, let's say 3, we again ensure that the Min Heap only contains the k largest elements.

upon each insertion.

10

11

12

13

15

16

17

18

19

29

8

9

10

16

17

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

32

33

35

39

40

10

11

12

13

14

15

16

18

21

22

23

24

25

26

33

34

36

35 }

17 }

34 };

36 /\*\*

\*/

return minHeap.top();

Typescript Solution

let minHeap: MinHeap;

data: number[];

interface MinHeap {

let kthLargestSize: number;

heapify: () => void;

size: () => number;

kthLargestSize = k;

nums.forEach(num => {

return minHeap.peek()!;

minHeap = createMinHeap();

kthLargestAdd(num);

comparator: ComparatorFunction;

offer: (value: number) => void;

bubbleUp: (index: number) => void;

bubbleDown: (index: number) => void;

// Initializing the Kth largest element finder

swap: (index1: number, index2: number) => void;

function KthLargestInit(k: number, nums: number[]): void {

// Create a MinHeap with default values and a comparator

peek: () => number | null;

poll: () => number | null;

\* The provided code snippet illustrates the use of the KthLargest class.

type ComparatorFunction = (a: number, b: number) => number;

\* It demonstrates the instantiation of a KthLargest object and subsequent calls to its add method.

Python Solution

from heapq import heappush, heappop

self.k = k

for num in nums:

self.add(num)

26 # print(kthLargest.add(10)) # returns 5

27 # print(kthLargest.add(9)) # returns 8

28 # print(kthLargest.add(4)) # returns 8

import java.util.PriorityQueue;

def add(self, val: int) -> int:

heappush(self.min\_heap, val)

if len(self.min\_heap) > self.k:

heappop(self.min\_heap)

# Store the size k to know the kth largest value

# The root of the min heap is the kth largest value

// Class to find the kth largest element in a stream of numbers

private PriorityQueue<Integer> minHeap;

public KthLargest(int k, int[] nums) {

for (int num : nums) {

add(num);

public int add(int val) {

minHeap.offer(val);

if (minHeap.size() > k) {

this.minHeap = new PriorityQueue<>(k);

// Priority queue to maintain the smallest 'k' elements seen so far

// Initialize a min-heap with the capacity to hold 'k' elements

\* Adds a new number into the stream and returns the kth largest element.

// If the size of the min-heap exceeds 'k', remove the smallest element

// Add the initial elements to the kth largest tracker

\* @param val The new number to be added to the stream.

// Always add the new value to the min-heap

\* @return The kth largest element after adding the new number.

The kth position to track in the list of largest elements.

\* @param nums An array of initial numbers to be added to the kth largest tracker.

// The kth position we are interested in for the largest element

# If the size of the heap exceeds k, remove the smallest element

# Add the initial elements to the heap

# Add the new value to the min heap

 Add 9: self.q becomes [5, 8, 9] after removing 4 because the size exceeded k. Now the 3rd largest element is 5. So after adding the element 9, our Min Heap has elements [5, 8, 9], and the 3rd largest (kth largest) element is 5.

Now the Min Heap contains k largest elements, the kth largest (3rd largest) among them is 4 (the root of the Min Heap).

Add 3: self.q remains [4, 5, 8] because 3 is not among the 3 largest elements. The 3rd largest element is still 4.

4. Let's insert a larger element, 9. The Min Heap needs to accommodate this change, and the smallest element in the heap should

The class now effectively and efficiently maintains the 3rd largest element as we add new values into the stream. The add method

updates the Min Heap in logarithmic time, making the operation swift compared to the linear time complexity of sorting the numbers

- class KthLargest: def \_\_init\_\_(self, k: int, nums: List[int]): # Initialize a min heap to store the k largest elements self.min\_heap = []
- return self.min\_heap[0] 20 21 22 # Example usage: # kthLargest = KthLargest(3, [4, 5, 8, 2]) 24 # print(kthLargest.add(3)) # returns 4 25 # print(kthLargest.add(5)) # returns 5

```
11
12
       /**
13
        * Constructor to initialize the data structure and populate with initial elements.
14
15
        * @param k
```

\*/

/\*\*

\*/

**Java Solution** 

class KthLargest {

private int k;

this.k = k;

```
minHeap.poll();
41
42
43
           // The root of the min-heap represents the kth largest element
44
           return minHeap.peek();
45
46
47 }
48
   /* Usage example:
    * KthLargest kthLargest = new KthLargest(3, new int[]{4, 5, 8, 2});
    * kthLargest.add(3); // returns 4
    * kthLargest.add(5);
                         // returns 5
    * kthLargest.add(10); // returns 5
    * kthLargest.add(9); // returns 8
54
    * kthLargest.add(4); // returns 8
56
    */
57
C++ Solution
 1 #include <queue>
   #include <vector>
   // Define a class to find kth largest element in a stream of numbers.
  class KthLargest {
 6 public:
       // Declare a min heap to keep track of the k largest elements.
       std::priority_queue<int, std::vector<int>, std::greater<>> minHeap;
       // Store the value of 'k' to know which largest element to keep track of.
       int kthSize;
10
11
12
       // Constructor for initializing the KthLargest class.
13
       // k - The kth position to track.
       // nums - Initial list of numbers from which we find the kth largest element.
14
       KthLargest(int k, std::vector<int>& nums) {
15
           kthSize = k;
16
17
           // Add initial numbers to the heap
           for (int num : nums) {
18
19
               add(num);
20
21
22
23
       // Function to add a number to the stream and return the kth largest element.
24
       int add(int val) {
25
           // Add the new number to the min heap.
26
           minHeap.push(val);
           // If the heap size is greater than k, remove the smallest element.
28
           if (minHeap.size() > kthSize) {
29
               minHeap.pop();
30
31
           // Return the kth largest element (top of the min heap).
```

#### 27 // Function to add a new value to the data collection function kthLargestAdd(val: number): number { minHeap.offer(val); 30 31 if (minHeap.size() > kthLargestSize) { 32 minHeap.poll();

});

```
function createMinHeap(): MinHeap {
         const heap: MinHeap = {
 39
 40
             data: [],
             comparator: (a, b) \Rightarrow a - b,
 41
 42
             heapify() {
 43
                 if (this.size() < 2) return;</pre>
 44
                 for (let i = 1; i < this.size(); i++) {
 45
                     this.bubbleUp(i);
 46
 47
             },
 48
             peek() {
 49
                 return this.size() === 0 ? null : this.data[0];
 50
             },
 51
             // Adds a new value and bubbles it up to maintain heap property
 52
             offer(value) {
                 this.data.push(value);
 53
 54
                 this.bubbleUp(this.size() - 1);
 55
             // Removes and returns the root value of the heap
 56
             poll() {
 57
                 if (this.size() === 0) {
 58
 59
                     return null;
 60
 61
                 const result = this.data[0];
 62
                 const last = this.data.pop();
                 if (this.size() !== 0) {
 63
 64
                     this.data[0] = last!;
 65
                     this.bubbleDown(0);
 66
 67
                 return result;
 68
 69
             // Bubbles a value up the heap until the heap property is restored
 70
             bubbleUp(index) {
                 while (index > 0) {
 72
                     const parentIndex = (index - 1) >> 1;
 73
                     if (this.comparator(this.data[index], this.data[parentIndex]) < 0) {</pre>
 74
                          this.swap(index, parentIndex);
                          index = parentIndex;
 75
 76
                      } else {
 77
                         break;
 78
 79
 80
             // Bubbles a value down the heap until the heap property is restored
 81
 82
             bubbleDown(index) {
                 const lastIndex = this.size() - 1;
 83
 84
                 while (true) {
 85
                      let findIndex = index;
 86
                     const leftIndex = index * 2 + 1;
 87
                     const rightIndex = index * 2 + 2;
 88
                     if (
 89
                          leftIndex <= lastIndex &&</pre>
 90
                          this.comparator(this.data[leftIndex], this.data[findIndex]) < 0</pre>
 91
 92
                          findIndex = leftIndex;
 93
                     if (
 94
 95
                          rightIndex <= lastIndex &&
 96
                          this.comparator(this.data[rightIndex], this.data[findIndex]) < 0</pre>
 97
 98
                          findIndex = rightIndex;
 99
                     if (index !== findIndex) {
100
                          this.swap(index, findIndex);
101
                          index = findIndex;
102
                     } else {
103
104
                         break;
105
106
107
             // Swaps two values in the heap
108
             swap(index1, index2) {
109
                  [this.data[index1], this.data[index2]] = [this.data[index2], this.data[index1]];
110
111
112
             // Returns the number of elements in the heap
             size() {
113
                 return this.data.length;
114
115
         };
116
117
118
         heap.heapify();
119
         return heap;
120 }
121
122 // Example of usage
123 KthLargestInit(3, [4, 5, 8, 2]);
124 console.log(kthLargestAdd(3)); // Should return the kth largest element after adding 3
125 console.log(kthLargestAdd(5)); // Should return the kth largest element after adding 5
126 console.log(kthLargestAdd(10)); // Should return the kth largest element after adding 10
    console.log(kthLargestAdd(9)); // Should return the kth largest element after adding 9
128 console.log(kthLargestAdd(4)); // Should return the kth largest element after adding 4
129
Time and Space Complexity
Time Complexity:
```

### • The \_\_init\_\_(self, k: int, nums: List[int]) method has a time complexity of O(n \* log(k)) where n is the length of the nums list. This is because for every element in nums, the add operation is called which takes O(log(k)) time due to the heap operation and we perform this n times.

- The add(self, val: int) -> int method has a time complexity of O(log(k)). This is because the heappush operation could take up to O(log(k)) time to maintain the heap properties when adding a new element, and similarly heappop operation, which is called only when the heap size exceeds k, takes O(log(k)).
- elements at any time.

• The space complexity of the whole class is O(k) since the heap that is maintained by the class never contains more than k

**Space Complexity:**