

第九章 排序

- 排序概念及其算法性能分析
- 插入排序
- 快速排序
- 选择排序
- 归并排序
- 基于链表的排序算法
- 分配排序
- 内部排序算法的分析
- 本章小结

9.1 排序的概念及其算法性能分析

- **排序**：将一组杂乱无章的数据按一定的规律顺序排列起来。
- **数据表(*datalist*)**：它是待排序数据对象的有限集合。
- **排序码(*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**排序码**。每个数据表用哪个属性域作为排序码，要视具体的应用需要而定。

- **排序算法的稳定性：**如果在对象序列中有两个对象 $r[i]$ 和 $r[j]$ ，它们的排序码 $k[i] = k[j]$ ，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。**
- 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**受对象排序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。
- **算法执行时所需的附加存储**：评价算法好坏的另一标准。

待排序数据表的类定义

```
#include <iostream.h>
const int DefaultSize = 100;
template <class Type> class DataList {
template <class Type> class Element {
friend class DataList <Type>;
private:
    Type key; //排序码
    field otherdata; //其它数据成员
public:
    Element ( ) : key(0), otherdata(NULL) { }
    Type GetKey ( ) { return key; } //提取排序码
    void SetKey ( const Type x ) { key = x; } //修改
```

```
Element <Type> & operator =  
    ( Element <Type> &x )  
    { key = x->key; otherdata = x->otherdata; }  
//赋值
```

```
int operator == ( Element <Type> &x )  
    { return key == x->key; } //判this == x
```

```
int operator <= ( Element <Type> &x )  
    { return key <= x->key; } //判this ≤ x
```

```
int operator < ( Element <Type> &x )  
    { return key < x->key; } //判this < x
```

```
int operator > ( Element <Type> &x )  
    { return key > x->key; } //判this > x
```

```
};
```

```
template <class Type> class DataList {
```

```
private:
```

```
    Element <Type> *Vector; //存储向量
```

```
    int MaxSize, CurrentSize; //最大与当前个数
```

```
public:
```

```
    Datalist ( int MaxSz = DefaultSize ) :
```

```
        MaxSize(Maxsz), CurrentSize (0)
```

```
    { Vector = new Element <Type> [MaxSz]; }
```

```
void Sort ( ); //排序
```

```
void Swap ( Element <Type> &x,
```

```
            Element <Type> &y ) { //对换
```

```
    Element <Type> temp = x;
```

```
    x = y; y = temp; }
```

```
};
```



9.2 插入排序 (Insert Sorting)

基本方法：每步将一个待排序的对象，按其排序码大小，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。

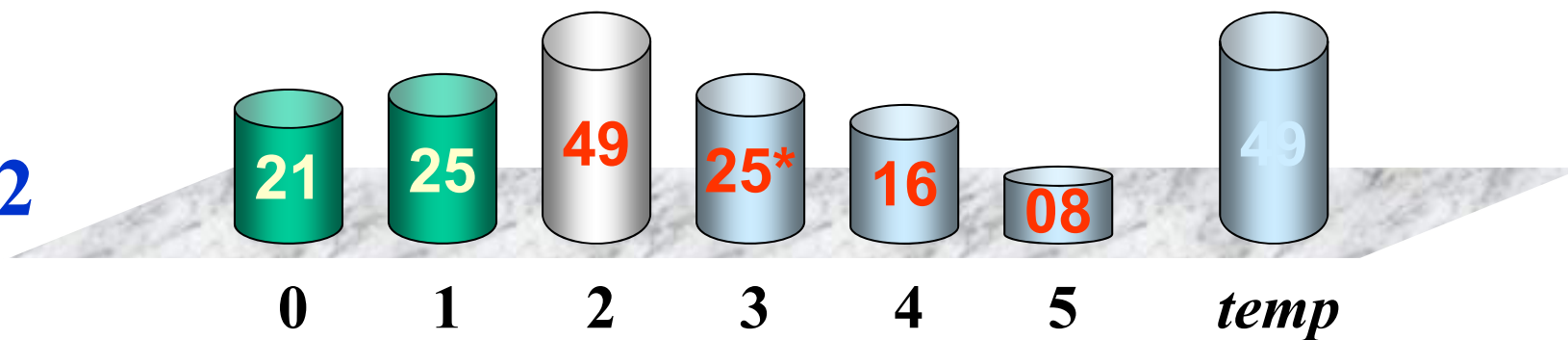
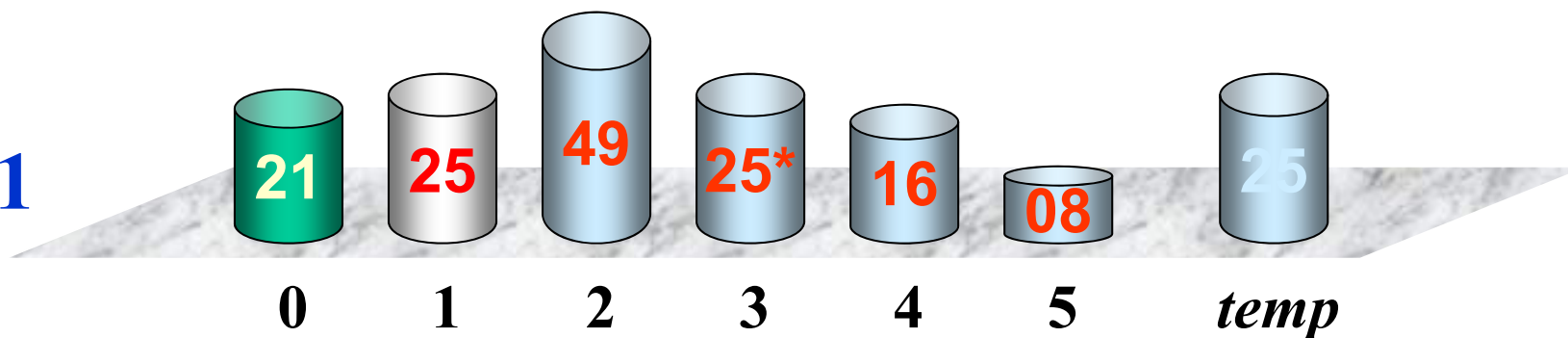
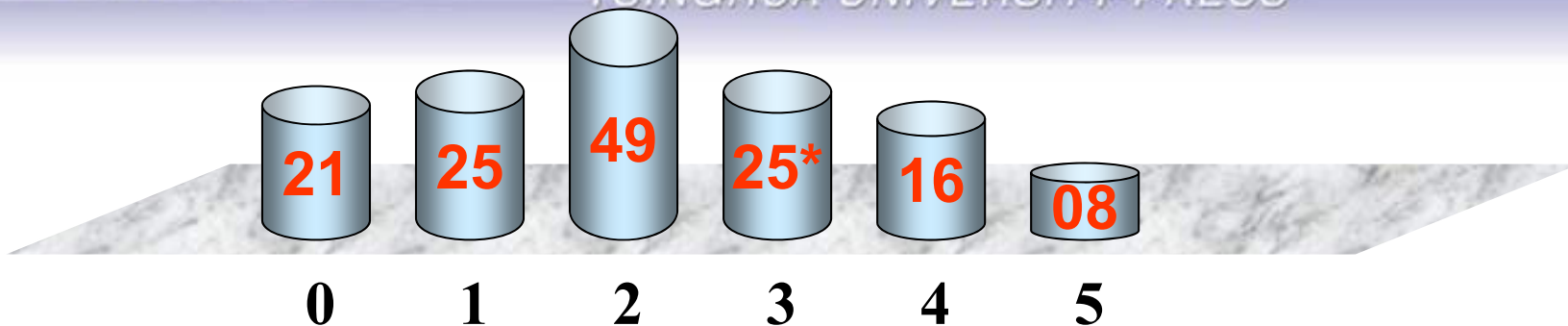
- 分类原理，根据往已经排好序的有序数据表中寻找插入位置的方法的不同而区分。
 - 直接插入排序(Insert Sort)
 - 折半插入排序(Binary Insert Sort)
 - 希尔排序(Shell Sort)

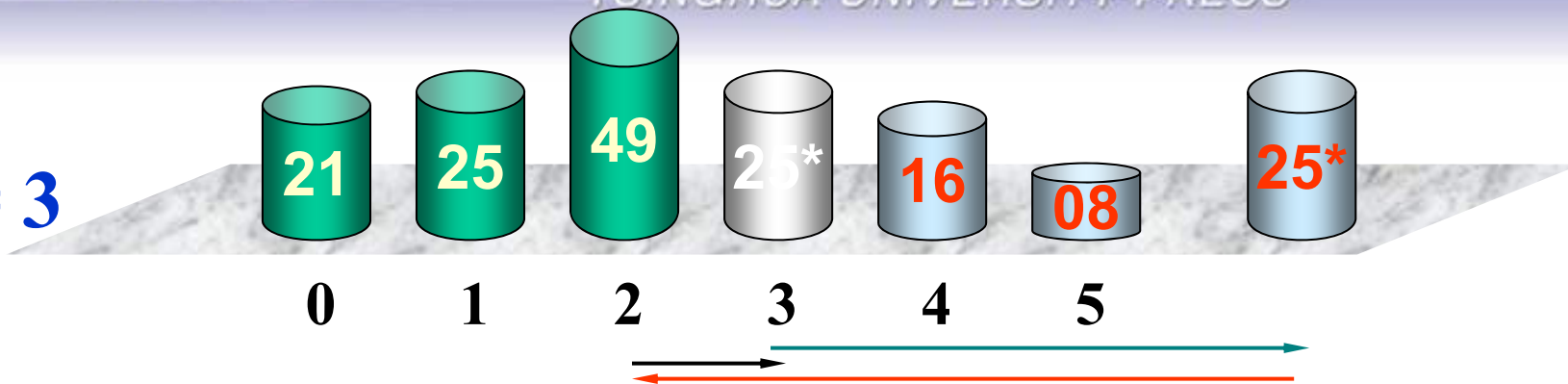
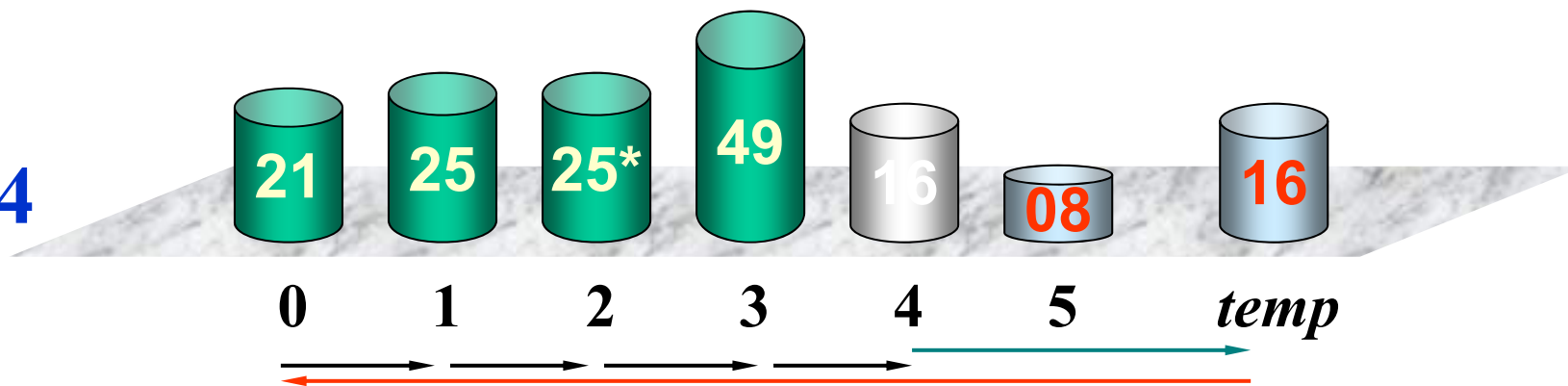
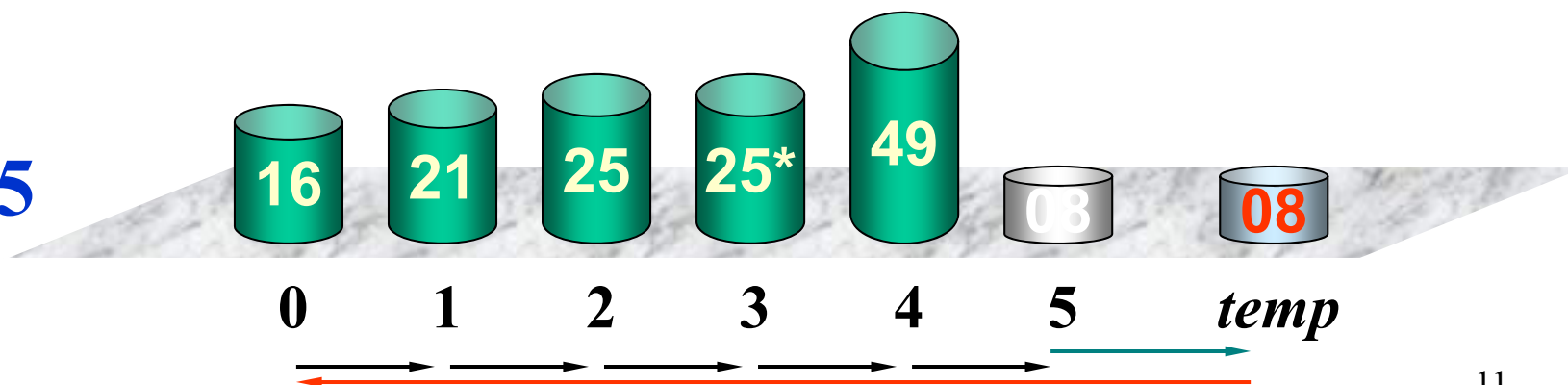
直接插入排序 (Insert Sort)

- **基本思想：**当插入第 i ($i \geq 1$) 个对象时，前面的 $V[0], V[1], \dots, V[i-1]$ 已经排好序。这时，用 $V[i]$ 的排序码与 $V[i-1], V[i-2], \dots$ 的排序码顺序进行比较，找到插入位置即将 $V[i]$ 插入，原来位置上的对象向后顺移。

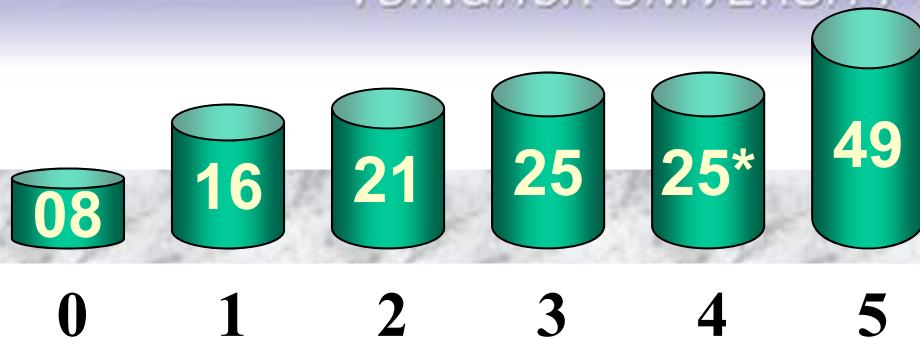


各趟排序结果

 $i = 1$ $i = 2$ 

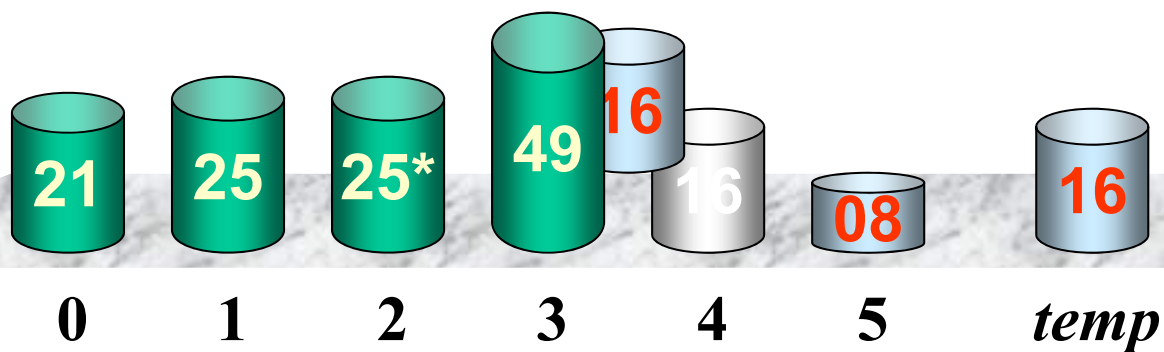
$i = 3$  $i = 4$  $i = 5$ 

完成

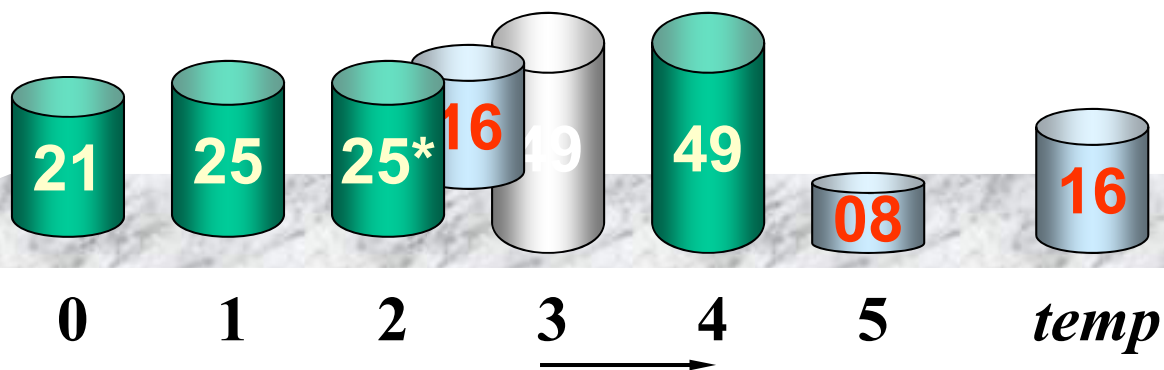


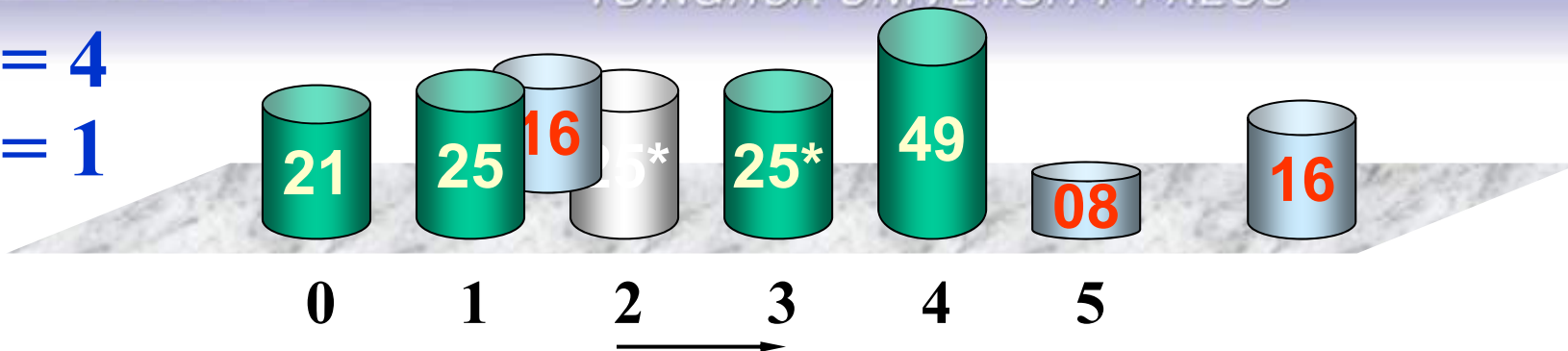
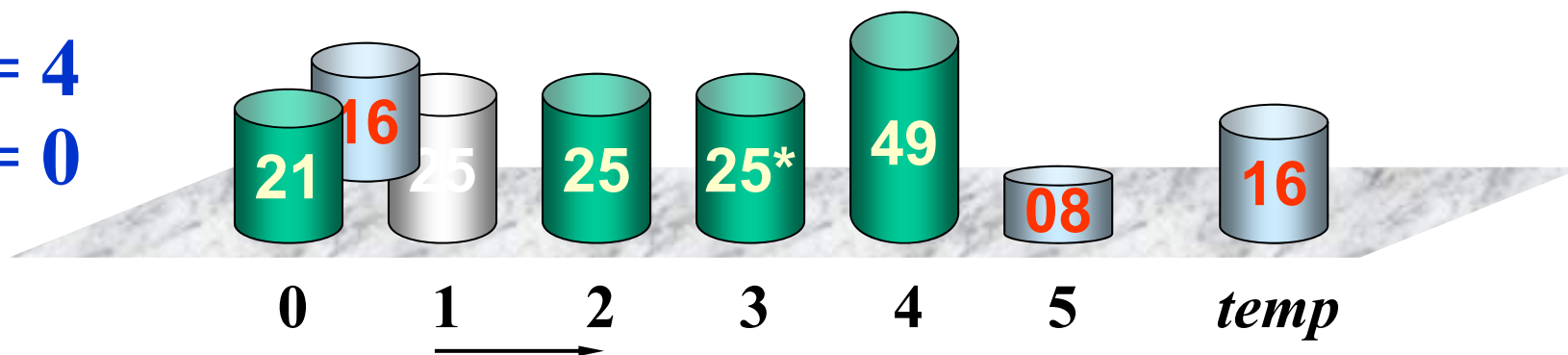
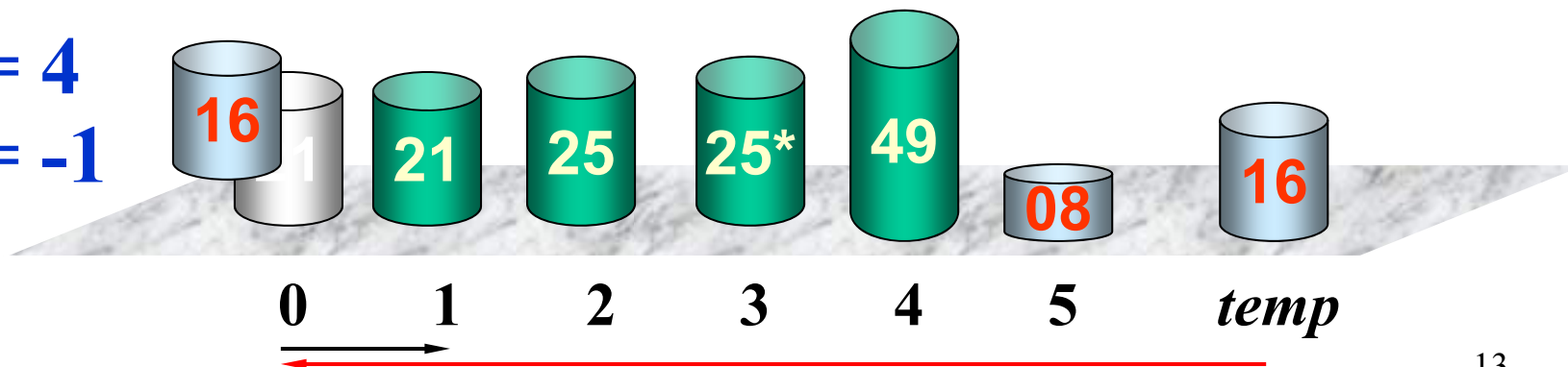
$i = 4$ 时的排序过程

$i = 4$
 $j = 3$



$i = 4$
 $j = 2$



$i = 4$
 $j = 1$  $i = 4$
 $j = 0$  $i = 4$
 $j = -1$ 


插入排序动画

45

34

78

12

34 

32

29

64

直接插入排序的算法

```
template <class Type> void DataList <Type> ::
```

```
InsertSort ( ) {
```

```
//按排序码key非递减顺序对表进行排序
```

```
    Element <Type> temp; int i, j;
```

```
    for ( i = 1; i < CurrentSize; i++ ) {
```

```
        temp = Vector[i];
```

```
        for ( j = i; j > 0; j-- ) //从后向前顺序比较
```

```
            if ( temp < Vector[j-1] )
```

```
                Vector[j] = Vector[j-1];
```

```
            else break;
```

```
        Vector[j] = temp;    }
```

```
}
```

算法分析

- 设待排序对象个数为 $CurrentSize = n$ ，则该算法的主程序执行 $n-1$ 趟。
- 排序码比较次数和对象移动次数与对象排序码的初始排列有关。
- 最好情况下，排序前对象已按排序码从小到大有序，每趟只需与前面有序对象序列的最后一个对象比较 1 次，移动 2 次对象，总排序码比较次数为 $n-1$ 次，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*个对象都做排序码比较，并且每做1次比较就要做1次数据移动，则总排序码比较次数*KCN*和对象移动次数*RMN*分别为：

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

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

- 平均情况下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)

- **基本思想**：设在顺序表中有一个对象序列 $V[0], V[1], \dots, V[n-1]$ 。其中， $V[0], V[1], \dots, V[i-1]$ 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 DataList <Type> :: BineryInsSort ( )
{
    Element <Type> temp; int Left, Right;
    for ( int i = 1; i < CurrentSize; i++ ) {
        Left = 0; Right = i-1; temp = Vector[i];
        while ( Left <= Right ) {
            int middle = ( Left + Right )/2;
            if ( temp < Vector[middle] )
                Right = middle - 1;
            else Left = middle + 1; }
        for ( int k = i-1; k >= Left; k-- )
            Vector[k+1] = Vector[k];
        Vector[Left] = temp;
    }
}
```

算法分析

- 折半搜索比顺序搜索查找快，所以折半插入排序就**平均性能**来说比直接插入排序要快。
- 它所需的排序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 $\lfloor \log_2 i \rfloor + 1$ 次排序码比较，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 $n=2^k$ ）用折半插入排序所进行的排序码比较次数为：

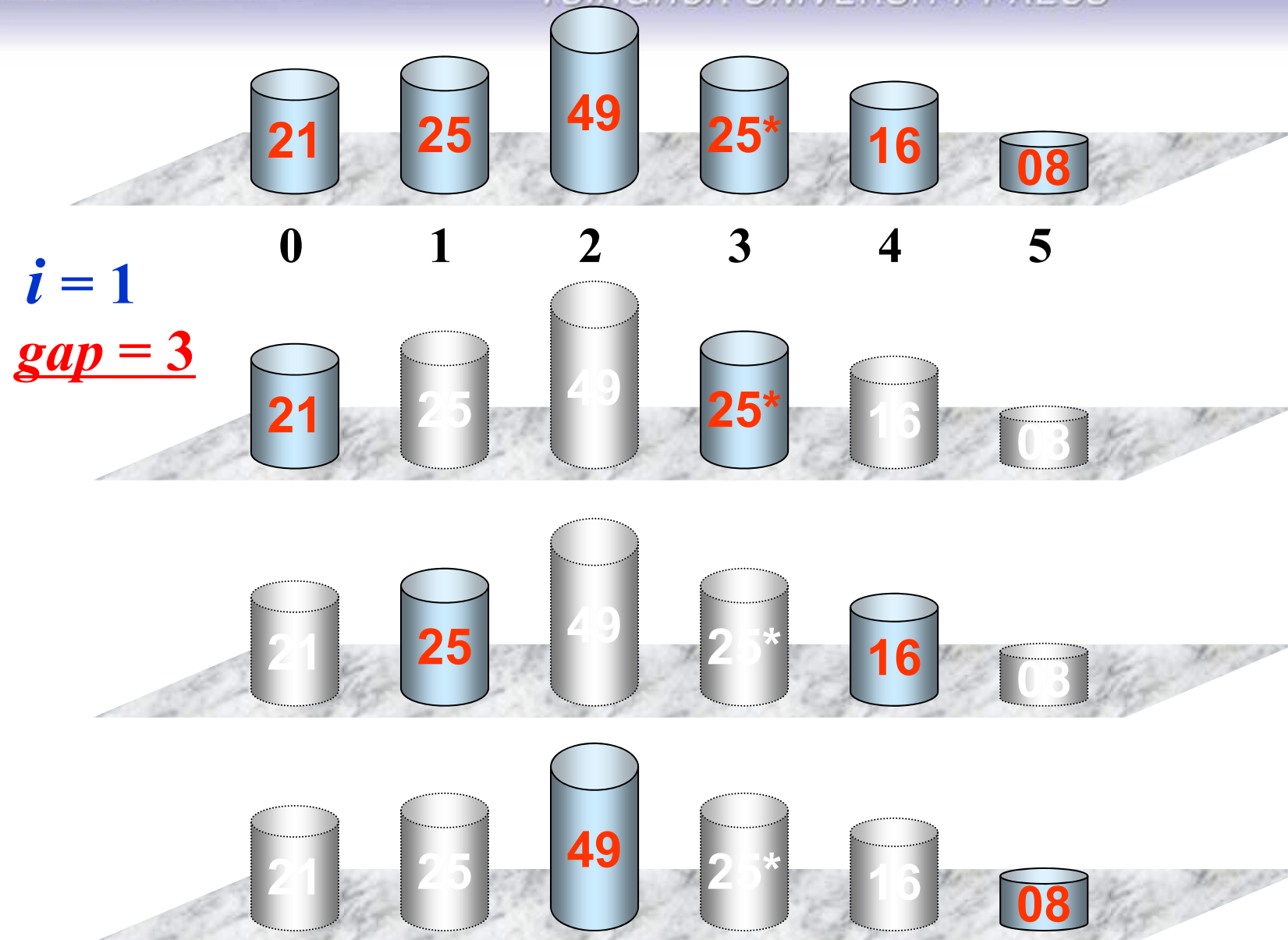
$$\sum_{i=1}^{n-1} (\lfloor \log_2 i \rfloor + 1) \approx n \cdot \log_2 n$$

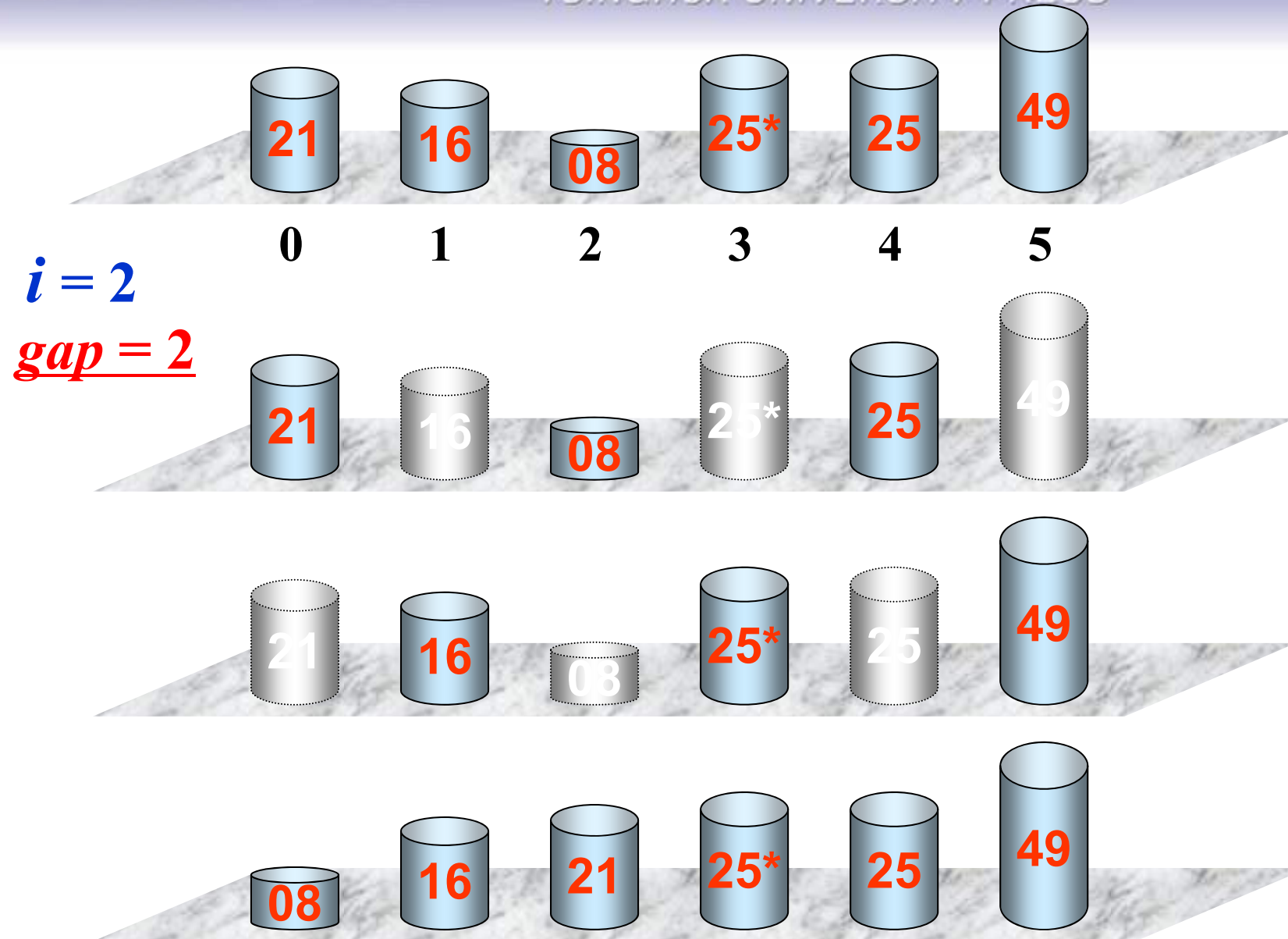
- 折半插入排序是一个**稳定**的排序方法。

- 当 n 较大时，总排序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排序码排好序或接近有序时，直接插入排序比折半插入排序执行的排序码比较次数要少。折半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。

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。



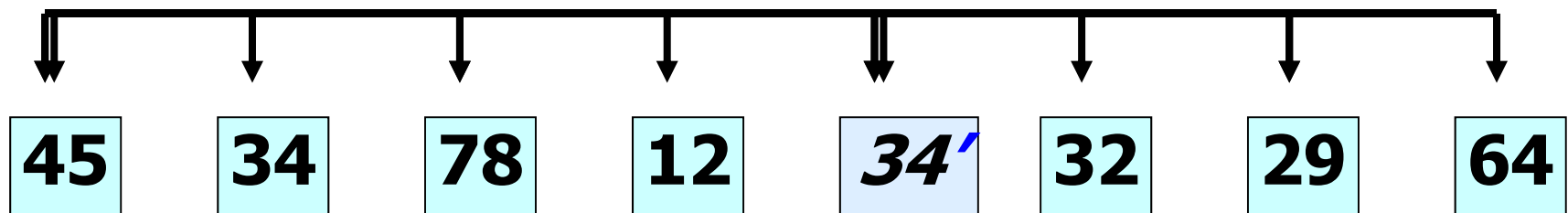




- 开始时 gap 的值较大，子序列中的对象较少，排序速度较快；随着排序进展， gap 值逐渐变小，子序列中对象个数逐渐变多，由于前面工作的基础，大多数对象已基本有序，所以排序速度仍然很快。

希尔排序过程

- 先给定一组严格递减的正整数增量 d_0, d_1, \dots, d_{t-1} , 且取 $d_{t-1} = 1$ 。
- 对于 $i=0, 1, \dots, t-1$, 进行下面各遍的处理:
 - ◆ 将序列分成 d_i 组, 每组中结点的下标相差 d_i
 - ◆ 对每组节点使用插入排序



希尔排序的算法

```
template <class Type> void DataList <Type> ::  
ShellSort ( ) {  
    Element <Type> temp;  
    int gap = CurrentSize / 2; //gap是子序列间隔  
    while ( gap != 0 ) { //循环, 直到gap为零  
        for ( int i = gap; i < CurrentSize; i++ ) {  
            temp = Vector[i]; //直接插入排序  
            for ( int j = i; j >= gap; j -= gap )  
                if ( temp < Vector[j-gap] )  
                    Vector[j] = Vector[j-gap];  
                else break;  
            Vector[j] = temp; }  
        gap = ( int ) ( gap / 2 ); }  
}
```

- gap 的取法有多种。最初Shell提出取 $gap = \lfloor n/2 \rfloor$, $gap = \lfloor gap/2 \rfloor$, 直到 $gap = 1$ 。Knuth提出取 $gap = \lfloor gap/3 \rfloor + 1$ 。还有人提出都取奇数为好, 也有人提出各 gap 互质为好。
- 对特定的待排序对象序列, 可以准确地估算排序码的比较次数和对象移动次数。
- 想要弄清排序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利用大量实验统计资料得出: 当 n 很大时, 排序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。



冒泡排序(bubble sort)

- 基本思想:依次比较**相邻**的两个元素的顺序，如果顺序不对，则将两者交换，重复这样的操作直到整个序列被排好序。
- 实际是通过比较与交换使得待排序列一个最值元素“上浮”到序列一端，然后缩小排序范围

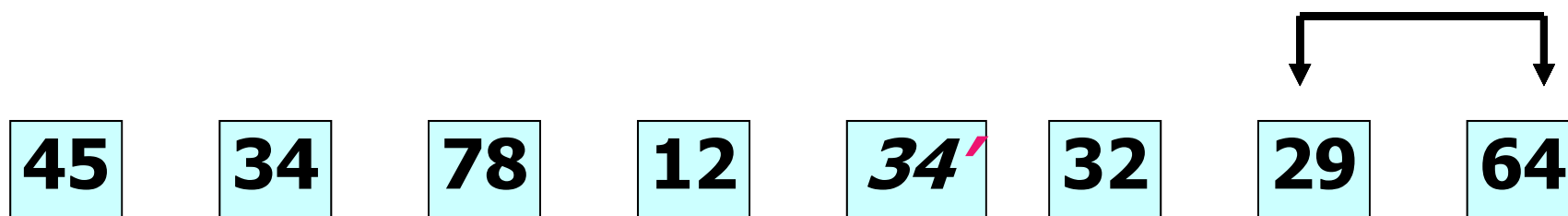
冒泡排序步骤

- 假设待排序的序列为 $a_0, a_1, a_2, \dots, a_{n-1}$
- 起始时排序范围是从 a_0 至 a_{n-1} ,
- 自右向左对相邻两结点进行比较, 让较大向右移, 让较小向左移。当比较完当前排序范围后, 键值最小的元素被移动到序列的 a_0 位置, 故 a_0 无需再参加下一次比较
- 下一次比较的范围为 a_1 至 a_{n-1} 。

冒泡排序过程

D	A	T	A	S	T	R	U	C	T	U	R	E
A	D	A	T	C	S	T	R	U	E	T	U	R
A	A	D	C	T	E	S	T	R	U	R	T	U
A	A	C	D	E	T	R	S	T	R	U	T	U
A	A	C	D	E	R	T	R	S	T	T	U	U
A	A	C	D	E	R	R	T	S	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U
A	A	C	D	E	R	R	S	T	T	T	U	U

冒泡排序动画



程序9-2 冒泡排序方法

```
template <class Item>
```

```
void BubbleSort(Item a[], int l, int r)
```

```
{
```

```
// 1. 比较相邻的元素。如果第一个比第二个大，就交换它们。
```

```
// 2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最  
// 后一对这时，最后的元素应该会是最大的数。
```

```
// 3. 针对所有的元素重复以上的步骤，除了最后一个。
```

```
// 4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何  
// 一对数字需要比较。
```

```
    for (int i=l; i<r; ++i)           // 进行r-l趟过程
```

```
        for (int j=i; j<r-1; ++j)    // 从左至右比较相邻记录
```

```
            if (a[j]>a[j+1])           // 若a[j]>a[j+1]
```

```
                swap(a[j],a[j+1]);    // 交换a[j]和a[j+1]
```

```
}
```

冒泡排序性能分析

■ 时间代价:

- 最佳情况（排序）： $\frac{n(n-1)}{2}$ 次比较，0次移动， **$O(n^2)$** (可通过改进达到 $O(n)$)
- 最差情况（逆序）： **$O(n^2)$**

- 比较与移动次数均为：
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- 平均情况： **$O(n^2)$**
- 移动与比较次数都很多，速度很慢

■ 空间代价： **$O(1)$**

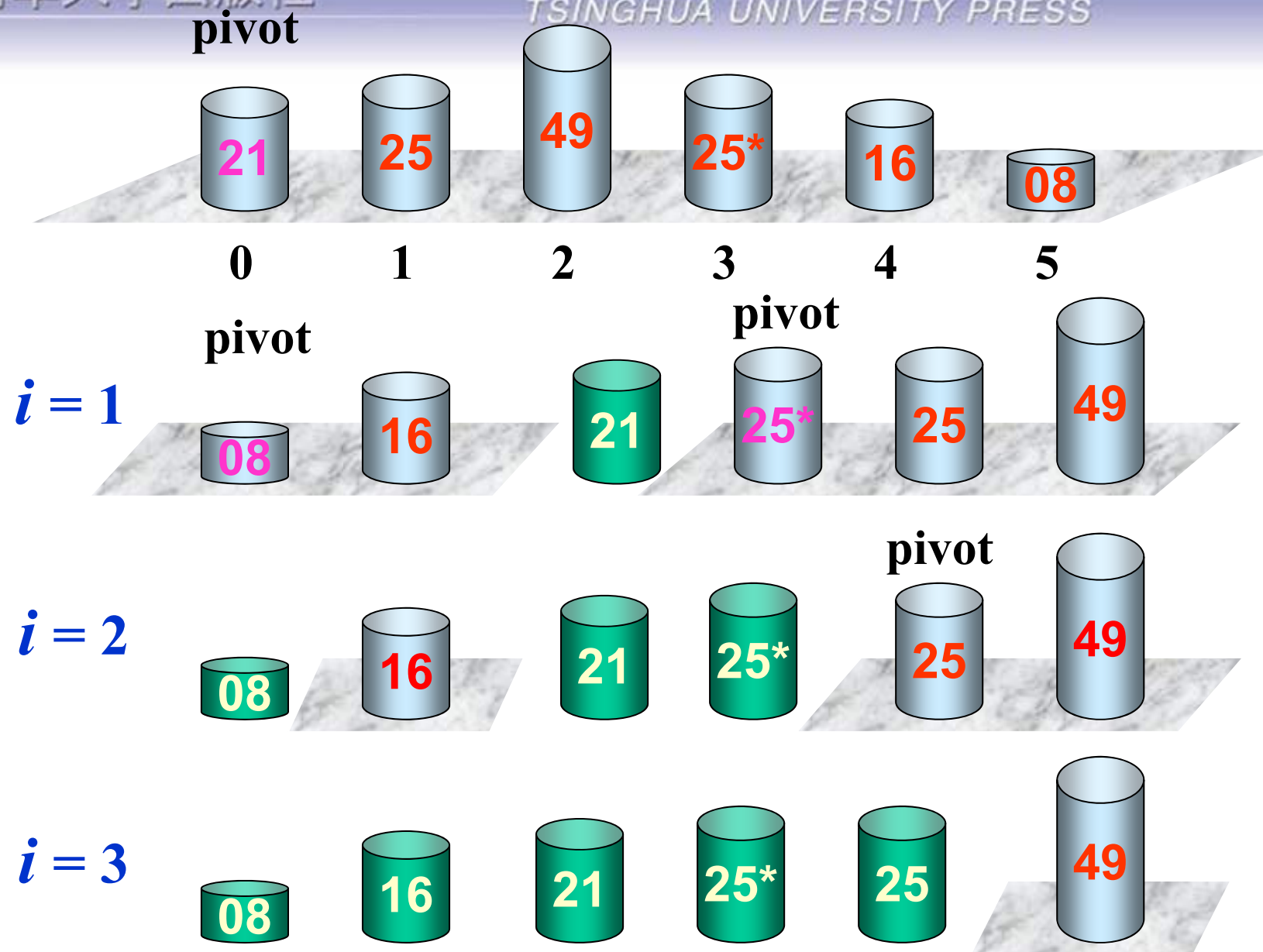
■ 稳定

9.3 快速排序 (Quick Sort)

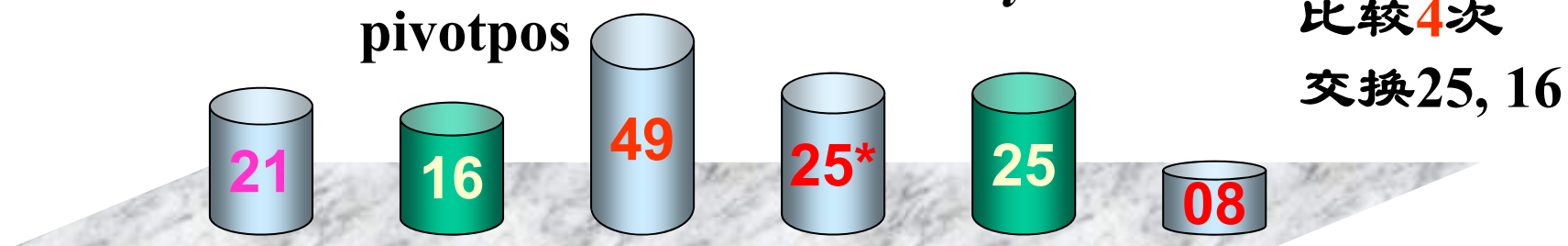
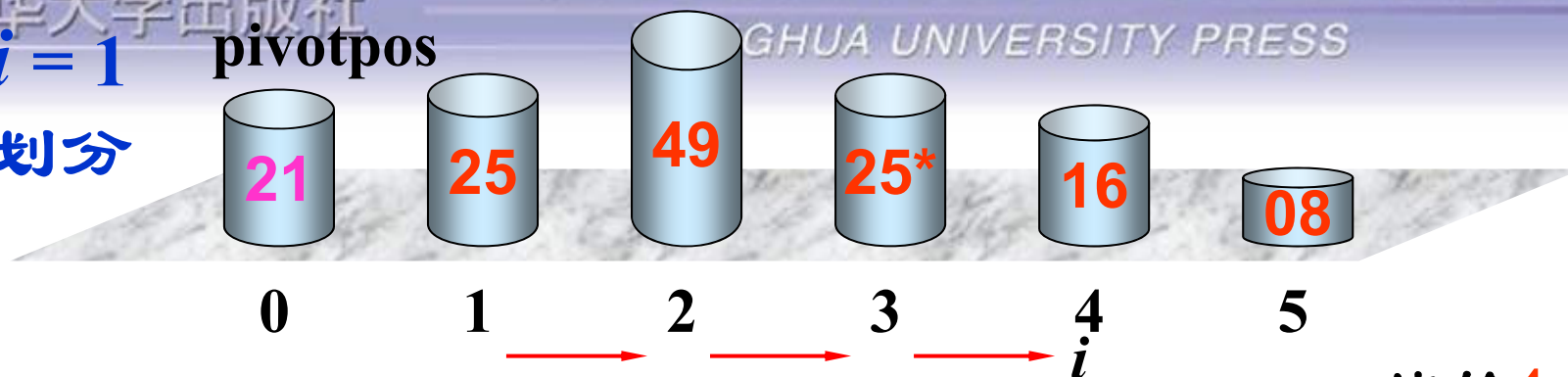
- **基本思想**：任取待排序对象序列中的某个对象（例如，取第一个对象）作为基准，按照该对象的排序码大小，将整个对象序列划分为左右两个子序列：
 - ◆ 左侧子序列中所有对象的排序码都小于或等于基准对象的排序码。
 - ◆ 右侧子序列中所有对象的排序码都大于基准对象的排序码。
- 基准对象则排在这两个子序列中间（这也是该对象最终应安放的位置）。
- 然后，分别对这两个子序列重复施行上述方法，直到所有的对象都排在相应位置上为止。

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RightList;  
        QuickSort ( LeftList );  
        QuickSort ( RightList );  
        将两个子序列LeftList和RightList合并  
        为一个序列List;  
    }  
}
```



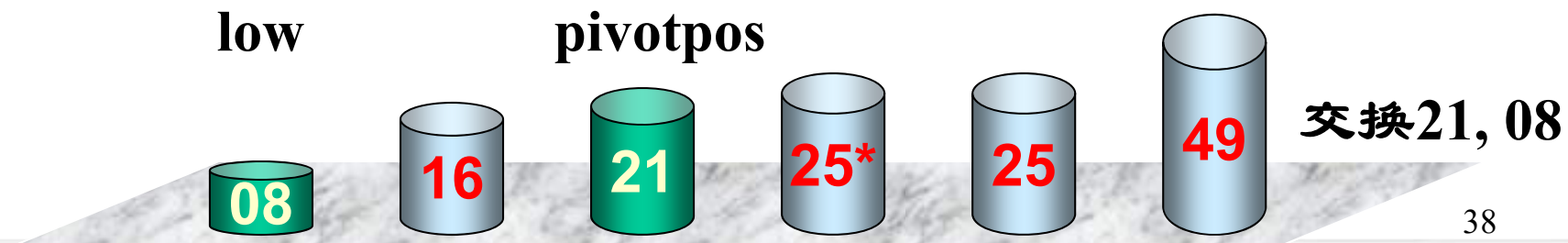
$i = 1$
划分



比较4次
交换25, 16



比较1次
交换49, 08



交换21, 08

一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'

	49	38	65	97	76	13	27	49'
--	----	----	----	----	----	----	----	-----

low ↑

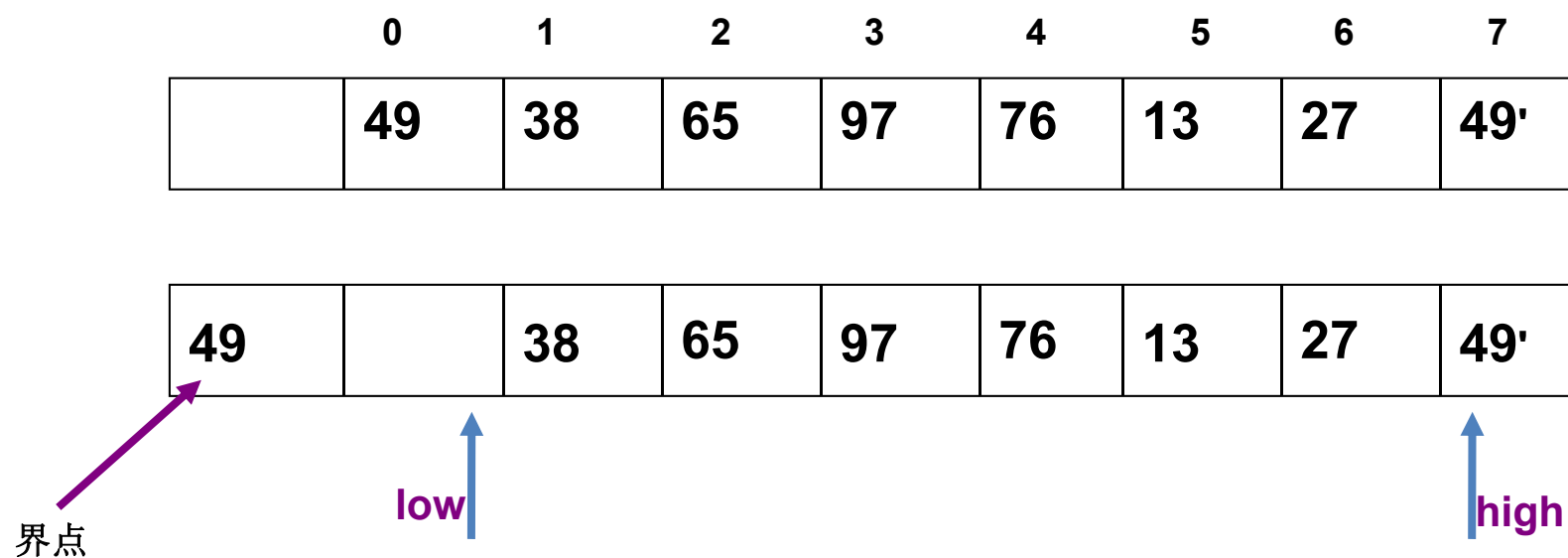
↑ high

一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：

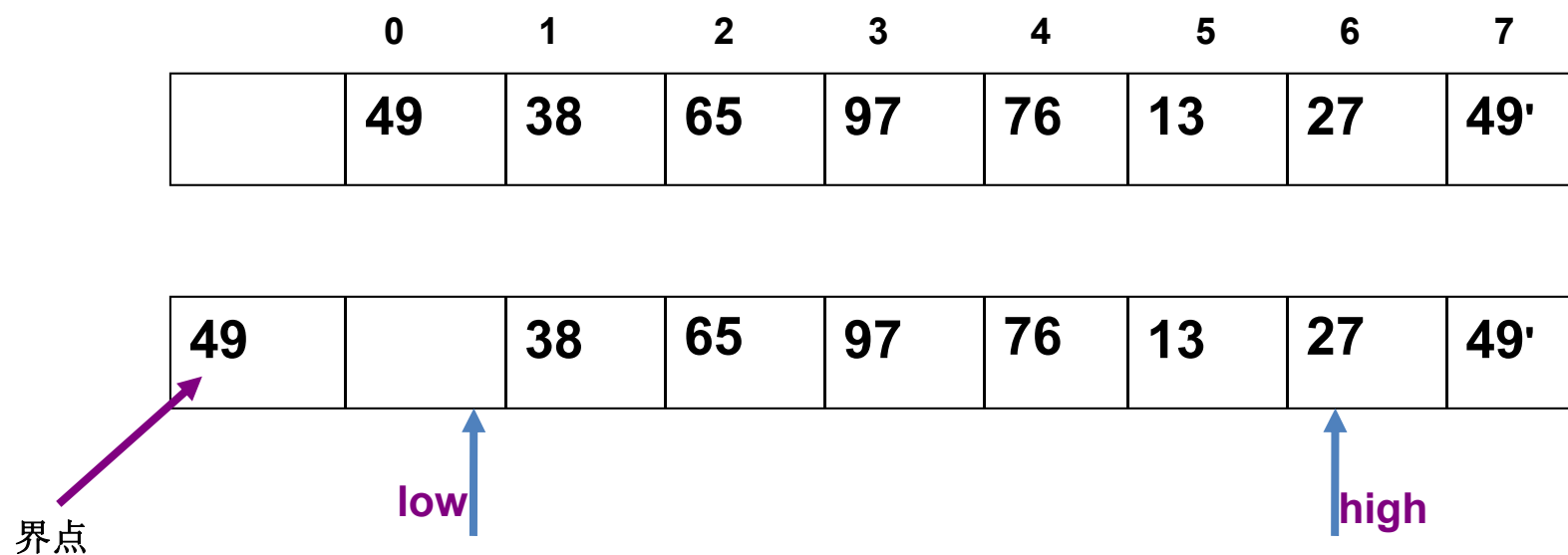


一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区首元素，**high**指向待划分区尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**；

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：



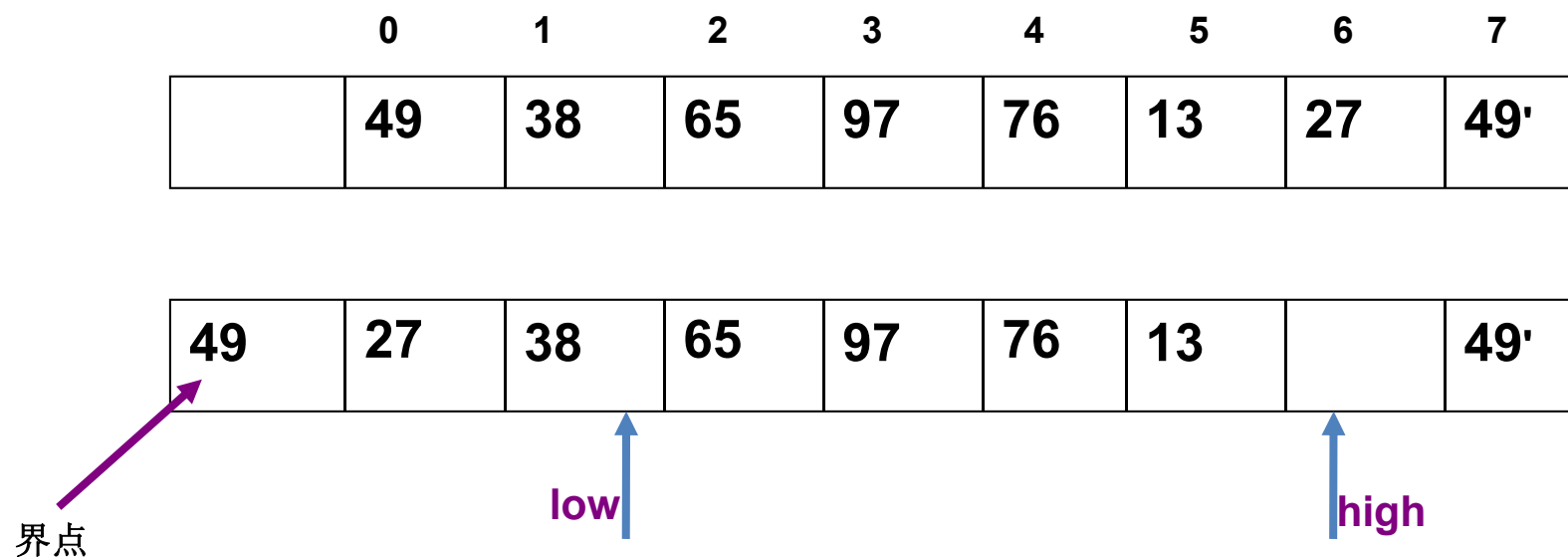
一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**;

4. **R[low]=R[high], low++;**

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：



一次划分的具体过程示例

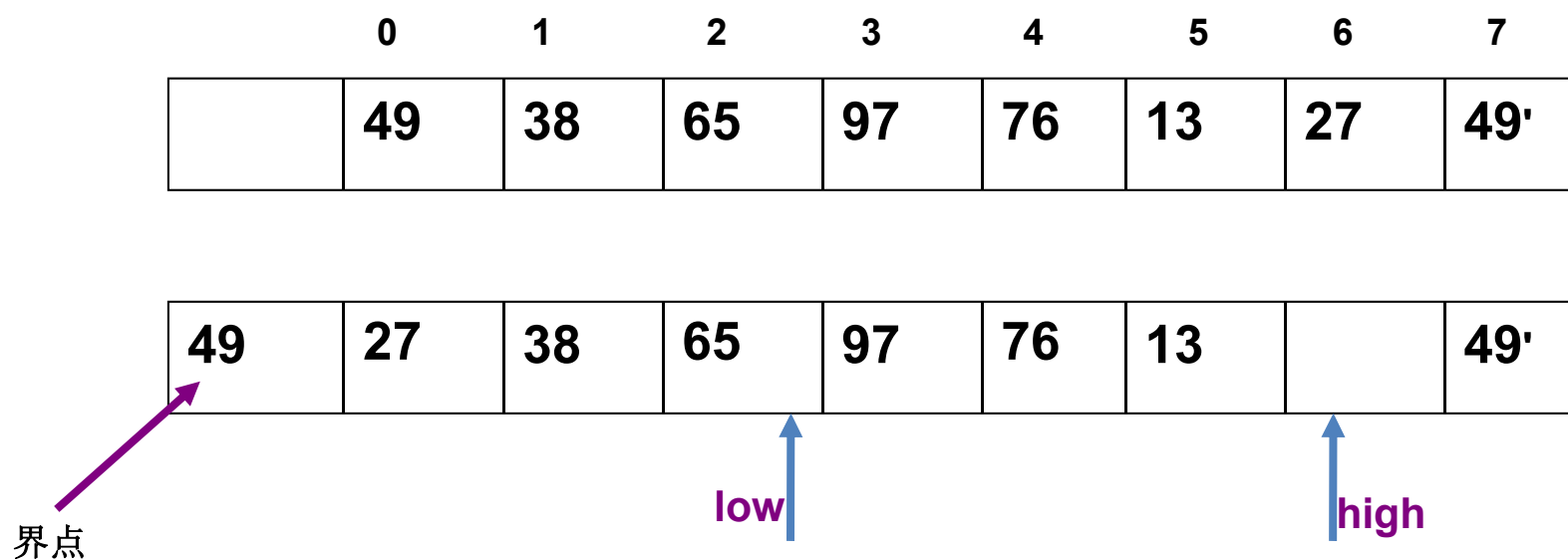
一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**；

4. **R[low]=R[high], low++**;

5. **low**从前往后移动直到**R[low].key>=t.key**;

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：



一次划分的具体过程示例

一次划分的具体过程为：

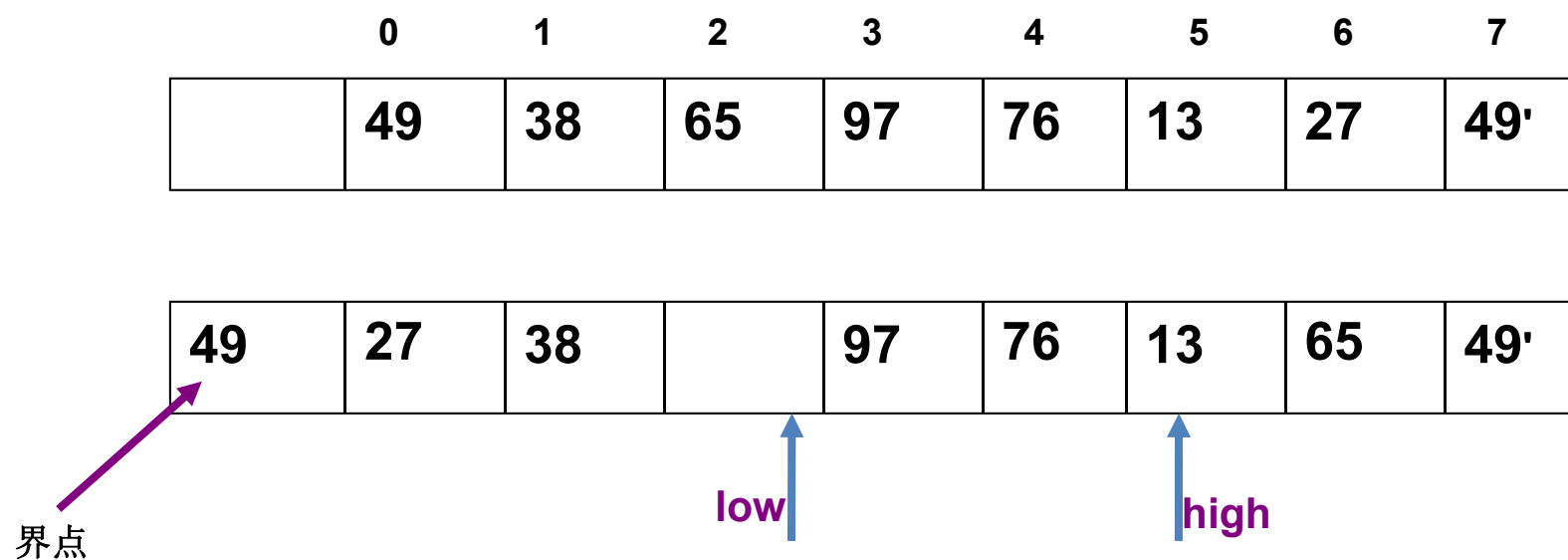
1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**；

4. **R[low]=R[high], low++**；

5. **low**从前往后移动直到**R[low].key>=t.key**；

6. **R[high]=R[low], high--**；

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：



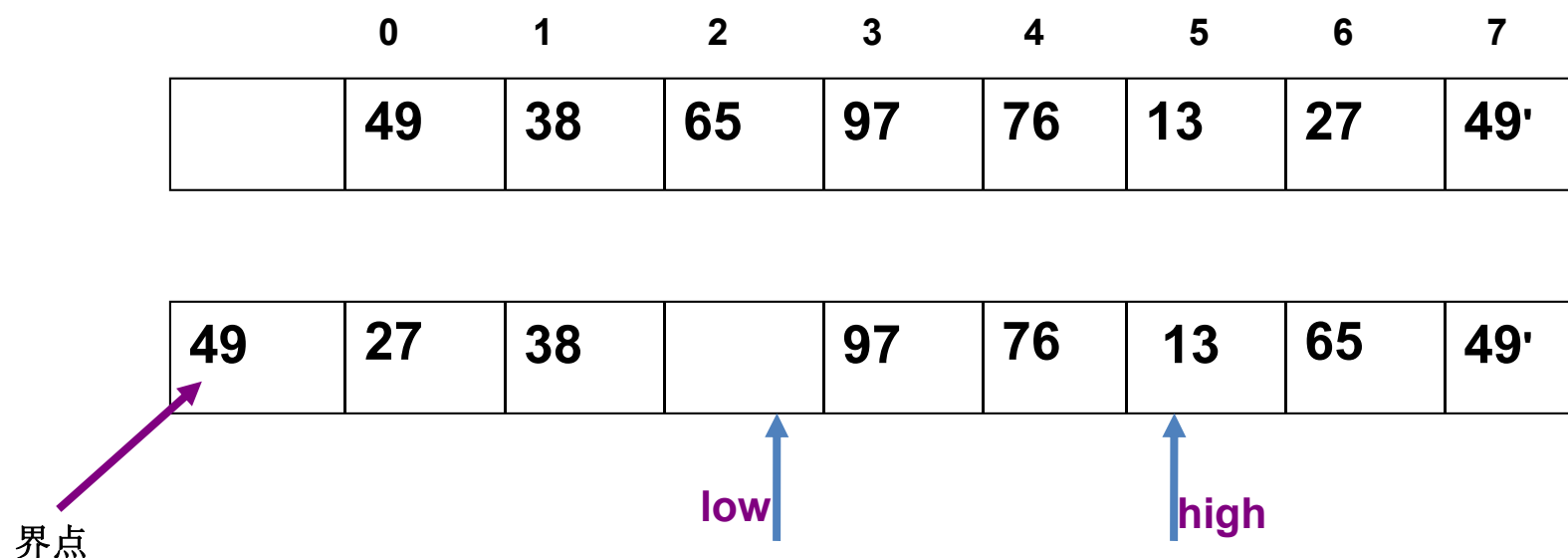
一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**；

4. **R[low]=R[high], low++**；
5. **low**从前往后移动直到**R[low].key>=t.key**；
6. **R[high]=R[low], high--**；
7. **goto 3**；

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：

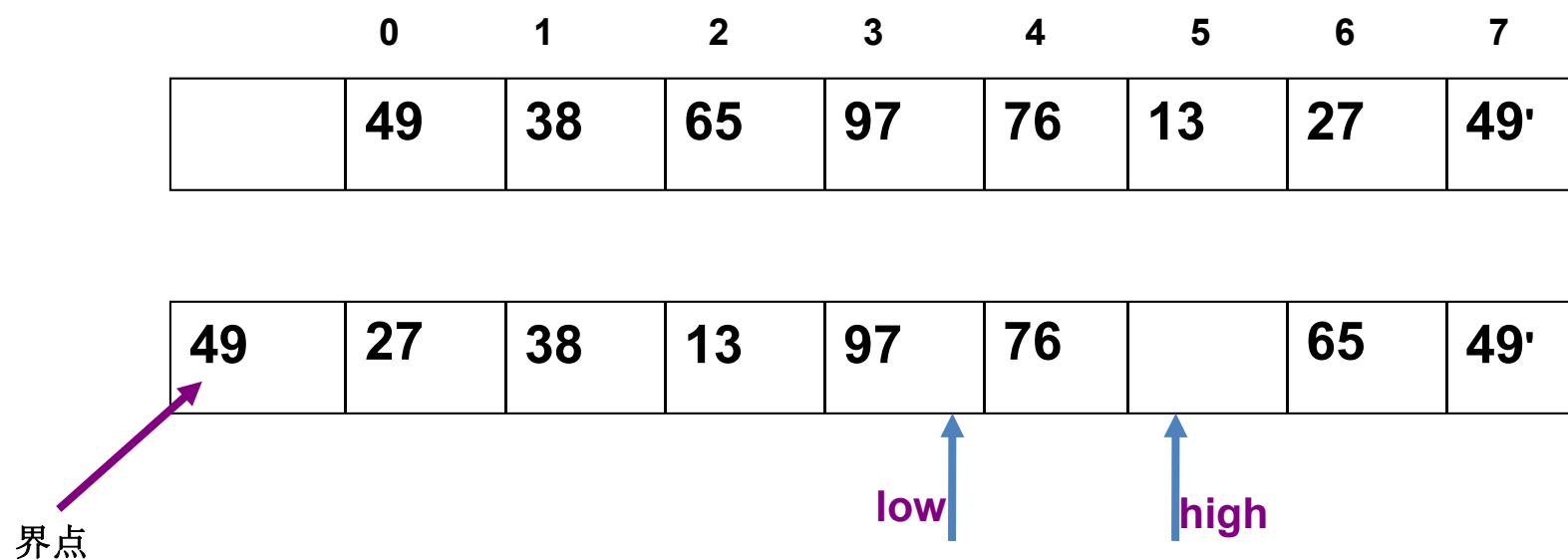


一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**;
4. **R[low]=R[high], low++**;
5. **low**从前往后移动直到**R[low].key>=t.key**;
6. **R[high]=R[low], high--**;
7. **goto 3**;

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：



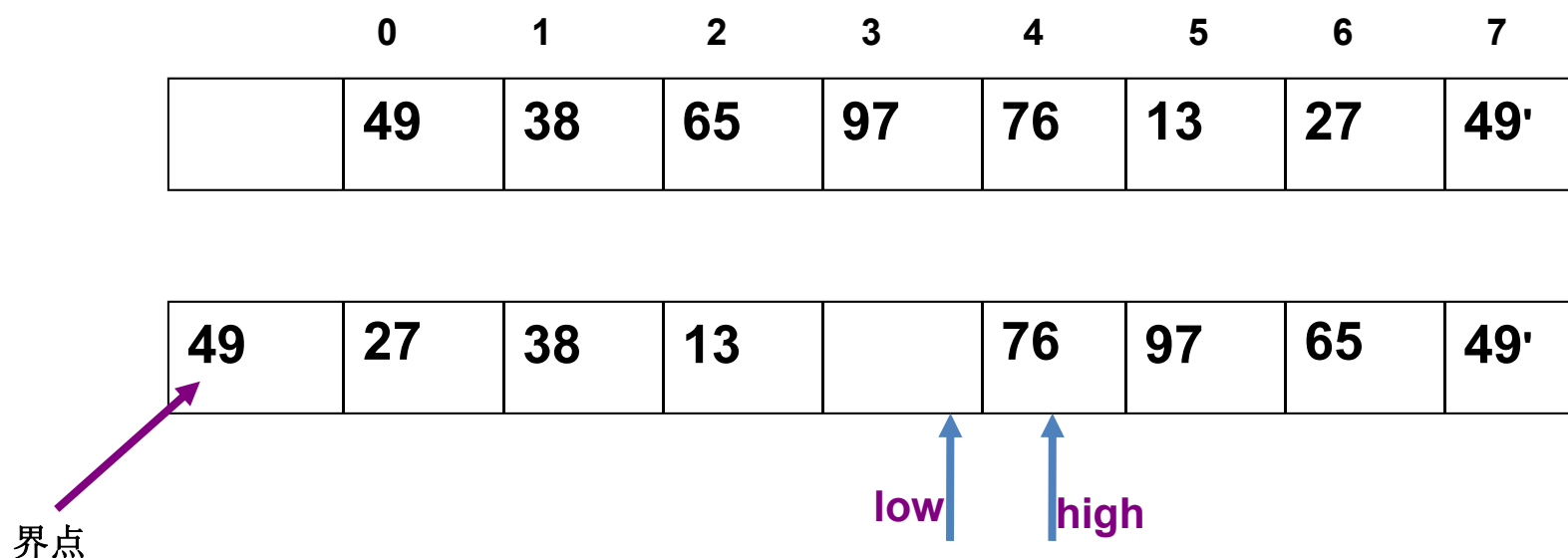
一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=R[low]** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**；

4. **R[low]=R[high], low++**；
5. **low**从前往后移动直到**R[low].key>=t.key**；
6. **R[high]=R[low], high--**；
7. **goto 3**；

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：




一次划分的具体过程示例

一次划分的具体过程为：

1. **low**指向待划分区域首元素，**high**指向待划分区域尾元素；
2. **t=t** (为了减少数据的移动, 将作为标准的元素暂存到**t**中, 最后再放入最终位置)；
3. **high**从后往前移动直到**R[high].key<t.key**；
4. **R[low]=R[high], low++**；
5. **low**从前往后移动直到**R[low].key>=t.key**；
6. **R[high]=R[low], high--**；
7. **goto 3**；
8. 直到**low==high**时，**R[low]=t**(即将作为标准的元素放到其最终位置)。

如：将序列 49、38、65、97、76、13、27、49' 一次划分的过程为：

	0	1	2	3	4	5	6	7
	49	38	65	97	76	13	27	49'
	27	38	13	49	76	97	65	49'


low **high**

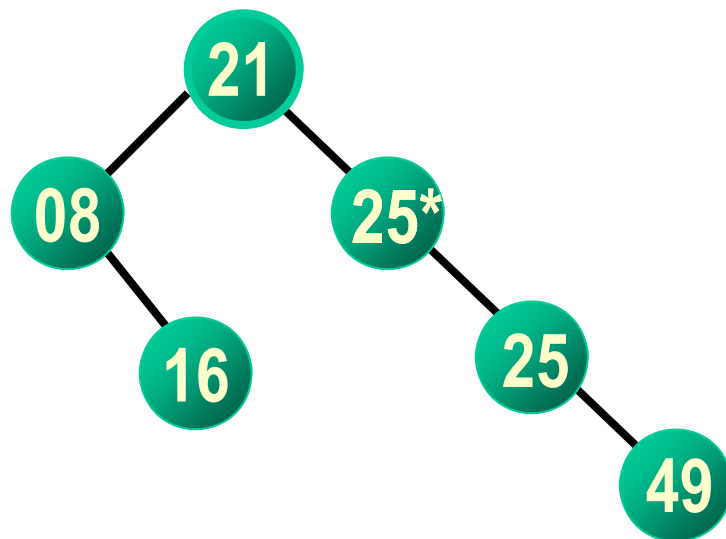
概括地说, 一次划分就是从表的两端交替地向中间进行扫描, 将小的放到左边, 大的放到右边, 作为标准的元素放到中间。

快速排序的算法

```
template <class Type> void DataList <Type> ::  
QuickSort ( const int left, const int right ) {  
//在序列left~right中递归地进行快速排序  
    if ( left < right ) {  
        int pivotpos = Partition ( left, right ); //划分  
        //对左序列同样处理  
        QuickSort ( left, pivotpos-1 );  
        //对右序列同样处理  
        QuickSort ( pivotpos+1, right );  
    }  
}
```

```
template <class Type> int DataList <Type> ::  
Partition ( const int low, const int high ) {  
    int pivotpos = low; //基准位置  
    Element <Type> pivot = Vector[low];  
    for ( int i = low+1; i <= high; i++ )  
        if ( Vector[i] < pivot ) {  
            pivotpos++;  
            if ( pivotpos != i )  
                Swap ( Vector[pivotpos], Vector[i] );  
        } //小于基准对象的交换到区间的左侧去  
    Swap ( Vector[low], 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，只要是排序码小于基准对象排序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高度。
- 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：

$$\begin{aligned} T(n) &\leq cn + 2T(n/2) \quad //c\text{是一个常数} \\ &\leq cn + 2 (cn/2 + 2T(n/4)) = 2cn + 4T(n/4) \\ &\leq 2cn + 4 (cn/4 + 2T(n/8)) = 3cn + 8T(n/8) \\ &\dots\dots\dots \\ &\leq cn\log_2 n + nT(1) = O(n\log_2 n) \end{aligned}$$

- 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高度一致，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排序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。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 $n-i$ 次排序码比较才能找到第 i 个对象的安放位置，总的排序码比较次数将达到：

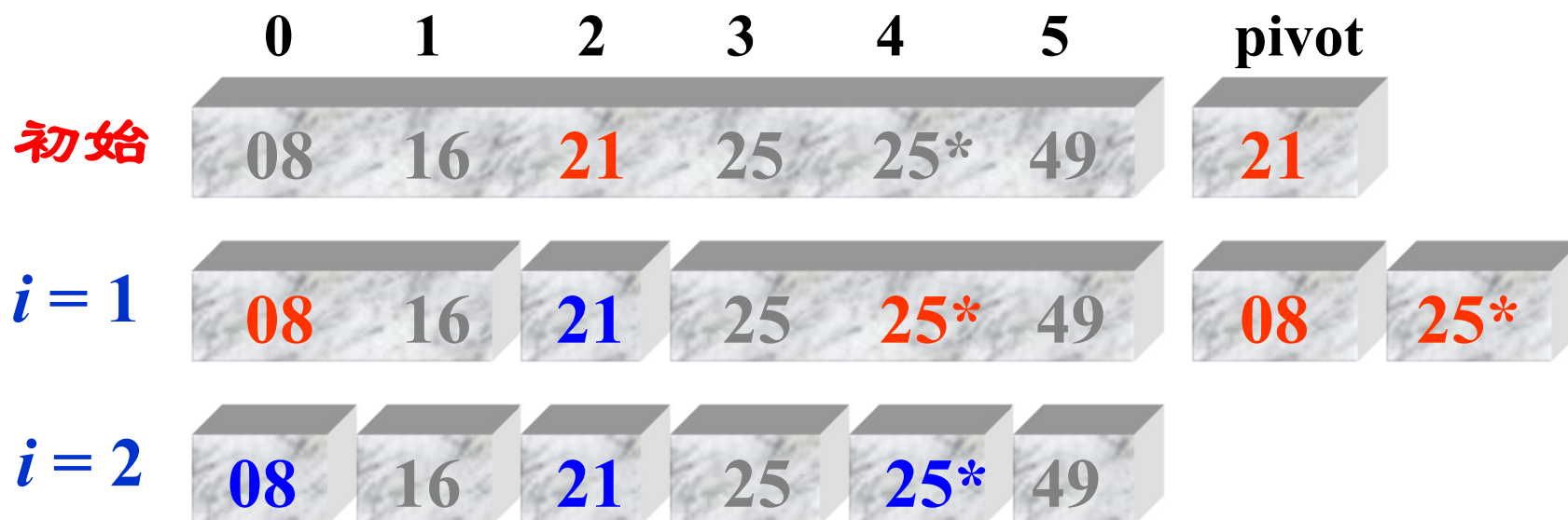
$$\sum_{i=1}^{n-1} (n-i) = \frac{1}{2} n(n-1) \approx \frac{n^2}{2}$$

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

用第一个对象作为基准对象

快速排序退化的例子

- 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（栈）将达到 $O(n)$ 。
- 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**，取其排序码居中者作为基准对象。



用居中排序码对象作为基准对象


```
Type mid ( Type a, Type b, Type c ) {  
    Type first = a, second; //first记录最小  
    if ( b < first ) { second = first; first = b; }  
    else second = b; //second记录次小  
    if ( c < first ) { second = first; first = c; }  
    else if ( c < second ) second = c;  
    return second;  
}
```

- 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。



快速排序性能分析

- 时间代价:

- 最差情况（划分基准每次均为最值）： $O(n^2)$

- 平均情况： $O(n \log n)$

- 比较次数为： $2n \ln n$

- 空间代价(递归压栈)： $O(\log n)$

- 不稳定

快速排序的一些改进策略

- 程序递归空间不足:
 - ◆ 使用栈
- 避免最坏情况
 - ◆ 划分基准的选择
- 加快短序列排序
 - ◆ 使用插入排序

改进策略——栈的使用

- 程序递归需要系统栈空间: $O(\log n)$ （最坏情况 $O(n)$ ），若排序序列过长，系统会因缺少空间。
- 改进：
 - ◆ 用栈将递归改非递归，栈存储待排序列范围
 - ◆ 优先把长序列压栈，则栈的深度最多为 $\log_2 n$

自底向上的快速排序方法

```
inline void push_interval(stack<int> &s, int a, int b)
{
    s.push(b);    s.push(a);
}
```

```
template <class Item>
```

```
void QuickSortBU(Item a[], int l, int r)
{
```

```
    stack<int> s;    int i;
```

```
    //将初始区间(l,r)压栈
```

```
    push_interval(s,l,r);
```

```
    //如果栈不空, 循环; 栈空: 说明初始区间(l,r)已经排好序, 并
        //被弹出栈外.
```

```
    while(!s.empty()){
```

```
        //获取当前栈顶存放的区间
```

```
        l=s.top();    s.pop();
```

```
        r=s.top();    s.pop();
```

```

//如果当前区间左右边界重合, 则结束当前循环(并开始下次
//循环)
if (r<=l) continue;
//调用程序9-7中的partition函数
i=partition(a,l,r);
//比较两个序列的长度, 将长的区间范围先压入栈中后, 再
//将短序列的区间压进栈
// 以保证栈的深度比较小
if (i-l > r-i){
    push_interval(s,l,i-1); push_interval(s,i+1,r);
}
else{
    push_interval(s,i+1,r); push_interval(s,l,i-1);
}
}
}

```

改进策略——划分基准的选择

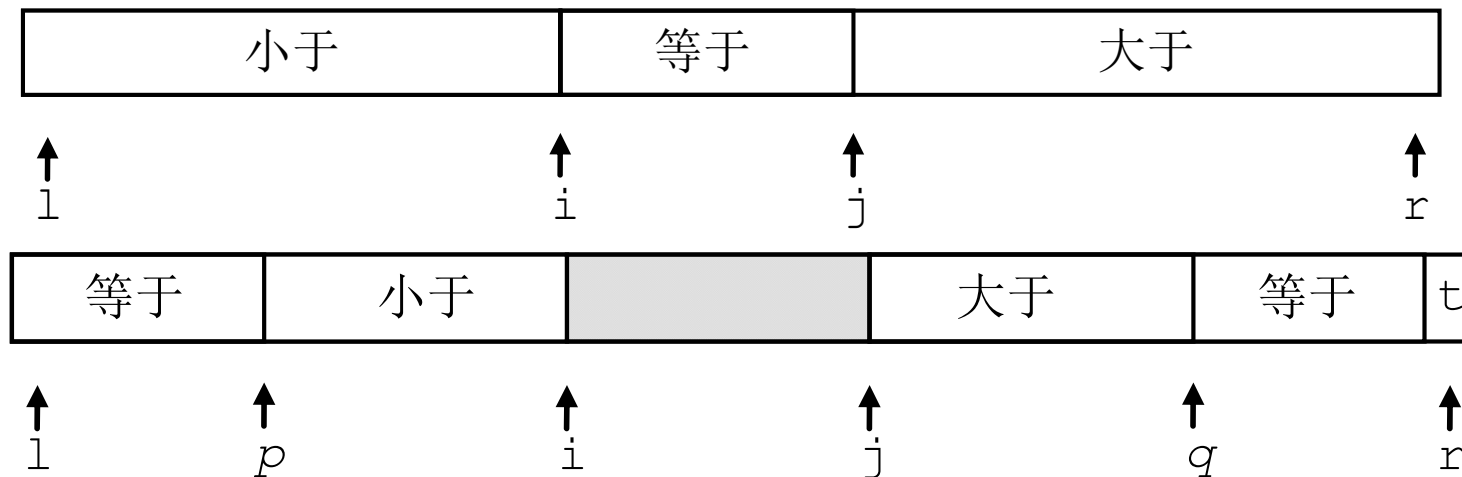
- 当划分基准不能将序列一分为二，则快速排序会退还成 $O(n^2)$ 。
- 改进：
 - ◆ 随机选择降低最坏发生的概率（最坏情况还是会发生）
 - ◆ 从序列中选择三项，然后选择三项中的中间项

改进策略——短子序列

- 对于短序列，快速排序效率不如一些简单的排序方法。而插入排序在这种情况下表现更好。
- 短子序列长度 M ：研究表明， M 在5~25时，效率比 $M=1$ 提高10%。经验表明， $M=8\sim 10$ 效率会更高

重复值

- 当存在大量重复项时，排序算法会退化。
- 改进——三路划分（小于、等于、大于）：
 - ① 扫描左子序列时，将与控制值相等的项放置在该子序列的最左端
 - ② 扫描右子序列时，将与控制值相等的项放置在该子序列的最右端
 - ③ 当*i*和*j*相遇后，循环将等于控制值的项交换到序列中央



快速排序的三路划分实现

```
template <class Item>
```

```
int operator==(const Item &A, const Item &B)
```

```
{
```

```
    return !less(A,B)&&!less(B,A);
```

```
}
```

```
template <class Item>
```

```
void QuickSort(Item a[], int l, int r)
```

```
{
```

```
    int k; Item v=a[r];
```

```
//控制值选取
```

```
    if (r<=l)return;
```

```
//递归退出条件
```

```
    int i=l-1, j=r, p=l-1, q=r;
```

```
    for(;;){
```

```
//从序列的最左向右扫描数据，直到找到一个比控制值大的项a_i
```

```
        while (a[++i]<v);
```

```
//从序列的最右向左扫描，直到找到一个比控制值小的项a_j
```

```
        while (v<a[--j]) if (j==l) break;
```

```

//此时,  $a_j < t < a_i$ , 交换 $a_i$ 和 $a_j$ , 继续上述的扫描和交换过程, 直到 $i \geq j$ 
    if ( $i \geq j$ ) break;
    swap( $a[i], a[j]$ );
    //将与控制值相等的记录交换到左侧队列的最左端,
    //同时调整p
    if ( $a[i] == v$ ) { $p++$ ; swap( $a[p], a[i]$ );}
    //将与控制值相等的记录交换到右侧队列的最右端,
    //同时调整q
    if ( $v == a[j]$ ) { $q--$ ; swap( $a[q], a[j]$ );}
}
swap( $a[i], a[r]$ );  $j = i - 1$ ;  $i = i + 1$ ;
//重复值交换: 将与控制值相同的记录交换到序列正中
for ( $k = l$ ;  $k \leq p$ ;  $k++$ ,  $j--$ ) swap( $a[k], a[j]$ );
for ( $k = r - 1$ ;  $k \geq q$ ;  $k--$ ,  $i++$ ) swap( $a[k], a[i]$ );
// 三路划分完成, 递归对左子序列和右子序列做快速排序
QuickSort( $a, l, j$ ); // 左子序列三路划分快速排序
QuickSort( $a, i, r$ ); // 右子序列三路划分快速排序
}

```

9.4 选择排序

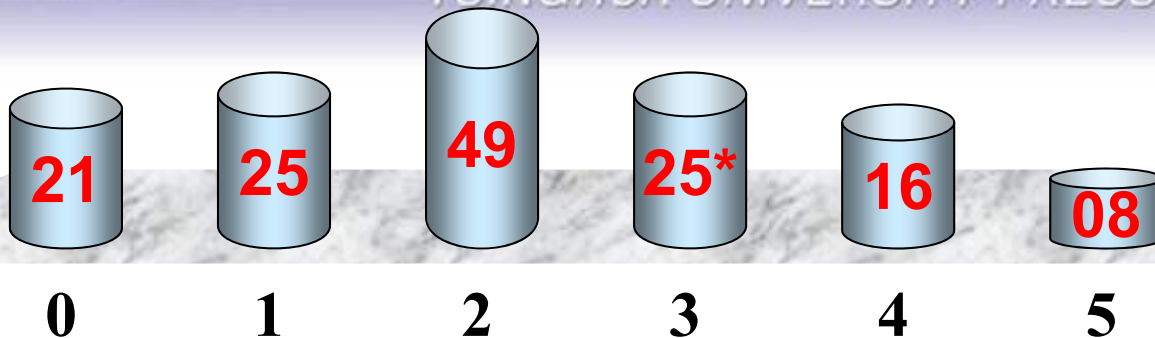
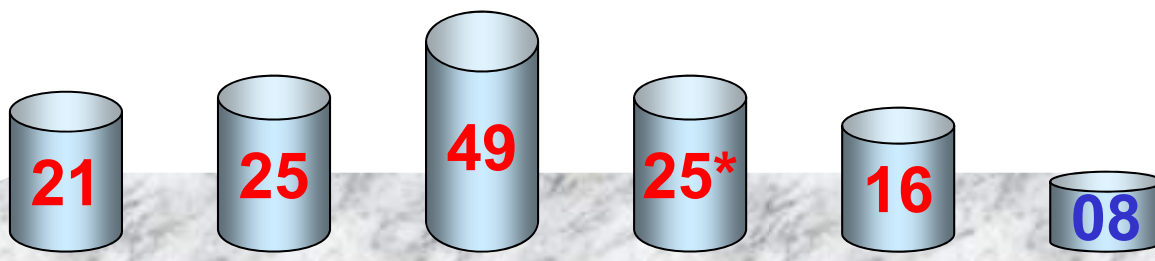
基本思想：每一趟（例如，第 i 趟， $i = 0, 1, \dots, n-2$ ）在后面 $n-i$ 个待排序对象中选出排序码最小的对象，作为有序对象序列的第 i 个对象。待到第 $n-2$ 趟作完，待排序对象只剩下1个，就不用再选了。

- 直接选择排序
- 锦标赛排序
- 堆排序

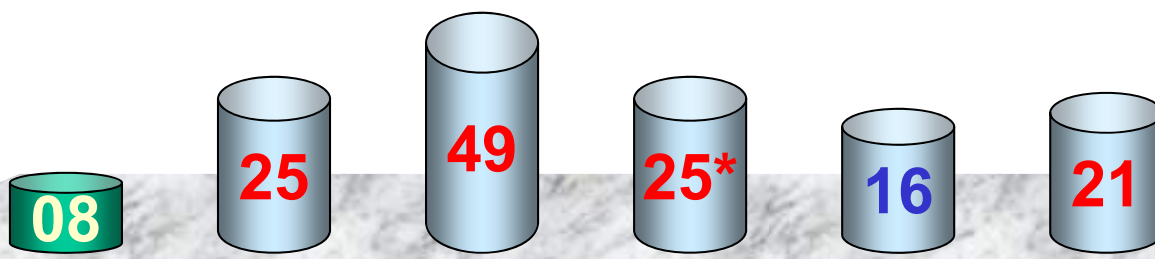
直接选择排序 (Select Sort)

- 直接选择排序是一种简单的排序方法，它的基本步骤是：
 - ① 在一组对象 $V[i] \sim V[n-1]$ 中选择具有最小排序码的对象；
 - ② 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；
 - ③ 在这组对象中剔除这个具有最小排序码的对象。在剩下的对象 $V[i+1] \sim V[n-1]$ 中重复执行第①、②步，直到剩余对象只有一个为止。

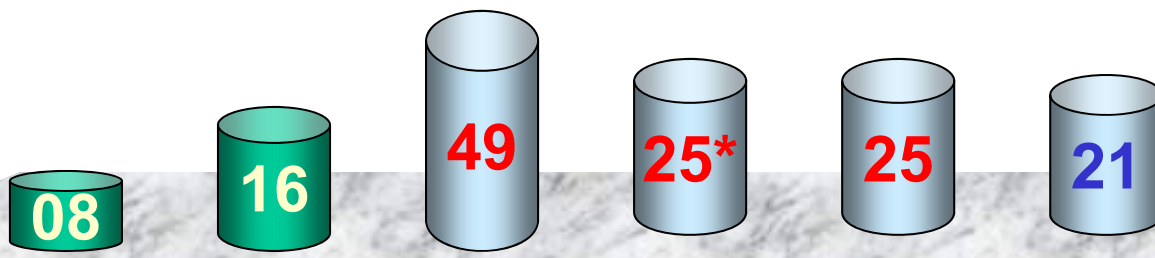
初始

 $i = 0$ 

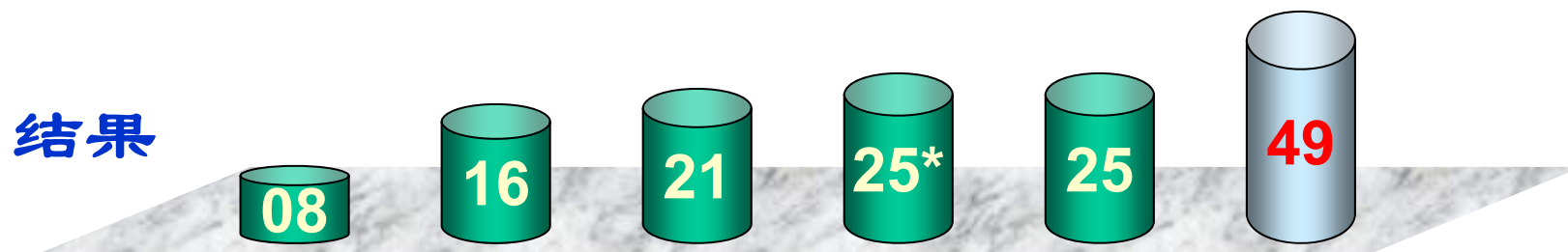
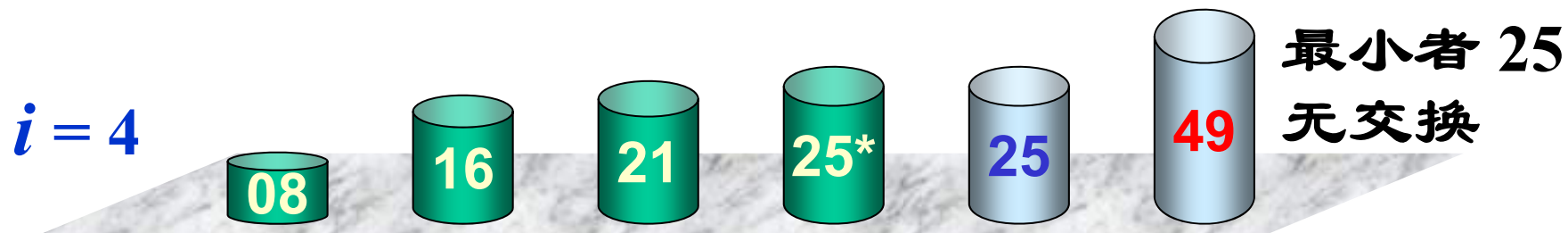
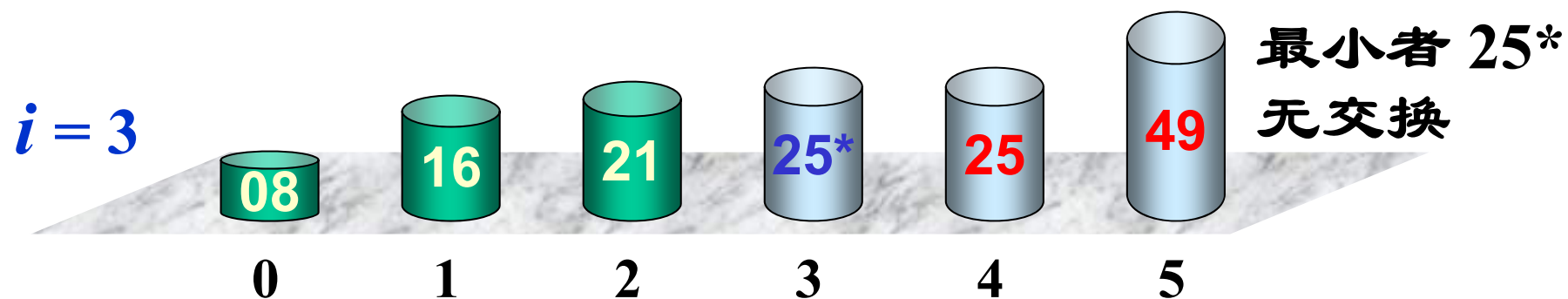
最小者 08
交换 21, 08

 $i = 1$ 

最小者 16
交换 25, 16

 $i = 2$ 

最小者 21
交换 49, 21

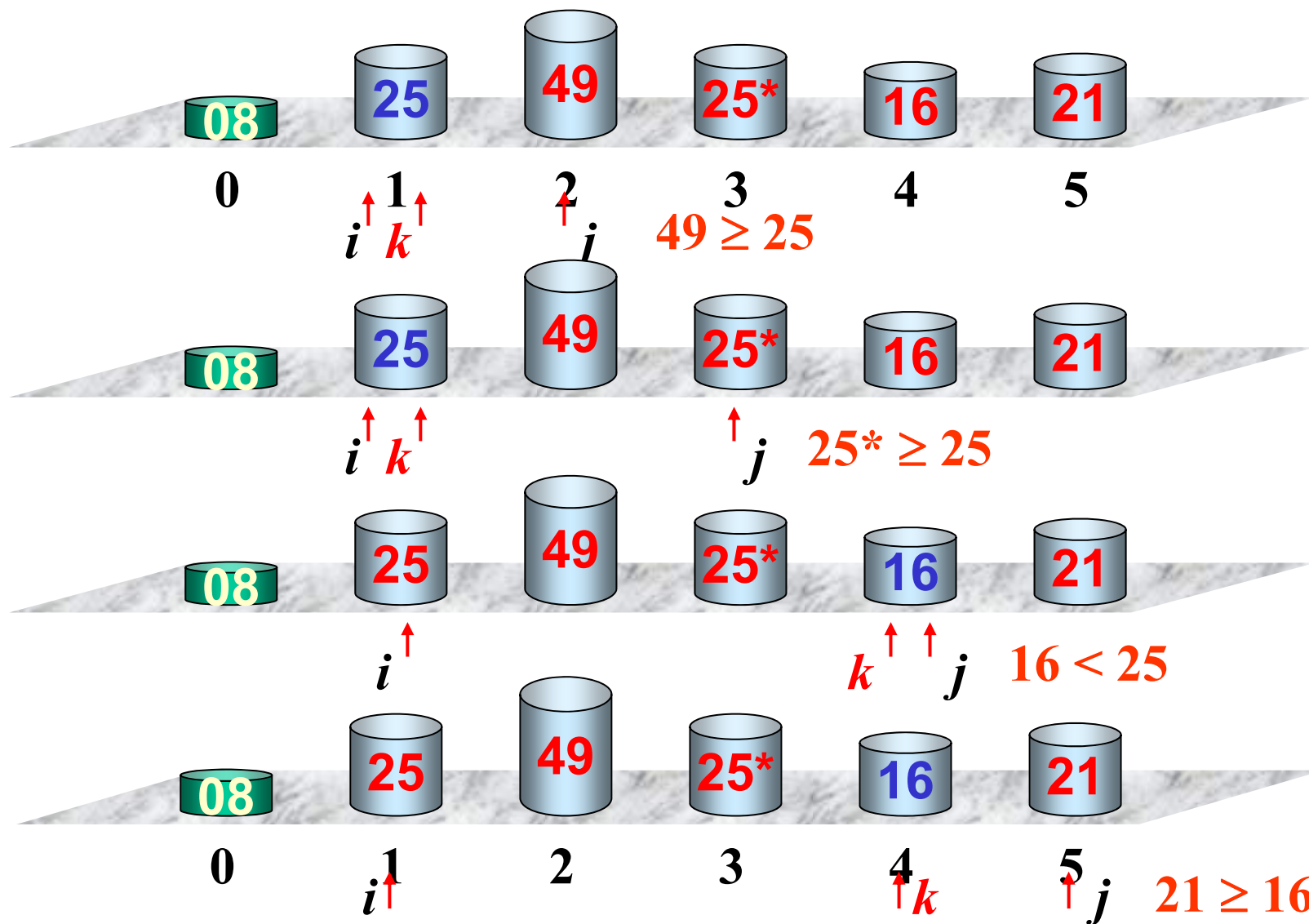


各趟排序后的结果

直接选择排序的算法

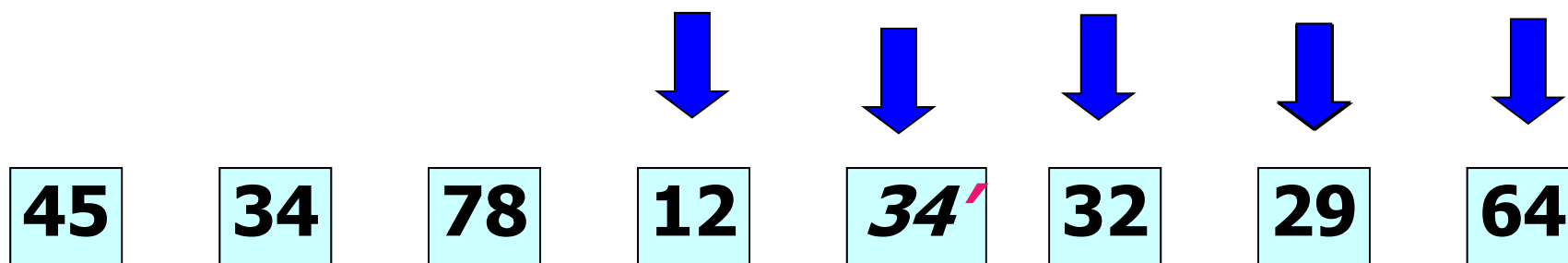
```
template <class Type> void DataList <Type> ::  
SelectSort ( ) {  
    for ( int i = 0; i < CurrentSize-1; i++ ) {  
        int k = i; //选择具有最小排序码的对象  
        for ( int j = i+1; j < CurrentSize; j++ )  
            if ( Vector[j] < Vector[k] )  
                k = j; //当前具最小排序码的对象  
        if ( k != i ) //对换到第 i 个位置  
            Swap ( Vector[i], Vector[k] );  
    }  
}
```


$i=1$ 时选择排序的过程



k 指示当前序列中最小者

选择排序动画



- 直接选择排序的排序码比较次数 KCN 与对象的初始排列无关。设整个待排序对象序列有 n 个对象，则第 i 趟选择具有最小排序码对象所需的比较次数总是 $n-i-1$ 次。总的排序码比较次数为：

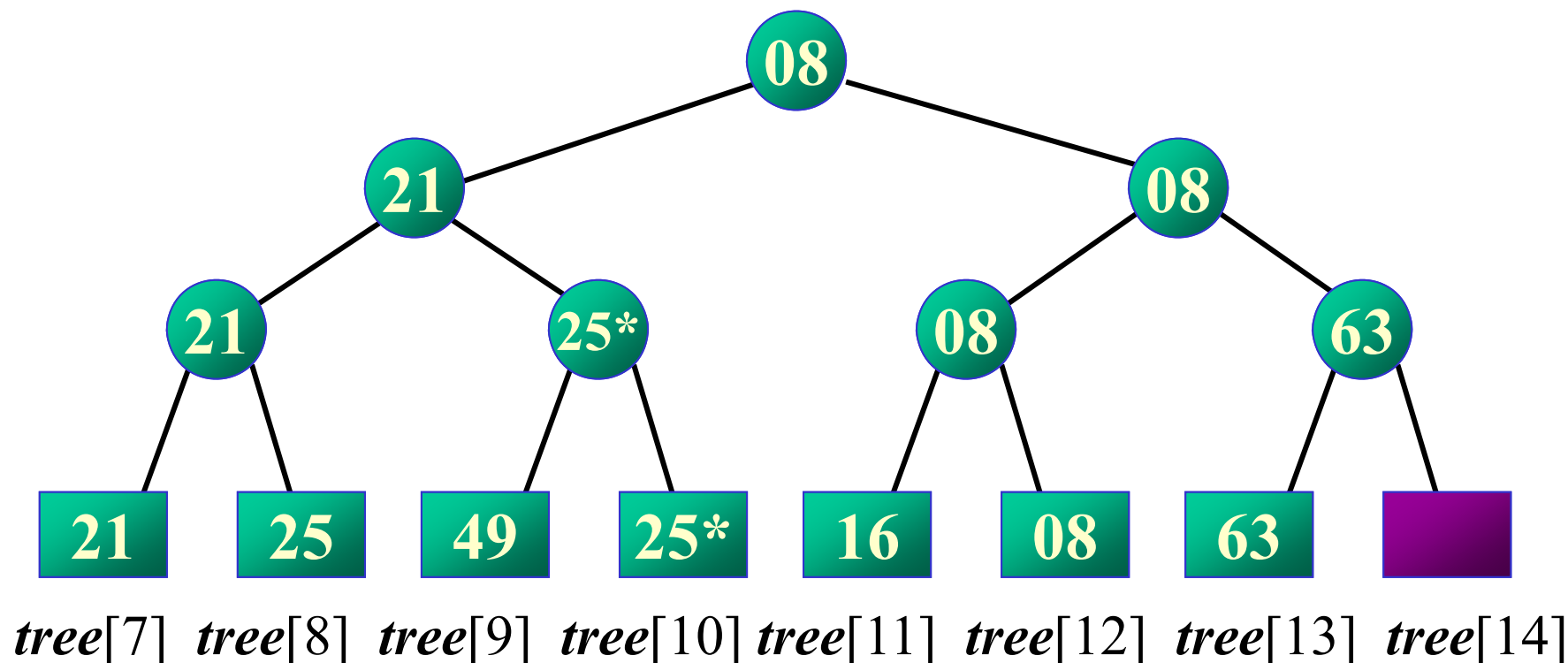
$$KCN = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2}$$

- 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排序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 个对象的排序码，进行两两比较，得到 $\lceil n/2 \rceil$ 个比较的优胜者（排序码小者），作为第一步比较的结果保留下来。然后，对这 $\lceil n/2 \rceil$ 个对象再进行排序码的两两比较，...，如此重复，直到选出一个排序码最小的对象为止。
- 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排序码。

Winner



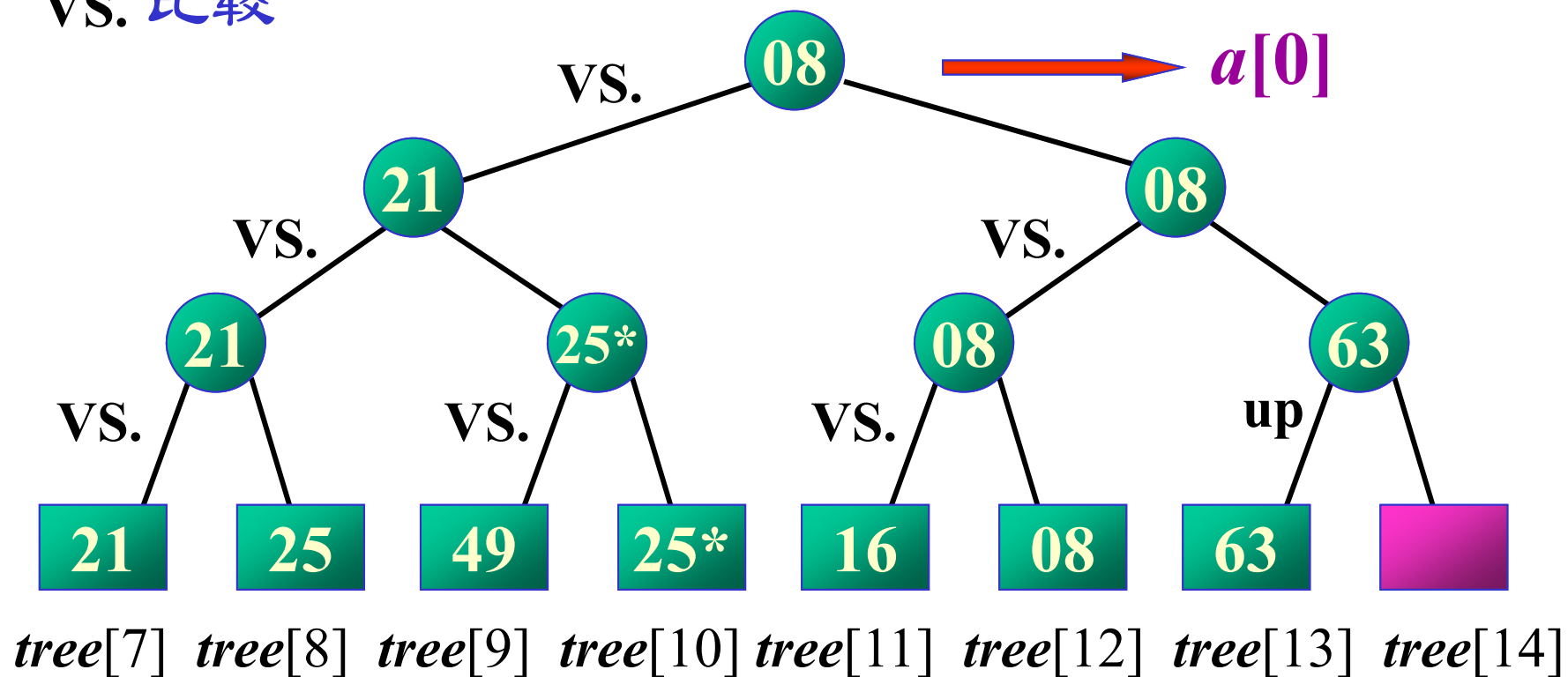
- 如果 n 不是2的 k 次幂，则让叶结点数补足到满足 $2^{k-1} < n \leq 2^k$ 的 2^k 个。叶结点上面一层的非叶结点是叶结点排序码两两比较的结果，最顶层是根。

胜者树的概念

- 每次两两比较的结果是把排序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排序码 `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排序码的对象。

up 对手不参选
VS. 比较

Winner (胜者)

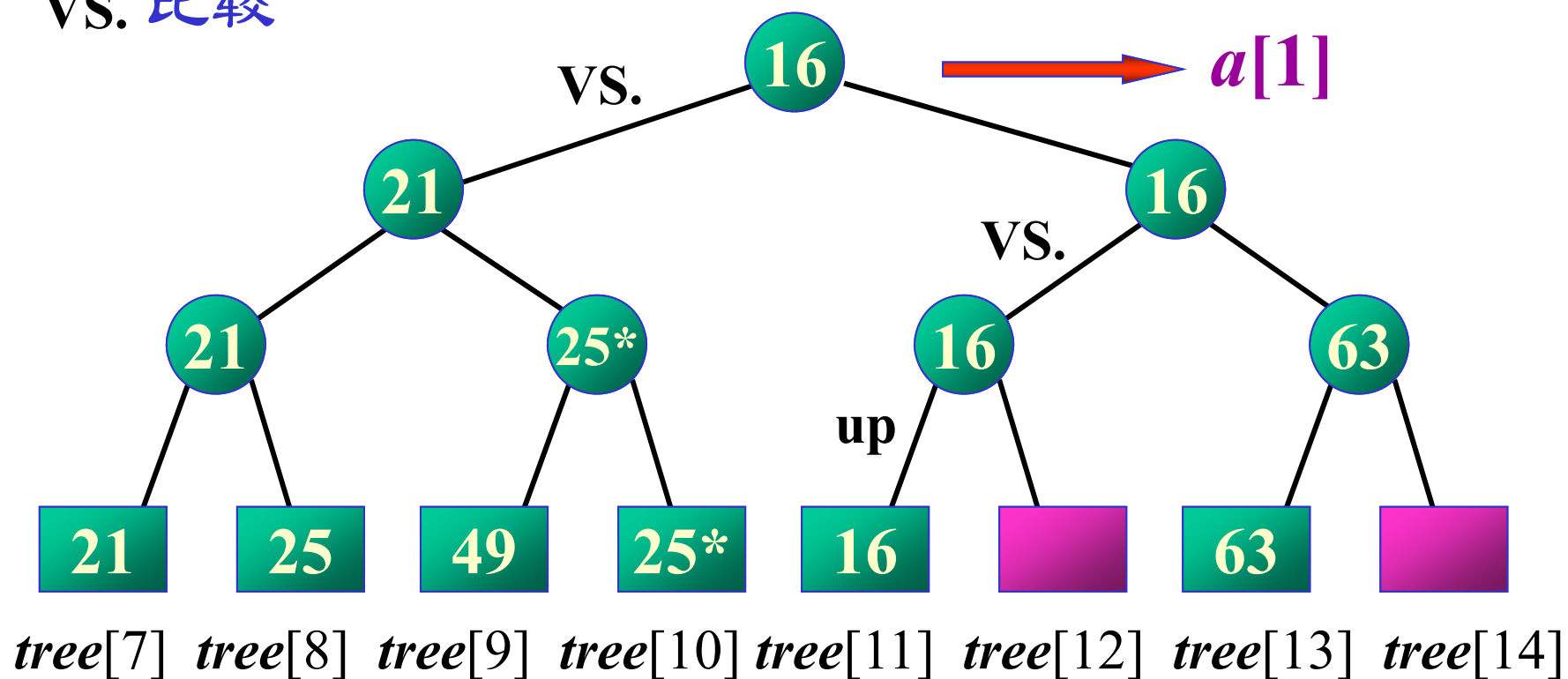


形成初始胜者树 (最小排序码上升到根)

排序码比较次数 : 6

up 对手不参选
VS. 比较

Winner (胜者)

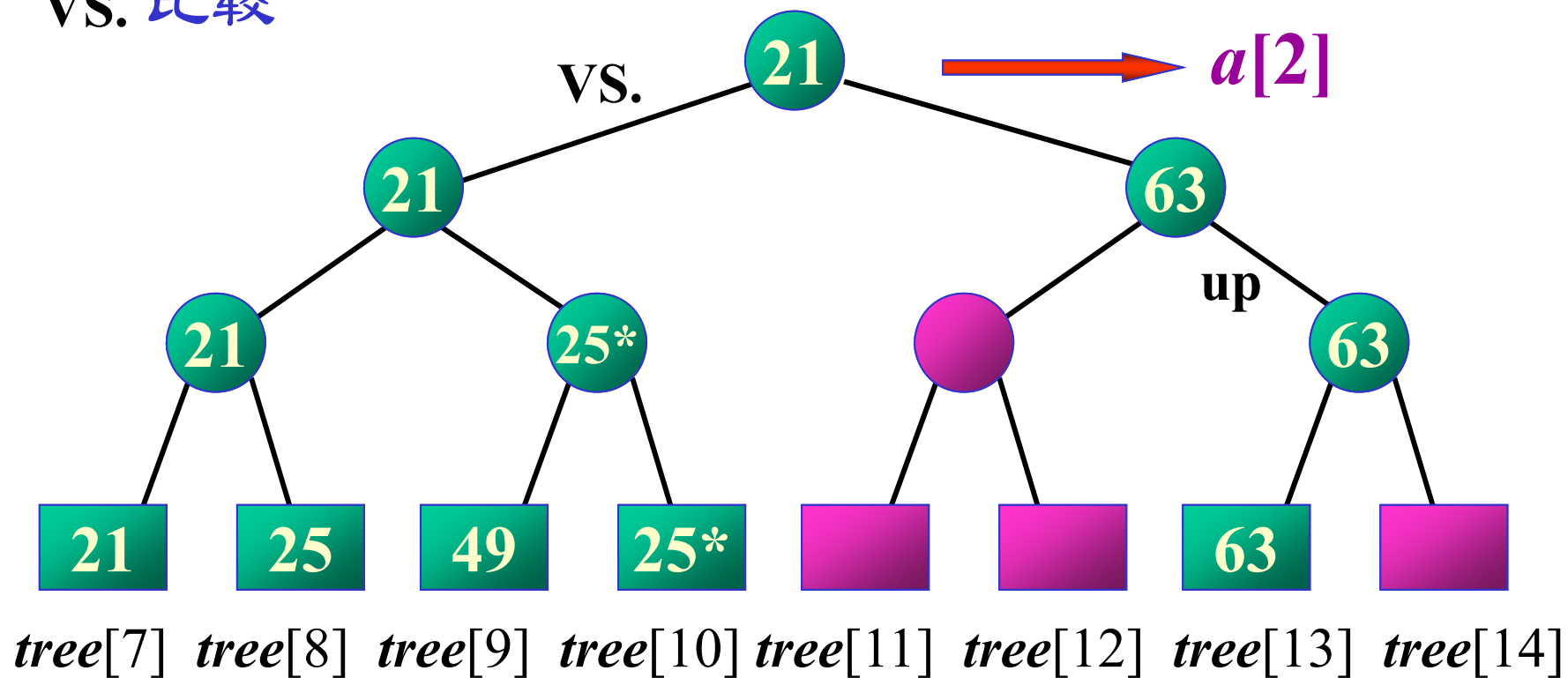


输出冠军并调整胜者树后树的状态

排序码比较次数：2

up 对手不参选
VS. 比较

Winner (胜者)

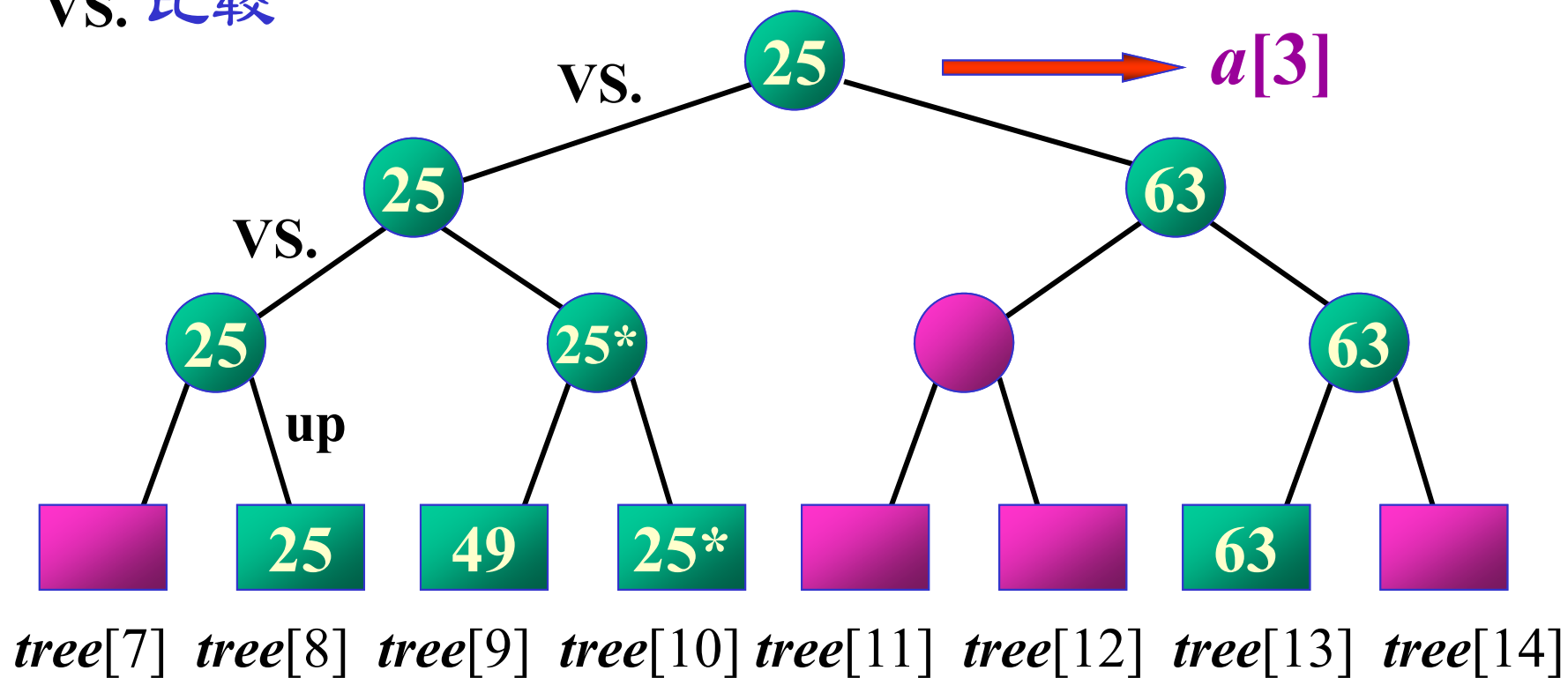


输出亚军并调整胜者树后树的状态

排序码比较次数：1

up 对手不参选
VS. 比较

Winner (胜者)

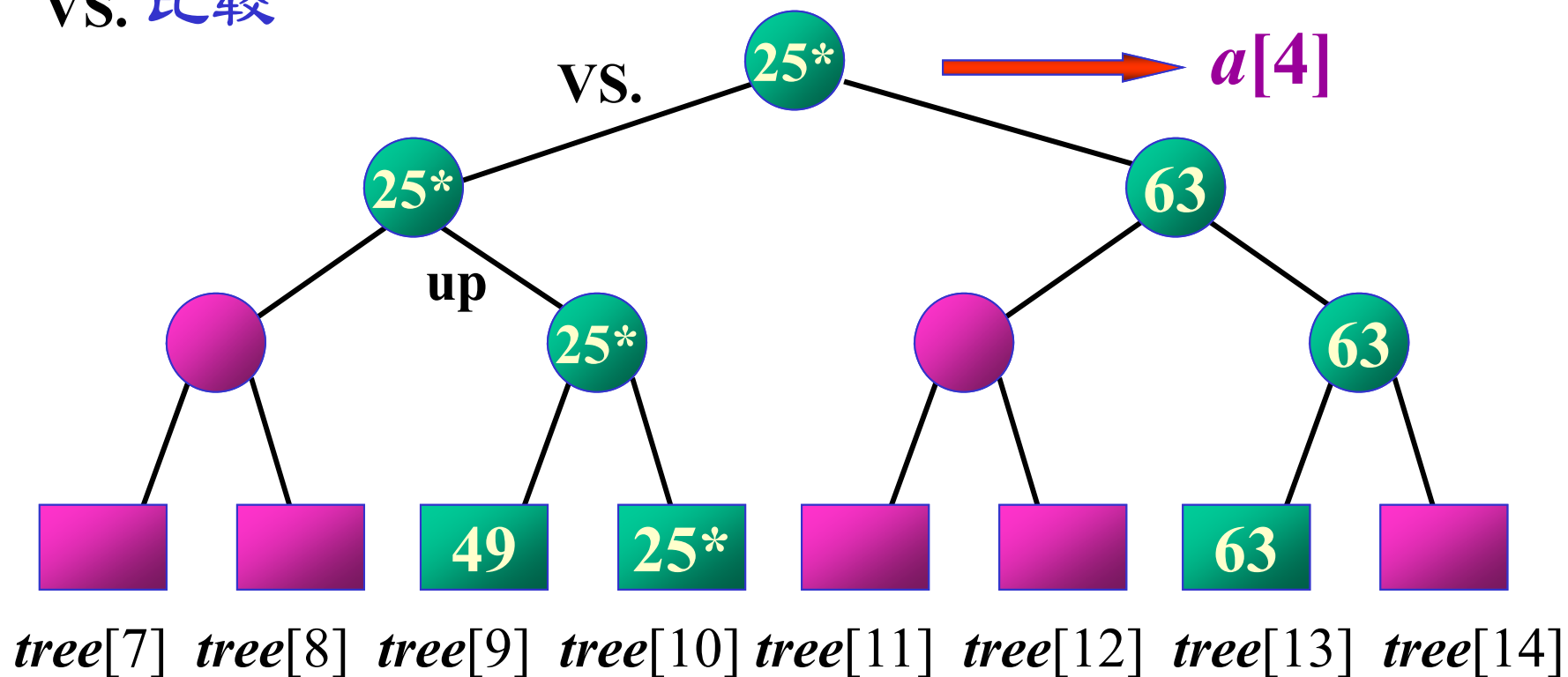


输出第三名并调整胜者树后树的状态

排序码比较次数：2

up 对手不参选
VS. 比较

Winner (胜者)

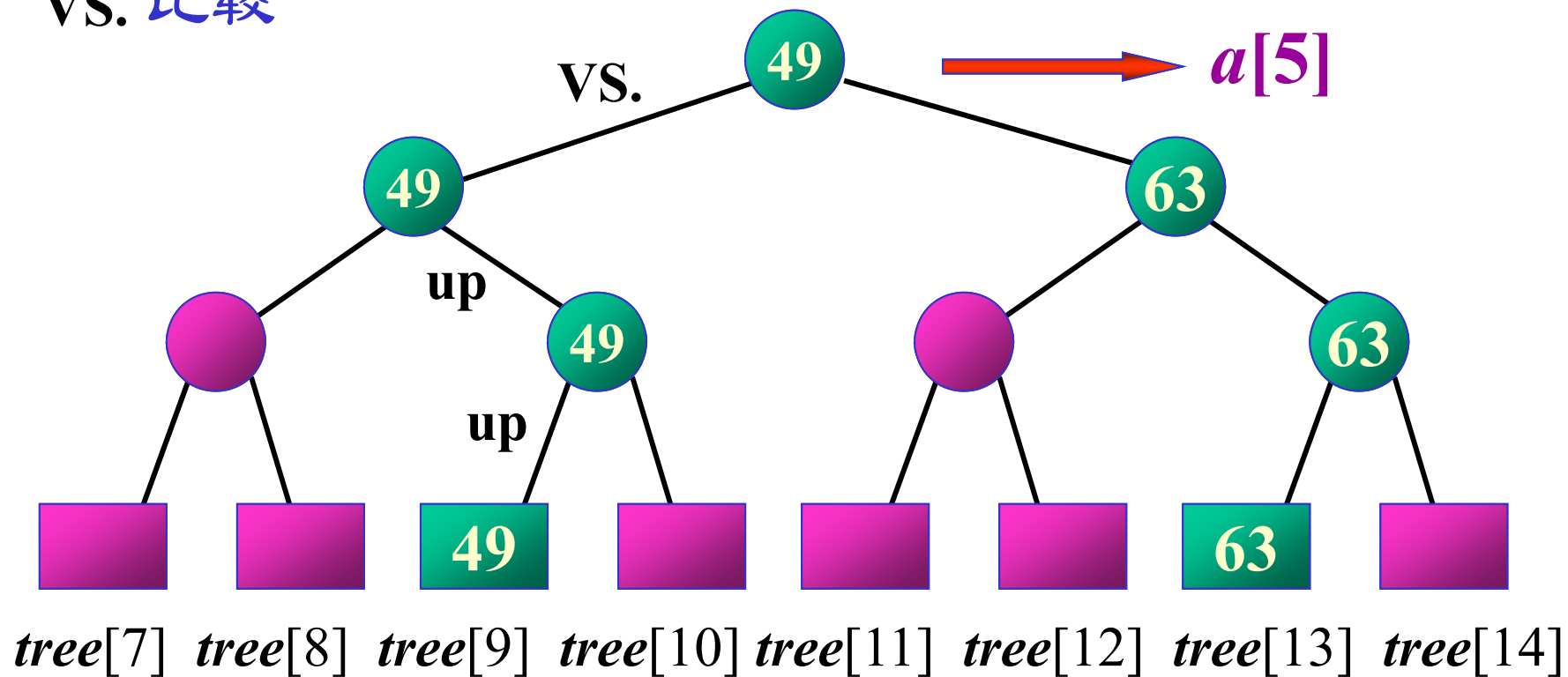


输出第四名并调整胜者树后树的状态

排序码比较次数：1

up 对手不参选
VS. 比较

Winner (胜者)

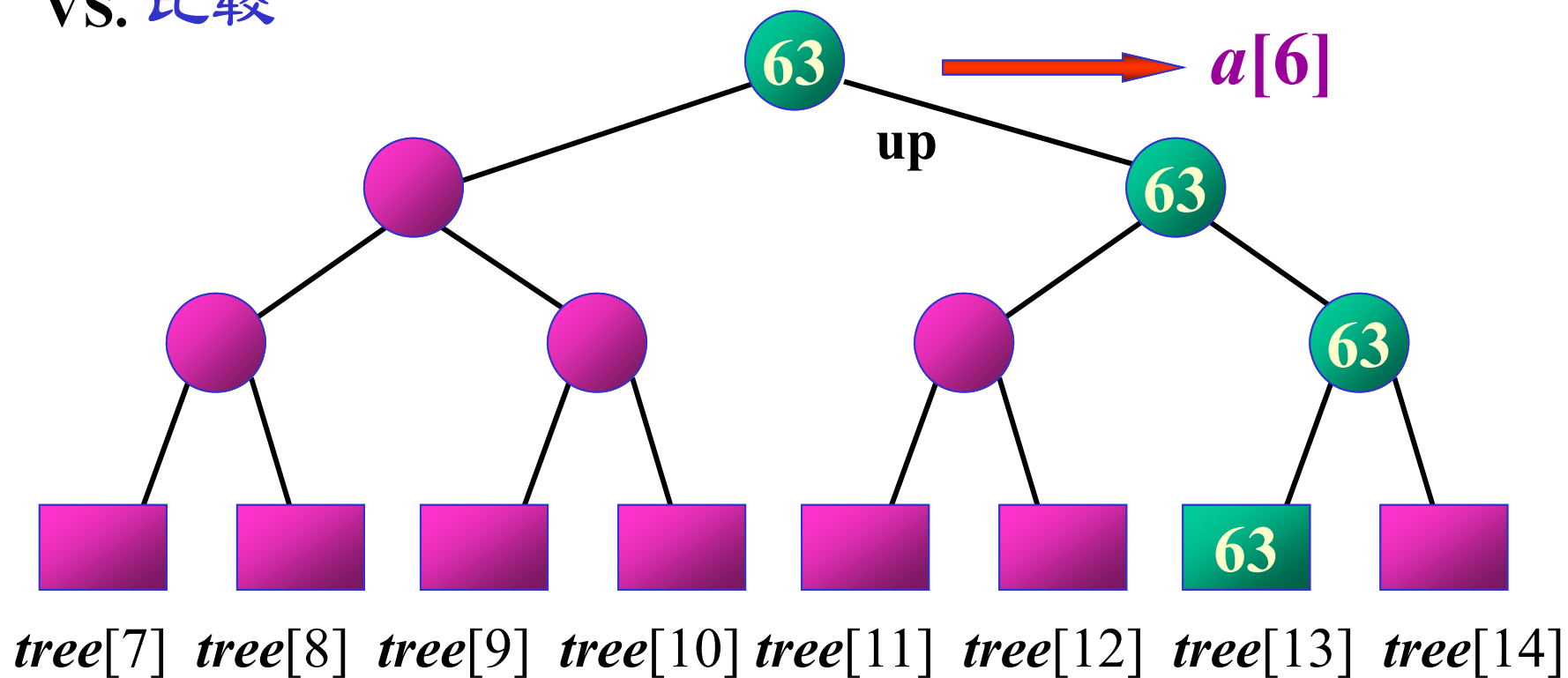


输出第五名并调整胜者树后树的状态

排序码比较次数：1

up 对手不参选
VS. 比较

Winner (胜者)



全部比赛结果输出时树的状态

排序码比较次数：0

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
    int TreeSize = 2*bottomRowSize-1; //总结点数
    int loadindex = bottomRowSize-1; //内结点数
    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 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
}
}
```

- 锦标赛排序构成的胜者树是满的完全二叉树，其深度为 $\lceil \log_2 n \rceil$ ，其中 n 为待排序元素个数。
- 除第一次选择具有最小排序码的对象需要进行 $n-1$ 次排序码比较外，重构胜者树选择具有次小、再次小排序码对象所需的排序码比较次数均为 $O(\log_2 n)$ 。总排序码比较次数为 $O(n \log_2 n)$ 。
- 对象的移动次数不超过排序码的比较次数，所以锦标赛排序总时间复杂度为 $O(n \log_2 n)$ 。

- 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

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

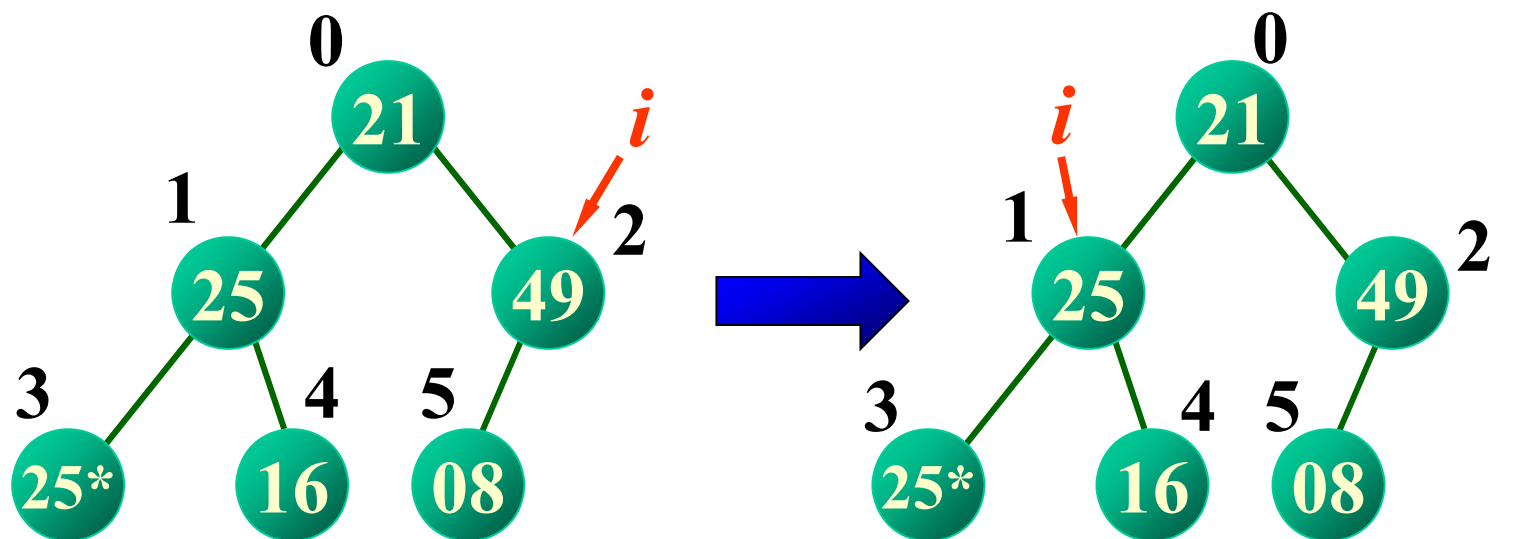
的 k ，使用 $2*2^k-1$ 个结点。

- 每个结点包括排序码、结点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：
 - ◆ 根据初始输入数据，利用堆的调整算法 **FilterDown()** 形成初始堆；
 - ◆ 通过一系列对象交换和重新调整堆进行排序。

建立初始的最大堆

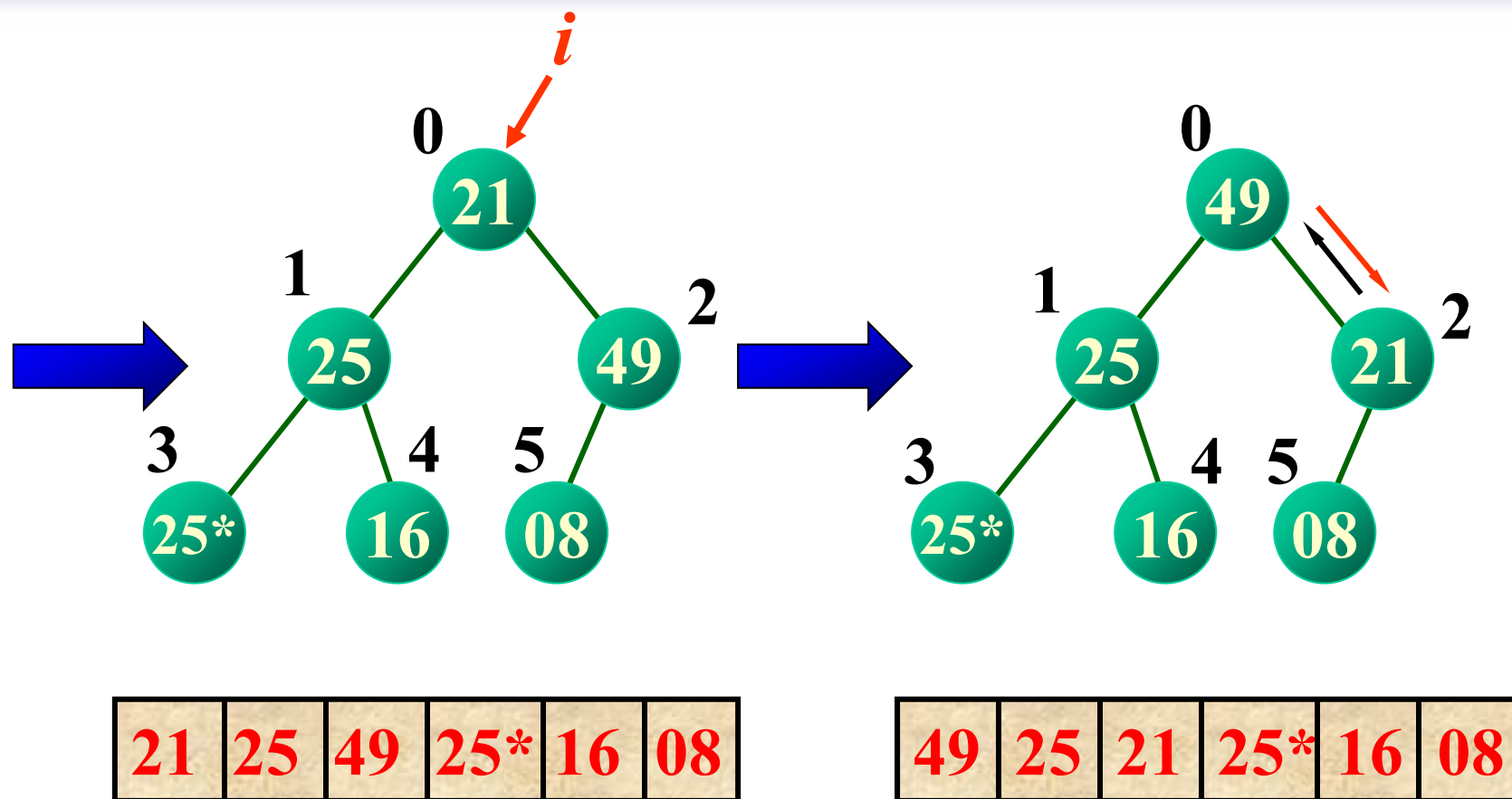


21	25	49	25*	16	08
----	----	----	-----	----	----

初始排序码集合

21	25	49	25*	16	08
----	----	----	-----	----	----

$i = 2$ 时的局部调整



$i = 1$ 时的局部调整

$i = 0$ 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 ) { //最后位置  
        if ( child +1 < EndOfHeap &&  
            heap[child] < heap[child+1] )  
            child = child+1; //选两子女的大者  
        if ( temp >= heap[child] )  
            break; //temp排序码大, 不做调整  
        else {
```



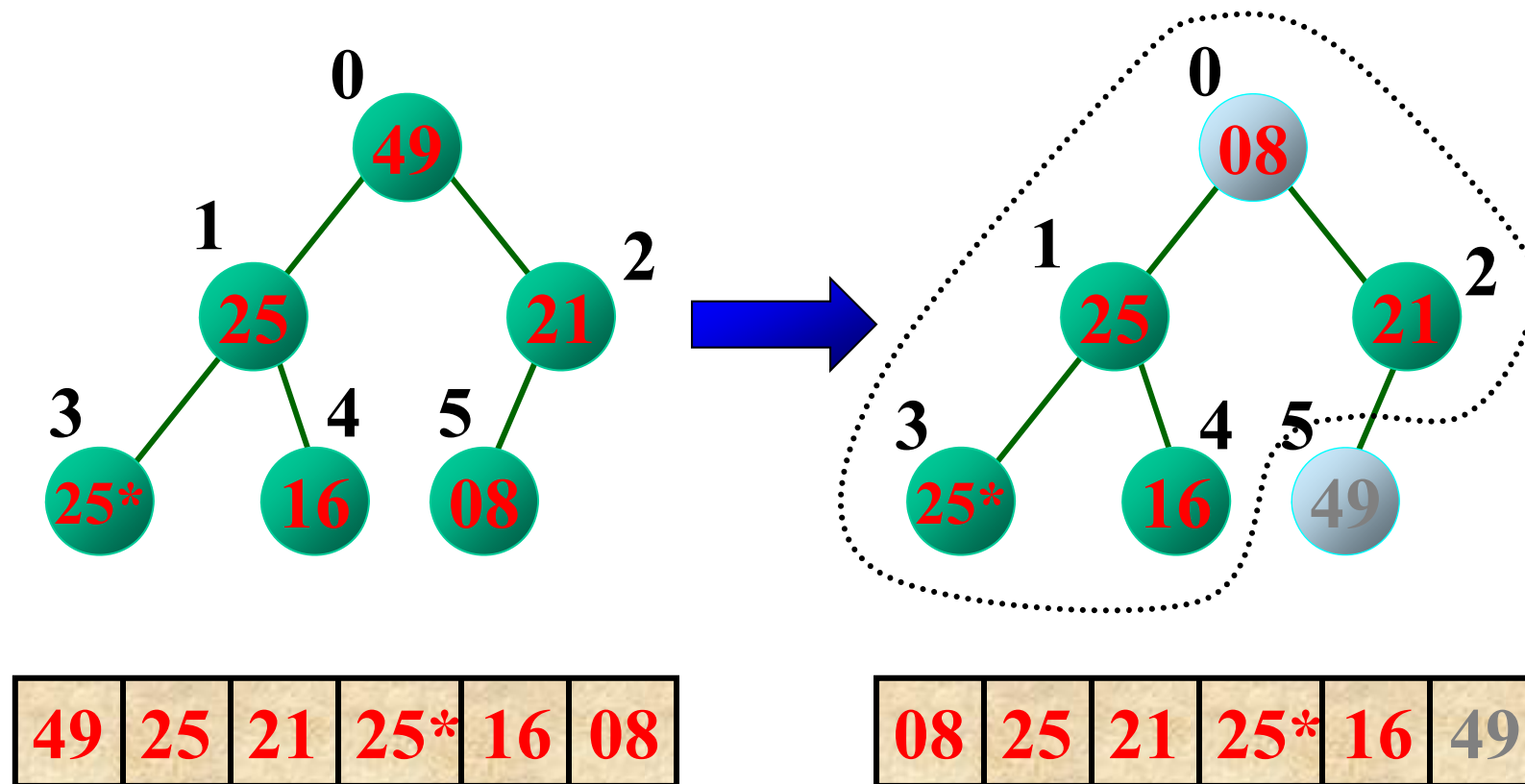
```
    heap[current] = heap[child]; //大子女上移
    current = child; //向下继续调整
    child = 2*child+1;
}
}
heap[current] = temp; //回放至合适位置
}
```

将表转换成堆

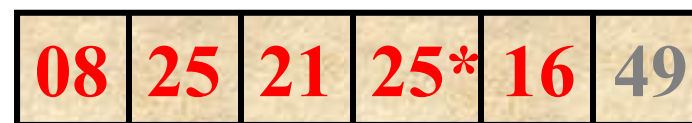
```
for ( int i = ( CurrentSize-2) / 2 ; i >= 0; i-- )
    FilterDown ( i, CurrentSize-1 );
```

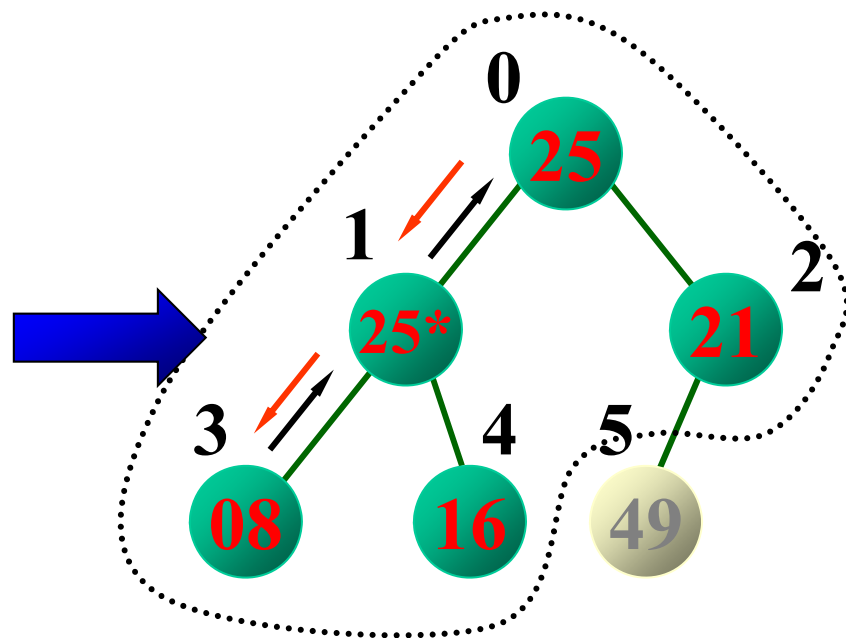
基于初始堆进行堆排序

- 最大堆堆顶 $V[0]$ 具有最大的排序码，将 $V[0]$ 与 $V[n-1]$ 对调，把具有最大排序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 $\text{FilterDown}(0, n-2)$ ，重新建立最大堆，具有次最大排序码的对象又上浮到 $V[0]$ 位置。
- 再对调 $V[0]$ 和 $V[n-2]$ ，调用 $\text{FilterDown}(0, n-3)$ ，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。



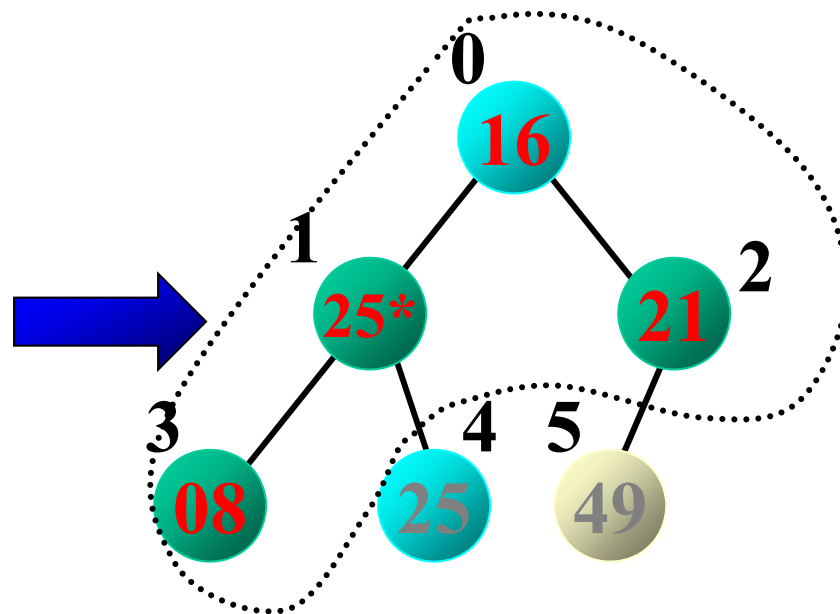
初始最大堆

交换 0 号与 5 号对象，
5 号对象就位



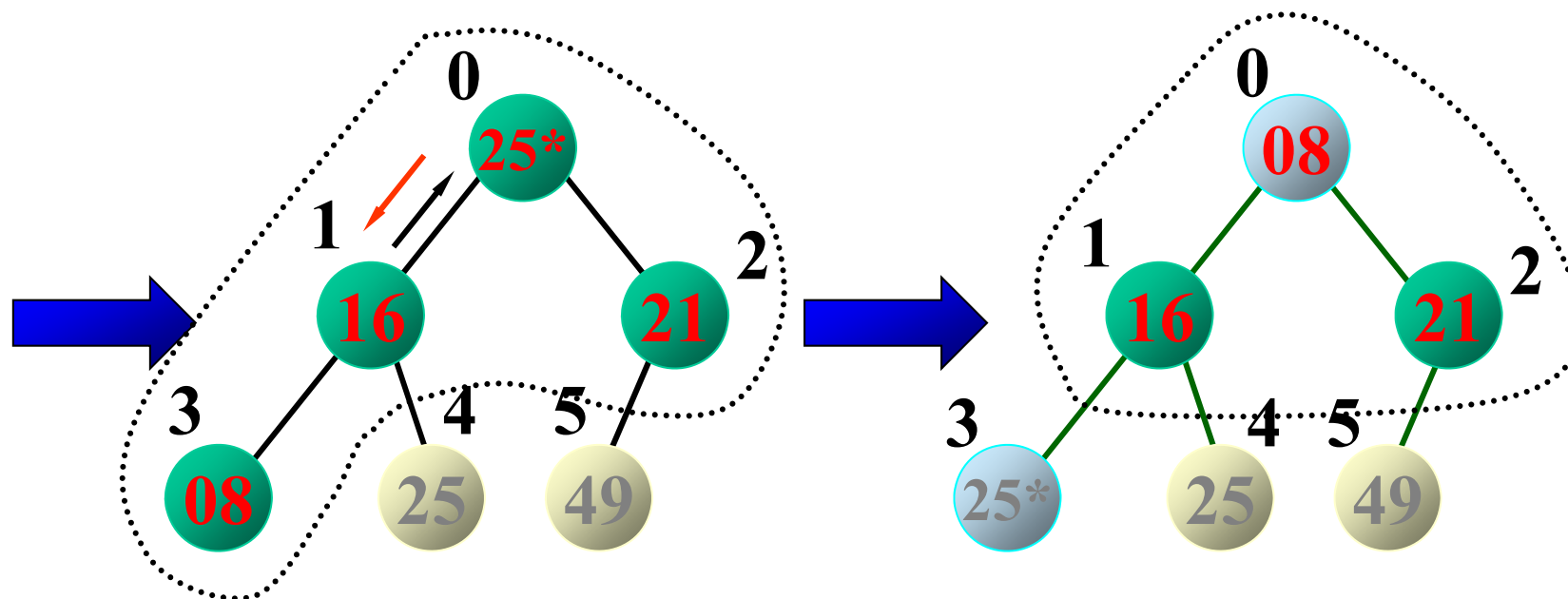
25	25*	21	08	16	49
----	-----	----	----	----	----

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

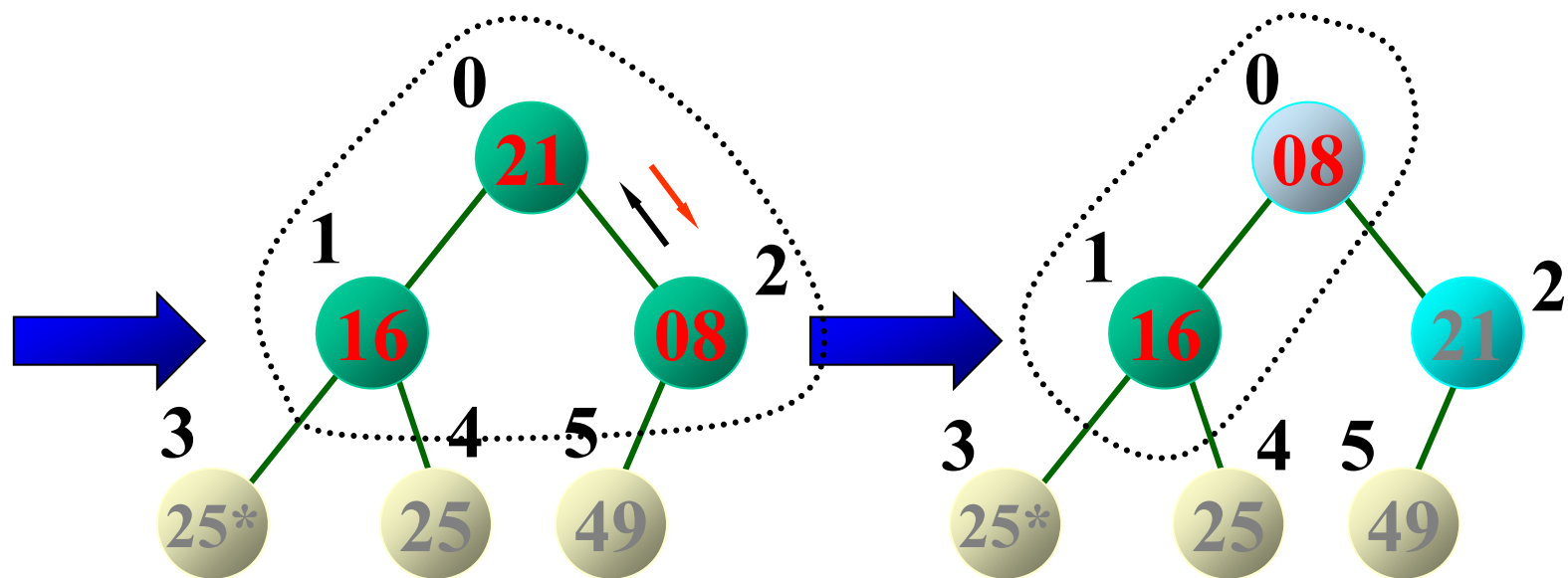


25*	16	21	08	25	49
-----	----	----	----	----	----

从 0 号到 3 号重新
调整为最大堆

08	16	21	25*	25	49
----	----	----	-----	----	----

交换 0 号与 3 号对象，
3 号对象就位

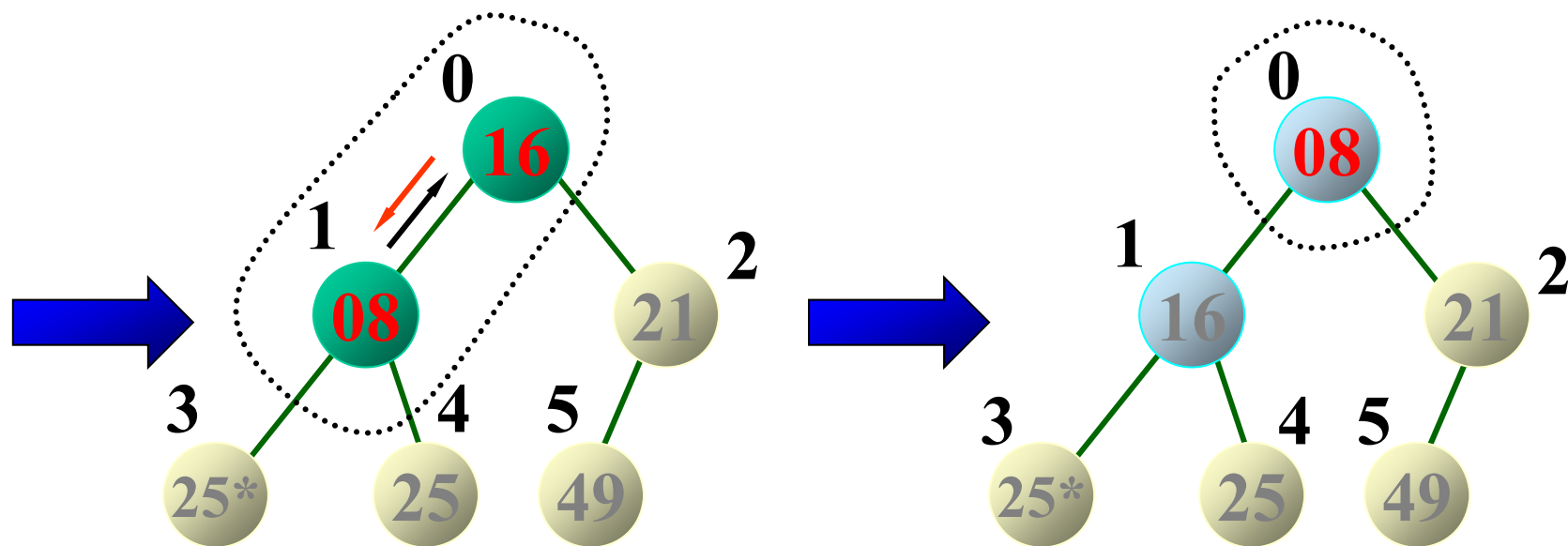


21	16	08	25*	25	49
----	----	----	-----	----	----

从 0 号到 2 号重新
调整为最大堆

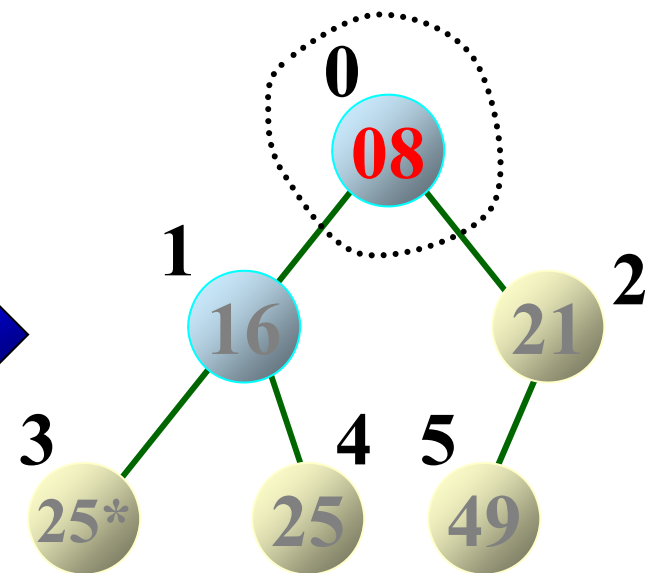
08	16	21	25*	25	49
----	----	----	-----	----	----

交换 0 号与 2 号对象，
2 号对象就位



16	08	21	25*	25	49
----	----	----	-----	----	----

从 0 号到 1 号重新
调整为最大堆



08	16	21	25*	25	49
----	----	----	-----	----	----

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排序码非递减有序  
    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  
    }  
}
```


- 若设堆中有 n 个结点，且 $2^{k-1} \leq n < 2^k$ ，则对应的完全二叉树有 k 层。在第 i 层上的结点数 $\leq 2^i$ ($i = 0, 1, \dots, k-1$)。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。

$$\begin{aligned} 2 \cdot \sum_{i=0}^{k-2} 2^i \cdot (k-i-1) &= 2 \cdot \sum_{j=1}^{k-1} 2^{k-j-1} \cdot j = \\ &= 2 \cdot 2^{k-1} \sum_{j=1}^{k-1} \frac{j}{2^j} \leq 2 \cdot n \sum_{j=1}^{k-1} \frac{j}{2^j} < 4n \end{aligned}$$

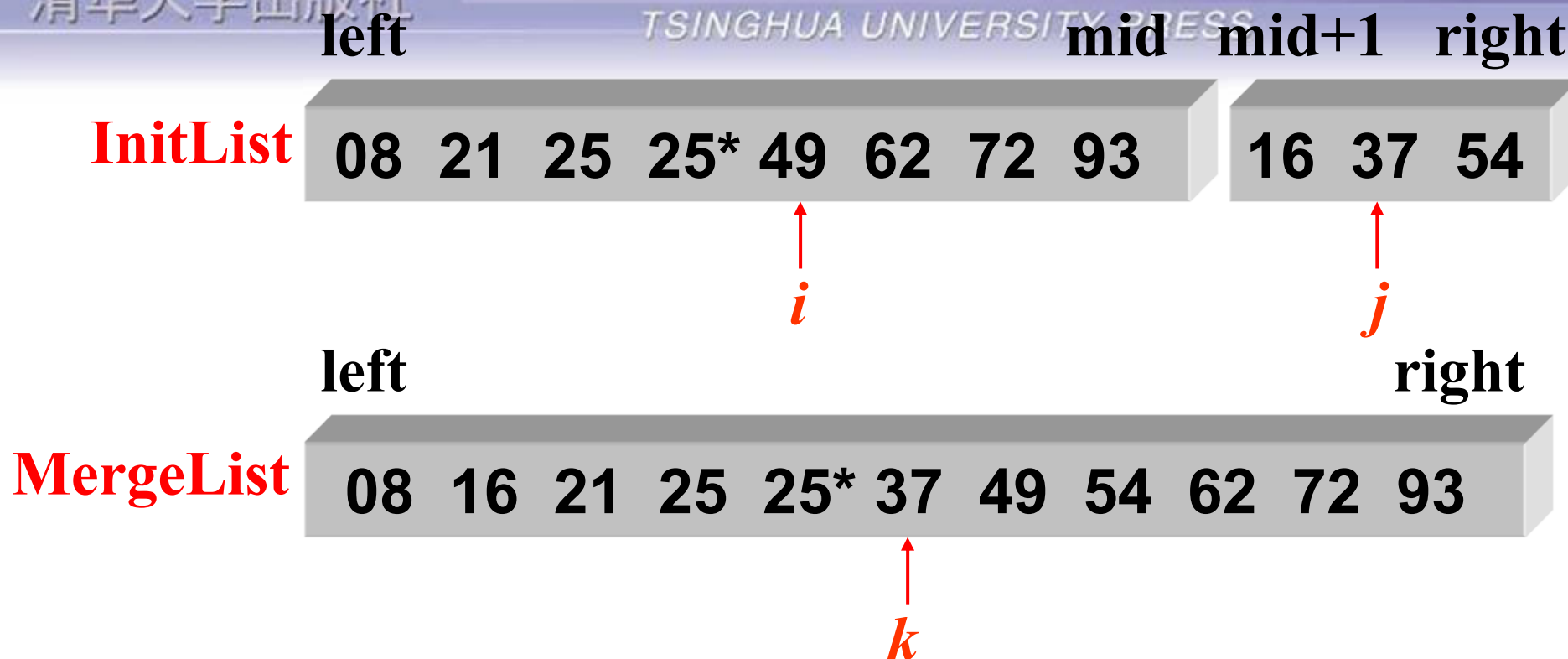
- 第二个for循环中调用了 $n-1$ 次FilterDown()算法, 该循环的计算时间为 $O(n \log_2 n)$ 。因此, 堆排序的时间复杂度为 $O(n \log_2 n)$ 。
- 该算法的附加存储主要是在第二个for循环中用来执行对象交换时所用的一个临时对象。因此, 该算法的空间复杂度为 $O(1)$ 。
- 堆排序是一个不稳定的排序方法。



9.5 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** 中两个有序表 $V[l] \dots V[m]$ 和 $V[m+1] \dots V[n]$ 。它们可归并成一个有序表，存于另一对象序列 **MergedList** 的 $V[l] \dots V[n]$ 中。
- 这种归并方法称为 两路归并 (2-way Merging)。
- 设变量 i 和 j 分别是表 $V[l] \dots V[m]$ 和 $V[m+1] \dots V[n]$ 的当前检测指针，变量 k 是存放指针。



- ◆ 当*i*和*j*都在两个表的表长内变化时，根据对应项的排序码的大小，依次把排序码小的对象排放到新表*k*所指位置中。
- ◆ 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中。

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 DataList <Type> ::  
Merge ( DataList <Type> &MergedList,  
        const int left, const int mid, const int right ) {  
    int i = left, j = mid+1, k = left;  
    while ( i <= mid && j <= right ) //两两比较  
        if ( Vector[i] <= Vector[j] ) {  
            MergedList.Vector[k] = Vector[i];  
            i++; k++;  
        }  
        else {  
            MergedList.Vector[k] = Vector[j];  
            j++; k++;  
        }  
}
```

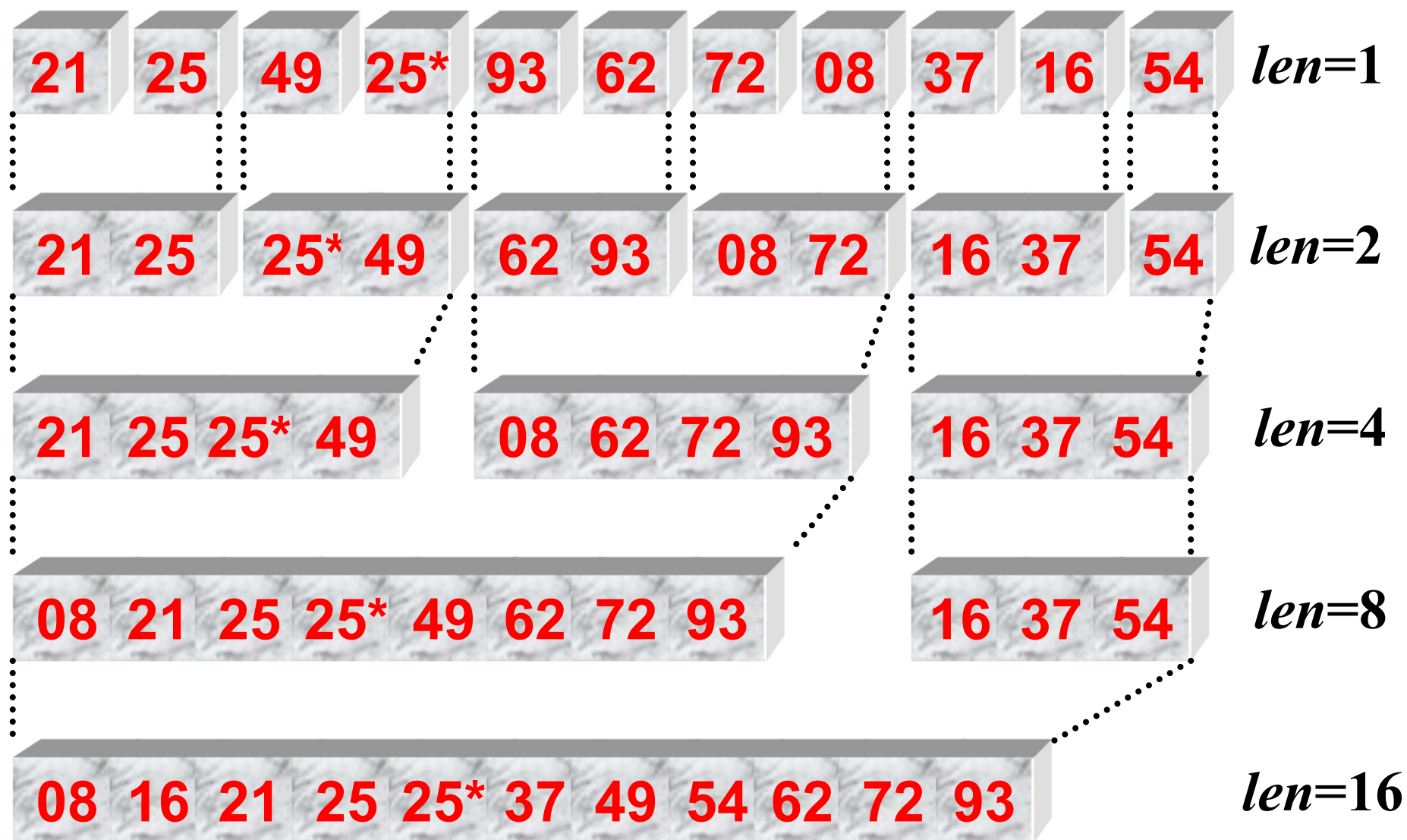
```
        j++; k++;
    }
    if ( i <= mid )
        for ( int n1 = k, n2 = i; n2 <= mid;
              n1++, n2++ )
            MergedList.Vector[n1] = Vector[n2];
    else
        for ( int n1 = k, n2 = j; n2 <= right;
              n1++, n2++ )
            MergedList.Vector[n1] = Vector[n2];
}
```

一趟归并排序的情形

- 设 $\text{InitList.Vector}[0]$ 到 $\text{InitList.Vector}[n-1]$ 中 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，归并成长度为 $2len$ 的归并项，结果放到 MergedList.Vector 中。
- 如果 n 不是 $2len$ 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 Merge 算法将它们归并成一个长度小于 $2len$ 的归并项。
 - ◆ 只剩下一个归并项，其长度小于或等于 len ，将它直接抄到 MergedList.Vector 中。


```
template <class Type> void Datalist <Type> ::  
MergePass ( DataList <Type> &MergedList,  
            const int len ) {  
    int i = 0;  
    while ( i+2*len <= CurrentSize-1 ) {  
        Merge ( MergedList, i, i+len-1, i+2*len-1 );  
        i += 2 * len; //循环两两归并  
    }  
    if ( i+len <= CurrentSize-1 )  
        Merge ( MergedList, i, i+len-1, CurrentSize-1 );  
    else for ( int j = i; j <= CurrentSize-1; j++ )  
        MergedList.Vector[j] = Vector[j];  
}
```

迭代的归并排序算法



(两路) 归并排序的主算法

```
template <class Type> void DataList <Type> ::  
MergeSort ( ) {
```

```
//按对象排序码非递减的顺序对表中对象排序
```

```
    DataList <Type> &TempList(MaxSize);
```

```
    int len = 1;
```

```
    while ( len < CurrentSize ) {
```

```
        MergePass ( TempList, len ); len *= 2;
```

```
        TempList.MergePass ( this, len ); len *= 2;
```

```
    }
```

```
}
```

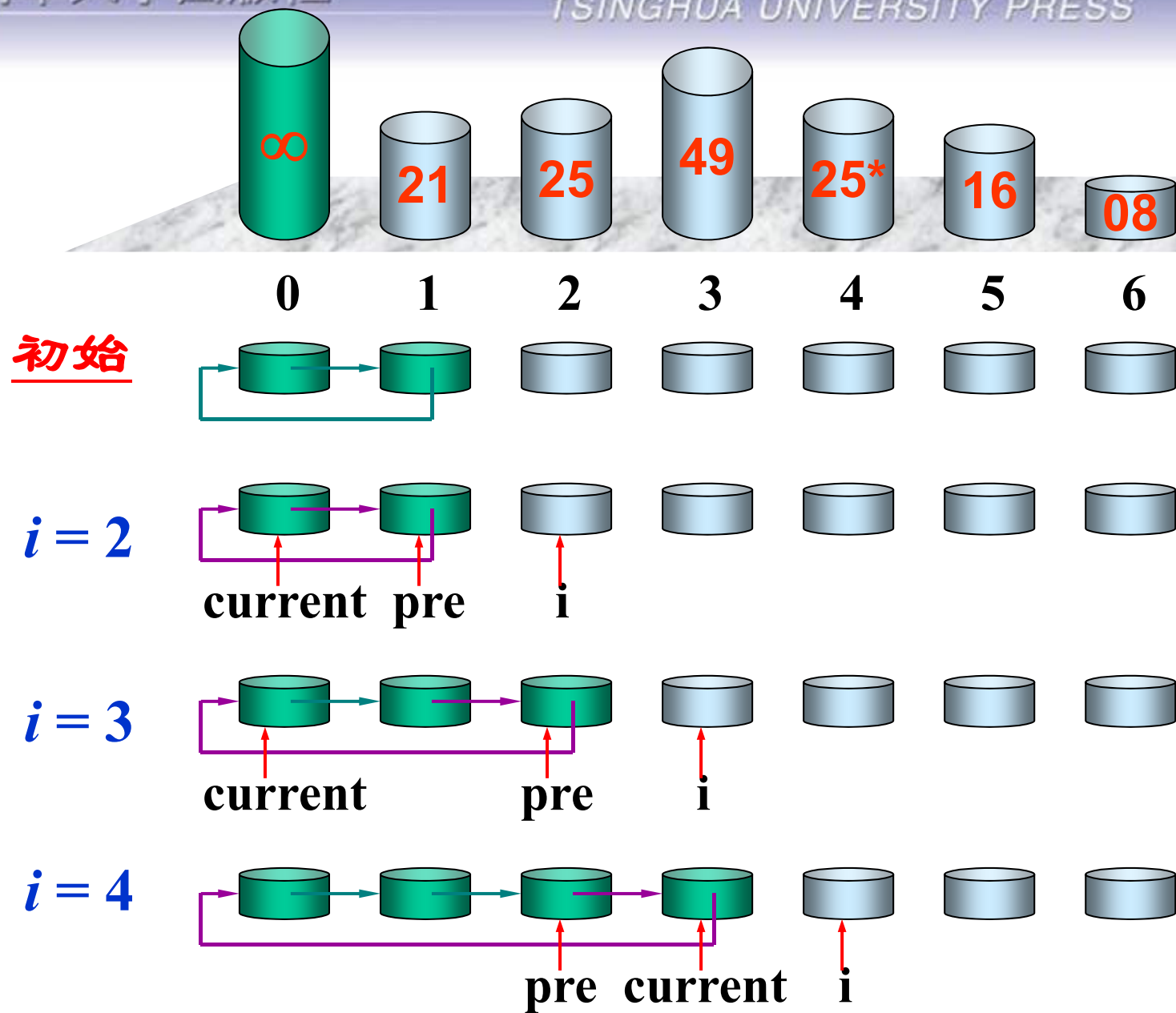
- 在迭代的归并排序算法中，函数MergePass() 做一趟两路归并排序，要调用Merge()函数 $\lceil n/(2*\text{len}) \rceil \approx O(n/\text{len})$ 次，函数MergeSort()调用MergePass()正好 $\lceil \log_2 n \rceil$ 次，而每次Merge()要执行比较 $O(\text{len})$ 次，所以算法总的时间复杂度为 $O(n\log_2 n)$ 。
- 归并排序占用附加存储较多，需要另外一个与原待排序对象数组同样大小的辅助数组。这是这个算法的缺点。
- 归并排序是一个稳定的排序方法。

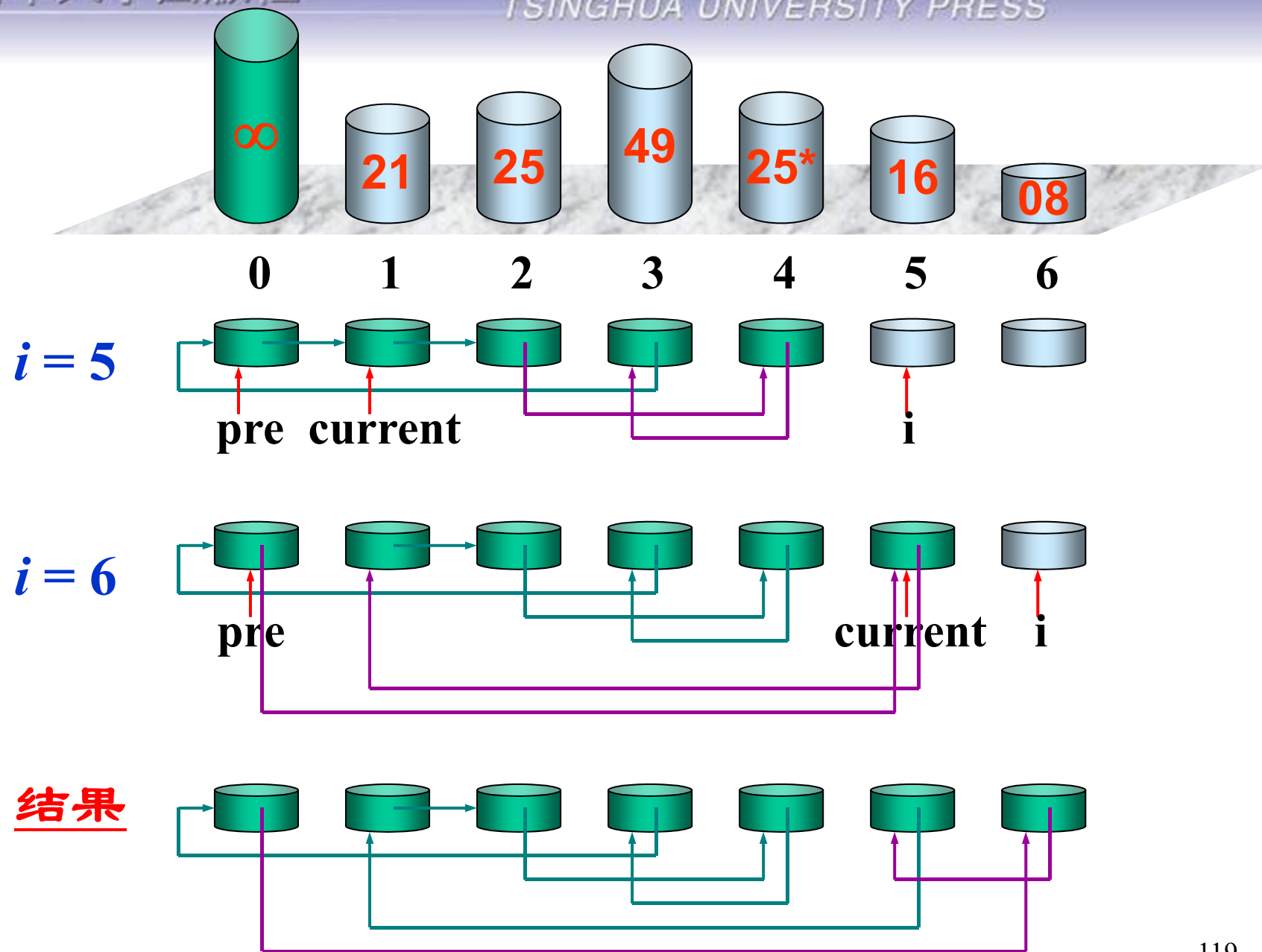


9.6 基于链表的排序算法

链表插入排序

- **基本思想**：在每个对象的结点中增加一个**链接指针数据成员link**。
- 对于数组中的一组对象 $V[1], \dots, V[n]$ ，若 $V[1], \dots, V[i-1]$ 已经通过指针link，按其排序码从小到大链接起来。现要插入 $V[i], i = 2, 3, \dots, n$ ，则必须在前面 $i-1$ 个链接起来的对象当中，循链检测比较，找到 $V[i]$ 应插入的位置，把 $V[i]$ 链入，并修改相应链接指针。从而，得到 $V[1], \dots, V[i]$ 的一个通过指针排好的链表。如此重复执行，直到把 $V[n]$ 也插入到链表中排好序为止。





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; //结点的排序码  
    int link; //结点的链接指针  
public:  
    Element ( ) : key(0), link (NULL) { }  
    Type GetKey ( ) { return key; }  
    void SetKey ( const Type x ) { key = x; }  
    int GetLink ( ) { return link; }  
    void SetLink ( const int l ) { link = l; }  
};
```



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

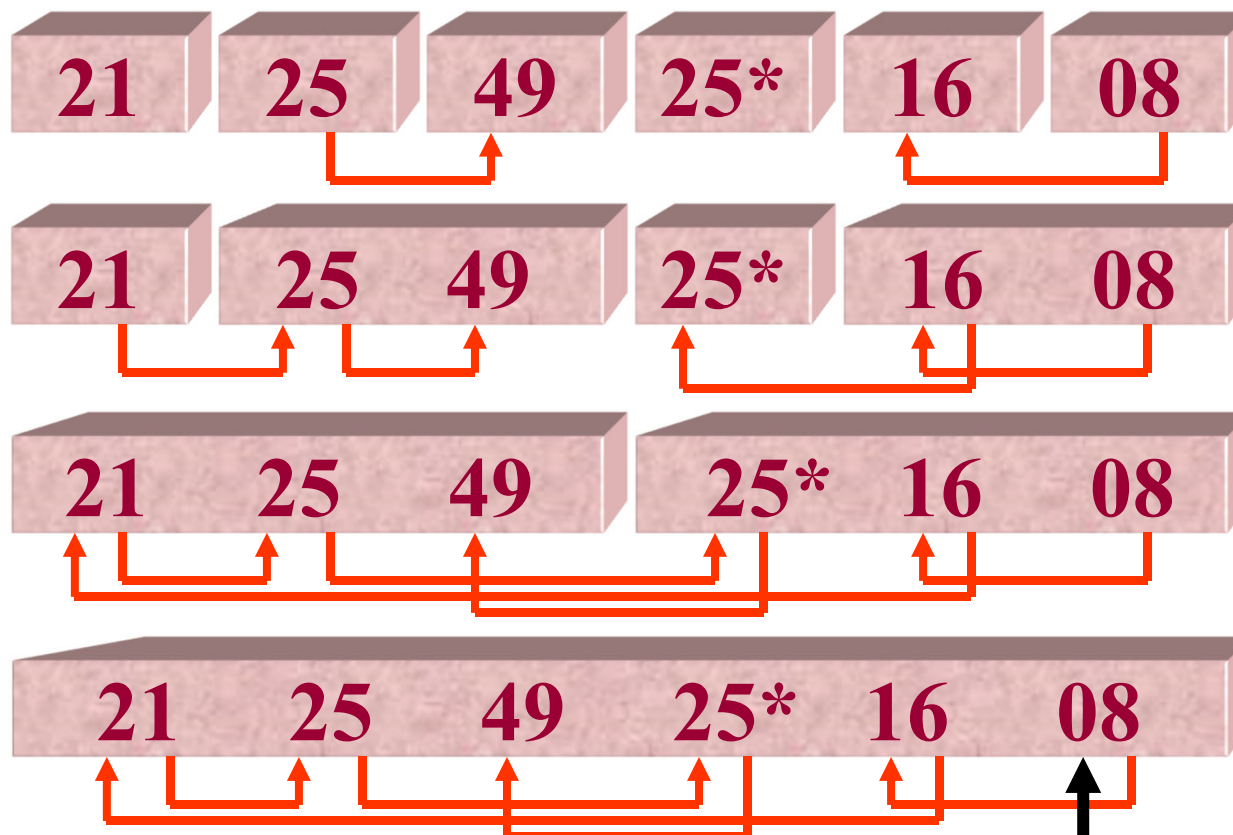
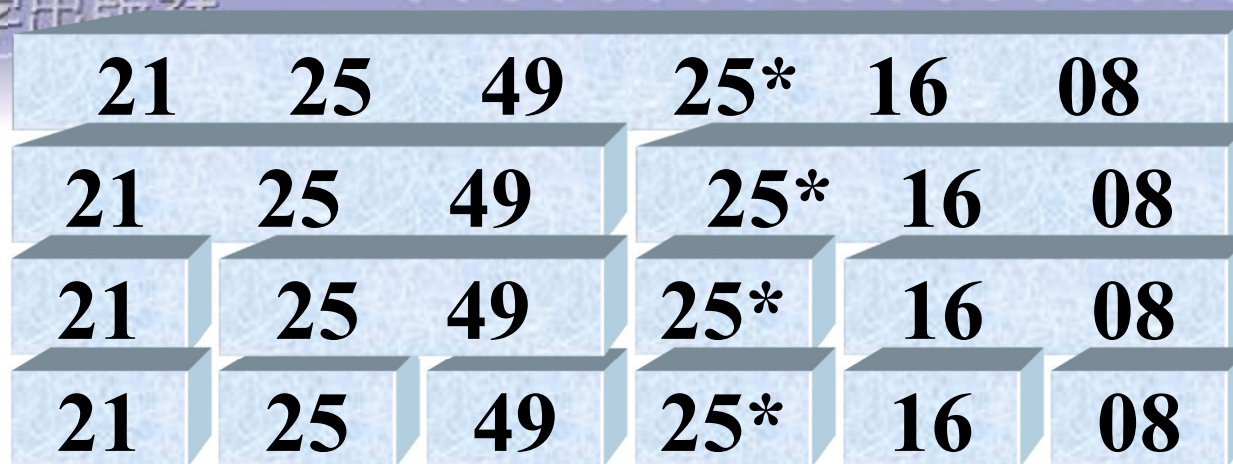
链表插入排序的算法

```
template <class Type> int Staticlinklis <Type> ::  
LinkedInsSort ( ) {  
    Vector[0].key = MaxNum;  
    Vector[0].link = 1; Vector[1].link = 0;  
    //元素V[0]与V[1]形成循环链表  
    for ( int i = 2; i <= CurrentSize; i++ ) {  
        int current = Vector[0].link; //检测指针  
        int pre = 0; //检测指针的前驱指针  
        while ( Vector[current].key <= Vector[i].key )  
        { //找插入位置  
            pre = current; //pre跟上, current向前走  
            current = Vector[current].link; }  
        Vector[i].link = current; Vector[pre].link = i;  
        //在pre与current之间链入    }  
    }
```

- 使用链表插入排序，每插入一个对象，最大排序码比较次数等于链表中已排好序的对象个数，最小排序码比较次数为1。故总的排序码比较次数最小为 $n-1$ ，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，用了 n 个附加域和一个附加对象。
- 算法在Vector[pre].key == Vector[i].key时，将Vector[i]插在Vector[pre]的后面，所以链表插入排序方法是稳定的。

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：待排序对象序列的排序码为{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。再实施归并，逐步退出递归调用的过程。



静态链表的两路归并算法

```
template <class Type> int StaticlinkList <Type> ::  
ListMerge ( const int start1, const int start2 ) {  
    int k = 0, i = start1, j = start2;  
    while ( i && j )  
        if ( Vector[i].key <= Vector[j].key ) {  
            Vector[k].link = i; k = i;  
            i = Vector[i].link; }  
        else {  
            Vector[k].link = j; k = j;  
            j = Vector[j].link; }  
    if ( i == 0 ) Vector[k].link = j;  
    else Vector[k].link = i;  
    return Vector[0].link;  
}
```

递归的归并排序算法

```
template <class Type> int StaticlinkList <Type> ::  
MergeSort ( const int left, const int right ) {  
    if ( left >= right ) return left;  
    int mid = ( left + right ) / 2;  
    return ListMerge ( MergeSort ( left, mid ),  
                        MegerSort ( mid+1, right ) );  
    //以中点mid为界  
    //分别对左半部和右半部进行表归并排序  
}
```

- 链表的归并排序方法的**递归深度**为 $O(\log_2 n)$ ，对象排序码的比较次数为 $O(n\log_2 n)$ 。
- 链表的归并排序方法是一种**稳定**的排序方法。



9.7 分配排序 – 基数排序(Radix Sort)

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

多排序码排序

- 以扑克牌排序为例，每张扑克牌有两个“排序码”：**花色**和**面值**。其有序关系为：
 - ◆ 花色： $\clubsuit < \diamondsuit < \heartsuit < \spadesuit$
 - ◆ 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**多排序码排序**。排序后形成的有序序列叫做**词典有序序列**。
- 对于上例两排序码的排序，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 $\{V_0, V_1, \dots, V_{n-1}\}$ ，且每个对象 V_i 中含有 d 个排序码。

- 如果对于序列中任意两个对象 V_i 和 V_j ($0 \leq i < j \leq n-1$) 都满足:

$$(K_i^1, K_i^2, \dots, K_i^d) < (K_j^1, K_j^2, \dots, K_j^d)$$

则称序列对排序码 (K^1, K^2, \dots, K^d) 有序。其中, K^1 称为最高位排序码, K^d 称为最低位排序码。

- 如果排序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排序码排序。

- **实现多排序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**最高位排序码 K^1** 排序，得到若干对象组，对象组中各对象都有相同**排序码 K^1** 。
 - ◆ 再分别对每组中对象根据**排序码 K^2** 进行排序，按 **K^2** 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**排序码 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**最低位排序码** K^d 对所有对象进行一趟排序，再依据**次低位排序码** K^{d-1} 对上一趟排序的结果再排序，依次重复，直到依据**排序码** 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排序码进行排序时，不需要再分组，而是整个对象组都参加排序。
- **LSD和MSD方法**也可应用于对一个排序码进行的排序。此时可将**单排序码** K_i 看作是一个子排序码组：

$$(K_i^1, K_i^2, \dots, K_i^d)$$

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”对单排序码进行排序。在这种方法中，把单排序码 K_i 看成是一个 d 元组：
$$(K_i^1, K_i^2, \dots, K_i^d)$$
- 其中，每一个分量 K_i^j ($1 \leq j \leq d$)也可看成是一个排序码。

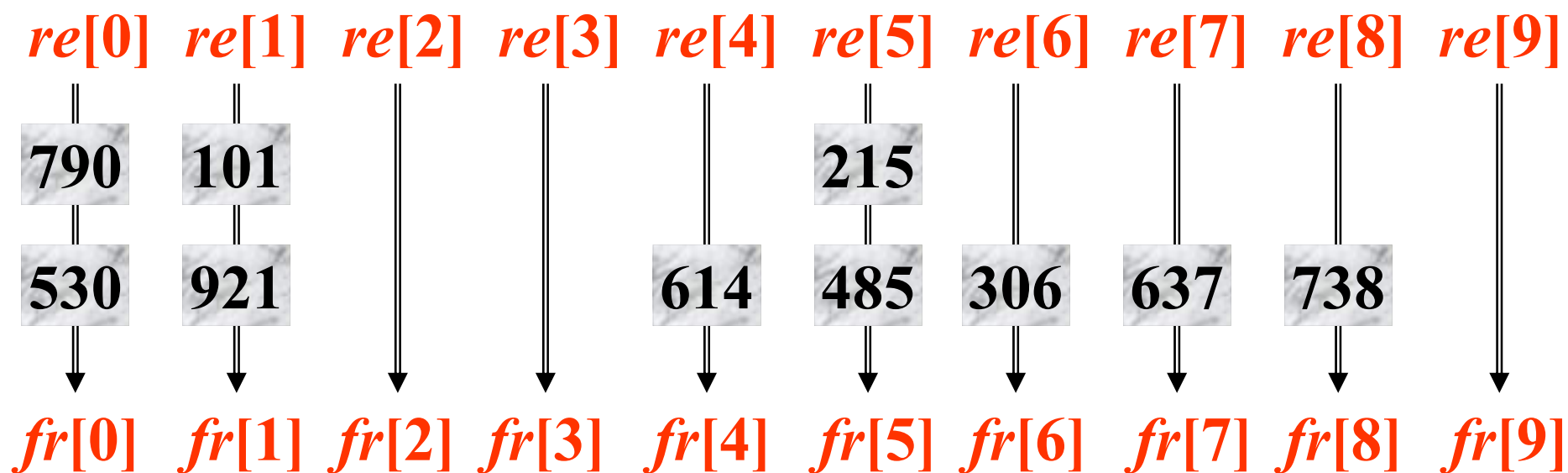
- 分量 K_i^j ($1 \leq j \leq d$) 有 $radix$ 种取值, 称 $radix$ 为基数。例如, 排序码 984 可以看成是一个 3 元组 (9, 8, 4), 每一位有 0, 1, ..., 9 等 10 种取值, 基数 $radix = 10$ 。排序码 'data' 可以看成是一个 4 元组 (d, a, t, a), 每一位有 'a', 'b', ..., 'z' 等 26 种取值, $radix = 26$ 。
- 针对 d 元组中的每一位分量, 把对象序列中的所有对象, 按 K_i^j 的取值, 先 “分配” 到 rd 个队列中去。然后, 再按各队列的顺序, 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排序码 K_0, K_1, \dots, K_{n-1} , 依次对各位的分量, 让 $j = d, d-1, \dots, 10$, 分别用“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, 所有对象就按其排序码的值从小到大排好序了。
- 各队列采用链式队列结构, 分配到同一队列的排序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。

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 $i = 3$)



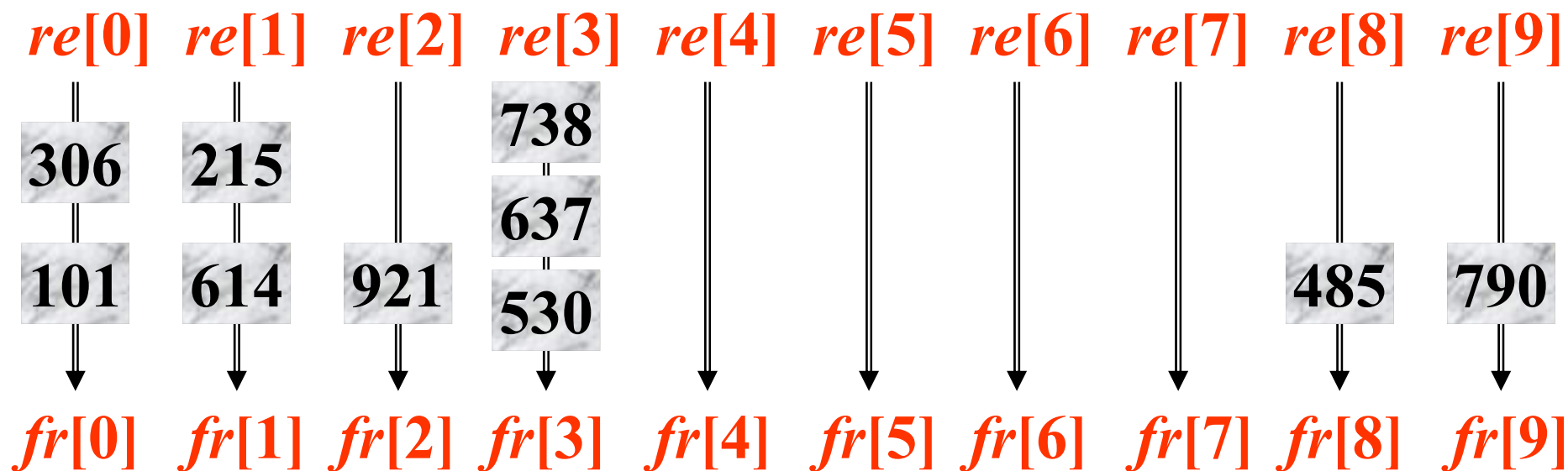
第一趟收集

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

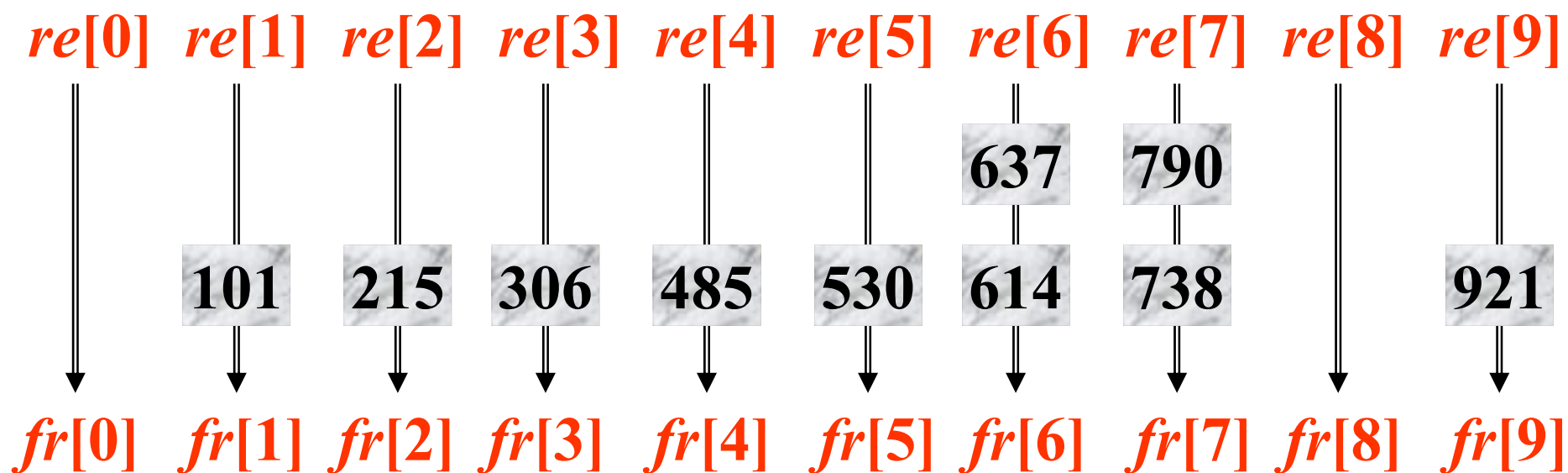
101 → 306 → 614 → 215 → 921 → 530 → 637 → 738 → 485 → 790

138

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

139

链表基数排序

```
template <class Type> void StaticlinkList <Type> ::  
RadixSort ( const int d, const int radix ) {  
    int rear[radix], front[radix];  
    for ( int i = 1; i < CurrentSize; i++ )  
        Vector[i].link = i+1;  
    Vector[n].link = 0; //静态链表初始化  
    int current = 1; //链表扫描指针  
    for ( i = d-1; i >= 0; i-- ) { //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 = Vector[current].key[i];
    //取当前对象排序码的第i位
    if ( front[k] == 0 ) //原链表为空
        front[k] = current; //队头链入
    else //原链表非空, 链尾链入
        Vector[rear[k]].link = current;
    rear[k] = current; //修改链尾指针
    current = Vector[current].link; //下一对象
}
j = 0; //从0号队列开始收集
while ( front[j] == 0 ) j++; //空队列跳过
```

```
Vector[0].link = current = front[j]; //新链头
int last = rear[j]; //新链尾
for ( k = j+1; k < radix; k++ )
//逐个队列链接
    if ( front[k] ) {
        Vector[last].link = front[k];
        last = rear[k];
    }
Vector[last].link = 0; //链收尾
} //下一趟
}
```

- 若每个排序码有 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排序码位数较少的情况，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。

内部排序方法的比较

- 基于“比较-分配”的排序时间复杂度理论下界 $\Omega(n \cdot \log n)$
 - ◆ 比较树证明
- 简单排序：插入排序、冒泡排序、选择排序：
 - ◆ 不需要额外辅助空间
 - ◆ 平均时间复杂度均为 $O(n^2)$
 - ◆ 在元素个数较少时，插入排序性能好，比大多排序都快

内部排序方法的比较

- 高级排序：快速排序、堆排序、归并排序、希尔排序：
 - ◆快速排序：平均性能最好、最坏情况时间复杂度 $O(n^2)$
 - ◆堆排序：平均性能仅次于快速排序、不需额外的存储空间
 - ◆归并排序：稳定的排序方法，没有最坏情况。
 - ◆希尔排序：算法时间复杂度有待研究

内部排序方法的比较

- 基于分配的排序：桶排序、基数排序
 - ◆ 桶排序：待排元素均匀分配时，时间复杂度为 $O(n)$
 - ◆ 基数排序：
 - 具有线性时间复杂度
 - 对整数与字符串有很高效率
 - 限于具体数据，效率与通用性不如基于“比较-交换”的排序

9.8内部排序方法的比较

排序方法	平均时间	最坏时间	最佳时间	辅助空间	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$ (改进)	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
归并排序	$O(n*\log n)$	$O(n*\log n)$	$O(n*\log n)$	$O(n)$	稳定
快速排序	$O(n*\log n)$	$O(n^2)$	$O(n*\log n)$	$O(\log n)$	不稳定
堆排序	$O(n*\log n)$	$O(n*\log n)$	$O(n*\log n)$	$O(1)$	不稳定
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$	稳定

选择排序的方法

- 当待排序记录数 n 较**大**时，若要求排序**稳定**，则采用**归并**排序。
- 当待排序记录数 n 较**大**，关键字**分布随机**，而且**不要求稳定**时，可采用**快速**排序；
- 当待排序记录数 n 较**大**，关键字会**出现正、逆序**情形，可采用**堆**排序（或**归并**排序）。
- 当待排序记录数 n 较**小**，记录已**接近有序**或随机分布时，可采用直接插入排序。



随堂练习

例1：设有**5000**个无序的元素，希望用最快速度挑出其中前**10**个最大的元素。在快速排序、堆排序、归并排序、基数排序和希尔排序方法中，采用哪一种方法最好？为什么？

例2：对由 **n** 个元素组成的线性表进行快速排序时，所需进行的比较次数与这 **n** 个元素的初始排列有关。

(1) 当 **$n=7$** 时，在最好情况下需进行多少次比较？

(2) 当 **$n=7$** 时，给出一个最好情况的初始排列的实例。

(3) 当 **$n=7$** 时，在最坏情况下需进行多少次比较？

(4) 当 **$n=7$** 时，给出一个最坏情况的初始排列的实例。

例3：某个待排序的序列是一个可变长度的字符串序列，这些字符串一个接一个地存储于唯一的字符数组中。请改写快速排序算法，对这个字符串序列进行排序。

例1：设有**5000**个无序的元素，希望用最快速度挑出其中前**10**个最大的元素。在快速排序、堆排序、归并排序、基数排序和希尔排序方法中，采用哪一种方法最好？为什么？

所列几种排序方法的速度都很快，但快速排序、归并排序、基数排序和希尔排序都是在排序结束之后才能确定数据元素的全部顺序，而无法知道排序过程中部分元素的有序性。但堆排序则每次输出一个最小（或最大）的元素，然后对堆进行调整，保证堆顶的元素是未排序元素中的最小（或最大）的。因此，选取前**10**个最大元素采用堆排序方法最好。

例2：对由 **n** 个元素组成的线性表进行快速排序时，所需进行的比较次数与这 **n** 个元素的初始排列有关。

(1) 在最好情况下，每一趟快速排序后均能划分出左、右两个相等的区间，即设线性表的长度 **$n=2^k-1$** ，则第一趟快速排序后划分得到两个长度均为 **$n/2$** 的子表，第二快速排序后划分得到**4**个长度为 **$n/4$** 的子表，以此类推，总共需进行 **$k=\log_2(n+1)$** 遍划分，即子表长度为**1**时排序完毕。因此，当 **$n=7$** 时 **$k=3$** ，也即在最好情况下第一个元素由两头向中间扫描到正中位置，即需与其余**6**个元素都进行比较后找到最终存储位置，因此需要比较**6**次。第二趟分别对左、右两个子表（长度均为**3**，即 **$k=2$** ）进行排序，与第一趟类似，需与子表中其余**2**个元素进行比较后找到其最终存储位置，也即两个子表共需比较**4**次，并且继续划分出的每个子表长度均为**1**，即排序完毕。故总共需比较**10**次。

例2：对由n个元素组成的线性表进行快速排序时，所需进行的比较次数与这n个元素的初始排列有关。

(2) 由(1)所知，每趟排序都应使第一个元素存储于表的正中位置，因此最好的初始排列的例子为：4, 7, 5, 6, 3, 1, 2。

(3) 快速排序最坏的情况是，每趟用来划分的基准元素总是定位于表的第一位置或最后一个位置，这样划分的左、右子表一个长度为0，另一个仅是原表长减1。这样，快速排序的效率蜕化为冒泡排序，其时间复杂度为 $O(n^2)$ ，即比较次数为 $6+5+4+3+2+1=21$ 次。

(4) 由(3)可知，快速排序最坏的情况是初始序列有序。所以，当 $n=7$ 时，最坏情况的初始排列的例子为：1, 2, 3, 4, 5, 6, 7或者7, 6, 5, 4, 3, 2, 1。

9.8 内部排序算法的分析

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	$\log_2 n$	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	



本章小结

- 知识点

- 内排序

- 插入排序（3种）
 - 快速排序
 - 选择排序（3种）
 - 归并排序
 - 基于链表的排序
 - 基数排序
 - 各种内排序算法的比较分析

- **课程习题**

- **笔做题——9.20, 9.24, 9.25**
(以作业形式提交)
- **上机题——9.19, 9.21, 9.23**
- **思考题——9.3, 9.4, 9.5, 9.6, 9.7, 9.9, 9.11,
9.12, 9.14**