



数据结构与算法

Data Structure and Algorithm

第10章 排序



目 录

10. 1 基本概念

10. 2 插入排序

10. 3 交换排序

10. 4 选择排序

10. 5 归并排序

10. 6 基数排序

退出



10. 1 基本概念

10.1.1 排序介绍

排序（**Sorting**）是数据处理中一种很重要的运算，同时也是很常用的运算，一般数据处理工作25%的时间都在进行排序。

简单地说，排序就是把一组记录（元素）按照某个域的值的递增（即由小到大）或递减（即由大到小）的次序重新排列的过程。



表10-1 学生档案表

学号	姓名	年龄	性别
99001	王晓佳	18	男
99002	林一鹏	19	男
99003	谢宁	17	女
99004	张丽娟	18	女
99005	周涛	20	男
99006	李小燕	16	女



10.1.2 基本概念

1. 内排序与外排序

按照排序过程中使用内外存的不同将排序方法分为内排序和外排序。

若排序过程全部在内存中进行，则称为内排序；

若排序过程需要不断地进行内存和外存之间的数据交换，则称为外排序。

本章仅讨论内排序。



2. 稳定性

因为排序码可以不是记录的关键字，同一排序码值可能对应多个记录。对于具有同一排序码的多个记录来说，若采用的排序方法使排序后记录的相对次序不变，则称此排序方法是稳定的，否则称为不稳定的。

例如：

数据项	1	2	2+	8	3	8+	
排序后	1	2	2+	3	8	8+	稳定
	1	2+	2	3	8+	8	不稳定



3. 内部排序分类

排序




插入排序（直插排序、二分排序、希尔排序）

交换排序（冒泡排序、快速排序）

选择排序（直选排序、树型排序、堆排序）

归并排序（二路归并排序、多路归并排序）

分配排序（多关键字排序、基数排序）



排序

简单排序法 $O(n^2)$

先进排序法 $O(n\log n)$

基数排序 $O(d*n)$





4. 排序的时间复杂性

排序过程主要是对记录的排序码进行比较和记录的移动过程。因此排序的时间复杂性可以算法执行中的数据比较次数及数据移动次数来衡量。当一种排序方法使排序过程在最坏或平均情况下所进行的比较和移动次数越少，则认为该方法的时间复杂性就越好，分析一种排序方法，不仅要分析它的时间复杂性，而且要分析它的空间复杂性、稳定性和简单性等。




10. 2 插入排序

基本思想：每步将一个待排序的记录，按其关键字的大小插入前面已经排序序列中，知道全部插入完为止。

具体方法：

1. 直接插入排序（straight insertion sort）
 2. 折半插入排序（binary insertion sort）
 3. 希尔排序（shell sort）
- 



10.2.1 直接插入排序

1. 直接插入排序的基本思想

基本思想：把 n 个待排序的元素看成为一个有序表和一个无序表，开始时有序表中只包含一个元素，无序表中包含有 $n-1$ 个元素，排序过程中每次从无序表中取出第一个元素，把它的排序码依次与有序表元素的排序码进行比较，将它插入到有序表中的适当位置，使之成为新的有序表。

例如， $n=6$ ，数组R的六个排序码分别为：17，3，25，14，20，9。它的直接插入排序的执行过程如图9-1所示。

	R[0]	R[1]	R[2]	R[3]	R[4]	R[5]	R[6]
初始状态		(17)	3	25	14	20	9
第一次插入	(3)	(3	17)	25	14	20	9
第二次插入	(25)	(3	17	25)	14	20	9
第三次插入	(14)	(3	14	17	25)	20	9
第四次插入	(20)	(3	14	17	20	25)	9
第五次插入	(9)	(3	9	14	17	20	25)

图10-1 直接插入排序示例

2. 直接插入的算法实现

```
void InsertSort(Sqlist &L)
```

```
//对顺序表L做直接插入排序
```

```
{ for ( int i=2; i<L.length;++ i)
```

```
    //i表示插入次数，共进行n-1次插入
```

```
    if(LT(L.r[i].key,L.r[i-1].key)){
```

```
        L.r[0]=L.r[i];        //复制哨兵
```

```
        for(j=i-2; LT(L.r[0].key,L.r[j].key); --j)
```

```
            L.r[j+1]=L.r[j];
```

```
        L.r[j+1]=L.r[0];
```

```
    }
```

```
}//InsertSort
```

3. 直接插入排序的效率分析

1) 空间：它只需要一个元素的辅助空间

2) 时间：

正序： $C_{\min} = 0$

$$M_{\min} = n-1$$


逆序 $C_{\max} = 2 + \dots + n = (n+2)(n-1)/2$

$$M_{\max} = 3 + 4 + \dots + n + 1 = (n+4)(n-1)/2$$

因此，直接插入排序的时间复杂度为 $O(n^2)$ 。

3) 稳定性：稳定的

结论：适合元素个数不多，且已基本有序的情况



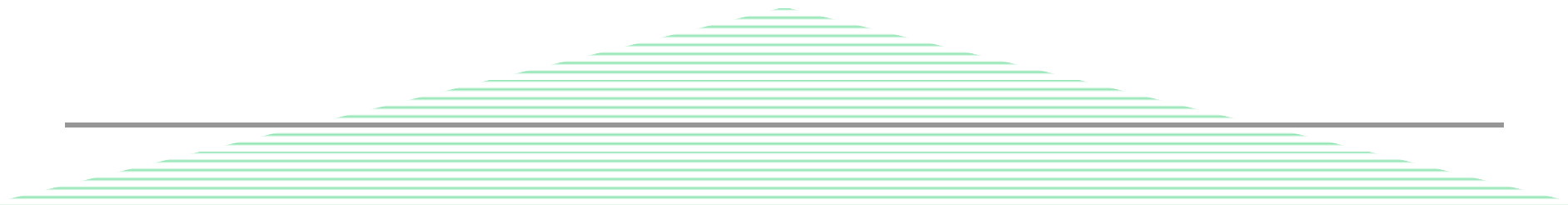
10.2.2二分插入排序

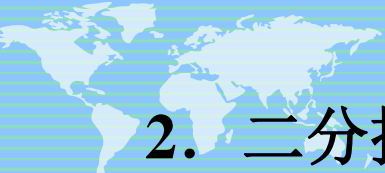
1. 二分插入排序的基本思想

基本思想是：在有序表中采用二分查找的方法查找待排元素的插入位置,在直接插入的基础上，减少比较次数。

其处理过程：先将第一个元素作为有序序列，进行 $n-1$ 次插入，用二分查找的方法查找待排元素的插入位置，将待排元素插入。

例如：[6 13 28 39 41 72 85] 20






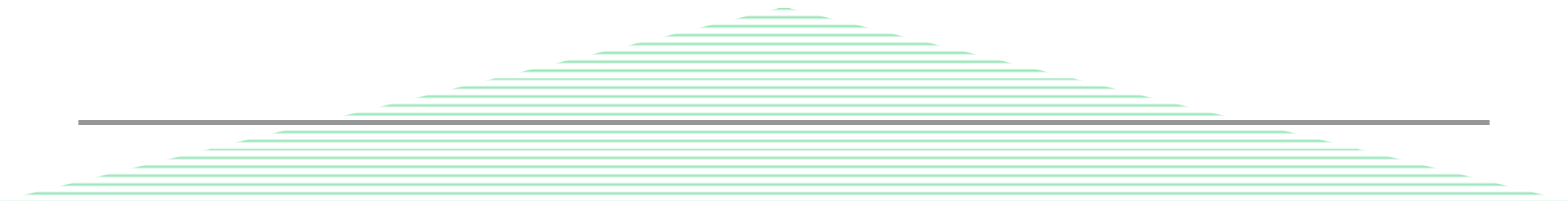
2. 二分插入排序的算法实现

其算法如下：

```
void BInsertSort(SqList &L){  
    //对顺序表L做折半插入排序  
    for( i=2; i<=L.length; ++i) //共进行n-1次插入  
    { L.r[0]=L.r[i];  
        low=1 ;   high=i-1;  
        while(low<=high){  
            m=(low+high)/2;    //取中点  
            if LT(L.R[0].key,L.r[m].key) high=m-1; //取左区间  
            else low=m+1;      //取右区间  
        }  
        for( j=i-1;j>=high+1;- -j) L.r[j+1]=L.r[j] //元素后移空出插入位  
        L.r[high+1]=L.r[0];  
    }  
}
```

3. 二分插入排序的效率分析

- 1)空间:与直接插入排序基本一致;
 - 2)时间上, 前者的比较次数比直接插入查找的最坏情况好, 最好的情况坏, 两种方法的元素的移动次数相同, 因此二分插入排序的时间复杂度仍为 $O(n^2)$ 。
 - 3)稳定性: 二分插入算法与直接插入算法的元素移动一样是顺序的, 因此该方法也是稳定的。
- 



10.2.3 希尔排序

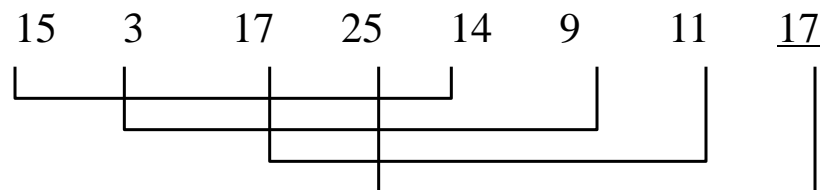
1. 希尔排序的基本思想

希尔排序（Shell Sort）又称为“缩小增量排序”。是1959年由D.L.Shell提出来的。该方法的基本思想是：先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。

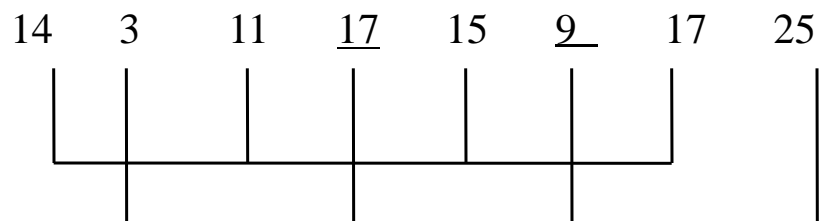
因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前两种方法有较大提高。

例如， $n=8$ ，数组R的八个元素分别为：15，3，17，25，14，9，11，17。下面用图10-2给出希尔排序算法的执行过程。

初始状态， $d=4$



第一趟结果， $d=2$



第二趟结果， $d=1$



第三趟结果



图10-2 希尔排序算法的执行过程



3. shell排序的算法实现

其算法如下：

```
Void ShellSort(SqList &L, int dlta[], int t){  
    //按增量序列dlta[0...t-1]对顺序表L作希尔排序  
    for(k=0;k<t;++ k)  
        ShellInsert(L, dlta[k]);  
} //ShellSort
```




```
void ShellInsert(SqList &L,int dk){  
    //对顺序表L作一趟希尔插入排序  
    for( i=dk+1; i<=L.length; ++i)  
        if LT(L.R[i].key,L.r[i-dk].key) {  
            L.r[0]=L.r[i];  
            for( j=i-dk;j>0&&LT(L.r[0].key,L.r[j].key);j-=dk)  
                L.r[j+dk]=L.r[j] ;  
            L.r[j+dk]=L.r[0];  
        }  
} //ShellInsert
```



4. 希尔排序的效率分析

- 1) 空间：一个记录
- 2) 增量序列不同，移动和比较次数不同，增量最后一趟必为1。希尔排序的时间复杂性在 $O(n \log_2 n)$ 和 $O(n^2)$ 之间，大致为 $O(n^{1.3})$ 。
- 3) 稳定性：不稳定



10. 3 交换排序

基本思想：


两两比较待排序记录的关键字，并交换不满足顺序要求的那些偶对，直到全部满足为止。

具体方法：

冒泡排序（Bubble Sorting）

快速排序（Quick Sorting）





10.3.1 冒泡排序 (Bubble Sorting)

1. 冒泡排序的基本思想

基本思想是：在每一趟排序中，从第一个记录到第 n 个记录相邻两关键字比较，若为逆序，则交换位置，每次排出一个在未排序序列中关键字为最大的记录，直到在一趟排序过程中没有进行过交换记录的操作。

例如， $n=6$ ，数组R的六个排序码分别为：17，3，25，14，20，9。下面用图10-3给出冒泡排序算法的执行过程。

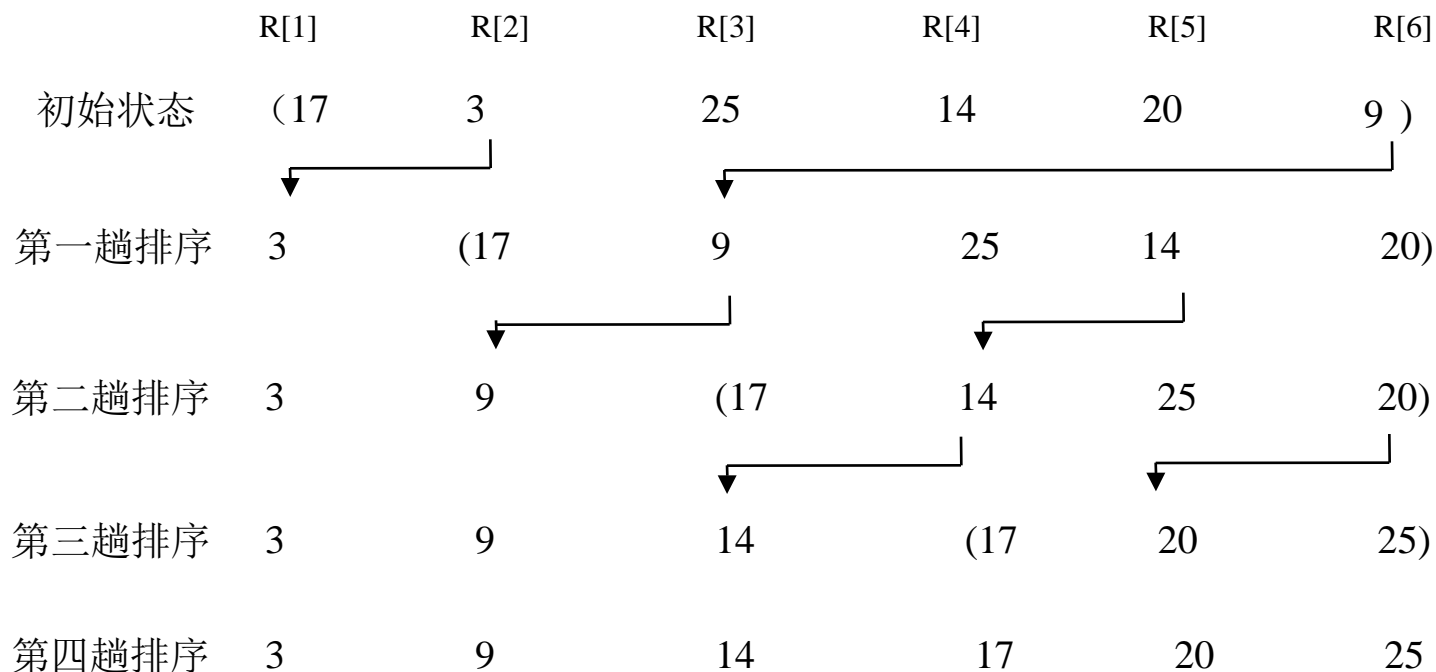


图10-3 冒泡排序示例



2. 冒泡排序的算法实现

```
void Bubblesort( SqList &L ){  
    //对顺序表L作冒泡排序，特征位flag，0表示此次气泡无交换，结束  
    for (int i=1; i<n &&flag==1; i++)  
    { //i表示趟数，最多n-1趟  
        flag=0; //开始时元素未交换  
        for (int j=1; j<=n-i; j++)  
            if (L.r[j+1].key<L.r[j].key) { //发生逆序  
                L.r[j+1].key<->L.r[j].key  
                flag=1; } //交换，并标记发生了交换  
        } //for  
    } // Bubblesort
```



3. 冒泡排序的效率分析

1) 空间：它只需要一个元素的辅助空间

2) 时间：

正序 一趟完成排序

$$C_{\min} = n-1 \quad M_{\min} = 0$$

逆序 $n-1$ 趟完成排序

$$C_{\max} = (n-1) + (n-2) + \dots + 1 = n(n-1)/2$$

$$M_{\max} = 3[(n-1) + (n-2) + \dots + 1] = 3n(n-1)/2$$

因此，直接插入排序的时间复杂度为 $O(n^2)$ 。

3) 稳定性：稳定的

因为冒泡排序算法只进行元素间的顺序移动，所以是一个稳定的算法。由于其中的元素移动较多，所以属于内排序中速度较慢的一种。



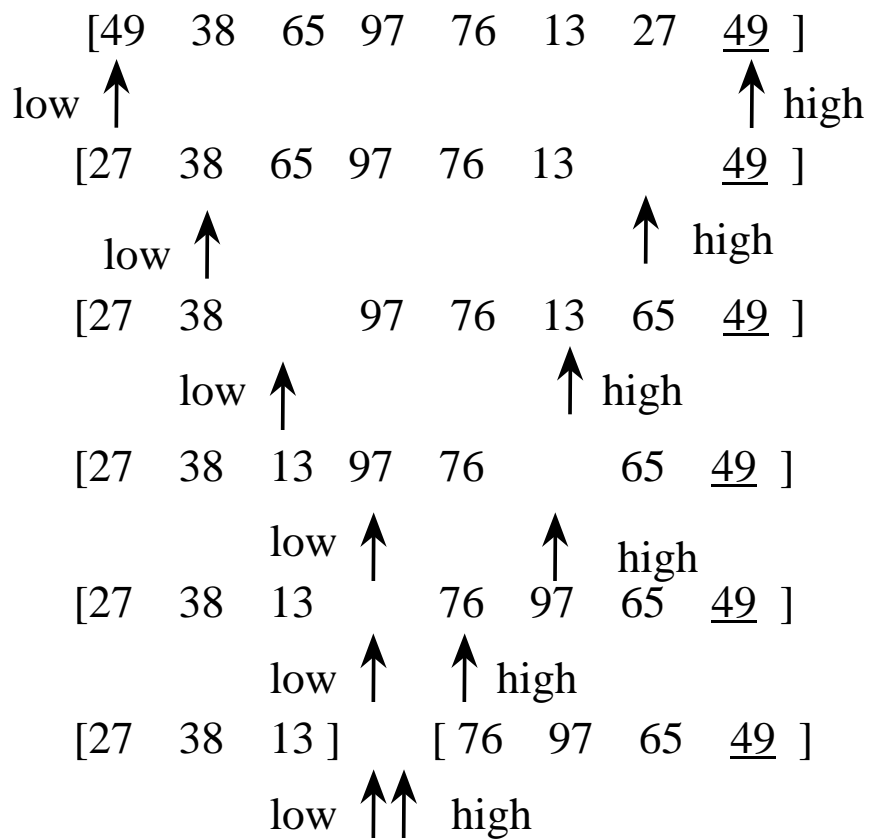
10.3.2 快速排序

1. 快速排序的基本思想

它的基本思想是：任取待排序序列中的某个元素作为基准（一般取第一个元素），通过一趟排序，将待排元素分为左右两个子序列，左子序列元素的排序码均小于或等于基准元素的排序码，右子序列的排序码则大于基准元素的排序码，然后分别对两个子序列继续进行排序，直至整个序列有序。

这个数据元素称作为枢轴（pivotkey），一般情况取待排序序列的第一个数据元素。

例如，给定排序码为：（49，38，65，97，76，13，27，49），具体划分如图9-4所示。



10-4 快速排序的一次划分



2. 快速排序的算法实现

快速排序一次算法1:

```
int Partition(SqList &L,int low,int high){  
    //交换顺序表L中子表L.r[low...high]的记录，使枢轴记录到位，并返回其所  
    //在位置，此时在它之前（后）的记录均不大（小）于它  
    pivotkey=L.r[low].key;  
    While(low<high){  
        while(low<high&&L.r[high].key>=pivotkey)  --high;  
        L.r[low]<->L.r[high];  
        while(low<high&&L.r[low].key<=pivotkey)  ++high;  
        L.r[low]<->L.r[high];  
    }  
    return low;  
}  
//Partition
```



快速排序一次算法2:

```
int Partition(SqList &L,int low,int high){  
    //交换顺序表L中子表L.r[low...high]的记录，使枢轴记录到位，并返回其  
    //所在位置，此时在它之前（后）的记录均不大（小）于它  
    L.r[0] = L.r[low];    pivotkey=L.r[low].key;  
    While(low<high){  
        while(low<high&&L.r[high].key>=pivotkey)  --high;  
        L.r[low] = L.r[high];  
        while(low<high&&L.r[low].key<=pivotkey)  ++high;  
        L.r[high] = L.r[low];  
    }  
    L.r[low] = L.r[0];    return low;  
} //Partition
```



快速排序的递归算法：

```
Void QuickSort(SqList &L){  
    //对顺序表L作快速排序  
    QSort(L,1,L.length);  
} //QuickSort
```

```
void Qsort(SqList &L,int low,int high) {  
    //对顺序表L中的子序序列L.r[low...high]作快速排序  
    if(low<high){  
        pivotloc = Partition(L,low,high);  
        QSort(L,low,pivotloc-1);  
        QSort(L,pivotloc+1,high);  
    }  
} //QSort
```




3. 快速排序的效率分析

1)时间

快速排序的最好时间复杂度应为 $O(n\log_2 n)$ ，快速排序的平均时间复杂度也为 $O(n\log_2 n)$ 。

若快速排序出现最坏的情形（每次能划分成两个子区间，但其中一个为空），蜕化为冒泡排序，因此快速排序的最坏时间复杂度为 $O(n^2)$ 。

2)空间

快速排序所占用的辅助空间为栈的深度，故最好的空间复杂度为 $O(\log_2 n)$ ，最坏的空间复杂度为 $O(n)$ 。

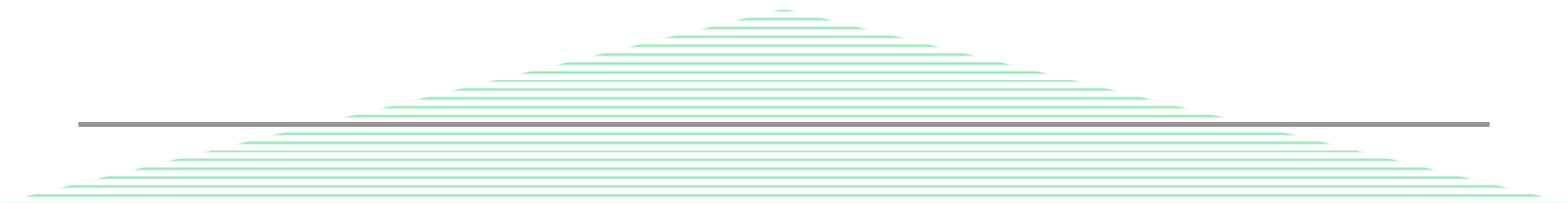
3)稳定性：不稳定



实验四：快速排序的实现

输入任意的无序数据序列，利用快速排序算法实现数据序列的从小到大排序。

实验拓展：10.6（a）和10.6（b）性能作对比





10. 4 选择排序

基本思想：

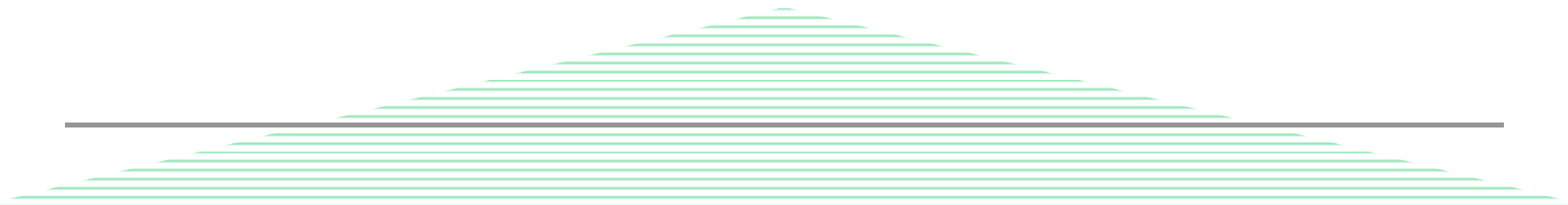
每步从待排序的记录中选出关键字最小的记录，顺序放在已排序的记录序列的最后，直到全部排完。

具体方法：

直接选择排序（straight selection sorting）

树形选择排序（tree selection sort）

堆排序 (heap sort)

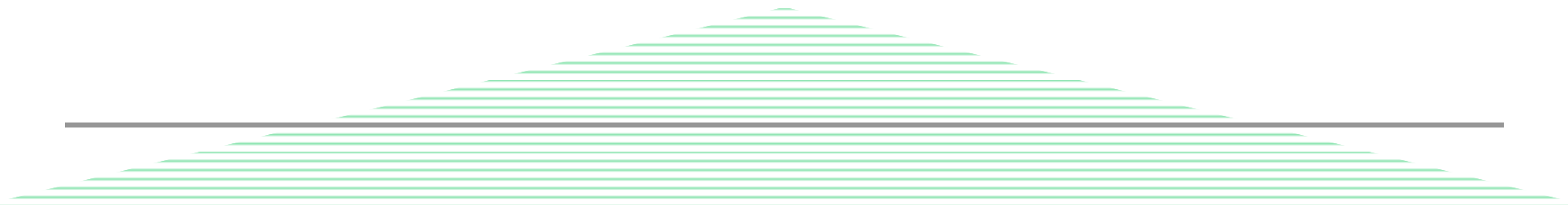




10.4.1 直接选择排序

1. 直接选择排序的基本思想

直接选择排序（straight selection sorting）也是一种简单的排序方法。它的基本思想是：首先从所有记录中选出关键字最小的记录，把它与第一个记录交换，然后在其余的记录中再选出关键字最小的记录与第二个记录交换，以此类推，直到所有记录排序完成，共 $n-1$ 趟。





例如，给定 $n=8$ ，
数组 R 中的8个元
素的排序码为：
(8, 3, 2, 1, 7
, 4, 6, 5)，则
直接选择排序过
程如图10-5所示
。

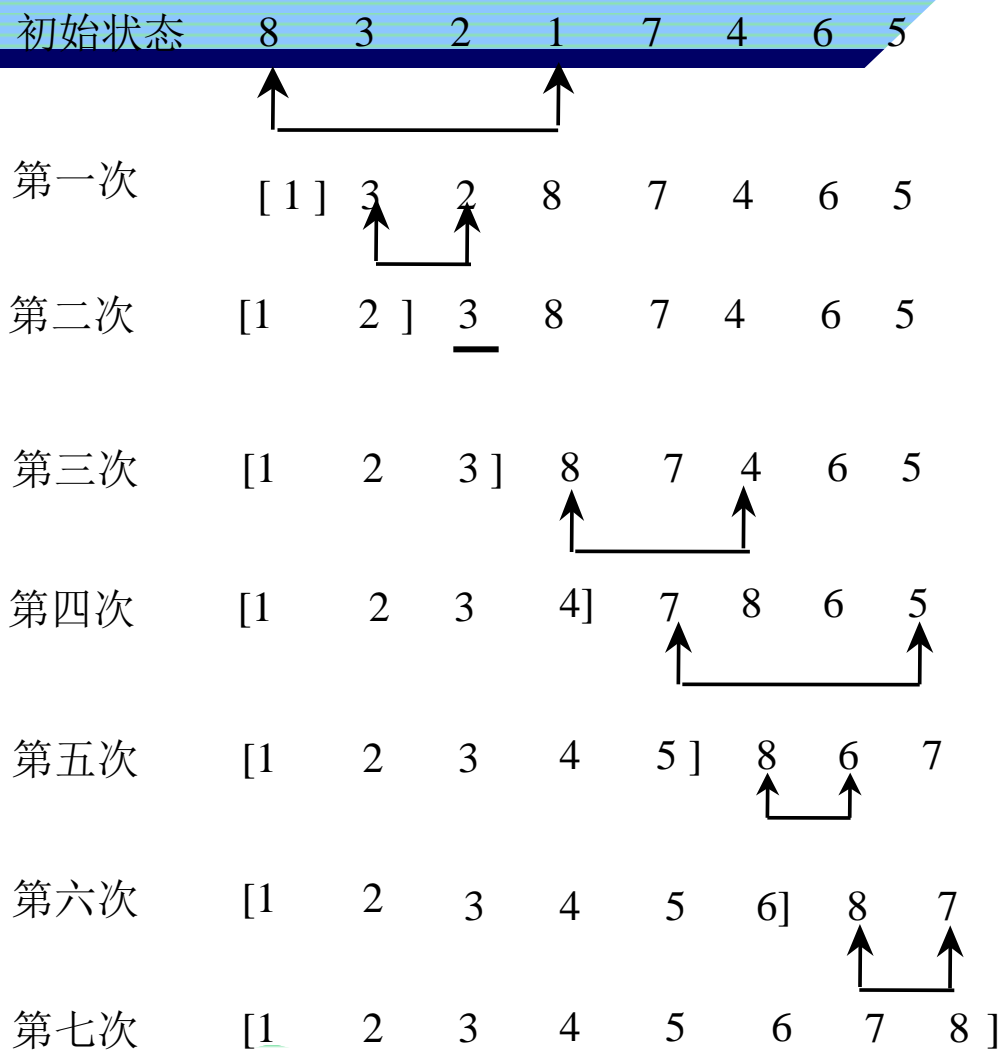

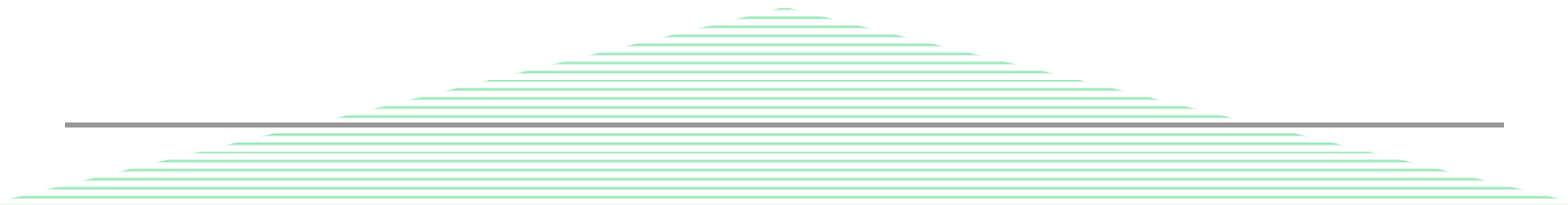



图 10-5 直接选择排序的过程示例



2. 直接选择排序的算法实现

```
Void SelectSort(SqList &L){  
    //对顺序表L作简单选择排序  
    for(i=1;i<L.length;++i) { //选择第i小的记录，并交换到位  
        j=SelectMinKey(L,i);    //在L.r[i...L.length]中选择Key最小记录  
        if(i!=j) L.r[i]<->L.r[j]; //与第i个记录交换  
    }  
} //SelectSort
```





3. 直接选择排序的效率分析

1) 时间

比较次数：与初始顺序没有关系

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

移动次数：与初始顺序有关系

正序 $M_{\min}=0$

每趟都执行交换 $M_{\max} = \sum_{i=1}^{n-1} 3 = 3(n-1)$

由此可知，直接选择排序的最好最差都为 $O(n^2)$ 数量级

2) 空间：一个记录交换空间

3) 稳定性

由于在直接选择排序中存在着不相邻元素之间的互换，因此，直接选择排序是一种不稳定的排序方法。

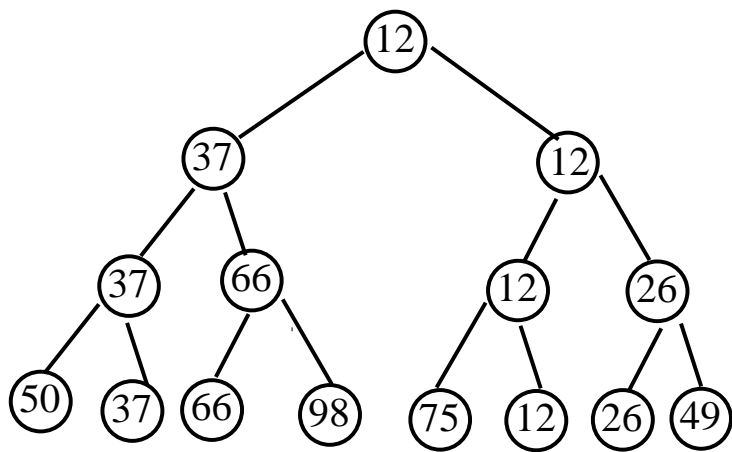


10.4.2 树形选择排序

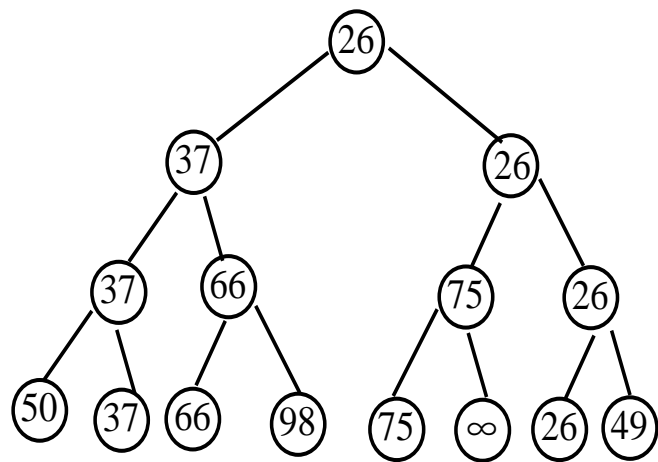
1. 基本思想

树形选择排序（tree selection sorting），又称锦标赛排序（tournament sorting），是一种按照锦标赛的思想进行选择排序的方法，即若甲胜乙，而乙胜丙，则认为甲胜丙。首先对 n 个记录的排序码进行两两比较，然后在其中 $\lceil n/2 \rceil$ 个较小者之间再进行两两比较，如此重复，直到选出最小排序码为止，用一棵倒置的 n 个叶子结点的完全二叉树表示。

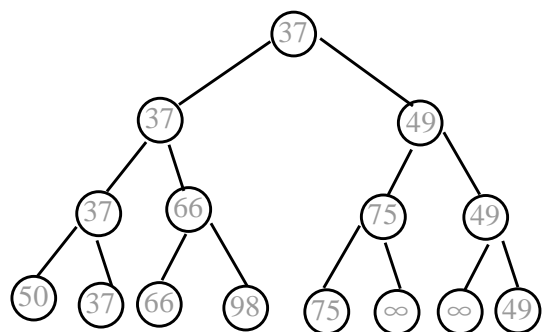
例如，给定排序码头 50, 37, 66, 98, 75, 12, 26, 49，树形选择排序过程见图10-7。



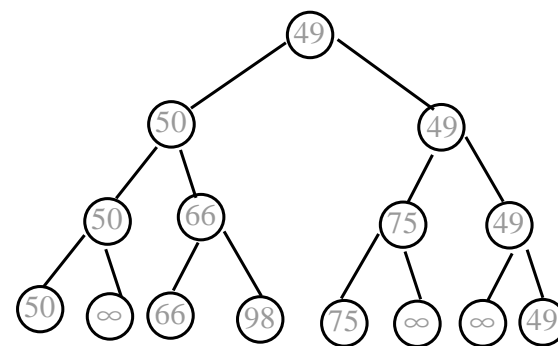
(a) 经过 7 次比较得到最小值 12



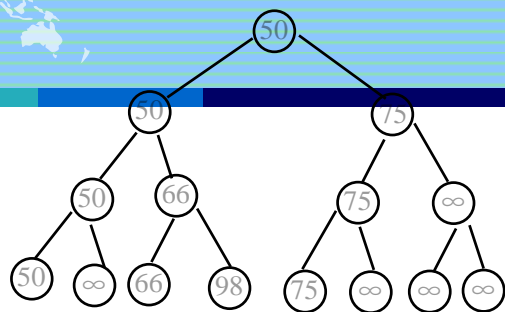
(b) 输出 12 后，经过 2 次比较得到第二小值 26



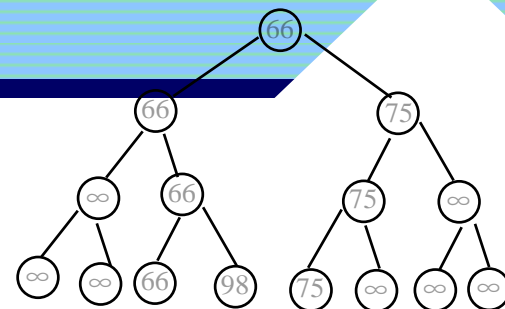
(c) 输出 12, 26 后, 经过 2 次比较得到第三小值 37



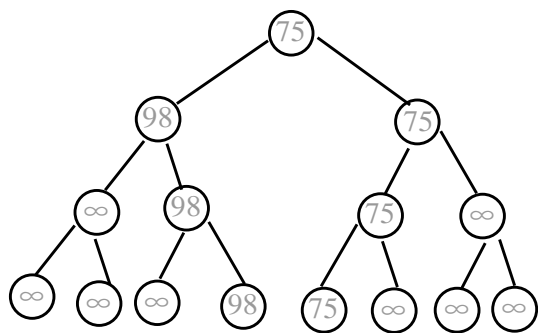
(d) 输出 12,26,37 后, 经过 2 次比较得到第四小值 49



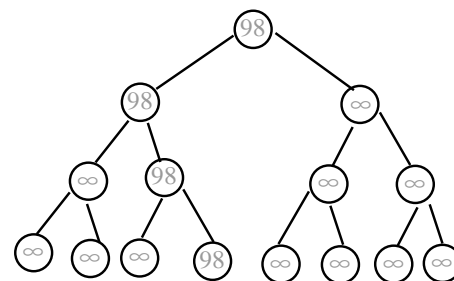
(e) 输出 12,26,37,49 后, 经过 1 次比较得到第五小值 50



(f) 输出 12,26,37,49,50 后, 经过 1 次比较得到第六小值 66



(g) 输出 12,26,37,49,50,66 后, 经过 1 次比较得到第七小值 75



(h) 输出 12,26,37,49,50,66,75 后, 经过 1 次比较得到第八小值 98

图 9-7 树形选择排序示意过程



10.4.3 堆排序

1. 堆的定义

若有 n 个元素的排序码 $k_1, k_2, k_3, \dots, k_n$ ，当满足如下条件：


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

$$(2) \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases}$$

其中 $i=1, 2, \dots, \lfloor n/2 \rfloor$

则称此 n 个元素的排序码 $k_1, k_2, k_3, \dots, k_n$ 为一个堆。

若将此排序码按顺序组成一棵完全二叉树，则（1）称为小顶堆或小根堆（二叉树的所有根结点值小于或等于左右孩子的值），（2）称为大顶堆或大根堆（二叉树的所有根结点值大于或等于左右孩子的值）。



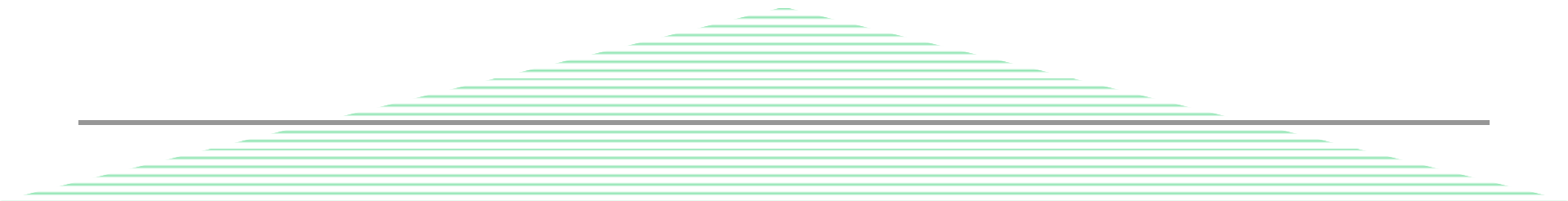
2. 堆排序的基本思想

将排序码初始序列组成一棵完全二叉树，则堆排序可以包含两个阶段：

2.1 建立初始堆

2.2 利用堆进行排序

若排序是从小到大排列，则可以用建立大顶堆实现堆排序，若排序是从大到小排列，则可以用建立小顶堆实现堆排序。

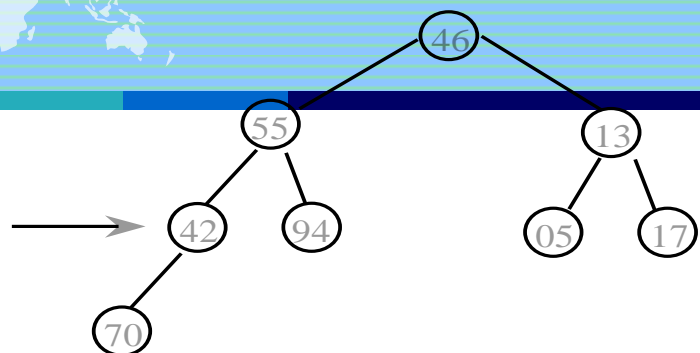




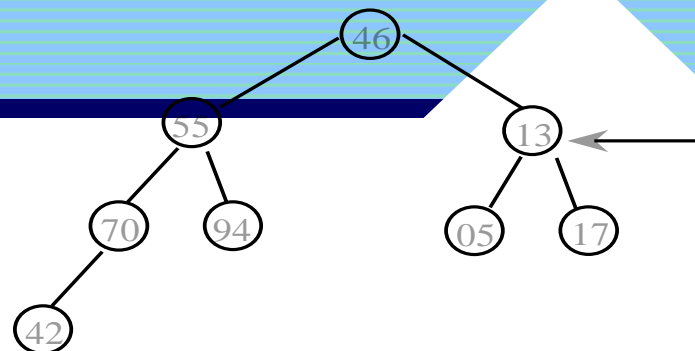
2. 1 初始序列建堆的基本思想

将排序码 $k_1, k_2, k_3, \dots, k_n$ 表示成一棵完全二叉树，然后从第 $\lfloor n/2 \rfloor$ 个排序码（倒数第一个非终端结点）开始筛选，使由该结点作根结点组成的子二叉树符合堆的定义，然后从第 $\lfloor n/2 \rfloor - 1$ 个排序码重复刚才操作，直到第一个排序码止。这时候，该二叉树符合堆的定义，初始堆已经建立。

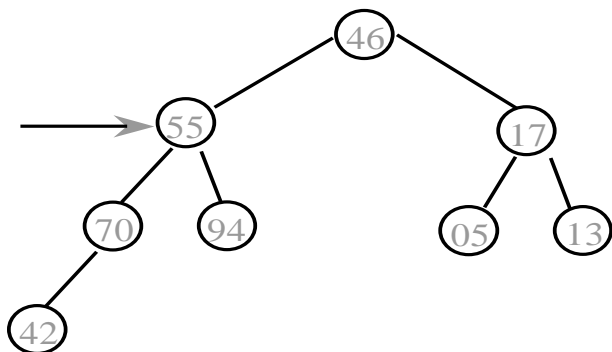
例如，给定排序码46, 55, 13, 42, 94, 05, 17, 70，建立初始大顶堆的过程如图10-8所示。



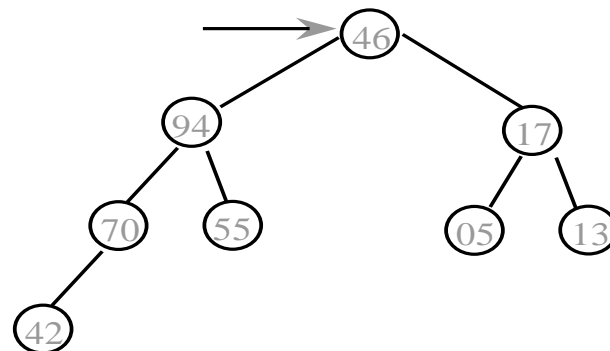
(a) 初始无序的结点，从 42 开始调整



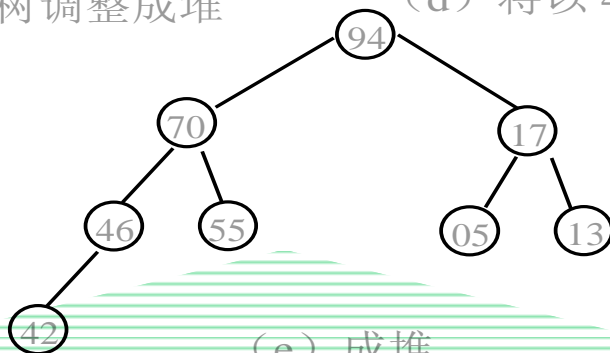
(b) 将以 13 为根的子树调整成堆



(c) 将以 55 为根的子树调整成堆




(d) 将以 46 为根的子树调整成堆



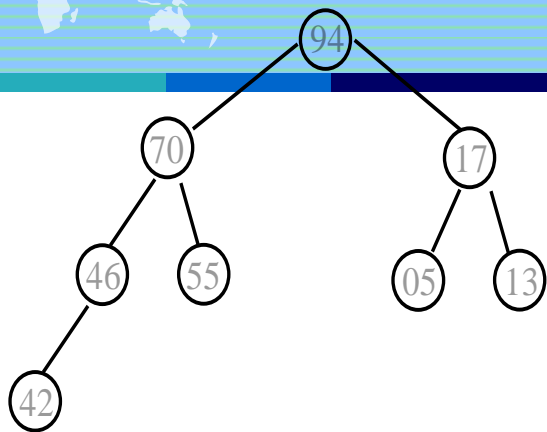
(e) 成堆

图 9-8 建立初始大根堆的过程示意图

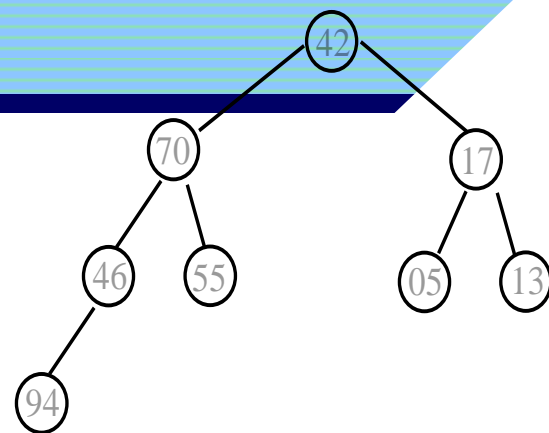


2. 2 堆排序的基本思想

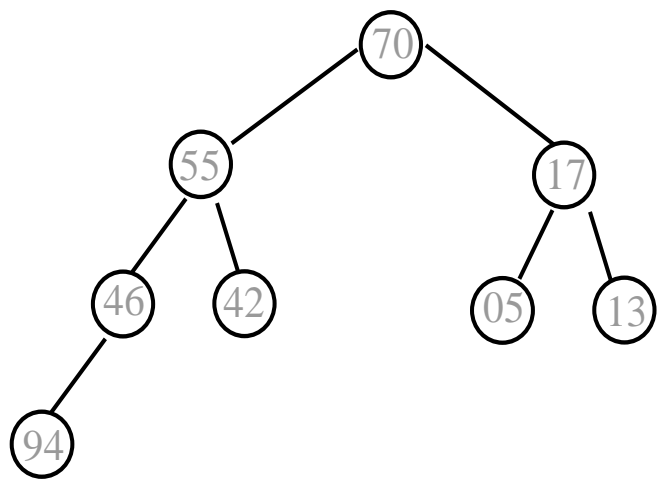
若在输出堆顶的最大值后，使得剩余 $n-1$ 个元素的序列重又建成一个堆，则得到 n 个元素中的次小值，如此反复执行，便能得到一个有序序列。



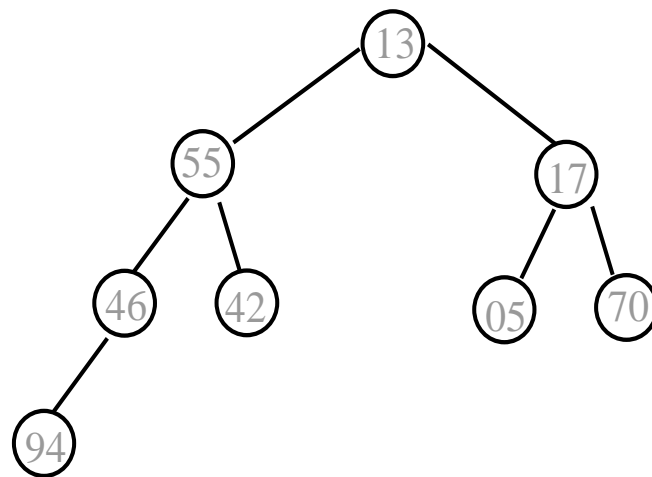
(a) 初始堆



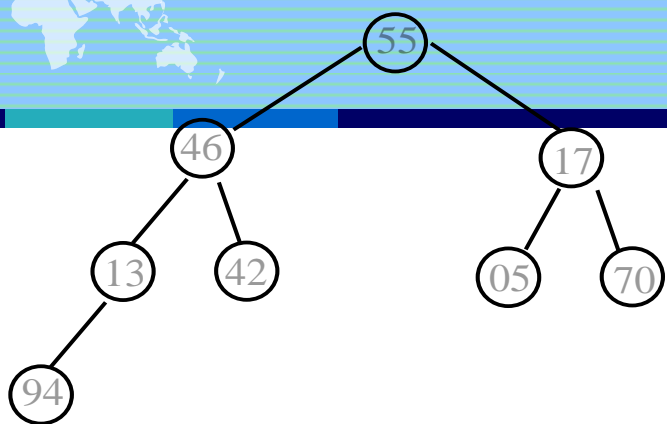
(b) 94 与 42 交换



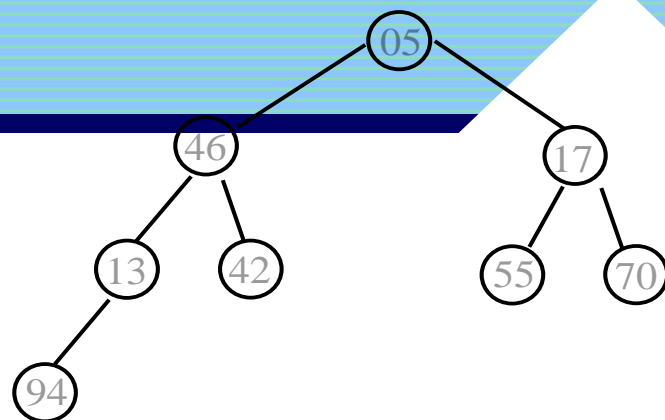
(c) 前 7 个排序码重新建成堆



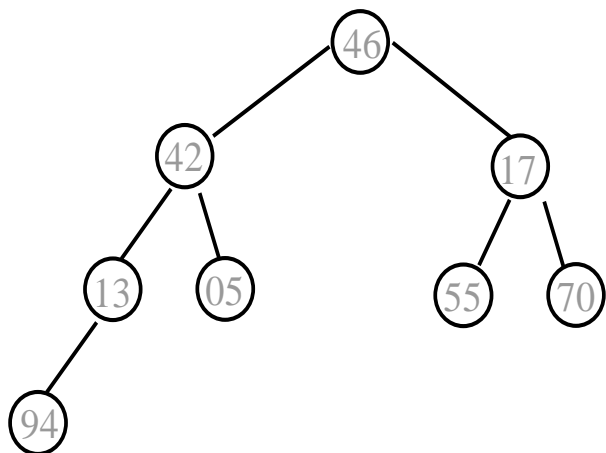
(d) 70 和 13 交换



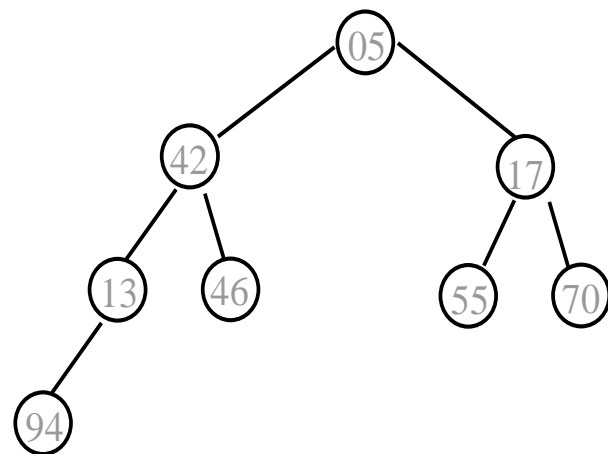
(e) 前 6 个排序码重新建成堆



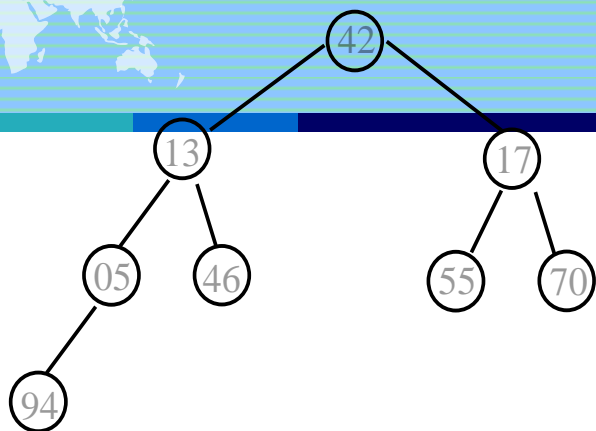
(f) 55 和 05 交换



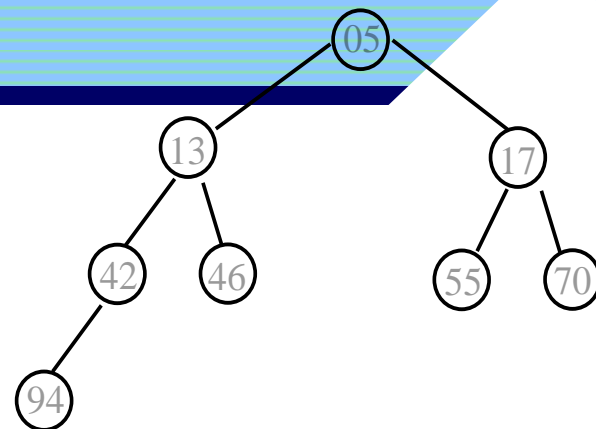
(g) 前 5 个排序码重新建成堆



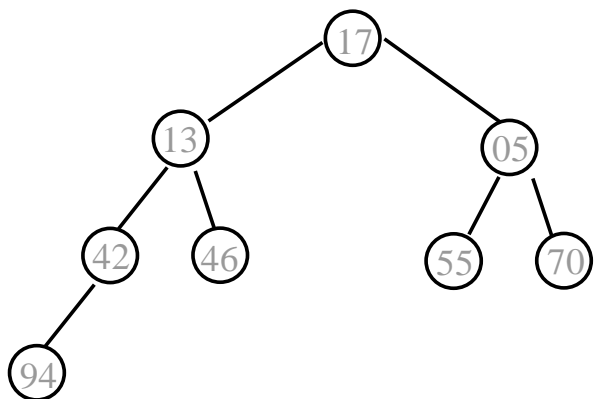
(h) 46 和 05 交换



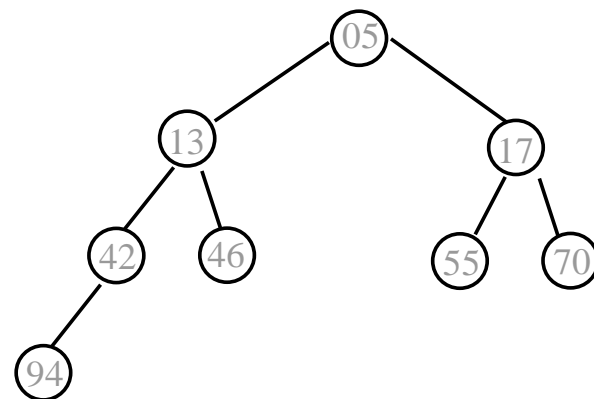
(i) 前 4 个排序码重新建成堆



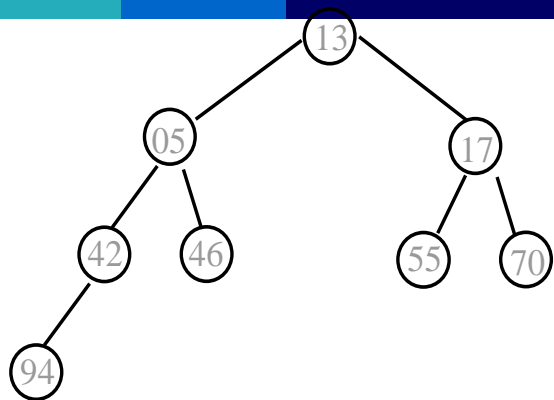
(j) 42 和 05 交换



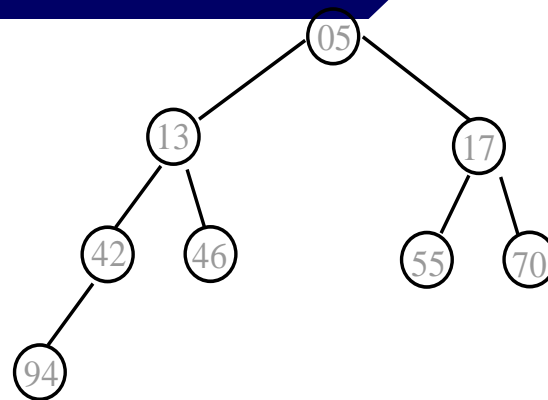
(k) 前 3 个排序码重新建成堆



(l) 17 和 05 交换



(m) 前 2 个排序码重新建成堆



(n) 13 和 05 交换

图 9-9 堆排序过程示意图

从图9-9 (n) 可知，将其结果按完全二叉树形式输出，则得到结果为：05，13，17，42，46，55，70，94，即为堆排序的结果。



3. 算法实现

```
Typedef SqList HeapType;           //堆采用顺序表存储表示
Void HeapAdjust(HeapType &H, int s, int m){
    //已知H.r[s..m]中记录的关键字除H.r[s].key之外均满足堆的定义,
    //本函数调整H.r[s]的关键字, 使H.r[s..m]成为一个大顶堆
    rc=H.r[s];
    for(j=2*s;j<=m;j*=2){
        if(j<m&&LT(H.r[j].key, H.r[j+1].key)) ++j;
        if( !LT(rc.key, H.r[j].key) ) break;
        H.r[s]=H.r[j]; s=j;
    }
    H.r[s]=rc;
}
```



```
Void HeapSort(HeapType &H){  
    //对顺序表H进行堆排序  
    for(i=H.length/2;i>0;--i)    //把H.r[1..H.length]建成大顶堆  
        HeapAdjust(H, i, H.length);  
    for(i=H.length;j>1;--i){  
        H.r[1]<-->H.r[i]; //将堆顶记录和当前未经排序子序列H r[1..i]中  
                           //最后一个记录相互交换  
        //将H.r[1..i-1]重新调整为大顶堆  
        HeapAdjust(H,1,i-1);  
    }  
} //HeapSort
```



4. 堆排序的效率分析

1) 时间 在整个堆排序中，共需要进行 $n + \lfloor n/2 \rfloor - 1$ 次筛选运算，每次筛选运算进行双亲和孩子或兄弟结点的排序码的比较和移动次数都不会超过完全二叉树的深度，所以，每次筛选运算的时间复杂度为 $O(\log_2 n)$ ，故整个堆排序过程的时间复杂度为 $O(n \log_2 n)$ 。

2) 空间：堆排序占用的辅助空间为1（供交换元素用），故它的空间复杂度为 $O(1)$ 。

3) 稳定性：堆排序是一种不稳定的排序方法，例如，给定排序码：2，1，2，它的排序结果为：1，2，2。



10.5 归并排序

已知待排序的记录分为若干部分，每个部分内的记录是有序的，将这些已排序的部分进行合并，得到完全排序的有序序列。

将两个或两个以上的有序表组成一个新的有序表，若每部归并都是将两个字文件合成一个子文件，称为“二路归并排序”。



10.5.1 二路归并排序

1.二路归并排序的基本思想

将两个有序子区间（有序表）合并成一个有序子区间，一次合并完成后，有序子区间的数目减少一半，而区间的长度增加一倍，当区间长度从1增加到 n （元素个数）时，整个区间变为一个，则该区间中的有序序列即为我们所需的排序结果。

例如，给定排序码46，55，13，42，94，05，17，70，二路归并排序过程如图9-10所示。

初始状态: [46] [55] [13] [42] [94] [05] [17] [70]
└──┘ └──┘ └──┘ └──┘
一趟归并: [46 55] [13 42] [05 94] [17 70]
└────────┘ └────────┘
二趟归并: [13 42 46 55] [05 17 70 94]
└────────────────┘
三趟归并: [05 13 17 42 46 55 70 94]

图 9-10 二路归并排序过程示意图

初始状态: [46] [55] [13] [42] [94] [05] [17] [70]
═══════ ════════ ════════ ════════
└──┘ └──┘ └──┘ └──┘
一趟归并: [46 55] [13 42] [05 94] [17 70]
└────────┘ └────────┘
二趟归并: [13 42 46 55] [05 17 70 94]
└────────────────┘
三趟归并: [05 13 17 42 46 55 70 94]

图10-11 二路归并排序过程示意图

例如，给定排序码46，55，13，42，94，05，17，70，二路归并排序过程如图9-10所示。

非递归过程

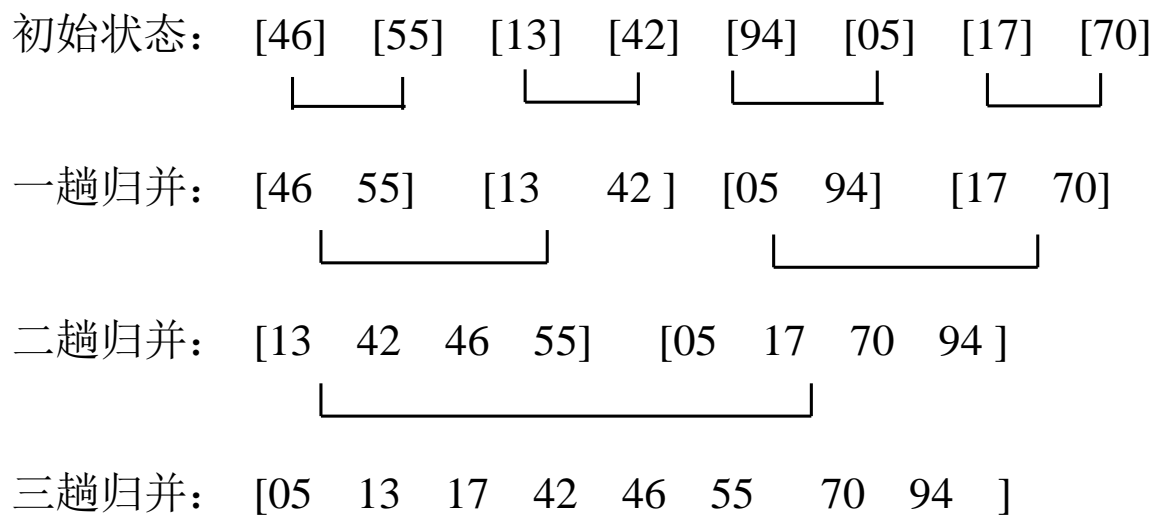



图 9-10 二路归并排序过程示意图



递归过程：

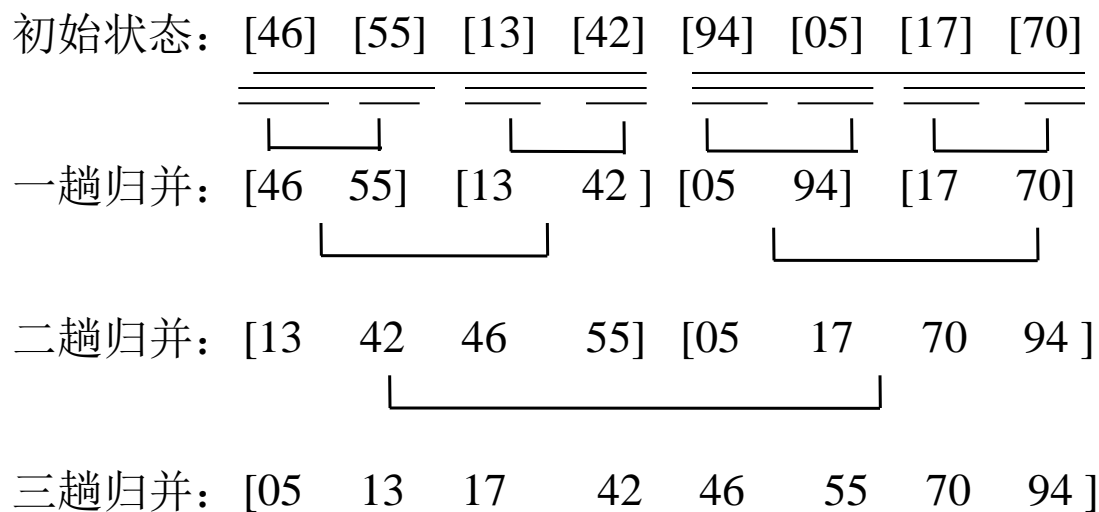
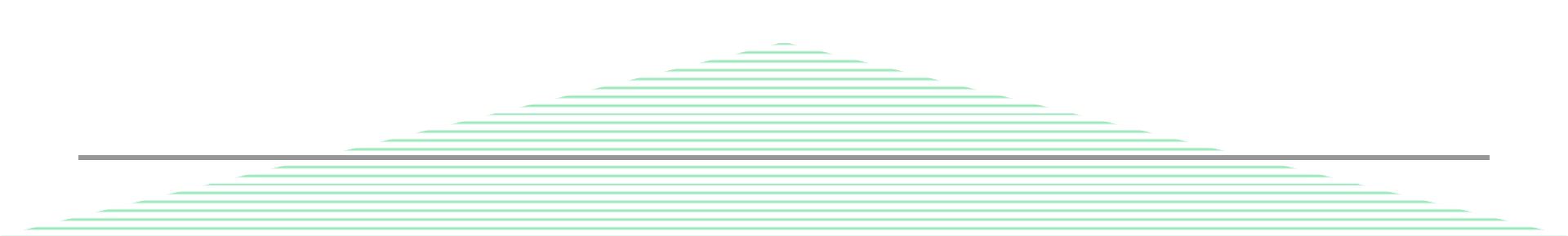


图10-11 二路归并排序过程示意图





2. 算法实现

```
Void MergeSort(SqList &L){  
    //对顺序表L作归并排序  
    MSort(L.r,L.r,1,L.length);  
} //MergeSort
```

```
Void MSort(RcdType SR[],RcdType  
&TR1[],int s,int t){  
    //将SR[s..t]归并排序为TR1[s..t]  
    if(s==t) TR1[s] = SR[s];  
    else{  
        m=(s+t)/2;  
        MSort(SR,TR2, s, m);  
        MSort(SR,TR2, m+1,T);  
        Merge(TR2,TR1,s,m,t);  
    }  
} //MSort
```




```
Void Merge(RcdType SR[], RcdType &TR[], int i, int m, int n){  
    //将有序的SR[i..m]和SR[m+1..n]归并为有序的TR[i..n]  
    for(j=m+1,k=i; i<=m&& j<=n; ++k){  
        if LQ(SR[i].key, SR[j].key)  TR[k]=SR[i++];  
        else TR[k]=SR[j++];  
    }  
    if(i<=m)  TR[k..n] = SR[i..m];  
    if(j<=n)  TR[k..n] = SR[j..n];  
} //Merge
```



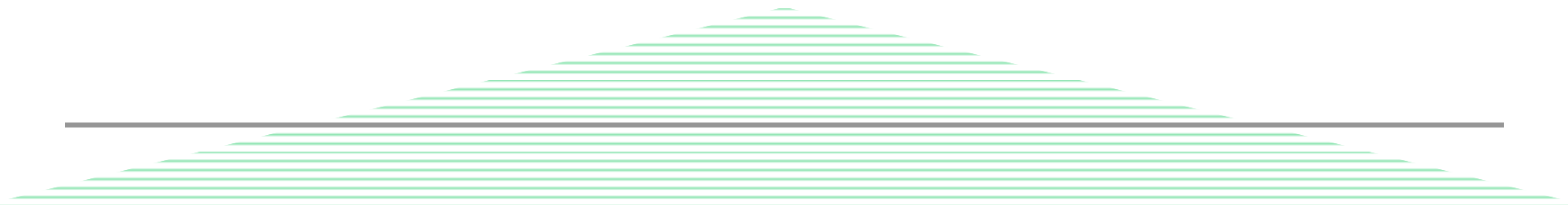
3. 二路归并排序的效率分析

- 1) 时间：最好最差一样，二路归并排序的时间复杂度为 $O(n\log_2 n)$ 。
- 2) 空间：利用二路归并排序时，需要利用与待排序数组相同的辅助数组作临时单元，故该排序方法的空间复杂度为 $O(n)$ ，比前面介绍的其它排序方法占用的空间大。
- 3) 稳定性：稳定



10. 6 基数排序（多关键字排序的思想）

基数排序（radix sorting）是和前面所述各类排序方法完全不同的一种排序方法。在前面几节中，实现排序主要是通过排序码之间的比较和移动两项操作来进行的，而基数排序不需要进行排序码的比较，它是一种借助多关键字（多个排序码）排序的思想来实现单关键字排序的排序方法。



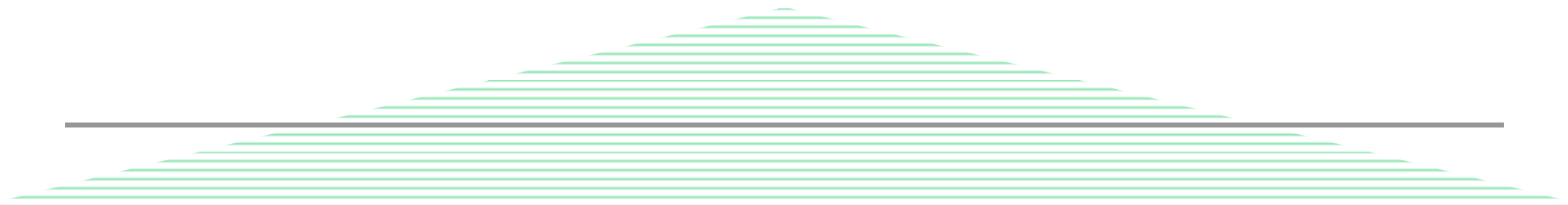


10.6.1 多关键字排序

在实际应用中，有时的排序会需要按几种不同排序码来排序。



例如1：描述一个班级的学生入学信息，既要按总分排序，又要按英语成绩排序，则是一种典型的多关键字排序。

例如2：将一副扑克牌中52张牌按从小到大排列，（规定花色大小为：梅花<方块<红桃<黑桃，面值大小规定为：2<3<4<...<10<J<Q<K<A），则一副扑克牌的排序也是多关键字排序。



例如：扑克牌52张牌排序

两个关键字： K^0 ， K^1

k^0 :    

k^1 : 3 4 5 6 7 8 9 10 J Q k A 2

整理有序两种方法：

1. 先按不同“花色”分成有次序的四堆，每一堆的牌均具有相同的“花色”，然后分别对每一堆按“面值”大小整理有序，再收集起来。

2. 先按不同“面值”分成13堆，接着将这13堆牌从小到大收集一起，然后将这副牌整个颠倒过来再重新按不同“花色”分成4堆，最后将这4堆牌自小至大合在一起。



有两种方法：

MSD法：最高位优先法（Most Significant Digit First）

$k^0 \rightarrow k^1 \rightarrow \dots \rightarrow k^{d-1}$ (d为关键字个数)

排序过程：分配→排序→收集

LSD法：最低位优先法（Least Significant Digit first）

$k^{d-1} \rightarrow k^{d-2} \rightarrow \dots \rightarrow k^0$

排序过程：分配→收集（d次）



10.6.2 链式基数排序

1. 基本思想：LSD思想，若干次分配→收集操作。

例如，给定排序码序列123，78，65，9，108，7，8，3，68，309，基数排序的步骤见图10-11。

分析：数字 $0 \leq k \leq 999$

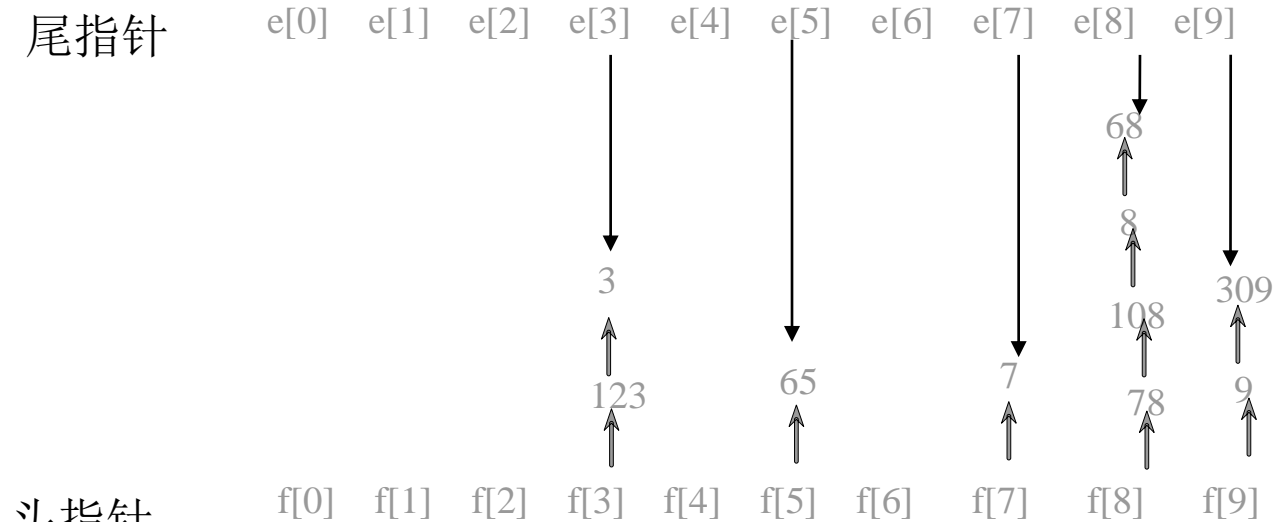
■ 关键字个数三个： $k^0 k^1 k^2$ 并且 $0 \leq k^j \leq 9$

■ 基=10 基：基数的取值范围。



→123→78→65→9→108→7→8→3→68→309

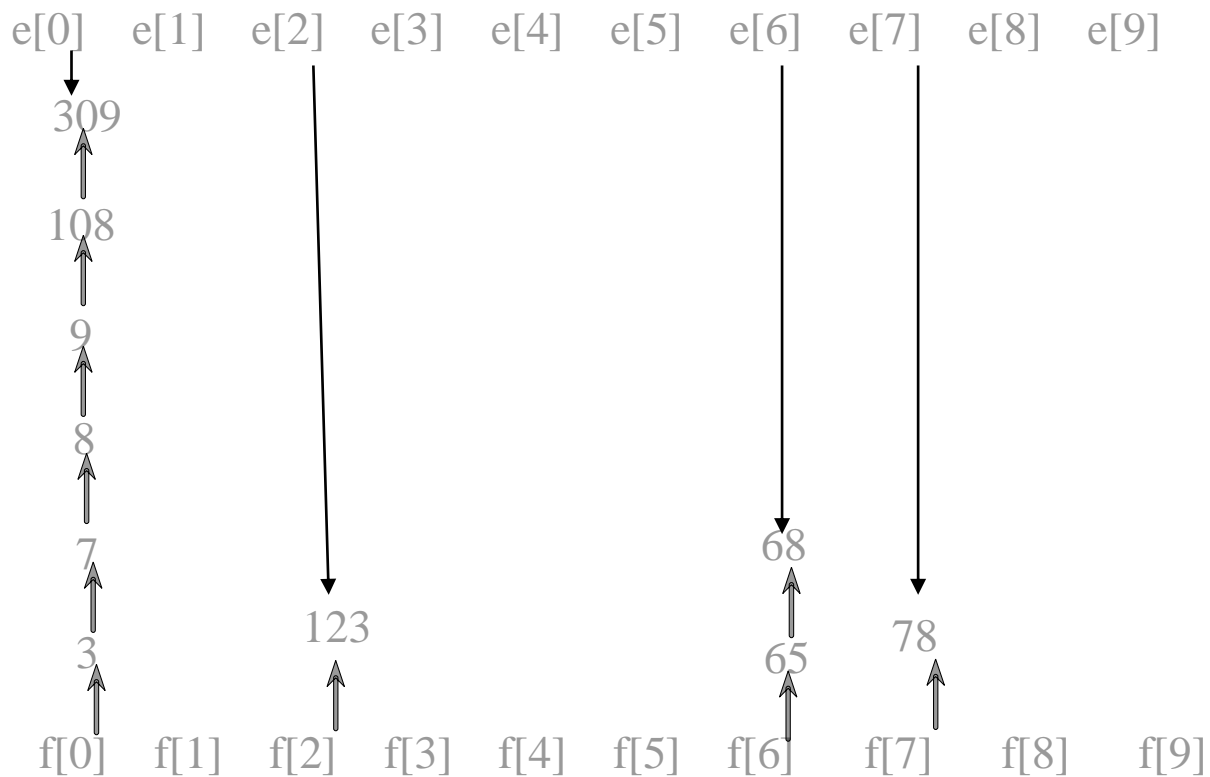
(a) 初始状态



(b) 第一趟分配（按个位，有十个队列）

→123→3→65→7→78→108→8→68→9→309

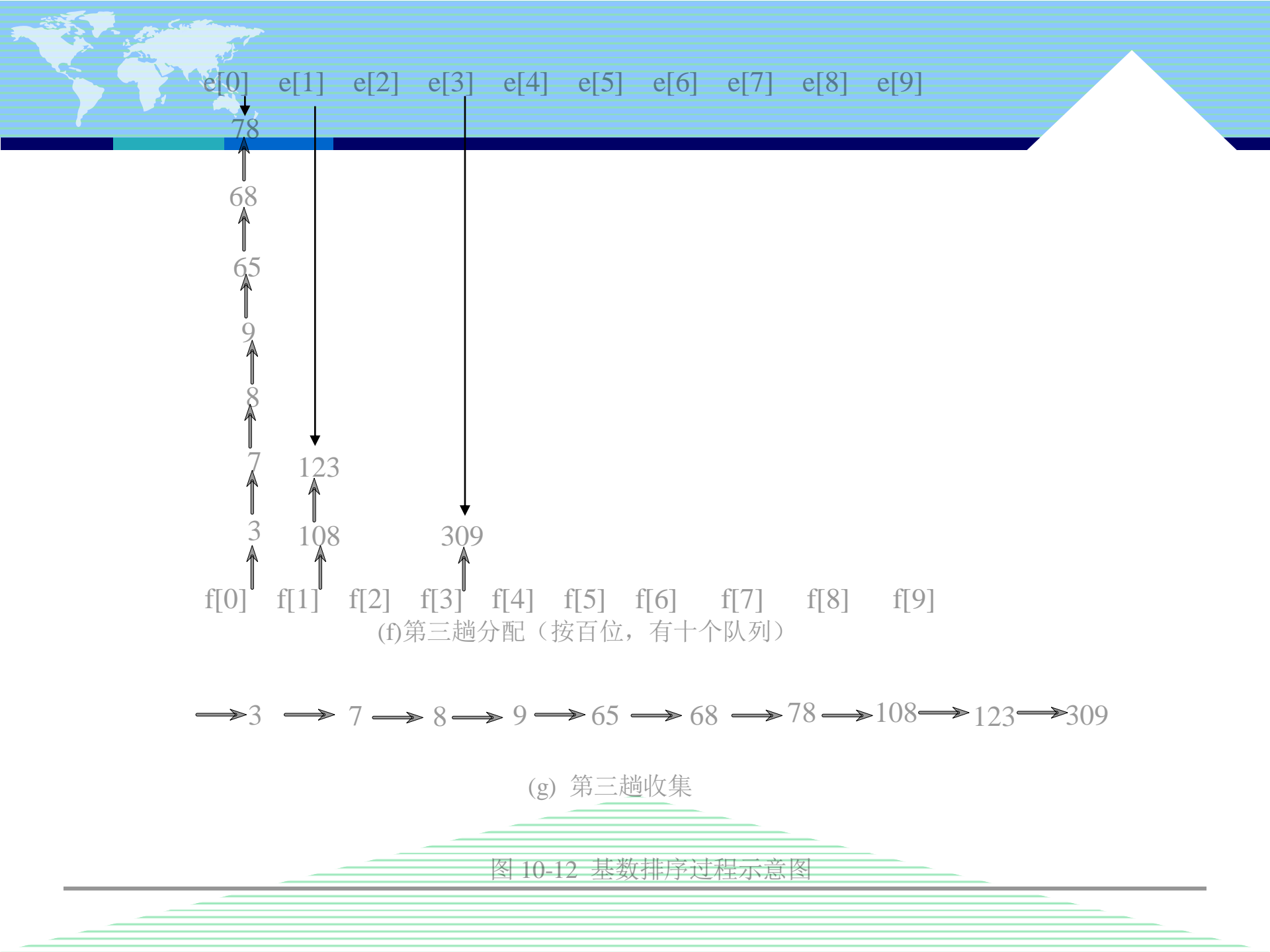
(c) 第一趟收集



(d)第二趟分配（按十位，有十个队列）

→ 3 → 7 → 8 → 9 → 108 → 309 → 123 → 65 → 68 → 78

(e) 第二趟收集



2. 算法实现

数据结构定义


```
#define MAX_NUM_OF_KEY 8           //关键字项数的最大值
#define RADIX 10                   //关键字基数，此时是十进制整数的基数
#define MAX_SPACE 10000
typedef struct{
    KeysType key[MAX_NUM_OF_KEY]; //关键字
    infoType otheritems;          //其他数据项
    int next;
}SLCell;                          //静态链表的结点类型
typedef struct{
    SLCell r[MAX_SPACE];          //静态链表的可利用空间，r[0]为头结点
    int keynum;                   //记录的当前关键字个数
    int recnum;                   //静态链表的当前长度
}SLList;                          //静态链表类型
typedef int ArryType[RADIX];     //指针数组类型
```




2. 算法实现

算法10.15 一次分配操作

```
Void Distribute(SLCell &r, int i, ArrType &f, ArrTye &e){  
//静态链表L的r域中记录已按（keys[0],...,keys[i-1]）有序。  
//本算法按第i个关键字keys[i]建立RADIX个子表，使同一子表中记录的  
//keys[i]相同。F[0..RADIX-1]和e[0..RADIX-1]分别指向各表中第一个  
//和最后一个记录  
    for(i=0;j<Radix;++j) f[j]=0;    //各子表初始化为空表  
    for(p=r[0].next; p ;p=r[p].next){  
        j=ord(r[p].keys[i]); //ord将记录中第i个关键字映射道[0..RADIX-1]  
        if(!f[j]) f[j] = p;  
        else r[e[j]].next = p;  
        e[j] = p;  
    }  
} //Distribute
```



算法10.16 一次收集操作

```
Void Collect(SLCell &r, int i, ArrType f, ArrType e){  
//本算法按keys[i]自小到大将f [0..Radix-1] 所指各子表依次链接成一个链  
//表， e[0..RADIX-1]为各子表的尾指针。  
    for(j=0; !f [j] ; j=succ(j)); //找第一个非空子表， succ为求后继函数  
    r[0].next = f[j]; t=e[j]; //r[0].next指向第一个非空子表中第一个结点  
    while ( j<RADIX ){  
        for( j=succ(j); j<RADIX-1&&!f[j]; j=succ(j) ); //找下一个非空子表  
        if( f [j] ) {r[t].next = f [j]; t=e[j]; }           //链接两个非空子表  
    }  
    r[t].next = 0;           // t指向最后一个非空子表中的最后一个结点  
} //Collect
```



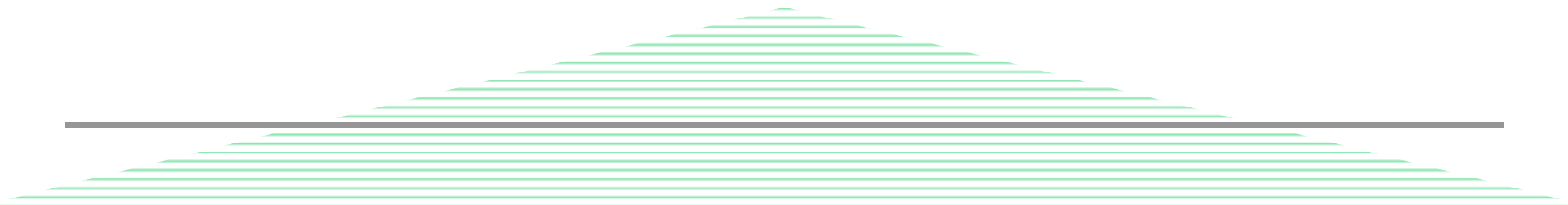
算法10.16 链式基数排序


```
Void RadixSort(SLList &L){  
    //L使采用静态链表表示的顺序表  
    //对L作基数排序，使得L成为按关键字自小到大的有序静态链表，  
    //L.r[0]为头结点  
    for( i=0;i<L.recnum;++i) L.r[i].next =i+1;  
    L.r[L.recnum].next = 0; //将L改造成静态链表  
    for( i=0;i<L.keynum; ++i){ //按最底位优先依次对各关键字进行分配和收集  
        Distribute(L.r, i, f, e); //第i趟分配  
        Collect(L.r, i, f, e);    //第i趟收集  
    }  
} //RadixSort
```



3. 基数排序的效率分析

设有 n 个记录，每个记录含有 d 个关键字，每个关键字取值范围 rd 个值。

- 1) $O(d(n+rd))$ ，一趟分配 $O(n)$,一趟收集 $O(rd)$
 - 2) $2rd$ 个队列指针， n 个指针域空间
 - 3) 由于基数排序中值相同的元素的相对位置在分配和收集中，不会发生变化，所以基数排序是一种稳定的排序方法。
- 




10. 7 各种内排序方法的比较和选择

10.7.1 各种内排序方法的比较

1. 从时间复杂度比较

从平均时间复杂度来考虑，直接插入排序、冒泡排序、直接选择排序是三种简单的排序方法，时间复杂度都为 $O(n^2)$ ，而快速排序、堆排序、二路归并排序的时间复杂度都为 $O(n\log_2 n)$ ，希尔排序的复杂度介于这两者之间。

若从最好的时间复杂度考虑，则直接插入排序和冒泡排序的时间复杂度最好，为 $O(n)$ ，其它的最好情形同平均情形相同。若从最坏的时间复杂度考虑，则快速排序的为 $O(n^2)$ ，直接插入排序、冒泡排序、希尔排序同平均情形相同，但系数大约增加一倍，所以运行速度将降低一半，最坏情形对直接选择排序、堆排序和归并排序影响不大。



2. 从空间复杂度比较


归并排序的空间复杂度最大，为 $O(n)$ ，快速排序的空间复杂度为 $O(\log_2 n)$ ，其它排序的空间复杂度为 $O(1)$ 。

3. 从稳定性比较

直接插入排序、冒泡排序、归并排序是稳定的排序方法，而直接选择排序、希尔排序、快速排序、堆排序是不稳定的排序方法。

4. 从算法简单性比较

直接插入排序、冒泡排序、直接选择排序都是简单的排序方法，算法简单，易于理解，而希尔排序、快速排序、堆排序、归并排序都是改进型的排序方法，算法比简单排序要复杂得多，也难于理解。



10.7.2 各种内排序方法的选择

1. 从时间复杂度选择

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

2. 从空间复杂度选择

尽量选空间复杂度为 $O(1)$ 的排序方法，其次选空间复杂度为 $O(\log_2 n)$ 的快速排序方法，最后才选空间复杂度为 $O(n)$ 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，排序码分布是随机，而对稳定性不做要求时，则采用快速排序为宜。



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

(3) 当待排序元素的个数 n 大，排序码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，则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 小，元素基本有序或分布较随机，且要求稳定时，则采用直接插入排序为宜。

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



作业：

10.1（要有详细演示过程）、10.33