优先级队列(堆)

【本节目标】

1. 掌握堆的概念及实现
2. 掌握 PriorityQueue 的使用

1. 优先级队列

1.1 概念

前面介绍过队列，**队列是一种先进先出(FIFO)的数据结构**，但有些情况下，**操作的数据可能带有优先级，一般出队列时，可能需要优先级高的元素先出队列**，该中场景下，使用队列显然不合适，比如：在手机上玩游戏的时候，如果有来电，那么系统应该优先处理打进来的电话；初中那会班主任排座位时可能会让成绩好的同学先挑座位。

在这种情况下，**数据结构应该提供两个最基本的操作，一个是返回最高优先级对象，一个是添加新的对象**。这种数据结构就是**优先级队列(Priority Queue)。**

2. 优先级队列的模拟实现

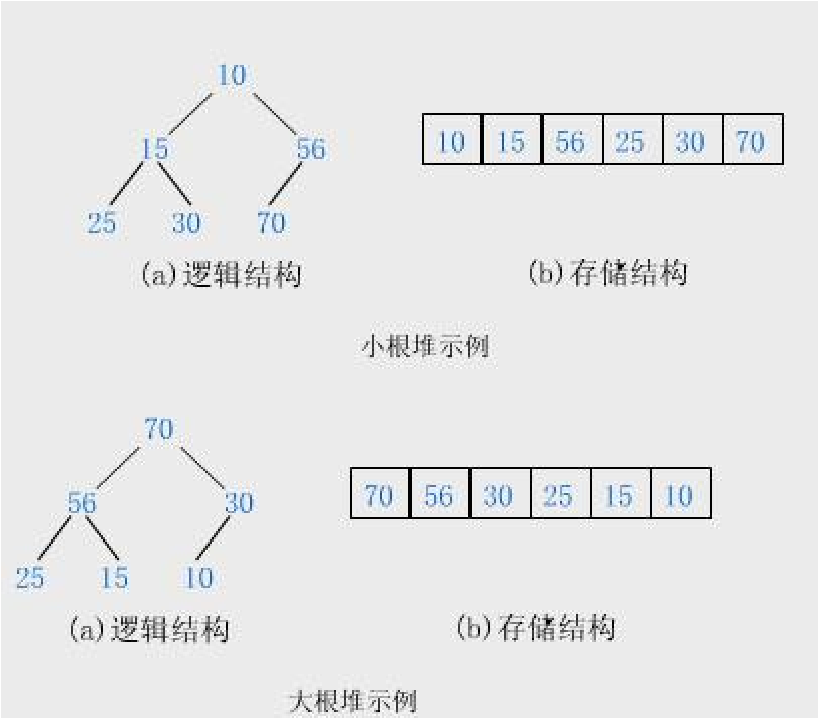
JDK1.8中的**PriorityQueue底层使用了堆这种数据结构**，而堆实际就是在完全二叉树的基础上进行了一些调整。

2.1 堆的概念

如果有一个**关键码的集合K = {k0，k1， k2，…，kn-1}**，把它的所有元素**按完全二叉树的顺序存储方式存储 在一个一维数组中**，并满足：**Ki <= K2i+1 且 Ki<= K2i+2** (Ki >= K2i+1 且 Ki >= K2i+2) i = 0，1，2…，则**称为 小堆**(或大堆)。将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。

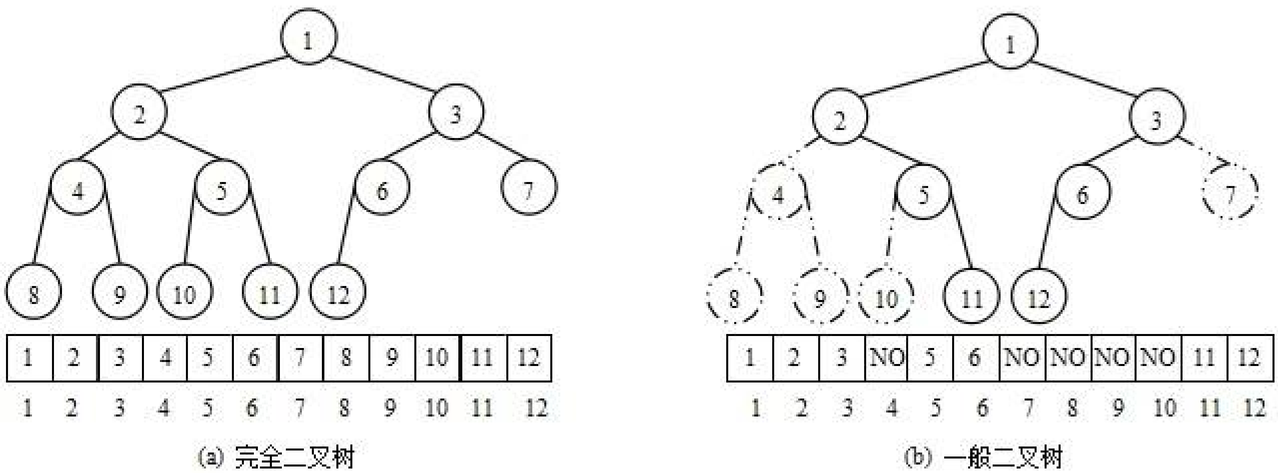
堆的性质：

- 堆中某个节点的值总是不大于或不小于其父节点的值；
- 堆总是一棵完全二叉树。



2.2 堆的存储方式

从堆的概念可知，**堆是一棵完全二叉树，因此可以层序的规则采用顺序的方式来高效存储**，



注意：对于**非完全二叉树，则不适合使用顺序方式进行存储**，因为为了能够还原二叉树，**空间中必须要存储空节点，就会导致空间利用率比较低。**

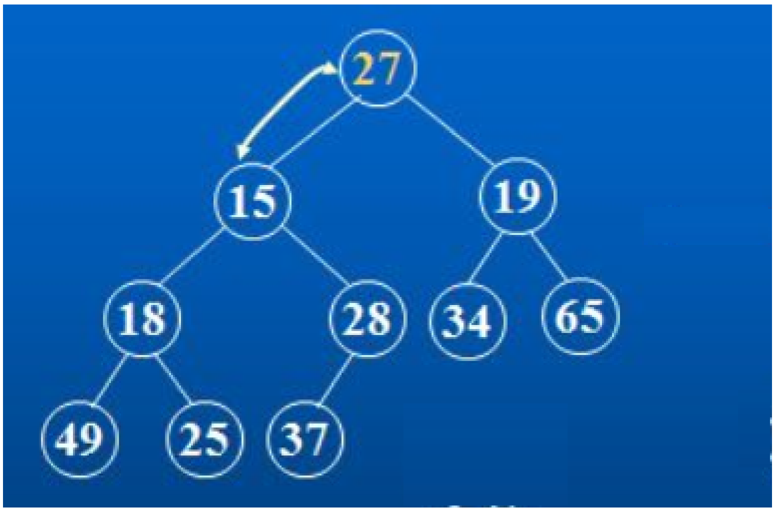
将元素存储到数组中后，可以根据二叉树章节的性质5对树进行还原。假设i为节点在数组中的下标，则有：

- 如果i为0，则i表示的节点为根节点，否则i节点的双亲节点为 (i - 1)/2
- 如果2 * i + 1 小于节点个数，则节点i的左孩子下标为2 * i + 1，否则没有左孩子
- 如果2 * i + 2 小于节点个数，则节点i的右孩子下标为2 * i + 2，否则没有右孩子

2.3 堆的创建

2.3.1 堆向下调整

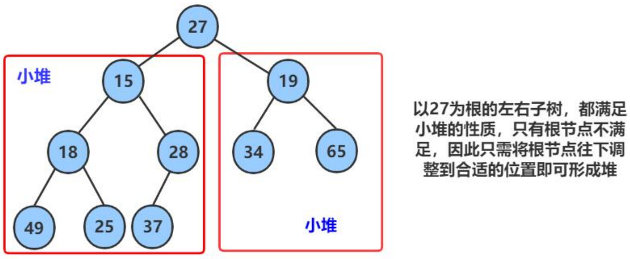
对于集合{ 27,15,19,18,28,34,65,49,25,37 }中的数据，如果将其创建成堆呢？

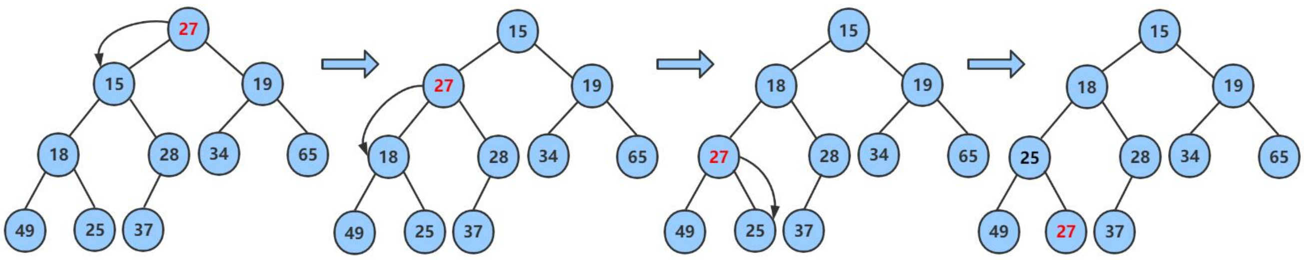


仔细观察上图后发现：**根节点的左右子树已经完全满足堆的性质，因此只需将根节点向下调整好即可。**

向下过程(以小堆为例)：

1. 让parent标记需要调整的节点，child标记parent的左孩子(注意：parent如果有孩子一定先是有左孩子)
2. 如果parent的左孩子存在，即:child < size， 进行以下操作，直到parent的左孩子不存在
    - parent右孩子是否存在，存在找到左右孩子中最小的孩子，让child进行标
    - 将parent与较小的孩子child比较，如果：
        - parent小于较小的孩子child，调整结束
        - 否则：交换parent与较小的孩子child，交换完成之后，parent中大的元素向下移动，可能导致子树不满足对的性质，因此需要继续向下调整，即parent = child；child = parent*2+1; 然后继续2。





```
public void shiftDown(int[] array, int parent) {
    // child先标记parent的左孩子，因为parent可能右左没有右
    int child = 2 * parent + 1;
    int size = array.length;

    while (child < size) {

        // 如果右孩子存在，找到左右孩子中较小的孩子,用child进行标记

        if(child+1 < size && array[child+1] < array[child]){
```

```
            child += 1;
        }

        // 如果双亲比其最小的孩子还小，说明该结构已经满足堆的特性了
        if (array[parent] <= array[child]) {
            break;
        }else{
            // 将双亲与较小的孩子交换
            int t = array[parent];
            array[parent] = array[child];
            array[child] = t;

            // parent中大的元素往下移动，可能会造成子树不满足堆的性质，因此需要继续向下调整
            parent = child;
            child = parent * 2 + 1;
        }
    }
}
```

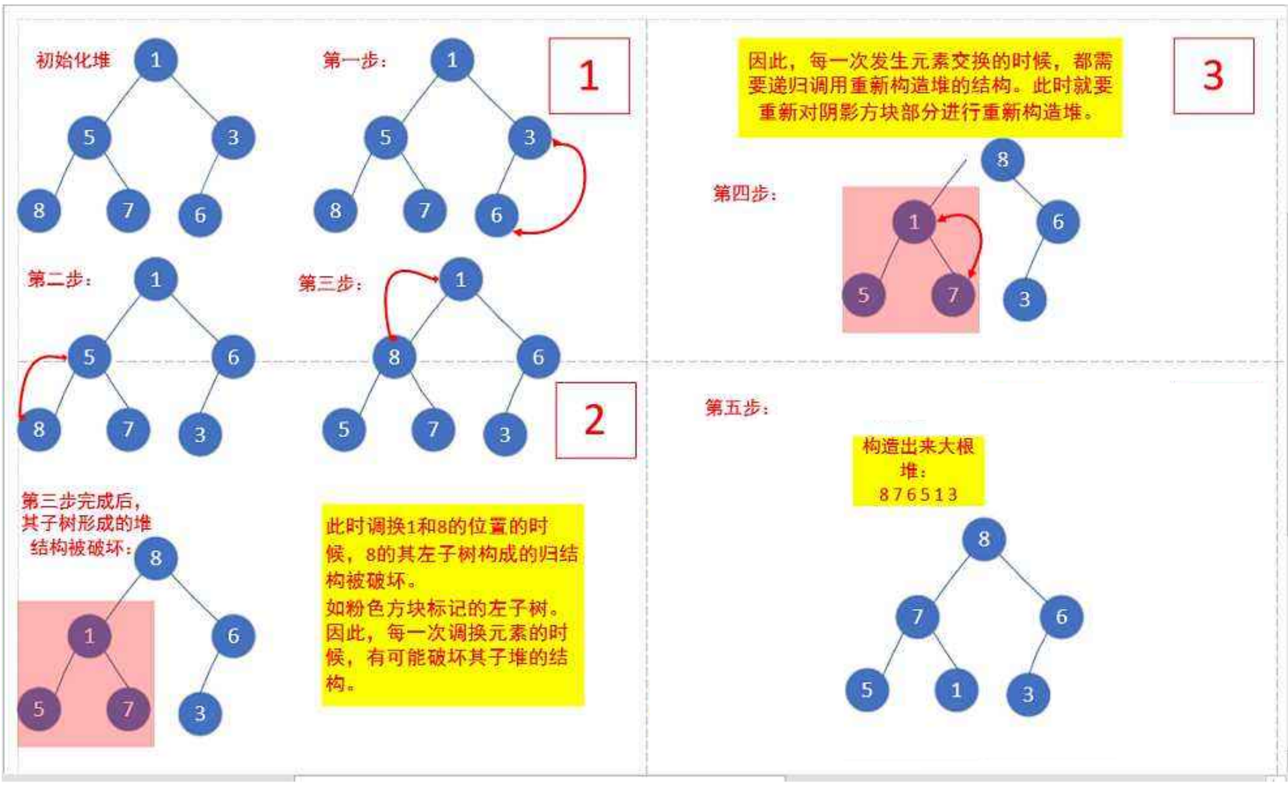
注意：在调整以parent为根的二叉树时，必须要满足parent的左子树和右子树已经是堆了才可以向下调整。

时间复杂度分析：

最坏的情况即图示的情况，**从根一路比较到叶子，比较的次数为完全二叉树的高度，即时间复杂度为** $O(log_2n)$

2.3.2 堆的创建

那对于普通的序列{ 1,5,3,8,7,6 }，即根节点的左右子树不满足堆的特性，又该如何调整呢？



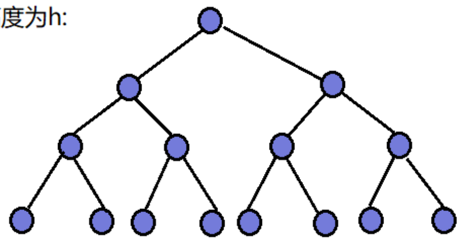
参考代码：

```java
public static void createHeap(int[] array) {
    // 找倒数第一个非叶子节点，从该节点位置开始往前一直到根节点，遇到一个节点，应用向下调整
    int root = ((array.length-2)>>1);
    for (; root >= 0; root--) {
        shiftDown(array, root);
    }
}
```

2.3.3 建堆的时间复杂度

因为堆是完全二叉树，而满二叉树也是完全二叉树，此处为了简化使用满二叉树来证明(时间复杂度本来看的就是近似值，多几个节点不影响最终结果)：

假设树的高度为h:



第1层，2^0个节点，需要向下移动h-1层

第2层，2^1个节点，需要向下移动h-2层

第3层，2^2个节点，需要向下移动h-3层

第4层，2^3个节点，需要向下移动h-4层

......

第h-1层，2^{h-2}个节点，需要向下移动1层

则需要移动节点总的移动步数为：

$$T(n) = 2^0 * (h-1) + 2^1 * (h-2) + 2^2 * (h-3) + 2^3 * (h-4) + \ldots + 2^{h-3} * 2 + 2^{h-2} * 1 \quad ①$$

$$2 * T(n) = 2^1 * (h-1) + 2^2 * (h-2) + 2^3 * (h-3) + 2^4 * (h-4) + \ldots + 2^{h-2} * 2 + 2^{h-1} * 1 \quad ②$$

②−① 错位相减：

$$T(n) = 1 - h + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{h-2} + 2^{h-1}$$

$$T(n) = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + \ldots + 2^{h-2} + 2^{h-1} - h$$

$$T(n) = 2^h - 1 - h$$

$$n = 2^h - 1 \qquad h = \log_2(n+1)$$

$$T(n) = n - \log_2(n+1) \approx n$$

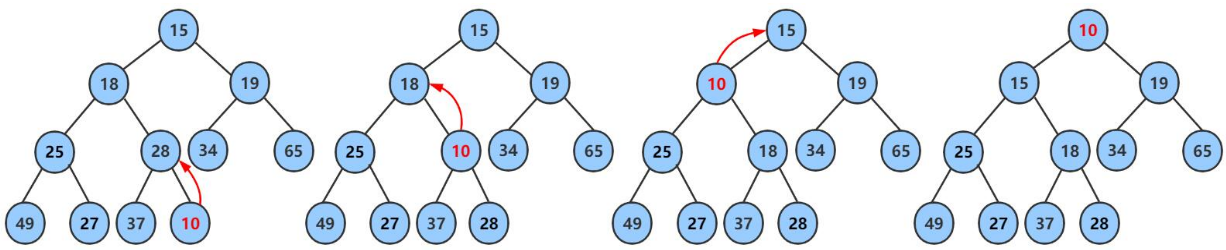
因此：**建堆的时间复杂度为O(N)。**

2.4 堆的插入与删除

2.4.1 堆的插入

堆的插入总共需要两个步骤：

1. 先将元素放入到底层空间中(注意：空间不够时需要扩容)
2. 将最后新插入的节点向上调整，直到满足堆的性质



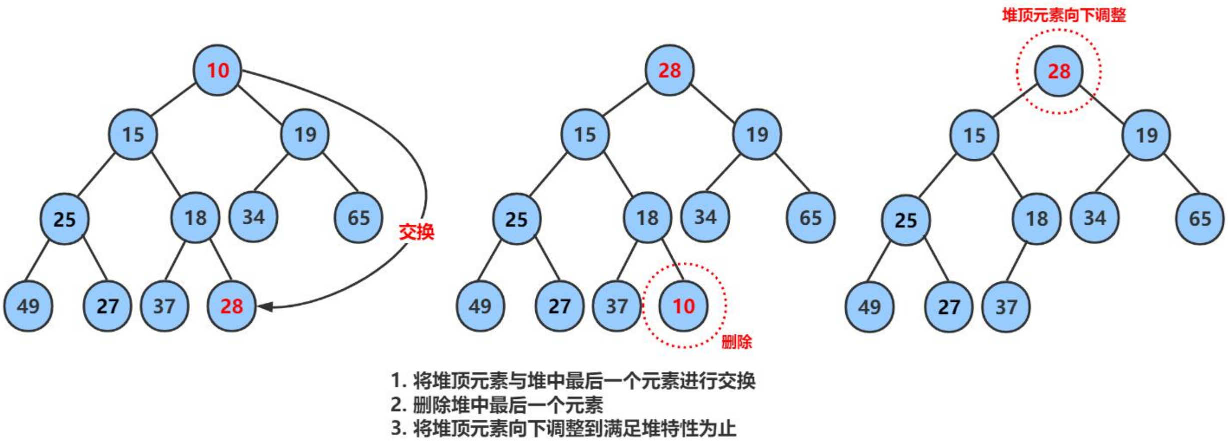
1. 先将元素插入到堆的末尾，即最后一个孩子之后
2. 插入之后如果堆的性质遭到破坏，将新插入节点顺着其双亲往上调整到合适位置即可

```
public void shiftUp(int child) {
  // 找到child的双亲
   int parent = (child - 1) / 2;

  while (child > 0) {
    // 如果双亲比孩子大，parent满足堆的性质，调整结束
    if (array[parent] > array[child]) {
      break;
    }
    else{
      // 将双亲与孩子节点进行交换
      int t = array[parent];
      array[parent] = array[child];
      array[child] = t;

      // 小的元素向下移动，可能到值子树不满足对的性质，因此需要继续向上调增
      child = parent;
      parent = (child - 1) / 1;
    }
  }
}
```

2.4.2 堆的删除

注意：堆的删除一定删除的是堆顶元素。具体如下：

1. 将堆顶元素对堆中最后一个元素交换
2. 将堆中有效数据个数减少一个
3. 对堆顶元素进行向下调整



2.5 用堆模拟实现优先级队列

```
public class MyPriorityQueue {
  // 演示作用，不再考虑扩容部分的代码
  private int[] array = new int[100];
  private int size = 0;

  public void offer(int e) {
    array[size++] = e;
```

```
        shiftUp(size - 1);
    }

    public int poll() {
        int oldValue = array[0];
        array[0] = array[--size];
        shiftDown(0);
        return oldValue;
    }

    public int peek() {
        return array[0];
    }
}
```

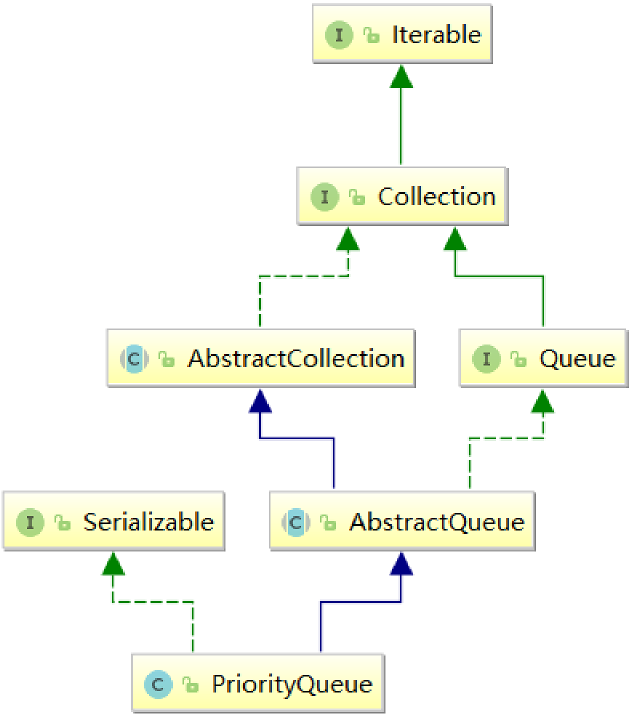
常见习题：

```
1.下列关键字序列为堆的是:()
  A: 100,60,70,50,32,65    B: 60,70,65,50,32,100    C: 65,100,70,32,50,60
  D: 70,65,100,32,50,60    E: 32,50,100,70,65,60    F: 50,100,70,65,60,32

2.已知小根堆为8,15,10,21,34,16,12，删除关键字8之后需重建堆，在此过程中，关键字之间的比较次数是()
  A: 1     B: 2     C: 3    D: 4


4.最小堆[0,3,2,5,7,4,6,8],在删除堆顶元素0之后，其结果是()
  A: [3，2，5，7，4，6，8]  B: [2，3，5，7，4，6，8]
  C: [2，3，4，5，7，8，6]  D: [2，3，4，5，6，7，8]


[参考答案]
  1.A    2.C       4.C
```

3.常用接口介绍

3.1 PriorityQueue的特性

Java集合框架中提供了**PriorityQueue**和**PriorityBlockingQueue**两种类型的优先级队列，**PriorityQueue是线程不安全的，PriorityBlockingQueue是线程安全的**，本文主要介绍PriorityQueue。



关于PriorityQueue的使用要注意：

1. 使用时必须导入PriorityQueue所在的包，即：

   ```
   import java.util.PriorityQueue;
   ```

2. PriorityQueue中放置的**元素必须要能够比较大小，不能插入无法比较大小的对象，否则会抛出ClassCastException异常**
3. 不能**插入null对象，否则会抛出NullPointerException**
4. **没有容量限制，可以插入任意多个元素，其内部可以自动扩容**
5. **插入和删除元素的时间复杂度为**$O(log_2 N)$
6. **PriorityQueue底层使用了堆数据结构**
7. **PriorityQueue默认情况下是小堆**---即每次获取到的元素都是最小的元素

3.2 PriorityQueue常用接口介绍

1. **优先级队列的构造**

   此处只是列出了PriorityQueue中常见的几种构造方式，其他的学生们可以参考帮助文档。

构造器	功能介绍
PriorityQueue()	创建一个空的优先级队列，默认容量是11
PriorityQueue(int initialCapacity)	创建一个初始容量为initialCapacity的优先级队列，注意：initialCapacity不能小于1，否则会抛IllegalArgumentException异常
PriorityQueue(Collection<? extends E> c)	用一个集合来创建优先级队列

```
static void TestPriorityQueue(){
      // 创建一个空的优先级队列，底层默认容量是11
      PriorityQueue<Integer> q1 = new PriorityQueue<>();

      // 创建一个空的优先级队列，底层的容量为initialCapacity
      PriorityQueue<Integer> q2 = new PriorityQueue<>(100);

      ArrayList<Integer> list = new ArrayList<>();
      list.add(4);
      list.add(3);
      list.add(2);
      list.add(1);

      // 用ArrayList对象来构造一个优先级队列的对象
      // q3中已经包含了三个元素
      PriorityQueue<Integer> q3 = new PriorityQueue<>(list);
      System.out.println(q3.size());
      System.out.println(q3.peek());
   }
```

注意：默认情况下，PriorityQueue队列是小堆，如果需要大堆需要用户提供比较器

```
// 用户自己定义的比较器：直接实现Comparator接口，然后重写该接口中的compare方法即可
class IntCmp implements Comparator<Integer>{
   @Override
   public int compare(Integer o1, Integer o2) {
      return o2-o1;
   }
}

public class TestPriorityQueue {
   public static void main(String[] args) {
      PriorityQueue<Integer> p = new PriorityQueue<>(new IntCmp());
      p.offer(4);
      p.offer(3);
      p.offer(2);
      p.offer(1);

      p.offer(5);
```

```
        System.out.println(p.peek());
    }
}
```

此时创建出来的就是一个大堆。

2. **插入/删除/获取优先级最高的元素**

函数名	功能介绍
boolean offer(E e)	插入元素e，插入成功返回true，如果e对象为空，抛出NullPointerException异常，时间复杂度$O(log_2 N)$，注意：空间不够时候会进行扩容
E peek()	获取优先级最高的元素，如果优先级队列为空，返回null
E poll()	移除优先级最高的元素并返回，如果优先级队列为空，返回null
int size()	获取有效元素的个数
void clear()	清空
boolean isEmpty()	检测优先级队列是否为空，空返回true

```
static void TestPriorityQueue2(){
    int[] arr = {4,1,9,2,8,0,7,3,6,5};

    // 一般在创建优先级队列对象时，如果知道元素个数，建议就直接将底层容量给好
    // 否则在插入时需要不多的扩容
    // 扩容机制：开辟更大的空间，拷贝元素，这样效率会比较低
    PriorityQueue<Integer> q = new PriorityQueue<>(arr.length);
    for (int e: arr) {
        q.offer(e);
    }

    System.out.println(q.size());   // 打印优先级队列中有效元素个数
    System.out.println(q.peek());   // 获取优先级最高的元素

    // 从优先级队列中删除两个元素之和，再次获取优先级最高的元素
    q.poll();
    q.poll();
    System.out.println(q.size());   // 打印优先级队列中有效元素个数
    System.out.println(q.peek());   // 获取优先级最高的元素

    q.offer(0);
    System.out.println(q.peek());   // 获取优先级最高的元素

    // 将优先级队列中的有效元素删除掉，检测其是否为空
    q.clear();
    if(q.isEmpty()){
        System.out.println("优先级队列已经为空!!!");
```

```
    }
    else{
      System.out.println("优先级队列不为空");
    }
  }
```

注意：以下是JDK 1.8中，PriorityQueue的扩容方式：

```
 private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

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);
}

private static int hugeCapacity(int minCapacity) {
    if (minCapacity < 0) // overflow
        throw new OutOfMemoryError();
    return (minCapacity > MAX_ARRAY_SIZE) ?
        Integer.MAX_VALUE :
        MAX_ARRAY_SIZE;
}
```

优先级队列的扩容说明：

- 如果容量小于64时，是按照oldCapacity的2倍方式扩容的
- 如果容量大于等于64，是按照oldCapacity的1.5倍方式扩容的
- 如果容量超过MAX_ARRAY_SIZE，按照MAX_ARRAY_SIZE来进行扩容

3.3 oj练习

top-k问题：最大或者最小的前k个数据。比如：世界前500强公司

[top-k问题：最小的K个数]

```
class Solution {
    public int[] smallestK(int[] arr, int k) {
        // 参数检测
        if(null == arr || k <= 0)
            return new int[0];

        PriorityQueue<Integer> q = new PriorityQueue<>(arr.length);

        // 将数组中的元素依次放到堆中

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

```java
            q.offer(arr[i]);
        }

        // 将优先级队列的前k个元素放到数组中
        int[] ret = new int[k];
        for(int i = 0; i < k; ++i){
            ret[i] = q.poll();
        }

        return ret;
    }
}
```

该解法只是PriorityQueue的简单使用，并不是topK最好的做法，那topk该如何实现？下面介绍：

4. 堆的应用

4.1 PriorityQueue的实现

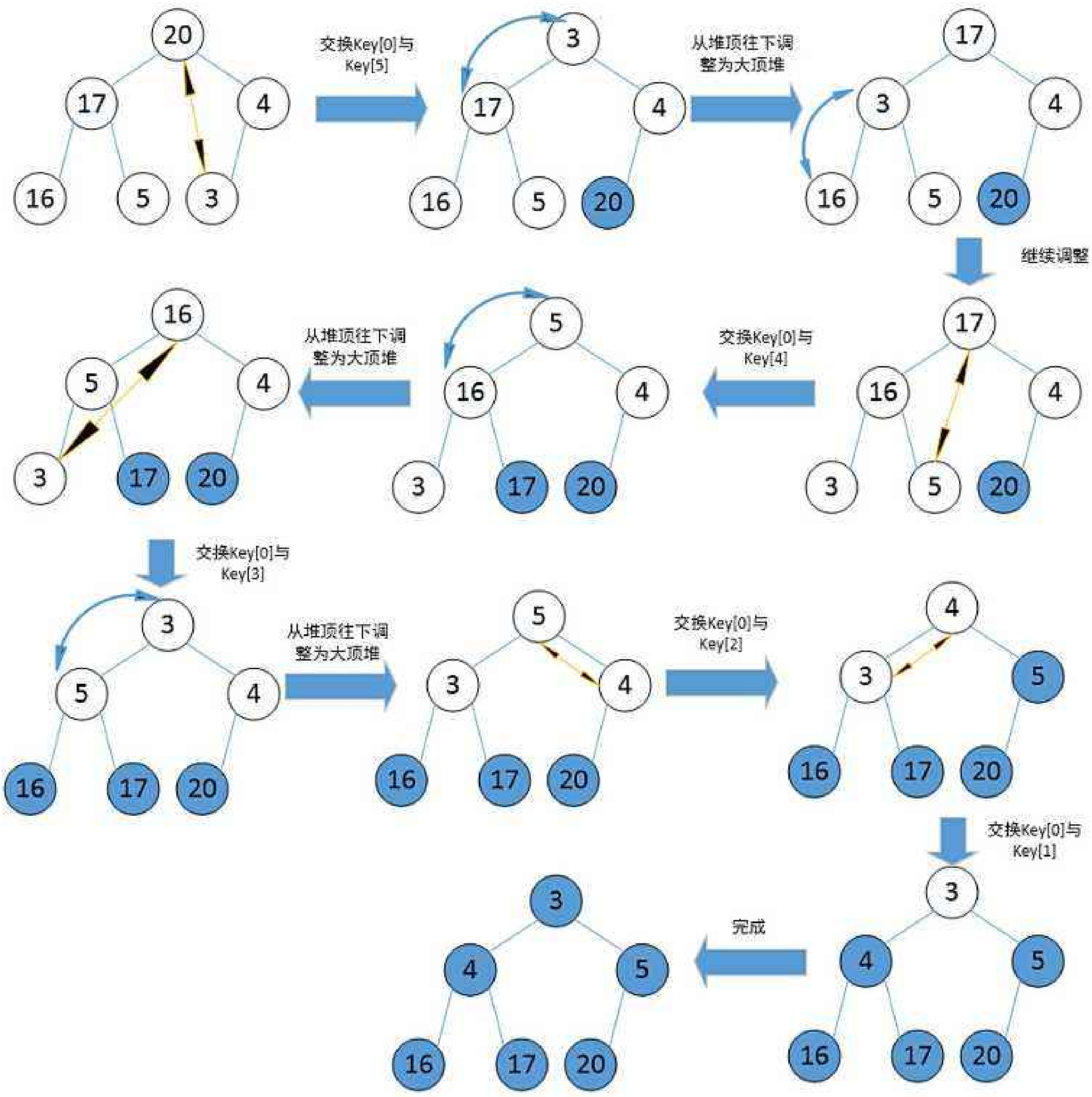
用堆作为底层结构**封装优先级队列**

4.2 堆排序

堆排序即利用堆的思想来进行排序，总共分为两个步骤：

1. **建堆**
    - 升序：建大堆
    - 降序：建小堆
2. **利用堆删除思想来进行排序**

建堆和堆删除中都用到了向下调整，因此掌握了向下调整，就可以完成堆排序。



常见习题：

1.一组记录排序码为(5 11 7 2 3 17),则利用堆排序方法建立的初始堆为()
A: (11 5 7 2 3 17)　B: (11 5 7 2 17 3)　C: (17 11 7 2 3 5)
D: (17 11 7 5 3 2)　E: (17 7 11 3 5 2)　F: (17 7 11 3 2 5)

答案：C

4.3 Top-k问题

TOP-K问题：即求数据集合中前K个最大的元素或者最小的元素，一般情况下数据量都比较大。

比如：专业前10名、世界500强、富豪榜、游戏中前100的活跃玩家等。

对于Top-K问题，能想到的最简单直接的方式就是排序，但是：如果数据量非常大，排序就不太可取了(可能数据都不能一下子全部加载到内存中)。最佳的方式就是用堆来解决，基本思路如下：

1. **用数据集合中前K个元素来建堆**
    - 前k个最大的元素，则建小堆
    - 前k个最小的元素，则建大堆
2. **用剩余的N-K个元素依次与堆顶元素来比较，不满足则替换堆顶元素**

将剩余N-K个元素依次与堆顶元素比完之后，堆中剩余的K个元素就是所求的前K个最小或者最大的元素。

【具体代码实现，见下个课件】