第5章 排 序

排序是数据结构的一种重要运算。本章5.1节～5.6节介绍内排序的各种方法，5.7节介绍外排序方法。此外，堆排序也是一种典型的选择排序，有关堆排序算法，将在第8章介绍。

5.1 基本概念

在讨论排序的概念之前，首先引入排序码的概念。所谓排序码是结点中的一个或多个字段，其值作为排序运算中的依据。排序码可以是关键码，这时排序即是按关键码对文件进行排序；排序码也可以不是关键码，这时可能有多个结点的排序码具有相同的值，因而排序结果就可能不唯一。排序码的数据类型可以是整数，也可以是实数、字符串，乃至复杂的组合数据类型。

习惯上，在排序中将结点称为记录，将一系列结点构成的线性表称为文件。在本书中后续涉及排序时，都要使用记录和文件这两个概念，请读者将它们和外存中的记录、文件等概念加以区别。

排序(sorting)又称分类。假定具有 n 个记录$\{A_1, A_2, \cdots, A_n\}$的文件，每个记录有一个排序码 K_i，$\{K_1, K_2, \cdots, K_n\}$是相应的排序码的集合。排序运算就是将上述文件中的记录按排序码非递减(或非递增)的次序排列成有序序列。

由于各种待排序的文件中，记录的大小和数量不等，有的文件记录本身较大、数量很多，有的文件的记录本身较小、数量较少。对于较小的文件，可以一次将文件全部调入内存进行排序处理；而对于很大的文件，无法一次全部调入内存进行排序处理，因而在排序过程中需要涉及内外存之间的数据交换。在排序过程中，文件全部放在内存处理的排序算法称为“内排序”；在排序过程中，不仅需要使用内存，而且还要使用外存的排序算法称为“外排序”。按照所采用的策略的不同，排序方法可以分为5种类型，即插入排序、选择排序、交换排序、分配排序、归并排序。当然，由于关注的重点不同，一个具体的排序算法既可以看成是这种排序，也可以看成是那种排序，也就是说，一个具体的排序算法究竟应该属于上述5种类型中的哪一种并不是唯一的。

在待排序的文件中，可能存在着多个具有相同排序码的记录。如果一个排序算法对于任意具有相同排序码的多个记录在排序之后，这些具有相同排序码的记录的相对次序仍然保持不变，则称该排序算法为“稳定的”；否则称该排序算法是“不稳定的”。排序的方法很多，就其性能而言，很难说哪一种算法是最好的。每一种算法都有各自的优缺点，适合于不同的应用领域。有两个评价排序算法性能的重要指标：一个是算法执行时所需的时间；另一个是算法执行时所需的内存空间。其中，时间开销是衡量一个排序算法好坏的最重要的性能指标。为便于分析，排序算法的时间开销通常用算法执行中的比较次数和

记录移动次数表示。许多排序算法执行排序时所耗费的时间不仅与算法本身有关，而且与待排序文件的记录顺序有关，衡量这些排序算法的性能可以采用最大执行时间和平均执行时间。

为讨论方便起见，假定排序要求都是按非递减序进行排序的。

本章后续各节将分别讨论插入排序、选择排序、交换排序、分配排序、归并排序和外排序，给出典型排序算法的面向对象的具体实现描述。同时，对主要排序算法的性能进行必要的分析和讨论。执行排序算法所需要的空间量一般都不大，对算法性能好坏的影响并不大，我们只给出结果，而不加以讨论。

5.2 插入排序

插入排序的基本思想是：每次选择待排序的记录序列的第1个记录，按照排序码的大小将其插入已排序的记录序列中的适当位置，直到所有记录全部排序完毕。

5.2.1 直接插入排序

直接插入排序是一种最简单的排序方法，整个排序过程为：先将第1个记录视为一个有序的记录序列，然后从第2个记录开始，依次将未排序的记录插入这个有序的记录序列中，直到整个文件中的全部记录排序完毕。在排序过程中，前面的记录序列是已经排好序的，而后面的记录序列有待排序处理。

例5-1 假设有5个元素构成的数组，其排序码依次为50、20、40、75、35，整个数组完成直接插入排序的过程如图5.1所示。

初始：	[50] 20 40 75 35	从50开始
第1趟扫描后：	[20] [50] 40 75 35	将20插入位置0；50后移到位置1
第2趟扫描后：	[20] [40] [50] 75 35	将40插入位置1；50后移到位置2
第3趟扫描后：	[20] [40] [50] [75] 35	记录75位置不变
第4趟扫描后：	[20] [35] [40] [50] [75]	将35插入位置1；后面各记录右移

图5.1 直接插入排序的过程

下面给出直接插入排序的函数DirectInsertionSort，该函数的参数是存放被排序文件的数组A和文件中所包含的记录个数（即数组A的大小）n。考察第i（$1 \leqslant i \leqslant n-1$）遍的情况，子文件A[0]到A[$i-1$]已按非递减序排列，这一遍将记录A[$i$]插入到子文件A[0]到A[$i-1$]中。将A[$i$]向子文件A[0]到A[$i-1$]的前部移动，移动A[$i$]前，将A[$i$]的排序码与记录A[$i-1$]、A[$i-2$]等进行比较。在小于或等于A[$i$]的排序码的第1个记录A[$j$]或到达第1个记录A[0]处停止扫描。当A[$i$]往前移动时，要将子文件中每个遇到的记录A[$j$]后移一个位置。当找到A[$i$]的正确位置$j$后，将其插入到位置$j$。假设文件的排序码是ElementType类型的，key是它的一个排序码。

sort.h文件如下：

```
typedef int ElementType;
struct forSort
{
    ElementType key;
};
typedef struct forSort ForSort;

void InitForSort(ForSort * FS, int a)
{
    FS->key=a;
}
```

算法 5.1 直接插入排序。

```
void DirectInsertionSort(ForSort A[], int n)
{
    int i,j;
    ForSort temp;

    for(i=1; i<n; i++)
    {
        j=i;
        temp=A[i];
        while(j >0 && temp.key<A[j-1].key)
        {
            A[j]=A[j-1];
            j--;
        }
        A[j]=temp;
    }
}
```

将记录 A[0],A[1],…,A[$n-1$]采用直接插入排序,需要进行 $n-1$ 趟扫描。一般地,在第 i 趟,插入发生在 A[0]到 A[i],平均大约需要 $i/2$ 次比较。总的比较次数为

$$1/2+2/2+\cdots+(n-1)/2=n(n-1)/4$$

显然,插入排序不需要交换。按比较次数衡量,算法的时间复杂度为 $O(n^2)$。最好的情况是待排序文件的记录已经是排好序的,在第 i 趟,插入发生在 A[i]处,每趟只需一次比较,总的比较次数为 $n-1$ 次,算法的时间复杂度为 $O(n)$。最坏的情况是待排序文件的记录已按非递增序排序,每次插入发生在 A[0]处,第 i 趟需要进行 i 次比较,总的比较次数为 $n(n-1)/2$,算法的时间复杂度为 $O(n^2)$。

直接插入排序是稳定的。

5.2.2 折半插入排序

将直接插入排序中寻找 A[i]的插入位置的方法改为采用折半比较,便得到折半插入排序算法。在处理 A[i]时,A[0],A[1],…,A[$i-1$]已经按排序码排好序。所谓折半比

较，就是在插入 A[i]时，取 A$\left[\left\lfloor\frac{i-1}{2}\right\rfloor\right]$的排序码与 A[$i$]的排序码进行比较，如果 A[$i$]的排序码小于 A$\left[\left\lfloor\frac{i-1}{2}\right\rfloor\right]$的排序码，说明 A[$i$]只能插入 A[0]到 A$\left[\left\lfloor\frac{i-1}{2}\right\rfloor\right]$之间，故可以在 A[0]到 A$\left[\left\lfloor\frac{i-1}{2}\right\rfloor-1\right]$之间继续使用折半比较；否则 A[$i$]只能插入 A$\left[\left\lfloor\frac{i-1}{2}\right\rfloor\right]$到 A[$i-1$]之间，故可以在 A$\left[\left\lfloor\frac{i-1}{2}\right\rfloor+1\right]$到 A[$i-1$]之间继续使用折半比较。如此反复，直到最后能够确定插入的位置为止。一般地，在 A[k]和 A[r]之间采用折半，其中间结点为 A$\left[\left\lfloor\frac{k+r}{2}\right\rfloor\right]$，经过一次比较，可以排除一半的记录，把可能插入的区间减少了一半，故称折半。执行折半插入排序的前提是文件记录必须按顺序存储。

例 5-2 将例 5-1 中的 5 个记录采用折半插入排序，在前 4 个记录已经排序的基础上，插入最后一个记录的比较过程如图 5.2 所示。

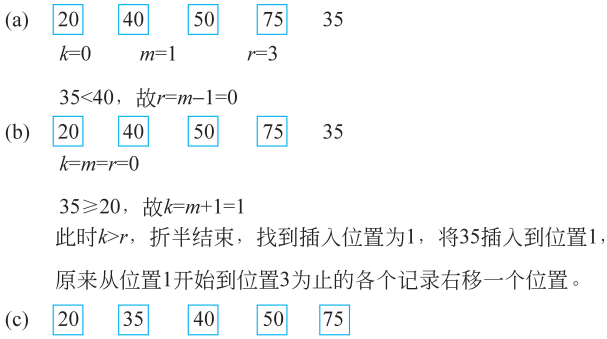

图 5.2 折半查找过程

下面给出折半插入排序的函数 BinaryInsertionSort，该函数的参数是存放被排序文件的数组 A 和文件中所包含的记录个数（即数组 A 的大小）n。

算法 5.2 折半插入排序。

```
/*用折半插入排序法对文件 A[0],A[1],…,A[n-1]排序*/
void BinaryInsertionSort(ForSort A[],int n)
{
    int i,k,r;
    ForSort temp;

    for(i=1;i<n;i++)
    {
        temp=A[i]; /*采用折半法在已排序的子文件 A[0]~A[i-1]找 A[i]的插入位置*/
        k=0;
        r=i-1;
        while(k<=r)
        {
            int m;
```

```
            m=(k+r)/2;
            if(temp.key<A[m].key)
                r=m-1;                    /*在前半部分继续找插入位置*/
            else
                k=m+1;                    /*在后半部分继续找插入位置*/
        }

        /*找到插入位置为k,先将A[k]~A[i-1]右移一个位置*/
        for(r=i;r>k;r--)
            A[r]=A[r-1];
        A[k]=temp;                        /*将temp插入*/
    }
}
```

在算法中,采用 $k>r$ 来控制折半的结束,此时,k 就是 A[i]应该插入的位置。在判别下一步应该是在前半部分还是在后半部分继续使用折半时,采用 temp.key<A[m].key 作为依据,保证了排序过程是稳定的。

使用折半插入排序时,需进行的比较次数与记录的初始状态无关,仅依赖于记录的个数。在插入第 i 个记录时,如果 $i=2^j(0\leqslant j\leqslant\lfloor\log_2 n\rfloor)$,则无论排序码取什么值,都需要恰好经过 $j=\log_2 i$ 次比较才能确定应该插入的位置;如果 $2^j<i\leqslant 2^{j+1}$,则需要的比较次数大约为 $j+1$。因此,将 n 个记录用折半插入排序所要进行的总的比较次数约为(为推导简便起见,假设 $n=2^k$):

$$\begin{aligned}\sum_{i=1}^{n}\lceil\log_2 i\rceil &= 0+1+2+2+3+3+3+3+\cdots+k+\cdots+k\\ &= 2^0+2^1+2^2+\cdots+2^{k-1}+2^1+\cdots 2^{k-1}+2^2+\cdots+\\ &\quad 2^{k-1}+\cdots+2^{k-2}+2^{k-1}+2^{k-1}\\ &= \sum_{i=1}^{k}\sum_{j=i}^{k}2^{j-1}\\ &= \sum_{i=1}^{k}(2^k-2^{i-1})\\ &= k\cdot 2^k-2^k+1\\ &= n\log_2 n-n+1\\ &\approx n\log_2 n\end{aligned}$$

即折半插入排序的时间复杂度为 $O(n\log_2 n)$。当 n 较大时,显然要比直接插入排序的最大比较次数少得多,但是大于直接插入排序的最小比较次数。算法 BinaryInsertionSort 的记录移动次数与算法 DirectInsertionSort 相同,最坏情况是待排序文件中的记录已按非递增序排好序,此时总的移动次数为 $n(n-1)/2$;最好情况是待排序文件中的记录已按非递减序排好序,此时总的移动次数为 $2(n-1)$。

5.2.3 Shell 排序

Shell 排序法又称希尔排序法、缩小增量排序法。Shell 排序的基本思想是:先选定

一个整数 $s_1<n$，把待排序文件中的所有记录分成 s_1 个组，所有距离为 s_1 的记录分在同一组内，并对每一组内的记录进行排序。然后，取 $s_2<s_1$，重复上述分组和排序的工作。当到达 $s_i=1$ 时，所有记录在同一个组内排好序。

各组内的排序通常采用直接插入法。由于开始时 s 的取值较大，每组内记录数较少，所以排序比较快。随着 s_i 的不断缩小，每组内的记录数逐步增多，但由于已经按 s_{i-1} 排好序，因此排序速度也比较快。

例 5-3 设某文件中待排序记录的排序码分别为 28、13、72、85、39、41、6、20。用 Shell 排序法对该文件进行排序，取 $s_1=\frac{n}{2}=4$，$s_{i+1}=\left\lfloor\frac{s_i}{2}\right\rfloor$，排序过程如图 5.3 所示。

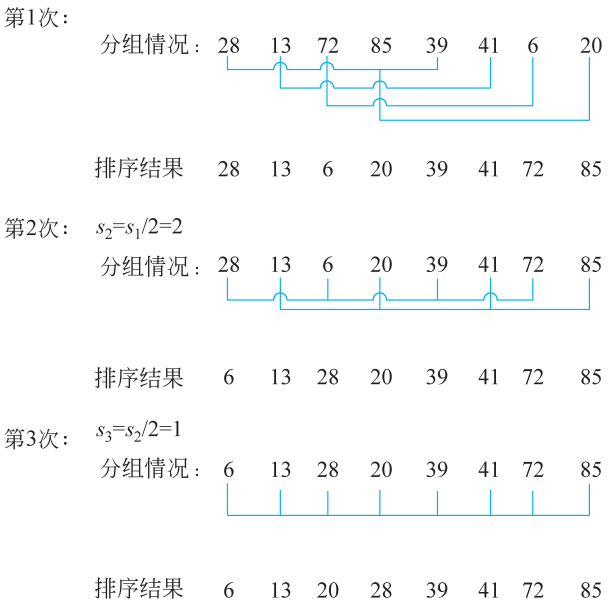

图 5.3 Shell 排序过程

下面给出 Shell 排序的函数 ShellSort，该函数的参数是存放被排序文件的数组 A、文件中所包含的记录个数（即数组 A 的大小）n、首次分组间隔（增量）s。

算法 5.3 Shell 排序。

```
/*用 Shell 排序对文件 A[0],A[1],…,A[n-1]排序,初始增量为 s*/
void ShellSort(ForSort A[],int n,int s)
{
    int i,j,k;
    ForSort temp;
    /*分组排序,初始增量为 s,每循环一次增量减半,直到增量为 0 时结束*/
    for(k=s;k>0;k>>=1)
    {
        for(i=k;i<n;i++)                    /*分组排序*/
        {
            temp=A[i];
            j=i-k;
```

```
                /*组内排序,将 temp 直接插入组内合适的记录位置*/
                while(j>=0&&temp.key<A[j].key)
                {
                    A[j+k]=A[j];
                    j-=k;
                }
                A[j+k]=temp;
            }
        }
    }
```

一般而言,Shell 排序算法的速度要快于直接插入排序。具体分析比较复杂,请参见 Knuth 所著的《计算机程序设计技巧第 3 卷》。Shell 排序的平均比较次数和平均移动次数为 $O(n^{1.3})$。在 Shell 排序算法中,对各组内的排序,也可以采用直接插入法或其他排序算法。

Shell 排序是不稳定的。

5.3 选择排序

选择排序的基本思想是:每次从待排序的记录中选出排序码最小的记录,再在剩下的记录中选出次最小的记录,重复这个选择过程,直到完成全部排序。本节介绍直接选择排序和树形选择排序。

5.3.1 直接选择排序

基本思想:每次从待排序的记录中选出排序码最小的记录,顺序放在已排序的记录序列的最后,直到完成全部排序。

例 5-4 设某文件中待排序的记录的排序码分别为 42、32、31、12、25、11、43、10、8。用直接选择排序的排序过程如图 5.4 所示。

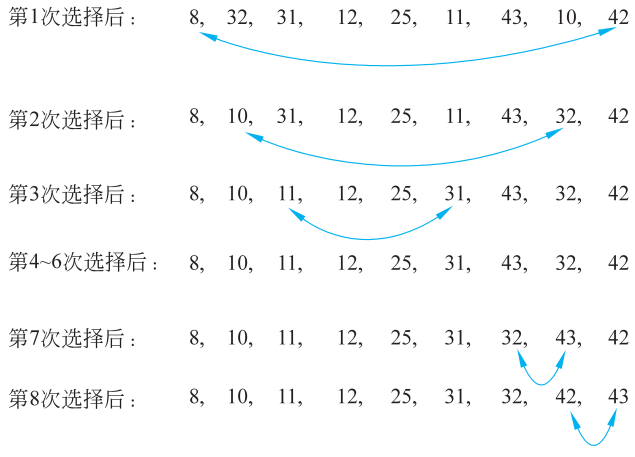

图 5.4 直接选择排序

算法 5.4 直接选择排序。

```
void DirectSelectSort(ForSort A[], int n)
{
    int i, j,k;
    ForSort temp;

    for (i=0; i<n-1; i++)
    {
        k=i;
        for(j=i+1;j<n;j++)
            if(A[j].key<A[k].key)
                k=j;
        if(i!=k)
        {
            temp=A[k];
            A[k]=A[i];
            A[i]=temp;
        }
    }
}
```

直接选择排序的比较次数与排序码的初始顺序无关，总的比较次数为

$$\sum_{i=1}^{n-1}(n-i)=\sum_{i=2}^{n}(i-1)=\frac{n(n-1)}{2}$$

当初始文件已排序时，移动次数最少为 0 次，最多为 $3(n-1)$ 次，即对应每趟选择后都要执行交换的情况。选择排序是不稳定的。

5.3.2 树形选择排序

树形选择排序又称为竞赛树排序或胜者树。

基本思想：把 n 个排序码两两进行比较，取出 $\lceil \frac{n}{2} \rceil$ 个较小的排序码作为第 1 步比较的结果保存下来，再把 $\lceil \frac{n}{2} \rceil$ 个排序码两两进行比较，重复上述过程，一直比较出最小的排序码为止。

例 5-5 设某文件中待排序记录的排序码分别为 40、35、30、13、24、15、42、14、17。用树形选择排序的排序过程如图 5.5 所示。

重复上述选择过程，共进行 8 次选择后完成整个文件的排序码排序，图 5.6 中的∞代表比 n 个记录的排序码都大的整数。

n 个记录的排序码用树形选择排序，总的比较次数为

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

移动次数不超过比较次数，总的时间开销为 $O(n\log_2 n)$。此外，存储开销方面需要增加相当多的存储空间保留中间结果。第 8 章将介绍时间开销为 $O(n\log_2 n)$，空间开销仅为 $O(1)$ 的另一树形选择排序方法（堆排序）。

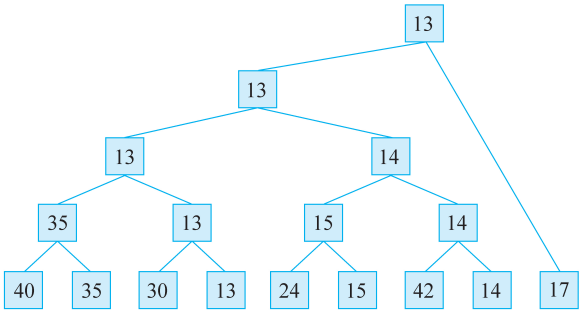

图 5.5 第一次树形选择排序选出最小排序码 13

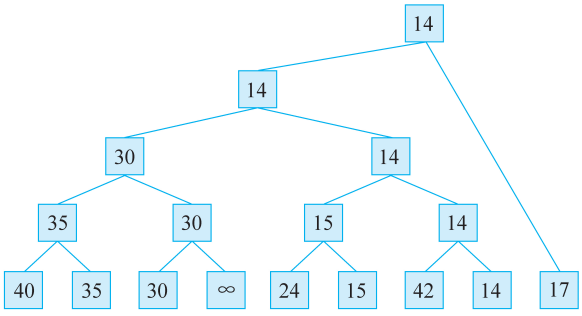

图 5.6 第二次树形选择排序选出最小排序码 14

5.4 交换排序

交换排序的基本思想是：每次将待排序文件中的两个记录的排序码进行比较，如果不满足排序要求，则交换这两个记录在文件中的顺序，直到文件中任意两个记录之间都满足排序要求为止。常用的交换排序包括起泡排序和快速排序。

5.4.1 起泡排序

起泡排序是最简单的交换排序。起泡排序的排序过程如下：首先比较第 1 个记录和第 2 个记录的排序码，如果不满足排序要求，则交换第 1 个记录和第 2 个记录的位置；然后对第 2 个记录（可能是新交换过来的最初的第 1 个记录）和第 3 个记录进行同样的处理；重复此过程，直到处理完第 $n-1$ 个记录和第 n 个记录为止。上述过程称为一次起泡过程，这个过程的处理结果就是将排序码最大（非递减序）或最小（非递增序）的那个记录交换到最后一个记录位置，到达这个记录在最后排序后的正确位置。然后，重复上述起泡过程，但每次只对前面的未排好序的记录进行处理，直到所有的记录均排好序为止。

在每次起泡过程中，可以设立一个标志位，用于标示每次起泡过程中是否进行过记录交换。如果某次起泡过程中未发生交换，则表明整个记录已经达到了排序要求。显然，对 n 个记录的排序处理最多需要 $n-1$ 次起泡过程。

例 5-6 设某文件中待排序记录的排序码分别为 28、6、72、85、39、41、13、20。用起泡排序法对该文件进行排序，排序过程如图 5.7 所示。

图 5.7 起泡排序过程

下面给出起泡排序的函数 BubbleSort，该函数的参数是存放被排序文件的数组 A 和文件中所包含的记录个数（即数组 A 的大小）n。

算法 5.5 起泡排序。

```
/*用起泡排序法对文件 A[0],A[1],…,A[n-1]排序*/
void BubbleSort(ForSort A[],int n)
{
    int i,j;
    Bool flag;
    ForSort temp;

    for(i=n-1,flag=(Bool)1;i>0&&flag;i--)
    {
        flag=FALSE;                          /*设置未交换标志*/
        for(j=0;j<i;j++)
            if(A[j+1].key<A[j].key)
            {
                flag=TRUE;                   /*有交换发生,置标志*/
                temp=A[j+1];                 /*交换*/
                A[j+1]=A[j];
                A[j]=temp;
            }
    }
}
```

在执行起泡排序前，如果待排序文件中的记录顺序已经满足排序要求，则只需一次起泡过程即可，此时比较次数和移动次数均为最少，比较次数为 $n-1$ 次，移动次数为 0 次；如果待排序文件中的记录顺序是与排序要求逆序的，则需要进行 $n-1$ 次排序，此时比较次数和移动次数均达到最大，比较次数为