Programming Assignment — Using a heap

In class, we discussed finding a median. The algorithm assumes that we are given a large array and then find the median. There is a related problem, namely, finding the current median while processing elements one by one.

Example:

- Initially we have no data, and there is no median.
- Then, we enter 1. The median is now 1.
- Then, we enter 3. The list is [1,3]. The median is now 2 $= \frac{1}{2}(1+3)$.
- Then, we enter 1. The list is [1,1,3] and the median is 1.
- Then, we enter 4. The list is [1,1,3,4] and the median is 2 $= \frac{1}{2}(1+3)$.
- Then, we enter 5. The list is [1,1,3,4,5] and the median is 3.

A more extensive example is below.

An online algorithm:

The basic idea is to break the list into two parts of about equal size so that the smaller numbers are in the left list, the larger numbers are in the right list, and it is easy to get the maximum from the left and the minimum from the right.

Fortunately, there is an implementation of heaps in Python that we can use, named heapq. heapq implements a minimum heap. As is typical for a heap, inserting and popping the smallest element takes $\log(n)$, where *n* is the number of elements in the heap. Since the underlying data structure is just a Python list, getting the number of elements in the heap is constant, as the length of an array is stored separately in a Python list.

We can use the same implementation for a maximum heap by inserting the negative of elements and negating any element that we pop.

To implement insertion, we always add to the left heap if the left heap is smaller, and if they have the same number of elements, then we add to the right heap. We have to be careful, because we need to maintain the invariant that all elements in the left heap are smaller or equal to or than all the elements in the right heap.

The algorithm guarantees that the number of elements in the left heap is either equal to the number of elements in the right heap, or one less. If the number of elements in the heaps are equal, then the medium is the average between the maximum of the left heap and the minimum of the right heap, otherwise, it is the minimum of the right heap.

Here is an example that shows what happens if we want to insert an element that is too large into the left heap.

Inserting 5, need to insert into the left, but this violates the condition

5 < 3, the minimum of the right heap

Move the minimum of the right heap into the left heap

Now insert 5 into the right heap

In effect, we incremented the number of elements in the left heap

The next insertion is happening into the right heap.

Inserting 2, need to insert into the right, but this violates the condition

2 < 3, the maximum of the right heap

Move the maximum of the left heap into the right heap

Now, insert the 2 into the left heap

Effectively, we incremented the number of elements in the left heap.

Extended Example:

```
def test():
    m = Median()
    for i in [1,3,1,4,5,6,1,2,4,3,8,9,1,2,2,1,6,7, 1, 1, 1, 1, 1,
2,1,1,1,1,1,1,1]:
        m.add(i)
        print('inserted', i)
        print(([-x for x in sorted(m.small)][::-1]), sorted(m.big))
        print('median is now', m.get_median())
```

```
>>> test()
inserted 1
[] [1]
median is now 1
inserted 3
[3] [1]
median is now 2.0
inserted 1
[1] [1, 3]
median is now 1
inserted 4
[1, 1] [3, 4]
median is now 2.0
inserted 5
[1, 1] [3, 4, 5]
median is now 3
inserted 6
[1, 1, 3] [4, 5, 6]
median is now 3.5
inserted 1
[1, 1, 1] [3, 4, 5, 6]
median is now 3
inserted 2
[1, 1, 1, 2] [3, 4, 5, 6]
median is now 2.5
inserted 4
[1, 1, 1, 2] [3, 4, 4, 5, 6]
median is now 3
inserted 3
[1, 1, 1, 2, 3] [3, 4, 4, 5, 6]
median is now 3.0
inserted 8
[1, 1, 1, 2, 3] [3, 4, 4, 5, 6, 8]
median is now 3
inserted 9
[1, 1, 1, 2, 3, 3] [4, 4, 5, 6, 8, 9]
median is now 3.5
inserted 1
[1, 1, 1, 1, 2, 3] [3, 4, 4, 5, 6, 8, 9]
median is now 3
inserted 2
[1, 1, 1, 1, 2, 2, 3] [3, 4, 4, 5, 6, 8, 9]
median is now 3.0
inserted 2
[1, 1, 1, 1, 2, 2, 2] [3, 3, 4, 4, 5, 6, 8, 9]
median is now 3
inserted 1
[1, 1, 1, 1, 1, 2, 2, 2] [3, 3, 4, 4, 5, 6, 8, 9]
median is now 2.5
inserted 6
[1, 1, 1, 1, 1, 2, 2, 2] [3, 3, 4, 4, 5, 6, 6, 8, 9]
median is now 3
inserted 7
[1, 1, 1, 1, 1, 2, 2, 2, 3] [3, 4, 4, 5, 6, 6, 7, 8, 9]
```

```
median is now 3.0
inserted 1
[1, 1, 1, 1, 1, 1, 2, 2, 2] [3, 3, 4, 4, 5, 6, 6, 7, 8, 9]
median is now 3
inserted 1
[1, 1, 1, 1, 1, 1, 1, 2, 2, 2] [3, 3, 4, 4, 5, 6, 6, 7, 8, 9]
median is now 2.5
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 2, 2] [2, 3, 3, 4, 4, 5, 6, 6, 7, 8, 9]
median is now 2
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2] [2, 3, 3, 4, 4, 5, 6, 6, 7, 8, 9]
median is now 2.0
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2] [2, 2, 3, 3, 4, 4, 5, 6, 6, 7, 8, 9]
median is now 2
inserted 2
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2] [2, 2, 3, 3, 4, 4, 5, 6, 6, 7, 8,
9]
median is now 2.0
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2] [2, 2, 2, 3, 3, 4, 4, 5, 6, 6, 7,
8, 9]
median is now 2
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2] [2, 2, 2, 3, 3, 4, 4, 5, 6, 6,
7, 8, 9]
median is now 2.0
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [2, 2, 2, 2, 3, 3, 4, 4, 5, 6,
6, 7, 8, 9]
median is now 2
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [2, 2, 2, 2, 3, 3, 4, 4, 5,
6, 6, 7, 8, 9]
median is now 1.5
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 2, 2, 2, 2, 3, 3, 4, 4,
5, 6, 6, 7, 8, 9]
median is now 1
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 2, 2, 2, 2, 3, 3, 4,
4, 5, 6, 6, 7, 8, 9]
median is now 1.0
inserted 1
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1] [1, 1, 2, 2, 2, 2, 3, 3,
4, 4, 5, 6, 6, 7, 8, 9]
median is now 1
```

Your task:

Implement the algorithm for a class Median. Your class should implement an add-an-element function, a median function, a how-many-elements function, and a function that determines whether a given number is in the list.

```
import heapq


class Median:
    def __init__(self):
        self.small = []
        self.big = [ ]

    def peek_big(self):
        return self.big[0]

    def peek_small(self):
        if self.small:
            return -self.small[0]

    def pop_big(self):
        return heapq.heappop(self.big)

    def pop_small(self):
        return - heapq.heappop(self.small)

    def add_small(self, element):
        heapq.heappush(self.small, -element)

    def add_big(self, element):
        heapq.heappush(self.big, element)

    def get_length(self):
         pass

    def get_median(self):
        pass

    def add(self, element):
        pass


    def is_in(self, element):
        pass
```

Run-time

To insert an element, we determine the length of the two sub-heaps (time $O(1)$), might pop an element from a heap ($O(\log(n))$ time), insert that element into the other heap ($O(\log(n))$ time),

and then insert the original element into the first heap ($O(\log(n))$ time). This is still $O(\log(n))$ time. In the best case, we pop, compare, and then insert, which is also $O(\log(n))$ time.

To calculate the median, we determine the lengths of the two heaps (time $O(1)$) and then pop one or two elements ($O(\log(n))$ time).

If we were to just maintain a sorted array, we need $O(n)$ time, which is worse.