Technique: Heap

Level: Hard

Online Median: Given a stream of integers, find their median. If an integer is added to the stream, you should be able to update the median quickly.

For example,

```
3,2,1,4,5    -> Median = 3
5,6,3,7      -> Median = (5+6)/2 = 5.5
```

Questions to Clarify:

Q. What is the median if there are even number of elements?
A. Make it the average of the two middle numbers.

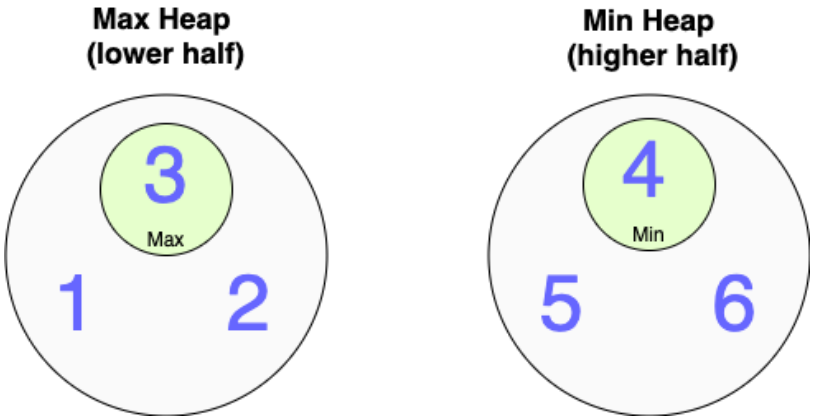
Q. What should we return if there are no items?
A. Throw an exception.

Q. Can the user ask for a median at any time?
A. Yes, you should have a function to get the median at any point.

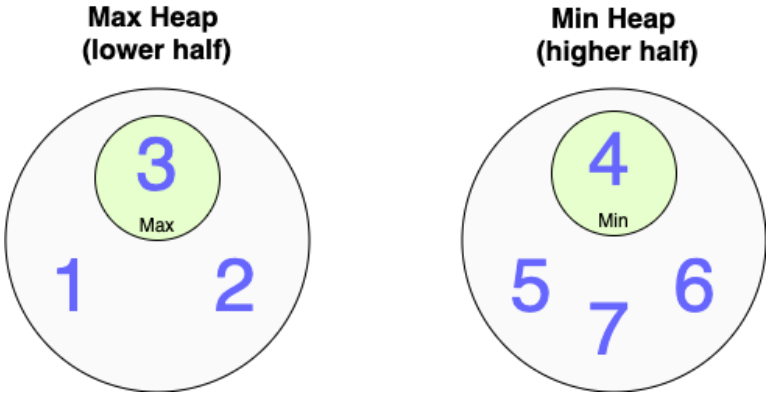
Solution:

The median element is the middle element - if the stream was sorted. One way to solve this problem is to store elements of the stream in sorted order. This would take O(n) time for every number inserted.

We can improve this by using two heaps - each storing half of the elements. The first max-heap stores the lower half, and the second min-heap stores the higher half. For example,



In the above stream, the median will be `(3+4)/2`. Let's see a case with odd number of elements. We add `7` to the above dataset, and get the following:

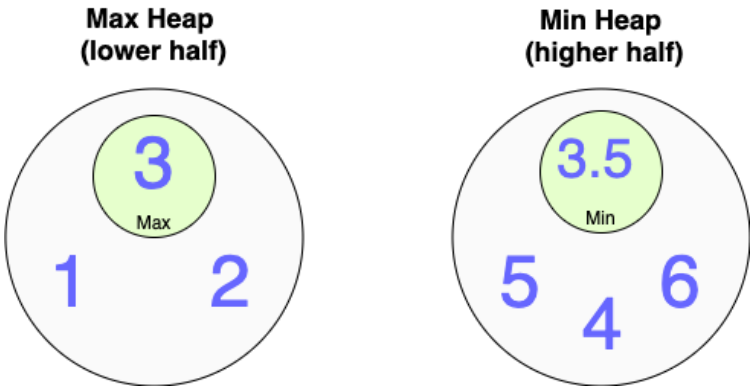


The key trick in this problem is updating the heaps. If you can figure that out, you've figured out the whole thing.

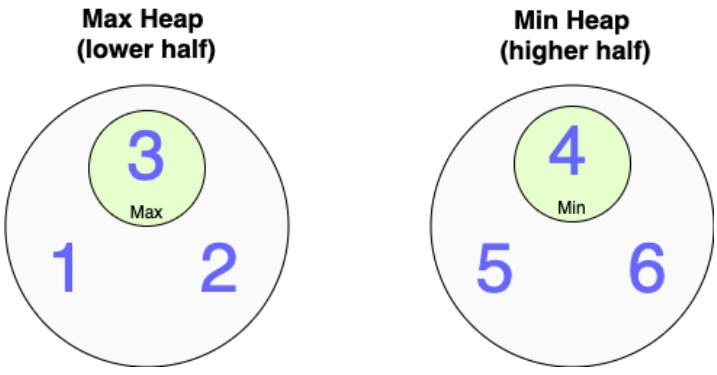
There are multiple ways to do this, but here's our favorite:

If there are odd numbers in the dataset, we ensure that the size of the high heap is one larger than the size of the low heap. So with odd elements, the median will always be on high.

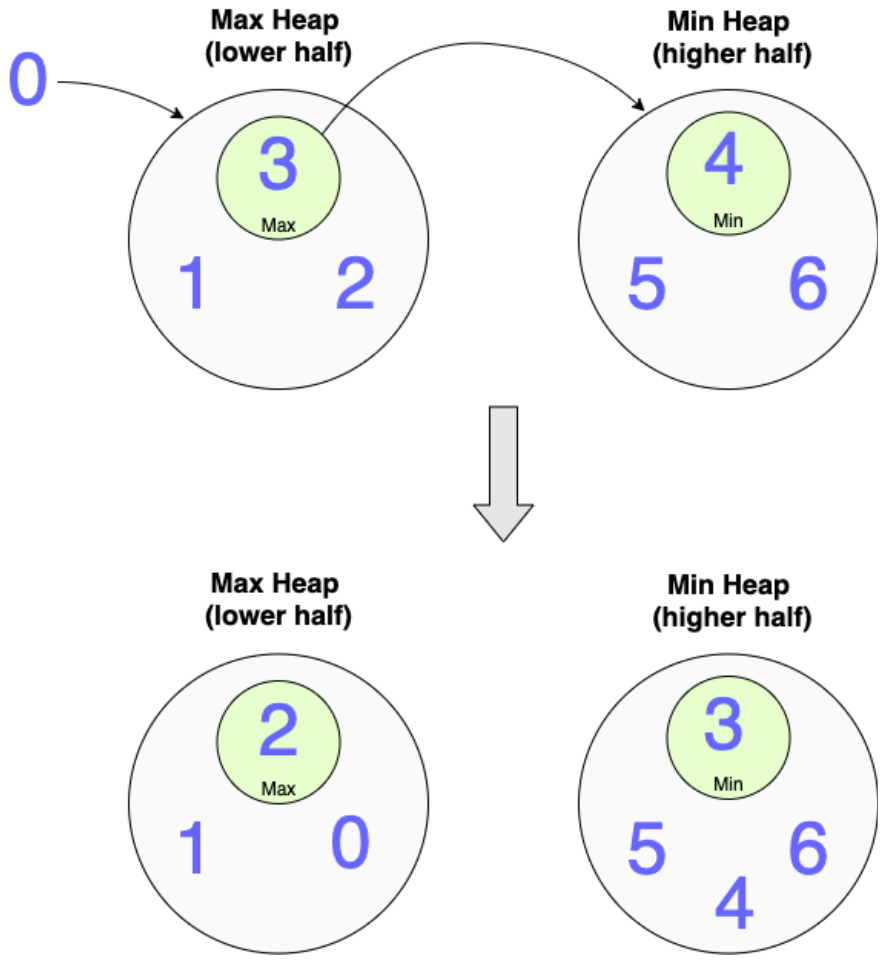
For example, let's say we added `3.5` to the first diagram. `3.5` can be on either heap, but we ensure that it goes on the high heap.



But that's a convenient example. What if we added `0` instead?



0 should go on low. If we just add 0 to low, low will become larger than the high. We don't want that. So before placing 0 on low, we move the top element from low to high. We can then add 0 to low, because now low won't exceed high.



So when we add a new element, we ensure:

1. The heaps are equal in size, or
2. high is one larger than low.

We need to consider several test cases in this solution. They are outlined below.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a few lines quickly to plan out your solution. Your actual solution should be in a real language and use good syntax.)

low = empty max heap
high = empty min heap
```

```
getMedian()
     if low and high are equal in size:
         return average of both's top values

     otherwise, return high's top value

insert(number)
     if high is empty:
         this is the first item, add number to high

     if low and high are equal in size:
         if number is smaller than low.top, we need to place it in low:
             remove low's top element and place it on high
             add number to low
         else:
             add number to high
      else: (high has 1 number more than low)
         if number is greater than high.top, it needs to go onto high:
             move high's top element to low
             add number to high
         else:
             add number to low
```

Test Cases:
Edge Cases: empty stream
Base Cases: 1 element in stream, 2 elements in stream
Regular Cases: even elements in stream, odd elements in stream,
adding elements in lower half, higher half,
equal to lower half's top, equal to higher half's top

Time Complexity:
Insert: O(logn)
Median Lookup: O(1)

Space Complexity: O(n)

```
public static class RunningMedian {

    private static PriorityQueue<Integer> low;
    private static PriorityQueue<Integer> high;

    public RunningMedian() {
        low = new PriorityQueue<Integer>(Collections.reverseOrder()); // max heap
        high = new PriorityQueue<Integer>(); // min heap
    }
```

```
    public static double getMedian() throws EmptyDatasetException {
        if (low.size() == 0 && high.size() == 0)
            throw new EmptyDatasetException();

        if (low.size() == high.size())
            return low.peek() + (high.peek() - low.peek())/2.0;

        return high.peek();
    }

    public static void insert(int number) {
        if (high.isEmpty()) {
            high.add(number);
            return;
        }

        if (low.size() == high.size()) {
            if (number < low.peek()) {
                high.add(low.remove());
                low.add(number);
            } else {
                high.add(number);
            }
        } else {
            if (number > high.peek()) {
                low.add(high.remove());
                high.add(number);
            } else {
                low.add(number);
            }
        }
    }

}

public static class EmptyDatasetException extends Exception {

}
```