

第6章 排 序

如果将数据按照一定排序标准进行排序, 数据处理的效率常常就会显著提高。例如, 一个学生成绩管理系统中, 每个学生的信息包含了学号、姓名、课程、成绩等属性。如果这个系统是按照学号排好顺序的, 当需要查找一个学生的信息时, 可以根据这个学生的学号, 用折半查找的算法很快地找到该学生的信息。利用已经排好的数据可以带来很多方便, 这种方法也适用于计算机科学。

在计算机科学中, 排序算法是一种重要的操作。合理的排序算法能够大幅提高计算机处理数据的性能。在对数据记录进行排序之前, 要先确定排序的标准。待排序的数据记录可能包含一个或多个属性, 如学生成绩管理系统中, 每条记录都包含了学号、姓名、课程等属性。用来作为排序依据的属性称为关键字域, 简称关键字, 例如前面例子中的学号。排序就是根据关键字的大小将无序的多条记录, 调整为有序的序列。

6.1 排序的基本概念

给定多条记录的一个序列 $R = \{r_1, r_2, \dots, r_n\}$, 其对应的关键字为 $K = \{k_1, k_2, \dots, k_n\}$, 排序要解决的问题可以描述为:

输入: 序列 $R = \{r_1, r_2, \dots, r_n\}$, 关键字 $K = \{k_1, k_2, \dots, k_n\}$

输出: 新序列 $R' = \{r'_1, r'_2, \dots, r'_n\}$ 及 $K = \{k'_1, k'_2, \dots, k'_n\}$, 使得 $k'_1 \leq k'_2 \leq \dots \leq k'_n$ 。

如果记录的关键字都没有重复出现, 那么排序算法可以得到唯一的结果; 否则, 排序的结果可能不唯一。当关键字可以重复出现时, 假设 $k_i = k_j$, 且在排序前的序列 R 中, R_i 领先于 R_j , 若在排序后的序列 R' 中 R_i 仍领先于 R_j , 则称排序算法是稳定的; 否则, 若排序后的序列 R' 中, R_j 领先于 R_i , 则称排序算法是不稳定的。例如, 序列 $\{3, 15, 8, \mathbf{8}, 6, 9\}$, 其中用粗体来区分数字 8。若排序后得到 $\{3, 6, \mathbf{8}, \mathbf{8}, 9, 15\}$, 则该排序方法是不稳定的。

当数据量较小时, 可以完全存放在计算机随机存储器中进行排序; 但是如果数据量比较大时, 如银行账户信息等, 数据不能全部存放计算机随机存储器中, 排序过程中需要在内、外存之间移动数据。因此可以将排序分为内部排序和外部排序。内部排序指的是待排序记录存放在计算机随机存储器中进行的排序过程。外部排序指的是待排序记录的数量很大, 以致内存一次不能容纳全部记录, 在排序过程中需要对外存进行访问的排序过程。

排序算法有很多种, 要确定哪种算法好就必须建立一种评价标准。由于排序过程中最基本的运算是关键字的比较和数据的移动, 因此以关键字比较次数和数据的移动次数来度量排序算法的时间复杂度。确定比较次数或数据移动次数的精确值通常是不必要或者不可能的, 本章中用 O 符号近似给出比较和移动次数的数量级。排序算法的效率往往与数据的初始顺序有很大关系, 所以在分析排序算法性能时, 要考虑最好情况、最坏情

况以及平均情况下算法的时间复杂度和空间复杂度。

假设待排序的数据元素个数为 n ，并且对于数据元素存在“ $<$ ”和“ $>$ ”运算。通过比较确定 n 个数据元素的相对次序的排序算法称为基于比较的排序算法。本章主要介绍几种常用的基于比较的排序算法以及基数排序，并给出它们的代码实现。同时也分析比较了不同算法的复杂度。如不特殊说明，本章所指的排序是按照不减序的排序。

6.2 插 入 排 序

6.2.1 直接插入排序

直接插入排序是一种简单的排序算法，由 $n-1$ 趟排序组成。第 p 趟排序后保证从第 0 个位置到第 p 个位置上的元素为有序状态。第 $p+1$ 趟排序是将第 $p+2$ 个元素插入到前面 $p+1$ 个元素的有序表中。图 6-1 显示了应用直接插入排序算法的每一趟的排序情况。

初始数据序列	32	18	65	48	27	9
第 1 趟排序之后	18	32	65	48	27	9
第 2 趟排序之后	18	32	65	48	27	9
第 3 趟排序之后	18	32	48	65	27	9
第 4 趟排序之后	18	27	32	48	65	9
第 5 趟排序之后	9	18	27	32	48	65

图 6-1 每趟直接插入排序结果

上面的例子中：

第 1 趟排序是将元素 18 插入到前面 1 个元素的有序序列 {32} 中，形成新的有序序列 {18, 32}。第 2 趟排序是将元素 65 插入到前面 2 个元素的有序序列 {18, 32} 中，形成新的有序序列 {18, 32, 65}。同理，第 5 趟排序是将元素 9 插入到前面 5 个元素的有序序列 {18, 27, 32, 48, 65} 中，形成新的有序序列 {9, 18, 27, 32, 48, 65}，得到最后的有序序列。

推广到一般情形，第 p 趟排序后使得数据元素 $data[0]$ ， $data[1]$ ， \dots ， $data[p]$ 形成一个有序序列，进行第 $p+1$ 趟排序时，要将 $data[p+1]$ 插入到前面的有序序列中。首先用一个临时空间 $temp$ 存储 $data[p+1]$ ，然后将 $temp$ 与 $data[p]$ 进行比较，如果前者小，则将 $data[p]$ 移动到 $data[p+1]$ ；继续将 $temp$ 与 $data[p-1]$ 进行比较，如果前者小，则将 $data[p-1]$ 移动到 $data[p]$ ；重复这个过程，直到 $temp$ 不小于 $data[i]$ （或者 $data[0] \dots data[p]$ 都向后移动），则将 $temp$ 移动到 $data[i+1]$ 的位置（或者 $data[0]$ 的位置）。

直接插入排序的算法如例 6.1 所示。

【例 6.1】直接插入排序。

```
template<class T>
void InsertionSort(T Data[],int n)
```

```

{ //利用直接插入排序对于 n 个数据元素进行不减序排序
    int p,i;
    for( p = 1; p < n; i++) //循环, p 表示插入趟数, 共进行 n-1 趟插入
    {
        T temp = Data[p]; //把待插入元素赋给 temp
        i = p - 1;
        while( i>= 0 && Data[i] > temp) //把比 temp 大的元素都向后移动
        {
            Data[i+1] = Data[i];
            i--;
        }
        Data[i+1] = temp; //i+1 为 temp 的位置, 将 temp 插入到这个位置
    }
}

```

直接插入排序算法主要应用比较和移动两种操作。从空间上来看, 它只需要一个元素的辅助空间, 用于位置的交换, 有些教材也将这类排序算法称为原地 (In Place) 排序算法。

从时间分析, 首先外层循环要进行 $n-1$ 次。但每一趟插入排序的比较和移动次数并不相同。第 p 趟插入时最好的情况是数据已经排好序, 每趟插入操作进行一次比较, 两次移动; 最坏情况是比较 p 次, 移动 $p+2$ 次 (逆序) ($p=1,2,\dots,n-1$)。记 M 为执行一次排序算法移动的次数, C 为比较的次数, 则有如下结论:

$$C_{\min} = n-1, \quad M_{\min} = 2(n-1);$$

$$C_{\max} = 1+2+\dots+(n-1), \quad M_{\max} = 3+4+\dots+(n+1) = (n^2+3n-4)/2;$$

假设数据元素在各个位置的概率相等, 即 $1/n$, 则平均的比较次数和移动次数为:

$$C_{\text{ave}} = (n^2+n-2)/4, \quad M_{\text{ave}} = (n^2+5n-6)/4;$$

因此, 直接插入排序的时间复杂度为 $O(n^2)$ 。对于随机顺序的数据来说, 移动和比较的次数接近最坏情况。

由于直接插入算法的元素移动是顺序的, 该排序算法是稳定的, 感兴趣的同学可以自己证明。

6.2.2 折半插入排序

直接插入排序算法是利用有序表的插入操作来实现对数据集合的排序。在进行第 $p+1$ 趟的插入排序时, 需要在前面的有序序列 $\text{data}[0], \text{data}[1], \dots, \text{data}[p]$ 中找到 $\text{data}[p+1]$ 的对应的位置 i , 同时将 $\text{data}[i], \text{data}[i+1], \dots, \text{data}[p]$ 都向后移动一个位置。由于有序表是排好序的, 故可以用折半查找 (二分法) 操作来确定 $\text{data}[p+1]$ 对应的位置 i , 这就是折半插入排序算法的思想。

例如, 对有序表 $\{5, 6, 7, 8, 9, 10\}$, 当插入元素 4 时查找元素 4 的插入位置的过程如图 6-2 所示。可以看到由于使用了折半查找的方法在有序表中查找待插入元素的对应

位置而大大提高了查找速度。在本例中，原本需要 5 次比较操作才能确定元素 4 的插入位置，现在只需要 2 次比较操作就可以完成。

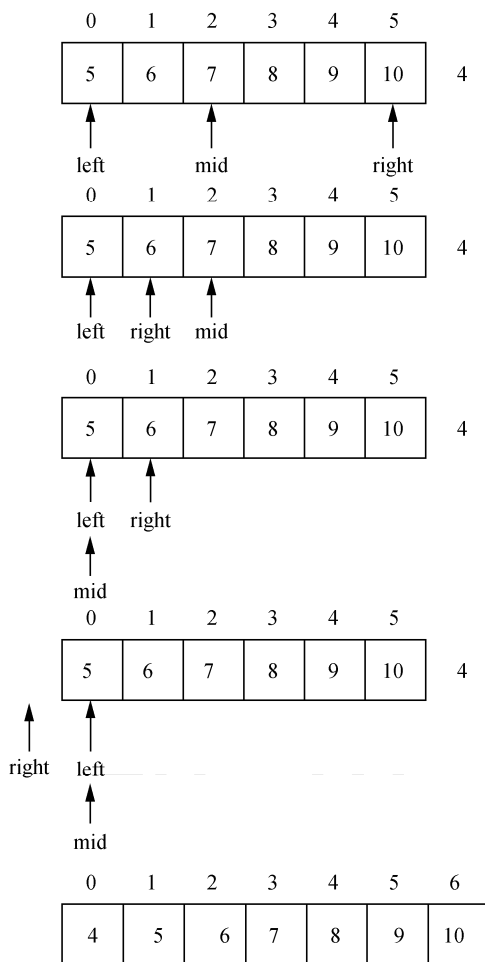


图 6-2 折半插入排序

折半插入排序算法如例 6.2 所示。

【例 6.2】折半插入排序算法。

```
template<class T>
void BinaryInsertionSort(T Data[],int n)    //参数: 待排序数据和待排序元
                                              //素个数
{
    int left,mid,right,p;                    //声明一些变量
    for( p = 1; p < n; p++)                  //共进行 n-1 次插入
    {
        T temp = Data[p];                   //保存待插入数据
```

```

left = 0, right = p-1;           //初始化 left 和 right 的值
while(left <= right)             //执行折半查找
{
    mid = (left + right) / 2;     //求出中心点
    if( Data[mid] > temp )       //中心点元素值比待插入数据大
        right = mid -1;         //更新右区间值
    else
        left = mid + 1;          //更新左区间值
}
for( int i = p-1; i >= left; i--)//执行移动操作
    Data[i+1] = Data[i];
Data[left] = temp;               //将待插入元素插入到有序表中
}
}

```

折半插入排序算法与直接插入排序算法相比,需要的辅助空间与直接插入排序基本一致;时间上,折半插入排序的比较次数比直接插入排序算法的最坏情况好,最好的情况下,时间复杂度为 $O(n \log_2 n)$;折半插入排序算法的元素的移动次数与直接插入排序相同,复杂度仍为 $O(n^2)$ 。

折半插入排序算法与直接插入排序算法的元素移动一样是顺序的,因此该排序算法也是稳定的。

6.2.3 希尔排序

折半插入排序算法改进了直接插入排序算法中确定待插入元素位置的操作,希尔排序则是利用直接插入排序的一些特点进行另一种改进。

前面章节对直接插入排序算法的分析说明,如果待排序的数据是有序的,那么最好的时间复杂度是 $O(n)$,另外对于短序列来说,插入排序也是比较有效的排序算法。希尔(Shell)排序正是利用直接插入排序的这两个性质而进行改进。

希尔排序的基本思想是,先将待排序数据序列划分成为若干子序列分别进行直接插入排序;待整个序列中的数据基本有序后,再对全部数据进行一次直接插入排序。对于子序列的排序可以采用任意简单的排序算法,本书中对于子序列的排序采用的是直接插入排序算法。例如,对于序列{65, 34, 25, 87, 12, 38, 56, 46, 14, 77, 92, 23},可以划分成图 6-3 所示的 6 个子序列。对于每个子序列使用直接插入排序,结果为 56, 34, 14, 77, 12, 23, 65, 46, 25, 87, 92, 38。对于第 1 趟希尔排序的结果要继续划分,但是要缩小增量。

如果初始序列为 $data[0], data[1], \dots, data[n-1]$,子序列中元素的间隔为 d ,则子序列可以描述为 $data[i], data[i+d], data[i+2*d], \dots, data[i+k*d]$ (其中 $0 \leq i < d, i+k*d < n$)。希尔排序中通过不断的缩小增量,来将原始序列分成若干个子序列。例如,增量初始的时候可以选为待排序的元素个数的一半,即 $\left\lfloor \frac{n}{2} \right\rfloor$ ($\lfloor \rfloor$ 表示的是向下取整),在后来的迭

代过程中不断缩小增量，下一次的增量为上一次的一半，即第 2 趟时选择增量为 $\left\lfloor \frac{n}{4} \right\rfloor$ ，

以此类推，直到增量变为 1 时为止。这时序列已经基本有序，对整个序列进行一次插入排序即可完成数据排序。增量序列的选择对于希尔排序的效率有较大的影响，在最初的希尔排序算法中，通过每次将原始的序列等分为两个子序列来进行划分。实际上，Donald Knuth 证明，即使只有两个增量 $(16n/\pi)^{1/3}$ 和 1，希尔排序的效率也要高于插入排序，这时希尔排序算法的时间复杂度为 $O(n^{1.3})$ 。目前，已经提出许多增量选择策略，其中满足下列条件的序列是一种比较合适的增量序列。

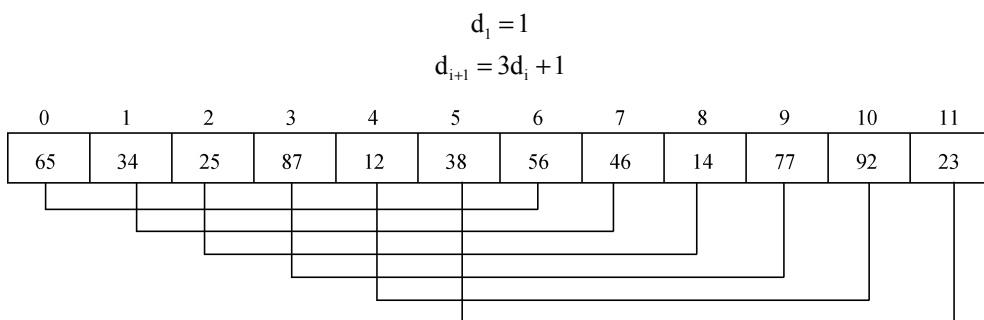


图 6-3 希尔排序的子序列划分示例

按照这个递归公式，选择的增量序列为 $\{1, 4, 13, 40, 121, 364, 1093, 3280, \dots\}$ ，可以根据待排序序列的数据个数来确定该增量序列的最大值。

对于序列 $\{65, 34, 25, 87, 12, 38, 56, 46, 14, 77, 92, 23\}$ ，进行希尔排序的执行过程如图 6-4 所示。按照原始的序列选择策略，一共有 12 个数据，算法执行第一趟的时候，所有数据被分为 $\frac{12}{2}=6$ 组，每组有 2 个元素，对每组的两个元素进行排序。第二趟的时候增量为 $\frac{12}{4}=3$ ，将第一趟排序后的结果分为 3 组，分别对这三组元素执行直接插入排序算法。最后一趟，增量为 $\left\lfloor \frac{12}{8} \right\rfloor = 1$ ，对所有元素执行一次插入排序。

希尔排序算法如例 6.3 所示。

【例 6.3】希尔排序算法。

```
template<class T>
void ShellSort(T Data[],int n)
{
    int d = n/2;                                //增量初始化为数组大小的一半
    while(d>=1)                                  //循环遍历增量的所有可能
    {
        for(int k = 0; k<d; k++)                //遍历所有的子序列
        {
```

```

for( int i = k+d; i < n; i+=d) //对每一个子序列执行直接插
                                //入排序
{
    T temp = Data[i];
    int j = i - d;
    while( j>= k && Data[j] > temp)
    {
        Data[j+d] = Data[j];
        j -= d;
    }
    Data[j+d] = temp;
}
d = d/2;                        //增量变为上次的一半
}

```

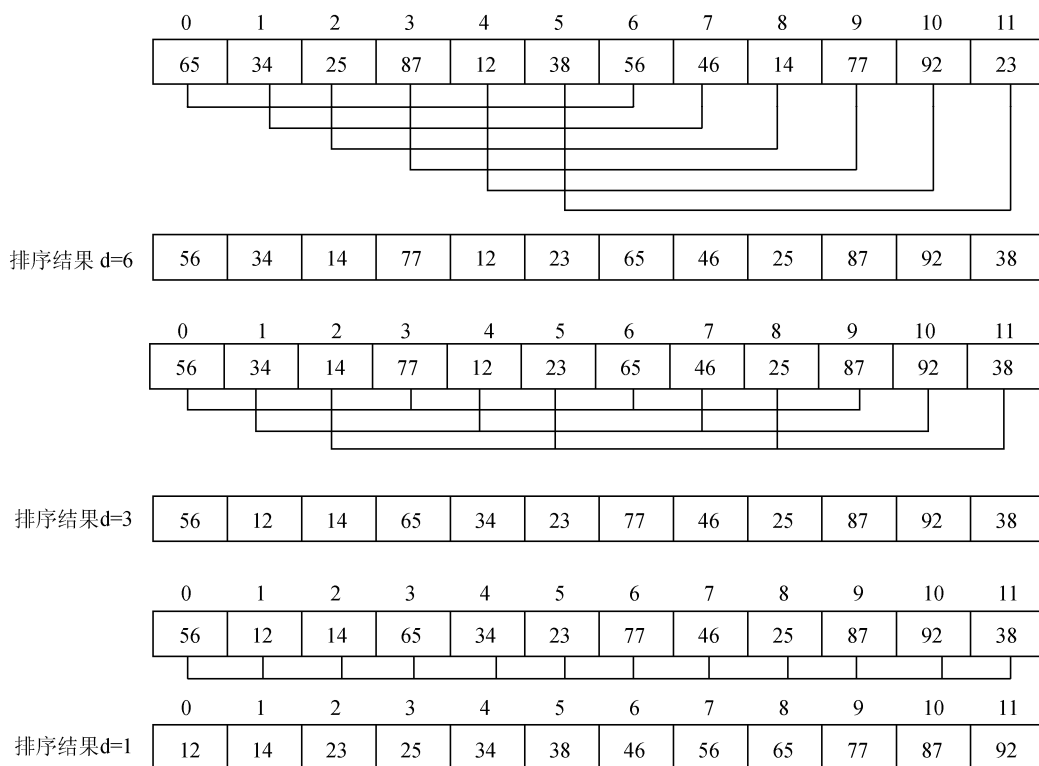


图 6-4 希尔排序算法流程

希尔排序算法复杂度依赖于增量序列的选择，分析过程比较复杂。但希尔排序的时间复杂度在 $O(n \log_2 n)$ 和 $O(n^2)$ 之间，大致为 $O(n^{1.3})$ ，和直接插入排序算法相比，大大

减少了算法的复杂度。

希尔算法本身并不是稳定的。例如，对于初始序列{65, 34, **34**, 87}，增量序列为{2, 1}，则希尔排序的结果为{**34**, 34, 65, 87}。

6.3 交 换 排 序

交换排序主要是通过两个元素之间的交换操作，达到排序的目的。最常用的交换排序算法有冒泡排序算法和快速排序算法。

6.3.1 冒泡排序

冒泡排序通过不断比较相邻元素的大小，然后决定是否对这两个元素进行交换操作，从而达到排序的目的。

具体来说对于序列 $R = \{r_1, r_2, \dots, r_n\}$ ，冒泡排序按照下述的步骤对序列 R 进行 $n-1$ 趟排序。

第一趟排序：

首先将第一个元素 r_1 和第二个元素 r_2 相比较，若为逆序，则交换 r_1 和 r_2 ；然后比较第二个元素 r_2 和第三个元素 r_3 的大小，若为逆序，则交换 r_2 和 r_3 。以此类推，直至比较第 $n-1$ 个元素 r_{n-1} 和第 n 个元素 r_n 的大小，若为逆序，则交换两者。经过这样的处理过程之后最大的元素就到了序列 R 的最右端，这样就完成第一轮排序。如果将 R 看作是一个垂直的柱体（序号大的元素在上方），其中排序的目的是使得最大的元素在顶部，最小的元素在底部，则上述过程就好像是湖底的一个气泡慢慢冒出来，这就是冒泡排序名称的由来。

第二趟排序，由于最大的元素已经在 R 的最右端了，因此只需要对记录 $\{r_1, r_2, \dots, r_{n-1}\}$ 进行上述的排序过程就可以了。以此类推不断扫描数据序列，直到所有的元素都排好序为止。

冒泡排序算法如例 6.4 所示。

【例 6.4】冒泡排序算法。

```
template<typename T>
void BubbleSort(T Data[],int n)
{
    for(int i = 0; i < n; i++)           //外层循环控制排序的每一趟
    {
        for(int j = 1; j < n-i; j++)     //内层循环控制本趟中的冒泡操作
        {
            if(Data[j] < Data[j-1])     //如果是逆序的，则交换这两个元素
            {
                T t = Data[j];
                Data[j] = Data[j-1];
            }
        }
    }
}
```



```
Data[j-1] = t;
```

图 6-5 演示了序列{10, 5, 7, 8, 6, 9}执行冒泡排序的过程。

从图 6-5 可以看到，对于冒泡排序算法来说，每趟结束时，不仅能挤出一个最大值到最后面位置（或者最小值到最前面位置），还能同时部分理顺其他元素。而且一旦下趟没有交换发生，还可以提前结束排序过程，实际上图 6-5 中第四趟之后的操作都没有必要。在实现冒泡排序算法的时候，可以用一个变量来记录一趟排序过程中是否执行了交换操作，如果没有交换操作，则可以提前结束程序算法的执行，从而进一步提高排序算法的效率。改进后的冒泡排序算法如例 6.5 所示。

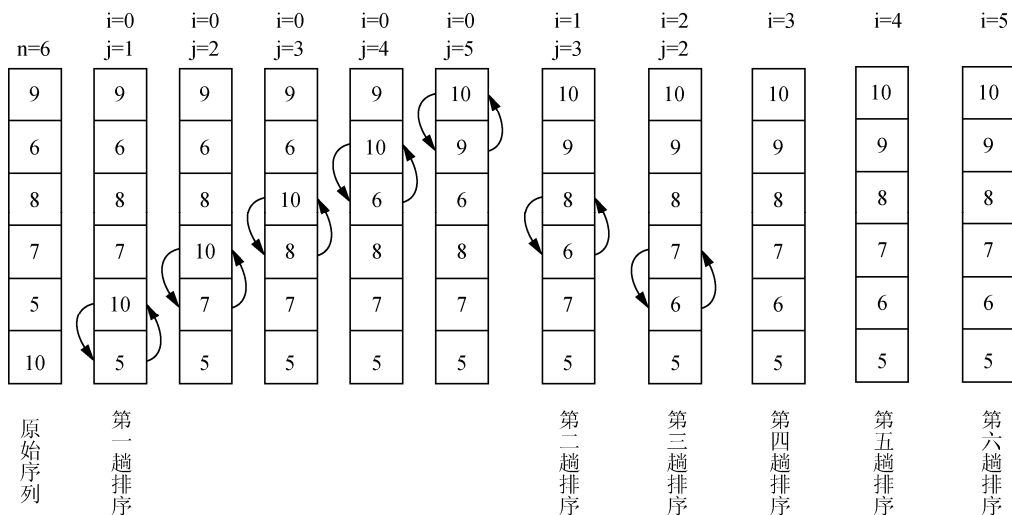


图 6-5 冒泡排序算法流程

【例 6.5】冒泡排序的改进算法。

```
template<class T>
void BubbleSort(T Data[],int n)
{
    int flag = 0 ; //标记每一趟的冒泡排序过程中是否发生了交换
    for(int i = 0; i < n; i++) //外层循环控制排序的每一趟
    {
        flag = 0 ;
        for(int j = 1; j < n-i; j++) //内层循环控制本趟中的冒泡操作
        {
            if(Data[j] < Data[j-1]) //如果是逆序的，则交换这两个元素
            {
                flag = 1 ;
                T t = Data[j];
```

```

        Data[j] = Data[j-1];
        Data[j-1] = t;
    }
}
if (flag == 0)    //如果某一趟的冒泡过程中没有发生交换则结束排序
    return;
}
}

```

显然，冒泡排序算法的效率和待排序序列的初始顺序密切相关。若待排序的元素为正序，则是冒泡排序的最好情况，此时只需进行一趟排序，比较次数为 $n-1$ 次，移动元素次数为 0 次；若初始待排序的元素为逆序，则是冒泡排序的最坏情况，此时需要执行 $n-1$ 趟排序，第 i 趟 ($1 \leq i \leq n$) 做了 $n-i$ 次关键字比较，执行了 $3(n-i)$ 次数据元素交换，因此比较总次数和记录移动次数分别为：

$$\text{比较次数: } \sum_{i=1}^{n-1} (n-i) = \frac{1}{2}n(n-1)$$

$$\text{移动次数: } 3 \sum_{i=1}^{n-1} (n-i) = \frac{3}{2}n(n-1)$$

所以，冒泡排序算法的时间复杂度最坏情况为 $O(n^2)$ 。冒泡排序算法的平均时间复杂度也是 $O(n^2)$ ，这可以根据每趟的平均比较次数和平均移动次数计算出来，感兴趣的读者可以自己计算一下。冒泡排序算法每次都要考虑相邻两个元素的大小关系，若为逆序则交换，移动操作较多，所以属于内排序中速度较慢的一种。另外，由于冒泡排序将注意力集中在要向上移动的数据上，因此，有些已经在正确位置上的元素（如图 6-5 中的数字 8 和数字 7）也可能被移动走，从而导致冗余的比较和交换动作。

由于冒泡排序算法只进行元素间的顺序移动，所以是一个稳定的排序算法。

6.3.2 快速排序

回顾 6.2.3 学习的希尔排序算法，它是首先将原始的序列划分为若干子序列，然后对各个子序列分别进行排序；然后再次执行相同的操作，将整个序列划分为新的更少的子序列，并排序；这样一直做下去，直到整个序列都排好序为止。将序列进行划分的目的是将原始的问题变为容易解决的小问题。其实希尔排序算法里面，体现了计算机科学中的“分治法”的思想，它是算法设计中的一种重要的思想。本节中介绍的快速排序算法也是基于“分治法”思想提出的一种排序算法。快速排序是名副其实的，因为在实际应用中，它几乎是最快的排序算法，被评选为 20 世纪十大算法之一。

快速排序算法主要由下面的三步组成：

1) 分割：取序列的一个元素作为轴元素，利用这个轴元素，把序列分成三段，使得所有小于等于轴的元素放在轴元素的左边，大于轴的元素放到轴元素的右边。此时，轴元素已经被放到了正确的位置。

2) 分治：对左段和右段中的元素递归调用 1) 中例程，分别对左段和右段中的元素进行排序。

3) 合并：对于快速排序来说，每个元素都已被放到正确位置，因此，合并过程不需要执行其他操作，整个序列已排好序。

在分割步骤中需要找到一个“轴元素”，也称该元素为哨兵。轴元素的选取对于快速排序算法的性能有较大的影响，目前已经提出很多不同的轴元素选择策略，最简单的办法就是选择第一条记录或者是最后一条记录作为轴元素。但用这样的策略选择轴元素时，当输入的序列是已经排好序或是逆序的时候，此时产生的两个子序列严重不平衡，每次都会有一个子序列是空的，导致算法的效率是最低的。也可以用中值作为轴，但实际应用表明这种选择策略也不是很理想。另外一种常用的策略是每次随机化的从待排序的序列中选择一个元素作为轴，这种策略可以获得较好的平均性能。

图 6-6 解释了对一个数据集的快速排序做法。这里的轴元素（随机地）选为 67，数据集中其余元素被分成两个集合 L、R。递归地将集合 L 排序得到 12, 25, 37, 46，集合 B 类似处理，此时整个数据集的排序就得到了。

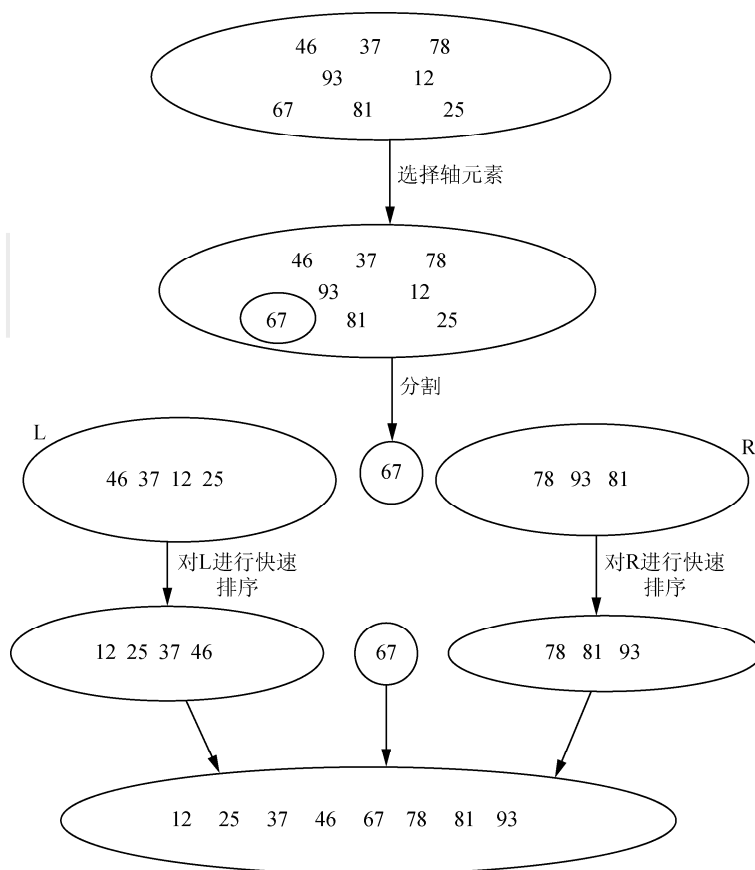


图 6-6 快速排序各步骤描述举例

本书在实现快速排序算法时，用待排序列左边第一个元素作为轴元素。在实际应用中如果不是选择待排序列中的第一元素作为轴，则可以先将轴元素和第一个元素交换，从而转换成用第一个元素作为轴的情形。

根据快速排序的步骤，确定轴元素之后就要将待排序列分割成两个部分，使得轴左边的元素都小于等于轴对应的元素，右边的元素都大于轴对应的元素。在实践中有两种常用的分割策略。

第一种分割策略是，首先用一个临时变量对首元素（即轴元素）进行备份，取两个指针 `left` 和 `right`，它们的初始值分别是待排序列两端的下标，其中 `left` 指向序列最左边的下标，`right` 指向序列最右边的下标。在整个排序过程中保证 `left` 不大于 `right`，用下面的方法不断移动两个指针：

首先从 `right` 所指的位置向左搜索，找到第一个小于或等于轴的元素，把这个元素移动到 `left` 的位置。

再从 `left` 所指的位置开始向右搜索，找到第一个大于轴的元素，把它移动到 `right` 所指的位置。

重复上述过程，直到 `left=right`，最后把轴元素放在 `left` 所指的位置。

经过上面的处理之后，所有大于轴的元素被放在轴的右边，所有小于等于轴的元素被放在轴的左边，从而达到了对序列进行划分的目的。

图 6-7 给出了对序列 {45, 32, 61, 98, 74, 17, 22, 53} 执行一次分割时的执行过程。

这种分割策略的具体实现如例 6.6 所示。

【例 6.6】快速排序分割策略一。

```
template <class T>
int Partition(T Data[],int left, int right)
//实现对 data[left]到 data[right]的分割操作，并返回划分后轴元素对应的位置
{
    T pivot = Data[left];           //选择最左边的为轴元素
    while(left < right)              //外层循环控制遍历数组
    {
        while(left< right && Data[right] > pivot)//控制 right 指针的
移动
            right--;
        Data[left] = Data[right];    //将 right 指向的数据移动到 left 位置
        while(left < right && Data[left] <= pivot)//控制 left 指针移动
            left++;
        Data[right] = Data[left];    //将 left 指向的数据移动到 right 位置
    }
    Data[left] = pivot;              //将轴元素放到 left 位置
    return left;                    //返回轴元素的新位置，实现分治
}
```

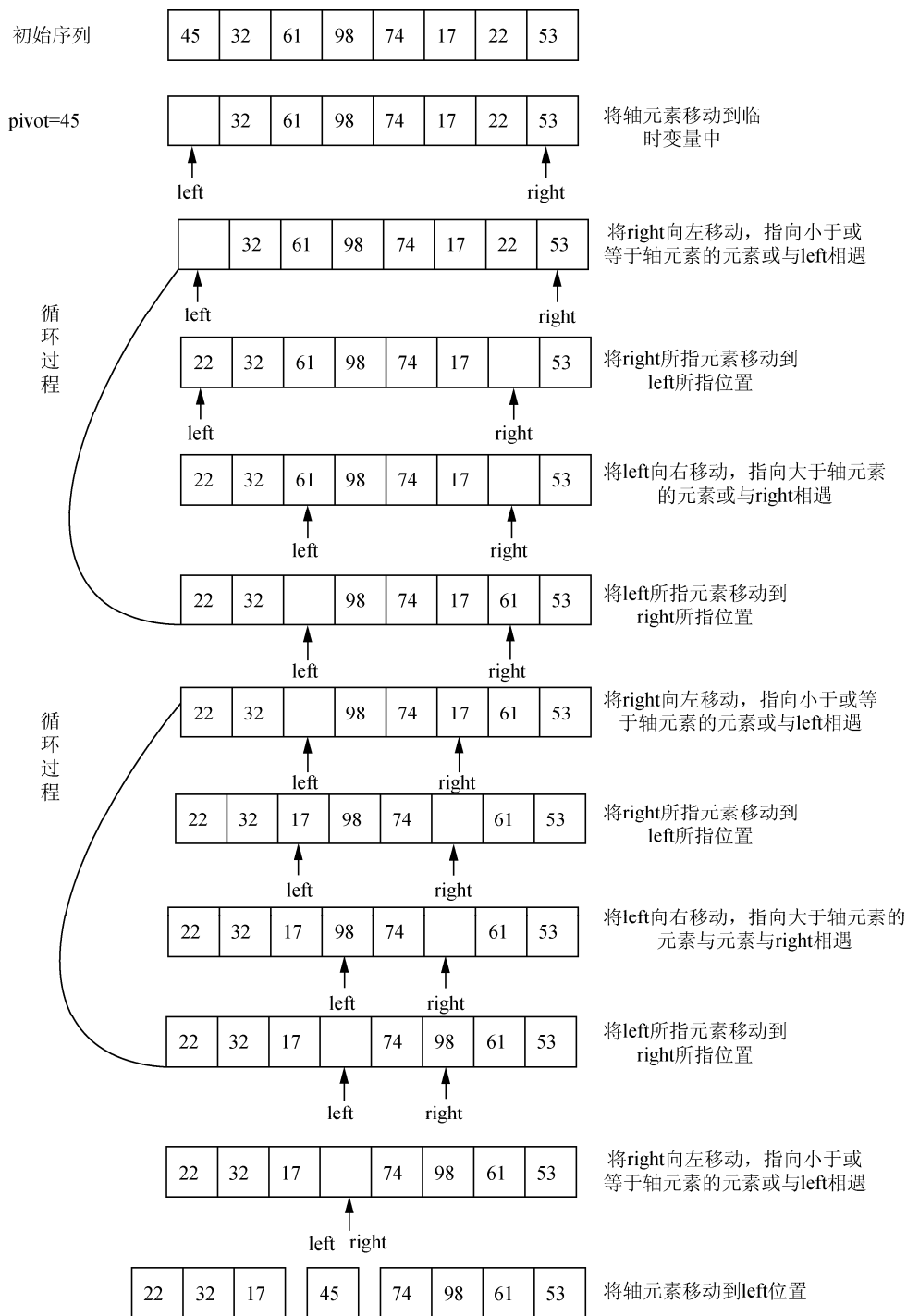


图 6-7 快速排序算法的分割策略一

第二种分割策略与第一种分割策略不同的是，分别从待排序序列的两边相向遍历，即从左向右遍历确定第一个大于轴元素的元素，从右向左遍历确定第一个不大于轴元素的元素，然后交换二者。

例如对上面的初始待排序序列，依然要先备份轴元素 45，然后定义两个扫描变量 `left`, `right`, `left` 初始值为待排序序列中第二个元素位置（第一个元素被选为轴元素），`right` 初始值为待排序序列中最后一个元素位置；当 `left` 不大于 `right` 时，向右移动 `left`，使其停在第一个大于 45 的元素位置，同时向左移动 `right`，使其停在第一个不大于 45 的元素位置，然后交换 `left` 和 `right` 位置的元素；然后继续移动 `left`、`right`，交换相应的元素，重复这个过程；直至 `left` 大于 `right`。此时 `right` 以及 `right` 左边的元素一定是不大于轴元素的，而 `left` 以及 `left` 右边的元素一定是大于轴元素的，将第一个元素（即轴元素）与 `right` 所指的元素交换，从而完成分割过程。具体过程如图 6-8 所示。

这种分割策略的具体实现如例 6.7 所示。

【例 6.7】快速分割策略二。

```
template <class T>
int Partition(T Data[],int start, int end)
{//实现对 data[start]到 data[end]的分割操作，并返回划分后轴元素对应的位置
    T pivot = Data[start] ;
    int left =start, right = end ;           //初始化 left,right
    while(left < =right)                     //外层循环控制遍历数组
    {
        while(left<= right && Data[left] <= pivot)//控制 left 指针的
            left++;
        while(left < =right && Data[right] >pivot) //控制 right 指针
            right--;
        if(left<right)
        {
            swap(Data[right],Data[left]); //交换 Data[right]和 Data[left]
            left ++; right--;
        }
    }
    Swap(Data[start] ,Data[right]); //交换 Data[right]和轴元素 Data[start]
    return right;                    //返回轴元素的新位置，实现分治
}
```

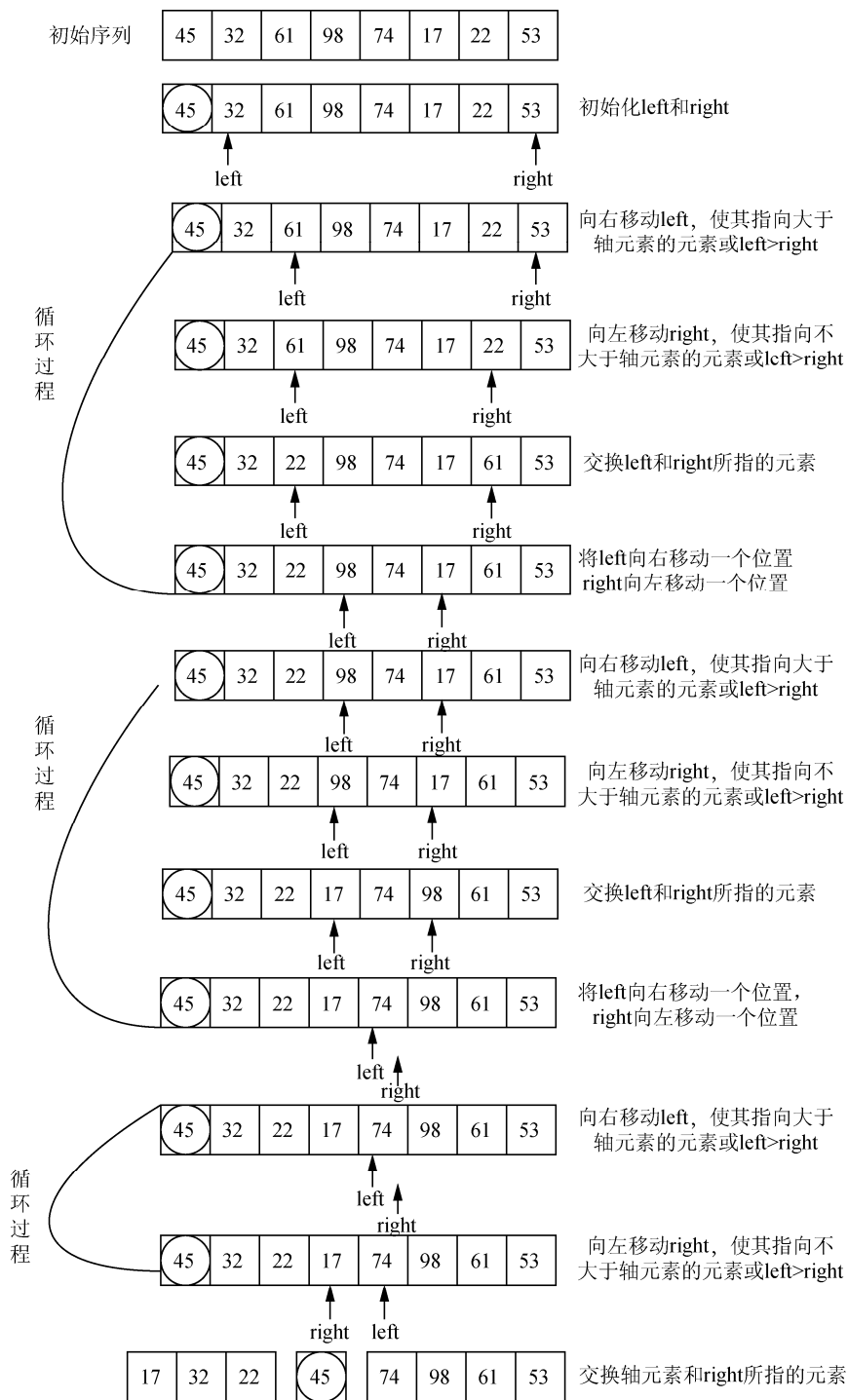


图 6-8 快速排序算法的分割策略二

快速排序算法就是递归调用上述分割策略来不断划分子序列从而完成排序,具体实现如例 6.8 所示。

【例 6.8】快速排序。

```
template <class T>
void QuickSort(T Data[],int left,int right)
{//用分治法实现快速排序算法
    if(left < right)                                //控制分治的结束条件
    {
        int p = Partition(Data,left,right); //实现分割并找到分离的位置
        QuickSort(Data,left,p-1);          //对左边的子序列进行快速排序
        QuickSort(Data,p+1,right);          //对右边的子序列进行快速排序
    }
}
```

对于 n 个元素的序列进行快速排序时,第一次分割显然需要 $n-1$ 次关键字的比较。快速排序的运行时间等于两个递归调用的运行时间加上分割时间。下面分最坏情形、最好情形和平均情形来分析快速排序的时间复杂度。

快速排序的最坏情形出现在输入序列有序时,每一次分割都将轴元素划分在序列的一端。即对 k 个元素的有序序列划分得到子序列长度分别为 0 和 $k-1$ 。因此快速排序的递归过程总的比较次数为

$$W(n) = \sum_{k=2}^{k=n} (k-1) = \frac{n(n-1)}{2} = O(n^2)$$

由此可知,在最坏情形下,快速排序和直接插入排序、冒泡排序一样差。

快速排序的最好情形是每次分割都是最平衡的,也就是每次分割之后,一个子序列的长度为 $\lfloor n/2 \rfloor$, 另一个子序列的长度为 $\lfloor n/2 \rfloor$, 其中 n 表示当前待划分序列的长度。

这时,快速排序的比较次数可以通过下面的递推方程获得:

$$\begin{aligned} W(1) &= 0 \\ W(n) &\approx n-1+2W\left(\frac{n}{2}\right) \\ &\approx n-1+2\left(\frac{n}{2}-1\right)+4W\left(\frac{n}{4}\right) \\ &\approx n-1+2\left(\frac{n}{2}-1\right)+4\left(\frac{n}{4}-1\right)+8W\left(\frac{n}{8}\right) \\ &\approx n \log n + (1+2+4+\cdots+\left\lfloor \frac{n}{2} \right\rfloor) \\ &\approx n \log n + n = O(n \log n) \end{aligned}$$

快速排序的平均时间性能分析较为困难。首先假设待排元素的关键字互不相等,且 n 个元素的关键字的所有不同的排列以相等的概率出现在输入序列中,因此轴元素的最终位置 i 可以取值为 1, 2, \cdots , n 。快速排序算法的平均比较次数满足下面的递归方程:

$$W(1) = 0$$

$$\begin{aligned} W(n) &= n-1 + \sum_{i=1}^n \frac{1}{n} (W(i-1) + W(n-i)) \quad n > 1 \\ &= n-1 + \frac{2}{n} \sum_{i=0}^{n-1} W(i) = n-1 + \frac{2}{n} \sum_{i=1}^{n-1} W(i) \\ &= n-1 + \frac{2}{n} (W(1) + W(2) + \dots + W(n-1)) \end{aligned}$$

推得

$$\begin{aligned} W(n-1) &= n-2 + \frac{2}{n-1} \sum_{i=1}^{n-2} W(i) \\ &= n-2 + \frac{2}{n-1} (W(1) + W(2) + \dots + W(n-2)) \end{aligned}$$

计算 $nW(n) - (n-1)W(n-1)$ 得到:

$$\begin{aligned} nW(n) - (n-1)W(n-1) &= n(n-1) + 2 \sum_{i=1}^{n-1} W(i) - (n-1)(n-2) - 2 \sum_{i=1}^{n-2} W(i) \\ &= 2W(n-1) + 2(n-1) \end{aligned}$$

即

$$\frac{W(n)}{n+1} = \frac{W(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

设

$$C(n) = \frac{W(n)}{n+1}, \quad C(1) = 0$$

则有

$$C(n) = \begin{cases} 0 & n=1 \\ C(n-1) + \frac{2(n-1)}{n(n+1)} & n>1 \end{cases}$$

这是一个递归方程, 经过简单计算可以得到:

$$C(n) = 2 \sum_{i=1}^n \frac{i-1}{i(i+1)} = 2 \sum_{i=1}^n \frac{1}{i} - 4 \sum_{i=1}^n \frac{1}{i(i+1)}$$

由 Harmonic 级数, 有

$$\sum_{i=1}^n \frac{1}{i} \approx \ln n + \gamma, \quad \gamma \text{ 是 Euler 常数, 约为 } 0.577$$

另外,

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

$$\text{所以 } C(n) \approx 2(\ln n + \gamma) - \frac{4n}{n+1}$$

从而得到

$$W(n) = (n+1)C(n) \approx 2(n+1)\ln n + 2(n+1)\gamma - 4n = O(n \log n)$$

由以上分析可知,快速排序的平均时间性能优于前面介绍的几种排序方法。快速排序空间的开销主要是递归调用时所使用的栈,因此快速排序空间开销和递归调用的栈的深度成正比,故最好的空间复杂度为 $O(\log n)$,最坏的空间复杂度为 $O(n)$ 。

快速排序算法是一种不稳定的算法,当选择好轴元素之后执行交换操作时,可能会破坏原序列中拥有相同值的元素的顺序。例如,对于序列{6, 7, 5, 2, 5, 8},利用第一种分割策略进行分割后得到的序列为{5, 2, 5, 6, 7, 8}。感兴趣的读者可以继续执行分割过程进行验证。

6.4 选择排序

选择排序的基本思想是在当前待排序列中选取关键字最小(最大)的记录作为当前待排序列中的第1个记录。对于具有 n 个元素的待排序列,选择排序则要经过 $n-1$ 趟这样的选择过程。第 i 趟选择的时候从序列中找出关键字第 i 小的元素,并和第 i 个元素交换。而前面介绍的冒泡排序算法的第 i 趟排序则最多需要 $n-i$ 次交换操作,因此简单选择排序算法的交换次数大大减少了。选择排序每次需要从待排序的序列中选出一个最小(最大)的元素,应用不同的查找方法就对应于不同的选择排序算法。本节假设初始待排序数据元素存储在一维数组中。

6.4.1 简单选择排序

简单选择排序算法就是利用线性查找的方法从一个序列中找到最小的元素,即第 i 趟的排序操作为:通过 $n-i$ 次关键字的比较,从 $n-i+1$ 个元素中选出关键字最小的元素,并和第 $i-1$ ($1 \leq i \leq n-1$)个元素交换。简单选择排序算法也称为直接选择排序算法。

简单排序算法的实现见例6.9。

【例 6.9】 简单选择排序算法。

```
template<class T>
void SelectionSort(T Data[],int n)
{
    for(int i = 1; i < n; i++)        //共进行 n-1 趟选择
    { //第 i 趟时的待排序列为 Data[i-1],...,Data[n-1]
        int k = i-1;                //记录当前待排序列的第一个元素
        for(int j = i; j < n; j++)    //找到待排序列中最小元素的下标
        {
            if( Data[j] < Data[k])
                k = j;
        }
        if( k!= i-1)                //交换最小元素到当前待排序列的第一个位置
```

```

    {
        T t = Data[k];
        Data[k] = Data[i-1];
        Data[i-1] = t;
    }
}

```

图 6-9 给出了对序列{10, 5, 7, 8, 6, 9}执行简单选择排序算法的执行流程。从图 6-9 中可以看出, 执行一趟简单选择排序后, 就会将一个元素放到正确的位置上。算法每次都会找到待排序元素的最小值。比如, 第一趟排序结束后, 将最小元素 5 放到下标为 0 的位置处, 第二趟排序结束后, 将次小元素 6 放到了下标为 1 的位置处, 等等。直到将最后一个元素, 也就是最大的元素放到最后一个位置处, 序列得到正确的排序, 整个算法运行结束。

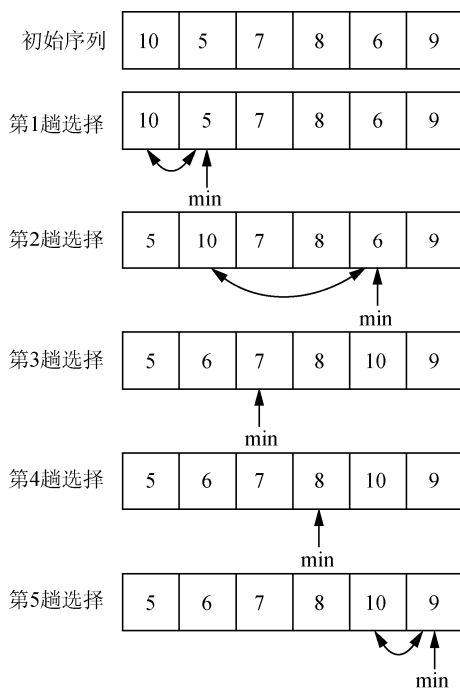


图 6-9 简单选择排序

简单选择排序算法需要进行 $n-1$ 趟选择, 而且第 i 趟选择需要进行 $n-i$ 次比较, 最多执行 1 次数据交换, 最少交换 0 次, 因此简单排序算法的时间效率是 $O(n^2)$ 。简单选择排序算法比较次数较多, 而移动次数较少。空间开销中, 由于只需要使用一个临时变量来记录最小位置, 因此空间复杂度为 $O(1)$ 。简单选择排序算法是不稳定的排序算法,

例如对于序列{6, 2, 6, 3}, 算法执行的结果是{2, 3, 6, 6}。

根据简单选择排序算法的时间性能分析可知, 简单选择排序的主要操作是进行关键字的比较。第 i 趟选择过程一定要进行 $n-i$ 次关键字的比较, 并没有利用第 $i-1$ 趟的比较所得的信息。若能利用前一趟的选择过程中的比较信息, 就可以减少之后各趟选择排序中所用的比较次数。下面介绍的堆排序就充分利用了每一趟选择过程中的比较信息。

6.4.2 堆排序

前一节中介绍的简单选择排序是利用线性查找的方法从一个序列中找到最小的元素。堆排序则是利用堆找到序列中的最大值, 从而实现选择排序。

根据第3章中堆数据结构的定义, 堆数据结构可以被视为一棵完全二叉树, 可以用数组来实现堆的存储。其中对应的二叉树的每个结点和数组中对应位置处的元素对应。堆有两种: 最小堆和最大堆。在最大堆中要求每一个结点对应的值都应该大于或等于该结点的子结点的值 (叶子结点除外); 最小堆则要求每一个对点对应的值都小于或等于该结点的子结点的值 (叶子结点除外)。图 6-10 中给出了一个最大堆的实例。

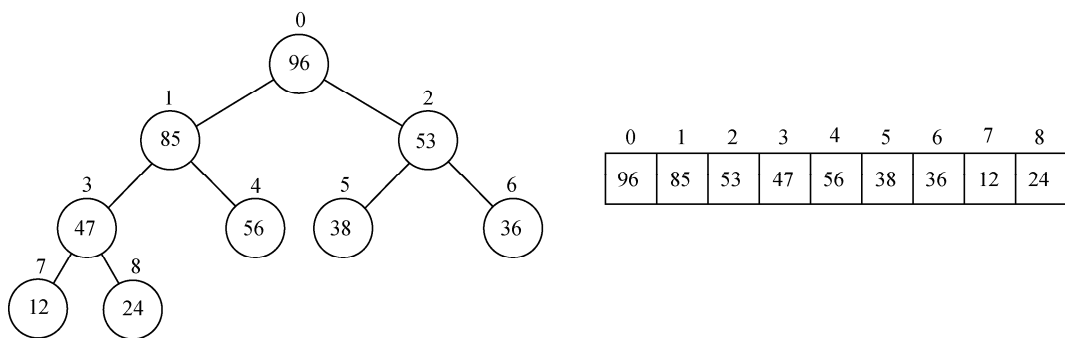


图 6-10 最大堆及其对应的数组

堆排序算法就是用最大堆来得到最大元素, 也就是对应堆的根结点的值。具体步骤是:

- 1) 将初始待排数据初始化为一个最大堆, 初始化当前待排序列的元素个数 n ;
- 2) 将堆顶元素和当前最后一个元素进行交换, $n=n-1$;
- 3) 调整堆结构;
- 4) 如果当前待排序列元素个数 $n>1$ 则重复步骤 2) 和步骤 3)。

图 6-11 给出了对图 6-10 中的最大堆执行堆排序的过程。每次将最后一个元素和堆的根结点进行交换之后, 都要执行一次调整操作。由于以根的两个子结点为根的两棵子树仍然保持最大堆的性质, 只需要调用 SiftDown 操作从堆的根结点向下进行调整即可。

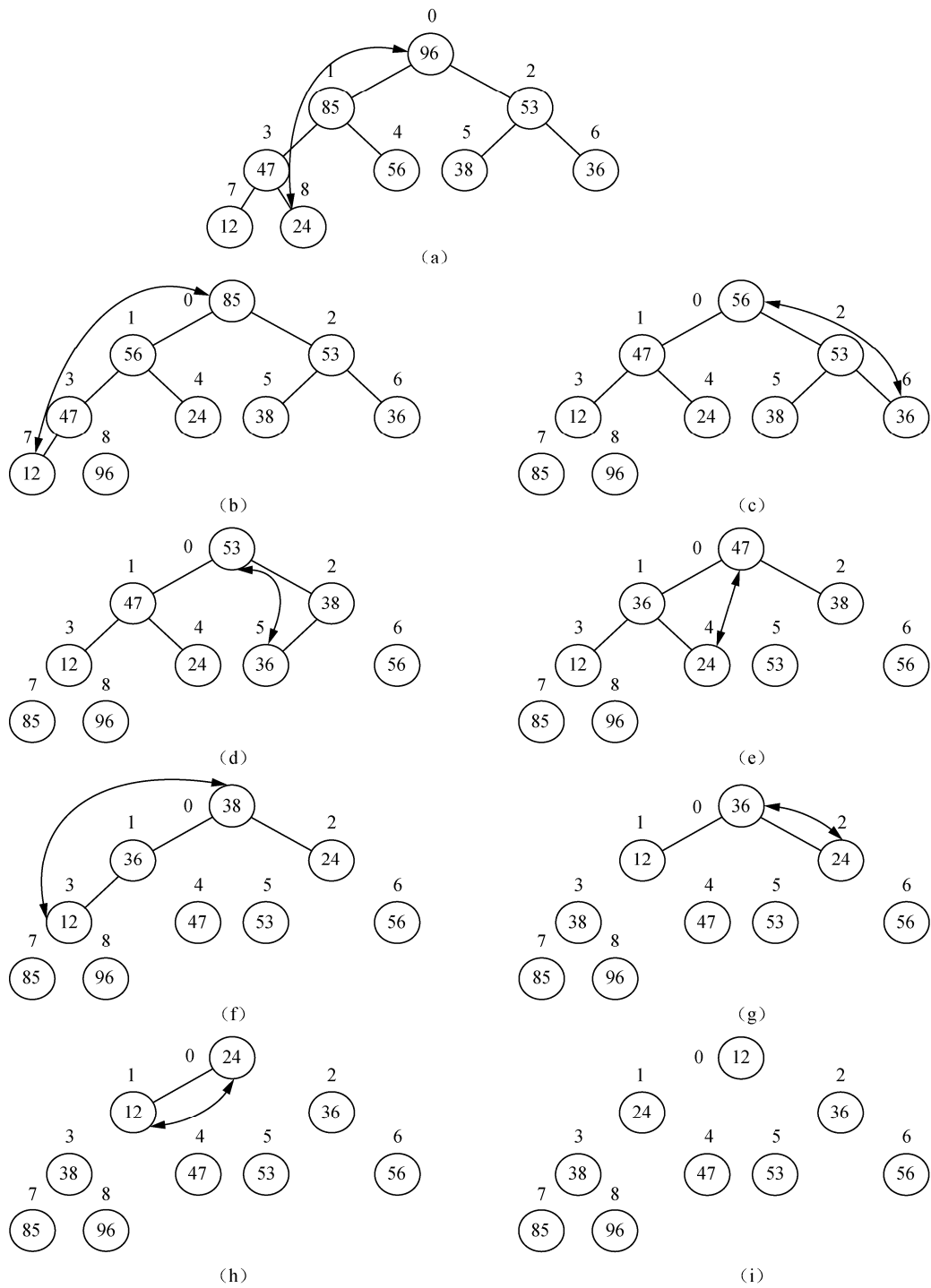


图 6-11 堆排序

根据第3章堆的相关操作的定义,可以容易地给出堆排序中各个步骤的代码实现。

首先定义如下的 **SiftDown** 操作,用来保持以结点 i 为根的最大堆的性质。对于一个结点 i 来说,当它被破坏之后,由于以它的左右子结点为根的树都保持了最大堆的性质,所以需要根据结点 i 以及它的左右子结点的值来对 i 进行调整。然后用同样的方式对可能遭到破坏的子结点为根的子树进行调整。

```
template<class T>
void SiftDown(T Data[], int i, int n)
{//用来保持以结点 i 为根的最大堆的性质, n 是所有元素的个数
    int l = 2*i + 1, r = 2*i + 2, min = i; //找到 i 结点的两个孩子的下标
    if( l < n && Data[min] < Data[l]) //和左子结点进行比较
        min = l;
    if( r < n && Data[min] < Data[r]) //和右子结点进行比较
        min = r;
    if( min != i ) //判断是否需要进行调整
    {
        T t = Data[min];
        Data[min] = Data[i];
        Data[i] = t;
        SiftDown(Data, min, n); //递归对子结点进行调整
    }
}
```

根据 **SiftDown** 操作,很容易完成初始化最大堆的操作。

```
template<class T>
void BuildHeap(T Data[], int n)
{
    int p = n/2 - 1; //求出非叶子结点的最大下标
    for(int i = p; i >= 0; i--)
    {
        SiftDown(Data, i, n); //调用 SiftDown 函数,保持最大堆的性质
    }
}
```

堆排序算法首先调用 **BuildHeap** 操作将输入数组构造为一个最大堆,然后每次循环都找到待排序序列中的最大值,并将该元素和数组中最后一个元素进行交换。这时,最后一个元素就被放到了正确的位置上,因而,对应堆的大小应该减 1,也就是说,堆中的元素始终只包含待排序的元素。由于根结点可能已经不满足最大堆的性质,需要对根结点执行一次调整操作 (**SiftDown**)。

基于上面的各个步骤的实现,堆排序的实现如例 6.10 所示。

【例 6.10】堆排序算法。

```
template<class T>
void HeapSort(T Data[], int n)
```

```

{
    BuildHeap(Data,n);           //首先建立一个最大堆
    for(int i = n-1; i > 0; i--) //进行循环
    {
        T t = Data[0];           //每次取出最大元素后不断调整最大堆
        Data[0] = Data[i];
        Data[i] = t;
        MaxHeap(Data,0,i);
    }
}

```

对于调整最大堆的操作 SiftDown 来说, 最多执行 $O(\log_2 n)$ 次数据元素的交换操作; 初始化堆的时间复杂度为 $O(n)$ 。堆排序算法中共调用了 $n-1$ 次 SiftDown 操作, 以及一次初始化堆的操作, 所以堆排序的时间复杂度为 $O(n\log_2 n)$ 。堆排序过程只需要临时变量来进行交换操作, 故堆排序空间开销为 $O(1)$ 。堆排序是不稳定的排序算法。当数据量较大的时候堆排序的效率体现的很明显, 在小数据集上, 堆排序算法优势不是很明显。

6.5 归 并 排 序

归并排序与快速排序类似, 都是应用分治思想设计的排序算法。与快速排序不同的是, 归并排序使问题的划分策略尽可能简单, 着重于合并两个已排好序的数据序列。归并排序是由 John von Neumann 提出来的, 是计算机中使用的早期排序算法之一。

若一个序列只有一个元素, 则它是有序的, 归并排序不执行任何操作。否则, 归并排序算法利用如下的递归步骤进行排序:

- 1) 先把序列划分为长度基本相等的子序列。
- 2) 对每个子序列归并排序。
- 3) 把排好序的子序列合并为最后的结果。

其中第 3 步中需要将两个子序列合并为一个有序序列, 也就是归并过程。假设有两个已排好序的序列 A (长度为 n_A), B (长度为 n_B), 将它们合并为一个有序的序列 C (长度为 $n_C = n_A + n_B$)。归并用到的方法其实很简单: 把 A, B 两个序列的最小元素进行比较, 把其中较小的元素作为 C 的第一个元素; 在 A, B 剩余的元素中继续挑最小的元素进行比较, 确定 C 的第二个元素, 依次类推, 直到 A 或 B 中所有元素都被添加到序列 C 中, 此时将余下的元素直接添加到序列的最后面, 就可以完成对 A 和 B 的归并。由于 A 和 B 已经排好序了, 每次挑最小元素时, 仅需要比较序列 A 和序列 B 的最前面的元素就可以了, 因而归并过程的复杂度为 $O(n_C)$ 。

类似于第二章介绍的线性表的操作, 将一个序列中的两个有序子序列归并的过程如例 6.11 所示。

【例 6.11】合并一个序列中的两个有序子序列。

```
template<class T>
//函数 Merge, 参数 Data 是待归并数组, 其中对 Data[start,mid]和 Data[mid+1,
//end]
//之间的数据进行归并
void Merge(T Data[],int start,int mid,int end)
{
    int len1 = mid - start + 1,len2 = end - mid;    //分别表示两个归
                                                    //并区间的长度

    int i,j,k;
    T* left = new T[len1];        //临时数组用来存放 Data[start,mid]数据
    T* right = new T[len2];       //临时数组用来存放 Data[mid+1,end]
    for(i = 0; i < len1; i++)    //执行数据复制操作
        left[i] = Data[i+start];
    for(i = 0; i < len2; i++)    //执行数据复制操作
        right[i] = Data[i+mid+1];
    i = 0,j=0;
    for(k = start;k<end; k++)    //执行归并
    {
        if( i == len1 || j == len2)
            break;
        if( left[i] <= right[j])
            Data[k] = left[i++];
        else
            Data[k] = right[j++];
    }
    while( i < len1)//若 Data[start,mid]还有待归并数据, 则放到 Data 后面
        Data[k++] = left[i++];
    while( j < len2)    //对 Data[mid+1,end]间的数据执行同样的操作
        Data[k++] = left[j++];
    delete[] left;    //释放内存
    delete[] right;
}
```

图 6-12 中给出了序列{8, 4, 5, 6, 2, 1, 7, 3}执行归并排序算法时的流程, 其中前 3 步中执行的是分解步骤, 也就是每次都将原始的序列划分为两个子序列。直到第三步中每一个子序列的长度都为 1, 然后执行归并过程。每次都将两个子序列进行归并, 直到所有序列合并为一个序列, 完成归并排序。

基于上面的归并操作, 归并排序算法实现如例 6.12 所示。

【例 6.12】归并排序。

```
template<class T>
void MergeSort(T Data[],int start,int end)
{//对 Data[start]-Data[end]之间的序列进行归并排序
    if( start < end)
```



```

{
    int mid = (start + end)/2;    //计算中间位置
    MergeSort(Data, start, mid); //对左边子序列归并排序
    MergeSort(Data, mid+1, end); //对右边子序列归并排序
    Merge(Data, start, mid, end); //归并左、右两边的有序序列
}
}

```

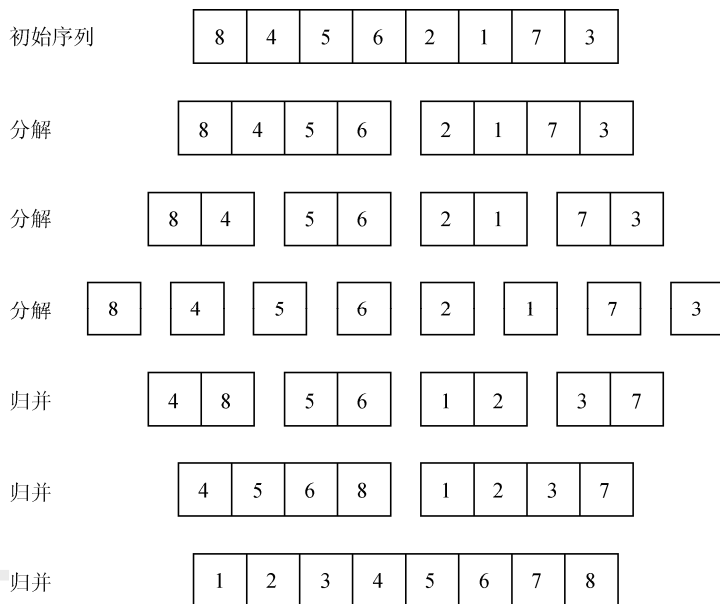


图 6-12 归并排序流程

归并排序算法所需要的时间主要包括分解操作、两个子序列的排序操作和归并操作三部分，由于每一次划分只需要计算划分的位置即可，是一个常数时间，故归并排序算法的时间开销主要由两个子序列的排序操作和对这两个子序列的归并操作确定。

将对长度为 n 的序列执行归并排序的时间复杂度记为 $W(n)$ ，由于归并排序是一个递归算法，因此很容易得到最坏情况下，对两个子序列进行归并排序的算法复杂度分别为 $W(\lceil n/2 \rceil)$ 和 $W(\lfloor n/2 \rfloor)$ 。将两个序列执行归并操作的时间复杂度是和归并后的序列的长度成正比的，可以记为 cn 。因此，算法时间复杂度为 $W(n) = W(n/2) + W(n/2) + cn$ ，且 $W(1) = 0$ ，所以最坏情况下算法的复杂度为 $O(n \log n)$ 。和前面算法相比较，归并排序的算法复杂度比较低，但是在执行归并过程中，算法需要额外的 $O(n)$ 的辅助空间，这是算法的一个缺点。

这里介绍的归并排序中每次迭代都将待排序列分割为两个等长的子序列，因此也称为二路归并排序。归并排序中也可以将待排序列划分为多个子序列，例如，每次可以将序列分割为 3 等分，或 4 等分等。

和快速排序相比，归并排序不需要选择轴元素，归并排序是一个稳定的算法。但由

于需要额外申请辅助空间，实际应用中，归并排序算法的效果往往没有快速排序好。

6.6 比较排序算法的时间复杂度下界

堆排序和归并排序在最坏情况下和平均情况下的时间复杂度都达到了 $O(n \log_2 n)$ ，显然比直接插入排序、冒泡排序、选择排序等算法快得多。那么对于这类 $O(n \log_2 n)$ 级的比较排序算法是否还可以进一步改进呢？本节将讨论比较排序算法的时间复杂度下界。

如果以每次比较作为结点，则每个以比较为基础的排序算法都可以用一个二叉判定树来表示，其中一个中间结点表示一次比较，叶子结点表示排序的一种结果。比如，有一个序列 $\{a, b, c\}$ (a, b, c 互不相等)，则一个通过先比较 a 和 b ，再比较 a 和 c ，最后比较 b 和 c 的排序算法对应的判定树如图 6-13 所示。其中每个结点的左分支表示满足该结点的比较条件，而该结点的右分支表示不满足该结点的比较条件。

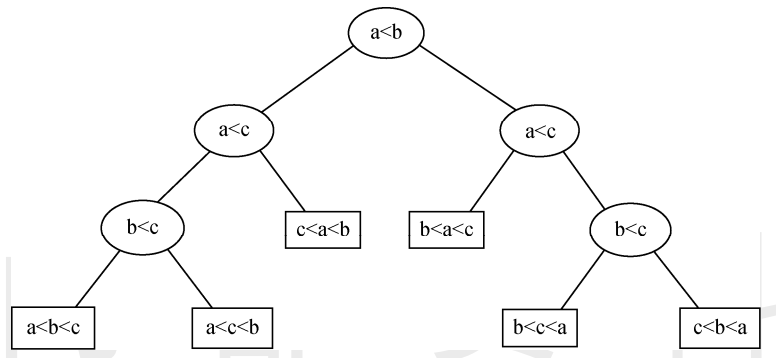


图 6-13 判定树实例

根据图 6-13，假设输入满足 $a < c < b$ ，算法执行路线为 $(a < b) \rightarrow (a < c) \rightarrow (b < c)$ ，共需要 3 次比较。若输入满足 $b < a < c$ ，则算法执行路线为 $(a < b) \rightarrow (a < c)$ 共需要 2 次比较。任何以比较为基础的排序算法都可以表示为一棵判定树，其中树的形状和大小分别表示了排序算法的功能和需要排序的元素个数；而树的高度则表示了算法的运行时间。对于长度为 n 的序列，共有 $n!$ 种不同的排列方式，而每一种排列方式都对应一种排序结果，因此任何排序判定树都有 $n!$ 个叶子。

对于有 $n!$ 个叶子结点的二叉树来说，树的高度至少为 $\lceil \log_2 n! \rceil$ ，因此最坏情况下，比较排序算法至少要做 $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.443n \rceil$ 次比较。平均情况下，比较排序算法的比较次数的下界也是 $\lceil \log_2 n! \rceil \approx \lceil n \log_2 n - 1.443n \rceil$ 。因此基于比较的排序算法的时间复杂度的下界为 $O(n \log_2 n)$ ，也就是说具有 $O(n \log_2 n)$ 复杂度的比较排序算法在渐近意义下是最优的算法。

6.7 基数排序

基数排序是日常生活中经常用到的一种排序方法。例如,为了排序通讯录中联系人,可以根据字母表中字母将联系人分成很多堆,每一堆包含了姓名以相同字母开头的联系人信息。然后,每一堆联系人信息使用相同的方法进行排序,即根据联系人姓名的第二个字母划分堆。重复进行下去,直到划分出的堆的数目等于最长的联系人姓名的字母个数为止。类似的排序还有电子邮件的排序、图书馆的卡片排序等。

基数排序是一种借助多关键字排序的思想对单逻辑关键字进行排序的方法。多关键字排序的一个典型的问题就是对扑克牌进行排序。

已知扑克牌 52 张牌面的次序关系定义为:

花色: $\clubsuit < \diamondsuit < \heartsuit < \spadesuit$

面值: $2 < 3 < \cdots < A$

其中,花色的优先级最高,为主关键字,而面值为次关键字。按照关键字的先后顺序,基数排序算法可以分为两种:高位优先法和低位优先法。

1. 高位优先法 (Most Significant Digit First, MSDF)

在高位优先法中,先按照优先级最高的关键字进行排序,将序列分为若干个子序列,每个子序列中的最高优先级的关键字都是相同的。然后按照次优先级关键字进行排序,将序列划分为更小的若干子序列,依次重复,直到用最低优先级的关键字对子序列进行排序。最后将所有的子序列连接在一起,就成为一个有序序列。

例如,对扑克牌排序来说,若按照高位优先法进行基数排序,则首先按照不同“花色”将扑克牌分成有次序的 4 堆,使得每一堆具有相同的花色;然后分别对每一堆按“面值”大小整理有序。这里也是利用了分治法的思想,对序列进行排序。

2. 低位优先法 (Least Significant Digit First, LSDF)

低位优先法与高位优先法相反,先按照优先级最低的关键字进行排序,依次重复,直至按照最高优先级的关键字排序,序列就会成为有序序列。

对扑克牌排序问题,若利用低位优先法,则首先按“面值”将扑克牌分成 13 堆,称之为分配操作,然后将这 13 堆牌从小到大叠在一起,称之为收集操作。然后将整副扑克牌按照“花色”再次执行分配和收集操作,这时整个扑克牌就会成为一个有序的序列。

一般来说,低位优先法要比高位优先法简单。低位优先法通过若干次的分配操作和收集操作就可以完成排序,执行的次数取决于关键字的多少;而高位优先法在执行分配操作以后要处理各个子集的排序问题,一般是一个递归的过程。

基数排序方法就是将待排序的数据元素的单逻辑关键字拆分成若干个关键字,然后利用 MSDF 或 LSDF 排序。本书仅介绍基于低位优先的基数排序算法。

例如,若关键字是十进制数值,且值域为 $0 \leq K \leq 999$,则可将 K 看作是由 3 个关

键字 $K^0K^1K^2$ 组成。例如，821 可以是由 8 2 1 组成的。

图 6-14 给出了对序列 {278,109,63,930,589,184,505,269,8,83} 执行基数排序的流程。在该示例中，采用的是基于低位优先的基数排序。这十个数字中最高位是百位，故关键字的个数为 3，需要执行 3 次分配和收集操作。第一趟分配时，根据个位数字将数字分配到不同的堆中，然后执行收集操作。执行第二趟和第三趟时，分别根据十位和百位数字执行分配和收集操作。

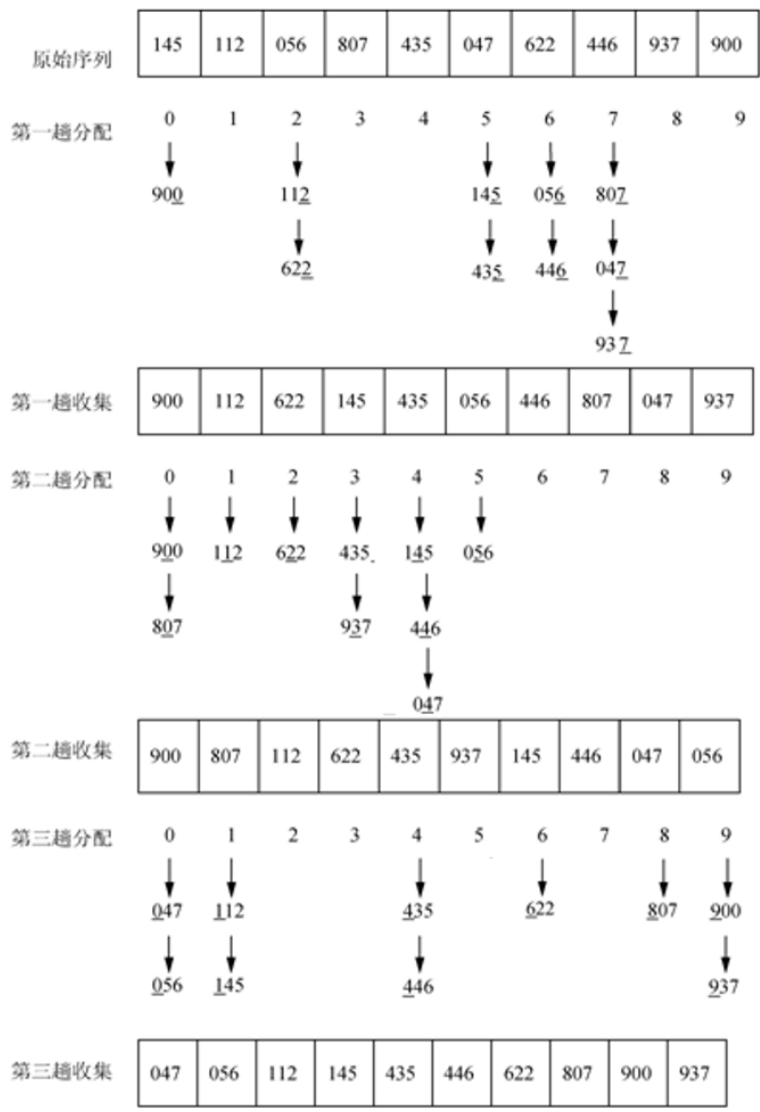


图 6-14 基数排序执行流程

若数据的关键字可以拆分成 d 个关键字，基数排序在执行分配操作时需要将 n 个记

录分配到不同的堆中，第 i 次分配操作时，堆的个数等于第 i 个关键字的属性值的个数 r_i ，但每个堆中元素的个数是不确定的，最多可能是 n 个。通常用数组来保存每个堆中的元素，这时，每个堆都需要分配大小为 n 的数组，此时基数排序算法的空间开销为 m ，其中 $r = \max\{r_1, r_2, \dots, r_d\}$ ，代表关键字属性值个数的最大值。显然这种实现方法空间开销比较大，其中有大量的空间都被浪费了。另外一种常用的方法是基于链式队列来实现堆中元素的存储。可以把这些数据设计成一个队列数组（设队列数组名为 tub ），队列数组的每个元素中包括两个域： $front$ 域和 $rear$ 域。 $front$ 域用于指示队头， $rear$ 域用于指示队尾。当第 i 个队列中有数据元素要放入时，就在队列数组的相应元素 $tub[i]$ 中的队尾位置插入一个结点。图 6-15 中给出了链式队列存储结构的示意图，其中属性值共有 r_i 种可能取值，用这种方法实现时空间复杂度为 $O(n+r)$ 。

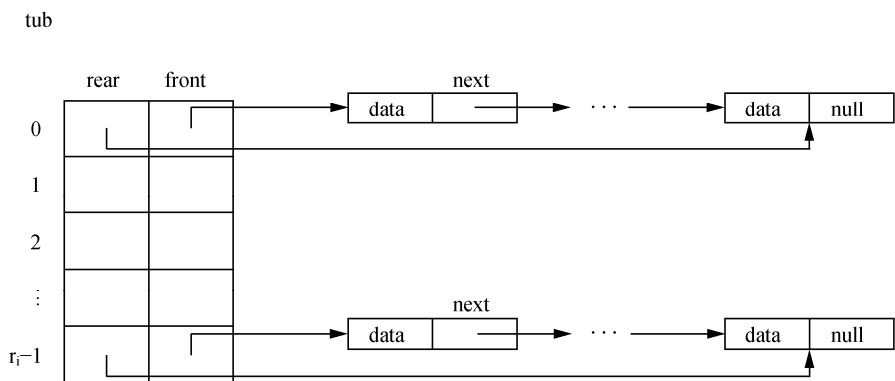


图 6-15 链式队列结构

基数排序不用比较和移动，而用分配和收集操作来进行排序，时间效率高。若关键字的个数记为 d ，则基数排序共需要执行 d 次分配和收集操作，算法的时间复杂度为 $O(dn)$ 。根据基数排序的流程可以看出它是一种稳定的排序算法。

例 6.13 给出了基于低位优先的基数排序的算法实现。

【例 6.13】基于低位优先的基数排序。

```
const int RADIX = 10; //定义基数，用于区分不同进制
template<class T>
struct LinkNode{ //定义链式结构的结点
    T data;
    LinkNode* next;
};
template<class T>
struct TubNode{ //定义队列数组结点
    LinkNode<T>*rear;
    LinkNode<T>*front;
};
/*
```

分配操作

输入: Data[] 数据数组, n 元素个数, ith 第 i 次分配操作

输出: 执行分配操作之后的一个队列数组。

```

*/
template<class T>
TubNode<T>* Distribute(T Data[],int n,int ith)
{
    //申请内存
    TubNode<T>* tube = new TubNode<T>[RADIX];
    //执行初始化
    memset(tube,0,sizeof(TubNode<T>)*RADIX);
    LinkNode<T>* t;
    for(int i = 0; i<n; i++)
    {
        T v = Data[i];
        int j = ith-1;
        while(j-->0)                //求第 i 个位上的数字
            v = v/RADIX;
        v = v%RADIX;
        t = new LinkNode<T>;        //申请新的链表结点
        t->data = Data[i];
        t->next = NULL;
        if( tube[v].front)          //如果对应的队列不为空,则放到末尾
        {
            tube[v].rear->next = t;
            tube[v].rear = t;
        }
        else                        //否则,初始化队列头和尾
        {
            tube[v].front = tube[v].rear = t;
        }

    }
    return tube;
}
/*

```

收集操作, 将 tube 中数据收集到 Data 数组中

输入: Data 数据数组, tube 执行完分配操作之后的队列数组

输出: 无

```

*/
template<class T>
void Collect(T Data[],TubNode<T>* tube)
{
    LinkNode<T>*t,*p;                //临时变量
    int index = 0;                    //数据下标
    for(int i = 0; i<RADIX; i++)
    {

```



```

        p = t = tube[i].front;
        while(t)
        {
            Data[index++] = t->data; //复制对应数据
            t = t->next;
            delete p;                //释放内存
            p = t;
        }
        delete[] tube;              //释放内存
    }

    /*
    基数排序，堆 Data[] 中数据进行排序，并将结果放入到 Data[] 中
    输入：Data[] 数据，n 元素个数，keys 关键字个数
    输出：无
    */
    template<class T>
    void RadixSort(T Data[], int n, int keys) //基数排序
    {
        TubNode<T>* tube;
        for(int i = 0; i < keys; i++)        //循环执行 keys 次分配和收集操作
        {
            tube = Distribute<T>(Data, n, i+1);
            Collect<T>(Data, tube);
        }
    }

```

6.8 各种内部排序算法的比较和选择

本章所介绍的各种内部排序算法的性能比较如表 6-1 所示。

表 6-1 各种排序算法性能比较

排序算法	比较次数			移动次数		辅助空间	稳定性
	最好时间	平均时间	最坏时间	最好情形	最坏情形		
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	0	$O(n^2)$	$O(1)$	稳定
希尔排序		$O(n^{1.3})$		0	$O(n^2)$	$O(1)$	不稳定
直接选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	0	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(\log_2 n)$	$O(n^2)$	$O(1)$	不稳定
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	0	n	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$		$O(\log_2 n)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$		$O(n)$	稳定
基数排序 (基于链式队列)	$O(dn)$	$O(dn)$	$O(dn)$			$O(n)$	稳定
基数排序(基于 顺序队列)	$O(dn)$	$O(dn)$	$O(dn)$			$O(n)$	稳定

在实际应用中,选择有效的排序算法时需要考虑算法的时间开销、空间开销,以及是否稳定等诸多因素,通常可以依据以下原则:

1. 从时间复杂度选择

对元素个数较多的排序,可以选择快速排序、堆排序、归并排序;元素个数较少时,可以选简单的排序方法。

一般情况下简单的排序方法比复杂的方法只有 20%的效率损失。如果在程序中只用到一次排序,并且只是对很小的数据集进行排序,那么使用复杂且效率较高的算法可能并不值得。但是如果有大量的数据需要进行排序,那么 20%的效率是不能被忽视的。对于一些数量较少的数据来说,简单的算法常常比复杂的算法执行的更好,只是在数据规模很大时,复杂算法的效率优势才能明显体现出来。

2. 从空间复杂度选择

尽量选择空间复杂度为 $O(1)$ 的排序方法,其次选择空间复杂度为 $O(\log_2 n)$ 的快速排序方法,最后才选空间复杂度为 $O(n)$ 的二路归并排序的排序方法。

3. 一般选择规则

1) 当待排序元素的个数 n 较大,排序的关键字随机分布,而且对稳定性不做要求时,采用快速排序为宜。

2) 当待排序元素的个数 n 较大,内存空间允许,且要求排序稳定时,采用二路归并排序为宜。

3) 当待排序元素的个数 n 较大,排序的关键字分布可能会出现正序或逆序的情形,且对稳定性不做要求时,采用堆排序或二路归并排序为宜。

4) 当待排序元素的个数 n 较小,元素基本有序或分布较随机,且要求稳定时,采用直接插入排序为宜。

5) 当待排序元素的个数 n 较小,对稳定性不做要求时,则采用直接选择排序为宜;若排序的关键字不接近逆序,也可以采用直接插入排序。冒泡排序一般很少采用。

6.9 外 部 排 序

前面介绍的各种排序算法都是内部排序算法,即待排序的数据全部存储在内存中。但是当数据规模很大时,内存是远远不够的,这时就要利用外部排序方法来完成排序。

外部排序指的是大文件的排序,即待排序的记录存储在外存储器上,待排序的文件无法一次装入内存,需要在内存和外部存储器之间进行多次数据交换,以达到排序的目的。对外存文件中的数据进行排序后的结果仍然被放到原有文件中。

一般来说外部排序分为两个阶段:预处理和归并处理。在预处理阶段,根据内存的

大小将有 n 个记录的磁盘文件分批读入内存，采用有效的内存排序方法进行排序，将其预处理为若干个有序的子文件，这些有序子文件成为“初始顺串”。在归并阶段，利用归并的方法将这些初始顺串逐趟合并成一个有序文件。

6.9.1 置换选择排序

预处理阶段通常使用置换选择排序来生成初始顺串。置换选择排序是堆排序的一种变形，能够在对数据文件扫描一遍的前提下，使得所生成的各个初始顺串有更大的长度，从而减少了初始顺串的个数，有利于在合并时减少对数据的扫描遍数。置换选择排序算法描述如下：

1. 初始化堆

- 1) 从磁盘读入 M 个记录放到内存数组中。
- 2) 设置堆末尾标准 $LAST=M-1$ 。
- 3) 建立最小堆。

2. 重复以下步骤直到堆为空，即 $LAST=-1$

- 1) 把具有最小关键字的记录 Min 也就是根结点送到输出缓冲区。
- 2) 设 R 是输入缓冲区中的下一条记录，如果 R 的关键字大于刚刚输出的关键字 Min ，则把 R 放到根结点；否则使用数组中 $LAST$ 位置的记录代替根结点，将 R 放入到 $LAST$ 所在位置，并且设置 $LAST=LAST-1$ 。
- 3) 重新调整堆。

例如文件中存储的数据为 28, 94, 96, 11, 35, 17, 12, 75, 15, 58, 41, 81, 99, 10, 8。设内存数组的大小为 5，则利用置换选择排序生成第一个初始顺串为 $Ta=\{11, 17, 28, 35, 75, 94, 96\}$ ，其过程如图 6-16 所示。生成第一个顺串后，要重新将数组调整为最小堆，并置 $LAST$ 为 4，重复图 6-16 的过程就可以将剩余的数据生成初始顺串 $Tb=\{12, 15, 41, 58, 81, 99\}$ 、 $Tc=\{8, 10\}$ 。

如果存储堆的数组大小是 M ，则一个顺串的最小长度就是 M 个记录，因为至少原来在堆中的那些记录将成为顺串的一部分。例如当输入的数据是逆序时，那么顺串的长度只能是 M 。生成初始顺串的最好情况是输入为正序时，则一次性就能把整个文件生成一个顺串。因此生成的初始顺串的长度是大小不一的，但是每个顺串都是有序的，利用图 6-17 所示的扫雪机模型可以预计平均情况下初始顺串长度是数组长度的两倍，即为 $2M$ 。

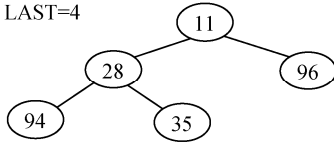
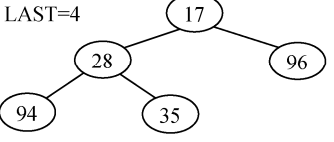
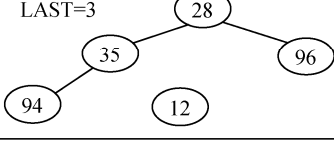
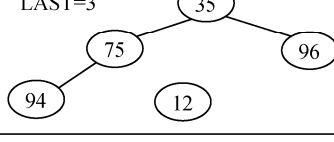
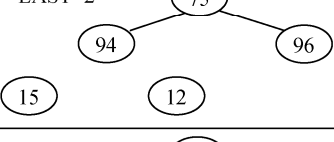
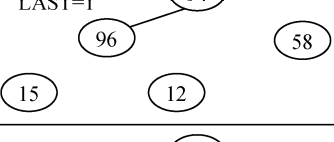
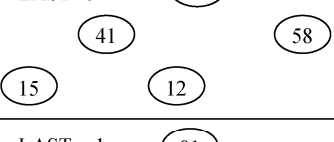
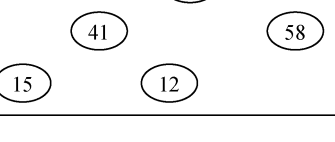
输入	存储	输出
17	LAST=4 	11
12	LAST=4 	17
75	LAST=3 	28
15	LAST=3 	35
58	LAST=2 	75
41	LAST=1 	94
81	LAST=0 	96
	LAST=-1 	

图 6-16 置换选择排序示例

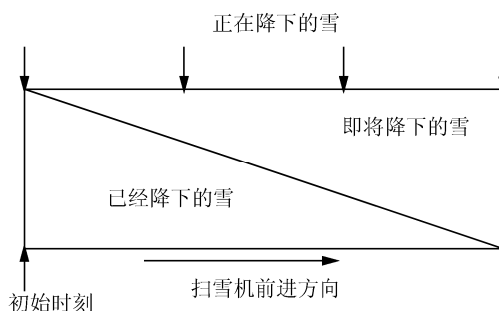


图 6-17 扫雪机模型分析

6.9.2 多路归并

归并阶段，是将多个初始顺串归并成一个有序序列。例如，对上面的初始顺串 $Ta=\{11, 17, 28, 35, 75, 94, 96\}$ 、 $Tb=\{12, 15, 41, 58, 81, 99\}$ 、 $Tc=\{8, 10\}$ ，可以利用二路归并，先将 Ta 和 Tb 归并成一个有序串 $\{11, 12, 15, 17, 28, 35, 41,$

$58, 75, 81, 94, 96, 99\}$ ，然后再和 Tc 归并得到最后的有序序列，如图 6-18 所示。其中每个结点表示外存上一个文件，将 Ta 和 Tb 归并成文件 Tab 时需要分别从 Ta 和 Tb 中顺序读取数据，归并排序后写入到文件 Tab 中，这称为一趟归并；然后将 Tab 与 Tc 中的数据顺序读取出来并归并排序后写入到文件 $Tabc$ 中。

如果初始顺串的个数为 m 个，那么采用二路归并时至少需要 $\log_2 m$ 趟归并过程，每一趟归并都要读写文件，效率较低。在实际应用中通常采用多路归并，这样可以减少扫描次数。在多路归并中，最直接的方法就是经过多次比较来找出所要的记录，时间代价较大。一般情况下使用选择树来进行多路归并。如果要归并 m 个顺串，则选择树是含 m 个叶结点， $m-1$ 个内部结点的完全二叉树，有赢者树和败者树两种类型。

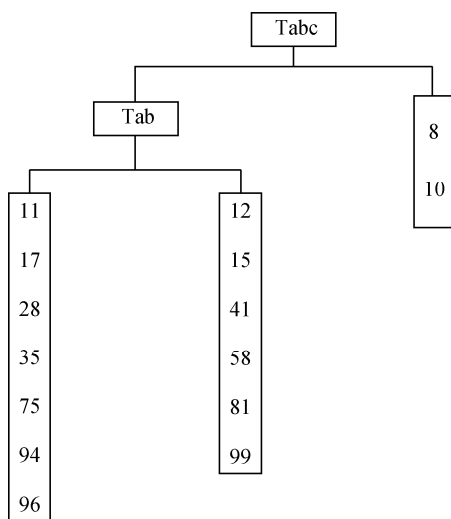


图 6-18 利用二路归并归并初始顺串

1. 赢者树

赢者树的叶子结点是初始顺串中当前要处理的数据，内部结点是其左右子结点的最小值所在的顺串的索引（假设从小到大排序）。图 6-19 是一棵实现 5 路归并的赢者树，图中的方形结点表示叶结点，分别为 5 个初始顺串中当前参加归并选择的记录关键字；图中圆形结点表示内部结点，分别为其子结点中赢者所对应的顺串索引。因此，树的根

结点就是最终的赢者的索引，即为下一个要输出的记录所对应的顺串索引。例如，根据图 6-19，下一个要输出的是顺串 4 的当前记录。输出该记录后，顺串 4 的下一个记录成为顺串 4 的当前记录，相应的关键字将进入赢者树，这时需要重新调整赢者树。

调整赢者树时，只需要沿着从发生改变的叶结点到根结点的路径进行调整。调整过程如下：

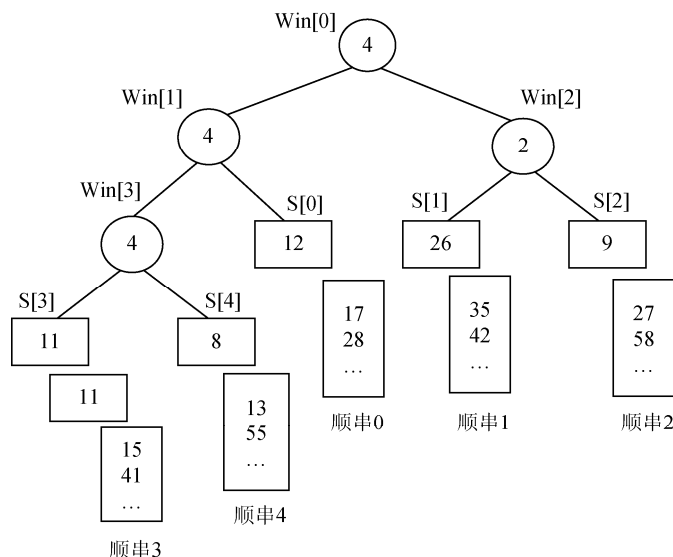


图 6-19 实现 5 路归并的赢者树

1) 将发生改变的叶结点作为当前结点 p 。

2) 如果 p 是根结点则调整结束。

否则，设 p 的兄弟结点为 q ，如果 q 不空，则比较 p 和 q 标记的关键字，用小的关键字所对应的索引修正 p 、 q 的父结点 pre 的值；如果 q 空，则用 p 所对应的索引修改父结点 pre 的值；

3) $p=pre$ ；重复执行 (2)。

对于图 6-19，当输出顺串 4 中的记录 8 后， $S[4]$ 的值改变为顺串 4 的下个记录 13，按照调整步骤，由于此时顺串 3 的当前记录 11 小于顺串 4 的当前记录 13，因此将 $S[4]$ 的父结点 $Win[3]$ 调整为 3；并继续调整 $Win[1]=3$ ， $Win[0]=2$ 。调整后的赢者树如图 6-20 所示。

2. 败者树

败者树是另外一种选择树，其结构与赢者树类似。如果来实现从小到大排序，则将两个关键字中较大的称为败者。在败者树中，内部结点是其左右子树中推选出的赢者进行比较后的败者对应的顺串的索引。如图 6-21 就是一棵败者树，叶结点是各个顺串当前参与比较的记录；内部结点 $Lost[4]$ 是其左右两个子结点 $S[3]$ 、 $S[4]$ 比较后败者所对应的顺串索引，即 3。这一轮比较后向上推选出的赢者为 $S[4]$ ；同样 $Lost[3]$ 是其左右两个

叶结点 $S[1]$ 、 $S[2]$ 比较后败者所对应的顺序索引 1，并向上推选赢者 $S[2]$ ； $Lost[2]$ 是其左子树推选出的赢者 $S[4]$ 于其右边叶结点 $S[0]$ 比较后的败者所对应的顺序索引 0，继续向上推选赢者 $S[4]$ ； $Lost[1]$ 是其左右子树推选出的赢者 $S[4]$ 、 $S[2]$ 比较后的败者所对应的顺序索引 2，继续向上推选赢者 $S[4]$ ； $Lost[0]$ 的值就是最后推选出的赢者对应的顺序索引 4。

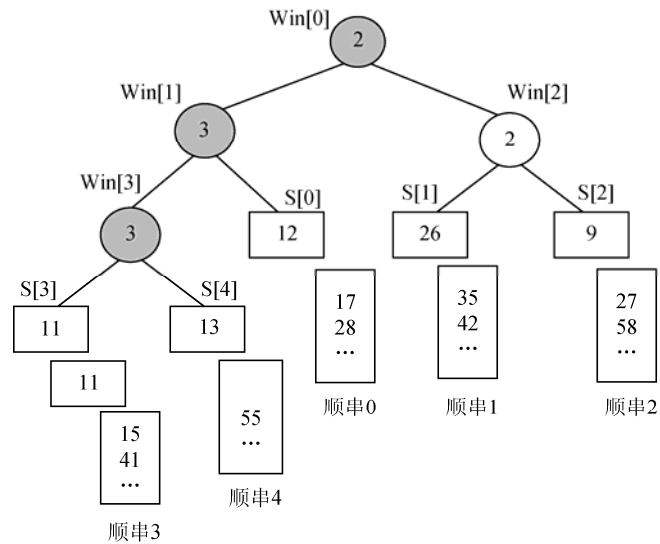


图 6-20 调整后的赢者树

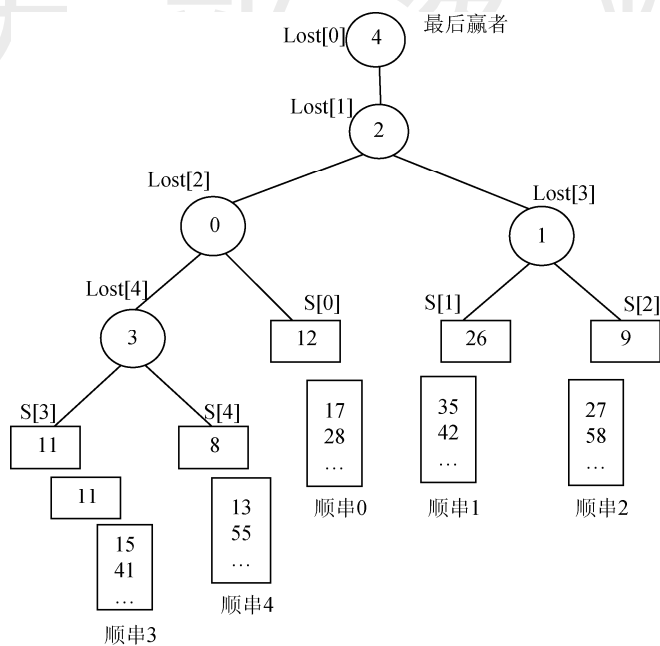


图 6-21 实现 5 路归并的败者树

败者树中 $Lost[0]$ 的值就是下一个要输出的记录所对应的顺串索引。输出该记录后，相应顺串的下一个记录成为新的当前记录，相应的关键字将进入败者树，这时也需要重新调整败者树。调整过程也是沿着从发生改变的叶结点到根结点的路径进行调整。调整过程如下：

- 1) 设发生改变的叶结点的父结点为 $Lost[i]$ 。
- 2) 如果 $i > 0$ ，则比较 $Lost[i]$ 的左右子树推选出的赢者，将败者索引作为 $Lost[i]$ 的值，将赢者继续向上推选；否则，则以败者树推选出的最后赢者索引作为 $Lost[0]$ 的值，结束调整。
- 3) $i = i/2$ ；重复执行 (2)。

对于图 6-21，当输出顺串 4 中的记录 8 后，叶结点 $S[4]$ 的值改变为 13，按照败者树的调整步骤，比较 $Lost[4]$ 的左右子树推选出的赢者 ($S[3]$ 、 $S[4]$)，将败者索引 4 作为 $Lost[4]$ 的值，并向上推选 $S[3]$ ；继续比较 $Lost[2]$ 的左右子树推选出的赢者 ($S[3]$ 、 $S[0]$)，将败者索引 0 作为 $Lost[2]$ 的值，并向上推选 $S[3]$ ；继续比较 $Lost[1]$ 的左右子树推选出的赢者 ($S[3]$ 、 $S[2]$)，将败者索引 3 作为 $Lost[1]$ 的值，并向上推选 $S[2]$ ；最后将 $Lost[0]$ 的值调整为 2，产生下一个要输出的记录对应索引。调整后的败者树如图 6-22 所示。

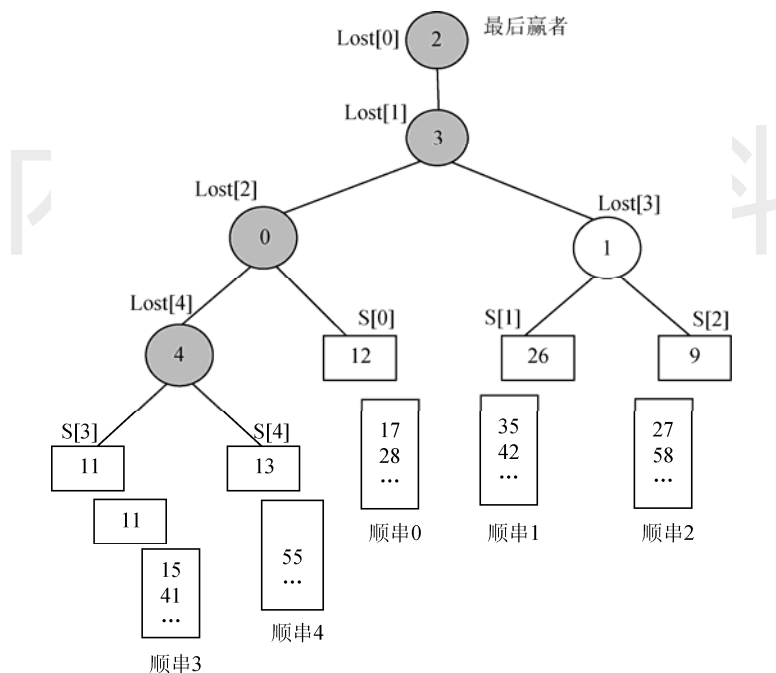


图 6-22 调整后的败者树

如果对 m 个顺串使用 k 路赢者树或败者树来归并，则需要 $\log_k m$ 趟归并。每一趟归并需要初始化含 k 个叶结点的赢者树或败者树，时间复杂度为 $O(k)$ ；读入一个新的关键

字并调整的时间为 $O(\log k)$ 。所以归并产生 n 个元素的顺串的时间复杂度为 $O(k+n\log k)$ 。

6.10 排序的应用：书库信息排序

1. 需求描述

在现代社会当中，生活节奏越来越快，生活中的所有事物同样也都在提速来满足人们日新月异的需求。那么在具有大量书籍信息的书库中，怎样才能快速找到用户满意的图书呢？有的用户可能希望按照书名检索，有的用户可能需要按照书目的发行日期查找，还有的用户可能需要买一本价格尽可能低的书等。本节介绍一个简单的书库系统，实现多种排序供用户检索。

本应用需要构建一个书库，保存所有待检索的书籍。另外可以完成如下的排序：

- 1) 按照书籍的 ISBN 的字典顺序打印书籍信息。
- 2) 按照书籍的价格从小到大打印书籍信息。
- 3) 按照书籍的发行日期排序后打印书籍信息。

最后，本应用还需要构建一个书籍文件来存储测试所用的一些书籍信息，来测试应用是否能够良好的工作。

2. 问题分析

本应用主要涉及排序算法、读写文件知识以及书籍、书库的概念。根据需求分析，书籍信息需要包含一个发行日期的信息；书库用来保存所有的图书信息，可以向一个书库添加一本书籍，并且书库可以将书籍信息展示给用户。由于不同用户可能希望以不同的排序方式看到所有的书籍信息，因此，书库可以实现以下几种排序方式：

1) 按照发行日期排序。书库默认以图书的发行时间为顺序来整理、保存书籍，后发行的书籍通常入库时间也比较晚，因此如果按照发行日期对书库中的书籍排序，使用插入排序比较合理。

2) 按照 ISBN 号排序。目前书籍的 ISBN 号都是十位数字串，即 ISBN 号的长度是固定的，且每一位上的字符变化范围都为 0~9，这恰好符合基数排序的条件，因此使用基数排序对书籍按照 ISBN 号排序是合理的。

3) 按照书价格排序。书库中书籍的价格一般都是随机的，因此选择随机情况下性能很好的排序算法——快速排序来对书籍进行排序。

程序运行之前，先建立一个文本文件来存放书籍信息（模拟要入库的书籍），其中每一行存放一本书籍信息。程序运行时，首先建立一个书库，然后逐条读入文件中的书籍信息，并添加到书库当中，添加书籍的过程可以看成是插入排序的过程。

3. 概要设计

本应用需要定义日期类（Date）、书籍类（Book）、书库类（BookList）。其中日期类

包含年 (year), 月 (month) 以及日 (day) 三个字段。为方便存取等操作, 需要重载输入输出流操作符 (>>和<<)。另外, 在按照发行日期排序时, 要对日期进行大小比较, 因此还应重载比较运算符 (>或者<)。

书籍类包含书名 (name), 价格 (price), 日期 (Date), ISBN 号 (ISBN) 几个字段。书籍在书库中以双向链表的方式存储, 所以书籍类中要有两个指针域 (next、pre)。为了方便操作, 书籍类重载了<<运算符。

对于书库类, 其本质是维护一个关于书籍指针的双向链表, 并存储了其头指针 (head) 和尾指针 (tail) 两个结点。此外, 书库类需要提供按照发行日期、价格、ISBN 号以及书名进行排序的操作。书库类还要提供打印所有书籍信息操作。

各个类定义及类之间的关系如图 6-23 所示。

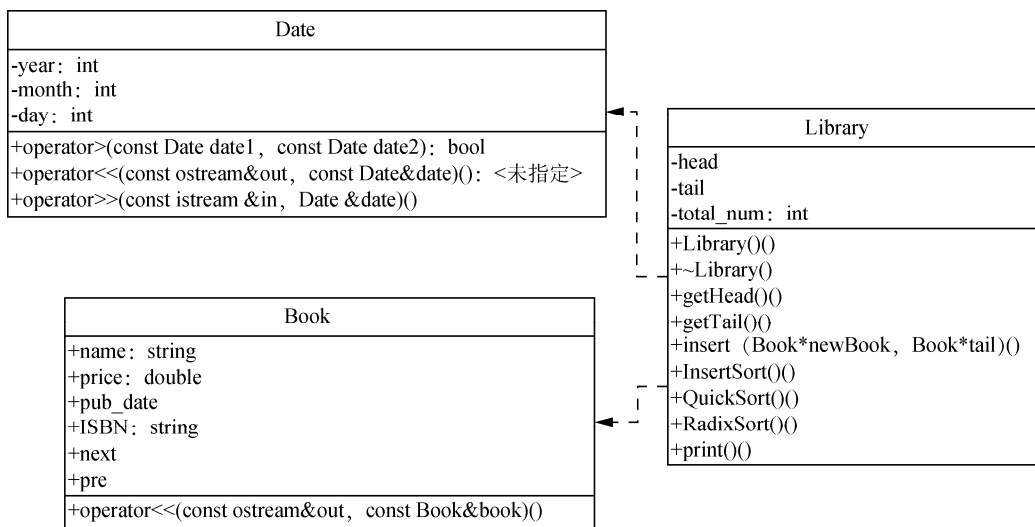


图 6-23 书库信息排序应用中的类及类之间的关系

4. 详细设计

下面分析本应用及主要功能的实现流程。

本应用在书库初始化及书籍信息加载完成之后, 接收用户输入的命令, 解析后做出相应的反馈, 其流程如图 6-24 所示。通过一个循环控制来接收并解析用户的命令: 如果用户选择退出, 则退出应用。否则, 如果用户选择了按照以日期的方式排序方式查看书籍信息, 则书库 Lib 会调用插入排序 InsertSort()函数对书籍进行排序; 如果用户选择了以 ISBN 排序方式查看书籍信息, 则书库 Lib 调用 RadixSort()函数对书籍进行排序; 如果用户选择了以按照价格排序方式查看书籍信息, 则书库 Lib 会调用其 QuickSort()函数对书籍进行排序, 最后, Lib 调用其打印函数 print()将按照一定规则排好序的书籍信息打印出来。

加载书籍信息涉及文件读写以及插入排序的过程, 具体如图 6-25 所示。

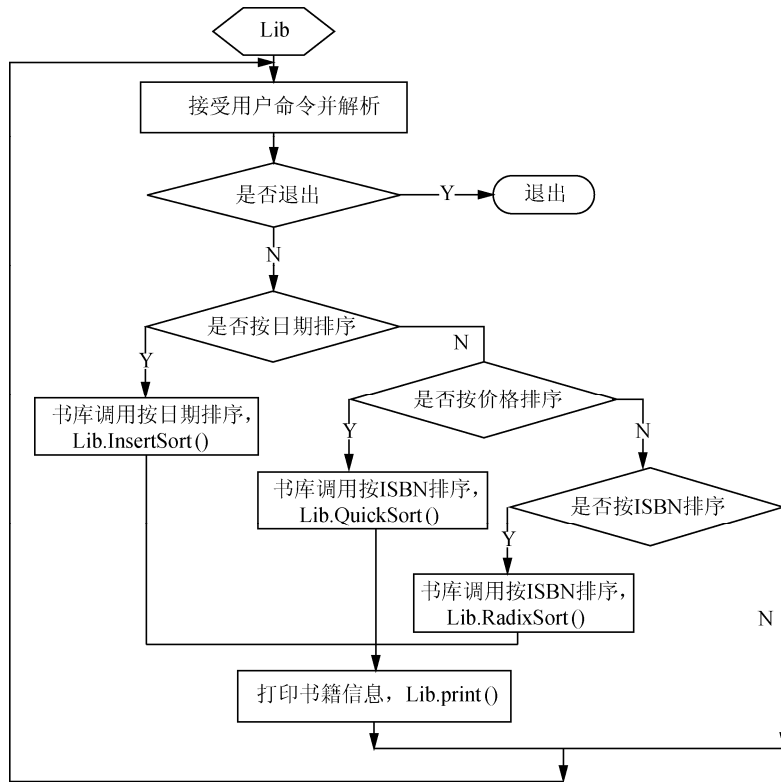


图 6-24 书库信息排序应用的主要功能流程

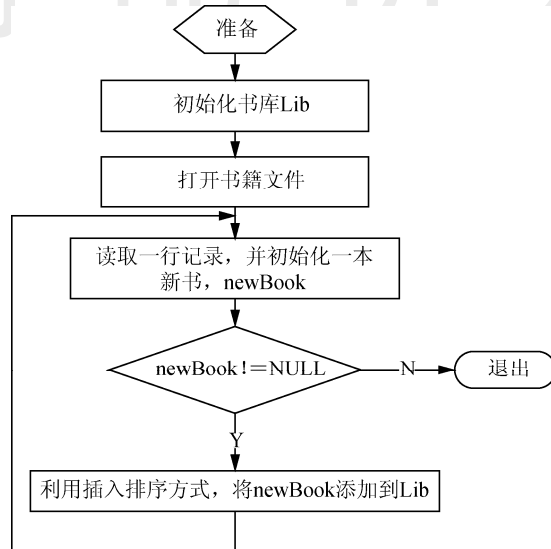


图 6-25 书籍信息加载流程

关于本应用涉及的插入排序、快速排序及基数排序的理论和实现请参见本章的相关知识。

习 题

1. 分别使用下面的排序算法将序列 60, 40, 120, 185, 20, 135, 150, 130, 45 按非减序排序。

(1) 直接插入排序; (2) 冒泡排序; (3) 快速排序; (4) 堆排序; (5) 归并排序。

2. 写出使用增量序列 {1, 3, 7} 对输入数据 9, 8, 6, 5, 4, 3, 2, 1, 0 运行希尔排序得到的结果。

3. 用基数排序对下列字符串进行排序 fghe, ef, abc, degh, fg, ba, a, uvw, xyzfg。

4. 如果不使用递归, 则如何实现归并排序?

5. 分析下面四种排序算法在待排数据相等、正序和逆序时的时间性能。

(1) 直接插入排序; (2) 希尔排序; (3) 快速排序; (4) 归并排序。

6. 设计并实现一个有效的对 n 个整数重排的算法, 使得所有负数位于非负数之前, 给出算法的性能分析。

7. 试给出一个同时找到 n 个元素中最大元素与最小元素的有效算法, 并说明理由。

8. 如果待排数据有 5 个元素, 基于比较的排序算法至少需要多少次比较? 如果待排数据有 7 个元素, 基于比较的排序算法至少需要多少次比较? 分别给出相应算法的实现。

9. 设输入文件数据 (101, 51, 19, 61, 3, 71, 31, 17, 19, 100, 55, 20, 9, 30, 50, 6, 90); 可以利用的内存数组大小为 6, 写出使用置换-选择算法生成的初始顺序; 写出实现 6 路归并的初始赢者树和败者树。

10. 多项式相加是常用的代数操作。一个熟悉的规则是, 如果两个数据项包含相同幂次的变量, 则相加, 得到的数据项仍保持原来的变量和幂次, 但其系数等于两个数据项的系数之和。如: $2x^3y + 4x^3y = 6x^3y$ 。为了便于相加, 首先要对数据项中的变量排序, 然后再进行合并相加。请给出实现。