一、基础概念

1. 基本介绍

① PriorityQueue是优先级队列，它首先实现了队列接口(Queue)，与LinkedList类似，它的队列长度也没有限制，与一般队列的区别是，**它有优先级的概念，每个元素都有优先级，队头的元素永远都是优先级最高的**。

② PriorityQueue内部是用堆实现的，是一种比较通用的实现了堆的性质的数据结构。

2. Queue接口

(1) 基本介绍

public interface Queue<E> extends Collection<E> {

boolean add(E e);

boolean offer(E e);

E remove();

E poll();

E element();

E peek();

}

(2) Queue扩展了Collection，主要操作有三个：

① 在尾部添加元素 (add, offer)。

② 查看头部元素 (element, peek)，返回头部元素，但不改变队列。

③ 删除头部元素 (remove, poll)，返回头部元素，并且从队列中删除。

二、PriorityQueue底层实现

1. 基本属性

private static final int DEFAULT\_INITIAL\_CAPACITY = 11; //默认底层数组大小

transient Object[] queue //内存实际数组

private int size = 0; //数组中元素的数量

private final Comparator<? super E> comparator; //比较器对象

2. 构造方法

(1) 默认构造方法

public PriorityQueue() {

//调用另一个传参的方法，使用的是默认数组大小11，比较器对象为null

this(DEFAULT\_INITIAL\_CAPACITY, null);

}

(2) 传入数组大小的构造方法

public PriorityQueue(int initialCapacity) {

//调用另一个传参的方法，使用的是自定义数组大小,比较器对象为null

this(initialCapacity, null);

}

(3) public PriorityQueue(Comparator<? super E> comparator) {

//调用另一个传参的方法，使用的是默认数组大小11,比较器对象为自定义

this(DEFAULT\_INITIAL\_CAPACITY, comparator);

}

(4) 核心构造方法 – 以上三个都是调用这个方法

public PriorityQueue(int initialCapacity,

Comparator<? super E> comparator) {

if (initialCapacity < 1)

throw new IllegalArgumentException();、

//新建一个Object数组

this.queue = new Object[initialCapacity];

this.comparator = comparator;

}

说明：

与TreeMap/TreeSet类似，为了保持一定顺序，PriorityQueue要求，要么元素实现Comparable接口，要么传递一个比较器Comparator。

三、具体方法实现

1. 往队尾添加元素add()/offer()（入队）

(1) public boolean offer(E e) {

if (e == null)

throw new NullPointerException();

modCount++;

int i = size;

//如果数组长度不够，调用grow方法进行动态扩容

if (i >= queue.length)

grow(i + 1);

size = i + 1;

//当第一次添加时，直接添加到第一个位置；否则放入最后一个位置，并且向上调整

if (i == 0)

queue[0] = e;

else

siftUp(i, e);

return true;

}

(2) private void grow(int minCapacity) {

int oldCapacity = queue.length;

//如果原长度小于64，则扩充为原长度的两倍+2，否则扩充为150%

int newCapacity = oldCapacity + ((oldCapacity < 64) ?

(oldCapacity + 2) : (oldCapacity >> 1));

//如果大于最大值，就取integer的最大值

if (newCapacity - MAX\_ARRAY\_SIZE > 0)

\*以“” newCapacity = hugeCapacity(minCapacity);

//拷贝数组到扩充后的新数组

queue = Arrays.copyOf(queue, newCapacity);

}

(3) private void siftUpUsingComparator(int k, E x) { //k表示size，x表示插入的对象

while (k > 0) {

int parent = (k - 1) >>> 1; //得到父节点索引

Object e = queue[parent]; //得到父节点对象

//k表示当前索引的位置，p表示k的父节点位置，e表示父节点对象

//当e大于x时，则将e下调到k的位置，将x上升到p的位置，sitUp

//最后k的位置就是要插入的位置

if (comparator.compare(x, (E) e) >= 0)

break;

queue[k] = e;

k = parent;

}

queue[k] = x;

}

2. 查看头部元素

public E peek() {

return (size == 0) ? null : (E) queue[0];

}

3. 删除头部元素（出队）

(1) public E poll() {

if (size == 0)

return null;

int s = --size;

modCount++;

//保存头部元素

E result = (E) queue[0];

//保存最后一个元素

E x = (E) queue[s];

//将最后一个元素置为null

queue[s] = null;

//当元素超过两个时，对x最后一个元素进行sitDown

if (s != 0)

siftDown(0, x);

//返回被删除的头部元素

return result;

}

(2) private void siftDownUsingComparator(int k, E x) {

//表示最后一个非叶子节点

int half = size >>> 1;

//当k为叶子节点时就不用再寻找，k>=half就表示为是叶子节点

while (k < half) {

//child表示较小的子节点编号，一开始表示左子节点

int child = (k << 1) + 1;

Object c = queue[child];

//表示右子节点

int right = child + 1;

//当左子节点大于右子节点时，将child更改为右子节点

//找出较小的那个子节点

if (right < size &&comparator.compare((E) c, (E) queue[right]) > 0)

c = queue[child = right];

//当X小于child时就满足条件，如果大于就交换两者位置

if (comparator.compare(x, (E) c) <= 0)

break;

queue[k] = c;

//k表示当前位置

k = child;

}

queue[k] = x;

}

4. 查找元素

(1) public boolean contains(Object o) {

return indexOf(o) != -1;

}

(2) private int indexOf(Object o) {

//遍历数组，当找到返回其索引值，否则返回-1

if (o != null) {

for (int i = 0; i < size; i++)

if (o.equals(queue[i]))

return i;

}

return -1;

}

5. 根据值删除元素

(1) public boolean remove(Object o) {

//找出其索引位置

int i = indexOf(o);

if (i == -1)

return false;

else {

//调用removeAt方法删除

removeAt(i);

return true;

}

}

(2) private E removeAt(int i) {

modCount++;

int s = --size;

//当删除的元素为最后一个元素时，直接设为null就行。

if (s == i)

queue[i] = null;

else {

//取出最后一个元素，并将最后一个位置设为空

E moved = (E) queue[s];

queue[s] = null;

//由于最后一个元素肯定会比较大，所以先将元素sitDown

siftDown(i, moved);

//如果sitDown失败，就是位置没有发生变动，再将元素sitUp

if (queue[i] == moved) {

siftUp(i, moved);

//说明向上移动成功，返回数组最后一个元素，其他情况返回null

if (queue[i] != moved)

return moved;

}

}

return null;

}

四、总结

1. **实现了优先级队列**，最先出队的总是优先级最高的，即排序中的第一个。

2. **优先级可以有相同的，内部元素不是完全有序的**，如果遍历输出，除了第一个，其他没有特定顺序。

3. 查看头部元素的效率很高，为O(1)，入队、出队效率比较高，为O(log2(N))，构建堆heapify的效率为O(N)。

4. 根据值查找和删除元素的效率比较低，为O(N)。

五、堆的应用

1. 求前k个最大元素

(1) 背景

**求前K个最大的元素，元素个数不确定，数据量可能很大，甚至源源不断到来**，但需要知道到目前为止的最大的前K个元素。这个问题的变体有：求前K个最小的元素，求第K个最大的，求第K个最小的。

(2) 利用堆解决问题的思路

① 一个基本的思路是**维护一个长度为K的数组**，最前面的K个元素就是目前最大的K个元素，以后每来一个新元素的时候，都先找数组中的最小值，将新元素与最小值相比，如果小于最小值，则什么都不用变，如果大于最小值，则将最小值替换为新元素。

② 使用最小堆维护这K个元素，最小堆中，根即第一个元素永远都是最小的，新来的元素与根比就可以了，如果小于根，则堆不需要变化，否则用新元素替换根，然后向下调整堆即可，调整的效率为O(log2(K))，这样，总体的效率就是O(N\*log2(K))，这个效率非常高，而且存储成本也很低。**使用最小堆之后，第K个最大的元素也很容易获得，它就是堆的根。**

(3) 源码

public void add(E e) {

//当数组长度小于k时，直接添加元素

if(p.size()<k){

p.add(e);

return;

}

Comparable<? super E> head = (Comparable<? super E>)p.peek();

if(head.compareTo(e)>0){

//当添加的元素小于头部元素时，不添加

return;

}

//新元素替换掉原来的最小值成为头部元素

p.poll();

p.add(e);

}

2. 求中值

(1) 背景

中值就排序后中间那个元素的值，如果元素个数为奇数，中值是没有歧义的，但如果是偶数，中值可能有不同的定义，可以为偏小的那个，也可以是偏大的那个，或者两者的平均值，或者任意一个，这里，我们假定任意一个都可以。

(2) 思路（数组的大小是固定的）

一个简单的思路是排序，排序后取中间那个值就可以了，排序可以使用Arrays.sort()方法，效率为O(N\*log2(N))。

(3) 利用堆解决的思路（数组里的值是不断增大的）

可以使用两个堆，一个最大堆，一个最小堆，思路如下：

① 假设当前的中位数为m，最大堆维护的是<=m的元素，最小堆维护的是>=m的元素，**但两个堆都不包含m。**

② 当新的元素到达时，比如为e，将e与m进行比较，若e<=m，则将其加入到最大堆中，否则将其加入到最小堆中。

③ 第二步后，如果此时最小堆和最大堆的元素个数的差值>=2 ，则将m加入到元素个数少的堆中，然后从元素个数多的堆将根节点移除并赋值给m。

(4) 具体流程

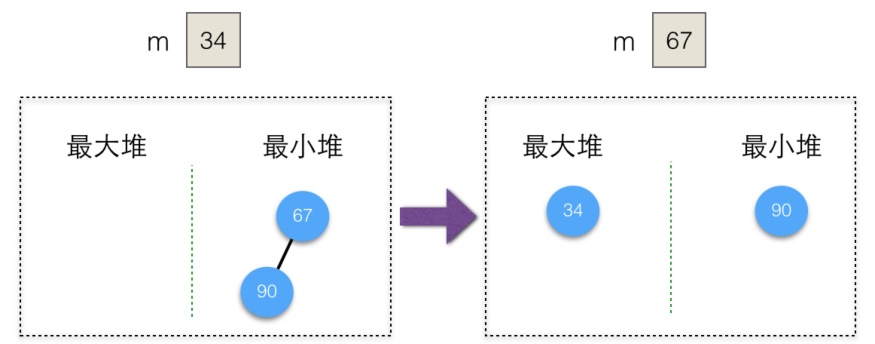
① 输入元素依次是：34, 90, 67, 45,1

② 输入第一个元素时，m即为34。

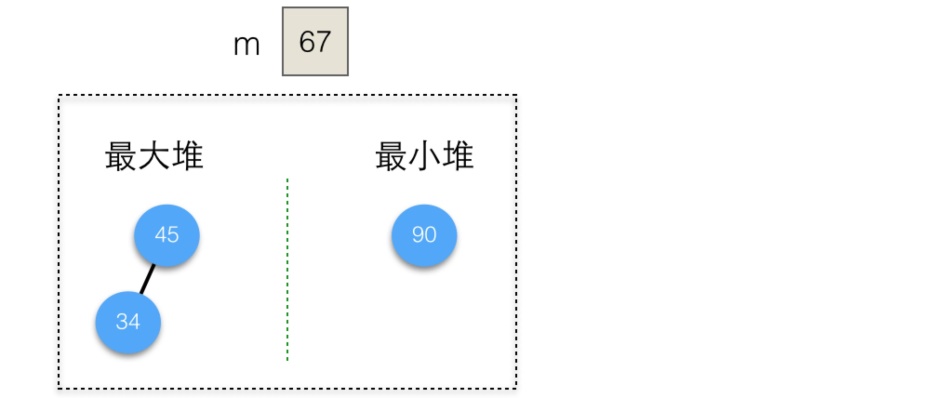
输入第二个元素时，90大于34，加入最小堆，中值不变，如下所示：



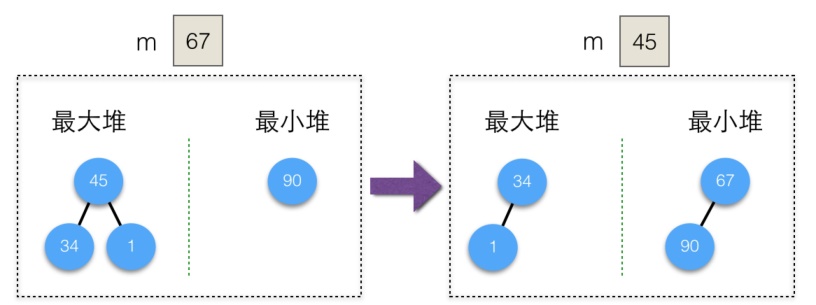
③ 输入第三个元素时，67大于34，加入最小堆，但加入最小堆后，最小堆的元素个数为2，需调整中值和堆，现有中值34加入到最大堆中，最小堆的根67从最小堆中删除并赋值给m，如下图所示：



④ 输入第四个元素45时，45小于67，加入最大堆，中值不变，如下图所示：



⑤ 输入第五个元素1时，1小于67，加入最大堆，此时需调整中值和堆，现有中值67加入到最小堆中，最大堆的根45从最大堆中删除并赋值给m，如下图所示：



(5) 核心方法

public void add(E e){

if(m==null){ //第一个元素

m = e;

return;

}

if(compare(e, m)<=0){

//小于中值, 加入最大堆

maxP.add(e);

}else{

minP.add(e);

}

if(minP.size()-maxP.size()>=2){

//最小堆元素个数多，即大于中值的数多

//将m加入到最大堆中，然后将最小堆中的根移除赋给m

maxP.add(this.m);

this.m = minP.poll();

}else if(maxP.size()-minP.size()>=2){

minP.add(this.m);

this.m = maxP.poll();

}

}