HashMap/Table和Heap

### 理解HashTable/Map

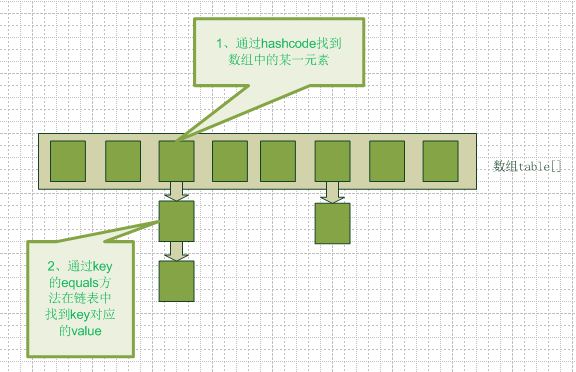
1. **结构**

HashMap和HashTable都使用哈希表来存储键值对。在数据结构上是基本相同的，都创建了一个继承自Map.Entry的私有的内部类Entry，每一个Entry对象表示存储在哈希表中的一个键值对。

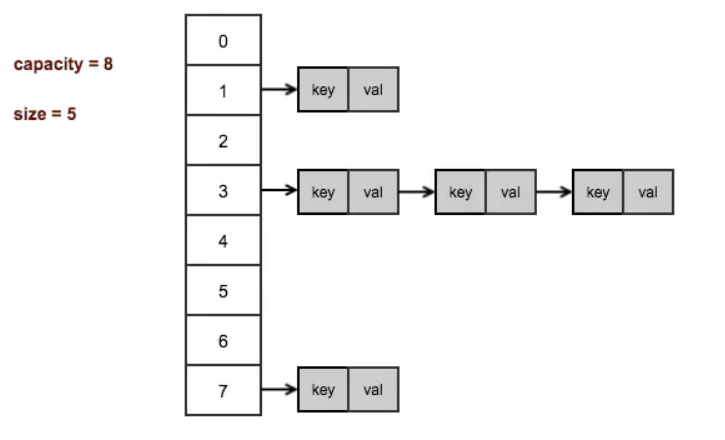
Entry对象唯一表示一个键值对，有四个属性：

-K key 键对象  
-V value 值对象  
-int hash 键对象的hash值  
-Entry entry 指向链表中下一个Entry对象，可为null，表示当前Entry对象在链表尾部

如下图示：



**在之前的作业中，有同学提出过相关的问题，所以这里我更深入地讲一下：**

Bucket就是这个table，当我们往hashMap或者HashTable中put元素的时候，先根据key的hash值得到这个元素在数组中的位置（即下标），然后就可以把这个元素放到对应的位置中了。****

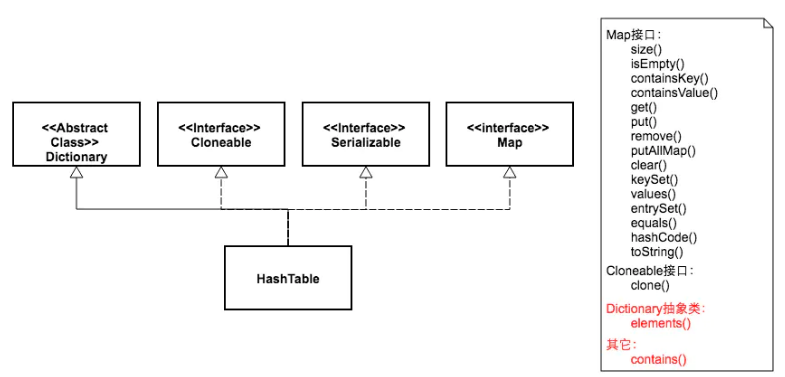
上图画出的是一个桶数量为8，存有5个键值对的HashMap/HashTable的内存布局情况。可以看到HashMap/HashTable内部创建有一个Entry类型的引用数组，用来表示哈希表，数组的长度，即是哈希桶的数量。而数组的每一个元素都是一个Entry引用，从Entry对象的属性里，也可以看出其是链表的节点，每一个Entry对象内部又含有另一个Entry对象的引用。

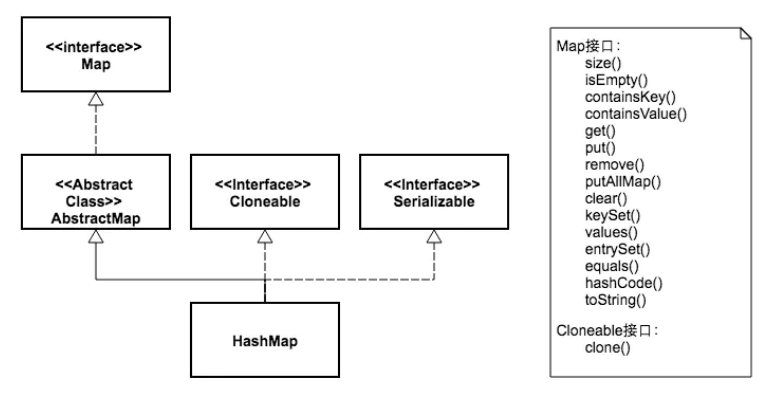
1. **HashMap和HashTable的区别**
2. 继承不同：

Hash Table继承了Dictionary class，implements Map接口

Hash Map继承了AbstractMap class，implements Map接口

具体的图可以看这个：





可惜，Dictionary类是一个已经被废弃的类（见其源码中的注释）。父类都被废弃，自然而然也没人用它的子类Hash table了。

1. 对于put的值的要求不同：

Hashtable既不支持Null key也不支持Null value。Hashtable的put()方法的注释中有说明。HashTable在遇到null时，会抛出NullPointerException异常

//以下代码及注释来自java.util.HashTable

public synchronized V put(K key, V value) {

// 如果value为null，抛出NullPointerException

if (value == null) {

throw new NullPointerException();

}

// 如果key为null，在调用key.hashCode()时抛出NullPointerException // ...

}

在HashMap中，null可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为null。当get()方法返回null值时，即可以表示 HashMap中没有该键，也可以表示该键所对应的值为null。因此，在HashMap中不能由get()方法来判断HashMap中是否存在某个键， 而应该用containsKey()方法来判断。HashMap在实现时对null做了特殊处理，将null的hashCode值定为了0，从而将其存放在哈希表的第0个bucket中。我们以put方法为例，看一看代码的细节：

//以下代码及注释来自java.util.HasMap

public V put(K key, V value) {

if (table == EMPTY\_TABLE) {

inflateTable(threshold);

}

// 当key为null时，调用putForNullKey特殊处理

if (key == null)

return putForNullKey(value);

// ...

}

private V putForNullKey(V value) {

// key为null时，放到table[0]也就是第0个bucket中

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null) {

V oldValue = e.value; e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(0, null, value, 0);

return null;

}

从这里我们可以看出，对于HashMap来说，所有的null value都被放在了table的index0的位置，所有其他的Entry都是从index为1开始一个个往下找的

1. 初始的capacity和扩容的方式：

HashTable默认的初始大小为11，之后每次扩充为原来的2n+1。HashMap默认的初始化大小为16，之后每次扩充为原来的2倍。如果在创建时给定了初始化大小，那么HashTable会直接使用你给定的大小，而HashMap会将其扩充为2的幂次方大小。也就是说HashTable会尽量使用素数、奇数。而HashMap则总是使用2的幂作为哈希表的大小。

所以单从这一点上看，HashTable的哈希表大小选择，似乎更高明些。但另一方面我们又知道，在取模计算时，如果模数是2的幂，那么我们可以直接使用位运算来得到结果，效率要大大高于做除法。所以从hash计算的效率上，又是HashMap更胜一筹。

1. Hash值的计算不同：

HashTable使用的是hash function，我们可以直接看java自带的jdk代码：

//以下代码及注释来自java.util.HashTable

// hash 不能超过Integer.MAX\_VALUE

//所以要取其最小的31个bit int hash = hash(key);

int index = (hash & 0x7FFFFFFF) % tab.length;

// 直接计算key.hashCode()

private int hash(Object k) {

// hashSeed will be zero if alternative hashing is disabled.

return hashSeed ^ k.hashCode();

}

HashMap为了加快hash的速度，将哈希表的大小固定为了2的幂。当然这引入了哈希分布不均匀的问题，所以HashMap为解决这问题，又对hash算法做了一些改动。HashMap由于使用了2的幂次方，所以在取模运算时不需要做除法，只需要位的与运算就可以了。但是由于引入的hash冲突加剧问题，HashMap在调用了对象的hashCode方法之后，又做了一些位运算在打散数据。我们可以看一下代码：

//以下代码及注释来自java.util.HashMap

int hash = hash(key);

int i = indexFor(hash, table.length);

// 在计算了key.hashCode()之后，做了一些位运算来减少哈希冲突

final int hash(Object k) {

int h = hashSeed;

if (0 != h && k instanceof String) {

return sun.misc.Hashing.stringHash32((String) k);

}

h ^= k.hashCode();

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

// 取模不再需要做除法

static int indexFor(int h, int length) {

// assert Integer.bitCount(length) == 1 : "length must be a non-zero power of 2";

return h & (length-1);

}

如果你有细心读代码，还可以发现一点，就是HashMap和HashTable在计算hash时都用到了一个叫hashSeed的变量。这是因为映射到同一个hash桶内的Entry对象，是以链表的形式存在的，而链表的查询效率比较低，所以HashMap/HashTable的效率对哈希冲突非常敏感，所以可以额外开启一个可选hash（hashSeed），从而减少哈希冲突。但是这个优化在JDK 1.8中已经去掉了，因为JDK 1.8中，映射到同一个哈希桶（数组位置）的Entry对象在LinkedList的长度大于8的时候，使用了红黑树来存储。但是在HashTable中却是始终使用LinkedList来储存hashCode相等的那些Entry的

1. **HashTable/Map的作用和算法example**

以空间换时间复杂度，增加所需要的空间，但是大幅度地减小时间上的开销

**题目：**

**给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target  的那 两个 整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。**

**Example1：**

输入：nums = [2,7,11,15], target = 9

输出：[0,1]

解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1]

**Example2：**

输入：nums = [3,2,4], target = 6

输出：[1,2]

**Example3：**

输入：nums = [3,3], target = 6

输出：[0,1]

**思路：**

**最容易想到的方法是枚举数组中的每一个数 x，寻找数组中是否存在 target - x。当我们使用遍历整个数组的方式寻找 target - x 时，需要注意到每一个位于 x 之前的元素都已经和 x 匹配过，因此不需要再进行匹配。而每一个元素不能被使用两次，所以我们只需要在 x 后面的元素中寻找 target – x, 时间复杂度为O(n^2)**

public int[] twoSum(int[] nums, int target) {

int n = nums.length;

for (int i = 0; i < n; ++i) {

for (int j = i + 1; j < n; ++j) {

if (nums[i] + nums[j] == target) {

return new int[]{i, j};

}

}

}

return new int[0];

}

1. **由于哈希查找的时间复杂度为 O(1)，所以可以利用哈希容器 map 降低时间复杂度**
2. **遍历数组 nums，i 为当前下标，每个值都判断map中是否存在 target-nums[i] 的 key 值**
3. **如果存在则找到了两个值，如果不存在则将当前的 (nums[i],i) 存入 map 中，继续遍历直到找到为止, 如果最终都没有结果则抛出异常**

public int[] twoSum(int[] nums, int target) {

Map<Integer, Integer> res = new HashMap<Integer, Integer>();

for (int i = 0; i < nums.length; ++i) {

if (res.containsKey(target - nums[i])) {

return new int[]{res.get(target - nums[i]), i};

}

res.put(nums[i], i);

}

return new int[0];

}

1. **复杂度分析:**

**时间复杂度：O(N)，其中 N 是数组中的元素数量。对于每一个元素 x，我们可以 O(1) 地寻找 target – x**

**空间复杂度：O(N)，其中 N 是数组中的元素数量。**

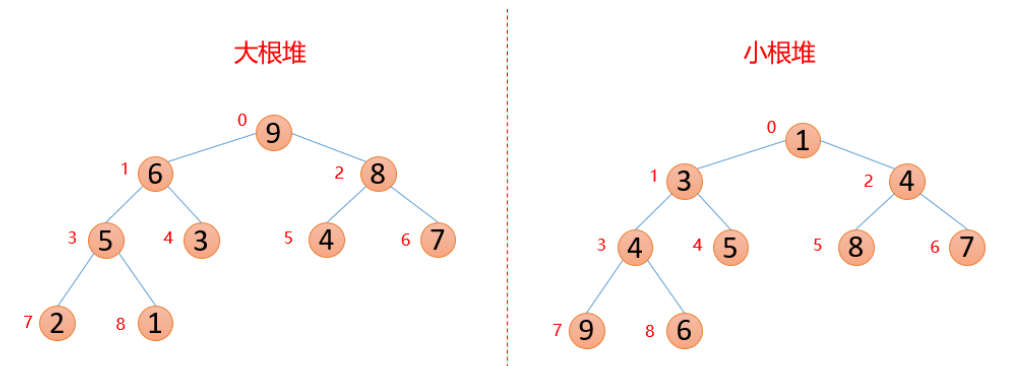
### Heap：

1. **什么是Heap：**

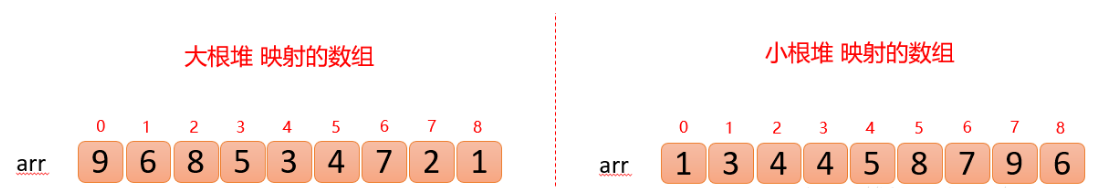
堆的结构可以分为大根堆（MaxHeap）和小根堆（MinHeap），是一个[完全二叉树](https://baike.baidu.com/item/%E5%AE%8C%E5%85%A8%E4%BA%8C%E5%8F%89%E6%A0%91/7773232?fr=aladdin)，而堆排序是根据堆的这种数据结构设计的一种排序

1. **Heap的结构与array**

性质：每个结点的值都大于其左孩子和右孩子结点的值，称之为大根堆；每个结点的值都小于其左孩子和右孩子结点的值，称之为小根堆。如下图：



我们对上面的图中每个数都进行了标记，上面的结构映射成数组就变成了下面这个样子：



1. **Heap在array中的index**

1. 父结点索引：( I – 1 ) / 2（这里计算机中的除以2，省略掉小数）

2. 左孩子索引：2 \* I + 1

3. 右孩子索引：2 \* I + 2

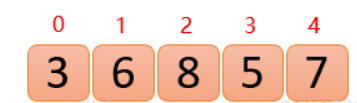
1. **构造Heap和heapSort**

**1. 首先将待排序的数组构造成一个大根堆，此时，整个数组的最大值就是堆结构的顶端**

**2. 将顶端的数与末尾的数交换，此时，末尾的数为最大值，剩余待排序数组个数为n-1**

**3. 将剩余的n-1个数再构造成大根堆，再将顶端数与n-1位置的数交换，如此反复执行，便能得到有序数组**

假设存在以下数组：



**主要思路：**

**第一次保证0~0位置满足heap要求**

**第二次保证0~1位置满足heap要求**

**第三次保证0~2位置满足heap要求...**

**直到保证0~n-1位置（每次新插入的数据都与其父结点进行比较，如果插入的数比父结点大，则与父结点交换，否则一直向上交换，直到小于等于父结点，或者来到了顶端）**

下面为对应的流程图：

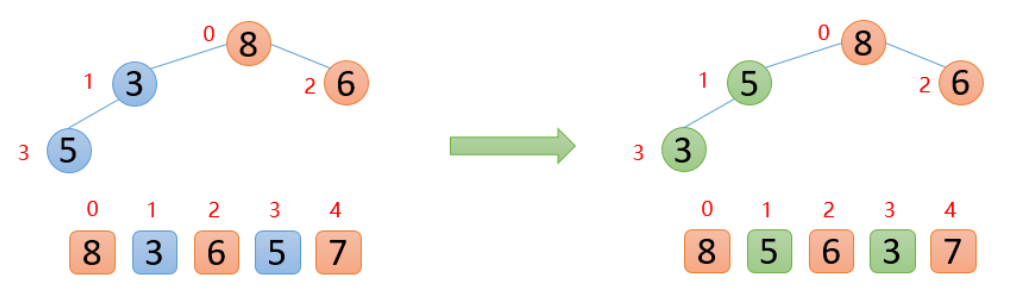
1. 插入6的时候，6大于他的父结点3，即arr(1)>arr(0)，则交换；此时，保证了0~1位置是大根堆结构，如下图：



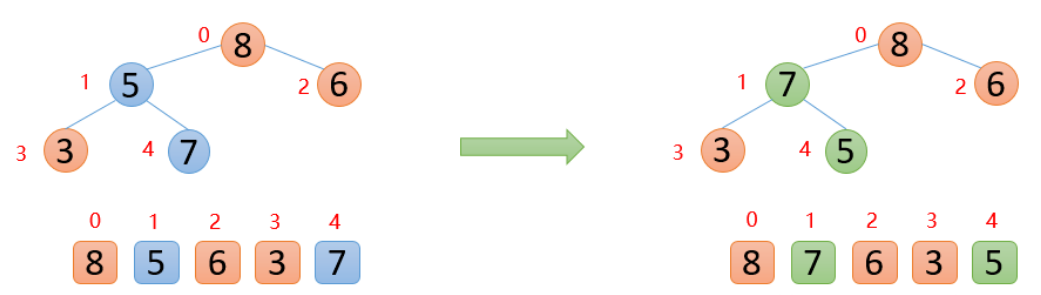
1. 插入8的时候，8大于其父结点6，即arr(2)>arr(0),则交换；此时，保证了0~2位置是大根堆结构，如下图：

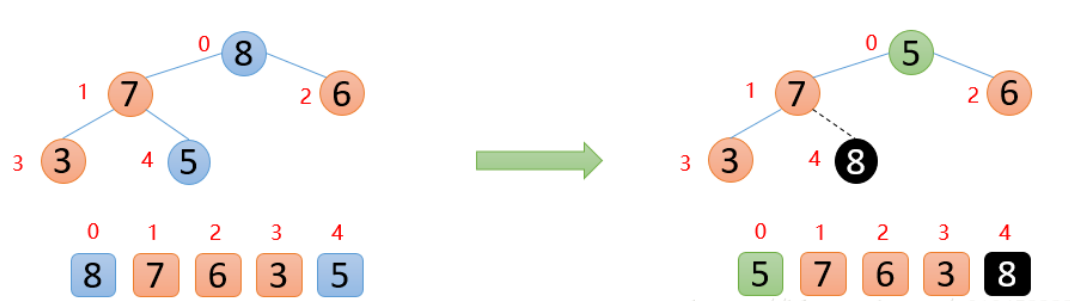


1. 插入5的时候，5大于其父结点3，则交换，交换之后，5又发现比8小，所以不交换；此时，保证了0~3位置大根堆结构，如下图：

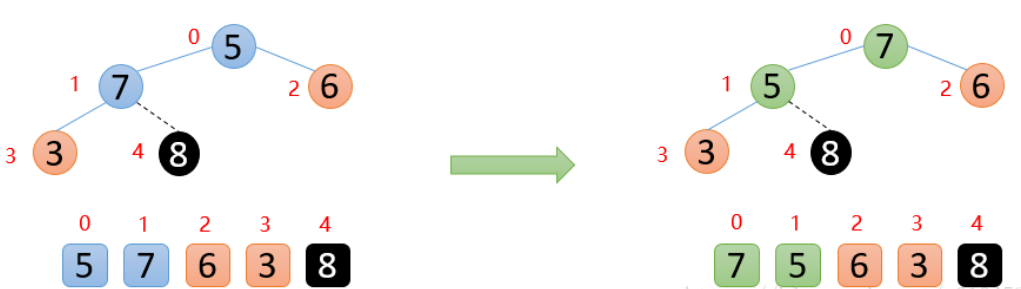


1. 插入7的时候，7大于其父结点5，则交换，交换之后，7又发现比8小，所以不交换；此时**整个数组已经是大根堆结构**

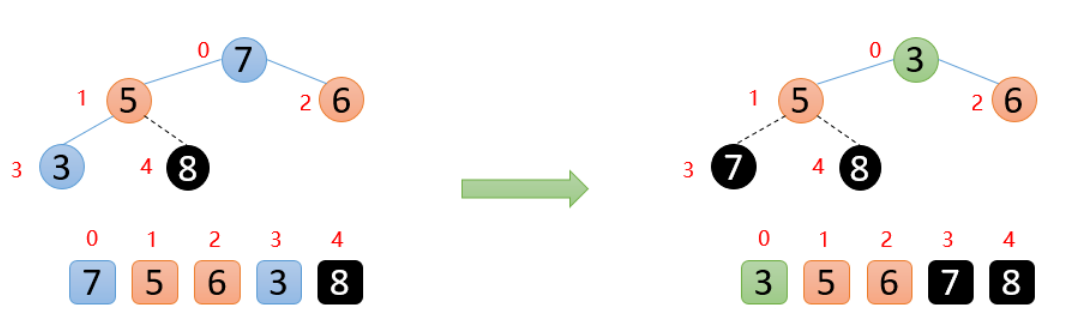


1. 此时，我们已经得到一个大根堆，下面将顶端的数与最后一位数交换，然后将剩余的数再构造成一个大根堆：
2. 此时最大数8已经来到末尾，则固定不动，后面只需要对顶端的数据进行操作即可，拿顶端的数与其左右孩子较大的数进行比较，如果顶端的数大于其左右孩子较大的数，则停止，如果顶端的数小于其左右孩子较大的数，则交换，然后继续与下面的孩子进行比较

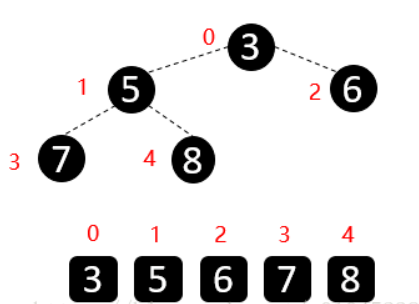
下图中，5的左右孩子中，左孩子7比右孩子6大，则5与7进行比较，发现5<7，则交换；交换后，发现5已经大于他的左孩子，说明剩余的数已经构成大根堆，后面就是重复固定最大值，然后构造大根堆



1. 如下图：顶端数7与末尾数3进行交换，固定好7



1. 剩余的数开始构造大根堆 ，然后顶端数与末尾数交换，固定最大值再构造大根堆，重复执行上面的操作，最终会得到有序数组



1. **总结**

**对于上面的操作，我们有如下的总结：**

**1、首先将无需数组构造成一个大根堆（新插入的数据与其父结点比较）**

**2、固定一个最大值，将剩余的数重新构造成一个大根堆，重复这样的过程**

1. **算法和应用**

**PriorityQueue是Heap的一个应用，它能自动地给每次添加的元素经行排序，然后poll的时候会自动poll出目前最小的那个元素，也就是说实现了MinHeap的思想**

**题目：**

**给定整数数组 nums 和整数 k，请返回数组中第**k**个最大的元素。**

**请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。**

**Example 1：**

输入: [3,2,1,5,6,4] 和 k = 2

输出: 5

**Example 2：**

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4

输出: 4

**思路：**

**把数组全部排序好，这样就可以拿到第k大的元素，这样是一种解法，但是我们是需要第K大的元素，不一定要全部排序好再去拿，只针对部分元素进行排序，这样的复杂度显然可以降低的，也就是可以转化为：使用堆排序来解决这个问题——建立一个大顶堆，做k−1 次删除操作后,堆顶元素就是我们要找的答案（堆排序过程中，不全部下沉，下沉nums.length-k+1,然后堆顶可以得到我们top k答案了）**

**代码示例：**

public int findKthLargest(int[] nums, int k) {

//构造小顶堆

PriorityQueue<Integer> queue = new PriorityQueue<Integer>((a,b)->a - b);

for(int i = 0;i < nums.length;i ++){

//当队列size小于k时，就加入元素；

if(queue.size() < k){

queue.add(nums[i]);

}

//否则判断小顶堆队首元素是否小于当前元素，如果小于说明有更大的元素要加进来；

else if(queue.peek() <= nums[i]){

queue.poll();

queue.add(nums[i]);

}

}

//返回队首元素

return queue.peek();

}