C语言排序方法总结

C语言排序方法

学的排序算法有：插入排序，合并排序，冒泡排序，选择排序，希尔排序，堆排序，快速排序，计数排序，基数排序，桶排序（没有实现）。比较一下学习后的心得。

1.普及一下排序稳定，所谓排序稳定就是指：如果两个数相同，对他们进行的排序结果为他们的相对顺序不变。例如A={1,2,1,2,1}这里排序之后是A = {1,1,1,2,2} 稳定就是排序后第一个1就是排序前的第一个1，第二个1就是排序前第二个1，第三个1就是排序前的第三个1。同理2也是一样。这里用颜色标明了。不稳定呢就是他们的顺序不应和开始顺序一致。也就是可能会是A={1,1,1,2,2}这样的结果。

2.普及一下原地排序：原地排序就是指不申请多余的空间来进行的排序，就是在原来的排序数据中比较和交换的排序。例如快速排序，堆排序等都是原地排序，合并排序，计数排序等不是原地排序。

3.感觉谁最好，在我的印象中快速排序是最好的，时间复杂度：n*log(n)，不稳定排序。原地排序。他的名字很棒，快速嘛。当然快了。我觉得他的思想很不错，分治，而且还是原地排序，省去和很多的空间浪费。速度也是很快的，n*log(n)。但是有一个软肋就是如果已经是排好的情况下时间复杂度就是n*n,不过在加入随机的情况下这种情况也得以好转，而且他可以做任意的比较，只要你能给出两个元素的大小关系就可以了。适用范围广，速度快。

4.插入排序：n*n的时间复杂度，稳定排序，原地排序。插入排序是我学的第一个排序，速度还是很快的，特别是在数组已排好了之后，用它的思想来插入一个数据，效率是很高的。因为不用全部排。他的数据交换也很少，只是数据后移，然后放入要插入的数据。（这里不是指调用插入排序，而是用它的思想）。我觉得，在数据大部分都排好了，用插入排序会给你带来很大的方便。数据的移动和交换都很少。

5.冒泡排序，n*n的时间复杂度，稳定排序，原地排序。冒泡排序的思想很不错，一个一个比较，把小的上移，依次确定当前最小元素。因为他简单，稳定排序，而且好实现，所以用处也是比较多的。还有一点就是加上哨兵之后他可以提前退出。

6.选择排序，n*n的时间复杂度， 稳定排序，原地排序。选择排序就是冒泡的基本思想，从小的定位，一个一个选择，直到选择结束。他和插入排序是一个相反的过程，插入是确定一个元素的位置，而选择是确定这个位置的元素。他的好处就是每次只选择确定的元素，不会对很多数据进行交换。所以在数据交换量上应该比冒泡小。

7.插入排序，选择排序，冒泡排序的比较，他们的时间复杂度都是n*n。我觉得他们的效率也是差不多的，我个人喜欢冒泡一些，因为要用它的时候数据多半不多，而且可以提前的返回已经排序好的数组。而其他两个排序就算已经排好了，他也要做全部的扫描。在数据的交换上，冒泡的确比他们都多。呵呵。举例说明插入一个数据在末尾后排序，冒泡只要一次就能搞定，而选择和插入都必须要n*n的复杂度才能搞定。就看你怎么看待咯。

8.合并排序：n*log(n)的时间复杂度， 稳定排序，非原地排序。他的思想是分治，先分成小的部分，排好部分之后合并，因为我们另外申请的空间，在合并的时候效率是0(n)的。速度很快。貌似他的上限是n*log(n)，所以如果说是比较的次数的话，他比快速排序要少一些。对任意的数组都能有效地在n*log(n)排好序。但是因为他是非原地排序，所以虽然他很快，但是貌似他的人气没有快速排序高。

9.堆排序：n*log(n)的时间复杂度， 非稳定排序，原地排序。他的思想是利用的堆这种数据结构，堆可以看成一个完全二叉树，所以在排序中比较的次数可以做到很少。加上他也是原地排序，不需要申请额外的空间，效率也不错。可是他的思想感觉比快速难掌握一些。还有就是在已经排好序的基础上添加一个数据再排序，他的交换次数和比较次数一点都不会减少。虽然堆排序在使用的中没有快速排序广泛，但是他的数据结构和思想真的很不错，而且用它来实现优先队列，效率没得说。堆，还是要好好学习掌握的。

10.希尔排序：n*log(n)的时间复杂度(这里是错误的，应该是n^lamda(1 < lamda < 2), lamda和每次步长选择有关。)， 非稳定排序，原地排序。主要思想是分治，不过他的分治和合并排序的分治不一样，他是按步长来分组的，而不是想合并那样左一半右一半。开始步长为整个的长度的一半。分成nLen/2个组，然后每组排序。接个步长减为原来的一半在分组排序，直到步长为1，排序之后希尔排序就完成了。这个思路很好，据说是插入排序的升级版，所以在实现每组排序的时候我故意用了插入排序。我觉得他是一个特别好的排序方法了。他的缺点就是两个数可能比较多次，因为两个数据会多次分不过他们不会出现数据的交换。效率也是很高的。

11.快速排序，堆排序，合并排序，希尔排序的比较，他们的时间复杂的都是n*log(n)，我认为在使用上快速排序最广泛，他原地排序，虽然不稳定，可是很多情况下排序根本就不在意他是否稳定。他的比较次数是比较小的，因为他把数据分成了大和小的两部分。每次都确定了一个数的位置，所以理论上说不会出现两个数比较两次的情况，也是在最后在交换数据，说以数据交换上也很少。合并排序和堆排序也有这些优点，但是合并排序要申请额外的空间。堆排序堆已经排好的数据交换上比快速多。所以目前快速排序用的要广泛的多。还有他很容易掌握和实现。

12.计数排序：n的时间复杂度，稳定排序，非原地排序。他的思想比较新颖，就是先约定数据的范围不是很大，而且数据都是整数(或能定位到整数)的情况，然后直接申请一个空间。把要排序的数组A的元素值与申请空间B的下标对应，然后B中存放该下标元素值的个数，从而直接定位A中每个元素的位置。这样效率只为n。因为比较很特殊，虽然很快，但是用的地方并不多。

13.基数排序：n的时间复杂度，稳定排序，非原地排序。他的思想是数据比较集中在一个范围，例如都是4位数，都是5位数，或数据有多个关键字，我们先从各位开始排，然后排十位，依次排到最高位，因为我们可以用一个n的方法排一位，所以总的方法为d*n的复杂度。关键字也一样，我们先排第3个关键字，在排第3个关键字，最后排第一个关键字。只有能保证每个关键字在n的时间复杂度完成，那么整个排序就是一个d*n的时间复杂度。所以总的速度是很快的。不过有一点就是要确保关键字能在n的时间复杂度完成。

14.桶排序：n的时间复杂度，稳定排序，非原地排序。主要思路和基数排序一样，也是假设都在一个范围例如概率都在0-1，而且分布还挺均匀，那么我们也是和基数排序一样对一个数把他划分在他指定的区域。然后在连接这些区域就可以了。书上对每个区域使用链表的存储，我认为在寸小区域的时候也会有时间在里面。所以只是理论上的n时间复杂度。这种思路是不错的。呵呵。

15.计数排序，基数排序，桶排序的比较，我觉得他们都很有思想，不过都是在特定情况下才能发挥最大的效果。虽然效率很高，但是用的不会很广泛。他们之间我更喜欢计数排序，来个映射的方式就直接找到了自己的位置，很高明。和基数排序和同排序只是理论上的n时间复杂度，基数排序要确定一个关键字的排序是n复杂度的，桶排序要确定每个区域的排序是n复杂度的。

16.排序算法的最后感悟：黑格尔说过：存在即合理。所以这些排序的算法都是很好的，他确实给了我们思想上的帮助。感谢前人把精华留给了我们。我得到的收获很大，总结一下各自排序的收获：

冒泡：好实现，速度不慢，使用于轻量级的数据排序。

插入排序：也使用于小数据的排序，但是我从他的思想中学到怎么插入一个数据。呵呵，这样就知道在排好的数据里面，不用再排序了，而是直接调用一下插入就可以了。

选择排序：我学会了怎么去获得最大值，最小值等方法。只要选择一下，不就可以了。

合并排序：我学会分而治之的方法，而且在合并两个数组的时候很适用。

堆排序：可以用它来实现优先队列，而且他的思想应该给我加了很多内力。

快速排序：本来就用的最多的排序，对我的帮助大的都不知道怎么说好。

希尔排序：也是分治，让我看到了分治的不同，原来还有这种思想的存在。

计数排序，基数排序，桶排序：特殊情况特殊处理。

插入排序

插入排序主要思想是：把要排序的数字插入到已经排好的数据中。

例如12356是已经排好的序，我们将4插入到他们中，时插入之后也是排好序的。这里显而易见是插入到3的后面。变为123456.实现思路：插入排序就是先是一个有序的数据，然后把要插入的数据插到指定的位置，而排序首先给的就是无序的，我们怎么确定先得到一个有序的数据呢？答案就是：如果只有一个，当然是有序的咯。我们先拿一个出来，他是有序的，然后把数据一个一个插入到其中，那么插入之后是有序的，所以直到最后都是有序的。。结果就出来了！当然在写的时候还是有一个技巧的，不需要开额外的数组，下标从第二个元素开始遍历知道最后一个，然后插入到前面已经有序的数据中。这样就不会浪费空间了。插入排序用处还是很多的，特别是链表中，因为链表是指针存放的，没有数组那么好准确的用下标表示，插入是简单有效的方法。

源代码

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 //插入排序从下到大，nData为要排序的数据,nNum为数据的个数,该排序是稳定的排序
5 bool InsertionSort(int nData[], int nNum)
6 {
7     for (int i = 1; i < nNum; ++i)       //遍历数组，进行插入排序
8     {
9         int nTemp = nData[i];
10        for (int j = 0; j < i; ++j)       //对该数，寻找他要插入的位置
11        {
```

```
12          if (nData[j] > nTemp)    //找到位置，然后插入该位置，之后的数据后移
13          {
14              for (int k = i; k > j; --k)    //数据后移
15              {
16                  nData[k] = nData[k -1];
17              }
18              nData[j] = nTemp;       //将数据插入到指定位置
19              break;
20          }
21      }
22  }
23
24  return true;
25 }
26
27 int main()
28 {
29  int nData[10] = {4,10,9,8,7,6,5,4,3,2};    //创建10个数据，测试
30  InsertionSort(nData, 10);       //调用插入排序
31
32  for (int i = 0; i < 10; ++i)
33  {
34      printf("%d ", nData[i]);
35  }
36
37  printf("\n");
38  system("puase");
39  return 0;
40 }
```

冒泡排序

冒泡排序的主要思路：

我们把要排序的数组A = {3,4,2,1} 看成一组水泡， <!--[endif]-->就像冒泡一样，轻的在上面，重的在下面，换成数据，就是小的在上面，大的在下面。 我们先把最轻的冒出到顶端，然后冒出第二轻的在最轻的下面，接着冒出第三轻的。依次内推。直到所有都冒出来了为止。

3.我们怎么做到把最轻的放在顶端呢？我们从最底下的数据开始冒，如果比他上面的数据小，就交换（冒上去），然后再用第二第下的数据比较(此时他已经是较轻的一个)，如果他比他上面的小，则交换，把小的冒上去。直到比到第一位置，得到的就是最轻的数据咯，这个过程就像是冒泡一样，下面的和上面的比较，小的冒上去。大的沉下来。

画个图先：

最初	第一次结果	第二次结果	第三次结果
3	3	3	1
4	4	1	3
2	1	4	4
1	2	2	2

开始：1 和2 比，1比2小，浮上，然后1跟4比，再1跟3比，这样结构就变为1，3，4，2。最小的位置确定了，然后我们确定第二小的，同理2 vs

4, 2 vs 3 得到2， 再确定第3小数据，3 vs 4得到3，最后就是4为最大的数据，我们冒泡就排好了。

注：这里红色的1,2是前一次比较1 vs 2交换的结构。后面也一样。

大概思路就这样了，奉上源代码：

```
#include <stdio.h>
#include <stdlib.h>

//冒泡排序, pnData要排序的数据， nLen数据的个数
int BubbleSort(int* pnData, int nLen)
{
    bool isOk = false;       //设置排序是否结束的哨兵

    //i从[0,nLen-1)开始冒泡，确定第i个元素
    for (int i = 0; i < nLen - 1 && !isOk; ++i)
    {
        isOk = true;       //假定排序成功

        //从[nLen - 1, i）检查是否比上面一个小，把小的冒泡浮上去
        for (int j = nLen- 1; j > i; --j)
        {
            if (pnData[j] < pnData[j - 1])    //如果下面的比上面小，交换
            {
                int nTemp = pnData[j];
                pnData[j] = pnData[j - 1];
                pnData[j - 1] = nTemp;
                isOk = false;
            }
        }
    }

    return 1;
}

int main()
{
    int nData[10] = {4,10,9,8,7,6,5,4,3,2};    //创建10个数据，测试
    BubbleSort(nData, 10);       //调用冒泡排序

    for (int i = 0; i < 10; ++i)
    {
        printf("%d ", nData[i]);
    }

    printf("\n");
    system("pause");
    return 0;
```

```
}
```

选择排序

选择排序和冒泡排序思路上有一点相似，都是先确定最小元素，再确定第二笑元素，最后确定最大元素。他的主要流程如下：

1.加入一个数组A = {5,3,6,2,4,7}，我们对他进行排序

2.确定最小的元素放在A[0]位置，我们怎么确定呢，首先默认最小元素为5,他的索引为0,然后用它跟3比较，比他打，则认为最小元素为3,他的索引为1，然后用3跟6比，发现比他小，最小元素还是3，然后跟2比，最小元素变成了2，索引为3，然后跟4比，跟7比。当比较结束之后，最小元素也尘埃落定了。就是2，索引为3，然后我们把他放在A[0]处。为了使A[0]原有数据部丢失，我们使A[0](要放的位置) 与A[3]（最小数据的位置）交换。这样就不可以了吗？

3.然后我们在来找第二小元素，放在A[1]，第三小元素，放在A[2]。。当寻找完毕，我们排序也就结束了。

4.不过，在找的时候要注意其实位置，不能在找A[2]的时候，还用A[2]的数据跟已经排好的A[0],A[1]比，一定要跟还没有确定位置的元素比。还有一个技巧就是我们不能每次都存元素值和索引，我们只存索引就可以了，通过索引就能找到元素了。

5.他和冒泡的相似和区别，冒泡和他最大的区别是他发现比他小就交换，把小的放上面，而选择是选择到最小的在直接放在确定的位置。选择也是稳定的排序。

基本思路就这样了，奉上源代码：

```
#include <stdio.h>
#include <stdlib.h>

//选择排序, pnData要排序的数据， nLen数据的个数
int SelectSort(int* pnData, int nLen)
{
   //i从[0,nLen-1)开始选择，确定第i个元素
   for (int i = 0; i < nLen - 1; ++i)
   {
      int nIndex = i;

      //遍历剩余数据，选择出当前最小的数据
      for (int j = i + 1; j < nLen; ++j)
      {
         if (pnData[j] < pnData[nIndex])
         {
            nIndex = j;
         }
      }

      //如果当前最小数据索引不是i，也就是说排在i位置的数据在nIndex处
      if (nIndex != i)
      {
         //交换数据，确定i位置的数据。
         int nTemp = pnData[i];
         pnData[i] = pnData[nIndex];
         pnData[nIndex] = nTemp;
      }
   }

   return 1;
```

```
}

int main()
{
    int nData[10] = {4,10,9,8,7,6,5,4,3,2};    //创建10个数据，测试
    SelectSort(nData, 10);        //调用选择排序

    for (int i = 0; i < 10; ++i)
    {
        printf("%d ", nData[i]);
    }

    printf("\n");
    system("pause");
    return 0;
}
```

希尔排序

希尔排序，他的主要思想借用了合并排序的思想。不过他不是左边一半右边一半，而是按照步长来分，随着步长减少，分成的组也越少。然后进行各组的插入排序。主要思路就是这样。

```
#include <stdio.h>
#include <stdlib.h>

//对单个组排序
int SortGroup(int* pnData, int nLen, int nBegin,int nStep)
{
    for (int i = nBegin + nStep; i < nLen; i += nStep)
    {
        //寻找i元素的位置，
        for (int j = nBegin; j < i; j+= nStep)
        {
            //如果比他小，则这里就是他的位置了
            if (pnData[i] < pnData[j])
            {
                int nTemp = pnData[i];
                for (int k = i; k > j; k -= nStep)
                {
                    pnData[k] = pnData[k - nStep];
                }
                pnData[j] = nTemp;
            }
        }
    }
    return 1;
}
//希尔排序, pnData要排序的数据， nLen数据的个数
```

```
int ShellSort(int* pnData, int nLen)
{
    //以nStep分组，nStep每次减为原来的一半。
    for (int nStep = nLen / 2; nStep > 0; nStep /= 2)
    {
        //对每个组进行排序
        for (int i = 0 ;i < nStep; ++i)
        {
            SortGroup(pnData, nLen, i, nStep);
        }
    }

    return 1;
}

int main()
{
    int nData[10] = {4,10,9,8,7,6,5,4,3,2};    //创建10个数据，测试
    ShellSort(nData, 10);        //调用希尔排序

    for (int i = 0; i < 10; ++i)
    {
        printf("%d ", nData[i]);
    }

    printf("\n");
    system("pause");
    return 0;
}
```

合并排序合并排序的主要思想是：把两个已经排序好的序列进行合并，成为一个排序好的序列。例如：13579 2468这两个序列，各自都是排好序的，然后我们进行合并，成为123456789这样一个排好序的序列。貌似这个跟排序关系不大，因为排序给的是一个乱的序列，而合并是合并的两个已经排序好的序列。且慢，我们可以把需要排序的数据分解成N个子序列，当分解的子序列所包含数据个数为1的时候，那么这个序列不久是有序了吗？然后再合并。这个就是有名的"分治。例如321分成3，2，1三个序列，1这个序列是有序的啦。同理2，3都是有序的。然后我们逐一的合并他们。3，2合并为23，然后在23与1合并为123。哈哈，排序成功。合并排序主要思路就是这样了。

但是，问题又出来了，怎么合并两个有序列呢？我相信我应该理解了数组的存储方式，所以直接用数组说事啦。。我们先把下标定位到各有序子序列的开始，也把合并之后数组的下标定位到最初。那么下标对应的位置就是他们当前的最小值了。然后拿他们来比较，把更小的那个放到合并之后数组的下标位置。这样，合并后数组的第一个元素就是他们的最小值了。接着，控制合并后数组的下标后移一个，把比较时小数字所在序列对应的下标后移一个。这样。下次比较的时候，他得到就是他的第二小，（第一下已经合并了）就是当前最小值了，在于另一个序列的当前最小值比较，用小的一个放到合并后数组的相应位置。依次类推。接着当数据都合并玩了结束，合并完成。（这样说忒空泛了，云里雾里的，BS一下以前的我。）

1357 2468 来做例子：

(1回合)　1357 2468 00000(合并后数据空)

(2) 357 2468 100000(0表示空) 因为1 < 2所以把1放到合并后位置中了（这里1并不是丢掉了，而是下标变为指向3了，1是没有写而已。呵呵。理解为数组的下标指向了3）

(3) 357 468 120000 因为3 > 2，所以把而放进去

(4) 57 468 123000 同理3 < 4

(5) 57 68 1234000 同理5 > 4

(6) 7 68 1234500 同理5 > 6

(7) 7 8 1234560 同理7 > 6

(8) 0(空了) 8 12345670 同理7 < 8

(9) 0 0 12345678 弄最后一个

当然，这些只是思路。并不是一定一成不变的这样。合并OK，那么我们就可以用合并排序了哦！哈哈。。不过注意，那个321全部弄成一个单个数字，然后一个一个合并这样来合并似乎不是很好，貌似还有更好的解决方案。哈哈，对了，就是我先分一半来合并。如果这一半是排好序的，那么合并不久简单了吗？但是我怎么让一般排好序呢。呵呵简单，我一半在分一半合并排序，在分一半合并排序，直到分到两个都是1个了，就合并，ok!

例如，81726354：

(1)分成9172 6354

(2)把8172 分成 81 和72 把6354分成63和54

(3)81分成8和1,哦能合并了哦。合并为18, 同理72，63，54，也可以分解成单个合并为27,36,45

(4) 现在变为了 18, 27, 36, 45了，这个时侯，18 和27能合并了，合并为1278 同理36,合并为45 3456

(5) 好了最好吧，1278和3456合并为12345678.ok排序成功。

这样把一个问题分解为两个或多个小问题，然后在分解，最后解决小小问题，已达到解决打问题的目的。

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 //合并排序的合并程序他合并数组nData中位置为[nP,nM) 和[nM,nR).这个是更接近标准的思路
 5 bool MergeStandard(int nData[], int nP, int nM, int nR)
 6 {
 7     int n1 = nM - nP;       //第一个合并数据的长度
 8     int n2 = nR - nM;       //第二个合并数据的长度
 9
10     int *pnD1 = new int[n1 + 1];       //申请一个保存第一个数据的空间
11     int *pnD2 = new int[n2 + 1];       //申请二个保存第一个数据的空间
12
13     for (int i = 0; i < n1; ++i)       //复制第一个数据到临时空间里面
14     {
15         pnD1[i] = nData[nP + i];
16     }
17     pnD1[n1] = INT_MAX;                  //将最后一个数据设置为最大值(哨兵)
18
19     for (int i = 0; i < n2; ++i)       //复制第二个数据到临时空间里面
20     {
21         pnD2[i] = nData[nM + i];
22     }
23     pnD2[n2] = INT_MAX;                  //将最后一个数据设置为最大值(哨兵)
24
25     n1 =  n2 = 0;
```

```
26
27     while(nP < nR)
28     {
29         nData[nP++] = pnD1[n1] <  pnD2[n2] ? pnD1[n1++] : pnD2[n2++];       //取出当前最小值到指定位置
30     }
31
32     delete pnD1;
33     delete pnD2;
34     return true;
35 }
36
37 //合并排序的合并程序他合并数组nData中位置为[nP,nM) 和[nM,nR).
38 bool Merge(int nData[], int nP, int nM, int nR)
39 {
40     //这里面有几个注释语句是因为当时想少写几行而至。看似短了，其实运行时间是一样的，而且不易阅读。
41
42     int nLen1 = nM - nP;        //第一个合并数据的长度
43     int nLen2 = nR - nM;        //第二个合并数据的长度
44     int* pnD1 = new int[nLen1];    //申请一个保存第一个数据的空间
45     int* pnD2 = new int[nLen2];   //申请一个保存第一个数据的空间
46
47     int i = 0;
48     for ( i = 0; i < nLen1; ++i)       //复制第一个数据到临时空间里面
49     {
50         pnD1[i] = nData[nP + i];
51     }
52
53     int j = 0;
54     for (j = 0; j < nLen2; ++j)       //复制第二个数据到临时空间里面
55     {
56         pnD2[j] = nData[nM + j];
57     }
58
59     i = j = 0;
60     while (i < nLen1 && j < nLen2)
61     {
62         //nData[nP++] = pnD1[i] < pnD2[j] ? pnD1[i++] : pnD2[j++];       //取出当前最小值添加到数据中
63
64         if (pnD1[i] < pnD2[j])       //取出最小值，并添加到指定位置中，如果pnD1[i] < pnD2[j]
65         {
66             nData[nP] = pnD1[i];    //取出pnD1的值，然后i++，定位到下一个个最小值。
67             ++i;
68         }
69         else                        //这里同上
70         {
71             nData[nP] = pnD2[j];
72             ++j;
```

```
73        }
74        ++nP;                    //最后np++，到确定下一个数据
75    }
76
77    if (i < nLen1)                //如果第一个数据没有结束（第二个数据已经结束了）
78    {
79        while (nP < nR)            //直接把第一个剩余的数据加到nData的后面即可。
80        {
81            //nData[nP++] = pnD1[i++];
82            nData[nP] = pnD1[i];
83            ++nP;
84            ++i;
85        }
86    }
87    else                        //否则（第一个结束，第二个没有结束）
88    {
89        while (nP < nR)            //直接把第一个剩余的数据加到nData的后面即可。
90        {
91            //nData[nP++] = pnD2[j++];
92            nData[nP] = pnD2[j];
93            ++nP;
94            ++j;
95        }
96    }
97
98    delete pnD1;        //释放申请的内存空间
99    delete pnD2;
100
101    return true;
102 }
103
104 //合并的递归调用,排序[nBegin, nEnd)区间的内容
105 bool MergeRecursion(int nData[], int nBegin, int nEnd)
106 {
107    if (nBegin >= nEnd - 1)        //已经到最小颗粒了,直接返回
108    {
109        return false;
110    }
111
112    int nMid = (nBegin + nEnd) / 2;          //计算出他们的中间位置，便于分治
113    MergeRecursion(nData, nBegin, nMid);    //递归调用，合并排序好左边一半
114    MergeRecursion(nData, nMid, nEnd);        //递归调用，合并排序好右边一半
115    //Merge(nData, nBegin, nMid, nEnd);        //将已经合并排序好的左右数据合并，时整个数据排序完成
116    MergeStandard(nData, nBegin, nMid, nEnd);//(用更接近标准的方法合并)
117
118    return true;
119 }
```

```
120
121 //合并排序
122 bool MergeSort(int nData[], int nNum)
123 {
124     return MergeRecursion(nData, 0, nNum);       //调用递归，完成合并排序
125 }
126
127 int main()
128 {
129     int nData[10] = {4,10,3,8,5,6,7,4,9,2};    //创建10个数据，测试
130
131     MergeSort(nData, 10);
132     for (int i = 0; i < 10; ++i)
133     {
134         printf("%d ", nData[i]);
135     }
136
137     printf("\n");
138     system("pause");
139     return 0;
140 }
141
```

堆排序

```
#include <stdio.h>
#include <stdlib.h>

//交换两个整数。注意一定要if判断是否两个相等，如果
//不相等才交换，如果相等也交换会出错的。a^a = 0
inline void Swap(int& a, int& b)
{
    if (a != b)
    {
        a^= b;
        b^= a;
        a^= b;
    }
}

//维持一个最大堆
int Heapify(int* npData, int nPos, int nLen)
{
    int nMax = -1;                    //暂存最大值
    int nChild = nPos * 2;              //他的左孩子位置

    while(nChild <= nLen)              //判断他是否有孩子
```

```
    {
        nMax = npData[nPos];          //是当前最大值为他

        if (nMax < npData[nChild])       //与左孩子比较
        {
            nMax = npData[nChild];       //如果比左孩子小，就时最大值为左孩子
        }

        //同理与右孩子比较，这里要注意，必须要保证有右孩子。
        if (nChild + 1 <= nLen && nMax < npData[nChild + 1])
        {
            ++nChild;                 //赋值最大值的时候把孩子变为右孩子，方便最后的数据交换
            nMax = npData[nChild];

        }

        if (nMax != npData[nPos])       //判断是否该节点比孩子都打，如果不大
        {
            Swap(npData[nPos], npData[nChild]);   //与最大孩子交换数据
            nPos = nChild;                    //该节点位置变为交换孩子的位置
            nChild *= 2;                      //因为只有交换后才使不满足堆得性质。
        }
        else                          //都最大了，满足堆得性质了。退出循环
        {
            break;
        }
    }

    return 1;                     //维持结束。
}

//建立一个堆
int BuildHeap(int* npData, int nLen)
{
    //从nLen / 2最后一个有叶子的数据开始，逐一的插入堆，并维持堆得平衡。
    //因为堆是一个完全二叉树，所以nlen/2+1- nLen之间肯定都是叶子。
    //叶子还判断什么呢。只有一个数据，肯定满足堆得性质咯。
    for (int i = nLen / 2; i >= 1; --i)
    {
        Heapify(npData, i, nLen);
    }

    return 1;
}

//堆排序
```

```
int HeapSort(int* npData, int nLen)
{
    BuildHeap(npData, nLen);        //建立一个堆。

    while(nLen >= 1)                //逐一交和第一个元素交换数据到最后
    {                               //完成排序
        Swap(npData[nLen], npData[1]);
        --nLen;
        Heapify(npData, 1, nLen);//交换之后一定要维持一下堆得性质。
    }                               //不然小的成第一个元素,就不是堆了。

    return 1;
}

//main函数，
int main()
{
    int nData[11] = {0,9,8,7,6,5,4,3,2,1,0};    //测试数据，下标从1开始哦。
    HeapSort(nData, 10);                         //堆排序

    for (int i = 1; i <= 10; ++i)                //输出排序结果。
    {
        printf("%d ", nData[i]);
    }
    printf("\n");
     system("pause");
    return 0;
}
```

用堆排序实现优先队列

1.一个是他是一个数组（当然你也可以真的用链表来做。）。

2.他可以看做一个完全二叉树。注意是完全二叉树。所以他的叶子个数刚好是nSize / 2个。

3.我使用的下标从1开始，这样好算，如果节点的位置为i,他的父节点就是i/2,他的左孩子结点就是i*2,右孩子结点就是i*2+1，如果下标从0开始，要复杂一点。

4.他的父节点一定不比子节点小（我所指的是最大堆）。

由这些性质就可以看出堆得一些优点：

1.可以一下找到最大值，就在第一个位置heap[1].

2.维持堆只需要log(2,n)(n是数据个数)的复杂度，速度比较快。他只需要比较父与子之间的大小关系，所以比较次数就是树的高度，而他是一个完全二叉树，所以比较次数就是log(2,n)。

具体实现：

具体实现就看看源代码吧！

```
#include <stdio.h>
#include <stdlib.h>

//定义一个堆得结构体，
```

```
struct MyHeap
{
    int* pnData;    //指向数据的指针
    int nSize;      //当前堆中的元素个数
};

//调整数据，维持堆得性质，这个和上次heapify的作用一样
//只是这个时从子道父节点这样的判断而已。
int IncreaseKey(MyHeap* pHeap, int nPos)
{
    //循环和他父节点判断，只要 nPos > 1他就有父节点
    while(nPos > 1)
    {
        int nMax = pHeap->pnData[nPos];
        int nParent = nPos / 2;

        //如果他比父节点大，交换数据，并使判断进入父节点
        //（因为只有父节点可能会影响堆得性质。他的数据改变了。）
        if (nMax > pHeap->pnData[nParent])
        {
            pHeap->pnData[nPos] = pHeap->pnData[nParent];
            pHeap->pnData[nParent] = nMax;
            nPos = nParent;
        }
        else        //否则堆没有被破坏，退出循环
        {
            break;
        }
    }

    return 1;
}

//插入数据，这里pnHeap为要插入的队，nLen为当前堆得大小。
//nData为要插入的数据，这里注意报保证堆得空间足够。
int Insert(MyHeap* pHeap, int nData)
{
    ++pHeap->nSize;             //添加数据到末尾
    pHeap->pnData[pHeap->nSize] = nData;
    IncreaseKey(pHeap, pHeap->nSize);
    return 1;
}

//弹出堆中对大元素，并使堆得个数减一
int PopMaxHeap(MyHeap* pHeap)
{
```

```
    int nMax = pHeap->pnData[1]; //得到最大元素

    //不要忘记维持堆得性质，因为最大元素已经弹出了，主要思路就是
    //同他最大孩子填充这里。

    int nPos = 1;           //起始位1，因为他弹出，所以是这里开始破坏堆得性质的
    int nChild = nPos * 2;    //他的左孩子的位置，

    //循环填充，用最大孩子填充父节点
    while(nChild <= pHeap->nSize)
    {
        int nTemp = pHeap->pnData[nChild];
        if (nChild + 1 <= pHeap->nSize &&
            nTemp < pHeap->pnData[nChild + 1])
        {
            ++nChild;
            nTemp = pHeap->pnData[nChild];
        }
        pHeap->pnData[nPos] = nTemp;
        nPos = nChild;
        nChild *= 2;
    }
    //最好一个用最末尾的填充。
    pHeap->pnData[nPos] = pHeap->pnData[pHeap->nSize];
    --pHeap->nSize;          //堆个数量减一
    return nMax;          //返回最大值。
}

//程序入口main
int main()
{
    MyHeap myHeap;          //定义一个堆
    myHeap.pnData = (int*)::malloc(sizeof(int) *11); //申请数据空间
    myHeap.nSize = 0;          //初始大小为0

    for (int i = 1; i <= 10; ++i)       //给优先队列堆里添加数据
    {
        Insert(&myHeap, i);
    }

    for (int i = 1; i <= 10; ++i)       //测试优先队列是否建立成功
    {
        printf("%d ", myHeap.pnData[i]);
    }
    printf("\n");
```

```
    while(myHeap.nSize > 0)  //逐一弹出队列的最大值。并验证
    {
        printf("%d ", PopMaxHeap(&myHeap));
    }
    printf("\n");

    ::free(myHeap.pnData);        //最后不要忘记释放申请的空间
    system("pause");
    return 0;
}
```

基数排序

据说他的时间复杂度也是O(n)，他的思路就是：

没有计数排序那么理想，我们的数据都比较集中，都比较大，一般是4，5位。基本没有小的数据。

那我们的处理很简单，你不是没有小的数据嘛。我给一个基数，例如个位，个位都是[0-10)范围内的。先对他进行归类，把小的放上面，大的放下面，然后个位排好了，在来看10位，我们也这样把小的放上面，大的放下面，依次内推，直到最高位排好。那么不就排好了吗？我们只需要做d(基数个数)的循环就可以了。时间复杂度相当于O(d * n) 因为d为常量，例如5位数，d就是5.所以近似为O(n)的时间复杂度。这次自己写个案例：

最初的数据	排好个位的数据	排好十位的数据	排好百位的数据
981	981	725	129
387	753	129	387
753	955	753	456
129	725	955	725
955	456	456	753
725	387	981	955
456	129	387	981

这里只需循环3次就出结果了。

<!--[if !supportLists]-->3. <!--[endif]-->但是注意，我们必须要把个位排好。但是个位怎么排呢？这个是个问题。书上说的是一叠一叠的怎么合并，我是没有理解的。希望知道的有高手教我一下。

我是用的一个计数排序来排各位的，然后排十位。效率应该也低不到哪里去。

思路就这样咯。奉上源代码：

```
#include <stdio.h>
#include <stdlib.h>

//计数排序，npRadix为对应的关键字序列，nMax是关键字的范围。npData是具体要
//排的数据，nLen是数据的范围，这里必须注意npIndex和npData对应的下标要一致
//也就是说npIndex[1] 所对应的值为npData[1]
int RadixCountSort(int* npIndex, int nMax, int* npData, int nLen)
{
    //这里就不用说了，计数的排序。不过这里为了是排序稳定
    //在标准的方法上做了小修改。

    int* pnCount  = (int*)malloc(sizeof(int)* nMax);      //保存计数的个数
```

```
    for (int i = 0; i < nMax; ++i)
    {
        pnCount[i] = 0;
    }
    for (int i = 0; i < nLen; ++i)    //初始化计数个数
    {
        ++pnCount[npIndex[i]];
    }

    for (int i = 1; i < 10; ++i)  //确定不大于该位置的个数。
    {
        pnCount[i] += pnCount[i - 1];
    }

    int * pnSort  = (int*)malloc(sizeof(int) * nLen);    //存放零时的排序结果。

    //注意：这里i是从nLen-1到0的顺序排序的，是为了使排序稳定。
    for (int i = nLen - 1; i >= 0; --i)
    {
        --pnCount[npIndex[i]];
        pnSort[pnCount[npIndex[i]]] = npData[i];
    }

    for (int i = 0; i < nLen; ++i)        //把排序结构输入到返回的数据中。
    {
        npData[i] = pnSort[i];
    }
    free(pnSort);                        //记得释放资源。
    free(pnCount);
    return 1;
}

//基数排序
int RadixSort(int* nPData, int nLen)
{
    //申请存放基数的空间
    int* nDataRadix    = (int*)malloc(sizeof(int) * nLen);

    int nRadixBase = 1;    //初始化倍数基数为1
    bool nIsOk = false; //设置完成排序为false

    //循环，知道排序完成
    while (!nIsOk)
    {
        nIsOk = true;
        nRadixBase *= 10;
```

```
        for (int i = 0; i < nLen; ++i)
        {
            nDataRadix[i] = nPData[i] % nRadixBase;
            nDataRadix[i] /= nRadixBase / 10;
            if (nDataRadix[i] > 0)
            {
                nIsOk = false;
            }
        }
        if (nIsOk)        //如果所有的基数都为0，认为排序完成，就是已经判断到最高位了。
        {
            break;
        }
        RadixCountSort(nDataRadix, 10, nPData, nLen);
    }

    free(nDataRadix);

    return 1;
}

int main()
{
    //测试基数排序。
    int nData[10] = {123,5264,9513,854,9639,1985,159,3654,8521,8888};
    RadixSort(nData, 10);
    for (int i = 0; i < 10; ++i)
    {
        printf("%d ", nData[i]);
    }
    printf("\n");

    system("pause");
    return 0;
}
```

计数排序：

貌似计数排序的复杂度为o(n)。很强大。他的基本思路为：

<!--[if !supportLists]-->1. <!--[endif]-->我们希望能线性的时间复杂度排序，如果一个一个比较，显然是不实际的，书上也在决策树模型中论证了，比较排序的情况为nlogn的复杂度。

<!--[if !supportLists]-->2. <!--[endif]-->既然不能一个一个比较，我们想到一个办法，就是如果我在排序的时候就知道他的位置，那不就是扫描一遍，把他放入他应该的位置不就可以了嘛。

<!--[if !supportLists]-->3. <!--[endif]-->要知道他的位置，我们只需要知道有多少不大于他不就可以了吗？

<!--[if !supportLists]-->4. <!--[endif]-->以此为出发点，我们怎么确定不大于他的个数呢？我们先来个约定，如果数组中的元素都比较集中，都在[0, max]范围内。我们开一个max的空间b数组，把b数组下标对应的元素和要排序的A数组下标对应起来。这样不就可以知道不比他大的有多少个了吗？我们只要把比他小的位置元素个数求和，就是不比他大的。例如：A={3,5,7};我们开一个大小为8的数组b，把a[0] = 3 放入b[3]中，使b[3] = 0; 同理 b[5] = 1; b[7] = 2;其他我们都设置为-1，哈哈我们只需要遍历一下b数组，如果他有数据，就来出来，铁定是当前最小的。如果要知道比a[2]小的数字有多少个，值只需要求出b[0] – b[6]的有数据的和就可以了。这个0(n)的速度不是盖得。

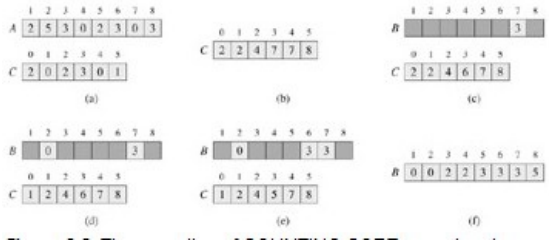
<!--[if !supportLists]-->5. <!--[endif]-->思路就是这样咯。但是要注意两个数相同的情况A = {1,2,3,3,4}，这种情况就不可以咯，所以还是有点小技巧的。

<!--[if !supportLists]-->6. <!--[endif]-->处理小技巧：我们不把A的元素大小与B的下标一一对应，而是在B数组对应处记录该元素大小

的个数。这不久解决了吗。哈哈。例如A = {1,2,3,3,4}我们开大小为5的数组b;记录数组A中元素值为0的个数为b[0] = 0, 记录数组A中元素个数为1的b[1] = 1,同理b[2] = 1, b[3] = 2, b[4] = 1;好了，这样我们就知道比A[4](4)小的元素个数是多少了：count = b[0] + b[1] + b[2] + b[3] = 4;他就把A[4]的元素放在第4个位置。

还是截张书上的图：

<!--[if !vml]-->



<!--[endif]-->

再次推荐《算法导论》这本书，在我的上次的随笔中有下载链接。哈哈。真正支持还是需要买一下纸版。呵呵。

<!--[if !supportLists]-->7. 不过在编程的时候还是要注意细节的，例如我不能每次都来算一下比他小的个数。呵呵，思路就这样了。奉上源代码：

```
#include <stdio.h>
#include <stdlib.h>

//计数排序
int CountSort(int* pData, int nLen)
{
    int* pCout = NULL;          //保存记数数据的指针
    pCout = (int*)malloc(sizeof(int) * nLen);    //申请空间
    //初始化记数为0
    for (int i = 0; i < nLen; ++i)
    {
        pCout[i] = 0;
    }

    //记录排序记数。在排序的值相应记数加1。
    for (int i = 0; i < nLen; ++i)
    {
        ++pCout[pData[i]];       //增
    }

    //确定不比该位置大的数据个数。
    for (int i = 1; i < nLen; ++i)
    {
        pCout[i] += pCout[i - 1];    //不比他大的数据个数为他的个数加上前一个的记数。
    }

    int* pSort = NULL;          //保存排序结果的指针
    pSort = (int*)malloc(sizeof(int) * nLen);    //申请空间

    for (int i = 0; i < nLen; ++i)
    {
        //把数据放在指定位置。因为pCout[pData[i]]的值就是不比他大数据的个数。
        //为什么要先减一，因为pCout[pData[i]]保存的是不比他大数据的个数中包括了
```

```
        //他自己，我的下标是从零开始的!所以要先减一。
        --pCout[pData[i]];    //因为有相同数据的可能，所以要把该位置数据个数减一。
        pSort[pCout[pData[i]]] = pData[i];

    }

    //排序结束，复制到原来数组中。
    for (int i = 0; i < nLen; ++i)
    {
        pData[i] = pSort[i];
    }

    //最后要注意释放申请的空间。
    free(pCout);
    free(pSort);

    return 1;
}

int main()
{
    int nData[10] = {8,6,3,6,5,8,3,5,1,0};
    CountSort(nData, 10);
    for (int i = 0; i < 10; ++i)
    {
        printf("%d ", nData[i]);
    }
    printf("\n");

    system("pause");
    return 0;
}
```