

数据结构与算法

第九章 排序

插入排序
交换排序
选择排序
归并排序
基数排序

第九章 排序

□ 假设含 n 个记录的序列为

$$\{R_1, R_2, \dots, R_n\}$$

其相应的关键字序列为

$$\{K_1, K_2, \dots, K_n\}$$

需确定 $1, 2, \dots, n$ 的一种排列 P_1, P_2, \dots, P_n , 使其相应的关键字满足如下的非递减(或非递增)关系

$$K_{P_1} \leq K_{P_2} \leq \dots \leq K_{P_n}$$

即使记录的序列成为一个按关键字有序的序列

$$\{R_{P_1}, R_{P_2}, \dots, R_{P_n}\}$$

这样一种操作称为排序。

□ 排序定义——将一个数据元素（或记录）的任意序列，重新排列成一个按关键字有序的序列叫~

排序基本操作

□ 排序基本操作

- 比较两个关键字大小
- 将记录从一个位置移动到另一个位置

□ 数据结构(记录)

```
typedef struct {  
    KeyType key;  
    InfoType otherinfo;  
} RecType;  
typedef struct {  
    RecType r[MAXSIZE + 1];  
    int length;  
} SqList;
```

排序分类

□ 按待排序记录所在位置

- 内部排序:待排序记录存放在内存

- 外部排序:排序过程中需对外存进行访问的排序

□ 按排序所需工作量

- 简单的排序方法: $T(n)=O(n^2)$

- 先进的排序方法: $T(n)=O(\log n)$

- 基数排序: $T(n)=O(d \cdot n)$

□ 排序稳定性:假设 $K_i=K_j$ ($1 \leq i \leq n, 1 \leq j \leq n, i \neq j$), 且在排序前的序列中 R_i 领先于 R_j (即 $i < j$)。

- 若在排序后的序列中 R_i 仍领先于 R_j , 则称是稳定的

- 若可能使排序后的序列中 R_j 领先于 R_i , 则称为不稳定的

排序分类

□ 插入排序

- 直接插入排序
- 折半插入排序
- 希尔排序(缩小增量法)

□ 交换排序

- 冒泡排序
- 快速排序

□ 选择排序

- 简单选择排序
- 堆排序

□ 归并排序

□ 基数排序

9.1 插入排序

- 插入排序是一种简单的排序方法，它的基本操作是将一个记录插入到已排好序的有序表中，从而得到一个新的、记录数增1的有序表。
- 直接排序种类
 - 直接插入排序
 - 折半插入排序
 - 希尔排序(缩小增量法)

直接插入排序

- 排序过程：整个排序过程为 $n-1$ 趟插入，即先将序列中第1个记录看成是一个有序子序列，然后从第2个记录开始，逐个进行插入，直至整个序列有序
- 算法描述

直接插入排序例

例

i=1 (49) 38 65 97 76 13 27

i=2 38 (38 49) 65 97 76 13 27

i=3 65 (38 49 65) 97 76 13 27

i=4 97 (38 49 65 97) 76 13 27

i=5 76 (38 49 65 76 97) 13 27

i=6 13 (13 38 49 65 76 97) 27

i=7 27 (13 27 38 49 65 76 97)

排序结果: (13 27 38 49 65 76 97)

直接插入排序

□ 排序过程：整个排序过程为 $n-1$ 趟插入，即先将序列中第1个记录看成是一个有序子序列，然后从第2个记录开始，逐个进行插入，直至整个序列有序

□ 算法描述

```
Void InsertSort(SqList &L) {  
    for(i=2; i<=L.length; ++i)  
        if(LT(L.r[i].key, L.r[i-1].key)){  
            L.r[0]=L.r[i];  
            L.r[i]= L.r[i-1];  
            for(j = i -2; LT(L.r[0].key, L.r[j].key); --j)  
                L.r[j+1] = r[j];  
            L.r[j+1] = L.r[0];  
        }  
}
```

直接插入排序算法评价

□ 时间复杂度——若待排序记录按关键字

$$T(n)=O(n^2)$$

■ 从小到大排列(正序)

☆ 关键字比较次数: $\sum_{i=2}^n 1 = n - 1$

☆ 记录移动次数: 0

■ 从大到小排列(逆序)

☆ 关键字比较次数: $\sum_{i=2}^n i = \frac{(n+2)(n-1)}{2}$

☆ 记录移动次数: $\sum_{i=2}^n (i+1) = \frac{(n+4)(n-1)}{2}$

■ 是随机的, 取平均值

☆ 关键字比较次数: $\frac{n^2}{4}$

☆ 记录移动次数: $\frac{n^2}{4}$

□ 空间复杂度:

■ $S(n)=O(1)$

折半插入排序

- 排序过程：用折半查找方法确定插入位置的排序叫~
- 算法描述

- 算法评价

- 时间复杂度： $T(n)=O(n^2)$
- 空间复杂度： $S(n)=O(1)$

折半插入排序例

例

i=1		(30)	13	70	85	39	42	6	20
i=2	13	(13	30)	70	85	39	42	6	20
⋮									
i=7	6	(6	13	30	39	42	70	85)	20
i=8	20	(6	13	30	39	42	70	85)	20
		↑			↑			↑	
		l			m			h	
i=8	20	(6	13	30	39	42	70	85)	20
		↑	↑	↑					
		l	m	h					
i=8	20	(6	13	30	39	42	70	85)	20
			↑	↑	↑				
			h	l					
i=8	20	(6	13	20	30	39	42	70	85)

折半插入排序

□ 排序过程：用折半查找方法确定插入位置的排序叫~

□ 算法描述

```
void BInsertSort(SqList &L) {  
    for(i=2; i<=L.length; ++i)  
        L.r[0] = L.r[i];  
        low = 1; high = i - 1;  
        while (low <= high) {  
            m = (low+high)/2;  
            if(LT(L.r[0].key, L.r[m].key)) high=m-1;  
            else low = m+1;  
        }  
        for(j=i-1; j>=high+1; --j) L.r[j+1] = L.r[j];  
        L.r[high+1] = L.r[0];  
    }  
}
```

□ 算法评价

■ 时间复杂度： $T(n)=O(n^2)$

■ 空间复杂度： $S(n)=O(1)$

希尔排序

□ 排序过程：先取一个正整数 $d_1 < n$ ，把所有相隔 d_1 的记录放一组，组内进行直接插入排序；然后取 $d_2 < d_1$ ，重复上述分组和排序操作；直至 $d_i = 1$ ，即所有记录放进一个组中排序为止

希尔排序例

例 初始: 49 38 65 97 76 13 27 48 55 4

取 $d_1=5$

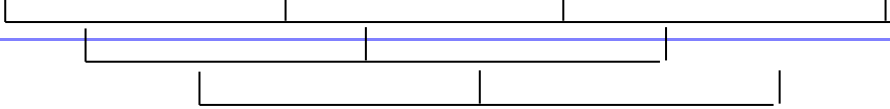
一趟分组: 49 38 65 97 76 13 27 48 55 4



一趟排序: 13 27 48 55 4 49 38 65 97 76

取 $d_2=3$

二趟分组: 13 27 48 55 4 49 38 65 97 76



二趟排序: 13 4 48 38 27 49 55 65 97 76

取 $d_3=1$

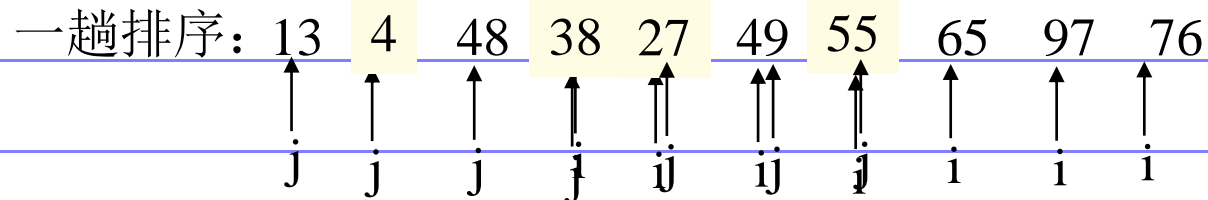
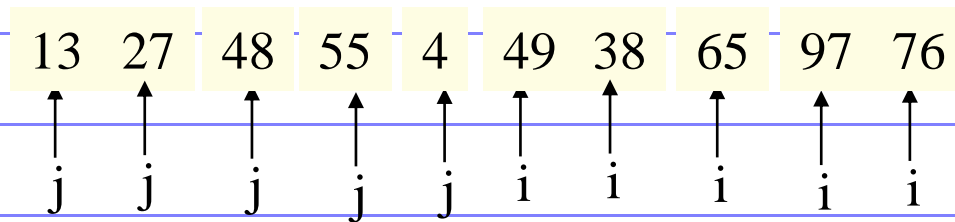
三趟排序: 4 13 27 38 48 49 55 65 76 97

希尔排序例

```
#define T 3
```

```
int d[]={5,3,1};
```

例



二趟排序: 13 4 48 38 27 49 55 65 97 76

希尔排序

□ 算法描述

```
void ShellInsert(SqList &L, int dk) {  
    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); i-=dk) {  
                L.[j+dk] = L.r[j];  
                L.r[j+dk] = L.r[0];  
            }  
        }  
}  
void ShellSort(SqList &L, int dlta[], int t) {  
    for(k=0; k<t; ++k) ShellInsert(L, dlta[k]);  
}
```

希尔排序特点

□ 子序列的构成不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列

■ 希尔排序可提高排序速度，因为

✧ 分组后 n 值减小， n^2 更小，而 $T(n)=O(n^2)$ ，所以 $T(n)$ 从总体上看是减小了

✧ 关键字较小的记录跳跃式前移，在进行最后一趟增量为1的插入排序时，序列已基本有序

■ 增量序列取法

✧ 无除1以外的公因子

✧ 最后一个增量值必须为1

9.2 交换排序

- 交换排序在原表上进行。在排序过程中，比较后交换元素位置。
- 主要方法
 - 冒泡排序
 - 快速排序

冒泡排序

□ 排序过程

- 将第一个记录的关键字与第二个记录的关键字进行比较，若为逆序 $r[1].key > r[2].key$ ，则交换；然后比较第二个记录与第三个记录；依次类推，直至第 $n-1$ 个记录和第 n 个记录比较为止——**第一趟冒泡排序**，结果关键字最大的记录被安置在最后一个记录上
- 对前 $n-1$ 个记录进行第二趟冒泡排序，结果使关键字次大的记录被安置在第 $n-1$ 个记录位置
- 重复上述过程，直到“在一趟排序过程中没有进行过交换记录的操作”为止

冒泡排序例

例	38	38	38	38	13	13	13
	49	49	49	13	27	27	27
	65	65	13	27	30	30	30
	76	13	27	30	38	38	
	13	27	30	49	49		
	27	30	65	65			
	30	76	76				
	97	97					
初始	第一趟	第二趟	第三趟	第四趟	第五趟	第六趟	

冒泡排序算法评价

□ 时间复杂度 $T(n)=O(n^2)$

■ 最好情况（正序）

☆ 比较次数： **$n-1$**

☆ 移动次数：**0**

■ 最坏情况（逆序）

☆ 比较次数

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

☆ 移动次数：

$$3 \sum_{i=1}^n (n-i) = \frac{3}{2}(n^2 - n)$$

□ 空间复杂度：

■ **$S(n)=O(1)$**

快速排序

- 基本思想：通过一趟排序，将待排序记录分割成独立的两部分，其中一部分记录的关键字均比另一部分记录的关键字小，则可分别对这两部分记录进行排序，以达到整个序列有序
- 排序过程：对 $r[s.....t]$ 中记录进行一趟快速排序，附设两个指针 i 和 j ，设枢轴记录 $rp=r[s]$ ， $x=rp.key$
 - 初始时令 $i=s, j=t$
 - 首先从 j 所指位置向前搜索第一个关键字小于 x 的记录，并和 rp 交换
 - 再从 i 所指位置起向后搜索，找到第一个关键字大于 x 的记录，和 rp 交换
 - 重复上述两步，直至 $i=j$ 为止
 - 再分别对两个子序列进行快速排序，直到每个子序列只含有一个记录为止

快速排序例

例

初始关键字:

			X=49				
	27	38	13	49	76	97	65 50
	↑	↑	↑	↑↑	↑	↑	↑
	i	i	i	ij	j	j	j

完成一趟排序: (27 38 13) 49 (76 97 65 50)

分别进行快速排序: (13) 27 (38) 49 (50 65) 76 (97)

快速排序结束: 13 27 38 49 50 65 76 97


```

int qusort(int s[],int start,int end)    //自定义函数 qusort()
{
    int i,j;    //定义变量为基本整型
    i=start;    //将每组首个元素赋给i
    j = end;    //将每组末尾元素赋给j
    s[0]=s[start];    //设置基准值
    while(i<j)
    {
        while(i<j&& s[0]<s[j])
            j--;    //位置左移
        if(i<j)
        {
            s[i]=s[j];    //将s[j]放到s[i]的位置上
            i++;    //位置右移
        }
        while(i<j&& s[i]<=s[0])
            i++;    //位置左移
        if(i<j)
        {
            s[j]=s[i];    //将大于基准值的s[j]放到s[i]位置
            j--;    //位置左移
        }
    }
    s[i]=s[0];    //将基准值放入指定位置
    if (start<i)
        qusort(s, start, j-1);    //对分割出的部分递归调用qusort()函数
    if (i<end)
        qusort(s, j+1, end);
    return 0;
}

```

快速排序算法评价

□ 时间复杂度 $T(n) = O(n^2)$

■ 最好情况（每次总是选到中间值作枢轴）

$$T(n) = O(n \log_2 n)$$

■ 最坏情况（每次总是选到最小或最大元素作枢轴）

$$T(n) = O(n^2)$$

□ 空间复杂度：需栈空间以实现递归

■ 最坏情况： $S(n) = O(n)$

■ 一般情况： $S(n) = O(\log_2 n)$

9.3 选择排序

□ 选择排序(selection sort)的基本思想是；每一趟在 $n-i+1$ ($i=1, 2, \dots, n-1$)个记录中选取关键字最小的记录作为有序序列中第 i 个记录。

- 简单选择排序
- 树形选择排序
- 堆排序

简单选择排序

□ 排序过程

- 首先通过 $n-1$ 次关键字比较，从 n 个记录中找出关键字最小的记录，将它与第一个记录交换
- 再通过 $n-2$ 次比较，从剩余的 $n-1$ 个记录中找出关键字次小的记录，将它与第二个记录交换
- 重复上述操作，共进行 $n-1$ 趟排序后，排序结束

简单选择排序例



简单选择排序算法评价

□ 时间复杂度 $T(n) = O(n^2)$

■ 记录移动次数

☆ 最好情况：0

☆ 最坏情况： $3(n-1)$

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

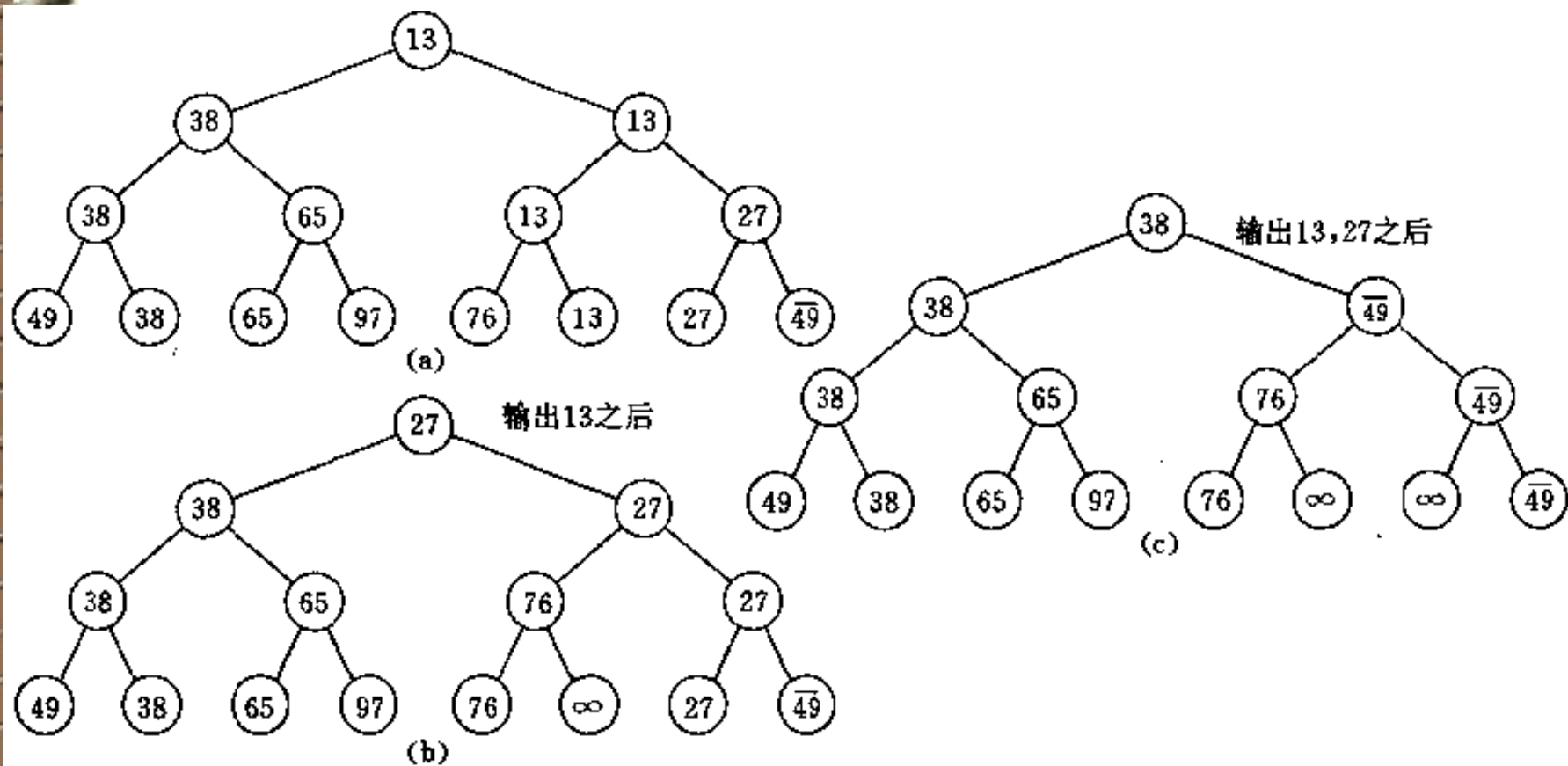
□ 空间复杂度：

■ $S(n) = O(1)$

树形选择排序

- ❑ 树形选择排序(Tree Selection Sort), 又称锦标赛排序(Tournament Sort), 是一种按照锦标赛的思想进行选择排序的方法。首先对 n 个记录的关键字进行两两比较, 然后在其中 $[n/2]$ 个较小者之间再进行两两比较, 如此重复, 直至选出最小关键字的记录为止。这个过程可用一棵有 n 个叶子结点的完全二叉树表示。
- ❑ 由于含有 n 个叶子结点的完全二叉树的深度为 $\lceil \log_2 n \rceil + 1$, 则在树形选择排序中, 除了最小关键字之外, 每选择一个次小关键字仅需进行 $\lceil \log_2 n \rceil$ 次比较, 因此, 它的时间复杂度为 $O(n \log_2 n)$ 。但是, 这种排序方法尚有辅助存储空间较多、和“最大值”进行多余的比较等缺点。

树形选择排序



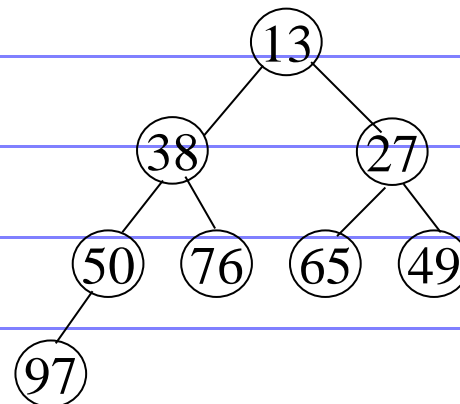
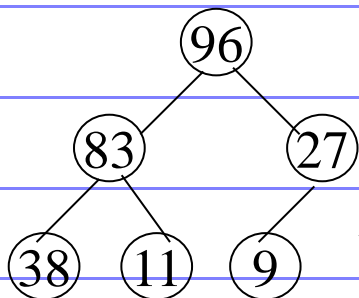
堆排序

□ 堆的定义：n个元素的序列(k_1, k_2, \dots, k_n)，当且仅当满足下列关系时，称之为堆

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases} \quad (i=1, 2, \dots, \lfloor n/2 \rfloor)$$

例 (96, 83, 27, 38, 11, 9)

例 (13, 38, 27, 50, 76, 65, 49, 97)



可将堆序列看成完全二叉树，则堆顶元素（完全二叉树的根）必为序列中n个元素的最小值或最大值

堆排序方法

- 堆排序：将无序序列建成一个堆，得到关键字最小（或最大）的记录；输出堆顶的最小（大）值后，使剩余的 $n-1$ 个元素重又建成一个堆，则可得到 n 个元素的次小值；重复执行，得到一个有序序列，这个过程叫~
- 堆排序需解决的两个问题：
 - 如何由一个无序序列建成一个堆？
 - 如何在输出堆顶元素之后，调整剩余元素，使之成为一个新的堆？

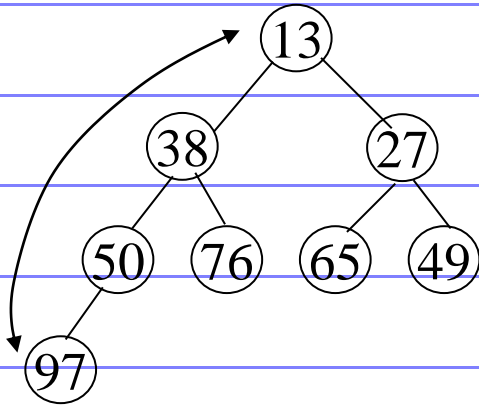
堆排序——筛选

□ 第二个问题解决方法——筛选

- 输出堆顶元素之后，以堆中最后一个元素替代之；然后将根结点值与左、右子树的根结点值进行比较，并与其中小者进行交换；重复上述操作，直至叶子结点，将得到新的堆，称这个从堆顶至叶子的调整过程为“筛选”

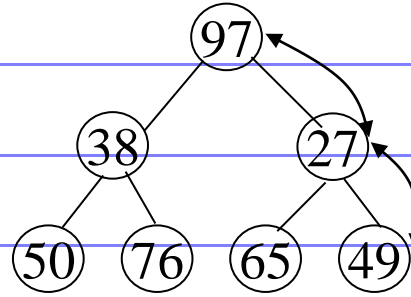
堆排序筛选例

例



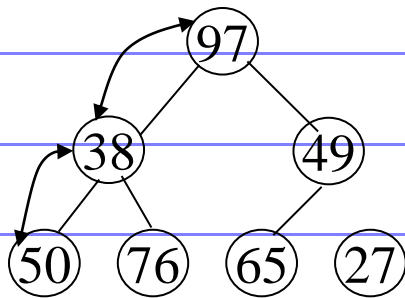
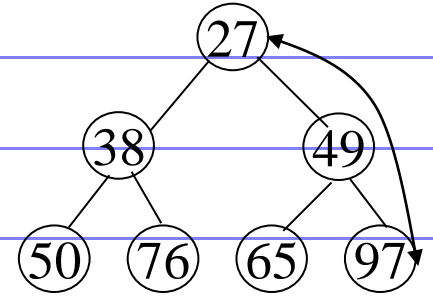
13

输出: 13



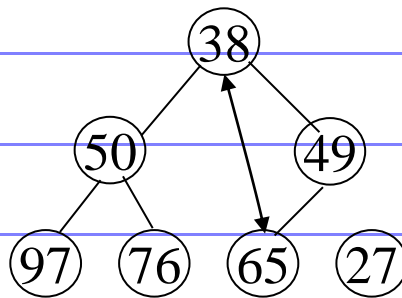
13

输出: 13



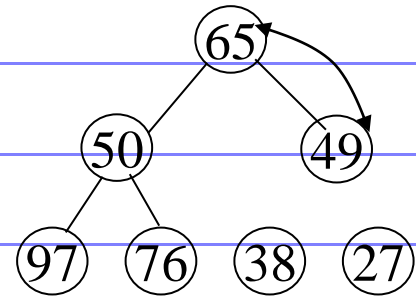
13

输出: 13 27



13

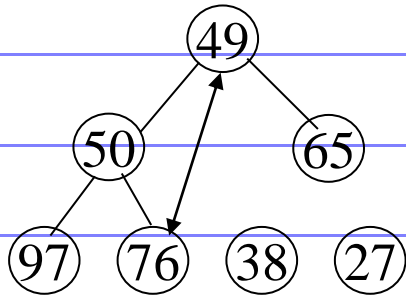
输出: 13 27



13

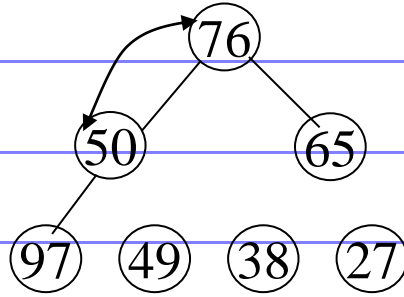
输出: 13 27 38

堆排序筛选例



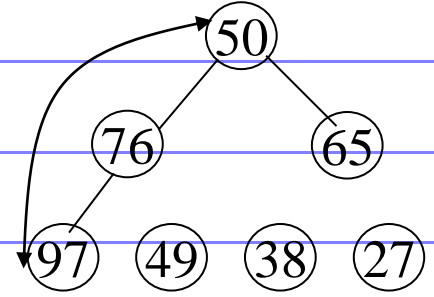
⑬

输出: 13 27 38



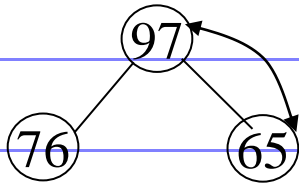
⑬

输出: 13 27 38 49



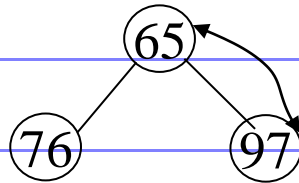
⑬

输出: 13 27 38 49



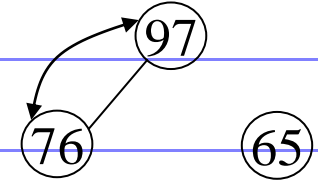
⑬

输出: 13 27 38 49 50



⑬

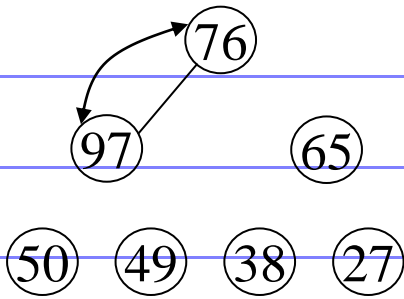
输出: 13 27 38 49 50



⑬

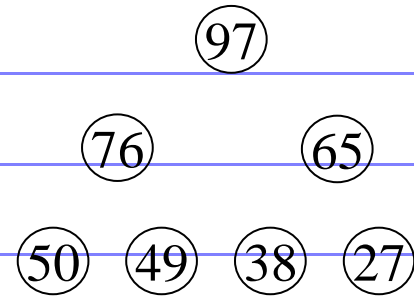
输出: 13 27 38 49 50 65

堆排序筛选例



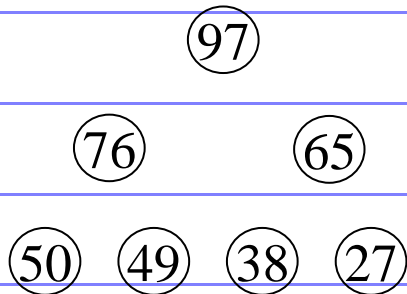
⑬

输出: 13 27 38 49 50 65



⑬

输出: 13 27 38 49 50 65 76



⑬

输出: 13 27 38 49 50 65 76 97

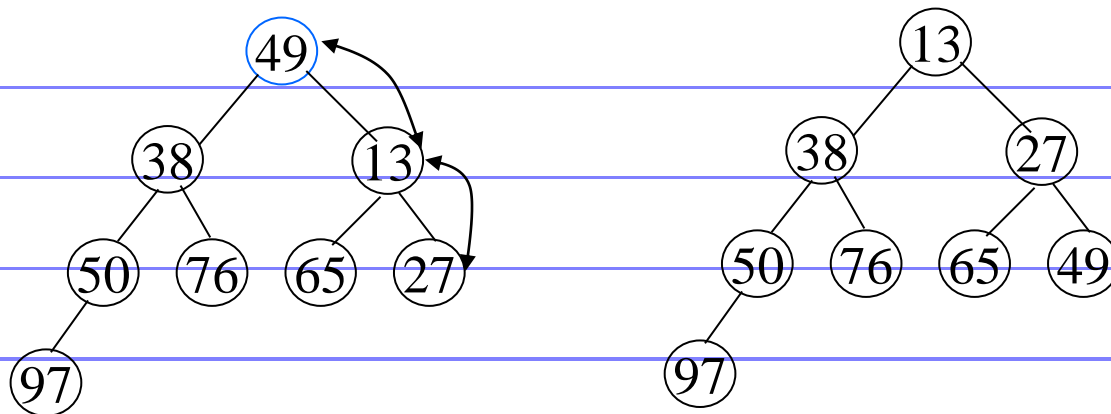
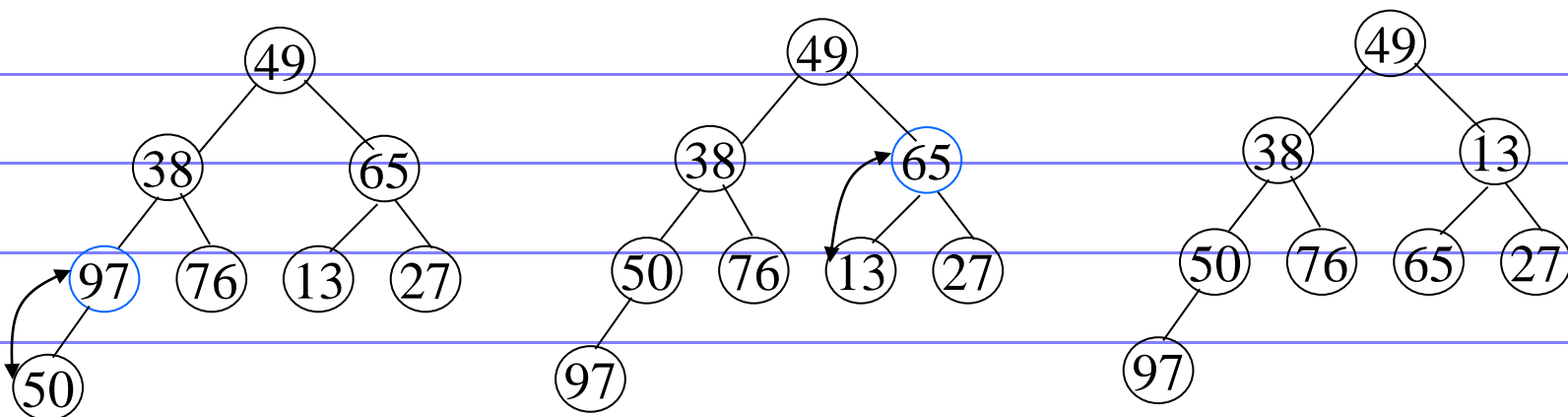
堆排序——堆的构造

□ 第一个问题解决方法

- 方法：从无序序列的第 $\lfloor n/2 \rfloor$ 个元素（即此无序序列对应的完全二叉树的最后一个非终端结点）起，至第一个元素止，进行反复筛选

堆排序例

例 含8个元素的无序序列 (49, 38, 65, 97, 76, 13, 27, 50)



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，调整H.r[s]的关键字，使H成为一个大顶堆

int j;

RedType rc;

rc = H.r[s];

for (j=2*s; j<=m; j*=2) { // 沿key较大的孩子结点向下筛选

if (j<m && H.r[j].key<H.r[j+1].key) ++j; // j为key较大的记录的下标

if (rc.key >= H.r[j].key) break; // rc应插入在位置s上

H.r[s] = H.r[j]; s = j;

}

H.r[s] = rc; // 插入

} // HeapAdjust

void HeapSort(HeapType &H) { // 对顺序表H进行堆排序

int i;

RedType temp;

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; i>1; --i) {

temp=H.r[i];

H.r[i]=H.r[1];

H.r[1]=temp; // 将堆顶记录和当前未经排序子序列Hr[1..i]中最后一个记录相互交换

HeapAdjust(H, 1, i-1); // 将H.r[1..i-1] 重新调整为大顶堆

}

} // HeapSort

堆排序算法评价

□ 时间复杂度：

■ 最坏情况下 $T(n)=O(n\log n)$

□ 空间复杂度：

■ $S(n)=O(1)$

9.4 归并排序

- 归并——将两个或两个以上的有序表组合成一个新的有序表，叫~
- 2-路归并排序：
 - 设初始序列含有 n 个记录，则可看成 n 个有序的子序列，每个子序列长度为1
 - 两两合并，得到 $\lfloor n/2 \rfloor$ 个长度为2或1的有序子序列
 - 再两两合并，.....如此重复，直至得到一个长度为 n 的有序序列为止

归并排序例

例

初始关键字: [49] [38] [65] [97] [76] [13] [27]

一趟归并后: [38 49] [65 97] [13 76] [27]

二趟归并后: [38 49 65 97] [13 27 76]

三趟归并后: [13 27 38 49 65 76 97]

归并排序算法评价

□ 时间复杂度:

■ $T(n) = O(n \log_2 n)$

□ 空间复杂度:

■ $S(n) = O(n)$

```
1 public static int[] mergeSort(int[] a, int low, int high){  
2     int mid = (low+high)/2;  
3     if(low<high){  
4         mergeSort(a, low, mid);  
5         mergeSort(a, mid+1, high);  
6         merge(a, low, mid, high);  
7     }  
8     return a;  
9 }
```

```
12 public static void merge(int[] a, int low, int mid, int high) {
13     int[] temp = new int[high-low+1];
14     int i= low;
15     int j = mid+1;
16     int k=0;
17     while(i<=mid && j<=high){
18         if(a[i]<a[j]){
19             temp[k++] = a[i++];
20         }else{
21             temp[k++] = a[j++];
22         }
23     }
24     while(i<=mid){
25         temp[k++] = a[i++];
26     }
27     while(j<=high){
28         temp[k++] = a[j++];
29     }
30     for(int x=0;x<temp.length;x++){
31         a[x+low] = temp[x];
32     }
33 }
```

9.5 基数排序

□ 多关键字排序

◇ 定义：

□ 基数排序：借助“分配”和“收集”对单逻辑关键字进行排序的一种方法

例 对52张扑克牌按以下次序排序：

$\clubsuit 2 < \clubsuit 3 < \dots < \clubsuit A < \diamond 2 < \diamond 3 < \dots < \diamond A <$

$\heartsuit 2 < \heartsuit 3 < \dots < \heartsuit A < \spadesuit 2 < \spadesuit 3 < \dots < \spadesuit A$

两个关键字：花色（ $\clubsuit < \diamond < \heartsuit < \spadesuit$ ）

面值（ $2 < 3 < \dots < A$ ）

并且“花色”地位高于“面值”

多关键字排序方法

- ❑ 最高位优先法(MSD): 先对最高位关键字 k_1 (如花色)排序, 将序列分成若干子序列, 每个子序列有相同的 k_1 值; 然后让每个子序列对次关键字 k_2 (如面值)排序, 又分成若干更小的子序列; 依次重复, 直至就每个子序列对最低位关键字 k_d 排序; 最后将所有子序列依次连接在一起成为一个有序序列
- ❑ 最低位优先法(LSD): 从最低位关键字 k_d 起进行排序, 然后再对高一位的關鍵字排序,依次重复, 直至对最高位关键字 k_1 排序后, 便成为一个有序序列
- ❑ MSD与LSD不同特点
 - 按MSD排序, 必须将序列逐层分割成若干子序列, 然后对各子序列分别排序
 - 按LSD排序, 不必分成子序列, 对每个关键字都是整个序列参加排序; 并且可不通过关键字比较, 而通过若干次分配与收集实现排序

链式基数排序

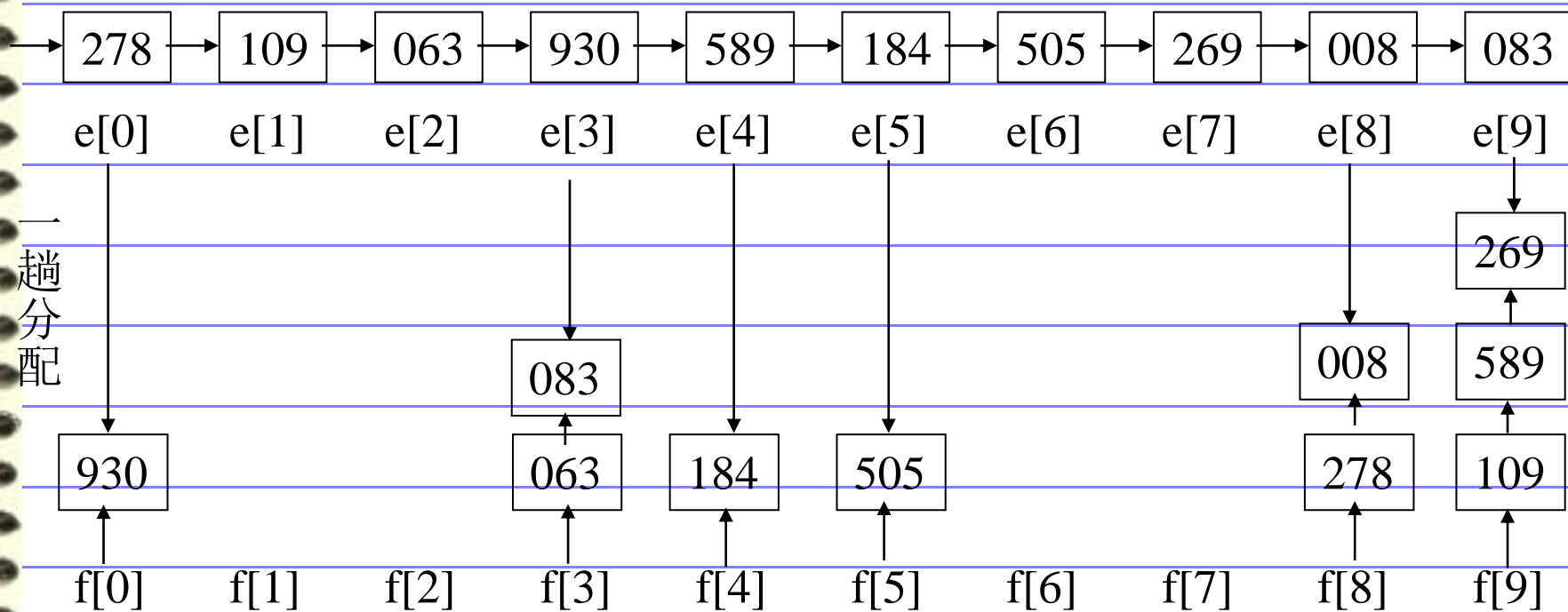
□ 链式基数排序：用链表作存储结构的基数排序

□ 步骤

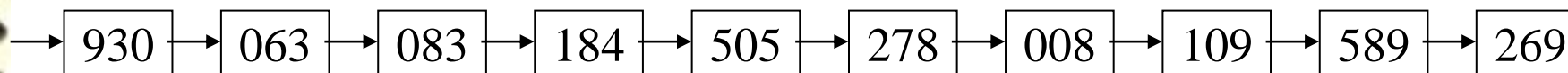
- 设置10个队列， $f[i]$ 和 $e[i]$ 分别为第 i 个队列的头指针和尾指针
- 第一趟分配对最低位关键字（个位）进行，改变记录的指针值，将链表中记录分配至10个链队列中，每个队列记录的关键字的个位相同
- 第一趟收集是改变所有非空队列的队尾记录的指针域，令其指向下一个非空队列的队头记录，重新将10个队列链成一个链表
- 重复上述两步，进行第二趟、第三趟分配和收集，分别对十位、百位进行，最后得到一个有序序列

例

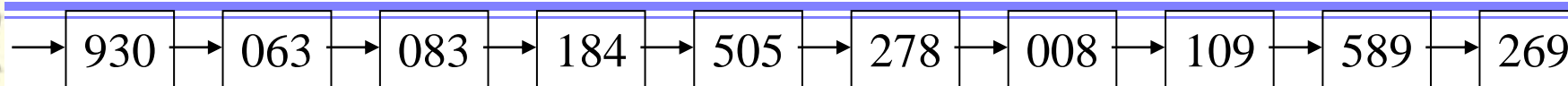
初始状态:



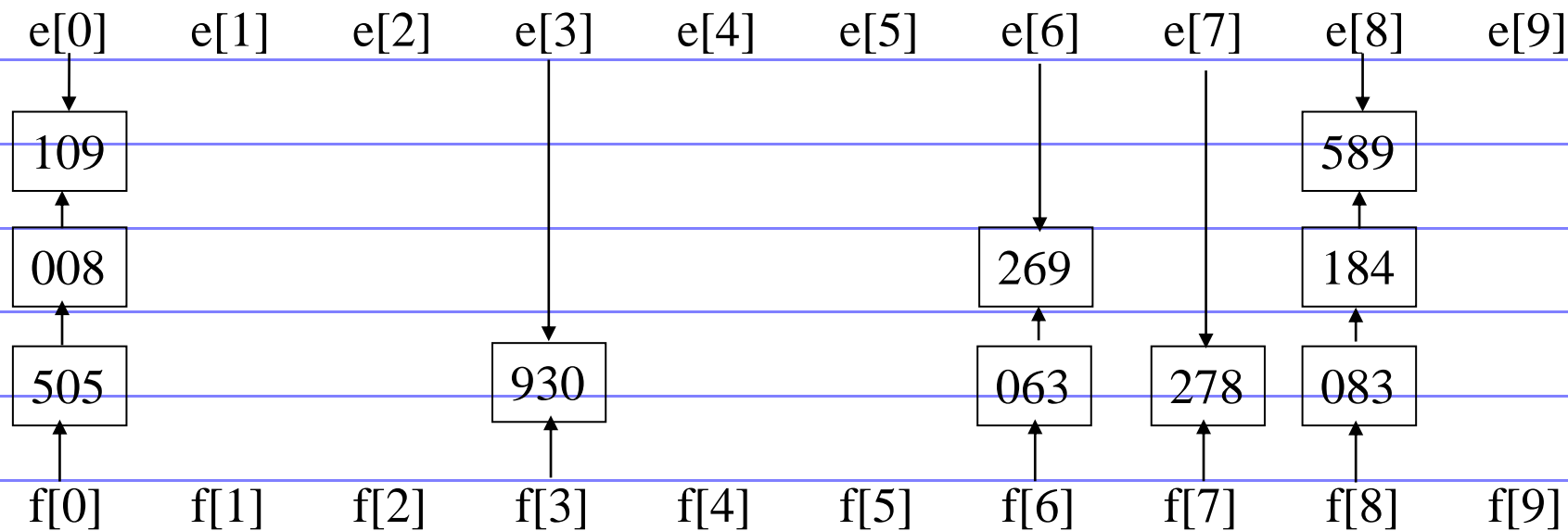
一趟收集:



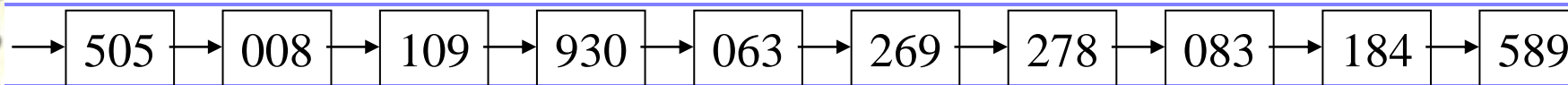
一趟收集:



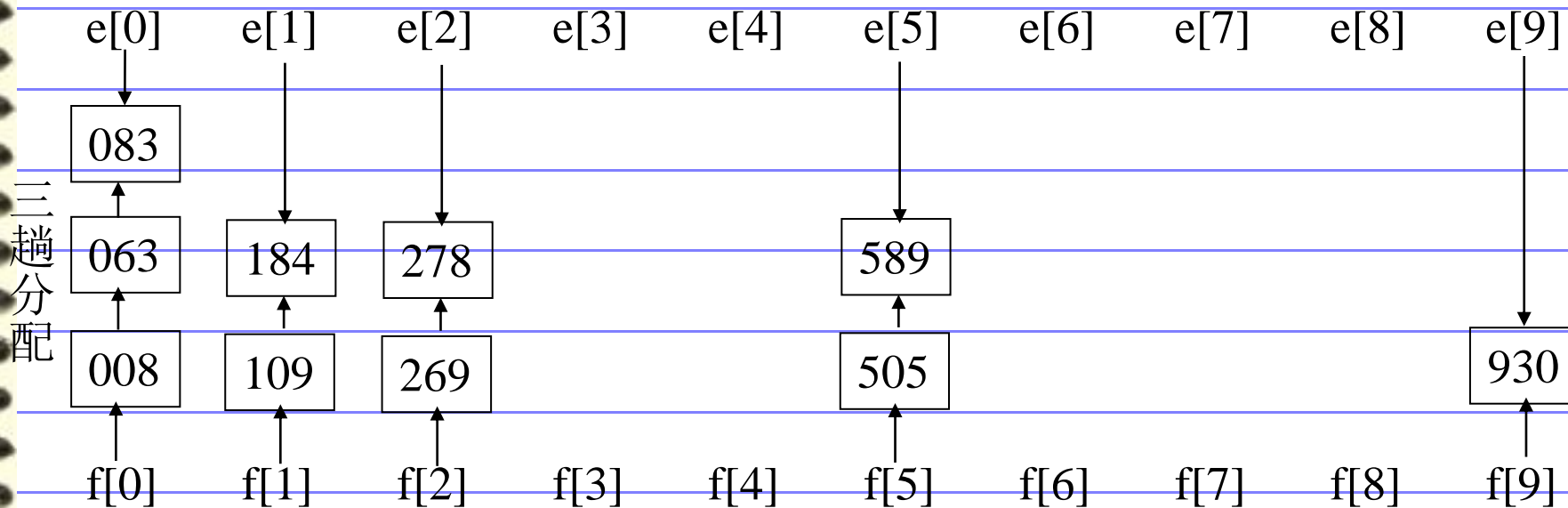
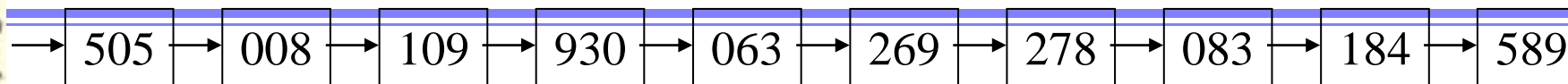
一趟分配



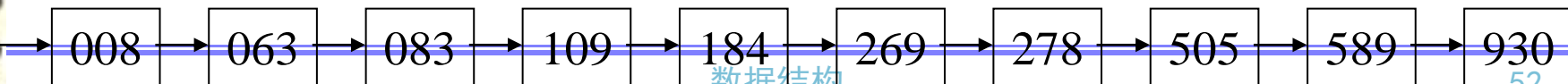
二趟收集:



二趟收集:



三趟收集:



链式基数排序算法评价

□ 时间复杂度：

- 分配： $T(n)=O(n)$

- 收集： $T(n)=O(rd)$

$T(n)=O(d(n+rd))$

其中： n ——记录数

d ——关键字数

rd ——关键字取值范围

□ 空间复杂度：

- $S(n)=2rd$ 个队列指针+ n 个指针域空间

各种内部排序方法的比较讨论

排序方法	平均时间	最坏情况	辅助存储
简单排序	$O(n^2)$	$O(n^2)$	$O(1)$
快速排序	$O(n \log n)$	$O(n^2)$	$O(\log n)$
堆排序	$O(n \log n)$	$O(n \log n)$	$O(1)$
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n)$
基数排序	$O(d(n + rd))$	$O(d(n + rd))$	$O(rd)$

各种内部排序方法的比较讨论

- (1)从平均时间性能而言，快速排序最佳，其所需时间最省，但快速排序在最坏情况下的时间性能不如堆排序和归并排序。而后两者相比较的结果是，在 n 较大时，归并排序所需时间较堆排序省，但它所需的辅助存储量最多。
- (2)上表中的“简单排序”包括除希尔排序之外的所有插入排序，起泡排序和简单选择排序，其中以直接插入排序为最简单，当序列中的记录“基本有序”或 n 值较小时，它是最佳的排序方法，因此常将它和其它的排序方法结合在一起使用。
- (3)基数排序的时间复杂度也可写成 $O(d \times n)$ 。因此它最适用于 n 值很大而关键字较小的序列。若关键字也很大而序列中大多数记录的“最高位关键字”均不同，则亦可先按“最高位关键字”不同将序列分成若干“小”的子序列，而后进行直接插入排序。
- (4)从方法的稳定性来比较，基数排序是稳定的内排序方法，所有时间复杂度为 $O(n^2)$ 的简单排序法也是稳定的，然而，快速排序、堆排序和希尔排序等时间性能较好的排序方法都是不稳定的。