

第 8 章 排序	1
8.1 概述	1
8.1.1 排序及其分类.....	1
8.1.2 排序算法的分析指标.....	2
8.2 插入排序.....	2
8.2.1 直接插入排序.....	3
8.2.2 希尔排序.....	5
8.3 交换排序.....	6
8.3.1 冒泡排序.....	6
8.3.2 快速排序.....	8
8.4 选择排序.....	11
8.4.1 直接选择排序.....	12
8.4.2 堆排序.....	13
8.5 归并排序.....	23
8.5.1 归并.....	23
8.5.2 归并排序.....	24
本章小结	27

第 8 章 排序

8.1 概述

排序是日常工作和软件设计中最常用的运算之一。例如，在每年的高考之后，为了能确定录取的分数线，需要对所有考生的分数进行排序；对电话号码簿之类的数据表排序可以更快地实现查找。由于需要排序的数据表的基本特性可能存在差异，使得查找方法也相应地有所不同。本章介绍几种最常见的排序方法，并讨论其性能和特点，在此基础上进一步讨论各种方法的适用场合，以便在实际应用时能根据具体问题选择合适的排序方法。

8.1.1 排序及其分类

所谓**排序**（ Sorting ）就是将数据表调整为按**关键字**从小到大或从大到小的次序排列的过程。

对一个数据表来说，不同的要求可能会选择不同的字段作为其关键字，如在档案表中，职务、职称、年龄等均可作为关键字来排序。

排序的要求和方法较多，对此有不同的分类方法，下面先介绍有关的排序分类方法。

(1) **增排序和减排序**

如果排序的结果是按关键字从小到大的次序排列的，就是增排序，否则就是减排序。

(2) **内部排序和外部排序**

如果在排序过程中，数据表中的所有数据均在内存中，则这类排序为内部排序，否则为外部排序。

许多读者在学习程序设计语言中所接触过的排序大多是在数组中进行的，而数组是保存在内存中的，所以那些排序就是内部排序。在一些场合下，数据表中的内容可能较多，超出数组的存储容量，如某省的高考成绩数据库。在这种情况下，排序过程中就需要将部分数据存放在外部存储器中，另一部分数据放在内存中排序，在将内存中的部分数据排序完毕后再保存到外部存储器中，然后重新调出另外的数据来排序。这一过程要反复进行，直到全部排出次序为止。这就是外部排序。考虑到其难度，本教材中没有讨论有关外部排序的内容。

(3) **稳定排序和不稳定排序**

在排序过程中，如果关键字相同的两个元素的相对次序不变，则称为稳定的排序，否则是不稳定的排序。

稳定排序的概念在一些比赛或选举中可能会涉及到：如一个单位在投票选举某个岗位时有现任和一个新的竞选者，如果两人的得票数相同，应选谁呢？如果采用稳定的方式，则应是现任留任，否则就选用新的竞选者。

(4) **排序的基本方法**

虽然存在多种排序算法，但按照各算法所采用的基本方法可将其划分为：**插入排序、交换排序、选择排序、归并排序和基数排序**。本章以这些基本方法为线索来介绍有关排序的算法。

8.1.2 排序算法的分析指标

与许多算法一样，对各种排序算法性能的评价同样侧重于其时间性能和空间性能方面，对某些算法还可能要涉及到其他一些相关性能的分析。

1. **时间性能分析**

在分析排序算法的时间性能时，主要以算法中用得最多的基本操作的执行次数（或者其数量级）来衡量，这些操作主要是比较元素、移动或交换元素。在一些情况下，可能还要用这些次数的平均数来表示。

2. **空间性能分析**

排序算法的空间性能主要是指在排序过程中所占用的辅助空间的情况，即是用来临时存储数据的内存的情况，在特殊情况下还可能指用于程序运行所需要的辅助空间。

8.2 插入排序

插入排序算法的**基本思想**是：将待排序表看作是左右两部分，其中左边为有序区，右边为无序区，整个排序过程就是将右边无序区中的元素逐个插入到左边的有序区中，以构成新

的有序区。本节介绍基于这一思想的两个排序算法，即直接插入排序算法和希尔排序算法。

8.2.1 直接插入排序

直接插入排序是插入类排序算法中较简单、直接的排序方法，**基本思想**是：将整个待排序表看作是左右两部分，其中左边为有序区，右边为无序区，整个排序过程就是将右边无序区中的元素逐个**插入**到左边的有序区中，以构成新的有序区。

下面先讨论插入一个元素到有序区中的操作的实现，在此基础上讨论整个排序算法。

假设当前数据表左边的有序区中已经有 $i-1$ 个元素了（下面用方括号表示有序区），现在要将无序区中的第一个元素（即整个表的第 i 个元素） a_i 插入到该有序区中，以构成新的有序区，如下所示。

$([a_1, a_2, \dots, a_{i-1}], a_i, \dots, a_n)$

从功能上说，在往有序区中插入元素以构成新的有序区时，需要完成如下操作：

- ① 搜索插入的位置；
- ② 移动元素以腾出空位；
- ③ 插入元素 a_i 。

其中步骤 ③ 的实现自然是简单的。对其中的步骤 ①，即搜索插入位置的实现过程类似于本书中另外章节所介绍的查找运算，可用顺序查找或二分查找方法来实现。考虑到移动元素只能从后往前逐个进行。因此，为节省运算时间，可将搜索和移动元素放在一起同步进行，即从后往前顺序地搜索和移动元素。由此可得其粗略描述如下：

```
temp=A[i];    //用临时变量 temp 保存元素值，以腾出 A[i]的空间
j=i-1;        //用变量 j 依次指示子表中的元素，其初值是当前空位置的前一个元素
while (j>=1 && A[j].key>temp.key) //从后往前搜索插入位置并腾出空位
{
    //当前面的元素大于待插入元素时要后移
    A[j+1]=A[j];
    j=j-1;
}
A[j+1]=temp;    //插入元素
```

下面讨论完整的排序方法：

显然，开始排序时的有序区中最多只能保证有一个元素，因而需将下标为 $2 \sim n$ 的元素依次插入到有序区中，即要进行 $n-1$ 次插入操作。由此可得完整的排序算法如下：

```
void insertSort(elementType A[n+1])
{
    for (i=2; i<=n; i++)    //i 表示待插入元素的下标
    {
        temp=A[i];        //临时保存待插入元素，以腾出 A[i]空间
        j=i-1;            //j 指示当前空位置的前一个元素
        while (j>=1 && A[j].key>temp.key) //搜索插入位置并腾出空位
        {
            A[j+1]=A[j];
            j=j-1;
        }
    }
}
```

```

        A[j+1]=temp;                //插入元素
    }
}

```

图 8-1 为插入排序过程示例：

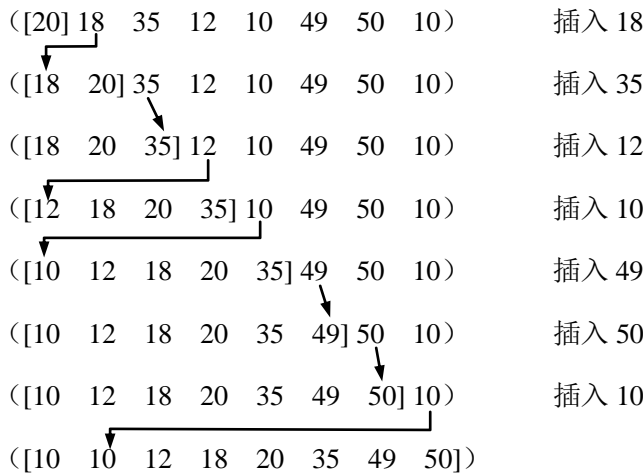


图 8-1 直接插入排序过程示例

上述直接插入排序算法虽然比较简洁易懂，然而，其时间性能不够理想：因为在比较每个元素时，都要先判断其下标是否越界。采用“**监视哨**”的方法就可以省略这一需要对每个元素都要执行的判断：

将数组最前面的元素 $A[0]$ 作为**监视哨**，用以临时存放待插入的元素 $A[i]$ 的值（因此，临时变量 `temp` 就可以由 $A[0]$ 来代替了）。这样，就要用该元素来搜索插入位置了。如果数组中的元素都比待插入元素大，则会比较到监视哨（事实上，此时是和自己比较），并且因为相等而结束搜索。由此可知不用判断下标的范围了。

```

void insertSort(elementType A[n+1])
{
    for (i=2; i<=n; i++)                //i 表示待插入元素的下标
    {
        A[0]=A[i];                      //设置监视哨保存待插入元素，以腾出 A [i] 空间
        j=i-1;                          //j 指示当前空位置的前一个位置
        while (A[j].key>A[0].key)        //搜索插入位置并腾出空位
        {
            A[j+1]=A[j];
            j=j-1;
        }
        A[j+1]=A[0];                    //插入元素
    }
}

```

【算法分析】

① **稳定性**：由于算法在搜索插入位置的过程中遇到相等的元素时就停止了，所以该算法为稳定的排序算法。

② **空间性能**：该算法仅需要一个记录的辅助存储空间，即监视哨的空间。

③ **时间性能**: 整个算法循环 $n-1$ 次, 每次循环中的基本操作为比较和移动元素, 其总次数取决于数据表的初始特性, 可能有以下几种典型的情况:

(1) 数据表开始时已经有序, 因而每次循环中只需比较一次, 移动两次, 所以整个排序算法的比较和移动元素次数分别为 $(n-1)$ 和 $2(n-1)$, 因而其时间复杂度为 $O(n)$ 。

(2) 当数据表为逆序时, 每次循环中比较和移动元素的次数达到最大值, 分别为 i 和 $i+1$ 次, 因而整个算法的比较和移动元素次数达到最大值, 分别为 $\sum_{i=2}^n i = (n+2)(n-1)/2$ 和

$\sum_{i=2}^n i+1 = (n+4)(n-1)/2$ 。因而算法的时间复杂度为 $O(n^2)$ 。

(3) 一般情况下, 可认为出现各种排列的概率相同, 取上述两者的平均值作为其时间性能, 因而时间复杂度为 $O(n^2)$ 。

8.2.2 希尔排序

如前所述, 直接插入排序算法的时间性能取决于数据的初始特性。一般情况下, 时间复杂度为 $O(n^2)$, 但是当序列为正序或基本有序 (即表中逆序的元素较少, 或者说表中每个元素距离其最终位置的差距不大) 时, 时间复杂度为 $O(n)$ 。因此, 若能在此之前将排序序列调整为基本有序, 则排序的效率会大大提高。另一方面, 如果元素个数较少, 则直接插入排序的效率也较高。正是基于这样的考虑, 出现了希尔排序 (Shell Sort)。

希尔排序的**基本思想**是: 将待排序列划分为若干组, 在每组内进行直接插入排序, 以使整个序列基本有序, 然后再对整个序列进行直接插入排序。

这种排序的关键是如何分组——如果简单地逐段分割, 难以达到基本有序的目的。为此采用间隔方法分组, 分组方法为: 对给定的一个步长 $d(d>0)$, 将下标相差为 d 的倍数的元素分在一组, 这样共得到 d 组。这样一来, 又引出另一问题: d 取什么值? 事实上, d 的取值有多个, 典型的取值依次为 $d_1 = n/2$, $d_2 = d_1/2$, ..., $d_k = 1$, 这样, 随着步长 d_i 的逐渐缩小, 每组规模不断扩大。当步长取值为 1 时, 整个序列为一组执行直接插入排序, 这是希尔排序所必须的。通过前面若干趟的初步排序, 使得此时的序列基本有序, 因此只需较少的比较和移动次数。图 8-2 为希尔排序的示例。

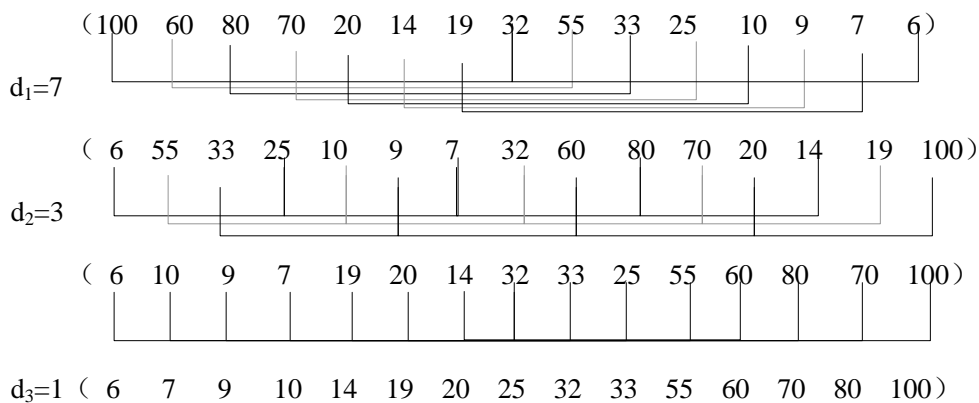


图 8-2 希尔排序过程示例

希尔排序的算法如下：

```
void shellSort(elementType A[n+1], int dh)
    //dh 为起始步长，约定  $1 < dh \leq n/2$  且  $dh$  为偶数
{
    while (dh >= 1)                //通过步长控制排序的执行过程
    {   for (i=dh+1; i<=n; i++)      //依次插入元素到前面的有序表中
        {   插入 A[i] 到 A[1..i-dh] 中的适当位置
            temp=A[i];
            j=i;                    //保存待插入元素，腾出空位，并用 j 指示空位置
            while (j>dh && temp.key<A[j-dh].key)
            {   A[j]=A[j-dh];        //移动元素
                j=j-dh;              //j 前移
            }
            A[j]=temp;              //插入
        }
        dh=dh/2;                    //假设步长每次缩短一半
    }
}
```

关于本算法的补充说明：

① 本算法中约定初始步长 dh 为已知。

② 本算法中采用简单的取步长值的方法：从第二项起的每个步长为其前一步长的一半。然而，在实际应用中，可能有多种取步长的方法，并且不同的取值方法对算法的时间性能有一定的影响。因而一种好的取步长的方法是改进希尔排序算法的时间性能的关键。

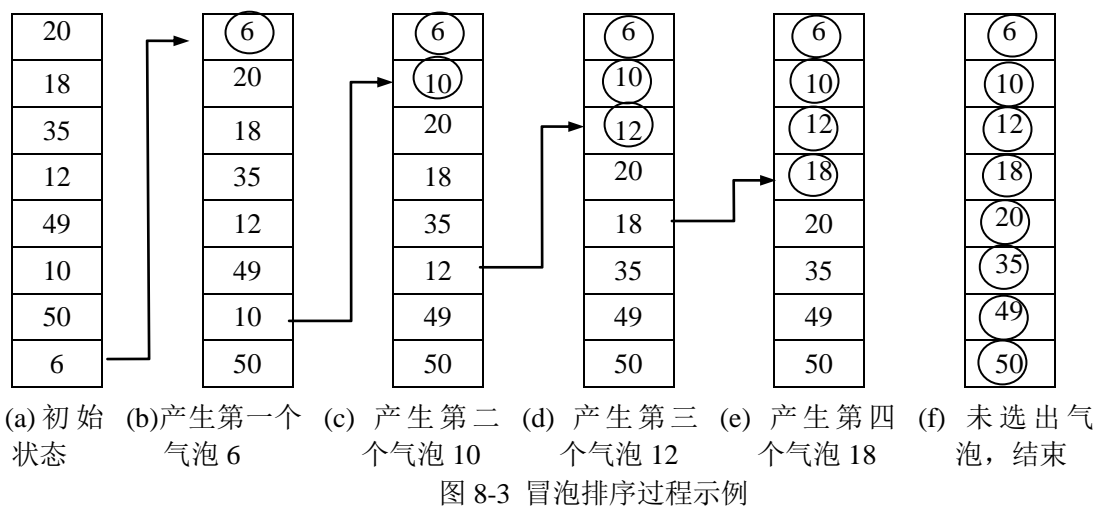
如上所述，希尔排序时间性能的分析是一个复杂问题。考虑到每一趟都是在上一趟的基础上进行的，故可认为是基本有序，因而各趟的时间复杂度为 $O(n)$ 。由于按每次取步长的一半的方式进行，故需要的趟数为 $\log_2 n$ ，由此可知，整个排序算法的时间复杂度为 $O(n \log_2 n)$ 。另外，该算法显然是不稳定的。

8.3 交换排序

交换排序的**基本思想**是：两两比较待排序列的元素，发现倒序即**交换**。下面讨论基于这种思想的两个排序：冒泡排序和快速排序。

8.3.1 冒泡排序

在这类基于交换思想的排序算法中，较为简单的一种是**冒泡排序**(bubble sort)。冒泡排序的基本思想是：从一端开始，**逐个比较相邻的两个元素，发现倒序即交换**。典型的做法是从后往前，或从下往上逐个比较相邻元素，发现倒序即进行交换，本书默认为按这一方向进行。这样一遍下来，一定能将其中最大(或最小)的元素交换到其最终位置上（按此处的约定，为最上面），就像水中的气泡那样冒到水面上，故此得名。然而，一趟只能使一个“气泡”到位，所以必须对余下元素重复上述过程，即要重复 $n-1$ 次冒泡操作。图 8-3 为冒泡排序过程示例。



简单的算法如下：

```
void bubbleSort(elementType A[n+1])
{
    for (i=1; i<n; i++)          //控制选择 n-1 次“气泡”涌
        for (j=n; j>=i+1; j--)  //控制从下往上依次比较相邻的元素
            if (A[j].key<A[j-1].key) //判断是否倒序
                A[j]<=>A[j-1];      //交换
}
```

分析：本算法在各趟的比较次数依次为 $n-1, n-2, \dots, 1$ ，因而时间复杂度为 $O(n^2)$ 。

如果在某一趟排序过程中，没有进行任何交换，说明已经有序，则可以结束排序。同理，在初始序列为正序时，第一趟比较下来，也可结束，因而其时间复杂度可以达到 $O(n)$ 。由此可知，运用这一条件可以提高算法的实践性能。为实现这一要求，可在每趟排序中设置是否进行过交换的标志。这样，在每趟排序结束时，以此作为是否继续的条件。由此得到改进的冒泡排序算法如下：

```
void bubble_sort(elementtype A[n+1])
{
    i=1;
    do
    {
        exchanged=FALSE;          //exchanged 为是否交换的标志涌
        for (j=n; j>=i+1; j--)    //控制一趟中从下往上依次比较相邻的元素
            if (A[j].key<A[j-1].key) //判断是否倒序
                { A[j]<=>A[j-1]; exchanged=TRUE; } //交换，并作标记
        i++;
    } while (i<=n-1 && exchanged==TRUE);
}
```

分析：本算法的时间复杂度依赖于待排序序列的初始特性，典型地有如下几种情况：

- ① 当初始序列为正序时，仅一趟比较下来即可结束，因而所进行的比较元素的次数为 $n-1$ 次，而交换次数为 0，所以时间复杂度为 $O(n)$ 。
- ② 当初始序列为逆序时，每一趟中的比较和交换元素的次数均达最大值，其中第 i 趟中

的比较和交换次数均为 $(n-i)$ ，因而整个算法的比较和交换次数为 $n(n-1)/2$ ，故算法的时间复杂度为 $O(n^2)$ 。

③ 假设一般情况下出现各种排列的概率相同，则将上述两种情况的时间复杂度的平均值作为其时间复杂度，因此为 $O(n^2)$ 。

该算法显然是稳定的排序算法。

在上述冒泡排序中，如果序列中的最大值在第一个位置上，则即使其它元素的排列为正序，也需进行 $n(n-1)/2$ 次比较——即比较次数达到最大。此时，若按反方向进行排序，则只需两趟即可结束。这是否意味着这一方向效果更好呢？显然不能这么说，因为采用这种方向会遇到同样的问题。也有这样改进的：交替地按从上到下和从下到上的方向进行。虽然这种改进可避免上述问题，但改进程度有限。

8.3.2 快速排序

由于冒泡排序算法中是以相邻元素来比较和交换的，因此，若一个元素离其最终位置较远，则需要执行较多次数的比较和移动操纵。是否可以改变一下比较的方式，以使比较和移动操作更少一些？快速排序算法即是对冒泡排序算法的改进。

1. 快速排序的基本思想

快速排序的**基本思想**是：首先，选定一个元素作为**中间元素**，然后将表中所有元素与该中间元素相比较，将表中比中间元素小的元素调到表的前面，将比中间元素大的元素调到后面，再将中间数放在这两部分之间以作为分界点，这样便得到一个划分。然后再对左右两部分分别进行快速排序（即对所得到的两个子表再采用相同的方式来划分和排序，直到每个子表仅有一个元素或为空表为止。此时便得到一个有序表）。

也就是说，快速排序算法通过一趟排序操作将待排序序列划分成左右两部分，使得左边任一元素不大于右边任一元素，然后再分别对左右两部分分别进行(同样的)排序，直至整个序列有序为止。

由此可见，对数据序列进行划分是快速排序算法的关键。下面先讨论划分的实现，在此基础上讨论快速排序算法的实现。

2. 划分方法

为实现划分，首先需要解决“中间数”的选择：作为参考点的中间数的选择没有特别的死规定，可有多种选择方法，如选择第一个元素、中间的某个元素、最后一个元素或其他形式等。较典型的方法是选第一个元素，下面采用的就是这种方法。

对给定的中间数实现划分时，需要解决的问题仍有较大的难度：

① 按什么次序比较各元素？

② 当发现“小(或大)”的元素要往前(后)面放置时，具体放在什么位置？

下面讨论划分的具体实现：

① 由于中间元素所占的空间有可能要被其它元素占用，为此，可先保存该元素的值到其它位置以腾出其空间。为此，可执行语句“ $x=A[low]$ ；”（ low 为该表的第一个元素的下标）。

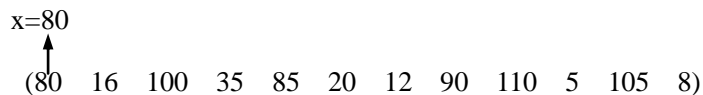
② 这样一来，前面便有一个空位置(用整型变量 low 指示)，此时可以从最后边往前搜索一个比中间数小的元素，并将其放置到前面的这个空位上。

③ 此时，后面便有了一个空位置(用整型变量 $high$ 指示)，可从最前面开始往后搜索一个比中间数大的元素，并将其放置到后面的这个位置上。

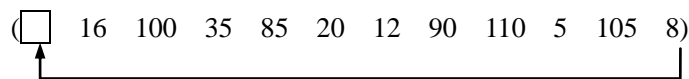
重复①, ②, 直到两边搜索的空位重合(即 $low==high$), (此时说明在该空位的前面没有了大的元素, 后面没有了小的元素) 因而可将中间数放在该空位中。

由此可知, 这种方法是按照由两头向中间交替逼近的次序进行的。图 8-4(a)为一趟划分过程的操纵示例。

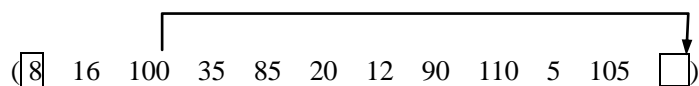
图 8-4(b)为图 8-4(a)操作过程的合并, 将所有单个移动在一个图中标出: 移动元素用箭头表示, 其旁边的数表示其执行序号, 在下一行写出其结果, 其中移走一个元素时, 在下一行用一方框表示。

$x=80$


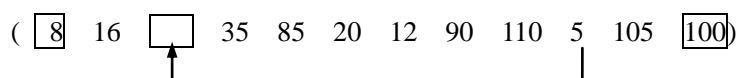
选中间数并腾位后, 从最右边选出比中间数小的数 8 移到前面的空位中:

()

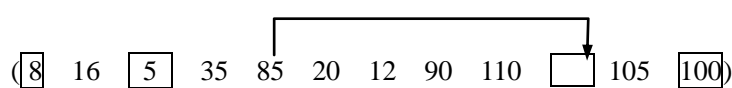
从最左边选出比中间数大的数 100 移到后面的空位中:

()

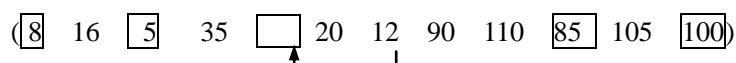
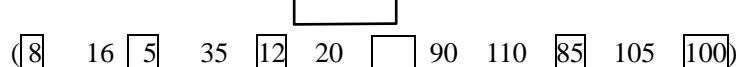
从最右边选出比中间数小的数 5 移到前面的空位中

()

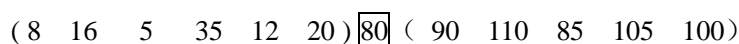
从最左边选出比中间数大的数 85 移到后面的空位中:

()

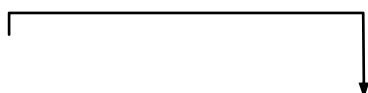
从最右边选出比中间数小的数 12 移到前面的空位中:

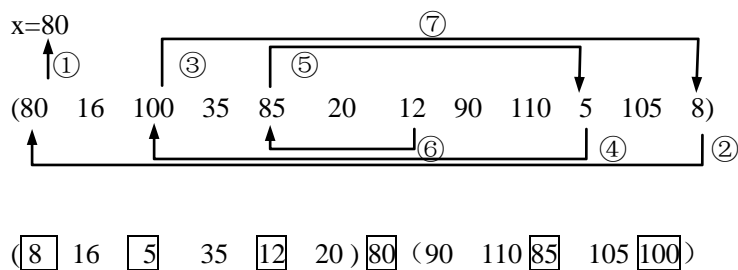
(
)

最后, 从两边搜索到空位重合, 此时将中间元素 80 放在该空位中, 并将两边分别划分为一个子表:

(8 16 5 35 12 20)  (90 110 85 105 100)

(a) 划分过程分步示意图





(b)划分过程分步操作的标注

图 8-4 划分过程示例

3. 划分算法及其应用

下面讨论划分算法的设计。由前面的讨论可知，划分过程中的操作步骤如下：

① 保存中间元素的值到临时变量 x 以腾出其空间，并用变量 i 指示该空位。即执行语句 $x=A[\text{low}]$;

② 从最后边往前搜索比中间数小的元素，并将其放置到前面的这个空位上。从而使后面空出一个位置(用整型变量 high 指示);

③ 从最前面开始往后搜索一个比中间数大的元素，并将其放置到后面的这个空位置 high 上，从而使前面空出一个位置(用整型变量 low 指示);

重复②，③，直到从两边搜索的空位重合（即 $\text{low}==\text{high}$ ），（此时说明在该空位的前面没有了大的元素，后面没有了小的元素）因而可将中间数放在该空位中。

由操作过程的描述可知，整个过程交替地从后往前搜索小的元素和从前往后搜索大的元素放置到另一端的空的位置中，并形成新的空位置，直到两个方向的搜索位置重合为止。因而整个算法中的搜索过程要用循环语句来控制，其能够循环的条件是 low 、 high 不相等。

上述搜索元素需要用一个循环语句来控制，循环的条件之一当然是所指定的大小关系。除此之外是否还需要其他条件来控制？

由于整个划分的过程既可能是以前面搜索到元素为结束，也可能是以后面搜索到元素为结束。因此，在每个搜索元素的操作中都需要判断搜索操作的结束条件，即除了大小关系外，还要加上搜索位置是否重合这一条件。

另外，考虑到快速排序算法中需要对划分之后的子表继续划分，因此其划分区域发生了变化，为此，将区域的两端也作为参数。由此可得算法如下：

```
void partition( elementType A[], int low, int high, int *mid )
{
    x=A[low];           //保存枢轴元素到 x，用最低端元素作为分区枢轴
    while(low<high)
    {
        //从高端寻找一个关键字值比枢轴关键字小的元素，移到 A[low]
        while(low<high && A[high].key>=A[0].key)
            high--;      //A[high].key 不小于分区关键字，high 直接左移
        A[low]=A[high];  //高端找到一个比枢轴关键字小的元素，移到 A[low]
        //从低端寻找一个关键字值比枢轴关键字大的元素，移到 A[high]
        while(low<high && A[low].key<=A[0].key)
            low++;       //A[low].key 不大于枢轴关键字，low 下标右移
        A[high]=A[low];  //低端找到一个比枢轴关键字大的元素，移到 A[high]
    }
}
```

```

    }
    //至此, low==high, 找到枢轴元素 x 的最终存储位置
    A[low]=x;           //或 A[high]=x, x 放置到目标位置
    (*mid)=low;        //或 (*mid)=high, 取得分界点下标, 返回, 以继续对子表进行划分
}

```

时间复杂度分析: 由于该算法从两端交替搜索到重合为止, 因而其时间复杂度是 $O(n)$ 。

4. 快速排序算法

下面来讨论快速排序算法的设计。如前所述, 整个快速排序是在一趟划分之后, 对两部分分别进行快速排序, 因而是一个递归形式的算法。考虑到快速排序要对数组中不同区间的子表进行排序, 因而需要将表的两个端点的下标作为参数。算法如下:

```

void QuickSort( elementType A[], int low, int high )
{
    //对数组 A 中的下标从 low 到 high 的元素组成的子表进行快速排序
    int mid;
    if(low<high)        //子表中至少 2 个元素
    {
        partition( A, low, high, &mid );    //划分
        QuickSort( A, low, mid-1 );        //递归对左子表快速排序
        QuickSort( A, mid+1, high );        //递归对右子表快速排序
    }
}

```

【算法分析】

① 稳定性: 快速排序算法显然是不稳定排序。

② 时间复杂度: 如前所述, 一趟划分算法的时间复杂度为 $O(n)$, 因此, 要分析整个快速排序算法的时间复杂度, 就要分析其划分的趟数。这可能有多种情况:

(a)理想情况下, 每次所选的中间元素正好能将子表几乎等分为两部分, 为便于分析, 认为是等分。这样, 经过 $\log_2 n$ 趟划分便可使所划分的各子表的长度为 1。由于一趟划分所需的时间与元素个数成正比, 因而可认为是 cn , 其中 c 为某个常数。所以整个算法的时间复杂度为 $O(n\log_2^n)$ 。

(b)另一极端情况是: 每次所选的中间元素为其中最大或最小的元素, 这将使每次划分所得的两个子表中的一个变为空表, 另一子表的长度为原表长度-1, 因而需要进行 $n-1$ 趟划分, 而每趟划分中需扫描的元素个数为 $n-i+1$ (i 为趟数), 因而整个算法的时间复杂度为 $O(n^2)$ 。

(c)一般情况下, 从统计意义上说, 所选择的中间元素是最大或最小元素的概率较小, 因而可以认为快速排序算法的平均时间复杂度为 $O(kn\log_2^n)$, 其中 k 为某常数。经验证明, 在所有同量级的此类排序方法中, 快速排序算法的常数因子 k 最小。因此, 从平均时间性能来说, 快速排序是目前被认为是最好的一种内部排序方法。

8.4 选择排序

选择排序的**基本思想**是: 在每一趟排序中, 在待排序子表中**选出**关键字最小或最大的元素放在其最终位置上。基于这一思想的排序有多种, 本书介绍两种排序, 即直接选择排序和

堆排序。

8.4.1 直接选择排序

直接选择排序算法采用的方法较直观：通过在待排序子表中完整地比较一遍以确定最大（小）元素，并将该元素放在子表的最前（后）面。这是选择排序中最简单的一种，算法如下：

```
void selectSort(elementType A[n])
{
    for (i=0; i<n-1; i++)          //控制选择 n-1 次
    {
        min=i;                    //min 指示搜索的最小元素，开始时假定子表的第一个元素最小
        for (j=i+1; j<n; j++)      //用后面的各元素与最小元素来比较
            if (A[j].key<A[min].key)
                min=j;              //找出比当前最小数更小的
        if (min!=i)
            A[min]<=>A[i];          //将最小数放到最终位置上
    }
}
```

对下面数据表的直接选择排序的各趟比较数据和相应的结果如下：

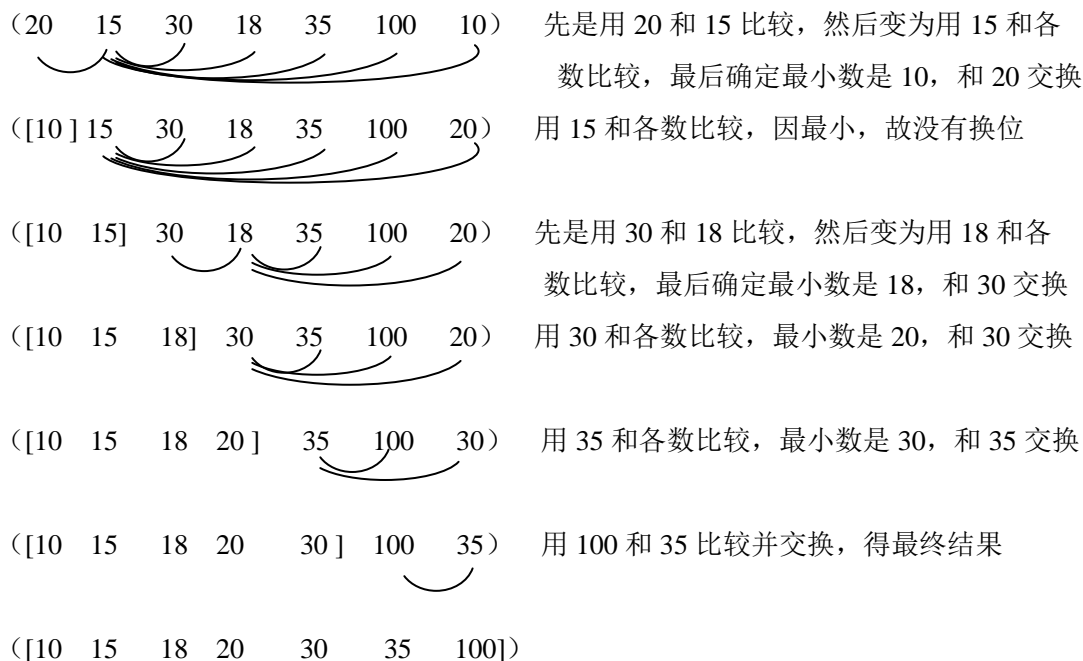


图 8-5 直接选择排序示例及操作说明

【算法分析】

① 稳定性：该算法是不稳定排序，因为关键字相同的元素在排序过程中可能会交换次序。例如对数据表 (3, 3, 2) 排序时，由于要将第一个 3 与 2 交换而导致两个 3 之间的换位。

② 时间复杂度：该算法共进行了 $n(n-1)/2$ 次比较，交换次数最多为 $n-1$ 次，因而时间复杂度为 $O(n^2)$ 。

通过模拟执行可发现：排序过程中可能存在许多次的重复比较，因而造成时间复杂度的增大。如果能减少这些重复比较，便可能会改进算法的时间性能。下面的堆排序是这种改进算法中的一种。

8.4.2 堆排序

所谓**堆排序**就是利用**堆**来进行的一种排序。什么是堆？利用堆又是如何实现排序的？

1. 堆的定义及模型表示

定义：n 个元素的序列 (a_1, a_2, \dots, a_n) 当且仅当满足下面关系时，称之为堆。（其中 k_i 是元素 a_i 的关键字）

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad \begin{matrix} (2i \leq n) \\ (2i+1 \leq n) \end{matrix}$$

初学者可能不易察觉这一定义所揭示的关系。细心的读者对堆的定义中的元素的下标之间的关系可能会觉得似曾相识，事实上，在二叉树的性质中接触过。

若将此序列的各元素按其下标对应到完全二叉树中相同编号的各结点（见图 8-6 所示），则堆的定义可用完全二叉树中的有关术语解释为：每一结点均不大于（对应左边的条件）或不小于（对应右边的条件）其左右孩子结点的值。由此可知，若序列 (a_1, a_2, \dots, a_n) 是堆，则堆顶（完全二叉树中的根）必为序列中的最小或最大值。为便于描述，将根最大的堆称为**大根堆**（见图 8-7（a）），类似地有**小根堆**的概念（见图 8-7（b））。

$(a_1, a_2, a_3, a_4, a_5, a_6, a_7, \dots)$

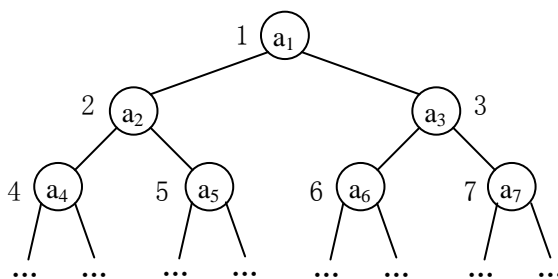


图 8-6 堆的完全二叉树描述示意图

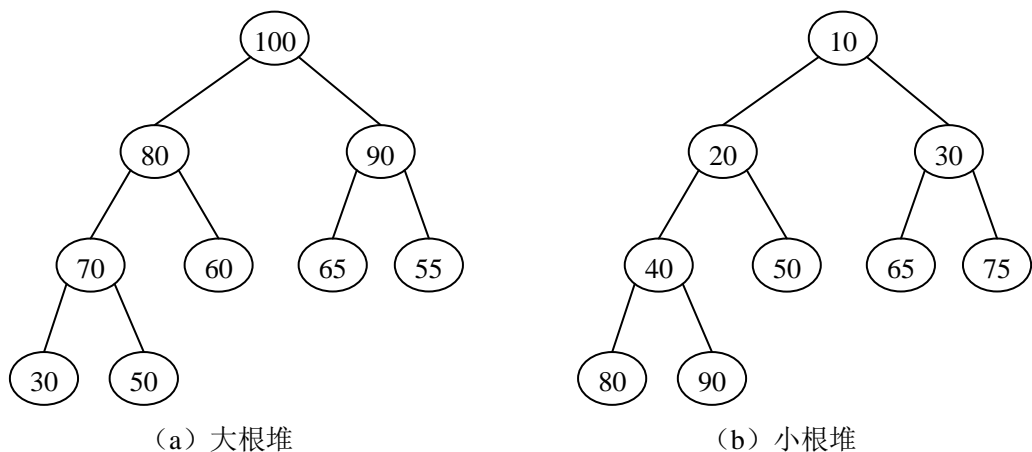


图 8-7 堆的示例

2. 堆的筛选

堆的筛选操作也叫**调整堆**，即将给定序列调整为满足堆的条件的序列。下面以大根堆为例介绍堆的筛选算法。

假定一棵完全二叉树已经是堆，但现在根结点的值被更换为 x 。由于 x 的出现可能破坏了此树的堆属性，如图 8-8 所示，因此需要将其重新调整为堆。

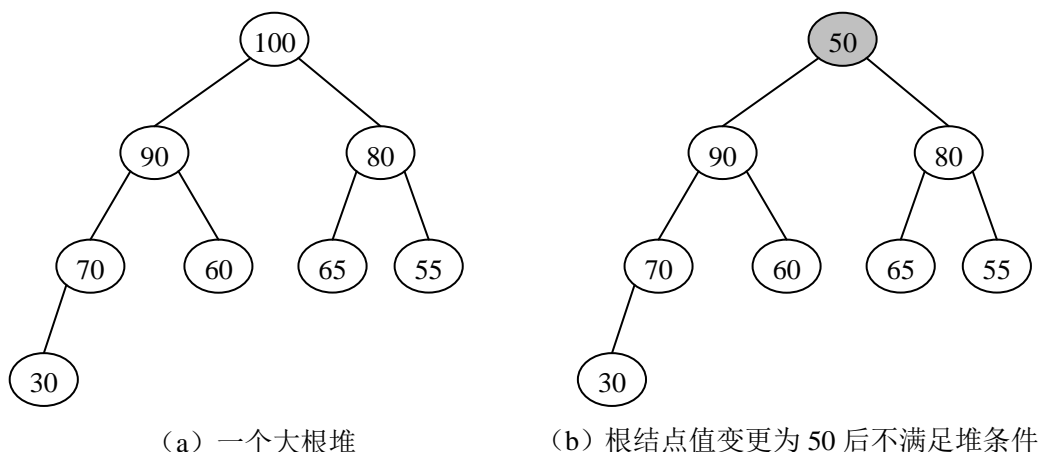


图 8-8 根结点值变更导致堆变化示意图

如何将变化后的序列重新调整为堆呢？

虽然此时的完全二叉树可能变为不是堆，但只有根结点的值发生了变化，其余结点之间仍然满足堆的条件，即根结点的左右子树仍然是堆。如图 8-8 所示。理解这一点有助于对调整过程的认识。分析如下：

① 由于左右子树是堆，故此时的左右孩子结点的值（如图 8-8 中的 90 和 80）分别是两棵子树中的最大数，因此整棵树的结点最大值只可能从根（值为 x ）及其左右孩子结点中产生，故可通过比较这三者，取最大者作为调整后的根即可。

② 如果当前的根（值为 x ）是最大的值，则整棵树即是堆，无需调整；否则将左右孩子结点中的最大值与 x 互换，最大值调到根的位置上来。这样又出现了一个新的问题：即 x 换到新的位置后，以 x 为根的子树又可能不是堆。对此可采用相同的方法进行筛选（调整），直

到为 x 找到合适的位置使整棵树重新成为堆。

例如，图 8-9 (a) 为二叉树调整为堆的示意图，图 8-9 (b) 是调整后的堆。调整过程如下：

① 90、80、 $x=50$ 进行比较（箭头所示），确定 90 最大。

② 90 上移到 50 的位置（50 在此之前应临时保留起来）。

③ 对 90 上移所留下的空位，确定填充者：比较其左右孩子 70、60 和 $x=50$ ，选择最大者 70。（本来应把 $x=50$ 先换到原来 90 的位置，再比较取最大值互换。但因为 50 保存在一个变量中，可以省略写入操作，直接进行比较，取最大值写入此位置，这样可提高效率）。

④ 70 上移到父结点。

⑤ 对 70 上移所留下的空位，确定填充者：比较其左孩子 30（无右孩子）和 50 中的最大者（为 50），故 50 最终放到此处。调整过程到此结束。

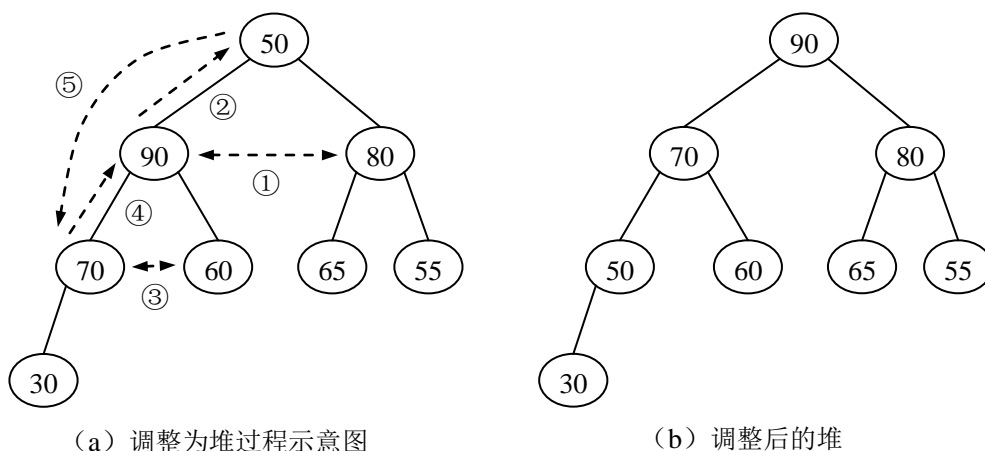


图 8-9 调整堆（筛选）过程示意图

下面讨论筛选（调整）算法的设计。首先确定所需要的参数：从整个排序算法的要求出发，调整的可能是整棵树，也可能是其中的一棵子树，这样就需要知道子树根结点（当前堆顶）的编号，因此需要将堆顶的下标（编号）作为参数。另外，在堆排序中要不断输出根，使得树中的结点不断减少，因此，也需要将当前树中的结点数（也是当前完全二叉树中最大的结点编号，或剩下待排序列的元素个数）作为参数。加上存放全部待排序列元素的数组参数，总共 3 个参数。

由前面所讨论的筛选方法可知，筛选操作的粗略程序描述如下（设当前根的下标为 k ，当前树的结点总数是 m ）：

① 保存当前根的值到一个变量（不妨设为 x ），假定此结点下标为 i 。

② 比较 i 结点的左右孩子（下标分别是多少？）和 x 值，取最大值。可能有几种情况：

(1) i 结点无左右孩子：说明已经到了叶子结点，故 x 最大，填到 i 结点中；

(2) i 结点左右孩子的值小于 x ：说明 i 即为 x 的最终位置，故 x 填到 i 结点中；

(3) 否则，将左右孩子中的最大者填充到 i 结点中，从而出现新的空位，更新 i 指向此新的空位（如何赋值？），转到 ② 继续执行。

由此得筛选（调整）算法如下：

【筛选算法描述】

//调整以 k 为根的子树序列使成为堆。

//其中 k 为子树根。 m 为子树的结点数，也是最大结点（元素）编号（下标）
//假设 k 的左右子树均为堆，即以 $2k$ 和 $2k+1$ 为根的左右子树均是堆

```
void Sift( elementType A[], int k, int m )
{
    elementType x;
    int i,j;
    bool finished=false; //设置筛选未结束标志
    i=k;                  //i 为当前子树根结点编号
    j=2*i;                 //j 为当前根结点 i 左孩子结点的编号
    x=A[i];                //x 保存当前根结点 i 的元素值，空出位置存储调整过来的根
    while(j<=m && !finished)
    {
        if(j<m && A[j]<A[j+1]) //j 指向当前根结点 i 的左右孩子中值较大者
            j++;
        //x 值（i 结点的值）大于左右孩子的值，
        //已经为大根堆，无需调整，置 finished=true
        if(x>=A[j])
            finished=true;
        else //否则， $x<A[j]$ ，需将  $A[j]$  调整为当前子树的根(堆顶)
        {
            A[i]=A[j];
            i=j; //x 调到 j 后，可能破坏 j 为根结点的子树的堆，
                //需要继续调整（往下筛），更新子树根结点 i 为 j。
                //本来此处应是  $A[j]=x$ ； $i=j$ ； $x=A[i]$ ；但执行交换后 x 值并未变化，
                //所以，省略 x 值的交换操作，以提高效率，
                //只要为 x 寻找合适的位置即可。
            j=2*j; //i 仍为当前子树根结点，j 指向 i 的左孩子。
                //回去循环继续把 i 为根的子树调整为堆。
        }
    }
    A[i]=x; //循环结束，i 即为 x 的最终位置，至此以 k 为根的二叉树已经调整为堆。
}
```

上述过程有点类似我们用筛子筛东西，大的物体留下来，小的物体被筛下去，因此被称为筛选算法。

3. 堆排序

下面讨论堆排序算法。若要求按从小到大次序进行排序（即增排序），则需要借助于大根堆。

我们首先来讨论**输出根**操作。假设待排序列放在数组 $A[1...n]$ 中，按完全二叉树顺序存储方式存放， $A[1]$ 为树根，结点 i 的左孩子是 $2i$ ，右孩子是 $2i+1$ 。并假设当前序列已经是大

根堆，即根结点值（或者说是第一个元素） $A[1]$ 最大。进行递增排序时，此最大值 $A[1]$ 必须要放置到 $A[n]$ 位置，那么原来的 $A[n]$ 就要放到其它位置，方便起见，就将 $A[1]$ 和 $A[n]$ 位置互换(即执行 $A[n] \leftrightarrow A[1]$)，称这一操作为**输出根**。

在输出根之后，最后一个元素已经放置到排序后的最终位置，此结点（元素）不必再考虑，我们感兴趣的的就是余下的 $n-1$ 个元素所构成的子表了，即输出根后树上的结点数减 1。由于原表中的最后一个元素（即 $A[n]$ ）调整到了树根（ $A[1]$ ）的位置，可能使得剩下的子树（序列）不满足堆的条件，因而需要将其重新调整为堆。剩下的子树调整为堆后，又可得到下一个最大值，再输出根。反复进行输出根、调整堆的操作，每次输出根后树上结点数减 1，直到树上全部结点都被输出，排序即完成。

下面讨论利用堆进行排序的具体方法。可以分两种情况分别讨论：

① 如果初始序列是堆，则可通过反复执行如下操作而最终得到一个有序序列：

输出根：即将根（第一个元素）与当前子序列中的最后一个元素交换。

调整堆：将输出根之后的子序列调整为堆（元素个数比输出前少 1 个）。

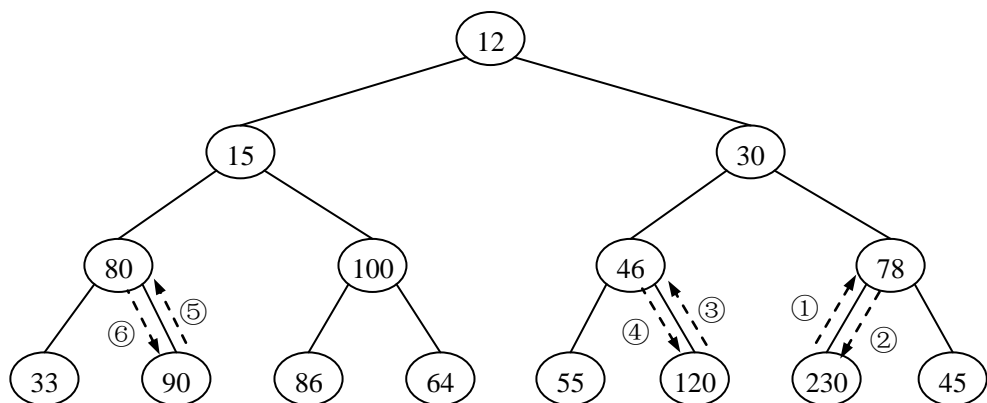
② 如果初始序列不是堆，则首先要将其先建成堆，然后再按 ① 的方式来实现。

现在的问题是：如何由一个无序序列建成一个堆？

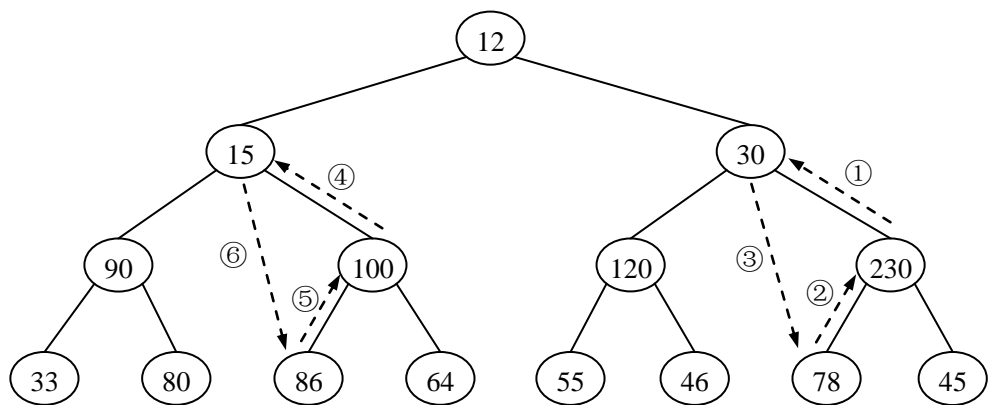
事实上，由无序序列建堆可通过反复调用筛选操作来实现。为此，需满足筛选的条件，即**左右子树必须为堆**。因此，建堆过程要从下往上逐棵子树地进行筛选（调整）。我们知道只有一个结点（根）的子树无需调整，所以从最小的高度为 2 的子树开始调整即可，然后调整高度为 3 的子树，直到调整整棵树。从易于编程的角度出发，从完全二叉树最后一棵高度为 2 的子树开始调整，其根结点编号为 $n/2$ （最后一个结点 n 的父结点），然后按自右往左、自下而上次序依次进行调整，直到 1 号结点（根）。即调整子树根的下标从 $n/2$ 开始，逐次减 1，直到下标为 1 的结点。

例：由初始序列（12,15,30,80,100,46,78,33,90,86,64,55,120,230,45）建堆的过程如图 8-9 所示。其中各操作的标注已经较直接，故不再用文字叙述。

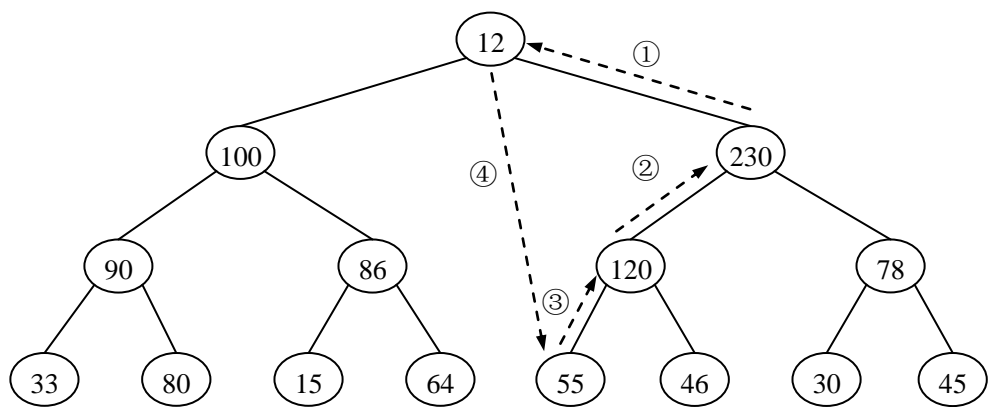
对应的二叉树形式及下标从 7（ $15/2$ ）到 4 的调整过程（用序号表示调整序号）：



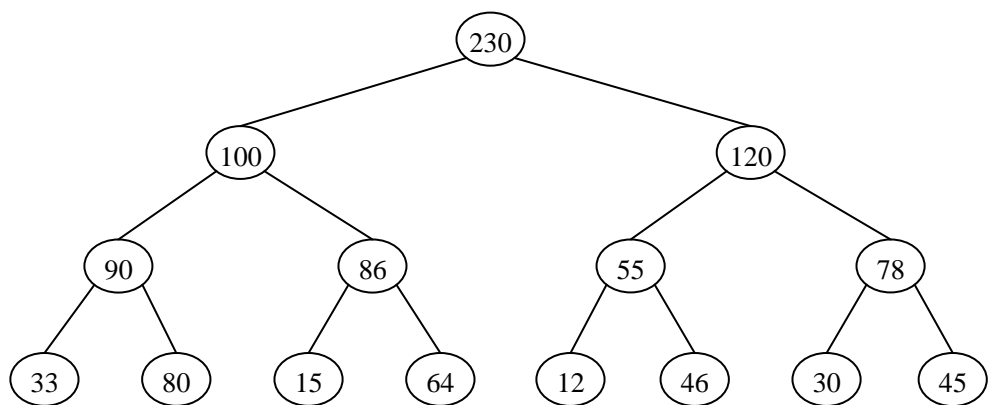
(a) 根下标从 7 到 4 的子树调整过程示意图



(b) 根下标从 3 到 2 的子树调整过程示意图



(c) 根下标为 1 的树调整过程示意图



(d) 最终结果

图 8-10 初始建堆操作过程示例图

最终所得到的堆: (230,100,120,90,86,55,78,33,80,15,64,12,46,30,45)

由此可得建初堆算法描述如下：

【建初堆算法描述】

//从原始待排序序列创建初堆

```
void CreateHeap(elementType A[], int n)
{
    //n 为结点个数（元素个数）
    int i;
    //初建堆--由初始序列产生堆（此处为大根堆）
    //从第 n/2 结点开始往上筛，直到 1 号结点（根、堆顶）
    for(i=n/2; i>=1; i--)
    {
        Sift(A, i, n); //每次调用此函数，都将 i 为根结点的子树调整为堆。
    }
}
```

最后我们讨论堆排序，堆排序就是对已经建成的堆反复执行输出根、调整堆的操作，直到树上全部结点都输出，排序即完成。仍用上面的例子来演示堆排序的过程。经过建初堆后，待排序列 $A[] = \{(230, 100, 120, 90, 86, 55, 78, 33, 80, 15, 64, 12, 46, 30, 45)\}$ ，对应的完全二叉树如图 8-10(d)。

第一步：输出根 230，即使 $A[15]=230$ ， $A[1]=45$ ，如图 8-11(a)。230 进入最终位置，不需再参与排序，此后待排序列元素减 1 为 14，树的结点数亦为 14（230 对应结点可从树上删去）。此后 $A[] = \{(45, 100, 120, 90, 86, 55, 78, 33, 80, 15, 64, 12, 46, 30)\}$ ，230}。

调整堆：由于 45 调到 $A[1]$ ，需要调整堆，如图 8-11(b)。 $A[] = \{(120, 100, 78, 90, 86, 55, 45, 33, 80, 15, 64, 12, 46, 30)\}$ ，230}。

第二步：输出根 120，即 $A[14]=120$ ， $A[1]=30$ ，如图 8-11(c)。 $A[] = \{(30, 100, 78, 90, 86, 55, 45, 33, 80, 15, 64, 12, 46)\}$ ，120, 230}。

调整堆如图 8-11(d)。 $A[] = \{(100, 90, 78, 80, 86, 55, 45, 33, 30, 15, 64, 12, 46)\}$ ，120, 230}。

第三步：输出根 100，即 $A[13]=100$ ， $A[1]=46$ ，如图 8-11(e)。 $A[] = \{(46, 90, 78, 80, 86, 55, 45, 33, 30, 15, 64, 12)\}$ ，100, 120, 230}。

调整堆如图 8-11(f)。 $A[] = \{(90, 86, 78, 80, 64, 55, 45, 33, 30, 15, 46, 12)\}$ ，100, 120, 230}。

第四步：输出根 90，即 $A[12]=90$ ， $A[1]=12$ ，如图 8-11(g)。 $A[] = \{(12, 86, 78, 80, 64, 55, 45, 33, 30, 15, 46)\}$ ，90, 100, 120, 230}。

调整堆如图 8-11(h)。 $A[] = \{(86, 80, 78, 33, 64, 55, 45, 12, 30, 15, 46)\}$ ，90, 100, 120, 230}。

这个过程持续下去，最终即可完成堆排序。

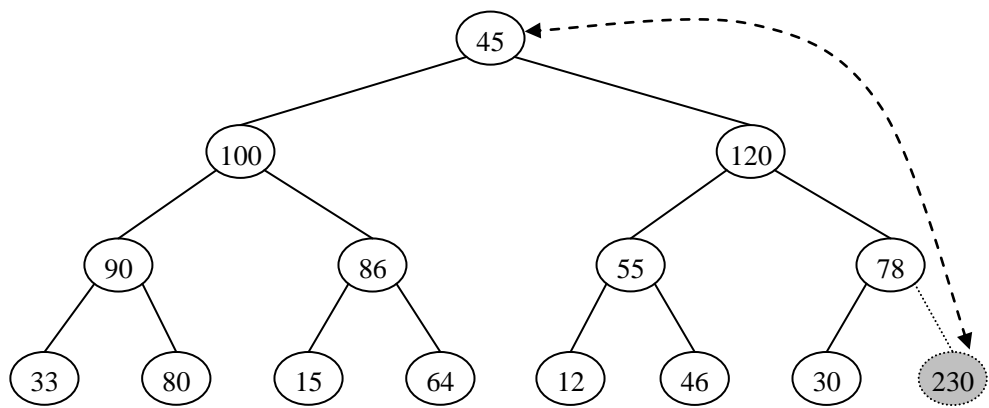


图 8-11(a) 输出根 230 示例图

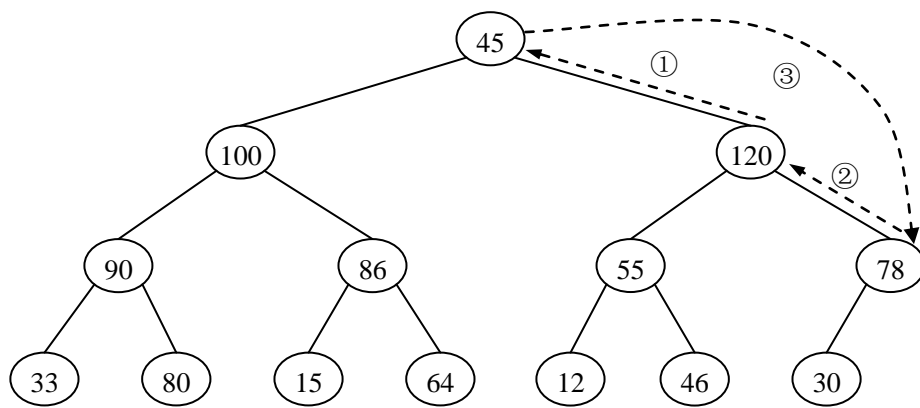


图 8-11(b) 输出根 230 后调整堆

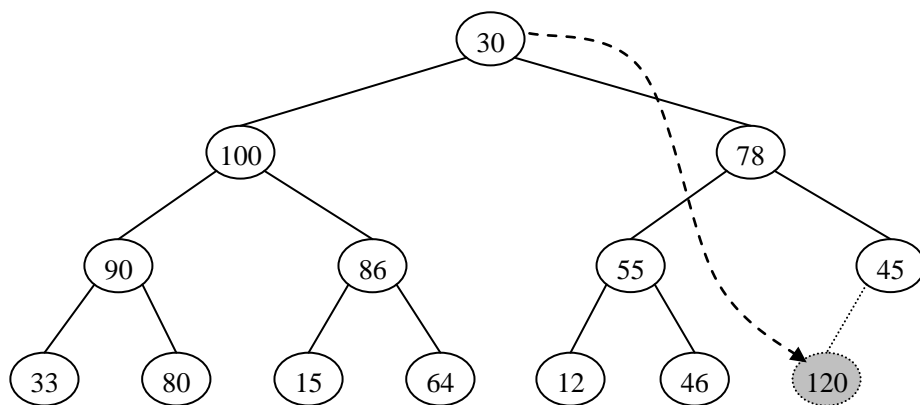


图 8-11(c) 输出根 120

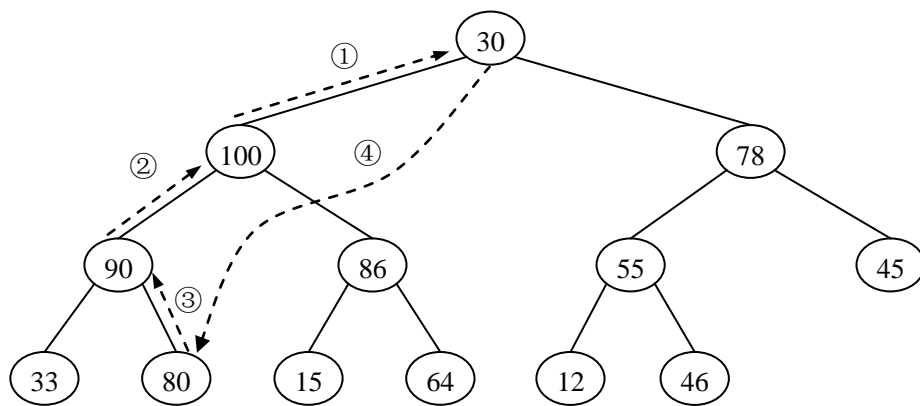


图 8-11(d) 输出根 120 后调整堆

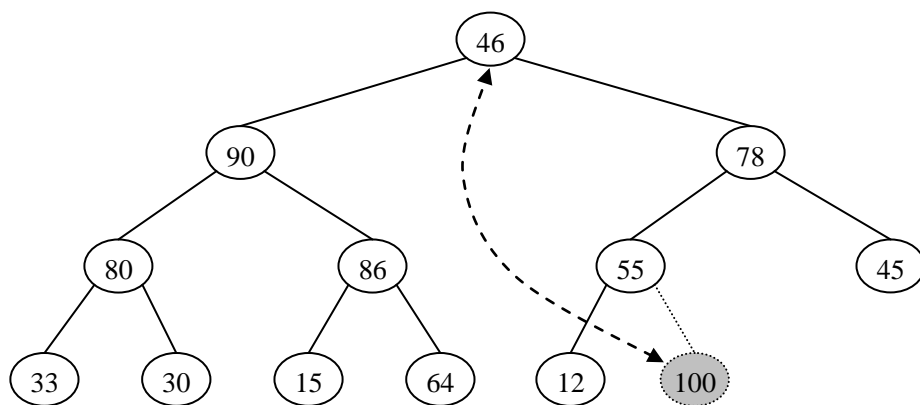


图 8-11(e) 输出根 100

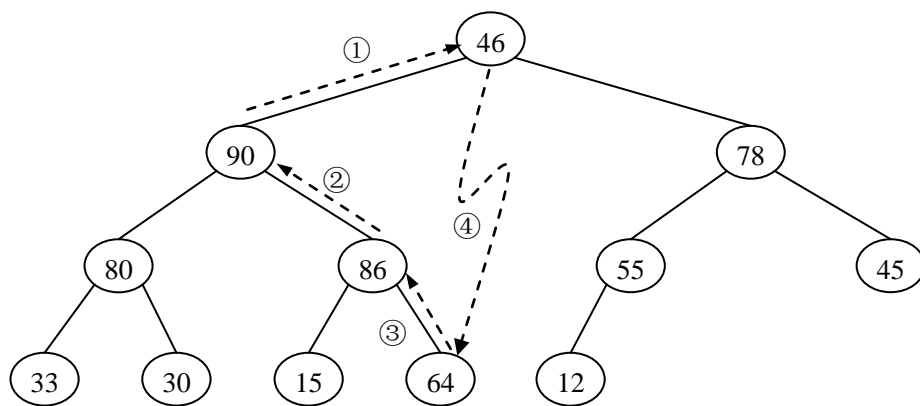


图 8-11(f) 输出根 100 后调整堆

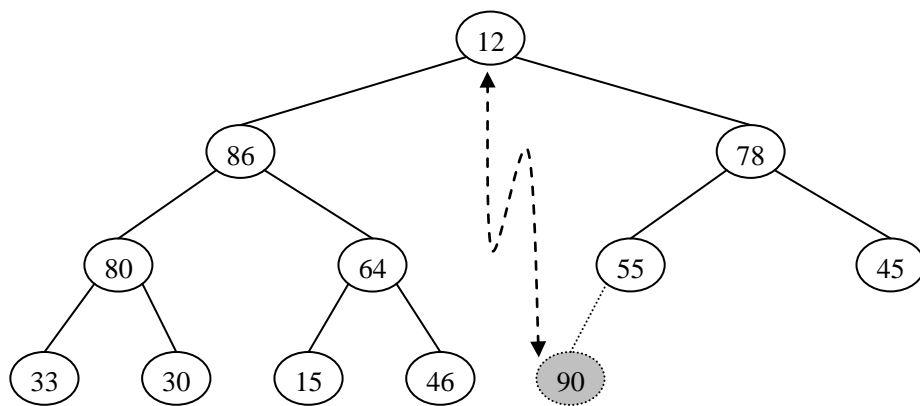


图 8-11(g) 输出根 90

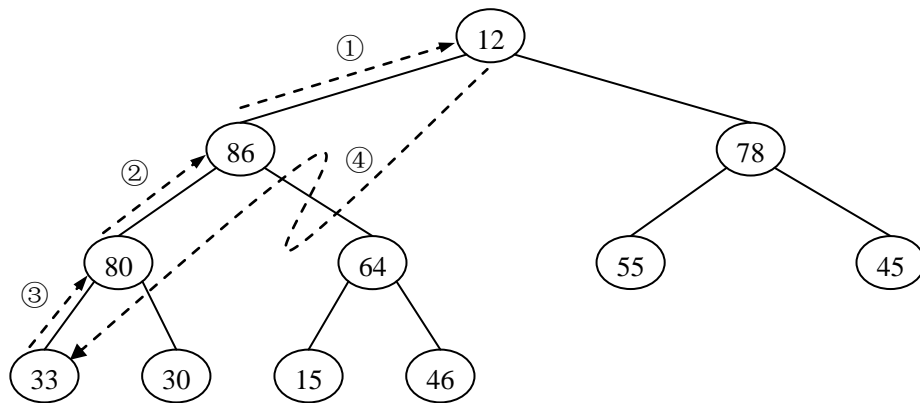


图 8-11(h) 输出根 90 后调整堆

由讨论得堆排序算法如下：

【堆排序算法描述】

```
void HeapSort(elementType A[], int n)
```

```
{    //排序结果仍放在数组 A[] 中。A[0] 单元作为临时存储变量。
```

```
    int i;
```

```
    CreateHeap(A,n);    //由无序序列建初堆
```

```
    //由堆序列产生排序序列，此时整棵树（完全二叉树）为堆（此处为大根堆）
```

```
    for( i=n;i>=2;i-- )    //循环输出根、调整堆操作。
```

```
    {
```

```
        A[0]=A[i];    //完全二叉树最后一个结点 A[i] 保存到 A[0]，
```

```
        //空出位置输出根 A[1]，即当前子树的根（堆顶）
```

```

        A[i]=A[1]; //输出根，即 A[1]保存到排序后的最终位置 i。
        A[1]=A[0]; //原第 i 元素暂作为“根”。
                //又 A[1]=A[0]后可能破坏了当前树的堆属性，
                //需从根结点 1 开始重新调整为堆
                //因为输出根，此时树的结点数为 i-1。
        Sift(A,1,i-1);
    }
}

```

【算法分析】堆排序算法花费时间最多的是建初始堆和调整堆时所进行的筛选。对深度为 k 的堆，筛选算法中所进行的关键字的比较次数至多为 $2(k-1)$ 次，而 n 个结点的完全二叉树的高度约为 $\log_2 n + 1$ ，因此调整堆（共 $n-1$ 次）总共进行的关键字的比较次数不超过 $\log_2(n-1) + \log_2(n-2) + \dots + (\log_2 2)$ ，而建初始堆所进行的比较次数不超过 $4n$ ，因此算法的时间复杂度为 $O(n \log_2 n)$ 。

8.5 归并排序

归并排序（Merge Sort）是一种基于归并方法完成的排序。所谓归并是指将两个或两个以上的有序表合并成一个新的有序表。本节介绍的算法是通过反复将 2 个有序子表归并完成排序，故称为**二路归并排序**。归并排序是分治法（Divide and Conquer）求解问题的经典案例之一。是一种高效的排序算法，最坏的时间复杂度为 $O(n \log_2 n)$ 。归并排序可用于大数据量的外排序，还可以设计成并行算法进行并行计算。

二路归并排序通过将两个较短的有序表归并为一个更长的有序表，逐步完成整表的排序。那么，初始的有序表怎么得到呢？我们可以通过不同的方式将原始的表划分为只有 1 个元素的子表（长度为 1），每个子表只有 1 个元素是有序的；然后将两两长度为 1 的有序子表归并为长度为 2 的有序子表；再归并为长度为 4 的有序子表；依次类推，直至归并出长度为 n 的有序表，整表排序完成。根据有序子表划分和归并执行过程的不同，归并排序有自底向上和自顶向下两种具体的排序算法。

下面首先讨论归并的实现，在此基础上分别讨论自底向上和自顶向下两种归并排序算法的实现。

8.5.1 归并

我们在第二章顺序表的内容中讨论过将两个有序表 $A=(a_1, a_2, \dots, a_m)$ 和 $B=(b_1, b_2, \dots, b_n)$ 高效归并为一个有序表 $C=(c_1, c_2, \dots, c_{m+n})$ 的例子，时间复杂度为 $O(m+n)$ 。但在归并算法中两个有序子表是在同一个待排序表 $A[n]$ 中通过下标划分出来的。我们可以通过 low、mid、high 三个下标变量，划分出 $A[\text{low}..\text{mid}]$ 和 $A[\text{mid}+1..\text{high}]$ 两个子表。假定这两个子表是有序的，我们将其归并为一个更长的 $A[\text{low}..\text{high}]$ 有序子表。每次归并的结果需要先缓存在另一个表上，归并结束后再复制回 $A[n]$ 中，因此，我们需要一个与 $A[n]$ 相同大小的缓存表 $T[n]$ 。我们改造第二章的归并算法描述如下：

```

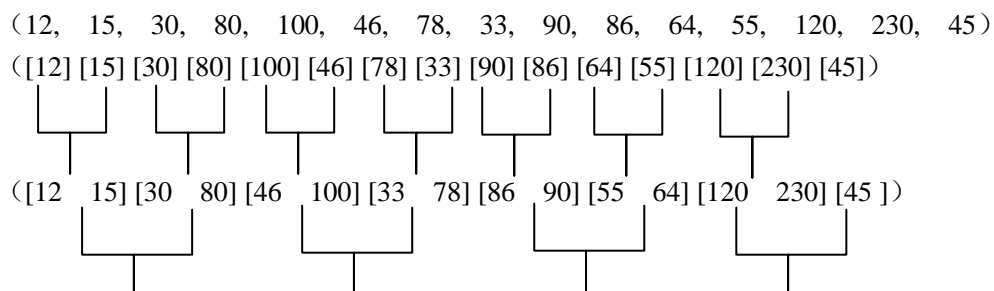
void Merge( elementType A[], int low, int mid, int high )
{
    elementType T[n];    //缓存表，用于存放归并后的元素
    int i=low, j=mid+1, k=low;
        //i 为 A[low...mid]的下标; j 为 A[mid+1...high]的下标; k 为 T[]的下标
    while(i<=mid && j<=high)    //A[]中两个子表都有元素
    {
        if(A[i].key<=A[j].key)    //A[i]较小，放入临时表
            T[k++]=A[i++];
        else    //A[j]较小，放入临时表
            T[k++]=A[j++];
    }
        //下面处理一个子表结束，另一个子表未结束情况。
    while( i<=mid )    //A[low..mid]未结束
        T[k++]=A[i++];
    while( j<=high )    //A[mid+1...high]子表未结束
        T[k++]=A[j++];
        //临时表 T[]中已归并元素，拷回原表 A[]
    for( i=low; i<=high; i++ )
        A[i]=T[i];
}

```

A[n]中一趟全部子表的两两归并，要扫描比较 A[n]中的所有 n 个元素，因此一趟归并，不管子表怎样划分，总时间复杂度都是 O(n)。

8.5.2 自底向上归并排序

自底向上的归并排序，首先将整个表看成是 n 个有序子表，每个子表 1 个元素，即子表长度 len=1。然后选择相邻子表两两归并，得到 n/2 个长度 len=2 的有序子表；再两两归并，得到 n/4 个长度 len=4 的有序子表；反复这个过程，直至得到一个长度为 n 的有序表为止。以上过程通过循环控制完成，循环条件为 len<n。每趟循环有序子表的长度 len 增长一倍，即 len=len*2。当 len>=n 时归并完成。子表长度按 1、2、4、...、n 规律变化。图 8-10 为这种排序的示例。



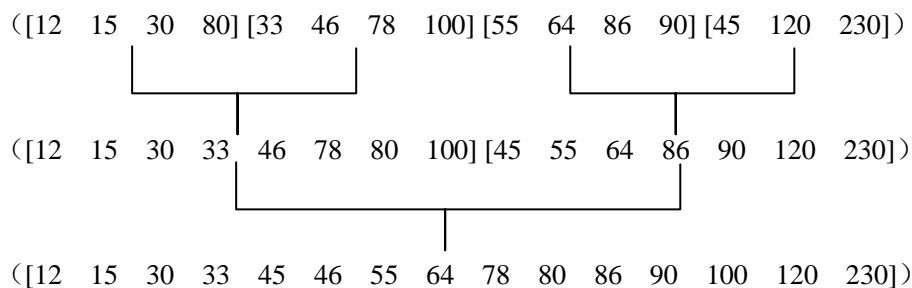


图 8-10 自底向上归并排序示例

上述一趟归并时，前面部分由循环控制进行等长子表（假定长度为 len ）的两两归并，最后可能出现两种情况：①最后剩下一个子表，子表长度 $\leq len$ ，这个子表本轮不做处理，直接进入下一轮归并；②最后剩下两个子表，可能等长，也可能最后一个子表的长度小于 len ，需要单独进行归并。由此得到自底向上归并排序算法框架如下：

```
MergeSortUp( A[ ] )
{
    while( len < n )    //有序子表长度小于原序列长度
    {
        ① 循环调用 Merge( A, low, mid, high ) 函数，
            对长度为 len 的相邻 2 个等长有序子序列进行归并；
        ② 如果最后剩下 2 个不等长子表，单独调用 Merge( ) 函数进行归并；
        ③ len=2*len;    //一趟归并结束，有序子表长度倍增；
    }
}
```

假定每趟归并用整型变量 i 控制，每趟都从 1 开始（ $A[0]$ 单元未用），有序子表长度为 len 。

两两等长 len 的有序子表归并时，假设第一个子表第一个元素的下标为 i ，则有 $low=i$ ， $mid=i+len-1$ ， $high=i+2len-1$ ，即当前归并的两个子表的范围为 $A[i \dots i+len-1]$ 和 $A[i+len \dots i+2len-1]$ 。两个子表归并完成后， i 应该指向后面两个相邻子表的第一个元素，即 $i=i+2*len$ ，也就是 i 的循环步长为 $2*len$ 。那么等长子表归并的结束条件是什么呢？考虑到最后最多剩下两个等长的子表，所以有 $i \leq n-2*len$ ，此即为相邻等长子表归并结束的条件。

什么情况下最后会剩下 2 个不等长子表呢（也可能是等长的）？前面相邻子表归并结束后， i 指向剩下部分第一个子表的第一个元素，范围为 $A[i \dots i+len-1]$ ，如果第二个子表至少有 1 个元素，则必有 $i+len \leq n$ ，所以最后存在两个不等长子表（也可能等长）的条件为 $i+len \leq n$ 。

【自底向上归并排序算法描述】

```
void MergeSortUp( elementType A[ ] )
{
    int len=1;        //子表长度
    while( len < n )   //控制归并趟数
    {
        //控制相邻等长子表归并
        for(i=1; i <= listLen-2*len; i=i+2*len)
        {
            //调用二路归并算法
            Merge(A, i, i+len-1, i+2*len-1); //low=i, mid=i+len-1, high=i+2*len-1
        }
    }
}
```

```

        if( i+len<=n )    //如果最后剩下 2 个有序子表，单独归并。
            Merge( A, i, i+len-1, n );
        len=2*len;        //有序表长度倍增
    }
}

```

此算法每趟都要扫描比较 $A[n]$ 中的每个元素，故每趟的比较次数为 n 。又有序表长度倍增，最多 $\log_2 n$ 趟即完成排序。故算法的时间复杂度为 $O(n\log_2 n)$ 。算法需要 $T[n]$ 缓存归并结果，故空间复杂度为 $O(n)$ 。是稳定排序。

8.5.3 自顶向下归并排序

自顶向下归并排序的子表划分和归并过程与上述自底向上归并排序不同。它首先对原始待排表 $A[n]$ 进行等长二分划分；然后递归地对左右子表进行划分；每当得到 2 个有序子表，立即调用 $\text{Merge}(A[], \text{low}, \text{mid}, \text{high})$ 函数进行归并；如此划分、归并交替进行，直到归并出整个有序表。

假定子表第一个元素下标 low ，最后一个元素下标 high 。等分划分的分界点下标 $\text{mid}=(\text{low}+\text{high})/2$ ，左子表范围 $A[\text{low}...\text{mid}]$ ，右子表范围 $A[\text{mid}+1...\text{high}]$ 。左右子表都有序后，调用 $\text{Merge}(A[], \text{low}, \text{mid}, \text{high})$ 函数归并出更长有序子表。算法递归执行，划分类似二叉树的先序遍历，归并类似二叉树的后序遍历。算法描述如下。

【自顶向下归并排序算法描述】

```

void MergeSortDown( elementType A[], int low, int high )
{
    int mid;    //划分点下标
    if( low<high )
    {
        mid=(low+high)/2;           //等长二分划分
        MergeSortDown(A, low, mid); //递归对 A[low...mid]进行归并排序
        MergeSortDown(A, mid+1, high); //递归对 A[mid+1...high] 进行归并排序
        Merge(A, low, mid, high);    // 2 个有序子表归并为更长有序序列
    }
}

```

此算法因为等分划分，最多经 $\log_2 n$ 次划分即得到长度为 1 的有序子表。不管归并过程如何，所有相同长度等长有序子表的一次归并需要比较 n 次。估算法时间复杂度仍为 $O(n\log_2 n)$ 。

【算法分析】

① 空间性能： $O(n)$ ，由于在顺序表中不能进行就地归并，因而需另外开辟存储区 $T[n]$ 以存放归并结果，所以归并排序需要与原表等量的辅助存储空间。

② 时间性能：由于将两个有序表合并为一个有序表的时间复杂度为两表长之和的数量级，因此，一趟排序的时间复杂度为 $O(n)$ 。又因总共需 $\log_2 n$ 趟归并，故时间复杂度为 $O(n\log_2 n)$ 。

③ 稳定性：归并排序是稳定的。

归并排序是一种高效的排序算法，算法的时间性能与元素初始状态无关，适用记录较多情况。归并排序可用于并行运算。常用于外部排序。

本章小结

排序是软件设计中最常用的运算之一，有多种排序的算法，衡量排序算法的时间性能主要是以算法中用得最多的基本操作的数量为基本单位的，这些基本操作包括比较元素、移动元素和交换元素。

依据排序所用的基本思想，可将排序算法划分为插入排序、交换排序、选择排序和归并排序。

插入排序算法的基本思想是：将待排序表看作是左右两部分，其中左边为有序区，右边为无序区，整个排序过程就是将右边无序区中的元素逐个插入到左边的有序区中，以构成新的有序区。直接插入排序是这类排序算法中最基本的一种，然而，其时间性能取决于数据表的初始特性，在数据表基本有序的情况下，时间复杂度为 $O(n)$ ，最坏情况下的时间复杂度为 $O(n^2)$ ，平均时间复杂度也为 $O(n^2)$ 。希尔排序算法是一种改进的插入排序，其基本思想是：将待排序列划分为若干组，在每组内进行直接插入排序，以使整个序列基本有序，然后再对整个序列进行直接插入排序。其时间性能不取决于数据表的初始特性，为 $O(n\log_2^n)$ 。

交换排序的基本思想是：两两比较待排序列的元素，发现倒序即交换。基于这种思想的排序有冒泡排序和快速排序两种。冒泡排序的基本思想是：从一端开始，逐个比较相邻的两个元素，发现倒序即交换。然而，其时间性能取决于数据表的初始特性，最好情况下，时间复杂度为 $O(n)$ ，最坏情况下的时间复杂度为 $O(n^2)$ ，平均时间复杂度也为 $O(n^2)$ 。快速排序是一种改进的交换排序，其基本思想是：以选定的元素为中间元素，将数据表划分为左右两部分，其中左边所有元素不大于右边所有元素，然后再对左右两部分分别进行快速排序。在理想情况下，快速排序算法的时间复杂度为 $O(n\log_2 n)$ 。然而，如果数据表已经有序，则算法的时间复杂度最差，达到 $O(n^2)$ 。

选择排序的基本思想是：在每一趟排序中，在待排序子表中选出关键字最小或最大的元素放在其最终位置上。直接选择排序和堆排序是基于这一思想的两个排序算法。直接选择排序算法采用的方法较直观：通过在待排序子表中完整地比较一遍以确定最大（小）元素，并将该元素放在子表的最前（后）面。堆排序就是利用堆来进行的一种排序，其中堆是一个满足特定条件的序列，该条件用完全二叉树模型表示为每个结点不大于（小于）其左右孩子的值。利用堆排序可使选择下一个最大（小）数的时间加快，因而提高算法的时间复杂度，达到 $O(n\log_2^n)$ 。

归并排序是一种基于归并的排序，其基本操作是指将两个或两个以上的有序表合并成一个新的有序表。

每种算法都基于一定的基本思想，各有其特点和应用背景。