Daily Coding Problem #33

Problem

This problem was asked by Microsoft.

Compute the running median of a sequence of numbers. That is, given a stream of numbers, print out the median of the list so far on each new element.

Recall that the median of an even-numbered list is the average of the two middle numbers.

For example, given the sequence [2, 1, 5, 7, 2, 0, 5], your algorithm should print out:

```
2
1.5
2
3.5
2
```

```
2
2
```

Solution

For this problem, the trick is to use two heaps: a min-heap and a max-heap. We keep all elements smaller than the median in the max-heap and all elements larger than the median in the min-heap. We'll keep these heaps balanced so that the median is always either the root of the min-heap or the max-heap (or both).

When we encounter a new element from the stream, we'll first add it to one of our heaps: the max-heap if the element is smaller than the median, or the min-heap if it's bigger. We can make the max-heap the default heap if they're equal or there are no elements.

Then we re-balance if necessary by moving the root of the larger heap to the smaller one. It's only necessary if the a heap is larger than the other by more than 1 element.

Finally, we can print out our median: it will just be the root of the larger heap, or the average of the two roots if they're of equal size.

Since Python has really terrible support for heaps, we'll pretend we have some heap objects that have the standard interface:

```
def get_median(min_heap, max_heap):
    if len(min_heap) > len(max_heap):
        return min_heap.find_min()
    elif len(min_heap) < len(max_heap):
        return max_heap.find_max()
    else:
        min_root = min_heap.find_min()
        max_root = max_heap.find_max()
```

```
        return (min_root + max_root) / 2

def add(num, min_heap, max_heap):
    # If empty, then just add it to the max heap.
    if len(min_heap) + len(max_heap) <= 1:
        max_heap.insert(num)
        return

    median = get_median(min_heap, max_heap)
    if num > median:
        # add it to the min heap
        min_heap.insert(num)
    else:
        max_heap.insert(num)

def rebalance(min_heap, max_heap):
    if len(min_heap) > len(max_heap) + 1:
        root = min_heap.extract_min()
        max_heap.insert(root)
    elif len(max_heap) > len(min_heap) + 1:
        root = max_heap.extract_max()
        min_heap.insert(root)

def print_median(min_heap, max_heap):
    print(get_median(min_heap, max_heap))

def running_median(stream):
    min_heap = minheap()
    max_heap = maxheap()
    for num in stream:
        add(num, min_heap, max_heap)
```

```
    rebalance(min_heap, max_heap)
    print_median(min_heap, max_heap)
```

This runs in O(N) space. In terms of time, each new element takes O(log N) time to manipulate the heaps, so this will run in O(N log N) time.