Find Median from Data Stream

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.

Examples:
`[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.

For example:

```
add(1)
add(2)
findMedian() -> 1.5
add(3)
findMedian() -> 2
```

Credits:
Special thanks to @Louis1992 for adding this problem and creating all test cases.

Solution 1

I keep two heaps (or priority queues):

- Max-heap `small` has the smaller half of the numbers.
- Min-heap `large` has the larger half of the numbers.

This gives me direct access to the one or two middle values (they're the tops of the heaps), so getting the median takes O(1) time. And adding a number takes O(log n) time.

Supporting both min- and max-heap is more or less cumbersome, depending on the language, so I simply negate the numbers in the heap in which I want the reverse of the default order. To prevent this from causing a bug with -2^{31} (which negated is itself, when using 32-bit ints), I use integer types larger than 32 bits.

Using larger integer types also prevents an overflow error when taking the mean of the two middle numbers. I think almost all solutions posted previously have that bug.

Update: These are pretty short already, but by now I wrote even shorter ones.

Java

```java
class MedianFinder {

    private Queue<Long> small = new PriorityQueue(),
                        large = new PriorityQueue();

    public void addNum(int num) {
        large.add((long) num);
        small.add(-large.poll());
        if (large.size() < small.size())
            large.add(-small.poll());
    }

    public double findMedian() {
        return large.size() > small.size()
               ? large.peek()
               : (large.peek() - small.peek()) / 2.0;
    }
};
```

Props to larrywang2014's solution for making me aware that I can use Queue in the declaration instead of PriorityQueue (that's all I got from him, though (just saying because I just saw he changed his previously longer addNum and it's now equivalent to mine)).

C++

```cpp
class MedianFinder {
    priority_queue<long> small, large;
public:

    void addNum(int num) {
        small.push(num);
        large.push(-small.top());
        small.pop();
        if (small.size() < large.size()) {
            small.push(-large.top());
            large.pop();
        }
    }

    double findMedian() {
        return small.size() > large.size()
               ? small.top()
               : (small.top() - large.top()) / 2.0;
    }
};
```

Big thanks to jianchao.li.fighter for telling me that C++'s priority_queue is a max-queue (see comments below).

Python

```python
from heapq import *

class MedianFinder:

    def __init__(self):
        self.heaps = [], []

    def addNum(self, num):
        small, large = self.heaps
        heappush(small, -heappushpop(large, num))
        if len(large) < len(small):
            heappush(large, -heappop(small))

    def findMedian(self):
        small, large = self.heaps
        if len(large) > len(small):
            return float(large[0])
        return (large[0] - small[0]) / 2.0
```

written by StefanPochmann original link here

Solution 2

Not sure why it is marked as hard, i think this is one of the easiest questions on leetcode.

```
class MedianFinder {
    // max queue is always larger or equal to min queue
    PriorityQueue<Integer> min = new PriorityQueue();
    PriorityQueue<Integer> max = new PriorityQueue(1000, Collections.reverseOrder
());
    // Adds a number into the data structure.
    public void addNum(int num) {
        max.offer(num);
        min.offer(max.poll());
        if (max.size() < min.size()){
            max.offer(min.poll());
        }
    }

    // Returns the median of current data stream
    public double findMedian() {
        if (max.size() == min.size()) return (max.peek() + min.peek()) /  2.0;
        else return max.peek();
    }
};
```

written by kennethliaoke original link here

Solution 3

Same idea as before, but really exploiting the symmetry of the two heaps by switching them whenever a number is added. Still O(log n) for adding and O(1) for median. Partially inspired by peisi's updated solution.

Update: Added a new Java version (the first one).

Java

```java
class MedianFinder {

    Queue<Integer> q = new PriorityQueue(), z = q, t,
                   Q = new PriorityQueue(Collections.reverseOrder());

    public void addNum(int num) {
        (t=Q).add(num);
        (Q=q).add((q=t).poll());
    }

    public double findMedian() {
        return (Q.peek() + z.peek()) / 2.;
    }
};
```

Or:

```java
class MedianFinder {

    Queue[] q = {new PriorityQueue(), new PriorityQueue(Collections.reverseOrder(
))};
    int i = 0;

    public void addNum(int num) {
        q[i].add(num);
        q[i^=1].add(q[i^1].poll());
    }

    public double findMedian() {
        return ((int)(q[1].peek()) + (int)(q[i].peek())) / 2.0;
    }
};
```

Python

```
from heapq import *

class MedianFinder:

    def __init__(self):
        self.heaps = None, [], []
        self.i = 1

    def addNum(self, num):
        heappush(self.heaps[-self.i], -heappushpop(self.heaps[self.i], num * self
.i))
        self.i *= -1

    def findMedian(self):
        return (self.heaps[self.i][0] * self.i - self.heaps[-1][0]) / 2.0
```

Or:

```
from heapq import *

class MedianFinder:

    def __init__(self):
        self.data = 1, [], []

    def addNum(self, num):
        sign, h1, h2 = self.data
        heappush(h2, -heappushpop(h1, num * sign))
        self.data = -sign, h2, h1

    def findMedian(self):
        sign, h1, h2 = d = self.data
        return (h1[0] * sign - d[-sign][0]) / 2.0
```

written by StefanPochmann original link here