第 11 章 排序算法

从操作角度看，排序是线性结构的一种操作，在当今的计算机系统中，花费在排序上的时间占系统 CPU 运行时间的比重很大。有资料表明，在一些商用计算机上，用在排序上的 CPU 时间达到 20%至 60%。为了提高排序效率，人们已对排序进行了许多研究，提出了许多方法/算法。从算法设计角度看。排序算法不仅对排序本身重要，而且展示了算法设计的某些重要原则和高超的技巧，所以也是重要的算法设计方法。因此，对于计算机专业人员来说，认真研究和掌握排序算法是十分重要的。

§11.1 概述

排序（Sorting）（也称整序）通常被理解为按规定的原则重新安排一组给定的对象的的排列次序。排序的主要目的是便于以后在已排序的集合中查找/检索某一成员。日常生活中，通过排序以便于检索的例子屡见不鲜。例如，电话号码簿、目录表、图书馆、词典、仓库以及一切需对所存贮的对象进行检索的地方都要先将对象加以排序。下面先介绍若干概念。

1．关键字：关键字形式上是记录中的某个字段（或某些字段的组合），在具体的操作中，用于代表其所在的记录。对排序操作，是以关键字字段的内容的大小进行排序的。记录中关键字字段的值称为键值。键值一般为数值或字符串。

2．排序：在数据结构中，排序被认为是定义在数据集合上的一种运算，其功能是将一组任意排列的数据元素（在排序中称为记录）重新排列成一个按键值有序的序列。具体地，排序定义如下定义：

设｛R_1，R_2，…，R_n｝为含 n 个记录的序列，其相应的键值序列为｛k_1，k_2，…k_n｝。排序是将这些记录重新排列为｛R_{i1}，R_{i2}，…，R_{in}｝，使得相应的键值满足条件 $k_{i1}\leqslant k_{i2}\leqslant\cdots\leqslant k_{in}$（**升序**），或满足 $k_{i1}\geqslant k_{i2}\geqslant\cdots\geqslant k_{in}$（**降序**）。这里的比较运算（≤或≥），或是数值比较，或是字符串比较。

3．排序的稳定性：假定在待排序的记录集中，存在多个具有相同键值的记录，若经过排序，这些记录的相对次序仍然保持不变（即在原序列中 $k_i=k_j$，$i<j$；而在排序后的序列中，R_i 仍在 R_j 之前），则称这种排序方法是**稳定的**，否则称为不稳定的。

4．内排序：由于记录的形式、数量和所存放的存储设备不同，排序所采用的方法也不同。按照排序过程涉及的存储设备的不同，排序分为内排序（internal sorting）和外排

序（external sorting）两大类。**内排序**是指在排序的整个过程中，数据全部存放在计算机的内存中，排序所需所有操作只需在内存中进行，不需进行内外存交换。内排序适用于记录个数不太多的小文件，同时也是外排序的基础。

5．外排序：是指在排序过程中，待排序的数据较多，不能一次全部装入内存继续排序，需进行内外存交换（即读入部分数据，处理后写回外存，然后再读入其他部分）。外排序适用于记录个数很多，不能一次将其全部都放入内存的大文件。

本章先集中讨论内排序的各种典型方法和算法，最后再简单介绍一下外排序。

6．多键排序：按多个关键字排序。这主要针对关键字值有重复的情况。首先按第一关键字排序，对第一关键字相等的记录，按第二关键字排序，而第二关键字也相等的记录再按第三关键字排，余类推。

多键排序有两种方式（设第 1 到第 n 个关键字分别为 $k_1, k_2, \ldots, k_n$）：

a) 依次对记录进行 n 次排序，第一次按 k_n 排序，第二次按 k_{n-1}，…，最后一次按 k_1 排。这种方法要求各趟排序所用算法是稳定的。

b) 先将关键字 $k_1, k_2, \ldots, k_n$ 分别视为字符串，将它们依次首尾连接在一起，形成一个字符串，然后，对记录集按新形成的字符串排序。

不论那种无法，多键排序都被转化为了单键排序，所以，我们只讨论单键排序的情况。

显然，方法 b)的速度要快于 a).

内排序的方法很多，如果按排序过程中依据的不同原则对内排序方法进行分类，则主要分为为：插入排序、交换排序、选择排序、归并排序等四类，另一类是分配排序。

为了比较各种排序算法的优劣，要分析算法的时间复杂性。为此，通常只考虑键值的比较次数和记录的移动次数，即以键值比较和记录移动为标准操作。当键值是字串的时候，比较要占用较多的时间，是影响时间复杂性的主要因素。而当记录很大时，为了交换记录的位置，移动记录也要占用较多的时间，是影响时间复杂性的另一个主要因素。评价排序的另一个主要标准是执行算法所需的附加空间。

排序一般在记录集上进行，也可以认为是在线性结构上进行。为了突出主题，我们不失一般性地假定，待排序的记录为一维数组 a[]，数组的每个元素是整数。另外，如无特别说明，都假定是按升序排序。

§11.2 插入排序

插入排序类似玩纸牌时整理手中纸牌的过程。插入排序的基本方法是：每步将一个待排序的记录按其关键字的大小插到前面已经排序的序列中的适当位置，直到全部记录插入完毕为止。

§11.2.1 直接插入排序

直接插入排序（straight insertion sort）是一种最简单的排序方法，它的基本思想是，将待排序的记录集分成两部分，第一部分已排好序，第二部分未排好序。排序中，每次都是从第二部分中取出一条记录，就将其插入到第一部分，并使其保持有序。初始时，任取一条记录作为第一部分（一条记录总是有序的）,而其他记录作为第二部分。显然，随着排序过程的进行，第一部分不断扩大，第二部分不断缩小。总有一次，第二部分变为空，则第一部分包含了所有的记录，并且是有序的。

图 11-1 给出了一个插入排序执行过程的例子。

下面考虑插入排序的计算机程序实现。很自然地，我们可以将插入排序粗略地描述为：

```
SortInsertion(int a[], long n)
{
  for (i=1; i<n; i++)
  将 a[i]插入到 a[0]—a[i-1]中，并使其保持有序
}
```

至于“将 a[i]插入到 a[0]—a[i-1]中，并使其保持有序”的实现，可如下进行：

```
x=a[i];
j=i-1;
while (j>=0 && x<a[j])
{
  a[j+1] = a[j]; //a[j]后移一位
  j--;
}
a[j+1] = x;
```

综合起来，就得到插入排序的程序：

```
int SortInsertion(int a[], long n)
{
   int x, i,j;

   for (i=1; i<n; i++)
   {
     x=a[i];
     j=i-1;
     while (j>=0 && x<a[j])
     {
```

```
      a[j+1] = a[j];
      j--;
    }//while
    a[j+1] = x;
  } //for
 return 0;
}
```

如果为了提高 while 循环的循环控制条件的判别速度，可在 while 进行前，将当前待插入的元素 a[i]放在 a 数组中的第一个元素位置（代替语句 x=a[i]），这样，while 控制条件中的“j>=0”就不需要了，从而起到节省时间的作用。

初始键值序列	[46]	58	15	45	90	18	10	62
i=2	[46	58]	15	45	90	18	10	62
i=3	[15	46	58]	45	90	18	10	62
i=4	[15	45	46	58]	90	18	10	62
i=5	[15	45	46	58	90]	18	10	62
i=6	[15	18	45	46	58	90]	10	62
i=7	[10	15	18	45	46	58	90]	62
i=8	[10	15	18	45	46	58	62	90]

图 11-1 直接插入排序过程示例

显然，该算法是稳定的，且时间复杂性为 O（n^2）。从空间来看，它只需要一个记录的辅助空间。

§11.2.2 其他插入排序算法

除了直接插入排序外，还有其他形式的插入排序，如折半插入排序、表插入排序和希尔排序等。它们是对直接插入排序的改进。改进的关键点是，如何尽快地找到插入位置。例如，折半插入排序是每次将当前待插入元素与第一部分的中点位置上的元素比较，这样当记录数目很多时，可大大减少为寻找插入点而进行的比较次数。具体程序的编制，留做练习。

§11.3 交换排序

这里的交换，就是根据记录集中两个记录键值的比较结果来对换这两个记录在序列中的位置。交换排序的特点是：将键值较大的记录向记录集的一端移动，键值较小的记录向另一端移动。

§11.3.1 冒泡排序

一个最简单的交换排序方法是冒泡排序法（bubble sort），它和气泡从水中往上冒的情况有些类似。具体做法是：先将第 1 个记录的键值和第 1 个记录的键值进行比较，如 a[0]＞a[1]，则将两个记录交换。然后比较第 2 个和第 3 个记录的键值，依同样的方法处理，接着同法处理第 3 个和第 4 个记录，等等，直到第 n-1 个记录和第 n 个记录进行比较交换，这称为一趟冒泡。一趟冒泡后，键值最大的记录传到了第 n 个位置。然后，对前 n-1 个记录进行同样操作，则具有第二大键值的记录被安置在第 n-1 个位置上。重复以上过程，每趟负责排好一个记录，经 n-1 趟后 n 个记录中有 n-1 个记录被排好，相当于全部 n 个记录排好。

例如，设初始记录集为

20 30 10 45 25 22 55 50

则第一趟冒泡的过程为：

20 **30** 10 45 25 22 55 50 //20 与 30 比较，未交换
20 **10** **30** 45 25 22 55 50 //30 与 10 比较，交换
20 10 **30** **45** 25 22 55 50 //30 与 45 比较，未交换
20 10 30 25 **45** 22 55 50 //45 与 25 比较，交换
20 10 30 25 **22** **45** 55 50 //45 与 22 比较，交换
20 10 30 25 22 **45** **55** 50 //45 与 55 比较，未交换
20 10 30 25 22 45 **50** **55** //55 与 50 比较，交换

这样，最大数 55 被交换到了最后。接下来，进行其他(n-2)趟冒泡（只写出每趟的结果）：

10 20 25 22 30 45 ***50*** ***55*** //第 2 趟
10 20 22 25 30 **45** ***50*** ***55*** //第 3 趟
10 20 22 25 **30** ***45*** ***50*** ***55*** //第 4 趟
10 20 22 **25** **30** **45** **50** ***55*** //第 5 趟
10 20 **22** **25** **30** **45** **50** **55** //第 6 趟
10 **20** **22** **25** **30** **45** **50** **55** //第 7 趟
10 **20** **22** **25** **30** **45** **50** **55** //第 8 趟

根据上面的讨论，冒泡排序的算法的框架为（设元素在 a[0]~a[n-1]中）：

```
for (i=n-1; i>0; i--)
  对 a[0]—a[i]进行冒泡;
```

进一步地，“a[0]—a[i]进行冒泡”的实现为：

```
for (j=0; j<i; j++)
  if (a[j]>a[j+1])
```

```
{ t = a[j];
  a[j] = a[j+1];
  a[j+1] = t;
}
```

综合起来，就得到完整的冒泡排序程序：

```
long SortBubble(int a[], long n)
{//将 a[0]~a[n-1]中的数按升序排列
 int t;
 long i, j;

 for (i=n-1; i>0; i--)
    for (j=0; j<i; j++)
      if (a[j]>a[j+1])
      { t = a[j];
        a[j] = a[j+1];
        a[j+1] = t;
      }
 return 0;
}
```

该算法主要由两个嵌套的循环构成，显然是 n^2 数量级的。

§11.3.2 冒泡算法的改进

我们分析一下前面的冒泡排序算法，发现其每次冒泡，都机械地从起点比较到当前终点。事实上，若在某一次冒泡中，从某个位置起，后面都未进行过交换，则表明该位置后面的各元素已排好序。例如，设某趟冒泡后，记录序列为：

序号：	0	1	2	3	4	5	6
记录：	20	10	19	18	30	40	50

下趟冒泡时，当交换完 20 与 18 后（序号 2 与 3），其后就没有再发生交换。这表明，序号 3 以后的元素，已经排好序，因此，下趟冒泡的终点到 2 号位置就可以了。这样，就可以减少冒泡的趟数。据此，我们可以通过控制冒泡次数改进前面的冒泡算法。具体做法是，将每次冒泡的终点由固定改为可调。在冒泡比较中，每交换一对元素，就将这对元素的第一个的序号（设为 k）记下，当一趟冒泡后，k 就是该趟中最后一次的交换位置，下趟比较的终点就可以设置为 k。这样处理后，显然，若在一次冒泡中未交换任何记录，或 k 已退为 0，则算法终止。

```
long SortBubble2(int a[], long n)
```

```
{//将 a[0]~a[n-1]中的数按升序排列
 long k, j, kk;
 int t;
 k=n-1;//用 k 记录上一趟的交换位置
 while (k>0)
 {kk=0;
  for (j=0; j<k; j++)
    if (a[j]>a[j+1])
    {t = a[j];
     a[j] = a[j+1];
     a[j+1] = t;
     kk = j; //记录比较的位置。为避免在循环中改变循环终值，不直接使用 k
    }
    k=kk;
  }//while
  return 0;
}
```

显然，该该算法对已接近排好序的数据集，速度较快。特别地，若数据集已排好序，则该算法只需扫描一次数据集。不过，虽然有了较大改进，但时间复杂度仍然为 $O(n^2)$.

§11.3.3 快速排序*

(一) 基本思想

快速排序（Quick Sort）是交换排序的另一种，实际上它是冒泡排序的另一种改进。快速排序的基本思想是：在待排序的 n 个记录中任取一个记录（例如就取第一个记录），以该记录的键值为标准（称为基准值），将所有记录分为三组，使得第一组中各记录的键值均小于或等于基准值，第二组只含基准值对应的一条记录，第三组中各记录的键值均大于基准值。这个过程称为一趟分割。一趟分割后，基准值的位置就是该值的最终位置（符合排序序列要求的位置）。然后，对所分成的第一和第三组再分别重复上述方法，直到所有记录都排在适当位置为止。

(二) 分割算法

实现一趟分割的算法称为分割算法。我们在第 2 章中已介绍过分割算法，给出了其实现函数

```
long Partition (long a[], long p1, long p2)
```

其功能是对 a[p1]~a[p2]进行分割，返回分割点的序号。具体实现方法就不再重复介绍。

(三) 快速排序递归程序

快速排序的递归实现是直接的，具体程序如下。

```
long SortQuick(ing a[], long p1, long p2)
{// 对 a[p1]~a[p2]中元素排序
  long p;
  if (p1<p2)
  {
    p = Partition(a, p1,p2);
    SortQuick(a, p1, p-1);
    SortQuick(a, p+1, p2);
    return p2-p1+1;
  }
  return 0;
}
```

(四) 快速排序非递归算法

为了写出非递归算法，我们先分析一下快速排序的过程。每次分割后，得到三部分，分割点（第二部分）对应的位置上的元素已排好，不需要再调整，但其他两部分需要继续分割。由于每次只能分割一部分，所以就需要将另一待分割的部分的界限保存起来，当当前分割的部分被分割完后，再分割这部分。注意，由于当前分割的部分也比较大，所以也需要多次分割，这样，就需要不断地分割，不断地保存（另一待分割部分）。那么，待分割部分（的界限）如何保存呢？为简单，可以保存在一个栈中。具体程序如下。

```
long SortQuick2(int a[], long p1, long p2)
{//对 a[p1]~a[p2]中元素排序
  long *s, top;
  long p,i, j;

  s=new long[2*(p2-p1+1)]; //申请栈空间。实际申请量可以是 log2(2*(p2-p1+1))。
  if (s==NULL) return -1;

  top=0;
  top++; s[top]=p1; //待排序元素的起点与终点进栈
  top++; s[top]=p2;

  while (top!=0)
  {//每次从栈中取一对范围值[i,j]，对序号为该范围内的元素进行新的一个回合的分割
```

```
  j = s[top]; top--;
  i = s[top]; top--;
  while (i<j)
  {//连续分割序号在[i,j]内的元素。每次都是将[i,j]一分为三，前后两部分中较大者进栈待分割，
   //较小部分继续在下次循环中分割，而分割点的位置已排好，不需再调整。
   p = Partition(a, i, j); //分割 a[i]~a[j],返回分割点到 p 中。至此，a[p]位置已排好
   if (p-i<j-p)
   {//前部较小，后部进栈,下次先分割前部
    if (j> p+1) //如果待进栈的部分元素个数不足 2，则不进栈
    {top++; s[top] = p+1;
     top++; s[top] = j;
    }
    j = p-1;
   }
   else
   {//后部较小，前部进栈，下次先分割后部
    if (p-1>i) //如果待进栈的部分元素个数不足 2，则不进栈
    {top++; s[top] = i;
     top++; s[top] = p-1;
    }
    i = p+1;
   }
  }// while (i<j)
 }// while (top!=0)
 return p2-p1+1;
}
```

下面给出利用上面的非递归快速排序程序进行排序的过程。假定分割的基准元素是区间中的第一个元素。方括号表示无序区。

初始关键字 [70 73 69 23 93 18 11 68]
第 1 趟分割后 [68 11 69 23 18] 70 [93 73]
第 2 趟分割后 [68 11 69 23 18] 70 73 93
第 3 趟分割后 [18 11 23] 68 69 70 73 93
第 4 趟分割后 11 18 23 68 69 70 73 93

快速排序是不稳定的，请读者自己举出一例说明。

可以证明，快速排序平均比较次数是 O（$n\log_2 n$）。快速排序的记录移动次数小于或等于比较次数，因此快速排序总执行时间为 O（$n\log_2 n$）.

从平均时间性能而言，快速排序最佳。但在最坏情况下，即对几乎是排序好的输入

序列，该算法的效率很低，近似于 O（n^2）。对于原始次序越乱的数据，该算法越有效。另外，该算法对于较大的数据集效果明显。

关于辅助存储的使用，这里使用了一个栈，这个栈的大小取决于分割调用的深度，最多不会超过 n。这里，n 为待排序的数据元素个数。如果每次都选较大的部分进栈，处理较短的部分，则栈内积累的元素就会较少，在这种情况下，对栈的大小的需求为 $O(\log_2 n)$。

§11.4 选择排序

选择排序（Selection sort）的基本思想是：依次选出第 1 小、第 2 小, …的记录，主要有两种形式的选择排序：直接选择排序和堆排序。

§11.4.1 直接选择排序

设待排序的记录集中记录的下标（位置）为 0~n-1，则直接选择排序的基本做法是：首先位置 0~n-1 的记录中选出键值最小的记录，把它与位置 0 的记录交换，然后，位置 1~n-1 的记录中选出键值最小的记录，把它与位置 1 的记录交换，依此类推，最后在位置 n-2 与 n-1 中选出最小者，把它与位置 n-2 的记录交换。至此，所有记录都按升序排列了。这种思想可描述如下：

```
for (i=0; i<n-1; i++)
   在 a[i]--a[n-1]中选择一个最小记录并令其与 a[i]交换位置;
```

而“在 a[i]—a[n-1]中选择一个最小记录并令其与 a[i]交换位置”的实现为：

```
k=i; //将 k 做为一面“旗帜”，指向当前扫描发现的最小记录
for (j=i+1; j<n; j++)
   if (a[k]>a[j]) k = j; //当前“旗帜”比新扫描到的记录大时，旗帜指向新记录
//退出循环后，“旗帜”所指即为扫描范围内的最小者
t = a[k]; //交换
a[k]=a[i];
a[i]=t;
```

由上面的程序片断，就很容易得到完整的直接选择排序程序，此略。

例如，对具有初始输入序列 8，4，3，6，9，2，7 的记录，采用直接选择排序进行排序，过程如下（方括号表示无序区）：

```
[8  4  3  6  9  2  7]
2  [4  3  6  9  8  7]
2  3  [4  6  9  8  7]
```

2 3 4 [6 9 8 7]
2 3 4 6 [9 8 7]
2 3 4 6 7 [8 9]
2 3 4 6 7 8 [9]

该选择排序算法主要部分是两层嵌套的 for 循环，显然其时间复杂性为 O（n^2）。

§11.4.2 堆排序

(一) 基本思想

对直接选择排序来说，为了从 n 个键值中找出最小的，需要进行 n-1 次比较。然后又在 n-1 个键值中找出次最小的。需进行 n-2 次比较，等等。事实上，后面这 n-2 次比较中有许多比较可能在前面的 n-1 次比较中已经做过，但由于第一次比较时这些结果没有保留下来，所以在第二次又重复进行。树形选择排序可以克服这一缺点，它的基本思想是：首先对 n 个记录的键值进行两两比较。然后在其中[n/2]个较小者之间再进行两两比较。如此重复，直至选出最小键值的记录为止。

可用一棵树来表示这一排序过程。树中的 n 个叶子代表待排序记录的键值。叶子上面一层是叶子两两比较的结果，依此类推，树根表示最后选择出来最小关键字。

在选择出最小键值后，将其输出，然后，再在剩余的树结点中，按类似的方法选择最小者，这样，一共经过 n-1 选择，就将全部记录按升序输出了。但不同的是，以前的比较结果，通过树记录下来了，使得后面的选择可以在它们的基础上进行。

如果专门设立树，则造成存储浪费。堆排序是一种巧妙的树形选择排序，它不需要专门设立树。

首先给出几个概念。

堆结构：即前面介绍的顺序二叉树。

堆：若某堆结构中，每个结点的值均小于它的两个儿子的值，则称该堆结构为堆。当然，也可以将该定义中的“小于”改为“大于”（成为另为一种次序的堆）。

那么，如何存储堆结构（顺序二叉树）呢？根据我们在“树形结构”一章中的介绍，最简单的方法是，按顺序二叉树的顺序存储，即将树中各元素，按从上到下（从根的方向到叶子方向），同层中从左到右的次序，存储在一维数组中。据此，我们很容易由树结构写出对应的数组存储，或反过来由数组画出对应的堆结构。图 11-1 给出一个堆和它的数组存储的例子。

在这种存储方法中，父子关系不存储，隐含在元素的序号中。父子关系的确定方法为：若各元素在数组中的序号为 1~n，则对序号为 i 的结点，它的父亲的序号为[i/2]，它的左儿的序号为 2i（当 2i>n 时无左儿），它的右儿的序号为 2i+1（当 2i+1>n 时无右儿）。

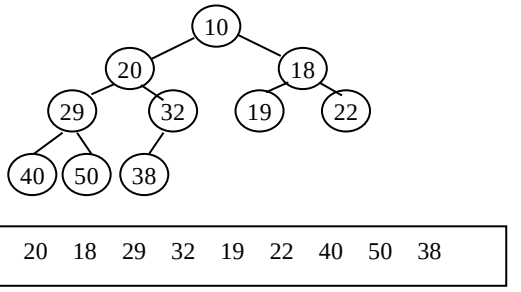


图 11-2 堆示例

堆排序的基本方法可描述为：

1 [建堆] 将原始数据调整为堆；

2 [输出] 输出堆顶元素

3 [堆调整] 将堆结构中剩余元素重新调整为堆

4 [判断] 若全部元素均已输出，则结束，输出次序即为排序次序。否则，转 2;

(二) 堆调整

由上面的讨论知，堆调整是堆排序的关键。

堆调整是针对这样的堆结构：堆顶（根）的左右子树都是堆，但考虑堆顶后，就可能不满足堆的定义了。堆调整就是要将这样一种堆结构调整为堆。

具体的调整方法是，首先将堆顶元素取出，放到临时存储器 x 中，然后，将 x 和堆顶的小儿子（值较小的儿子）比较，如果 x 比小儿子大，则将小儿子放到堆顶，而将 x 放到小儿子的位置，然后，按类似的方法调整以小儿子为根的新堆结构；如果 x 不比小儿子小，则不需要继续调整，将 x 放到堆顶即可;如果某次的（新）堆顶为叶子，则调整也终止。该过程显然是个递归过程，采用递归程序很自然。具体的堆调整递归程序如下。

在程序中要注意数组下标和树结点编号间的转换。树结点编号是从 1 起的连续自然数（父子关系的计算公式是基于这个假定的!），而数组下标为从 0 起的连续自然数，故需要换算。

图 11-1 通过一个例子说明了堆调整过程。

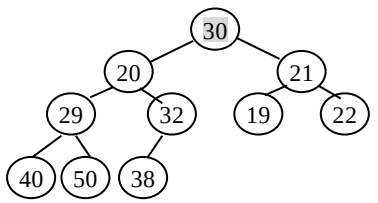
```
void HeapShift2(int a[], long p1, long p2)
{//将 a[p1]~a[p2]调整为堆。a[p1]~a[p2]对应的堆结构中，除去堆顶 a[p1]外，
//其他子树都是堆，堆结构的根的编号为 1。
//每个堆元素占一个数组元素。数组下标从 0 起。
 long j;
 int x;
 if (p1>=p2) return; //只有一个元素(叶子)或无元素

 j=2*(p1+1);//求 p1 的左儿子。注意数组下标和结点编号的换算
```

```
  j--; //将编号换算为下标
  if (j>p2) return ; //p1 无左儿子（即 p1 为叶子）时不需要调整
  if (j+1<=p2)
     if (a[j]>a[j+1]) j++;//令 j 为 p1 的值最小的儿子

  if (a[p1]<a[j]) return ;//堆顶比它的小儿子还小时，不需要继续调整
  x=a[p1]; //堆顶元素与小儿子交换
  a[p1]=a[j];
  a[j]=x;
  HeapShift2(a, j, p2); //继续对以小儿子为顶的新堆结构调整
}
```

现在考虑堆调整的非递归实现。调整的过程，实质上是让根结点 x（不满足堆定义的结点）逐步下滑的过程。每次都是向小儿子方向下滑（同时小儿子上升，即换位）。当 x 到达这样的位置后停止下滑：x 到达叶子位置，或 x 比它的当前小儿子还要小。这个过程可用一个循环实现，具体程序如下。

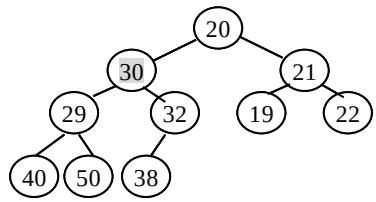
```
void HeapShift(int a[], long p1, long p2)
{//将 a[p1]~a[p2]调整为堆。a[p1]~a[p2]对应的堆结构中，除去堆顶 a[p1]外，
//其他子树都是堆,堆结构的根的编号为 1。
//每个堆元素占一个数组元素。数组下标从 0 起。
  long i, j;
  int x;
  if (p1>=p2) return; //只有一个元素或无元素

  i= p1+1;
  j=2*i;
  i--; j--; //将编号换算为下标
  if (j+1<=p2) //令 j 为 i 的值最小的儿子
     if (a[j]>a[j+1]) j++;

  x=a[i];
  while (j<=p2 && x>a[j])
  {
    a[i]=a[j];
    i =j;
    j = 2*(i+1);
    j--; //将编号换算为下标
    if (j+1<=p2)
```

```
        if (a[j]>a[j+1]) j++; //令 j 为 i 的值最小的儿子
    }
    a[i] = x;
    return ;
}
```



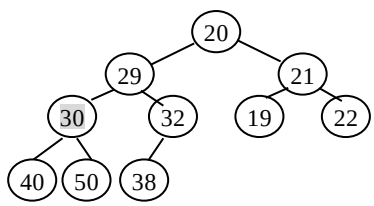
(a) 堆结构

根的两个儿子都是堆，但根不满足堆



(b) 调整

30 与小儿子 20 换位



(b) 调整

30 与小儿子 29 换位。在新位置，
30 比小儿子小，调整完毕

图 11-3 堆调整示例

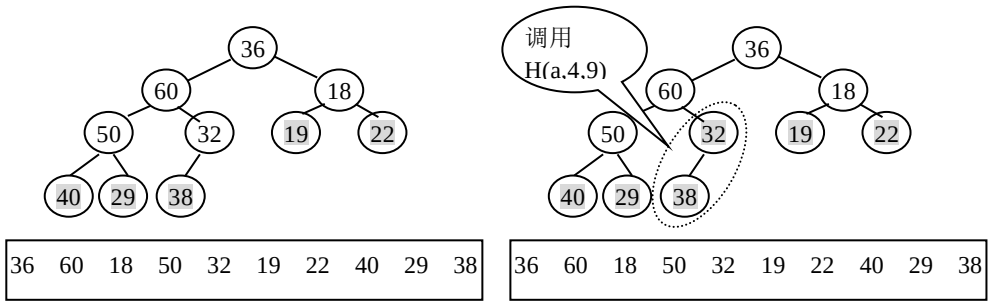
(三) 建堆

我们现在考虑如何将初始数据（初始堆结构）调整为堆。显然，对初始的堆结构，它不满足我们上面介绍的堆调整算法 HeapShift()的条件，所以不能直接使用该算法创建堆。但我们可以自底向上逐步调用 HeapShift()。首先，对以叶子为根的顺序二叉树，由于其只含一个结点，故可认为是堆，因此，以叶子为子树的树，都满足 HeapShift()的条件。我们从最后一个叶子的父亲结点开始，按从下到上、从右到左的次序，对各结点(k, k-1, …, 1)逐步调用 HeapShift()（即对以结点 x 为根的子树调用, x=k, k-1, …1）。显然，用根结点调用 HeapShift()后，整棵树就成了堆。该过程的程序实现为：

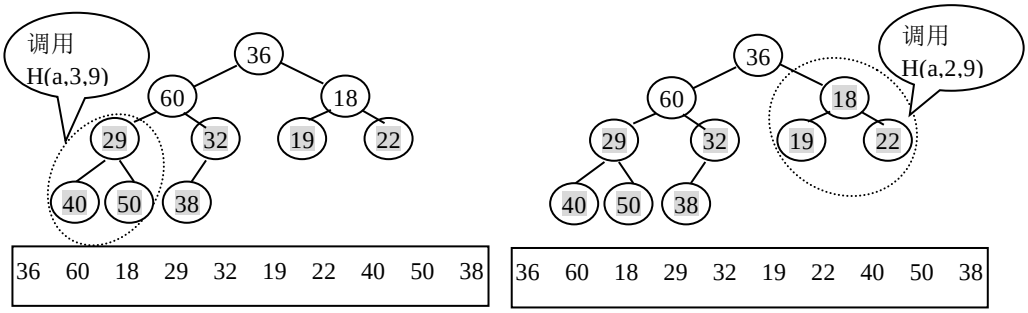
```
for (i=F1; i>=0; i--) HeapShift(a, i, F2);
```

这里，F1 为最后一个结点的父亲的（在数组中的）序号，F2 为以 i 为根的子树中的（层序下）最后一个结点的序号。如果最后结点的编号为 H，则它的父亲的编号为 H/2，因此，若堆结构在数组 a[0]~a[n-1]中，则 F1=(n-1+1)/2 – 1，注意，在该计算式中，进行

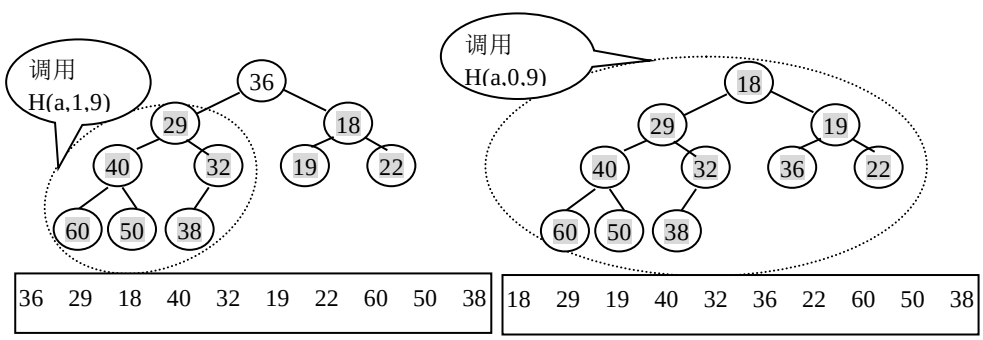
了数组序号与结点编号间的转换。至于 F2，由于不同的 i，F2 也不同，计算比较复杂，我们这里只简单地令它等于整棵树中的最后结点的编号，即 F2=n。尽管 n 不是准确的结束点，但由 HeapShift()的算法知，这样处理是没有问题的。



(a) 一个堆 **(b) 第 1 次调整**



(c) 第 2 次调整 **(d) 第 3 次调整**



(e) 第 4 次调整 **(f) 第 5 次调整**

图 11-4 建堆的例子

图 11-1 给出了一个建堆的例子。图中，用 H()表示堆调整算法 HeapShift()，带黑底的结点表示已经为某子堆中结点，每个图的下方的矩形内为对应堆结构的数组表示。

(四) 堆排序

有了上面的准备，就可以考虑堆排序程序实现了。

堆排序中的建堆问题已解决，这里还需解决的问题有堆顶“输出”和重新“调整”。

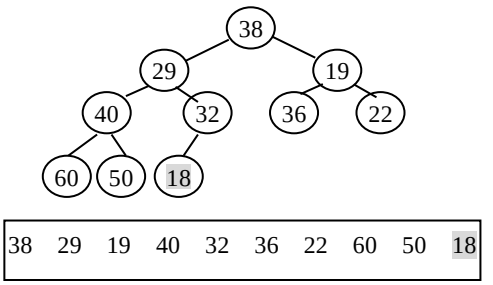
为了节省存储空间，每次“输出”元素，我们并不真正输出，而是将它与当前堆尾元素交换位置，而下次“输出”和“调整”时，我们将当前堆的堆尾视为上次堆尾的上个元素（堆顶不变），然后对新的堆进行“输出”和“调整”。具体过程如下：

```
for (i=n-1; i>0; i--) //每次对 a[0]—a[i]对应的堆进行输出、调整
{
将 a[0]与当前堆尾 a[i] 交换;
  HeapShift(a, 0, i-1); //调整新堆
}
```

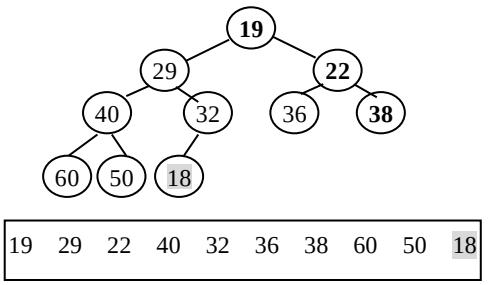
完整的堆排序程序如下：

```
long SortHeap(int a[], long n)
{//对 a[0]~a[n-1]排序
 long i;
 int x;

 for (i=((n-1)+1)/2; i>=1; i--)
    HeapShift(a, i-1, n-1); //建堆,注意数组下标和结点编号的换算

 for (i=n-1; i>0; i--) //每次对 a[0]-a[i]对应的堆进行输出、调整
 {
   x = a[i];
   a[i] = a[0];
   a[0] = x;
   HeapShift(a, 0, i-1); //调整新堆
 }
 return 0;
}
```

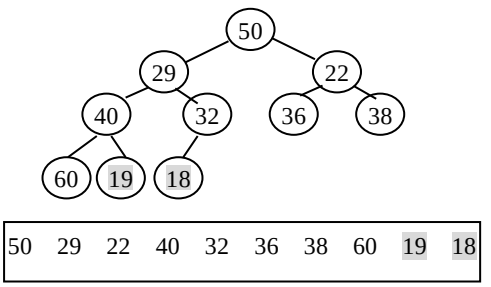
图 11-1 给出了一个关于上列程序运行过程的例子。该图针对的原始数据是图 11-1 建堆的原始数据，所以程序的建堆过程如图 11-1 所示，图 11-1 展示的是程序中的第二个 for 循环的运行过程。图中，带黑底的结点表示已排好序，即已不属于当前堆结构。粗体字结点表示在堆调整过程中移动过的结点。



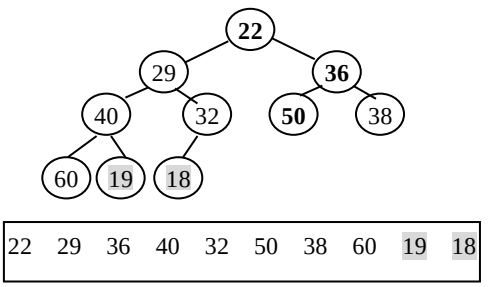
(a) 18 与 38 换位(在图 11-4 (f)的基础上)



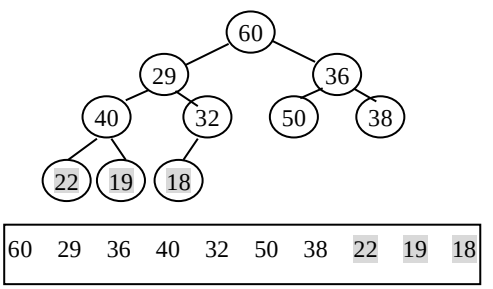
(b) 调整 a[0]~a[8]



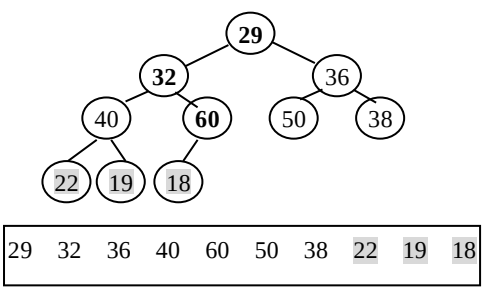
(c) 19 与 50 换位



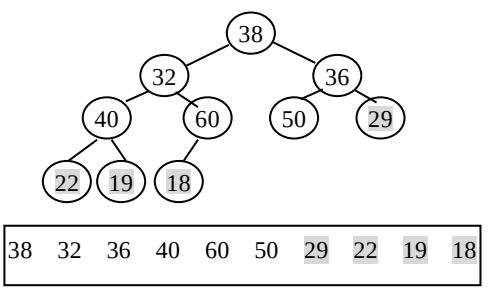
(d) 调整 a[0]~a[7]



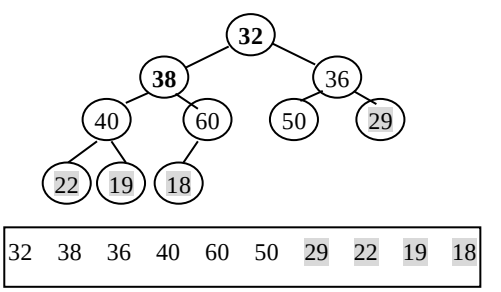
(e) 22 与 60 换位



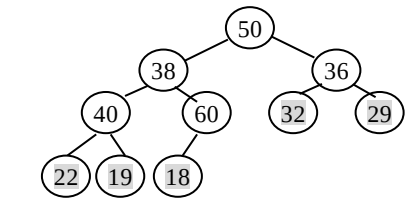
(f) 调整 a[0]~a[6]



(g) 29 与 38 换位

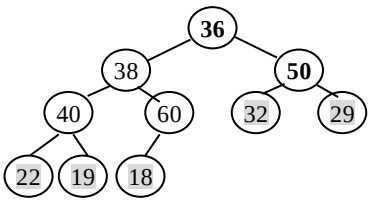


(h) 调整 a[0]~a[5]



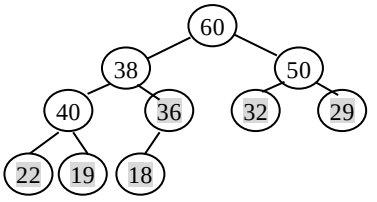
50	38	36	40	60	32	29	22	19	18

(i) 32 与 50 换位



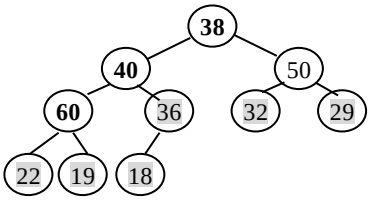
36	38	50	40	60	32	29	22	19	18

(j) 调整 a[0]~a[4]



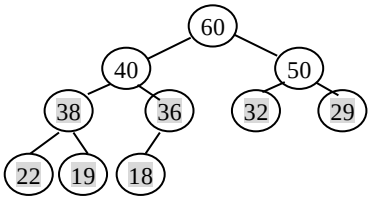
60	38	50	40	36	32	29	22	19	18

(k) 36 与 60 换位



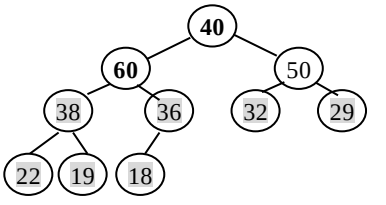
38	40	50	60	36	32	29	22	19	18

(l) 调整 a[0]~a[3]



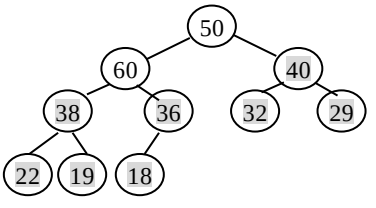
60	40	50	38	36	32	29	22	19	18

(m) 38 与 60 换位



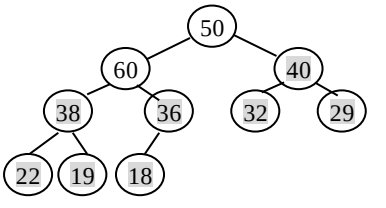
40	60	50	38	36	32	29	22	19	18

(n) 调整 a[0]~a[2]



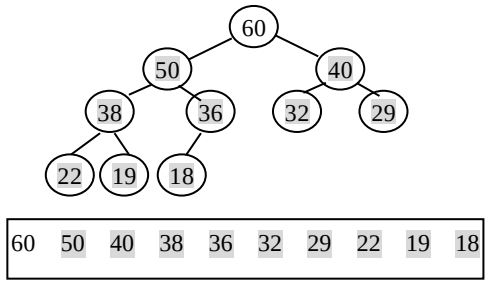
50	60	40	38	36	32	29	22	19	18

(o) 40 与 50 换位



50	60	40	38	36	32	29	22	19	18

(p) 调整 a[0]~a[1]



(q) 50 与 60 换位，完成

图 11-5 堆排序算法运行的例子

简单分析一下堆排序算法。首先考察堆调整算法 HeapShift()，由于算法每次都是从根起向叶子方向“下沉”结点，而且没有回溯，所以下沉的最大次数不超过树高度。由于 n 个结点的顺序二叉树的高度为$[\log_2 n]+1$,故堆调整算法的时间复杂度为 $O(\log_2 n)$。现在考察堆排序算法 SortHeap()，其主要时间花在建堆和下面的“输出”—“调整”（第二个 for 循环）上，它们都是在 for 循环中调用 HeapShift()，其中，建堆的 for 循环 n/2 次，时间复杂度为 $O(n\log_2 n)$而“输出”—“调整”循环次数为 n-1，时间复杂度也为 $O(n\log_2 n)$，故总的时间复杂度为 $O(n\log_2 n)$，这也是堆排序的最坏情况的时间复杂度。对最好情况，堆调整算法的处理时间比$[\log_2 n]+1$ 少一些，但没有数量级的变化。相对于快速排序来说，堆排序最大的优点是不需要附加空间。另外，堆排序是不稳定的。

§11.5 归并排序

归并排序是一种基于合并有序段的排序方法，即将若干有序段逐步合并，最后合并为一个有序段。对初始记录集，将每个记录视为一个独立的有序段，然后在此基础上逐步合并。因此，归并排序的基础是合并，下面首先讨论合并算法。

§11.5.1 二路合并

下面讨论两个有序段的合并。我们把两个有序段（设为升序）中的记录分别记为

a_s，…，a_m 和 a_{m+1}，…，a_n。

两个有序段的合并就是形成第三个有序段（升序）：

b_s，…，b_n

设 i, j 分别表示两个有序段中记录的下标，则两个有序段的合并可按下列方法进行：

①当 i≤m，且 j≤n 时，比较 a[i]和 a[j]键值的大小，若 a[i]<=a[j]，则将 a[i]顺序送入另一数组 b[]，同时使 i 加 1；若 a[i]>a[j]，则将 a[j]顺序送入另一数组 b[]，同时使 j

加 1。

②当 i＞m 或 j＞n 时，将剩余部分照抄到 b[]的末尾。

下面是具体的程序。

```
void Merge2Sorted(int a[], long s, long m, long n, int b[])
{//二路合并,将有序段 a[s]~a[m]和 a[m+1]~a[n]合并到 b[0]~b[n-s]
  long i, j, k;

  i=s; j=m+1; k=0;
  while (i<=m && j<=n)
     if (a[i] <= a[j]) b[k++]=a[i++];
     else b[k++]=a[j++];
  while (i<=m) b[k++]=a[i++];
  while (j<=n) b[k++]=a[j++];
  return;
}
```

对于该算法，由于每次比较就输出一个元素（输出到 b 中），所以，全部合并后需要比较（或复制）的次数为两个有序段的长度之和，即 n-s+1。

由于这里是每次合并两个有序段，所以称为二路合并。显然，也可以同时合并 n 个有序段（n 路合并，n>=2），具体算法实现留作练习。

§11.5.2 多段 2 路合并

设某记录集已分段有序，除最后一个段外，其他各段长度均相等。现考虑如何将它们中从头起的每个两两相邻的段，都分别合并为一个有序段。经过这种合并后，形成一个新的分段有序文件，其每个段（最后段可以例外）的长度都翻了倍，而段数为原来的 1/2（或 1/2 加 1）。这种操作称为多段合并。由于每次都是合并两个段，所以称多段 2 路合并。显然，也可以有多段 n 路合并。

```
void MMergeSorted(int a[], long p1,long p2, long len, int b[])
{//多段 2 路归并：a[p1]-a[p2]中，每 len 个元素为一个有序段，将从头起的每个连续的
//两段分别合并为一个有序段（即段 1 与段 2，段 3 与段 4，…分别合并）存入 b[]
  long i, j, k;

  i=p1;
  k=0;
  while (i+2*len-1 <= p2)
  {//当从 i 起的后面有完整的两段时，合并之
    Merge2Sorted(a,i, i+len-1, i+2*len-1, b+k);
    i += 2*len;
```

```
    k+=2*len;
  }
  if (i+len <= p2) //剩两段，但最后的段长不足 len
     Merge2Sorted(a, i, i+len-1, p2, b+k );
  else //剩一段,或不剩
     for (j=i; j<=p2; j++) b[k++]=a[j];
}
```

显然，该算法的执行时间近似为 s*p/2，即 O(s*m)。这里，p 为段数，而 s 为段长。

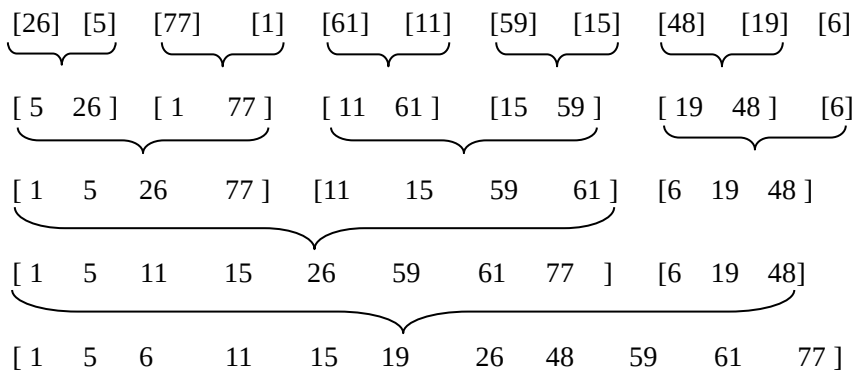
§11.5.3 二路归并排序

有了上面的几个算法的支持就可以容易地实现归并排序了。这种归并排序称为二路归并排序。二路归并排序的基本思想是：如果序列中有 n 个记录，可以先把它看成 n 个段，每个段中只包含一个记录，因而都是排好序的。二路归并排序是先将从头起的两连相邻的段合并（若段数为奇数，则最后一段照抄，下同），完成一趟归并，得到「n/2]个较大的有序段，每个段包含 2 个记录（最后一段长度可能不足）。然后，再进行第 2 趟归并，这趟是在上趟的基础上进行，归并后，得到「「n/2]/2]个有序段，每个段包含 4 个记录（最后一段长度可能不足）。如此反复多趟，直到最后合并成一个有序段（包含了全部记录），排序即告完成。

例如，设初始输入序列为

26，5，77，1，61，11，59，15，48，15，48，19 , 6

则对其采用归并排序法进行排序，其归并过程如下（方括号表示有序段，每行为一趟归并的结果）：



应用 MMergeSorted()，可以写出对包含 n 个记录的序列进行若干趟二路归并排序算法。具体程序如下。

```
void SortMerge(int a[], long p1, long p2, int b[])
{//2 路归并排序，将 a[p1]~a[p2]中的元素排序，结果在 b[0]~b[p2-p1]中
  long len;
```

```
  len=1;
  while (len<p2-p1+1)
  {
    MMergeSorted(a, p1, p2, len, b);
          //将 a[p1]~a[p2]中的长度为 len 的相邻有序段两两合并到 b[0]~b[p2-p1]
    len = len*2;
    MMergeSorted(b, 0,p2-p1, len, a+p1);
          //将 b[0]~a[p2-p1]中的长度为 len 的相邻有序段两两合并到 a[p1]~a[p2]
  }
}
```

粗略地分析一下该算法。设记录个数为 n，由于每趟都是两两合并（2 路合并），所以，每趟归并后段长度都是上次段长的 2 倍，特别地，最后一趟后，段长变为 n，所以，归并的趟数 m 满足：$n/2^m=1$，从而 $m=\log_2 n$. 由于每趟归并时，每个记录恰好被输出一次，所以，每趟归并的时间为 n，从而，完成全部归并的总时间代价为 $O(n\log_2 n)$。至于空间复杂度，显然需要附加一倍的存储开销。从这方面看，归并排序并不优秀，但是，归并排序很适合用在外排序中。另外，二路归并排序是稳定的。

从另一角度看，二路归并排序的过程是一棵二叉树。初始的 n 个记录做为 n 个叶子，第 1 趟归并是将这 n 个叶子两两归并，每两个结点归并后的结果，都分别用一个新结点表示，该新结点作为所合并的两个结点的父亲。下一趟是在上一趟的基础上按类似方法生成新结点，这样，最后一趟归并后，形成根结点。从这里也看出，2 路归并排序速度很快，相当于二叉树的“收缩”（树结点的生长的逆方向）。

在 n 较大时，归并排序所需时间较堆排序省。

§11.6 外排排简介

在许多实际应用系统中，经常遇到要对数据文件中的记录进行排序处理。由于文件中的记录往往很多、信息量庞大，整个文件所占据的存储单元远远超过一台计算机的内存容量。因此，无法把整个文件读入内存中进行排序。于是，就有必要研究适合于处理大型数据文件的排序技术。通常，这种排序往往需要借助于具有更大容量的外存设备才能完成。相对于仅用内存进行排序（又称为内排序）而言，这种排序方法就叫做外排序。

在实际应用中，由于使用的外存设备不同，通常又可以分为磁盘文件排序和磁带文件排序两大类。磁带排序和磁盘排序的基本步骤相类似，它们的主要不同之处在于初始归并段在外存贮介质中的分布方式，磁盘是直接存取设备，磁带是顺序式存取设备。

外部排序基本上由两个相对独立的阶段组成。首先，按可用内存大小，将外存上含 n 个记录的文件分成若干长度为 h 的子文件，依次读入内存并利用有效的内部排序方法

对它们进行排序，并将排序后得到的有序子文件重新写入外存，通称这些有序子文件为**归并段**或**顺串**。然后，对这些归并段进行逐趟归并，使归并段逐渐由小到大，直至得到整个有序文件为止。

对顺串的归并，最简单的是运用类似于内部排序中的归并算法。与内排序中的归并算法主要不同是，随着归并的进行，有序段（顺串）不断扩大，以至于内存缓冲区不能将它们完全容纳，需要分段从外存读入。同样在输出时，也需要分段输出到外存。

本章小结（注意：所有的"本章小节"都用楷体，但英文仍用 Times New Noman）

排序不仅是线性表的重要的操作，也代表着一类重要的算法设计方法。

本章共介绍了四种内排序算法，它们是：插入排序，交换排序（冒泡排序、快速排序）、选择排序（直接选择排序、堆排序）和归并排序。这几种均为基于比较的排序，即排序过程的实现主要靠关键字的关系大小比较。

另有一类排序算法，它们的（排序的）进行，不主要依赖关键字的关系比较。这类算法典型的有基数排序、桶排序、计数排序等，它们均可在线性时间复杂度内完成。但是，这些方法的适用范围一般都有限。这几种算法我们在这里没有介绍，有兴趣的读者可以参阅有关资料。

一般说来，比较简单的排序（如直接插入、直接选择、冒泡排序等）所需要的时间代价大，为 O（n^2），但在某些特殊情况下可能取得很好的效果。例如，当初始序列已排序或接近排序的情况下，直接插入排排序和改进的冒泡排序的时间代价可能达到 O（n）量级。

对一些较复杂的排序算法，如快速排序、堆排序、归并排序等，平均情况下的时间复杂度为 $O(n\log_2 n)$。比较起来，堆排序和归并排序对各种情况的时间复杂度差别不大，但对快速排序，其在最坏情况下退步为 $O(n^2)$，发生在初始记录已排好序的时侯。事实上，快速排序对初始记录排列很“乱”的情况更有效。

对基于比较的排序，复杂度是否还有比 $O(\log_2 n)$更低的算法？回答是否定。有人早已证明，对基于比较的排序，平均状况下的时间复杂度都不会比 $O(\log_2 n)$更低。这就知道我们，如果想寻找更快的排序方法，就不要盯在“比较”上了，必须超越这种思维才可能发现数量级更低的算法。但一个显然的事实是，再低也低不过 O(n)。

快速排序和归并排序都需要较大的辅助空间。对快速排序，如果不加控制（如递归算法），对辅助空间的需求可能达到 O(n)，但若每次对分割后的较大的部分进栈（先分割较小的部分），则平均情况下辅助空间的量为 $O(\log_2 n)$。对归并排序，固定需要与原始数据个数等量的辅助空间。归并排序方法多用于外排序。

在稳定性方面，快速排序、堆排序、直接选择排序等是不稳定的排序，其它排序是稳定的。

外排序的基本方法是归并。归并的思想本来是简单的；但为了减少访问外存的次数，减少外设的台数和输入输出缓冲区的个数，并使主机和外设尽量并行工作，这就给外排

序的讨论增加了难度。

在算法构思方面，值得认真地学习这几种排序算法。同样一个问题，考虑的角度不同，就导致不同的方法，例如，插入观点导致插入排序，交换观点导致冒泡排序、分割观点导致快速排序、选拔观点导致直接选择排序、树形选拔观点导致堆排序，合并观点导致归并排序。

习 题

1．对于给定的一组键值：83，40，63，13，84，35，96，57，39，79，61，15，分别画出应用直接插入排序、直接选择排序、快速排序、归并排序对上述序列排序中各趟的结果。

2．对上题给出的关键字，画出堆排序中的：a)建堆期间每次堆调整后的结果（树）；b)每次“输出-交换”后的结果（树）。

3．本章介绍的各排序方法中哪些是稳定的？哪些是不稳定的？对不稳定的举例说明。

4．设计一个针对单链表的直接选择排序算法。

5．插入排序中找插入位置的操作可以通过二分查找的方法来实现。试写一个针对这种改进的插入排序算法。

6．一个线性表中的元素为正整数或负整数。设计一个算法，将正整数和负整数分开，使线性表前一半为负整数，后一半为正整数。不要求对这些元素排序，但要求尽量减少交换次数。

7．冒泡也可以向前“冒”，就每次交换都从记录集的最后开始，将最小记录“冒”到最前面。请写出这种冒泡的子程序。

8．冒泡也可以从前后两端“同时”（实际实现时可以是交替）向中间“冒”，即两边“夹击”，将两边各自最大的记录夹在中间，然后再将它们中最大者置换到最后，完成一趟冒泡。请写出这种冒泡的子程序。

9．已知（k_1，k_2，…，k_n）是堆，试写一个算法将（k_1，k_2，…，k_n，k_{n+1}）调整为堆。按此思想写一个从空堆开始一个一个添入元素的建堆算法（提示：增加一个 k_{n+1} 后应从叶子方向调整）。

10．设计一个 n 路合并子程序。这里 n>=2，且要求 n 的具体值做为子程序参数给出给出。