第九章 排序

- 概述
- 插入排序
- 交换排序
- 选择排序
- 归并排序
- 基数排序
- 外排序
- 小结

概述

- 排序：将一组杂乱无章的数据按一定的规律顺次排列起来。
- 数据表(*datalist*)：它是待排序数据对象的有限集合。
- 关键码(*key*)：通常数据对象有多个属性域，即多个数据成员组成，其中有一个属性域可用来区分对象，作为排序依据。该域即为关键码。每个数据表用哪个属性域作为关键码，要视具体的应用需要而定。即使是同一个表，在解决不同问题的场合也可能取不同的域做关键码。

- 主关键码：如果在数据表中各个对象的关键码互不相同，这种关键码即主关键码。按照主关键码进行排序，排序的结果是唯一的。
- 次关键码：数据表中有些对象的关键码可能相同，这种关键码称为次关键码。按照次关键码进行排序，排序的结果可能不唯一。
- 排序算法的稳定性：如果在对象序列中有两个对象$r_{[i]}$和$r_{[j]}$，它们的关键码$k_{[i]} == k_{[j]}$，且在排序之前，对象$r_{[i]}$排在$r_{[j]}$前面。如果在排序之后，对象$r_{[i]}$仍在对象$r_{[j]}$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。

- 内排序与外排序：内排序是指在排序期间数据对象全部存放在内存的排序；外排序是指在排序期间全部对象个数太多，不能同时存放在内存，必须根据排序过程的要求，不断在内、外存之间移动的排序。
- 排序的时间开销：排序的时间开销是衡量算法好坏的最重要的标志。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。各节给出算法运行时间代价的大略估算一般都按平均情况进行估算。对于那些受对象关键码序列初始排列及对象个数影响较大的，需要按最好情况和最坏情况进行估算。

- 静态排序：排序的过程是对数据对象本身进行物理地重排，经过比较和判断，将对象移到合适的位置。这时，数据对象一般都存放在一个顺序的表中。
- 动态排序：给每个对象增加一个链接指针，在排序的过程中不移动对象或传送数据，仅通过修改链接指针来改变对象之间的逻辑顺序，从而达到排序的目的。
- 算法执行时所需的附加存储：评价算法好坏的另一标准。
- 静态排序过程中所用到的数据表类定义，体现了抽象数据类型的思想。

待排序数据表的类定义

```
#ifndef DATALIST_H
#define DATALIST_H
#include <iostream.h>
const int DefaultSize = 100;

template <class Type> class datalist {

template <class Type> class Element {
private:
    Type key;                  //关键码
    field otherdata;           //其它数据成员
public:
    Element ( ) : key(0), otherdata(NULL) { }
```

```
Type getKey ( ) { return key; }          //提取关键码
void setKey ( const Type x ) { key = x; }    //修改
Element<Type> & operator =                   //赋值
    ( Element<Type> & x ) { this = x; }
int operator == ( Type & x )             //判this == x
    { return ! ( this < x || x < this ); }
int operator != ( Type & x )             //判this != x
    { return this < x || x < this; }
int operator <= ( Type & x )             //判this ≤ x
    { return ! ( this > x ); }
int operator >= ( Type & x )             //判this ≥ x
    { return ! ( this < x ); }
int operator < ( Type & x )              //判this < x
    { return this > x; }
```

```
}

template <class Type> class datalist {
public:
    datalist ( int MaxSz = DefaultSize ) :    //构造函数
        MaxSize ( Maxsz ), CurrentSize (0)
        { Vector = new Element <Type> [MaxSz]; }
    void swap ( Element <Type> & x,           //对换
                Element <Type> & y )
        { Element<Type> temp = x;  x = y;  y = temp; }
private:
    Element <Type> * Vector;          //存储向量
    int MaxSize, CurrentSize;         //最大与当前个数
}
```

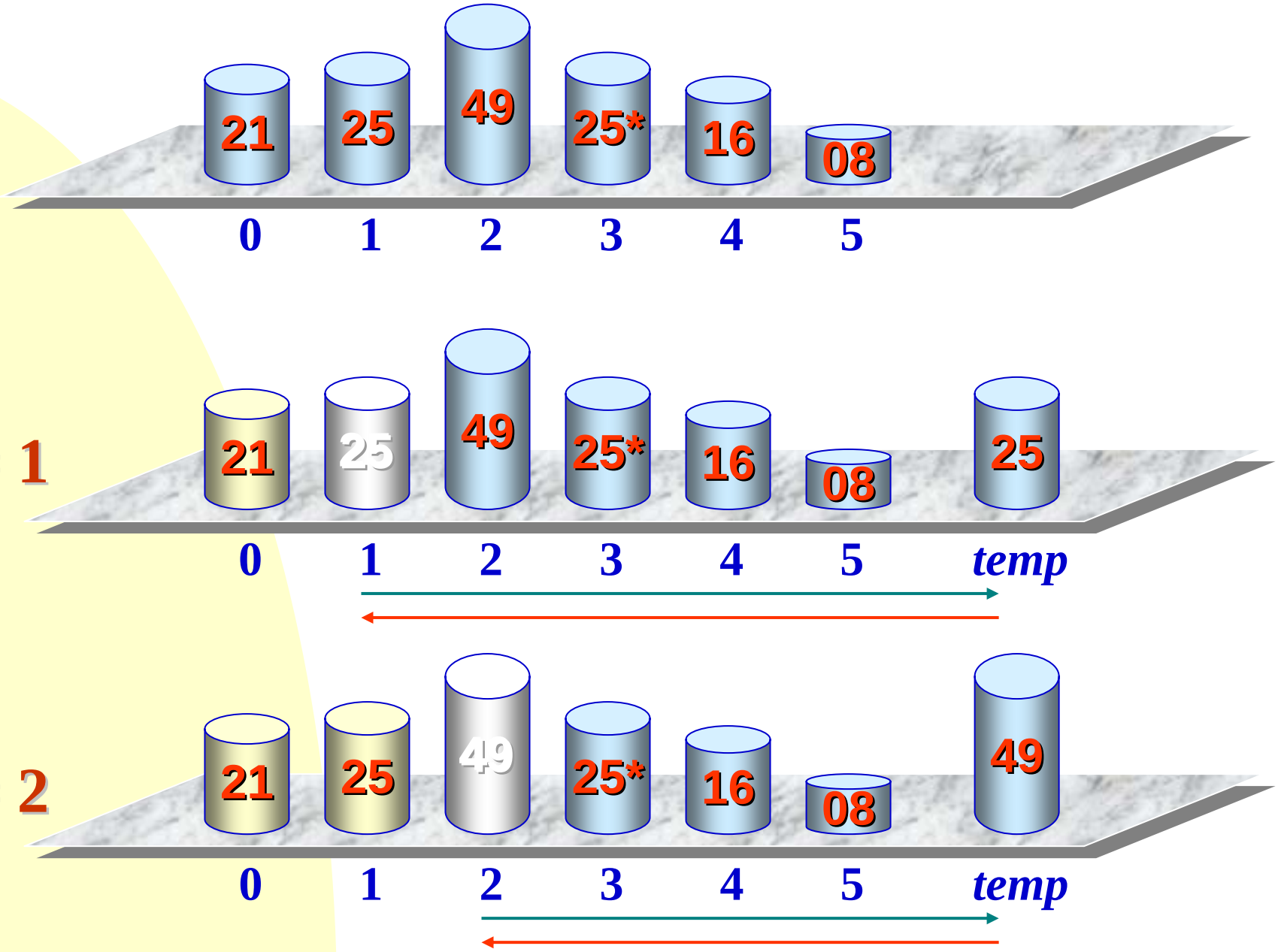
插入排序 (Insert Sorting)

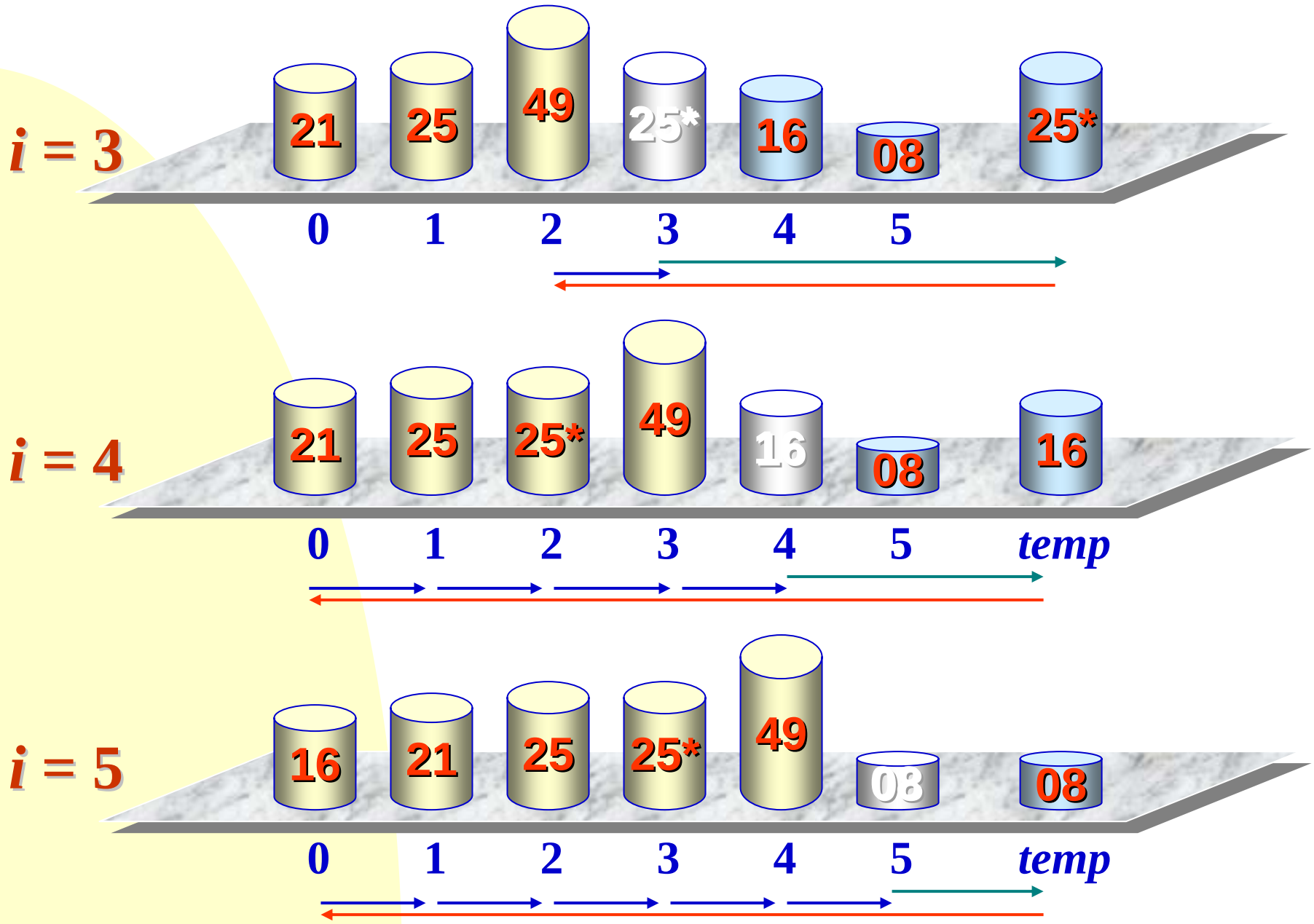
插入排序的基本方法是：每步将一个待排序的对象，按其关键码大小，插入到前面已经排好序的一组对象的适当位置上，直到对象全部插入为止。

直接插入排序 (Insert Sort)

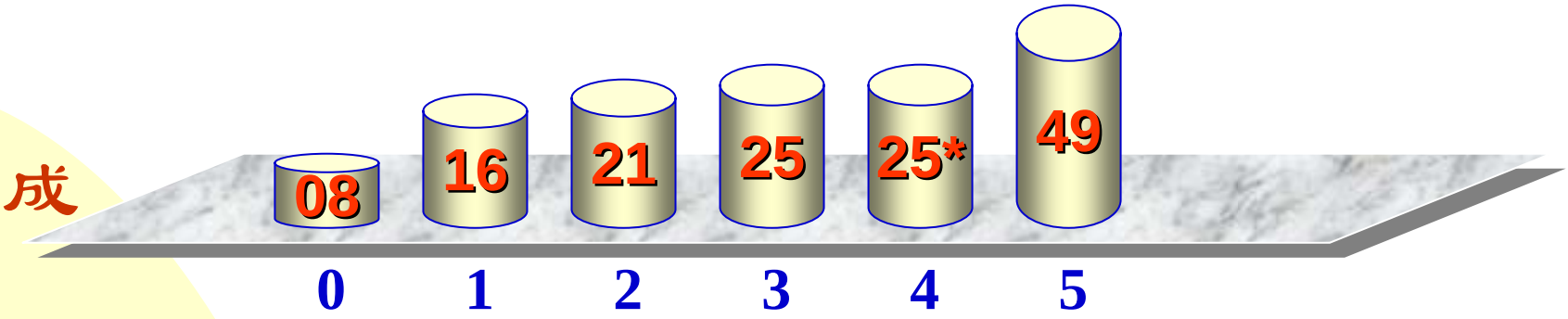
- 直接插入排序的基本思想是：当插入第i ($i \geq 1$)个对象时，前面的$V[0], V[1], \ldots, v[i-1]$已经排好序。这时，用$v[i]$的关键码与$v[i-1], v[i-2], \ldots$的关键码顺序进行比较，找到插入位置即将$v[i]$插入，原来位置上的对象向后顺移。

各趟排序结果


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

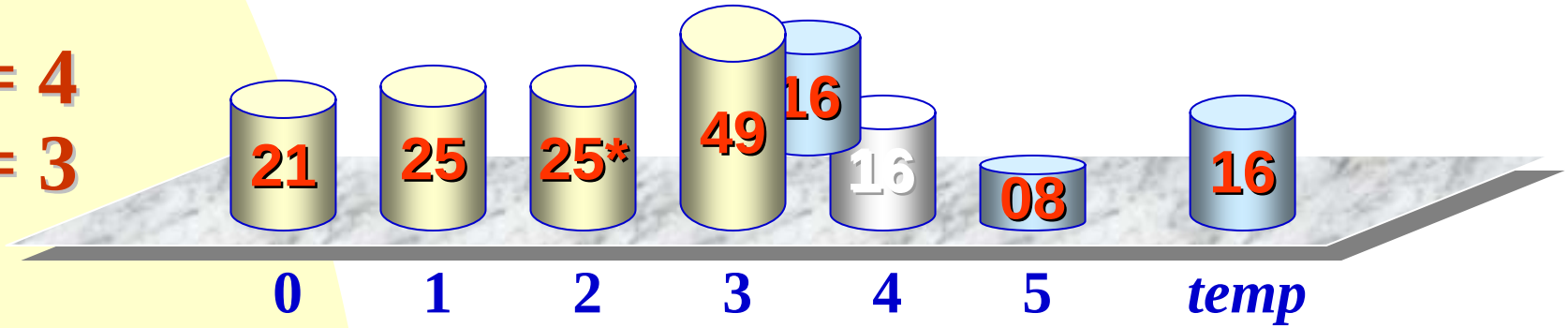

i = 3
21
25
49
25*
16
08
25*
0
1
2
3
4
5
i = 4
21
25
25*
49
16
08
16
0
1
2
3
4
5
temp
i = 5
16
21
25
25*
49
08
08
0
1
2
3
4
5
temp

完成

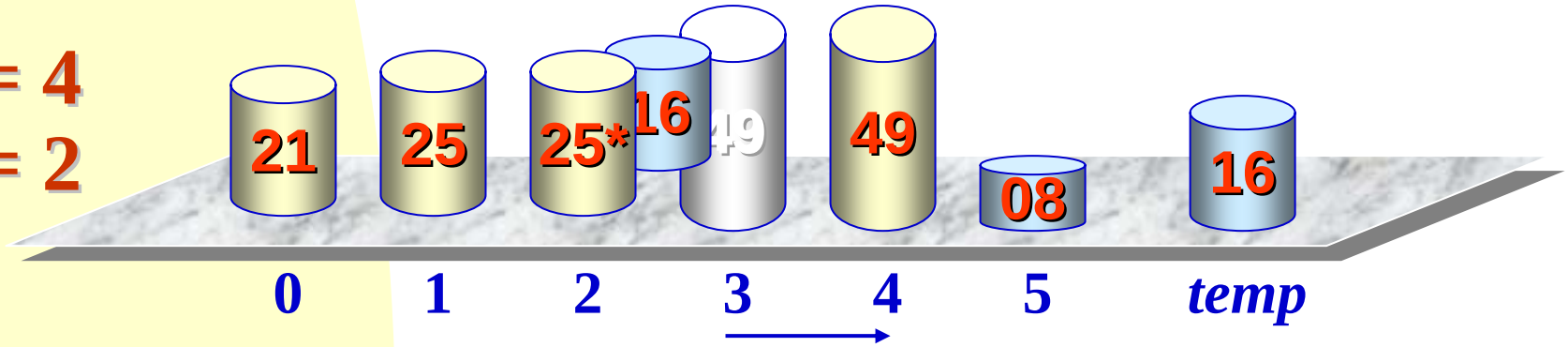


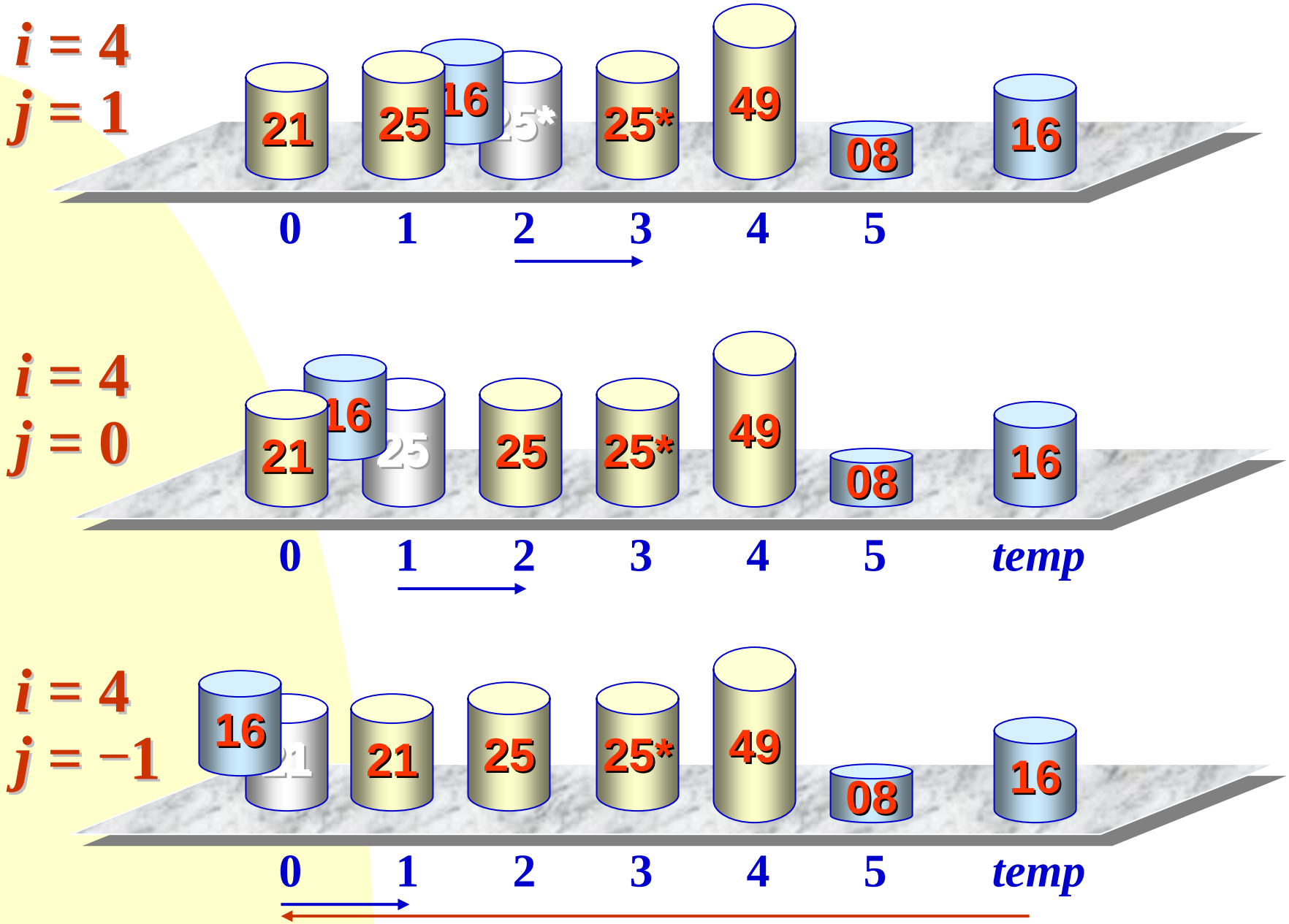
$i = 4$ 时的排序过程

$i = 4$
$j = 3$



$i = 4$
$j = 2$




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

直接插入排序的算法

```
template <class Type>
void InsertionSort ( datalist<Type> & list ) {
//按关键码 Key 非递减顺序对表进行排序
    for ( int i = 1; i < list.CurrentSize; i++ )
        Insert ( list, i );
}

template <class Type>
viod Insert ( datalist<Type> & list, int i ) {
   Element<Type> temp = list.Vector[i];
   int j = i;                    //从后向前顺序比较
```

```
while ( j > 0 &&
        temp.getKey ( ) < list.Vector[j−1].getKey ( ) )
    { list.Vector[j] = list.Vector[j −1]; j−−; }
  list.Vector[j] = temp;
}
```

算法分析

- 若设待排序的对象个数为***curremtsize = n***，则该算法的主程序执行***n-1***趟。
- 关键码比较次数和对象移动次数与对象关键码的初始排列有关。

- 最好情况下，排序前对象已经按关键码大小从小到大有序，每趟只需与前面的有序对象序列的最后一个对象的关键码比较 1 次，移动 2 次对象，总的关键码比较次数为 $n-1$，对象移动次数为 $2(n-1)$。
- 最坏情况下，第 i 趟时第 i 个对象必须与前面 i 个对象都做关键码比较，并且每做 1 次比较就要做 1 次数据移动。则总的关键码比较次数 KCN和对象移动次数RMN分别为

$$KCN = \sum_{i=1}^{n-1} i = n(n-1)/2 \approx n^2/2,$$

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

- 若待排序对象序列中出现各种可能排列的概率相同，则可取上述最好情况和最坏情况的平均情况。在平均情况下的关键码比较次数和对象移动次数约为 $n^2/4$。因此，直接插入排序的时间复杂度为 $o(n^2)$。
- 直接插入排序是一种稳定的排序方法。

折半插入排序 (Binary Insertsort)

- 折半插入排序基本思想是：设在顺序表中有一个对象序列 **V[0], V[1], …, v[*n*−1]**。其中，**v[0], V[1], …, v[*i*−1]** 是已经排好序的对象。在插入 **v[*i*]** 时，利用折半搜索法寻找 **v[*i*]** 的插入位置。

- 折半插入排序的算法

```
template <class Type>
void BinaryInsertSort ( datalist<Type> & list ) {
    for ( int i = 1; i < list.CurrentSize; i++)
        BinaryInsert ( list, i );
}
```

```
template <class Type>
void BinaryInsert ( datalist<Type> & list, int i ) {
    int left = 0,  Right = i−1;
    Element<Type> temp = list.Vector[i];
    while ( left <= Right ) {
        int middle = ( left + Right )/2;
        if ( temp.getKey ( ) <
                        list.Vector[middle].getKey ( ) )
            Right = middle − 1;
        else left = middle + 1;
    }
    for ( int k = i−1; k >= left; k−− )
        list.Vector[k+1] = list.Vector[k];
```

```
    list.Vector[left] = temp;
}
```

算法分析

- 对分查找比顺序查找快，所以对分插入排序就平均性能来说比直接插入排序要快。
- 它所需要的关键码比较次数与待排序对象序列的初始排列无关，仅依赖于对象个数。在插入第 i 个对象时，需要经过 $\lfloor \log_2 i \rfloor + 1$ 次关键码比较，才能确定它应插入的位置。因此，将 n 个对象（为推导方便，设为 $n=2^k$）用对分插入排序所进行的关键码比较次数为：

$$\sum_{i=1}^{n-1}\left(\lfloor \log_2 i \rfloor + 1\right) = \underbrace{1}_{2^0} + \underbrace{2+2}_{2^1} + \underbrace{3+\cdots+3}_{2^2} +$$

$$+ \underbrace{4+\cdots+4}_{2^3} + \cdots + \underbrace{k+\cdots+k}_{2^{k-1}} =$$

$$= (1 + 2 + 2^2 + \cdots + 2^{k-1}) + (2 + 2^2 + \cdots + 2^{k-1}) +$$

$$+ (2^2 + \cdots + 2^{k-1}) + \cdots + 2^{k-1} =$$

$$= \sum_{i=1}^{k}\sum_{j=i}^{k} 2^{j-1} = \sum_{i=1}^{k} 2^{i-1}(1 + 2 + \cdots + 2^{k-i}) =$$

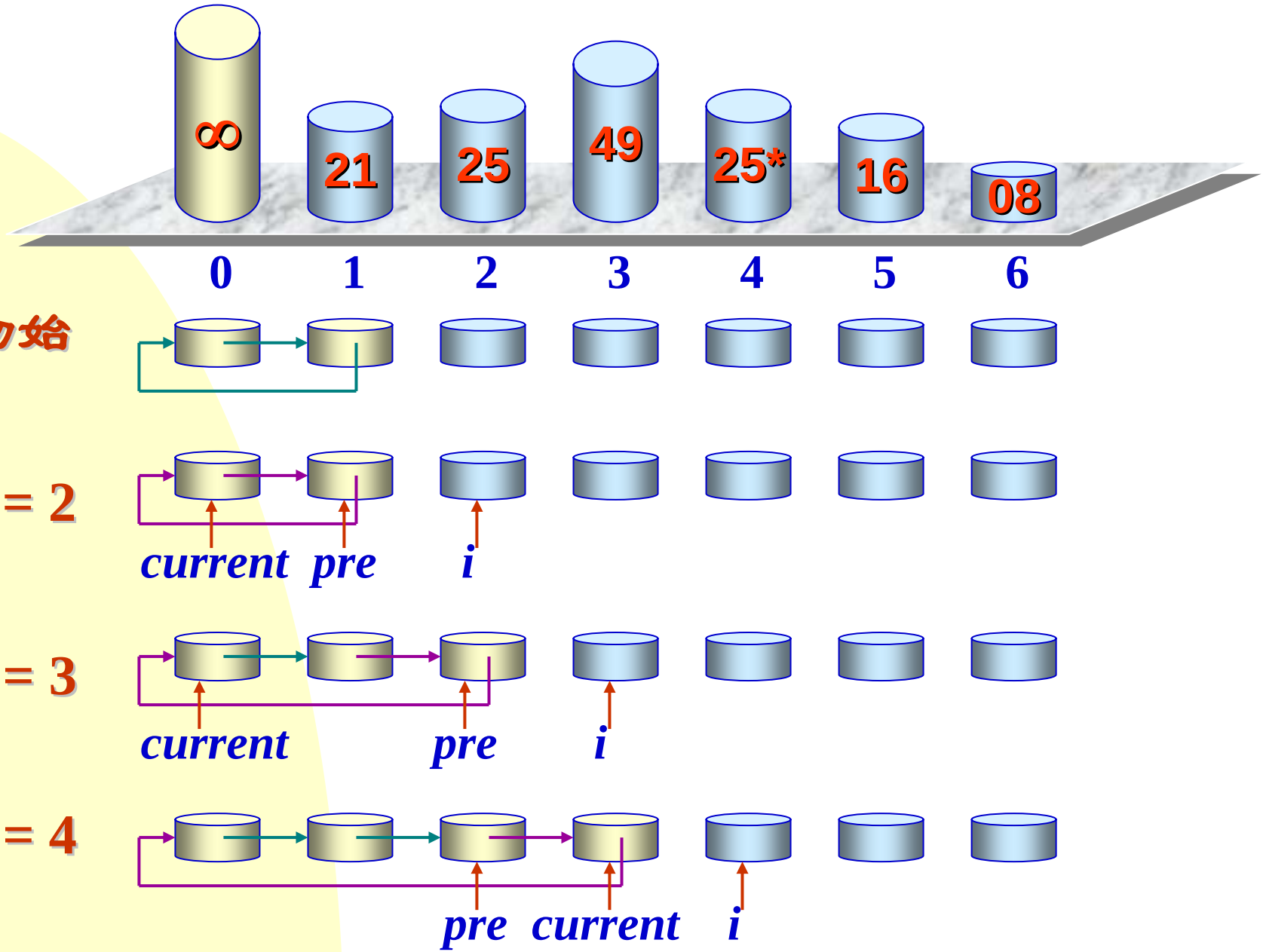
$$= \sum_{i=1}^{k} 2^{i-1}(2^{k-i+1} - 1) = \sum_{i=1}^{k} 2^k - \sum_{i=1}^{k} 2^{i-1} =$$

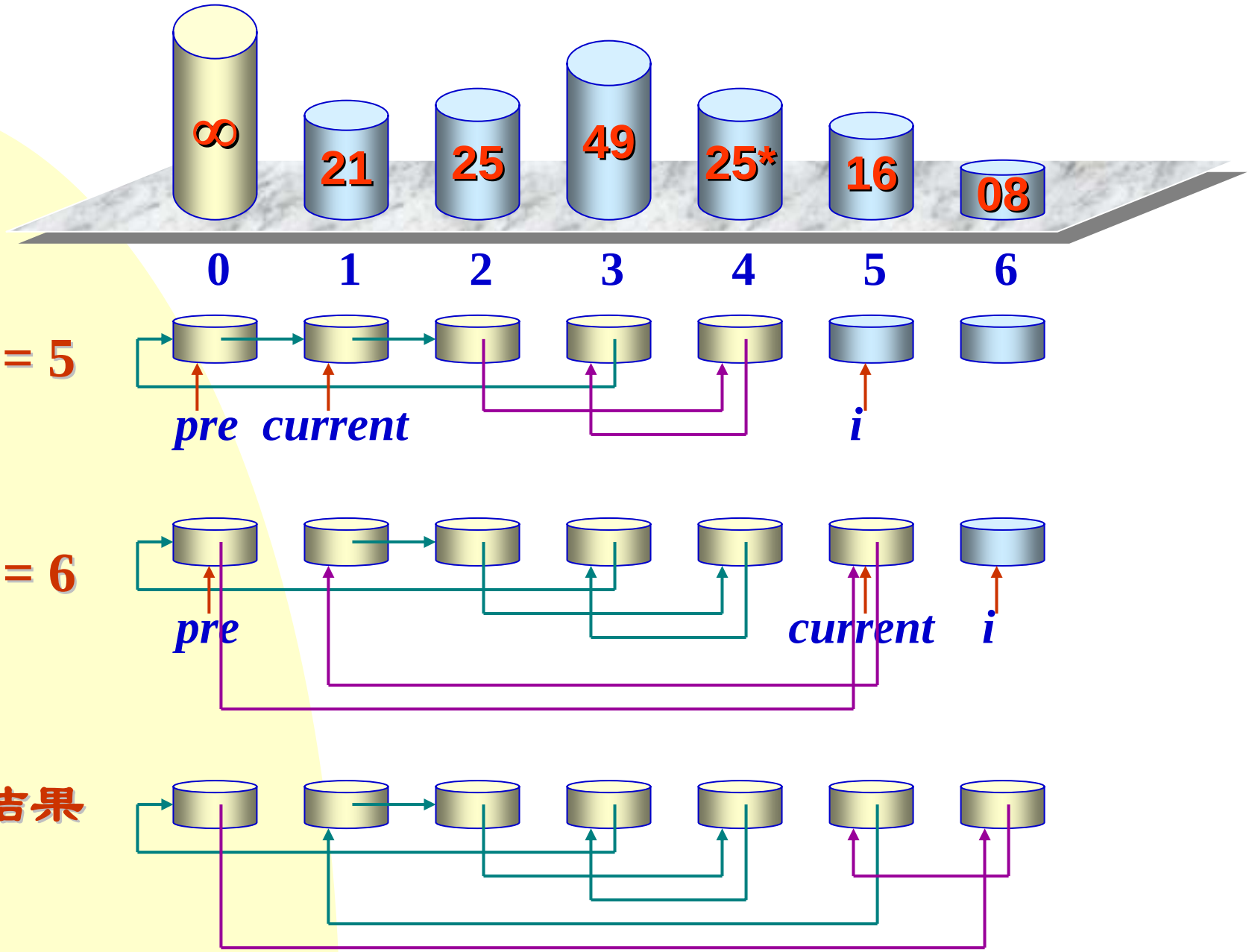
$$= k \cdot 2^k - (2^k - 1) = n \log_2 n - n + 1 \approx n \log_2 n$$

- 当 n 较大时，总关键码比较次数比直接插入排序的最坏情况要好得多，但比其最好情况要差。
- 在对象的初始排列已经按关键码排好序或接近有序时，直接插入排序比对分插入排序执行的关键码比较次数要少。对分插入排序的对象移动次数与直接插入排序相同，依赖于对象的初始排列。
- 对分插入排序是一个稳定的排序方法。

链表插入排序

- 链表插入排序的基本思想是：在每个对象的结点中增加一个链接指针数据成员 ***link***。
- 对于存放于数组中的一组对象**V[1], V[2], …, v[*n*]**，若**v[1], V[2], …, v[*i*-1]**已经通过指针***link***，按其关键码的大小，从小到大链接起来，现在要插入**v[*i*], *i* = 2, 3, …, *n***，则必须在前面***i*-1**个链接起来的对象当中，循链顺序检测比较，找到**v[*i*]**应插入(或链入)的位置，把**v[*i*]**插入，并修改相应的链接指针。这样就可得到**v[1], V[2], …, v[*i*]**的一个通过链接指针排列好的链表。
- 如此重复执行，直到把**v[*n*]**也插入到链表中排好序为止。


∞
21
25
49
25*
16
08
0
1
2
3
4
5
6
初始
i = 2
current
pre
i
i = 3
current
pre
i
i = 4
pre
current
i


∞
21
25
49
25*
16
08
0
1
2
3
4
5
6
i = 5
pre
current
i
i = 6
pre
current
i
结果

链表插入排序示例

	index	(0)	(1)	(2)	(3)	(4)	(5)	(6)
	key	MaxNum	21	25	49	25^*	16	08
状态	*link*	1	0	0	0	0	0	0
i=2	*key*	MaxNum	21	25	49	25^*	16	08
	link	1	2	0	0	0	0	0
i=3	*key*	MaxNum	21	25	49	25^*	16	08
	link	1	2	3	0	0	0	0
i=4	*key*	MaxNum	21	25	49	25^*	16	08
	link	1	2	4	0	3	0	0
i=5	*key*	MaxNum	21	25	49	25^*	16	08
	link	5	2	4	0	3	1	0
i=6	*key*	MaxNum	21	25	49	25^*	16	08
	link	6	2	4	0	3	1	5

用于链表排序的静态链表的类定义

```
template <class Type> class staticlinklist;

template <class Type> class Element {
private:
   Type key;          //结点的关键码
   int link;          //结点的链接指针
public:
   Element ( ) : key(0), link (NULL) { }
   Type getKey ( ) { return key; }          //提取关键码
   void setKey ( const Type x ) { key = x; }     //修改
    int getLink ( ) { return link; }          //提取链指针
    void setLink ( const int l ) { link = l; }  //设置指针
```

```
}
template <class Type> class staticlinklist {
public:
    dstaticlinklist ( int MaxSz = DefaultSize ) :
        MaxSize ( Maxsz ), CurrentSize (0)   //构造函数
        { Vector = new Element <Type> [MaxSz]; }
private:
    Element <Type> * Vector;                 //存储向量
    int MaxSize, CurrentSize;
                //向量中最大元素个数和当前元素个数
}
```

链表插入排序的算法

```
template <class Type>
int LinkInsertSort ( staticlinklis<Type> & list ) {
    list.Vector[0].setKey ( MaxNum );
    list.Vector[0].setLink ( 1 );
    list.Vector[1].setLink ( 0 );        //形成循环链表
    for ( int i = 2; i <= list.CurrentSize; i++ ) {
        int  int current = list.Vector[0].getLink ( );
        int pre = 0;                              //前驱指针
        while ( list.Vector[current].getKey ( ) <=
              list.Vector[i].getKey ( ) ) {      //找插入位置
            pre = current;        //pre跟上, current向前走
            current = list.Vector[current].getLink ( ); }
```

```
        list.Vector[i].setLink ( current );
        list.Vector[pre].setLink ( i );
            //在pre与current之间链入
    }
}
```

算法分析

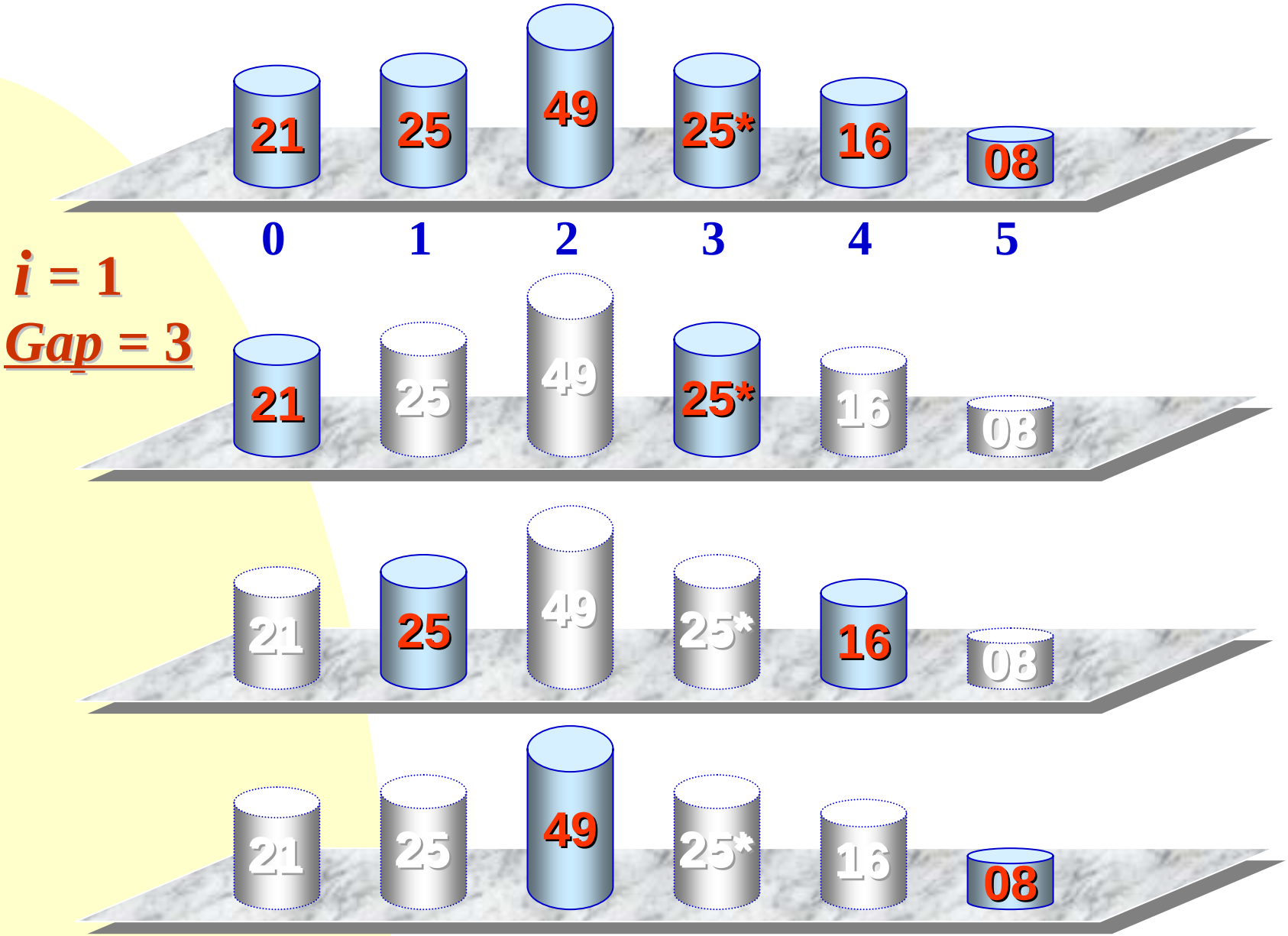
- 使用链表插入排序，每插入一个对象，最大关键码比较次数等于链表中已排好序的对象个数，最小关键码比较次数为**1**。故总的关键码比较次数最小为 $n-1$，最大为

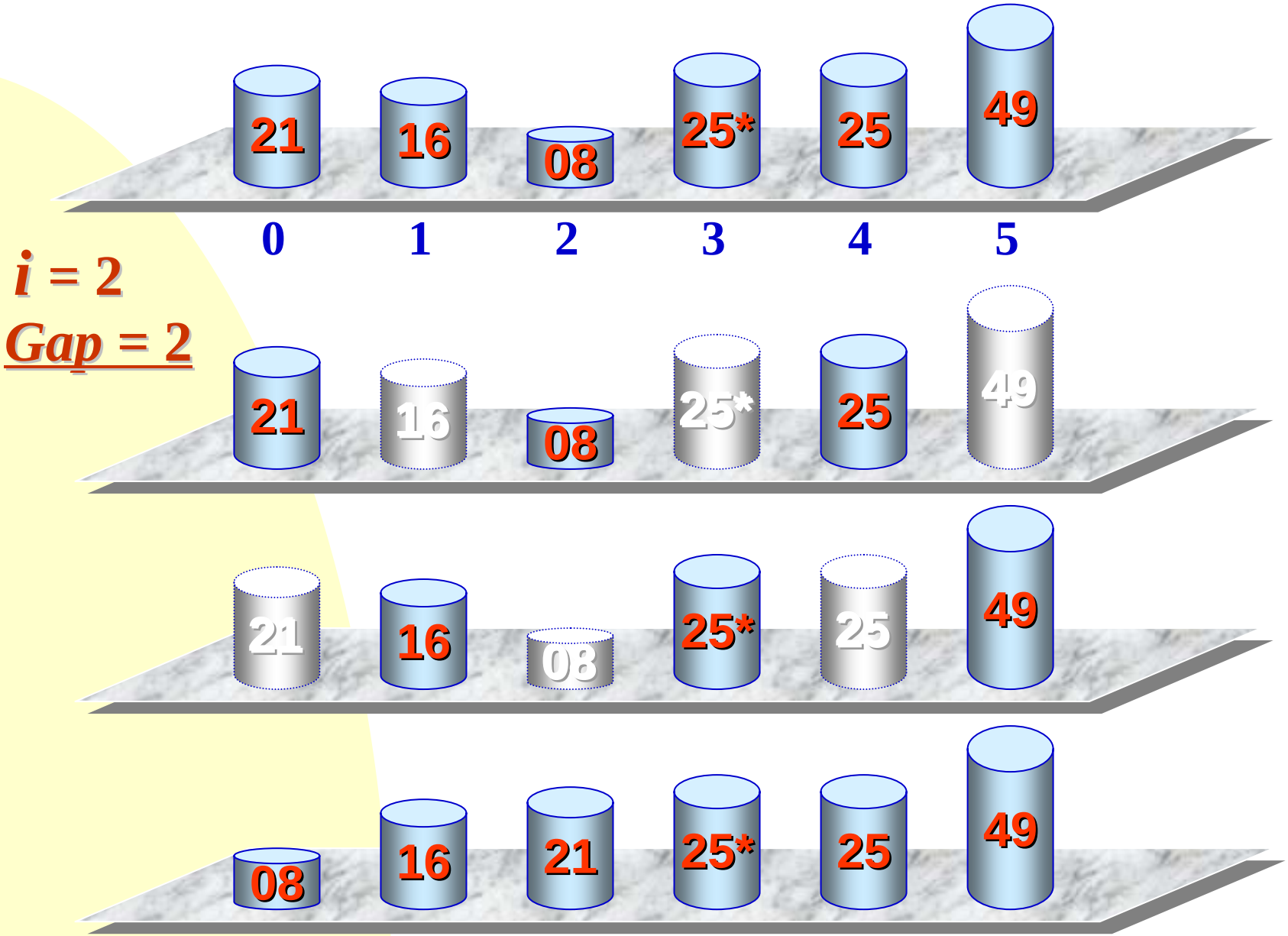
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

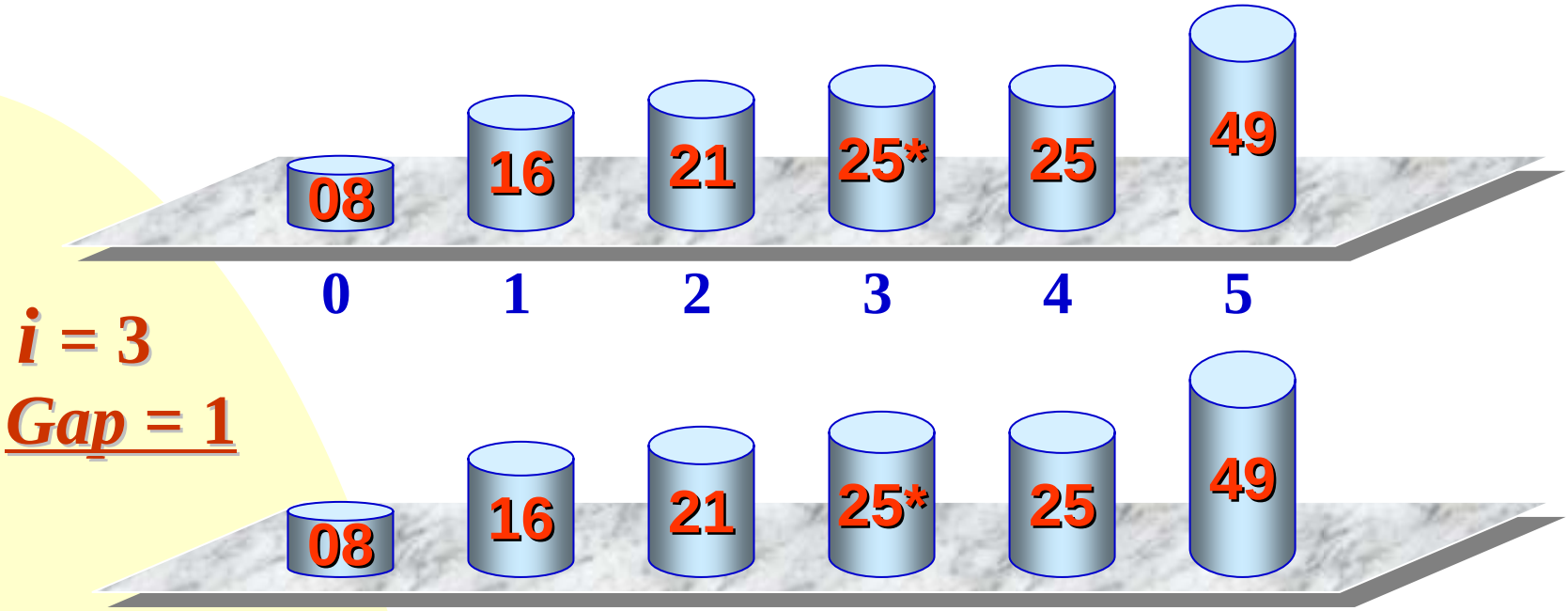
- 用链表插入排序时，对象移动次数为**0**。但为了实现链表插入，在每个对象中增加了一个链域 ***link***，并使用了***vector*[0]**作为链表的表头结点，总共用了 ***n*** 个附加域和一个附加对象。
- 算法从 ***i* = 2** 开始，从前向后插入。并且在***vector*[*current*].*Key* == *vector*[*i*].*Key***时，***current***还要向前走一步，***pre***跟到***current***原来的位置，此时，***vector*[*pre*].*Key* == *vector*[*i*].*Key***。将***vector*[*i*]**插在***vector*[*pre*]**的后面，所以，链表插入排序方法是稳定的。

希尔排序 (Shell Sort)

- 希尔排序方法又称为缩小增量排序。该方法的基本思想是：设待排序对象序列有 n 个对象，首先取一个整数 $gap < n$ 作为间隔，将全部对象分为 gap 个子序列，所有距离为 gap 的对象放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然后缩小间隔 gap，例如取 $gap = \lceil gap/2 \rceil$，重复上述的子序列划分和排序工作。直到最后取 $gap == 1$，将所有对象放在同一个序列中排序为止。


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


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



- 开始时 ***gap*** 的值较大，子序列中的对象较少，排序速度较快；随着排序进展，***gap*** 值逐渐变小，子序列中对象个数逐渐变多，由于前面工作的基础，大多数对象已基本有序，所以排序速度仍然很快。

希尔排序的算法

```
template <class Type>
void Shellsort ( datalist<Type> & list ) {
    int gap = list.CurrentSize / 2;
    // gap 是子序列间隔
    while ( gap ) {                          //循环,直到gap为零
        ShellInsert ( list, gap );     //一趟直接插入排序
        gap = gap == 2 ? 1 : ( int ) ( gap/2.2 );   //修改
    }
}
```

```
template <class Type> void
shellInsert ( datalist<Type> & list; const int gep ) {
//一趟希尔排序,按间隔gap划分子序列
    for ( int i = gap; i < list.CurrentSize; i++) {
        Element<Type> temp = list.Vector[i];
        int j = i;
        while ( j >= gap && temp.getKey ( ) <
                            list.Vector[j−gap].getKey ( ) ) {
            list.Vector[j] = list.Vector[j −gap];
            j −= gap;
         }
         list.Vector[j] = temp;
    }
}
```

- ***Gap*的取法有多种。最初 shell 提出取 *gap* = ⌊*n*/2⌋，*gap* =⌊*gap*/2⌋，直到*gap* = 1。后来 knuth 提出取*gap* =⌊*gap*/3⌋+1。还有人提出都取奇数为好，也有人提出各*gap*互质为好。**

算法分析

- 对特定的待排序对象序列，可以准确地估算关键码的比较次数和对象移动次数。
- 但想要弄清关键码比较次数和对象移动次数与增量选择之间的依赖关系，并给出完整的数学分析，还没有人能够做到。

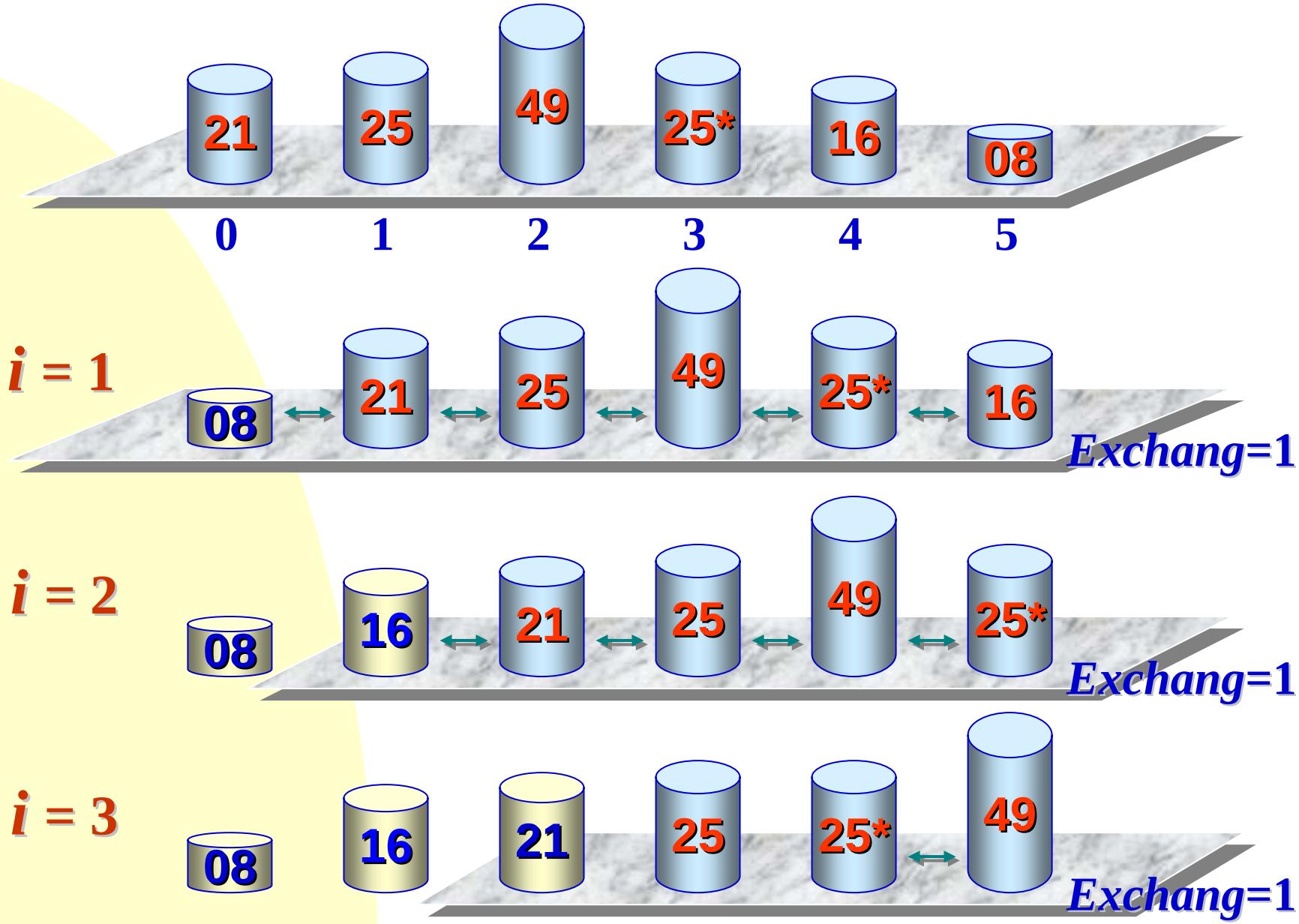
- **Knuth**利用大量的实验统计资料得出，当 n 很大时，关键码平均比较次数和对象平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内。这是在利用直接插入排序作为子序列排序方法的情况下得到的。

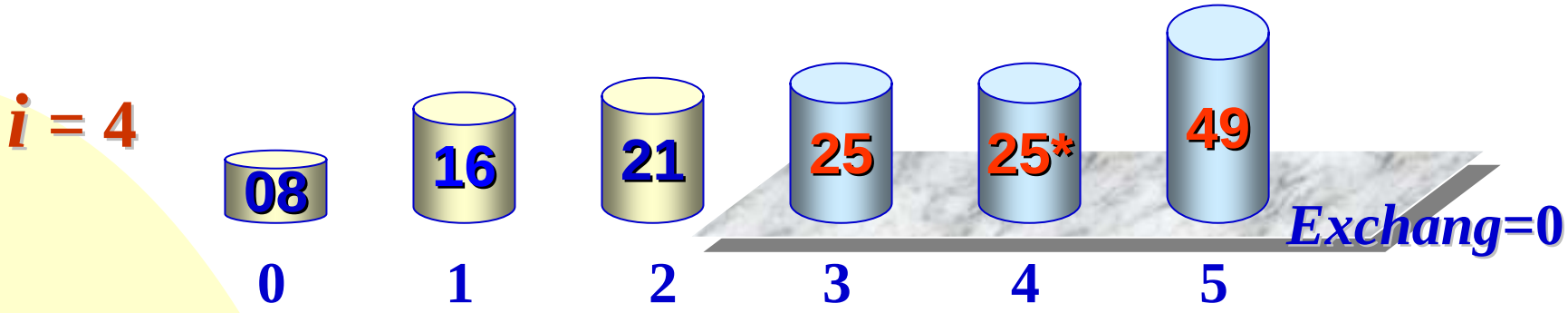
交换排序 (Exchange Sort)

交换排序的基本思想是两两比较待排序对象的关键码，如果发生逆序(即排列顺序与排序后的次序正好相反)，则交换之，直到所有对象都排好序为止。

起泡排序 (Bubble Sort)

- 起泡排序的基本方法是：设待排序对象序列中的对象个数为 n。最多作 $n-1$ 趟，$i = 1, 2, \ldots, n-2$ 。在第 i 趟中顺次两两比较$v[n-j-1].Key$和$v[n-j].Key$，$j = n-1, n-2, \ldots, i$。如果发生逆序，则交换$v[n-j-1]$和$v[n-j]$。


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



起泡排序的算法

```
template <class Type>
void BubbleSort (datalist<Type> & list ) {
    int pass = 1;  int exchange = 1;
    while ( pass < list.CurrentSize && exchange ){
        BubbleExchange ( list, pass, exchange );
        pass++;
    }
}
```

```
template <class Type>
 void BubbleExchange ( datalist<Type> & list ,
       const int i, int & exchange ) {
    exchange = 0;     //交换标志置为0,假定未交换
    for ( int j = list.CurrentSize−1;  j >= i;  j−− )
       if ( list.Vector[j−1].getKey ( ) >
                        list.Vector[j].getKey ( ) ) {    //逆序
          Swap ( list.Vector[j−1], list.Vector[j] );//交换
          exchange = 1;       //交换标志置为1,有交换
       }
}
```

- 第 i 趟对待排序对象序列$v[i-1], v[i], \ldots, v[n-1]$进行排序，结果将该序列中关键码最小的对象交换到序列的第一个位置$(i-1)$，其它对象也都向排序的最终位置移动。
- 当然在个别情形下，对象有可能在排序中途向相反的方向移动。
- 这样最多做$n-1$趟起泡就能把所有对象排好序。

算法分析

- 在对象的初始排列已经按关键码从小到大排好序时，此算法只执行一趟起泡，做 $n-1$ 次关键码比较，不移动对象。这是最好的情形。

- 最坏的情形是算法执行了$n-1$趟起泡，第 i 趟 $(1 \le i < n)$ 做了 $n-i$ 次关键码比较，执行了$n-i$次对象交换。这样在最坏情形下总的关键码比较次数KCN和对象移动次数RMN为：

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

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

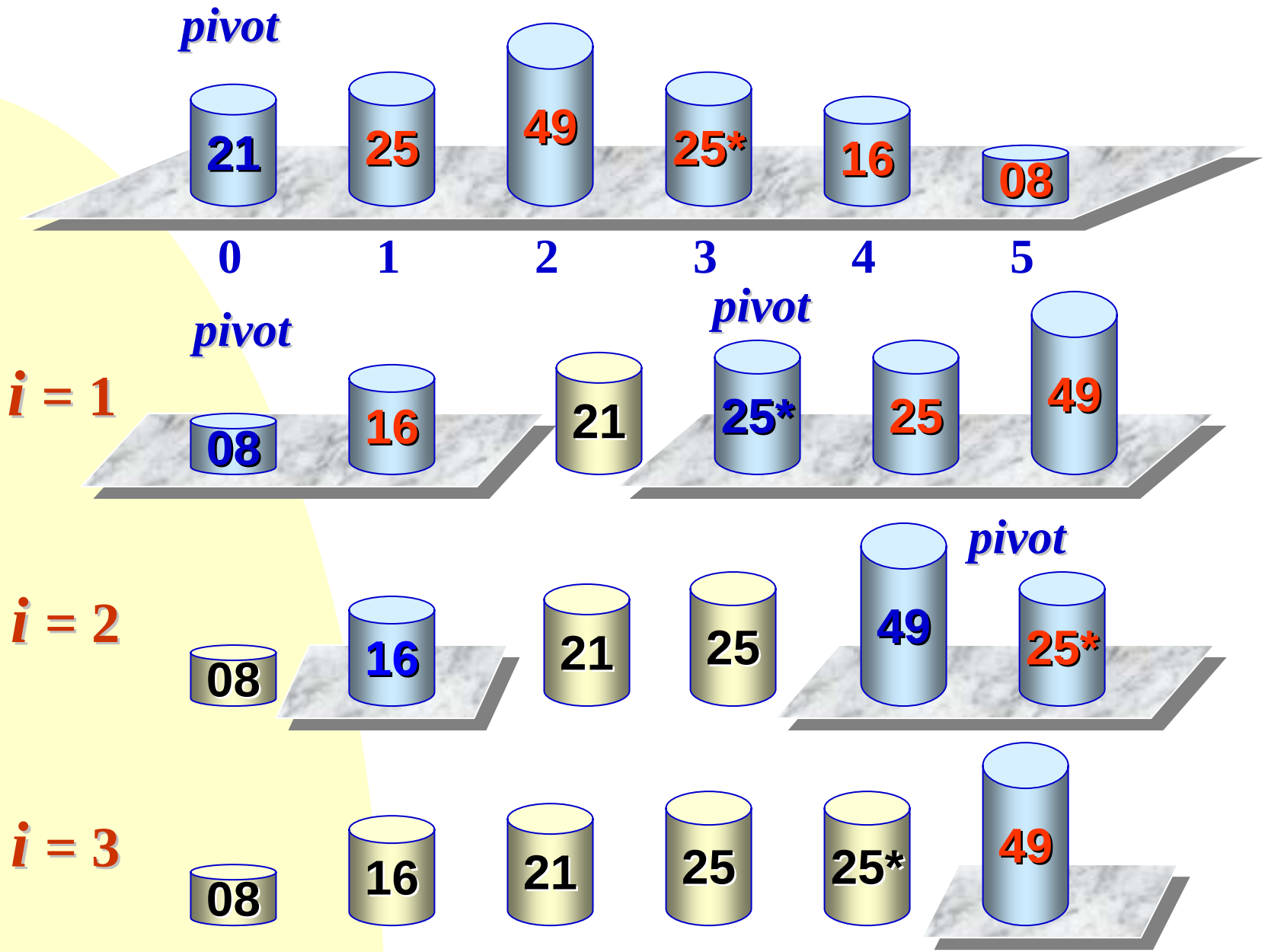
- 起泡排序需要一个附加对象以实现对象值的对换。
- 起泡排序是一个稳定的排序方法。

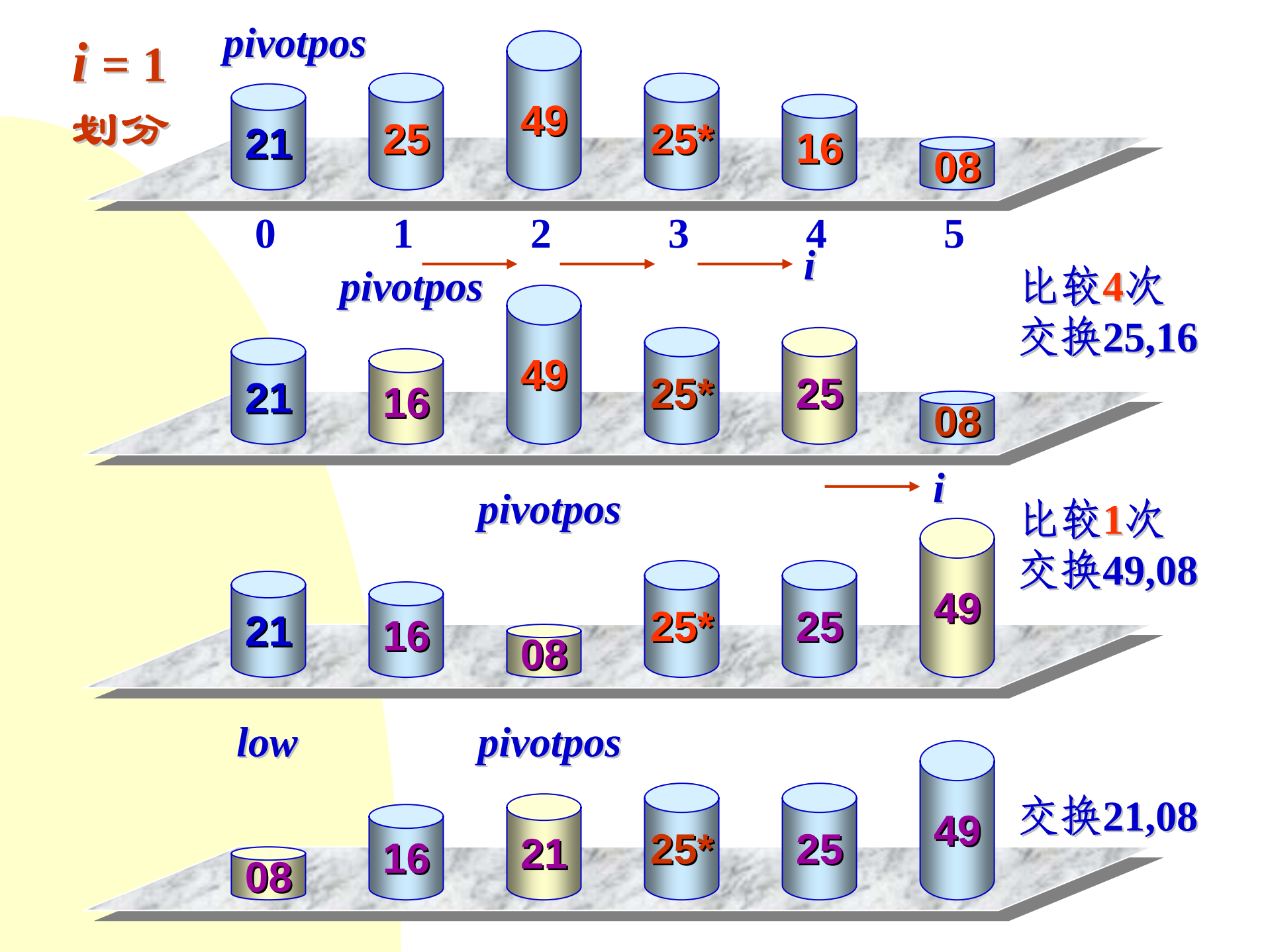
快速排序 (Quick Sort)

- 快速排序方法的基本思想是任取待排序对象序列中的某个对象 (例如取第一个对象) 作为基准，按照该对象的关键码大小，将整个对象序列划分为左右两个子序列:
  - 左侧子序列中所有对象的关键码都小于或等于基准对象的关键码
  - 右侧子序列中所有对象的关键码都大于基准对象的关键码
- 基准对象则排在这两个子序列中间(这也是该对象最终应安放的位置)。

- 然后分别对这两个子序列重复施行上述方法，直到所有的对象都排在相应位置上为止。
- 算法描述

```
QuickSort ( List ) {
    if ( List的长度大于1) {
        将序列List划分为两个子序列
            LeftList 和 Right List;
        QuickSort ( LeftList );
        QuickSort ( RightList );
        将两个子序列 LeftList 和 RightList
            合并为一个序列List;
    }
}
```


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

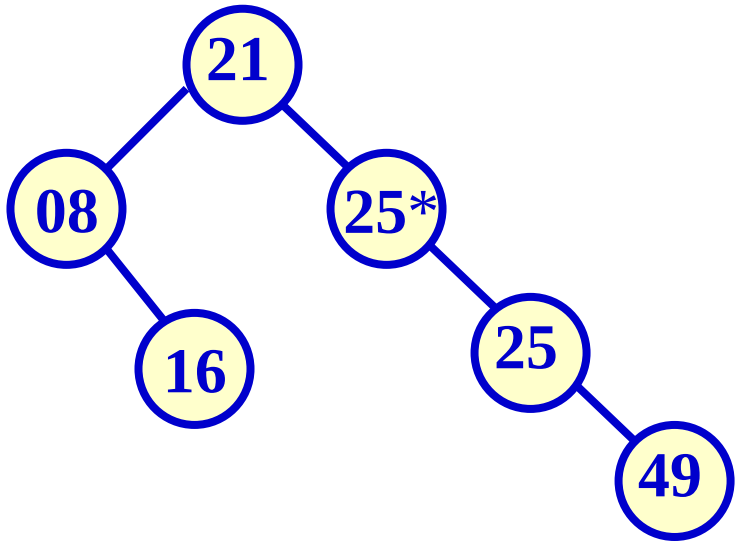

i = 1
划分
pivotpos
21
25
49
25*
16
08
0
1
2
3
4
5
i
pivotpos
比较4次
交换25,16
21
16
49
25*
25
08
i
pivotpos
比较1次
交换49,08
21
16
08
25*
25
49
low
pivotpos
交换21,08
08
16
21
25*
25
49

快速排序的算法

```
template <class Type>
void QuickSort ( datalist<Type> &list, const int left,
                 const int right ) {
//在待排序区间 left~right 中递归地进行快速排序
   if ( left < right) {
      int pivotpos = Partition ( list, left, right ); //划分
      QuickSort ( list, left, pivotpos−1);
       //在左子区间递归进行快速排序
      QuickSort ( list, pivotpos+1, right );
       //在左子区间递归进行快速排序
   }
}
```

```
template <class Type>
int Partition ( datalist<Type> &list, const int low,
                const int high ) {
   int pivotpos = low;                          //基准位置
   Element<Type> pivot = list.Vector[low];
   for ( int i = low+1; i <= high; i++ )
      if ( list.Vector[i].getKey ( ) < pivot.getKey( )
           && ++pivotpos != i  )
         Swap ( list.Vector[pivotpos], list.Vector[i] );
         //小于基准对象的交换到区间的左侧去
   Swap ( list.Vector[low], list.Vector[pivotpos] );
   return pivotpos;
}
```

- 算法*quicksort*是一个递归的算法，其递归树如图所示。



- 算法*partition*利用序列第一个对象作为基准，将整个序列划分为左右两个子序列。算法中执行了一个循环，只要是关键码小于基准对象关键码的对象都移到序列左侧，最后基准对象安放到位，函数返回其位置。

算法分析

- 从快速排序算法的递归树可知，快速排序的趟数取决于递归树的深度。
- 如果每次划分对一个对象定位后，该对象的左侧子序列与右侧子序列的长度相同，则下一步将是对两个长度减半的子序列进行排序，这是最理想的情况。
- 在n个元素的序列中，对一个对象定位所需时间为 $O(n)$。若设 $t(n)$ 是对 n 个元素的序列进行排序所需的时间，而且每次对一个对象正确定位后，正好把序列划分为长度相等的两个子序列，此时，总的计算时间为：

$$T(n) \le cn + 2\,t(n/2) \qquad // c\text{是一个常数}$$

$$\le Cn + 2\,(\,cn/2 + 2t(n/4)\,) = 2cn + 4t(n/4)$$

$$\le 2cn + 4\,(\,cn/4 + 2t(n/8)\,) = 3cn + 8t(n/8)$$

$$\cdots\cdots$$

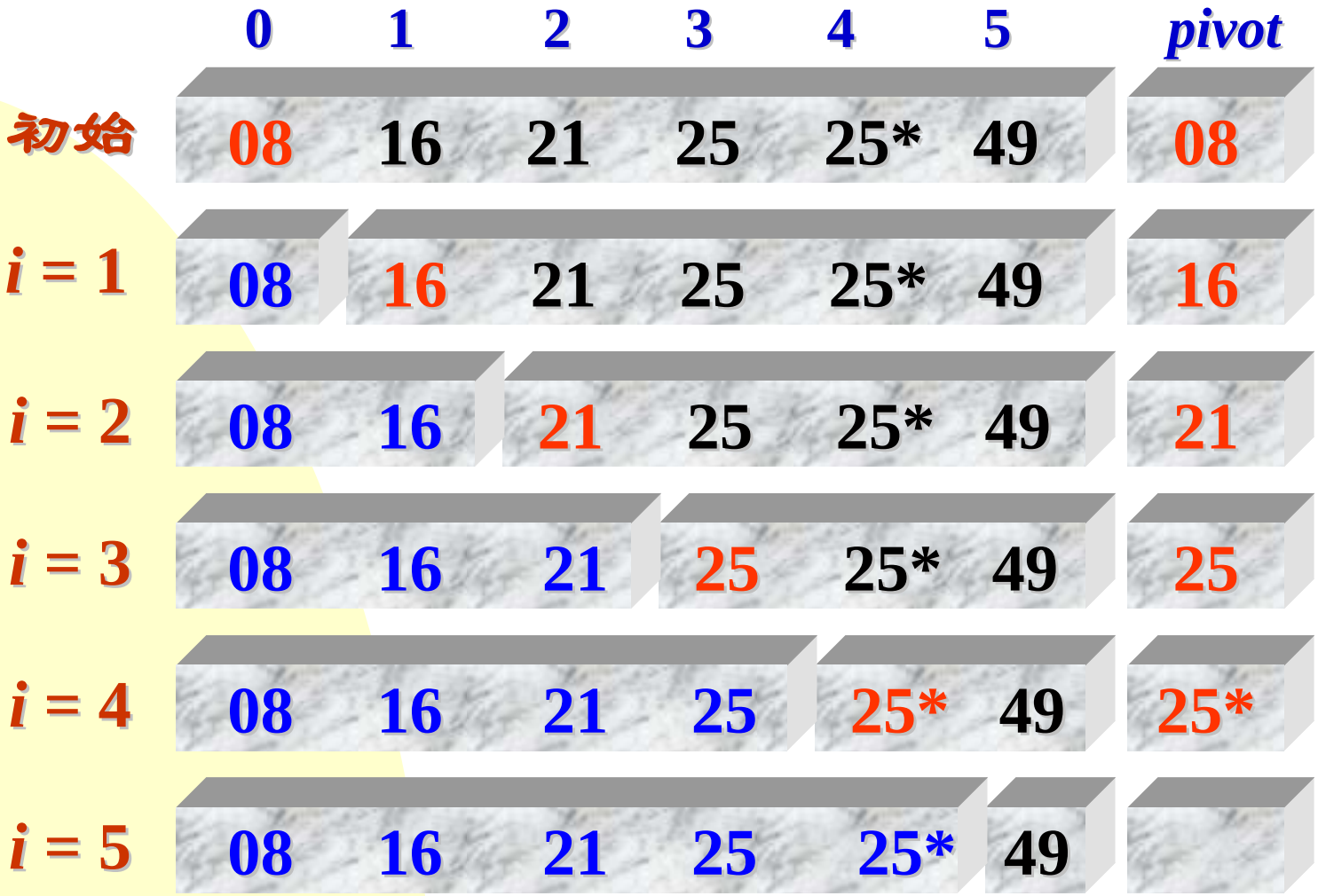
$$\le Cn\log_2 n + nt(1) = o(n\log_2 n)$$

- 可以证明，函数***quicksort***的平均计算时间也是$o(n\log_2 n)$。实验结果表明：就平均计算时间而言，快速排序是我们所讨论的所有内排序方法中最好的一个。

- 快速排序是递归的，需要有一个栈存放每层递归调用时的指针和参数。
- 最大递归调用层次数与递归树的深度一致，理想情况为$\lceil \log_2(n+1) \rceil$。因此，要求存储开销为 $o(\log_2 n)$。
- 在最坏的情况，即待排序对象序列已经按其关键码从小到大排好序的情况下，其递归树成为单支树，每次划分只得到一个比上一次少一个对象的子序列。这样，必须经过 $n-1$ 趟才能把所有对象定位，而且第 i 趟需要经过 $n-i$ 次关键码比较才能找到第 i 个对象的安放位置，总的关键码比较次数将达到

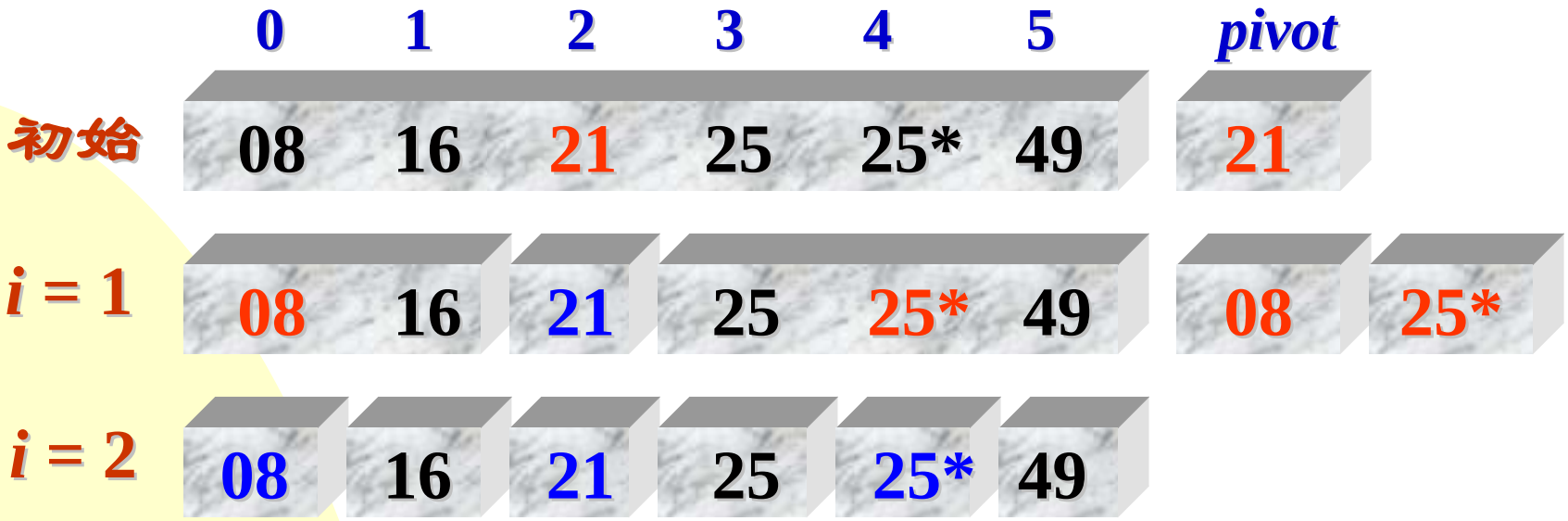
$$\sum_{i=1}^{n-1}(n-i)=\frac{1}{2}n(n-1)\approx\frac{n^2}{2}$$

- 其排序速度退化到简单排序的水平，比直接插入排序还慢。占用附加存储(即栈)将达到o(*n*)。
- 若能更合理地选择基准对象，使得每次划分所得的两个子序列中的对象个数尽可能地接近，可以加速排序速度，但是由于对象的初始排列次序是随机的，这个要求很难办到。
- 有一种改进办法：取每个待排序对象序列的第一个对象、最后一个对象和位置接近正中的3个对象，取其关键码居中者作为基准对象。



用第一个对象作为基准对象

快速排序退化的例子



用居中关键码对象作为基准对象

- 快速排序是一种不稳定的排序方法。
- 对于 *n* 较大的平均情况而言，快速排序是“快速”的，但是当 *n* 很小时，这种排序方法往往比其它简单排序方法还要慢。

选择排序

选择排序的基本思想是：每一趟 (例如第 i 趟，$i = 0, 1, \ldots, n-2$) 在后面 $n-i$ 个待排序对象中选出关键码最小的对象, 作为有序对象序列的第 i 个对象。待到第 $n-2$ 趟作完，待排序对象只剩下1个，就不用再选了。

直接选择排序 (Select Sort)

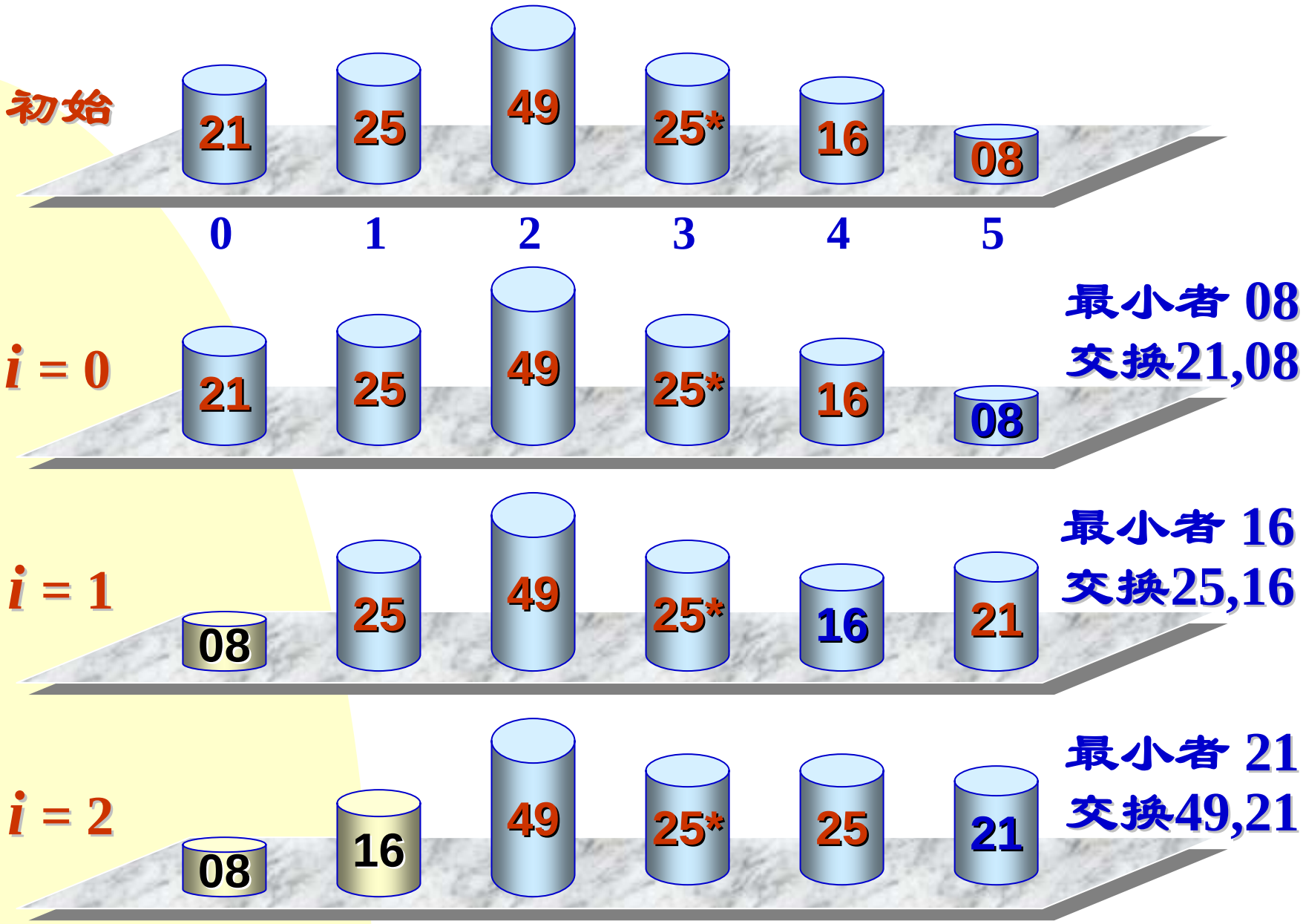
- 直接选择排序是一种简单的排序方法，它的基本步骤是:
  - ☆ 在一组对象$v[i] \sim v[n-1]$中选择具有最小关键码的对象;

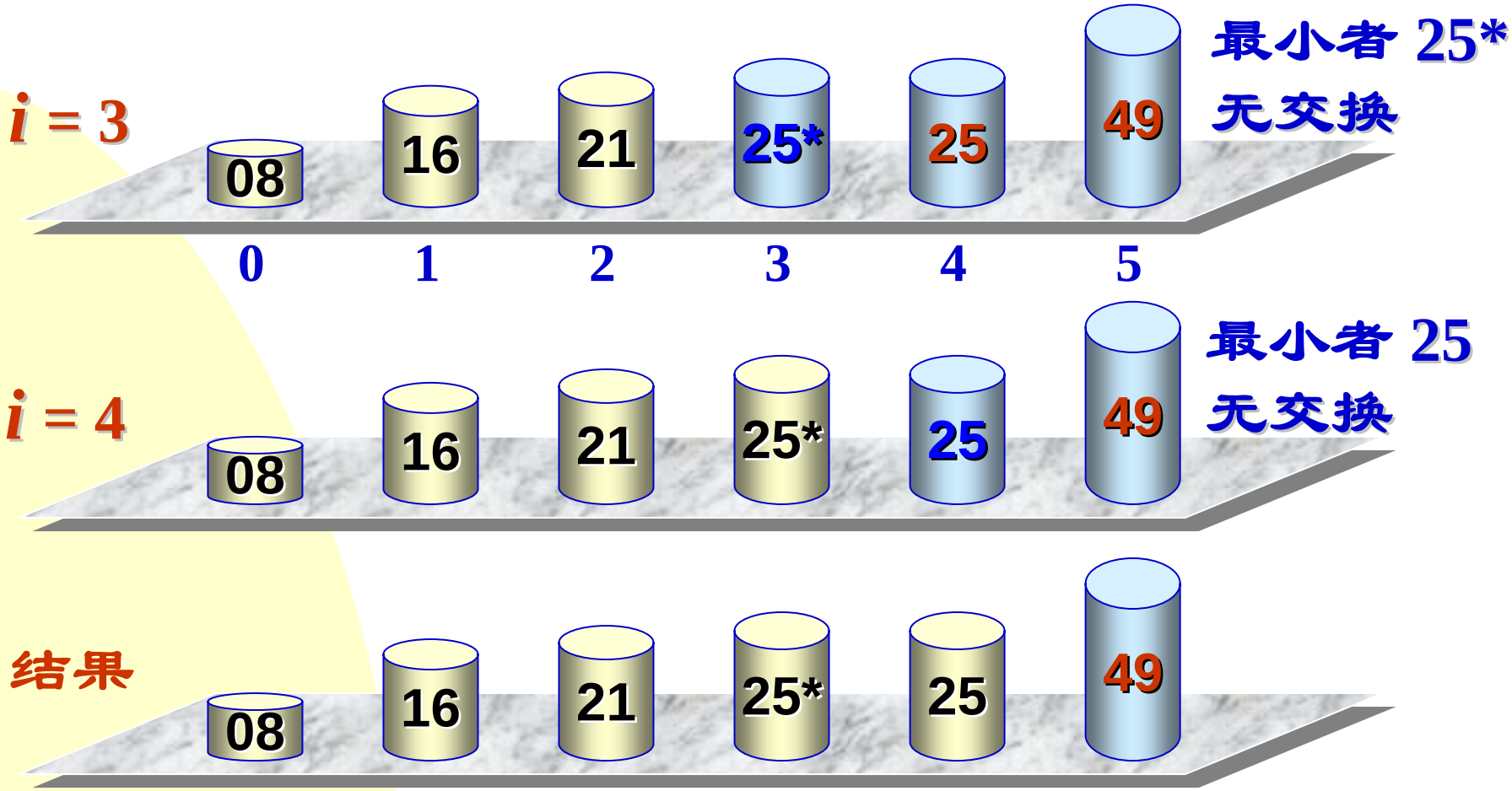
- 若它不是这组对象中的第一个对象，则将它与这组对象中的第一个对象对调；
- 在这组对象中剔除这个具有最小关键码的对象，在剩下的对象$v[i+1] \sim v[n-1]$中重复执行第☆、②步，直到剩余对象只有一个为止。

直接选择排序的算法

```
template <class Type>
void SelectSort ( datalist<Type> & list ) {
    for ( int i = 0; i < list.CurrentSize−1; i++ )
        SelectExchange ( list, i );
}
```

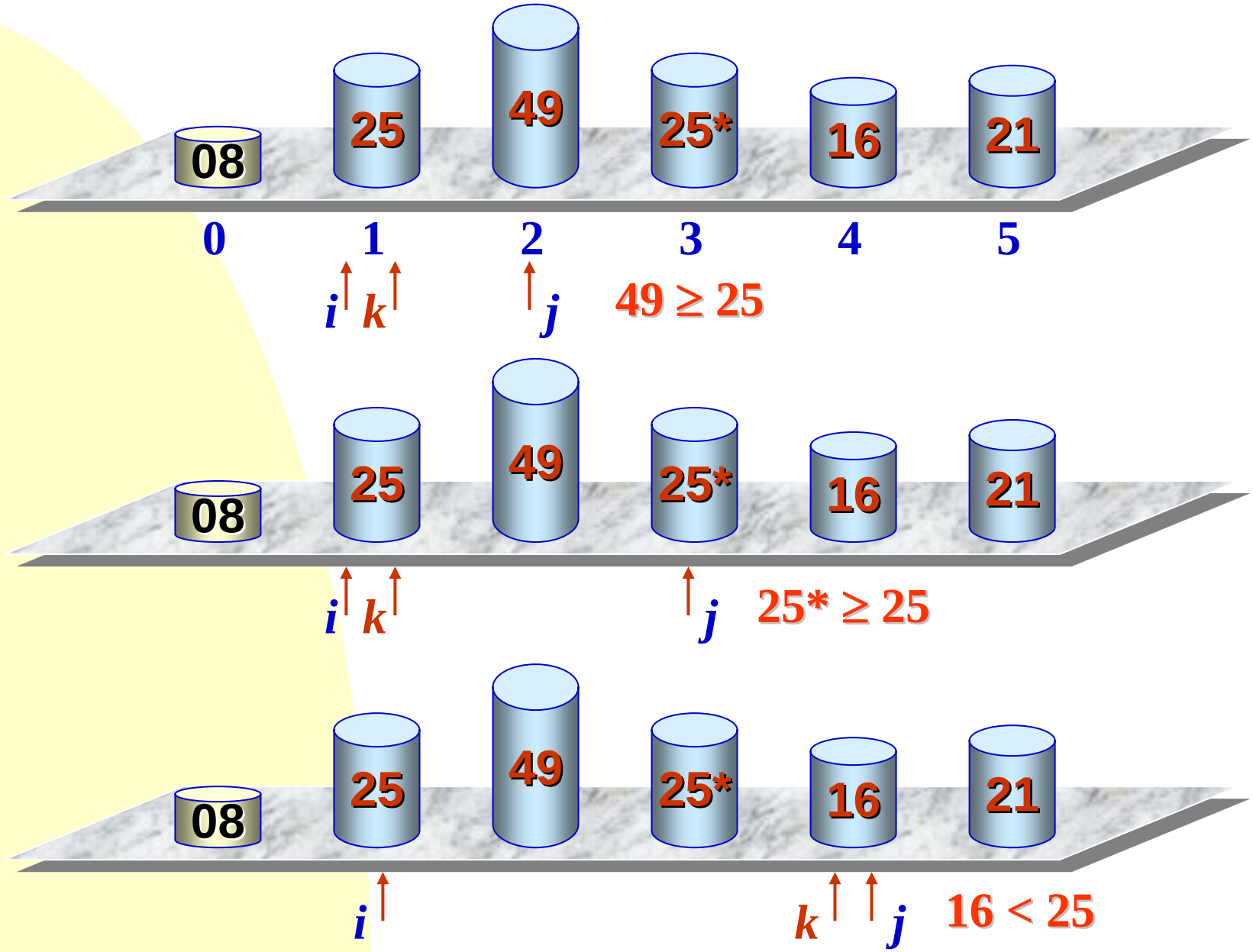
```
template <class Type>
void SelectExchange ( datalist<Type> & list,
          const int i ) {
    int k = i;
    for ( int j = i+1; j < list.CurrentSize; j++)
        if ( list.Vector[j].getKey ( ) <
                  list.Vector[k].getKey ( ) )
            k = j;          //当前具最小关键码的对象
        if ( k != i )       //对换到第 i 个位置
            Swap ( list.Vector[i], list.Vector[k] );
}
```

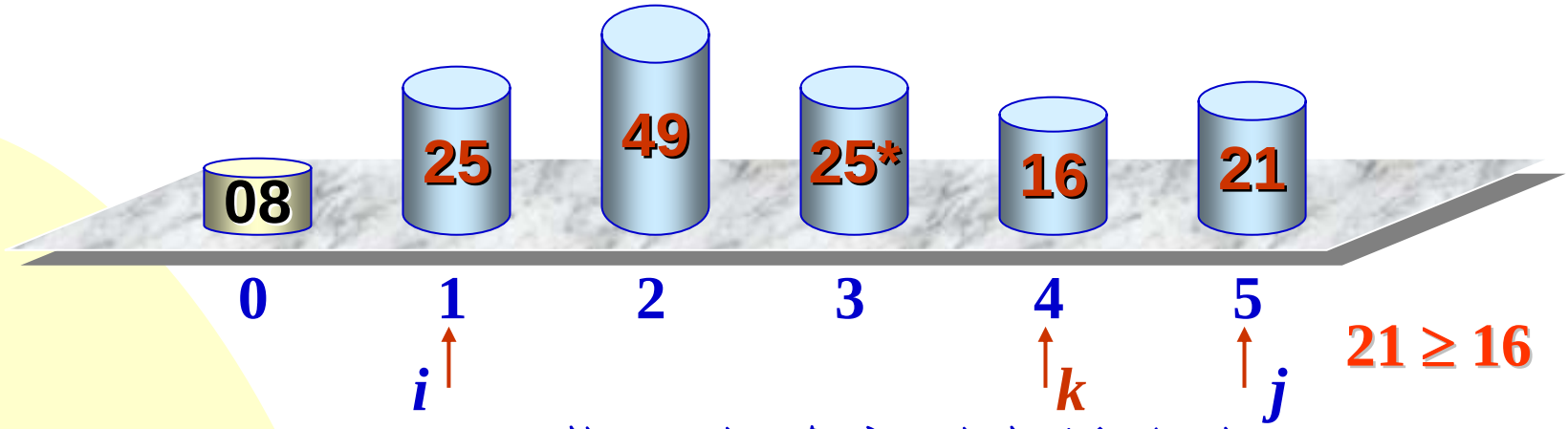

初始
21
25
49
25*
16
08
0
1
2
3
4
5
i = 0
最小者 08
交換21,08
21
25
49
25*
16
08
i = 1
最小者 16
交換25,16
08
25
49
25*
16
21
i = 2
最小者 21
交換49,21
08
16
49
25*
25
21


i = 3
08
16
21
25*
25
49
最小者 25*
无交换
0
1
2
3
4
5
i = 4
08
16
21
25*
25
49
最小者 25
无交换
结果
08
16
21
25*
25
49

各趟排序后的结果

i =1时选择排序的过程


08
25
49
25*
16
21
0
1
2
3
4
5
i
k
j
49 ≥ 25
08
25
49
25*
16
21
i
k
j
25* ≥ 25
08
25
49
25*
16
21
i
k
j
16 < 25



k 指示当前序列中最小者

算法分析

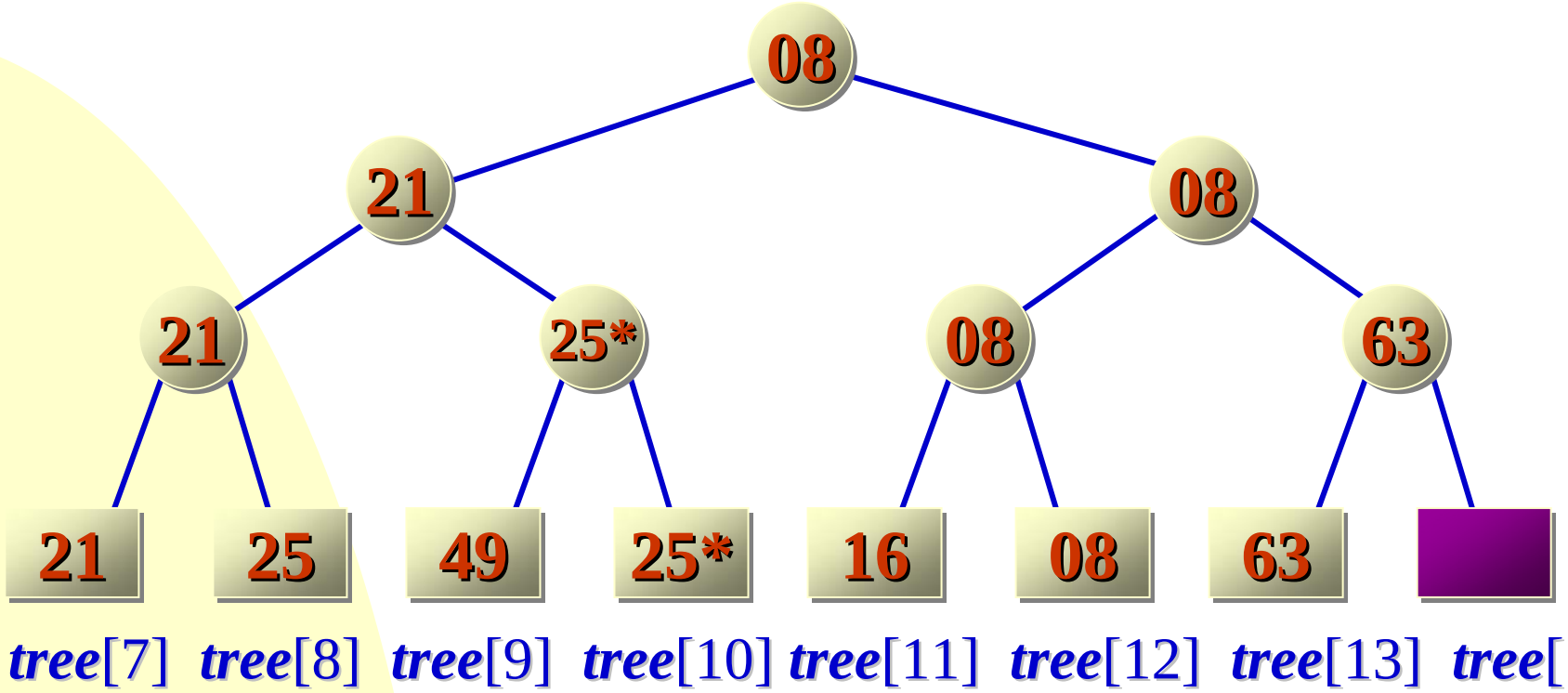
- 直接选择排序的关键码比较次数*KCN*与对象的初始排列无关。第 *i* 趟选择具有最小关键码对象所需的比较次数总是 *n*−*i*−1 次，此处假定整个待排序对象序列有 *n* 个对象。因此，总的关键码比较次数为

$$KCN = \sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2}$$

- 对象的移动次数与对象序列的初始排列有关。当这组对象的初始状态是按其关键码从小到大有序的时候，对象的移动次数$RMN = 0$，达到最少。
- 最坏情况是每一趟都要进行交换，总的对象移动次数为$RMN = 3(n-1)$。
- 直接选择排序是一种不稳定的排序方法。

锦标赛排序 (Tournament Tree Sort)

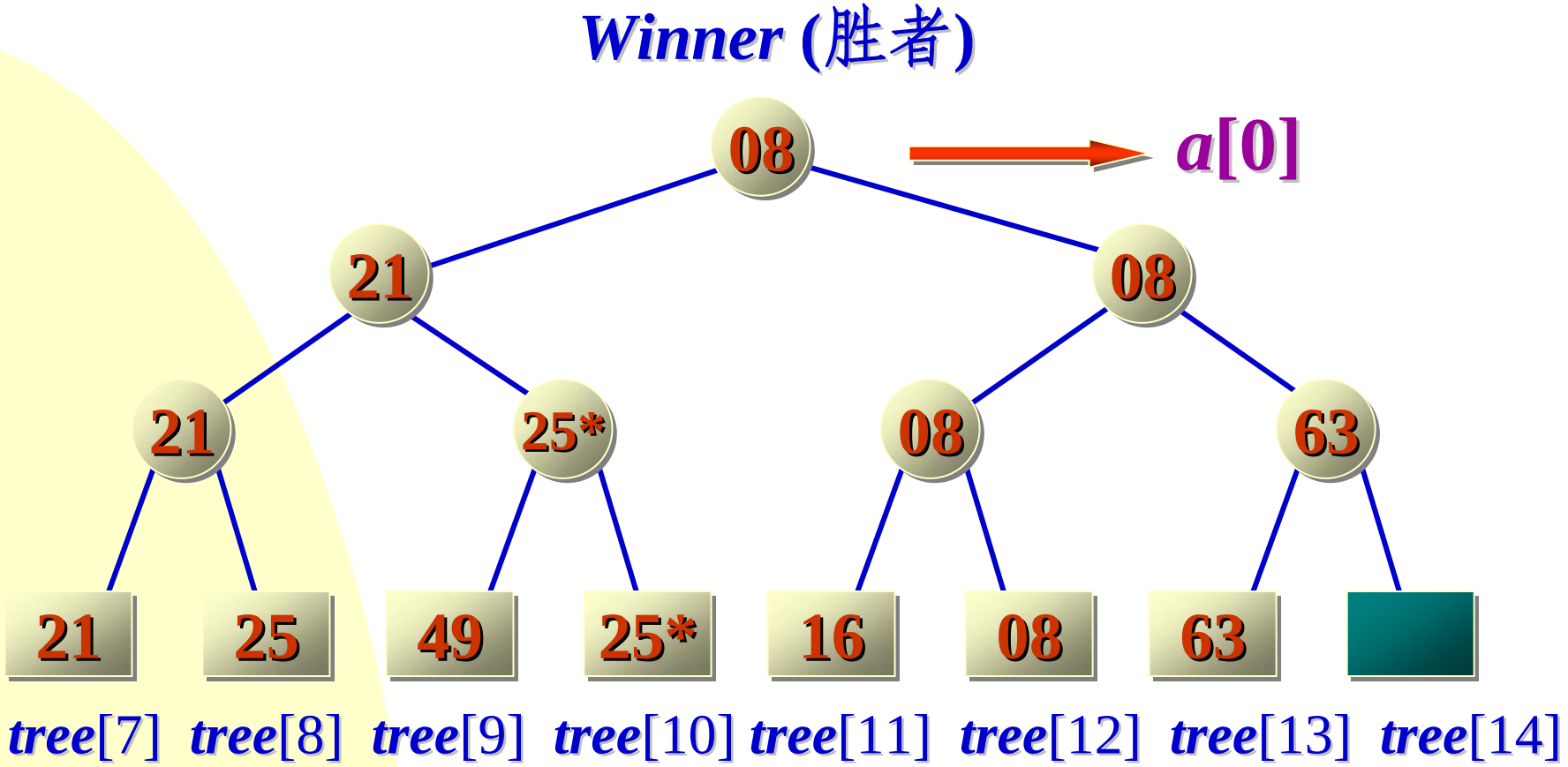
- 它的思想与体育比赛时的淘汰赛类似。首先取得 n 个对象的关键码，进行两两比较，得到$\lceil n/2 \rceil$个比较的优胜者(关键码小者)，作为第一步比较的结果保留下来。然后对这$\lceil n/2 \rceil$个对象再进行关键码的两两比较，…，如此重复，直到选出一个关键码最小的对象为止。
- 在图例中，最下面是对象排列的初始状态，相当于一棵满二叉树的叶结点，它存放的是所有参加排序的对象的关键码。



- 如果 n 不是2的 k 次幂，则让叶结点数补足到满足 $2^{k-1} < n \le 2^k$ 的2^k个。叶结点上面一层的非叶结点是叶结点关键码两两比较的结果。最顶层是树的根。

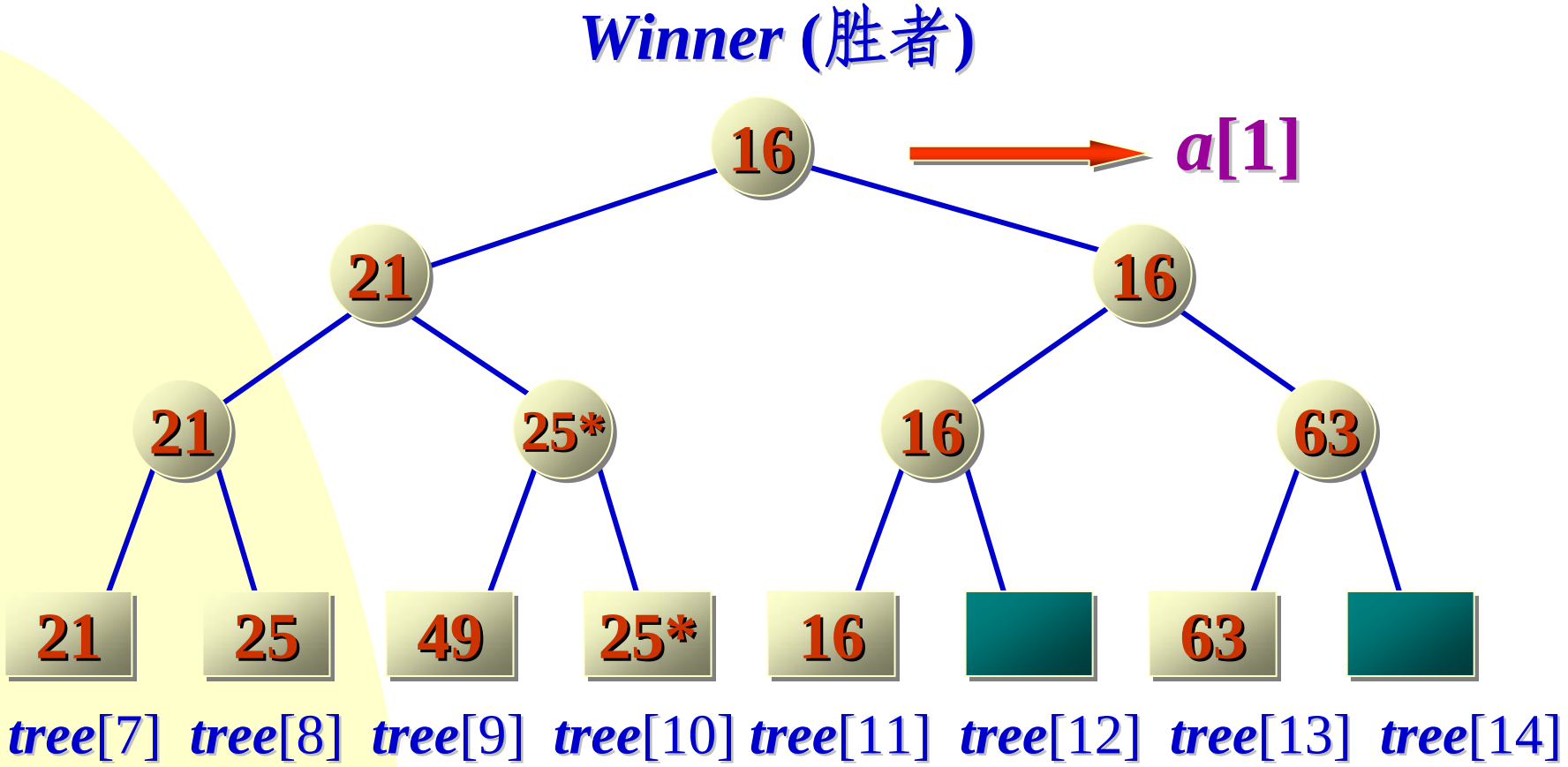
胜者树的概念

- 每次两两比较的结果是把关键码小者作为优胜者上升到双亲结点，称这种比赛树为胜者树。
- 位于最底层的叶结点叫做胜者树的外结点，非叶结点称为胜者树的内结点。每个结点除了存放对象的关键码 ***data*** 外，还存放了此对象是否要参选的标志 ***Active*** 和此对象在满二叉树中的序号***index***。
- 胜者树最顶层是树的根，表示最后选择出来的具有最小关键码的对象。



形成初始胜者树（最小关键码上升到根）

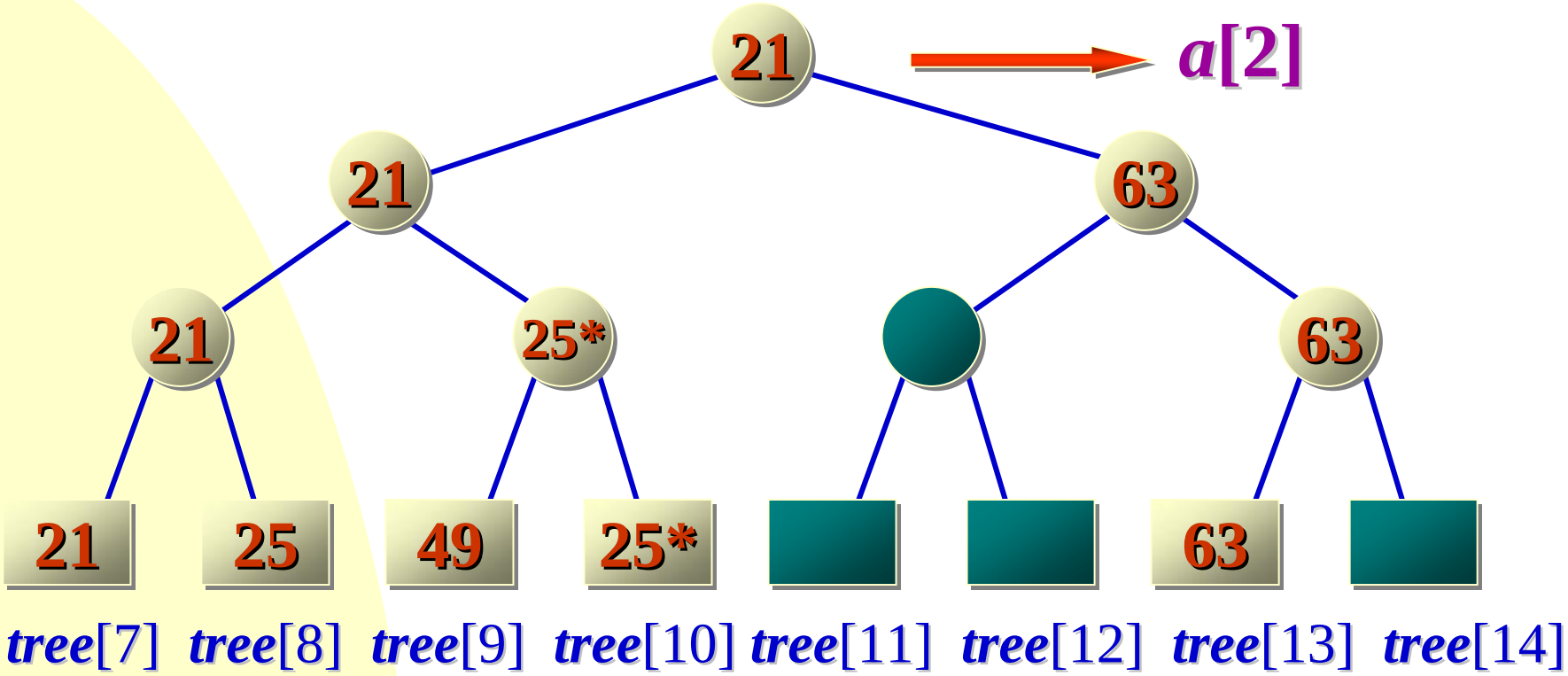
关键码比较次数：6



输出冠军并调整胜者树后树的状态

关键码比较次数：2

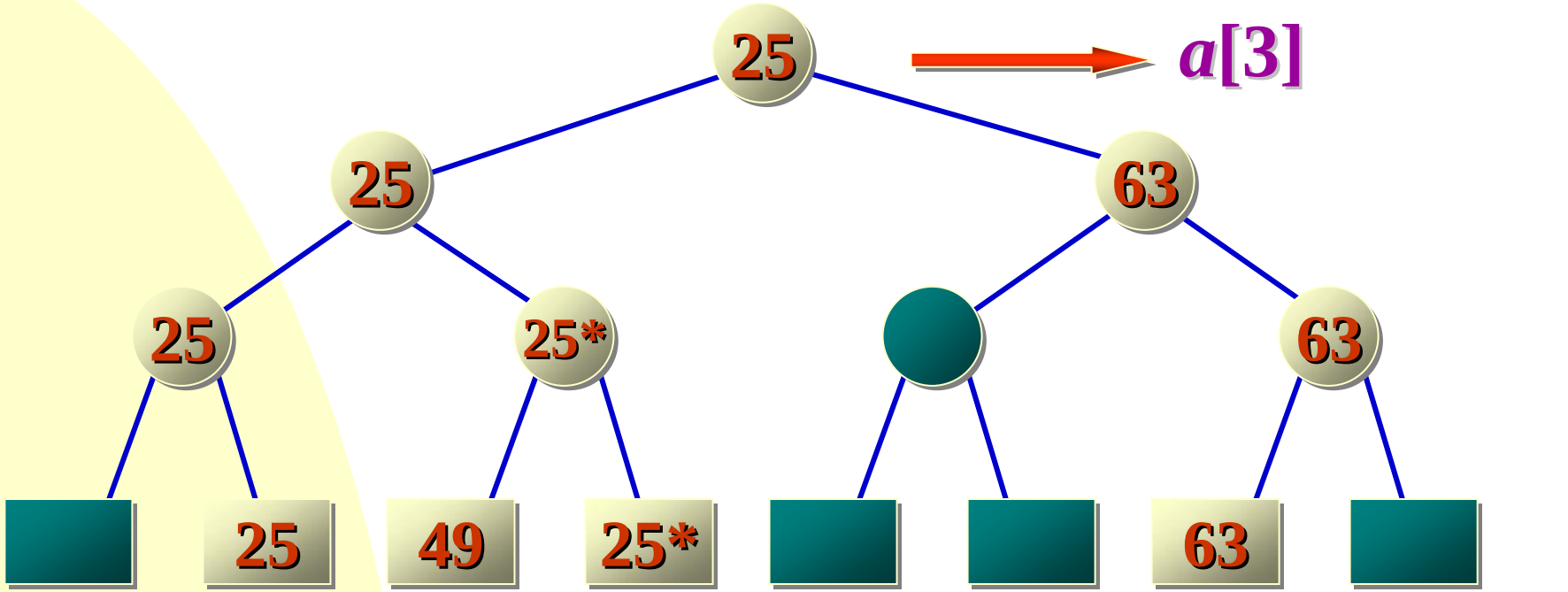
Winner (胜者)



输出亚军并调整胜者树后树的状态

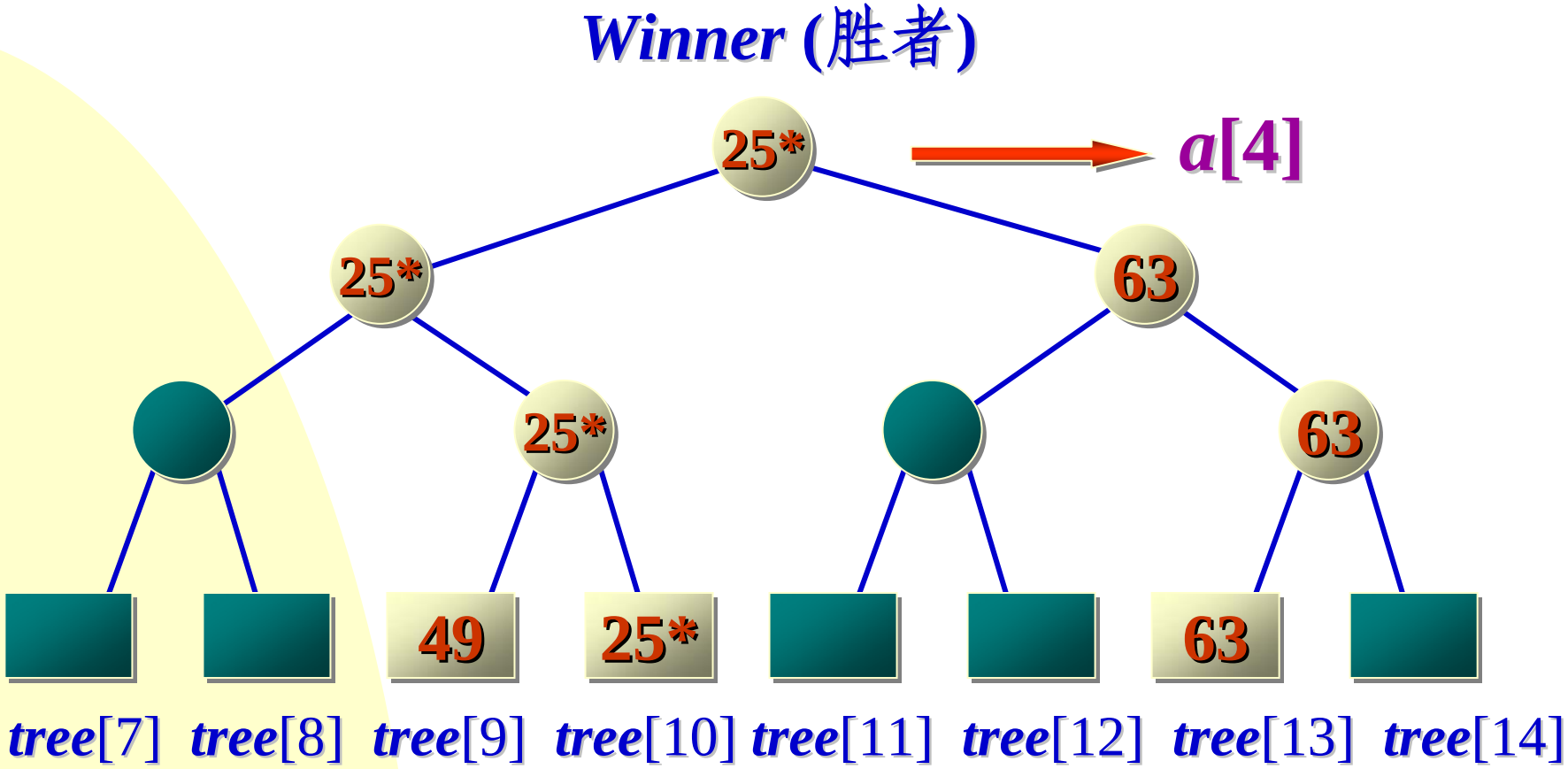
关键码比较次数：2

Winner (胜者)



输出第三名并调整胜者树后树的状态

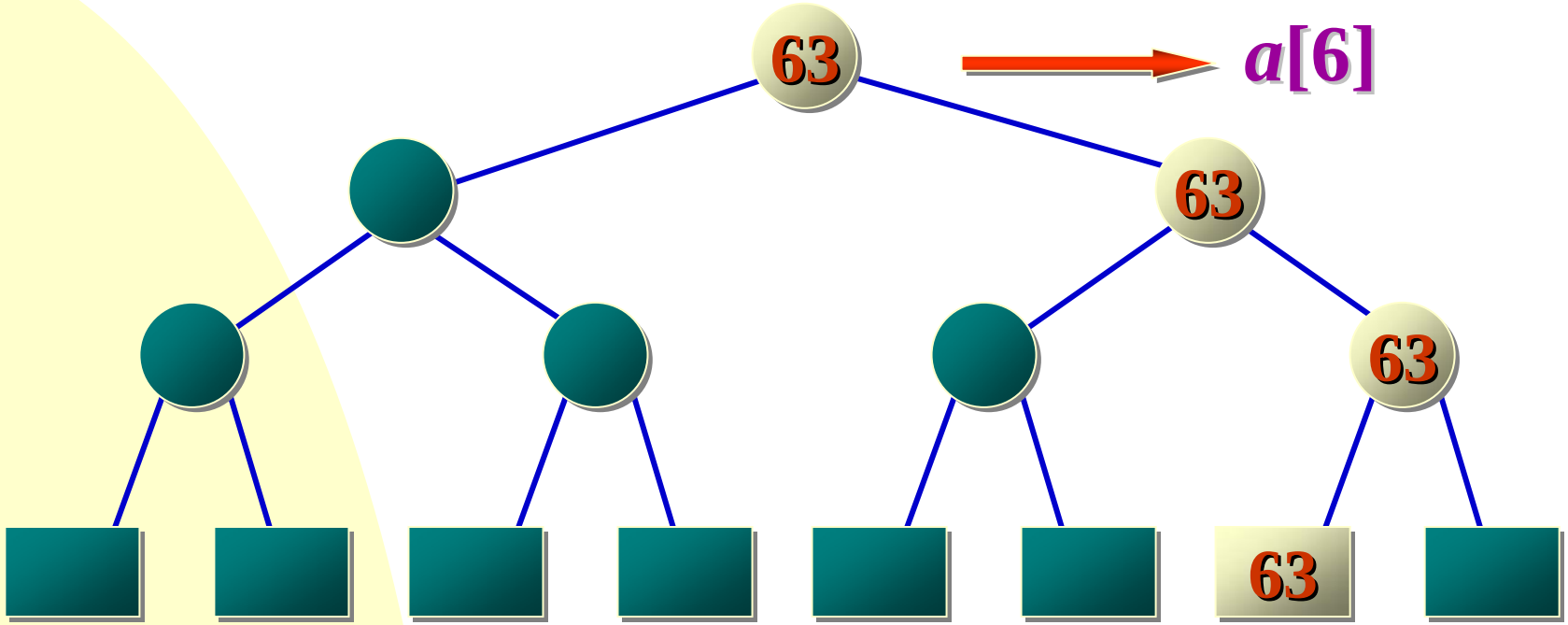
关键码比较次数：2



输出第四名并调整胜者树后树的状态

关键码比较次数：2

Winner (胜者)



全部比赛结果输出时树的状态

关键码比较次数：2

胜者树数据结点类的定义

```
template <class Type> class DataNode {
public:
    Type data;    //数据值
    int index;    //结点在满二叉树中顺序号
    int active;   //参选标志：=1, 参选, =0, 不参选
}
```

锦标赛排序的算法

```
template <class Type>
void TournamentSort ( Type a[ ], int n ) {
    DataNode<Type> *tree;
    DataNode<Type>  item;
```

```
int bottomRowSize = PowerOfTwo ( n );  //乘幂值
int TreeSize = 2*bottomRowSize−1; //总结点个数
int loadindex = bottomRowSize−1;   //内结点个数
tree = new DataNode<Type>[TreeSize];
int j = 0;          //从 a 向胜者树外结点复制数据
for ( int i = loadindex; i < TreeSize; i++) {
    tree[i].index = i;
    if ( j < n )
       { tree[i].active = 1;  tree[i].data = a[j++]; }
    else tree[i].active = 0;
}
i = loadindex;     //进行初始比较选择最小的项
while ( i ) {
```

```
    j = i;
    while ( j < 2*i ) {
        if ( !tree[j+1].active||          //胜者送入双亲
                tree[j].data <= tree[j+1].data )
            tree[(j−1)/2] = tree[j];
        else tree[(j−1)/2] = tree[j+1];
        j += 2;
    }
    i = (i−1)/2;        // i 退到双亲, 直到 i==0为止
}
for ( i = 0; i < n−1; i++) {  //处理其它n-1个数据
    a[i] = tree[0].data;                  //送出最小数据
    tree[tree[0].index].active = 0;  //失去参选资格
```

```
        UpdateTree ( tree, tree[0].index );    //调整
    }
    a[n−1] = tree[0].data;
}
```

锦标赛排序中的调整算法

```
template <class Type>
void UpdateTree ( DataNode<Type> *tree, int i ) {
    if ( i % 2 == 0 )
        tree[(i−1)/2] = tree[i−1];    //i偶数, 对手左结点
    else tree[(i−1)/2] = tree[i+1]; //i奇数, 对手右结点
    i = (i−1) / 2;                          //向上调整
```

```
    while ( i ) {                                  //直到 i==0
        if ( i %2 == 0) j = i−1;
        else j = i+1;
        if ( !tree[i].active || !tree[j].active )
          if ( tree[i].active )
              tree[(i−1)/2] = tree[i];       //i可参选, i上
          else  tree[(i−1)/2] = tree[j];   //否则, j上
        else                          //两方都可参选
          if ( tree[i].data < tree[j].data )
             tree[(i−1)/2] = tree[i];       //关键码小者上
          else  tree[(i−1)/2] = tree[j];
         i = (i−1) / 2;      // i上升到双亲
    }
}
```

算法分析

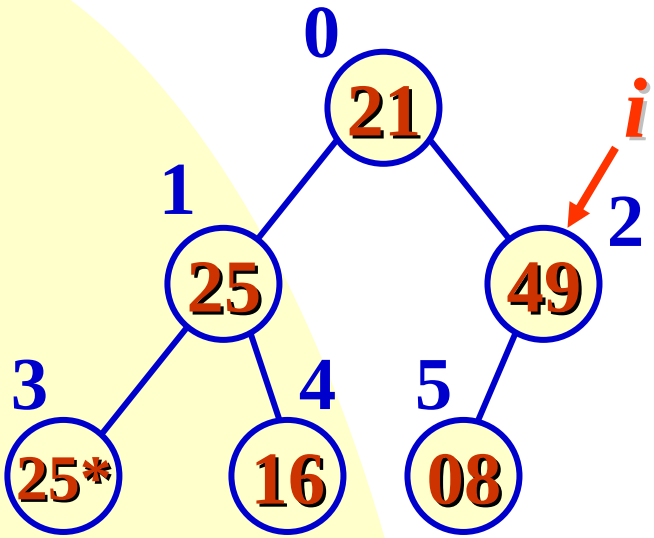
- 锦标赛排序构成的树是满的完全二叉树，其深度为$\lceil \log_2(n+1) \rceil$，其中 n 为待排序元素个数。
- 除第一次选择具有最小关键码的对象需要进行 $n-1$ 次关键码比较外，重构胜者树选择具有次小、再次小关键码对象所需的关键码比较次数均为$O(\log_2 n)$。总关键码比较次数为$O(n\log_2 n)$。
- 对象的移动次数不超过关键码的比较次数，所以锦标赛排序总的时间复杂度为$O(n\log_2 n)$。
- 这种排序方法虽然减少了许多排序时间，但是使用了较多的附加存储。

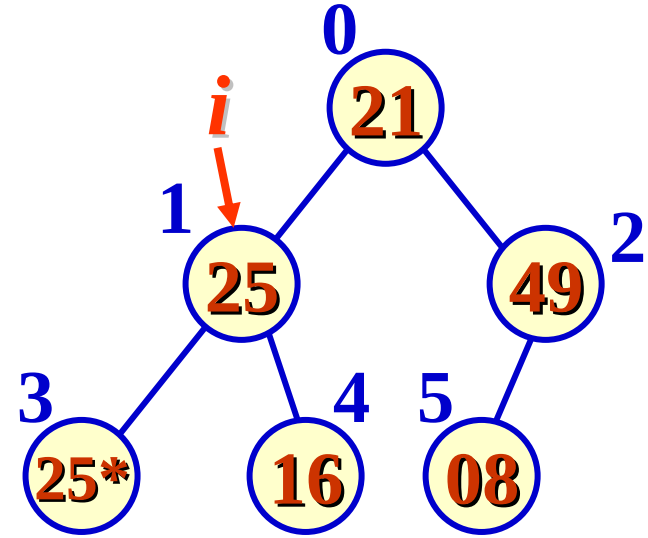
- 如果有 n 个对象，必须使用至少 $2n-1$ 个结点来存放胜者树。最多需要找到满足 $2^{k-1} < n \leq 2^k$ 的 k，使用 $2*2^k-1$ 个结点。每个结点包括关键码、对象序号和比较标志三种信息。
- 锦标赛排序是一个稳定的排序方法。

堆排序 (Heap Sort)

- 利用堆及其运算，可以很容易地实现选择排序的思路。
- 堆排序分为两个步骤：第一步，根据初始输入数据，利用堆的调整算法 *FilterDown( )* 形成初始堆，第二步，通过一系列的对象交换和重新调整堆进行排序。

建立初始的最大堆



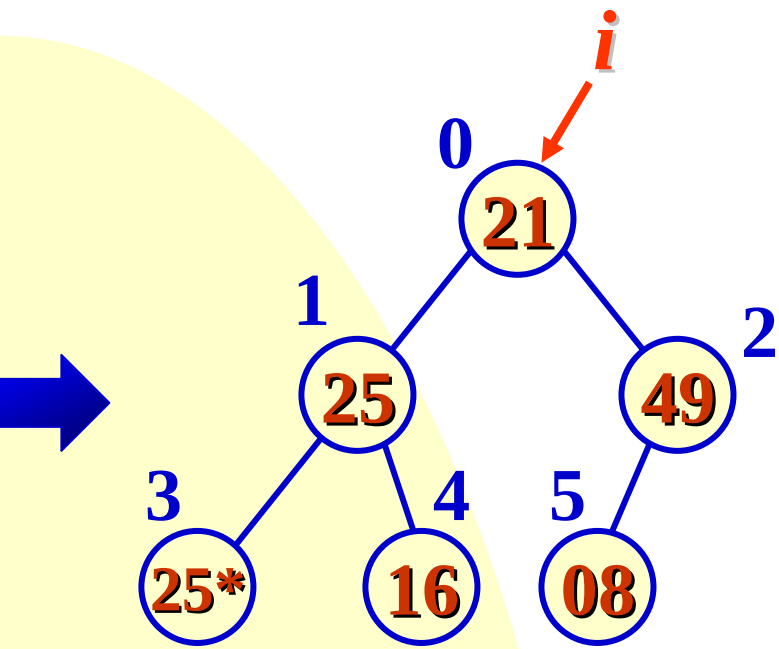


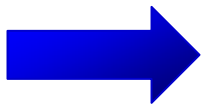
21	25	49	25*	16	08

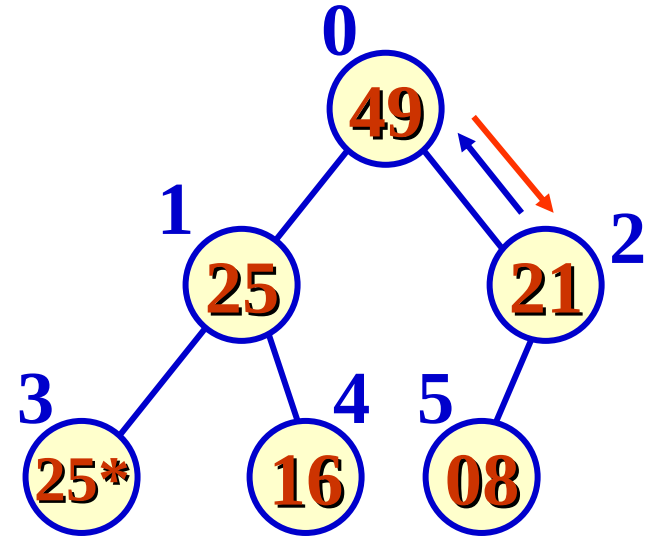
初始关键码集合

21	25	49	25*	16	08

$i = 2$ 时的局部调整







21	25	49	25*	16	08

i = 1 时的局部调整

49	25	21	25*	16	08

i = 0 时的局部调整
形成最大堆

最大堆的向下调整算法

```
template <class Type>
void MaxHeap<Type> ::
FilterDown ( const int i, const int EndOfHeap ) {
    int current = i;
    int child = 2*i+1;
    Type temp = heap[i];
    while ( child <= EndOfHeap ) {
        if ( child +1 < EndOfHeap &&
            heap[child].getKey( ) <
            heap[child+1].getKey( ) )
                child = child+1;
```

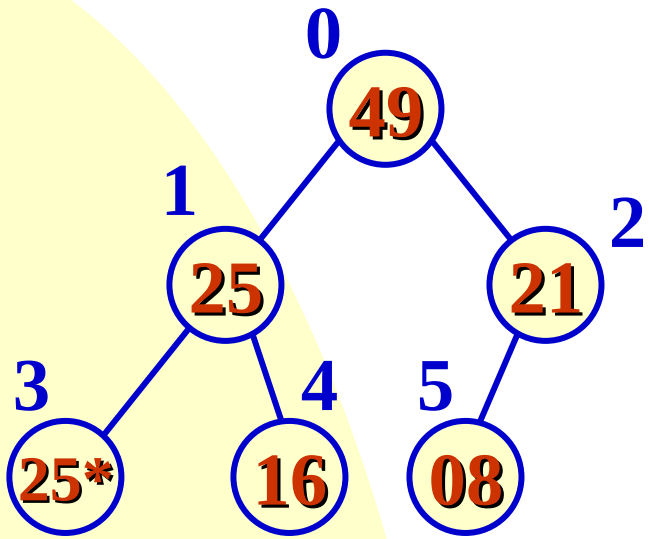
```
        if ( temp.getKey( ) >= heap[child].getKey( ) )
            break;
        else {
            heap[current] = heap[child];
            current = child;
            child = 2*child+1;
        }
    }
    heap[current] = temp;
}
```

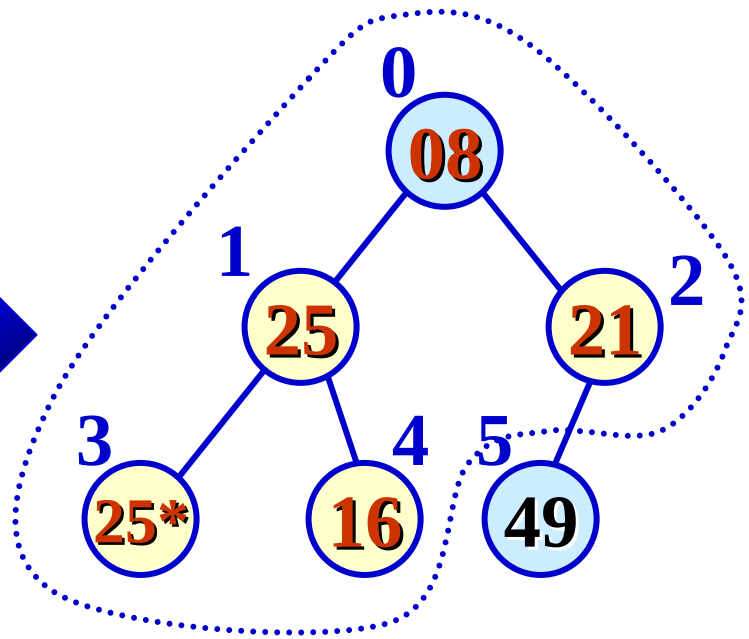
将表转换成堆

```
for ( int i = ( CurrentSize−2)/2 ;  i >= 0; i−− )
    FilterDown ( i, CurrentSize−1 );
```

基于初始堆进行堆排序

- 最大堆的第一个对象**V[0]**具有最大的关键码，将**V[0]**与**V[n]**对调，把具有最大关键码的对象交换到最后，再对前面的**n−1**个对象，使用堆的调整算法***FilterDown*(0, *n*−1)**，重新建立最大堆。结果具有次最大关键码的对象又上浮到堆顶，即**V[0]**位置。
- 再对调**V[0]**和**V[*n*−1]**，调用***FilterDown*(0, *n*−2)**，对前**n−2**个对象重新调整，….
- 如此反复执行，最后得到全部排序好的对象序列。这个算法即堆排序算法，其细节在下面的程序中给出。



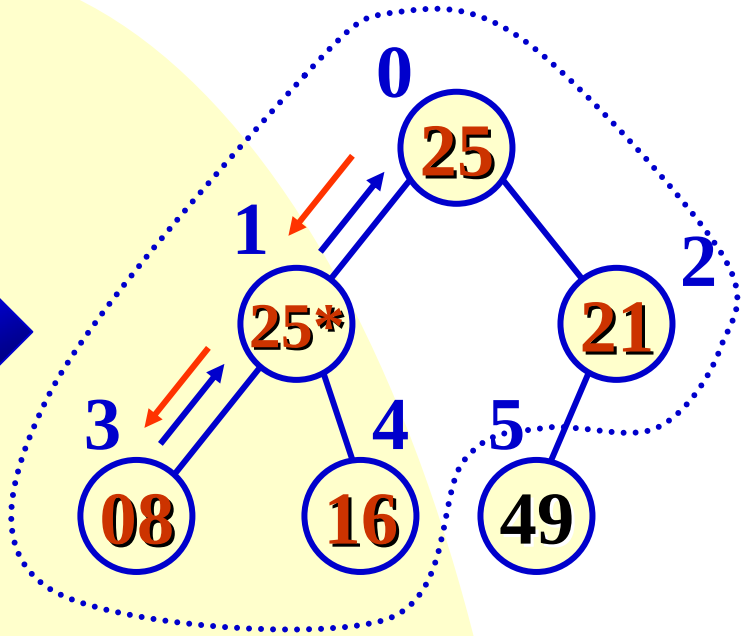


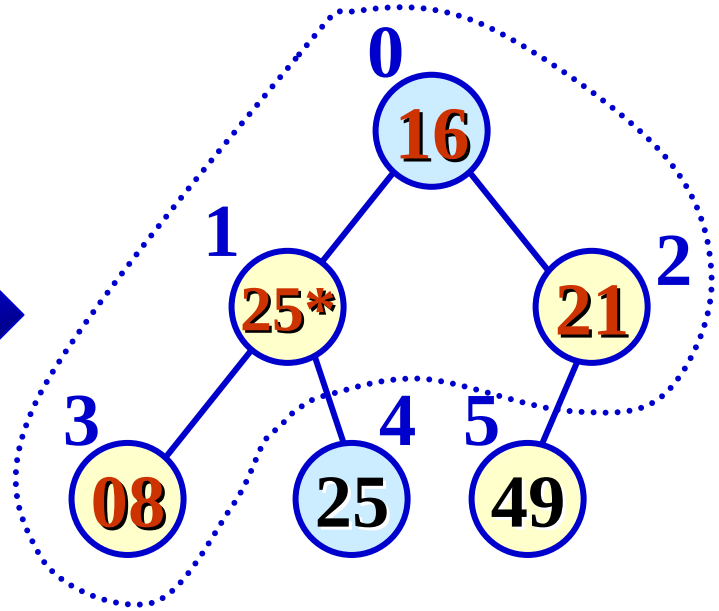
49	25	21	25*	16	08

初始最大堆

08	25	21	25*	16	49

交换 0 号与 5 号对象，
5 号对象就位



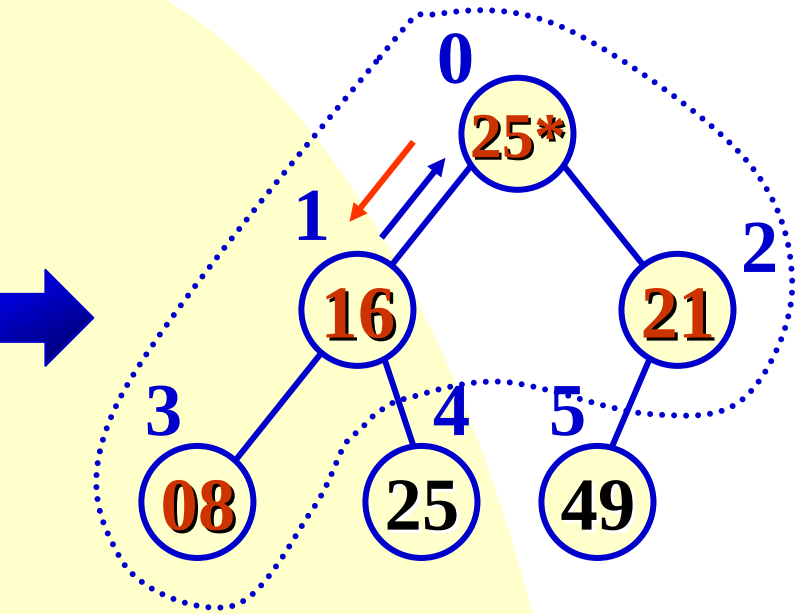


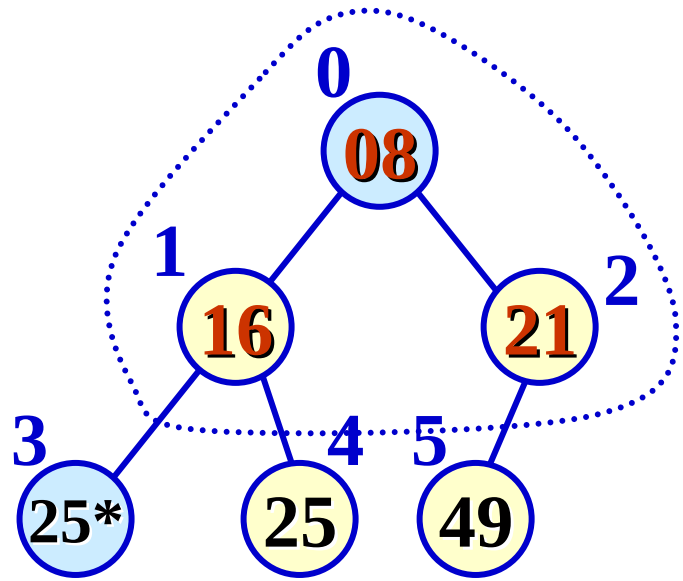
25	25*	21	08	16	49

从 0 号到 4 号 重新调整为最大堆

16	25*	21	08	25	49

交换 0 号与 4 号对象，4 号对象就位



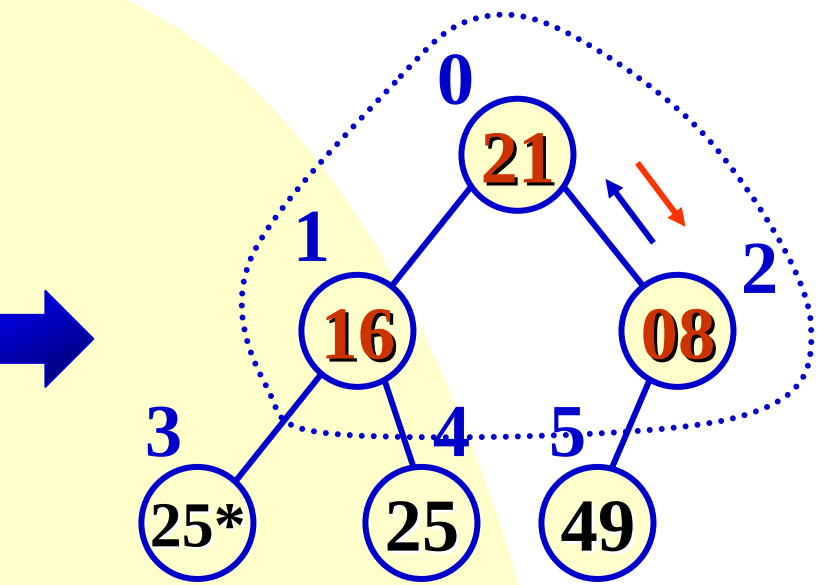


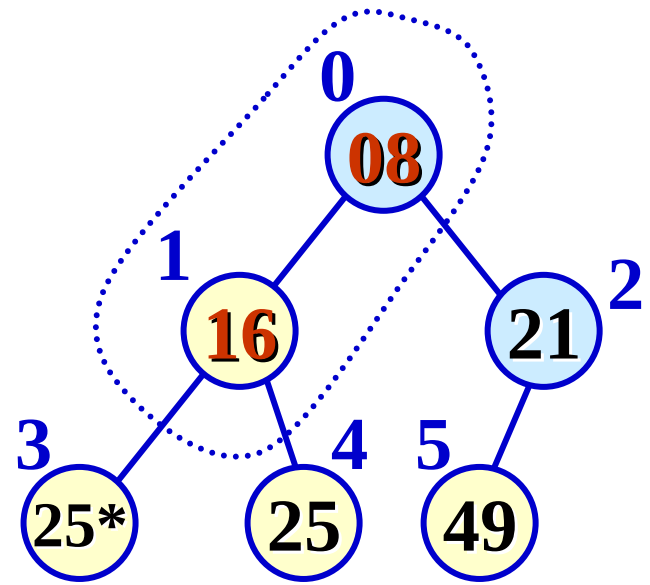
25*	16	21	08	25	49

08	16	21	25*	25	49

从 0 号到 3 号 重新调整为最大堆

交换 0 号与 3 号对象，3 号对象就位



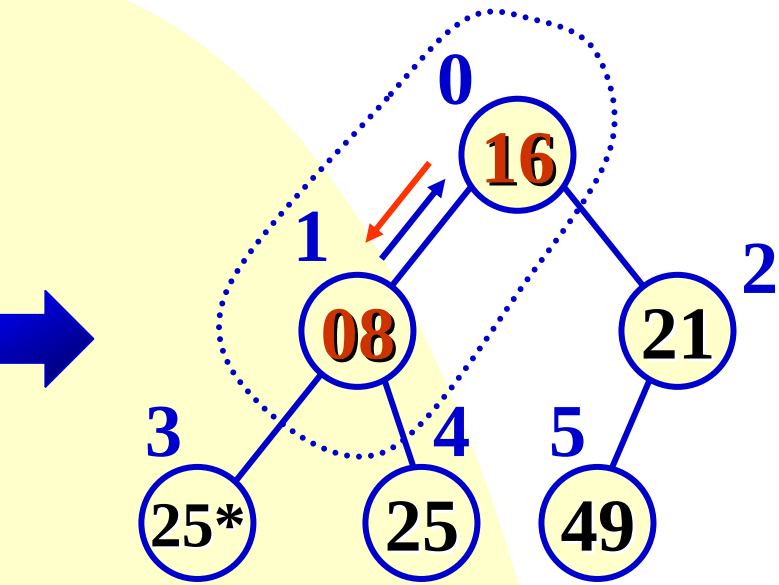


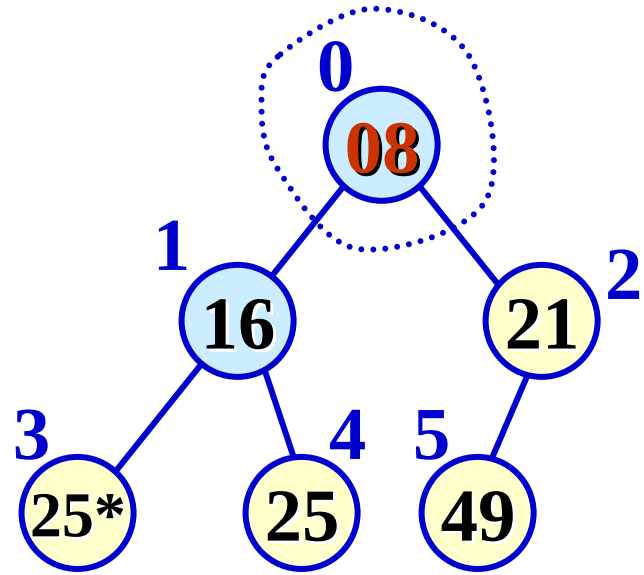
21	16	08	25*	25	49

08	16	21	25*	25	49

从 0 号到 2 号 重新调整为最大堆

交换 0 号与 2 号对象，2 号对象就位





16	08	21	25*	25	49

08	16	21	25*	25	49

从 0 号到 1 号 重新调整为最大堆

交换 0 号与 1 号对象，1 号对象就位

堆排序的算法

```
template <class Type> void MaxHeap<Type> ::
HeapSort ( ) {
//对表heap[0]到heap[n−1]进行排序,使得表中各
//个对象按其关键码非递减有序。
   for ( int i = ( CurrentSize−2 ) / 2; i >= 0; i−− )
      FilterDown ( i, CurrentSize−1 );        //初始堆
   for ( i = CurrentSize−1; i >= 1; i−−) {
      Swap ( heap[0],  heap[i] );              //交换
      FilterDown ( 0, i−1 );                   //重建最大堆
   }
}
```

算法分析

- 若设堆中有 n 个结点，且 $2^{k-1} \le n < 2^k$，则对应的完全二叉树有 k 层。在第 i 层上的结点数 $\le 2^i$ $(i = 0, 1, \ldots, k-1)$。在第一个形成初始堆的**for**循环中对每一个非叶结点调用了一次堆调整算法***FilterDown( )***，因此该循环所用的计算时间为：

$$2 \cdot \sum_{i=0}^{k-2} 2^i \cdot (k - i - 1)$$

- 其中，i 是层序号，2^i 是第 i 层的最大结点数，$(k-i-1)$是第 i 层结点能够移动的最大距离。

$$2\cdot\sum_{i=0}^{k-2}2^i\cdot(k-i-1)=2\cdot\sum_{j=1}^{k-1}2^{k-j-1}\cdot j=$$

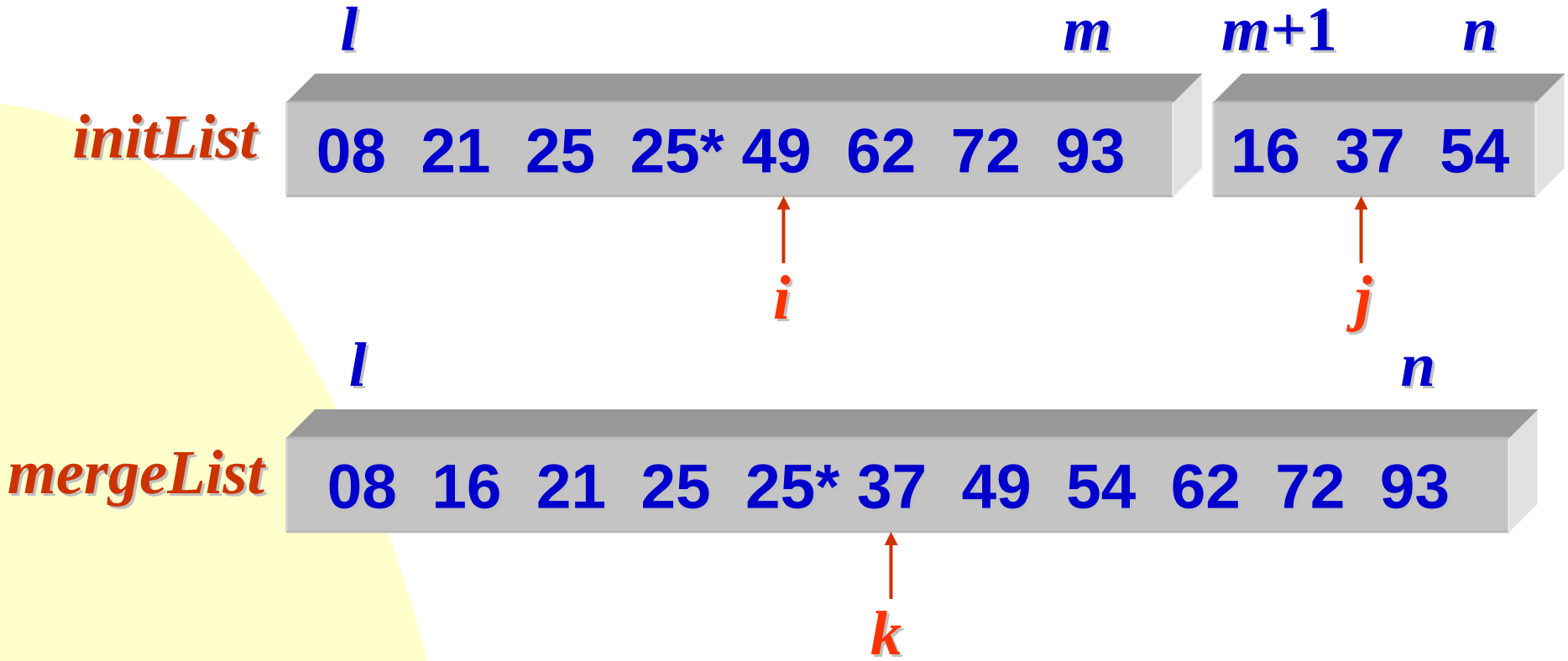
$$=2\cdot 2^{k-1}\sum_{j=1}^{k-1}\frac{j}{2^j}\le 2\cdot n\sum_{j=1}^{k-1}\frac{j}{2^j}<4n$$

- 在第二个**for**循环中，调用了$n-1$次*FilterDown( )*算法，该循环的计算时间为$O(n\log_2 n)$。因此，堆排序的时间复杂性为$O(n\log_2 n)$。
- 该算法的附加存储主要是在第二个**for**循环中用来执行对象交换时所用的一个临时对象。因此，该算法的空间复杂性为$O(1)$。
- 堆排序是一个不稳定的排序方法。

归并排序 (Merge Sort)

归并

- 归并，是将两个或两个以上的有序表合并成一个新的有序表。
- 对象序列 *initList* 中有两个有序表***V*[*l*] …*V*[*m*]**和***V*[*m*+1] …*V*[*n*]**。它们可归并成一个有序表，存于另一对象序列 *mergedList* 的***V*[*l*] …*V*[*n*]**中。
- 这种归并方法称为两路归并 **(2-way merging)**。
- 其基本思想是：设两个有序表**A**和**B**的对象个数**(**表长**)**分别为 ***al*** 和 ***bl***，变量 ***i*** 和 ***j*** 分别是表**A**和表**B**的当前检测指针。设表**C**是归并后的新有序表，变量 ***k*** 是它的当前存放指针。



- 当 *i* 和 *j* 都在两个表的表长内变化时，根据A[*i*]与B[*j*]的关键码的大小，依次把关键码小的对象排放到新表C[*k*]中；
- 当 *i* 与 *j* 中有一个已经超出表长时，将另一个表中的剩余部分照抄到新表C[*k*]中。

- 此算法的关键码比较次数和对象移动次数均为 $(m-l+1)+(n-m)=n-l+1$。

迭代的归并排序算法

- 迭代的归并排序算法就是利用两路归并过程进行排序的。其基本思想是:
- 假设初始对象序列有 n 个对象，首先把它看成是 n 个长度为 1 的有序子序列 (归并项)，先做两两归并，得到$\lceil n/2 \rceil$个长度为 2 的归并项 (如果 n 为奇数，则最后一个有序子序列的长度为 1)；再做两两归并，…，如此重复，最后得到一个长度为 n 的有序序列。

两路归并算法

```
template <class Type>
void merge ( datalist<Type> & initList,
        datalist <Type> & mergedList,  const int l,
        const int m,  const int n ) {
    int i = 0,  j = m+1,  k = 0;
    while ( i <= m && j <= n )        //两两比较
        if ( initList.Vector[i].getKey ( ) <=
                        initList.Vector[j].getKey ( ) ) {
            mergedList.Vector[k] = initList.Vector[i];
            i++;  k++;
        }
        else {
```

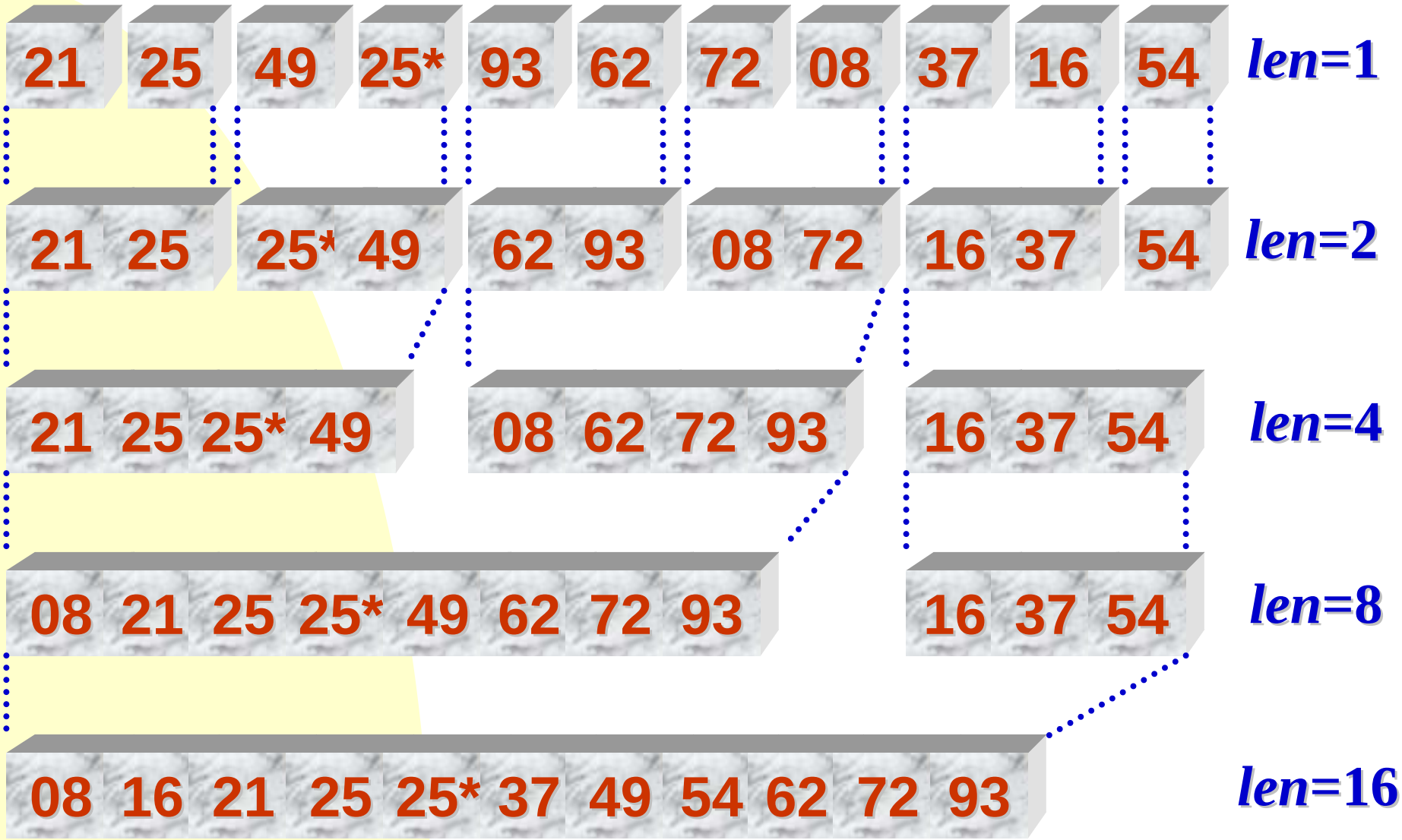
```
            mergedList.Vector[k] = initList.Vector[j];
            j++;  k++;
        }
    if ( i <= m )
        for ( int n1 = k, n2 = i;  n1 <= n && n2 <= m;
                                        n1++, n2++ )
            mergedList.Vector[n1] = initList.Vector[n2];
    else
        for ( int n1 = k, n2 = j;  n1 <= n && n2 <= n;
                                        n1++, n2++)
            mergedList.Vector[n1] = initList.Vector[n2];
}
```

一趟归并排序的情形

- 设数组*initList.Vector*[1]到*initList.Vector*[*n*]中的*n*个对象已经分为一些长度为*len*的归并项，将这些归并项两两归并，归并成一些长度为 2*len* 的归并项，结果放到*mergedList.Vector*中。
- 如果*n*不是2*len*的整数倍，则一趟归并到最后，可能遇到两种情形:
  - 剩下一个长度为*len*的归并项和另一个长度不足*len*的归并项，可用一次*merge*算法，将它们归并成一个长度小于2*len*的归并项。
  - 只剩下一个归并项，其长度小于或等于*len*，可将它直接抄到数组*MergedList.Vector*中。

```
template <class Type>
void MergePass ( datalist<Type> & initList,
     datalist<Type> & mergedList,  const int len ) {
   int i =0;
   while ( i+2*len <= initList.CurrentSize−1 ){
      merge ( initList,  mergedList,  i,  i+len−1,
                 i+2*len−1);
          i += 2 * len;
   }
   if ( i+len <= initList.CurrentSize−1 )
      merge ( initList,  mergedList,  i,  i+len−1,
                 initList.CurrentSize−1 );
   else for ( int j=i; j <= initList.CurrentSize−1; j++ )
          mergedList.Vector[j] = initList.Vector[j];
}
```

迭代的归并排序算法


21 25 49 25* 93 62 72 08 37 16 54
len=1
21 25 25* 49 62 93 08 72 16 37 54
len=2
21 25 25* 49 08 62 72 93 16 37 54
len=4
08 21 25 25* 49 62 72 93 16 37 54
len=8
08 16 21 25 25* 37 49 54 62 72 93
len=16

(两路)归并排序的主算法

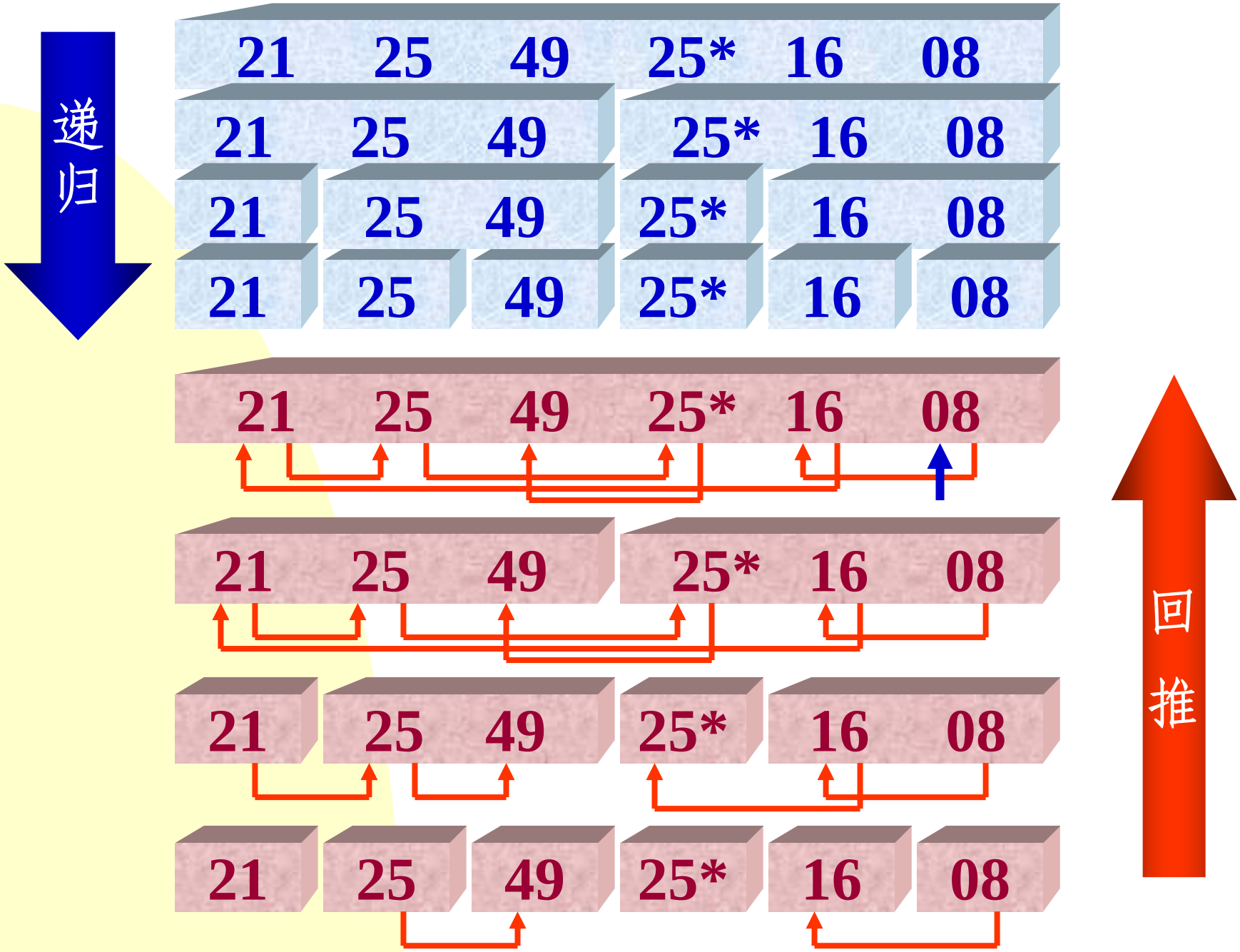
```
template <class Type>
void MergeSort ( datalist<Type> & list ) {
//按对象关键码非递减的顺序对表list中对象排序
   datalist<Type> & tempList ( list.MaxSize );
   int len = 1;
   while ( len < list.CurrentSize ) {
      MergePass ( list, tempList, len );   len *= 2;
      MergePass ( tempList, list, len );   len *= 2;
   }
   delete [ ]tempList;
}
```

算法分析

- 在迭代的归并排序算法中，函数 *MergePass*( ) 做一趟两路归并排序，要调用*merge* ( )函数 $\lceil n/(2*len) \rceil \approx O(n/len)$ 次，函数*MergeSort*( )调用*MergePass*( )正好$\lceil \log_2 n \rceil$ 次，而每次*merge*( )要执行比较$O(len)$次，所以算法总的时间复杂度为$O(n\log_2 n)$。
- 归并排序占用附加存储较多，需要另外一个与原待排序对象数组同样大小的辅助数组。这是这个算法的缺点。
- 归并排序是一个稳定的排序方法。

递归的表归并排序

- 与快速排序类似，归并排序也可以利用划分为子序列的方法递归实现。
- 在递归的归并排序方法中，首先要把整个待排序序列划分为两个长度大致相等的部分，分别称之为左子表和右子表。对这些子表分别递归地进行排序，然后再把排好序的两个子表进行归并。
- 图示：待排序对象序列的关键码为 **{ 21, 25, 49, 25*,16, 08 }**，先是进行子表划分，待到子表中只有一个对象时递归到底。再是实施归并，逐步退出递归调用的过程。


递归
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
21 25 49 25* 16 08
回推

- 归并时采用静态链表的存储表示，可以得到一种有效的归并排序算法。

静态链表的两路归并算法

```
template <class Type>
int ListMerge ( staticlinklist<Type> & list,
                const int start1,  const int start2 ) {
    int k = 0,  i = start1,  j = start2;
    while ( i && j )
        if ( list.Vector[i].getKey( ) <=
                        list.Vector[j].getKey( ) ) {
            list.Vector[k].setLink(i);  k = i;
            i = list.Vector[i].getLink( );
        }
```

```
        else {
            list.Vector[k].setLink(j);  k = j;
            j = list.Vector[j].getLink( );
        }
        if ( ! i ) list.Vector[k].setLink(j);
        else  list.Vector[k].setLink(i);
        return list.Vector[0].getLink( );
}
```

递归的归并排序算法

```
template <class Type>
int rMergeSort ( staticlinklist<Type> & list,
            const int left,  const int right ) {
```

```
    if ( left >= right ) return left;
    int middle = ( left + right ) / 2;
    return ListMerge ( list,
                rMergeSort ( list, left, middle ),
                rMegerSort ( list, middle+1, right ) );
    //以中点middle为界, 分别对左半部和右半部进
    //行表归并排序
}
```

算法分析

- 链表的归并排序方法的递归深度为$O(\log_2 n)$，对象关键码的比较次数为$O(n\log_2 n)$。
- 链表的归并排序方法是一种稳定的排序方法。

基数排序 (Radix Sort)

- 基数排序是采用“分配”与“收集”的办法，用对多关键码进行排序的思想实现对单关键码进行排序的方法。

多关键码排序

- 以扑克牌排序为例。每张扑克牌有两个“关键码”：花色和面值。其有序关系为：
  - 花色：♣ < ♦ < ♥ < ♠
  - 面值：2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A
- 如果我们把所有扑克牌排成以下次序：

♦ ♣ 2, …, ♣ A, ♦ 2, …, ♦ A, ♥ 2, …, ♥ A, ♠ 2, …, ♠ A

- 这就是多关键码排序。排序后形成的有序序列叫做词典有序序列。
- 对于上例两关键码的排序，可以先按花色排序，之后再按面值排序；也可以先按面值排序，再按花色排序。
- 一般情况下，假定有一个 n 个对象的序列 $\{V_0, V_1, \cdots, V_{n-1}\}$，且每个对象 V_i 中含有 d 个关键码 $\left(K_i^1, K_i^2, \cdots, K_i^d\right)$
- 如果对于序列中任意两个对象 V_i 和 V_j ($0 \le i < j \le n-1$) 都满足:

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right) < \left(K_j^1,\ K_j^2,\ \cdots,\ K_j^d\right)$$

- 则称序列对关键码 $(K^1, K^2, \ldots, K^d)$ 有序。其中，K^1 称为最高位关键码，K^d 称为最低位关键码。
- 如果关键码是由多个数据项组成的数据项组，则依据它进行排序时就需要利用多关键码排序。
- 实现多关键码排序有两种常用的方法
  - 最高位优先**MSD (Most Significant Digit first)**
  - 最低位优先**LSD (Least Significant Digit first)**
- 最高位优先法通常是一个递归的过程:
  - 先根据最高位关键码K^1排序,得到若干对象组，对象组中每个对象都有相同关键码K^1。

- 再分别对每组中对象根据关键码K^2进行排序，按K^2值的不同，再分成若干个更小的子组，每个子组中的对象具有相同的K^1和K^2值。
- 依此重复，直到对关键码K^d完成排序为止。
- 最后，把所有子组中的对象依次连接起来，就得到一个有序的对象序列。

- 最低位优先法首先依据最低位关键码K^d对所有对象进行一趟排序，再依据次低位关键码K^{d-1}对上一趟排序的结果再排序，依次重复，直到依据关键码K^1最后一趟排序完成，就可以得到一个有序的序列。使用这种排序方法对每一个关键码进行排序时，不需要再分组，而是整个对象组都参加排序。

- LSD和MSD方法也可应用于对一个关键码进行的排序。此时可将单关键码 K_i 看作是一个子关键码组：

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right)$$

链式基数排序

- 基数排序是典型的LSD排序方法，利用“分配”和“收集”两种运算对单关键码进行排序。在这种方法中，把单关键码 K_i 看成是一个 d 元组：

$$\left(K_i^1,\ K_i^2,\ \cdots,\ K_i^d\right)$$

- 其中的每一个分量 $K_i^j\ (1 \le j \le d)$ 也可看成是一个关键码。

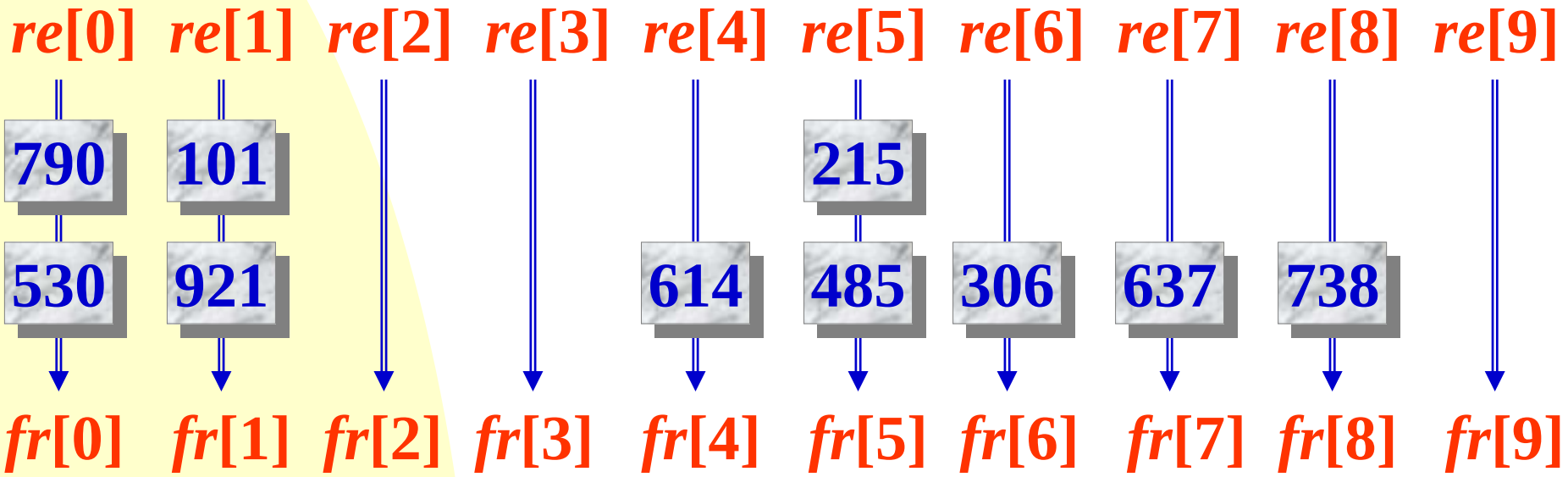
- 分量 $K_i^j(1 \le j \le d)$ 有*radix*种取值，则称*radix*为基数。例如，关键码**984**可以看成是一个**3**元组**(9, 8, 4)**，每一位有**0, 1, …, 9**等**10**种取值，基数 ***radix* = 10**。关键码**‘data’**可以看成是一个**4**元组**(d, a, t, a)**，每一位有**‘a’, ‘b’, …, ‘z’**等**26**种取值，***radix* = 26**。
- 针对*d*元组中的每一位分量，把对象序列中的所有对象，按 K_i^j 的取值，先“分配”到*rd*个队列中去。然后再按各队列的顺序，依次把对象从队列中“收集”起来，这样所有对象按取值 K_i^j 排序完成。

- 如果对于所有对象的关键码$K_0, K_1, \ldots, K_{n-1}$，依次对各位的分量，让 $j = d, d-1, \ldots, 1$，分别用这种“分配”、“收集”的运算逐趟进行排序，在最后一趟“分配”、“收集”完成后，所有对象就按其关键码的值从小到大排好序了。
- 各队列采用链式队列结构，分配到同一队列的关键码用链接指针链接起来。每一队列设置两个队列指针： **int *front* [*radix*]**指示队头， **int *rear* [*radix*]** 指向队尾。
- 为了有效地存储和重排 ***n*** 个待排序对象，以静态链表作为它们的存储结构。在对象重排时不必移动对象，只需修改各对象的链接指针即可。

基数排序的“分配”与“收集”过程　第一趟

614 → 738 → 921 → 485 → 637 → 101 → 215 → 530 → 790 → 306

第一趟分配（按最低位 $i = 3$ ）



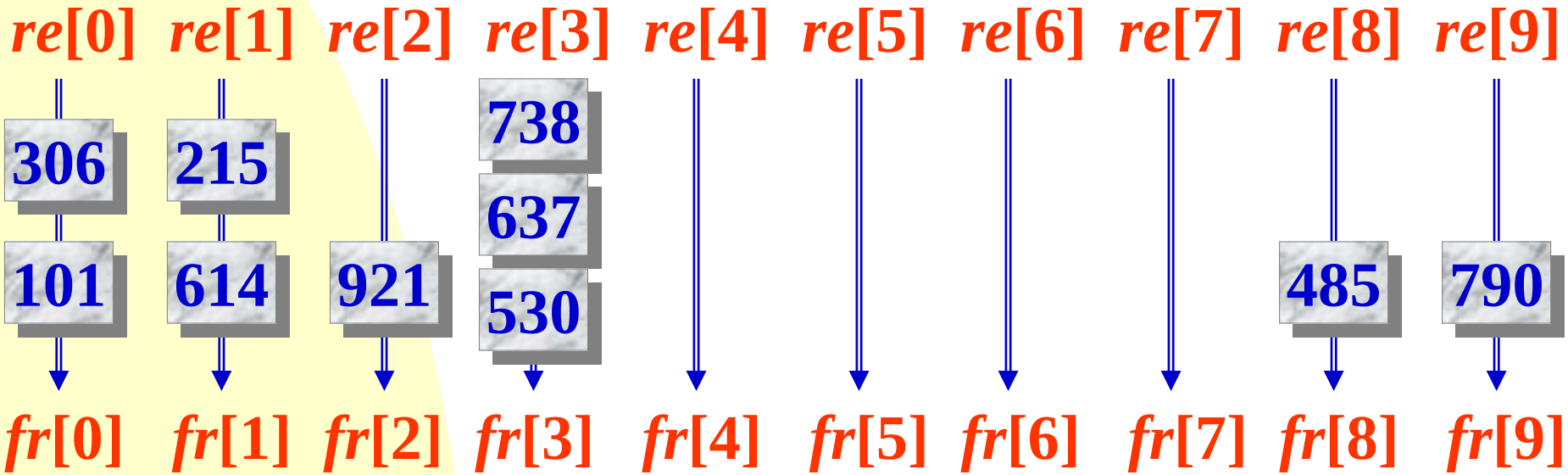
第一趟收集

530 → 790 → 921 → 101 → 614 → 485 → 215 → 306 → 637 → 738

基数排序的“分配”与“收集”过程　第二趟

530 → 790 → 921 → 101 → 614 → 485 → 215 → 306 → 637 → 738

第二趟分配（按次低位 $i = 2$）



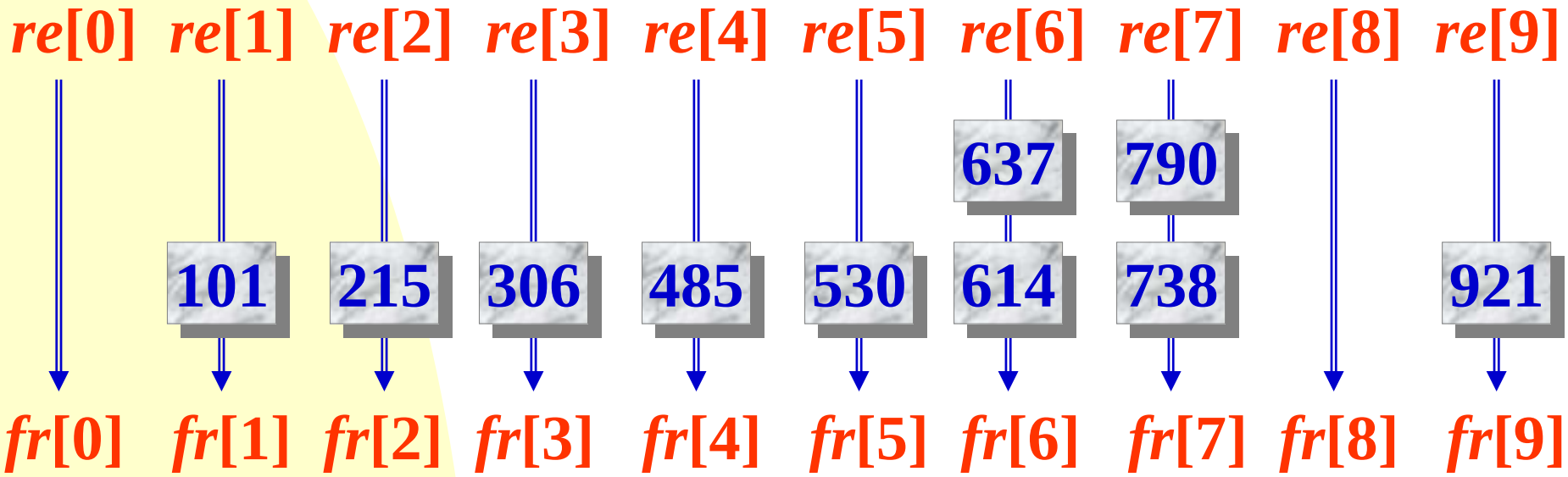
第二趟收集

101 → 306 → 614 → 215 → 921 → 530 → 637 → 738 → 485 → 790

基数排序的“分配”与“收集”过程　第三趟

101 → 306 → 614 → 215 → 921 → 530 → 637 → 738 → 485 → 790

第三趟分配（按最高位 $i = 1$ ）



第三趟收集

101 → 215 → 306 → 485 → 530 → 614 → 637 → 738 → 790 → 921

链表基数排序

```
template <class Type>
void RadixSort ( staticlinklist<Type> &list,
        const int d,  const int radix ) {
    int rear[radix],  front[radix];
    for ( int i = 1; i < list.CurrentSize; i++ )
        list.Vector[i].setLink(i+1);
    list.Vector[n].setLink(0);          //静态链表初始化
    int current = 1;                    //链表扫描指针
    for ( i = d−1; i >= 0; i−− ) {      //做 d 趟分配.收集
        for ( int j = 0; j < radix; j++ )  front[j] = 0;
        while ( current != 0 ) {        //逐个对象分配
```

```
        int k = list.Vector[current].getKey(key[i]);
        //取当前对象关键码的第 i 位
        if ( front[k] == 0)     //原链表为空,对象链入
            front[k] = current;
        else                              //原链表非空,链尾链入
            list.Vector[rear[k]].setLink (current);
        rear[k] = current;      //修改链尾指针
        current = list.Vector[current].getLink ( );
    }
    j = 0;              //从0号队列开始收集
    while ( front[j] == 0 ) j++;     //空队列跳过
    current = front[j];
```

```
        list.Vector[0].setLink (current);
        int last = rear[j];
        for ( k = j+1; k < radix; k++)  //逐个队列链接
            if ( front[k] ) {
                list.Vector[last].setLink(front[k]);
                last = rear[k];
            }
        list.Vector[last].setLink(0);
    }
}
```

算法分析

- 若每个关键码有d 位，需要重复执行d 趟“分配”与“收集”。每趟对 n 个对象进行“分配”，对$radix$个队列进行“收集”。总时间复杂度为$O(d(n+radix))$。
- 若基数$radix$相同，对于对象个数较多而关键码位数较少的情况，使用链式基数排序较好。
- 基数排序需要增加$n+2radix$个附加链接指针。
- 基数排序是稳定的排序方法。

外排序

当待排序的对象数目特别多时，在内存中不能一次处理。必须把它们以文件的形式存放于外存，排序时再把它们一部分一部分调入内存进行处理。这样，在排序过程中必须不断地在内存与外存之间传送数据。这种基于外部存储设备（或文件）的排序技术就是外排序。

外排序的基本过程

- 当对象以文件形式存放于磁盘上的时候，通常是按物理块存储的。
- 物理块也叫做页块，是磁盘存取的基本单位。

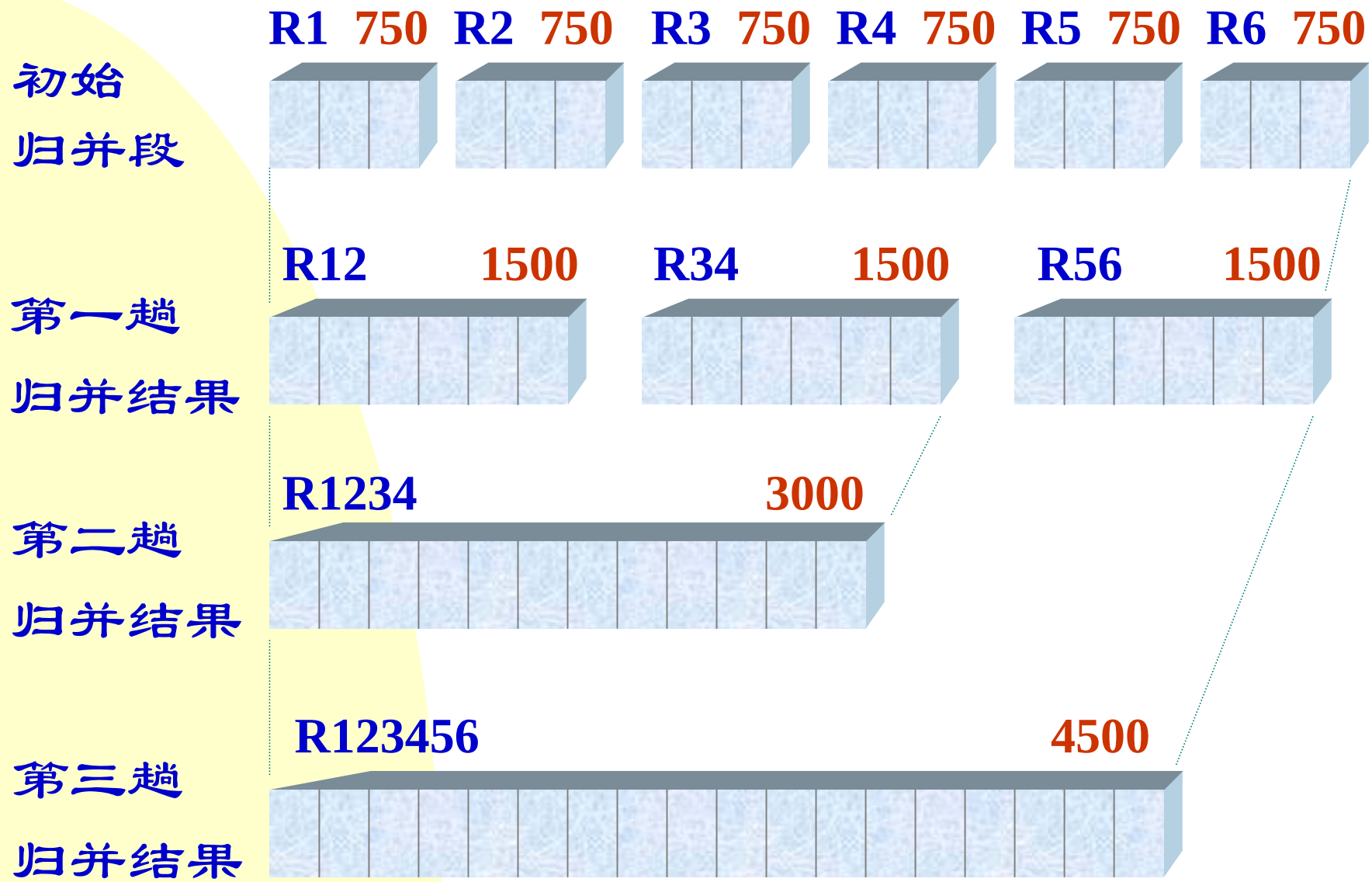
- 每个页块可以存放几个对象。操作系统按页块对磁盘上的信息进行读写。
- 本节所指的磁盘是由若干片磁盘组成的磁盘组，各个盘片安装在同一主轴上高速旋转。各个盘面上半径相同的磁道构成了柱面。各盘面设置一个读写磁头，它们装在同一动臂上，可以径向从一个柱面移到另一个柱面上。
- 为了访问某一页块，先寻找柱面，移动臂使读写磁头移到指定柱面上：寻查 (seek)。再根据磁道号(盘面号)选择相应读写磁头，等待指定页块转到读写磁头下：等待(latency)。因此，在磁盘组上存取一个页块的时间：

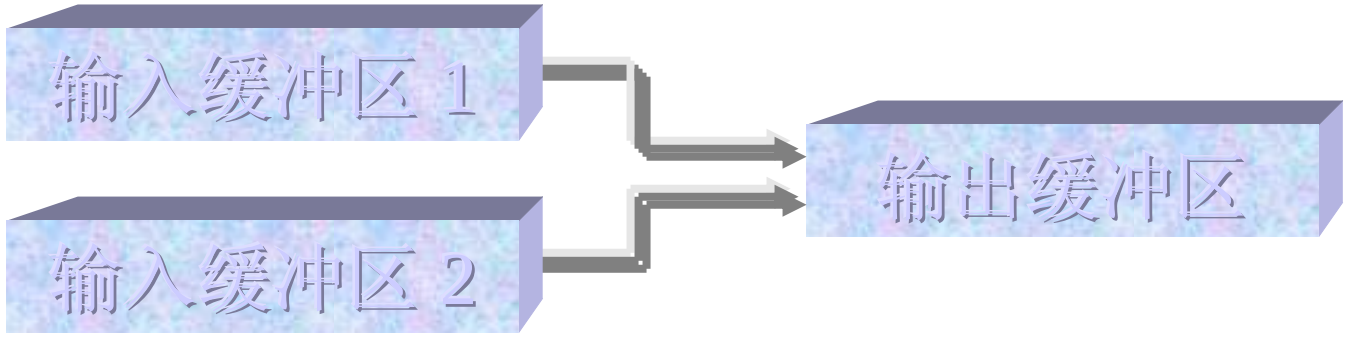
$$t_{io} = t_{seek} + t_{latency} + t_{rw}$$

- 基于磁盘进行的排序多使用归并排序方法。其排序过程主要分为两个阶段:
  - 第一个阶段建立用于外排序的内存缓冲区。根据它们的大小将输入文件划分为若干段，用某种内排序方法对各段进行排序。这些经过排序的段叫做初始归并段或初始顺串**(Run)**。当它们生成后就被写到外存中去。
  - 第二个阶段仿照内排序中所介绍过的归并树模式，把第一阶段生成的初始归并段加以归并，一趟趟地扩大归并段和减少归并段个数，直到最后归并成一个大归并段(有序文件)为止。

- 示例：设有一个包含4500个对象的输入文件。现用一台其内存至多可容纳750个对象的计算机对该文件进行排序。输入文件放在磁盘上，磁盘每个页块可容纳250个对象，这样全部对象可存储在 4500 / 250 = 18 个页块中。输出文件也放在磁盘上，用以存放归并结果。
- 由于内存中可用于排序的存储区域能容纳750个对象, 因此内存中恰好能存3个页块的对象。
- 在外归并排序一开始，把18块对象，每3块一组，读入内存。利用某种内排序方法进行内排序, 形成初始归并段, 再写回外存。总共可得到6个初始归并段。然后一趟一趟进行归并排序。

两路归并排序的归并树


R1 750
R2 750
R3 750
R4 750
R5 750
R6 750
初始
归并段
R12 1500
R34 1500
R56 1500
第一趟
归并结果
R1234 3000
第二趟
归并结果
R123456 4500
第三趟
归并结果



- 若把内存区域等份地分为 3 个缓冲区。其中的两个为输入缓冲区，一个为输出缓冲区，可以在内存中利用简单 2 路归并函数 ***merge*** 实现 2 路归并。
- 首先, 从参加归并排序的两个输入归并段 R_1 和 R_2 中分别读入一块，放在输入缓冲区1 和输入缓冲区2 中。然后，在内存中进行2路归并，归并出来的对象顺序存放到输出缓冲区中。

- 一般地，若总对象个数为 n，磁盘上每个页块可容纳 b 个对象，内存缓冲区可容纳 i 个页块，则每个初始归并段长度为 $len = i * b$，可生成 $m = \lceil n / len \rceil$ 个等长的初始归并段。
- 在做2路归并排序时，第一趟从 m 个初始归并段得到 $\lceil m/2 \rceil$ 个归并段，以后各趟将从 l ($l > 1$) 个归并段得到 $\lceil l/2 \rceil$ 个归并段。总归并趟数等于归并树的高度 $\lceil \log_2 m \rceil$。
- 根据 2 路归并树, 估计 2 路归并排序时间 t_{ES} 的上界为:

$$t_{ES} = m*t_{IS} + d*t_{IO} + S*u*t_{mg}$$

- 对4500个对象进行排序的例子，各种操作的计算时间如下：
  - 读18个输入块, 内部排序6段, 写18个输出块
    $= 6\ t_{IS} + 36\ t_{IO}$
  - 成对归并初始归并段 $R_1 \sim R_6$
    $= 36\ t_{IO} + 4500\ t_{mg}$
  - 归并两个具有1500个对象的归并段R_{12}和R_{34}
    $= 24\ t_{IO} + 3000\ t_{mg}$
  - 最后将 R_{1234} 和 R_{56} 归并成一个归并段
    $= 36\ t_{IO} + 4500\ t_{mg}$
- 合计 $t_{ES} = 6\ t_{IS} + 132\ t_{IO} + 12000\ t_{mg}$

- 由于 $t_{IO} = t_{seek} + t_{latency} + t_{rw}$，其中，$t_{seek}$和$t_{latency}$是机械动作，而$t_{rw}$、$t_{IS}$、$t_{mg}$是电子线路的动作，所以 $t_{IO} >> t_{IS}$，$t_{IO} >> t_{mg}$。想要提高外排序的速度，应着眼于减少 d。
- 若对相同数目的对象，在同样页块大小的情况下做 3 路归并或做 6 路归并(当然, 内存缓冲区的数目也要变化)，则可做大致比较:

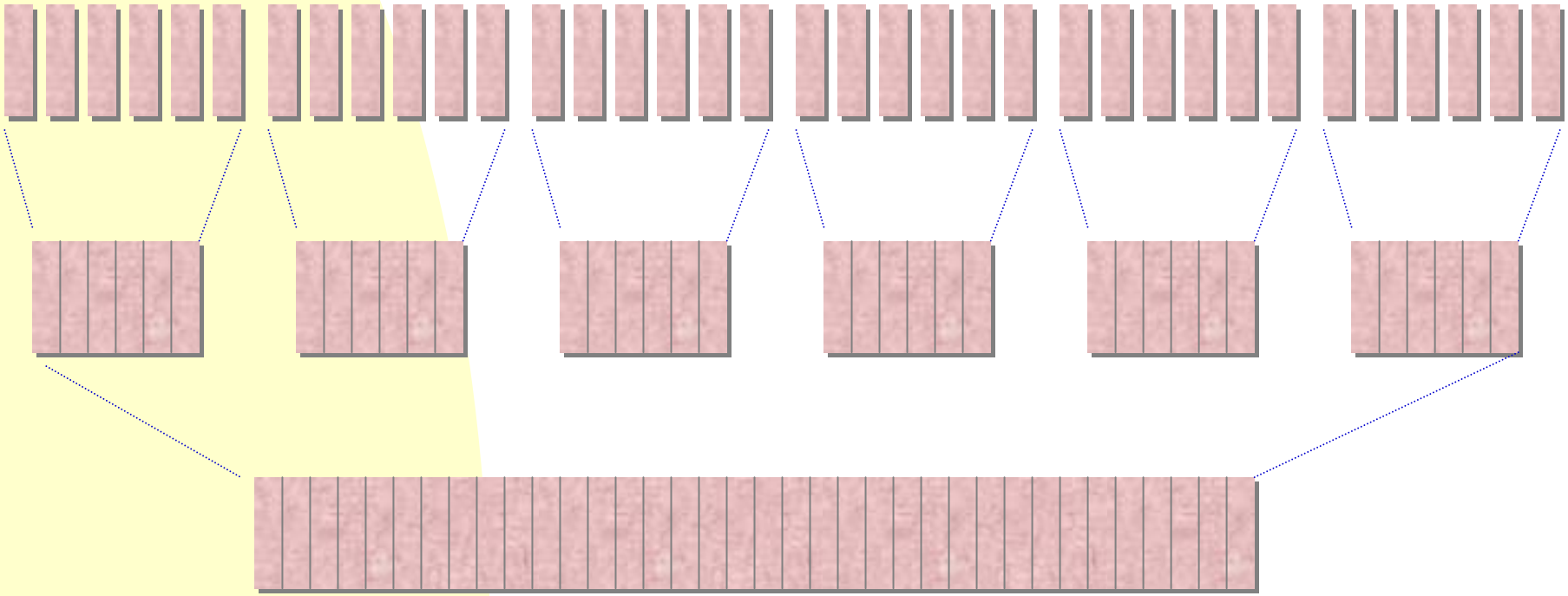
归并路数 k	总读写磁盘次数 d	归并趟数 S
2	132	3
3	108	2
6	72	1

- 因此，增大归并路数，可减少归并趟数，从而减少总读写磁盘次数d。

- 一般, 对 m 个初始归并段, 做 k 路平衡归并, 归并树可用正则 k 叉树(即只有度为 k 与度为0的结点的 k 叉树)来表示。
- 第一趟可将 m 个初始归并段归并为 $l=\lceil m/k \rceil$ 个归并段，以后每一趟归并将 l 个归并段归并成 $l=\lceil l/k \rceil$ 个归并段，直到最后形成一个大的归并段为止。树的高度 = $\lceil \log_k m \rceil$ = 归并趟数 S。
- 只要增大归并路数 k，或减少初始归并段个数 m，都能减少归并趟数 S，以减少读写磁盘次数 d，达到提高外排序速度的目的。
- 采用输入缓冲区、内部归并和输出缓冲区并行处理的方法，也能有效地提高外排序的速度。

k路平衡归并 (k-way Balanced merging)

- 做 k 路平衡归并时，如果有 m 个初始归并段，则相应的归并树有$\lceil \log_k m \rceil + 1$ 层，需要归并$\lceil \log_k m \rceil$趟。下图给出对有36个初始归并段的文件做6路平衡归并时的归并树。



- 做内部 k 路归并时，在 k 个对象中选择最小者，需要顺序比较 k-1 次。每趟归并 u 个对象需要做$(u\text{-}1)*(k\text{-}1)$次比较，$S$ 趟归并总共需要的比较次数为：

$$S*(u\text{-}1)*(k\text{-}1) = \lceil \log_k m \rceil * (u\text{-}1) * (k\text{-}1)$$
$$= \lceil \log_2 m \rceil * (u\text{-}1) * (k\text{-}1) / \lceil \log_2 k \rceil$$

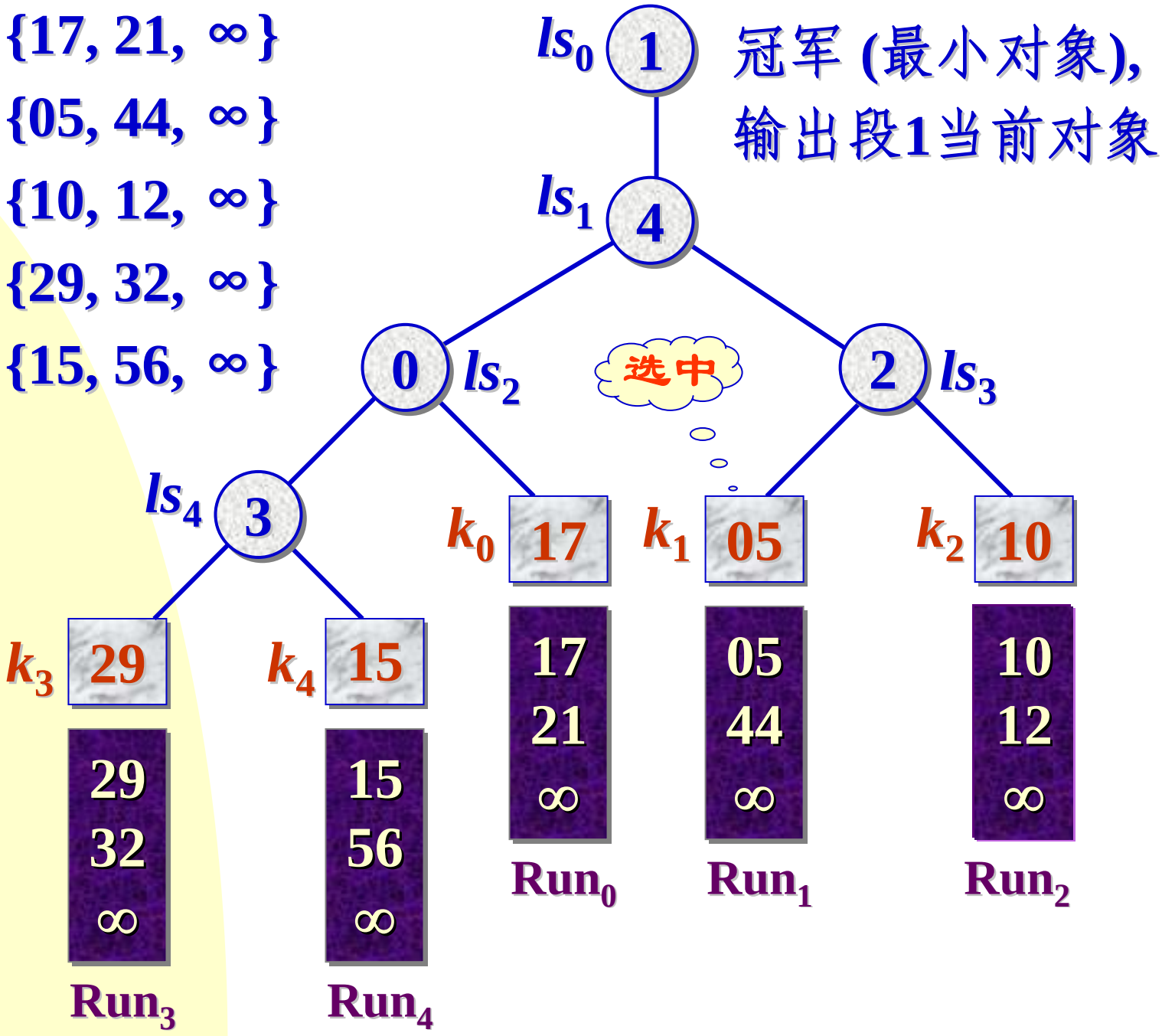
- 在初始归并段个数 m 与对象个数 u 一定时，$\lceil \log_2 m \rceil*(u\text{-}1) = \text{const}$，而 $(k\text{-}1) / \lceil \log_2 k \rceil$ 在 k 增大时趋于无穷大。因此，增大归并路数 k，会使得内部归并的时间增大。
- 使用“败者树”从 k 个归并段中选最小者，当 k 较大时 $(k \geq 6)$，选出关键码最小的对象只需比较 $\lceil \log_2 k \rceil$ 次。

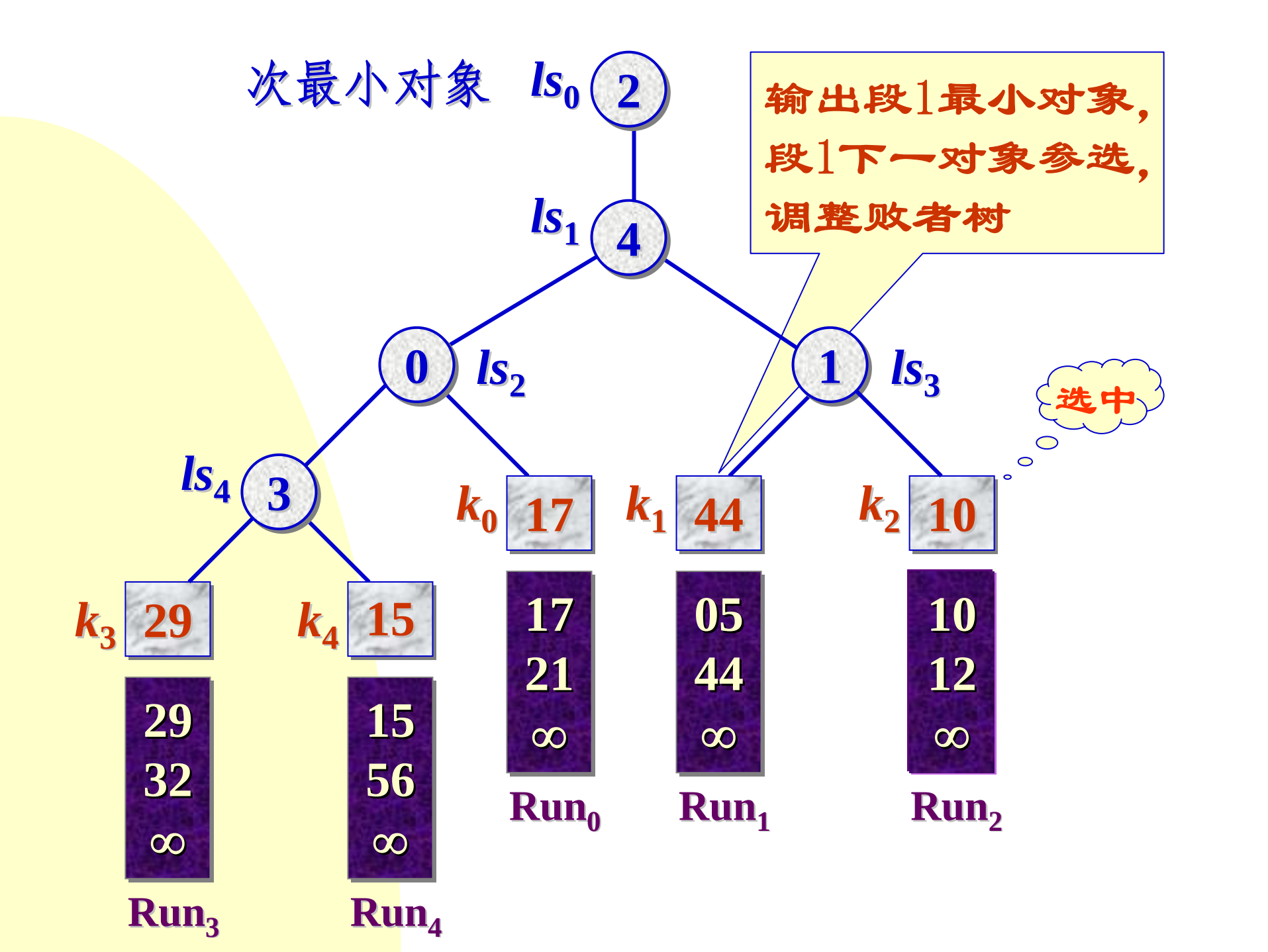
$$S*(u\text{-}1)*\lceil \log_2 k \rceil = \lceil \log_k m \rceil * (u\text{-}1) * \lceil \log_2 k \rceil$$
$$= \lceil \log_2 m \rceil * (u\text{-}1) * \lceil \log_2 k \rceil / \lceil \log_2 k \rceil$$
$$= \lceil \log_2 m \rceil * (u\text{-}1)$$

- 关键码比较次数与 ***k*** 无关，总的内部归并时间不会随 ***k*** 的增大而增大。
- 因此，只要内存空间允许，增大归并路数 ***k***，将有效地减少归并树深度，从而减少读写磁盘次数 ***d***，提高外排序的速度。
- 下面讨论利用败者树在 ***k*** 个输入归并段中选择最小者，实现归并排序的方法。

- 败者树是一棵正则的完全二叉树。其中
  - 每个叶结点存放各归并段在归并过程中当前参加比较的对象;
  - 每个非叶结点记忆它两个子女结点中对象关键码小的结点(即败者);
- 因此，根结点中记忆树中当前对象关键码最小的结点 (最小对象)。
- 败者树与胜者树的区别在于一个选择了败者(关键码大者)，一个选择了胜者(关键码小者)。
- 示例：设有5个初始归并段，它们中各对象的关键码分别是:

Run0: {17, 21, ∞}
Run1: {05, 44, ∞}
Run2: {10, 12, ∞}
Run3: {29, 32, ∞}
Run4: {15, 56, ∞}




次最小对象
ls_0 2
ls_1 4
0
ls_2
1
ls_3
ls_4 3
k_0 17
k_1 44
k_2 10
k_3 29
k_4 15
17
21
∞
05
44
∞
10
12
∞
29
32
∞
15
56
∞
Run_0
Run_1
Run_2
Run_3
Run_4
输出段l最小对象，
段l下一对象参选，
调整败者树
选中

- 败者树的高度为$\lceil \log_2 k \rceil$，在每次调整，找下一个具有最小关键码对象时，最多做$\lceil \log_2 k \rceil$次关键码比较。
- 在内存中应为每一个归并段分配一个输入缓冲区，其大小应能容纳一个页块的对象，编号与归并段号一致。每个输入缓冲区应有一个指针，指示当前参加归并的对象。
- 在内存中还应设立一个输出缓冲区，其大小相当于一个页块大小。它也有一个缓冲区指针，指示当前可存放结果对象的位置。每当一个对象 *i* 被选出，就执行**OutputRecord(*i*)**操作，将对象按输出缓冲区指针所指位置存放到输出缓冲区中。

- 在实现利用败者树进行多路平衡归并算法时，把败者树的叶结点和非叶结点分开定义。
- 败者树叶结点***key*[*k*]**有***k*+1**个，***key*[0]**到***key*[*k*-1]**存放各归并段当前参加归并的对象的关键码，***key*[*k*]**是辅助工作单元，在初始建立败者树时使用，存放一个最小的在各归并段中不可能出现的关键码：**−MaxNum**。
- 败者树非叶结点***loser*[*k*−1]**有 ***k*** 个，其中***loser*[1]**到***loser*[*k*−1]**存放各次比较的败者的归并段号，***loser*[0]**中是最后胜者所在的归并段号。另外还有一个对象数组 ***r*[*k*]**，存放各归并段当前参加归并的对象。

k 路平衡归并排序算法

```
void kwaymerge ( Element *r ) {
    r = new Element[k];              //创建对象数组
    int *key = new int[k+1];         //创建外结点数组
    int *loser = new int[k];         //创建败者树数组
    for ( int i = 0; i < k; i++ )    //传送参选关键码
      { InputRecord ( r[i] );  key[i] = r[i].key; }
    for ( i = 0; i < k; i++) loser[i] = k;
    key[k] = −MaxNum;                //初始化
    for ( i = k−1; i; i−− )          //调整形成败者树
        adjust ( key, loser, k, i );
```

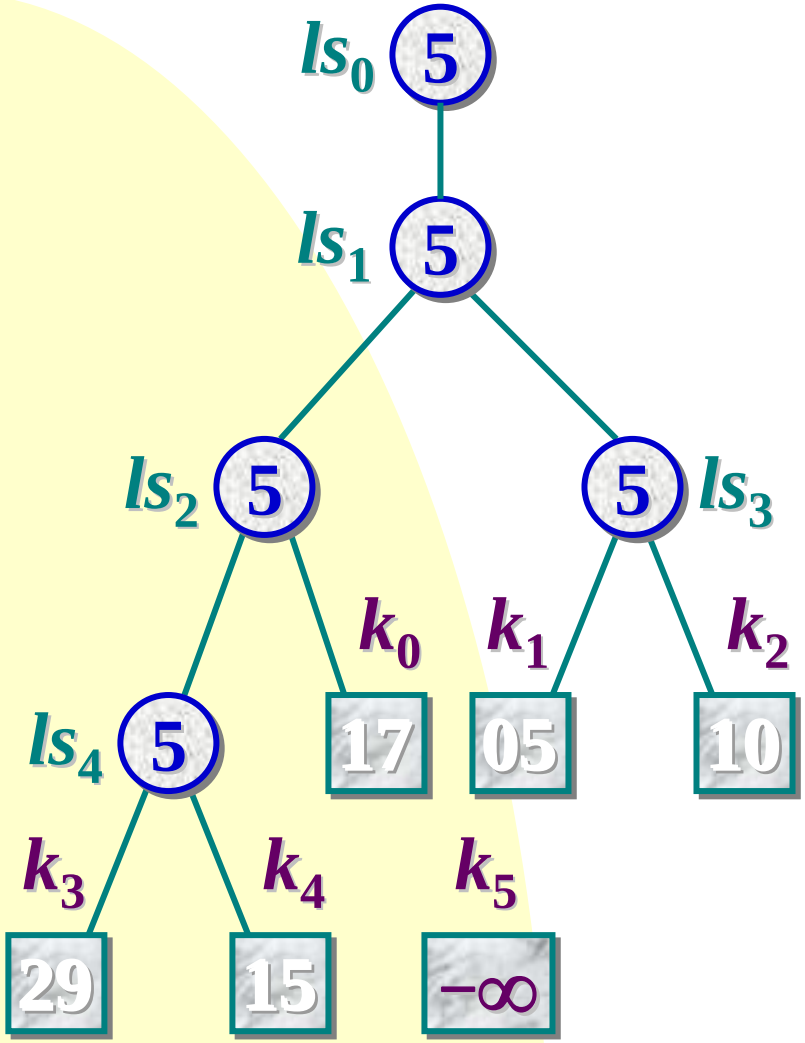
```
while ( key[loser[0]] != MaxNum ) {   //选归并段
    q = loser[0];                     //最小对象的段号
    OutputRecord ( r[q] );            //输出
    InputRecord ( r[q] );             //从该段补入对象
    key[q] = r[q].key;
    adjust ( key, loser, k, q );      //调整
}
Output end of run marker;             //输出段结束标志
delete [ ] r;  delete [ ] key;  delete [ ] loser;
}
```

自某叶结点key[q]到败者树根结点的调整算法

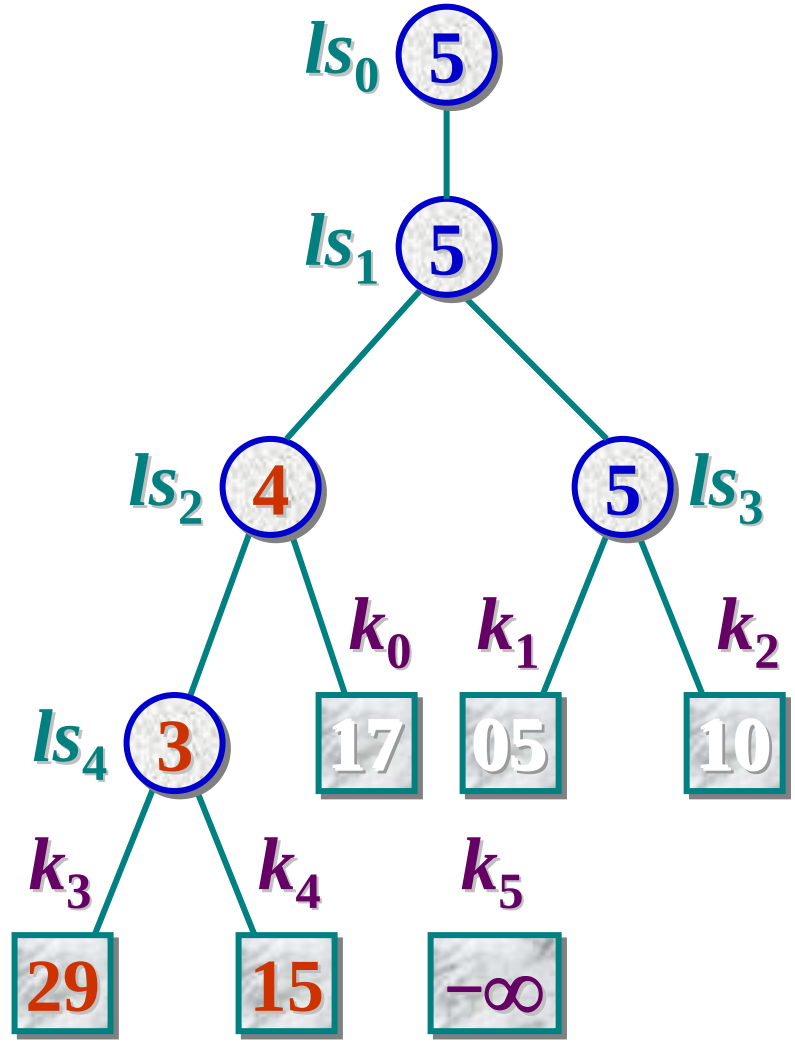
```
void adjust ( int key[ ]; int loser[ ]; const int k;
        const int q ) {
//q指示败者树的某外结点key[q], 从该结点起到根
//结点进行比较, 将最小 key 对象所在归并段的段
//号记入loser[0]。 k是外结点key[0..k-1]的个数。
    for ( int t = (k+q) / 2; t > 0; t /= 2 )  // t是q的双亲
        if ( key[loser[t]] < key[q]) {
                        //败者记入loser[t], 胜者记入q
            int temp = q;   q = loser[t];  loser[t] = temp;
        }               //q与loser[t]交换
    loser[0] = q;
}
```

- 以后每选出一个当前关键码最小的对象，就需要在将它送入输出缓冲区之后，从相应归并段的输入缓冲区中取出下一个参加归并的对象，替换已经取走的最小对象，再从叶结点到根结点，沿某一特定路径进行调整，将下一个关键码最小对象的归并段号调整到***loser*[0]**中。
- 最后，段结束标志**MaxNum**升入***loser*[0]**，排序完成，输出一个段结束标志。
- 归并路数 ***k*** 的选择不是越大越好。归并路数 ***k*** 增大时，相应需增加输入缓冲区个数。如果可供使用的内存空间不变，势必要减少每个输入缓冲区的容量，使内外存交换数据的次数增大。

利用败者树进行5路平衡归并的过程

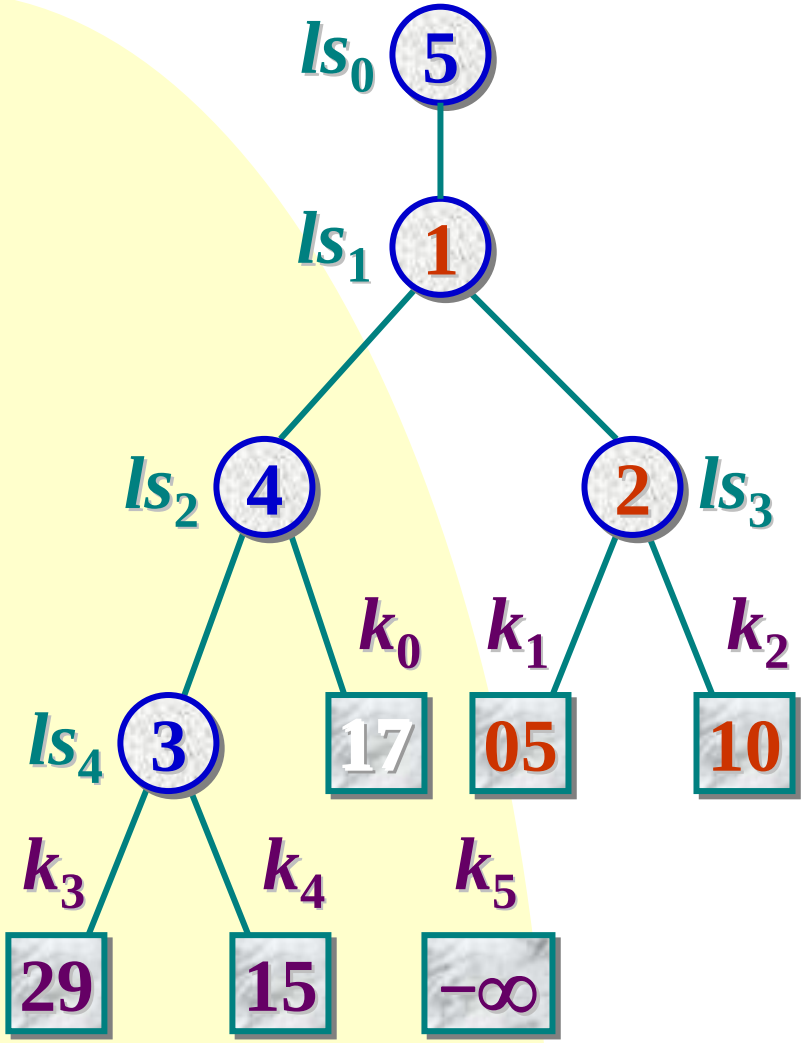


(1) 初始状态

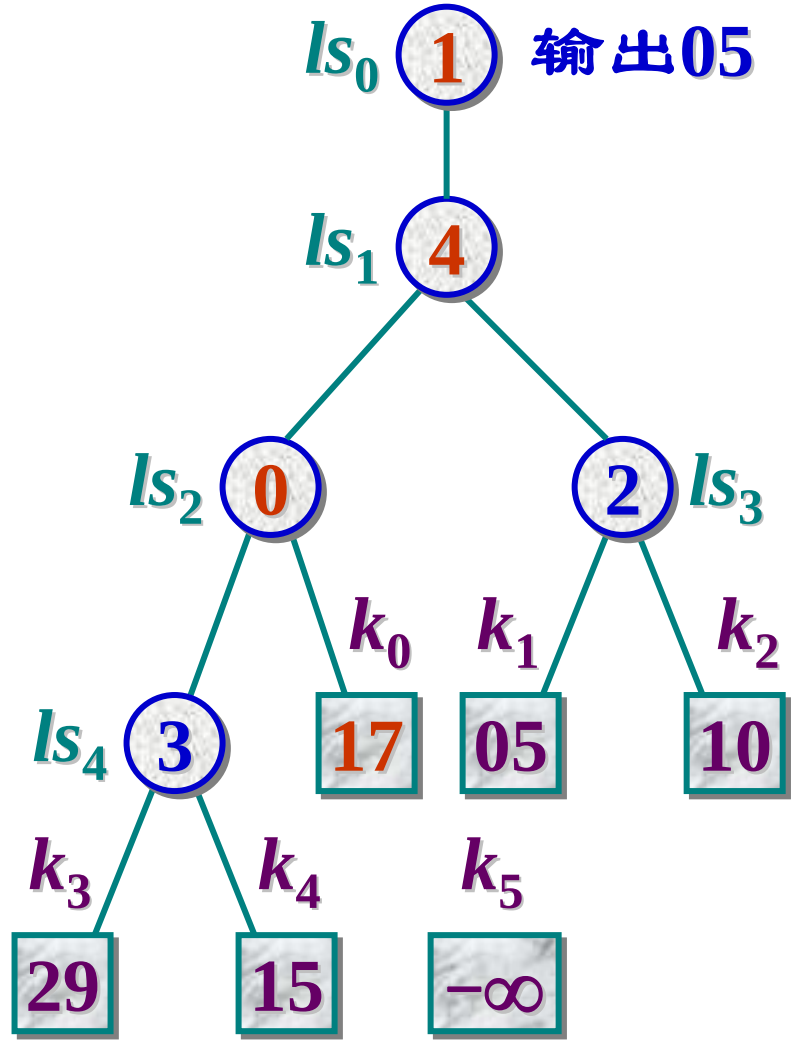


(2) 加入15, 29, 调整

利用败者树进行5路平衡归并的过程

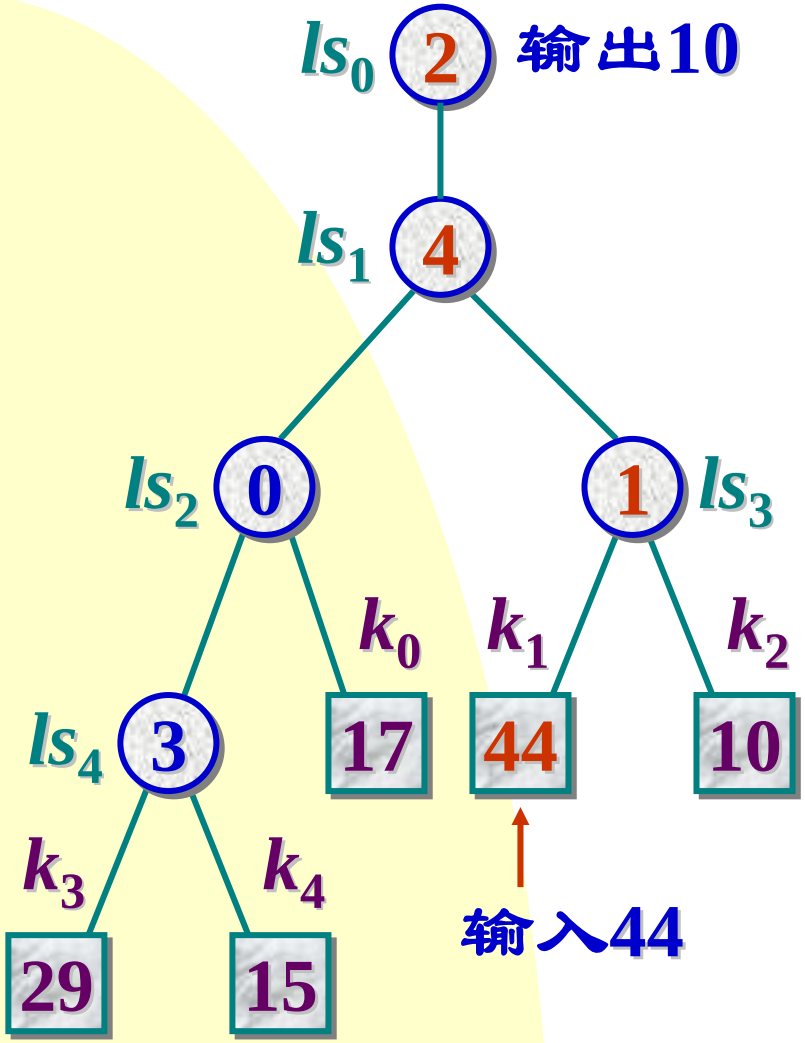


(3) 加入10, 05, 调整

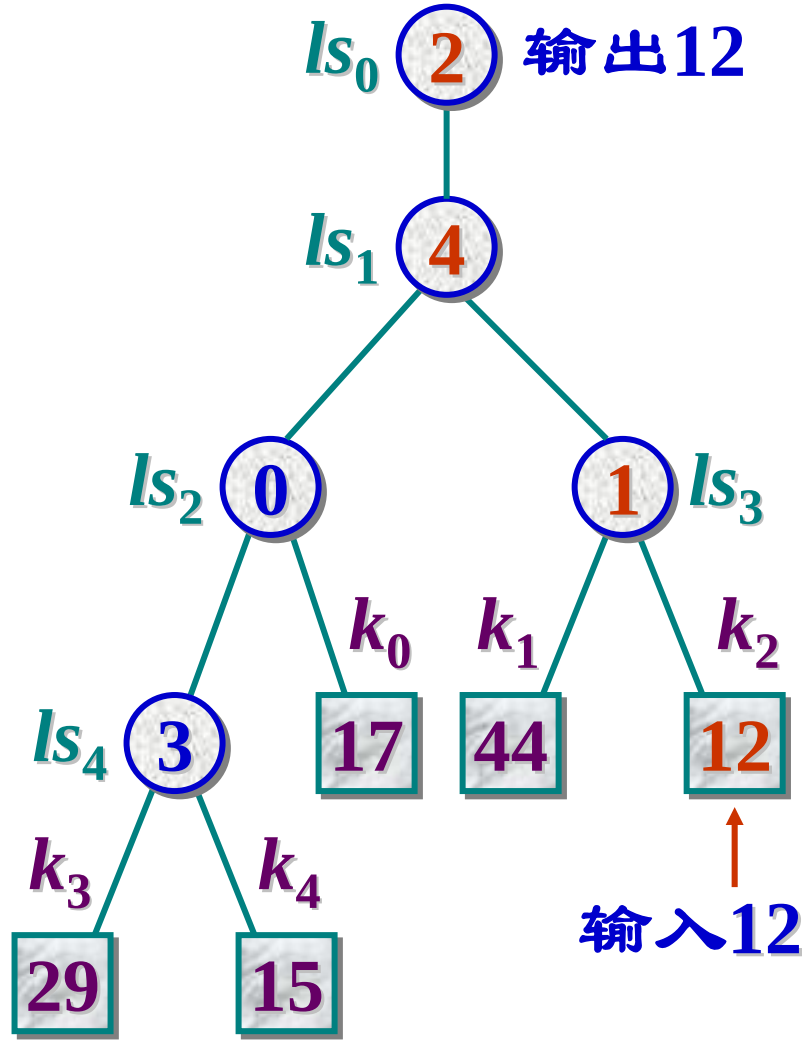


(4) 加入17, 调整

利用败者树进行5路平衡归并的过程

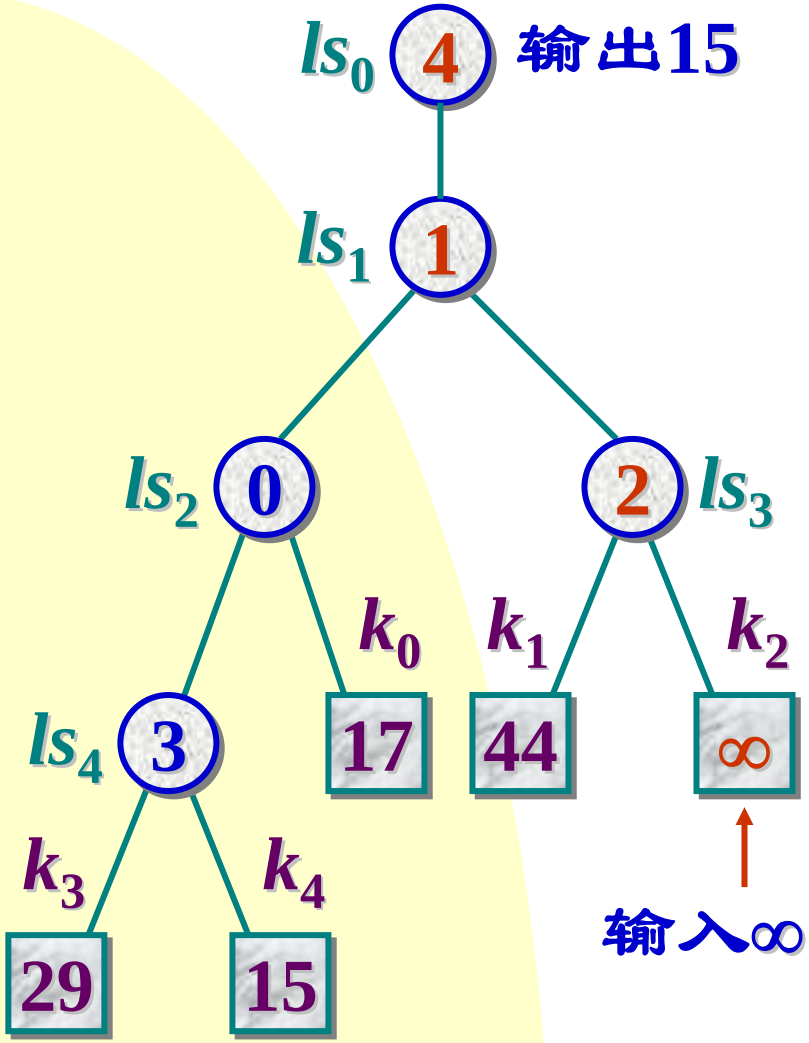


(5) 输出05后调整

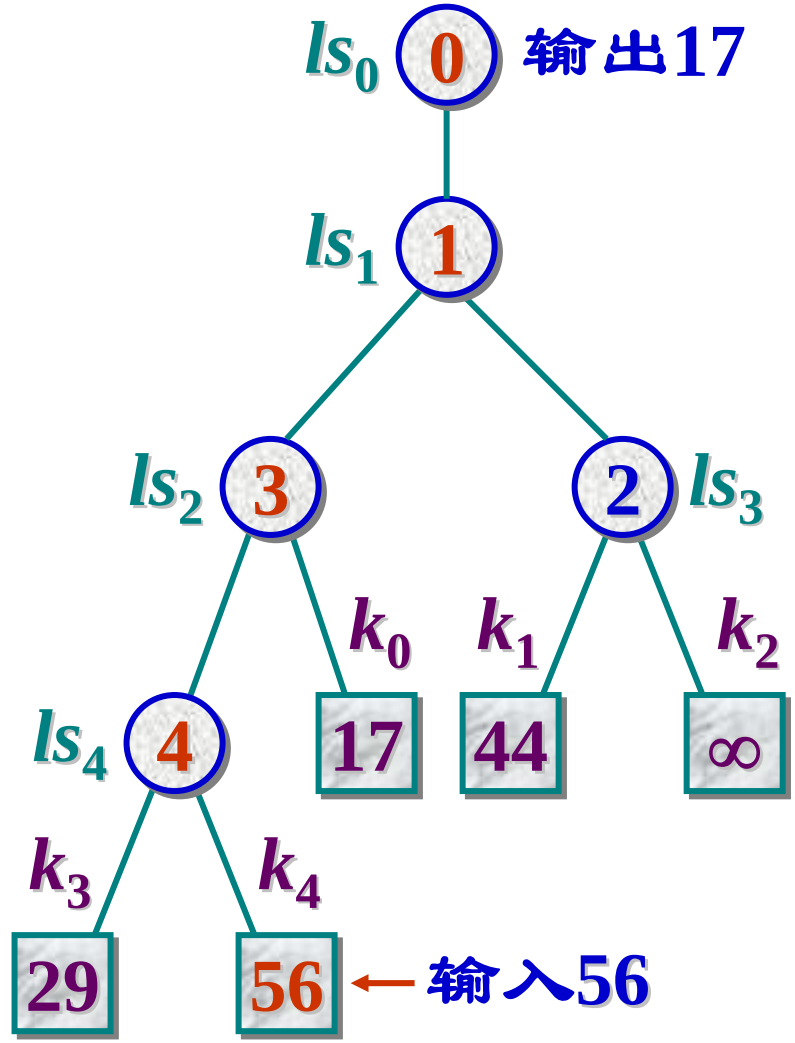


(6) 输出10后调整

利用败者树进行5路平衡归并的过程

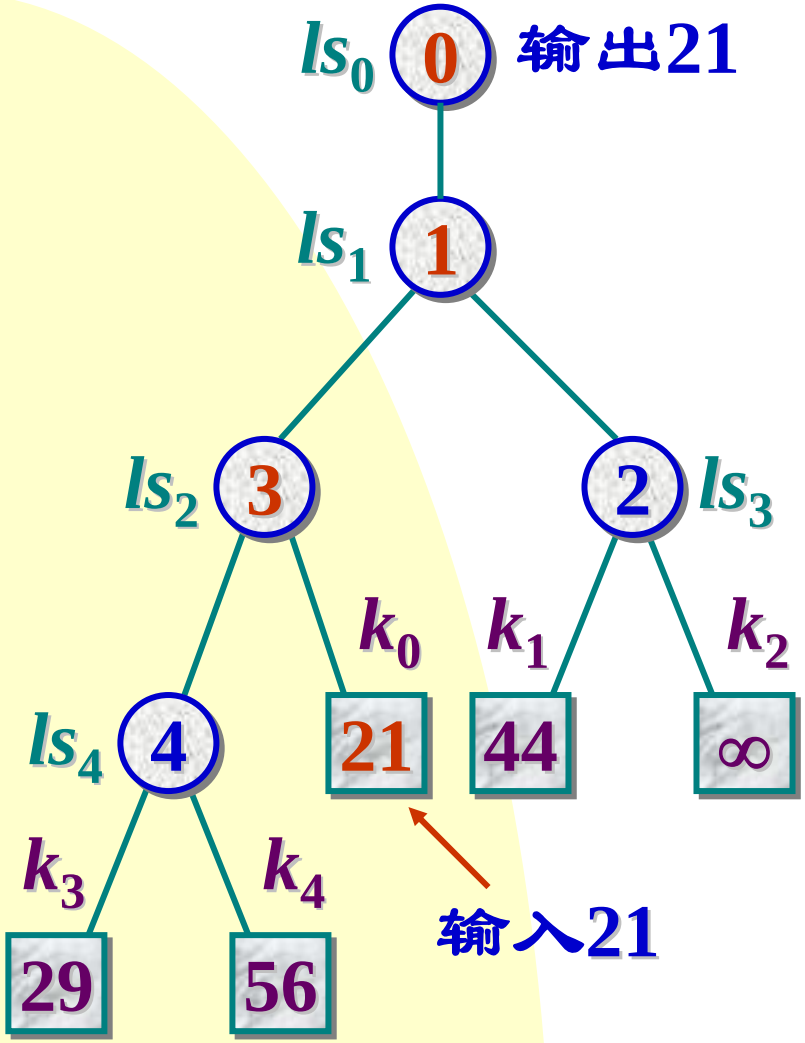


(7) 输出12后调整

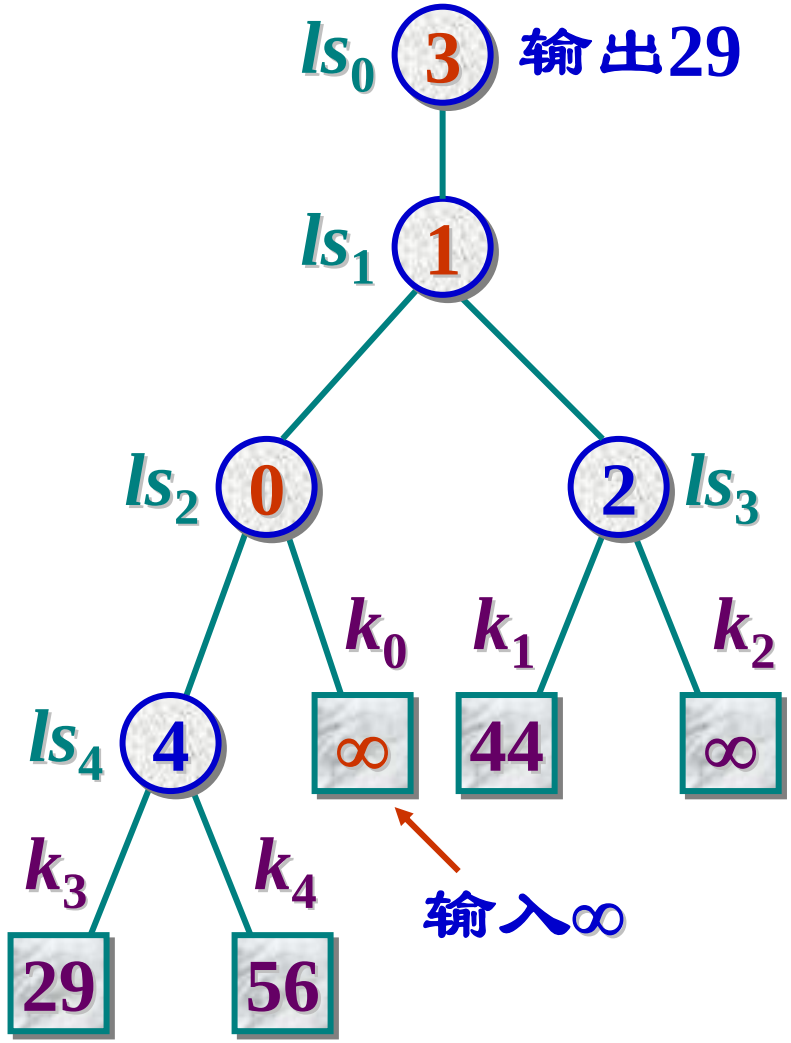


(8) 输出15后调整

利用败者树进行5路平衡归并的过程

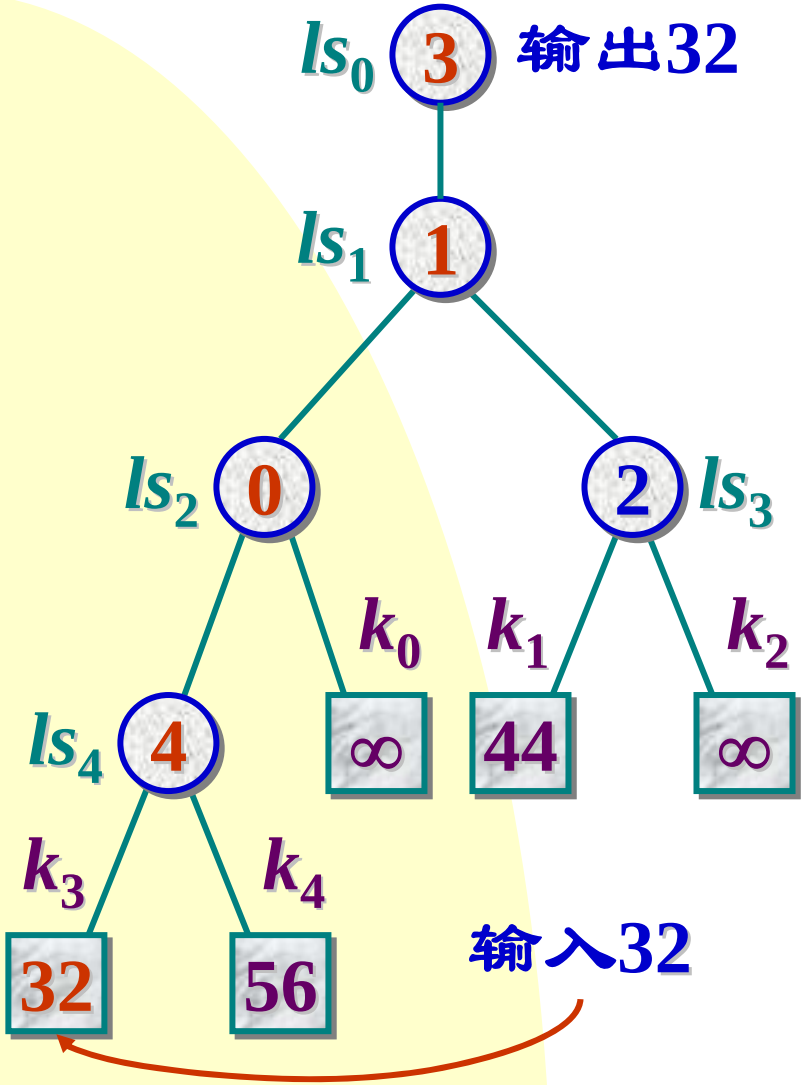


(9) 输出17后调整

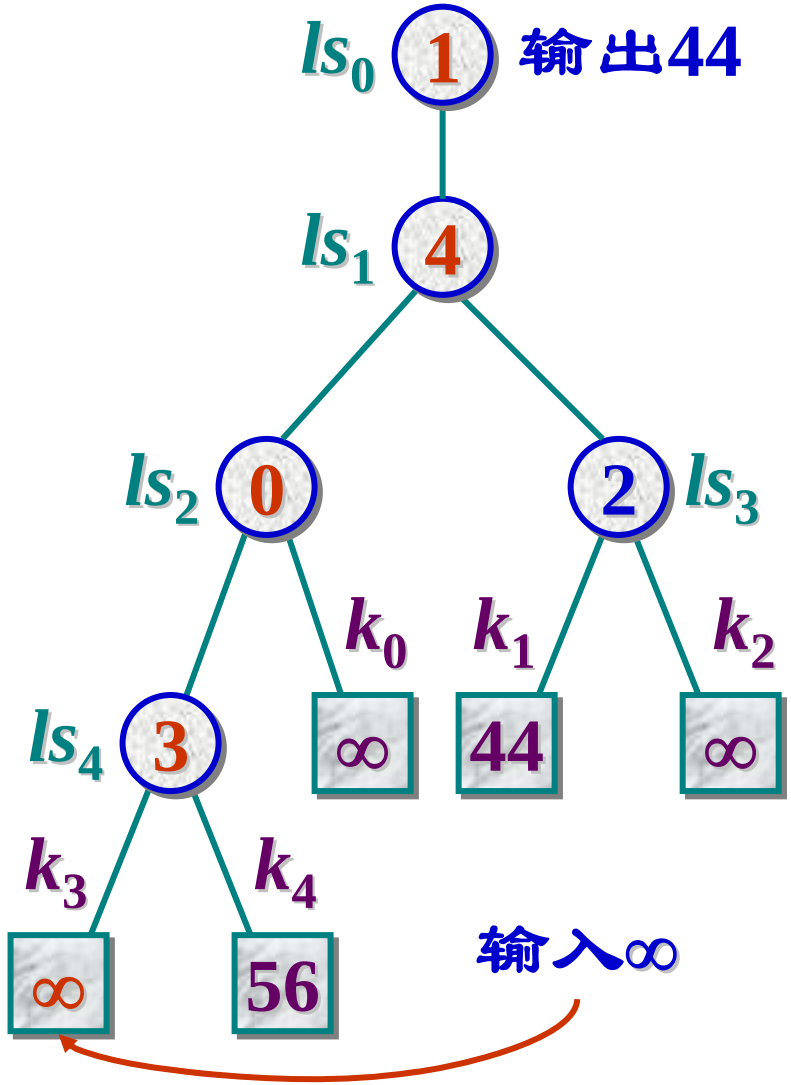


(10) 输出21后调整

利用败者树进行5路平衡归并的过程

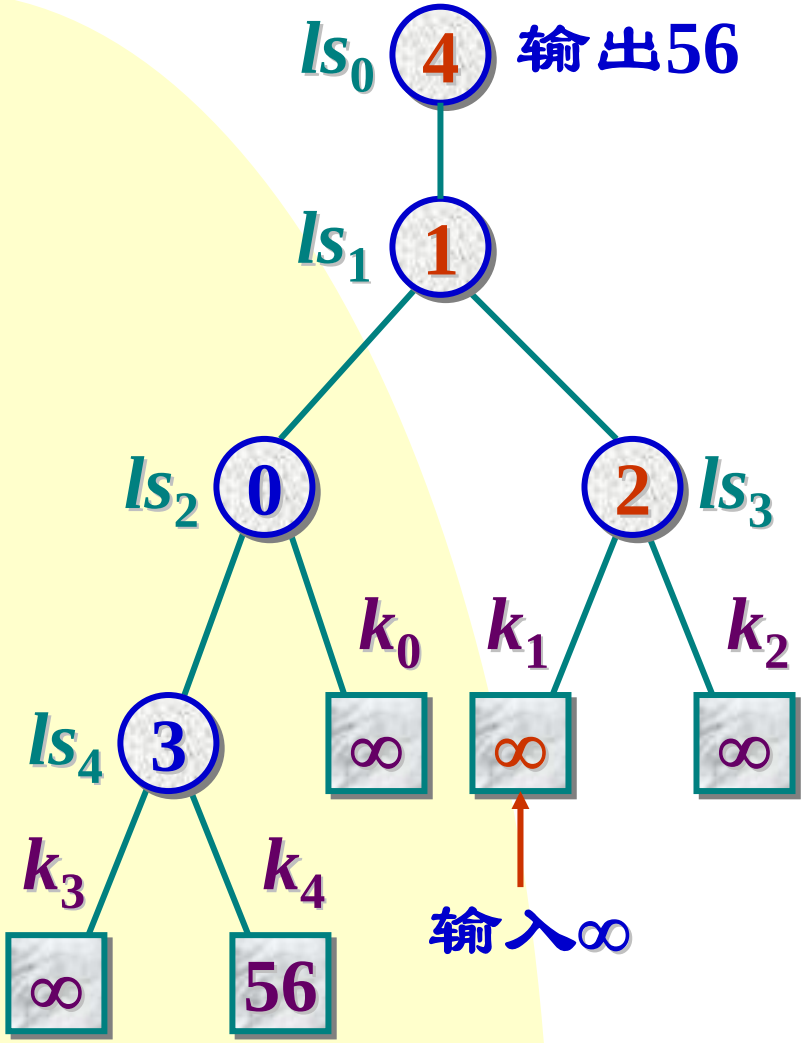


(11) 输出29后调整

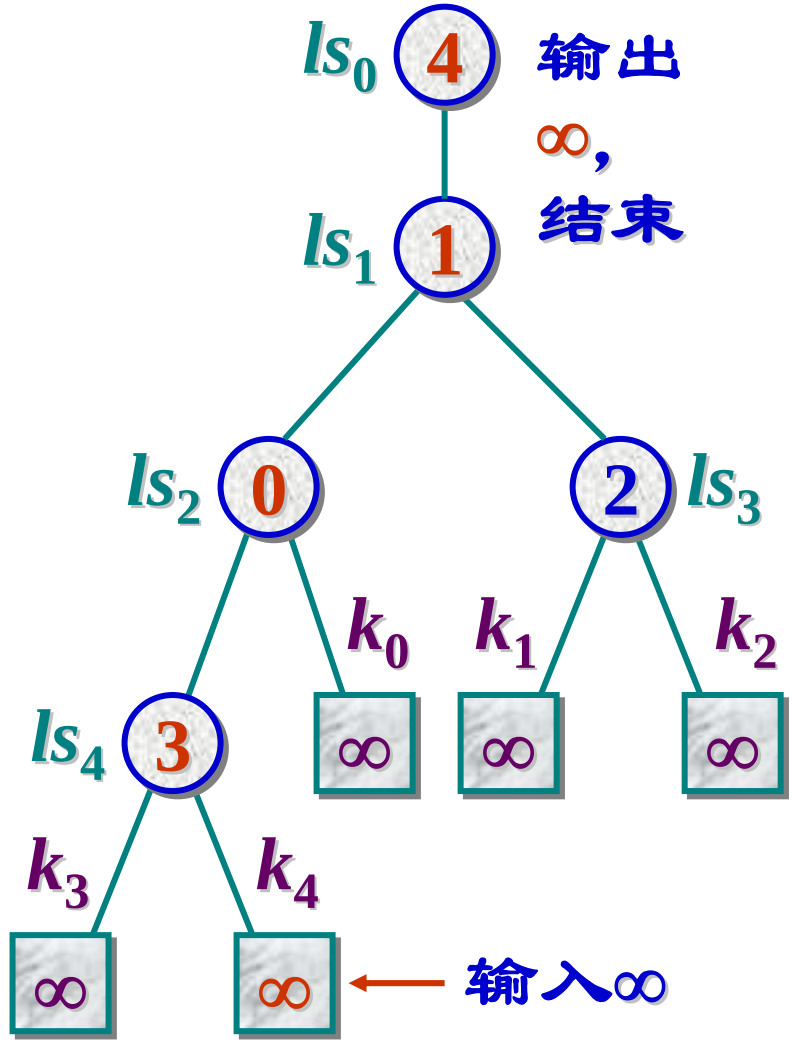


(12) 输出32后调整

利用败者树进行5路平衡归并的过程



(13) 输出44后调整



(14) 输出56后调整

初始归并段的生成 (Run Generation)

- 为了减少读写磁盘次数，除增加归并路数 ***k*** 外，还可减少初始归并段个数 ***m***。在总对象数***n*** 一定时，要减少 ***m***，必须增大初始归并段长度。
- 如果规定每个初始归并段等长，则此长度应根据生成它的内存工作区空间大小而定，因而***m*** 的减少也就受到了限制。
- 为了突破这个限制，可采用败者树来生成初始归并段。在使用同样大的内存工作区的情况下，可以生成平均比原来等长情况下大一倍的初始归并段，从而减少参加多路平衡归并排序的初始归并段个数，降低归并趟数。

- 图解举例说明如何利用败者树产生较长的初始归并段。设输入文件***FI***中各对象的关键码序列为 **{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }**。

选择和置换过程的步骤如下:

☆ 从输入文件***FI***中把 ***k*** 个对象读入内存中，并构造败者树。**(**内存中存放对象的数组***r***可容纳的对象个数为 ***k*** **)**

🕐 利用败者树在 ***r*** 中选择一个关键码最小的对象***r[q]***，其关键码存入***LastKey***作为门槛 。以后再选出的关键码比它大的对象归入本归并段，比它小的归入下一归并段。

🕐 将此***r[q]***对象写到输出文件***FO***中。**(*q*是叶结点序号)**

- 若*FI*未读完，则从*FI*读入下一个对象，置换*r*[*q*]及败者树中的*key*[*q*]。
- 调整败者树，从所有关键码比*LastKey*大的对象中选择一个关键码最小的对象*r*[*q*]作为门槛，其关键码存入*LastKey*。
- 重复 ~ ，直到在败者树中选不出关键码比*LastKey*大的对象为止。此时，在输出文件*FO*中得到一个初始归并段，在它最后加一个归并段结束标志。
- 重复 ~ ，重新开始选择和置换，产生新的初始归并段，直到输入文件*FI*中所有对象选完为止。

输入文件 *FI*	内存数组 *r*	输出文件 *FO*
21 05 44 10 12 56 32 29		
44 10 12 56 32 29	17 21 **05**	
10 12 56 32 29	**17** 21 44	05
12 56 32 29	10 **21** 44	05 17
56 32 29	10 12 **44**	05 17 21
32 29	10 12 **56**	05 17 21 44
29	10 12 32	05 17 21 44 56
29	10 12 32	05 17 21 44 56 ∞
29	**10** 12 32	
	29 **12** 32	10
	29 — 32	10 12
	— — **32**	10 12 29
	— — —	10 12 29 32
	— — —	10 12 29 32 ∞

- 若按在 *k* 路平衡归并排序中所讲的，每个初始归并段的长度与内存工作区的长度一致，则上述9个对象可分成3个初始归并段：

  Run0 { 05, 17, 21 }

  Run1 { 10, 12, 44 }

  Run2 { 29, 32, 56 }

- 但采用上述选择与置换的方法，可生成2个长度不等的初始归并段：

  Run0 { 05, 17, 21, 44, 56 }

  Run1 { 10, 12, 29, 32 }

- 在利用败者树生成不等长初始归并段的算法和调整败者树并选出最小对象的算法中，用两个条件来决定谁为败者，谁为胜者。
  - 首先比较两个对象所在归并段的段号，段号小者为胜者，段号大者为败者；
  - 在归并段的段号相同时，关键码小者为胜者，关键码大者为败者。
- 比较后把败者对象在对象数组 ***r*** 中的序号记入它的双亲结点中，把胜者对象在对象数组 ***r*** 中的序号记入工作单元 ***s*** 中，向更上一层进行比较，最后的胜者记入 ***loser*[0]**中。

利用败者树生成初始归并段

```
void generateRuns ( Element *r ) {
    r = new Element[k];
    int *key = new int[k];        //参选对象关键码数组
    int *rn = new int[k];         //参选对象段号数组
    int *loser = new int[k];      //败者树
    for ( int i = 0; i < k; i++ ){ loser[i] = 0; rn[i] = 0;}
    for ( i = k−1; i > 0; i−− ) {          //输入首批对象
        if ( end of input ) rn[i] = 2;     //中途结束
        else { InputRecord ( r[i] );       //从缓冲区输入
               key[i] = r[i].key;  rn[i] = 1;
        }
```

```
    SelectMin ( key, rn, loser, k, i, rq );   //调整
}
q = loser[0];     // q是最小对象在 r 中的序号
int rq = 1;       // r[q]的归并段段号
int rc = 1;       //当前归并段段号
int rmax = 1;     //下次将要产生的归并段段号
int LastKey = MaxNum;                  //门槛
while (1) {                            //生成一个初始归并段
    if ( rq != rc ) {                  //当前最小对象归并段大
        Output end of run marker;      //加段结束符
        if ( rq > rmax ) return;       //处理结束
        else rc = rq;                  //否则置当前段号等于rq
    }
```

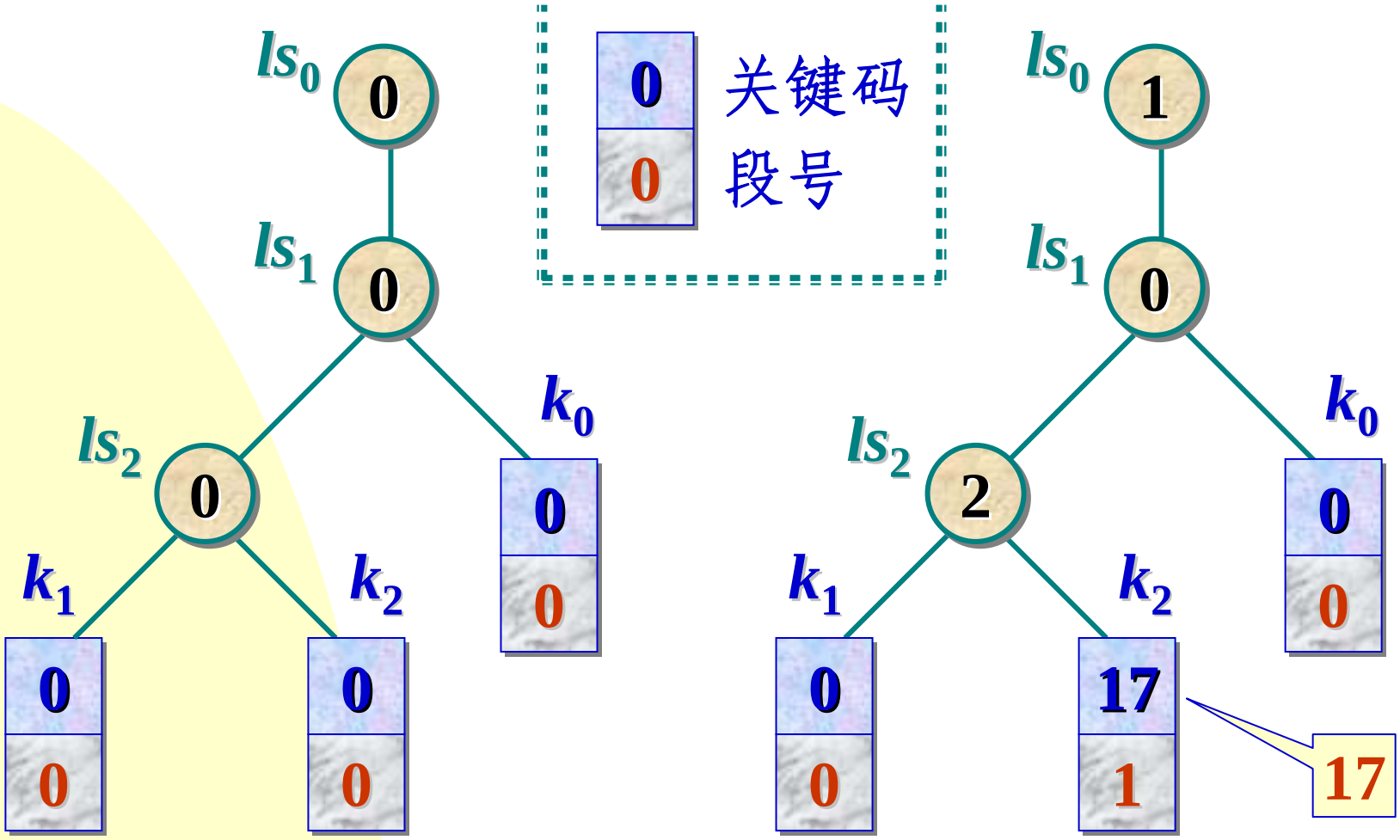
```
OutputRecord ( r[q] );        // rc==rq，输出
LastKey = Key[q];             //置新的门槛
if ( end of input ) rn[q] = rmax + 1; //虚设对象
else {                        //输入文件未读完
    InputRecord ( r[q] );     //读入到刚才位置
    key[i] = r[i].key;
    if ( key[q] < LastKey )   //小于门槛
        rn[q] = rmax = rq + 1; //应为下一段对象
    else rn[q] = rc;          //否则在当前段
}
rq = rn[q];
SelectMin ( key, rn, loser, k, q, rq );
q = loser[0];                 //新的最小对象
```

```
    } // end of while
    delete [ ] r; delete [ ] key;
    delete [ ] rn; delete [ ] loser;
}
```

在败者树中选择最小对象的算法

```
void SelectMin ( int key[ ]; int rn[ ];   int loser[ ];
      const int k;  const int q;   int &rq ) {
//q指示败者树的某外结点key[q], 从该结点向上到
//根结点loser[0]进行比较, 选择出LastKey对象。 k
//是外结点key[0..k−1]的个数。
```

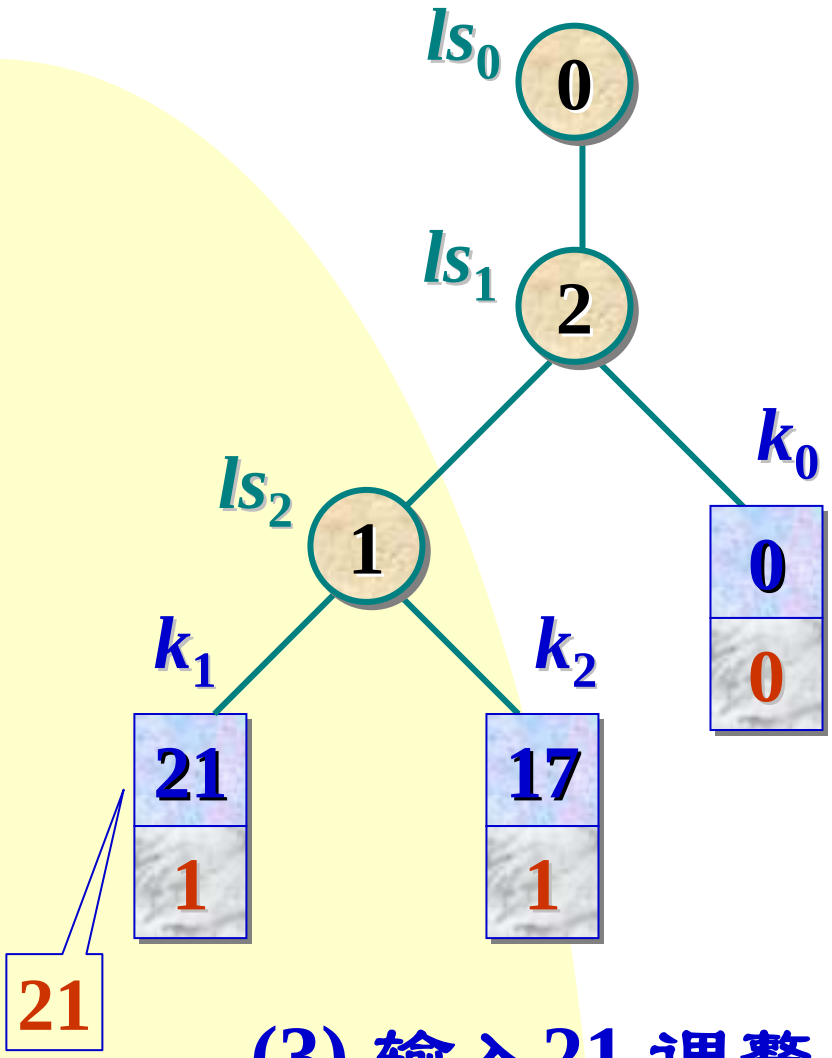
```
    for ( int t = (k+q)/2; t > 0; t /= 2 )
        if ( rn[loser[t]] < rq || rn[loser[t]] == rq &&
                key[loser[t]] < key[q] ) {
            //先比较段号再比较关键码, 小者为胜者
            int temp = q;  q = loser[t];  loser[t] = temp;
            //败者记入loser[t], 胜者记入q
            rq = rn[q];
        }
    loser[0] = q;       //最后的胜者
}
```



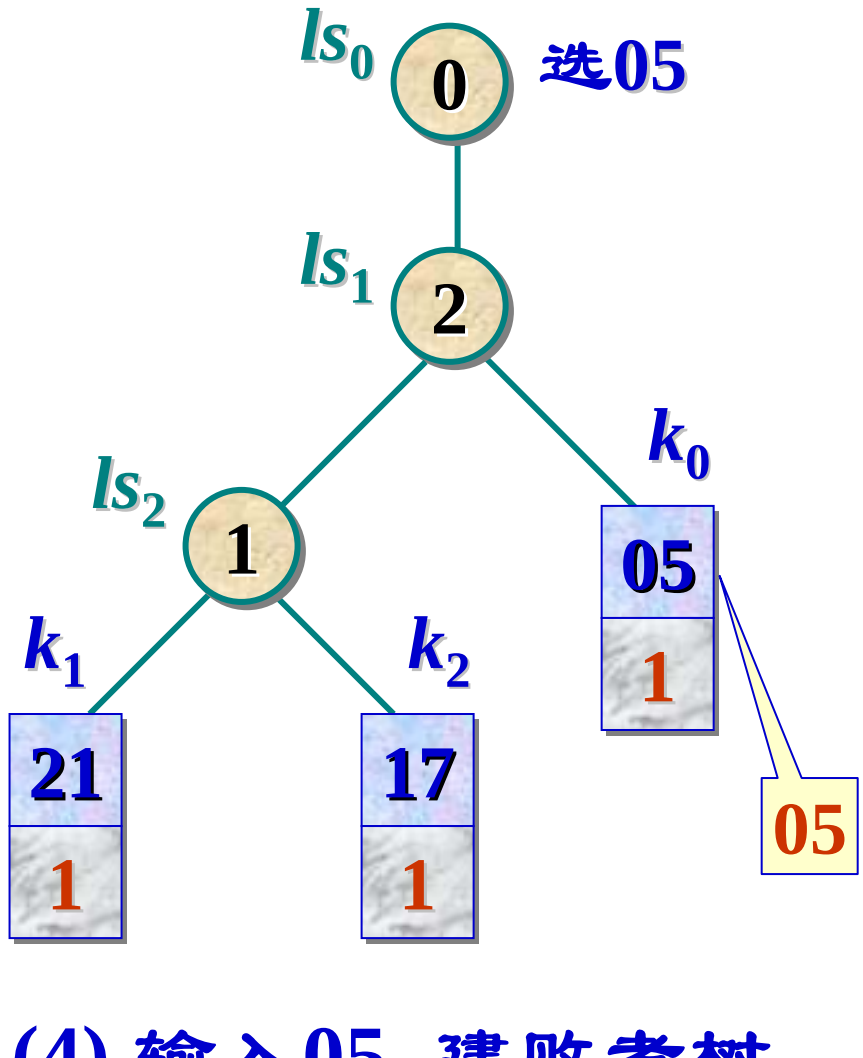
(1) 初始化

(2) 输入17, 调整

{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }

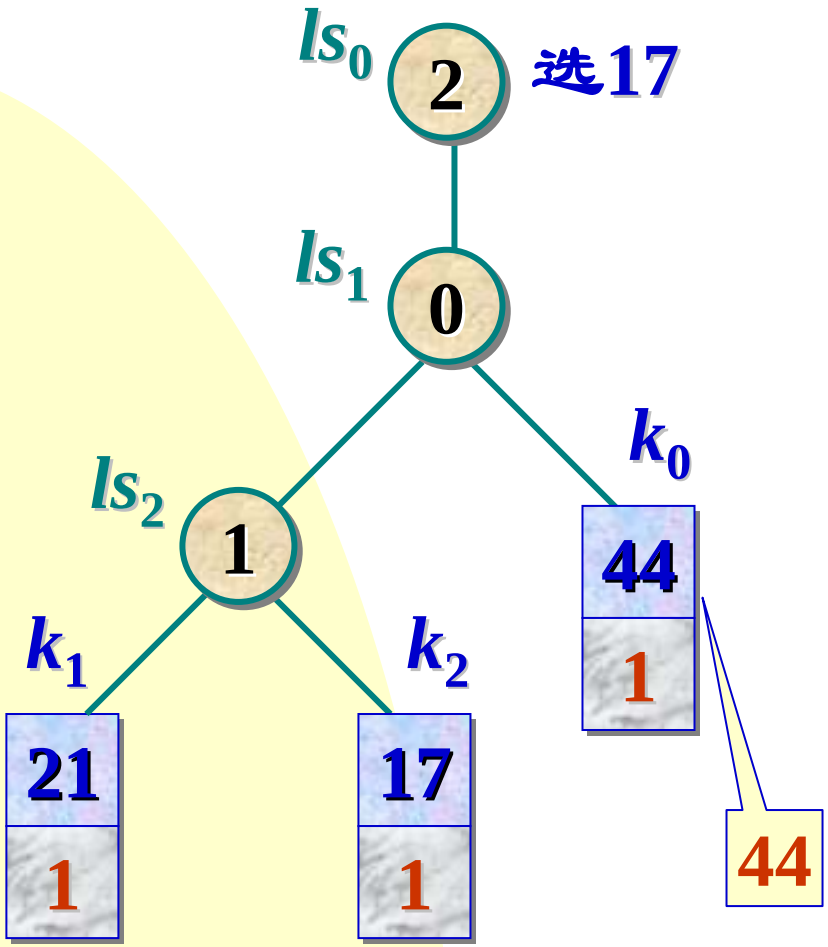


(3) 输入21,调整

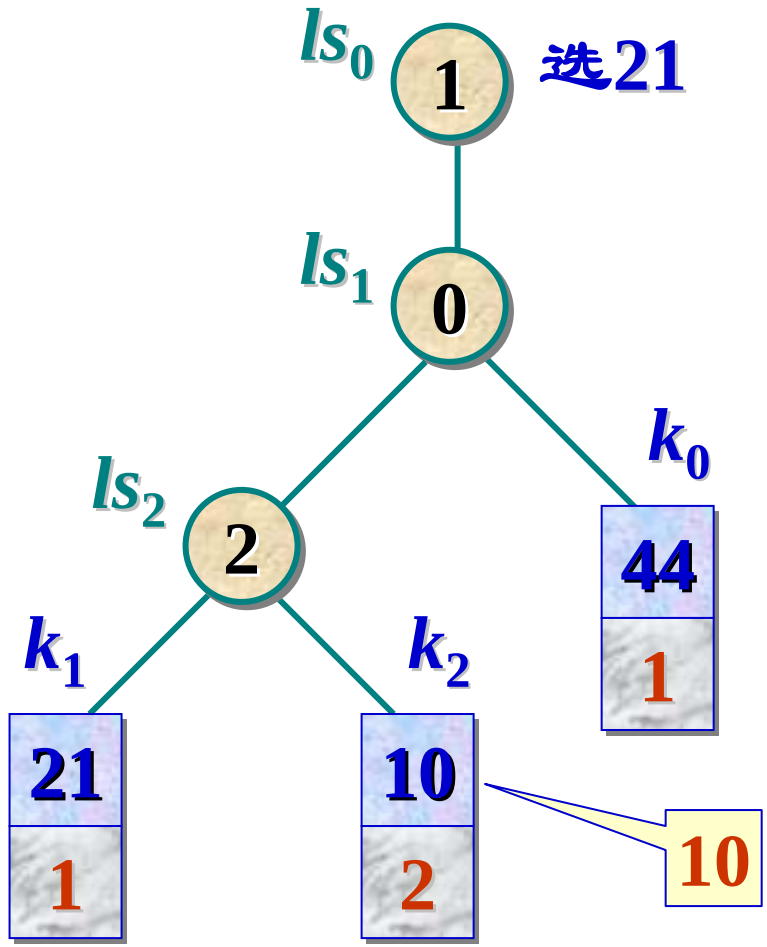


(4) 输入05, 建败者树

{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }

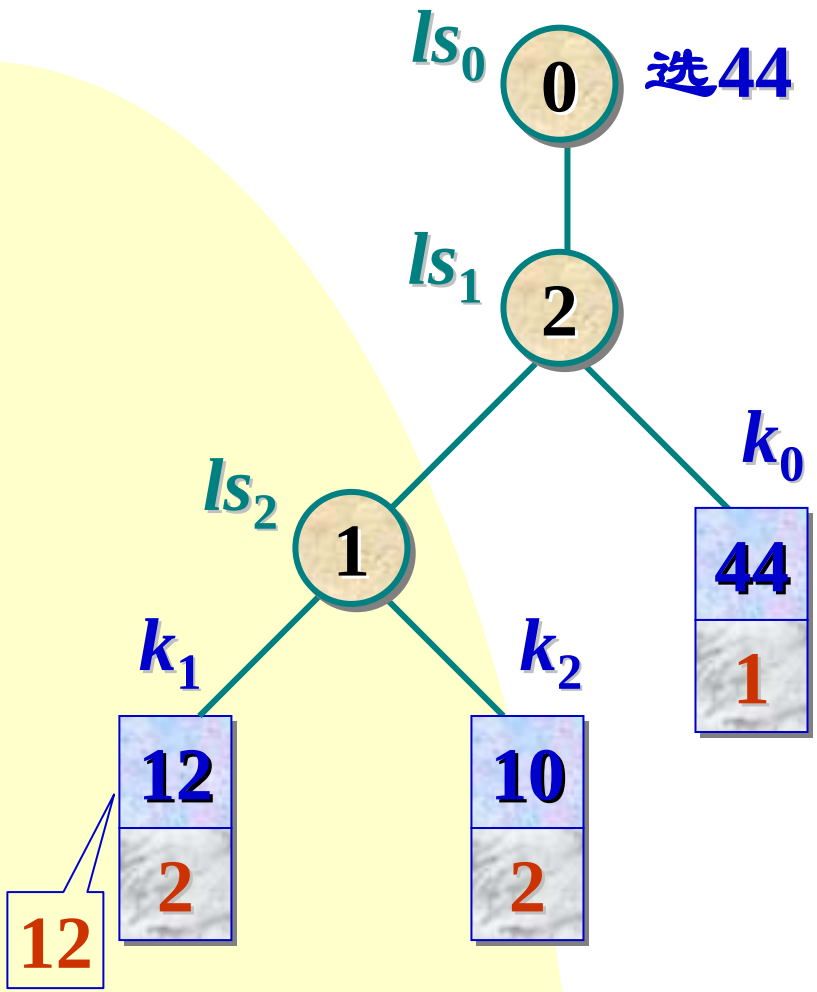


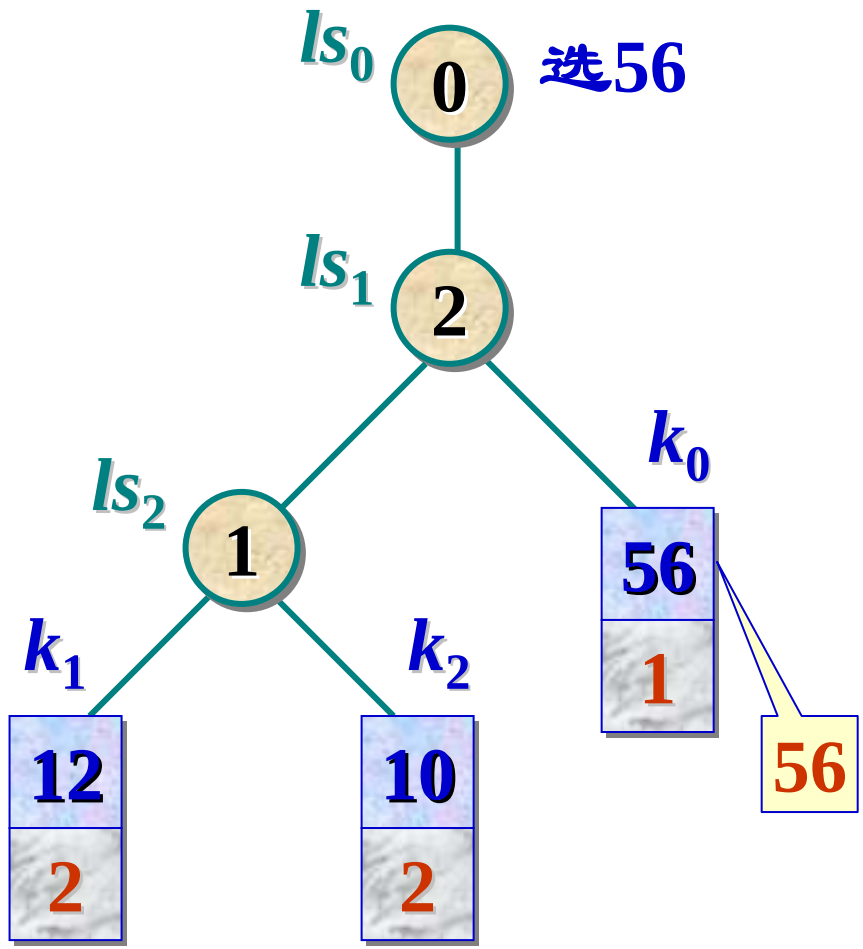
(5) *lastKey*=05, 置换 k_0, 选择17



(6) *lastKey*=17, 置换 k_2, 段号加1, 选择21

{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }

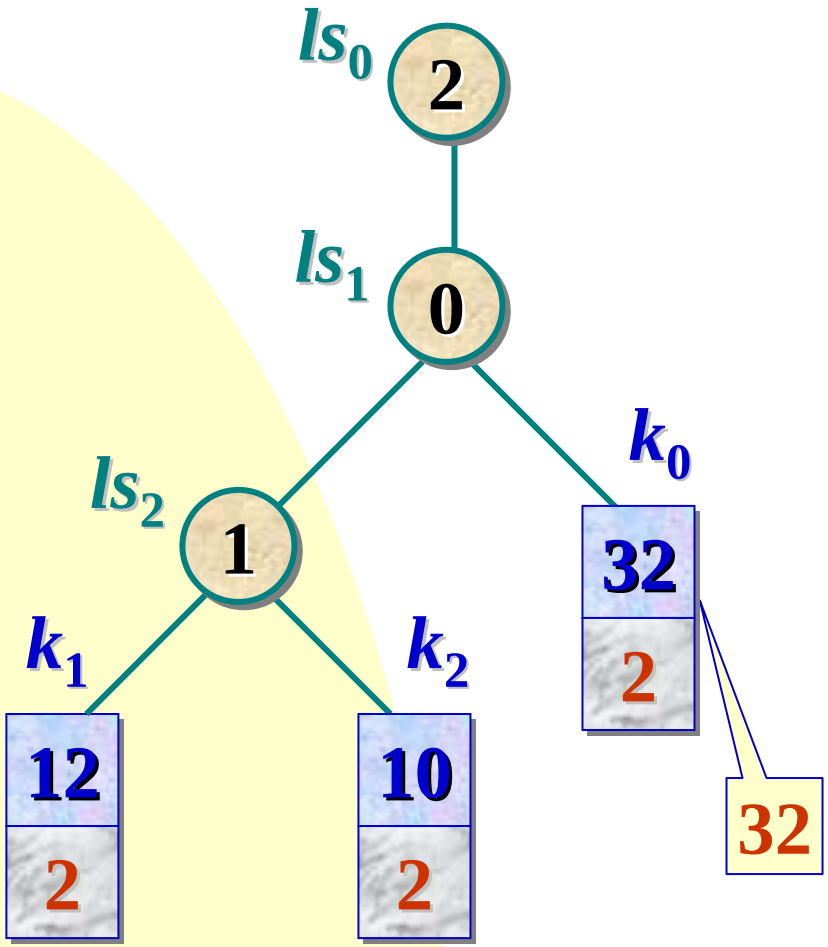




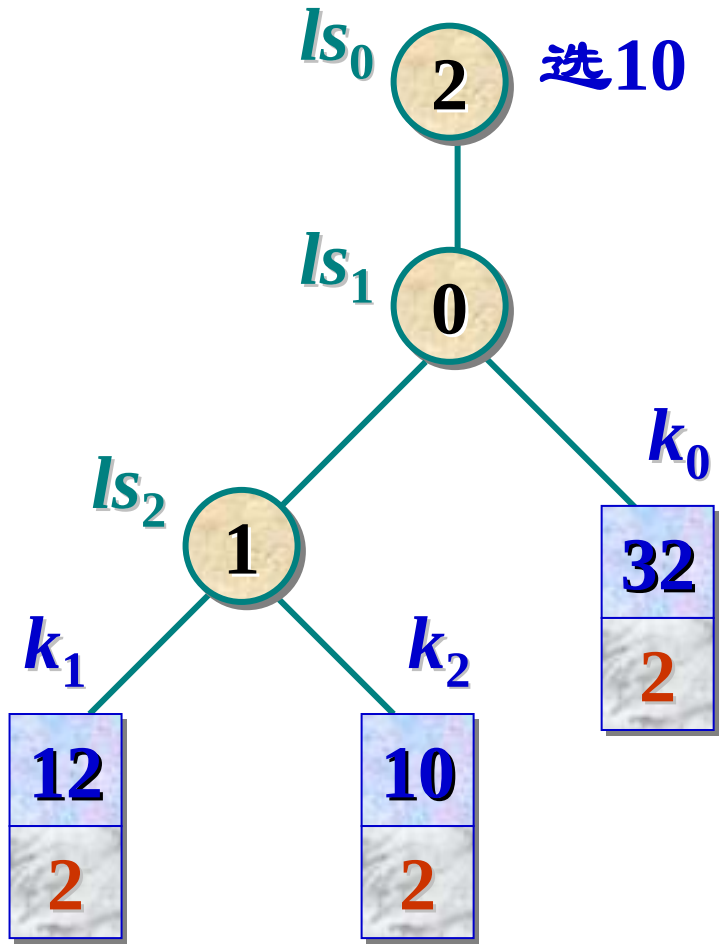
(7) *lastKey*=21, 置换 k_1, 段号加1, 选择44

(8) *lastKey*=44, 置换 k_0, 选择56

{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }

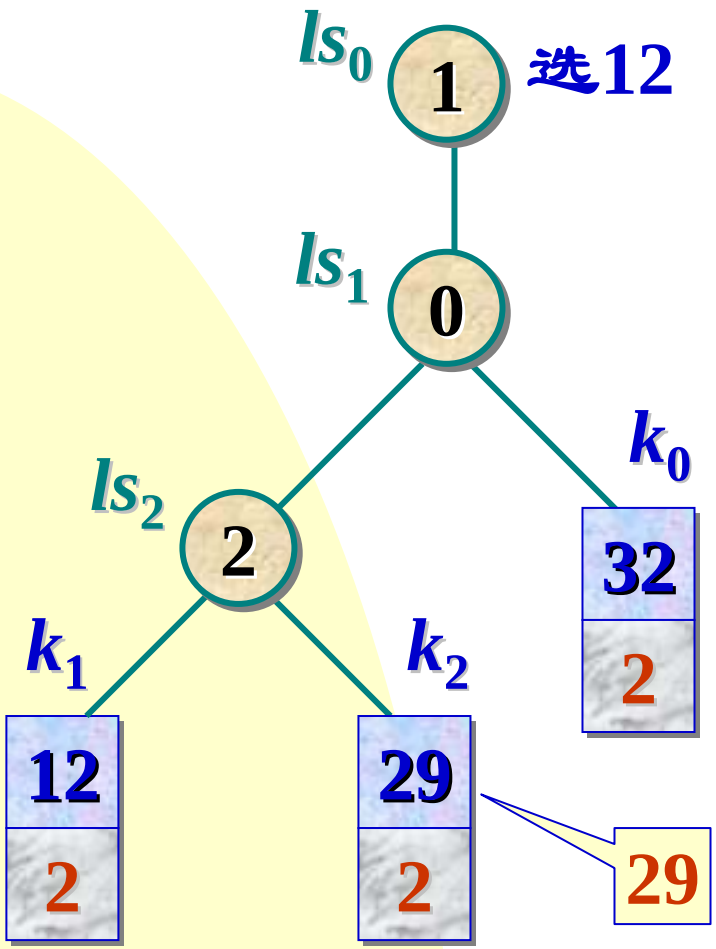


(9) *lastKey*=56, 置换 k_0, 段号加1, 本段结束

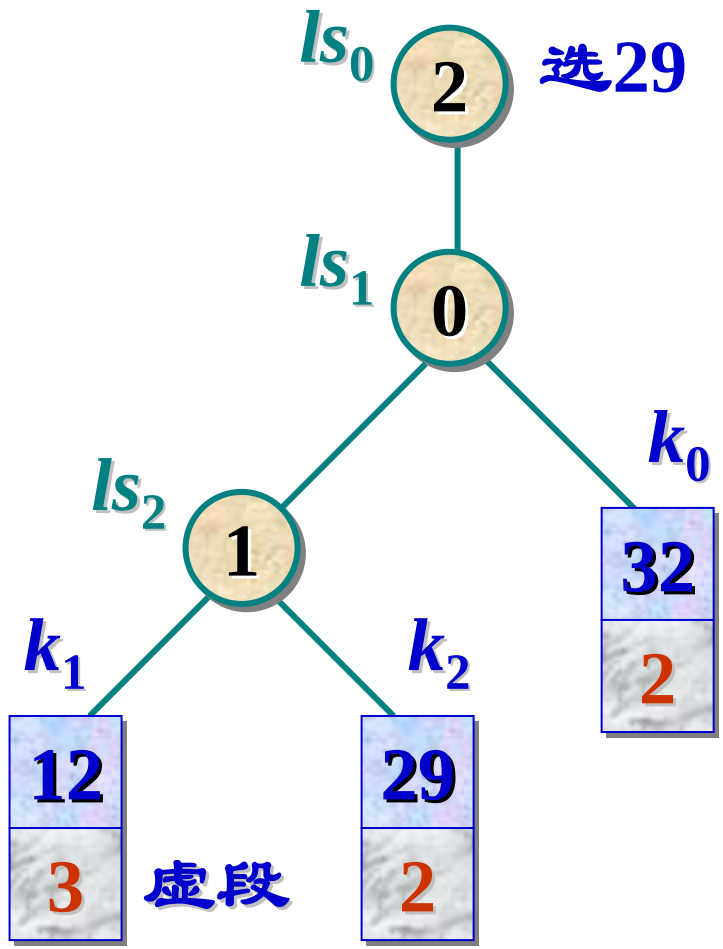


(10) 输出段结束标志, 选择10

{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }

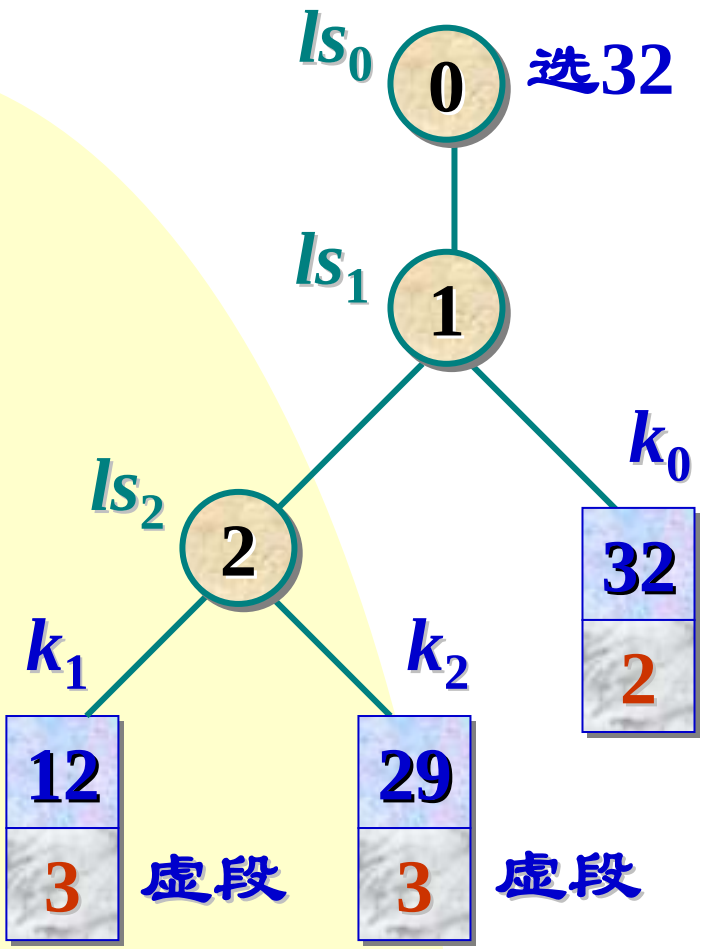


(11) *lastKey*=10, 置换 k_2, 选择12

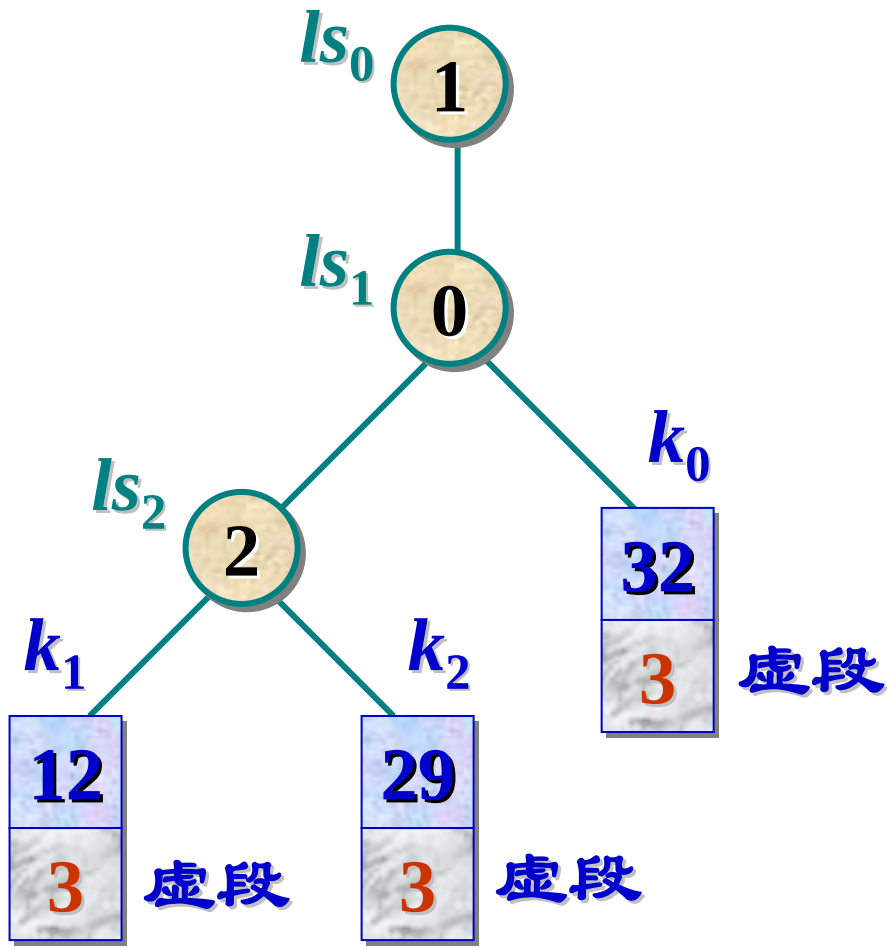


(12) *lastKey*=12, k_1 置虚段, 选择29

{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }

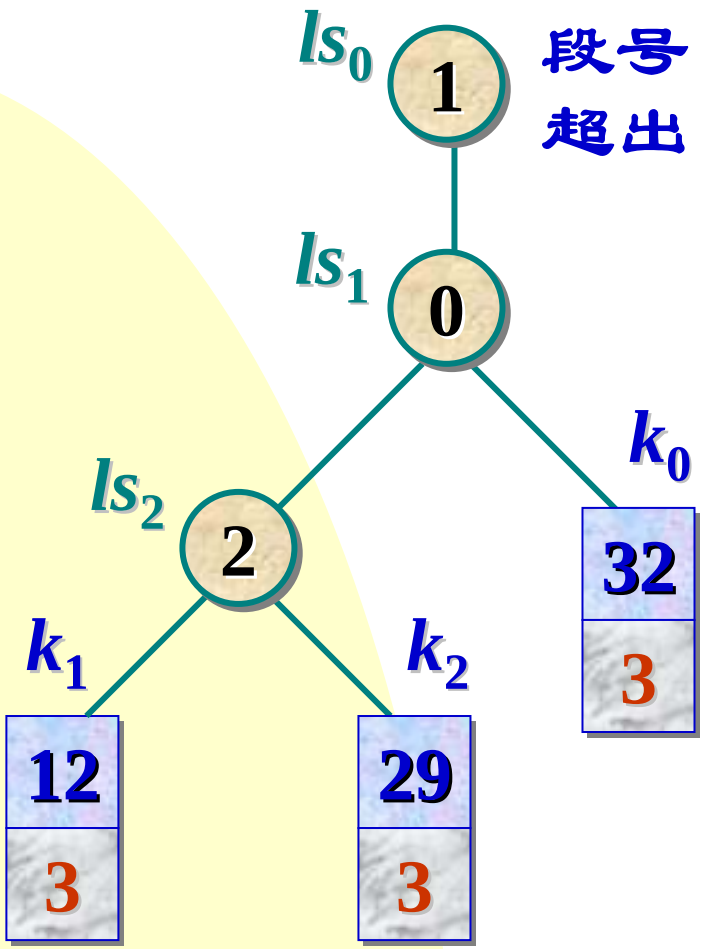


(13) *lastKey*=29, k_2 置虚段, 选择32



(14) *lastKey*=32, k_0 置虚段, 本段结束

{ 17, 21, 05, 44, 10, 12, 56, 32, 29 }



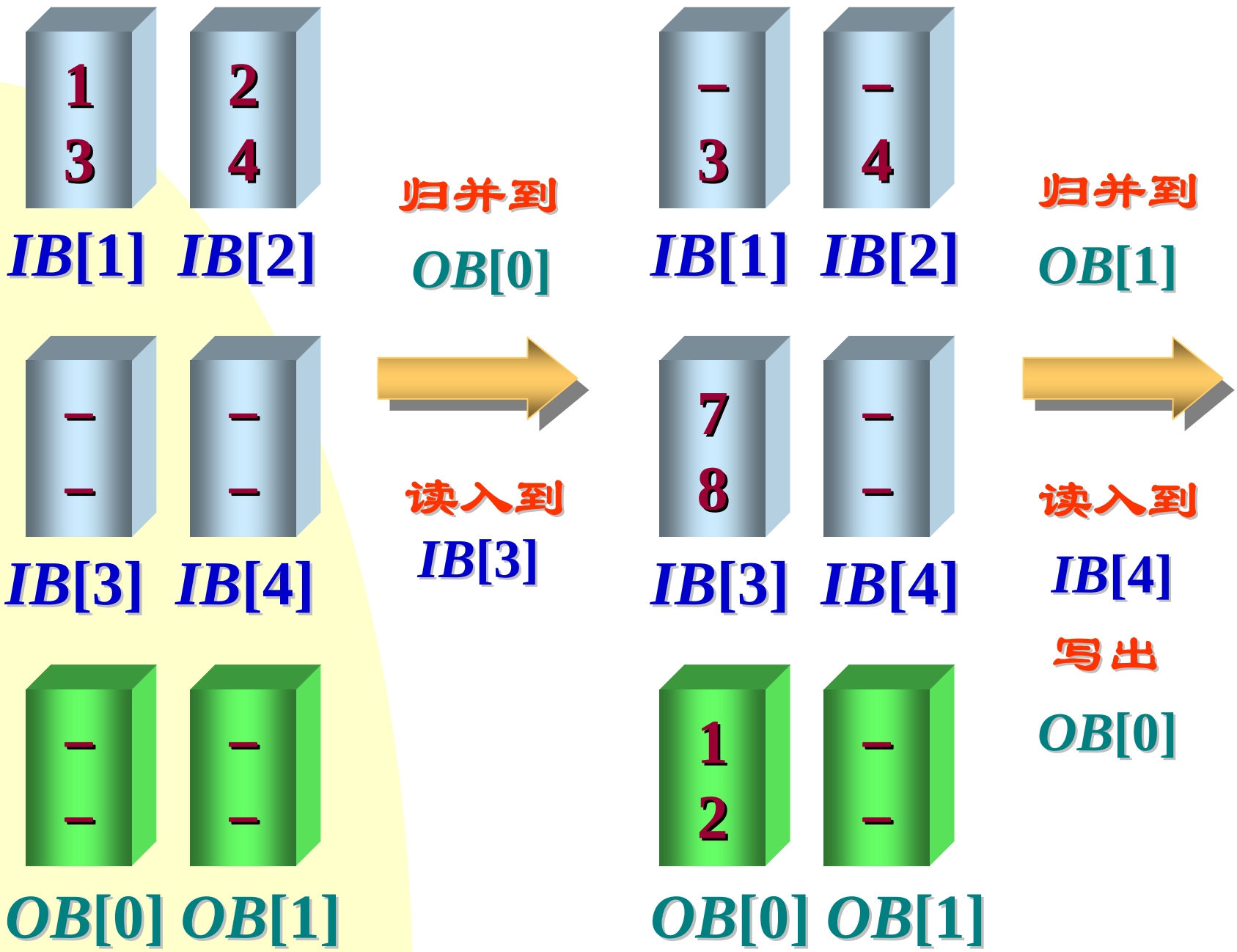
(15) 输出段结束标志，结束

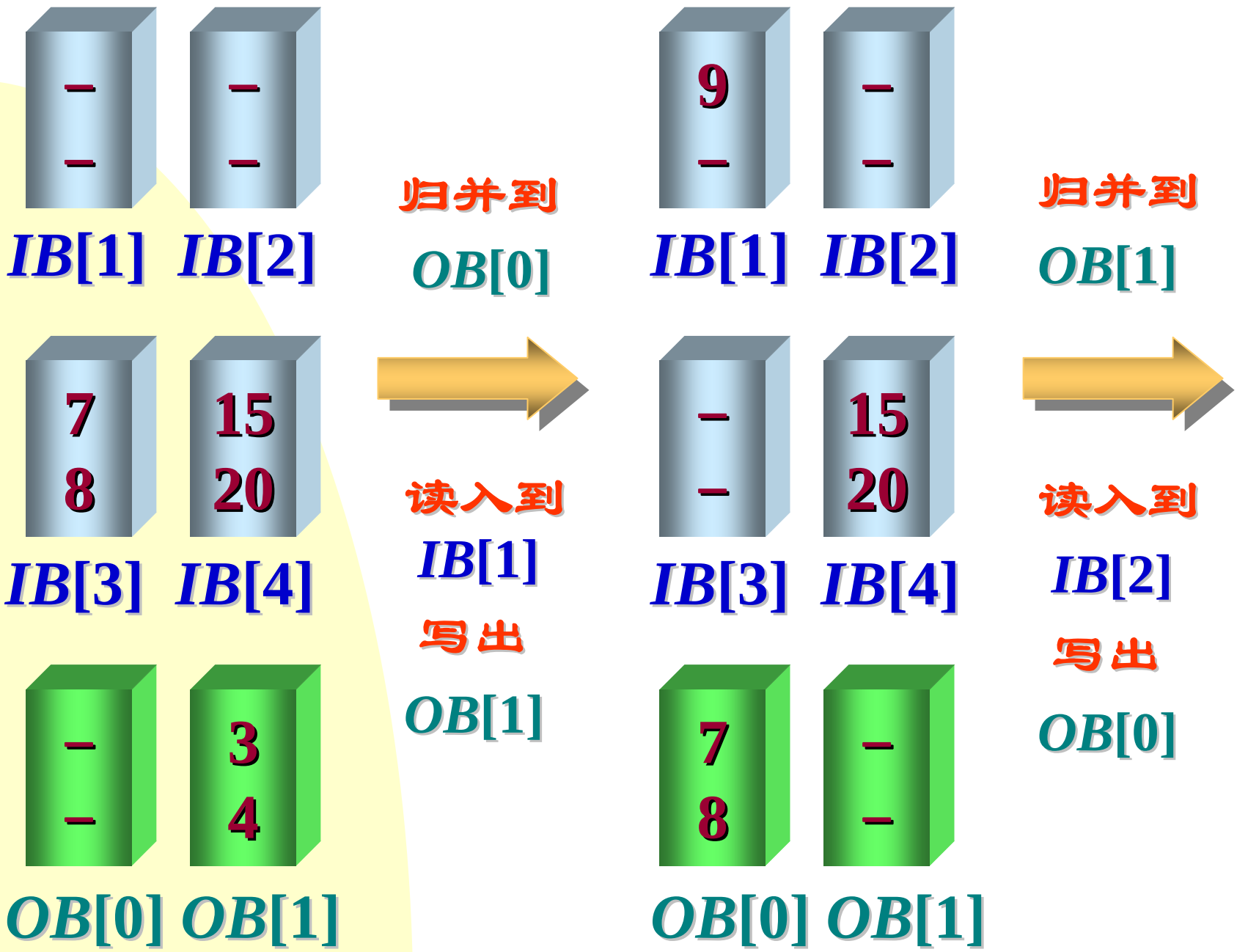
- 当输入的对象序列已经按关键码大小排好序时，只生成一个初始归并段。
- 在一般情况下，若输入文件有 n 个对象，生成初始归并段的时间开销是 $O(n\log_2 k)$，因为每输出一个对象，对败者树进行调整需要时间为 $O(\log_2 k)$。

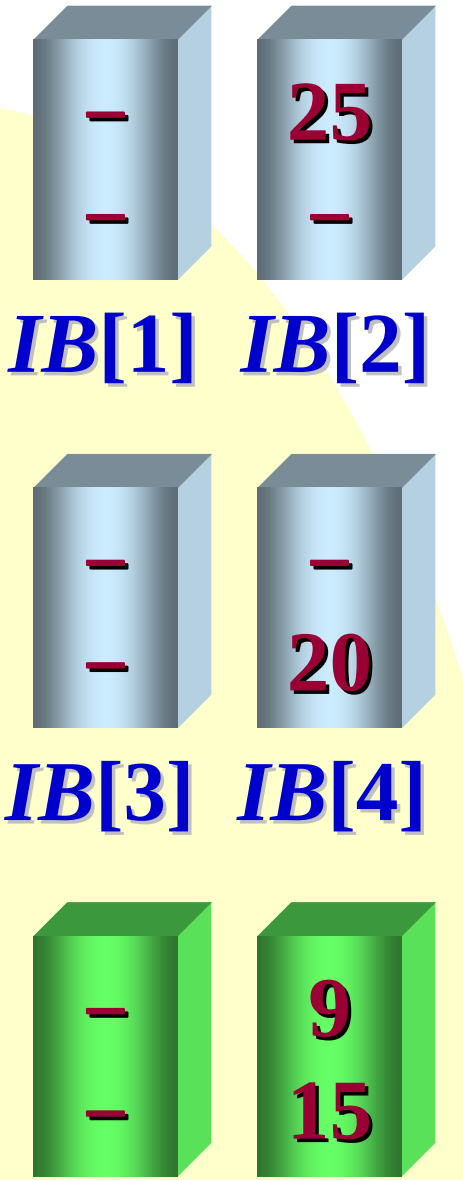
并行操作的缓冲区处理

- 如果采用 k 路归并对 k 个归并段进行归并，至少需要 k 个输入缓冲区和 1 个输出缓冲区。每个缓冲区存放一个页块的信息。
- 但要同时进行输入、内部归并、输出操作，这些缓冲区就不够了。例如，
  - 在输出缓冲区满需要向磁盘写出时，就必须中断内部归并的执行。
  - 在某一输入缓冲区空，需要从磁盘上再输入一个新的页块的对象时，也不得不中断内部归并。

- 由于内外存信息传输的时间与CPU的运行时间相比要长得多，所以使得内部归并经常处于等待状态。
- 为了改变这种状态，希望使输入、内部归并、输出并行进行。对于 k 路归并，必须设置 $2k$ 个输入缓冲区和 2 个输出缓冲区。
- 示例：给每一个归并段固定分配 2 个输入缓冲区，做 2 路归并。假设存在 2 个归并段:
  - **Run0**：对象的关键码是 **1, 3, 7, 8, 9**
  - **Run1**：对象的关键码是 **2, 4, 15, 20, 25**
- 假设每个缓冲区可容纳 2 个对象。需要设置 4 个输入缓冲区 $IB[i]$, $1 \le i \le 4$，2 个输出缓冲区 $OB[0]$和$OB[1]$。


1
3
2
4
IB[1]
IB[2]
归并到
OB[0]
–
3
–
4
IB[1]
IB[2]
归并到
OB[1]
–
–
–
–
IB[3]
IB[4]
读入到
IB[3]
7
8
–
–
IB[3]
IB[4]
读入到
IB[4]
写出
OB[0]
–
–
–
–
OB[0]
OB[1]
1
2
–
–
OB[0]
OB[1]


–
–
–
–
IB[1]
IB[2]
归并到
OB[0]
9
–
–
–
IB[1]
IB[2]
归并到
OB[1]
7
8
15
20
IB[3]
IB[4]
读入到
IB[1]
写出
OB[1]
–
–
15
20
IB[3]
IB[4]
读入到
IB[2]
写出
OB[0]
–
–
3
4
OB[0]
OB[1]
7
8
–
–
OB[0]
OB[1]



- 由示例可知，采用 **2*k*** 个输入缓冲区和 **2** 个输出缓冲区，可实现输入、输出和 ***k*** 路内部归并的并行操作。
- 但这 **2*k*** 个输入缓冲区如果平均分配给 ***k*** 个归并段，当其中某一个归并段比其它归并段短很多时，分配给它的缓冲区早早就空闲了，不能达到充分利用内存区的目的。

- 因此，不应为各归并段分别分配固定的两个缓冲区，缓冲区的分配应当是动态的，可根据需要为某一归并段分配缓冲区。但不论何时，每个归并段至少需要一个包含来自该归并段的对象的输入缓冲区。

k 路归并时动态分配缓冲区的实施步骤

☆ 为 k 个初始归并段各建立一个缓冲区的链式队列，开始时为每个队列先分配一个输入缓冲区。另外建立空闲缓冲区的链式栈，把其余 k 个空闲的缓冲区送入此栈中。输出缓冲区**OB**定位于**0**号输出缓冲区。

- 用 ***LastKey[i]*** 存放第 ***i*** 个归并段最后输入的关键码，用 ***NextRun*** 存放 ***LastKey[i]*** 最小的归并段段号；若有几个 ***LastKey[i]*** 都 是最小时，将序号最小的 ***i*** 存放到 ***NextRun*** 中。如果***LastKey[NextRun]*** $\neq \infty$，则从空闲缓冲区栈中取一个空闲缓冲区，预先链入段号为 ***NextRun*** 的归并段的缓冲区队列中。
- 使用函数 ***kwaymerge*** 对 ***k*** 个输入缓冲区队列中的对象进行 ***k*** 路归并，结果送入输出缓冲区 ***OB*** 中。归并一直持续到输出缓冲区 ***OB*** 变满或者有一个关键码为 ∞ 的对象被归并到***OB*** 中为止。

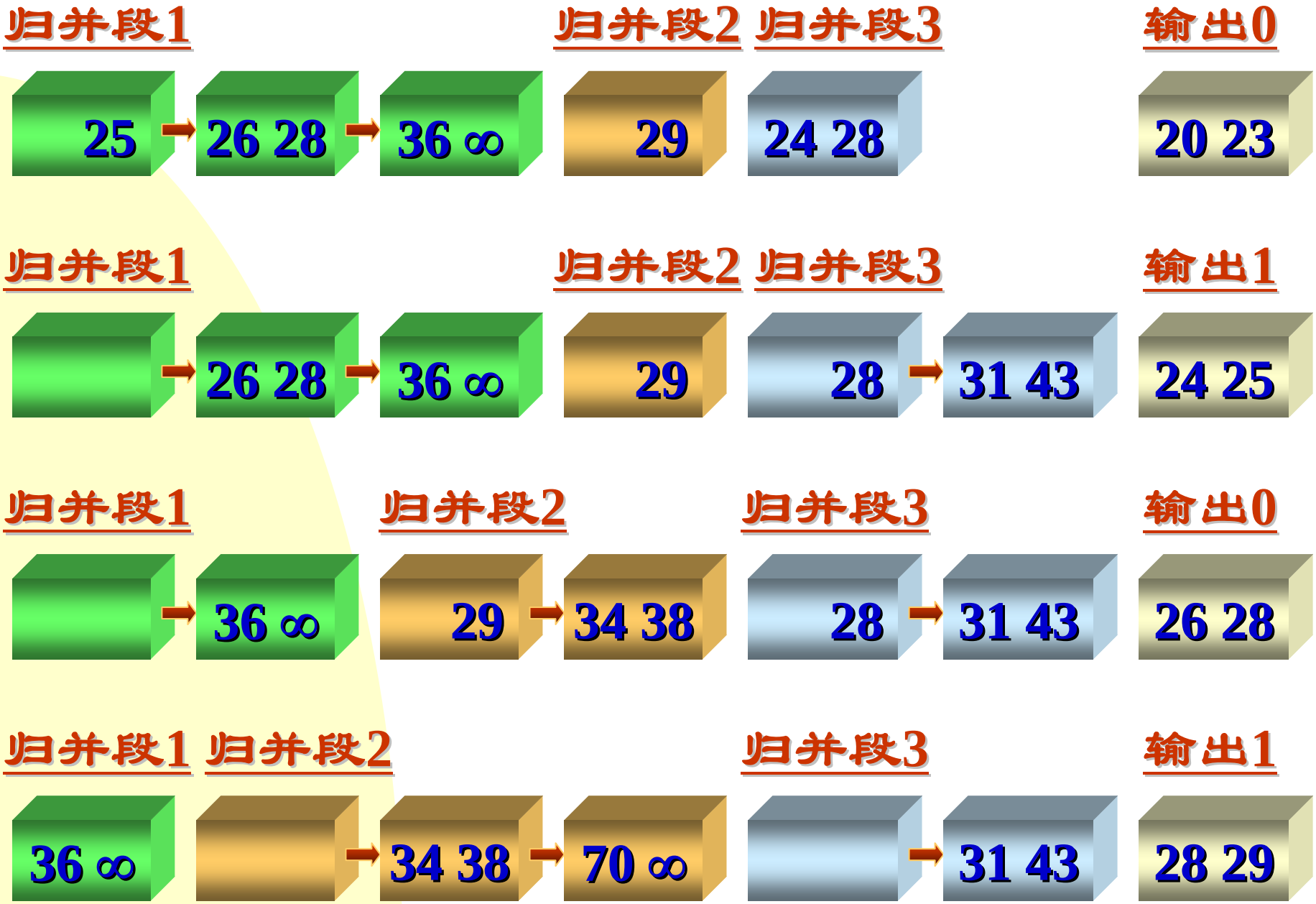
如果一个输入缓冲区变空，则 ***kwaymerge*** 进到该输入缓冲区队列中的下一个缓冲区，同时将变空的位于队头的缓冲区从队列中退出，加入到空闲缓冲区栈中。

但如果在输出缓冲区变满或关键码为 ∞ 的对象被归并到输出缓冲区***OB***的同时一个输入缓冲区变空，则 ***kwaymerge*** 不进到该输入缓冲区队列中的下一个缓冲区，变空的缓冲区也不从队列中退出，归并暂停。

一直等着，直到磁盘输入或磁盘输出完成为止，继续归并。

- 如果一个输入缓冲区读入完成，将它链入适当归并段的缓冲区队列中。然后确定满足***LastKey*[*NextRun*]** 的最小的 ***NextRun***，确定下一步将读入哪一个归并段的对象。
- 如果 ***LastKey*[*NextRun*]** $\neq \infty$，则从空闲缓冲区栈中取一个空闲缓冲区，从段号为 ***NextRun*** 的归并段中读入下一块，存放到这个空闲缓冲区中。
- 开始写出输出缓冲区***OB***的对象，再将输出缓冲区定位于**1**号输出缓冲区。
- 如果关键码为 ∞ 的对象尚未被归并到输出缓冲区***OB***中，转到继续操作；否则，一直等待，直到写出完成，然后算法结束。

- 示例：假设对如下三个归并段进行 3 路归并。每个归并段由3块组成，每块有2个对象。各归并段最后一个对象关键码 为∞。
  - 归并段1 { 20, 25 } { 26, 28 } { 36, ∞ }
  - 归并段2 { 23, 29 } { 34, 38 } { 70, ∞ }
  - 归并段3 { 24, 28 } { 31, 34 } { 50, ∞ }
- 设立6个输入缓冲区，2个输出缓冲区。利用动态缓冲算法，各归并段输入缓冲区队列及输出缓冲区状态的变化如图所示。


归并段1
归并段2
归并段3
输出0
25
26 28
36 ∞
29
24 28
20 23
归并段1
归并段2
归并段3
输出1
26 28
36 ∞
29
28
31 43
24 25
归并段1
归并段2
归并段3
输出0
36 ∞
29
34 38
28
31 43
26 28
归并段1
归并段2
归并段3
输出1
36 ∞
34 38
70 ∞
31 43
28 29

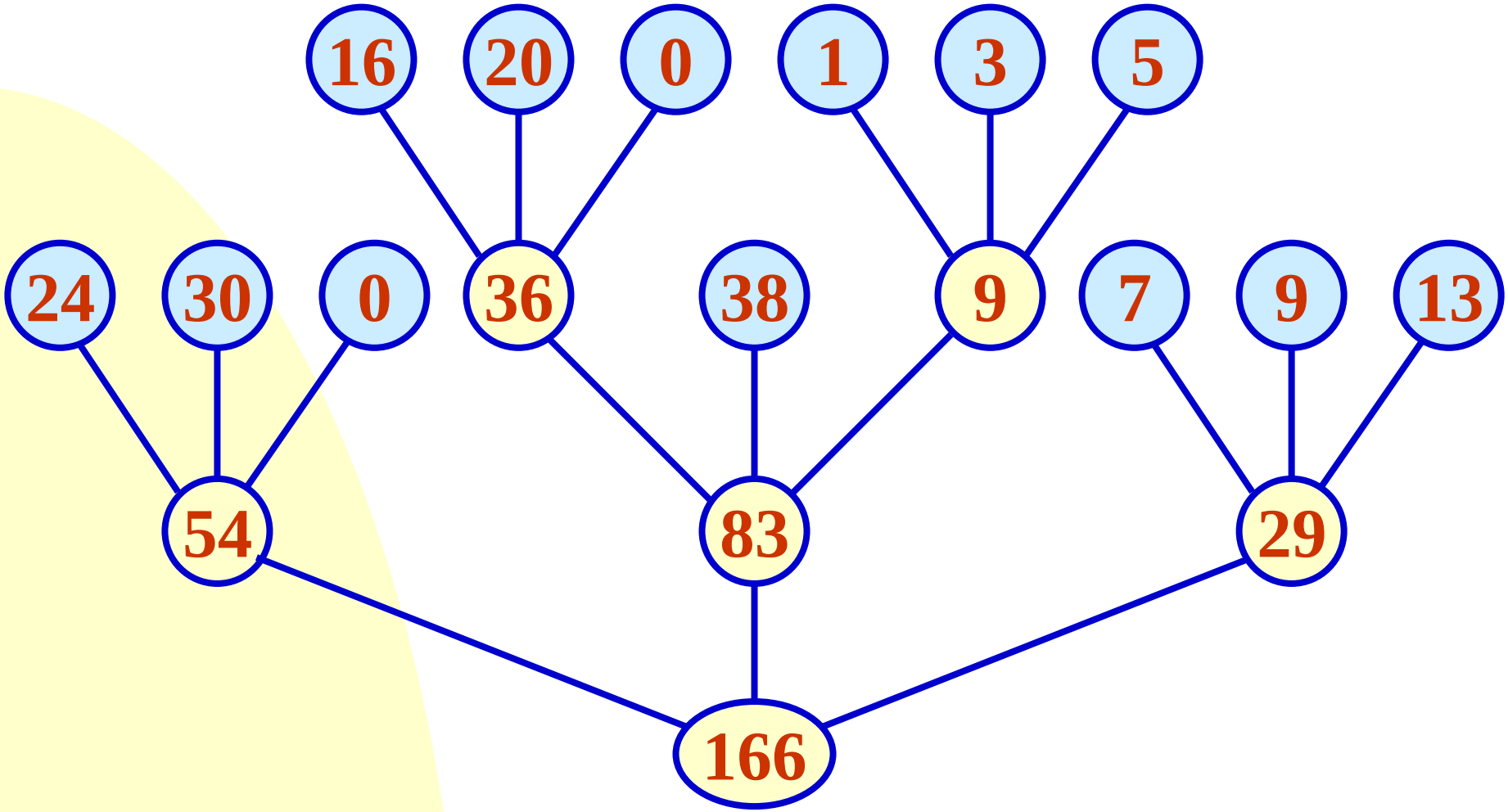

归并段1
归并段2
归并段3
输出0
36 ∞
38
70 ∞
43
50 ∞
31 34
归并段1
归并段2
归并段3
输出1
∞
70 ∞
43
50 ∞
36 38
归并段1
归并段2
归并段3
输出0
∞
70 ∞
∞
43 50
归并段1
归并段2
归并段3
输出1
∞
∞
70 ∞

- 对于较大的k, 为确定哪一个归并段的输入缓冲区最先变空，可对 $LastKey[i], 0 \le i \le k-1$，建立一棵败者树。通过 $\log_2 k$ 次比较就可确定哪一个归并段的缓冲区队列最先变空。
- 对于较大的k，函数 *kwaymerge* 使用了败者树进行归并。
- 除最初的 k 个输入页块的读入和最后一个输出页块的写出外，其它所有输入，输出和内部归并都是并行执行的。此外，也有可能在 k 个归并段归并完后，需要立即开始对另外 k 个归并段执行归并。所以，在对 k 个归并段进行归并的最后阶段，就开始下一批 k 个归并段的输入。
- 此算法假定所有的页块大小相同。

最佳归并树

- 归并树是描述归并过程的 ***m*** 叉树。因为每一次做 ***m*** 路归并都需要有 ***m*** 个归并段参加，因此，归并树是只有度为**0**和度为 ***m*** 的结点的正则 ***m*** 叉树。
- 示例：设有**13**个长度不等的初始归并段，其长度**(**对象个数**)**分别为

  0, 0, 1, 3, 5, 7, 9, 13, 16, 20, 24, 30, 38
- 其中长度为 **0** 的是空归并段。对它们进行 **3** 路归并时的归并树如图所示。



此归并树的带权路径长度为

$WPL = 24+30+38+7+9+13)*2+ (16+20+1+3+5)*3$

$= 377$。

- 在归并树中
  - 各叶结点代表参加归并的各初始归并段
  - 叶结点上的权值即为该初始归并段中的对象个 数
  - 根结点代表最终生成的归并段
  - 叶结点到根结点的路径长度表示在归并过程中的读对象次数
  - 各非叶结点代表归并出来的新归并段
  - 归并树的带权路径长度 ***WPL*** 即为归并过程中的总读对象数。因而，在归并过程中总的读写对象次数为 **2*WPL = 754**。

- 不同的归并方案所对应的归并树的带权路径长度各不相同。
- 为了使得总的读写次数达到最少，需要改变归并方案，重新组织归并树。
- 可将哈夫曼树的思想扩充到 ***m*** 叉树的情形。在归并树中，让对象个数少的初始归并段最先归并，对象个数多的初始归并段最晚归并，就可以建立总的读写次数达到最少的最佳归并树。
- 例如，假设有**11**个初始归并段, 其长度(对象个数)分别为

  1, 3, 5, 7, 9, 13, 16, 20, 24, 30, 38

  做**3**路归并。

- 为使归并树成为一棵正则三叉树，可能需要补入空归并段。补空归并段的原则为：
  - 若参加归并的初始归并段有 n 个，做 m 路平衡归并。因归并树是只有度为 0 和度为 m 的结点的正则 m 叉树，设度为 0 的结点有 $n_0(=n)$ 个，度为 m 的结点有 n_m 个，则有

$$n_0 = (m-1)n_m + 1$$

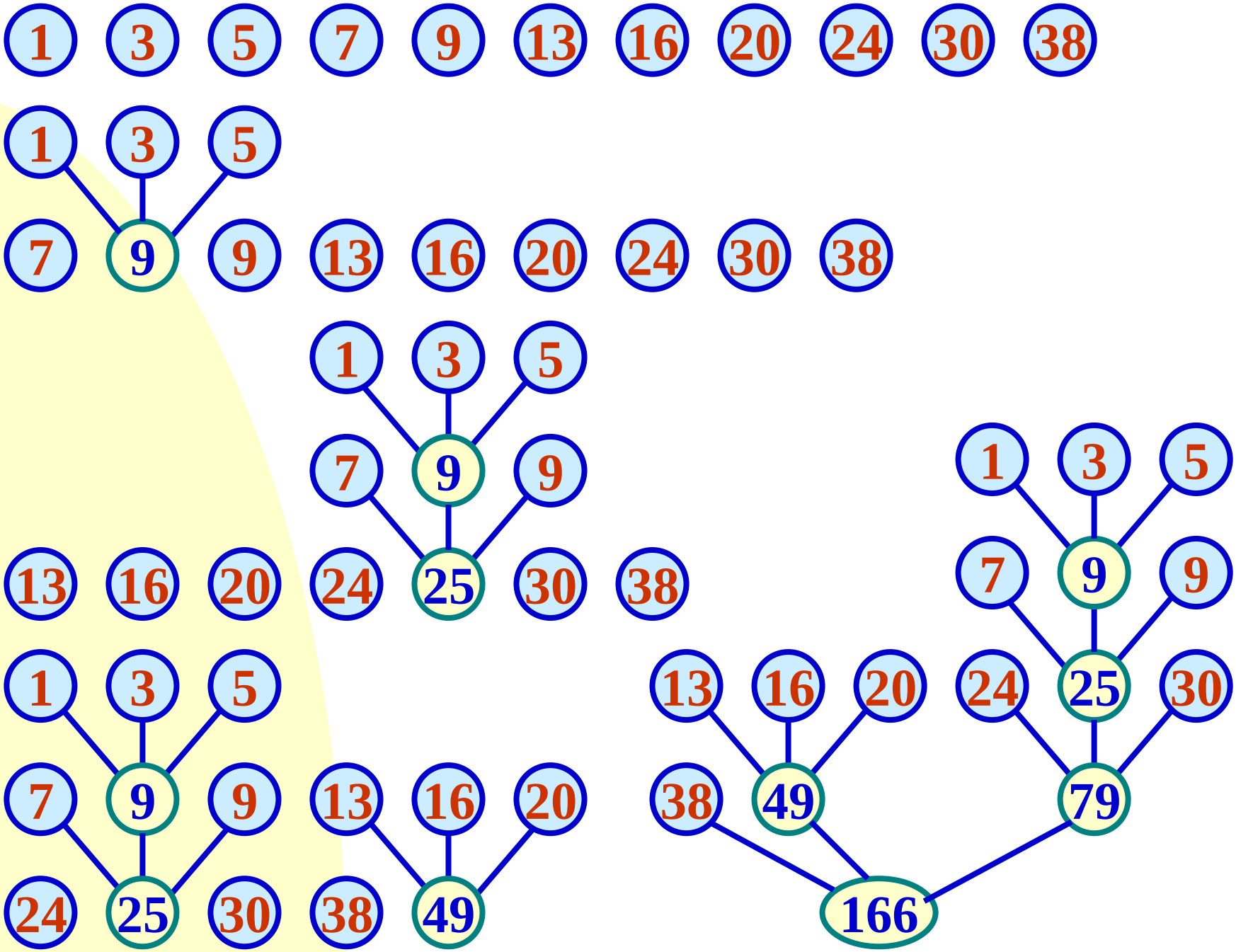
$$n_m = (n_0 - 1) / (m - 1)。$$

  - 如果 $(n_0-1) \% (m-1) = 0$, 则说明这 n_0 个叶结点(即初始归并段)正好可以构造 m 叉归并树。
  - 如果 $(n_0-1) \% (m-1) = u \neq 0$，则对于这 n_0 个叶结点，其中的 u 个不足以参加 m 路归并。

- 故除了有 n_m 个度为 m 的内结点之外，还需增加一个内结点。它在归并树中代替了一个叶结点位置，被代替的叶结点加上刚才多出的 u 个叶结点，再加上 $m-u-1$ 个对象个数为零的空归并段，就可以建立归并树。
- 在示例中，$n_0 = 11$，$m = 3$，$(11-1) \% (3-1) = 0$，可以不加空归并段，直接进行3路归并。

- 它的带权路径长度

$$WPL = 38*1 + (13+16+20+24+30)*2 + (7+9)*3 + (1+3+5)*4 = 328。$$


1 3 5 7 9 13 16 20 24 30 38
1 3 5
7 9 9 13 16 20 24 30 38
1 3 5
7 9 9
13 16 20 24 25 30 38
1 3 5
7 9 9
24 25 30
1 3 5
7 9 9
24 25 30
13 16 20
38 49
13 16 20
38 49
166
79

- 如果做5路归并，让 $m = 5$，则有

  $$(11-1) / (5-1) = 2$$

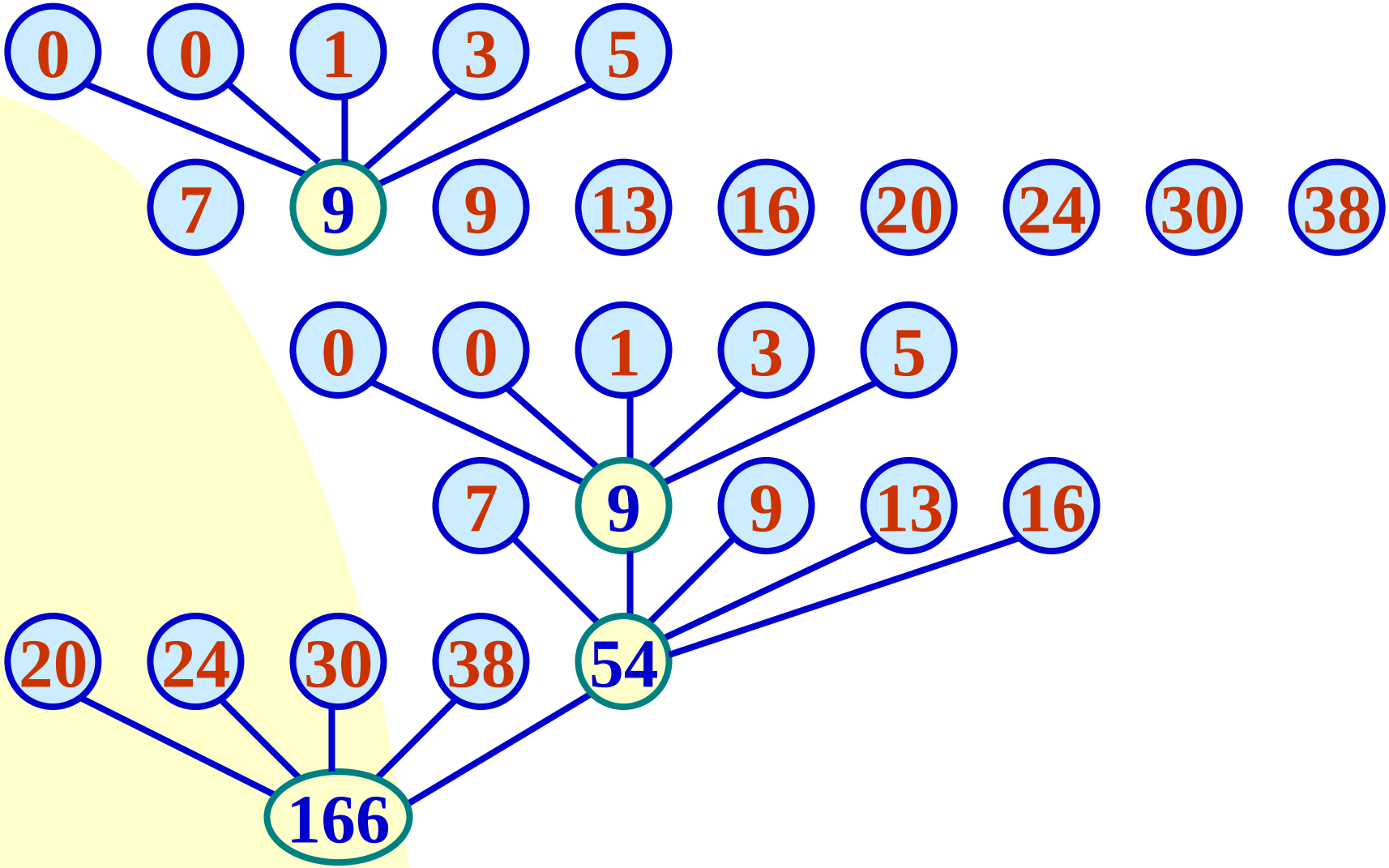
- 表示有2个度为5的内结点；但是，

  $$u = (11-1) \bmod (5-1) = 2 \neq 0$$

- 需要加一个内结点，它在归并树中代替了一个叶结点的位置，故一个叶结点参加这个内结点下的归并，需要增加的空初始归并段数为

  $$m-u-1 = 5-2-1 = 2$$

- 应当补充2个空归并段。则归并树如图所示。



带权路径长度 $WPL = (1+3+5)*3+(7+9+13+16)*2+(20+24+30+38) = 229$

小结 需要复习的知识点

- 排序的基本概念
  - 排序的基本概念
  - 关键码、初始关键码排列
  - 关键码比较次数、数据移动次数
  - 稳定性
  - 附加存储
- 插入排序
  - 用事例表明直接插入排序、折半插入排序的过程
  - 直接插入排序和折半插入排序的算法

- 排序的性能分析

- 当待排序的关键码序列已经基本有序时，用直接插入排序最快

- 选择排序
  - 用事例表明直接选择排序、锦标赛排序、堆排序的过程
  - 直接选择排序和堆排序的算法
  - 三种选择排序的性能分析
- 用直接选择排序在一个待排序区间中选出最小的数据时，与区间第一个数据对调，不是顺次后移。这导致方法不稳定。

- 当在 n 个数据（n很大）中选出最小的 5 ~ 8 个数据时，锦标赛排序最快。
- 锦标赛排序算法将待排序数据个数 n 补足到 2的 k 次幂

$$2^{k-1} < n \leq 2^k$$

- 在堆排序中将待排序的数据组织成完全二叉树的顺序存储。

- 交换排序
  - 用事例表明起泡排序和快速排序的过程
  - 起泡排序的算法

- 快速排序的递归算法和用栈实现的非递归算法

➢ 快速排序是一个递归的排序法

➢ 当待排序关键码序列已经基本有序时，快速排序显著变慢。

■ 二路归并排序

- 用事例表明二路归并排序的过程
- 二路归并排序的非递归算法
- 该算法的性能分析

➢ 归并排序可以递归执行

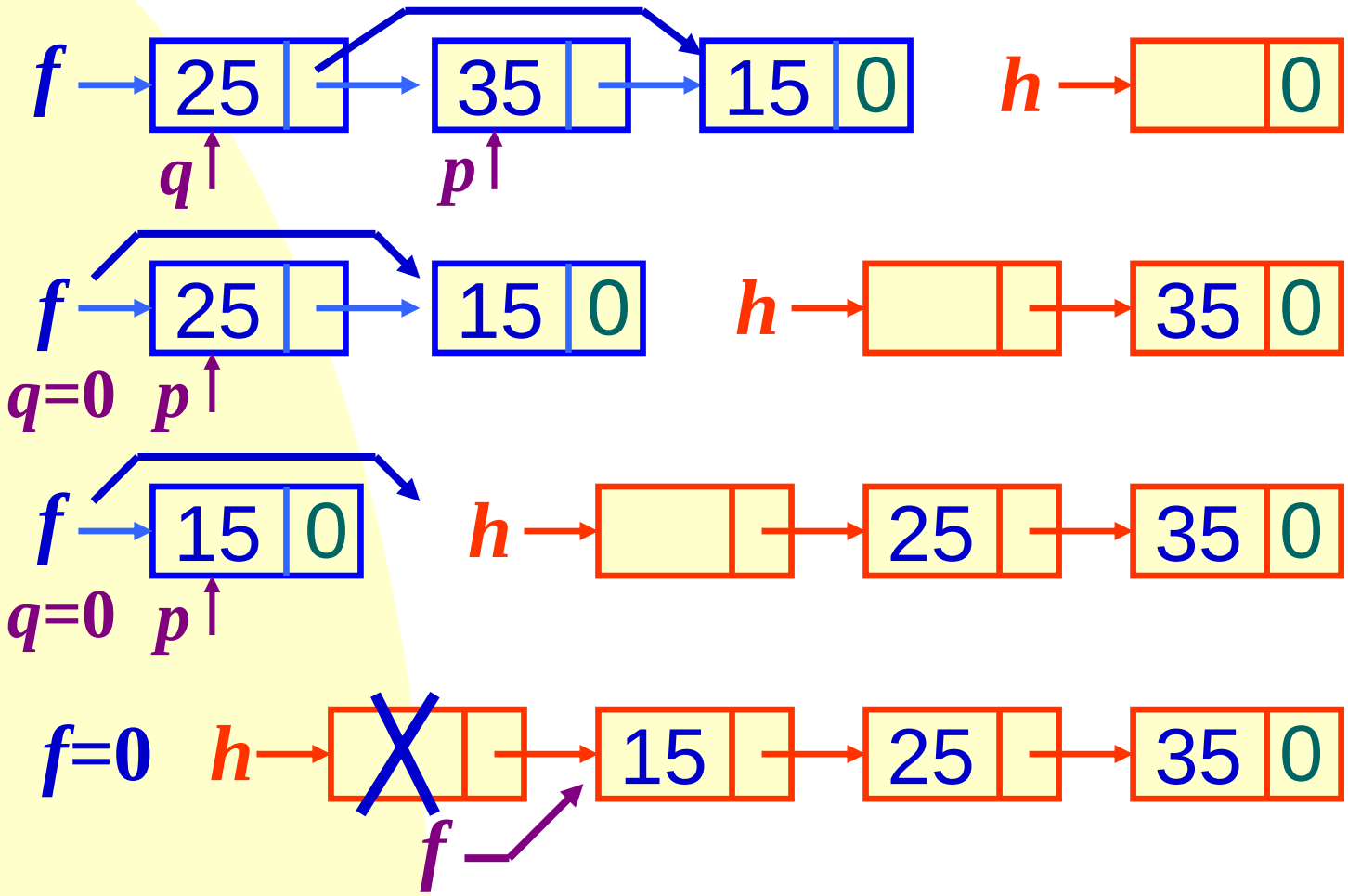
➢ 归并排序需要较多的附加存储。

- 归并排序对待排序关键码的初始排列不敏感，故排序速度较稳定。
- 外排序
  - 多路平衡归并排序的过程
  - 外排序的时间分析
  - 利用败者树进行多路平衡归并
  - 利用置换选择方法生成不等长的初始归并段
  - 最佳归并树的构造及***WPL***计算

各种排序方法的比较

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	
折半插入排序	$n\log_2 n$		0	n^2	√	1	
起泡排序	n	n^2	0	n^2	√	1	
快速排序	$n\log_2 n$	n^2	$n\log_2 n$	n^2	×	$\log_2 n$	n^2
简单选择排序	n^2		0	n	×	1	
锦标赛排序	$n\log_2 n$		$n\log_2 n$		√	n	
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	

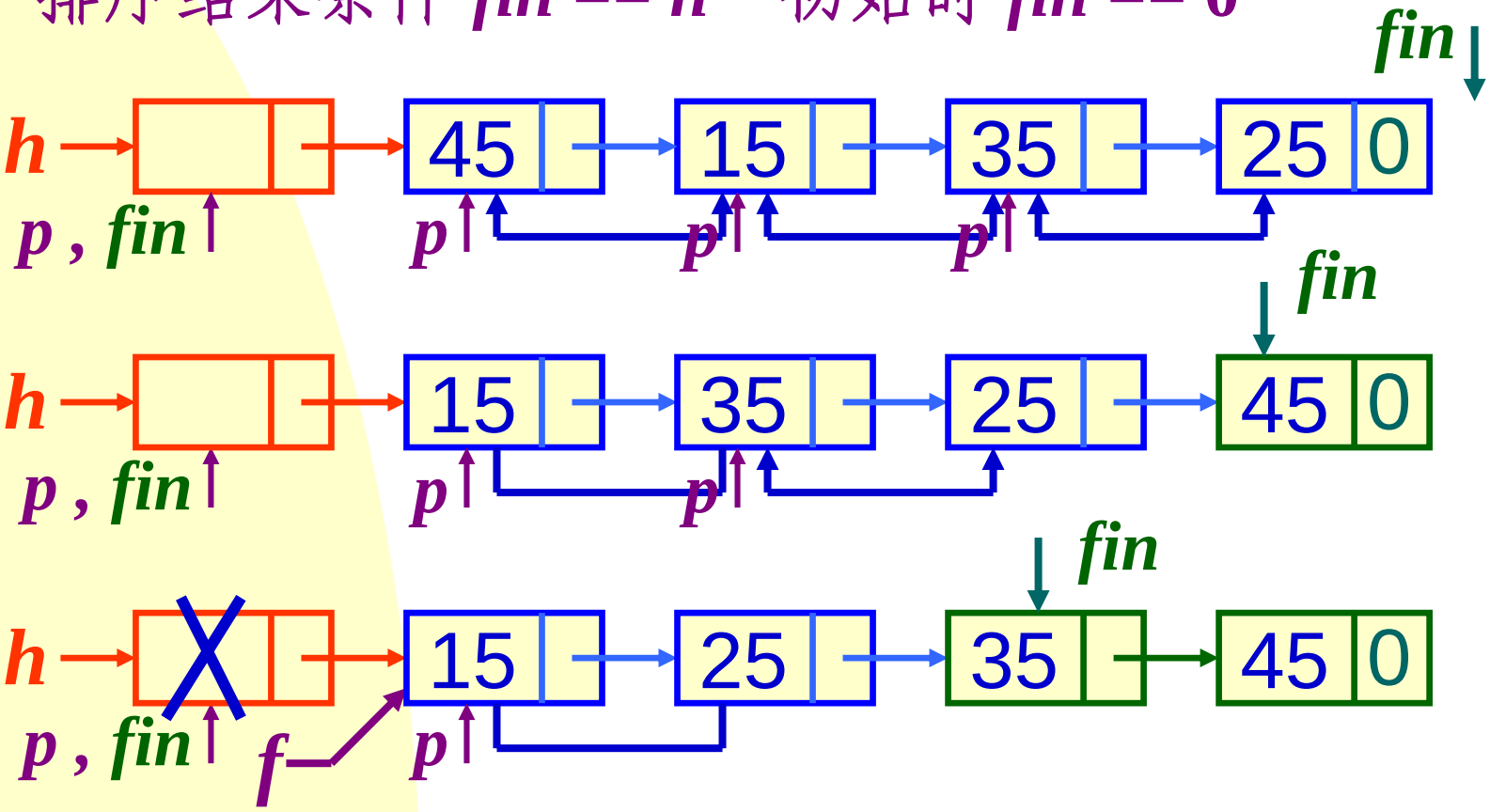
例1 链表选择排序算法示例



例2 链表起泡排序算法示例

每趟循环结束条件 $p \rightarrow link \rightarrow link == fin$

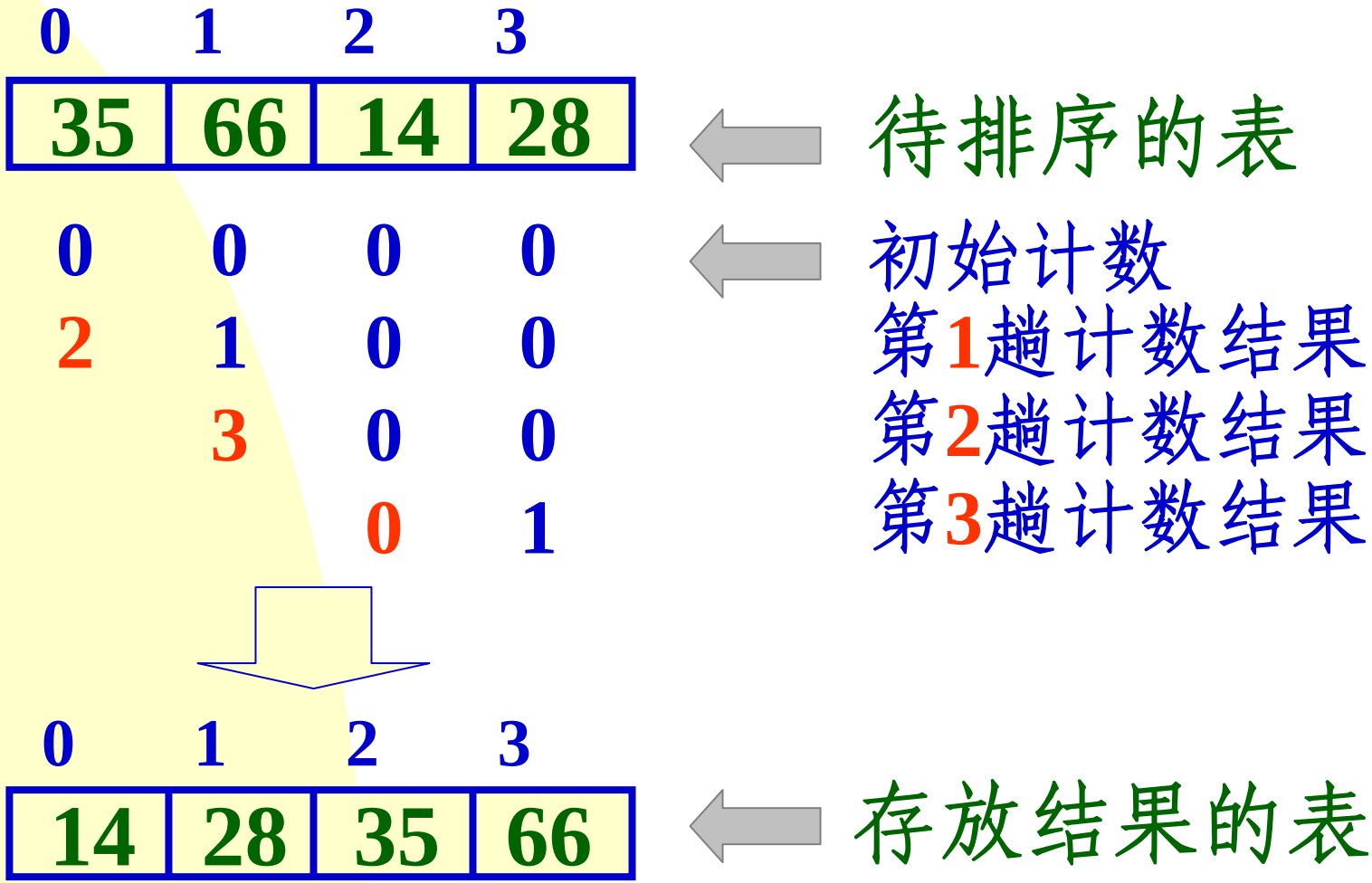
排序结束条件 $fin == h$　初始时 $fin == 0$



例3 计数排序（count Sorting）

- 有一个用数组表示的待排序的表。
- 算法针对表中的每个记录，扫描待排序的表一趟，统计表中有多少个记录的关键码比该记录的关键码小。
- 假设针对某一个记录，统计出的计数值为c，那么，这个记录在新的有序表中的合适的存放位置即为c。
- 排序结果存放到另一个新的表中。

- 表中所有待排序的关键码互不相同。



- 适用于计数排序的数据表定义

```
const int  DefaultSize = 100;
template <class Type> class datalist
template <class Type> class Element {
private:
    Type key;                  //关键码
    field otherdata;           //其它数据成员
public:
    Type getKey ( ) { return key; }
    void setKey ( const Type x ) { key = x; }
}
```

```
template <class Type> class datalist {
public:
  datalist ( int MaxSz = DefaultSize ) :
   MaxSize ( Maxsz ), CurrentSize (0) {
   Vector = new Element <Type> [MaxSz]; }
private:
    Element <Type> * Vector;
    int MaxSize, CurrentSize;
}
```

- 实现计数排序的算法

```
template <class Type>
void countsort ( datalist<Type> & initList,
   datalist<Type> & resultList ) {
//initList是待排序表，resultList是结果表
    int i, j;
    int *c = new int[initList.CurrentSize];
    // c是存放计数的临时表
    for ( i = 0; i < initList.CurrentSize; i++ )
        c[i] = 0;
    //初始化，计数值都为0
```

```
    for ( i = 0; i < initList.CurrentSize−1; i++ )
      for ( j = i+1; j < initList.CurrentSize; j++ )
        if (initList.Vector[j].getKey( ) <
            initList.Vector[i].getKey ( ) )
            c[i]++;
        else c[j]++;        //统计
    for ( i = 0; i < initList.CurrentSize; i++ )
      resultList.Vector[ c[i] ] = initList.Vector[i];
    resultList.CurrentSize = initList.CurrentSize;
}
```