DS—第十章

内部排序– Sorting in RAM

第九章回顾

- 相关概念术语
- 静态查找表

  1、顺序查找

  2、有序表查找

  3、索引顺序表查找
- 动态查找表

  1、二叉排序树

  2、平衡二叉树

  3、B-树、B+树
- 哈希表

  1、相关概念

  2、哈希函数

  3、处理冲突的方法

  4、哈希表的查找过程

- 排序概念、插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 各种排序方法比较

- 排序概念、插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 各种排序方法比较

排序相关概念

- **排序**：是计算机程序设计中的一种重要操作，它的功能是将一个数据元素（或记录）的任意序列，重新排列成一个按关键字有序的序列。

  注意：我们排序的是记录而非关键字，关键字只是排序的依据。但是我们为了讲课的方便，举例时可能只涉及到仅包含关键字的记录。

- **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。而外部排序指的是当排序记录的数量很大，以致内存一次不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。

- **排序的稳定性**：如果两记录Ri与Rj的关键字相同，在排序前Ri在Rj的前面，排序之后， Ri依然在Rj之前，我们称这种排序方法是稳定的，反之，是不稳定的。

排序表定义

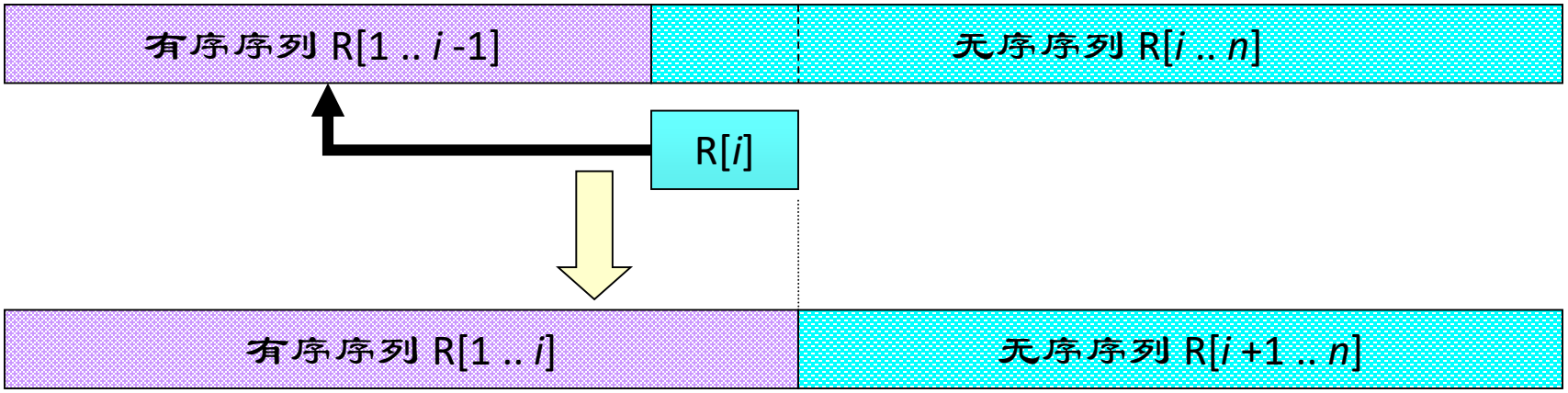
```
#define  MAXSIZE  20
typedef  int  KeyType;
typedef  struct
{
    KeyType  key;
    InfoType otherinfo;
}RedType;
Typedef struct
{
    RedType  r[MAXSIZE+1];
    int      length;
}SqList;
```

学习方法

- 排序思想及排序过程
- 排序算法的代码描述
- 排序算法的效率分析

直接插入排序

一趟直接插入排序的基本思想：



实现“一趟插入排序”可分三步进行：

1．在 R[1 .. *i* -1] 中查找 R[*i*] 的插入位置，

R[1 .. *j*].key ≤ R[*i*].key < R[*j*+1 .. *i* -1].key；

2．将 R[*j*+1 .. *i* -1] 中的所有记录均后移一个位置；

3．将 R[i] 插入（复制）到 R[j+1] 的位置上。

```
void InsertSort ( SqList &L ) {
  // 对顺序表 L 作直接插入排序。
  for ( i = 2; i <= L.length; ++ i )
    if (L.r[i].key < L.r[i -1].key) {
      L.r[0] = L.r[i];          // 复制为监视哨
      L.r[i] = L.r[i -1];
      for ( j = i - 2; L.r[0].key < L.r[ j ].key;  - - j )
        L.r[ j + 1] = L.r[ j ];       // 记录后移
      L.r[ j + 1] = L.r[0];       // 插入到正确位置
    }
} // Ins
```

初始状态

i =2

i =3

i =4

i =5

i =6

排序过程：先将序列中第 1 个记录看成是一个有序子序列，

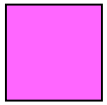
然后从第 2 个记录开始，逐个进行插入，直至整个序列有序。

算法分析

最好的情况：待排序记录按关键字从小到大排列（正序）

比较次数：$\sum_{i=2}^{n} 1 = n-1$　　**移动次数**：0

最坏的情况：待排序记录按关键字从大到小排列（逆序）



5 4 3 2 1

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

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

一般情况：待排序记录是随机的，取平均值。

比较次数和移动次数均约为：$\frac{n^2}{4}$

直接插入排序是稳定排序

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

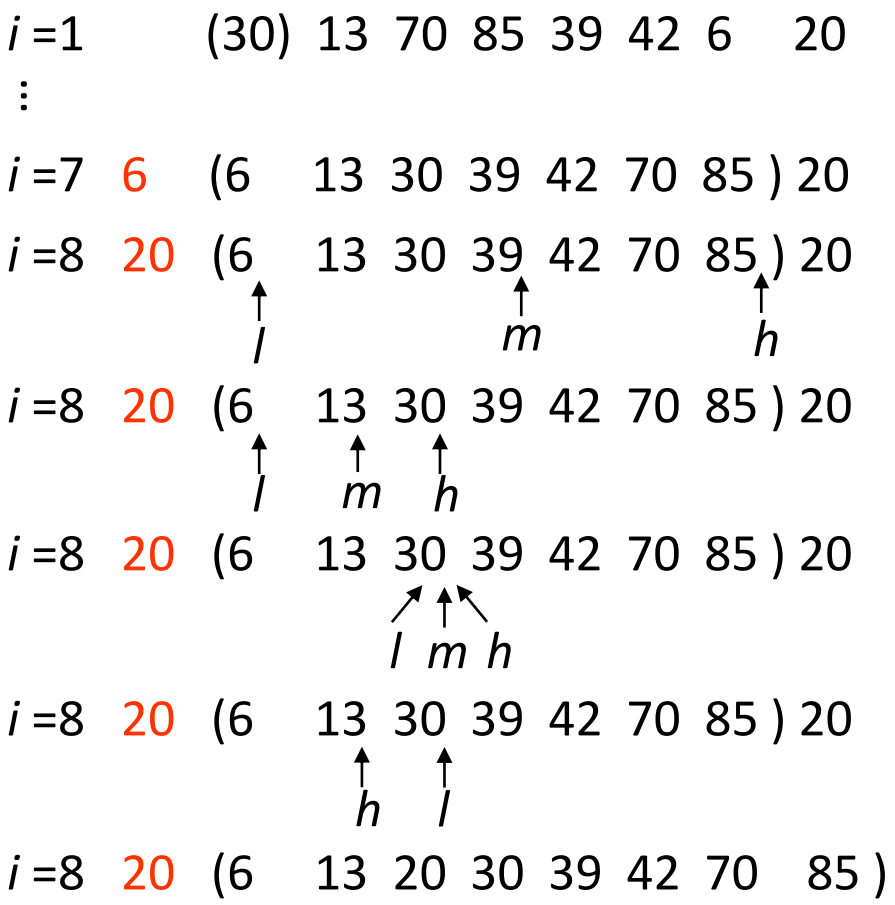
用折半查找方法确定插入位置的排序。

时间复杂度： $T(n)=O(n^2)$

仅减少了比较次数，
移动次数不变。

空间复杂度： $S(n)=O(1)$

折半插入排序是稳定排序



希尔排序（缩小增量排序）

基本思想：对待排序列先作“宏观”调整，再作“微观”调整。

排序过程：先取一个正整数 $d_1 < n$，把所有相隔 d_1 的记录放在一组内，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。其中 d_i 称为增量。

例：

第一趟分组，设 $d_1 = 5$

49	38	65	97	76	13	27	49	55	04

第二趟分组，设 $d_2 = 3$

13	27	49	55	04	49	38	65	97	76

第三趟分组，设 $d_3 = 1$

13	04	49	38	27	49	55	65	97	76

第三趟希尔排序

04	13	27	38	49	49	55	65	76	97

```
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);j-=dk)
                L.r[j+dk]=L.r[j];
            L.r[j+dk]=L.r[0];
        }
}

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]);
}
```

希尔排序分析

- 分组不是简单的“逐段分割”，而是将相隔某个增量的记录组成一个子序列。
- 增量序列取法

  希尔最早提出的选法是 $d_1 = \lfloor n/2 \rfloor$，$d_{i+1} = \lfloor d_i/2 \rfloor$。

  克努特 (Knuth) 提出的选法是 $d_{i+1} = \lfloor (d_i-1)/3 \rfloor$。

  还有许多其他取法。

  如何选择增量序列以产生最好的排序效果，至今仍没有从数学上得到解决。

  1)、没有除 1 以外的公因子；

  2)、最后一个增量值必须为 1。
- 希尔排序的时间复杂度约为：$O(n^{1.3})$

- 排序概念、插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 各种排序方法比较

起泡排序

初始关键字	第一趟排序后	第二趟排序后	第三趟排序后	第四趟排序后	第五趟排序后	第六趟排序后
49	38	38	38	38	13	13
38	49	49	49	13	27	27
65	65	65	13	27	38	38
97	76	13	27	49	49	
76	13	27	49	49		
13	27	49	65			
27	49	76				
49	97					

```
void  BubbleSort(SqList &L)
{
    change=true;
    for(i=1;i<L.length&&change;i++)
    {
        change=false;
        for(j=1;j<=L.length-i;j++)
            if(a[j]>a[j+1])
            {
                a[j]<->a[j+1];
                change=true;
            }
    }
}
```

性能分析

显然，如果初始序列为“正序”序列，则只需进行一趟排序，在排序过程中进行 $n-1$ 次关键字间的比较，且不移动记录；反之，如果初始序列为“逆序”序列，则需进行 $n-1$ 趟排序，需进行 $\sum_{i=n}^{2}(i-1)=n(n-1)/2$ 次比较，并作等数量级的记录移动。因此，总的时间复杂度为 $O(n^2)$。

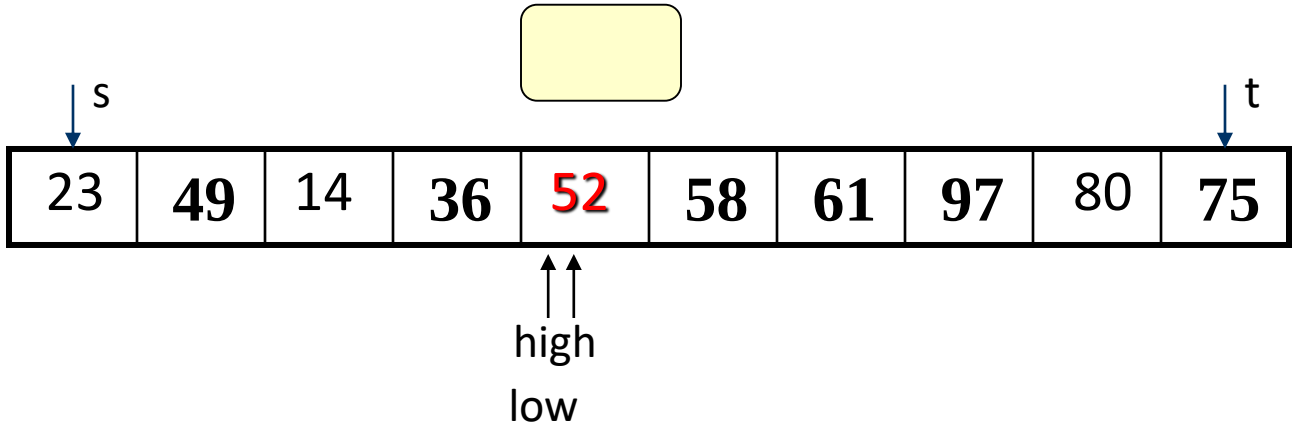
快速排序

基本思想：任选一个记录，以它的关键字作为“枢轴”，凡关键字小于枢轴的记录均移至枢轴之前，凡关键字大于枢轴的记录均移至枢轴之后。

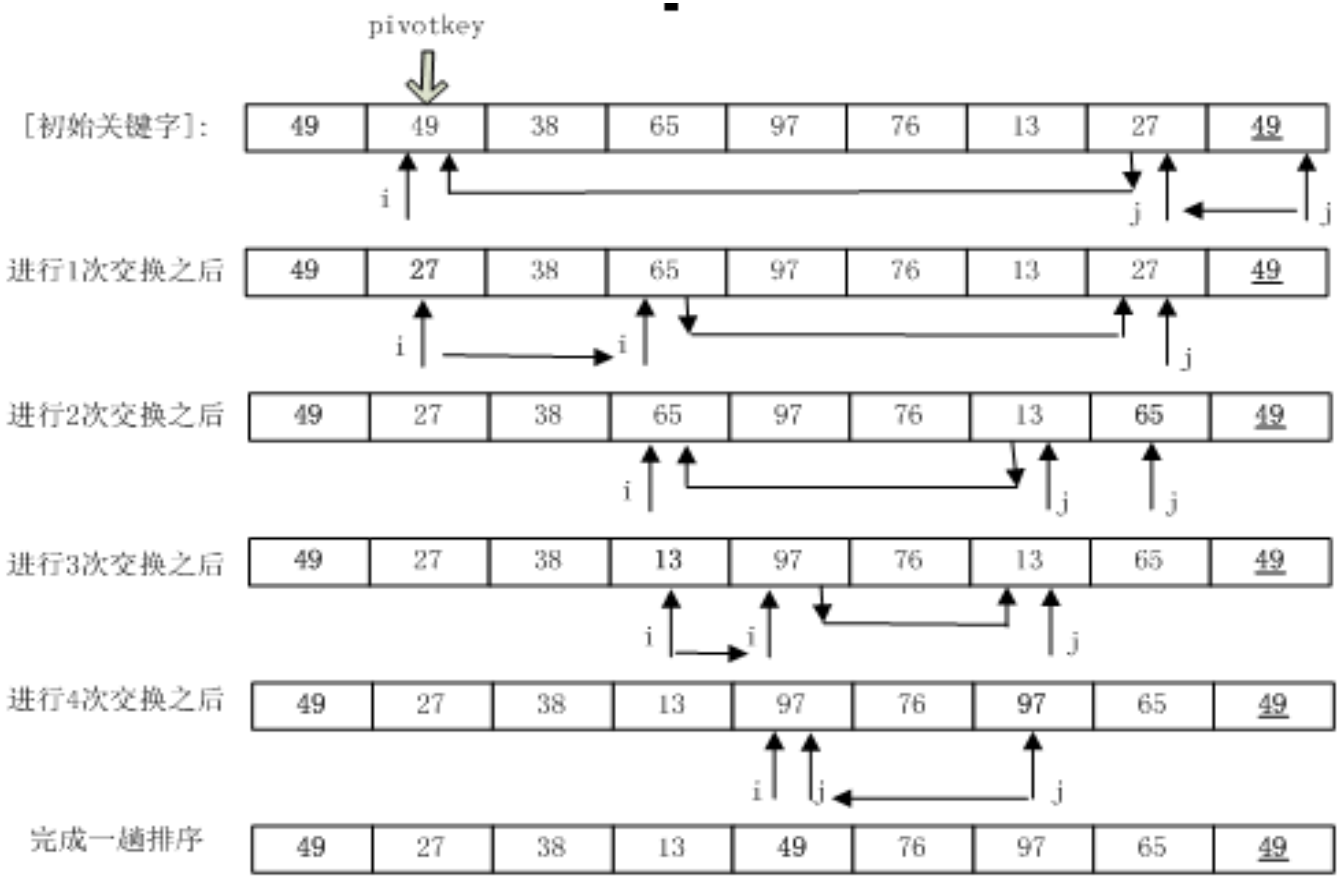
一般取第一个记录

附设两个指针 low 和 high，从 high 所指位置起向前搜索找到第一个关键字小于枢轴的关键字的记录与枢轴记录交换，然后从 low 所指位置起向后搜索找到第一个关键字大于枢轴的关键字的记录与枢轴记录交换，重复这两步直至 low = high 为止。

例：



设 R[s]=52 为枢轴。



(a)

初始状态	{49	38	65	97	76	97	65	49}
一次划分后	{27	38	13}	49	{76	97	65	49}
分别进行快排	{13} 结束	27	{38} 结束		{49	65}	76	{97} 结束
					49	{65} 结束		
有序序列	{13	27	38	49	49	65	76	97}

(b)

```
int  Partition(SqList &L,int low,int high)
{//以low位置上的元素为标准进行一次划分。
    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)
                low++;
        L.r[high]=L.r[low];
}
L.r[low] = L.r[0];
return low;
}
```

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

void QuickSort(SqList &L)
{对顺序表 L 进行快速排序。
    QSort(L,1,L.length);
}
```

效率分析

假设 T(n)表示对 n 个记录 L.r[1..n]进行快速排序所需的时间，则由算法 QuickSort 可见：$T(n)=T_{pass}(n)+T(k-1)+T(n-k)$。其中 $T_{pass}(n)$为对 n 个记录进行一趟快速排序（划分）所需要的时间，从算法 Partition 可以看出它和记录数 n 成正比，可以用 cn 表示；T(k-1)和 T(k-1)分别为对 L.r[1..k-1]和 L.r[k+1..n]中记录进行快速所需的时间。假设待排序列中的记录是随机排列的，则一趟排序之后，k 取 1 至 n 之间任何一值的概率相同，则快速排序所需时间的平均值为

$$T_{avg}(n)=cn+\frac{1}{n}\sum_{k=1}^{n}[T_{avg}(k-1)+T_{avg}(n-k)]\cdots\cdots\cdots①$$

由于当 k 从 1 增加到 n 时，$T_{avg}(k-1)$与 $T_{avg}(n-k)$是对称变化的，所以公式①可以变形为

$$T_{avg}(n)=cn+\frac{2}{n}\sum_{i=0}^{n-1}T_{avg}(i)\cdots\cdots\cdots②$$

由公式②得 $\sum_{i=0}^{n-1}T_{avg}(i)=\frac{n}{2}(T_{avg}(n)-cn)\cdots\cdots\cdots③$

又$\sum_{i=0}^{n-1} T_{avg}(i) = T_{avg}(n-1) + \sum_{0}^{n-2} T_{avg}(i)$（只是将公式展开一项而已）

$$= T_{avg}(n-1) + \frac{n-1}{2}(T_{avg}(n-1) - c(n-1)) \text{.....④}$$

由③、④联立得$\frac{n}{2}(T_{avg}(n) - cn) = T_{avg}(n-1) + \frac{n-1}{2}(T_{avg}(n-1) - c(n-1))$

即：$T_{avg}(n) - cn = \frac{2}{n}[T_{avg}(n-1) + \frac{n-1}{2}T_{avg}(n-1) - \frac{n-1}{2}cn + \frac{n-1}{2}c]$

所以：$T_{avg}(n) = \frac{2}{n}[\frac{n+1}{2}T_{avg}(n-1) - \frac{n-1}{2}cn + \frac{n-1}{2}c] + cn$

$$= \frac{n+1}{n}T_{avg}(n-1) - \frac{n-1}{n}cn + \frac{n-1}{n}c + cn$$

$$= \frac{n+1}{n}T_{avg}(n-1) + \frac{2n-1}{n}c$$

$$< \frac{n+1}{2}T_{avg}(1) + 2(n+1)(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1})c$$

$$< (\frac{b}{2} + 2c)(n+1)\ln(n+1) \qquad （\mathbf{n \geq 2}）$$

通常，快速排序被认为是，在所有同数量级（O(nlogn)）的排序方法中，其平均性能最好。可能蜕变为起泡排序，这样时间复杂度就为 $O(n^2)$。空间复杂度 O(n)~O(logn)。它是一种不稳定的排序方法。

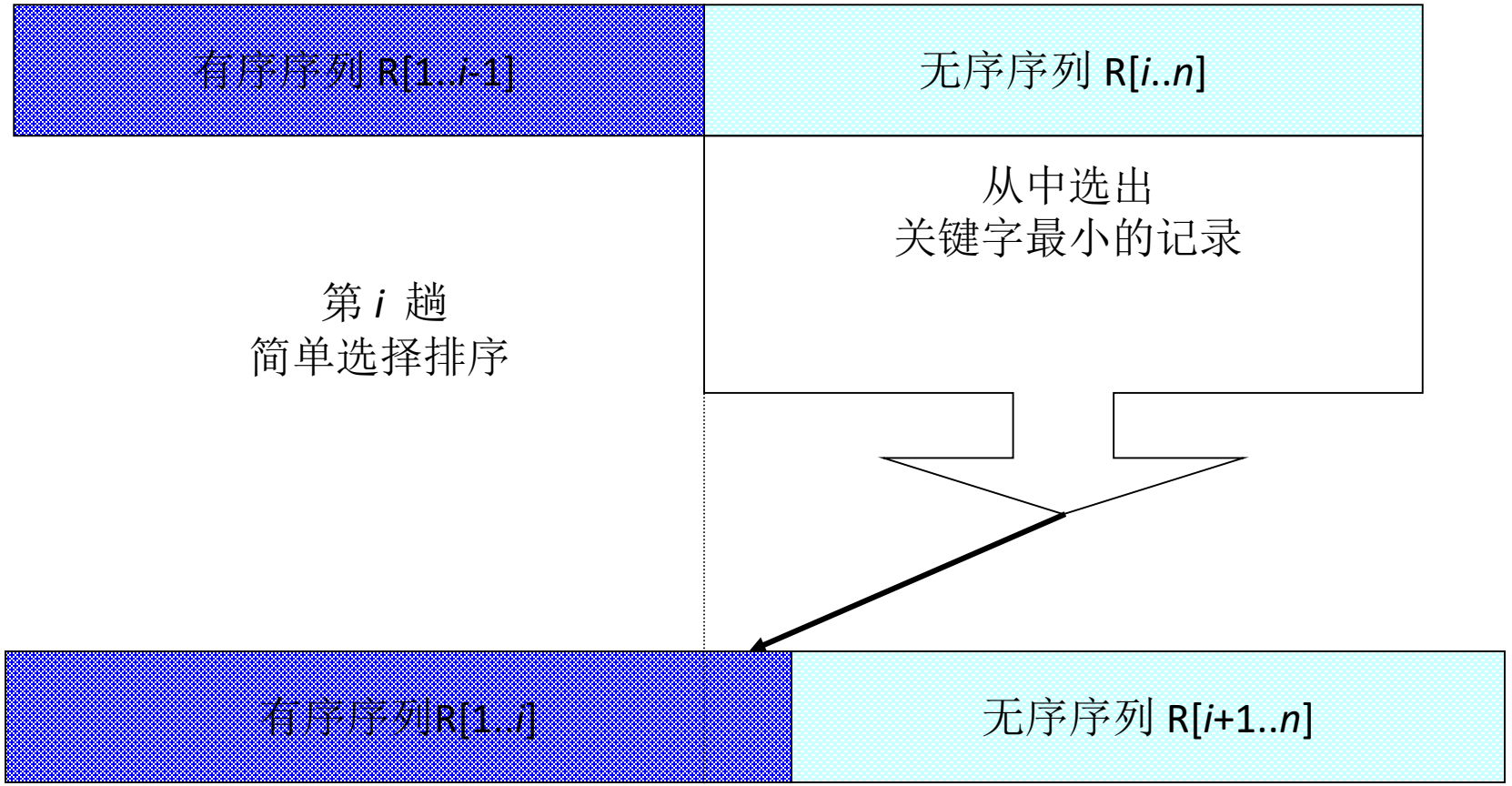
- 排序概念、插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 各种排序方法比较

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。

假设排序过程中，待排记录序列的状态为：



例：

									比较次数
							k ↓		
i =1	初始：	13	38	65	97	76	49	27]	n - 1
								↑ j	
i =2	一趟：	13	[38	65	97	76	49	27]	n - 2
i =3	二趟：	13	27	[65	97	76	49	38]	
i =4	三趟：	13	27	38	[97	76	49	65]	
i =5	四趟：	13	27	38	49	[76	97	65]	
i =6	五趟：	13	27	38	49	65	[97	76]	n - 6
排序结束：	六趟：	13	27	38	49	65	76	97	

比较次数：

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

时间复杂度：$O(n^2)$

空间复杂度：$O(1)$

移动次数：　正序：最小值为 0；　最大值为 3(n-1) 。

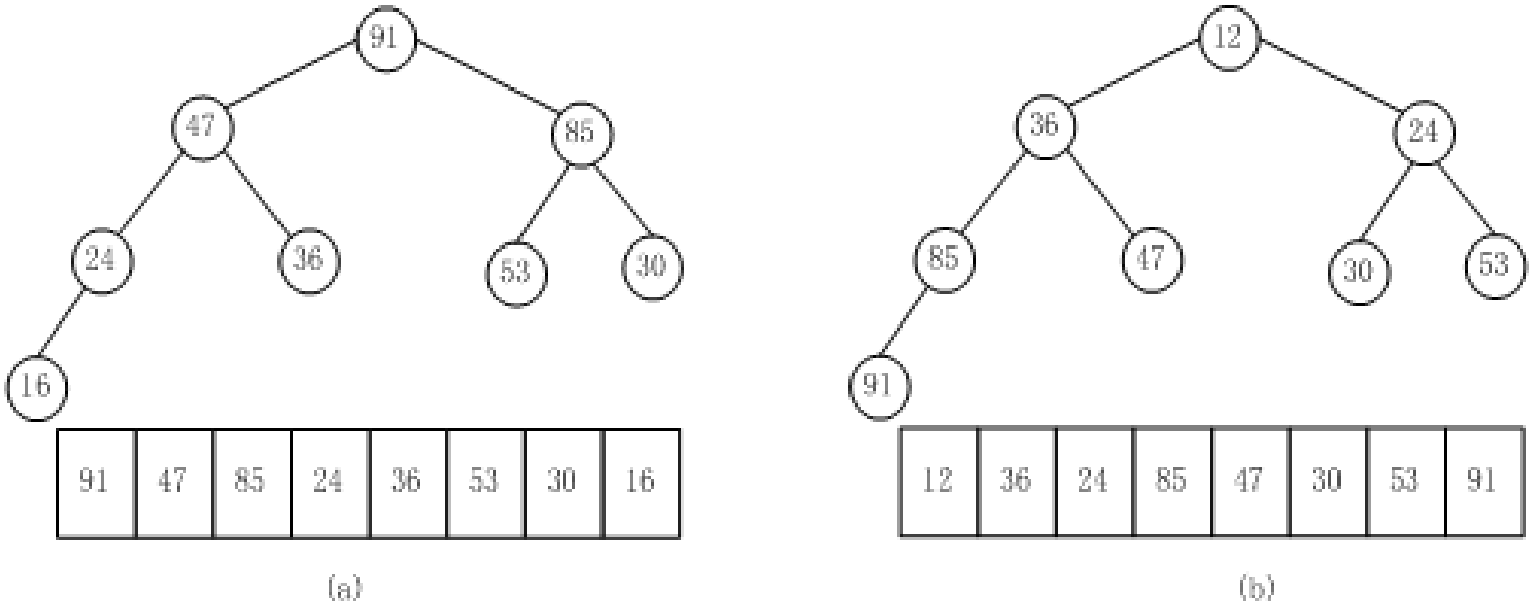
不稳定

前 $n-1$ 个为正序，第 n 个记录的关键字最小。

堆排序

n个元素的序列{k1，k2，…，kn}当且仅当满足下列关系是，称之为堆。

$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} 或 \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \left(i = 1,2,\ldots,\left\lfloor \frac{n}{2} \right\rfloor \right)$$



堆示例（a、大根堆　b、小根堆）

堆排序（续）

堆排序：将无序序列建成一个堆，得到关键字最小（大）的记录；输出堆顶的最小（大）值后，将剩余的 *n*-1 个元素重又建成一个堆，则可得到 *n* 个元素的次小值；如此重复执行，直到堆中只有一个记录为止，每个记录出堆的顺序就是一个有序序列，这个过程叫**堆排序**。

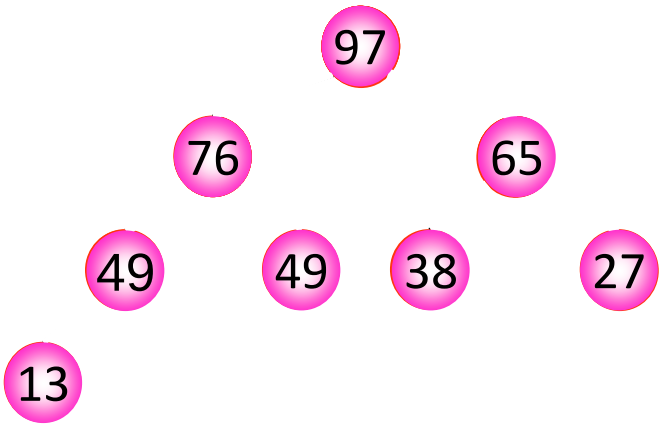
堆排序需解决的两个问题：

1、如何由一个无序序列建成一个堆？

2、在输出堆顶元素后，如何将剩余元素调整为一个新的堆？

堆排序——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”。

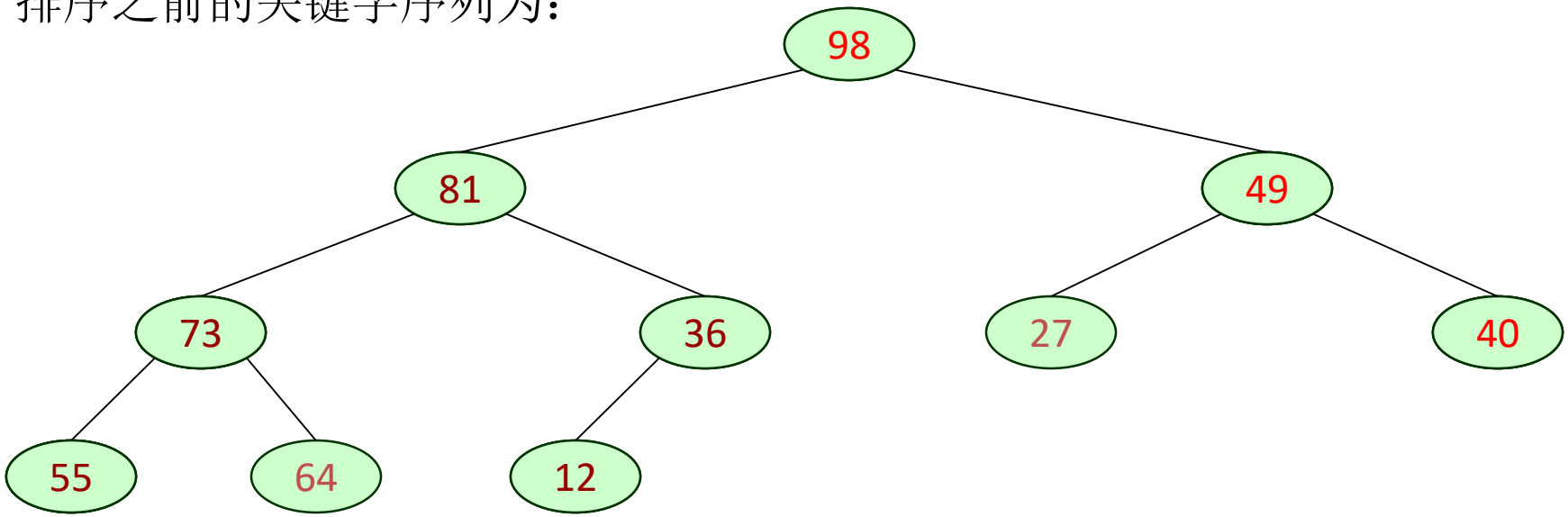


对深度为 k 的堆，“筛选”所需进行的关键字比较的次数至多为 $2(k-1)$。

堆排序——建堆

从无序序列的第⌊n/2⌋个元素（即无序序列对应的完全二叉树的最后一个内部结点）起，至第一个元素止，进行反复筛选。

例: 排序之前的关键字序列为：



现在，左/右子树都已经调整为堆，最后只要调整根结点，使整个二叉树是个“堆”即可。

建堆是一个从下往上进行“筛选”的过程。

```
void  HeapAdjust(SqList &H, int s, int m)
{
    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(SqList &H)
{
    for(i=H.length/2;i>0;i--)
        HeapAdjust(H,i,H.length);
    for(i=H.length;i>1;i--)
    {   H.r[1]<->H.r[i];
        HeapAdjust(H,1,i-1);
    }
}
```

效率分析

1. 对深度为 k 的堆，“筛选”所需进行的关键字比较的次数至多为 $2(k-1)$；
2. 对 n 个关键字，建成深度为 $h(=\lfloor \log_2 n \rfloor+1)$ 的堆，所需进行的关键字比较的次数至多 $4n$；
3. 调整“堆顶” $n-1$ 次，总共进行的关键字比较的次数不超过

$$2\ (\lfloor \log_2(n-1) \rfloor+\lfloor \log_2(n-2) \rfloor+\ \ldots+\log_2 2) < 2n(\lfloor \log_2 n \rfloor)$$

因此，堆排序的时间复杂度为 $O(n\log n)$，与简单选择排序 $O(n^2)$ 相比时间效率提高了很多。

空间复杂度：$S(n) = O(1)$

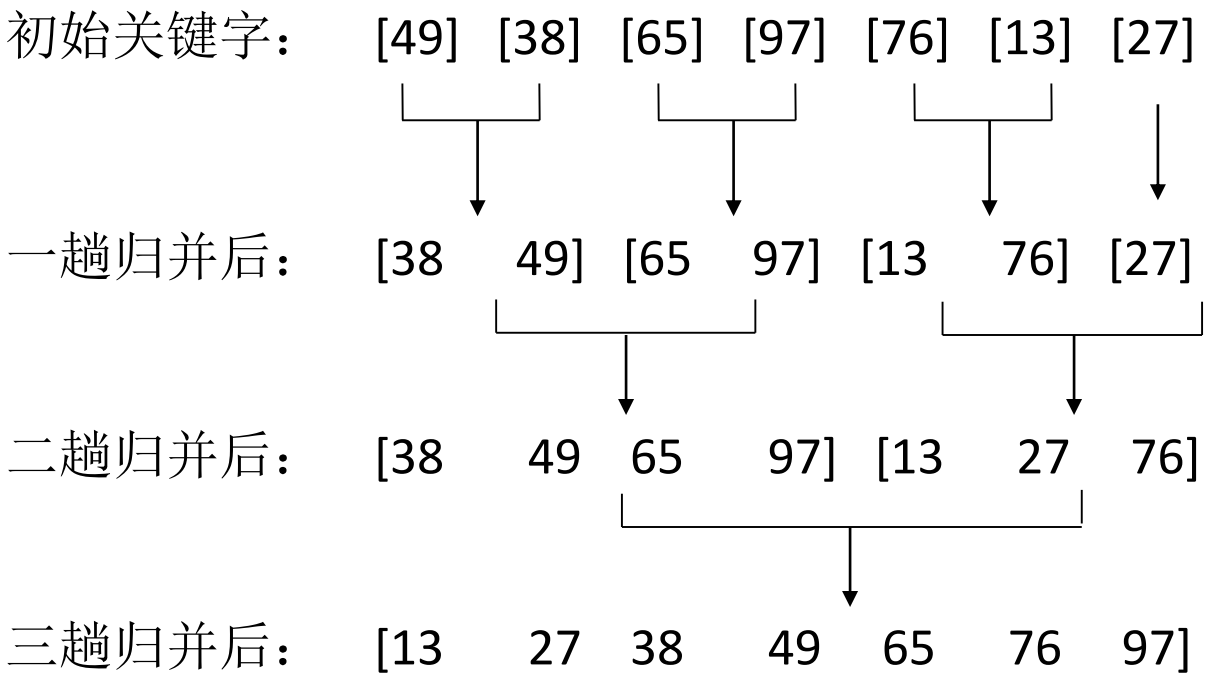
堆排序是一种速度快且省空间的排序方法。　　不稳定。

- 排序概念、插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 各种排序方法比较

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。

在内部排序中，通常采用的是**2-路归并排序**。即：将两个位置相邻的记录有序子序列归并为一个记录有序的序列。



看成是 n 个有序的子序列（长度为 1），然后两两归并。

得到$\lceil n/2 \rceil$个长度为 2 或 1 的有序子序列。

每趟归并的时间复杂度为$O(n)$，共需进行$\lceil \log_2 n \rceil$趟。

空间复杂度为：$O(n)$。 **时间复杂度为**：$O(n\log n)$。 **稳定。**

- 排序概念、插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 各种排序方法比较

多关键字排序

假设有 n 个记录的序列{R_1，R_2，…，R_n}且每个记录 R_i 中含有 d 个关键字（$K_i^0, K_i^1, \ldots, K_i^{d-1}$），则称序列{$R_1$，$R_2$，…，$R_n$}对关键字（$K_i^0, K_i^1, \ldots, K_i^{d-1}$）有序是指：对于序列中任意两个记录 R_i 和 R_j（$1 \le i < j \le n$）都满足下列关系：

$$(K_i^0, K_i^1, \ldots, K_i^{d-1}) < (K_j^0, K_j^1, \ldots, K_j^{d-1})$$

其中 K^0 称为最主位关键字，K^{d-1} 称为最次位关键字。

- **最高位优先法（MSD）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。
- **最低位优先法（LSD）不必分成若干子序列，只要按照低位优先排就行。(排序方法有限制)**

链式基数排序

在计算机上实现基数排序时，为减少所需辅助存储空间，应采用链表作存储结构，即链式基数排序，具体作法为：

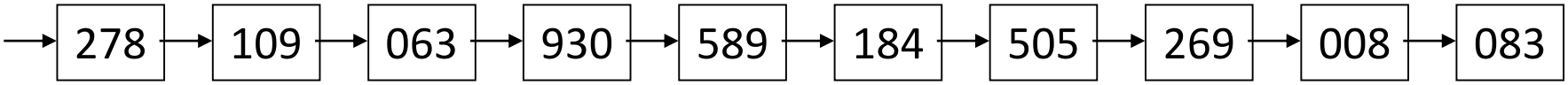
1、以静态链表存储待排记录，并令表头指针指向第一个记录；

2、“分配”时，按当前“关键字位”所取值，将记录分配到不同的“链队列”中，每个队列中记录的“关键字位”相同；

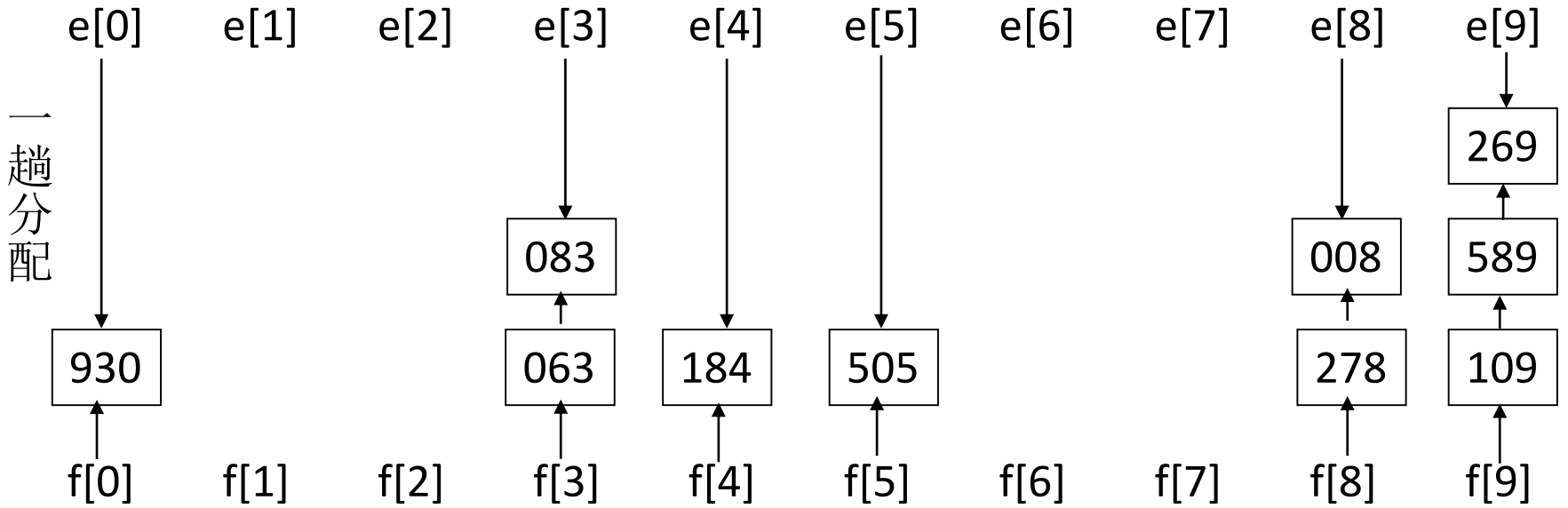
3、“收集”时，按当前关键字位取值从小到大将各队列首尾相链成一个链表；

4、对每个关键字位均重复 2 和 3 两步。

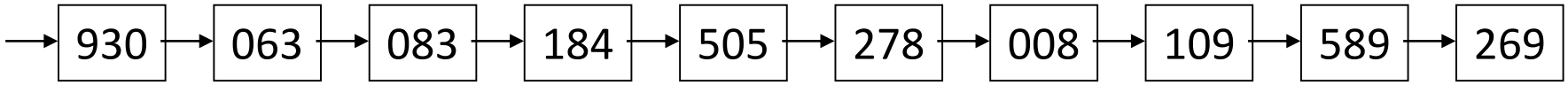
例：　以静态链表存储待排记录，并令表头指针指向第一个记录。



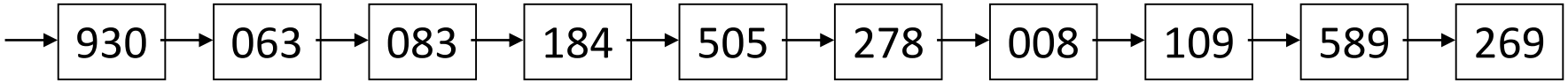
“分配”时，按当前“关键字位”所取值，将记录分配到不同的“链队列”中，每个队列中记录的“关键字位”相同。



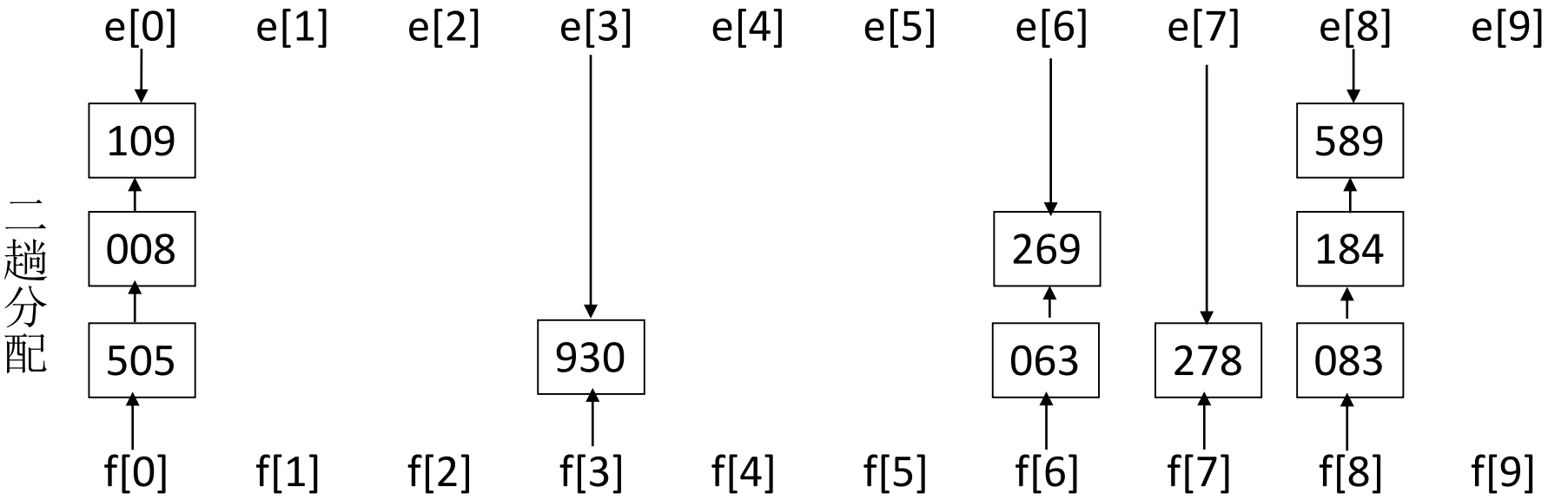
一趟收集：按当前关键字位取值从小到大将各队列首尾相链成一个链表；



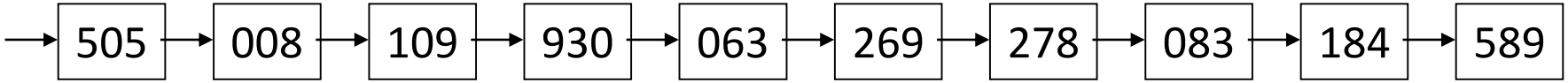
一趟收集：



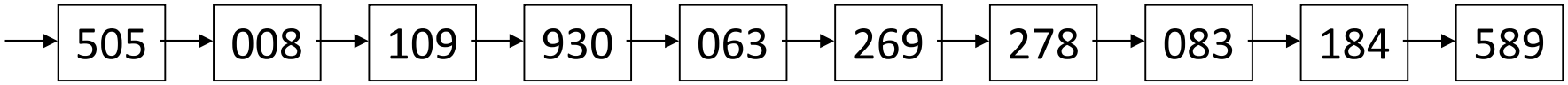
二趟分配



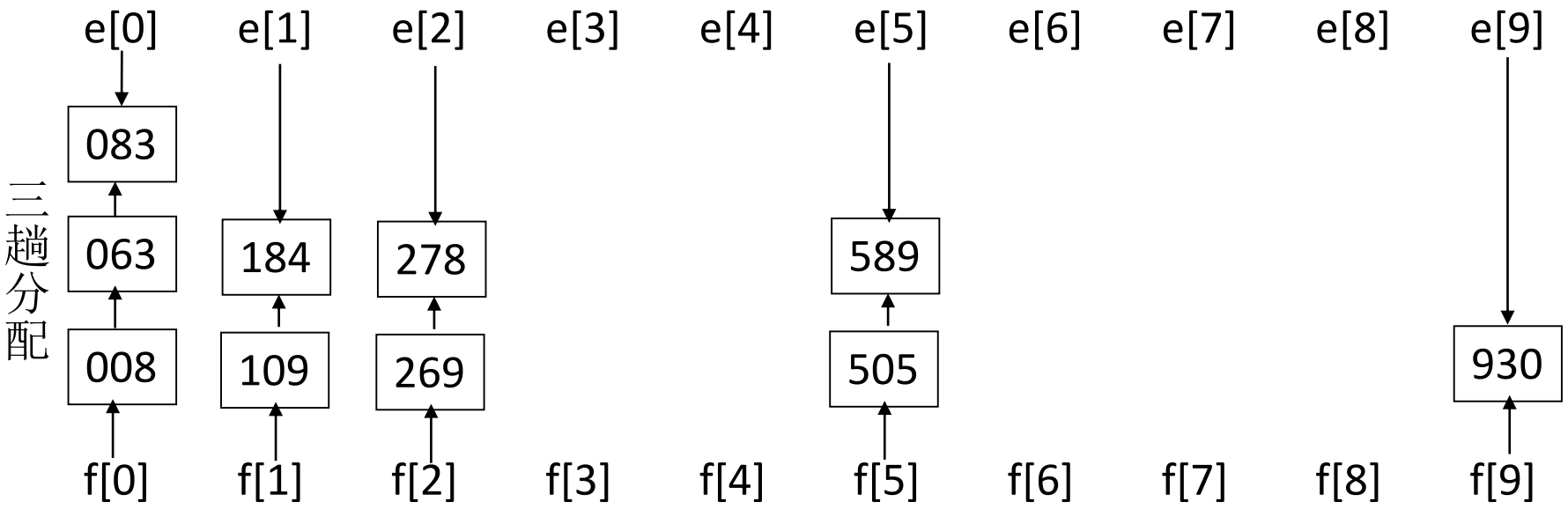
二趟收集：



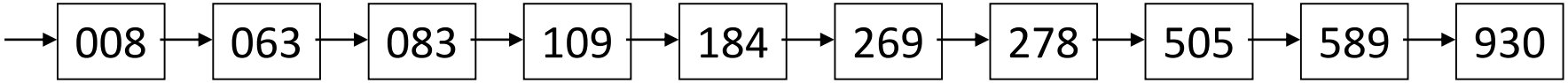
二趟收集：



三趟分配



三趟收集：



效率分析

时间复杂度：

假设：n —— 记录数

d —— 关键字位数

rd —— 关键字取值范围

（如十进制为10）

分配（每趟）：$T(n)=O(n)$

收集（每趟）：$T(n)=O(rd)$

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

空间复杂度：$S(n)=2rd$ 个队列指针 + n 个指针域空间

- 排序概念、插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 各种排序方法比较

排序方法比较

排序方法		平均时间复杂度	最坏时间复杂度	空间复杂度	稳定性
简单排序	直接插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
	折半插入排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
	起泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
	简单选择排序	$O(n^2)$	$O(n^2)$	O(1)	不稳定
高级排序	快速排序	O(nlogn)	$O(n^2)$	O(logn)	不稳定
	堆排序	O(nlogn)	O(nlogn)	O(1)	不稳定
	希尔排序	$O(n^{1+k})(0<k<1)$	$O(n^{1+k})(0<k<1)$	O(1)	不稳定
	归并排序	O(nlogn)	O(nlogn)	O(n)	稳定
	基数排序	O(d(n+rd))	O(d(n+rd))	O(rd)	稳定

小结

- 排序基本概念
- 插入排序
- 交换排序
- 选择排序
- 归并排序(算法的伪代码不要求)
- 基数排序(算法的伪代码不要求)

Thank You !