是你想的那个堆吗?

今日目标:

1：能够说出堆的定义及特点

2：能够说出二叉堆的定义及特性

3：能够说出二叉堆的插入，删除堆顶操作步骤

4：能够说出堆的企业应用

5：完成实战题目

1、堆的概述

1.1、定义及特点

堆（Heap）:是一种可以迅速找到数据集合中最大值或者最小值的数据结构。

堆又可以分为两种:

大顶堆（大根堆）：可以迅速找到最大值的堆叫大顶堆

小顶堆（小根堆）：可以迅速找到最小值的堆叫小顶堆

维基百科：堆

1.2、堆的实现形式及复杂度

堆本身是一个抽象的数据结构，它的实现可以有很多种，比如常见的：二叉堆，斐波拉契堆等等！其中面试最为常见的是二叉堆（相对容易实现）但却不是最好的。

对于堆而言，常见的几个操作如下（以大顶堆为例）

操作	复杂度
find-max (or *find-min*)	O(1)
delete-max(or delete-min)	O(log n)
insert(create)	O(log n)　or　O(1)

常见的实现及对应的复杂度如下图:

Operation	find-max	delete-max	insert	increase-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

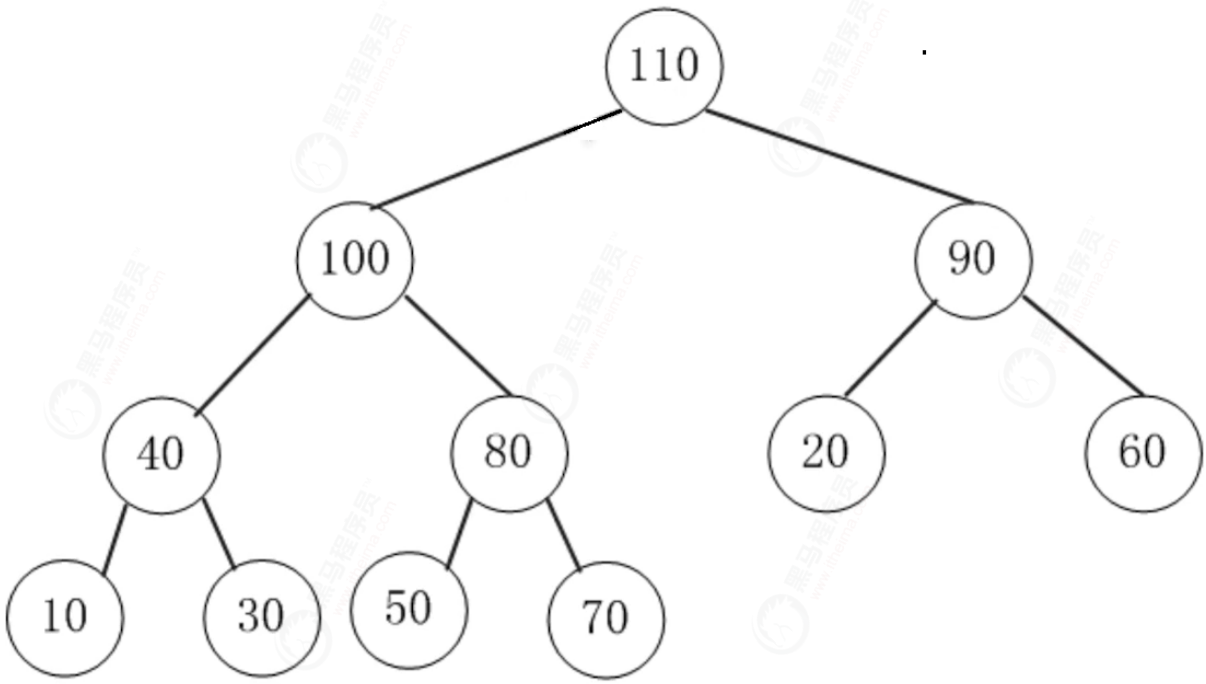
2、二叉堆

2.1、二叉堆的定义及特性

二叉堆是通过完全二叉树来实现的（注意不是二叉搜索），它具备以下特性：

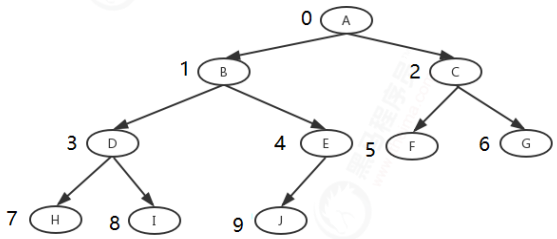
1. 堆是一个完全二叉树；
2. 堆中每一个节点的值都必须大于等于（或小于等于）其子树中每个节点的值

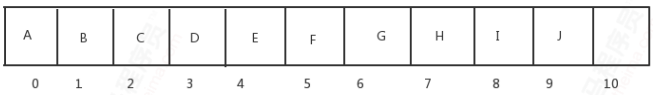
   如下图就是一个大顶堆



> 要点：完全二叉树如何存储？

对于完全二叉树除了最后一层，其他层的节点个数都是满的，最后一层的节点都靠左排列，当然想到完全二叉树我们立马能能够想到我们之前所讲到的完全二叉树可以用数组来进行存储，





存储特征：

对于任意一个节点假设它存储在数组下标为k的位置则

1：下标 2 * k +1 的位置存储的是它的左子节点

2：下标 2 * k + 2 的位置存储它的右子节点

对于任意一个节点假设它存储在数组下标为k的位置则：

1：下标 (k-1) /2 的位置存储的是它的父节点

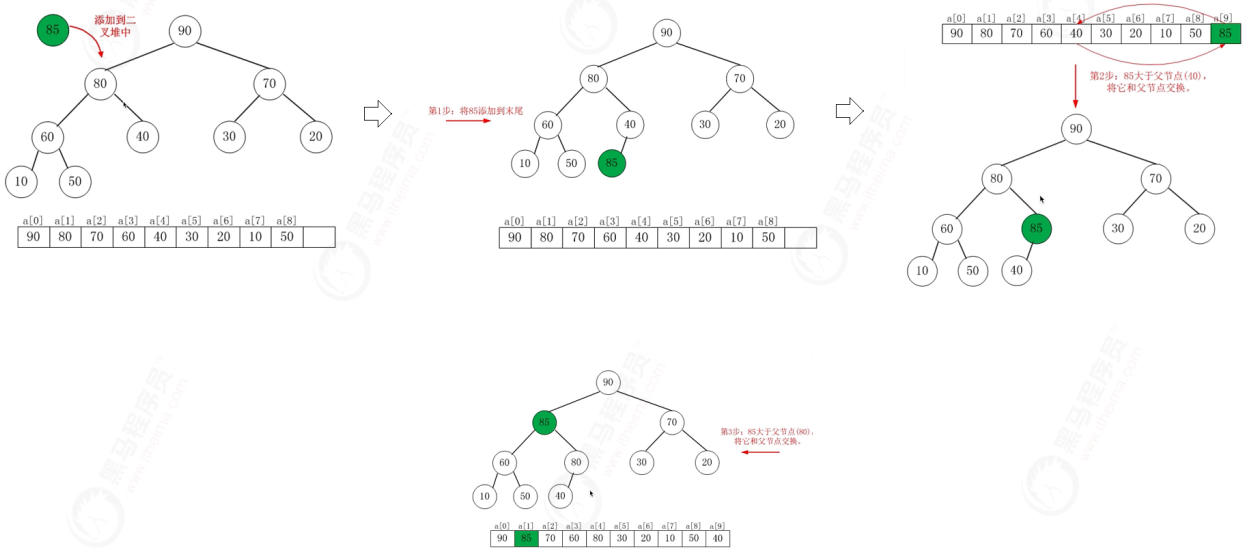
2.2、二叉堆的实现要点

2.2.1、插入操作

1：新元素一律插入到堆的尾部，

2：从尾部开始依次向上调整堆的结构直到满足堆的定义（一直到根节点结束），我们叫做：heapifyUp

操作过程如下图：



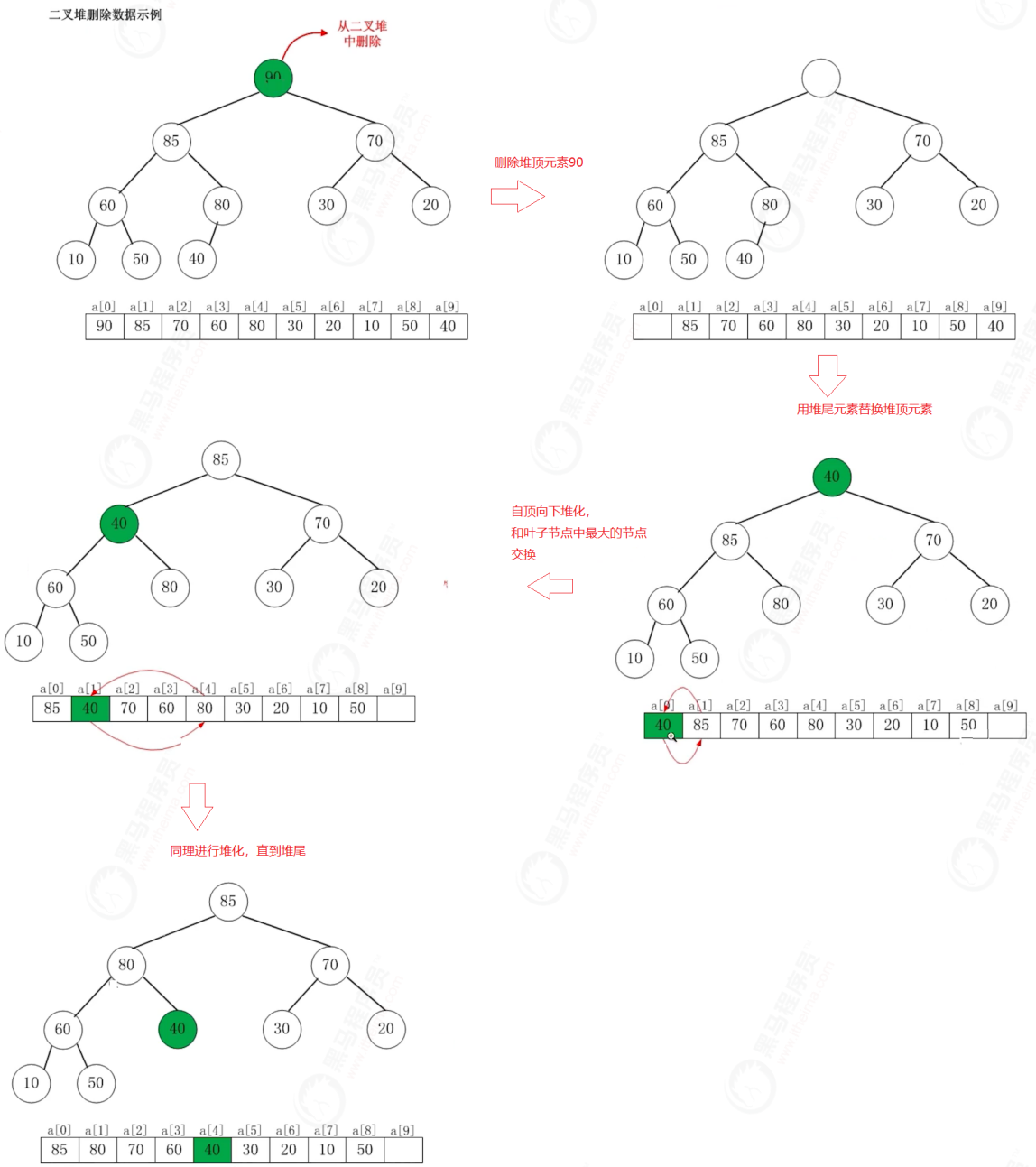
插入操作的时间复杂度是O(log n)

2.2.2、删除堆顶操作

1：将堆尾元素替换到顶部，（即堆顶元素被替代删除掉）

2：从顶部开始依次向下调整堆的结构直到满足堆的定义（一直到堆尾即可），我们叫做heapifyDown

操作过程如下图：



删除堆顶元素操作的时间复杂度是O(log n)

面试题目：2020-华为秋招

{0, 2, 1, 4, 3, 9, 5, 8, 6, 7} 是以数组形式存储的小顶堆，删除堆顶元素0后的结果是（）

- A:{2, 1, 4, 3, 8, 9, 5, 8, 6, 7}
- B:{1, 2, 5, 4, 3, 9, 8, 6, 7}
- C:{2, 3, 1, 4, 7, 9, 5, 8, 6}
- D:{1, 2, 5, 4, 3, 9, 7, 8, 6}

> 解析：根据删除堆顶元素的方式可得答案为D

2.2.3、获取最值操作

堆的特性就是可以迅速获取一堆元素中的最大值或者最小值，因此对于堆这种数据结构，我们返回其根节点即可，就是最大值或者最小值。时间复杂度是O(1)

总结

二叉堆是堆的一种常见且简单的实现，但并非最优的实现。在工程应用中有用基于二叉堆实现的，也有基于其他更高效的数据结构来实现的，不同的语言实现的不太一样！

> 课后作业：请自行实现一个二叉堆，满足以下操作
>
> 1：插入元素
>
> 2：删除堆顶元素
>
> 3：获取最值元素

3、堆的应用

3.1、优先级队列

队列我们都知道，满足先进先出（FIFO）的特点，但是对于优先级队列出队列的顺序不是按照进入的顺序来的，而是按照优先级来的，优先级高的先出队列，优先级低的后出队列。

如何来实现一个优先级队列呢?

实现的方法有很多但是我们最常用的就是用堆来实现，用堆实现一个优先级队列是最直接和高效的。并且在不同语言中就直接提供了具体的实现，比如java语言中的PriorityQueue，Python中也有第三方的库实现好了。

Java PriorityQueue：基于二叉小顶堆实现，堆顶元素是基于自然排序或者Comparator排序的最小元素。但是是非线程安全的。因此java中也提供了线程安全的优先级队列：PriorityBlockingQueue

对于优先级队列的使用场景有很多，这里例举一个：

> 问题：比如现在有1T的数据需要排序，请问你应该如何做?
>
> 首先我们可以用100个文件来存储这1T数据，那在每条数据产生的时候对数据进行哈希求值然后对100取模决定将该条数据存储到哪个文件中，这样可以将1T数据分配到100个文件中，接下来我们分别对每个文件中的数据进行排序，这个排序可以采用各种排序算法来解决，对每个小文件排好序之后剩余的就是将这100个文件合并成一个有序的大文件，在这个合并的过程中我们就可以使用到优先级队列。
>
> 具体做法是:
>
> 1：从100个文件中各取第一条数据，依次存入实现设定好的小顶堆中，这样堆顶元素就是优先级队列队首的元素，也就是最小的数据，我们将堆顶元素取出存储到最终合并好的大文件中，然后删除堆顶元素。

> 2：如果刚刚删除的堆顶元素来自15.txt，则从该文件中取出下一个数据存入到优先级队列中，然后再取堆顶元素存储到合并后的大文件中，删除堆顶元素，然后循环这个过程，就可以将100个文件中的数据依次的存储到最终的大文件中。

3.2、Top K问题

刚刚讲到利用堆实现一个优先级队列，此外堆还有另外一个非常重要的应用场景，那就是经典的求Top K问题。

对于Top K问题的求解我们大致可以分为两类情况：

情况1：针对一组事先已经确定好的静态数据

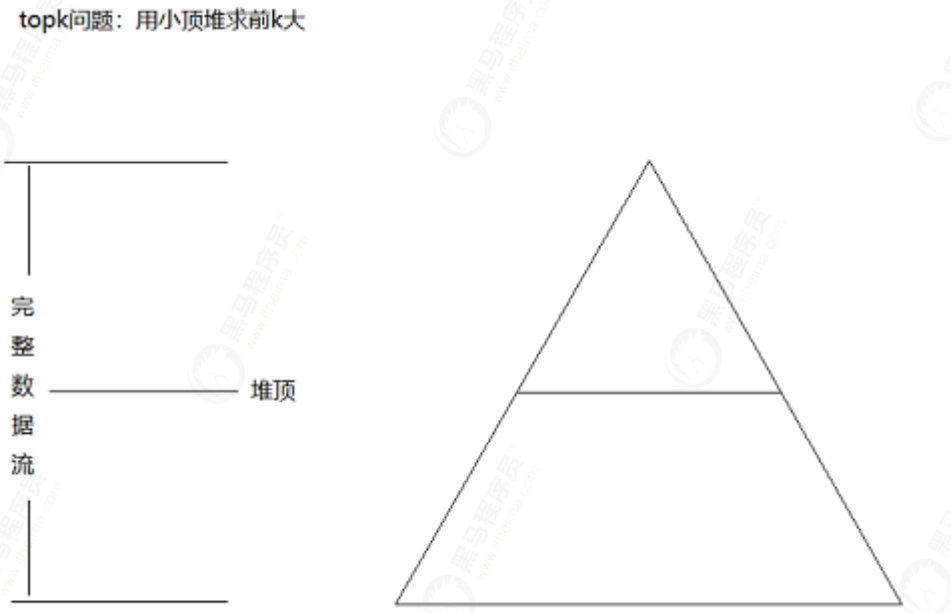
比如：从包含n个数据的数组中，查找前K大数据，我们可以维护一个**大小为K的小顶堆**，顺序遍历数组向堆中添加元素，会出现以下两种情况:

1：如果堆中元素不足k个，则一直向堆中添加数据

2：如果堆中元素已够k个，则判断新加入元素和堆顶元素的大小，如果新加入元素比堆顶元素小则不做处理继续遍历数组，如果数组元素比堆顶元素大，则将堆顶元素删除，将数组元素加入到堆中

这样等数组遍历完后，堆中的数据就是前K大数据了。

> 类似于：



数据 [4, 5, 2, 1, 8, 6, 3, 7] 求前3大元素

维护一个大小为3的小顶堆

1：如果堆中元素不足k个，则一直向堆中添加数据

2：如果堆中元素已够k个，则判断新加入元素和堆顶元素的大小，
如果新加入元素比堆顶元素小则不做处理继续遍历数组，
如果数组元素比堆顶元素大，则将堆顶元素删除，将数组元素加入到堆中

注意：如果是要找前K小的数据应该怎么做?

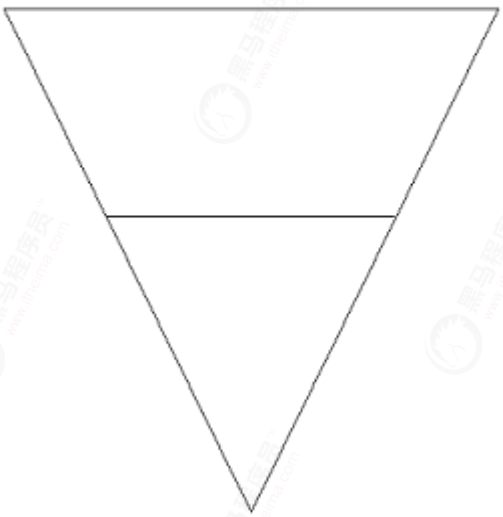
维护一个**大小为k的大顶堆**，同理，顺序遍历数据集依次向堆中添加元素，也会出现以下两种情况：

1：堆中元素不足k个，则一直向堆中添加数据

2：堆中元素已够k个，则判断新加入元素是否比堆顶元素小，如果比堆顶元素小则删除堆顶元素并加入新元素，如果并堆顶元素大则不做操作继续下一个数据。

类似于：

用大顶堆，求前k小



遍历数组需要 O(n) 的时间复杂度，一次堆化操作需要 O(log K) 的时间复杂度，所以最坏情况下，n个元素都入堆一次，时间复杂度就是 **O(n * logK)**

情况2：针对一组动态数据，可能随时有数据的加入和移除

这种情况其实就是实时 Top K。

操作思路是跟情况1一致，只不过所有操作都是实时的，有数据流时就操作。

> **注意**：TOP K问题的最高效解法是利用快排思想来解决！这个后续课程会讨论。
>
> 另外其实TOP K问题，求前k大，也可以用大顶堆来实现，但是我们需要将所有数据先添加入堆中，然后再依次从堆中删除堆顶元素，直到删除k个为止，这样也能实现，只不过时间复杂度是O(n * log n)，相对于前面讲的用小顶堆实现复杂度要略高。并且要实现实时 Top K不太好实现！
>
> 求前k小问题也同理可以用小顶堆来实现！

3.3、99%响应时间

比如在我们的工作中可能会遇到这样一个业务需求：如何实时统计业务接口的99%响应时间?

我们先来解释一个概念：99百分位数

99百分位数的概念可以类比中位数,中位数的概念就是将数据从小到大排列，处于中间位置，就叫中位数，这个数据会大于等于前面50%的数据。如果将一组数据从小到大排列，这个 99 百分位数就是大于前面 99% 数据的那个数据。 如果你还是不太理解，我再举个例子。假设有 100 个数据，分别是 1，2，3，……，100，那 99 百分位数就是 99，因为小于等于 99 的数占总个数的 99%。

弄懂了这个概念之后我们再来看 99% 响应时间。如果有 100 个接口访问请求，每个接口请求的响应时间都不同，比如 55 毫秒、100 毫秒、23 毫秒等，我们把这 100 个接口的响应时间按照从小到大排列，排在第 99 的那个数据就是 99% 响应时间，也叫 99 百分位响应时间。

我们总结一下，如果有 n 个数据，将数据从小到大排列之后，99 百分位数大约就是第 n*99% 个数 据，同类，80 百分位数大约就是第 n*80% 个数据。

那如何求解99%响应时间?

我们维护两个堆，一个大顶堆，一个小顶堆。假设当前总数据的个数是 n，大顶堆中保存 n * 99% 个数据，小顶堆中保存 n*1% 个数据。大顶堆堆顶的数据就是我们要找的 99% 响应时间。每次插入一个数据的时候，我们要判断这个数据跟大顶堆和小顶堆堆顶数据的大小关系，然后决定插入到哪个堆中。如果这个新插入的数据比大顶堆的堆顶数据小，那就插入大顶堆；如果这个新插入的数据比小顶堆的堆顶数据大，那就插入小顶堆。但是，为了保持大顶堆中的数据占 99%，小顶堆中的数据占 1%，在每次新插入数据之后，我们都要重新计算，这个时候大顶堆

和小顶堆中的数据个数，是否还符合 99:1 这个比例。如果不符合，我们就将一个堆中的数据移动到另一个堆。移动的方式就是取其中一个堆的堆顶数据存入另一个堆。

通过这样的方法，每次插入数据，可能会涉及几个数据的堆化操作，所以时间复杂度是 O(logn)。每次求 99% 响应时间的时候，直接返回大顶堆中的堆顶数据即可，时间复杂度是 O(1)。

> 最后：堆还有一个应用就是堆排序，这个在排序章节会进行讲解！

4、面试实战

剑指 Offer 40. 最小的k个数

字节，网易半年内面试题，剑指 Offer 40. 最小的k个数

典型的TOP K问题

> Top K 问题有两种不同的解法，一种解法使用堆（优先队列），另一种解法使用类似快速排序的分治法。这两种方法各有优劣，最好都掌握

解法1：排序，然后找前k个，时间复杂度是O(n * log n)

解法2：使用优先级队列（二叉小顶堆），小顶堆求前k小，需要先将所有数据添加到堆，然后依次取出堆顶元素，取k个

```
class Solution {
    //利用优先级队列(基于二叉小顶堆)
    public int[] getLeastNumbers(int[] arr, int k) {

        PriorityQueue<Integer> queue = new PriorityQueue();
        //将所有元素加入堆
        for (int i=0;i< arr.length;i++) {
            queue.offer(arr[i]);
        }
        //依次从堆取堆顶元素，取k个即可
        int[] res = new int[k];
        for (int i=0;i<k;i++) {
            res[i] = queue.poll();
        }
        return res;
    }
}
```

> 时间复杂度：O(n * log n)
>
> 空间复杂度：O(n)

解法3：求前k小，用大顶堆实现可以达到O(n * log k)的复杂度

```java
class Solution {
    public int[] getLeastNumbers(int[] arr, int k) {
        if (arr == null || arr.length ==0 || k ==0) {
            return new int[]{};
        }
        //默认是小顶堆,通过改变比较器可以成为大顶堆
        //Queue<Integer> queue = new PriorityQueue<>(k,(v1,v2)-> v2-v1);
        Queue<Integer> queue = new PriorityQueue<>(k,(v1,v2)-> Integer.compare(v2,v1));
        //将所有元素加入堆
        for (int i=0;i< arr.length;i++) {
            if (queue.size() < k) {
                queue.offer(arr[i]);
            }else {
                if (arr[i] <  queue.peek()) {
                    queue.poll();
                    queue.offer(arr[i]);
                }
            }
        }
        //依次从堆取堆顶元素，取k个即可,能保证返回数据的顺序性,O(k * log k)
        //这里因为看题目上下文并不需要返回数据的顺序性,所以可以直接遍历PriorityQueue,底层存储基于数组,O(n)
        int[] res = new int[k];
        int i=0;
        for (int e:queue) {
            res[i++] = e;
        }
        return res;
    }
}
```

> Comparator是一个函数式接口（有@FunctionalInterface注解），说明可以使用Lambda表达式完成比较操作，并且其中T指的是你要比较的类型
>
> 1. 在这个接口中必须要实现的方法是int compare(T o1, T o2);
> 2. 排序规则：传递的参数是第一个是o1，第二个是o2，比较的时候也是用o1- o2进行比较，那么就是升序；如果比较的时候是用反过来o2-o1进行比较，那么就是降序

同类题目：215. 数组中的第K个最大元素与面试题 17.14. 最小K个数

239. 滑动窗口最大值

使用堆作为滑动窗口，这样获取最大值只需要O(1)的时间复杂度，

```java
class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums.length ==0 || k==0) {
```

```
            return new int[]{};
        }
        int n = nums.length;
        int[] res = new int[n - k +1];
        int max = 0;

        //构造一个大小为k的大顶堆，作为我们的窗口
        Queue<Integer> queue = new PriorityQueue<>((v1,v2)-> v2-v1);
        for (int i=0;i<n;i++) {
            int start = i - k;
            if (start >= 0) {
                queue.remove(nums[start]);
            }
            queue.offer(nums[i]);
            if (queue.size() == k) {
                res[max++] = queue.peek();
            }
        }
        return res;
    }
}
```

> 注意：最终提交时未AC，超时，原因是：虽然offer是O(log k)的，但是remove操作是O(k)的，所以遍历完整个数组，时间复杂度是O(n * k)的
>
> 最终解法之前用单调队列完成过！

347. 前 K 个高频元素

字节，腾讯最近面试题，347. 前 K 个高频元素

hash+priorityQueue

> 1：借助 哈希表 来建立数字和其出现次数的映射，遍历一遍数组统计元素的频率
>
> 2：维护一个元素数目为 k 的最小堆
>
> 3：每次都将新的元素与堆顶元素（堆中频率最小的元素）进行比较
>
> 4：如果新的元素的频率比堆顶端的元素大，则弹出堆顶端的元素，将新的元素添加进堆中
>
> 最终，堆中的 kk 个元素即为前 kk 个高频元素

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        //使用hash统计每个元素出现的频率
        Map<Integer,Integer> hash = new HashMap();
        for (int num:nums) {

            //int value =  hash.containsKey(num) ? hash.get(num)+1 : 1;
```

```
            hash.put(num,hash.getOrDefault(num,1)+1);
        }
        //构造一个小顶堆(直接使用java中的实现PriorityQueue)
        //改造:元素的优先级的比较方式，按照元素的频率来定优先级,并非按照元素自身的大小关系来定优先级
        Queue<Integer> queue = new PriorityQueue(k,(v1,v2) ->  hash.get(v1)-hash.get(v2) );
        for (Integer key:hash.keySet()) {
            if (queue.size() < k) {
                queue.offer(key);
            }else {
                if ( hash.get(key) > hash.get(queue.peek())) { //注意判断是按照元素的频率去判断
                    queue.poll();
                    queue.offer(key);
                }
            }
        }

        int[] res = new int[k];
        int idx = k-1;
        while (idx >=0) {
            res[idx--] = queue.poll(); //堆顶是频率最小的
        }
        return res;
    }
}
```

其他题目：

264. 丑数 II和剑指 Offer 49. 丑数