

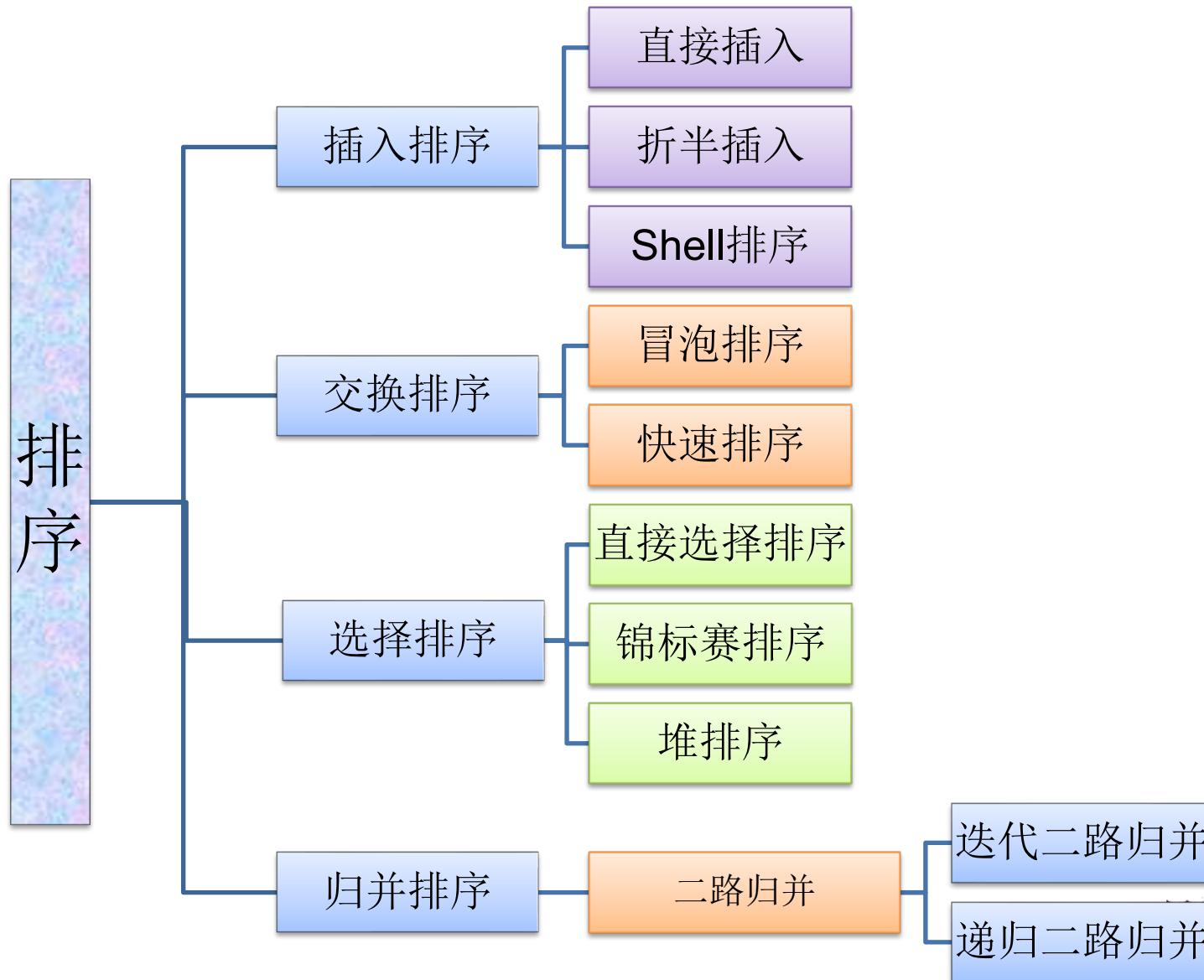


第九章 排序

- 9.1 概述
- 9.2 插入排序
- 9.3 交换排序
- 9.4 选择排序
- 9.5 归并排序



知识导图





第九章 排序

- 9.1 概述
- 9.2 插入排序
- 9.3 交换排序
- 9.4 选择排序
- 9.5 归并排序



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*): 它是待排序数据元素的有限集合。
- 排序码(*key*): 通常数据元素有多个属性域, 即多个数据成员组成, 其中有一个属性域可用来区分元素, 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: 如果在元素序列中有两个元素 $r[i]$ 和 $r[j]$, 它们的排序码 $k[i] == k[j]$, 且在排序之前, 元素 $r[i]$ 排在 $r[j]$ 前面。如果在排序之后, 元素 $r[i]$ 仍在元素 $r[j]$ 的前面, 则称这个排序方法是稳定的, 否则称这个排序方法是不稳定的。
- 内排序与外排序: 内排序是指在排序期间数据元素全部存放在内存的排序; 外排序是指在排序期间全部元素个数太多, 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。对于那些受元素排序码序列初始排列及元素个数影响较大的，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 T>
class Element {                                //数据表元素定义
public:
    T key;                                     //排序码
    field otherdata;                            //其他数据成员
    Element<T>& operator = (Element<T>& x) {
        key = x.key;   otherdata = x.otherdata;
        return this;
}
```



```
bool operator == (Element<T>& x)
{ return key == x.key; }      // 判 *this 与 x 相等
bool operator <= (Element<T>& x)
{ return key <= x.key; }      // 判 *this 小于或等于 x
bool operator >= (Element<T>& x)
{ return key >= x.key; }      // 判 *this 大于或等于 x
bool operator > (Element<T>& x)
{ return key > x.key; }      // 判 *this 大于 x
bool operator < (Element<T>& x)
{ return key < x.key; }      // 判 *this 小于 x
};
```



```
template <class T>
class dataList {                                //数据表类定义
private:
    Element <T>* Vector;                      //存储排序元素的向量
    int maxSize;                                //向量中最大元素个数
    int currentSize;                            //当前元素个数
public:
    dataList (int maxSz = DefaultSize) :      //构造函数
        maxSize(maxSz), currentSize(0)
    { Vector = new Element<T>[maxSize]; }
    int Length() { return currentSize; }         //取表长度
```



```
void Swap (Element<T>& x, Element<T>& y)
{ Element<T> temp = x; x = y; y = temp; }
Element<T>& operator [](int i)           //取第i个元素
{ return Vector[i]; }
int Partition (const int low, const int high);
                           //快速排序划分
};
```



第九章 排序

- 9.1 概述
- 9.2 插入排序
- 9.3 交换排序
- 9.4 选择排序
- 9.5 归并排序



插入排序 (Insert Sorting)

- 基本方法是：每步将一个待排序的元素，按其排序码大小，插入到前面已经排好序的一组元素的适当位置上，直到元素全部插入为止。

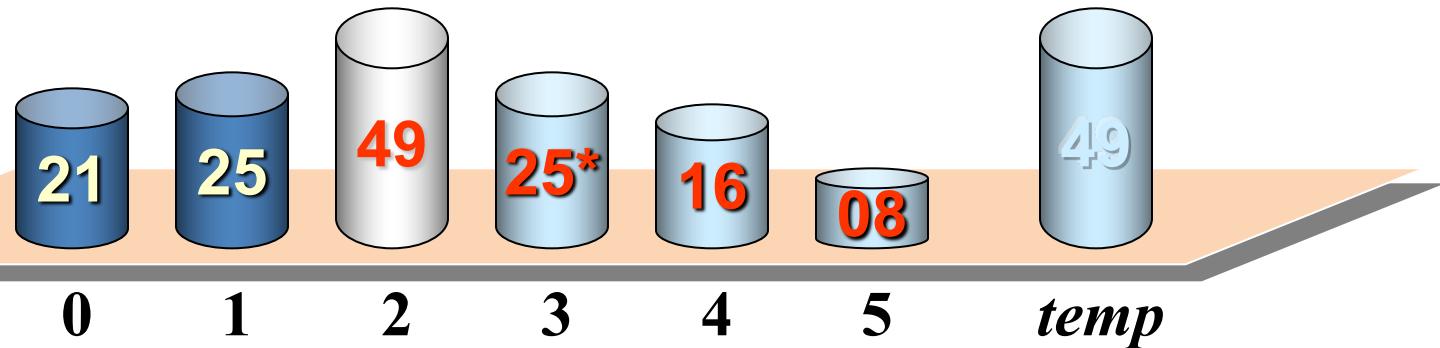
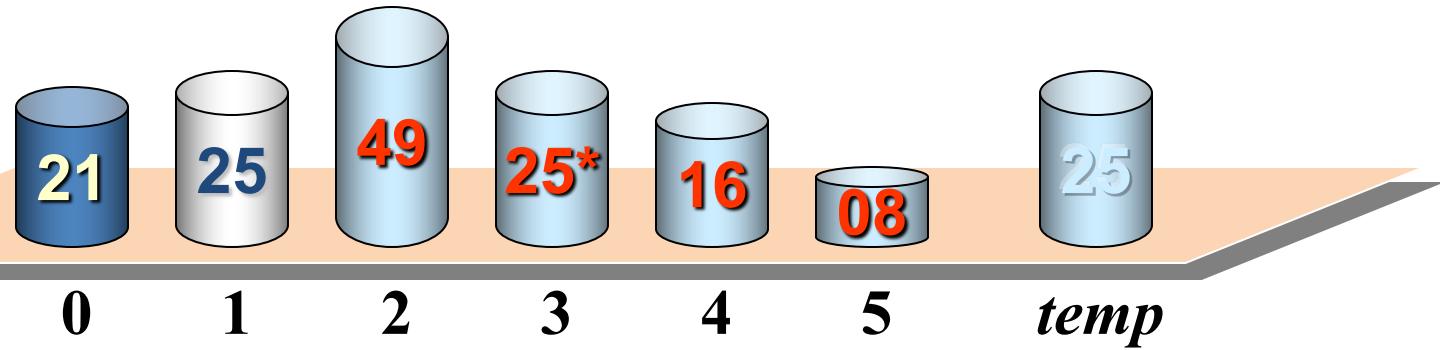
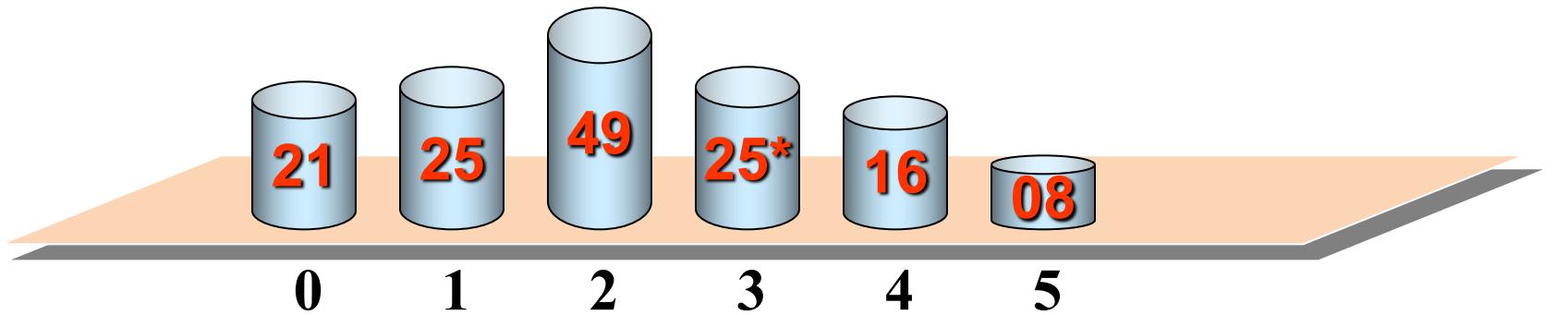
直接插入排序 (Insert Sort)

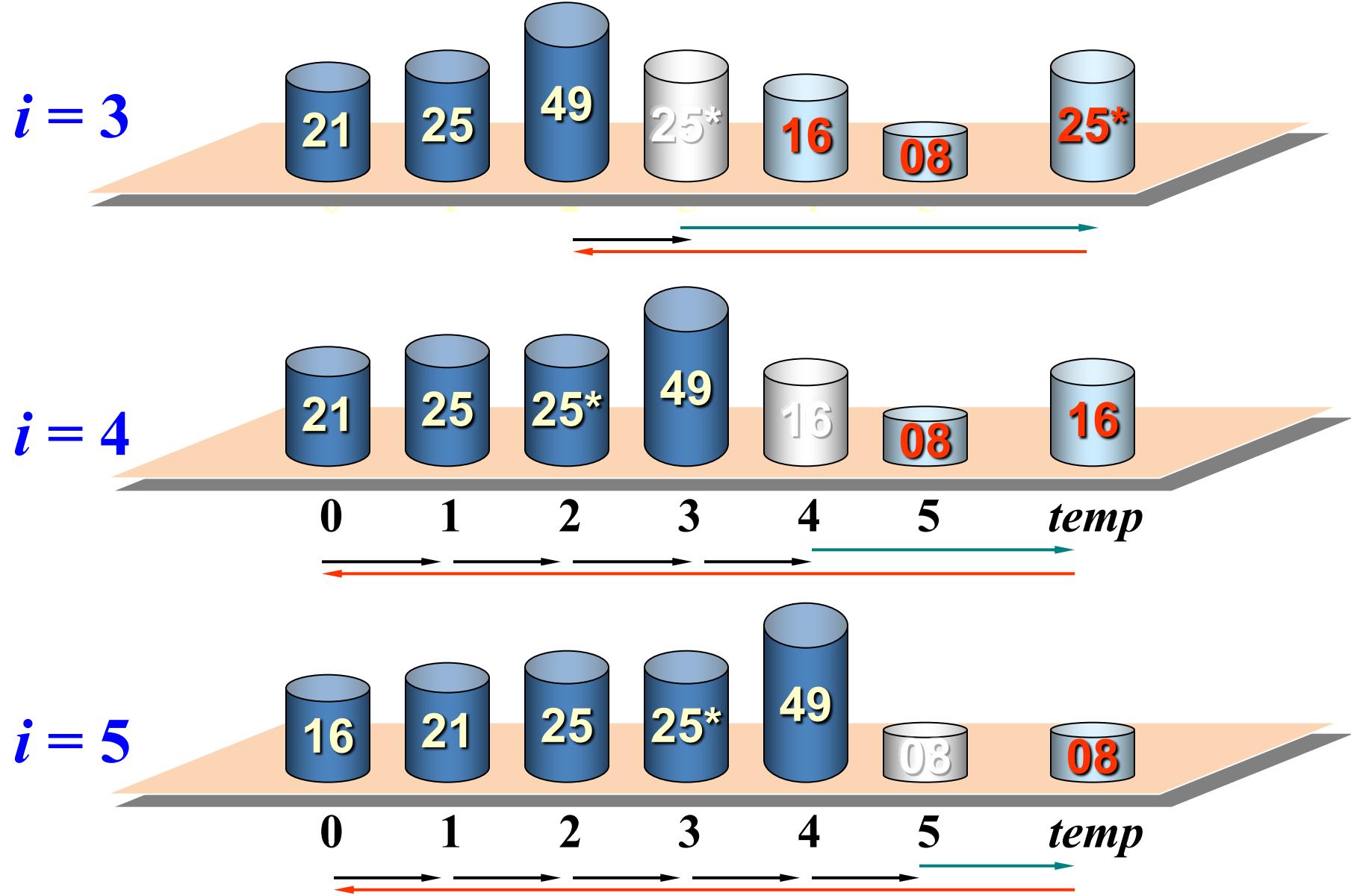
- 基本思想是：当插入第 i ($i \geq 1$) 个元素时，前面的 $V[0], V[1], \dots, V[i-1]$ 已经排好序。这时，用 $V[i]$ 的排序码与 $V[i-1], V[i-2], \dots$ 的排序码顺序进行比较，插入位置即将 $V[i]$ 插入，原来位置上的元素向后顺移。

各趟排序結果

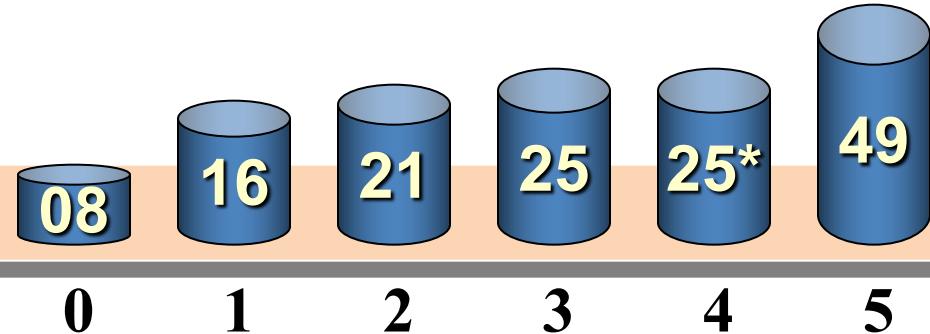
$i = 1$

$i = 2$



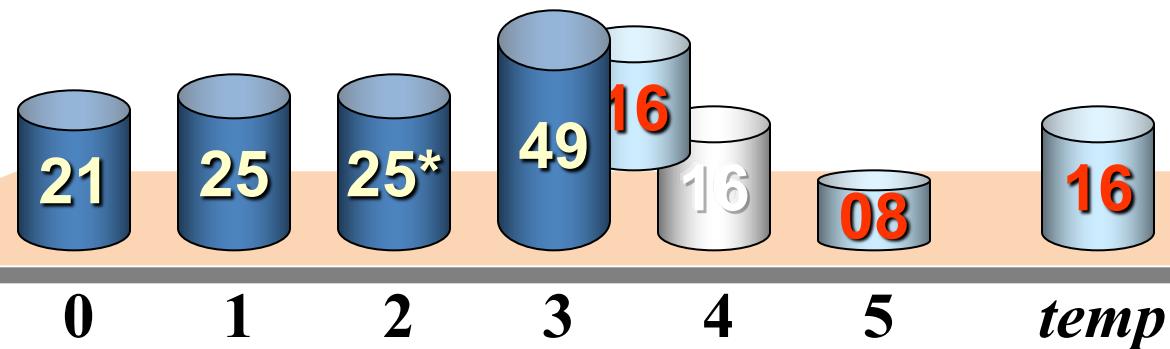


完成

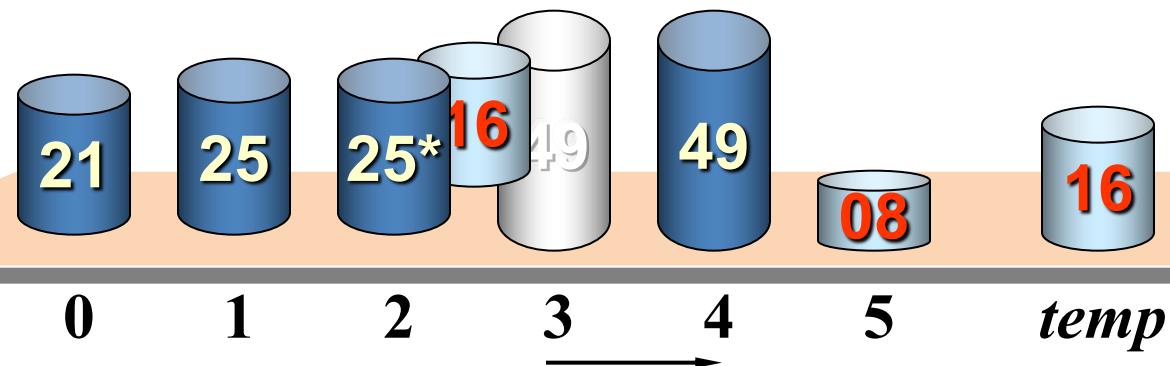


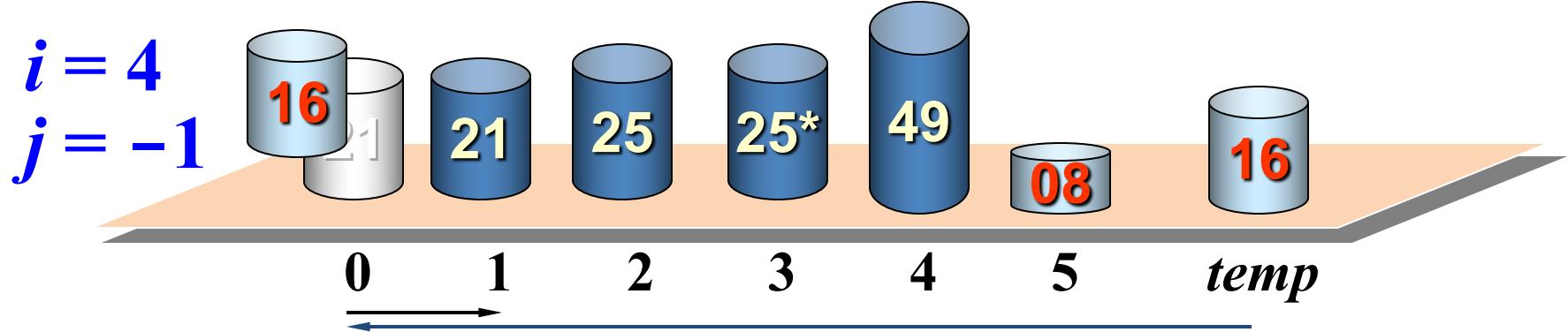
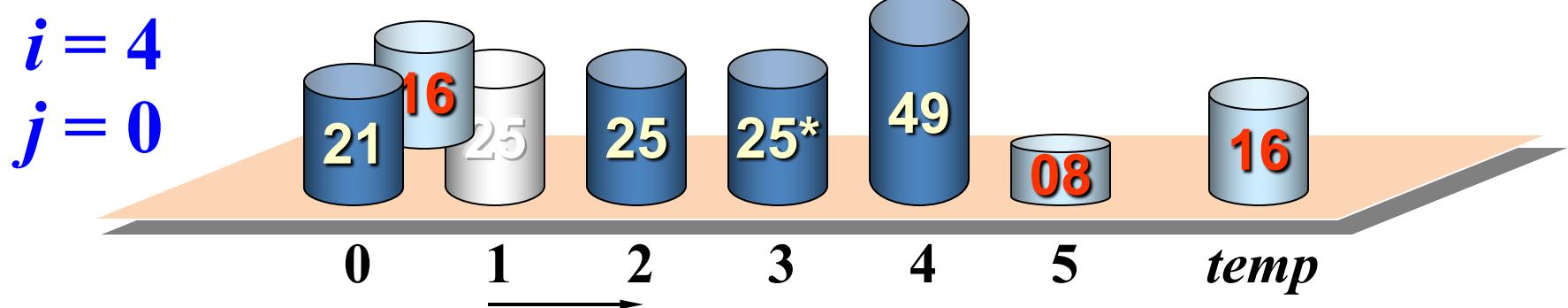
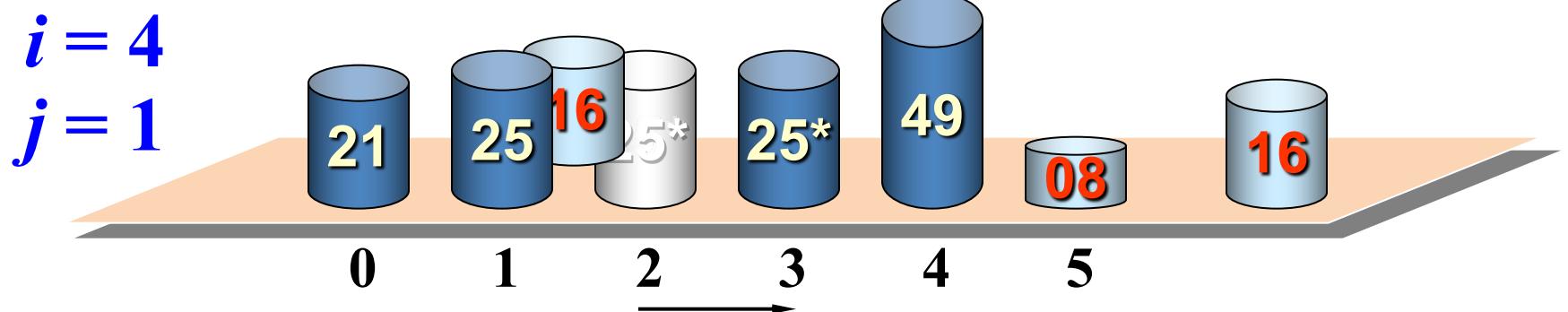
$i = 4$ 时的排序过程

$i = 4$
 $j = 3$



$i = 4$
 $j = 2$







直接插入排序的算法

```
#include "dataList.h"  
template <class T>  
void InsertSort (dataList<T>& L, int left, int right) {  
    //依次将元素L.Vector[i]按其排序码插入到有序表  
    //L.Vector[left],...,L.Vector[i-1]中,使得L.Vector[left]到L.Vector[i]有序。  
    Element<T> temp; int i, j;  
    for (i = left+1; i <= right; i++)  
        if (L[i] < L[i-1]) {  
            temp = L[i]; j = i-1;  
            do {  
                L[j+1] = L[j]; j--;  
            } while (j >= left && temp < L[j]);  
            L[j+1] = temp;  
        }  
};
```



算法分析

- 设待排序元素个数为 $\text{currentSize} = n$, 则该算法的主程序执行 $n-1$ 趟。
- 排序码比较次数和元素移动次数与元素排序码的初始排列有关。
- 最好情况下, 排序前元素已按排序码从小到大有序, 每趟只需与前面有序元素序列的最后一个元素比较1次, 总的排序码比较次数为 $n-1$, 元素移动次数为0。



算法分析

- 最坏情况下, 第 i 趟时第 i 个元素必须与前面 i 个元素都做排序码比较, 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 和元素移动次数 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 $\Theta(n^2)$ 。
- 直接插入排序是一种稳定的排序方法。



折半插入排序 (Binary Insert sort)

- 基本思想是：设在顺序表中有一个元素序列 $V[0], V[1], \dots, V[n-1]$ 。其中， $V[0], V[1], \dots, V[i-1]$ 是已经排好序的元素。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

```
#include "dataList.h"

template <class T>

void BinaryInsertSort (dataList<T>& L,
                      const int left, const int right) {
    //利用折半搜索, 在L.Vector[left]到L.Vector[i-1]中
    //查找L.Vector[i]应插入的位置, 再进行插入。

    Element<T> temp;
    int i, low, high, middle, k;
    for (i = left+1; i <= right; i++) {
        temp = L[i];  low = left;  high = i-1;
        while (low <= high) { //折半搜索插入位置
            middle = (low+high)/2;    //取中点
```



```
if (temp < L[middle])      //插入值小于中点值
    high = middle-1;        //向左缩小区间
else low = middle+1;        //否则, 向右缩小区间
}
for (k = i-1; k >= low; k--) L[k+1] = L[k];
                                //成块移动,空出插入位置
L[low] = temp;    //插入
}
};
```



算法分析

- 折半搜索比顺序搜索快, 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元素序列的初始排列无关, 仅依赖于元素个数。在插入第 i 个元素时, 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 个元素(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。



算法分析

- 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。
- 在元素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。
- 折半插入排序的元素移动次数与直接插入排序相同，依赖于元素的初始排列。



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：

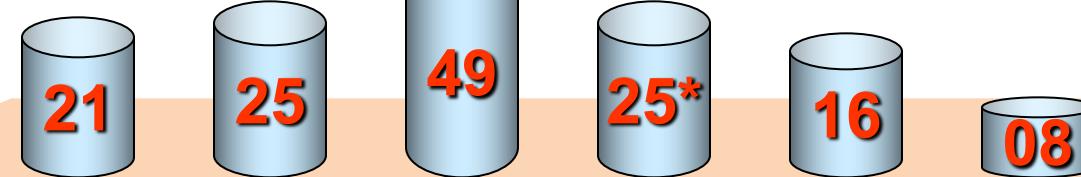
- 1) 设待排序元素序列有 n 个元素，首先取一个整数 $gap < n$ 作为间隔，将全部元素分为 gap 个子序列，所有距离为 gap 的元素放在同一个子序列中。
- 2) 每一个子序列中分别施行直接插入排序。



希尔排序 (Shell Sort)

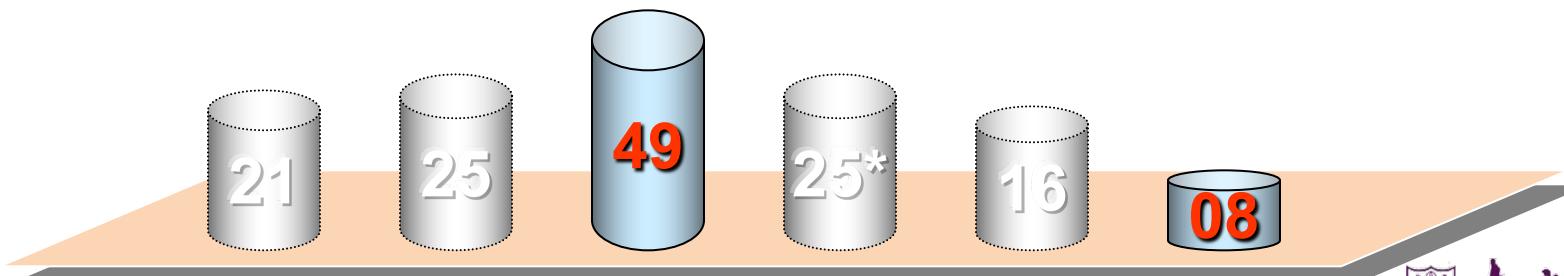
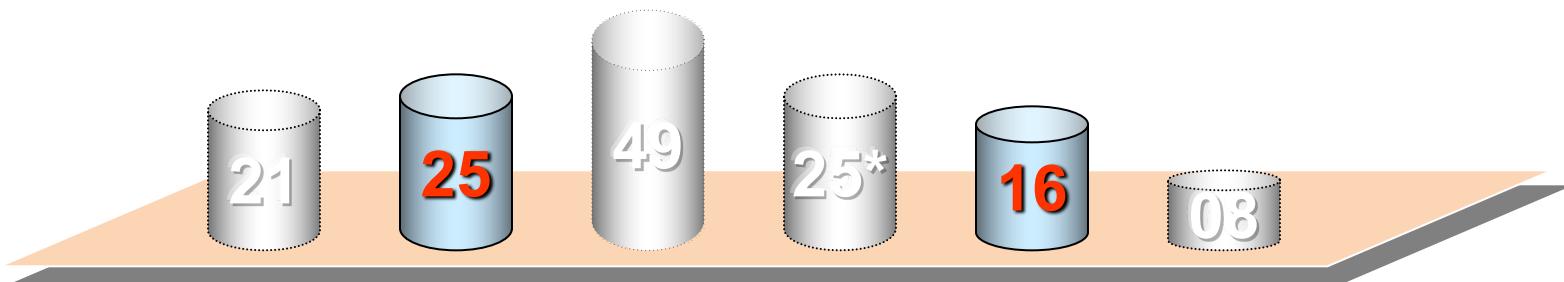
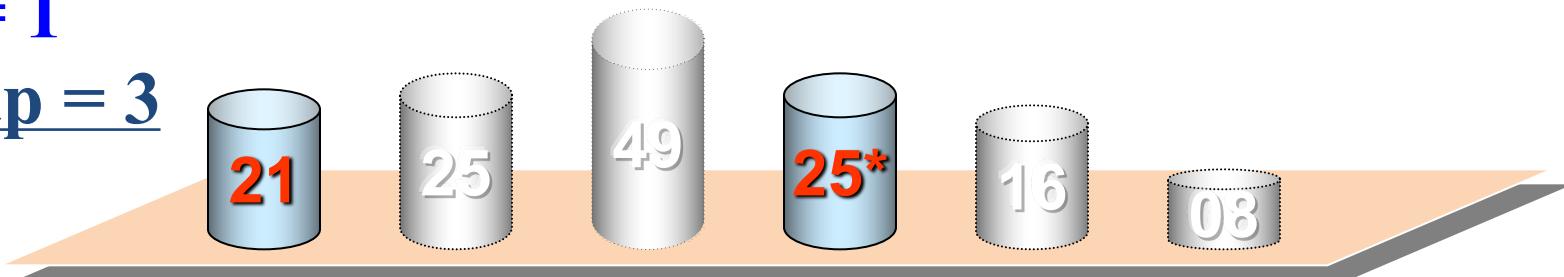
3) 然后缩小间隔 gap , 例如取 $gap = \lceil gap/2 \rceil$, 重复上述的子序列划分和排序工作。直到最后取 $gap == 1$, 将所有元素放在同一个序列中排序为止。

- 开始时 gap 的值较大, 子序列中的元素较少, 排序速度较快; 随着排序进展, gap 值逐渐变小, 子序列中元素个数逐渐变多, 由于前面工作的基础, 大多数元素已基本有序, 所以排序速度仍然很快。



$i = 1$

Gap = 3





0

1

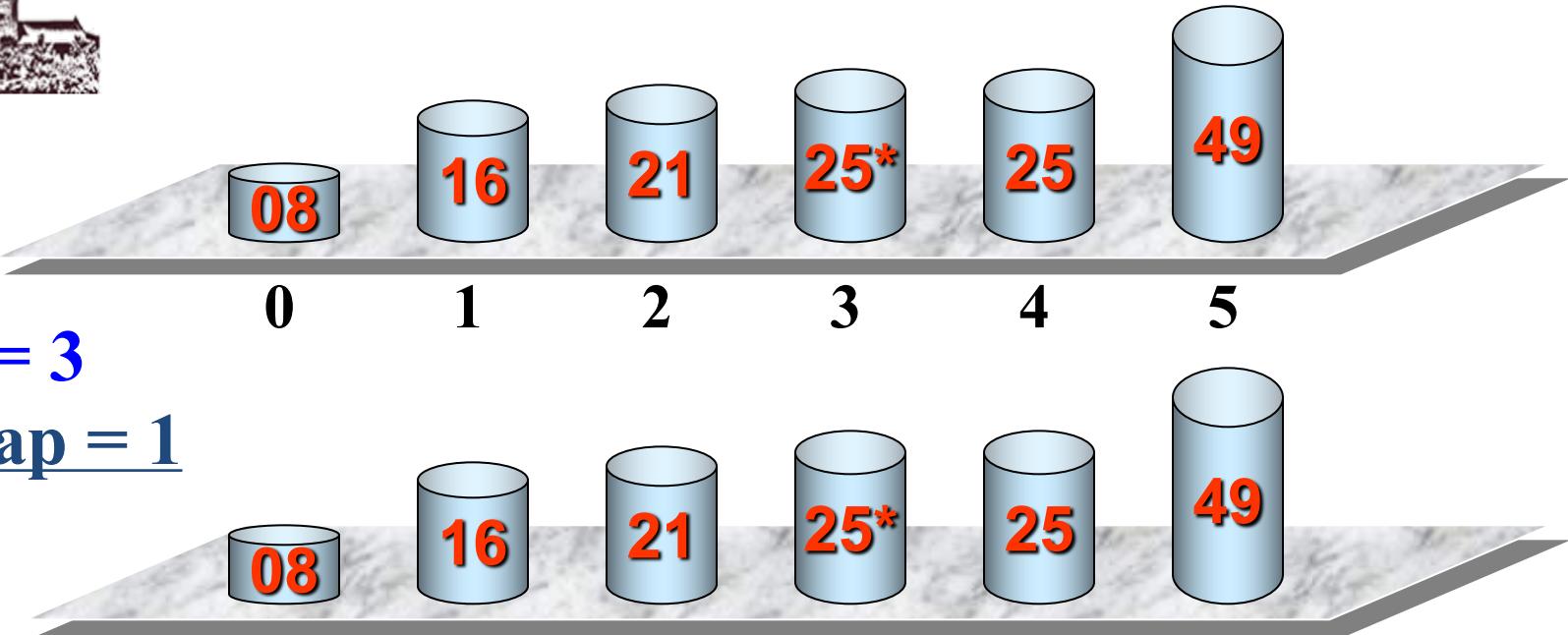
2

3

4

5

 $i = 2$ Gap = 2



希尔排序的算法

```
#include "dataList.h"  
template <class T>
```

```

void Shellsort (dataList<T>& L, const int left, const int right)
{
    int i, j, gap = right-left+1;           //增量的初始值
    Element<T> temp;
    do {
        gap = gap/2;                      //求下一增量值
        for (i = left+gap; i <= right; i++) //组内进行直接插入排序
            if (L[i] < L[i-gap]) {          //逆序
                temp = L[i]; j = i-gap;
                do {
                    L[j+gap] = L[j]; j = j-gap;
                } while (j >= left && temp < L[j]);
                L[j+gap] = temp; //将vector[i]回送
            } // end of if
    } while (gap > 1);
};

```

分组后在组内进行直接插入排序，将gap换成1后就是前面的直接插入排序。



算法分析

- 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。
- 对特定的待排序元素序列，可以准确地估算排序码的比较次数和元素移动次数。



算法分析

- 想要弄清排序码比较次数和元素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元素平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内。这是在利用直接插入排序作为子序列排序方法的情况下得到的。
- 希尔排序是一种不稳定的排序方法。



第九章 排序

- 9.1 概述
- 9.2 插入排序
- 9.3 交换排序
- 9.4 选择排序
- 9.5 归并排序

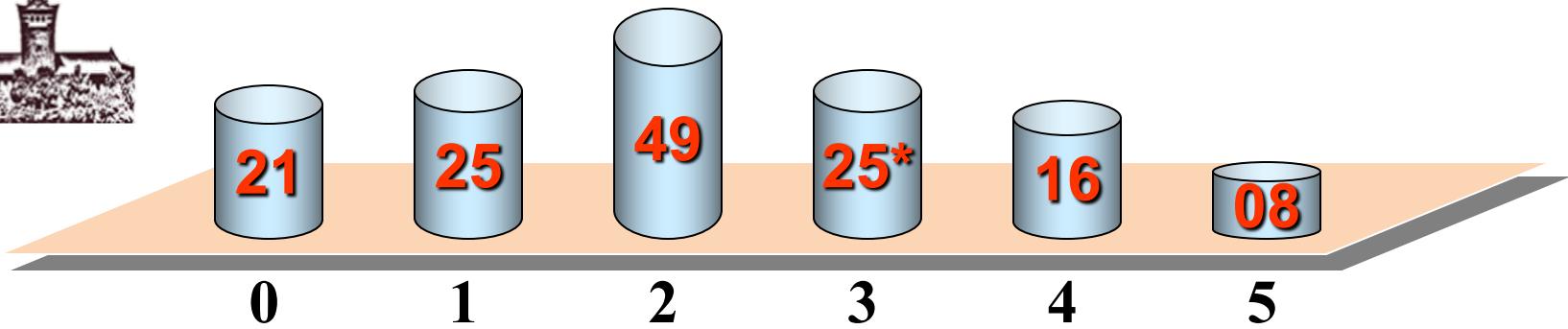


交换排序 (Exchange Sort)

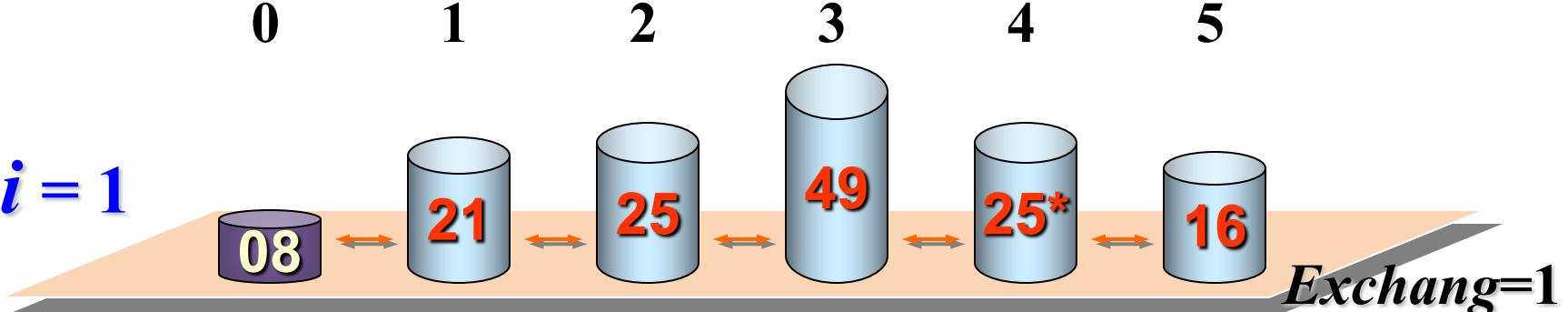
- 基本思想是两两比较待排序元素的排序码，如果发生逆序(即排列顺序与排序后的次序正好相反)，则交换之。直到所有元素都排好序为止。

冒泡排序 (Bubble Sort)

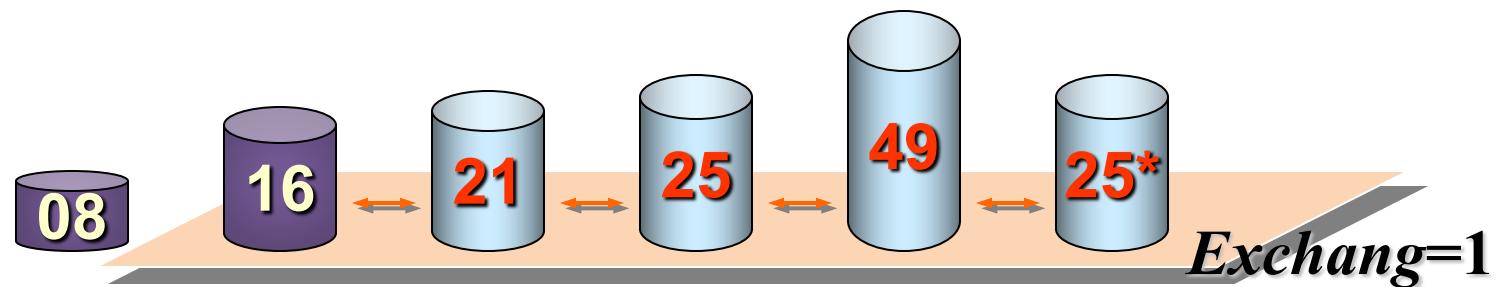
- 基本方法是：设待排序元素序列中的元素个数为 n 。最多作 $n-1$ 趟， $i = 1, 2, \dots, n-1$ 。在第 i 趟中从后向前， $j = n-1, n-2, \dots, i$ ，顺次两两比较 $V[j-1].key$ 和 $V[j].key$ 。如果发生逆序，则交换 $V[j-1]$ 和 $V[j]$ 。



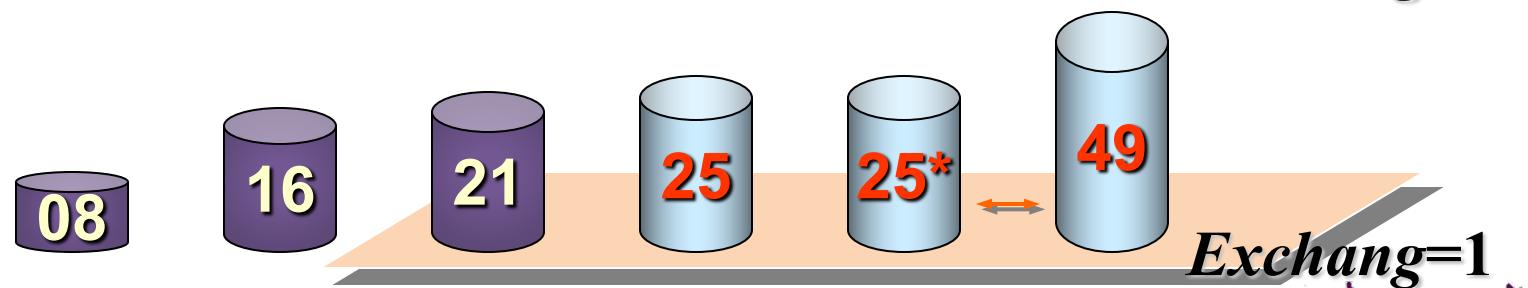
$i = 1$



$i = 2$

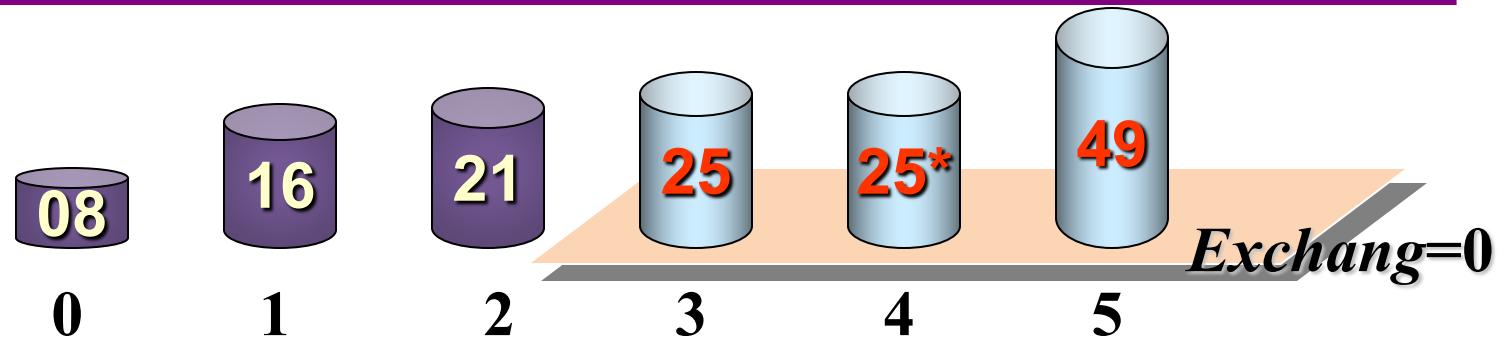


$i = 3$





$i = 4$





起泡排序的算法

```
template <class T>
void BubbleSort (dataList<T>& L, const int left,
                 const int right) {
    int pass = left+1, exchange = 1;
    while (pass <= right && exchange) {
        exchange = 0; //标志为0假定未交换
        for (int j = right; j >= pass; j--)
            if (L[j-1] > L[j]) { //逆序
                Swap (L[j-1], L[j]); //交换
                exchange = 1; //标志置为1,有交换
            }
        pass++;
    } // end of while
};
```



算法分析

- 第 i 趟对待排序元素序列 $V[i-1], V[i], \dots, V[right]$ 进行排序，结果将该序列中排序码最小的元素交换到序列的第一个位置($i-1$)。
- 最多做 $n-1$ 趟起泡就能把所有元素排好序。
- 在元素的初始排列已经按排序码从小到大排好序时，此算法只执行一趟起泡，做 $n-1$ 次排序码比较，不移动元素。这是最好的情形。



算法分析

- 最坏的情形是算法执行 $n-1$ 趟起泡, 第 i 趟 ($1 \leq i < n$) 做 $n-i$ 次排序码比较, 执行 $n-i$ 次元素交换。在最坏情形下总的排序码比较次数 KCN 和元素移动次数 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元素以实现元素值的对换。
- 起泡排序是一个稳定的排序方法。



快速排序 (Quick Sort)

- 基本思想是任取待排序元素序列中的某个元素(例如取第一个元素)作为基准, 按照该元素的排序码大小, 将整个元素序列划分为左右两个子序列:
 - ◆ 左侧子序列中所有元素的排序码都小于或等于基准元素的排序码
 - ◆ 右侧子序列中所有元素的排序码都大于基准元素的排序码



- 基准元素则排在这两个子序列中间(这也是该元素最终应安放的位置)。
- 然后分别对这两个子序列重复施行上述方法，直到所有的元素都排在相应位置上为止。



算法描述

```
QuickSort ( List ) {
```

```
    if ( List 的长度大于 1 ) {
```

Partition: 将序列 List 划分为两个子序列

LeftList 和 RightList;

```
        QuickSort ( LeftList );
```

```
        QuickSort ( RightList );
```

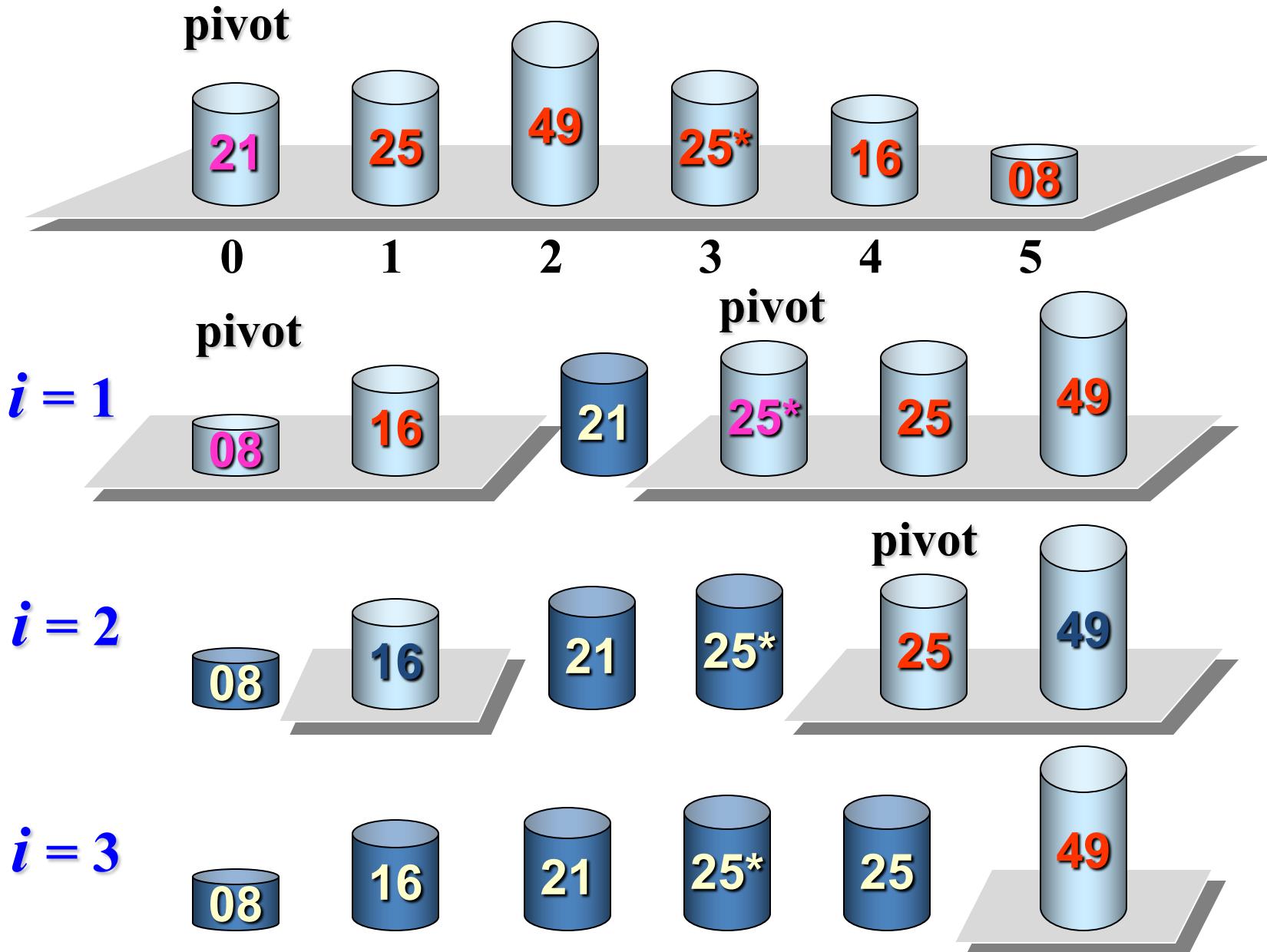
将两个子序列 LeftList 和 RightList

合并为一个序列 List;

```
}
```

```
}
```

快速排序算法示意



$i = 1$
划分

pivotpos



0

1

2

3

4

5

pivotpos



比较4次
交换25,16

i

pivotpos



比较1次
交换49,08

low

pivotpos



交换21,08



快速排序的算法

```
#include "dataList.h"

template <class T>
void QuickSort (dataList<T>& L,
                 const int left, const int right) {
    //对元素Vector[left], ..., Vector[right]进行排序,
    //pivot=L.Vector[left]是基准元素, 排序结束后它的
    //位置在pivotPos, 把参加排序的序列分成两部分,
    //左边元素的排序码都小于或等于它, 右边都大于它
    if (left < right) {           //元素序列长度大于1时
        int pivotpos = L.Partition (left, right); //划分
        QuickSort (L, left, pivotpos-1);
    }
}
```



```
    QuickSort (L, pivotpos+1, right);  
}  
};
```

```
template <class T>  
int dataList<T>::Partition (const int low, const int high) {  
    //数据表类的共有函数  
    int pivotpos = low;  
    Element<T> pivot = Vector[low];           //基准元素  
    for (int i = low+1; i <= high; i++)  
        //检测整个序列, 进行划分
```

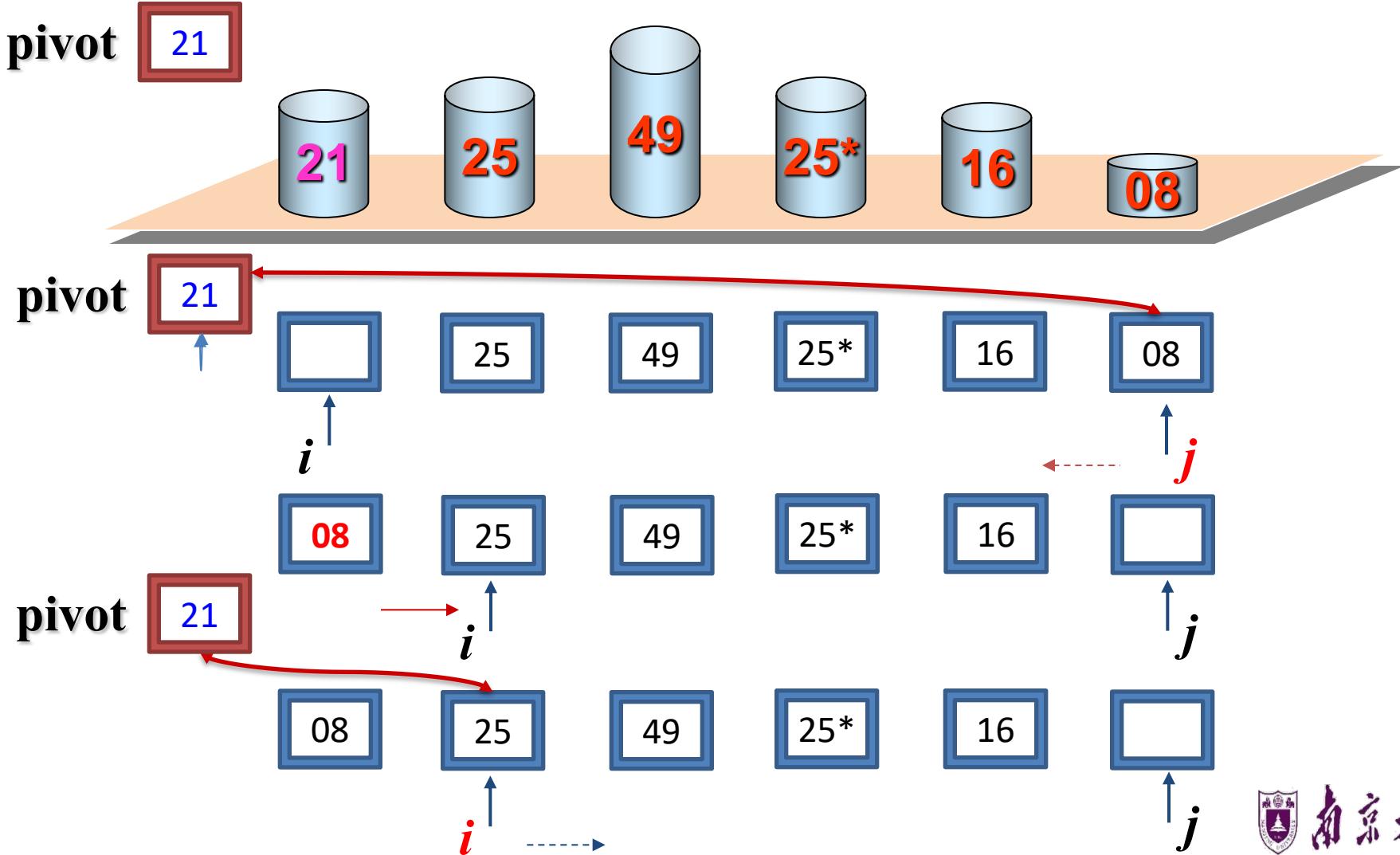


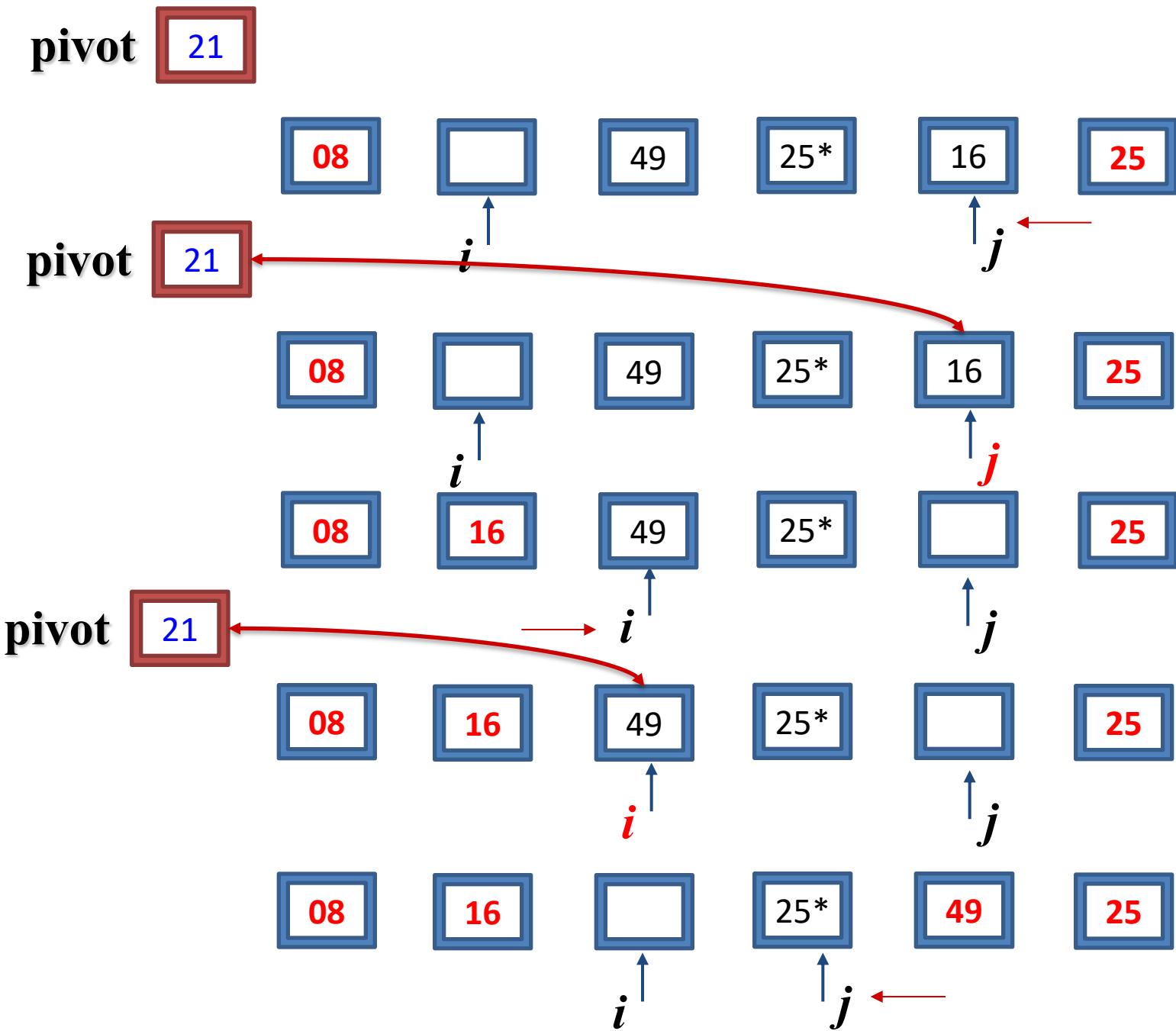
```
if (Vector[i] < pivot) {  
    pivotpos++;  
    if (pivotpos != i)  
        Swap(Vector[pivotpos], Vector[i]);  
}  
                                //小于基准的交换到左侧去  
Vector[low] = Vector[pivotpos];  
Vector[pivotpos] = pivot;  
                                //将基准元素就位  
return pivotpos; //返回基准元素位置  
};
```

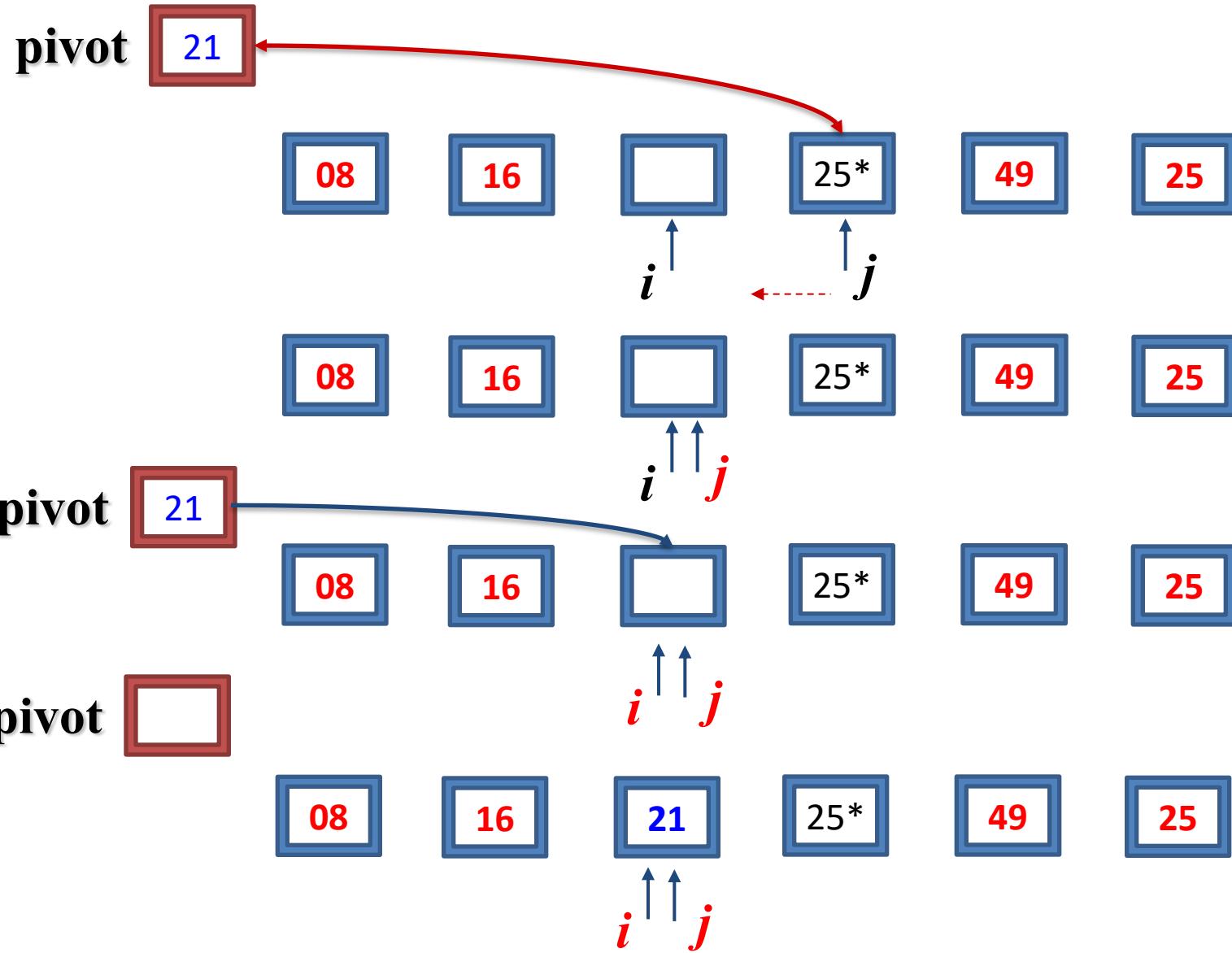


Partition补充算法

Pivot=21的一趟划分









Partition补充算法

```
int dataList<T>::Partition_Classic (int s, int t) {  
    int pivot = a[s]; i=s; j=t;  
    while (i < j)  
    {  
        while ((i < j) &&(a[j] >= pivot)) {j=j-1; }  
        if (i < j) {a[i]=a[j]; i=i+1;}  
  
        while ((i < j) &&(a[i] <= pivot)) {i=i+1; }  
        if (i < j) {a[j]=a[i]; j=j-1;}  
    }  
    a[i] = pivot; return i; //此时i==j  
}
```



单向 vs 双向

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 9 | 2 | 1 | 4 | 7 | 5 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 9 | 1 | 4 | 7 | 5 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 1 | 9 | 4 | 7 | 5 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 1 | 4 | 9 | 7 | 5 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 1 | 4 | 5 | 7 | 9 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 2 | 1 | 4 | 5 | 3 | 9 | 8 | 7 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 4 | 5 | 5 | 9 | 8 | 7 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | 9 | 2 | 1 | 4 | 7 | 5 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 2 | 1 | 4 | 7 | 5 | 8 | 3 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 9 | 2 | 1 | 4 | 7 | 5 | 8 | 9 | 6 |
|---|---|---|---|---|---|---|---|---|---|

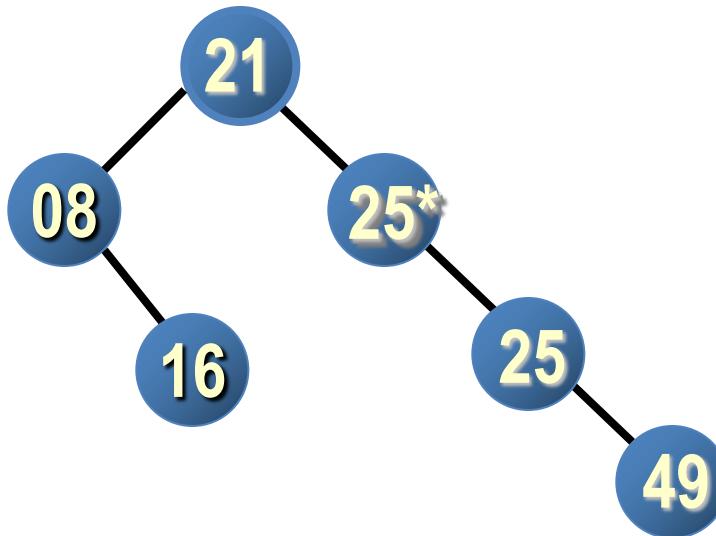
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 2 | 1 | 4 | 7 | 5 | 8 | 9 | 6 |
|---|---|---|---|---|---|---|---|---|---|

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 4 | 2 | 1 | 5 | 7 | 5 | 8 | 9 | 6 |
|---|---|---|---|---|---|---|---|---|---|



算法分析

- 算法quicksort是一个递归的算法，其递归树如图所示。



- 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。



算法分析

- 最大递归调用层数与递归树高度一致，理想情况为 $\lceil \log_2(n+1) \rceil$ 。存储开销为 $O(\log_2 n)$ 。
- 在最坏的情况下，即待排序元素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元素的子序列。必须经过 $n-1$ 趟才能把所有元素定位，而且第 i 趟需要经过 $n-i$ 次排序码比较才能找到第 i 个元素的安放位置，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

用第一个元素作为基准元素
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 $O(n)$ 。
- 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。
- 因此，当 n 很小时可以用直接插入排序方法。
- 随机化的快速排序可以时间复杂性到达 $O(n \log n)$ 。



- **Randomized QuickSort**
- 对于每个待排序的数组(子数组), 在**SPLIT**前, 先执行下面的步骤1和2。

| | | | | | | | | | | |
|---|---|---|---|---|----|----|----|----|----------|----|
| A | 2 | 3 | 6 | 7 | 10 | 11 | 18 | 19 | 20 | 21 |
| | | | | | | | | | <i>r</i> | |

1. Generate a random integer $r \in [1, n]$
2. Exchange $A[1]$ with $A[r]$