<https://mp.weixin.qq.com/s/eOAth_dUigoPCrM1hmUnEA>

# 拜托，别再问我什么是堆了!

## **前言**

堆是生产中非常重要也很实用的一种数据结构，也是面试中比如求 Top K 等问题的非常热门的考点，本文旨在全面介绍堆的基本操作与其在生产中的主要应用，相信大家看了肯定收获满满！本文将会从以下几个方面来讲述堆:

* 生产中的常见问题
* 堆的定义
* 堆的基本操作
* 堆排序
* 堆在生产中应用

## **生产中的常见问题**

我们在生产中经常碰到以下常见的问题：

1. 优先级队列的应用场景很广，它是如何实现的呢
2. 如何求 Top K 问题
3. TP99 是生产中的一个非常重要的指标，如何快速计算

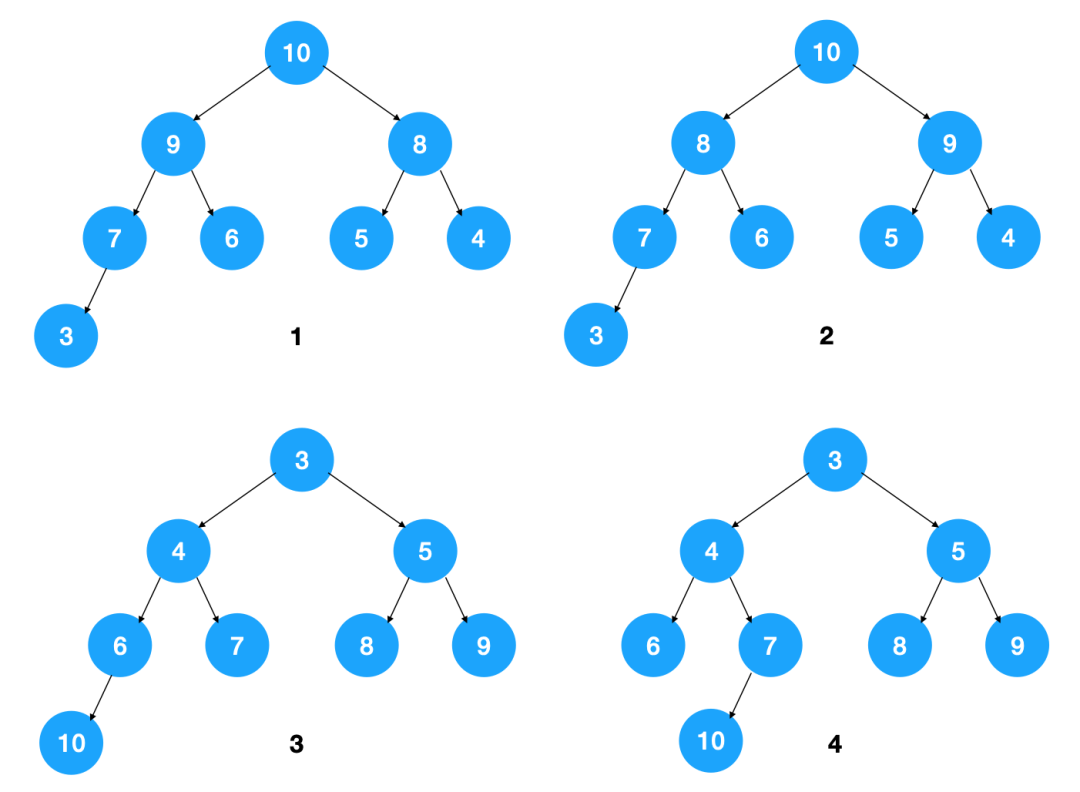
可能你已经猜到了，以上生产上的高频问题都可以用堆来实现，所以理解堆及掌握其基本操作十分重要！接下来我们就来一步步地来了解堆及其相关操作，掌握了堆，上面三个生产上的高频问题将不是问题。

## **堆的定义**

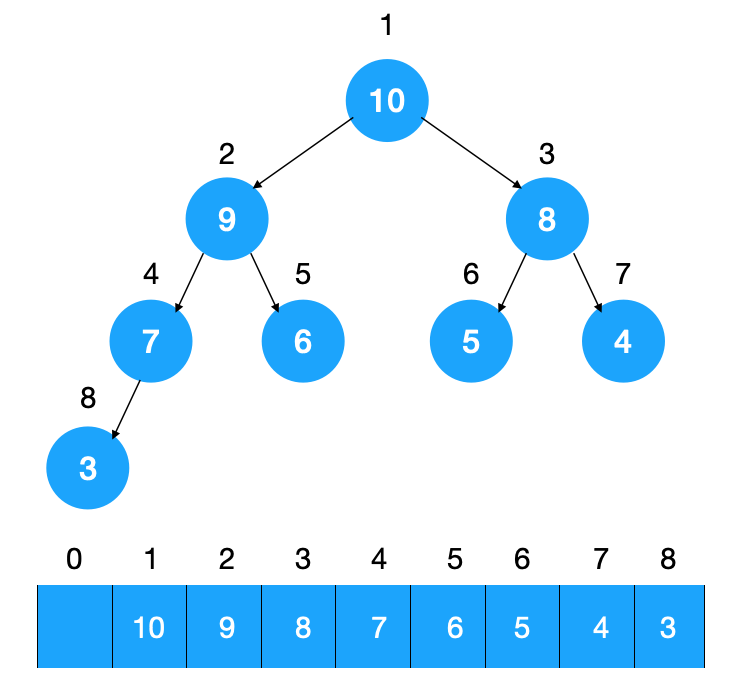
堆有以下两个特点

1. **堆是一颗完全二叉树**，这样实现的堆也被称为**二叉堆**
2. 堆中节点的值都大于等于（或小于等于）其子节点的值，堆中如果节点的值都大于等于其子节点的值，我们把它称为**大顶堆**，如果都小于等于其子节点的值，我们将其称为**小顶堆**。

简单回顾一下什么是完全二叉树，它的叶子节点都在最后一层，并且这些叶子节点都是靠左排序的。从堆的特点可知，下图中，1，2 是大顶堆，3 是小顶堆， 4 不是堆（不是完全二叉树）



从图中也可以看到，一组数据如果表示成大顶堆或小顶堆，可以有不同的表示方式，因为它只要求节点值大于等于（或小于等于）子节点值，未规定左右子节点的排列方式。堆的底层是如何表示的呢，从以上堆的介绍中我们知道堆是一颗完全二叉树，而完全二叉树可以用数组表示。



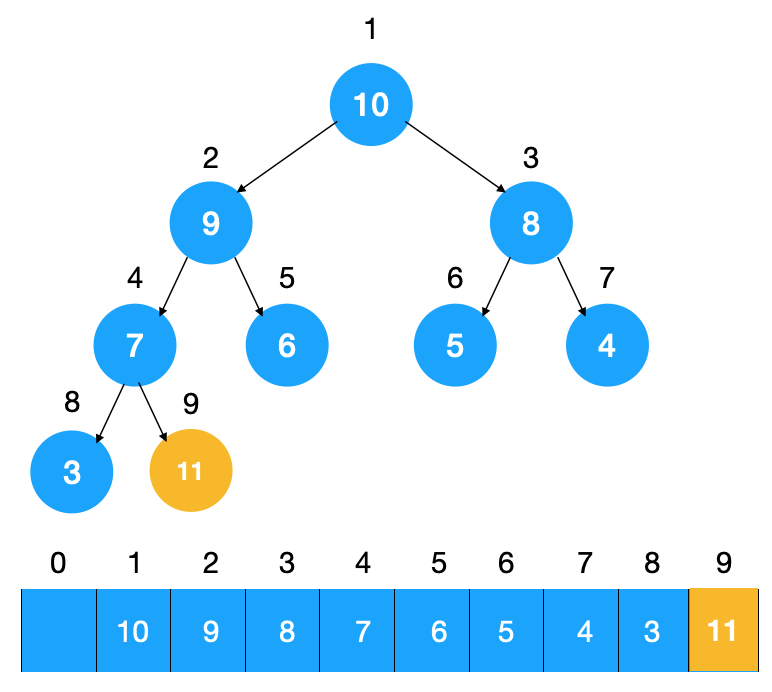
*如图示：给完全二叉树按从上到下从左到右编号，则对于任意一个节点来说，很容易得知如果它在数组中的位置为 i，则它的左右子节点在数组中的位置为 2i，2i + 1，通过这种方式可以定位到树中的每一个节点，从而串起整颗树。*一般对于二叉树来说每个节点是要存储左右子节点的指针，而由于完全二叉树的特点（叶子节点都在最后一层，并且这些叶子节点都是靠左排序的），用数组来表示它再合适不过，用数组来存储有啥好处呢，由于不需要存指向左右节点的指针，在这颗树很大的情况下能省下很多空间！

## **堆的基本操作**

堆有两个基本的操作，构建堆（往堆中插入元素）与删除堆顶元素，我们分别来看看这两个操作

* 往堆中插入元素

往堆中插入元素后（如下图示），我们需要继续满足堆的特性，所以需要不断调整元素的位置直到满足堆的特点为止（堆中节点的值都大于等于（或小于等于）其子节点的值）,我们把这种调整元素以让其满足堆特点的过程称为**堆化（heapify）**



由于上图中的堆是个大顶堆，所以我们需要调整节点以让其符合大顶堆的特点。怎么调整？不断比较子节点与父节点，如果子节点大于父节点，则交换，不断重复此过程，直到子节点小于其父节点。来看下上图插入节点 11 后的堆化过程。

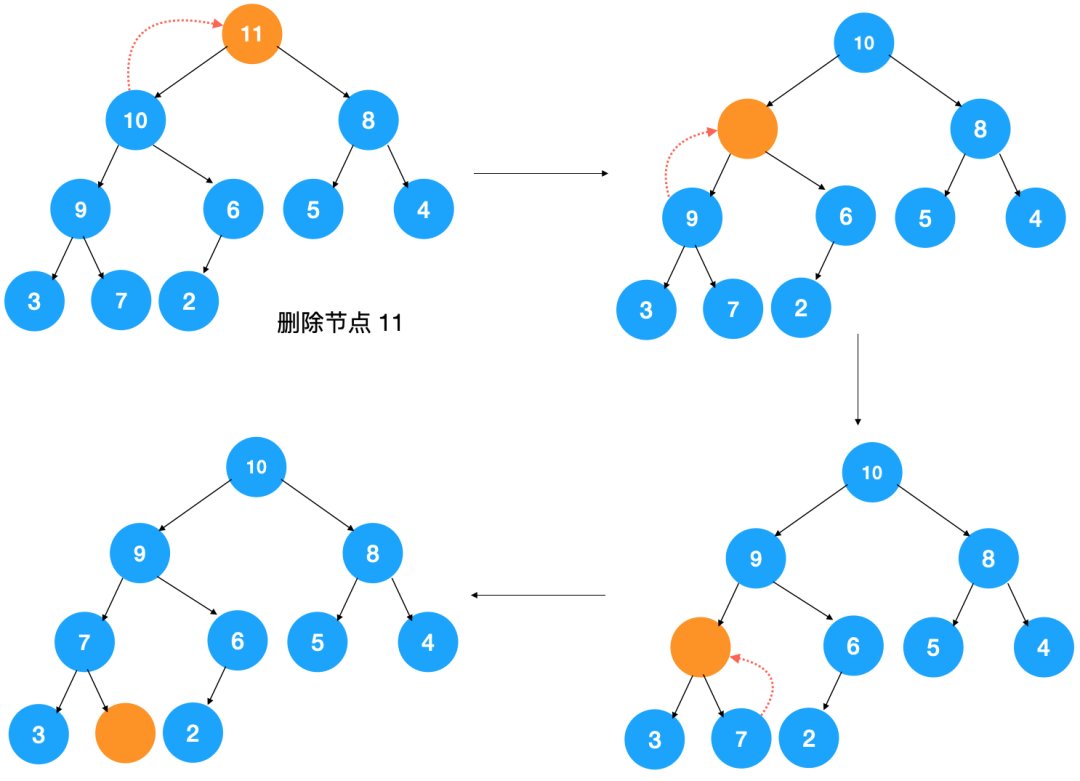
这种调整方式是先把元素插到堆的最后，然后自下而上不断比较子节点与父节点的值，我们称之为由下而上的堆化。有了以上示意图，不难写出插入元素进行堆化的代码：



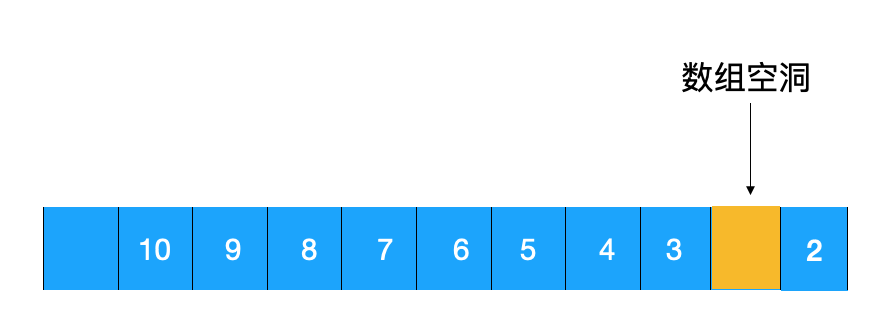
时间复杂度就是树的高度，所以为 O(logn)。

* 删除堆顶元素

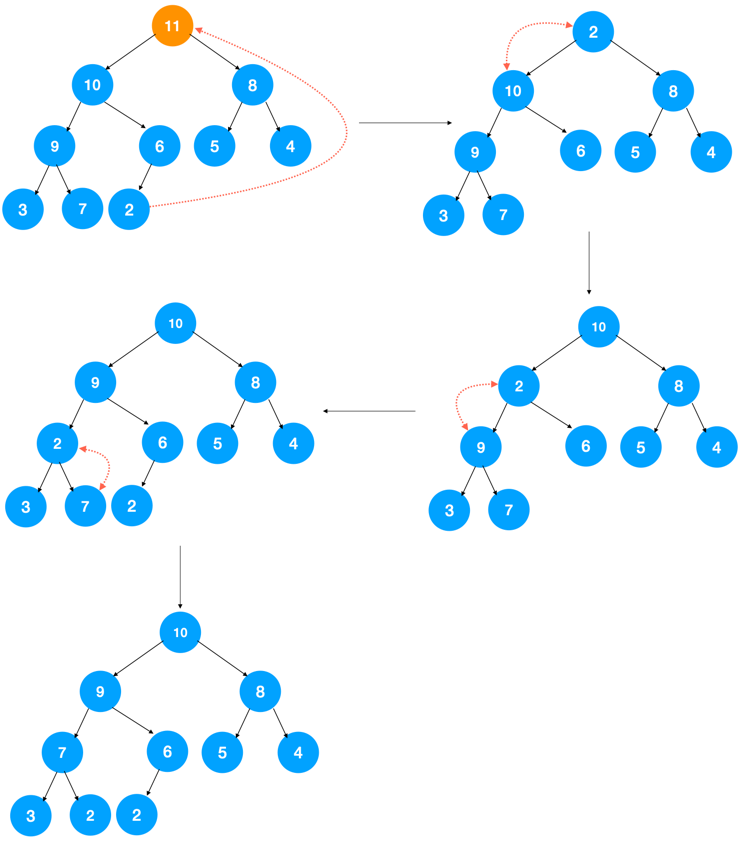
由于堆的特点（节点的值都大于等于（或小于等于）其子节点的值），所以其根节点（堆项）要么是所有节点中最大，要么是所有节点中最小的，当删除堆顶元素后，也需要调整子节点，以让其满足堆（大顶堆或小顶堆）的条件。假设我们要操作的堆是大顶堆，则删除堆顶元素后，要找到原堆中第二大的元素以填补堆顶元素，而第二大的元素无疑是在根节点的左右子节点上，假设是左节点，则用左节点填补堆顶元素之后，左节点空了，此时需要从左节点的左右节点中找到两者的较大值填补左节点...，不断迭代此过程，直到调整完毕，调整过程如下图示：



但是这么调整后，问题来了，如上图所示，在最终调整后的堆中，出现了数组空洞，对应的数组如下：



怎么解决？我们可以用最后一个元素覆盖堆顶元素，然后再自上而下地调整堆，让其满足大顶堆的要求，这样即可解决数组空洞的问题。



看了以上示意图，代码实现应该比较简单，如下：

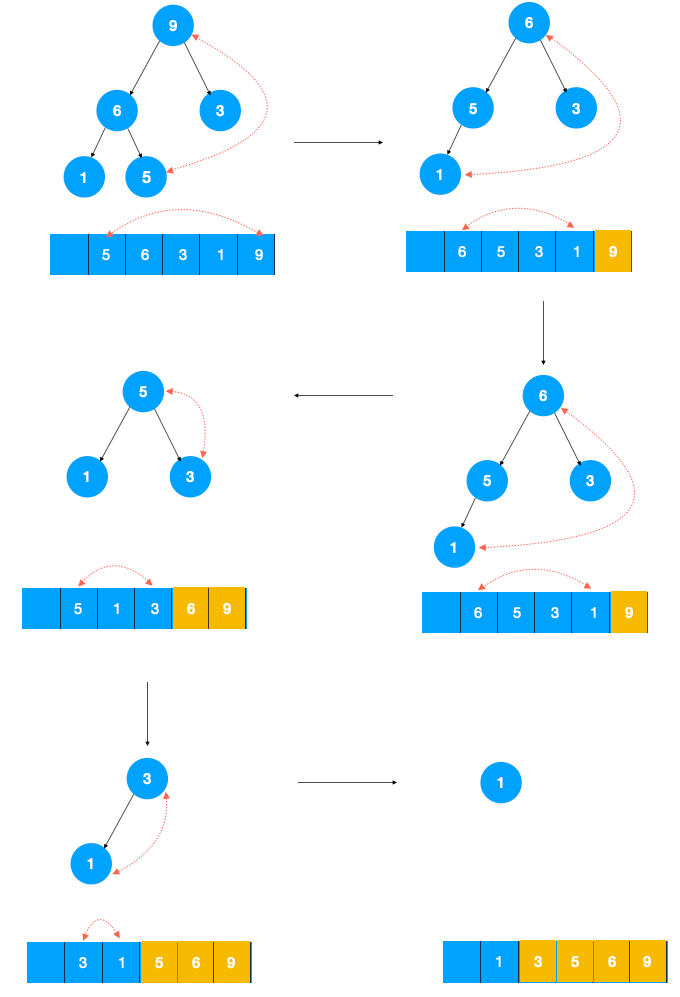




时间复杂度和插入堆中元素一样，也是树的高度，所以为 O(logn)。

## **堆排序**

用堆怎么实现排序？我们知道在大顶堆中，根节点是所有节点中最大的，于是我们有如下思路：假设待排序元素个数为 n（假设其存在数组中），对这组数据构建一个大顶堆，删除大顶堆的元素（将其与数组的最后一个元素进行交换），再对剩余的 n-1 个元素构建大顶堆，再将堆顶元素删除（将其与数组的倒数第二个元素交换），再对剩余的 n-2 个元素构建大顶堆...，不断重复此过程，这样最终得到的排序一定是从小到大排列的，堆排序过程如下图所示：

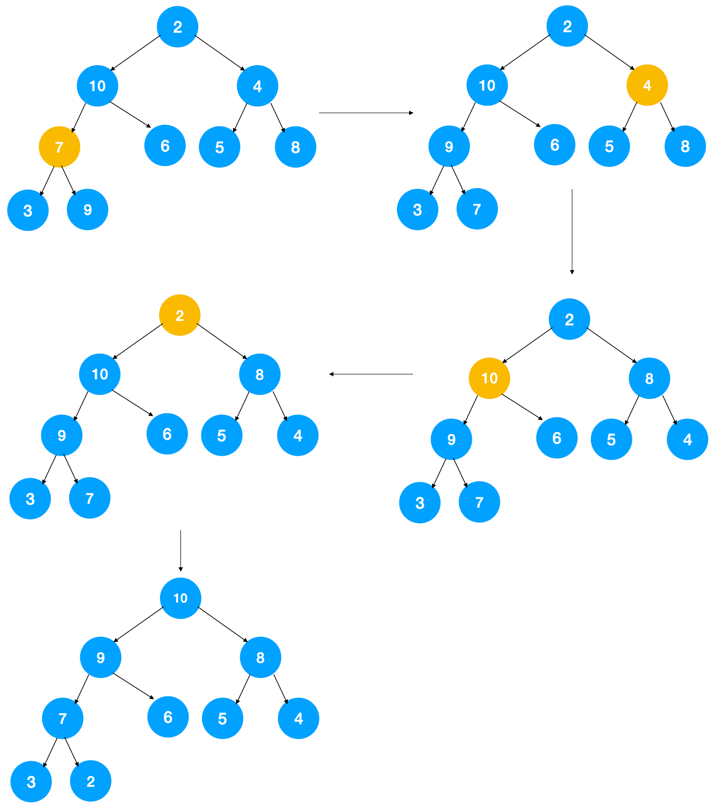


从以上的步骤中可以看到，重要的步骤就两步，建堆（堆化，构建大顶堆）与排序。先看下怎么建堆，其实在上一节中我们已经埋下了伏笔，上一节我们简单介绍了堆的基本操作，插入和删除，所以我们可以**新建一个数组**，遍历待排序的元素，每遍历一个元素，就调用上一节我们定义的 **insert(int value)** 方法，这个方法在插入元素到堆的同时也会堆化调整堆为大顶堆，遍历完元素后，最终生成的堆一定是大顶堆。

用这种方式生成的大顶堆空间复杂度是多少呢，由于我们新建了一个数组，所以空间复杂度是 O(n)，但其实堆排序是原地排序的（不需要任何额外空间），所以我们重点看下如何在不需要额外空间的情况下生成大顶堆。

其实思路很简单，对于所有**非叶子节点**，自上而下不断调整使其满足大顶堆的条件（每个节点值都大于等于其左右节点的值）即可，遍历到最后得到的堆一定是大顶堆！同时调整堆的过程中只是不断交换数组里的元素，没有用到额外的存储空间。那么非叶子节点的范围是多少呢，假设数组元素为 n，则数组下标为 1 到 n / 2 的元素是非叶子节点。下标 n / 2 + 1 到 n 的元素是叶子节点。

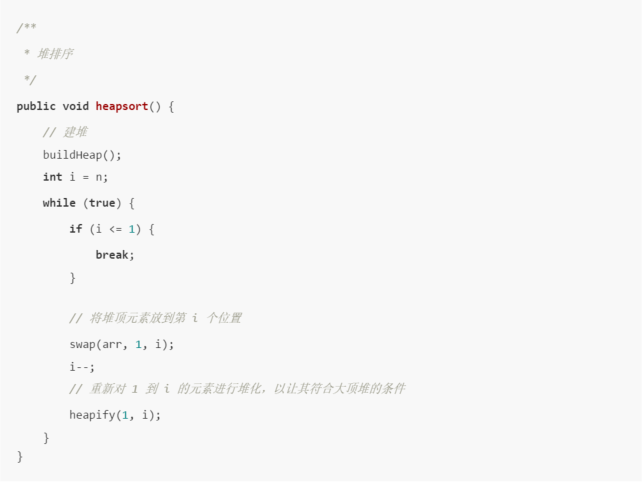
示意图如下：



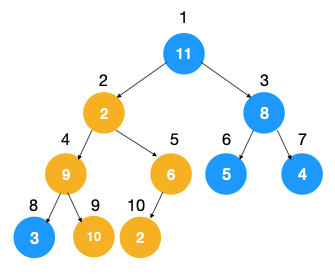
*如图示：对每个非叶子节点自上而下调整后，最终得到大顶堆。*有了以上思路，不难写出如下代码:



这样建堆的时间复杂度是多少呢，我们知道对每个元素进行堆化时间复杂度是 O(log n),那么对 1 到 n/2 个元素进行堆化，则总的时间复杂度显然是 O(n log n)（实际上如果详细推导，时间复杂度是 O(n)，这里不作展开，有兴趣的同学建议查一下资料看下 O(n) 是怎么来的）。知道怎么建堆，接下来排序就简单了，对 n 个元素来说，只要移除堆顶元素（将其与最后一个元素交换），再对之前的 n-1 个元素堆化，再移除堆顶元素（将其与倒数第二个元素交换）...，不断重复此过程即可，代码如下:



时间复杂度上文已经分析过了，就是 O(n log n)，居然和快排一样快！但堆排序实际在生产中用得并不是很多，Java 默认的数组排序（**Arrays.sort()**）底层也是用的快排，时间复杂度和快排一样快，为啥堆排序却并不受待见呢。主要有以下两个原因1、 快排在递归排序的过程中，都是拿 pivot 与**相邻的元素**比较，会用到计算机中一个非常重要的定理：**局部性原理**，啥叫局部性原理，可以简单理解为当 CPU 读取到某个数据的时候，它认为这个数据附近相邻的数据也有很大的概率会被用到，所以干脆把被读取到数据的附近的数据也一起加载到 Cache 中，这样下次还需要再读取数据进行操作时，就直接从 Cache 里拿数据即可（无需再从内存里拿了），数据量大的话，极大地提升了性能。**堆排序无法利用局部性原理**，为啥呢，我们知道在堆化的过程中，需要不断比较节点与其左右子节点的大小，左右子节点也需要比较其左右节点。



*如图示：在对节点 2 自上而下的堆化中，其要遍历数组中 4，5，9，10... 中的元素，这些元素并不是相邻元素，无法利用到局部性原理来提升性能。*

*如图示：在对节点 2 自上而下的堆化中，其要遍历数组中 4，5，9，10... 中的元素，这些元素并不是相邻元素，无法利用到局部性原理来提升性能*2、我们知道堆排序的一个重要步骤是把堆顶元素移除，重新进行堆化，每次堆化都会导致大量的元素比较，这也是堆排序性能较差的一个原因。3、堆排序不是稳定排序，因为我们知道在堆化开始前要先把首位和末位元素进行交换，如果这两元素值一样，就可能改变他们原来在数组中的相对顺序，而快排虽然也是不稳定排序，不过可以改进成稳定排序，这一点也是快排优于堆排序的一个重要的点。

## **堆在生产中应用**

堆排序虽然不常用，但堆在生产中的应用还是很多的，这里我们详细来看堆在生产中的几个重要应用

1. **优先级队列** 我们知道队列都是先进先出的，而在优先级队列中，元素被赋予了权重的概念，权重高的元素优先执行，执行完之后下次再执行权重第二高的元素...，显然用堆来实现优先级队列再合适不过了，只要用一个大顶堆来实现优先级队列即可，当权重最高的队列执行完毕，将其移除（相当于删除堆顶），再选出优先级第二高的元素（堆化让其符合大顶堆 的条件），很方便，实际上我们查看源码就知道， Java 中优先级队列  PriorityQueue 就是用堆来实现的。

**2、 求 TopK 问题**怎样求出 n 个元素中前 K 个最大/最小的元素呢。假设我们要求前 K 个最大的元素，我们可以按如下步骤来做

1. 取 n 个元素的前 K 个元素构建一个小顶堆
2. 遍历第 K + 1 到  n 之间的元素，每一个元素都与小顶堆的堆顶元素进行比较，如果小于堆顶元素，不做任何操作，如果大于堆顶元素，则将堆顶元素替换成当前遍历的元素，再堆化以让其满足小顶的要求，这样遍历完成后此小顶堆的所有元素就是我们要求的 TopK。

每个元素堆化的时间复杂度是 O(logK)，n 个元素时间复杂度是 O(nlogK)，还是相当给力的！

**3、 TP99 是生产中的一个非常重要的指标，如何快速计算**先来解释下什么是 TP99，它指的是在一个时间段内（如5分钟），统计某个接口（或方法）每次调用所消耗的时间，并将这些时间按从小到大的顺序进行排序，取第99%的那个值作为 TP99 值，举个例子， 假设这个方法在 5 分钟内调用消耗时间为从 1 s 到 100 s 共 100 个数，则其 TP99 为 99，这个值为啥重要呢，对于某个接口来说，这个值越低，代表 99% 的请求都是非常快的，说明这个接口性能很好，反之，就说明这个接口需要改进，那怎么去求这个值呢？思路如下：

1. 创建一个大顶堆和一个小顶堆，大顶堆的堆顶元素比小顶堆的堆顶元素更小，大顶堆维护 99% 的请求时间，小顶堆维护 1% 的请求时间
2. 每产生一个元素（请求时间），如果它比大顶堆的堆顶元素小，则将其放入到大顶堆中，如果它比小顶堆的堆顶元素大，则将其插入到小顶堆中，插入后当然要堆化以让其符合大小顶堆的要求。
3. 上一步在插入的过程中需要注意一下，可能会导致大顶堆和小顶堆中元素的比例不为 99:1，此时就要做相应的调整，如果在将元素插入大顶堆之后，发现比例大于 99：1，将需将大顶堆的堆顶元素移到小顶堆中，再对两个堆堆化以让其符合大小顶堆的要求，同理，如果发现比例小于 99: 1，则需要将小顶堆的堆顶元素移到大顶堆来，再对两者进行堆化。

以上的大小顶堆调整后，则大顶堆的堆顶元素值就是所要求的 TP99 值。有人可能会说以上的这些应用貌似用快排或其他排序也能实现，没错，确实能实现，但是我们需要注意到，在**静态数据**下用快排确实没问题，但在动态数据上，如果每插入/删除一个元素对所有的元素进行快排，其实效率不是很高，由于要快排要全量排序，时间复杂度是 O(nlog n)，而堆排序就非常适合这种**对于动态数据的排序**,对于每个新添加的动态数据，将其插入到堆中，然后进行堆化，时间复杂度只有 O(logK)

## **总结**

堆是一种非常重要的数据结构，在对****动态数据**进行排序时性能很高**，优先级队列底层也是普遍采用堆来管理，所以掌握堆的基本操作十分重要。另外我们也知道了 Java 的优先级队列（PriorityQueue）也是用堆来实现的，所以再次说明了掌握基本的数据结构非常重要，对于理解上层应用的底层实现十分有帮助！