第十章 内部排序

机电工程与自动化学院 L栋301

任卫红 助理教授

renweihong@hit.edu.cn

http://faculty.hitsz.edu.cn/renweihong

10.1 排序

第一节　排序

一、排序(Sorting)

- **排序：**将一个数据元素（或记录）的任意序列，重新排列成一个按**关键字有序的序列**
- **内部排序：**在排序期间数据对象全部存放在内存的排序；
- **外部排序：**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**。

第一节　排序

二、排序基本操作

排序的基本操作包括：

- **比较**：比较两个关键字的大小
- **移动**：将记录从一个位置移动至另一个位置

第一节　排序

三、排序时间复杂度

- 排序的时间复杂度可用算法执行中的记录关键字比较次数与记录移动次数来衡量。
- 算法执行时所需的附加存储：评价算法好坏的另一标准。

第一节　排序

四、排序方法的稳定性

- 如果在记录序列中有两个记录r[i]和r[j]，它们的关键字 key[i] == key[j]，且在排序之前，记录r[i]排在r[j]前面。
- 如果在排序之后，记录r[i]仍在记录r[j]的前面，则称这个排序方法是稳定的，
- 否则称这个排序方法是不稳定的。

例如：

排序前 (56, 34, **47**, 23, 66, 18, 82, **47**)

若排序后得到结果

(18, 23, 34, **47**, **47**, 56, 66, 82)

则称该排序方法是稳定的；

若排序后得到结果

(18, 23, 34, **47**, **47**, 56, 66, 82)

则称该排序方法是不稳定的。

10.2 插入排序

第二节　插入排序

- 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
  - 折半插入排序
  - 希尔排序

第二节　插入排序

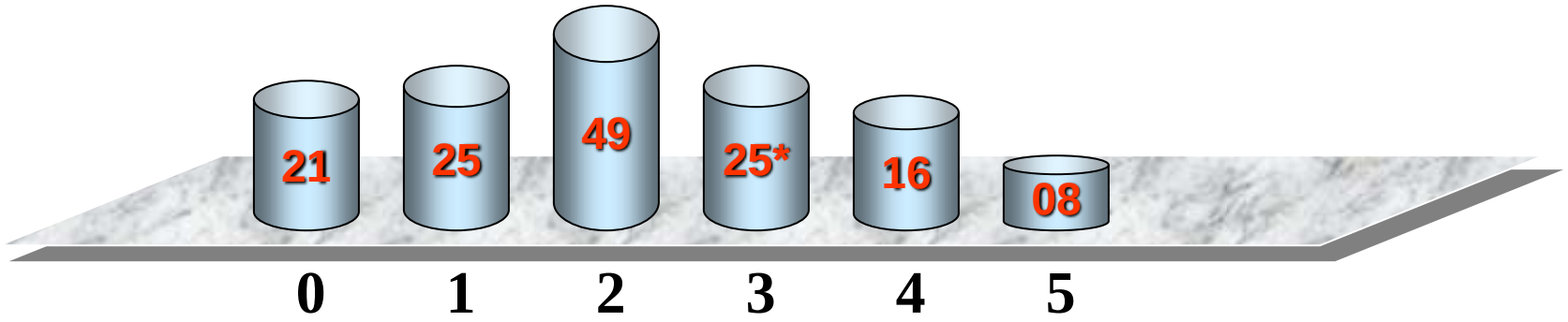
一、直接插入排序

- 当插入第i（i≥1）个对象时，前面的r[0]，r[1]，…，r[i-1]已经排好序。
- 用r[i]的关键字与r[i-1]，r[i-2]，…的关键字顺序进行比较(和顺序查找类似)，如果小于，则将r[x]向后移动(插入位置后的记录向后顺移)
- 找到插入位置即将r[i]插入

第二节　插入排序

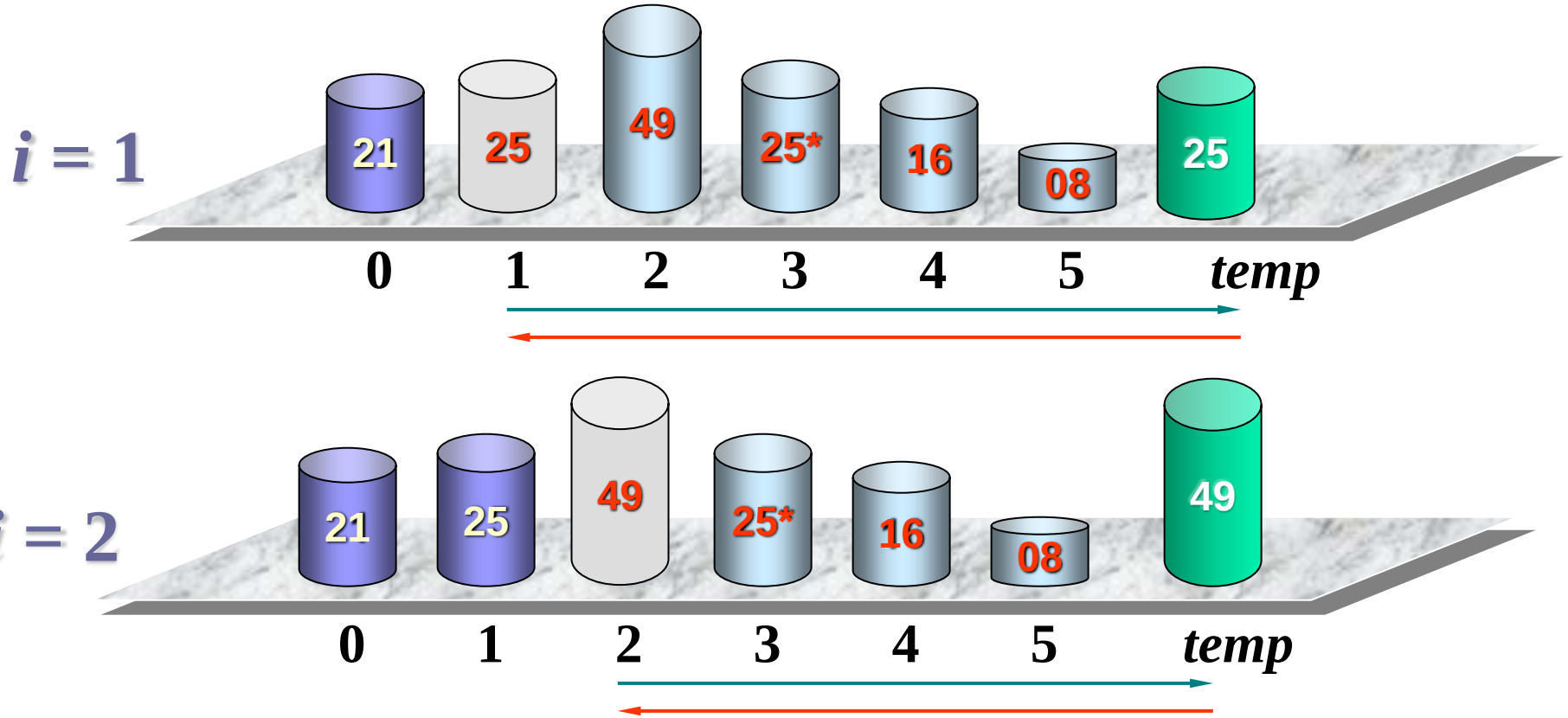
一、直接插入排序(举例)

- 已知待序的一组记录的初始排列为：21，25，49，25*，16，08



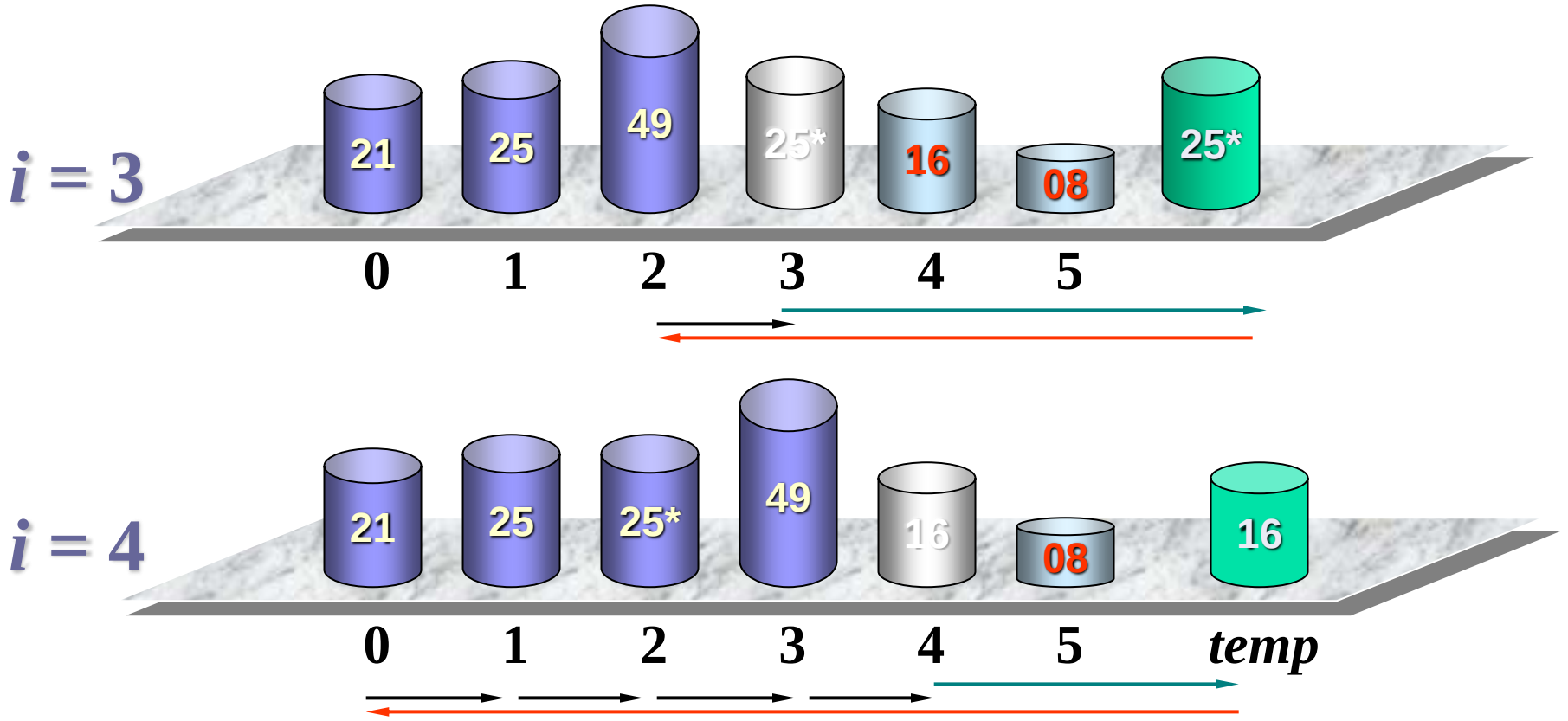
第二节　插入排序

一、直接插入排序(举例)



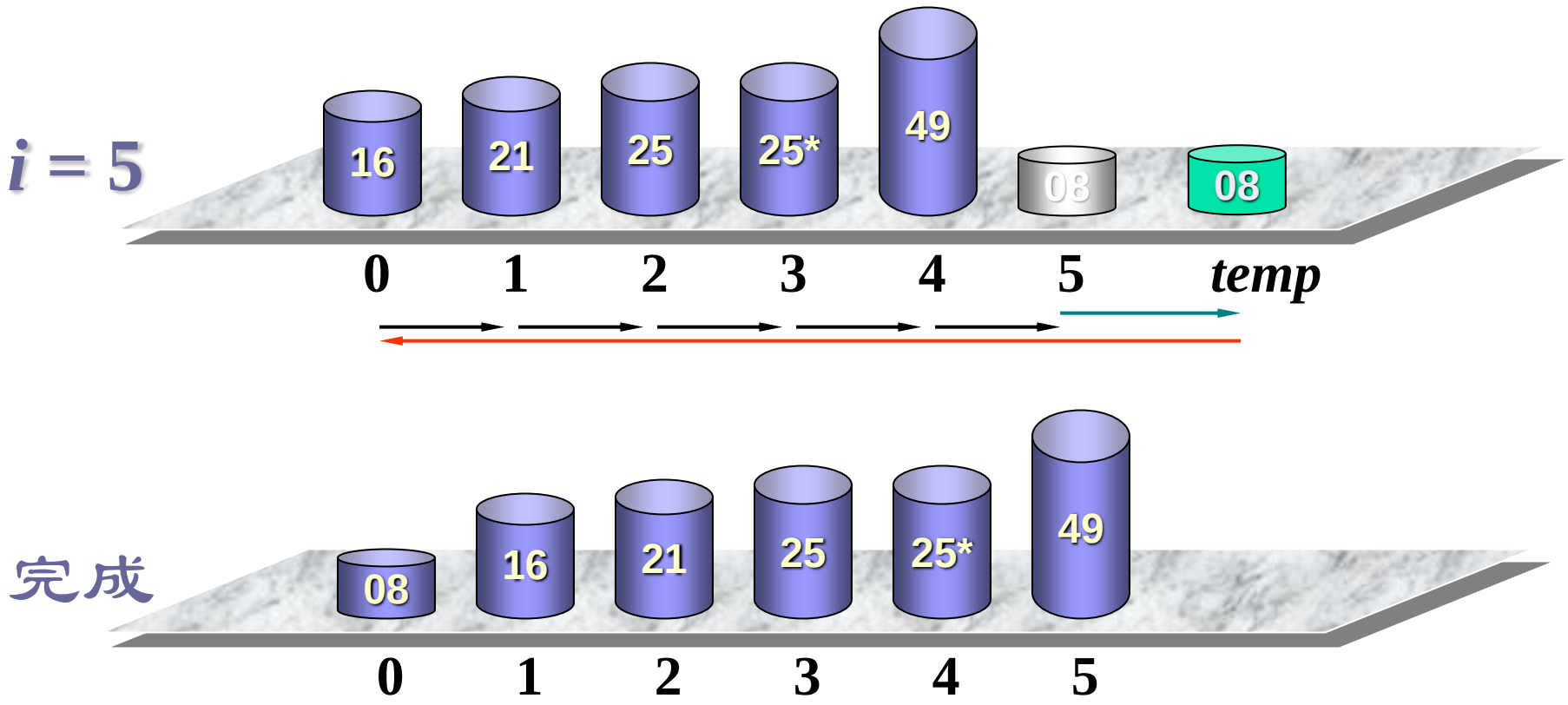
第二节　插入排序

一、直接插入排序(举例)



第二节　插入排序

一、直接插入排序(举例)



第二节　插入排序

一、直接插入排序(算法实现)

```
void InsertSort (int r[ ], int m ) {
   // 假设关键字为整型，放在向量r[]中
     int n, j, temp;
     for ( n = 1; n < m; n++ ) {
         temp = r[n];
         for ( j = n;  j > 0; j-- )  //从后向前顺序比较，并依次后移
             if ( temp < r[j-1] ) r[j] = r[j-1];
             else break;
         r[j] = temp;
     }
}
```

第二节　插入排序

一、直接插入排序(算法分析)

- 关键字比较次数和记录移动次数与记录关键字的初始排列有关。

- 最好情况下，排序前记录已按关键字从小到大有序，每趟只需与前面有序记录序列的最后一个记录比较1次，移动2次记录，总的关键字比较次数为 n-1，记录移动次数为 2(n-1)。

第二节　插入排序

一、直接插入排序(算法分析)

- **最坏情况下**，（i从1开始，下标从0开始）第i趟时第i个记录必须与前面i个记录都做关键字比较，并且每做**1次比较**就要**做1次数据移动**。则总关键字比较次数KCN和记录移动次数RMN分别为

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

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

第二节　插入排序

一、直接插入排序(算法分析)

- 在平均情况下的关键字比较次数和记录移动次数约为 $n^2/4$。

- 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
- 直接插入排序最大的优点是简单，在记录数较少时，是比较好的办法

第二节　插入排序

二、折半插入排序

- 折半插入排序在查找记录插入位置时，采用折半查找算法
- 折半查找比顺序查找快，所以折半插入排序在查找上性能比直接插入排序好
- 但需要移动的记录数目与直接插入排序相同(为$O(n^2)$)
- 折半插入排序的时间复杂度为$O(n^2)$。
- 折半插入排序是一种稳定的排序方法

第二节　插入排序

三、希尔排序

- 从直接插入排序可以看出，当待排序列为正序时，时间复杂度为O(n)
- 若待排序列基本有序时，插入排序效率会提高
- 希尔排序方法是先将待排序列分成若干子序列分别进行插入排序，待整个序列基本有序时，再对全体记录进行一次直接插入排序
- 希尔排序又称为**缩小增量排序**。

第二节　插入排序

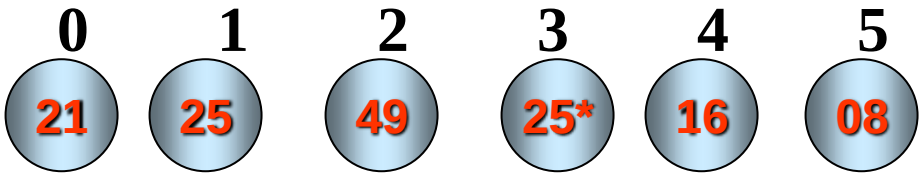
三、希尔排序(算法)

- 首先取一个整数 gap < n(待排序记录数) 作为间隔，将全部记录分为 gap 个子序列，所有距离为 gap 的记录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 = gap/2
- 重复上述的子序列划分和排序工作，直到最后取 gap = 1，将所有记录放在同一个序列中排序为止。

第二节　插入排序

三、希尔排序(举例)

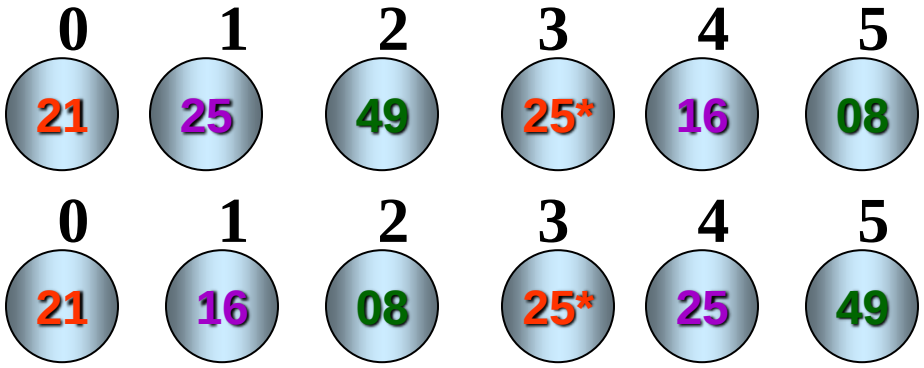
- 已知待序的一组记录的初始排列为：21，25，49，25*，16，08



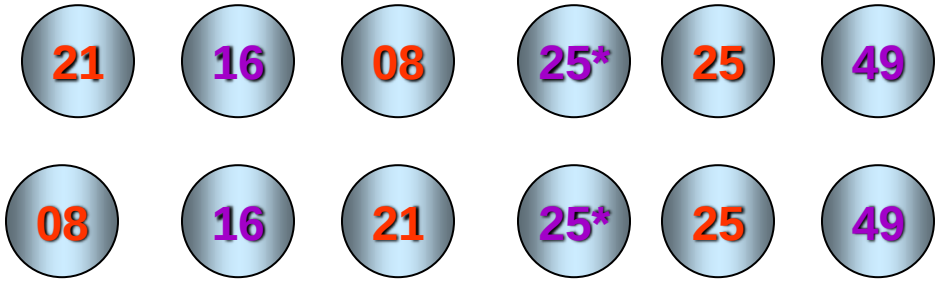
第二节　插入排序

三、希尔排序(举例)

Gap = 3



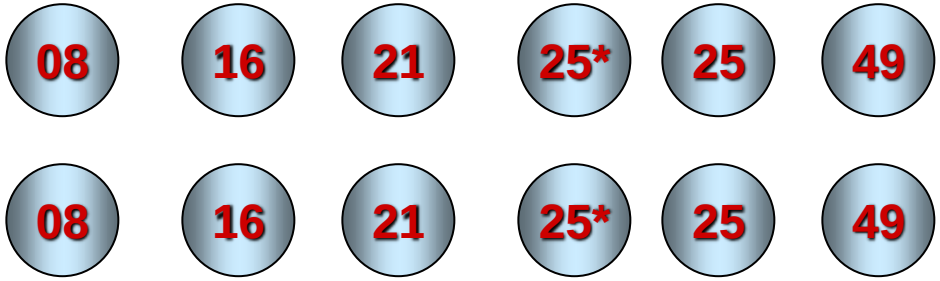
Gap = 2



第二节　插入排序

三、希尔排序(举例)

Gap = 1



第二节　插入排序

三、希尔排序(算法实现思路)

- Shellinsert(int gap)：修改直接插入排序算法，使其完成间隔为gap的元素之间进行比较的插入排序
  - 例：若，gap=3，则同时完成三个子序列的直接插入排序，且这三个子序列间的元素的间隔为3）
  - 例：若，gap=1，则1个子序列的直接插入排序，且这三个子序列间的元素的间隔为1）
- shellsort:主调函数，依次调用上面的shellinsert,gap取值依次减小，直至1.

第二节　插入排序

三、希尔排序(算法分析)

- 开始时 gap 的值较大，子序列中的记录较少，排序速度较快
- 随着排序进展，gap 值逐渐变小，子序列中记录个数逐渐变多，由于前面大多数记录已基本有序，所以排序速度仍然很快。
- Gap的取法有多种。 shell 提出取 gap = ⌊n/2⌋，gap = ⌊gap/2⌋，直到gap = 1。

第二节　插入排序

三、希尔排序(算法分析)

- 对特定的待排序记录序列，可以准确地估算关键字的比较次数和记录移动次数。
- 希尔排序所需的比较次数和移动次数约为$n^{1.3}$
- 当n趋于无穷时可减少到$nx(\log_2 n)^2$
- 希尔排序的时间复杂度约为$0(n\ x(\log_2 n)^2)$
- 希尔排序是一种不稳定的排序方法

10.3 快速排序

第三节　快速排序

一、起泡排序

- 设待排序记录序列中的记录个数为n（下标从1到n）。
- 一般地，第i趟起泡排序从1到n-i+1
- 依次比较相邻两个记录的关键字，如果发生逆序，则交换之
- 其结果是这n-i+1个记录中，关键字最大的记录被交换到第n-i+1的位置上，最多作n-1趟。

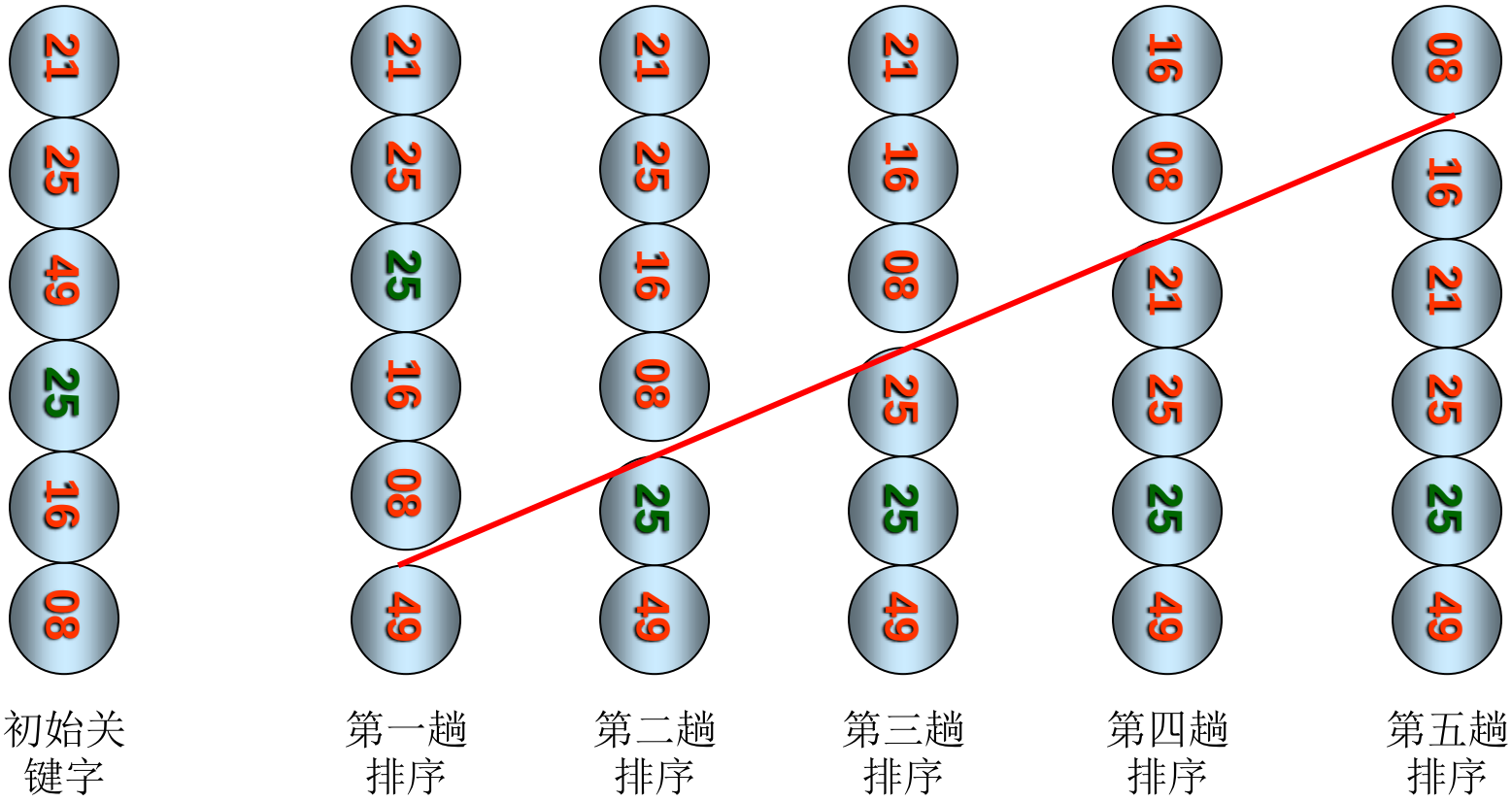
第三节　快速排序

一、起泡排序

- i=1时，为第一趟排序，关键字最大的记录将被交换到最后一个位置
- i=2时，为第二趟排序，关键字次大的记录将被交换到最后第二个位置
- ：
- 关键字小的记录不断上浮(起泡)，关键字大的记录不断下沉(每趟排序最大的一直沉到底)

第三节　快速排序

一、起泡排序



第三节　快速排序

一、起泡排序(性能分析)

- 最好情况：在记录的初始排列已经按关键字从小到大排好序时，此算法只执行一趟起泡，做n-1次关键字比较，不移动记录

第三节　快速排序

一、起泡排序(性能分析)

- **最坏情况**：执行n-1趟起泡，第i趟做n-i次关键字比较，执行n-i次记录交换，共计：

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

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

- 起泡排序的时间复杂度为$O(n^2)$
- 起泡排序是一种**稳定**的排序方法

第三节　快速排序

二、快速排序

- 任取待排序记录序列中的某个记录(例如取第一个记录)作为基准(枢),按照该记录的关键字大小,将整个记录序列划分为左右两个子序列:
- 左侧子序列中所有记录的关键字都小于或等于基准记录的关键字
- 右侧子序列中所有记录的关键字都大于基准记录的关键字

第三节　快速排序

二、快速排序

- 基准记录则排在这两个子序列中间(这也是该记录最终应安放的位置)。
- 然后分别对这两个子序列重复施行上述方法，直到所有的记录都排在相应位置上为止。
- 基准记录也称为**枢轴（或支点）记录**。

第三节　快速排序

二、快速排序(算法)

- 取序列第一个记录为枢轴记录，其关键字为Pivotkey
- 指针low指向序列第一个记录位置
- 指针high指向序列最后一个记录位置

第三节　快速排序

二、快速排序(算法)

- 一趟排序(某个子序列)过程

1. 从high指向的记录开始，向前找到第一个关键字的值小于Pivotkey的记录，将其放到low指向的位置，low+1
2. 从low指向的记录开始，向后找到第一个关键字的值大于Pivotkey的记录，将其放到high指向的位置，high-1
3. 重复1，2，直到low=high，将枢轴记录放在low(high)指向的位置

第三节　快速排序

二、快速排序(算法)

- 对枢轴记录前后两个子序列执行相同的操作，直到每个子序列都只有一个记录为止

第三节　快速排序

二、快速排序(举例)

pivotkey

21

初始关键字

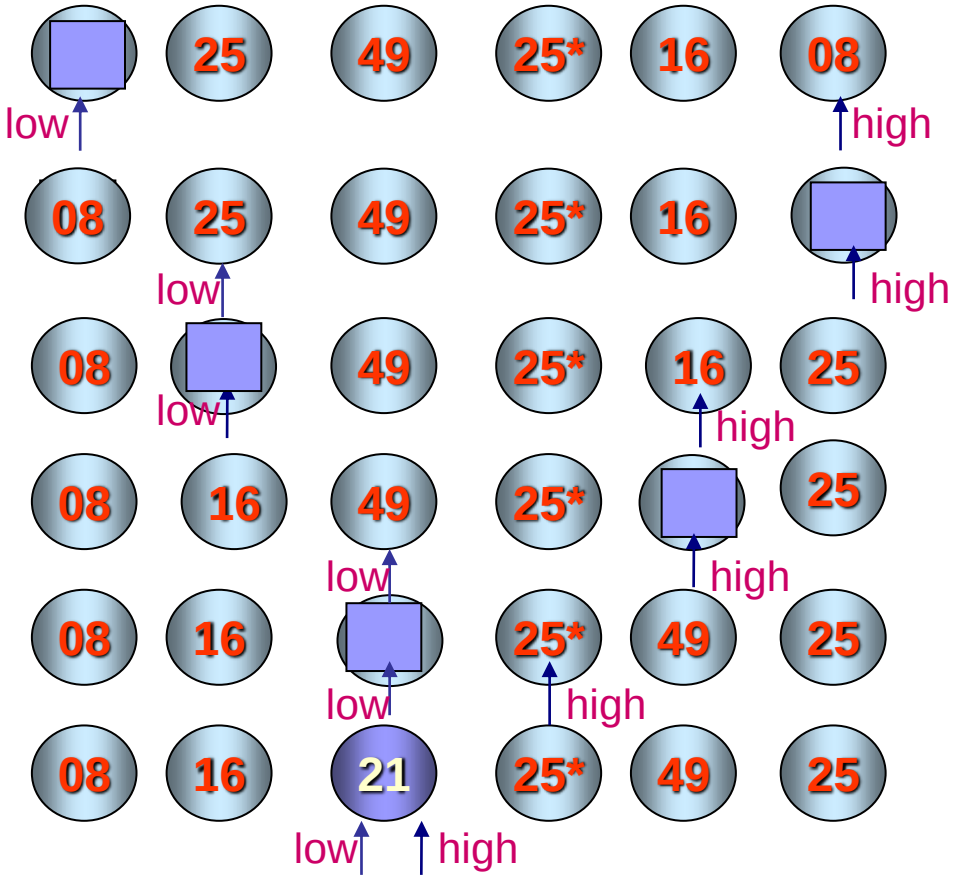
一次交换

二次交换

三次交换

high-1

完成一趟排序



第三节　快速排序

二、快速排序(举例)

- 绿色表示到位
- 蓝色表示枢轴

完成一趟排序

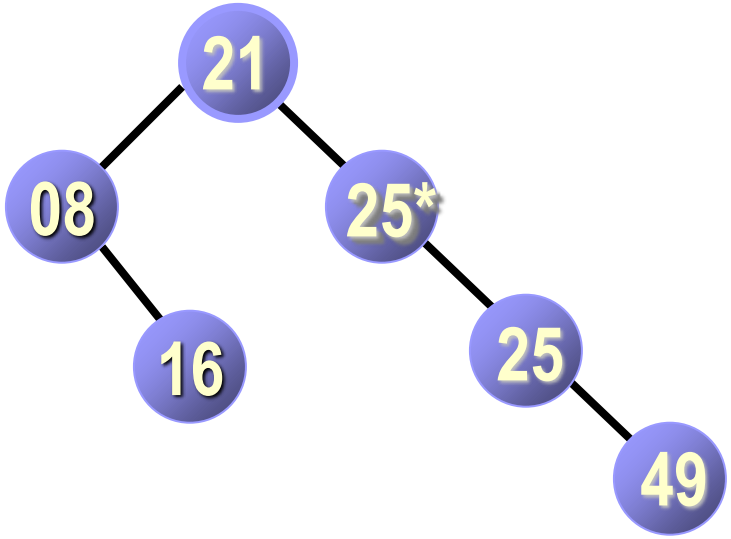
分别进行快速排序

有序序列

第三节　快速排序

二、快速排序(性能分析)

- 快速排序是一个递归过程，其递归树如图所示
- 利用序列第一个记录作为基准，将整个序列划分为左右两个子序列。只要是关键字小于基准记录关键字的记录都移到序列左侧



第三节　快速排序

二、快速排序(性能分析)

- 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记录定位后，该记录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。

第三节　快速排序

二、快速排序(性能分析)

- 在 n个元素的序列中,对一个记录定位所需时间为O(n)。若设 t(n) 是对 n 个元素的序列进行排序所需的时间，而且每次对一个记录正确定位后，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：

$T(n) \le cn + 2T(n/2)$　　// c 是一个常数

$\le cn + 2(cn/2 + 2T(n/4)) = 2cn + 4T(n/4)$

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

……

$\le cn \log_2 n + nT(1) = O(n \log_2 n)$

第三节　快速排序

二、快速排序(性能分析)

- 快速排序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。

- 但快速排序是一种不稳定的排序方法

第三节　快速排序

二、快速排序(性能分析)

- 在最坏情况下，即待排序记录序列已经按其关键字从小到大排好序，其递归树成为单支树，时间复杂度达$O(n^2)$
- 每次划分只得到一个比上一次少一个记录的子序列。
- 必须经过n-1 趟才能把所有记录定位，
- 而且第 i 趟需要经过 n-i 次关键字比较才能找到第 i 个记录的安放位置，总的关键字比较次数将达到

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

第三节　快速排序

二、快速排序(改进)

- 枢轴记录取low、high、(low+high)/2三者指向记录关键字居中的记录

10.4 选择排序

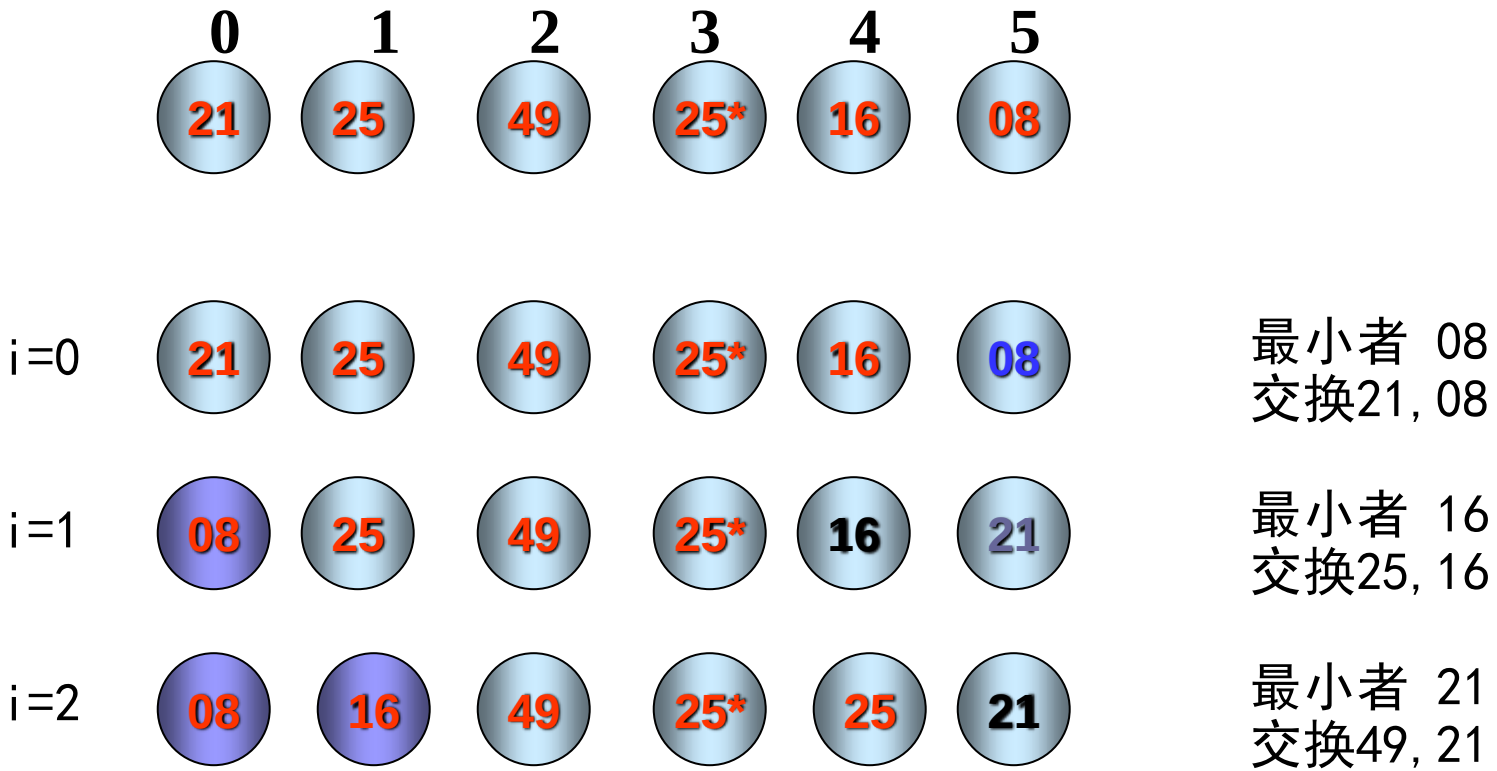
第四节　选择排序

一、简单选择排序

- 每一趟(例如第i趟, i=0, 1, …, n-2)在后面n-i个待排序记录中选出关键字最小的记录, 与第i个记录交换

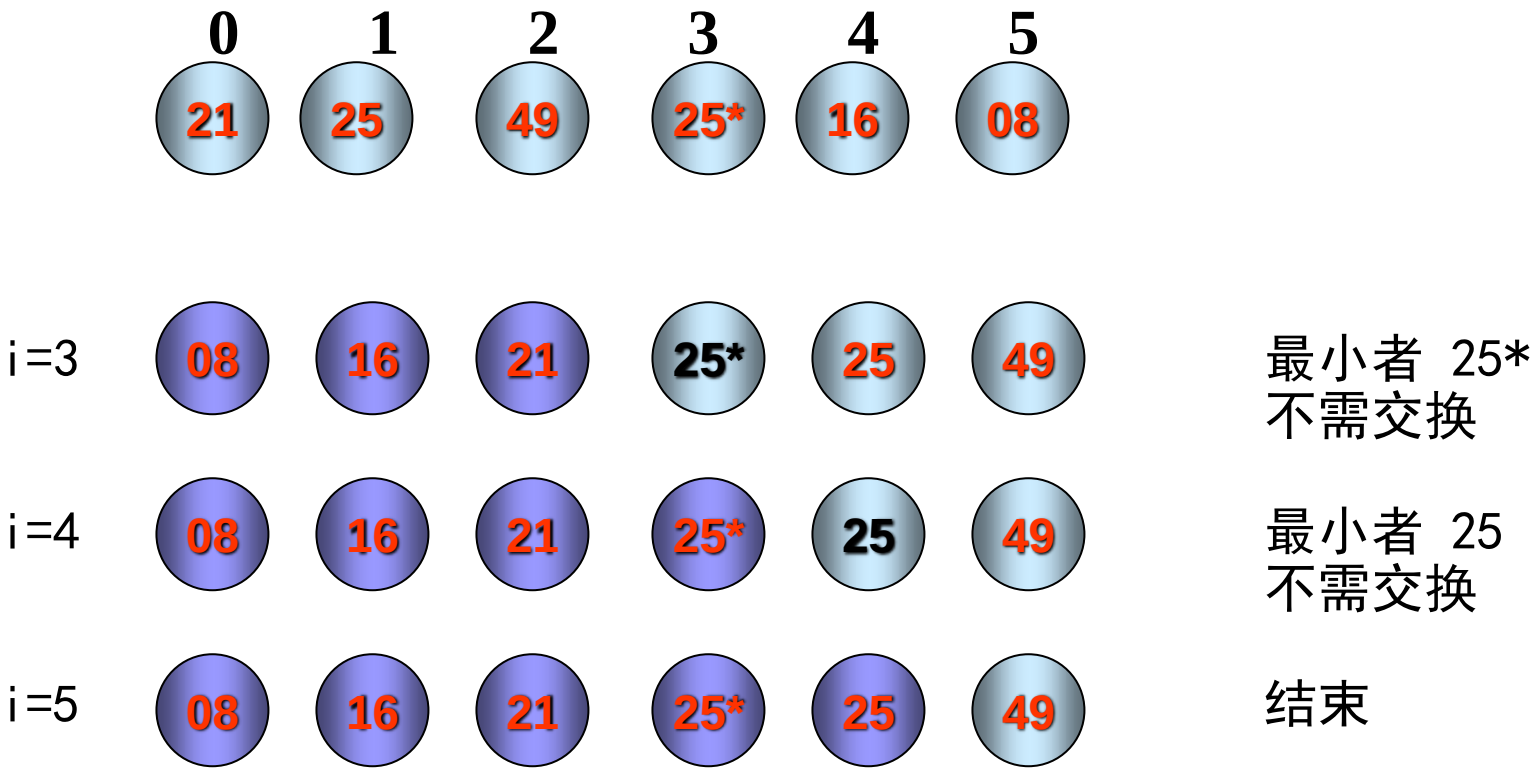
第四节　选择排序

一、简单选择排序(举例)



第四节　选择排序

一、简单选择排序(举例)



第四节　选择排序

一、简单选择排序(性能分析)

- 直接选择排序的关键字比较次数 KCN 与记录的初始排列无关。
- 设整个待排序记录序列有n个记录,则第i趟选择具有最小关键字记录所需的比较次数总是 n-i-1次。总的关键字比较次数为

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

第四节　选择排序

一、简单选择排序(性能分析)

- 记录的移动次数与记录序列的初始排列有关。当这组记录的初始状态是按其关键字从小到大有序的时候,记录的移动次数RMN=0,达到最少。
- 最坏情况是每一趟都要进行交换，总的记录移动次数为 RMN = 3(n-1)。
- 直接选择排序是一种不稳定的排序方法。

练习

- 49 38 65 97 76 13 27 49* 55 04

第四节　选择排序

二、堆排序

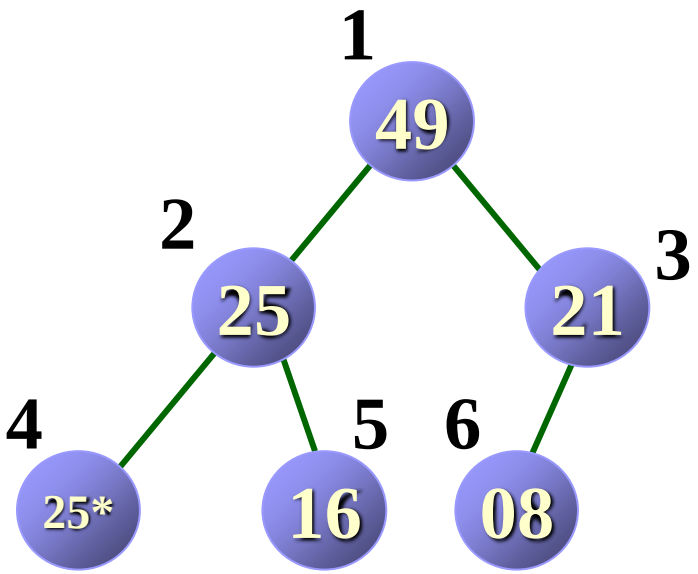
- 设有一个关键字集合，按完全二叉树的顺序存储方式存放在一个一维数组中。对它们从根开始，自顶向下，同一层自左向右从 1 开始连续编号。若满足

$$K_i \geq K_{2i} \;\&\&\; K_i \geq K_{2i+1}$$

则称该关键字集合构成一个堆(**最大堆**)

第四节　选择排序

二、堆排序(举例)



第四节　选择排序

二、堆排序

- 堆排序主要要解决两个问题：

1. 如何根据给定的序列建初始堆

2. 如何在交换掉根结点后，将剩下的结点调整为新的堆(筛选)

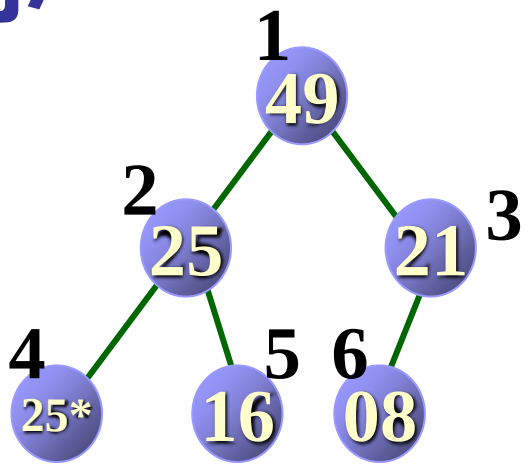
第四节　选择排序

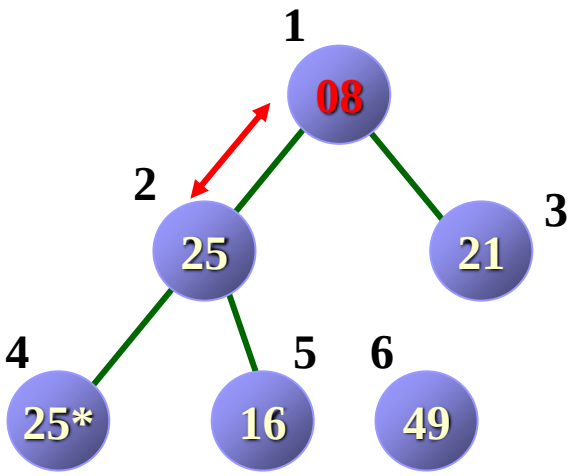
二、堆排序(筛选)——最大堆

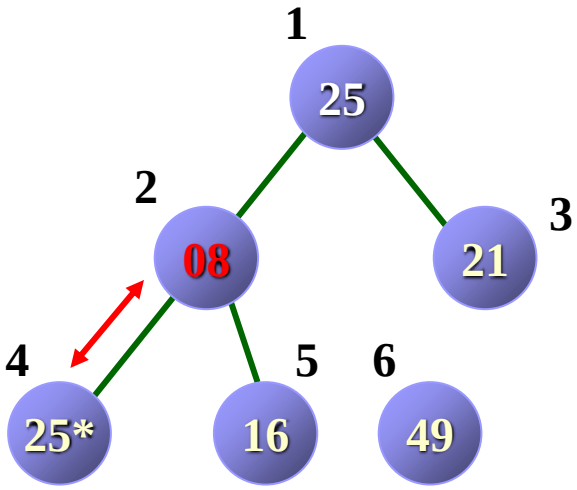
- 输出根结点
- 用最后结点代替根结点值
- 比较根结点与两个子结点的值，如果小于其中一个子结点，则选择大的子结点与根结点交换
- 继续将交换的结点与其子结点比较
- 直到叶子结点或者根节点值大于两个子结点

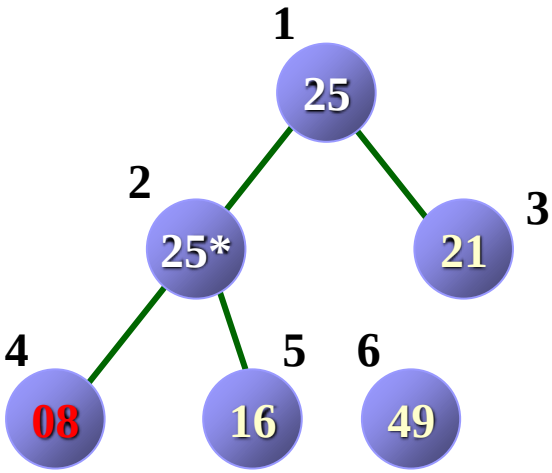
第四节　选择排序

二、堆排序（筛选举例，一趟筛选过程）









第四节　选择排序

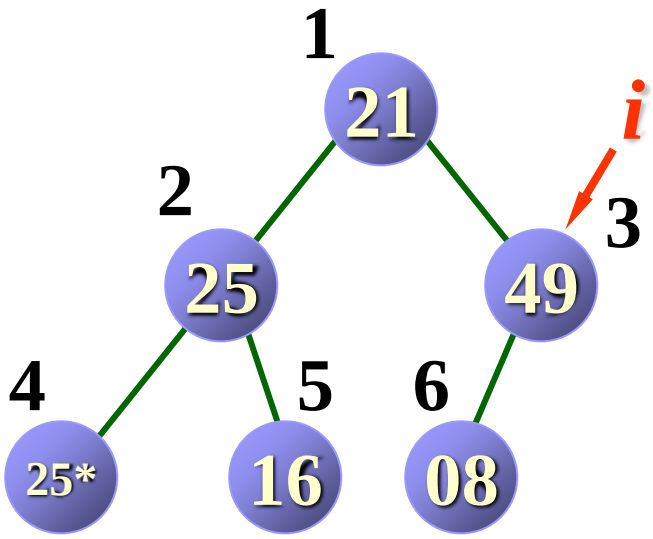
二、堆排序(创建初始堆)

- 根据给定的序列，从1至n按顺序创建一个完全二叉树
- 由最后一个非终端结点(第n/2个结点)开始至第1个结点，逐步做筛选

第四节　选择排序

二、堆排序(创建初始堆举例)

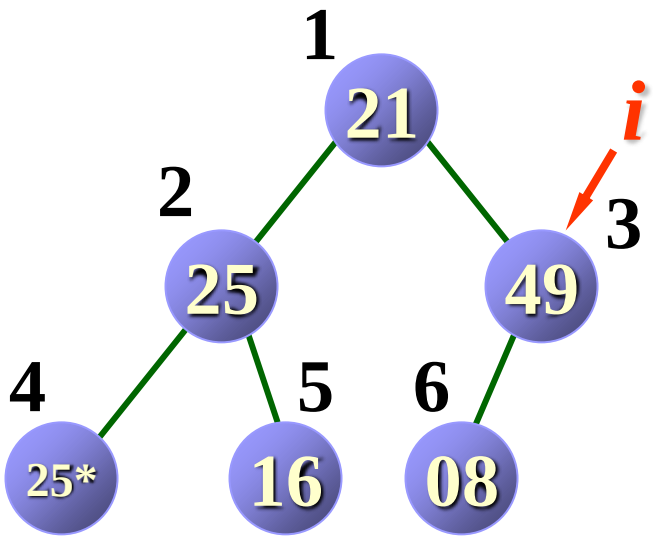
- 已知待序的一组记录的初始排列为：21, 25, 49, 25*, 16, 08



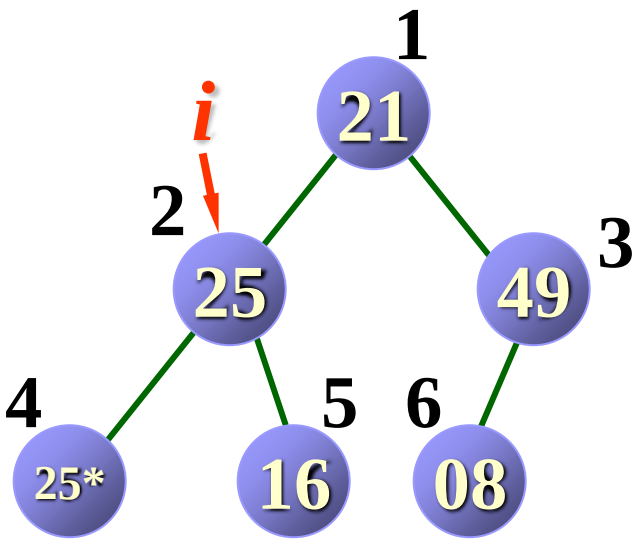
初始排序码集合

第四节　选择排序

二、堆排序(创建初始堆举例)



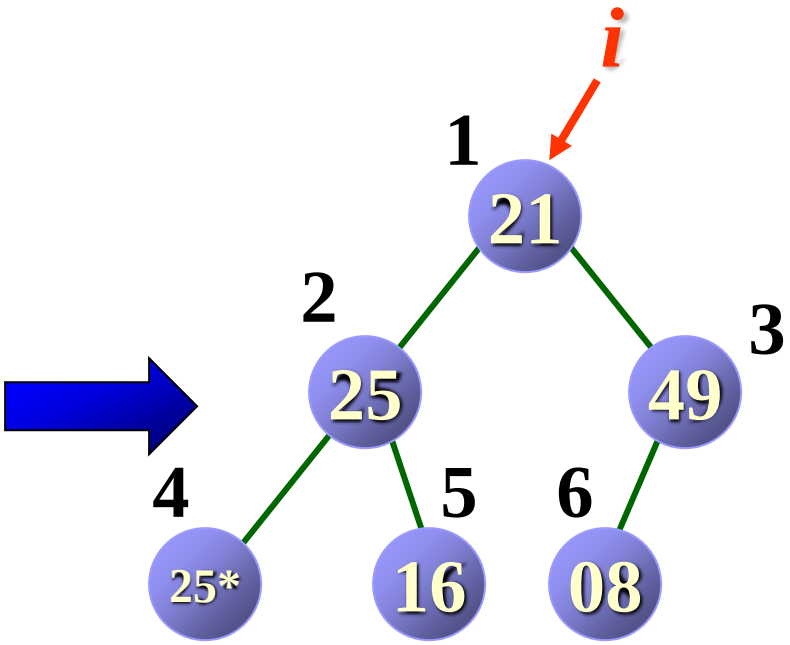
初始排序码集合



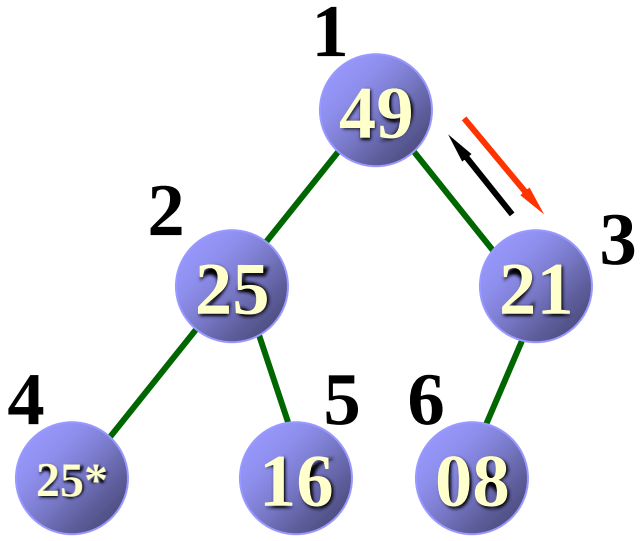
i = 3 时的局部调整

第四节　选择排序

二、堆排序(创建初始堆举例)



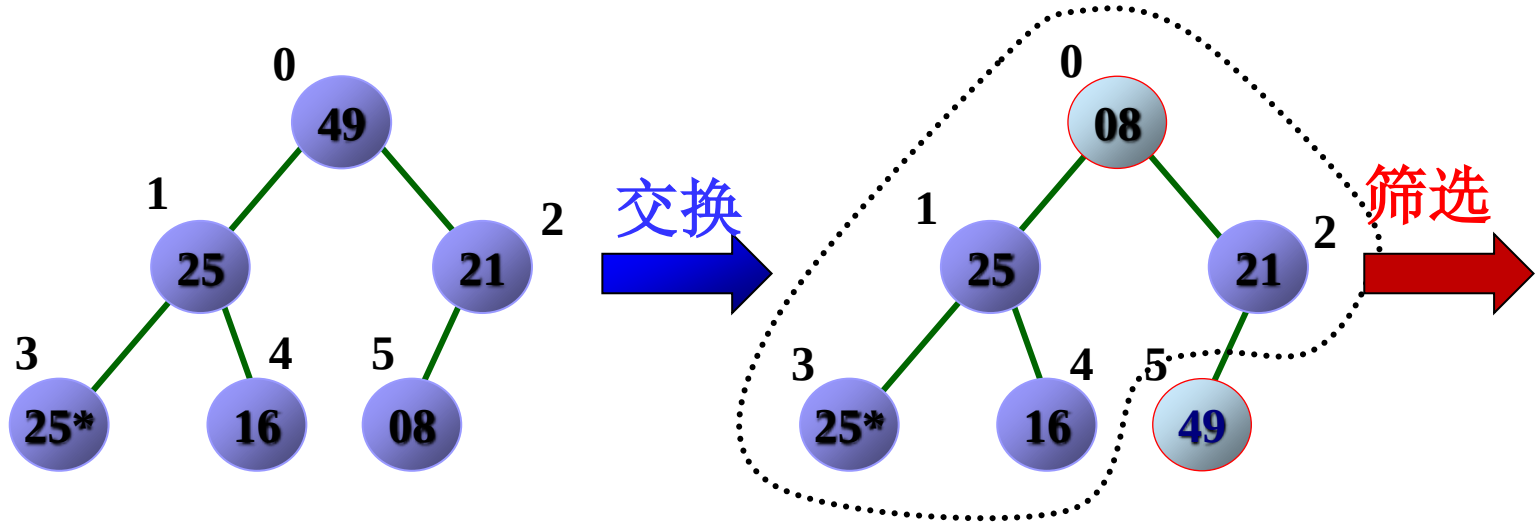
$i = 2$ 时的局部调整



$i = 1$ 时的局部调整
形成最大堆

第四节　选择排序

二、堆排序(举例)



49	25	21	25*	16	08

初始最大堆

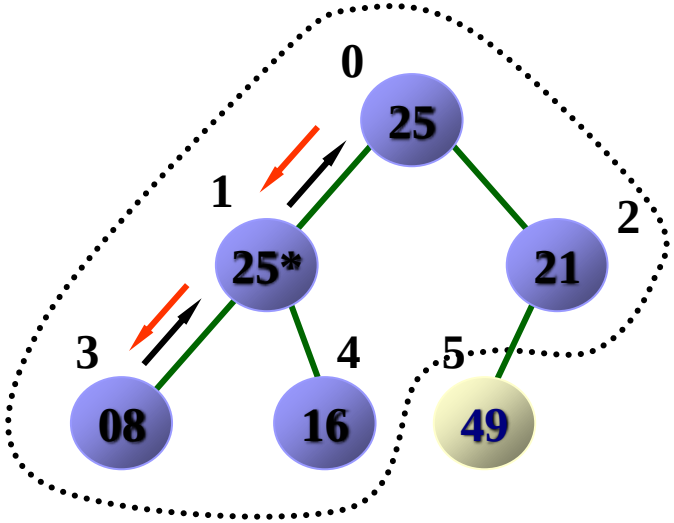
08	25	21	25*	16	49

交换 0 号与 5 号记录，
5 号记录就位

第四节　选择排序

二、堆排序(举例)

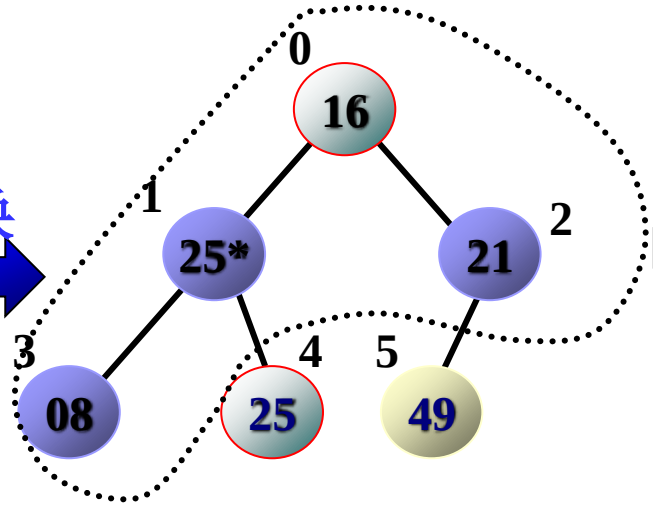
筛选



25	25*	21	08	16	49

从 0 号到 4 号 重新
调整为最大堆

交换



16	25*	21	08	25	49

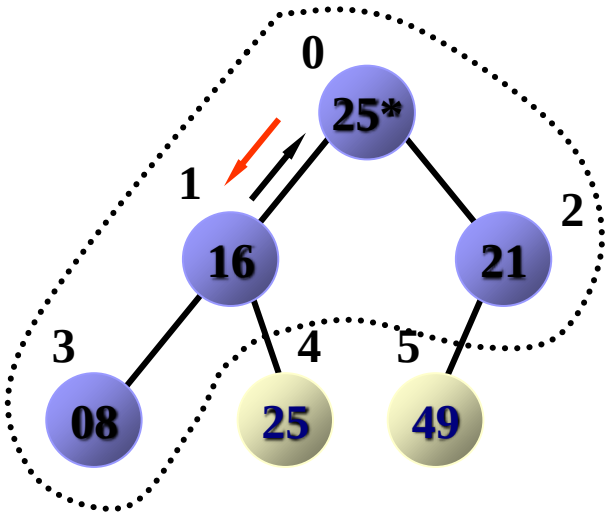
交换 0 号与 4 号记录,
4 号记录就位

筛选

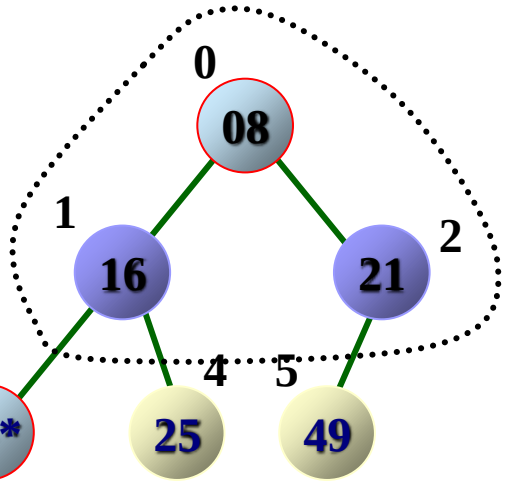
第四节　选择排序

二、堆排序(举例)

筛选



交换



筛选

25*	16	21	08	25	49

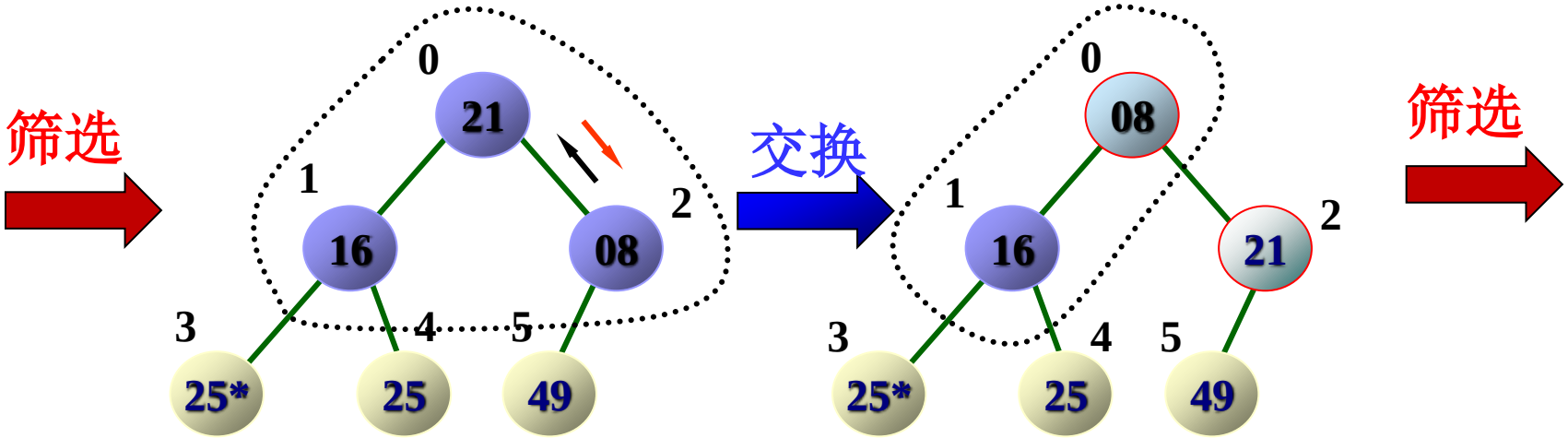
从 0 号到 3 号 重新
调整为最大堆

08	16	21	25*	25	49

交换 0 号与 3 号记录,
3 号记录就位

第四节　选择排序

二、堆排序(举例)



21	16	08	25*	25	49

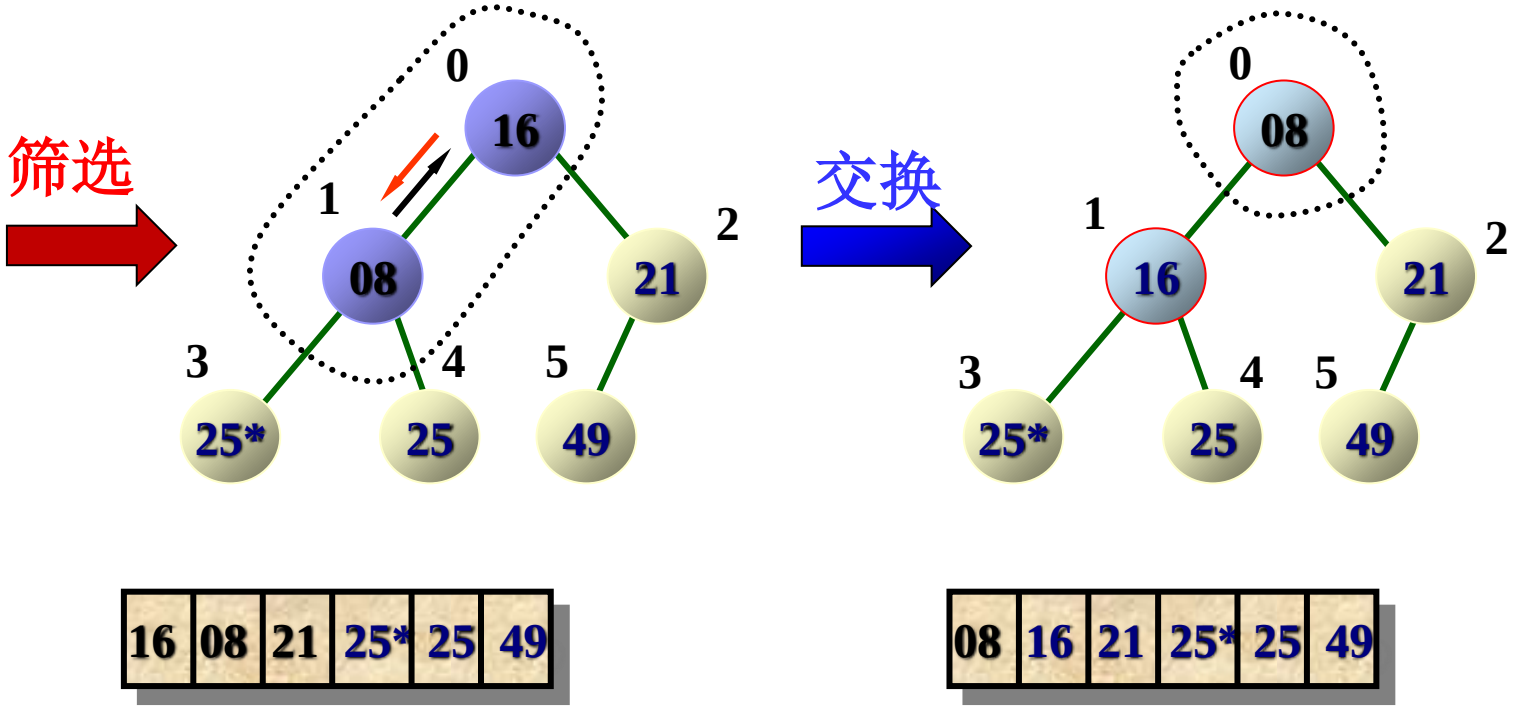
从 0 号到 2 号 重新调整为最大堆

08	16	21	25*	25	49

交换 0 号与 2 号记录，2 号记录就位

第四节　选择排序

二、堆排序(举例)



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 号记录，1 号记录就位

第四节　选择排序

二、堆排序(性能分析)

- 对于长度为n的序列，其对应的完全二叉树的深度为k ($2^{k-1} \leq n < 2^k$)
- 对深度为k的堆，筛选算法中进行的关键字比较次数至多为2(k-1)次
- 堆排序时间主要耗费在建初始堆和调整建新堆(筛选)上
- 建初始堆最多做n/2次筛选

第四节　选择排序

二、堆排序(性能分析)

- 对长度为n的序列，排序最多需要做n-1次调整建新堆(筛选)
- 因此共需要O(nxk)量级的时间
- $k = \log_2 n$
- 堆排序时间复杂度为$O(n\log_2 n)$
- 堆排序是一个不稳定的排序方法
- 记录数较多时，推荐堆排序

10.5 归并排序

第五节　归并排序

一、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。

第五节　归并排序

二、两路归并


```
typedef int SortData;
void merge ( SortData InitList[ ], SortData mergedList[ ],
                    int left, int mid,  int right ) {
   int i = left,  j = mid+1,  k = left;
   while ( i <= mid && j <= right )                    //两两比较将较小的并入
      if ( InitList[i] <= InitList[j] )  { mergedList [k] = InitList[i]; i++;  k++; }
      else                                { mergedList [k] = InitList[j]; j++;  k++; }
   while (i <= mid)  { mergedList[k] = InitList[i];  i++;  k++; }//将mid前剩余的
    并入
   while (j <= right){ mergedList[k] = InitList[j];  j++;  k++; }//将mid后剩余的
    并入
}
```

第五节　归并排序

二、两路归并(性能分析)

- 假设待归并两个有序表长度分别为m和n，则两路归并后，新的有序表长度为m+n
- 两路归并操作至多只需要m+n次移位和m+n次比较
- 因此两路归并的时间复杂度为O(m+n)

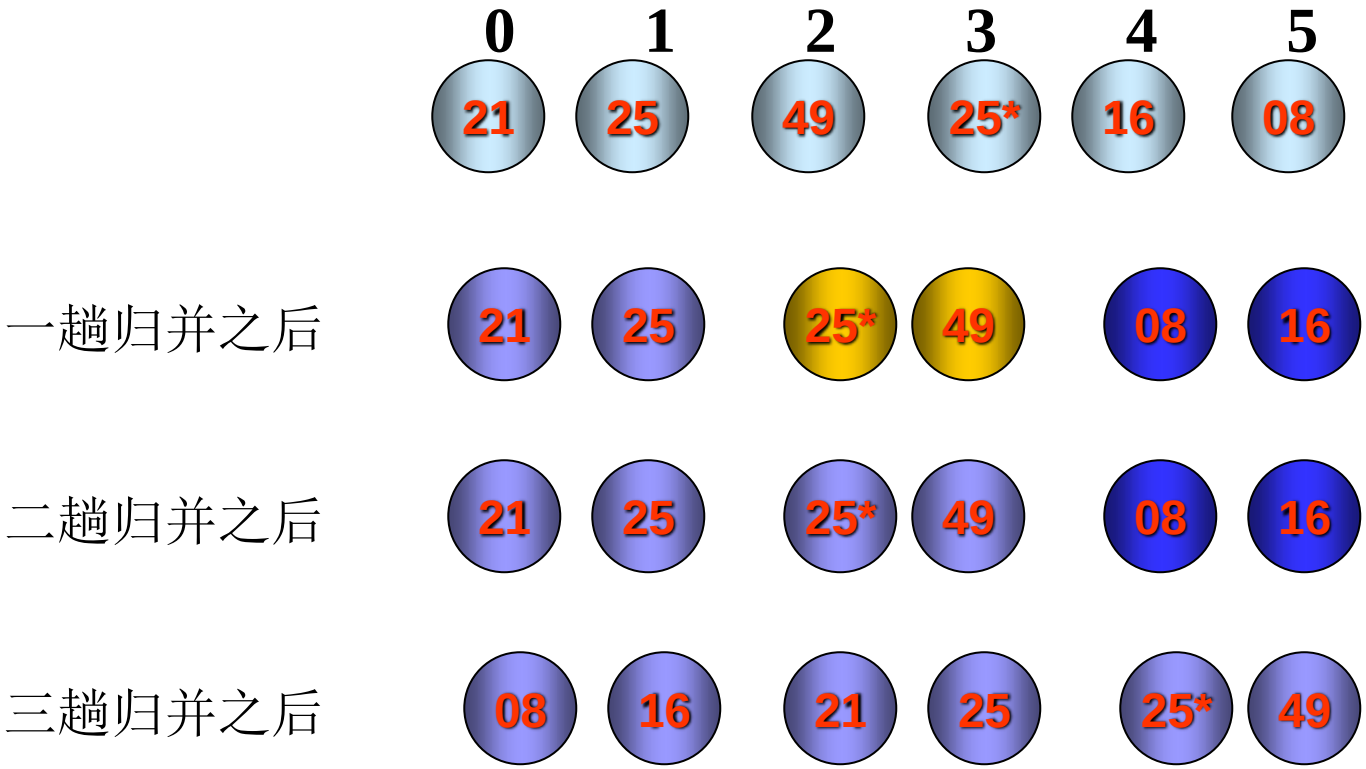
第五节　归并排序

三、2路－归并排序

- 将n个记录看成是n个有序序列
- 将前后相邻的两个有序序列归并为一个有序序列（两路归并）
- 重复做两路归并操作，直到只有一个有序序列为止

第五节　归并排序

三、2路一归并排序(举例)



第五节　归并排序

三、2路—归并排序(性能分析)

- 如果待排序的记录为n个，则需要做$\log_2 n$趟两路归并排序
- 每趟两路归并排序的时间复杂度为O(n)
- 因此2路—归并排序的时间复杂度为$O(n\log_2 n)$
- 归并排序是一种稳定的排序方法

第六节　基数排序

一、多关键字的排序

- 例：对52张扑克牌按以下次序排序：
  ♣2<♣3<……<♣A<♦2<♦3<……<♦A<
  ♥2<♥3<……<♥A<♠2<♠3<……<♠A
- 两个关键字：花色（♣<♦<♥<♠）
  面值（2<3<…<A）
- 并且“花色”地位高于“面值”

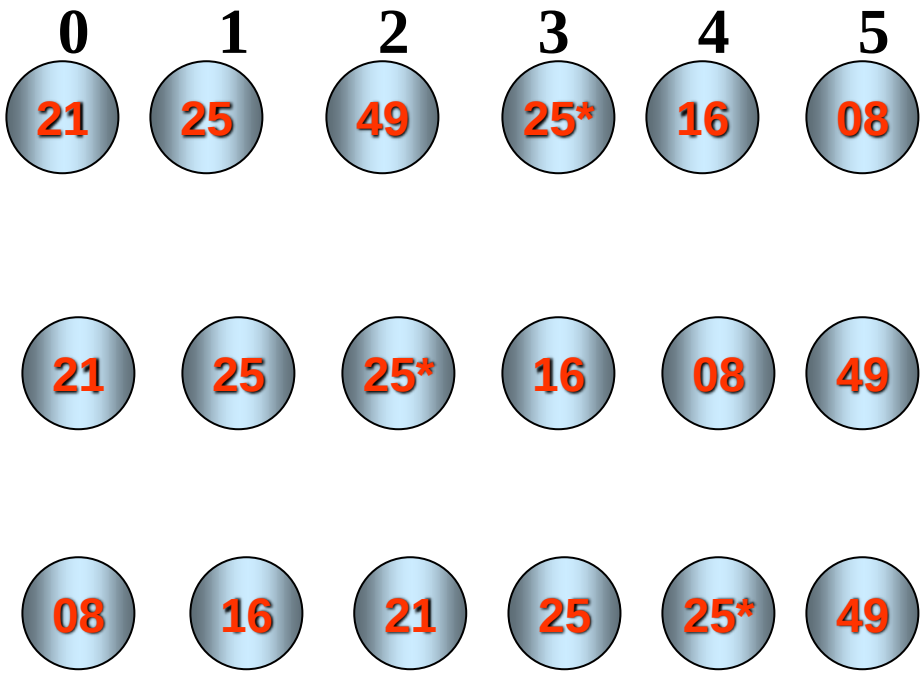
第六节　基数排序

一、多关键字的排序(最低位优先法LSD)

- 从最低位关键字kd起进行排序，
- 然后再对高一位的关键字排序，……
- 依次重复，直至对最高位关键字k1排序后，便成为一个有序序列

第六节　基数排序

一、多关键字的排序(最低位优先法LSD—举例)



第六节　基数排序

二、链式基数排序

- 基数排序：借助“分配”和“收集”对单逻辑关键字进行排序的一种方法
- 链式基数排序方法：用链表作存储结构的基数排序
- 设置10个队列，f[i]和e[i]分别为第i个队列的头指针和尾指针

第六节　基数排序

二、链式基数排序

- 第i趟分配：根据第i位关键字的值，改变记录的指针，将链表中记录分配至10个链队列中，每个队列中记录关键字的第i位关键字相同

- 第i趟收集：改变所有非空队列的队尾记录的指针域，令其指向下一个非空队列的队头记录，重新将10个队列链成一个链表

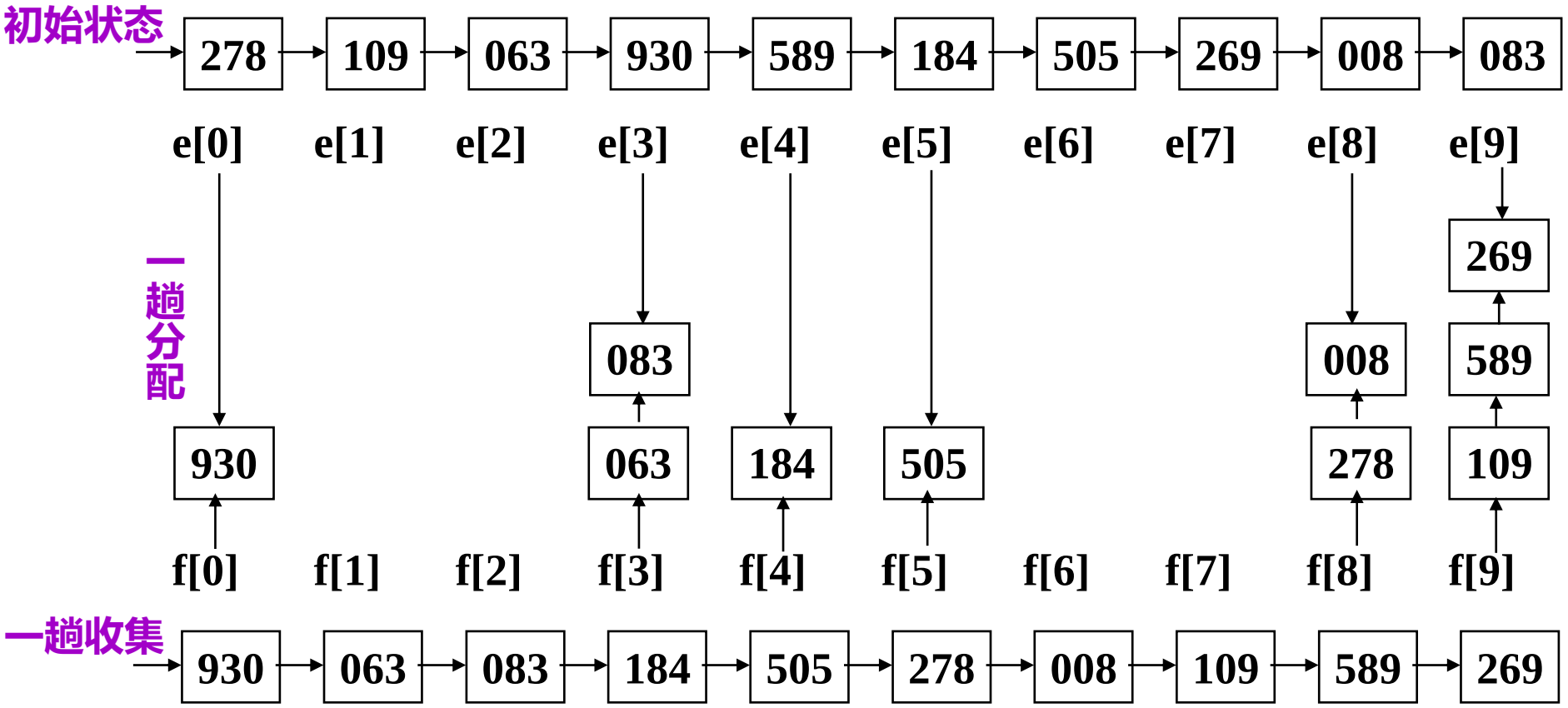
第六节　基数排序

二、链式基数排序

- 从最低位至最高位，逐位执行上述两步操作，最后得到一个有序序列

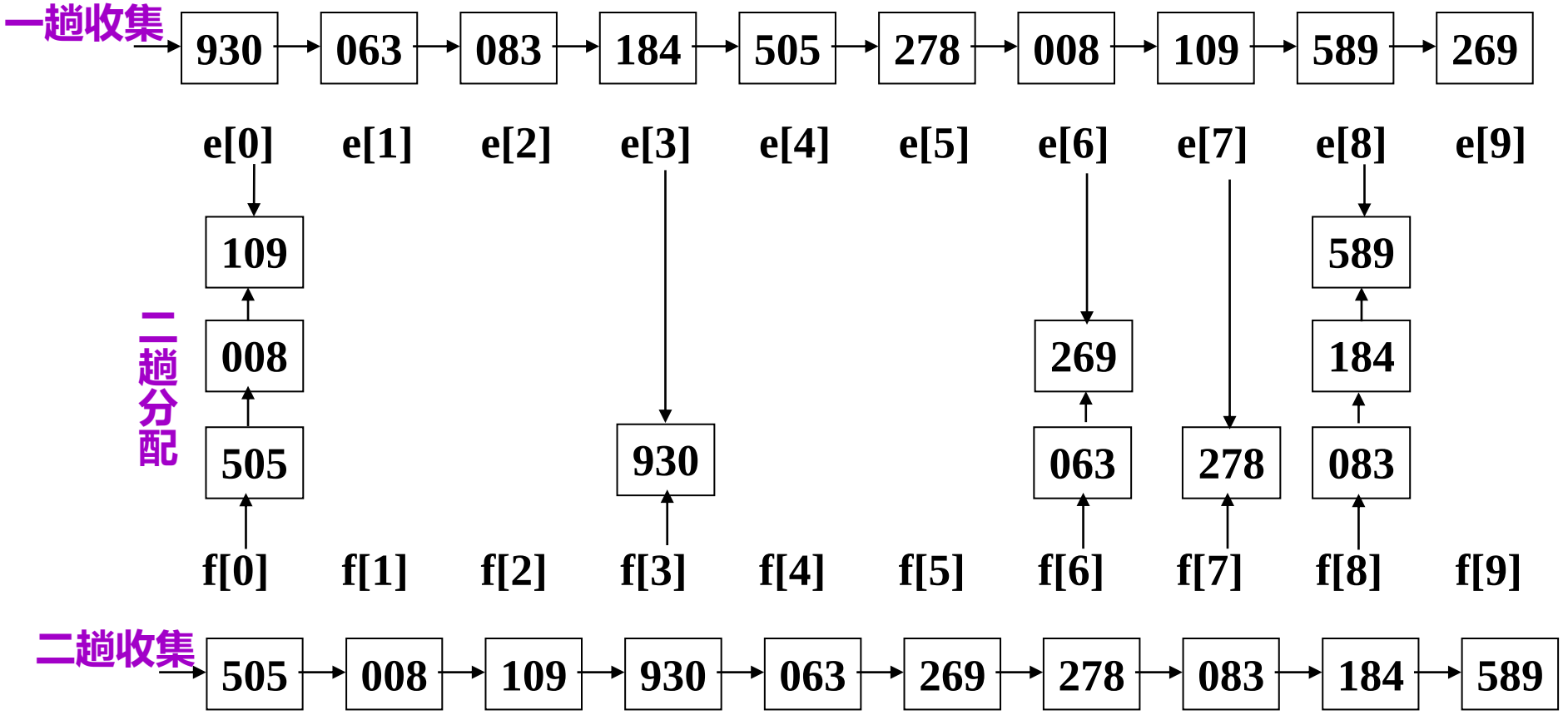
第六节　基数排序

二、链式基数排序（举例）



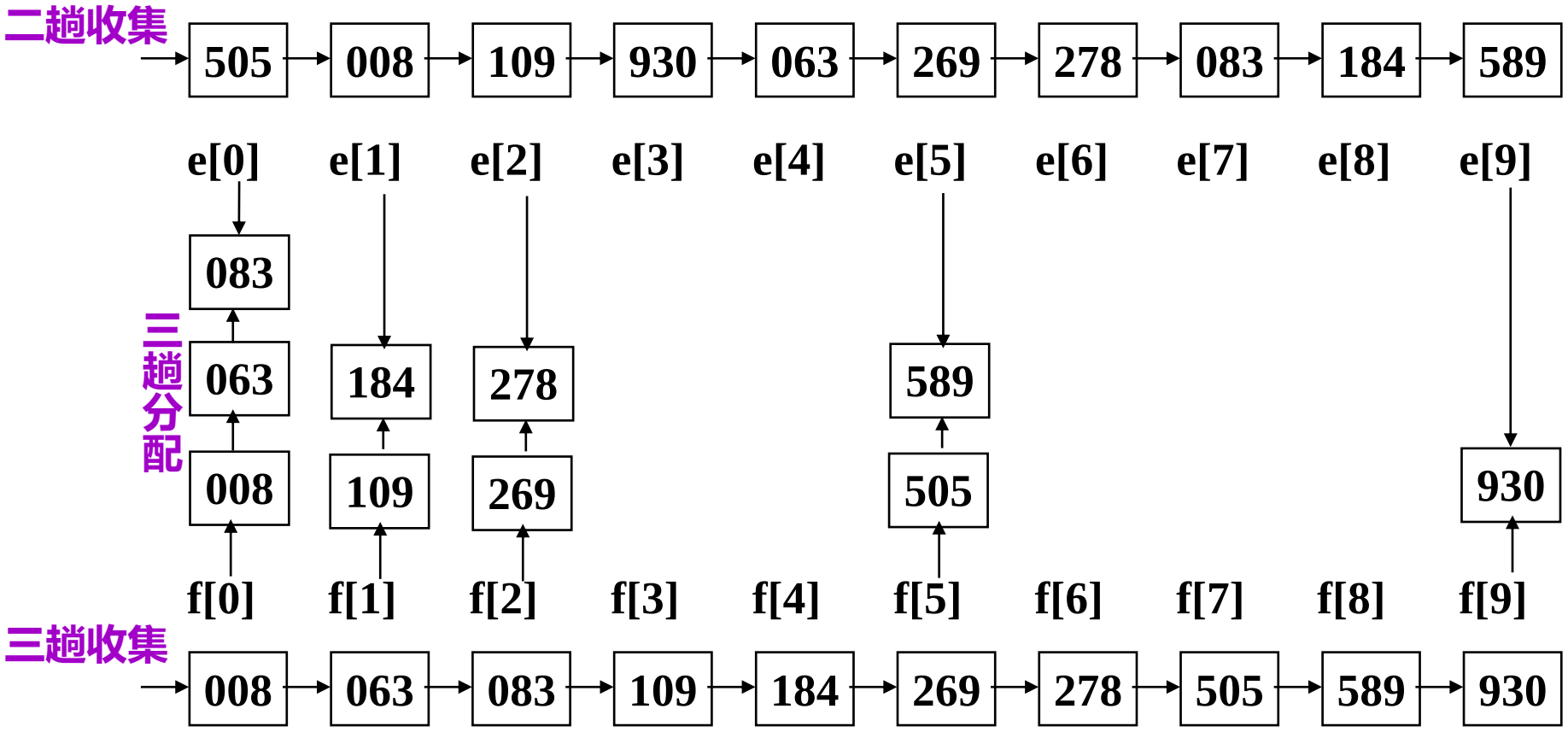
第六节　基数排序

二、链式基数排序(举例)



第六节　基数排序

二、链式基数排序(举例)



第六节　基数排序

二、链式基数排序(性能分析)

- 若每个关键字有 d 位,关键字的基数为radix
- 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))。

第六节　基数排序

二、链式基数排序(性能分析)

- 若基数radix相同，对于对象个数较多而关键字位数较少的情况，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。

第七节　各种排序方法比较

排序方法	平均时间	最坏情况	辅助存储	适合情况
插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	记录数不很多
希尔排序	$O(n(\log_2 n)^2)$	$O(n^2)$	$O(1)$	不太多
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	较多
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	较多
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	都可以
基数排序	$O(d(n+rd))$	$O(d(n+rd))$	$O(rd)$	关键字位数少

注：红色的是稳定的排序算法，黑色的是不稳定的排序算法，

一、时间性能

1. 平均的时间性能

时间复杂度为 $O(n\log n)$：

快速排序、堆排序和归并排序

时间复杂度为 $O(n^2)$：

直接插入排序、起泡排序和
简单选择排序

时间复杂度为 $O(n)$：

基数排序

2. 当待排记录序列按关键字顺序有序时

直接插入排序和起泡排序能达到O(n)的时间复杂度，快速排序的时间性能蜕化为$O(n^2)$。

3. 简单选择排序、堆排序和归并排序的时间性能不随记录序列中关键字的分布而改变。

二、空间性能

指的是排序过程中所需的辅助空间大小

1. 所有的**简单排序方法**(包括：直接插入、起泡和简单选择) **和堆排序**的空间复杂度**为O(1)**；
2. **快速排序为O(log*n*)，**为递归程序执行过程中，栈所需的辅助空间；
3. 归并排序所需辅助空间最多，其空间复杂度为 **O(n)**;
4. 链式基数排序需附设队列首尾指针，则空间复杂度为 **O(rd)**。

三、排序方法的稳定性能

1. 稳定的排序方法指的是，对于两个关键字相等的记录，它们在序列中的相对位置，在排序之前和经过排序之后，没有改变。

排序之前：$\{\cdots\cdots R_i(K)\cdots\cdots R_j(K)\cdots\cdots\}$

排序之后：$\{\cdots\cdots R_i(K)\ R_j(K)\cdots\cdots\cdots\cdots\}$

2. 当对多关键字的记录序列进行LSD方法排序时，必须采用稳定的排序方法。

例如：

排序前 (56, 34, **47**, 23, 66, 18, 82, **47**)

若排序后得到结果

(18, 23, 34, **47**, **47**, 56, 66, 82)

则称该排序方法是稳定的；

若排序后得到结果

(18, 23, 34, **47**, **47**, 56, 66, 82)

则称该排序方法是不稳定的。

3. 对于不稳定的排序方法，只要能举出一个实例说明即可。

例如：对{**4**，3，4，2} 进行快速排序，

得到 {2，3，4，**4**}

4. 快速排序、堆排序和希尔排序是不稳定的排序方法。

5. 所有时间复杂度为$O(n^2)$的简单排序算法都是稳定的（直接选择排序算法除外）。

6. 归并排序和基数排序是稳定的。

作业

- 有下列数据，请对其进行排序具体要求如下

  43 55 35 23 16 36 10 47 23* 89

1. 请写出进行直接插入排序，希尔排序，冒泡排序，快速排序，简单选择排序，堆排序，归并排序的前三趟排序结果，及最后的排序结果，并写出各种排序方法的时间复杂度，空间复杂度及稳定性
2. 请写出对其进行快速排序时第一趟划分的过程
3. 请画出对其进行基数排序的过程
4. 请画出对其进行堆排序时，建的初始堆，进行堆排的过程，请画出对应的数组内容