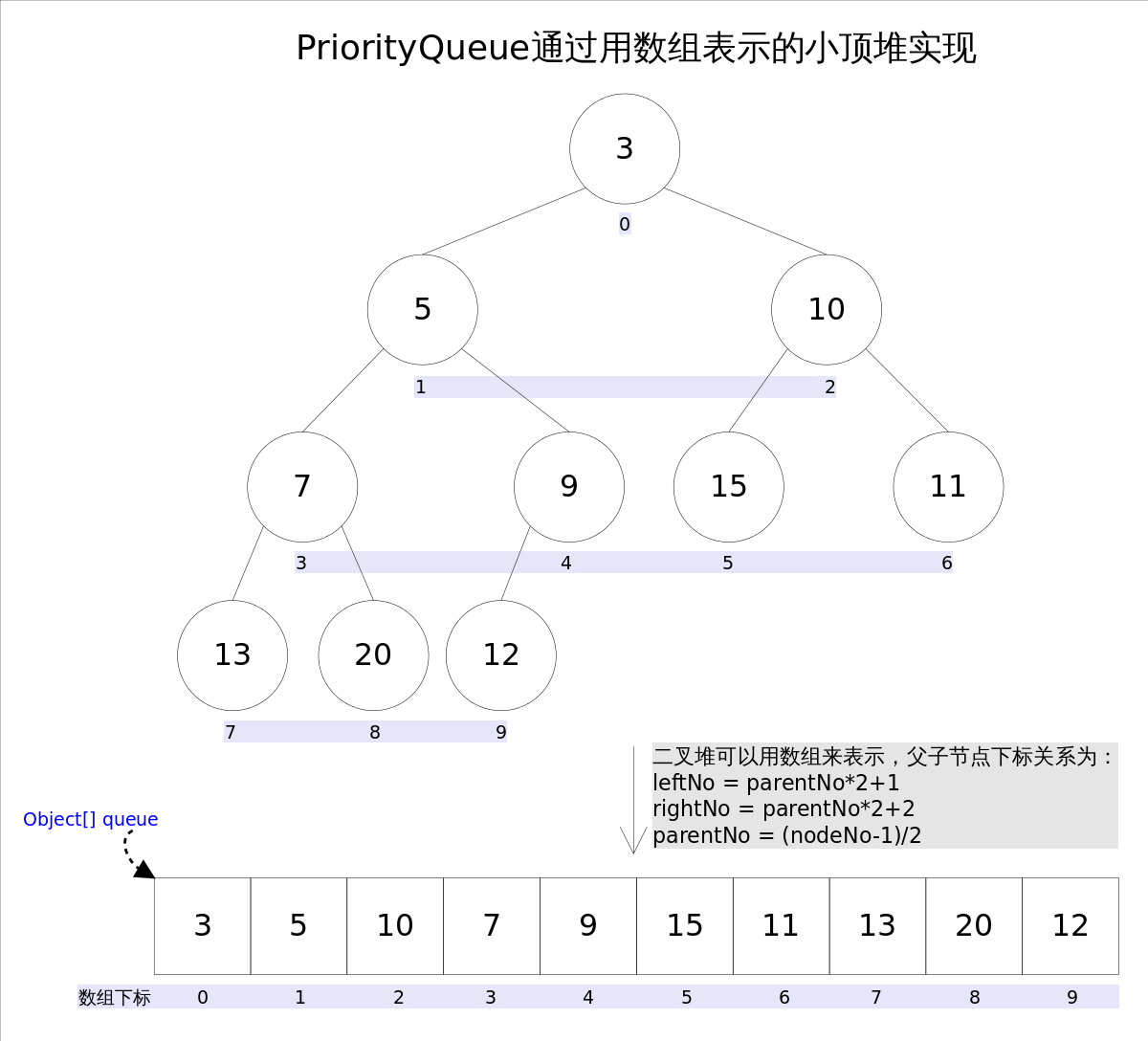
**1.总体介绍**

PriorityQueue通过二叉小顶堆实现，可以用一棵完全二叉树表示。优先队列，能保证每次取出的元素都是队列中权值最小的。元素大小的评判可以采用自然排序也可以通过传入的comparator比较器进行排序。

Java中的PriorityQueue实现了Queue接口，不允许放入null元素，其通过堆实现，具体说是通过完全二叉树实现的小顶堆（任意一个非叶子节点的权值都不大于其左右节点的权值），也就是意味着可以通过数组来作为PriorityQueue的底层实现。



**2.属性分析**

|  |
| --- |
| **默认初始容量11**  **private** **static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 11;  非私有以简化嵌套类访问，定义数组，存放数据  **transient** Object[] queue;  **private** **int** size = 0;数组里元素的个数  比较器，用于定义比较规则  **private** **final** Comparator<? **super** E> comparator;  被修改的次数  **transient** **int** modCount = 0; |

**3.构造函数**

|  |
| --- |
| **无参构造器，使用默认容量*DEFAULT\_INITIAL\_CAPACITY，调用*PriorityQueue(int initialCapacity,Comparator<? super E> comparator)**  **public** PriorityQueue() {  **this**(***DEFAULT\_INITIAL\_CAPACITY***, **null**);  }  **初始容量构造器，使用riorityQueue(int initialCapacity,Comparator<? super E> comparator)完成**  **public** PriorityQueue(**int** initialCapacity) {  **this**(initialCapacity, **null**);  }  **默认容量，传入比较器定义比较规则，调用riorityQueue(int initialCapacity, Comparator<? super E> comparator)**  **public** PriorityQueue(Comparator<? **super** E> comparator) {  **this**(***DEFAULT\_INITIAL\_CAPACITY***, comparator);  }  **核心构造器，为queue和comparator初始化**  **public** PriorityQueue(**int** initialCapacity,  Comparator<? **super** E> comparator) {  **if** (initialCapacity < 1)  **throw** **new** IllegalArgumentException();  **this**.queue = **new** Object[initialCapacity];  **this**.comparator = comparator;  }  **如果是sortedset类型，使用sortedset的比较器调用**initElementsFromCollection方法，如果是优先队列类型，就使用优先队列的比较器，调用initFromPriorityQueue，如果是其他类型，采用自然排序，调用initFromCollection  **public** PriorityQueue(Collection<? **extends** E> c) {  **if** (c **instanceof** SortedSet<?>) {  SortedSet<? **extends** E> ss = (SortedSet<? **extends** E>) c;  **this**.comparator = (Comparator<? **super** E>) ss.comparator();  initElementsFromCollection(ss);  }  **else** **if** (c **instanceof** PriorityQueue<?>) {  PriorityQueue<? **extends** E> pq = (PriorityQueue<? **extends** E>) c;  **this**.comparator = (Comparator<? **super** E>) pq.comparator();  initFromPriorityQueue(pq);  }  **else** {  **this**.comparator = **null**;  initFromCollection(c);  }  }  ----------------------------------------------------  **private** **void** initElementsFromCollection(Collection<? **extends** E> c) {  Object[] a = c.toArray();  **将c变为object数组，如果不是object数组，就重新copy成object数组**  **if** (a.getClass() != Object[].**class**)  a = Arrays.*copyOf*(a, a.length, Object[].**class**);  **int** len = a.length;  **此处len == 1 || this.comparator != null暂不明白，主要目的是不让数组中有null**  **if** (len == 1 || **this**.comparator != **null**)  **for** (**int** i = 0; i < len; i++)  **if** (a[i] == **null**)  **throw** **new** NullPointerException();  此时分为有序和无序，如果是有序，sortedset，直接数组输出就是有序数组，即符合小根堆；如果无序，也将其转为数组，并赋值为queue，然后使用heapify进行调整  **this**.queue = a;  **this**.size = a.length;  }  ----------------------------------------------------------  如果是priorityQueue类型，直接转为数组赋值，不需重新调整，**这里不明白else什么情况会出现**  **private** **void** initFromPriorityQueue(PriorityQueue<? **extends** E> c) {  **if** (c.getClass() == PriorityQueue.**class**) {  **this**.queue = c.toArray();  **this**.size = c.size();  } **else** {  initFromCollection(c);  }  }  先调用initElementsFromCollection给queue赋值，然后进行调整  **private** **void** initFromCollection(Collection<? **extends** E> c) {  initElementsFromCollection(c);  heapify();堆处理，后面详讲  }  --------------------------------------------------  **public** PriorityQueue(PriorityQueue<? **extends** E> c) {  **this**.comparator = (Comparator<? **super** E>) c.comparator();  initFromPriorityQueue(c);  }  **public** PriorityQueue(SortedSet<? **extends** E> c) {  **this**.comparator = (Comparator<? **super** E>) c.comparator();  initElementsFromCollection(c);  } |

**4.扩容机制**

|  |
| --- |
| private static final int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE - 8;    扩容机制；在size》=queue.length时，会调用grow进行扩容  **private** **void** grow(**int** minCapacity) {  **int** oldCapacity = queue.length;  // Double size if small; else grow by 50%  **int** newCapacity = oldCapacity + ((oldCapacity < 64) ?  (oldCapacity + 2) :  (oldCapacity >> 1));  // overflow-conscious code  **if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0)  newCapacity = *hugeCapacity*(minCapacity);  queue = Arrays.*copyOf*(queue, newCapacity);  }  获取oldCapacity=queue.length；如果old小于64，使用new = old + old + 2；  如果old大于64，使用new=old + old / 2;如果newCapacity比***MAX\_ARRAY\_SIZE***大，就调用hugeCapacity和ArrayList一样；最后复制原数组的元素，并返回一个新长度的数组  --------------------------------------------------------  **private** **static** **int** hugeCapacity(**int** minCapacity) {  **if** (minCapacity < 0) // overflow  **throw** **new** OutOfMemoryError();  **return** (minCapacity > ***MAX\_ARRAY\_SIZE***) ?  Integer.***MAX\_VALUE*** :  ***MAX\_ARRAY\_SIZE***;  } |

**5.添加元素**

|  |
| --- |
| **实际调用offer**  **public** **boolean** add(E e) {  **return** offer(e);  }  插入元素  **public** **boolean** offer(E e) {  **if** (e == **null**)  **throw** **new** NullPointerException();  modCount++;  **int** i = size;  **if** (i >= queue.length)  grow(i + 1);  size = i + 1;  **if** (i == 0)  queue[0] = e;  **else**  siftUp(i, e);  **return** **true**;  }  如果为空抛出异常，如果size>=queue.length，扩容，否则size+1；如果i=0；  表明queue为空，直接将第一个元素进行赋值，如果i>0;当前元素和上层元素进行调整，插入元素时从下往上，删除元素是，将最后一个元素放到堆顶，然后从上往下调整  ----------------------------------------------------  **private** **void** siftUp(**int** k, E x) {  **if** (comparator != **null**)  siftUpUsingComparator(k, x);  **else**  siftUpComparable(k, x);  }  如果比较器为空，使用x自带的排序方法或者实现comparable的compareTo方法，如果比较器不为空，使用比较器排序  --------------------------------------------------------------  使用比较器的向上调整  **private** **void** siftUpUsingComparator(**int** k, E x) {  **while** (k > 0) {  **int** parent = (k - 1) >>> 1;  Object e = queue[parent];  **if** (comparator.compare(x, (E) e) >= 0)  **break**;  queue[k] = e;  k = parent;  }  queue[k] = x;  }  K表示插入后的下标即size-1，k>0时循环：得到新插入节点的父亲的下标，parent=（k-1）/2，如果x比父亲大或相等，结束，queue[k] = x，插入后就是小根堆，如果x比父亲小，由于父亲肯定比孩子小，那么将父亲的值给插入的位置queue[k] = e;，继续向上找比x小的数，知道找到，或者到堆顶，结束  -------------------------------------------------------------  原理同上，只是比较的方法不同  **private** **void** siftUpComparable(**int** k, E x) {  Comparable<? **super** E> key = (Comparable<? **super** E>) x;  **while** (k > 0) {  **int** parent = (k - 1) >>> 1;  Object e = queue[parent];  **if** (key.compareTo((E) e) >= 0)  **break**;  queue[k] = e;  k = parent;  }  queue[k] = key;  } |

**6.查找，取出元素**

|  |
| --- |
| **返回堆顶元素，就是数组第一个**  **public** E peek() {  **return** (size == 0) ? **null** : (E) queue[0];  }  返回并移除堆顶元素，移除之后得调整  **public** E poll() {  **if** (size == 0)  **return** **null**;  **int** s = --size;  modCount++;  E result = (E) queue[0];  E x = (E) queue[s];  queue[s] = **null**;  **if** (s != 0)  siftDown(0, x);  **return** result;  }  如果size 为0，返回null，否则，size-1，得到堆顶元素result，然后得到最后一个元素赋值为x，把最后一个元素设为null；然后从堆顶开始，与x进行比较调整  ------------------------------------------------------------------  如果comparator不为空，调用siftDownUsingComparator调整，如果不为空，使用自然排序或者comparable实现的compareTo方法。  **private** **void** siftDown(**int** k, E x) {  **if** (comparator != **null**)  siftDownUsingComparator(k, x);  **else**  siftDownComparable(k, x);  }  ----------------------------------------------------------------  **private** **void** siftDownComparable(**int** k, E x) {  Comparable<? **super** E> key = (Comparable<? **super** E>)x;  **int** half = size >>> 1; // loop while a non-leaf  **while** (k < half) {  **int** child = (k << 1) + 1; // assume left child is least  Object c = queue[child];  **int** right = child + 1;  **if** (right < size &&  ((Comparable<? **super** E>) c).compareTo((E) queue[right]) > 0)  c = queue[child = right];  **if** (key.compareTo((E) c) <= 0)  **break**;  queue[k] = c;  k = child;  }  queue[k] = key;  }  使用x的comparable接口方式，得到最后一个元素的父亲，就是最后一层非叶子节点层，即倒数第二层；即half=size/2；如果k<half，循环进入；得到k的左右孩子child和right，如果右孩子的下标比size大或者相等，则没有右孩子；因此if条件，right<size并且左孩子>右孩子；child默认为小的那个下标，如果右孩子小，就将right赋值为child，记录小的对象为c；将c与key进行比较，如果key小于等于c，就结束循环，调整完成，否则将c赋值为queue[k]，k = child，继续向下调整。  -------------------------------------------------------------------  除了比较方法不一样，其他都一样  **private** **void** siftDownUsingComparator(**int** k, E x) {  **int** half = size >>> 1;  **while** (k < half) {  **int** child = (k << 1) + 1;  Object c = queue[child];  **int** right = child + 1;  **if** (right < size &&  comparator.compare((E) c, (E) queue[right]) > 0)  c = queue[child = right];  **if** (comparator.compare(x, (E) c) <= 0)  **break**;  queue[k] = c;  k = child;  }  queue[k] = x;  } |

**7.删除元素**

|  |
| --- |
| 删除指定元素，先得到元素的下标，然后调用removeAt进行删除  **public** **boolean** remove(Object o) {  **int** i = indexOf(o);  **if** (i == -1)  **return** **false**;  **else** {  removeAt(i);  **return** **true**;  }  }  ----------------------------------------------------  **private** **int** indexOf(Object o) {  **if** (o != **null**) {  **for** (**int** i = 0; i < size; i++)  **if** (o.equals(queue[i]))  **return** i;  }  **return** -1;  }  遍历数组，直到equal方法为true  ----------------------------------------------------------------  **private** E removeAt(**int** i) {  // assert i >= 0 && i < size;  modCount++;  **int** s = --size;  **if** (s == i) // removed last element  queue[i] = **null**;  **else** {  E moved = (E) queue[s];  queue[s] = **null**;  siftDown(i, moved);  **if** (queue[i] == moved) {  siftUp(i, moved);  **if** (queue[i] != moved)  **return** moved;  }  }  **return** **null**;  }  如果下标与-1后的size相同，说明删除最后一个元素，直接将其值置为null，否则，得到最后一个元素，将最后一个元素置为null，从第i个下标开始调整。  使用==判断相等  **boolean** removeEq(Object o) {  **for** (**int** i = 0; i < size; i++) {  **if** (o == queue[i]) {  removeAt(i);  **return** **true**;  }  }  **return** **false**;  } |