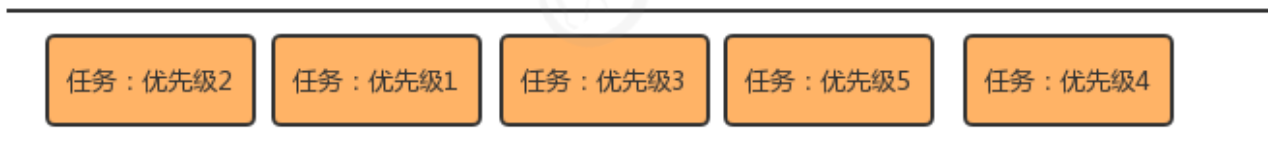


一、优先队列

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在某些情况下，我们可能需要找出队列中的最大值或者最小值，例如使用一个队列保存计算机的任务，一般情况下计算机的任务都是有优先级的，我们需要在这些计算机的任务中找出优先级最高的任务先执行，执行完毕后就需要把这个任务从队列中移除。普通的队列要完成这样的功能，需要每次遍历队列中的所有元素，比较并找出最大值，效率不是很高，这个时候，我们就可以使用一种特殊的队列来完成这种需求，优先队列。

计算机任务队列



优先队列按照其作用不同，可以分为以下两种：

最大优先队列：

可以获取并删除队列中最大的值

最小优先队列：

可以获取并删除队列中最小的值

1.1 最大优先队列

我们之前学习过堆，而堆这种结构是可以方便的删除最大的值，所以，接下来我们可以基于堆区实现最大优先队列。

1.1.1 最大优先队列API设计

类名	MaxPriorityQueue>
构造方法	MaxPriorityQueue(int capacity)：创建容量为capacity的MaxPriorityQueue对象
成员方法	1.private boolean less(int i,int j)：判断堆中索引i处的元素是否小于索引j处的元素 2.private void exch(int i,int j):交换堆中i索引和j索引处的值 3.public T delMax():删除队列中最大的元素,并返回这个最大元素 4.public void insert(T t)：往队列中插入一个元素 5.private void swim(int k):使用上浮算法，使索引k处的元素能在堆中处于一个正确的位置 6.private void sink(int k):使用下沉算法，使索引k处的元素能在堆中处于一个正确的位置 7.public int size():获取队列中元素的个数 8.public boolean isEmpty():判断队列是否为空
成员变量	1.private T[] imtes：用来存储元素的数组 2.private int N：记录堆中元素的个数

1.1.2 最大优先队列代码实现

```
1 //最大优先队列代码
2 public class MaxPriorityQueue<T extends Comparable<T>> {
3     //存储堆中的元素
4     private T[] items;
5     //记录堆中元素的个数
6     private int N;
7
8
9     public MaxPriorityQueue(int capacity) {
10         items = (T[]) new Comparable[capacity+1];
11         N = 0;
12     }
13
14     //获取队列中元素的个数
15     public int size() {
16         return N;
17     }
18
19     //判断队列是否为空
20     public boolean isEmpty() {
21         return N == 0;
22     }
23
24     //判断堆中索引i处的元素是否小于索引j处的元素
25     private boolean less(int i, int j) {
26         return items[i].compareTo(items[j]) < 0;
27     }
28
29     //交换堆中i索引和j索引处的值
30     private void exch(int i, int j) {
31         T tmp = items[i];
```

```

32     items[i] = items[j];
33     items[j] = tmp;
34 }
35
36 //往堆中插入一个元素
37 public void insert(T t) {
38     items[++N] = t;
39     swim(N);
40 }
41
42 //删除堆中最大的元素,并返回这个最大元素
43 public T delMax() {
44     T max = items[1];
45     //交换索引1处和索引N处的值
46     exch(1, N);
47     //删除最后位置上的元素
48     items[N] = null;
49     N--; //个数-1
50     sink(1);
51     return max;
52 }
53
54 //使用上浮算法,使索引k处的元素能在堆中处于一个正确的位置
55 private void swim(int k) {
56     //如果已经到了根结点,就不需要循环了
57     while (k > 1) {
58         //比较当前结点和其父结点
59         if (less(k / 2, k)) {
60             //父结点小于当前结点,需要交换
61             exch(k / 2, k);
62         }
63         k = k / 2;
64     }
65 }
66
67 //使用下沉算法,使索引k处的元素能在堆中处于一个正确的位置
68 private void sink(int k) {
69     //如果当前已经是最底层了,就不需要循环了
70     while (2 * k <= N) {
71         //找到子结点中的较大者
72         int max = 2 * k;
73         if (2 * k + 1 <= N) { //存在右子结点
74             if (less(2 * k, 2 * k + 1)) {
75                 max = 2 * k + 1;
76             }
77         }
78
79         //比较当前结点和子结点中的较大者,如果当前结点不小,则结束循环
80         if (!less(k, max)) {
81             break;
82         }
83         //当前结点小,则交换,
84
85         exch(k, max);

```

```

85         k = max;
86     }
87 }
88 }
89
90 //测试代码
91 public class Test {
92     public static void main(String[] args) throws Exception {
93         String[] arr = {"S", "O", "R", "T", "E", "X", "A", "M", "P", "L", "E"};
94         MaxPriorityQueue<String> maxpq = new MaxPriorityQueue<>(20);
95         for (String s : arr) {
96             maxpq.insert(s);
97         }
98         System.out.println(maxpq.size());
99         String del;
100         while(!maxpq.isEmpty()){
101             del = maxpq.delMax();
102             System.out.print(del+",");
103         }
104     }
105 }
106

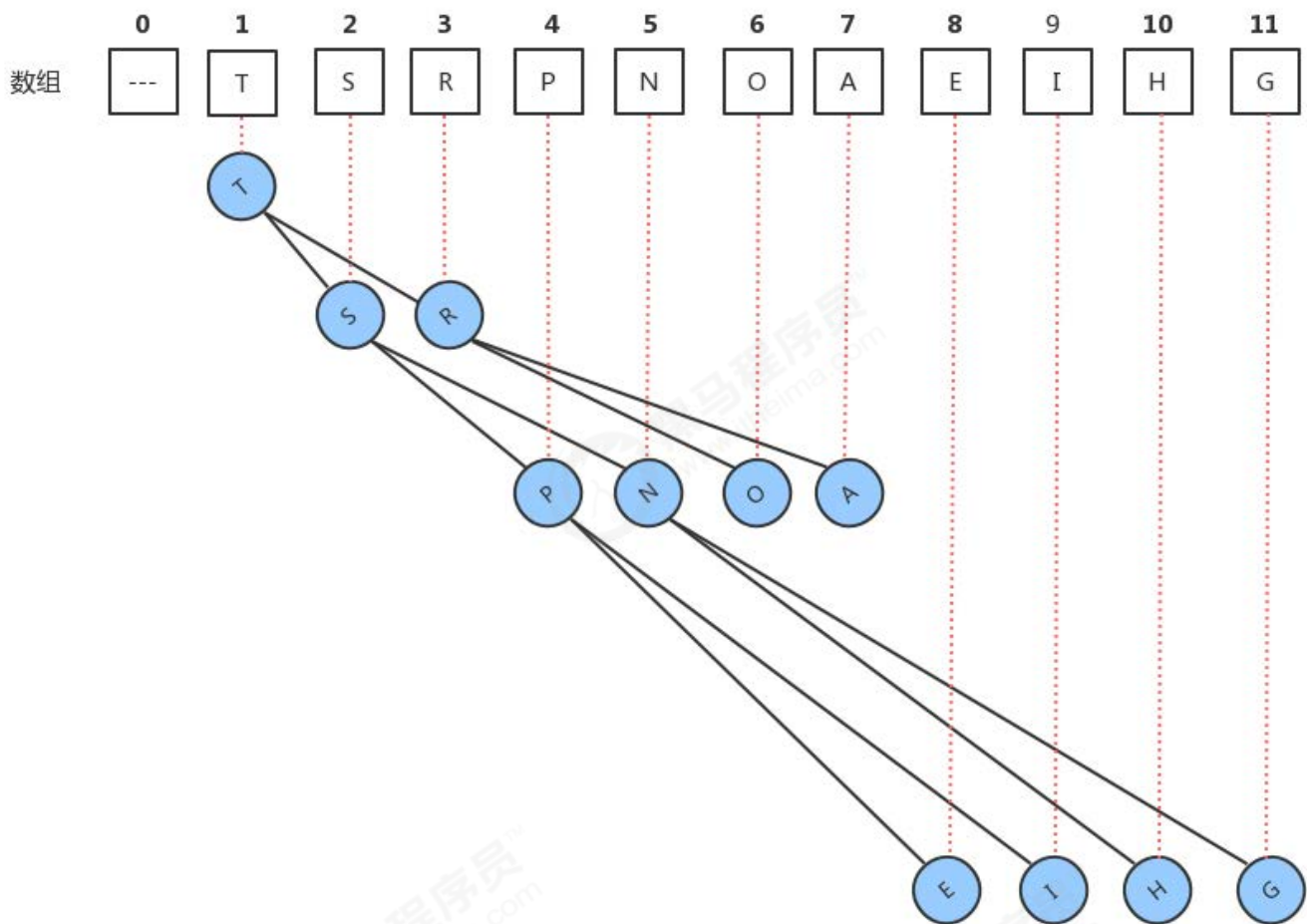
```

1.2 最小优先队列

最小优先队列实现起来也比较简单，我们同样也可以基于堆来完成最小优先队列。

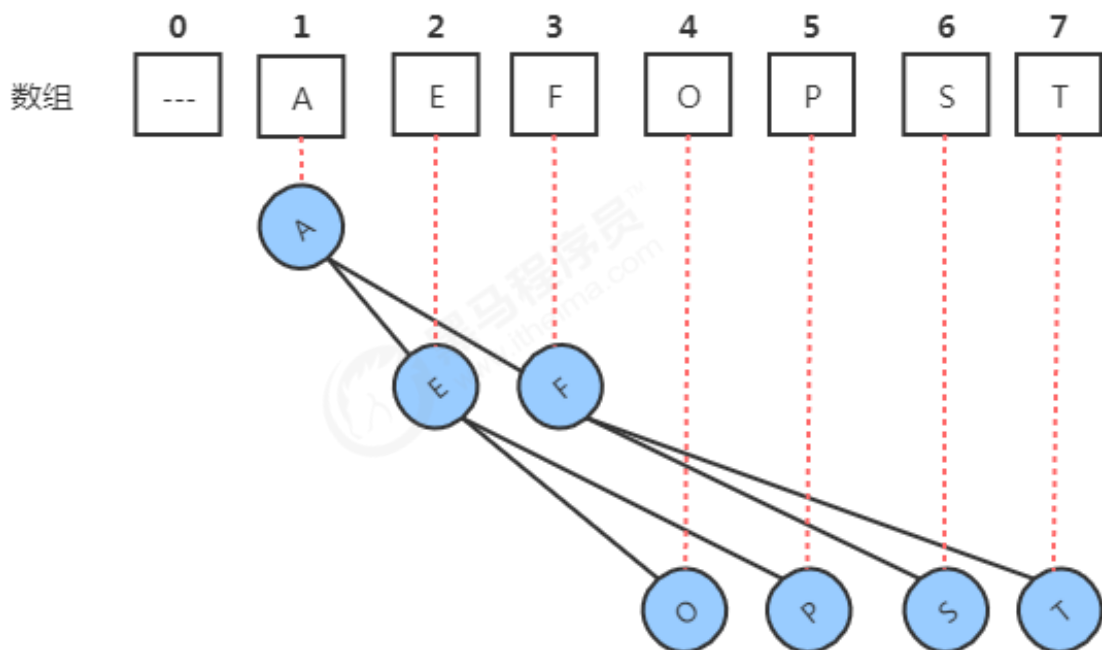
我们前面学习堆的时候，堆中存放数据元素的数组要满足都满足如下特性：

- 1.最大的元素放在数组的索引1处。
- 2.每个结点的数据总是大于等于它的两个子结点的数据。



其实我们之前实现的堆可以把它叫做最大堆，我们可以用相反的思想实现最小堆，让堆中存放数据元素的数组满足如下特性：

- 1.最小的元素放在数组的索引1处。
- 2.每个结点的数据总是小于等于它的两个子结点的数据。



这样我们就能快速的访问到堆中最小的数据。

1.2.1 最小优先队列API设计

类名	MinPriorityQueue<T>
构造方法	MinPriorityQueue(int capacity) : 创建容量为capacity的MinPriorityQueue对象
成员方法	1.private boolean less(int i,int j): 判断堆中索引i处的元素是否小于索引j处的元素 2.private void exch(int i,int j):交换堆中i索引和j索引处的值 3.public T delMin():删除队列中最小的元素,并返回这个最小元素 4.public void insert(T t) : 往队列中插入一个元素 5.private void swim(int k):使用上浮算法,使索引k处的元素能在堆中处于一个正确的位置 6.private void sink(int k):使用下沉算法,使索引k处的元素能在堆中处于一个正确的位置 7.public int size():获取队列中元素的个数 8.public boolean isEmpty():判断队列是否为空
成员变量	1.private T[] imtes : 用来存储元素的数组 2.private int N : 记录堆中元素的个数

1.2.2 最小优先队列代码实现

```
1 //最小优先队列代码
2 public class MinPriorityQueue<T extends Comparable<T>> {
3     //存储堆中的元素
4     private T[] items;
5     //记录堆中元素的个数
6     private int N;
7
8
9     public MinPriorityQueue(int capacity) {
10         items = (T[]) new Comparable[capacity+1];
11         N = 0;
12     }
13
14     //获取队列中元素的个数
15     public int size() {
16         return N;
17     }
18
19     //判断队列是否为空
20     public boolean isEmpty() {
21         return N == 0;
22     }
23
24     //判断堆中索引i处的元素是否小于索引j处的元素
25     private boolean less(int i, int j) {
26         return items[i].compareTo(items[j]) < 0;
27     }
28 }
```

```

29 //交换堆中i索引和j索引处的值
30 private void exch(int i, int j) {
31     T tmp = items[i];
32     items[i] = items[j];
33     items[j] = tmp;
34 }
35
36 //往堆中插入一个元素
37 public void insert(T t) {
38     items[++N] = t;
39     swim(N);
40 }
41
42 //删除堆中最小的元素,并返回这个最小元素
43 public T delMin() {
44     //索引1处的值是最小值
45     T min = items[1];
46     //交换索引1处和索引N处的值
47     exch(1, N);
48     //删除索引N处的值
49     items[N] = null;
50     //数据元素-1
51     N--;
52     //对索引1处的值做下沉,使堆重新有序
53     sink(1);
54     //返回被删除的值
55     return min;
56 }
57
58 //使用上浮算法,使索引k处的元素能在堆中处于一个正确的位置
59 private void swim(int k) {
60     //如果没有父结点,则不再上浮
61     while (k > 1) {
62         //如果当前结点比父结点小,则交换
63         if (less(k, k / 2)) {
64             exch(k, k / 2);
65         }
66         k = k / 2;
67     }
68 }
69
70 //使用下沉算法,使索引k处的元素能在堆中处于一个正确的位置
71 private void sink(int k) {
72     //如果没有子结点,则不再下沉
73     while (2 * k <= N) {
74         //找出子结点中的较小值的索引
75         int min = 2 * k;
76         if (2 * k + 1 <= N && less(2 * k + 1, 2 * k)) {
77             min = 2 * k + 1;
78         }
79         //如果当前结点小于子结点中的较小值,则结束循环
80         if (less(k, min)) {
81             break;

```

```

82     }
83     //当前结点大, 交换
84     exch(min, k);
85     k = min;
86 }
87 }
88 }
89
90 //测试代码
91 public class Test {
92     public static void main(String[] args) throws Exception {
93         String[] arr = {"S", "O", "R", "T", "E", "X", "A", "M", "P", "L", "E"};
94         MinPriorityQueue<String> minpq = new MinPriorityQueue<>(20);
95         for (String s : arr) {
96             minpq.insert(s);
97         }
98         System.out.println(minpq.size());
99         String del;
100         while(!minpq.isEmpty()){
101             del = minpq.delMin();
102             System.out.print(del+",");
103         }
104     }
105 }

```

1.3 索引优先队列

在之前实现的最大优先队列和最小优先队列，他们可以分别快速访问到队列中最大元素和最小元素，但是他们有一个缺点，就是没有办法通过索引访问已存在于优先队列中的对象，并更新它们。为了实现这个目的，在优先队列的基础上，学习一种新的数据结构，索引优先队列。接下来我们以最小索引优先队列列举。

1.3.1 索引优先队列实现思路

步骤一：

存储数据时，给每一个数据元素关联一个整数，例如insert(int k,T t),我们可以看做k是t关联的整数，那么我们的实现需要通过k这个值，快速获取到队列中t这个元素，此时有个k这个值需要具有唯一性。

最直观的想法就是我们可以用一个T[] items数组来保存数据元素，在insert(int k,T t)完成插入时，可以把k看做是items数组的索引，把t元素放到items数组的索引k处，这样我们再根据k获取元素t时就方便了，直接就可以拿到items[k]即可。

存放：{"S", "O", "R", "T", "E", "X", "A", "M", "P", "L", "E"}

T[] items : 用来保存数据元素

insert(0,"S")

insert(1,"O")

.....

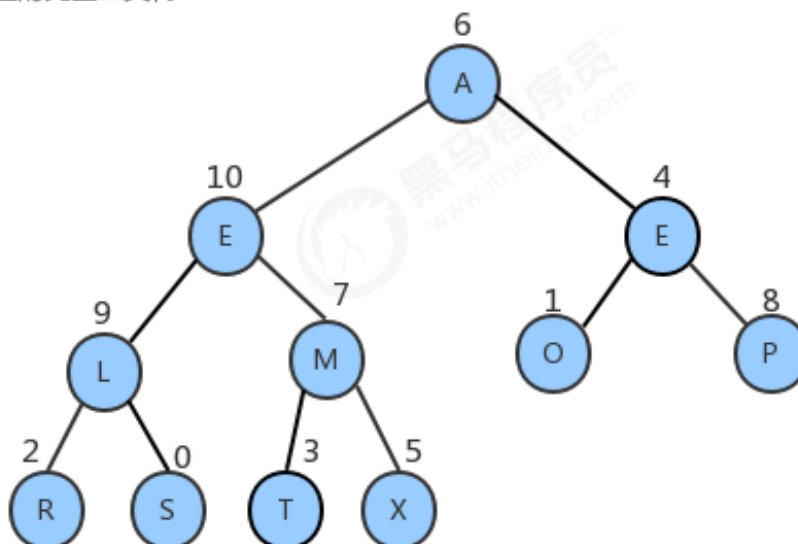
insert(9,"L")

步骤二：

步骤一完成后的结果，虽然我们给每个元素关联了一个整数，并且可以使用这个整数快速的获取到该元素，但是，items数组中的元素顺序是随机的，并不是堆有序的，所以，为了完成这个需求，我们可以增加一个数组int[]pq,来保存每个元素在items数组中的索引，pq数组需要堆有序，也就是说，pq[1]对应的数据元素items[pq[1]]要小于等于pq[2]和pq[3]对应的数据元素items[pq[2]]和items[pq[3]]。

	0	1	2	3	4	5	6	7	8	9	10	11
T[] items:	S	O	R	T	E	X	A	M	P	L	E	

最小堆有序后对应的完全二叉树：



int[] pq:存放元素在items里面的索引（堆调整后的顺序）：

	0	1	2	3	4	5	6	7	8	9	10	11
	--	6	10	4	9	7	1	8	2	0	3	5

堆有序： $items[pq[i]]$ 小于等于 $items[pq[2*i]]$ 和 $items[pq[2*i+1]]$

步骤三：

通过步骤二的分析，我们可以发现，其实我们通过上浮和下沉做堆调整的时候，其实调整的是pq数组。如果需要对items中的元素进行修改，比如让items[0]="H",那么很显然，我们需要对pq中的数据做堆调整，而且是调整pq[9]中元素的位置。但现在就会遇到一个问题，我们修改的是items数组中0索引处的值，如何才能快速的知道需要挑中pq[9]中元素的位置呢？

最直观的想法就是遍历pq数组，拿出每一个元素和0做比较，如果当前元素是0，那么调整该索引处的元素即可，但是效率很低。

我们可以另外增加一个数组，int[] qp,用来存储pq的逆序。例如：

在pq数组中：pq[1]=6;

那么在qp数组中，把6作为索引，1作为值，结果是：qp[6]=1;

	0	1	2	3	4	5	6	7	8	9	10	11
T[] items:	S	O	R	T	E	X	A	M	P	L	E	

int[] pq:存放元素在items里面的索引（堆调整后的顺序）：

	0	1	2	3	4	5	6	7	8	9	10	11
	--	6	10	4	9	7	1	8	2	0	3	5

int[] qp:存储pq数组的逆序：

	0	1	2	3	4	5	6	7	8	9	10	11
	9	6	8	10	3	11	1	5	7	4	2	--

当有了pq数组后，如果我们修改items[0]="H"，那么就可以先通过索引0，在qp数组中找到qp的索引：qp[0]=9，那么直接调整pq[9]即可。

1.3.2 索引优先队列API设计

类名	IndexMinPriorityQueue<
构造方法	IndexMinPriorityQueue(int capacity)：创建容量为capacity的IndexMinPriorityQueue对象
成员方法	1.private boolean less(int i,int j)：判断堆中索引i处的元素是否小于索引j处的元素 2.private void exch(int i,int j):交换堆中i索引和j索引处的值 3.public int delMin():删除队列中最小的元素,并返回该元素关联的索引 4.public void insert(int i,T t)：往队列中插入一个元素,并关联索引i 5.private void swim(int k):使用上浮算法，使索引k处的元素能在堆中处于一个正确的位置 6.private void sink(int k):使用下沉算法，使索引k处的元素能在堆中处于一个正确的位置 7.public int size():获取队列中元素的个数 8.public boolean isEmpty():判断队列是否为空 9.public boolean contains(int k):判断k对应的元素是否存在 10.public void changeItem(int i, T t):把与索引i关联的元素修改为t 11.public int minIndex():最小元素关联的索引 12.public void delete(int i):删除索引i关联的元素
成员变量	1.private T[] imtes：用来存储元素的数组 2.private int[] pq:保存每个元素在items数组中的索引，pq数组需要堆有序 3.private int [] qp:保存qp的逆序，pq的值作为索引，pq的索引作为值 4.private int N：记录堆中元素的个数

1.3.3 索引优先队列代码实现

```

1 //最小索引优先队列代码
2 package cn.itcast;
3
4 public class IndexMinPriorityQueue<T extends Comparable<T>> {

```

```

5 //存储堆中的元素
6 private T[] items;
7 //保存每个元素在items数组中的索引，pq数组需要堆有序
8 private int[] pq;
9 //保存pq的逆序，pq的值作为索引，pq的索引作为值
10 private int[] qp;
11 //记录堆中元素的个数
12 private int N;
13
14
15 public IndexMinPriorityQueue(int capacity) {
16     items = (T[]) new Comparable[capacity + 1];
17     pq = new int[capacity + 1];
18     qp = new int[capacity + 1];
19     N = 0;
20     for (int i = 0; i < qp.length; i++) {
21         //默认情况下，qp逆序中不保存任何索引
22         qp[i] = -1;
23     }
24 }
25
26 //获取队列中元素的个数
27 public int size() {
28     return N;
29 }
30
31 //判断队列是否为空
32 public boolean isEmpty() {
33     return N == 0;
34 }
35
36 //判断堆中索引i处的元素是否小于索引j处的元素
37 private boolean less(int i, int j) {
38     //先通过pq找出items中的索引，然后再找出items中的元素进行对比
39     return items[pq[i]].compareTo(items[pq[j]]) < 0;
40 }
41
42 //交换堆中i索引和j索引处的值
43 private void exch(int i, int j) {
44     //先交换pq数组中的值
45     int tmp = pq[i];
46     pq[i] = pq[j];
47     pq[j] = tmp;
48
49     //更新qp数组中的值
50     qp[pq[i]] = i;
51     qp[pq[j]] = j;
52 }
53
54 //判断k对应的元素是否存在
55 public boolean contains(int k) {
56     //默认情况下，qp的所有元素都为-1，如果某个位置插入了数据，则不为-1
57
58     return qp[k] != -1;

```

```

58     }
59
60     //最小元素关联的索引
61     public int minIndex() {
62         //pq的索引1处,存放的是最小元素在items中的索引
63         return pq[1];
64     }
65
66
67     //往队列中插入一个元素,并关联索引i
68     public void insert(int i, T t) {
69         //如果索引i处已经存在了元素,则不让插入
70         if (contains(i)) {
71             throw new RuntimeException("该索引已经存在");
72         }
73         //个数+1
74         N++;
75         //把元素存放到items数组中
76         items[i] = t;
77         //使用pq存放i这个索引
78         pq[N] = i;
79         //在qp的i索引处存放N
80         qp[i] = N;
81         //上浮items[pq[N]],让pq堆有序
82         swim(N);
83     }
84
85     //删除队列中最小的元素,并返回该元素关联的索引
86     public int delMin() {
87         //找到items中最小元素的索引
88         int minIndex = pq[1];
89         //交换pq中索引1处的值和N处的值
90         exch(1, N);
91         //删除qp中索引pq[N]处的值
92         qp[pq[N]] = -1;
93         //删除pq中索引N处的值
94         pq[N] = -1;
95         //删除items中的最小元素
96         items[minIndex] = null;
97         //元素数量-1
98         N--;
99         //对pq[1]做下沉,让堆有序
100        sink(1);
101        return minIndex;
102    }
103
104    //删除索引i关联的元素
105    public void delete(int i) {
106        //找出i在pq中的索引
107        int k = qp[i];
108        //把pq中索引k处的值和索引N处的值交换
109        exch(k, N);
110
        //删除qp中索引pq[N]处的值

```

```

111     qp[pq[N]] = -1;
112     //删除pq中索引N处的值
113     pq[N] = -1;
114     //删除items中索引i处的值
115     items[i] = null;
116     //元素数量-1
117     N--;
118     //对pq[k]做下沉，让堆有序
119     sink(k);
120     //对pq[k]做上浮，让堆有序
121     swim(k);
122 }
123
124 //把与索引i关联的元素修改为t
125 public void changeItem(int i, T t) {
126     //修改items数组中索引i处的值为t
127     items[i] = t;
128     //找到i在pq中的位置
129     int k = qp[i];
130     //对pq[k]做下沉，让堆有序
131     sink(k);
132     //对pq[k]做上浮，让堆有序
133     swim(k);
134 }
135
136
137 //使用上浮算法，使索引k处的元素能在堆中处于一个正确的位置
138 private void swim(int k) {
139     //如果已经到了根结点，则结束上浮
140     while (k > 1) {
141         //比较当前结点和父结点，如果当前结点比父结点小，则交换位置
142         if (less(k, k / 2)) {
143             exch(k, k / 2);
144         }
145         k = k / 2;
146     }
147 }
148
149 //使用下沉算法，使索引k处的元素能在堆中处于一个正确的位置
150 private void sink(int k) {
151     //如果当前结点已经没有子结点了，则结束下沉
152     while (2 * k <= N) {
153         //找出子结点中的较小值
154         int min = 2 * k;
155         if (2 * k + 1 <= N && less(2 * k + 1, 2 * k)) {
156             min = 2 * k + 1;
157         }
158         //如果当前结点的值比子结点中的较小值小，则结束下沉
159         if (less(k, min)) {
160             break;
161         }
162         exch(k, min);
163
164         k = min;

```

```
164     }
165 }
166
167 }
168
169
170 //测试代码
171 public class Test {
172     public static void main(String[] args) {
173         String[] arr = {"S", "O", "R", "T", "E", "X", "A", "M", "P", "L", "E"};
174         IndexMinPriorityQueue<String> indexMinPQ = new IndexMinPriorityQueue<>(20);
175         //插入
176         for (int i = 0; i < arr.length; i++) {
177             indexMinPQ.insert(i,arr[i]);
178         }
179
180         System.out.println(indexMinPQ.size());
181         //获取最小值的索引
182         System.out.println(indexMinPQ.minIndex());
183
184         //测试修改
185         indexMinPQ.changeItem(0,"Z");
186         int minIndex=-1;
187         while(!indexMinPQ.isEmpty()){
188             minIndex = indexMinPQ.delMin();
189             System.out.print(minIndex+",");
190         }
191     }
192 }
```