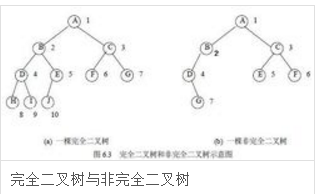
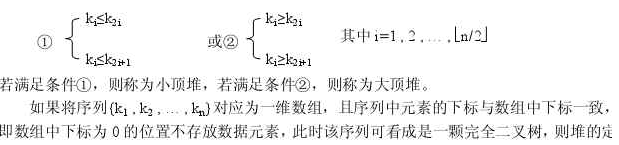
最大堆

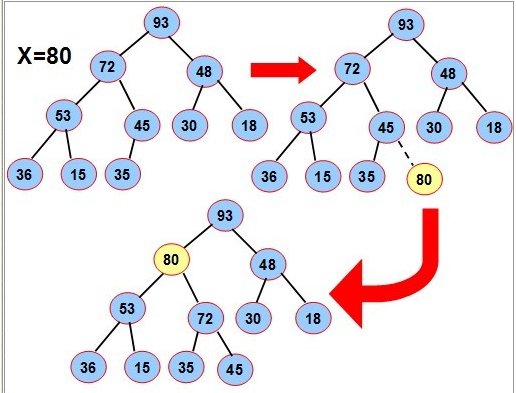
堆有最大堆和最小堆之分，最大堆就是每个节点的值都>=其左右孩子（如果有的话）值的完全二叉树。最小堆便是每个节点的值都<=其左右孩子值的完全二叉树。

一棵二叉树至多只有最下面的一层上的结点的度数可以小于2，并且最下层上的结点都集中在该层最左边的若干位置上，则此二叉树成为完全二叉树。

  
  
  设有n个元素的序列{k1,k2,...,kn},当且仅当满足下列关系时,称之为堆。



堆的三种基本操作(以下以最大堆为例)：   
⑴最大堆的插入     
  
    由于需要维持完全二叉树的形态，需要先将要插入的**结点x放在最底层的最右边**，插入后满足完全二叉树的特点；   
  然后把x依次向上调整到合适位置满足堆的性质,例如下图中插入80,先将80放在最后,然后两次上浮到合适位置.   
  时间：O(logn)。  “结点上浮”



程序：

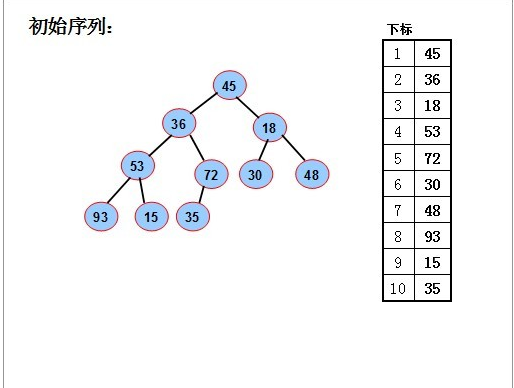
1. //向最大堆中插入元素, heap:存放堆元素的数组
2. **public** **static** **void** insert(List<Integer> heap, **int** value) {
3. //在数组的尾部添加
4. **if**(heap.size()==0)
5. heap.add(0);//数组下标为0的位置不放元素
6. heap.add(value);
7. //开始上升操作
8. // heapUp2(heap, heap.size() - 1);
9. heapUp(heap, heap.size() - 1);
11. }
13. //上升，让插入的数和父节点的数值比较，当大于父节点的时候就和父节点的值相交换
14. **public** **static** **void** heapUp(List<Integer> heap, **int** index) {
16. //注意由于数值是从下标为1开始，当index = 1的时候，已经是根节点了
17. **if** (index > 1) {
18. //求出父亲的节点
19. **int** parent = index / 2;
21. //获取相应位置的数值
22. **int** parentValue = (Integer) heap.get(parent);
23. **int** indexValue = (Integer) heap.get(index);
24. //如果父亲节点比index的数值小，就交换二者的数值
25. **if** (parentValue < indexValue) {
26. //交换数值
27. swap(heap, parent, index);
28. //递归调用
29. heapUp(heap, parent);
30. }
32. }
33. }

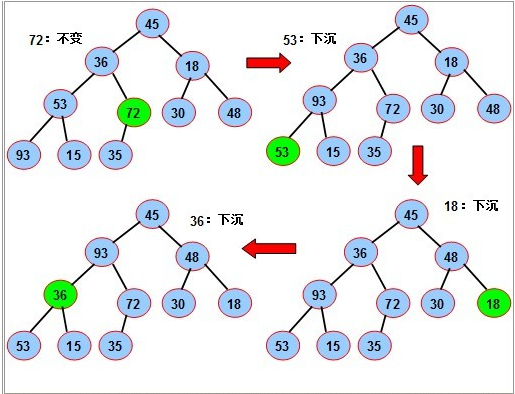
⑵删除   
   操作原理是：当删除节点的数值时，原来的位置就会出现一个孔,填充这个孔的方法就是，   
把**最后的叶子的值赋给该孔并下调到合适位置，最后把该叶子删除。**   
    
如图中要删除72,先用堆中最后一个元素来35替换72,再将35下沉到合适位置,最后将叶子节点删除。   
　　“结点下沉”

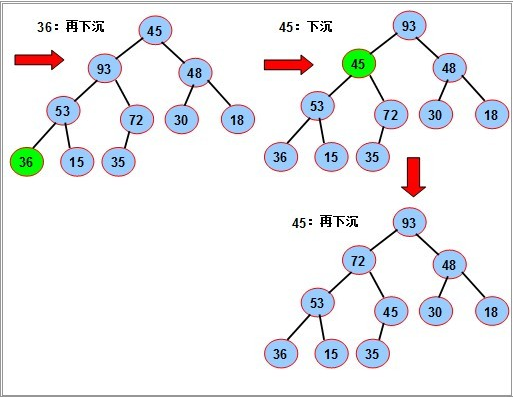
1. 程序:
2. /\*\*
3. \* 删除堆中位置是index处的节点
4. \* 操作原理是：当删除节点的数值时，原来的位置就会出现一个孔
5. \* 填充这个孔的方法就是，把最后的叶子的值赋给该孔，最后把该叶子删除
6. \* @param heap
7. \*/
8. **public** **static** **void** delete(List<Integer> heap,**int** index) {
9. //把最后的一个叶子的数值赋值给index位置
10. heap.set(index, heap.get(heap.size() - 1));
11. //下沉操作
12. //heapDown2(heap, index);
13. heapDown(heap, index);
14. //把最后一个位置的数字删除
15. heap.remove(heap.size() - 1);
16. }

19. /\*\*
20. \* 递归实现
21. \* 删除堆中一个数据的时候，根据堆的性质，应该把相应的位置下移，才能保持住堆性质不变
22. \* @param heap 保持堆元素的数组
23. \* @param index 被删除的那个节点的位置
24. \*/
25. **public** **static** **void** heapDown(List<Integer> heap, **int** index) {
26. //因为第一个位置存储的是空值，不在考虑之内
27. **int** n = heap.size() - 2;
29. //记录最大的那个儿子节点的位置
30. **int** child = -1;
32. //2\*index>n说明该节点没有左右儿子节点了，那么就返回
33. **if** (2 \* index > n) {
34. **return**;
35. } //如果左右儿子都存在
36. **else** **if** (2 \* index < n) {
38. //定义左儿子节点
39. child = 2 \* index;
40. //如果左儿子小于右儿子的数值，取右儿子的下标
41. **if** ((Integer) heap.get(child) < (Integer) heap.get(child + 1)) {
42. child++;
43. }
45. }//如果只有一个儿子（左儿子节点）
46. **else** **if** (2 \* index == n) {
47. child = 2 \* index;
48. }
50. **if** ((Integer) heap.get(child) > (Integer) heap.get(index)) {
51. //交换堆中的child，和index位置的值
52. swap(heap, child, index);
54. //完成交换后递归调用，继续下降
55. heapDown(heap, child);
56. }
57. }

⑶初始化   
方法1：插入法：   
  从空堆开始，依次插入每一个结点，直到所有的结点全部插入到堆为止。   
  时间：O(n\*log(n))   
  方法2：调整法：   
    序列对应一个完全二叉树；从最后一个分支结点（n div 2）开始，到根（1）为止，依次对每个分支结点进行调整（下沉），以便形成以每个分支结点为根的堆，当最后对树根结点进行调整后，整个树就变成了一个堆。   
  时间：O(n)   
对如图的序列,要使其成为堆,我们从最后一个分支结点(10/2),其值为72开始,依次对每个分支节点53,18,36 45进行调整(下沉).







1. 程序:
3. /\*根据树的性质建堆，树节点前一半一定是分支节点，即有孩子的，所以我们从这里开始调整出初始堆\*/
4. **public** **static** **void** adjust(List<Integer> heap){
5. **for** (**int** i = heap.size() / 2; i > 0; i--)
6. adjust(heap,i, heap.size()-1);

9. System.out.println("=================================================");
10. System.out.println("调整后的初始堆：");
11. print(heap);
12. }
14. /\*\*
15. \* 调整堆，使其满足堆得定义
16. \* @param i
17. \* @param n
18. \*/
19. **public** **static** **void** adjust(List<Integer> heap,**int** i, **int** n) {
21. **int** child;
22. **for** (; i <= n / 2; ) {
23. child = i \* 2;
24. **if**(child+1<=n&&heap.get(child)<heap.get(child+1))
25. child+=1;/\*使child指向值较大的孩子\*/
26. **if**(heap.get(i)< heap.get(child)){
27. swap(heap,i, child);
28. /\*交换后，以child为根的子树不一定满足堆定义，所以从child处开始调整\*/
29. i = child;
31. }  **else** **break**;
32. }
33. }

最大堆排序

1. //对一个最大堆heap排序
2. **public** **static** **void** heapSort(List<Integer> heap) {
4. **for** (**int** i = heap.size()-1; i > 0; i--) {
5. /\*把根节点跟最后一个元素交换位置，调整剩下的n-1个节点，即可排好序\*/
6. swap(heap,1, i);
7. adjust(heap,1, i - 1);
8. }
9. }

**堆排序是一种选择排序，其时间复杂度为O(nlogn)。**

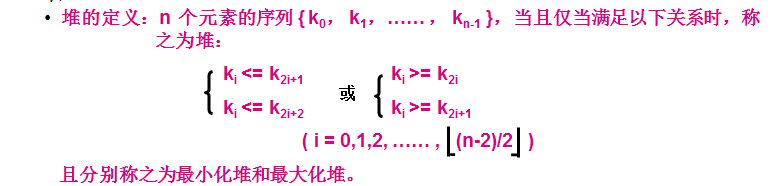
**堆的定义**

　　n个元素的序列{k1，k2，…,kn}当且仅当满足下列关系之一时，称之为堆。

　　情形1：ki<= k2i且ki<= k2i+1（**最小化堆**或**小顶堆**）

　　情形2：ki>= k2i且ki>= k2i+1 （**最大化堆**或**大顶堆**）

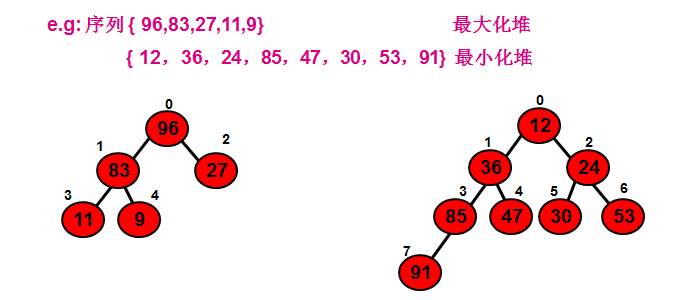
　　其中i=1,2,…,n/2向下取整;



　　若将和此序列对应的一维数组（即以一维数组作此序列的存储结构）看成是一个**完全二叉树**，则堆的含义表明，完全二叉树中所有非终端结点的值均不大于（或不小于）其左、右孩子结点的值。

　　由此，若序列{k1，k2，…,kn}是堆，**则堆顶元素（或完全二叉树的根）必为序列中n个元素的最小值（或最大值）。**

　　例如，下列两个序列为堆，对应的完全二叉树如图：



　　若在输出堆顶的最小值之后，使得剩余n-1个元素的序列重又建成一个堆，则得到n个元素的次小值。如此反复执行，便能得到一个有序序列，这个过程称之为**堆排序**。

　　堆排序（Heap Sort）只需要一个记录元素大小的辅助空间（供交换用），每个待排序的记录仅占有一个存储空间。

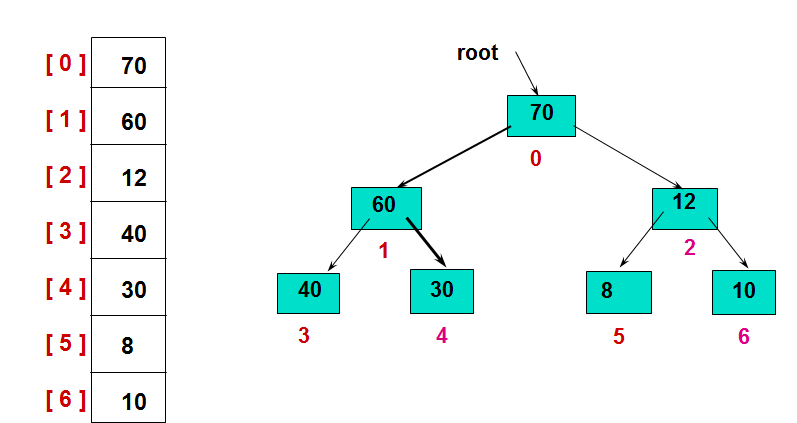
## **堆的存储**

　　一般用数组来表示堆，若根结点存在序号0处， i结点的父结点下标就为(i-1)/2。i结点的左右子结点下标分别为2\*i+1和2\*i+2。

　　（注：如果根结点是从1开始，则左右孩子结点分别是2i和2i+1。）

　　如第0个结点左右子结点下标分别为1和2。

　　如最大化堆如下：



　　左图为其存储结构，右图为其逻辑结构。

## **堆排序的实现**

　　实现堆排序需要解决两个问题：

　　　　1.如何由一个无序序列建成一个堆？

　　　　2.如何在输出堆顶元素之后，调整剩余元素成为一个新的堆？

　　先考虑第二个问题，一般在输出堆顶元素之后，视为将这个元素排除，然后用表中最后一个元素填补它的位置，自上向下进行调整：首先将堆顶元素和它的左右子树的根结点进行比较，把最小的元素交换到堆顶；然后顺着被破坏的路径一路调整下去，直至叶子结点，就得到新的堆。

　　我们称这个自堆顶至叶子的调整过程为“筛选”。

　　从无序序列建立堆的过程就是一个反复“筛选”的过程。

### **构造初始堆**

　　初始化堆的时候是对所有的非叶子结点进行筛选。

　　最后一个非终端元素的下标是[n/2]向下取整，所以筛选只需要从第[n/2]向下取整个元素开始，从后往前进行调整。

　　比如，给定一个数组，首先根据该数组元素构造一个完全二叉树。

　　然后从最后一个非叶子结点开始，每次都是从父结点、左孩子、右孩子中进行比较交换，交换可能会引起孩子结点不满足堆的性质，所以每次交换之后需要重新对被交换的孩子结点进行调整。

### **进行堆排序**

　　有了初始堆之后就可以进行排序了。

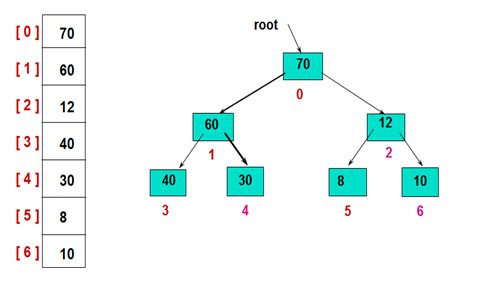
　　堆排序是一种选择排序。建立的初始堆为初始的无序区。

　　排序开始，首先输出堆顶元素（因为它是最值），将堆顶元素和最后一个元素交换，这样，第n个位置（即最后一个位置）作为有序区，前n-1个位置仍是无序区，对无序区进行调整，得到堆之后，再交换堆顶和最后一个元素，这样有序区长度变为2。。。

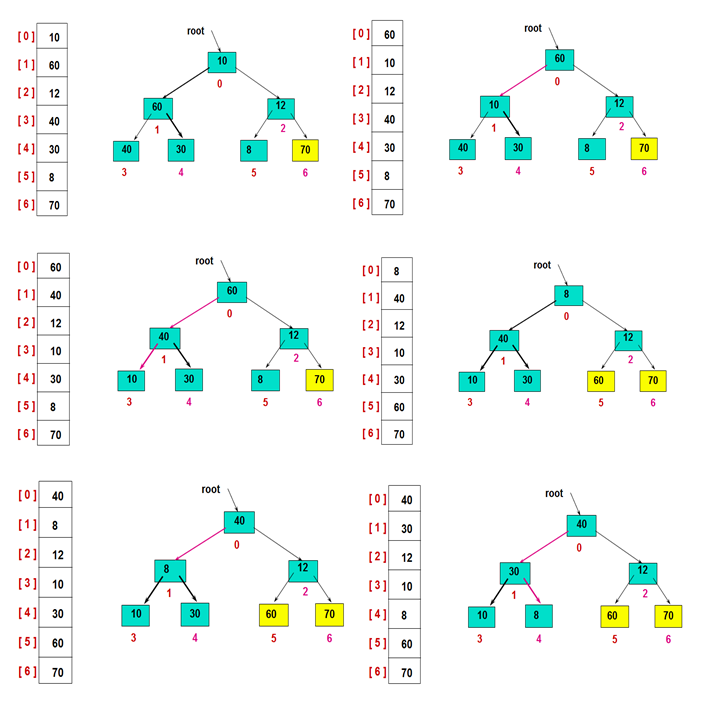
　　不断进行此操作，将剩下的元素重新调整为堆，然后输出堆顶元素到有序区。每次交换都导致无序区-1，有序区+1。不断重复此过程直到有序区长度增长为n-1，排序完成。

### **堆排序实例**

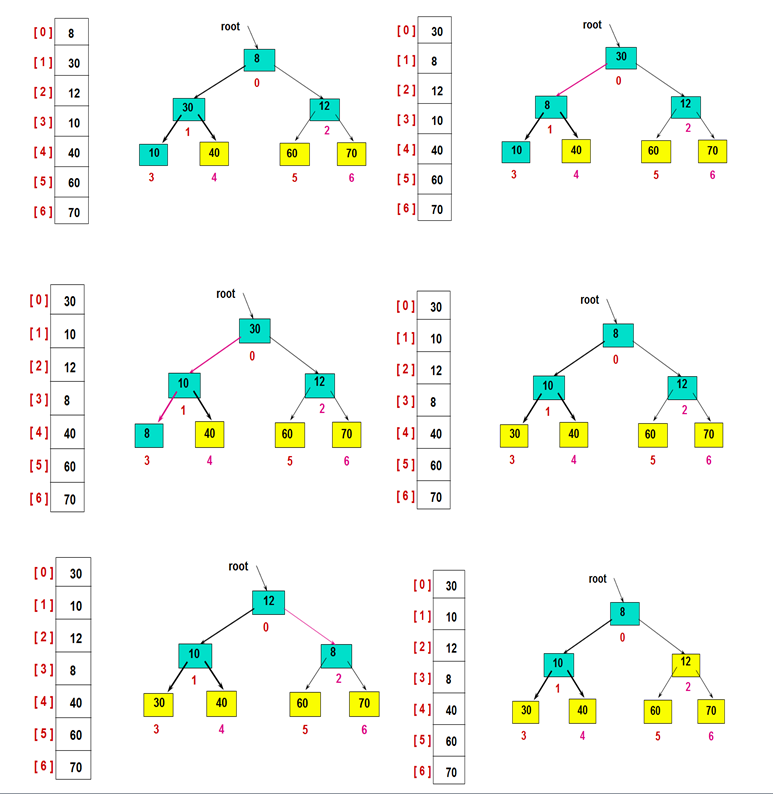
 　　首先，建立初始的堆结构如图：



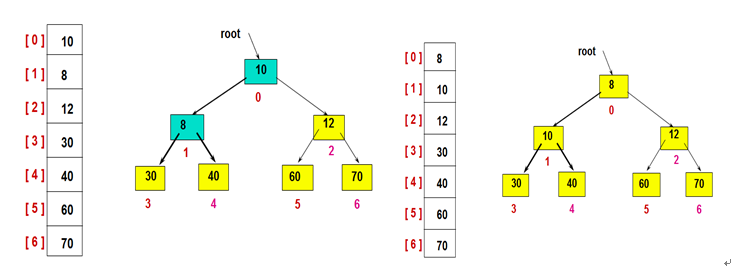
　　然后，交换堆顶的元素和最后一个元素，此时最后一个位置作为有序区（有序区显示为黄色），然后进行其他无序区的堆调整，重新得到大顶堆后，交换堆顶和倒数第二个元素的位置……



　　重复此过程：



　　最后，有序区扩展完成即排序完成：



　　由排序过程可见，**若想得到升序，则建立大顶堆，若想得到降序，则建立小顶堆**。

### **代码**

　　假设排列的元素为整型，且元素的关键字为其本身。

　　因为要进行升序排列，所以用大顶堆。

　　根结点从0开始，所以i结点的左右孩子结点的下标为2i+1和2i+2。

//堆筛选函数//已知H[start~end]中除了start之外均满足堆的定义//本函数进行调整，使H[start~end]成为一个大顶堆

typedef int ElemType;void HeapAdjust(ElemType H[], int start, int end)

{

ElemType temp = H[start];

for(int i = 2\*start + 1; i<=end; i\*=2)

{

//因为假设根结点的序号为0而不是1，所以i结点左孩子和右孩子分别为2i+1和2i+2

if(i<end && H[i]<H[i+1])//左右孩子的比较 {

++i;//i为较大的记录的下标 }

if(temp > H[i])//左右孩子中获胜者与父亲的比较 {

break;

}

//将孩子结点上位，则以孩子结点的位置进行下一轮的筛选

H[start]= H[i];

start = i;

}

H[start]= temp; //插入最开始不和谐的元素}

void HeapSort(ElemType A[], int n)

{

//先建立大顶堆

for(int i=n/2; i>=0; --i)//非叶子节点

{

HeapAdjust(A,i,n);

}

//进行排序

for(int i=n-1; i>0; --i)

{

//最后一个元素和第一元素进行交换

ElemType temp=A[i];

A[i] = A[0];

A[0] = temp;

//然后将剩下的无序元素继续调整为大顶堆

HeapAdjust(A,0,i-1);

}

}

**优先级队**

## 2.1 简介

              队列的特点是先进先出。通常都把队列比喻成排队买东西，大家都很守秩序，先排队的人就先买东西。但是优先队列有所不同，它不遵循先进先出的规则，而是根据队列中元素的优先权，优先权最大的先被取出。通常把优先队列比喻成现实生活中的打印。一个打印店里有很多打印机，每台机器的性能不一样，有的打印机打印很快，有的打印机打印速度很慢。当这些打印机陆陆续续打印完自己的任务时进入排队等候状态。如果我这个时候要打印一份文件，我选的不是第一个排队的打印机，而是性能最好，打印最快的打印机。

            重点:优先级队列，是要看优先级的，谁的优先级更高，谁就先得到权限。不分排队的顺序!

            基本操作：

**empty() 如果队列为空返回真**

**pop() 删除对顶元素**

**push() 加入一个元素**

**size() 返回优先队列中拥有的元素个数**

**top() 返回优先队列对顶元素**

                   在默认的优先队列中，优先级高的先出队。在默认的int型中先出队的为较大的数。

## 2.2 用堆实现优先队列

## 2.3 STL实现优先队列

使用方法：

头文件：

**#include <queue>**

声明方式：

1、普通方法：

**priority\_queue<int>q;**  
//通过操作，按照元素从大到小的顺序出队

Java中的

PriorityQueue是从JDK1.5开始提供的新的数据结构接口，它是一种基于优先级堆的极大优先级队列。优先级队列是不同于先进先出队列的另一种队列。每次从队列中取出的是具有最高优先权的元素。如果不提供Comparator的话，优先队列中元素默认按自然顺序排列，也就是数字默认是小的在队列头，字符串则按字典序排列（参阅 Comparable），也可以根据 Comparator 来指定，这取决于使用哪种构造方法。优先级队列不允许 null 元素。依靠自然排序的优先级队列还不允许插入不可比较的对象（这样做可能导致 ClassCastException）。

package com.test;

import java.util.Comparator;import java.util.PriorityQueue;import java.util.Queue;

public class PriorityQueueExample {

public static void main(String[] args) {

Queue<Integer> qi = new PriorityQueue<Integer>();

qi.add(5);

qi.add(2);

qi.add(1);

qi.add(10);

qi.add(3);

while (!qi.isEmpty()) {

System.out.print(qi.poll() + ",");

}

System.out.println();

System.out.println("-----------------------------");

// 自定义的比较器，可以让我们自由定义比较的顺序 Comparator<Integer> cmp;

cmp = new Comparator<Integer>() {

public int compare(Integer e1, Integer e2) {

return e2 - e1;

}

};

Queue<Integer> q2 = new PriorityQueue<Integer>(5, cmp);

q2.add(2);

q2.add(8);

q2.add(9);

q2.add(1);

while (!q2.isEmpty()) {

System.out.print(q2.poll() + ",");

}

}

}

output   
  
1,2,3,5,10,   
-----------------------------   
9,8,2,1,

# 3 应用

首先优先队列是由堆来实现的，所以以后用到优先队列的地方，可以直接用C++中的STL来直接实现，但是最好还是自己实现几次，否则我们只会成为傻瓜，什么东西就知道怎么用，不知道是如何实现的。

优先队列适用的范围很广，比如：

## 1、构造哈夫曼编码

              构造哈夫曼编码是找到节点集合中频率最小的两个点，然后合并节点在插入到集合中，再循环。。。

## 2、一些任务调度算法

            比如操作系统的线程的调度算法，有的是按照优先级来调度的，每次都执行优先级较高的线程

## 3、合并n个有序文件为一个有序文件

            首先把n个有序文件的第一个元素取出来，放到优先队列里面，然后取最小值，然后再插入元素导优先队列，取最小值。。。

## 由于优先队列内部是有堆实现的，所以适用于堆的都适用于优先队列

比如排序，找中位数，找最大的k个数

优先队列是队列的一种，不过它可以按照自定义的一种方式（数据的优先级）来对队列中的数据进行动态的排序

每次的push和pop操作，队列都会动态的调整，以达到我们预期的方式来存储。

例如：我们常用的操作就是对数据排序，优先队列默认的是数据大的优先级高

所以我们无论按照什么顺序push一堆数，最终在队列里总是top出最大的元素。

用法：

示例：将元素5，3，2，4，6依次push到优先队列中，print其输出。

1. 标准库默认使用元素类型的<操作符来确定它们之间的优先级关系。

priority\_queue<int> pq;

通过<操作符可知在整数中元素大的优先级高。  
故示例1中输出结果为： 6 5 4 3 2

2. 数据越小，优先级越高

priority\_queue<int, vector<int>, greater<int> >pq;

其中  
第二个参数为容器类型。  
第二个参数为比较函数。  
故示例2中输出结果为：2 3 4 5 6

3. 自定义优先级，重载比较符号

重载默认的 < 符号

struct node  
{  
 friend bool operator< (node n1, node n2)  
 {  
 return n1.priority < n2.priority;  
 }  
 int priority;  
 int value;  
};

这时，需要为每个元素自定义一个优先级。

注：重载>号会编译出错，因为标准库默认使用元素类型的<操作符来确定它们之间的优先级关系。  
而且自定义类型的<操作符与>操作符并无直接联系

代码：

#include<iostream>

#include<functional>

#include<queue>

using Namespace stdnamespace std;

struct node

{

friend bool operator< (node n1, node n2)

{

return n1.priority < n2.priority;

}

int priority;

int value;

};

int main()

{

const int len = 5;

int i;

int a[len] = {3,5,9,6,2};

//示例1

priority\_queue<int> qi;

for(i = 0; i < len; i++)

qi.push(a[i]);

for(i = 0; i < len; i++)

{

cout<<qi.top()<<" ";

qi.pop();

}

cout<<endl;

//示例2

priority\_queue<int, vector<int>, greater<int> >qi2;

for(i = 0; i < len; i++)

qi2.push(a[i]);

for(i = 0; i < len; i++)

{

cout<<qi2.top()<<" ";

qi2.pop();

}

cout<<endl;

//示例3

priority\_queue<node> qn;

node b[len];

b[0].priority = 6; b[0].value = 1;

b[1].priority = 9; b[1].value = 5;

b[2].priority = 2; b[2].value = 3;

b[3].priority = 8; b[3].value = 2;

b[4].priority = 1; b[4].value = 4;

for(i = 0; i < len; i++)

qn.push(b[i]);

cout<<"优先级"<<'\t'<<"值"<<endl;

for(i = 0; i < len; i++)

{

cout<<qn.top().priority<<'\t'<<qn.top().value<<endl;

qn.pop();

}

return 0;

}