295. Find Median from Data Stream

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value and the median is the mean of the two middle values.

- For example, for `arr = [2,3,4]`, the median is `3`.
- For example, for `arr = [2,3]`, the median is `(2 + 3) / 2 = 2.5`.

Implement the MedianFinder class:

- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within `10<sup>-5</sup>` of the actual answer will be accepted.

Example 1:

```
Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

Explanation
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);    // arr = [1]
medianFinder.addNum(2);    // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);    // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

Constraints:

- `-10<sup>5</sup> <= num <= 10<sup>5</sup>`
- There will be at least one element in the data structure before calling `findMedian`.
- At most `5 * 10<sup>4</sup>` calls will be made to `addNum` and `findMedian`.

Follow up:

- If all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your solution?

- If `99%` of all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your solution?

```python
import heapq


class Median:
    def __init__(self):
        self.left = []
        self.right = []

    def size(self):
        return len(self.left) + len(self.right)

    def peek(self):
        if self.size() == 0:
            return -1
        elif len(self.left) >= len(self.right):
            return -self.left[0]
        else:
            return self.right[0]

    def add(self, val):
        if len(self.right) > 0 and val > self.right[0]:
            heapq.heappush(self.right, val)
        else:
            heapq.heappush(self.left, -val)

        if len(self.left) - len(self.right) == 2:
            temp = -heapq.heappop(self.left)
            heapq.heappush(self.right, temp)
        elif len(self.right) - len(self.left) == 2:
            temp = heapq.heappop(self.right)
            heapq.heappush(self.left, -temp)

    def remove(self):
        if self.size() == 0:
            return -1
        elif len(self.left) >= len(self.right):
            return -heapq.heappop(self.left)
        else:
            return heapq.heappop(self.right)
```

```
median = Median()
median.add(5)
median.add(10)
median.add(12)
median.add(2)
median.add(29)
print(median.peek())
median.add(30)
print(median.peek())
```

Leetcode:

```
import heapq
class MedianFinder:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.left = []
        self.right = []

    def size(self):
        return len(self.left) + len(self.right)

    def addNum(self, val: int) -> None:
        if len(self.right) > 0 and val > self.right[0]:
            heapq.heappush(self.right, val)
        else:
            heapq.heappush(self.left, -val)

        if len(self.left) - len(self.right) == 2:
            temp = -heapq.heappop(self.left)
            heapq.heappush(self.right, temp)
        elif len(self.right) - len(self.left) == 2:
            temp = heapq.heappop(self.right)
            heapq.heappush(self.left, -temp)



    def findMedian(self) -> float:
        if self.size() == 0:
            return -1
```

```python
        elif len(self.left) == len(self.right):
            return (-self.left[0]+self.right[0])/2
        elif len(self.left) > len(self.right):
            return -self.left[0]
        else:
            return self.right[0]
```