

数据结构与算法

第六章-1 排序算法总览

裴文杰

计算机科学与技术学院 教授



第六章 排序

6.1 概述

6.2 插入排序

6.3 起泡（冒泡）排序

6.4 选择排序

6.5 基数排序

6.6 内部排序方法的比较



6.1 概述

- **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。
- **排序的目的**之一是方便数据查找。



6.1 概述

□ 排序方法的稳定和不稳定

在排序过程中，有若干记录的关键字相等，即 $K_i = K_j (1 \leq i \leq n, 1 \leq j \leq n, i \neq j)$ ，在排序前后，含相等关键字的记录的相对位置保持不变，即排序前 R_i 在 R_j 之前，排序后 R_i 仍在 R_j 之前，称这种排序方法是稳定的；

反之，若可能使排序后的序列中 R_i 在 R_j 之后，称所用排序方法是不稳定的。



6.1 概述

□ 内部排序和外部排序

在排序过程中，只使用计算机的内存存放待排序记录，称这种排序为“内部排序”。

排序期间文件的全部记录不能同时存放在计算机的内存中，要借助计算机的外存才能完成排序，称之为“外部排序”。

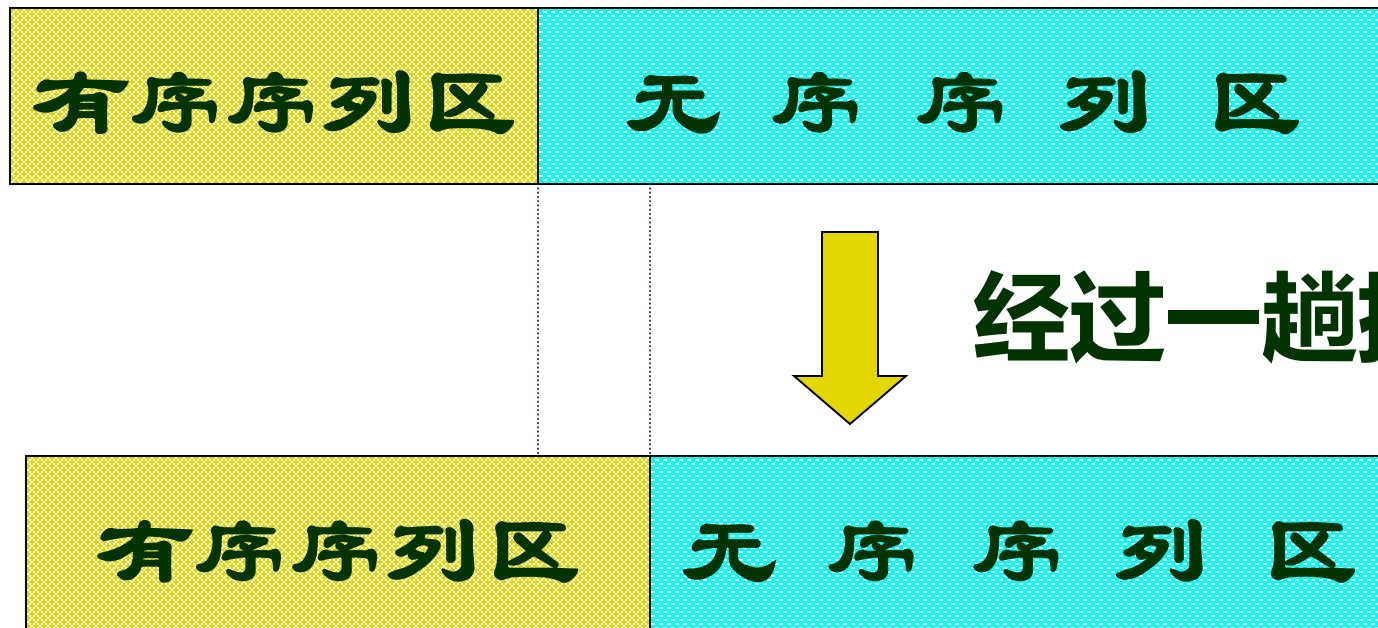
内外存之间的数据交换次数就成为影响外部排序速度的主要因素。



6.1 概述

□ 内部排序

内部排序的过程是一个逐步扩大记录的有序序列长度的过程。





6.1 概述

□ 内部排序

**基于不同的“扩大”有序序列长度的方法，
内部排序方法大致可分下列几种类型：**

插入类

交换类

选择类

归并类

其它方法



6.1 概述

□ 内部排序

1. 插入类

将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度。



6.1 概述

□ 内部排序

2. 交换类

通过“**交换**”无序序列中的记录
从而得到其中**关键字最小或最大**的记
录，并将它**加入**到有序子序列中，以
此方法增加记录的有序子序列的长度。



6.1 概述

□ 内部排序

3. 选择类

从记录的无序子序列中 **“选择”** 关键字**最小或最大**的记录，并将它**加入到有序子序列中**，以此方法**增加记录的有序子序列的长度**。



6.1 概述

□ 内部排序

4.归并类

通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度。

5.其它方法



6.1 概述

□ 内部排序

- **插入类** 将无序子序列中的一个或几个记录“插入”到有序序列中，从而增加记录的有序子序列的长度
- **选择类** 从记录的无序子序列中“选择”关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度
- **交换类** 通过“交换”无序序列中的记录从而得到其中关键字最小或最大的记录，并将它加入到有序子序列中，以此方法增加记录的有序子序列的长度
- **归并类** 通过“归并”两个或两个以上的记录有序子序列，逐步增加记录有序序列的长度
- **其它方法**



6.1 概述

□ 内部排序

排序算法的性能:

- **基本操作。**内排序在排序过程中的基本操作:
 - **比较:**关键字之间的比较;
 - **移动:**记录从一个位置移动到另一个位置。
- **辅助存储空间。**
 - 辅助存储空间是指在数据规模一定的条件下,除了存放待排序记录占用的存储空间之外,执行算法所需要的其他额外存储空间。
- **算法本身的复杂度。**



6.1 概述

□ 存储结构

```
#define MAXSIZE 1000 // 待排顺序表最大长度
typedef int KeyType; // 关键字类型为整数类型
typedef struct {
    KeyType key;           // 关键字项
    InfoType otherinfo;    // 其它数据项
} RcdType;               // 记录类型
typedef struct {
    RcdType r[MAXSIZE+1]; // r[0]闲置
    int length;           // 顺序表长度
} SqList;                // 顺序表类型
```

第六章 排序



6.1 概述

6.2 插入排序

6.3 气泡排序

6.4 选择排序

6.5 基数排序

6.6 内部排序方法的比较



6.2 插入排序

6.2.1 直接插入排序

6.2.2 折半插入排序

6.2.3 表插入排序

6.2.4 希尔排序



6.2.1 直接插入排序

不同的具体实现方法导致不同的算法描述

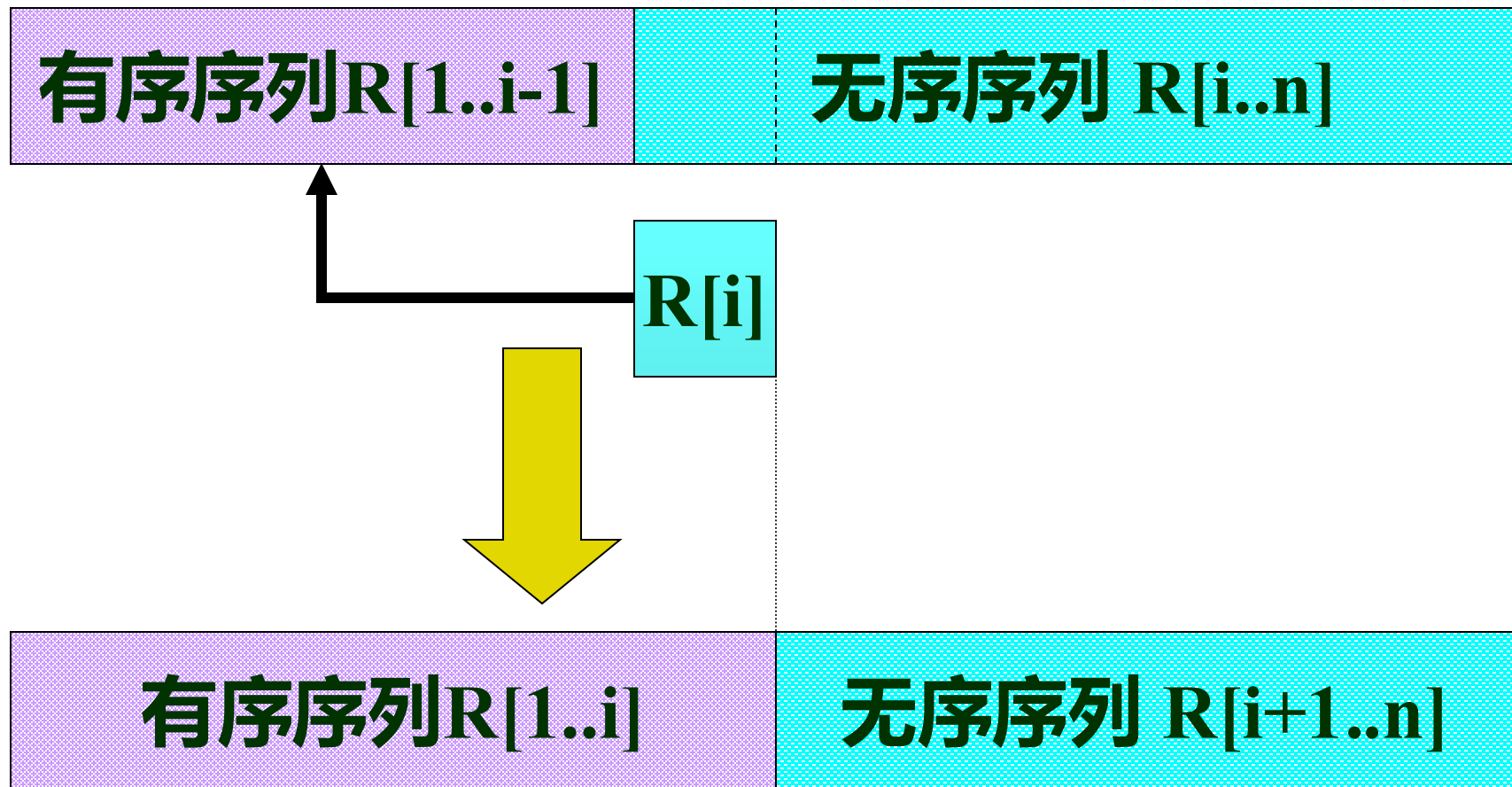
- 直接插入排序（基于顺序查找）
- 折半插入排序（基于折半查找）
- 表插入排序（基于链表存储）
- 希尔排序（基于逐趟缩小增量）



6.2.1 直接插入排序

□ 插入排序的思想

一趟直接插入排序的基本思想：





6.2.1 直接插入排序

□ 插入排序的思想

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,
 $R[1..j].key \leq R[i].key < R[j+1..i-1].key$;
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。



6.2.1 直接插入排序

□ 直接插入排序算法概述

(1)将序列中的第1个记录看成是一个有序的子序列;

(2)从第2个记录起逐个进行插入,直至整个序列变成按关键字有序序列为止;

整个排序过程需要进行比较、后移记录、插入适当位置。从第二个记录到第 n 个记录共需 $n-1$ 趟。



6.2.1 直接插入排序

例1 直接插入排序的过程。

初始关键字: (49) 38 65 97 76 13 27 49

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

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

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

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

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

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

i=8: (49) (13 27 38 49 49 65 76 97)



6.2.1 直接插入排序

利用 “顺序查找” 实现
“在 $R[1..i-1]$ 中查找 $R[i]$ 的插入位置”

算法的实现要点：

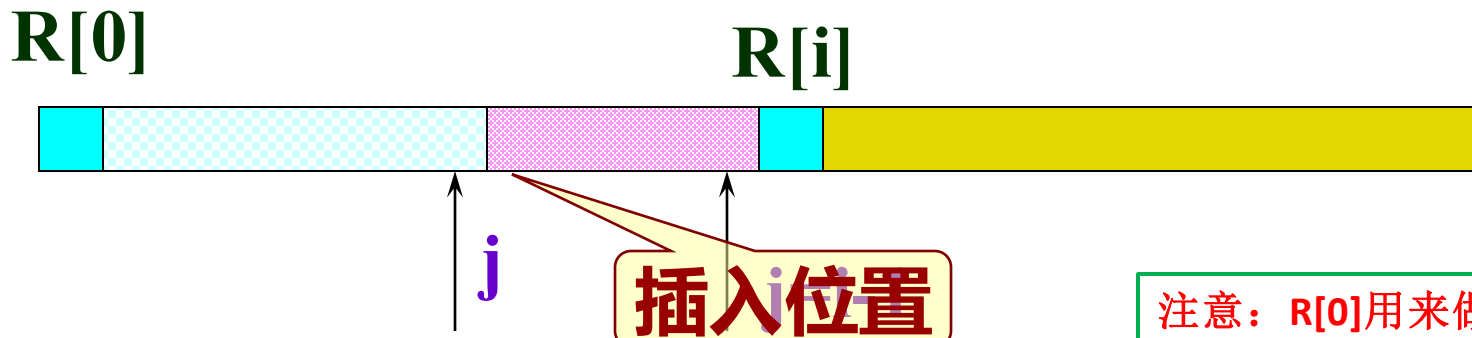


6.2.1 直接插入排序

□ 直接插入排序算法要点

从 $R[i-1]$ 起向前进行顺序查找,

监视哨设置在 $R[0]$;



$R[0] = R[i];$ // 设置“哨兵”

for ($j=i-1$; $R[0].key < R[j].key$; $--j$);

// 从后往前找

循环结束表明 $R[i]$ 的插入位置为 $j+1$



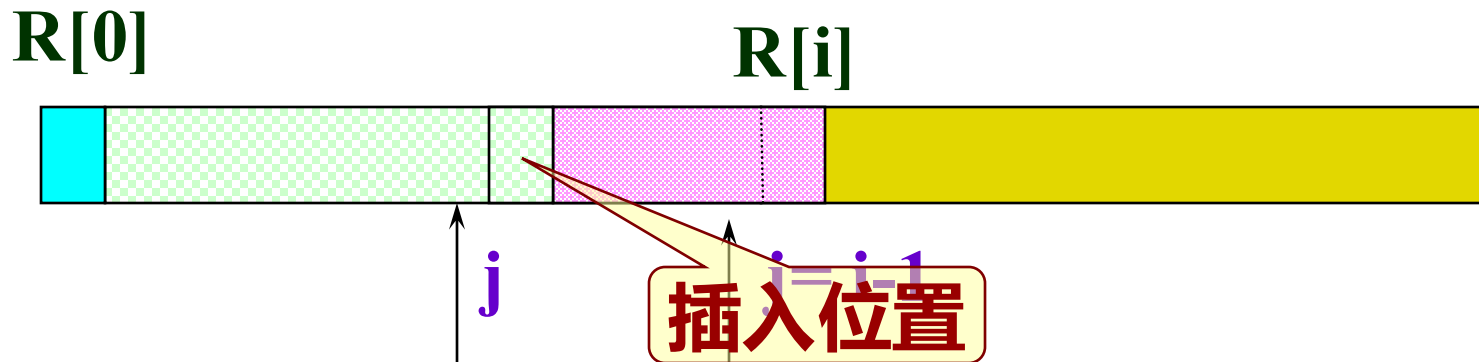
6.2.1 直接插入排序

□ 直接插入排序算法要点

对于在查找过程中找到的那些关键字不小于 $R[i].key$ 的记录，在查找的同时实现记录向后移动；

for ($j=i-1$; $R[0].key < R[j].key$; $--j$)

$R[j+1] = R[j];$



上述循环结束后可以直接进行“插入”



6.2.1 直接插入排序

□ 直接插入排序算法

```
void InsertionSort ( 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  
            for ( j=i-1; 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  
        }  
} // InsertSort
```



6.2.1 直接插入排序

□ 直接插入排序算法分析

实现内部排序的基本操作有两个：

- (1) “**比较**” 序列中两个关键字的大小；
- (2) “**移动**” 记录。



6.2.1 直接插入排序

□ 直接插入排序算法分析

最好的情况（关键字在记录序列中顺序有序）：

“比较” 的次数：

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

“移动” 的次数：

$$0$$

最坏的情况（关键字在记录序列中逆序有序）：

“比较” 的次数：

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

“移动” 的次数：

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



6.2.1 直接插入排序

□ 直接插入排序算法分析

(1) 稳定性

直接插入排序是稳定的排序方法。

(2) 算法效率

a. 时间复杂度

最好情况: 比较 $O(n)$, 移动 $O(1)$;

最坏情况: 比较 $O(n^2)$, 移动 $O(n^2)$;

平均 $O(n^2)$

b. 空间复杂度

$O(1)$ 。



6.2.2 折半插入排序

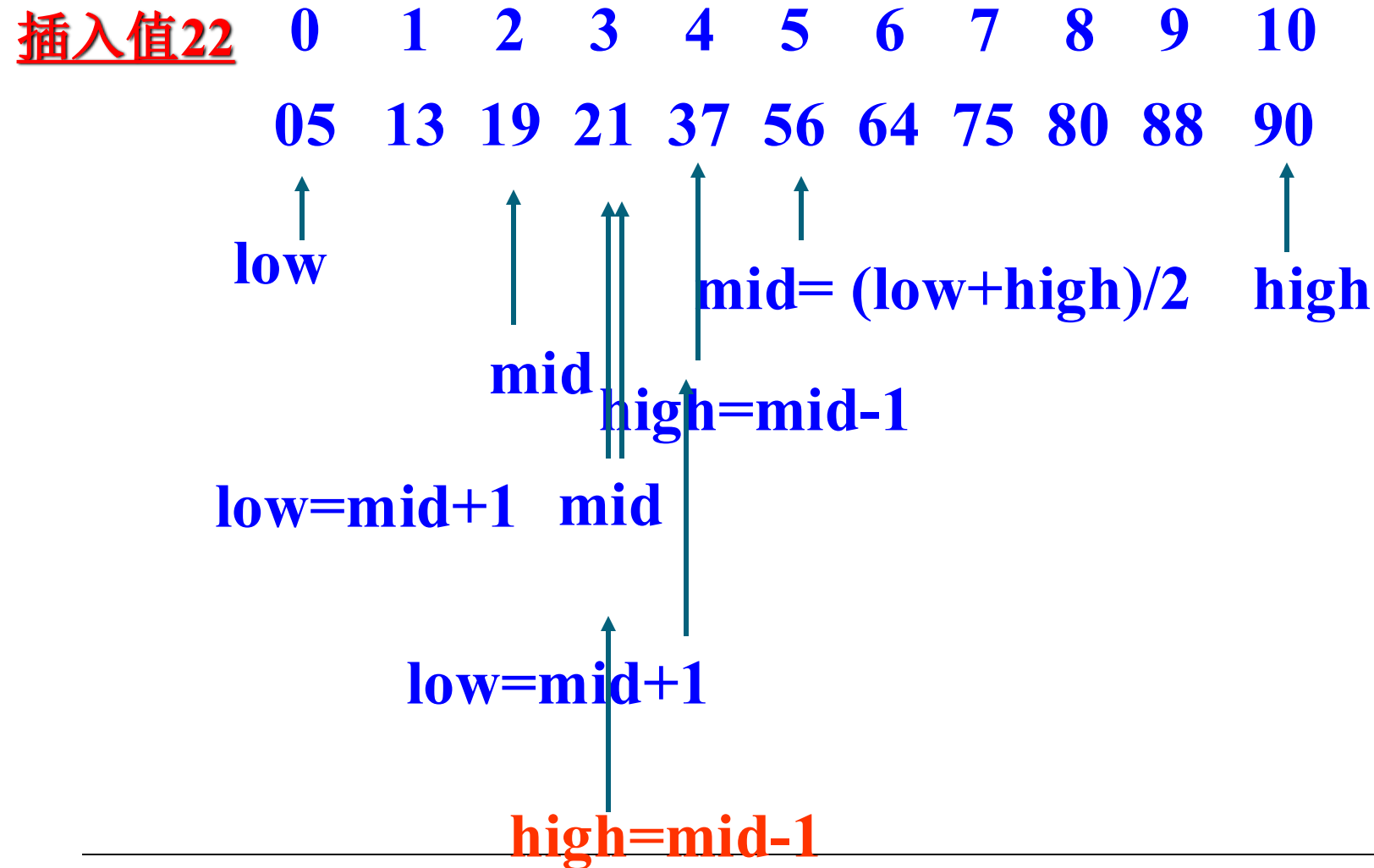
□ 折半插入排序思想

- (1)在直接插入排序中， $r[1..i-1]$ 是一个按关键字有序的有序序列；
- (2)可以利用折半查找实现“在 $r[1..i-1]$ 中查找 $r[i]$ 的插入位置”；
- (3)称这种排序为折半插入排序。



6.2.2 折半插入排序

□ 折半插入排序的演示过程





6.2.2 折半插入排序

□ 折半插入排序算法

```
void BiInsertionSort ( SqList &L )
```

```
{
```

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

```
        L.r[0] = L.r[i];           // 将 L.r[i] 暂存到 L.r[0]
```

```
        在 L.r[1..i-1]中折半查找插入位置; //见下一页
```

```
        for ( j=i-1; j>=high+1; --j )//插入位置为high+1
```

```
            L.r[j+1] = L.r[j];           // 记录后移
```

```
            L.r[high+1] = L.r[0];        // 插入
```

```
        }                               // for
```

```
    }                                   // BiInsertSort
```



6.2.2 折半插入排序

□ 折半插入排序算法

在 $L.r[1..i-1]$ 中折半查找插入位置

```
low = 1;  high = i-1;           //查找区间
while (low <= high) {
    m = (low+high)/2;           // 折半
    if (L.r[0].key < L.r[m].key)
        high = m-1;             // 插入点在低半区
    else low = m+1;             // 插入点在高半区
}
```




6.2.2 折半插入排序

不同的具体实现方法导致不同的算法描述

- 直接插入排序—（基于顺序查找）
 - 折半插入排序（基于折半查找）
 - 表插入排序（基于链表存储） → 改进排序的存储结构
 - 希尔排序（基于逐趟缩小增量）
- 移动相同数量的关键字

折半查找可以有效减少比较的次数，但是不能减少移动的次数。如何减小移动次数？



6.2.3 表插入排序

□ 表插入排序的思想

所谓表插入排序，是利用静态链表的形式，分两步完成排序。

一、对一个有序的循环链表，插入一新的元素，修改每个节点的后继指针的指向，使顺着这个指针的指向，元素是有序的。在这个过程中，我们不移动或交换元素，只是修改指针的指向。

二、顺着指针的指向调整元素的位置，使其在链表中真正做到物理有序。



6.2.3 表插入排序

□ 表插入排序的思想

思路：

- 1、**构建一新的结构体类型**，使其封装了值域和指针域。
并增加一结点，**当做头结点**，**为循环终止创造条件**，头结点值域存贮的值应不小于原序列中的最大值。
- 2、**初始化静态链表**：使第一个结点和头结点构成循环的链表。由于链表中只有一个元素，那当然是有序的。
- 3、把后续的结点**依次插入**到该循环链表中，调整各结点的指针指向，使其沿着指针方向是有序的。



6.2.3 表插入排序

□ 表插入排序的思想

小结：

- (1)目的是为了减少在排序过程中进行的“移动”记录的操作；
- (2)利用静态链表进行排序，排序中只修改指针指向；
- (3)在排序完成之后，一次性地调整各个记录相互之间的位置。



6.2.3 表插入排序

头指针

初始化静态链表：使第一个结点和头结点构成循环的链表

初始
状态

h

key域
next域
0是头指针

0	1	2	3	4	5	6	7	8
M	49	38	65	27	76	13	97	49*
1	0	-	-	-	-	-	-	-

0	1	2	3	4	5	6	7	8
M	49	38	65	27	76	13	97	49*
2	0	1	-	-	-	-	-	-

0	1	2	3	4	5	6	7	8
M	49	38	65	27	76	13	97	49*
2	3	1	0	-	-	-	-	-

0	1	2	3	4	5	6	7	8
M	49	38	65	27	76	13	97	49*
4	3	1	0	2	-	-	-	-



6.2.3 表插入排序

□ 表插入排序的存储结构

```
#define SIZE 100
```

```
typedef struct {
```

```
    RcdType rc; //记录项
```

```
    int      next; //指针项
```

```
}SLNode; //表结点类型
```

```
typedef struct {
```

```
    SLNode r[SIZE]; //0号单元为表头结点
```

```
    int      length; //链表当前长度
```

```
}SLinkListType;
```

6.2.3 表

□ 表插入排序的算法

```
void LInsertionSort (SLinkList
```

```
// 对记录序列SL[1..n]作表插
```

```
SL.r[0].key = MAXINT ;
```

```
SL.r[0].next = 1; SL.r[1].next = 2;
```

```
//初始状态，一个是有顺序的，头结点的下标为0
```

```
for ( i=2; i<=n; ++i ){//i指示要处理的位置
```

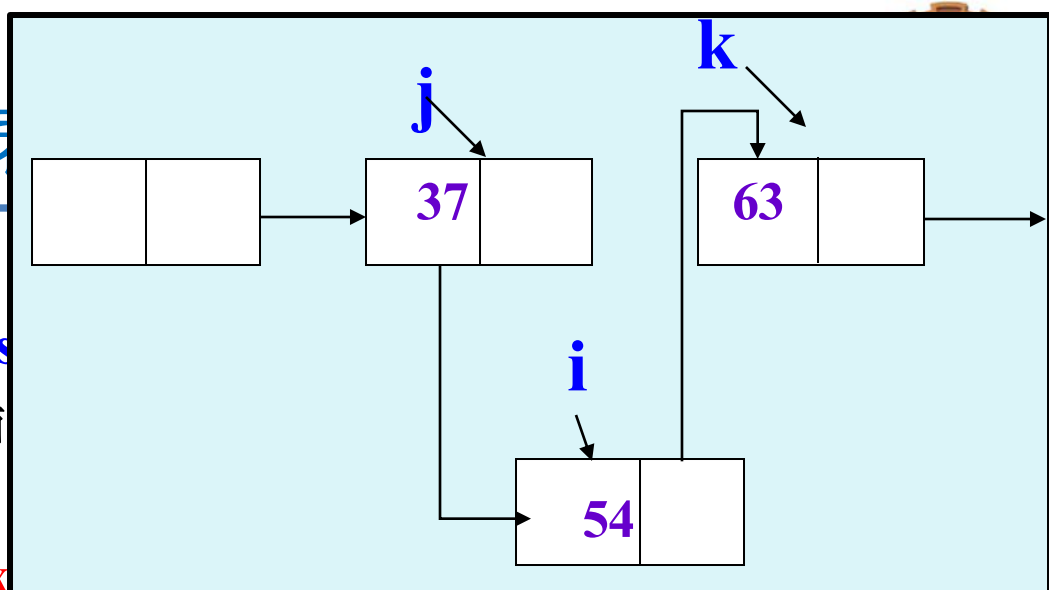
```
    for ( j=0, k = SL.r[0].next; SL.r[k].key<=SL.r[i].key ; ; ) //查找插入位置
```

```
        {j=k; k=SL.r[k].next;}
```

```
    SL.r[j].next = i; SL.r[i].next = k; // 结点i插入在结点j和结点k之间
```

```
}
```

```
// LinsertionSort
```



表插入排序的基本操作仍然是将一个记录插入到已排好序的有序表中，比较次数仍是 $O(n^2)$ ，但是不需移动，只是修改指针值即可。



6.2.3 表插入排序

□表插入排序的算法

表插入排序算法的结果只是求得一个有序链表，只能进行顺序查找，不能进行随机查找。

如果想要随机查找，则需要重新排列，使得它物理有序。怎么排序？



□根据建好的链表重新排列元素，使得物理有序

重新排列目的：将链表中第i个结点移动到数组中第i个位置

```
q = SL.r[p].next;
```



6.2.3 表插入排序

□ 根据建好的链表重新排列元素，使得物理有序

$i=1$
 $p=6$
 $q=7$

	0	1	2	3	4	5	6	7	8
M	49	38	65		76	13	27	52	
	6	8	1	5	0	4	7	2	3

$i=2$
 $p=7$
 $q=2$

	0	1	2	3	4	5	6	7	8
M	13	38	65	97	56	49	27	52	
	6	(6)	1	5	0	4	8	2	3

保留49的新的位置，
以便之后能找到49

p : 指示第 i 个记录的当前位置

i : 指示第 i 个记录应在的位置

q : 指示第 $i+1$ 个记录的当前位置



6.2.3 表插入排序

□ 根据建好的链表重新排列元素，使得物理有序

$i=2$
 $p=7$
 $q=2$

	0	1	2	3	4	5	6	7	8
M		13	38	65	97	76	49	27	52
			65	97	76	49	27		
6		(6)	1	5	0	4	8	2	3

交换

$i=3$
 $p=2$
 $q=1$

	0	1	2	3	4	5	6	7	8
M		13	27	65	97	76	49	38	52
6		(6)	(7)	5	0	4	8	1	3

保留38的
新的位置

p : 指示第 i 个记录的当前位置

i : 指示第 i 个记录应在的位置

q : 指示第 $i+1$ 个记录的当前位置



6.2.3 表插入排序

□ 根据建好的链表重新排列元素，使得物理有序

$i=3$
 $p=7$
 $q=1$

0	1	2	3	4	5	6	7	8
M	13	27	65	97	76	49	38	52
6	(6)	(7)	5	0	4	8	1	3

交换

$i=4$
 $p=1$
 $q=8$

0	1	2	3	4	5	6	7	8
M	13	27	38	97	76	49	65	52
6	(6)	(7)	(7)	0	4	8	5	3

p : 指示第 i 个记录的当前位置

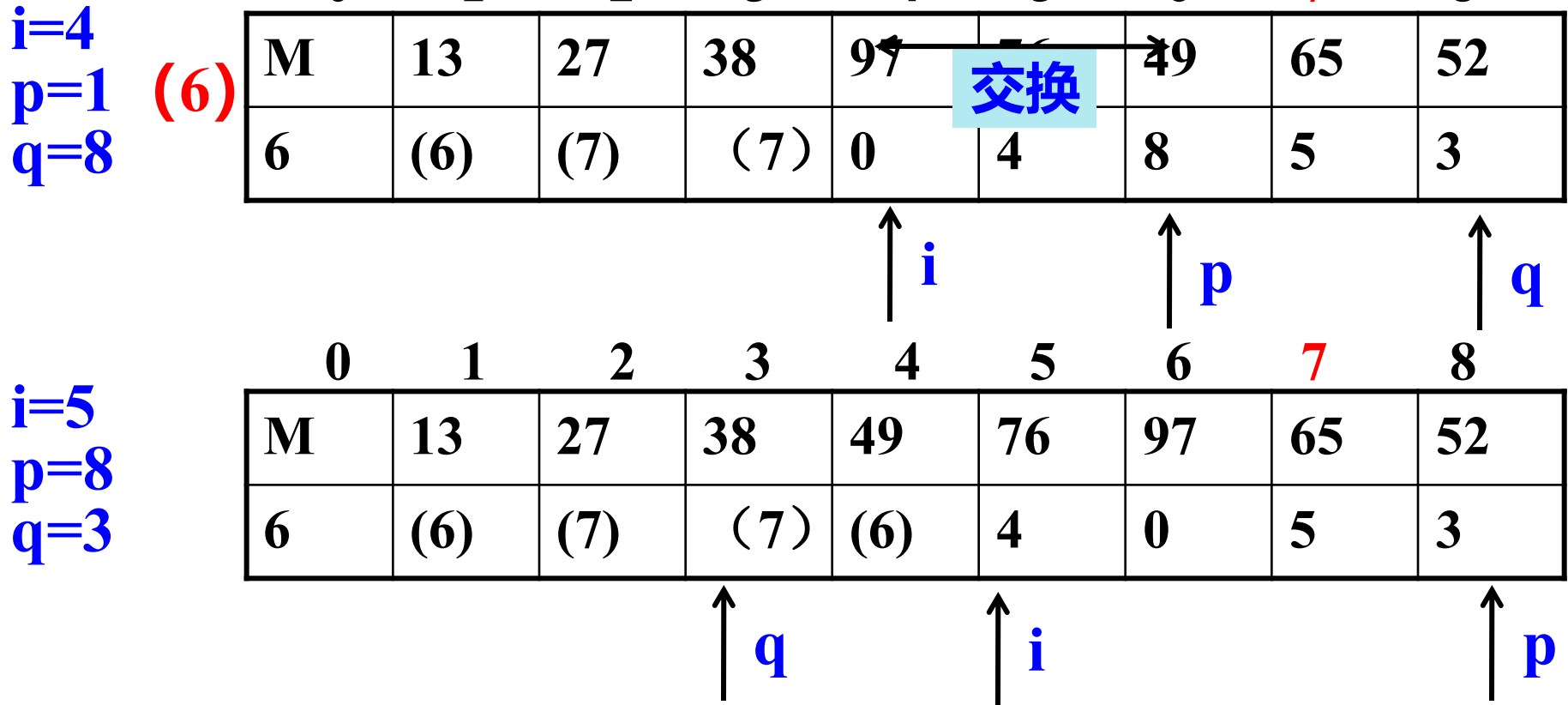
i : 指示第 i 个记录应在的位置

q : 指示第 $i+1$ 个记录的当前位置



6.2.3 表插入排序

□ 根据建好的链表重新排列元素，使得物理有序





6.2.3 表插入排序

□ 根据建好的链表重新排列元素，使得物理有序

$i=5$
 $p=8$
 $q=3$

0	1	2	3	4	5	6	7	8
M	13	27	38	49	76	97	65	52
6	(6)	(7)	(7)	(6)	4	0	5	3

交换

$i=6$
 $p=3$
 $q=5$

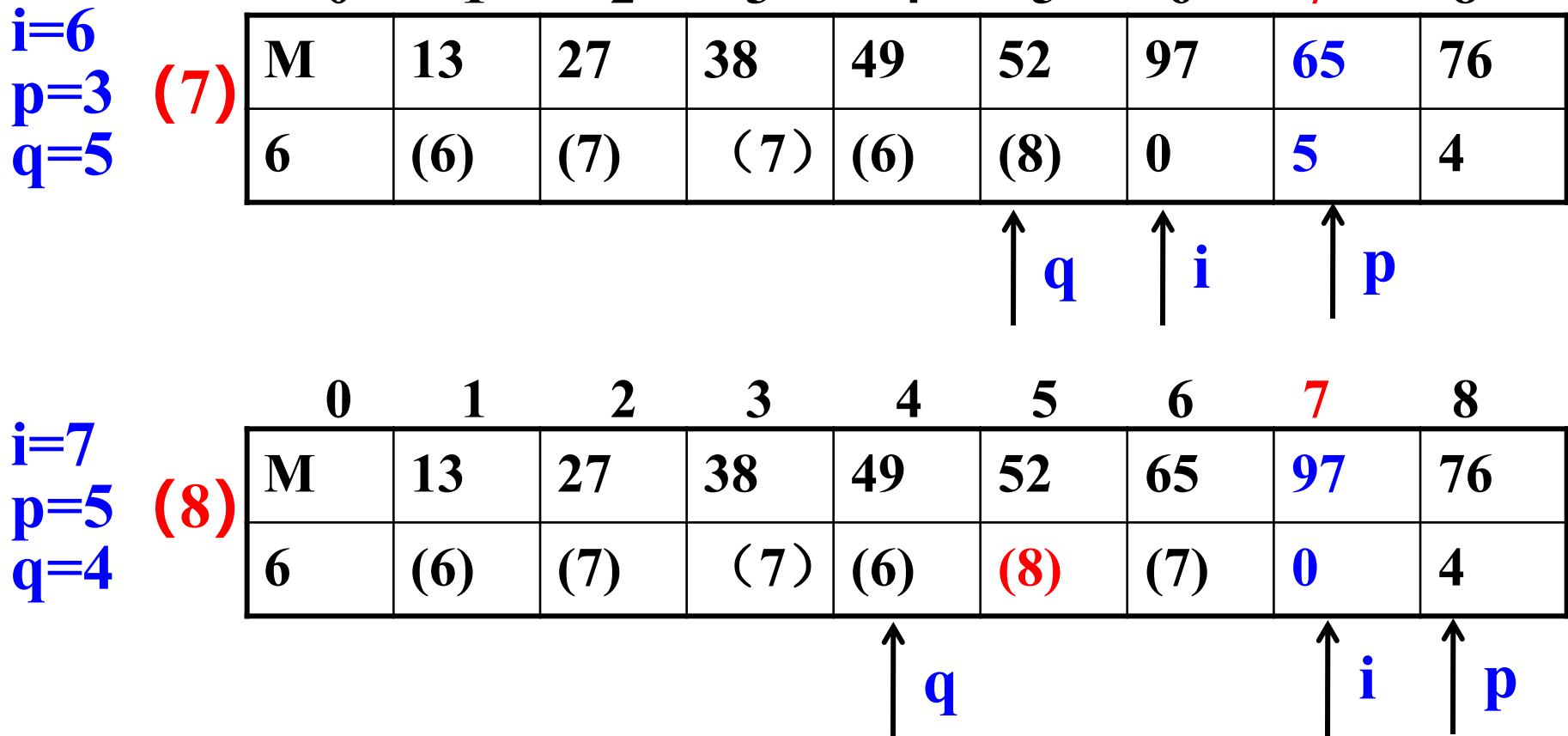
(7)

0	1	2	3	4	5	6	7	8
M	13	27	38	49	52	97	65	76
6	(6)	(7)	(7)	(6)	(8)	0	5	4



6.2.3 表插入排序

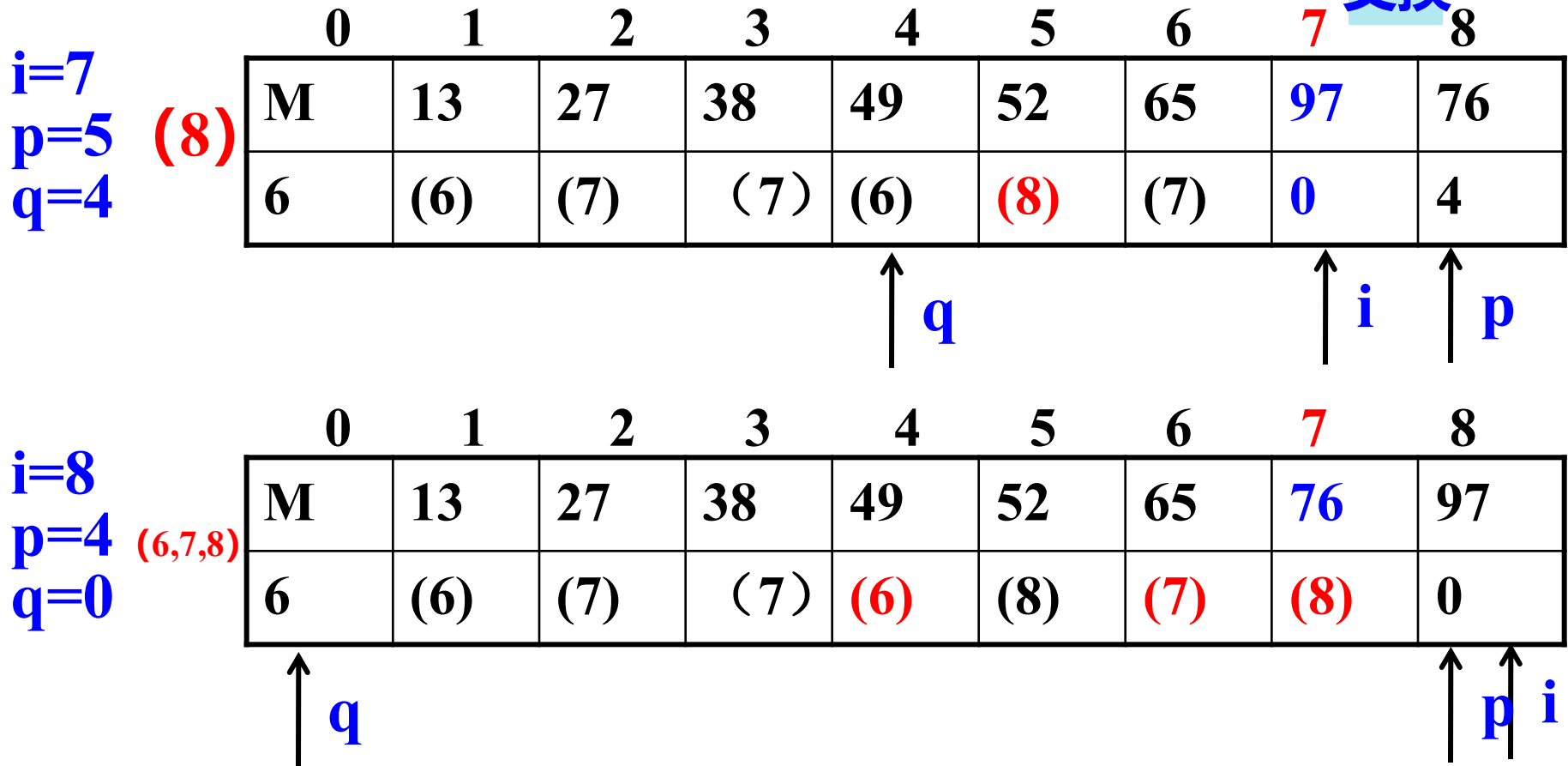
□ 根据建好的链表重新排列元素，使得物理有序





6.2.3 表插入排序

□ 根据建好的链表重新排列元素，使得物理有序 → 交换





6.2.3 表插入排序

□根据建好的链表重新排列元素，使得物理有序

```
void Arrange (SLinkListType &SL) {  
    p = SL.r[0].next;      // p指示第一个记录的当前位置  
    for ( i=1; i<SL.length; ++i ) { // SL.r[1..i-1]已经按照关键字排列  
        while (p<i) p = SL.r[p].next; //找到第i个记录，用p指示其当前位置  
        q = SL.r[p].next;           // q指示第i+1个记录所在的位置  
        if ( p!= i ) { //如果p与i相等，则表明已在正确的位置上，  
                        那就不需要调整了  
            SL.r[p]←→SL.r[i]; // 交换记录，使第i个记录到位  
            SL.r[i].next = p; //指向被移走的记录，以便通过while找到正确的值  
        }  
        p = q; // 为找第i+1个记录作准备  
    }  
} // Arrange
```



6.2.4 希尔排序

□ 希尔排序的思想

(1)对待排记录序列先作“宏观”调整，再作“微观”调整。

(2)所谓“宏观”调整，指的是，“跳跃式”的插入排序。



6.2.4 希尔排序

□ 希尔排序概述

(1)将记录序列分成若干子序列，分别对每个子序列进行插入排序。

例如：将 n 个记录分成 d 个子序列：

$\{R[1], R[1+d], R[1+2d], \dots, R[1+kd]\}$

$\{R[2], R[2+d], R[2+2d], \dots, R[2+kd]\}$

.....

$\{R[d], R[2d], R[3d], \dots, R[kd], R[(k+1)d]\}$

(2)其中， d 称为增量，它的值在排序过程中从大到小逐渐缩小，直至最后一趟排序减为 1。

希尔排序是按照不同步长对元素进行插入排序：

1. 当刚开始元素很无序的时候，步长最大，所以每次插入排序的元素个数很少，速度很快；
2. 当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。

所以，希尔排序的时间复杂度会比 $O(n^2)$ 好一些。

这两点也是希尔排序的设计动机和思路。



16	25	12	30	47	11	23	36	9	18	31
----	----	----	----	----	----	----	----	---	----	----

第一趟希尔排序，设增量 $d = 5$

11	23	12	9	18	16	25	36	30	47	31
----	----	----	---	----	----	----	----	----	----	----

第二趟希尔排序，设增量 $d = 3$

9	18	12	11	23	16	25	31	30	47	36
---	----	----	----	----	----	----	----	----	----	----

第三趟希尔排序，设增量 $d = 1$

9	11	12	16	18	23	25	30	31	36	47
---	----	----	----	----	----	----	----	----	----	----



6.2.4 希尔排序

□ 希尔排序算法

```
void ShellSort (SqList &L, int dlta[], int t)
{
    // dlta[]为增量序列
    for (k=0; k<t; ++k) // t为dlta长度
        ShellInsert(L, dlta[k]);
    //一趟增量为dlta[k]的插入排序
}
```



6.2.4 希尔排序

□ 希尔排序算法

```
void ShellInsert ( SqList &L, int dk ) { // 一趟希尔排序
    for ( i=dk+1; i<=L.length; ++i )
        // 将L.r[i]插入到间隔为dk的子序列中合适的位置
        if (L.r[i].key < L.r[i-dk].key) {
            L.r[0] = L.r[i];                // 暂存在R[0]
            for (j=i-dk; j>0 && L.r[0].key < L.r[j].key; j-=dk)
                L.r[j+dk] = L.r[j]; // 记录后移，查找插入位置
            L.r[j+dk] = L.r[0]; // 插入
        }
    }
```



6.2.4 希尔排序

□ 希尔排序算法分析

(1) 稳定性

希尔排序是不稳定的排序方法

(2) 算法效率

(1) 时间复杂度

平均 $O(n^{1.3})$ 到平均 $O(n^{1.5})$

(2) 空间复杂度

$O(1)$

至今仍未求得一种最好的增量序列($dlta[]$)!

希尔排序是按照不同步长对元素进行插入排序：

1. 当刚开始元素很无序的时候，步长最大，所以每次插入排序的元素个数很少，速度很快；
2. 当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。

所以，希尔排序的时间复杂度会比 $O(n^2)$ 好一些。

这两点也是希尔排序的设计动机和思路。

由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以shell排序是不稳定的；



第六章 排序

6.1 概述

6.2 插入排序

6.3 起泡（冒泡）排序

6.4 选择排序

6.5 归并排序

6.6 基数排序

6.7 内部排序方法的比较



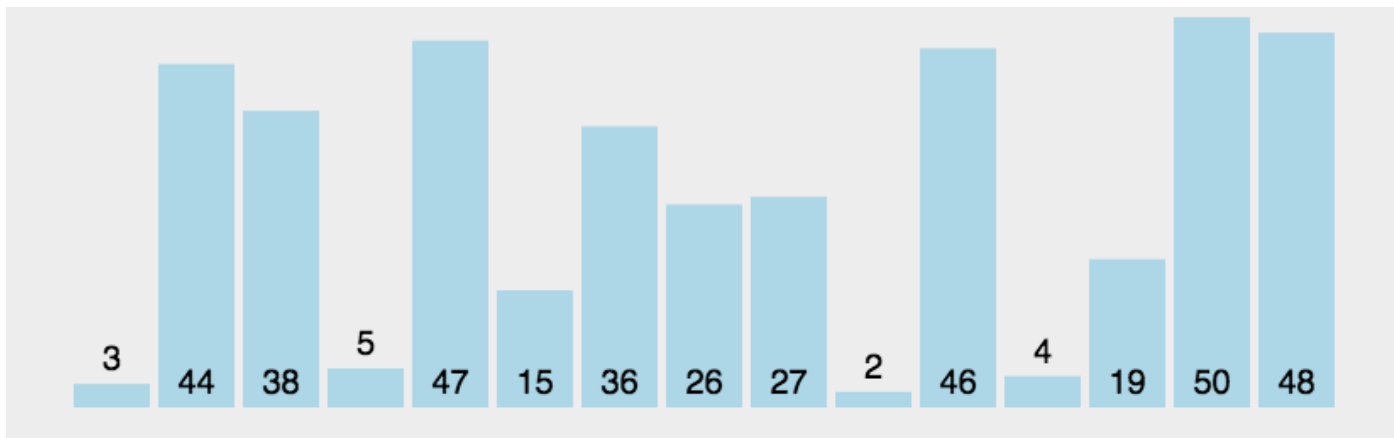
6.3.1 起泡排序

□ 基本思想

(1)从第一个记录开始，两两记录比较，若 $F[1].key > F[2].key$ ，则将两个记录交换；

(2)第1趟比较结果将序列中关键字最大的记录放置到最后一个位置，称为“沉底”，而最小的则上浮一个位置；第2趟比较将第二大的元素放置到倒数第二的位置，以此类推。

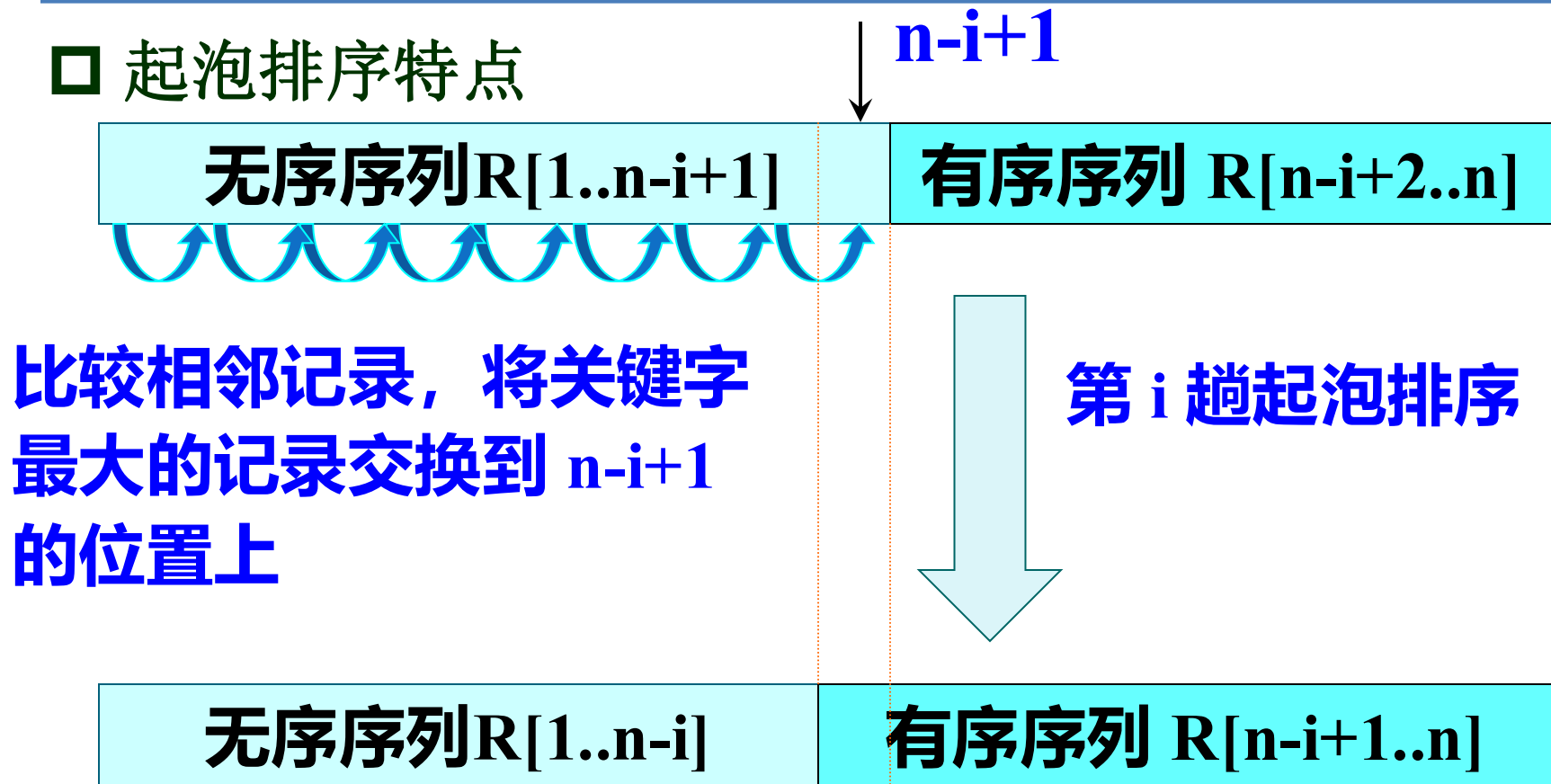
(3) n 个记录比较 $n-1$ 遍(趟)。





6.3.1 起泡排序

□ 起泡排序特点



气泡排序的特点：

有序序列中的关键字值都比无序序列中的大



6.3.1 起泡排序

□ 算法

```
void BubbleSort(SqList &L)
{
    int i, j, noswap; SqList temp;
    for(i=1;i<=n-1;i++)
    {
        noswap=TRUE;
        for(j=1;j<=n-i;j++)
            if (L.r[j].key>L.r[j+1].key)
            {
                temp=L.r[j]; L.r[j]=L.r[j+1];
                L.r[j+1]=temp;
                noswap=FALSE;
            }
        if (noswap) break; // 两两都不用互换, 说明已经排好序了
    }
}
```



6.3.1 起泡排序

□ 算法分析

最好的情况（关键字在记录序列中顺序有序）：
只需进行一趟起泡

“比较” 的次数：

$n-1$

“移动” 的次数：

0

最坏的情况（关键字在记录序列中逆序有序）：
需进行 $n-1$ 趟起泡，**每一次比较交换都移动3次**

“比较” 的次数：

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

“移动” 的次数：

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



6.3.1 起泡排序

□ 算法分析

(1) 稳定性

起泡排序是稳定的排序方法。

(2) 时间复杂性

最好情况：比较 $O(n)$, 移动 $O(1)$

最坏情况：比较 $O(n^2)$, 移动 $O(n^2)$

平均情况： $O(n^2)$

(3) 空间复杂性 $O(1)$



- **(希尔) Shell 排序改进了插入排序**
- **快速排序可以改进起泡排序**

**欲知详情，请看后续分解：
分治思想视角下的排序**

第六章 排序



6.1 概述

6.2 插入排序

6.3 起泡排序

6.4 选择排序

6.5 基数排序

6.6 内部排序方法的比较



6.4 选择排序

6.4.1 简单选择排序

6.4.2 树形选择排序

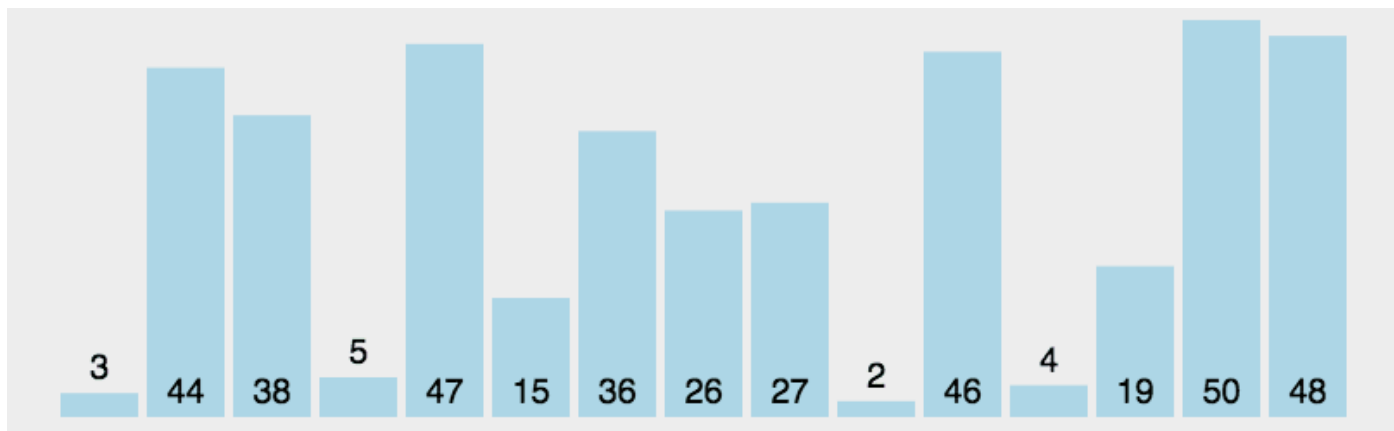
6.4.3 堆排序



6.4.1 简单选择排序

□ 基本思想

- (1)第一次从 n 个关键字中选择一个最小值，确定第一个；
- (2)第二次再从剩余元素中选择一个最小值，确定第二个；
- (3)共需 $n-1$ 次选择。





6.4.1 简单选择排序

□ 操作过程

设需要排序的表是 $A[n+1]$:

(1)第一趟排序是在无序区 $A[1]$ 到 $A[n]$ 中选出最小的记录, 将它与 $A[1]$ 交换, 确定最小值;

(2)第二趟排序是在 $A[2]$ 到 $A[n]$ 中选关键字最小的记录, 将它与 $A[2]$ 交换, 确定次小值;

(3)第 i 趟排序是在 $A[i]$ 到 $A[n]$ 中选关键字最小的记录, 将它与 $A[i]$ 交换;

(4)共 $n-1$ 趟排序。



6.4.1 简单选择排序

□ 操作过程

简单选择排序与气泡排序的区别在：

气泡排序每次比较后，如果发现顺序不对立即进行交换，而选择排序不立即进行交换而是找出最小关键字记录后再进行交换。



6.4.1 简单选择排序

□ 算 法

```
void SelectSort(SqList &L){  
    int i, j, low;  
    for(i=1;i<L.length;i++){  
        low=i;  
        for(j=i+1;j<=L.length;j++){  
            if(L.r[j].key<L.r[low].key)  
                low=j;  
        }  
        if(i!=low)  
            {L.r[0]=L.r[i];L.r[i]=L.r[low];L.r[low]=L.r[0];}  
    }  
}
```



6.4.1 简单选择排序

□ 算法分析

4. 算法分析

(1) 稳定性

简单选择排序方法是不稳定的。

(2) 时间复杂度

比较 $O(n^2)$, 移动最好 $O(1)$, 最差 $O(n)$

(3) 空间复杂度

为 $O(1)$ 。



6.4.2 树形选择排序

□ 引入

树形选择排序，又称锦标赛排序：按锦标赛的思想进行排序，目的是减少选择排序中的重复比较次数。

例如：4,3,1,2 在选择排序中3和4的比较次数共发生了三次。



6.4.2 树形选择排序

□ 引入

- ✓ 显然，在 n 个关键字中选出最小值，至少进行 $n-1$ 次比较，然而，继续在剩余 $n-1$ 个关键字中选择次小值就并非一定要进行 $n-2$ 次比较，若能利用前 $n-1$ 次比较所得信息，则可减少以后各趟选择排序中所用的比较次数。
- ✓ 实际上，体育比赛中的锦标赛便是一种选择排序。

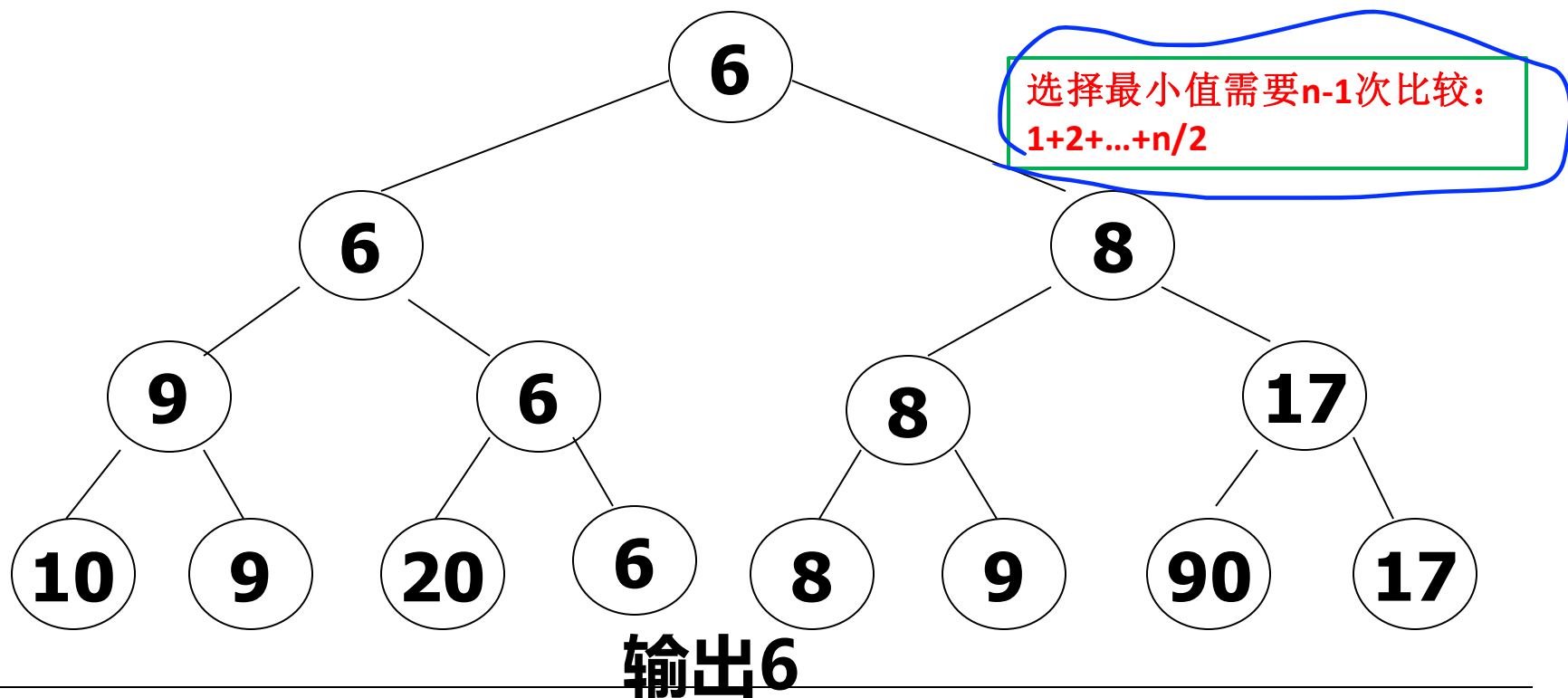
例如，在8个运动员决出前3名至多需要11场比赛，而不是7（全比一次）+6（去掉第一之后再全比一次）+5（去掉两个后再比一次）=18场比赛（它的前提是若乙胜丙，甲胜乙，则认为甲必能胜丙）



6.4.2 树形选择排序

□ 示例演示

按照一种**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 $\lceil n/2 \rceil$ (向上取整) 个较小者之间再进行两两比较，如此重复，直到选出最小关键字的记录为止。选出最小关键字。



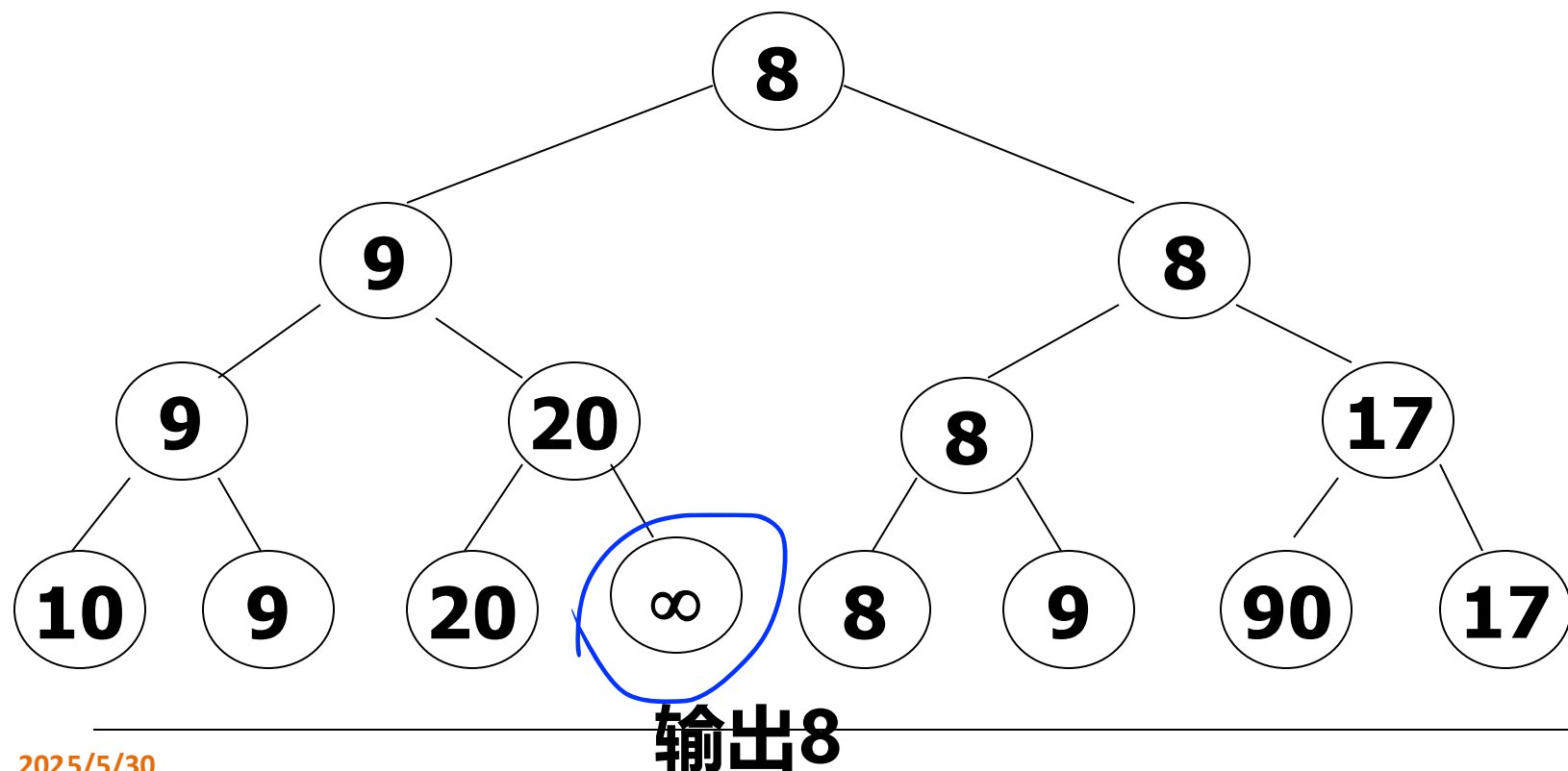
6.4.2

□ 示例演示

然后把最小关键字放最大值，然后选出次小关键字。

将最小关键字改为“最大值”后，从第二轮选择开始，每一次选择，只需要从该叶子结点开始，和其左（右）兄弟比较，修改从该叶子结点到根的路径上各结点的关键字即可。（其实和最大值比较也可以省略）

含有 n 个结点的完全二叉树的深度为： $\lceil \log_2 n \rceil + 1$ ，因此从第二轮选择开始，每一次选择需要进行 $\lceil \log_2 n \rceil$ 次比较。





6.4.2 树形选择排序

□ 算法分析

(1) 稳定性

树形选择排序方法是稳定的。

(2) 时间复杂度

比较 $O(n \log_2 n)$

(3) 空间复杂度

为 $O(n)$ 。

浪费存储空间比较大，和最大值进行多余的比较等缺点。

1964年J. Williams提出了堆排序。



6.4.3 堆排序

□ 引 入

堆排序属于选择排序:出发点是**利用前一次比较的结果,减少“比较”的次数。**

若能利用每趟比较后的结果,也就是在找出**关键字值最小**记录的同时,也找出**关键字值较小**的记录,则可减少后面的**选择**中所用的比较次数,从而提高整个排序过程的效率。

减少关键字之间的比较次数



查找最小值的同时,找出较小值



6.4.3 堆排序

□ 堆的定义

把具有如下性质的数组A表示的完全二叉树称为小根堆:

- (1) 若 $2*i \leq n$, 则 $A[i].key \leq A[2*i].key$;
- (2) 若 $2*i+1 \leq n$, 则 $A[i].key \leq A[2*i+1].key$

把具有如下性质的数组A表示的完全二叉树称为大根堆:

- (1) 若 $2*i \leq n$, 则 $A[i].key \geq A[2*i].key$;
- (2) 若 $2*i+1 \leq n$, 则 $A[i].key \geq A[2*i+1].key$



6.4.3 堆排序

□ 堆的性质（小根堆）

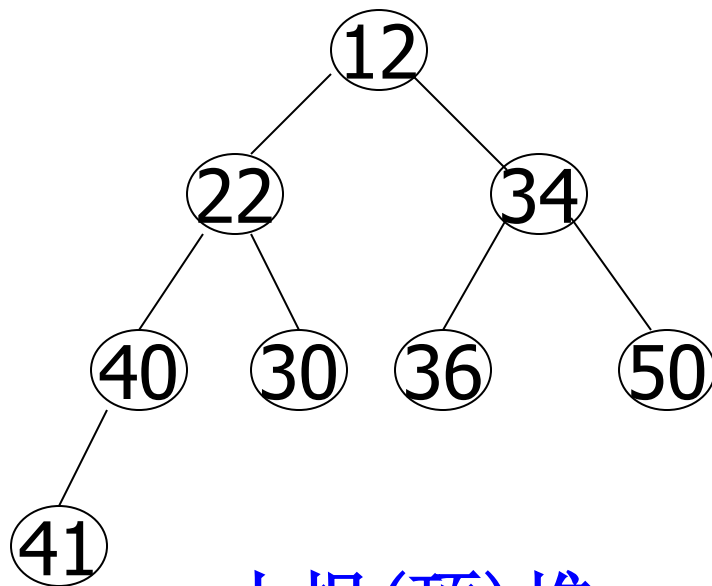
- 对于任意一个非叶结点的关键字，都不大于其左、右儿子结点的关键字。即 $A[i/2].key \leq A[i].key$ $1 \leq i/2 < i \leq n$
- 在堆中，以任意结点为根的子树仍然是堆。特别地，每个叶结点也可视为堆。每个结点都代表(是)一个堆。
 - ✓ 以堆（的数量）不断扩大的方式进行**初始建堆**。
- 在堆中（包括各子树对应的堆），其根结点的关键字是最小的。去掉堆中编号最大的叶结点后，仍然是堆。
 - ✓ 以堆的规模逐渐缩小的方式进行**堆排序**。



6.4.3 堆排序

□ 小根堆例子

0	1	2	3	4	5	6	7	8
	12	22	34	40	30	36	50	41



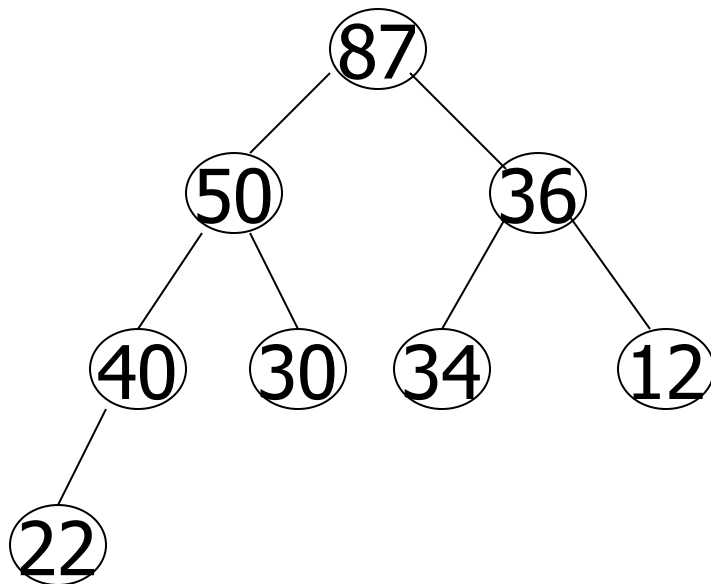
小根(顶)堆



6.4.3 堆排序

□ 大根堆例子

0	1	2	3	4	5	6	7	8
	87	50	36	40	30	34	12	22



大根(顶)堆



6.4.3 堆排序

□ 利用大根堆进行排序的示例演示

思路：

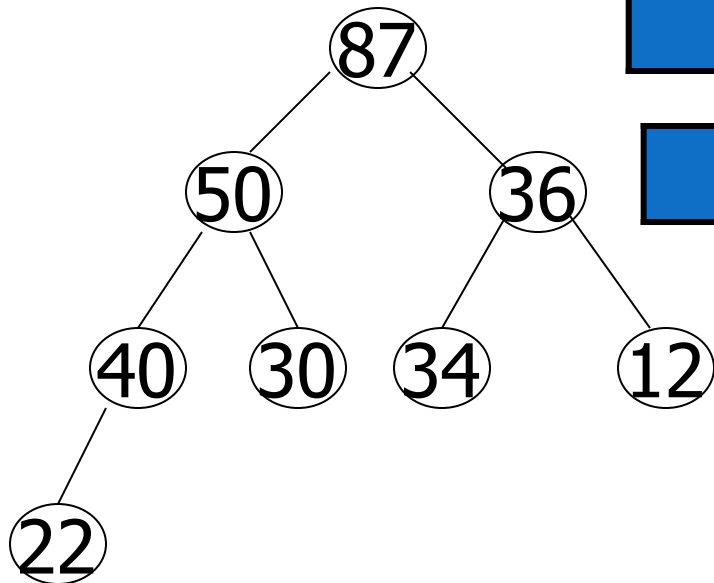
若将此序列对应的一维数组看成一个完全的二叉树，则堆的含义表明，完全二叉树中所有非终端节点的值均不大于（或不小于）其左右孩子节点的值。

若**输出堆顶**的最小值后，使得剩余 $n-1$ 个元素的序列重又建成一个堆，则得到 n 个元素的次小值。如此反复执行，便得到一个有序序列，这个过程称为**堆排序**。



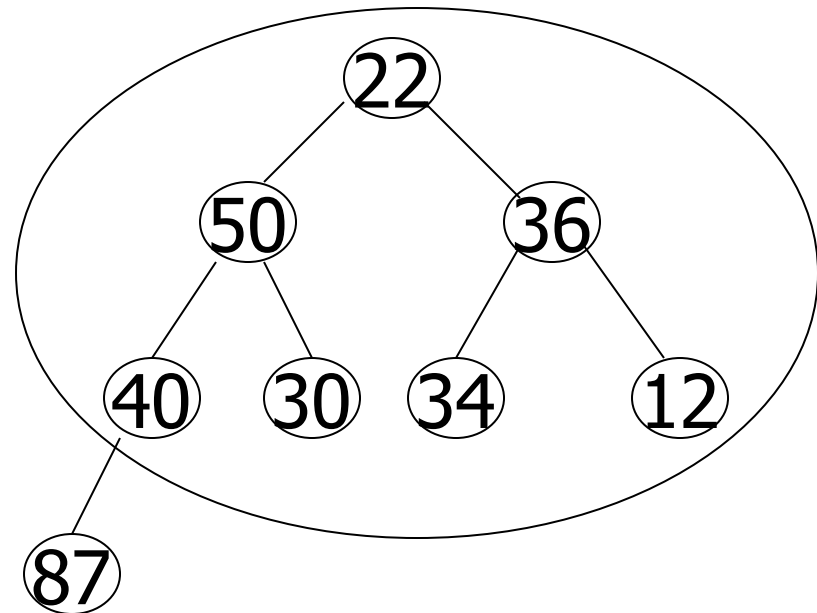
6.4.3 堆排序

□ 利用大根堆进行排序的示例演示



	87	50	36	40	30	34	12	22
--	----	----	----	----	----	----	----	----

	22	50	36	40	30	34	12	87
--	----	----	----	----	----	----	----	----





6.4.3 堆排序

□ 利用大根堆进行排序的示例演示

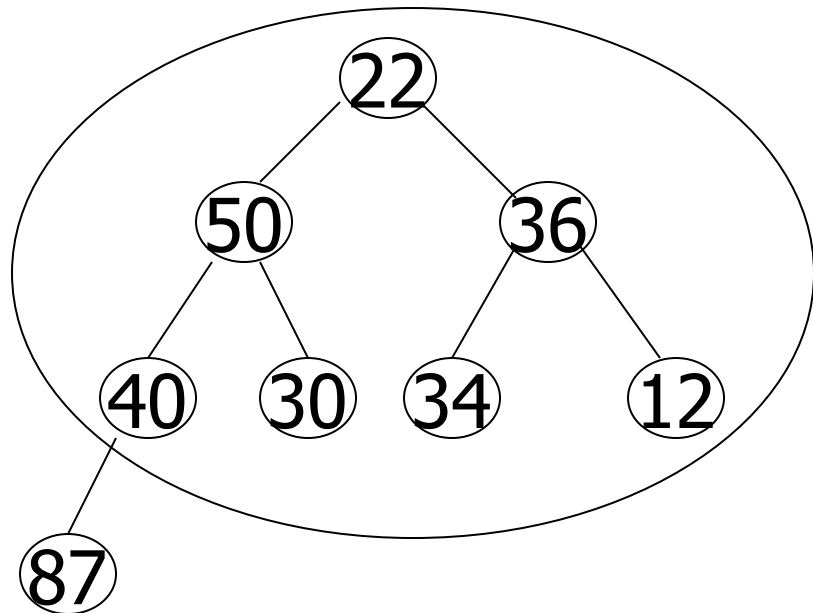
	22	50	36	40	30	34	12	87
--	----	----	----	----	----	----	----	----

思路：

87是最大值，应该放在数组的尾端，和尾端元素**22**进行交换，形成新的二叉树，如果这时候输出**87**，剩余的序列如何构成一个大根堆呢？

此时根节点的左右子树均为堆，则仅需**自上而下进行调整即可**。首先以堆顶元素和其左右子树根节点的值比较，由于左子树根节点的值大于右子树根节点的值，且大于根节点**22**，所以**50**和**22**交换；然后再交换**40**和**22**，形成新的大根堆。

从堆顶到叶子的调整过程称为“筛选”。





6.4.3 堆排序

□ 堆排序的思想

堆排序需解决两个问题：

- (1) 由一个无序序列建成一个堆。
- (2) 在输出堆顶元素之后，调整剩余元素成为一个新的堆。



6.4.3 堆排序

□ 堆排序的算法(采用大根堆)

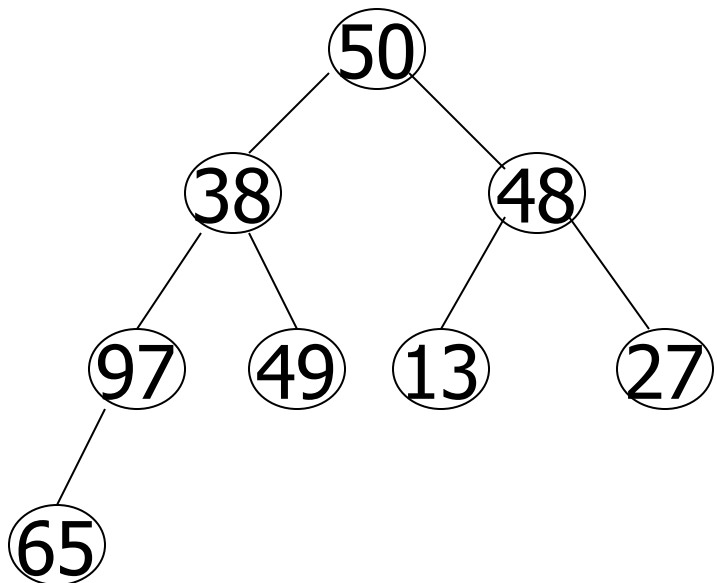
- (1) 按关键字建立 $A[1], A[2], \dots, A[n]$ 的大根堆;
- (2) 输出堆顶元素, 采用堆顶元素 $A[1]$ 与最后一个元素 $A[n]$ 交换, 最大元素得到正确的排序位置;
- (3) 此时前 $n-1$ 个元素不再满足堆的特性, 需重建堆;
- (4) 循环执行(2) (3)两步, 到排序完成。



6.4.3 堆排序

□ 如何由一个无序序列建成一个堆

例 对无序序列{50, 38, 48, 97, 49, 13, 27, 65}进行堆排序。



(1) 先建一个完全二叉树;

(2) 从最后一个非终端结点 (假设 i) 开始建堆, 不断进行“筛选”;

n 个结点, 最后一个非终端结点的下标是 $\lfloor n/2 \rfloor$;

(3) 然后从 i 依次往前重复步骤2

最后一个非终端结点为第4个结点（97），不需调整；然后看第3个结点（48），不需调整，再看第2个结点（38），需要自顶向下筛选。





6.4.3 堆排序

□ 建堆算法代码

```
for (i=H.length/2; i>0; --i)//对每个非终端结点进行调整  
    HeapAdjust ( H, i, H.length ); // 自顶向下筛选
```

筛选算法怎么写？



6.4.3 堆排序

□ 筛选算法

```
void HeapAdjust(HeapType &H, int s, int m){  
    int j;  
    RcdType rc;           //已知H.r[s...m]中的记录关键字除H.r[s].key  
    rc = H.r[s];          //之外均满足堆的定义，本函数调整H.r[s]的关  
                           //键字，使得H.r[s...m]成为大根堆；  
    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;  
        H.r[s] = H.r[j]; s = j;  
    }  
    H.r[s] = rc; //rc应插入在位置s上  
} // HeapAdjust
```




6.4.3 堆排序

□ 完整堆排序算法代码

```
void HeapSort(HeapType &H) {  
    int i;  
    RcdType temp;  
    for (i=H.length/2; i>0; --i) //第一步：建堆  
        HeapAdjust ( H, i, H.length );  
    for (i=H.length; i>1; --i) { //第二步：在输出堆顶元素之后，  
                                调整剩余元素成为一个新的堆  
        temp=H.r[i];H.r[i]=H.r[1];  
        H.r[1]=temp; //将堆顶记录和子序列最后一个记录交换  
        HeapAdjust(H, 1, i-1); //重新调整为大根堆  
    }  
} // HeapSort
```



6.4.3 堆排序

□ 算法分析

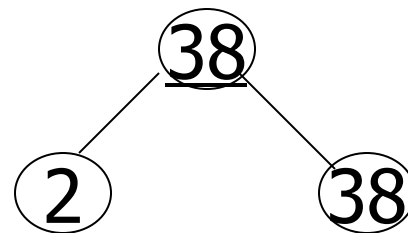
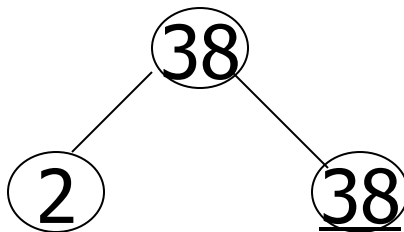
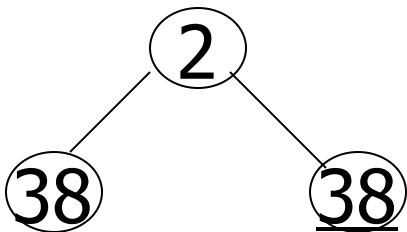
(1)堆排序是不稳定的排序。

(2)时间复杂度为 $O(n\log_2 n)$ 。

最坏情况下时间复杂度为 $O(n\log_2 n)$ 的算法。

(3)空间复杂度为 $O(1)$ 。

{2, 38, 38}





还有一种重要的排序：归并排序！

**欲知详情，请看后续分解：
分治视角下的排序**



第六章 排序

6.1 概述

6.2 插入排序

6.3 气泡排序

6.4 选择排序

6.5 基数排序

6.6 内部排序方法的比较



6.5.1 顺序基数排序

□ 基数排序的起源

(1)理论上可以证明, 对于基于关键字之间比较的排序, 无论用什么方法都至少需要进行 $\log_2 n!$ 次比较。

(2)由Stirling公式可知, $\log_2 n! \approx n \log_2 n - 1.44n + O(\log_2 n)$ 。所以基于关键字比较的排序时间的下界是 $O(n \log_2 n)$ 。因此不存在时间复杂性低于此下界的基于关键字比较的排序! 下节课证明

(3)只有不通过关键字比较的排序方法,才有可能突破此下界。

(4) **基数排序** (时间复杂性可达到线性级 $O(n)$)

- 不比较关键字的大小, 而根据**构成关键字的每个分量的值**, 排列记录顺序的方法, 称为**分配法排序 (基数排序)**。
- 而把关键字各个分量所有可能的取值范围的最大值称为**基数**或**桶**或**箱**

(5) **基数排序的适用范围**

- 显然, 要求关键字分量的取值范围必须是有限的, 否则可能要无限的箱。



6.5.1 顺序基数排序

□ 算法的基本思想

- 设待排序的序列的关键字都是位相同的**整数**（不相同，取位数的最大值），其位数为 $figure$ ，每个关键字可以各自含有 $figure$ 个**分量**，每个分量的值取值范围为 $0,1,\dots,9$ 即**基数**为10。依次从低位考查每个分量。
- 首先把全部数据装入一个队列A，然后按下列步骤进行：
 - 1.**初态**:设置10个队列，分别为 $Q[0], Q[1], \dots, Q[9]$ ，并且均为空
 - 2.**分配**:依次从队列中取出每个数据 $data$ ；第 $pass$ 遍处时，考查 $data.key$ 右起第 $pass$ 位数字，设其为 r ，把 $data$ 插入队列 $Q[r]$ ，取尽A，则全部数据被分配到 $Q[0], Q[1], \dots, Q[9]$ 。
 - 3.**收集**:从 $Q[0]$ 开始，依次取出 $Q[0], Q[1], \dots, Q[9]$ 中的全部数据，并按照取出顺序，把每个数据插入排队A。
 - 4.**重复**1,2,3步，对于关键字中有 $figure$ 位数字的数据进行 $figure$ 遍处理，即可得到按关键字有序的序列。



6.6.1 顺序基数排序

□ 事例演示

Q[0] 890 210

Q[1] 321 901

Q[2] 432 012

Q[3] 123 543

Q[4]

Q[5] 765

Q[6] 986

Q[7] 987

Q[8] 018 678 098

Q[9] 789 109

待排序关键字：321 986

123 432 543 018 765 678

987 789 098 890 109 901

210 012

经过这轮排序，已经将所有数
按照最后一位排好序



6.6.1 顺序基数排序

□ 事例演示

Q[0] 901 109

Q[1] 210 012 018

Q[2] 321 123

Q[3] 432

Q[4] 543

Q[5]

Q[6] 765

Q[7] 678

Q[8] 986 987 789

Q[9] 890 098

890 210 321 901 432 012 123

543 765 986 987 018 678 098

789 109

经过这轮排序，已经将所有数
按照后两位排好序



6.6.1 顺序基数排序

□ 事例演示

Q[0]	012	018	098						
Q[1]	109	123							
Q[2]	210								
Q[3]	321								
Q[4]	432			901	109	210	012	018	123
Q[5]	543			432	543	765	678	986	987
Q[6]	678			433	789	809	098		
Q[7]	765	789							
Q[8]	890								
Q[9]	901	986	987						



6.6.1 顺序基数排序

□ 排序过程

设待排记录A的关键字是figure位的正整数。

(1) 从最低位(个位)开始, 扫描关键字的pass位, 把等于0的插入Q[0],...,等于9的插入Q[9]。

(2) 将Q[0],...,Q[9]中的数据依次收集到A[]中。

(3) pass+1直到figure, 重复执行1, 2两步

本质: 借助辅助存储, 从低位到高位逐位排序的过程



6.6.1 顺序基数排序

□ 求关键字k的第p位算法

```
int RADIX(int k,int p)
{
    return((k/pow10(p-1))%10);
}
```



6.6.1 顺序基数排序

□ 基数排序算法 (粗糙伪代码)

```
void radixsort(int figure, QUEUE &A){  
    QUEUE Q[10];records data;  
    int pass, r, i; //pass用于位数循环,r取位数  
    for(pass=1;pass<=figure; pass++){  
        把10个队列置空;  
        while(!EMPTY(A)) {  
            取A中元素data,  
            计算data的关键字的第pass位的值r,  
            放入队列Q[r]中;  
        }  
        把10个队列中的值依次收集到A中;  
    }  
}
```



6.6.1 顺序基数排序

□ 基数排序算法（详细伪代码）

```
void radixsort(int figure, QUEUE &A){
    QUEUE Q[10]; records data;
    int pass, r, i;    //pass用于位数循环, r取位数
    for(pass=1; pass<=figure; pass++){
        for(i=0; i<=9; i++)
            MAKENULL(Q[i])//置空队列
        while(!EMPTY(A)) {
            data=FRONT(A); //取队头元素
            DEQUEUE(A); //删除队头元素
            r=RADIX(data.key, pass); //取位数数值
            ENQUEUE(data, Q[r]); //入队
        }
        for(i=0; i<=9; i++)
            While(!EMPTY(Q[i])) //对队列Qi中的每个元素收集到A中
            { data=FRONT(Q[i]);
              DEQUEUE(Q[i]); //
              ENQUEUE(data, A);
            }
    }
}
```



6.6.1 顺序基数排序

□ 问题分析

如果采用顺序(基数)队列，基数队列的长度怎么确定？

110 920 230 030 090 320 100 400

503 765 678 986 987 789 890 098

如果采用数组表示队列,队列的长度很难确定,太大造成浪费,小了会产生溢出。
一般采用链队列。



6.6.2 链式基数排序

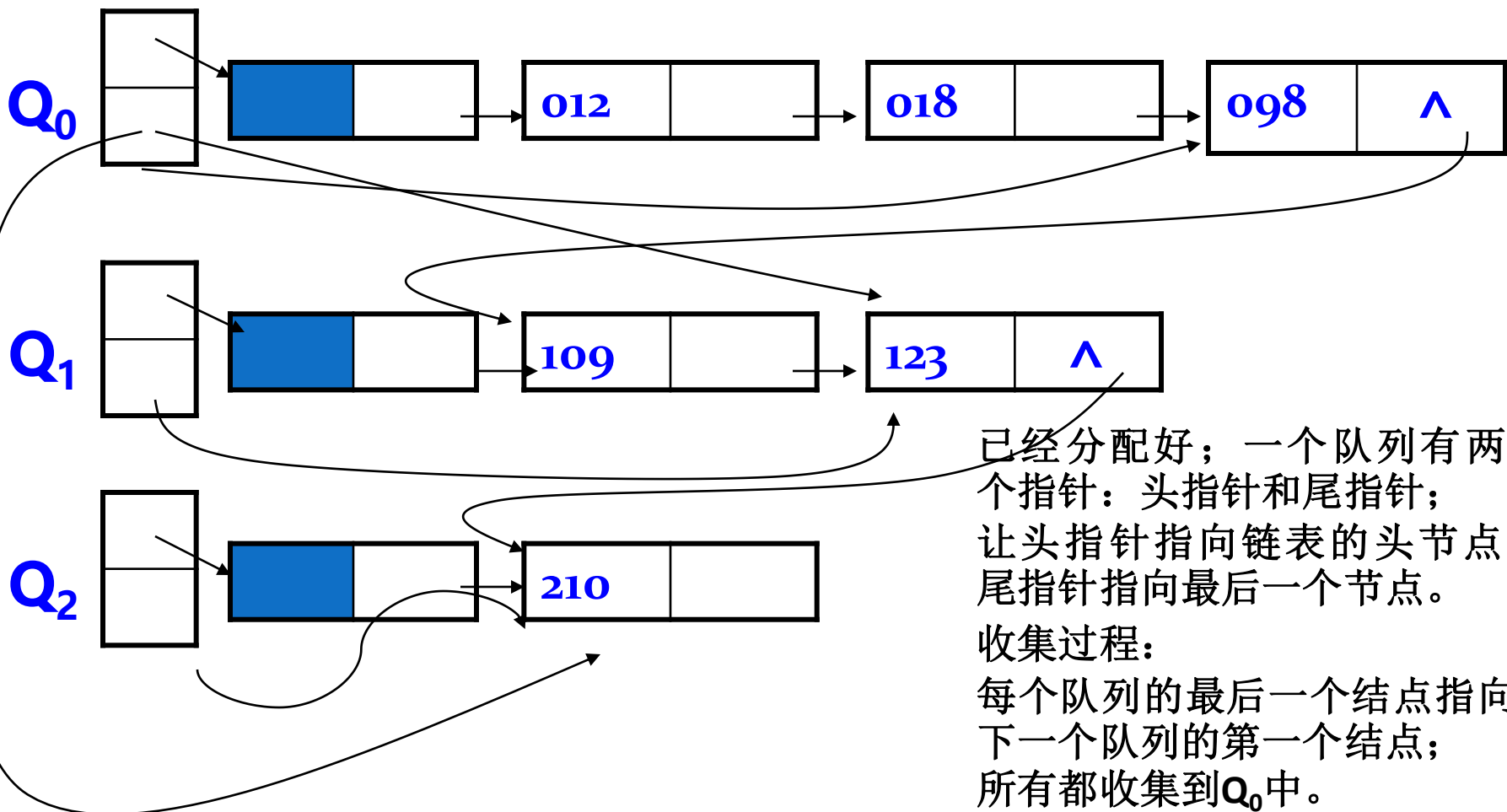
□ 存储结构

```
struct celltype{  
    records element;  
    celltype *next  
};//结点类型  
  
struct QUEUE{  
    celltype *front;  
    celltype *rear;  
};//队列定义
```



6.6.2 链式基数排序

□ 事例演示





6.6.2 链式基数排序

□ 收集算法关键字句

```
void CONCATENATE(QUEUE &Q[0],  
                  QUEUE &Q[1]){  
    if(!EMPTY(Q[1])){  
        Q[0].rear->next=Q[1].front->next;  
        Q[0].rear=Q[1].rear;  
    }//把Q0的最后一个节点指向Q1的第一个结点；修改Q0尾指针  
}
```



6.6.2 链式基数排序

□ 两种收集算法的比较

顺序收集算法

```
For(i=0;i<=9;i++)  
While(!EMPTY(Q[i]))  
{data=FRONT(Q[i]);  
  DEQUEUE(Q[i]);  
  ENQUEUE(data,A);  
} //顺序需要入队和出队操作;
```

链式收集算法

```
for(i=1;i<=9;i++)  
  CONCATENATE(Q[0],Q[i]);  
  A=Q[0];  
//但是链式只需要修改指针就可以了;  
(把所有的关键字都收集到Q[0]中)
```



6.6.2 链式基数排序

□ 算法分析

基数排序是稳定的
时间复杂性与空间复杂性分析：

设关键字位数为 d
则时间复杂性为 $O(dn)$
考虑到 d 是一个常数
时间复杂性为 $O(n)$
空间复杂性 $O(n)$



第六章 排序

6.1 概述

6.2 插入排序

6.3 气泡排序

6.4 选择排序

6.5 归并排序

6.6 基数排序

6.7 内部排序方法的比较



6.7 内部排序方法的比较

排序方法	比较次数		移动次数		稳定性	附加存储	
	最好	最差	最好	最差		最好	最差
直接插入排序	n	n^2	0	n^2	√	1	
冒泡排序	n	n^2	0	n^2	√	1	
快速排序	$n \log_2 n$	n^2	0	n^2	×	$\log_2 n$	n
简单选择排序	n^2		0	n	×	1	
堆排序	$n \log_2 n$		$n \log_2 n$		×	1	
归并排序	$n \log_2 n$		$n \log_2 n$		√	n	
基数排序		dn		√	n	



本章小结

- ✓ 熟练掌握：
 - 直接插入排序、希尔排序、冒泡排序、快速排序、简单选择排序、堆排序、归并排序、基数排序的思想和算法。充分了解各种排序算法的应用背景和优缺点。
- ✓ 重点学习：
 - 加强各种排序算法在实际应用中的训练，提高实际应用水平。