Find the Median of a Number Stream (medium)

We'll cover the following

Problem Statement

Design a class to calculate the median of a number stream. The class should have the following two methods:

1. `insertNum(int num)` : stores the number in the class
2. `findMedian()` : returns the median of all numbers inserted in the class

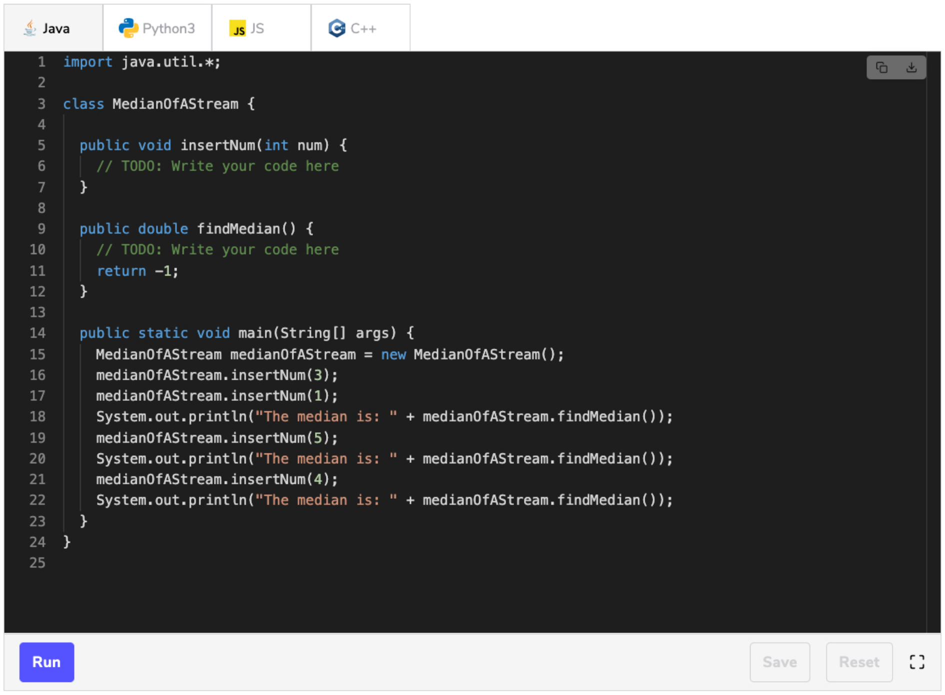
If the count of numbers inserted in the class is even, the median will be the average of the middle two numbers.

Example 1:

```
1. insertNum(3)
2. insertNum(1)
3. findMedian() -> output: 2
4. insertNum(5)
5. findMedian() -> output: 3
6. insertNum(4)
7. findMedian() -> output: 3.5
```

Try it yourself

Try solving this question here:



```java
import java.util.*;

class MedianOfAStream {

  public void insertNum(int num) {
    // TODO: Write your code here
  }

  public double findMedian() {
    // TODO: Write your code here
    return -1;
  }

  public static void main(String[] args) {
    MedianOfAStream medianOfAStream = new MedianOfAStream();
    medianOfAStream.insertNum(3);
    medianOfAStream.insertNum(1);
    System.out.println("The median is: " + medianOfAStream.findMedian());
    medianOfAStream.insertNum(5);
    System.out.println("The median is: " + medianOfAStream.findMedian());
    medianOfAStream.insertNum(4);
    System.out.println("The median is: " + medianOfAStream.findMedian());
  }
}
```

Solution

As we know, the median is the middle value in an ordered integer list. So a brute force solution could be to maintain a sorted list of all numbers inserted in the class so that we can efficiently return the median whenever required. Inserting a number in a sorted list will take $O(N)$ time if there are 'N' numbers in the list. This insertion will be similar to the Insertion sort. Can we do better than this? Can we utilize the fact that we don't need the fully sorted list - we are only interested in finding the middle element?

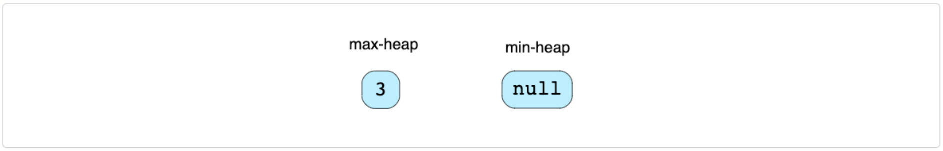
Assume 'x' is the median of a list. This means that half of the numbers in the list will be smaller than (or equal to) 'x' and half will be greater than (or equal to) 'x'. This leads us to an approach where we can divide the list into two halves: one half to store all the smaller numbers (let's call it `smallNumList`) and one half to store the larger numbers (let's call it `largNumList`). The median of all the numbers will either be the largest number in the `smallNumList` or the smallest number in the `largNumList`. If the total number of elements is even, the median will be the average of these two numbers.

The best data structure that comes to mind to find the smallest or largest number among a list of numbers is a Heap. Let's see how we can use a heap to find a better algorithm.

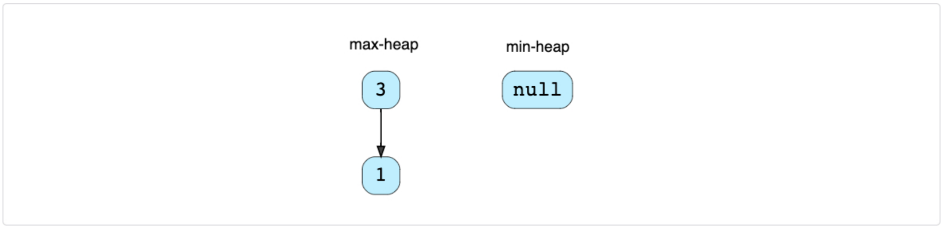
1. We can store the first half of numbers (i.e., `smallNumList`) in a **Max Heap**. We should use a **Max Heap** as we are interested in knowing the largest number in the first half.
2. We can store the second half of numbers (i.e., `largeNumList`) in a **Min Heap**, as we are interested in knowing the smallest number in the second half.
3. Inserting a number in a heap will take $O(logN)$, which is better than the brute force approach.
4. At any time, the median of the current list of numbers can be calculated from the top element of the two heaps.

Let's take the Example-1 mentioned above to go through each step of our algorithm:

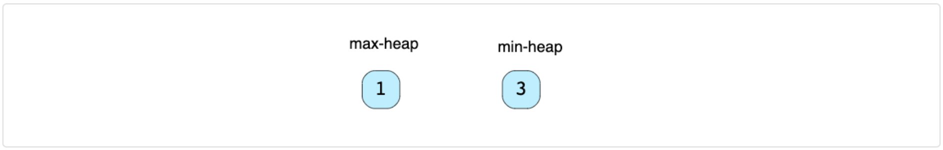
1. `insertNum(3)`: We can insert a number in the **Max Heap** (i.e. first half) if the number is smaller than the top (largest) number of the heap. After every insertion, we will balance the number of elements in both heaps, so that they have an equal number of elements. If the count of numbers is odd, let's decide to have more numbers in max-heap than the **Min Heap**.



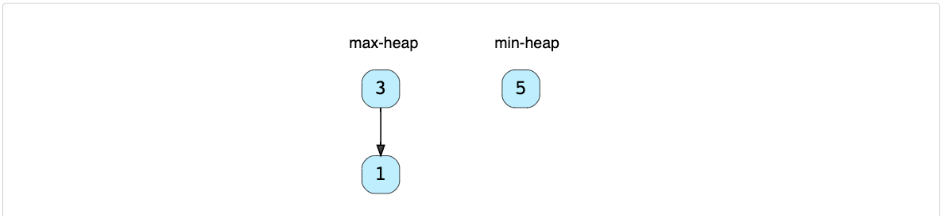
2. `insertNum(1)`: As '1' is smaller than '3', let's insert it into the **Max Heap**.



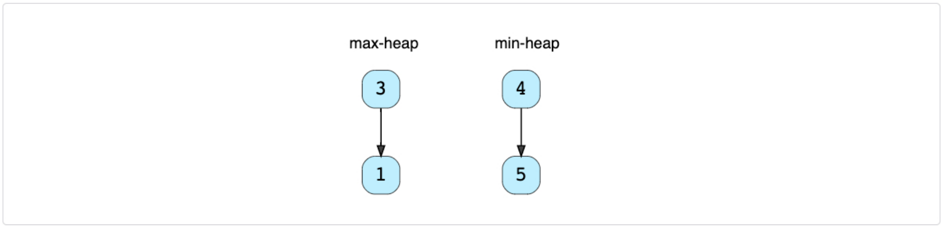
Now, we have two elements in the **Max Heap** and no elements in **Min Heap**. Let's take the largest element from the **Max Heap** and insert it into the **Min Heap**, to balance the number of elements in both heaps.



3. `findMedian()`: As we have an even number of elements, the median will be the average of the top element of both the heaps -> $(1 + 3)/2 = 2.0$
4. `insertNum(5)`: As '5' is greater than the top element of the **Max Heap**, we can insert it into the **Min Heap**. After the insertion, the total count of elements will be odd. As we had decided to have more numbers in the **Max Heap** than the **Min Heap**, we can take the top (smallest) number from the **Min Heap** and insert it into the **Max Heap**.



5. `findMedian()`: Since we have an odd number of elements, the median will be the top element of **Max Heap** -> 3. An odd number of elements also means that the **Max Heap** will have one extra element than the **Min Heap**.
6. `insertNum(4)`: Insert '4' into **Min Heap**.



7. `findMedian()`: As we have an even number of elements, the median will be the average of the top element of both the heaps -> $(3 + 4)/2 = 3.5$

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```java
import java.util.*;

class MedianOfAStream {

  PriorityQueue<Integer> maxHeap; //containing first half of numbers
  PriorityQueue<Integer> minHeap; //containing second half of numbers

  public MedianOfAStream() {
    maxHeap = new PriorityQueue<>((a, b) -> b - a);
    minHeap = new PriorityQueue<>((a, b) -> a - b);
  }

  public void insertNum(int num) {
    if (maxHeap.isEmpty() || maxHeap.peek() >= num)
      maxHeap.add(num);
    else
      minHeap.add(num);

    // either both the heaps will have equal number of elements or max-heap will have one
    // more element than the min-heap
    if (maxHeap.size() > minHeap.size() + 1)
      minHeap.add(maxHeap.poll());
    else if (maxHeap.size() < minHeap.size())
      maxHeap.add(minHeap.poll());
  }

  public double findMedian() {
    if (maxHeap.size() == minHeap.size()) {
```

Run | Save | Reset

Time complexity

The time complexity of the `insertNum()` will be $O(logN)$ due to the insertion in the heap. The time complexity of the `findMedian()` will be $O(1)$ as we can find the median from the top elements of the heaps.

Space complexity

The space complexity will be $O(N)$ because, as at any time, we will be storing all the numbers.