9.3 哈希表

以上两节讨论的查找表的各种结构的共同特点：

- 数据元素在表中的位置和它的关键字之间不存在一个确定的关系；
- 查找的过程为给定值依次和表中各数据元素的关键字进行比较；
- 查找的效率取决于和给定值进行比较的数据元素的关键字个数。

若能预先知道所查数据元素在表中的位置，则可提高查找的效率。即要求：数据元素在表中位置和其关键字之间存在一种确定的关系。

9.3 哈希表

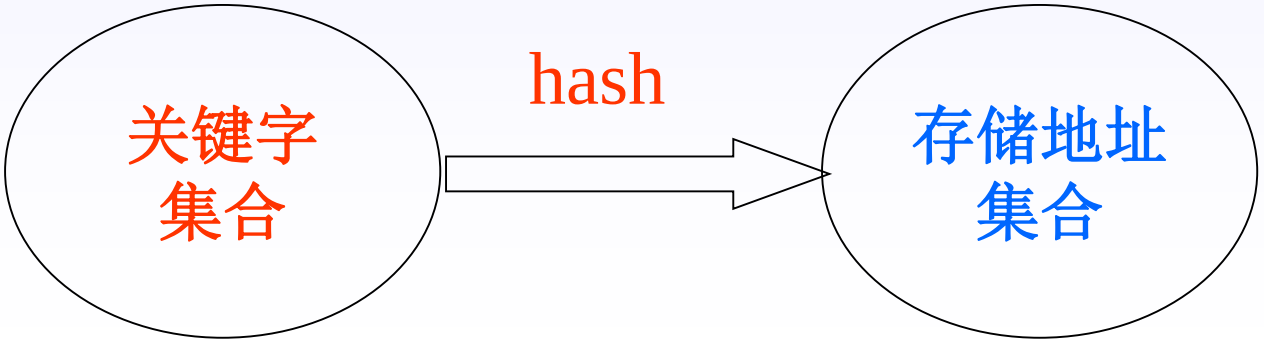
9.3.1 什么是哈希表

- 哈希方法在数据元素的存储位置与它的关键字之间建立一个确定的对应函数关系*Hash*( )，使每个关键字与结构中一个存储位置相对应： *Address*（a_i）＝ *Hash* (a_i.*key*)，

  其中： a_i是表中的一个元素

  Address(a_i)是a_i的存储地址

  a_i.key是a_i的关键字



9.3 哈希表

- 在存放数据元素时，依相同函数计算存储位置，并按此位置存放。这种方法就是哈希方法。在哈希方法中使用的存储位置与关键字之间的对应函数关系叫做哈希函数。而按此种想法构造出来的表或结构就叫做哈希表。
- 在查找时，首先对数据元素的关键字进行函数计算，把函数值当做数据元素的存储位置，在结构中按此位置取数据元素比较。若关键字相等，则查找成功。
- 使用哈希方法进行查找不必进行多次关键字的比较，查找速度比较快，可以直接到达或逼近具有此关键字的数据元素的实际存放地址。

9.3 哈希表

- 哈希函数是一个压缩映象函数。由于在设定哈希函数是需要考虑的关键字集合应包含所有可能的关键字，故比地址集合大得多。因此有可能经过哈希函数的计算，把不同的关键字映射到同一个哈希地址上，这就产生了冲突 (Collision)。

例：有一组数据元素，其关键字分别是12361, 7251, 3309, 30976, 采用的哈希函数是

$hash(x) = x \ \% \ 73$　其中，“%”是除法取余操作

则有：$hash(12361)=hash(7251)=hash(3309)=hash(30976) = 24$

即：对不同的关键字，通过哈希函数的计算，得到了同一哈希地址。

这些产生冲突的哈希地址相同的不同关键字称为同义词。

- 所以对于哈希方法，需要讨论以下两个问题：
  - 对于给定的一个关键字集合，选择一个计算简单且地址分布比较均匀的哈希函数，避免或尽量减少冲突；
  - 拟订解决冲突的方案。

9.3 哈希表

9.3.2 哈希函数的构造方法

哈希函数的要求：

- 哈希函数的定义域必须包括需要存储的全部关键字，如果哈希表允许有m个地址时，其值域必须在0到$m-1$之间。
- 哈希函数计算出来的地址应能均匀分布在整个地址空间中：若 *key*是从关键字集合中随机抽取的一个关键字，哈希函数应能以同等概率取0 到 $m-1$ 中的每一个值。
- 哈希函数应是简单的，能在较短的时间内计算出结果。

9.3 哈希表

1 直接定址法

此类函数取关键字的某个线性函数值作为哈希地址：

$$Hash(key) = a * key + b \quad \{a, b\text{为常数}\}$$

示例：有一组关键字如下：{ 942148, 941269, 940527, 941630, 941805, 941558, 942047, 940001 }。哈希函数为

$$Hash(key) = key - 940000$$

则有：

$Hash(942148) = 2148$　　$Hash(941269) = 1269$

$Hash(940527) = 527$　　$Hash(941630) = 1630$

$Hash(941805) = 1805$　　$Hash(941558) = 1558$

$Hash(942047) = 2047$　　$Hash(940001) = 1$

可以按计算出的地址存放记录。

9.3 哈希表

2 数字分析法

假设关键字集合中的每个关键字都是由s位数字组成(u1,u2, ...,us)，分析关键字集中的全体，并从中提取分布均匀的若干位或它们的组合作为地址。

- 示例：若哈希表地址范围有3位数字，下列各关键字的④⑤⑥位分布比较均匀，可做为记录的哈希地址。

①	②	③	④	⑤	⑥
9	4	2	1	4	8
9	4	1	2	6	9
9	4	0	5	2	7
9	4	1	6	3	0

- 当不知道全部关键字情况时，可以取关键字平方后的数字（与原来关键字的每一位都有关）中间几位作哈希地址。

9.3 哈希表

3 除留余数法

- 设哈希表的地址范围是 0到*m*-1, 取一个不大于 *m*，但最接近于或等于 *m* 的质数 *p*, 利用以下公式把关键字转换成哈希地址。哈希函数为：*hash* (*key*) = *key* % *p*　*p* ≤ *m*
- 示例：有一个关键字 *key* = 962148，哈希表大小 *m* = 25，即 *HT*[25]。取质数 *p*= 23。哈希函数 *hash* (*key*) = *key* % *p*。则哈希地址为:　*hash* (962148) = 962148 % 23 = 12。
- 可以按计算出的地址存放记录。需要注意的是，使用上面的哈希函数计算出来的地址范围是 0到 22，因此，从23到24这几个哈希地址实际上在一开始是不可能用哈希函数计算出来的，只可能在处理溢出时达到这些地址。

9.3 哈希表

4 折叠法

将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）做哈希地址。适于关键字位数很多，且每一位上数字分布大致均匀情况。

- 移位叠加：将分割后的几部分低位对齐相加
- 间界叠加：从一端沿分割界来回折送，然后对齐相加

例　关键字为：0442205864，哈希地址位数为4

```
  5 8 6 4
  4 2 2 0
      0 4
-------------
  1 0 0 8 8
```

移位叠加

H(key)=0088

```
  5 8 6 4
  0 2 2 4
      0 4
-------------
  6 0 9 2
```

间界叠加

H(key)=6092

9.3 哈希表

实际工作中需视情况的不同而采用不同的哈希函数，通常需要考虑的因素有：

A. 计算哈希函数所需时间；

B. 关键字的长度；

C. 哈希表的大小；

D. 关键字的分布情况；

E. 记录的查找频率。

总之，哈希函数是“杂凑”出来的。

9.3 哈希表

9.3.3 处理冲突的方法

解决冲突的方法又称为溢出处理技术。因为绝大多数哈希函数不能避免产生冲突，因此选择好的解决冲突的方法十分重要。

1 开放地址法

- 若设哈希表中的地址为 0 到 $m-1$，当要加入一个数据元素 R_2 时，用它的关键字 $R_2.key$，通过哈希函数 $hash(R_2.key)$ 的计算，得到它的存放位置 j，但是在存放时发现这个位置已经被另一个数据元素 R_1 占据了，即发生了冲突。为此，必须把 R_2 存放到表中“下一个”空位置中。如果表未满，则在允许的范围内必定还有空位置 。

9.3 哈希表

(1) 线性探查法 (Linear Probing)

- 需要查找或加入一个数据元素时，使用哈希函数计算位置号：　　$H_0 = hash(key)$
- 一旦发生冲突，在表中顺次向后寻找“下一个”空位置 H_i 的公式为：$H_i = (H_{i-1} + 1) \% m$，　$i = 1, 2, \ldots, m-1$

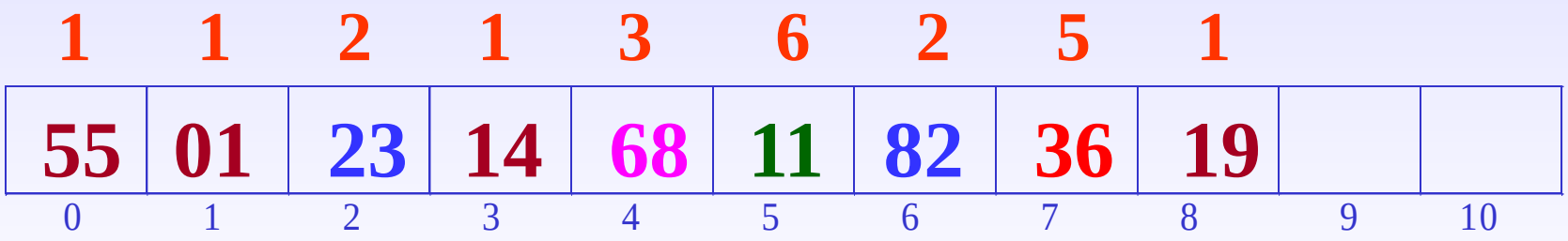
  即用以下的线性探查序列在表中寻找“下一个”空位置：

  $$H_0+1, H_0+2, \ldots, m-1, 0, 1, 2, \ldots, H_0-1$$

  亦可写成　$H_i = (H_0 + d_i) \% m$，　$d_i = 1, 2, \ldots, m-1$
- 当发生冲突时，探查下一个位置。当循环 $m-1$ 次后就会回到开始探查时的位置，说明待查关键字不在表内，而且表已满，不能再插入新关键字。

9.3 哈希表

- 设一组数据元素的关键字为(19,01,23,14,55,68,11,82,36)。哈希表为*HT*[11]，哈希函数为 *Hash (key) = key % 11*，采用线性探查法处理冲突，则上述关键字在哈希表中的存储位置如下图所示。



- 线性探查方法容易产生“堆积”，即在处理冲突过程中发生的两个第一个哈希地址不同的元素争夺同一个后续地址的现象，显然这将会导致查找时间增加。

9.3 哈希表

(2) 二次探查法 (quadratic probing)

- 为改善“堆积”问题，减少为完成查找所需的平均探查次数，可使用二次探查法。令哈希函数为： $H_0 = hash(x)$
- 二次探查法在表中寻找“下一个”空位置的公式为：

  $H_i = (H_0 + d_i) \% m$， $d_i = 1^2, -1^2, 2^2, -2^2, \ldots, k^2, -k^2$
- 式中的 m 是表的大小，当m是一个值为 $4k+3$ （ k是整数）的质数时， 采用二次探查法的H_i才能找到哈希表的每一个位置。
- 在做 $(H_0 + d_i) \% m$ 的运算时，$H_0 + d_i$ 可能为负数，因此实际算式可改为：

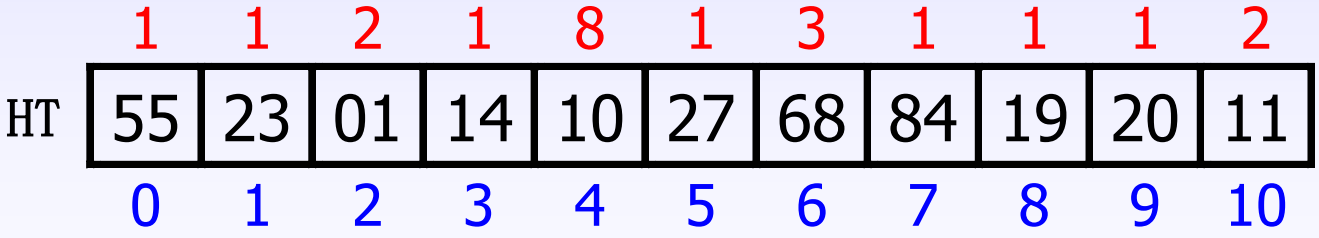
  if ($(H_0 + d_i) < 0$) $j = (H_0 + d_i + m) \% m$

9.3 哈希表

- 假设一组数据元素的关键字为 (19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10)。哈希表为*HT*[11]，采用的哈希函数是：

$$Hash(key) = key \% 11$$

采用二次探查法处理溢出，则上述关键字在哈希表中哈希位置如图所示。



其中，求关键字10的存储位置的过程为：H_0=10%11=10；

(10+1)%11=0； (10−1)%11=9； (10+4)%11=3； (10−4)%11=6；

(10+9)%11=8； (10−9)%11=1； (10+16)%11=4

9.3 哈希表

2 链地址法

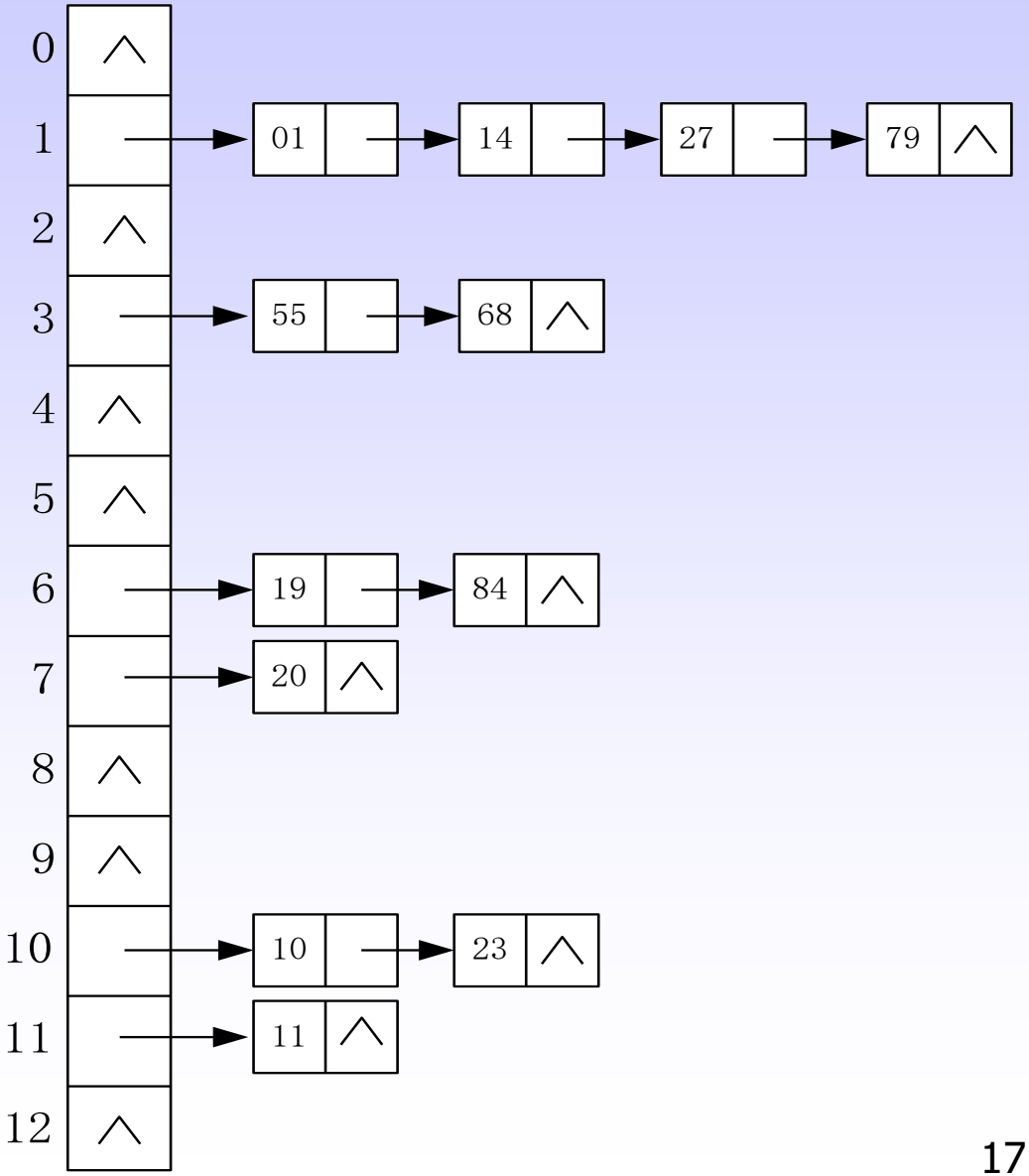
- 采用链地址法解决冲突时，首先对关键字集合用某一个哈希函数计算它们的存放位置。
- 若设哈希表地址空间的所有位置是从0到$m-1$，则关键字集合中的所有关键字被划分为m个子集，具有相同地址的关键字归于同一子集。同一子集中的关键字互称为同义词。
- 所有关键字互为同义词的数据元素都链接在同一个同义词子表中，各链表的表头结点组成一个向量。
- 向量的元素个数与地址空间的位置数一致。向量中的第 i 个元素中存放哈希函数为 i 的同义词子表的头指针。

9.3 哈希表

- 假设一组数据元素的关键字为 (19, 14, 23, 01, 68, 20, 84, 27, 55, 11, 10, 79)。哈希表为 *HT*[13]，采用的哈希函数是：

  Hash (key) = key % 13

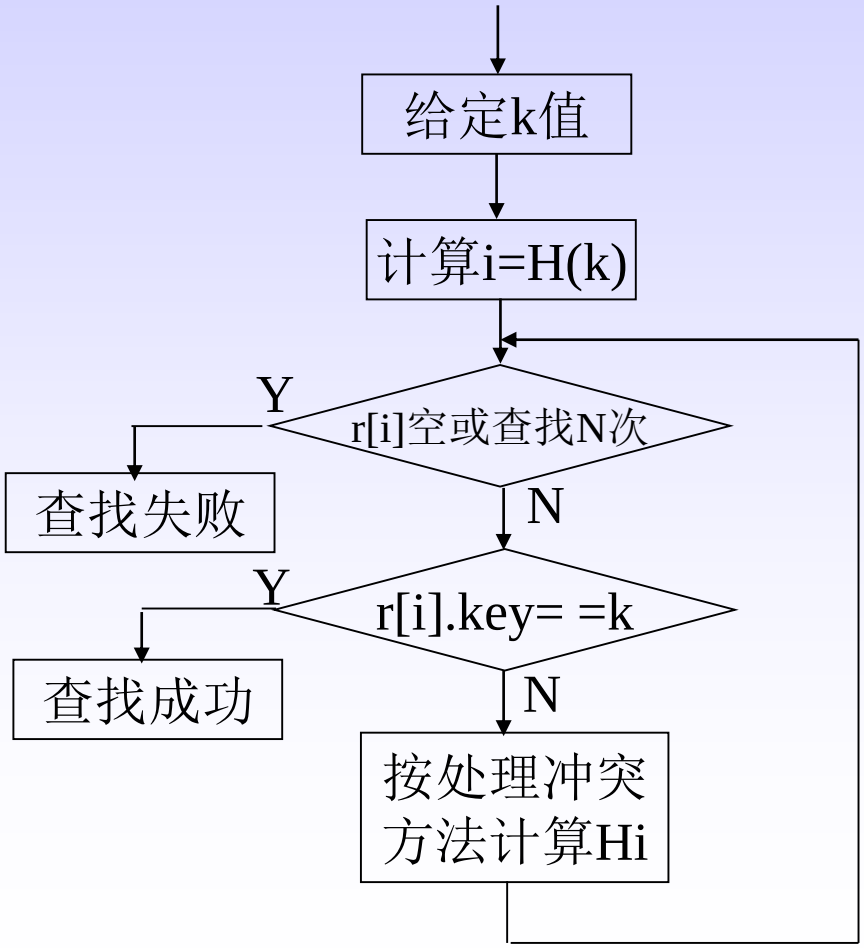
- 采用链地址法处理冲突，则上述关键字的哈希表如图所示。



9.3 哈希表

9.3.4 哈希表的查找与分析

哈希表的查找过程和造表过程一致，查找过程为:



9.3 哈希表

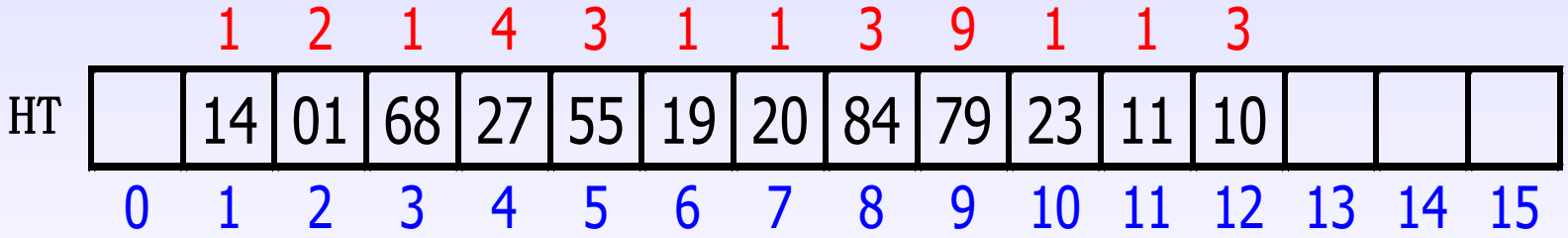
1 开放地址法

```
#define NULLKEY 0    // 正常关键字不为0
int SearchHash( ElemType *HT, KeyType key )
{  //采用线性探查法处理冲突，查找成功，返回i，否则返回-1。
    i = Hash( key ); j = 0;
    while( HT[i].key!=key && HT[i].key!=NULLKEY && j<MAX )
    {   i = (Hash(key)+j) % MAX;   j++; }
    if( HT[i].key==key )   return( i );
    else  return( -1 );
}
```

- 哈希表存放的数据元素的关键字不能重复。在插入新数据元素时，如果发现表中已有关键字相同的数据元素，则不再插入。

9.3 哈希表

- 用平均查找长度*ASL (Average Search Length)*衡量哈希方法的查找性能。 *ASL* 是指查找到表中已有数据元素的平均查找次数。它是找到表中各个已有数据元素的查找次数的平均值。例如：



$$ASL = \frac{1}{12}\sum_{i=1}^{12} Ci$$

$$= \frac{1}{12}(1\times 6 + 2 + 3\times 3 + 4 + 9)$$

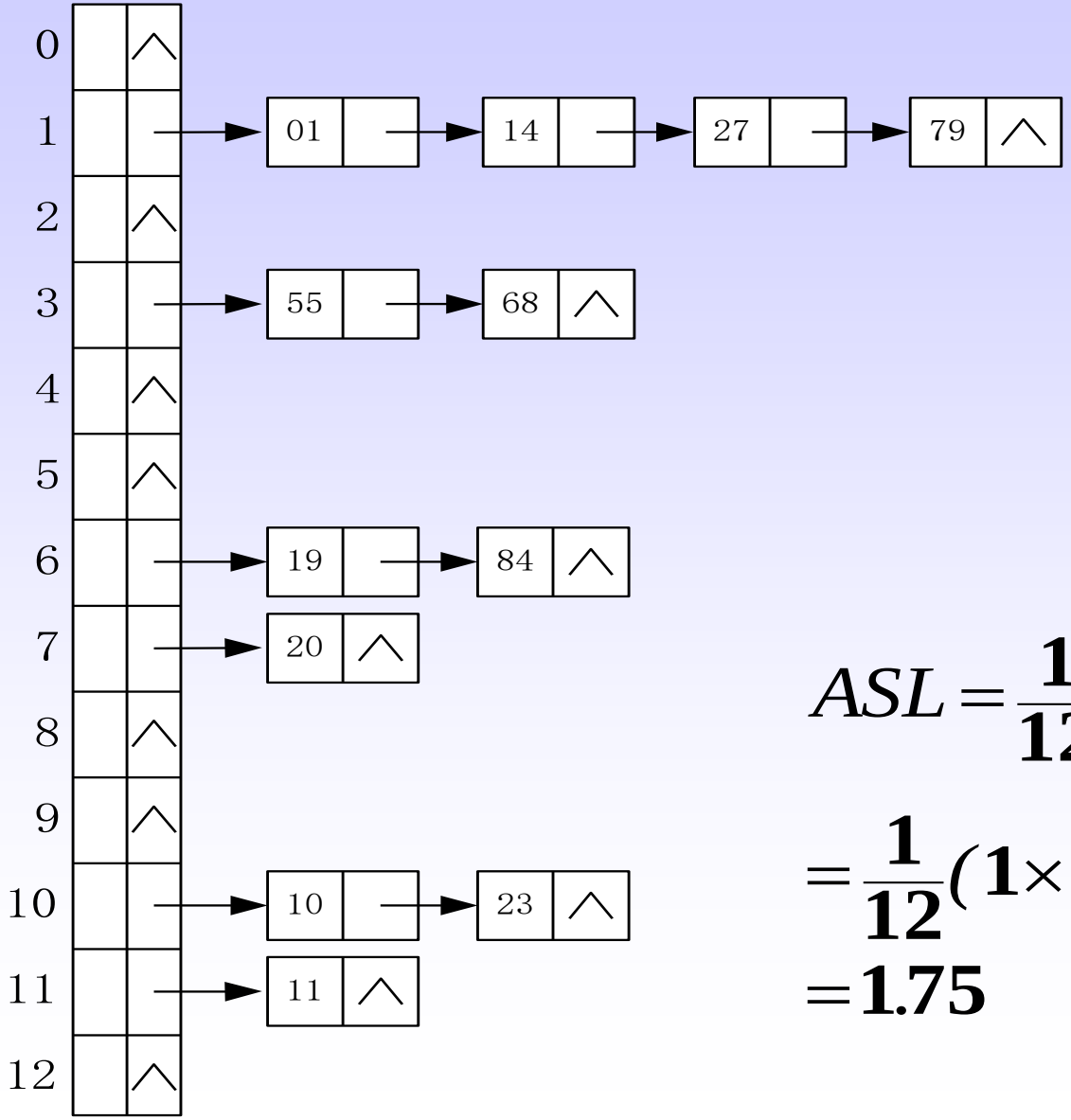
$$= 2.5$$

9.3 哈希表

2 链地址法

```
Typedef struct HNode
{       ElemType data;
        struct HNode *next;
} HNode;
HNode *SearchHash(HNode HT[], KeyType key, HNode *f)
{ //HT为每个链表的头结点组成的数组，查找不成功返回空指针；
  //否则，返回指向该结点的指针，f为其前驱的指针。
    p = HT[Hash(key)].next;    f = &HT[Hash(key)];
    while(p && p->data.key!=key)
    {  f = p; p = p->next; }
    return( p );
}
```

9.3 哈希表



$$ASL=\frac{1}{12}\sum_{i=1}^{12}Ci$$

$$=\frac{1}{12}(1\times6+2\times4+3+4)$$

$$=1.75$$

9.3 哈希表

```
Status InsertHash( HNode HT[], ElemType e )
{  p = SearchHash( HT, e.key, f );
   if( !p )
   {  s = ( HNode * )malloc(sizeof(HNode));
      s->data = e;  s->next = f->next;   f->next = s;  return( OK );
   }
   return( ERROR );
}
Status DeleteHash( HNode HT[], ElemType &e )
{  p = SearchHash(HT, e.key, f);
    if( p ) {  e=p->data;  f->next=p->next;  free(p); return(OK); }
    return( ERROR );
}
```

9.3 哈希表

哈希表分析

- 哈希表是一种直接计算记录存放地址的方法，它在关键字与存储位置之间直接建立了映象。
- 当选择的哈希函数能够得到均匀的地址分布时，在查找过程中可以不做多次探查。
- 由于很难避免冲突，就增加了查找时间。冲突的出现与哈希函数的选取 (地址分布是否均匀)、处理溢出的方法 (是否产生堆积)、 装填因子α（关键字的个数与存储位置的个数之比）的大小有关。

9.3 哈希表

$\alpha = n / m$	0.50		0.75		0.90		0.95	
散列函数种类	链地址法	开地址法	链地址法	开地址法	链地址法	开地址法	链地址法	开地址法
平方取中	1.26	1.73	1.40	9.75	1.45	310.14	1.47	310.53
除留余数	1.19	4.52	1.31	10.20	1.38	22.42	1.41	25.79
移位折叠	1.33	21.75	1.48	65.10	1.40	710.01	1.51	118.57
分界折叠	1.39	22.97	1.57	48.70	1.55	69.63	1.51	910.56
数字分析	1.35	4.55	1.49	30.62	1.52	89.20	1.52	125.59
理论值	1.25	1.50	1.37	2.50	1.45	5.50	1.48	10.50

上图给出一些实验结果，列出在采用不同的哈希函数和不同的处理溢出的方法时，查找关键字所需的对位置访问的平均次数。

9.3 哈希表

- 从图中可以看出，链地址法优于开地址法；除留余数法作哈希函数优于其它类型的哈希函数，最差的是折叠法。
- 当装填因子 α 较高时，选择的哈希函数不同，哈希表的查找性能差别很大。在一般情况下多选用除留余数法。
- 对哈希表技术进行的实验评估表明，它具有很好的平均性能，优于一些传统的技术，如平衡树。但哈希表在最坏情况下性能很不好。如果对一 个有 n 个关键字的哈希表执行一次查找或插入操作，最坏情况下需要 $O(n)$ 的时间。

9.3 哈希表

Knuth用地址分布均匀的哈希函数*Hash*( )计算位置存储, 对不同的溢出处理方法进行了概率分析。

处理溢出的方法		平均搜索长度 *ASL*	
		搜索成功 ***Sn***	搜索不成功(登入新记录) ***Un***
开地址法	线性探查法	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
	伪随机探查法 二次探查法 双散列法	$-\left(\frac{1}{\alpha}\right)\log_e(1-\alpha)$	$\frac{1}{1-\alpha}$
链地址法(同义词子表法)		$1+\frac{\alpha}{2}$	$\alpha+e^{-\alpha}\approx\alpha$

9.3 哈希表

- S_n 是查找一个随机选择的关键字 x_i $(1 \le i \le n)$ 所需的关键字比较次数的期望值
- U_n 是在长度为m的哈希表中n个位置已装入数据元素的情况下，装入第$n+1$项所需执行的关键字比较次数期望值。
- 哈希表的装填因子 α 表明了表中的装满程度。越大，说明表越满，再插入新元素时发生冲突的可能性就越大。
- 哈希表的查找性能，即平均查找长度依赖于哈希表的装填因子，不直接依赖于 n 或 m。
- 不论关键字集合有多大，我们总能选择一个合适的装填因子，以把平均查找长度限制在一定范围内。

第十章 排序

- 10.1 概述
- 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
- 10.7 小结

10.1.1 排序概述

- **排序(Sorting)：**

将一组数据按一定的规律顺次排列起来。假定包含*n*个纪录的序列为 $\mathbf{R}=(R_1,R_2,\cdots,R_n)$，对应的*关键字*为 $K_1,K_2,\cdots,K_n$ ，则排序是确定如下一个排列 $p_1,p_2,\cdots,p_n$，使得$K_{p_1}\le K_{p2}\le\cdots\le K_{p_n}$，从而得到一个有序序列： $R_{p_1},R_{p_2},\cdots,R_{p_n}$ 。

R	R_1	R_2	R_3
Key	K_1	K_2	K_3
	30	10	20

$p_1,p_2,\cdots,p_n=2,3,1$

(R_2,R_3,R_1)

10.1 概述

- 关键字(*key*):

通常数据元素有多个属性域，其中用来区分元素的属性域即为关键字。如果各个元素的关键字互不相同，这种关键字即主关键字。按照主关键字进行排序，排序的结果是唯一的。如果有些元素的关键字可能相同，这种关键字称为次关键字。按照次关键字进行排序，排序的结果可能不唯一。

10.1.1 概述

	学 号	姓 名	数学	外语
1	20051	刘大海	80	75
2	20042	王　伟	90	83
3	20066	吴晓英	82	88
4	20038	刘　伟	80	70
5	20052	王　洋	60	70

(a) 无序表

	学 号	姓 名	数学	外语
1	20038	刘　伟	80	70
2	20042	王　伟	90	83
3	20051	刘大海	80	75
4	20052	王　洋	60	70
5	20066	吴晓英	82	88

(b) 按学号排列的有序表

	学 号	姓 名	数学	外语
1	20052	王　洋	60	70
2	20051	刘大海	80	75
3	20038	刘　伟	80	70
4	20066	吴晓英	82	88
5	20042	王　伟	90	83

(c) 按数学成绩排列的有序表

	学 号	姓 名	数学	外语	总分
1	20052	王　洋	60	70	130
2	20038	刘　伟	80	70	150
3	20051	刘大海	80	75	155
4	20066	吴晓英	82	88	170
5	20042	王　伟	90	83	173

(d) 按总分成绩排列的有序表

10.1.1 概述

次关键字

- 排序算法的稳定性：如果在序列中有两个元素r_i和r_j，它们的关键字 $k_i = k_j$，且在排序之前，元素r_i排在r_j前面。如果在排序之后，元素r_i仍在元素r_j的前面，则称这个排序方法是稳定的，否则称这个排序方法是不稳定的。例如：

(10,25,22,42,25,30,18) 稳定排序 (10,18,22,25,25,30,42)

(10,25,22,42,25,30,18) 不稳定排序 (10,18,22,25,25,30,42)

10.1.1 概述

- 排序的时间开销：

衡量排序算法好坏的最重要的标志是*时间开销*。排序的时间开销可用算法执行中的数据比较次数与数据移动次数来衡量。一般来说算法运行时间按平均情况进行估算。对于那些受元素关键字序列初始排列及元素个数影响较大的，需要按最好和最坏情况进行估算。

10.1.1 概述

- 待排序记录的数据类型

```
#define MAXSIZE 20                          //宏定义序列长度
typedef int KeyType;                        // 定义关键字为整型

typedef struct {                            //定义一个元素（记录）
        KeyType   key;                      // 关键字项
        InfoType   otherinfo;               // 其它数据项
} RedType;                                  // 记录类型

typedef struct{                             //定义一个序列
        RedType   r[MAXSIZE+1];  // r[0]为监视哨
        int          length;                // 表长
} SqList;                                   // 顺序表类型
```

10.2 插入排序

插入排序：

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

- 直接插入排序
- 折半插入排序
- 表插入排序
- 希尔排序

10.1.2 插入排序算法

10.2.1 直接插入排序 (Straight Insertion Sort)

- 基本思想：

设待排序的序列为(r[1],r[2],...,r[n]) 。

首先,将初始序列中的元素r[1]看作有序子序列。

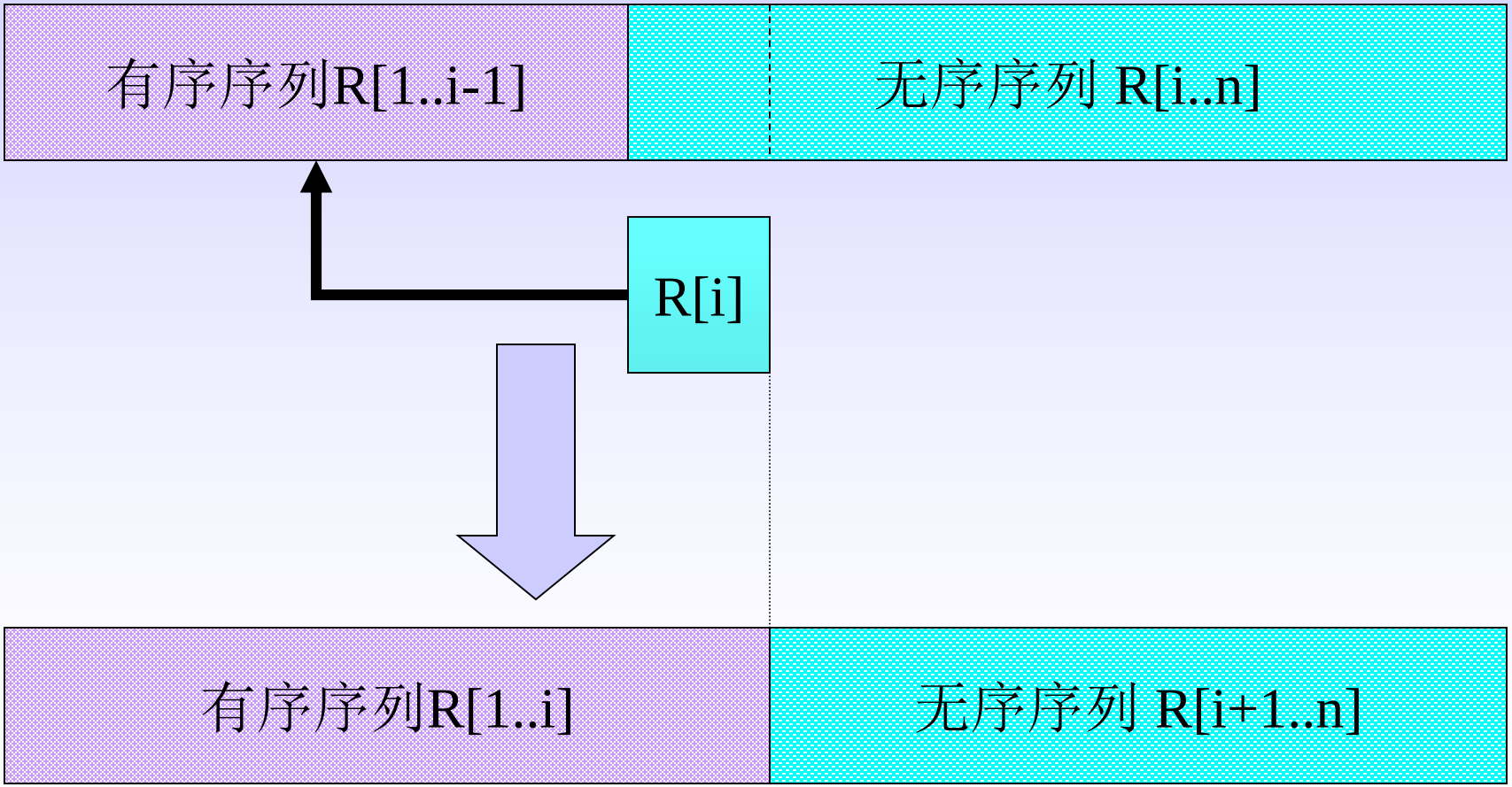
第1遍：将r[2]插入有序子序列中：若r[2].key<r[1].key，则r[2]插在 r[1]之前；否则，插在r[1]的后面。

第2遍：将元素r[3]插入前面已有2个元素的有序子序列中，得到3个元素的有序子序列。

以此类推，依次插入r[4],...,r[n],最后得到完整的有序序列。

10.2 插入排序

- 一趟直接插入排序的基本思想：



10.2 插入排序

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j+1；

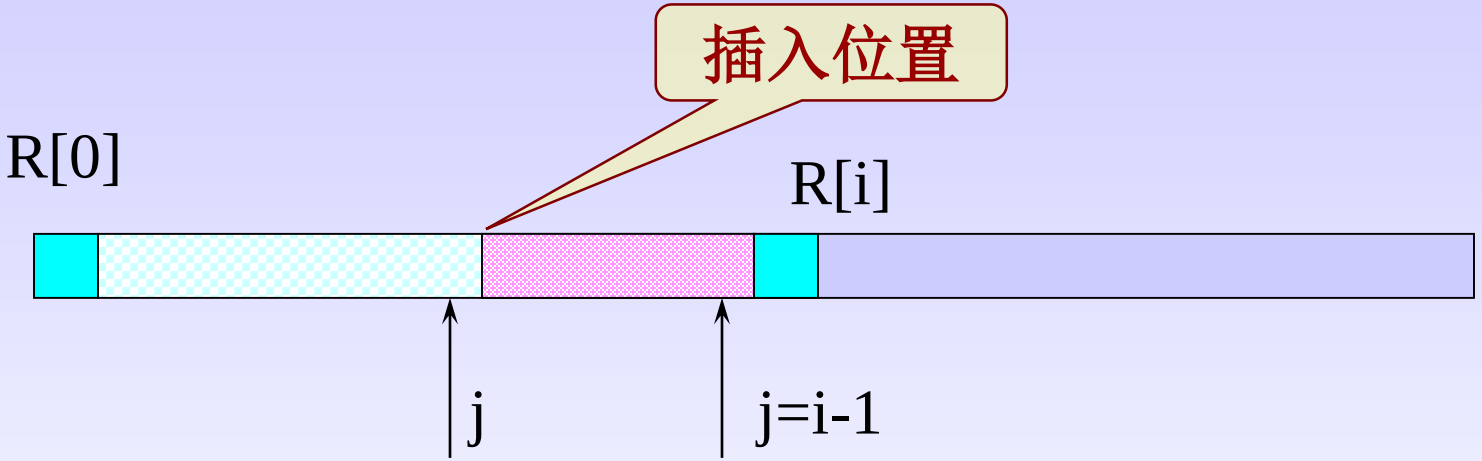
   $R[1..j].key \le R[i].key < R[j+1..i-1].key$

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。

10.2 插入排序

从R[i-1]起向前进行顺序查找，监视哨设置在R[0]；



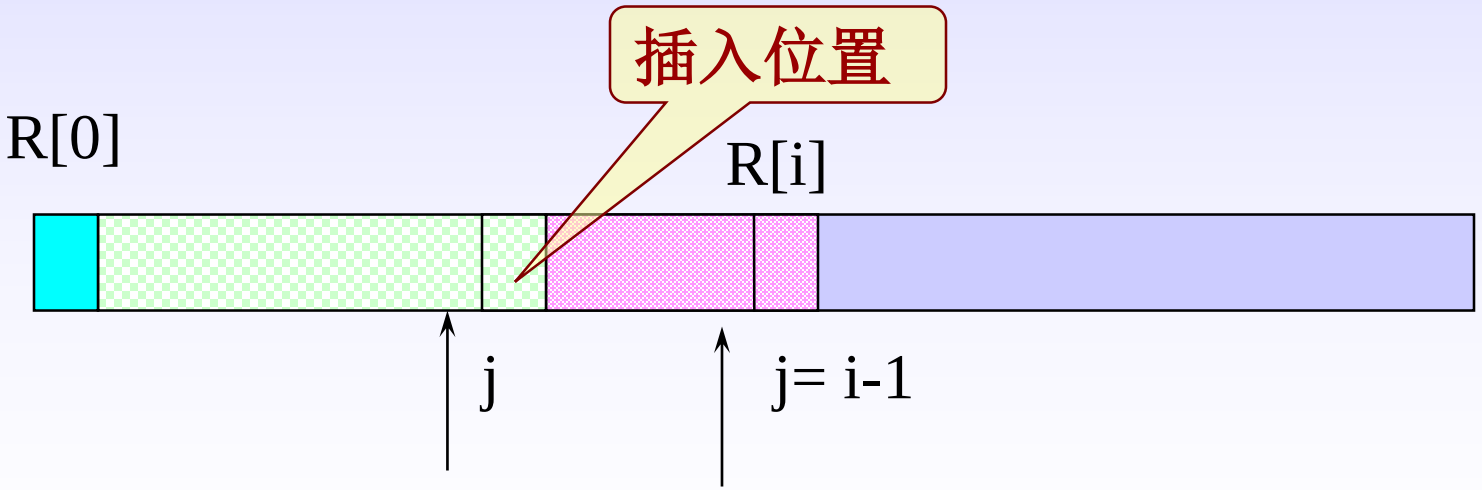

```
R[0] = R[i];            // 设置“哨兵”

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*

10.2 插入排序

对于在查找过程中找到的那些关键字大于**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”，**R[j+1] = R[0];**

10.2 插入排序

- 直接插入排序算法

```
void InsertSort(SqList &L) // 对顺序表L[1..n]作插入排序
{  for (i=2；i<=L.length；i++)   //操作第i个记录
   {  L.r[0]=L.r[i];          //待插记录L.r[i]存入监视哨中
      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]；       //插入记录r[0],即原r[i]
   }
}
```

r[0]的作用：

其一是在进入查找循环之前保存R[i]的副本；

其二是在循环中监视j是否越界（监视哨）。

10.2 插入排序

- 算法的实现

算法的基本思想

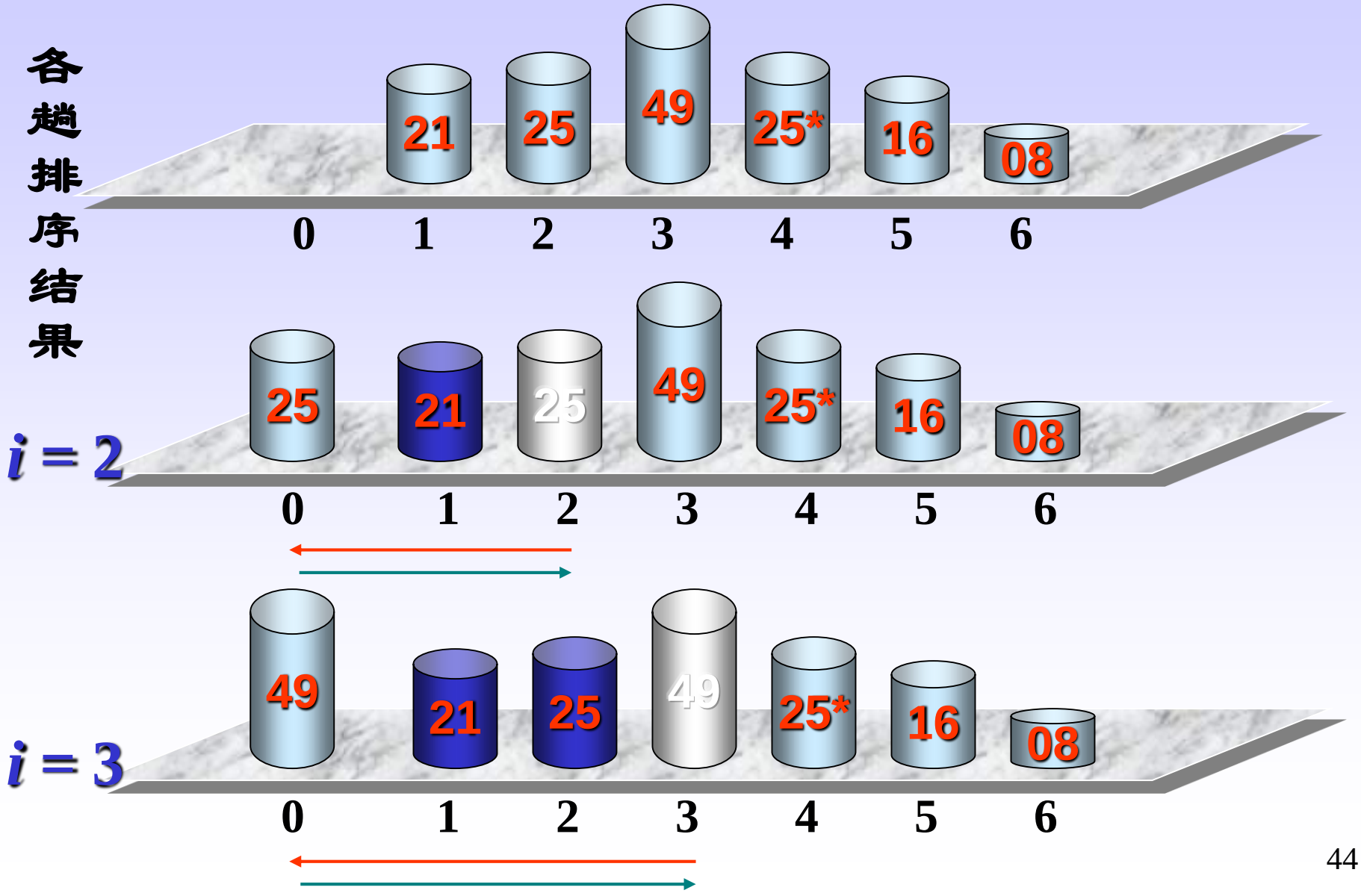
1、插入位置的查找

2、插入操作的实现

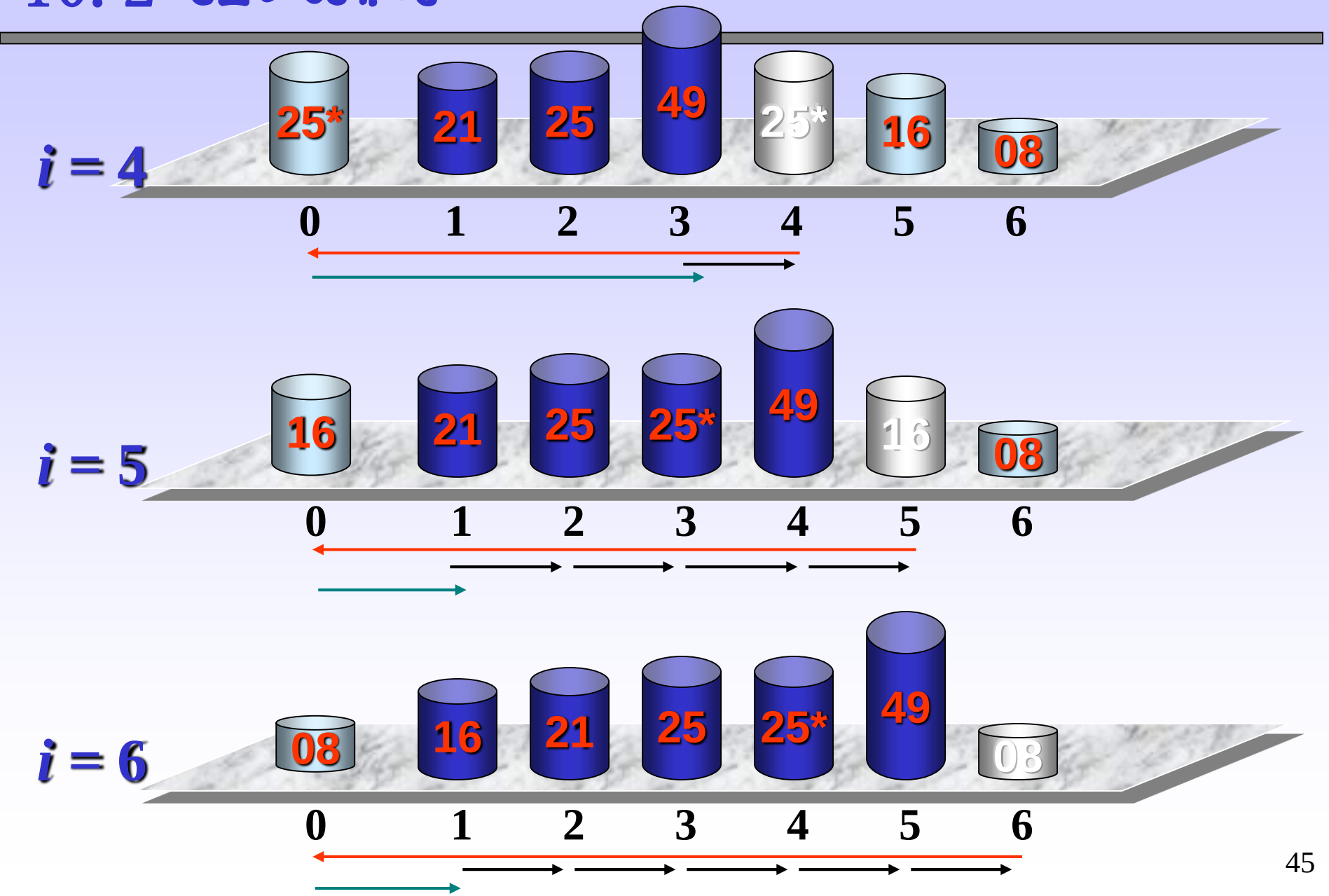
3、查找的同时进行移动

4、查找的顺序，哨兵设计的利用（边界）

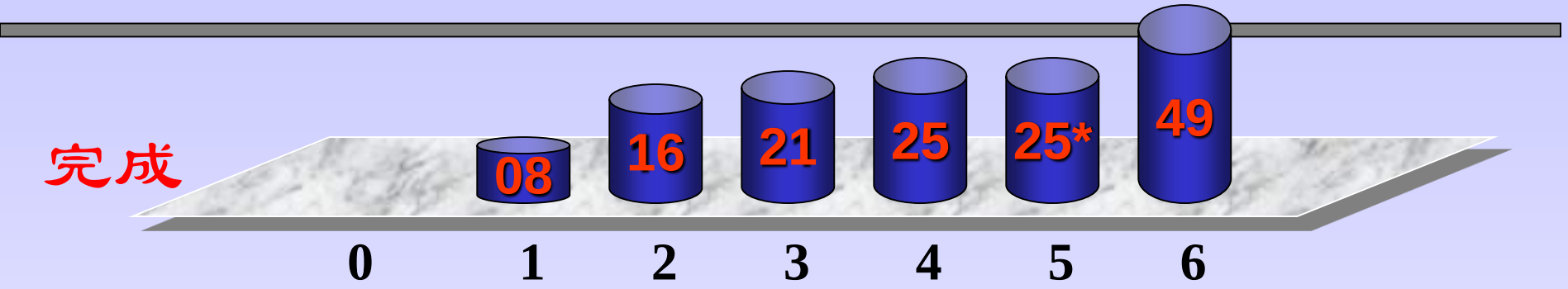
10.2 插入排序


各趟排序结果
21
25
49
25*
16
08
0 1 2 3 4 5 6
i = 2
25
21
25
49
25*
16
08
0 1 2 3 4 5 6
i = 3
49
21
25
49
25*
16
08
0 1 2 3 4 5 6

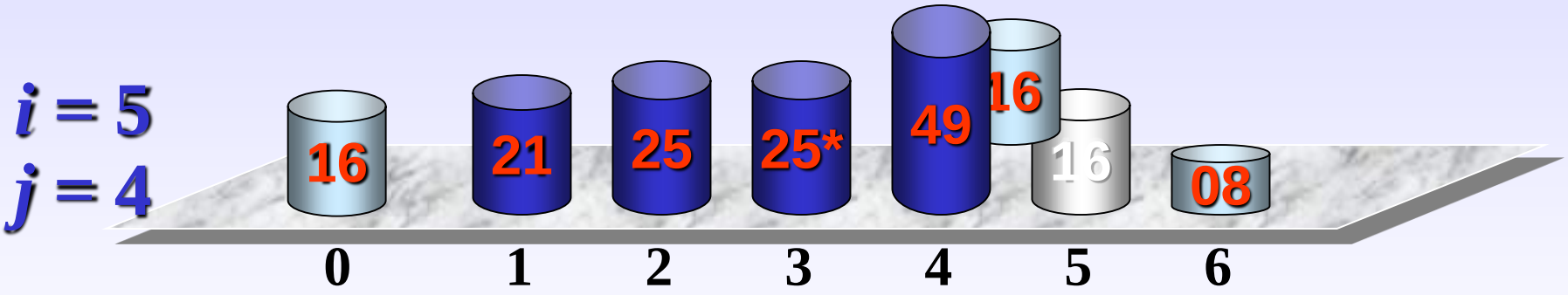
10.2 插入排序

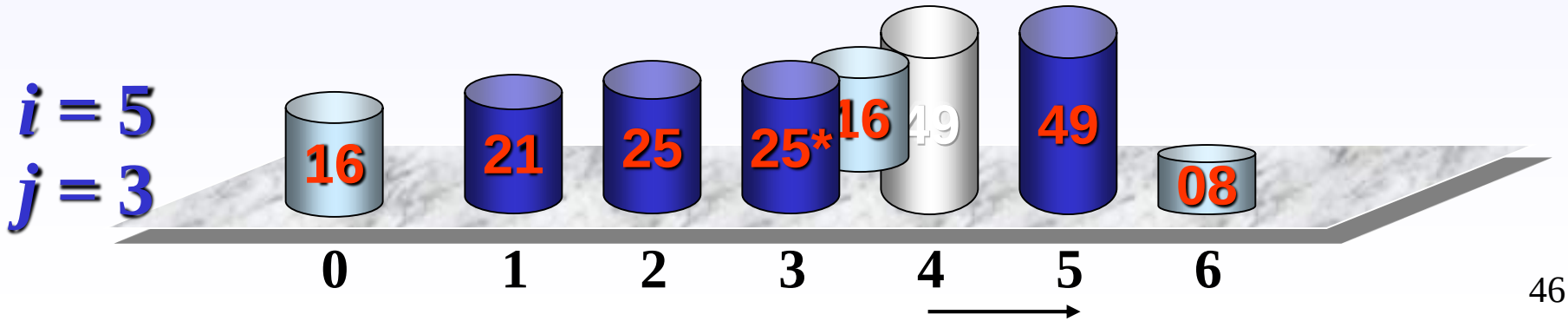

i = 4
25*
21
25
49
25*
16
08
0 1 2 3 4 5 6
i = 5
16
21
25
25*
49
16
08
0 1 2 3 4 5 6
i = 6
08
16
21
25
25*
49
08
0 1 2 3 4 5 6

10.2 插入排序

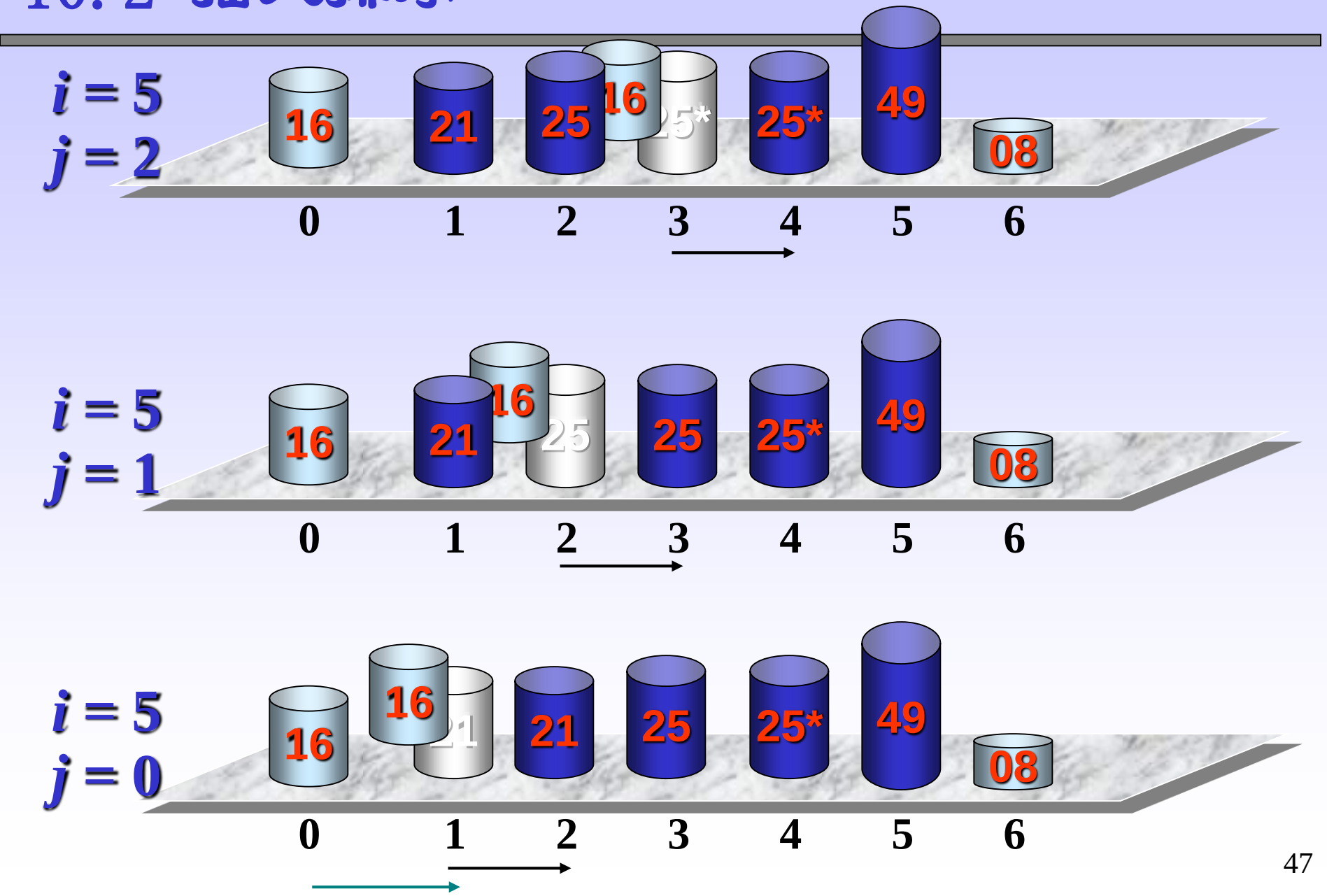


$i=5$时的排序过程





10.2 插入排序


i = 5
j = 2
16
21
25
16
25*
25*
49
08
0
1
2
3
4
5
6
i = 5
j = 1
16
21
16
25
25
25*
49
08
0
1
2
3
4
5
6
i = 5
j = 0
16
16
21
21
25
25*
49
08
0
1
2
3
4
5
6

10.2 插入排序

- 算法分析
- 关键字比较次数和元素移动次数与元素的初始排列有关。
  - *最好情况*：排序前元素已经按关键字递增有序，每趟只需与前面的有序序列的最后一个关键字比较 1 次，移动 2 次元素，总的比较次数为 n-1，移动次数为 2(n-1)。
  - *最坏情况*：排序前元素已经按关键字递减有序，第 i 趟时第 i 个元素必须与前面 i个关键字比较，且每次比较都要做数据移动。则总的比较次数和移动次数分别为：

$$\text{比较次数}=\sum_{i=2}^{n} i=\frac{(n+2)(n-1)}{2},\quad \text{移动次数}=\sum_{i=2}^{n}(i+1+2)=\frac{(n+4)(n-1)}{2}$$

10.2 插入排序

- 在平均情况下的关键字比较次数和元素移动次数约为 $n^2/4$。因此，直接插入排序的时间复杂度为 $o(n^2)$。

  平均比较次数：$$\sum_{i=2}^{n}\sum_{j=1}^{i}\frac{1}{i}j \approx n^2/4$$

- 在n 较小的时候，该算法性能较好，但是n很大的时候该算法会带来很大的计算复杂度。
- 直接插入排序是一种稳定的排序方法。

10.2 插入排序

10.2.2 折半插入排序

- 基本思想：设待排序的序列为(r[1],r[2],...,r[n])。

首先,将初始序列中的元素r[1]看作有序子序列。

第1遍：若r[2].key<r[1].key， 则r[2]插在 r[1]之前；否则，插在r[1]的后面。

第2遍：将元素r[3]插入前面已有2个元素的有序子序列中，得到3个元素的有序子序列。

以此类推，依次插入r[4],...,r[n],最后得到n个元素的递增有序序列。在插入r[*i*]时，利用折半搜索法寻找r[*i*]的插入位置。

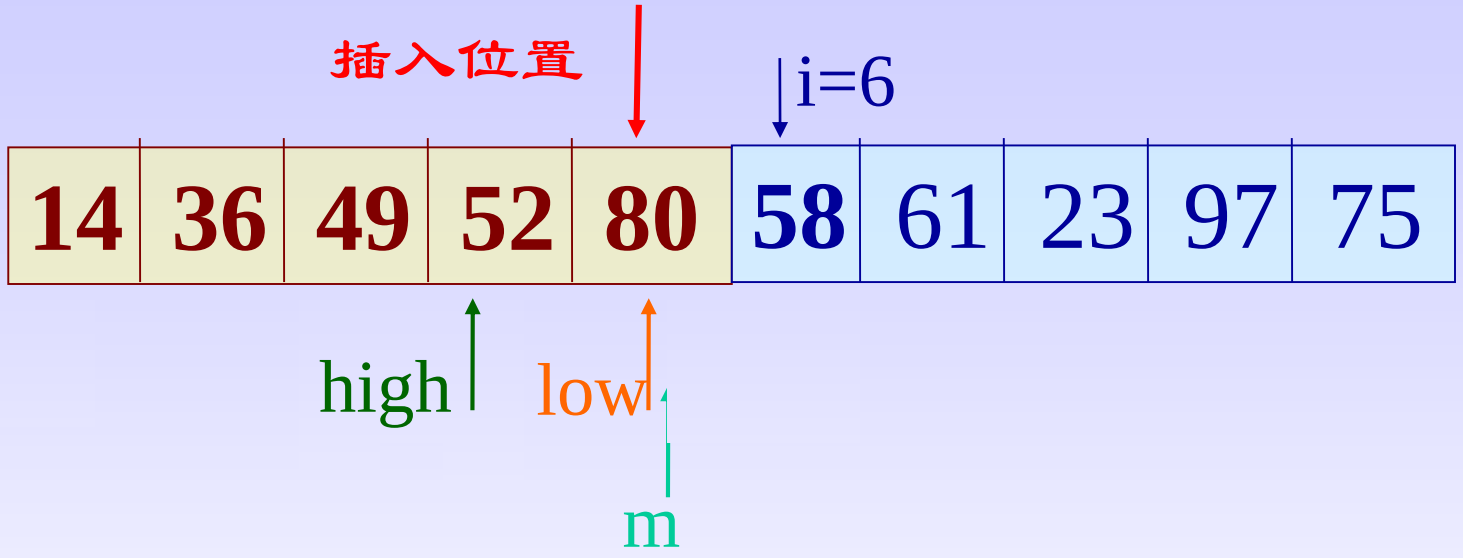
10.2 插入排序

折半插入排序算法

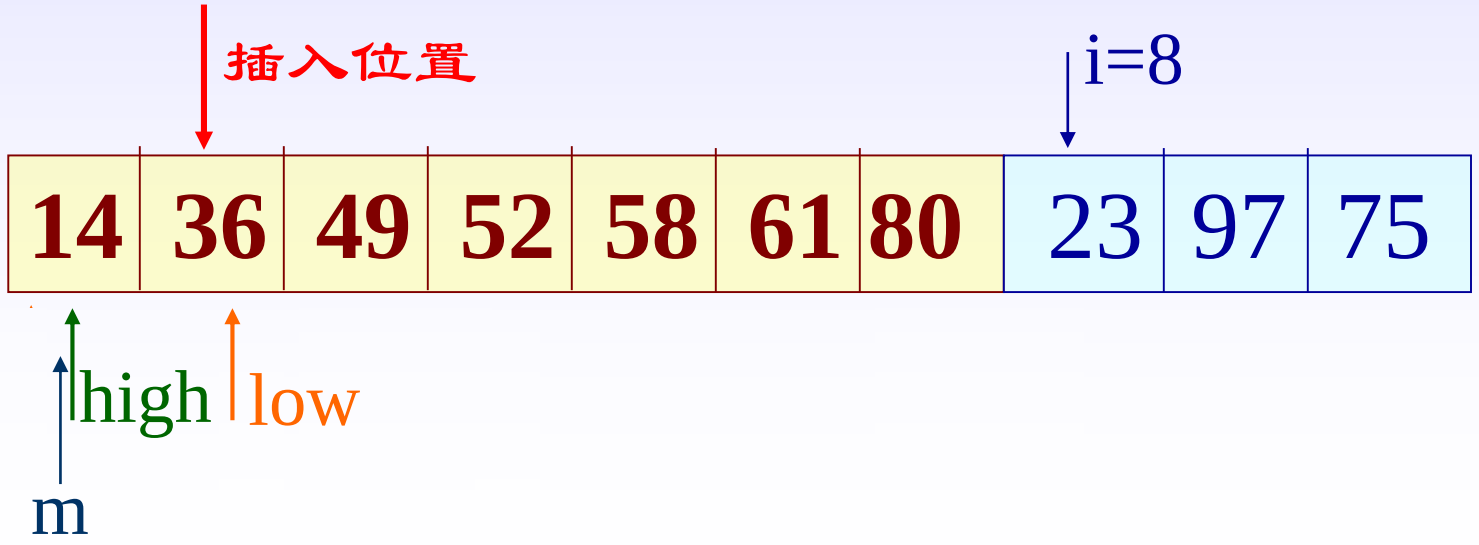
```
void BInsertSort(SqList &L) // 对顺序表L[1..n]作折半插入排序
{  for (i=2；i<=L.length；i++)
   { L.r[0]=L.r[i]；            //待插记录L.r[i]存入监视哨中
     low=1; high=i-1;      // 查找插入位置
     while(low<=high)
     {  mid=(low+high)/2;
        if(L.r[0].key<L.r[mid].key)  high=mid-1;
        else  low=mid+1;
     }
     for(j=i-1; j>=high+1; j--) L.r[j+1]=L.r[j];       //记录后移
     L.r[high+1]=L.r[0];        //插入记录r[0],即原r[i]
   }
}
```

10.2 插入排序

例如:



再如:



10.2 插入排序

- 算法分析

- 对于同一个序列，折半插入排序和直接插入排序具有相同的元素移动次数，但是具有较低的平均比较次数，为：

$$\sum_{i=1}^{n-1}(\log_2 i+1)$$

元素比较的复杂度为**O(nlog$_2$n)**；

- 算法总的复杂度仍为 $O(n^2)$。

10.2 插入排序

10.2.3 表插入排序

- 基本思想：

直接插入和折半插入排序具有相同且较多的记录移动次数。

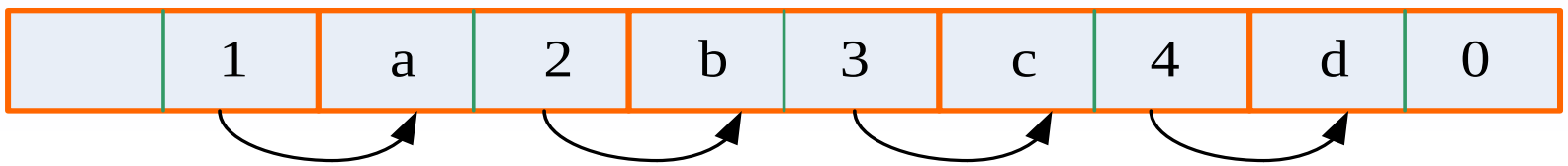
为了减少在排序过程中进行的“移动”记录的操作，必须改变排序过程中采用的存储结构。利用静态链表进行排序，并在排序完成之后，一次性地调整各个记录相互之间的位置，即将每个记录都调整到它们所应该在的位置上。

采用静态链表来存储记录！

10.2 插入排序

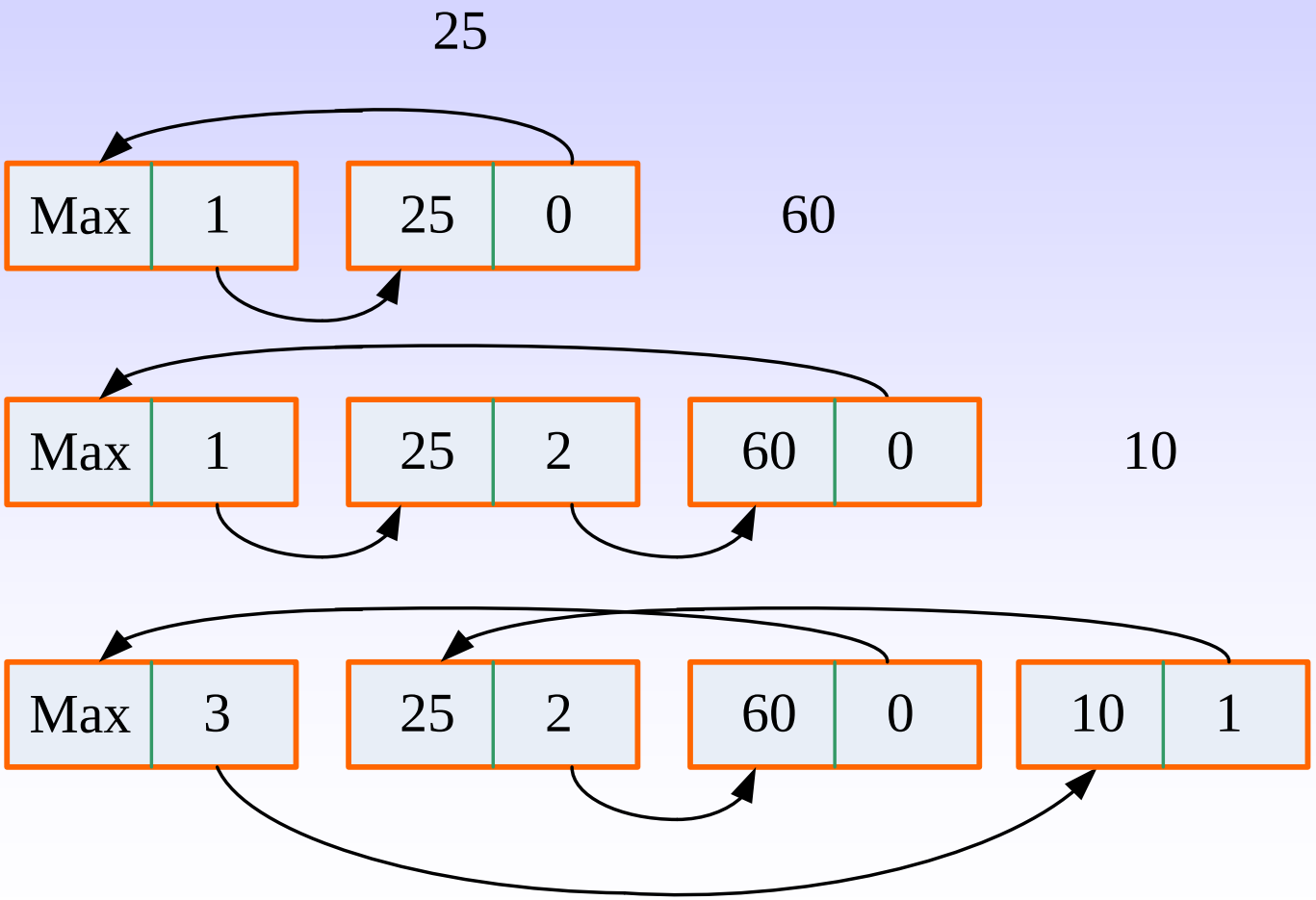
- 待排序记录的数据类型

```
#define MAXSIZE 20
typedef struct{
        RedType rc;                    //记录项
        int     next;                  //指针项
} SLNode;                              //表节点类型

typedef struct{
        SLNode  r[MAXSIZE+1];  // r[0]为监视哨
        int     length;        // 表长
} SqList;                              // 顺序表类型-静态链表
```



10.2 插入排序

- 表插入中的记录比较



10.2 插入排序

- 记录重排

```
void Arrange ( Elem &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 ) {
        while (p < i)  p = SL.r[p].next;
        q = SL.r[p].next;     // q指示尚未调整的表首
        if ( p! = i ) {
            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
        }
        p = q;    // p指示尚未调整的表首
            // 为找第i+1个记录作准备
    }
} // Arrange
```

参考教材
图10.3！

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

10.2 插入排序

10.2.4 希尔排序

- 直接插入排序中，当元素的关键字基本有序时，只需要较少的比较和移动，就可以完成排序。即此时直接插入排序的速度较快。
- 序列长度较小时，插入排序的效率也比较高
- 一开始，分成很多短的子序列分别进行排序，随着序列有序性的增强，逐渐加大序列长度，最后进行整序列排序。

10.2 插入排序

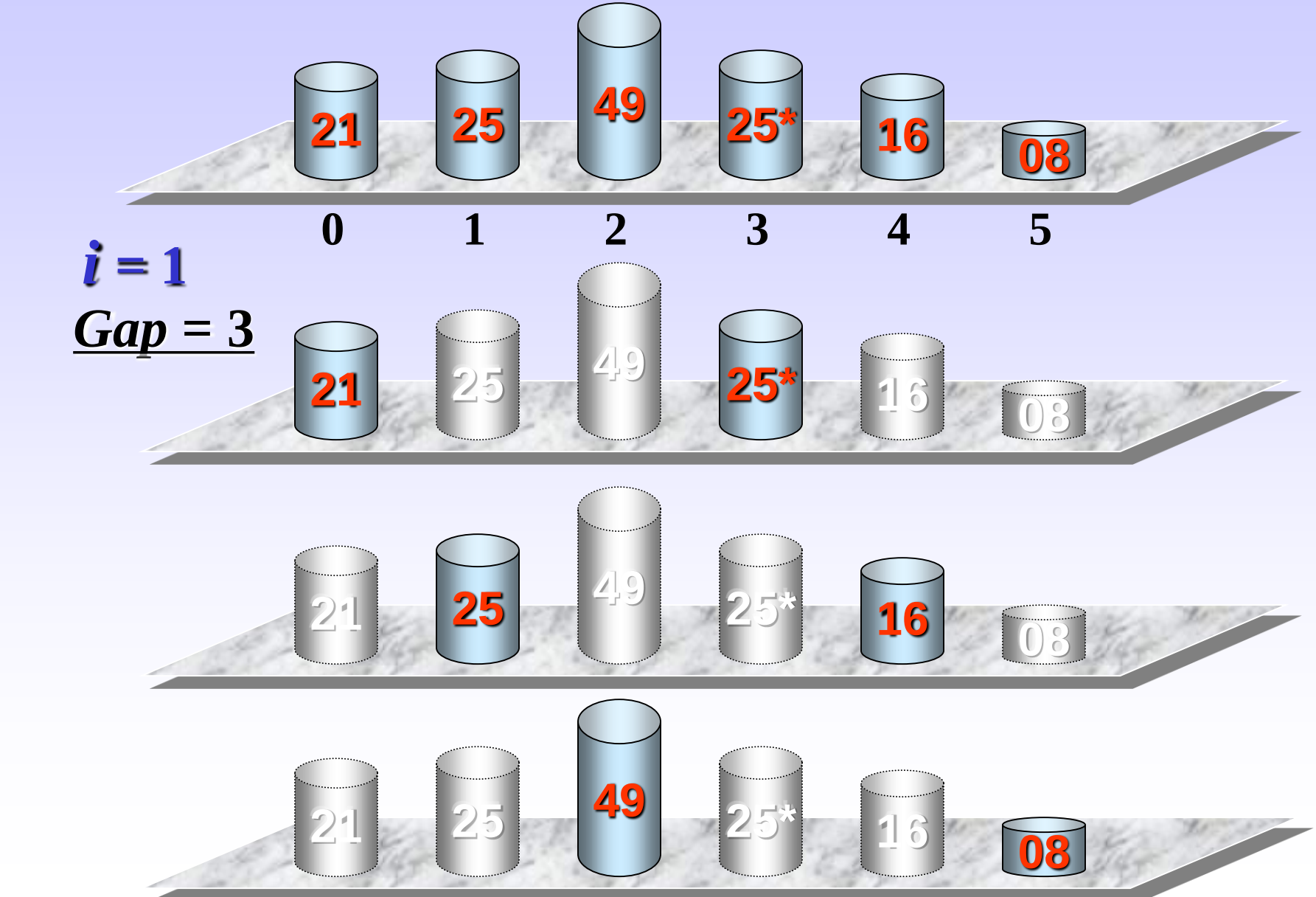
- **基本思想：** 设待排序元素序列有 n 个元素，

1) 首先取一个整数 $gap < n$ 作为间隔，将全部元素分为 gap 个子序列，所有距离为 gap 的元素放在同一个子序列中，在每一个子序列中分别做直接插入排序。

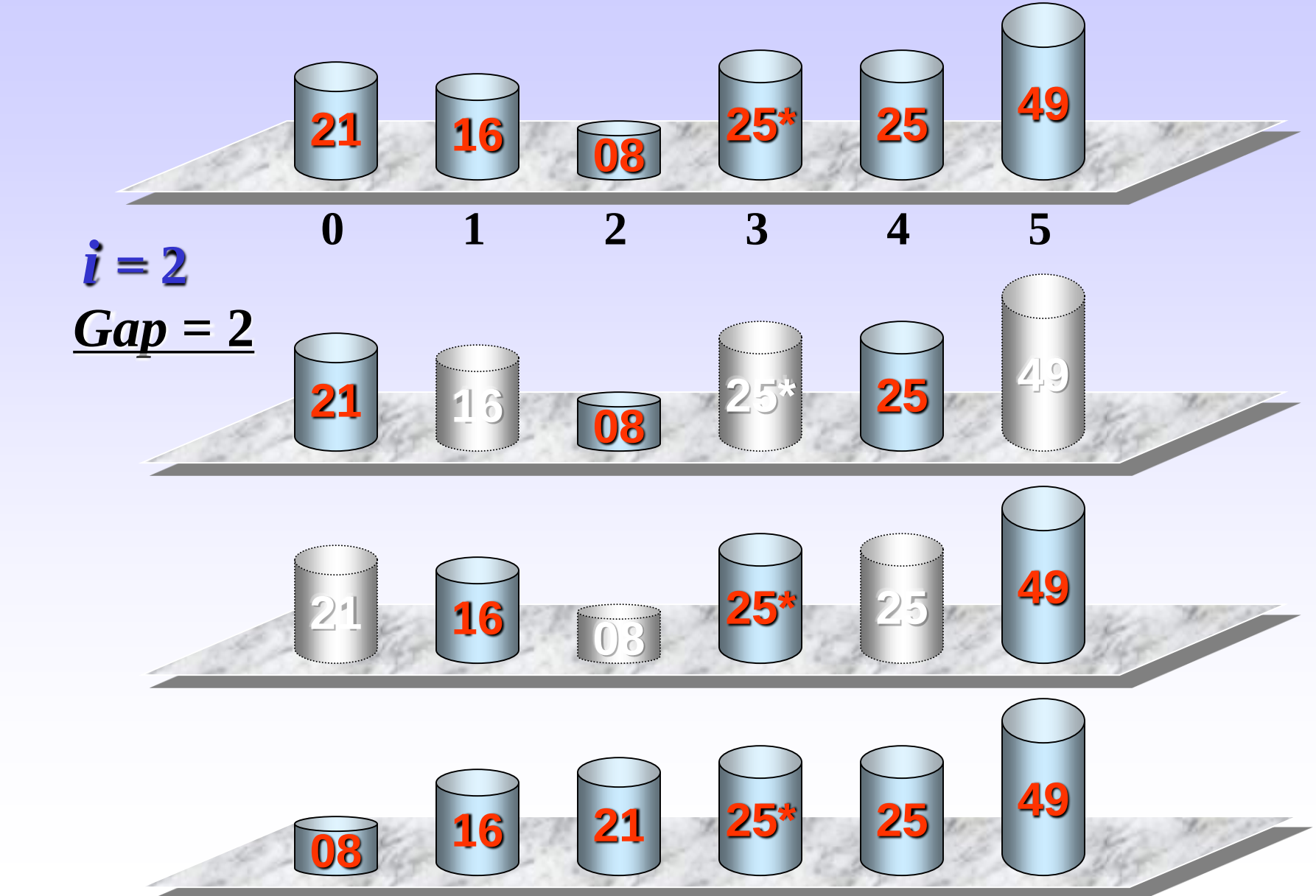
2) 然后缩小间隔 gap，如取$gap=\lceil gap/2 \rceil$，重复上述的子序列划分和排序工作。

3) 直到最后取$gap=1$，将所有元素放在同一个序列中排序为止。

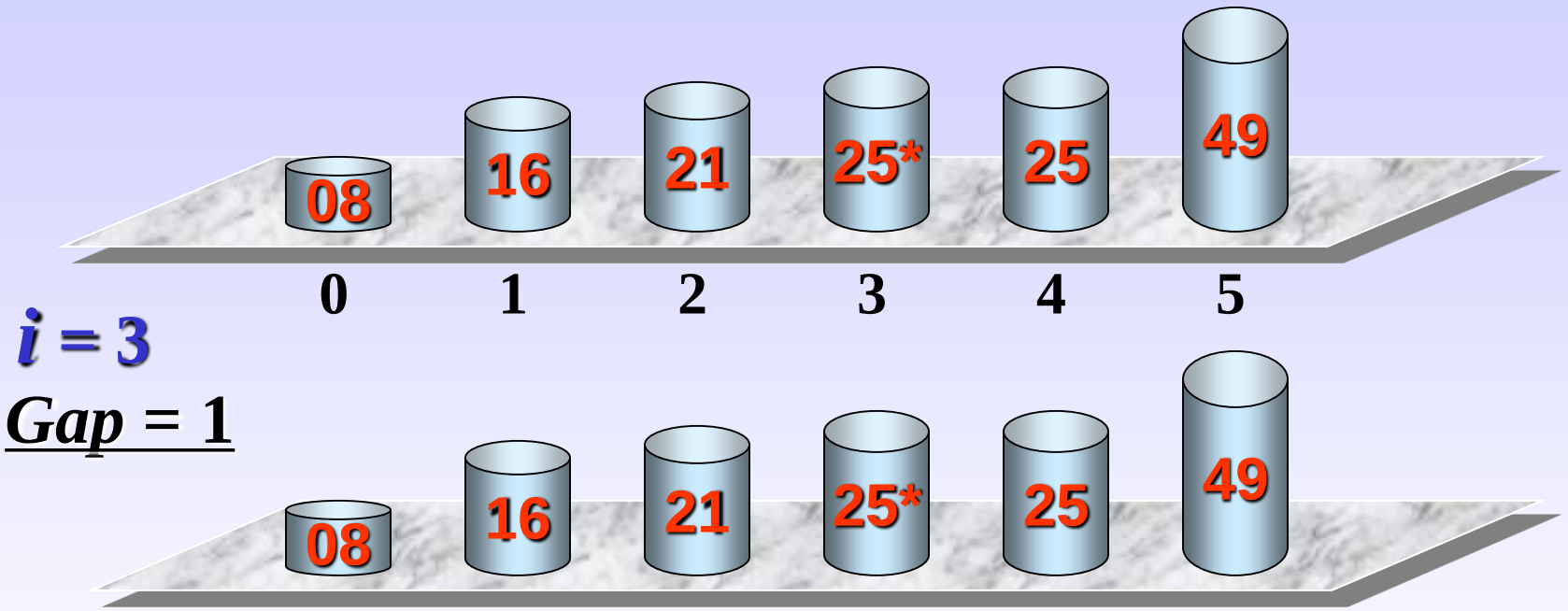
10.2 插入排序


21
25
49
25*
16
08
0
1
2
3
4
5
i = 1
Gap = 3
21
25
49
25*
16
08
21
25
49
25*
16
08
21
25
49
25*
16
08

10.2 插入排序


21
16
08
25*
25
49
0
1
2
3
4
5
i = 2
Gap = 2
21
16
08
25*
25
49
21
16
08
25*
25
49
08
16
21
25*
25
49

10.2 插入排序



10.2 插入排序

- 希尔排序算法

```
Void  ShellSort( SqList &L )
{  for( gap=L.length/2; gap>=1; gap/=2 )
   {   for( i=gap+1; i<=L.length; i++ )
       {   L.r[0]=L.r[i];
           for( j=i-gap; j>0&&L.r[0].key<L.r[j].key; j-=gap )
               L.r[j+gap]=L.r[j];
           L.r[j+gap]=L.r[0];
       }
   }
}
```

10.2 希尔排序的实现

- 子序列的划分，变步长如何实现的
- 每个子序列的直接插入排序是如何实现的
- 两者如何结合在一起

10.2 插入排序

- 希尔排序算法

```
Void  ShellSort( SqList &L )
{  for( gap=L.length/2; gap>=1; gap/=2 )
   {   for( i=gap+1; i<=L.length; i++ )
       {   L.r[0]=L.r[i];
           for( j=i-gap; j>0&&L.r[0].key<L.r[j].key; j-=gap )
               L.r[j+gap]=L.r[j];
           L.r[j+gap]=L.r[0];
       }
   }
}
```

10.2 插入排序

算法分析

- **Knuth经过大量实验统计得出，当*n*很大时，关键字平均比较次数和元素平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内。这是在利用直接插入排序作为子序列排序方法的情况下得到的。**
- **希尔排序是一种不稳定的排序方法。如元素序列：3，2，2*，5 排序后为：2*，2，3，5。**