0.手写优先队列

```java
/**
 * 大顶堆
 */
public class MyPriorityQueue {
    int[] queue;//定义一个数组用来处处数据
    int capacity = 10;// 优先队列的容量
    int size;//优先队列的大小

    // 构造方法
    public MyPriorityQueue(int capacity) {
        // 因为我打算根节点的编号从1开始，这样比较好计算 根节点=i 左节点=2*i  右节点=2*+1
        // 多开辟一个空间是索引为0的位置不存数据
        // 当然也可以从0开始，就是计算麻烦一点
        this.queue = new int[capacity + 1];
        this.capacity = capacity;
        this.size = 0;
    }


    public boolean offer(int val) {
        // 如果当前存储的数据已满，则返回false
        if (this.size >= this.capacity) return false;
        //向最后一个位置插入元素
        queue[++size] = val;
        // 定义一个变量，用来标记要交换的值的索引位置
        int index = size;
        // 当前值存储的位置不是根节点并且当前值大于根节点时需要调整位置
        while (index > 1 && queue[index] > queue[index / 2]) {
            // 交换过程
            int temp = queue[index];
            queue[index] = queue[index / 2];
            queue[index / 2] = temp;
            // 此时新插入的值的索引位置发生了变化需要更新
            index /= 2;
        }
        return true;
    }

    public int poll() {
        // 如果为空队列的话返回-1
        if (size == 0) return -1;
        // 先记录队首位置的元素，便于后面返回
        int result = queue[1];
        // 将队尾元素放到队首
        queue[1] = queue[size--];
        // 定义一个变量，用来标记要交换的值的索引位置
        int index = 1;
        while (index * 2 <= size) {// 如果当前节点有左子节点
            // 判断是否有右节点，  如果有，则判断左右子节点哪个更大
            // 定义一个变量，用来标记较大子节点的索引。先让它初始化为左节点的索引
            int maxIndex = index * 2;
            // 然后判断，如果有右节点，并且右节点值更大，则修改 maxIndex 为右节点的索引
            if (maxIndex + 1 <= size && queue[maxIndex] < queue[maxIndex + 1]) {
                maxIndex = maxIndex + 1;
            }
            // 如果当前值已经大于左右节点，则结束向下调整
            if (index == maxIndex) break;
            // 否则进行调整
            int temp = queue[index];
            queue[index] = queue[maxIndex];
            queue[maxIndex] = temp;
            // 更新当前值索引位置
            index = maxIndex;
        }
        return result;
    }

    public static void main(String[] args) {
        int[] ints = {3, 2, 1, 5, 6, 4};
        MyPriorityQueue myPriorityQueue = new MyPriorityQueue(15);
        for (int anInt : ints) {
            myPriorityQueue.offer(anInt);
        }
        for (int i = 0; i < ints.length; i++) {
            System.out.print(myPriorityQueue.poll() + ",");
        }
    }
}
```

1.剑指 Offer 40. 最小的k个数

普通方法

```
public int[] getLeastNumbers(int[] arr, int k) {
    PriorityQueue<Integer> priorityQueue = new PriorityQueue<>((o1, o2) -> o2 - o1);
    for (int i : arr) {
        priorityQueue.offer(i);
        if (priorityQueue.size() > k) priorityQueue.poll();
    }
    return priorityQueue.stream().mapToInt((Integer i) -> i.intValue()).toArray();
}
```

船长方法

```
    public int[] getLeastNumbers(int[] arr, int k) {
//        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>(new Comparator<Integer>() {
//            @Override
//            public int compare(Integer o1, Integer o2) {
//                return o2 - o1;
//            }
//        });
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>((o1, o2) -> o2 - o1);
        for (int i : arr) {
            priorityQueue.offer(i);
            if (priorityQueue.size() > k) priorityQueue.poll();
        }
        return priorityQueue.stream().mapToInt(i -> i).toArray();
    }
```

优化方法

```
public int[] getLeastNumbers(int[] arr, int k) {
    if (arr.length == 0 || k == 0) return new int[0];
    PriorityQueue<Integer> priorityQueue = new PriorityQueue<>((o1, o2) -> o2 - o1);
    for (int i = 0; i < k; i++) {
        priorityQueue.offer(arr[i]);
    }
    for (int i = k; i < arr.length; i++) {
        if (arr[i] < priorityQueue.peek()) {
            priorityQueue.poll();
            priorityQueue.offer(arr[i]);
        }
    }
    return priorityQueue.stream().mapToInt(i -> i).toArray();
}
```

2.1046. 最后一块石头的重量

```
    public int lastStoneWeight(int[] stones) {
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>((o1, o2) -> o2 - o1);
        for (int stone : stones) {
            priorityQueue.offer(stone);
        }
        while (priorityQueue.size() > 1) {
            int x = priorityQueue.poll();
            int y = priorityQueue.poll();
            if (x > y) {
                priorityQueue.offer(x - y);
            }
        }
        return priorityQueue.isEmpty() ? 0 : priorityQueue.poll();
    }
```

3.703. 数据流中的第 K 大元素

```
public class KthLargest {
    PriorityQueue<Integer> priorityQueue;
    int k;

    public KthLargest(int k, int[] nums) {
        priorityQueue = new PriorityQueue<Integer>();
        this.k = k;
        for (int num : nums) {
            add(num);
        }
    }

    public int add(int val) {
        priorityQueue.offer(val);
        if (priorityQueue.size() > k) priorityQueue.poll();
        return priorityQueue.peek();
    }
}
```

4.373. 查找和最小的K对数字

```
    public List<List<Integer>> kSmallestPairs(int[] nums1, int[] nums2, int k) {
        PriorityQueue<int[]> priorityQueue = new PriorityQueue<>(new Comparator<int[]>() {
            @Override
            public int compare(int[] o1, int[] o2) {
                return o2[2] - o1[2];
            }
        });
        for (int i = 0; i < nums1.length; i++) {
            for (int j = 0; j < nums2.length; j++) {
                if (priorityQueue.size() < k || (nums1[i] + nums2[j]) < priorityQueue.peek()[2]) {
                    priorityQueue.offer(new int[]{nums1[i], nums2[j], nums1[i] + nums2[j]});
                    if (priorityQueue.size() > k) priorityQueue.poll();
                } else break;
            }
        }
        List<List<Integer>> result = new ArrayList<>();
        while (!priorityQueue.isEmpty()) {
            int[] ints = priorityQueue.poll();
            result.add(new ArrayList<Integer>() {{
                this.add(ints[0]);
                this.add(ints[1]);
            }});
        }
        return result;
    }
```

5.215. 数组中的第K个最大元素

```
    public int findKthLargest(int[] nums, int k) {
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();
        for (int num : nums) {
            priorityQueue.offer(num);
            if (priorityQueue.size() > k) priorityQueue.poll();
        }
        return priorityQueue.peek();
    }
```

6.692. 前K个高频单词

```java
    public List<String> topKFrequent(String[] words, int k) {
        HashMap<String, Integer> map = new HashMap<>();
        for (String word : words) {
            map.put(word, map.getOrDefault(word, 0) + 1);
        }
        PriorityQueue<Map.Entry<String, Integer>> priorityQueue = new PriorityQueue<>(new Comparator<Map.Entry<String, Integer>>() {
            @Override
            public int compare(Map.Entry<String, Integer> o1, Map.Entry<String, Integer> o2) {
                return o1.getValue() == o2.getValue() ? o2.getKey().compareTo(o1.getKey()) : o1.getValue() - o2.getValue();
            }
        });
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            priorityQueue.offer(entry);
            if (priorityQueue.size() > k) priorityQueue.poll();
        }
        List<String> result = new ArrayList<>();
        while (!priorityQueue.isEmpty()) {
            result.add(0, priorityQueue.poll().getKey());
        }
        return result;
    }
```

7.面试题 17.20. 连续中值

```java
public class MedianFinder {
    PriorityQueue<Integer> smallHeap;
    PriorityQueue<Integer> bigHeap;

    /**
     * initialize your data structure here.
     */
    public MedianFinder() {
        smallHeap = new PriorityQueue<>();
        bigHeap = new PriorityQueue<>((o1, o2) -> o2 - o1);
    }

    public void addNum(int num) {
        smallHeap.offer(num);
        bigHeap.offer(smallHeap.poll());
        while (bigHeap.size() > smallHeap.size()) {
            smallHeap.offer(bigHeap.poll());
        }

    }

    public double findMedian() {
        if (smallHeap.size() == bigHeap.size()) {
            return (smallHeap.peek() + bigHeap.peek()) / 2.0d;
        }
        return smallHeap.peek();
    }
}
```

8.295. 数据流的中位数

```
public class MedianFinder {
    PriorityQueue<Integer> smallHeap;
    PriorityQueue<Integer> bigHeap;

    /**
     * initialize your data structure here.
     */
    public MedianFinder() {
        smallHeap = new PriorityQueue<>();
        bigHeap = new PriorityQueue<>((o1, o2) -> o2 - o1);
    }

    public void addNum(int num) {
        smallHeap.offer(num);
        bigHeap.offer(smallHeap.poll());
        while (bigHeap.size() > smallHeap.size()) {
            smallHeap.offer(bigHeap.poll());
        }


    }

    public double findMedian() {
        if (smallHeap.size() == bigHeap.size()) {
            return (smallHeap.peek() + bigHeap.peek()) / 2.0d;
        }
        return smallHeap.peek();
    }
}
```