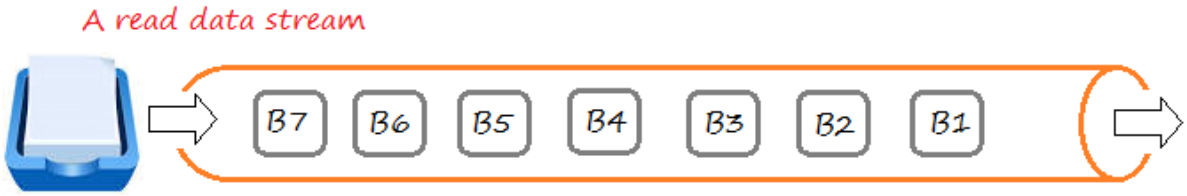
medium.com

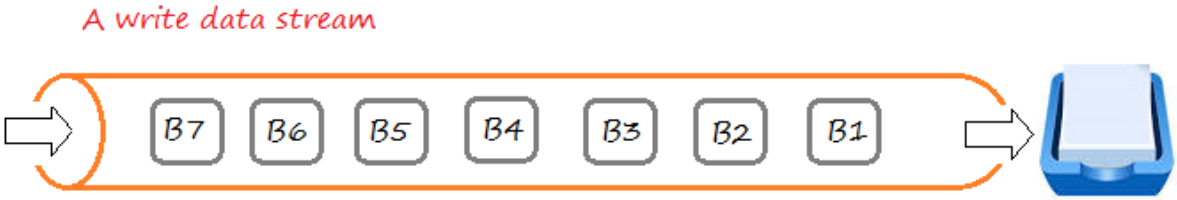
Find median from Data Stream - Algo Shaft - Medium

Kode Shaft

3-4 minutes

In this post we are gonna discuss how to find median in a stream of running integers





Problem Statement :

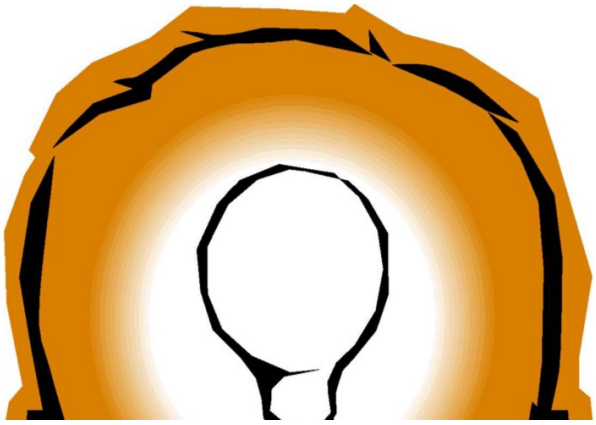
> Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.
>
> For example,
>
> [2,3,4], the median is 3
>
> [2,3], the median is (2 + 3) / 2 = 2.5

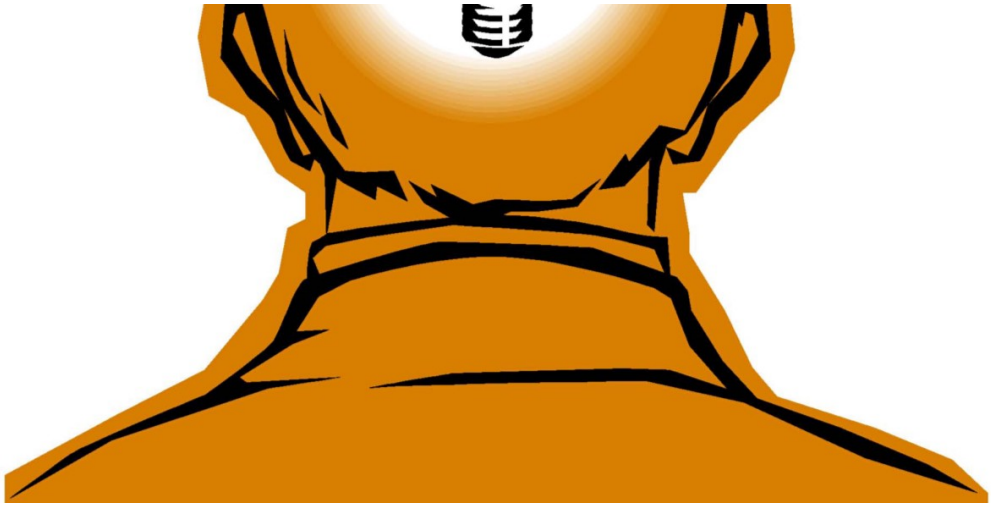
> Design a data structure that supports the following two operations:
>
> void addNum(int num) — Add a integer number from the data stream to the data structure.
>
> double findMedian() — Return the median of all elements so far.
>
> Example:
>
> addNum(1),
>
> addNum(2),
>
> findMedian() -> 1.5,
>
> addNum(3),
>
> findMedian() -> 2

Step by step solution to the problem :

First solution which comes to our mind for this problem is keeping an sorted array and whenever a new element comes put that in it's correct position in the sorted array. In this case the median of the sorted array can be our result.

The time complexity of the find() in above approach will be O(1) but while adding each time we have to increase the array size by one, copy to new array, then find median so it's quite expensive O(n). Let's think for sometime can we do better…….???





Heaps can rescue us in this situation. Idea is somehow if we can divide input numbers at every point into two half such that upper contain elements larger than lower and both the half are in sorted order, with a condition that absolute value of (no of elements in upper-no of elements in lower) will never be more than 1. Now there can be 3 cases:

a) no of elements in upper >no of elements in lower then clearly the last element in sorted upper is the median.

b) no of elements in upper <no of elements in lower then clearly the first element in sorted lower is the median.

c) no of elements in upper =no of elements in lower then median is (last element in sorted upper + first element in sorted lower)/2;

Initialization: We can implement upper by using minHeap and lower using MaxHeap.

Add:

Time complexity of this is O(logn).

Cases:

a) if both the heap is empty we are adding first element to minHeap(we can add to maxHeap also).

b) If **num** is < minHeap (which stored upper half in decreasing order) peak element , that means **num** has no place in minHeap as of now. So it can be placed in maxHeap provided **maxQueue.size()≤minQueue.size().**

Otherwise we pop the top element **maxTop** from maxHeap and compare it with **num,** then place minimum of (**maxTop,num)** to maxHeap and maximum of (**maxTop,num)** to minHeap.

c) If **num** is > minHeap (which stored upper half in decreasing order) peak element , that means **num** has no place in maxHeap as of now. So it can be placed in minHeap provided **maxQueue.size()≥minQueue.size().**

Otherwise we pop the top element **minTop** from minHeap and offer to maxHeap and offer **num** to minHeap.

Median Finding:

Time complexity of this is O(1).

Cases:

a) If both the heap size are equal then median is

(maxQueue.peek()+minQueue.peek())/2.0;

b) Otherwise median is at the peak of the heap whose size is more.

The complete code for this problem can be found in https://github.com/GyanTech877/algorithms/blob/master/heap/MedianFinder.java

Happy Learning 😎