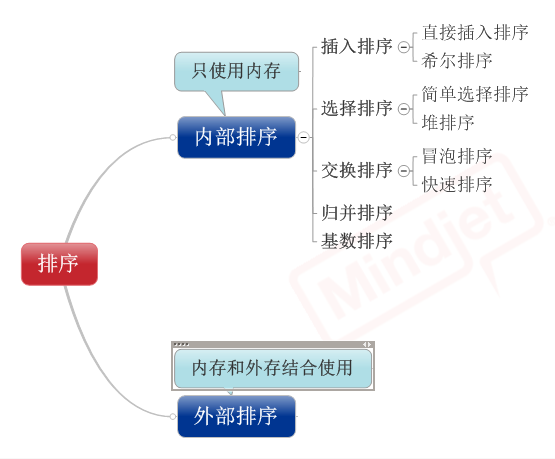
**程序员必知的8大排序（java实现）**

前几天，看到一篇前辈的博文“程序员必知的8大排序”，不禁的手痒起来，重新翻开严蔚敏老师的《数据结构》复习了一遍，然后一一的用java去实现，其中有不足之处，还望各位道友指正出来。

先来看看8种排序之间的关系：



1. **直接插入排序**

（1）基本思想：在要排序的一组数中，假设前面(n-1) [n>=2] 个数已经是排

好顺序的，现在要把第n个数插到前面的有序数中，使得这n个数

也是排好顺序的。如此反复循环，直到全部排好顺序。

（2）实例



（3）用java实现

**package** com.njue;

**public** **class** insertSort {

**public** insertSort(){

**int** a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,56,17,18,23,34,15,35,25,53,51};

**int** temp=0;

**for**(**int** i=1;i<a.length;i++){

**int** j=i-1;

temp=a[i];

**for**(;j>=0&&temp<a[j];j--){

a[j+1]=a[j]; //将大于temp的值整体后移一个单位

}

a[j+1]=temp;

}

**for**(**int** i=0;i<a.length;i++)

System.*out*.println(a[i]);

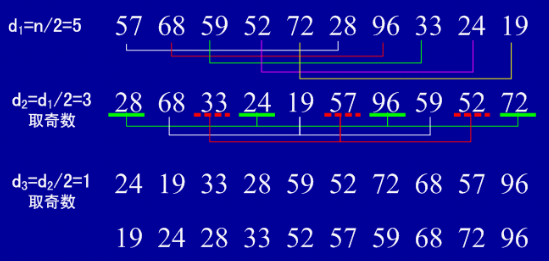
}

}

1. **希尔排序（最小增量排序）**

（1）基本思想：算法先将要排序的一组数按某个增量d（n/2,n为要排序数的个数）分成若干组，每组中记录的下标相差d.对每组中全部元素进行直接插入排序，然后再用一个较小的增量（d/2）对它进行分组，在每组中再进行直接插入排序。当增量减到1时，进行直接插入排序后，排序完成。

（2）实例：



（3）用java实现

**public** **class** shellSort {

**public** shellSort(){

**int** a[]={1,54,6,3,78,34,12,45,56,100};

**double** d1=a.length;

**int** temp=0;

**while**(**true**){

d1= Math.*ceil*(d1/2);

**int** d=(**int**) d1;

**for**(**int** x=0;x<d;x++){

**for**(**int** i=x+d;i<a.length;i+=d){

**int** j=i-d;

temp=a[i];

**for**(;j>=0&&temp<a[j];j-=d){

a[j+d]=a[j];

}

a[j+d]=temp;

}

}

**if**(d==1)

**break**;

}

**for**(**int** i=0;i<a.length;i++)

System.*out*.println(a[i]);

}

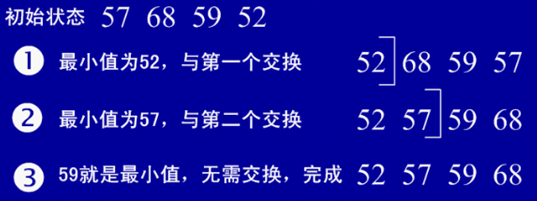
}

**3.简单选择排序**

（1）基本思想：在要排序的一组数中，选出最小的一个数与第一个位置的数交换；

然后在剩下的数当中再找最小的与第二个位置的数交换，如此循环到倒数第二个数和最后一个数比较为止。

（2）实例：



（3）用java实现

**public** **class** selectSort {

**public** selectSort(){

**int** a[]={1,54,6,3,78,34,12,45};

**int** position=0;

**for**(**int** i=0;i<a.length;i++){

**int** j=i+1;

position=i;

**int** temp=a[i];

**for**(;j<a.length;j++){

**if**(a[j]<temp){

temp=a[j];

position=j;

}

}

a[position]=a[i];

a[i]=temp;

}

**for**(**int** i=0;i<a.length;i++)

System.*out*.println(a[i]);

}

}

1. **堆排序**

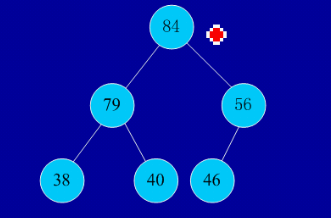
（1）基本思想：堆排序是一种树形选择排序，是对直接选择排序的有效改进。

堆的定义如下：具有n个元素的序列（h1,h2,...,hn),当且仅当满足（hi>=h2i,hi>=2i+1）或（hi<=h2i,hi<=2i+1）(i=1,2,...,n/2)时称之为堆。在这里只讨论满足前者条件的堆。由堆的定义可以看出，堆顶元素（即第一个元素）必为最大项（大顶堆）。完全二叉树可以很直观地表示堆的结构。堆顶为根，其它为左子树、右子树。初始时把要排序的数的序列看作是一棵顺序存储的二叉树，调整它们的存储序，使之成为一个堆，这时堆的根节点的数最大。然后将根节点与堆的最后一个节点交换。然后对前面(n-1)个数重新调整使之成为堆。依此类推，直到只有两个节点的堆，并对它们作交换，最后得到有n个节点的有序序列。从算法描述来看，堆排序需要两个过程，一是建立堆，二是堆顶与堆的最后一个元素交换位置。所以堆排序有两个函数组成。一是建堆的渗透函数，二是反复调用渗透函数实现排序的函数。

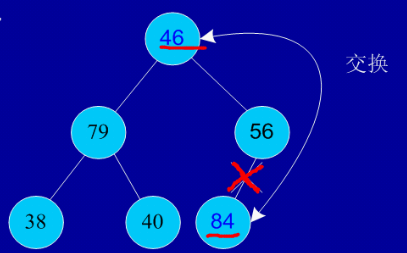
（2）实例：

初始序列：46,79,56,38,40,84

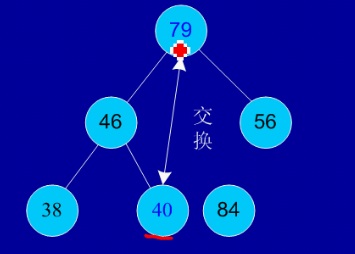
建堆：



交换，从堆中踢出最大数



剩余结点再建堆，再交换踢出最大数



依次类推：最后堆中剩余的最后两个结点交换，踢出一个，排序完成。

（3）用java实现

**import** java.util.Arrays;

**public** **class** HeapSort {

**int** a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,56,17,18,23,34,15,35,25,53,51};

**public** HeapSort(){

heapSort(a);

}

**public** **void** heapSort(**int**[] a){

System.*out*.println("开始排序");

**int** arrayLength=a.length;

//循环建堆

**for**(**int** i=0;i<arrayLength-1;i++){

//建堆

buildMaxHeap(a,arrayLength-1-i);

//交换堆顶和最后一个元素

swap(a,0,arrayLength-1-i);

System.*out*.println(Arrays.*toString*(a));

}

}

**private** **void** swap(**int**[] data, **int** i, **int** j) {

// **TODO** Auto-generated method stub

**int** tmp=data[i];

data[i]=data[j];

data[j]=tmp;

}

//对data数组从0到lastIndex建大顶堆

**private** **void** buildMaxHeap(**int**[] data, **int** lastIndex) {

// **TODO** Auto-generated method stub

//从lastIndex处节点（最后一个节点）的父节点开始

**for**(**int** i=(lastIndex-1)/2;i>=0;i--){

//k保存正在判断的节点

**int** k=i;

//如果当前k节点的子节点存在

**while**(k\*2+1<=lastIndex){

//k节点的左子节点的索引

**int** biggerIndex=2\*k+1;

//如果biggerIndex小于lastIndex，即biggerIndex+1代表的k节点的右子节点存在

**if**(biggerIndex<lastIndex){

//若果右子节点的值较大

**if**(data[biggerIndex]<data[biggerIndex+1]){

//biggerIndex总是记录较大子节点的索引

biggerIndex++;

}

}

//如果k节点的值小于其较大的子节点的值

**if**(data[k]<data[biggerIndex]){

//交换他们

swap(data,k,biggerIndex);

//将biggerIndex赋予k，开始while循环的下一次循环，重新保证k节点的值大于其左右子节点的值

k=biggerIndex;

}**else**{

**break**;

}

}

}

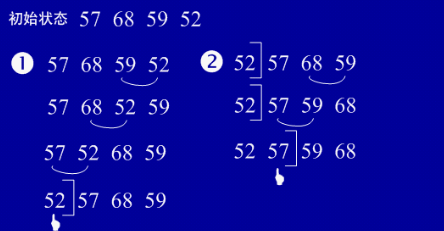
}

}

**5.冒泡排序**

（1）基本思想：在要排序的一组数中，对当前还未排好序的范围内的全部数，自上而下对相邻的两个数依次进行比较和调整，让较大的数往下沉，较小的往上冒。即：每当两相邻的数比较后发现它们的排序与排序要求相反时，就将它们互换。

（2）实例：



（3）用java实现

**public** **class** bubbleSort {

**public** bubbleSort(){

**int** a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,56,17,18,23,34,15,35,25,53,51};

**int** temp=0;

**for**(**int** i=0;i<a.length-1;i++){

**for**(**int** j=0;j<a.length-1-i;j++){

**if**(a[j]>a[j+1]){

temp=a[j];

a[j]=a[j+1];

a[j+1]=temp;

}

}

}

**for**(**int** i=0;i<a.length;i++)

System.*out*.println(a[i]);

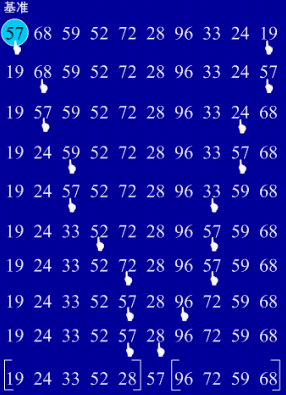
}

}

**6.快速排序**

（1）基本思想：选择一个基准元素,通常选择第一个元素或者最后一个元素,通过一趟扫描，将待排序列分成两部分,一部分比基准元素小,一部分大于等于基准元素,此时基准元素在其排好序后的正确位置,然后再用同样的方法递归地排序划分的两部分。

（2）实例：



（3）用java实现

**public** **class** quickSort {

**int** a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,56,17,18,23,34,15,35,25,53,51};

**public** quickSort(){

quick(a);

**for**(**int** i=0;i<a.length;i++)

System.*out*.println(a[i]);

}

**public** **int** getMiddle(**int**[] list, **int** low, **int** high) {

**int** tmp = list[low]; //数组的第一个作为中轴

**while** (low < high) {

**while** (low < high && list[high] >= tmp) {

high--;

}

list[low] = list[high]; //比中轴小的记录移到低端

**while** (low < high && list[low] <= tmp) {

low++;

}

list[high] = list[low]; //比中轴大的记录移到高端

}

list[low] = tmp; //中轴记录到尾

**return** low; //返回中轴的位置

}

**public** **void** \_quickSort(**int**[] list, **int** low, **int** high) {

**if** (low < high) {

**int** middle = getMiddle(list, low, high); //将list数组进行一分为二

\_quickSort(list, low, middle - 1); //对低字表进行递归排序

\_quickSort(list, middle + 1, high); //对高字表进行递归排序

}

}

**public** **void** quick(**int**[] a2) {

**if** (a2.length > 0) { //查看数组是否为空

\_quickSort(a2, 0, a2.length - 1);

}

}

}

**7、归并排序**

（1）基本排序：归并（Merge）排序法是将两个（或两个以上）有序表合并成一个新的有序表，即把待排序序列分为若干个子序列，每个子序列是有序的。然后再把有序子序列合并为整体有序序列。

（2）实例：



（3）用java实现

**import** java.util.Arrays;

**public** **class** mergingSort {

**int** a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,56,17,18,23,34,15,35,25,53,51};

**public** mergingSort(){

sort(a,0,a.length-1);

**for**(**int** i=0;i<a.length;i++)

System.*out*.println(a[i]);

}

**public** **void** sort(**int**[] data, **int** left, **int** right) {

// **TODO** Auto-generated method stub

**if**(left<right){

//找出中间索引

**int** center=(left+right)/2;

//对左边数组进行递归

sort(data,left,center);

//对右边数组进行递归

sort(data,center+1,right);

//合并

merge(data,left,center,right);

}

}

**public** **void** merge(**int**[] data, **int** left, **int** center, **int** right) {

// **TODO** Auto-generated method stub

**int** [] tmpArr=**new** **int**[data.length];

**int** mid=center+1;

//third记录中间数组的索引

**int** third=left;

**int** tmp=left;

**while**(left<=center&&mid<=right){

//从两个数组中取出最小的放入中间数组

**if**(data[left]<=data[mid]){

tmpArr[third++]=data[left++];

}**else**{

tmpArr[third++]=data[mid++];

}

}

//剩余部分依次放入中间数组

**while**(mid<=right){

tmpArr[third++]=data[mid++];

}

**while**(left<=center){

tmpArr[third++]=data[left++];

}

//将中间数组中的内容复制回原数组

**while**(tmp<=right){

data[tmp]=tmpArr[tmp++];

}

System.*out*.println(Arrays.*toString*(data));

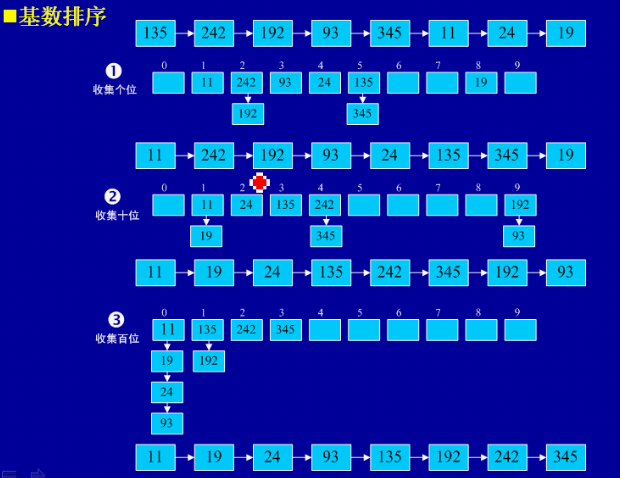
}

}

**8、基数排序**

（1）基本思想：将所有待比较数值（正整数）统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后,数列就变成一个有序序列。

（2）实例：



（3）用java实现

import java.util.ArrayList;

import java.util.List;

public class radixSort {

int a[]={49,38,65,97,76,13,27,49,78,34,12,64,5,4,62,99,98,54,101,56,17,18,23,34,15,35,25,53,51};

public radixSort(){

sort(a);

for(int i=0;i<a.length;i++)

System.out.println(a[i]);

}

public void sort(int[] array){

//首先确定排序的趟数;

int max=array[0];

for(int i=1;i<array.length;i++){

if(array[i]>max){

max=array[i];

}

}

int time=0;

//判断位数;

while(max>0){

max/=10;

time++;

}

//建立10个队列;

List<ArrayList> queue=new ArrayList<ArrayList>();

for(int i=0;i<10;i++){

ArrayList<Integer> queue1=new ArrayList<Integer>();

queue.add(queue1);

}

//进行time次分配和收集;

for(int i=0;i<time;i++){

//分配数组元素;

for(int j=0;j<array.length;j++){

//得到数字的第time+1位数;

int x=array[j]%(int)Math.pow(10, i+1)/(int)Math.pow(10, i);

ArrayList<Integer> queue2=queue.get(x);

queue2.add(array[j]);

queue.set(x, queue2);

}

int count=0;//元素计数器;

//收集队列元素;

for(int k=0;k<10;k++){

while(queue.get(k).size()>0){

ArrayList<Integer> queue3=queue.get(k);

array[count]=queue3.get(0);

queue3.remove(0);

count++;

}

}

}

}

}