**1.基本概念**

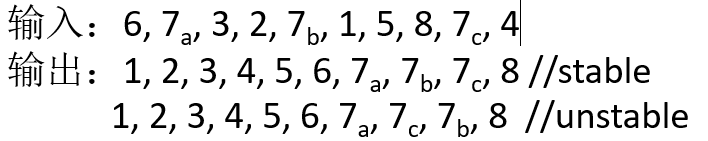
**评估指标**

时间开销：排序码的比较次数，元素移动次数

空间开销：**就地排序**：若排序算法所需辅助空间不依赖问题的规模n，即**空间复杂度O(1)**

非就地排序：空间复杂度**与n有关**

排序算法的稳定性：当输入含重复关键字/排序码时，**重复元素在输入、输出序列中的相对次序是保持不变的**，那么称该排序算法稳定



算法是否是**输入敏感**的(input-sensitive) ：算法的复杂度不仅与输入数据的规模有关，还与**输入数据本身所具有的特性 (无序程度)相关**

**排序算法分类**

**根据排序发生的位置**：

内部排序：在整个排序过程，数据元素全部存放在内存、不需要访问外存便能完成的排序；

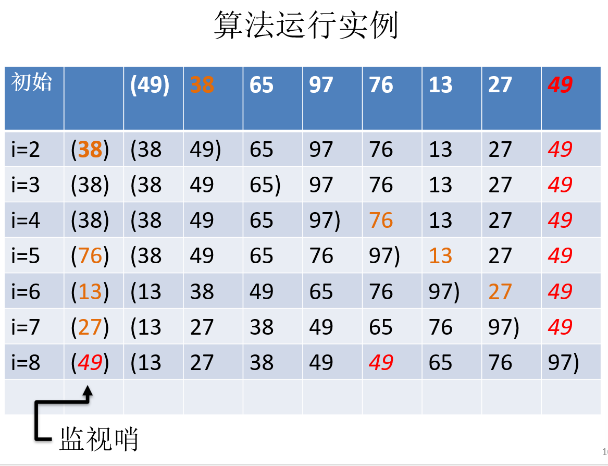
外部排序：参加排序的数据元素不能同时存放在内存，排序过程中必须不断在内、外存之间进行数据交换的排序；

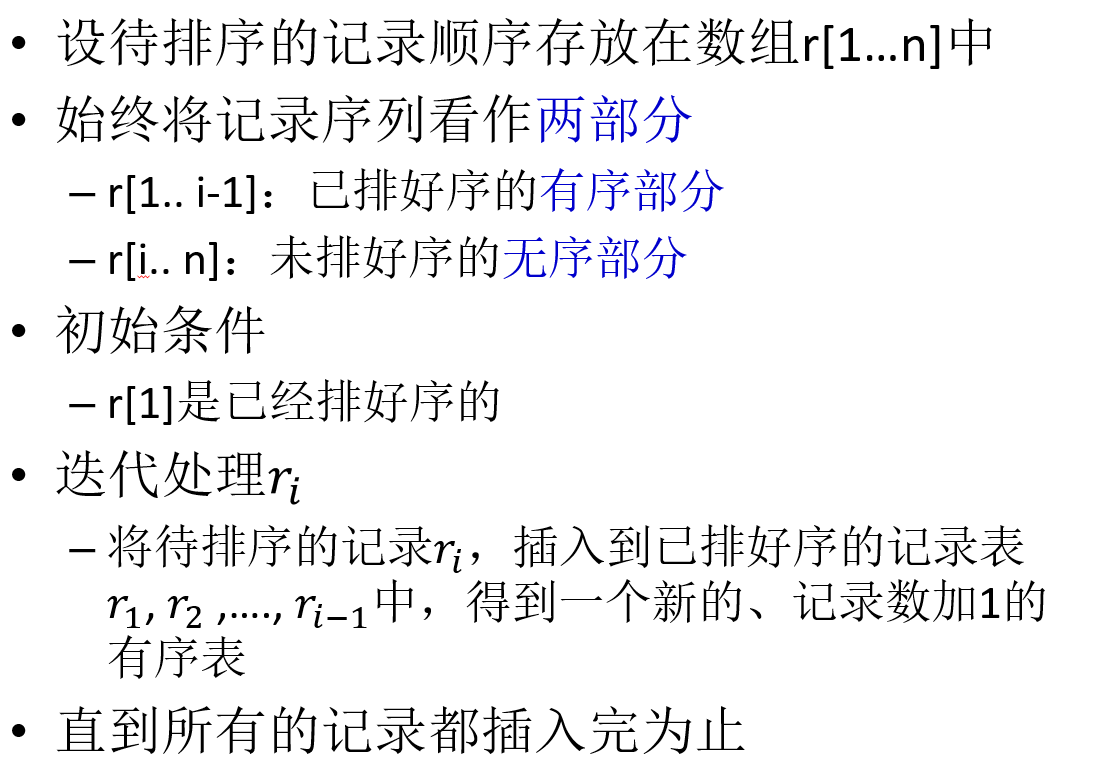
**根据排序方式**：

插入排序(直接插入排序，折半插入排序，2-路插入排序，表插入排序，希尔排序)、交换排序(起泡排序，快速排序)、选择排序(简单选择排序，树型选择排序，堆排序)、归并排序、基数排序

**2.插入排序**

基本方法：每趟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

**2.1直接插入排序(Straight Insertion Sort, 基于顺序查找)**

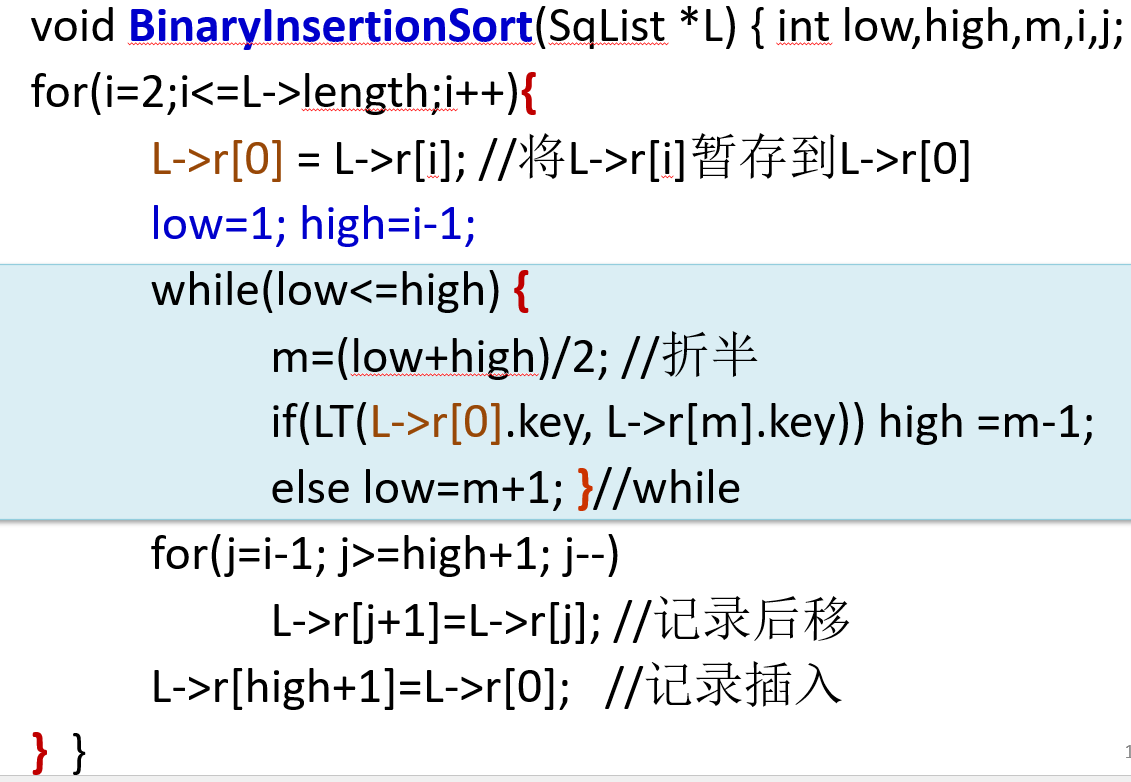


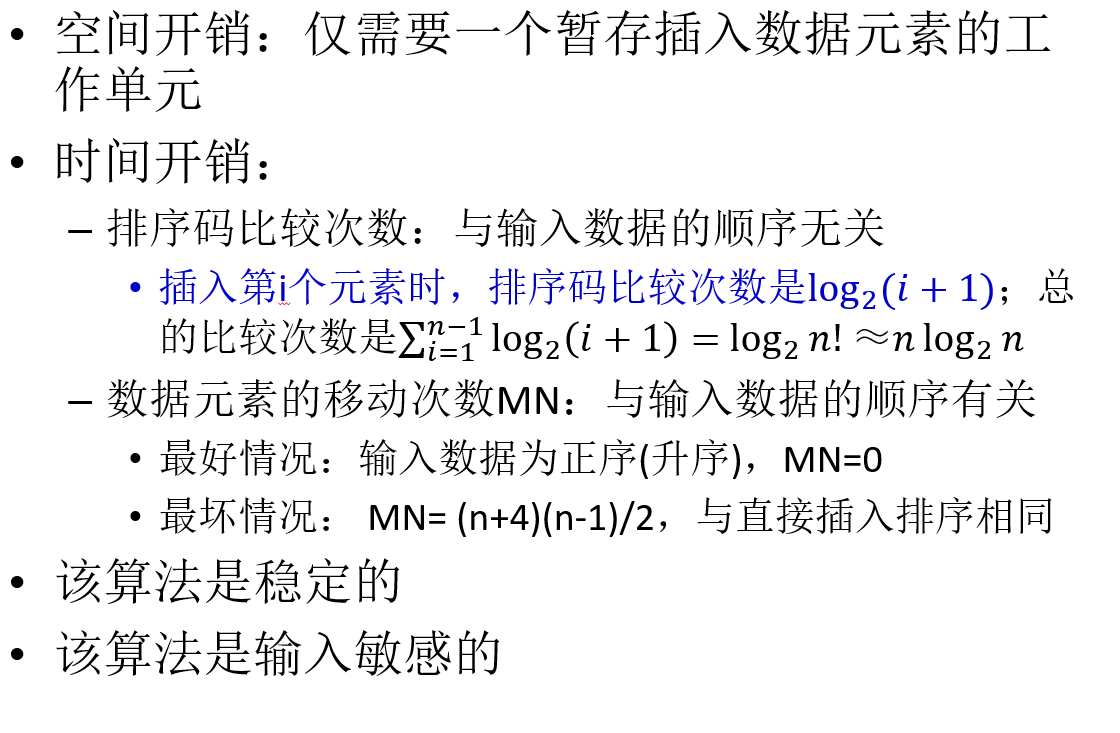
时间复杂度为O(𝑛^2)：最坏情况，第k次迭代，需k次比较；累计∑n𝑖=2(𝑖)次比较，移动记录∑n𝑖=2(𝑖+1)次

该算法是稳定的

该算法是输入敏感的：最好情况输入数据完全(或几乎)正序，每次迭代只需1次比较，累计O(n)时间

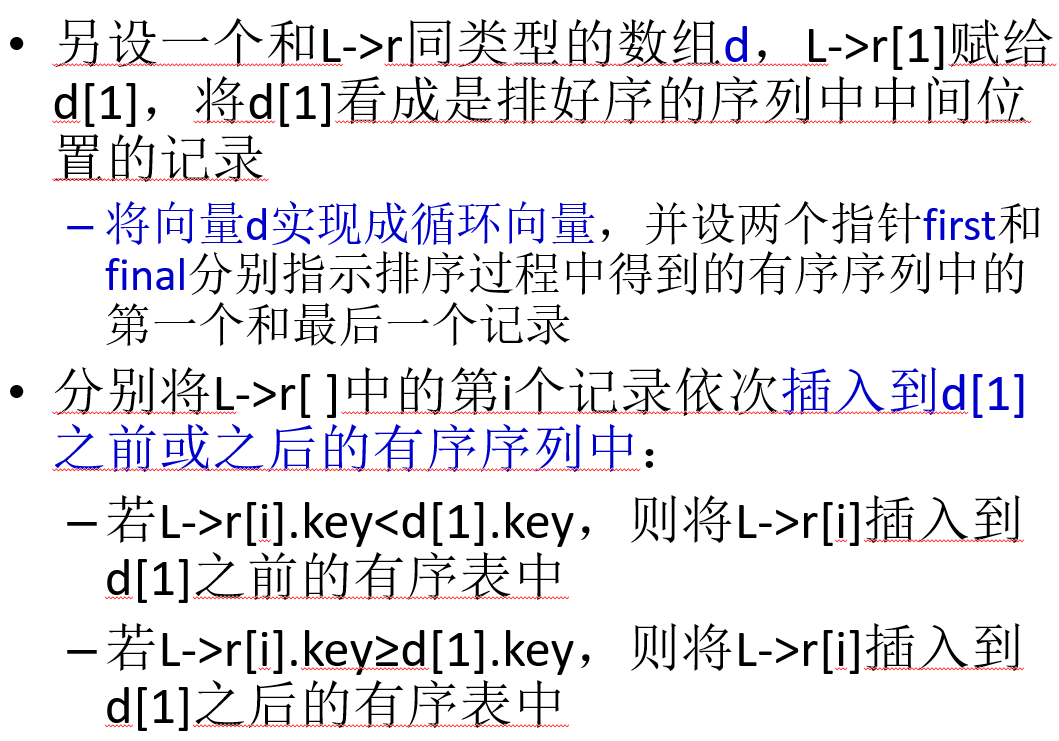
**2.2折半插入排序(Binary Insertion Sort, 基于折半查找)**

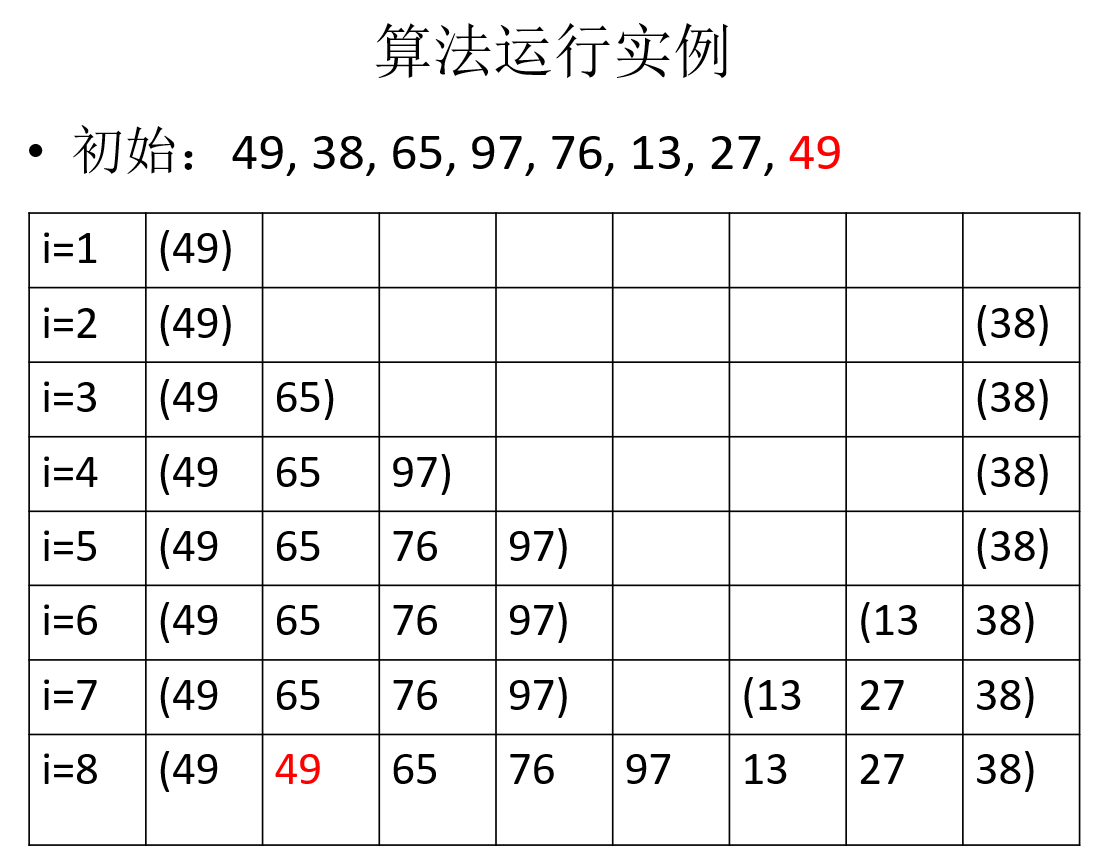


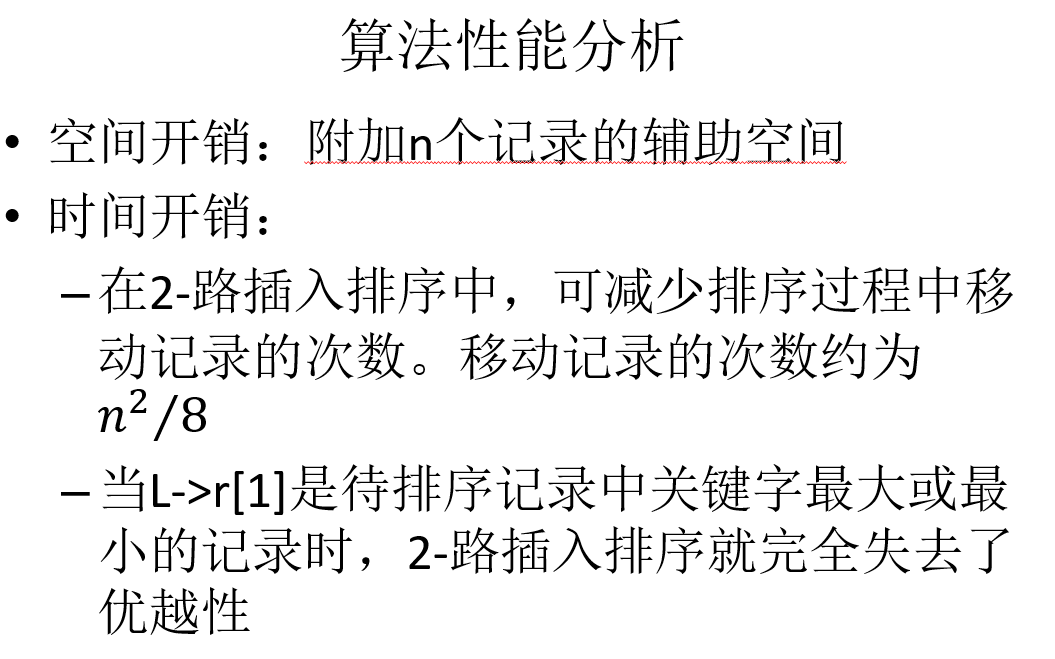


2.3 2-路插入排序(Two-way Insertion Sort)

可减少移动次数（但复杂度没变化）

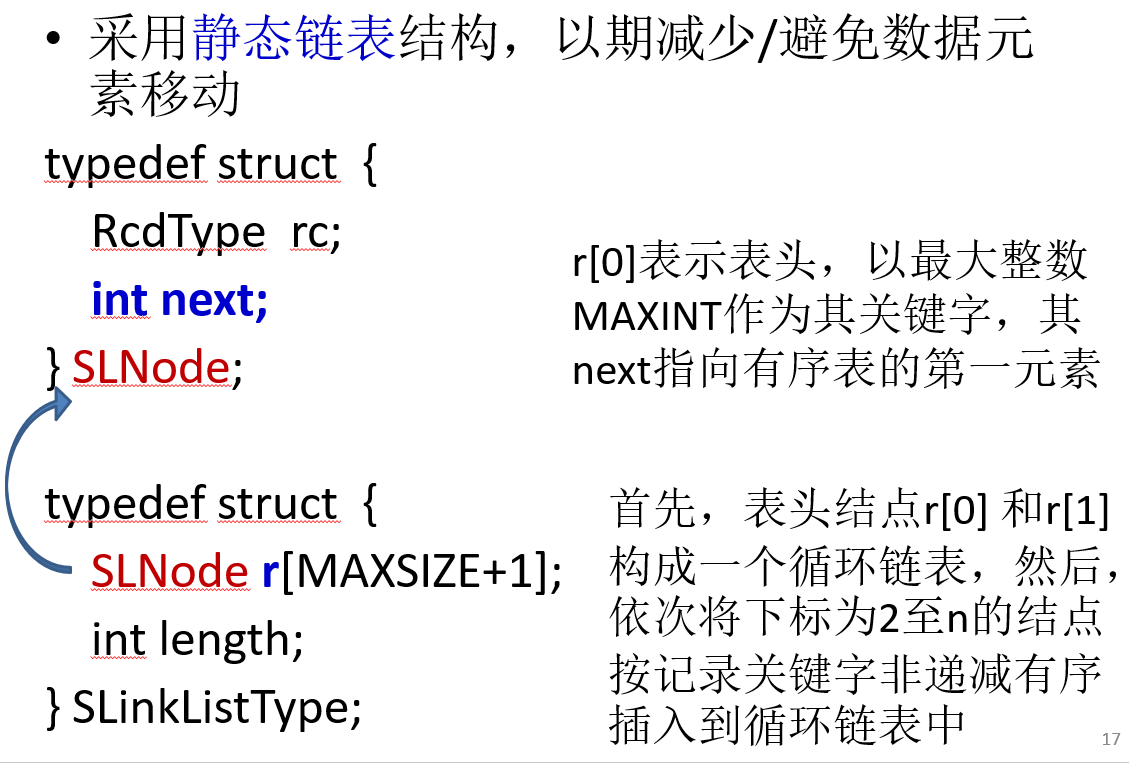






算法是稳定的、输入敏感的

2.4 表插入排序(List Insertion Sort, 基于静态链表存储)



r[0]作为表头（上界），r[1]作为（初始时的）表尾，构成一个循环链表

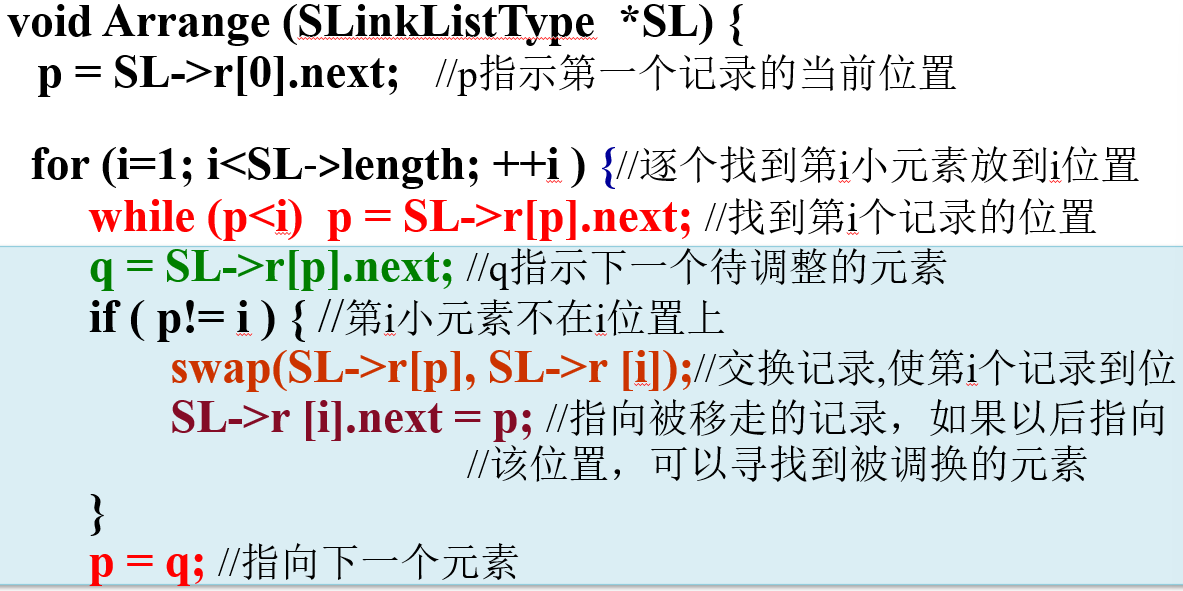
r[1]~r[n]用于存储待排序的数据

从r[r[0].next]开始，是一个升序排列的链表，最大值为r[0].rc=MAXINT

插入排序过程中不移动记录，只改变记录的链接顺序

为了方便查找，需要对结果进行重新排列求得一个有序数组

**重排列**



**性能分析**

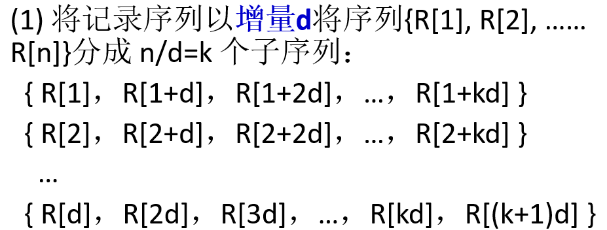
静态表插入排序算法：记录不移动，但比较次数未减少，时间开销O(𝑛^2)

调整算法：每一趟交换一对记录，平均时间开销为O(n)

总的时间开销为O(𝑛^2)

空间开销：需要O(n)的辅助空间

表插入排序是稳定的排序方法

**2.5希尔排序(Shell’s Sort, Diminishing Increment Sort,基于逐趟缩小增量)**

希尔排序是一种不稳定的插入排序方法

希尔排序的时间复杂性与各列内部排序的算法有关：内部排序不一定是高效的，但需要是输入敏感的（一般用直接插入排序作为子序列排序方法）

实际运行的时间更多地取决于所取“增量/gap”序列，尚未解决

Knuth：取 gap = ⌊𝑔𝑎𝑝∕3⌋+1

**3.交换排序**

**3.1 起泡排序(Bubble Sort)**

初始i=n；

每轮扫描a[0~i]：取j=0~i-1，每次比较a[j]和a[j+1]，若a[j]>a[j+1]，则交换。

一轮扫描过后i--；如果**整趟扫描都没有进行交换，则排序完成(使用sorted标志)**

**算法分析**

不变性：经k轮扫描交换后，最大的k个元素必然就位

正确性：经至多n轮扫描后，算法必然终止。且能给出正确的解答

时间复杂度：各个版本都相同：最好O(n)，最坏𝑂(𝑛^2)

起泡排序是**稳定的（使用大于号而非大于等于）**两相等值不会发生交换，保持原数组中的相对次序。

**3.2 快速排序(Quick Sort)**

(分治策略) 任取序列S中的记录m作为轴点(pivot)，将序列分为两个子序列，即，𝑆=𝑆\_𝐿+𝑚+𝑆\_𝑅，**将所有比轴点小的数都放到它前面，所有比轴点大的数都放到它后面**

**算法流程**

两个指针： i指向当前待放入的小于轴点的位置

j指向当前待放入的大于轴点的位置

以a[low]为pivot标杆，初始时i=low, j=high

外部大while循环(i<j)：

使用相互覆盖的两个while循环：

（1）先处理 j 指针指向的数据：如果指向的数据大于标杆元素，则j--；直到 j 遇到 比pivot小的元素，（或出现i>=j）结束第一个循环。然后用其覆盖掉 i 指向的元素。

（2）再处理 i指针指向的数据：如果指向的数据小于标杆元素，则i++；直到 i 遇到 比pivot大的元素，（或出现i>=j）结束第二个循环。然后用其覆盖掉 j指向的元素。

重复两个循环直到途中i>=j，将轴点安放到ij最后共同指向的位置。递归调用函数。



最好情况：**每次划分都(接近)平均**，轴点总是(接近)中央

T(n)=2T((n-1)/2)+O(n)= 𝑂(𝑛log2⁡𝑛)

最坏情况：每次划分都极不均衡

T(n) = T(n-1)+T(0)+O(n) = O(𝑛^2)

**平均时间开销**

**𝑻\_𝒂𝒗𝒈 (𝒏) < (𝒃/𝟐+𝟐𝒄)(𝒏+𝟏)ln(𝒏+𝟏)**

**时间开销**： 对于 n 较大的平均情况而言，快速排序是“快速”的

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

**空间开销**：(递归)需要**栈空间**

快速排序是一种**不稳定**的排序方法（等大的两个数，后面的可能会被high指针扔到前面）

**4. 选择排序 (Selection Sort)**

基本思想：每一趟 (例如第 i 趟，i = 1, …, n-1) 在后面 n-i +1个待排序元素中选出关键字最小的元素，作为有序元素序列的第 i 个元素

**4.1 简单选择排序**



时间开销：比较次数O(n2) 移动次数O(n)

时间开销： 比较次数n(n-1)/2

元素移动次数 最好0次，最坏3(n-1)次（n-1趟，每趟都要交换，每次交换 涉及三个数）

空间开销：O(1) 属于就地排序

**4.2 树形选择排序(Tree Selection Sort)/锦标赛排序(Tournament Sort)**

排序过程可以用一棵有n个叶子结点的完全二叉树表示(锦标赛树)

叶节点/外结点：待排序元素(选手)

内部节点：孩子中的胜者

判定原则：关键字比较，**排序码小者胜**

**胜者树(winner tree)：**

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**

**重构胜者树**：在输出最小关键字之后，欲选出**次小关键字**，仅需将叶子结点中的最小关键字的j记录修改为一个“最大值”，然后重新比较并修改结点的关键字，次优关键字自然会上升到根结点。

时间开销： 选出最小关键字：外结点赋值O(n) 重构胜者树：一次需O(log2𝑛)

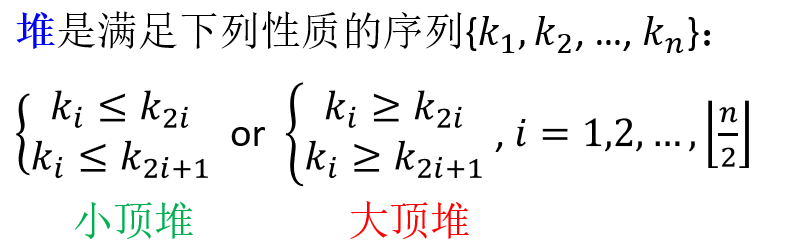
**时间复杂度为O(nlog2𝑛)**

空间开销： 空间复杂度为**O(n)**：胜者树的外结点n个，内结点n-1个

缺点：(1) **附加空间多**；(2) **需要和“最大值上界”进行多余的比较**

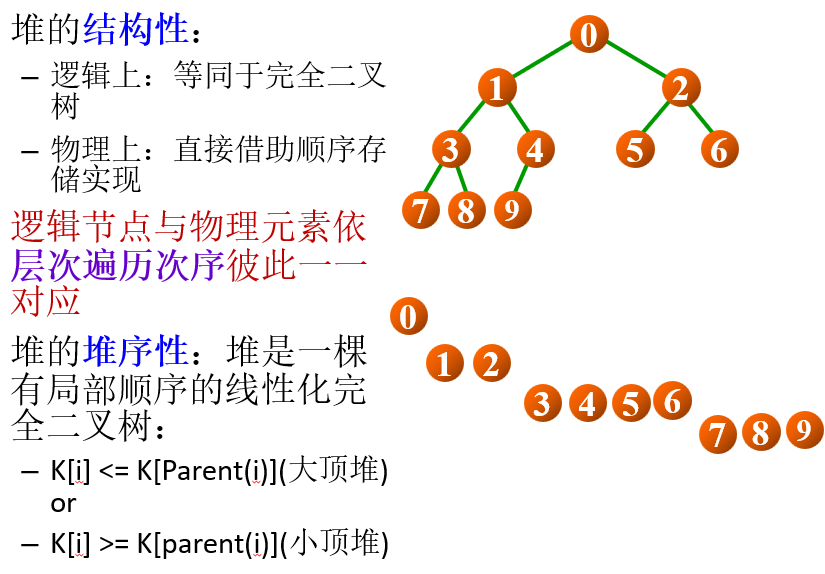
适用于：**只需要最小的前几个值时，可以减少排序操作**

**4.3 堆排序 (Heap Sort)**



堆可以看成是一颗完全二叉树的顺序存储，且此棵完全二叉树中所有非叶子结点的值均不大于(或不小于)其左、右孩子结点的值

**若序列{𝑘\_1,𝑘\_2, …, 𝑘\_𝑛}是堆，则堆顶元素𝑘\_1必定为该序列中 n个元素的最小值(或最大值)**



**堆排序**：不断输出堆的当前最小(大)值，从而实现对序列进行排序的一种排序方法

堆调整：在交换堆顶和堆底数字a后，删除堆底结点；

从堆顶结点开始，比较当前a所在结点与其左右子节点的大小，将最大的上升到a的位置（与a所在节点交换数据），直到a>左右子节点或到达叶节点。

构建堆：当前序列按照堆的逻辑组织方式看作一颗完全二叉树，（从以n/2位置的结点为根结点的子树开始，索引递减）从下往上堆这颗完全二叉树进行堆调整，使其成为小/大顶堆。

对 *n* 个关键字，建成深度为*h*(=⎣*log2n*⎦+1)的堆，所需进行的关键字比较的次数至多 4*n*

**方法1：自底向上建堆（向下调整）​​**

​**​适用场景​**​：高效构建初始堆，时间复杂度 ​**​O(n)​**​  
​**​步骤​**​：

1. ​**​定位最后一个非叶子节点​**​：  
   下标为 i = (n - 2) / 2（n 为数组长度）。
2. ​**​向前遍历至根节点​**​：  
   对每个节点执行向下调整（Heapify）。
3. ​**​向下调整逻辑​**​（以大根堆为例）：
   * 比较当前节点与左右子节点的值。
   * 若子节点更大，则与​**​最大子节点​**​交换位置。
   * 递归调整子树，直至满足堆性质或到达叶子节点

方法2：自顶向下建堆（向上调整）​​

​​适用场景​​：动态插入元素，时间复杂度 ​​O(n log n)​​。

​​步骤​​：

​​ 从第二个元素开始遍历​​（下标 i = 1 到 n-1）。

​​ 对每个元素执行向上调整​​：

比较当前节点与父节点的值。

若违反堆性质（如大根堆中当前节点 > 父节点），则交换。

重复直至根节点或满足堆性质

时间开销：堆排序的时间复杂度为O(nlogn)

空间开销：需要执行元素交换的临时空间，O(1)

堆排序是**就地排序**

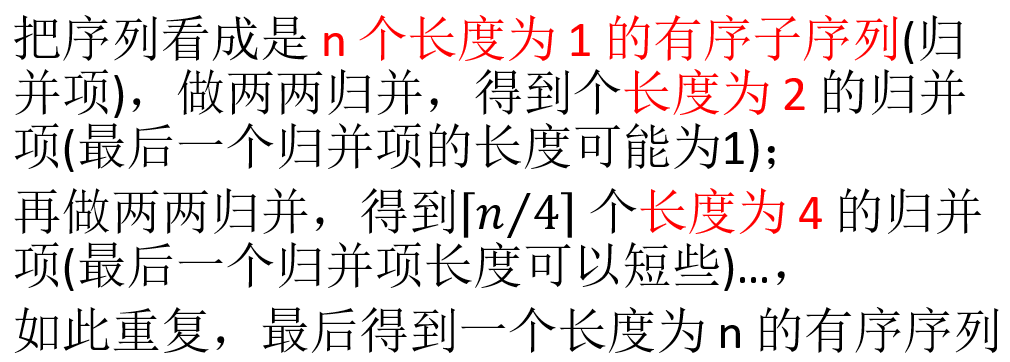
堆排序是一个**不稳定**的排序方法（同树排序）

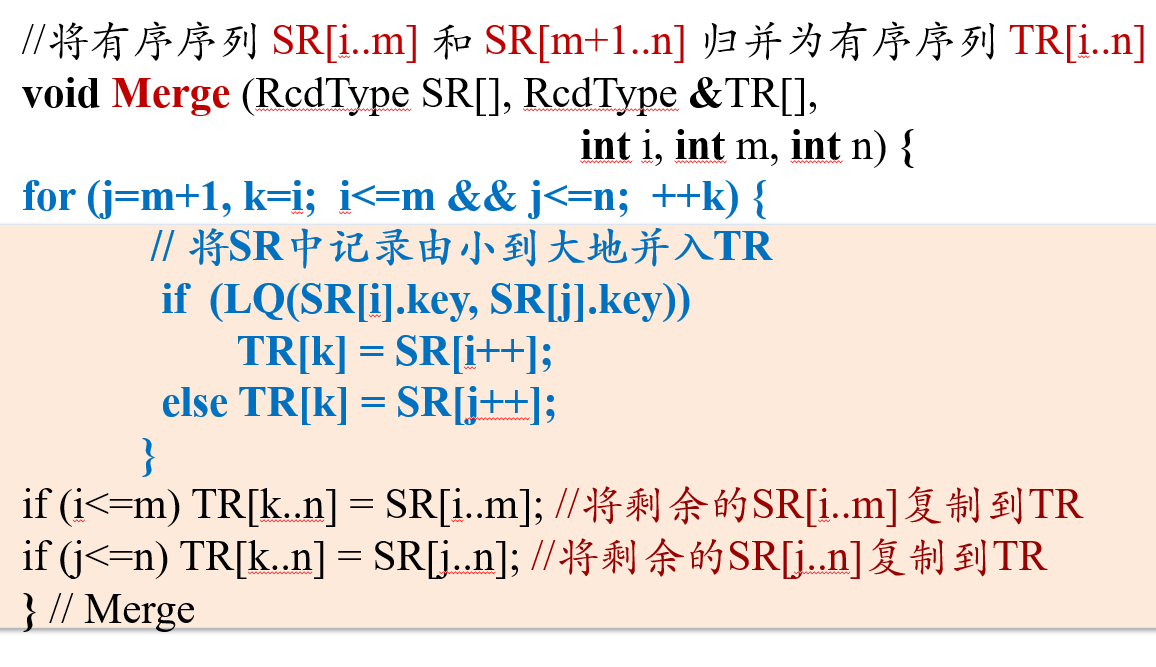
**5.归并排序**

归并(Merging or collating) 是将两个或两个以上的有序表合并成一个新的有序表

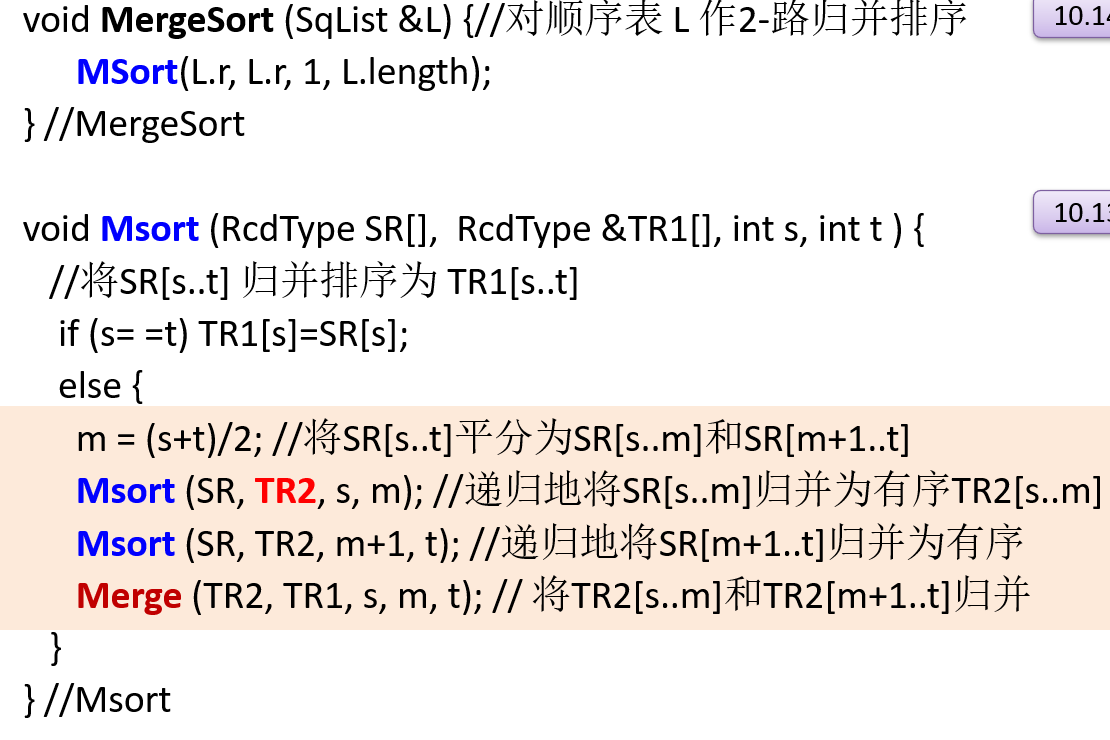
**2-路归并算法**

**非递归实现：**





**递归实现：**



时间复杂度为**Ο(nlogn)**，**最坏情况与最优情况复杂度一致**

空间开销：对等规模的辅助空间+递归开销

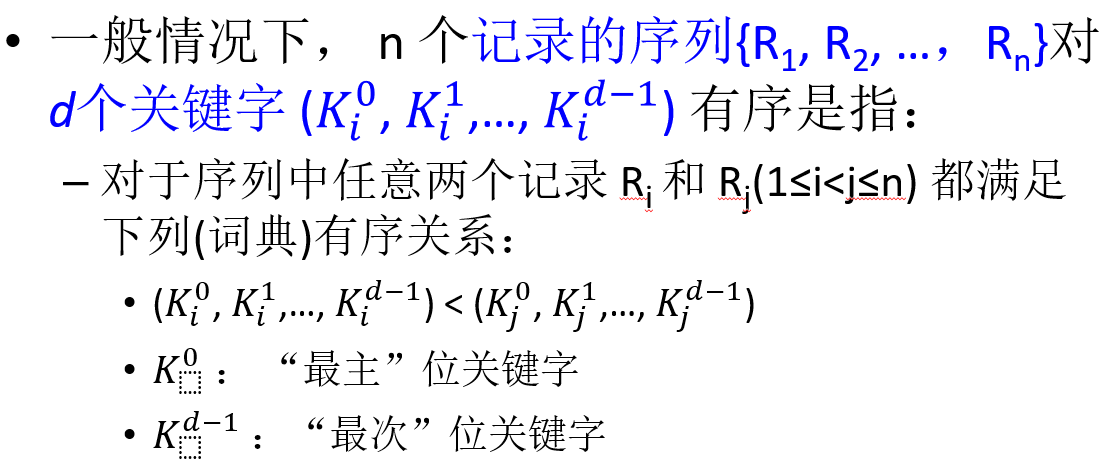
归并排序算法是**稳定**的

不需要随机读写，完全顺序访问：可扩展性极佳，十分适宜于外部排序；易于并行化

**6.基数排序**

借助“多关键字排序”的思想来实现“单关键字排序”的内部排序算法

**多关键字排序：**



**实现多关键字排序的两种方法：最高位优先法MSD、最低位优先法LSD**

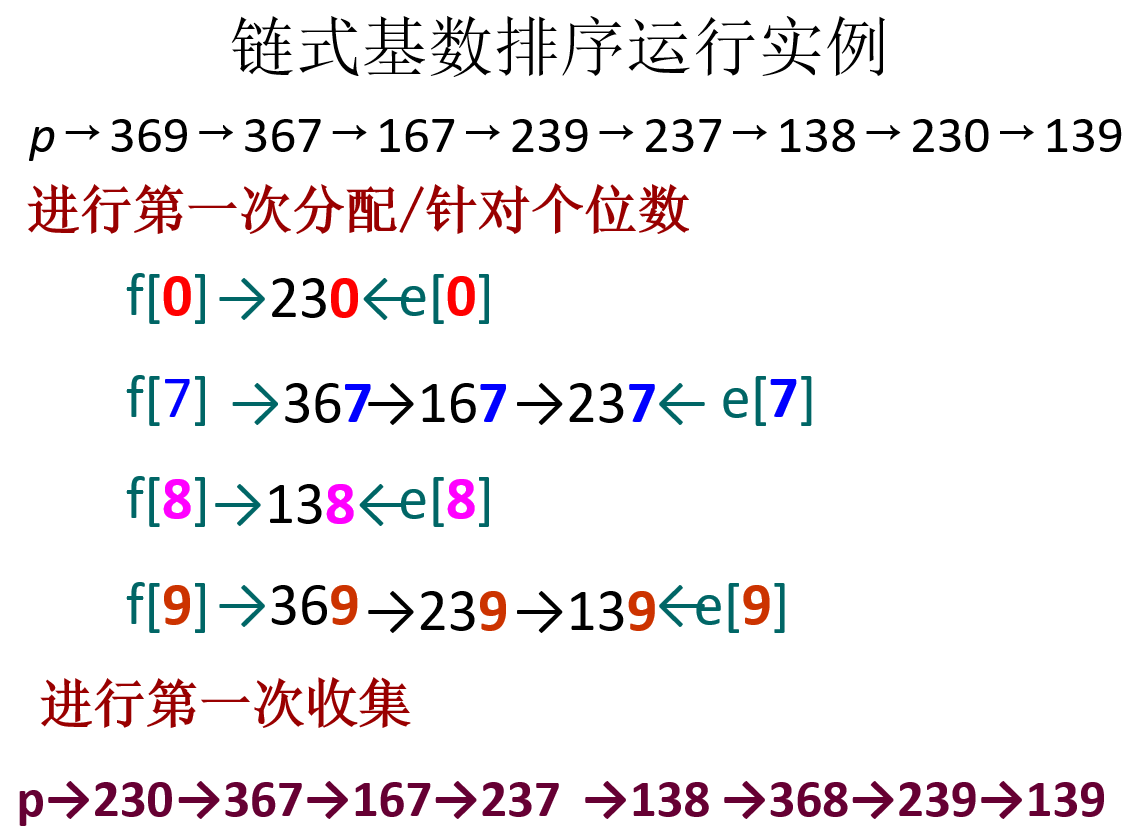


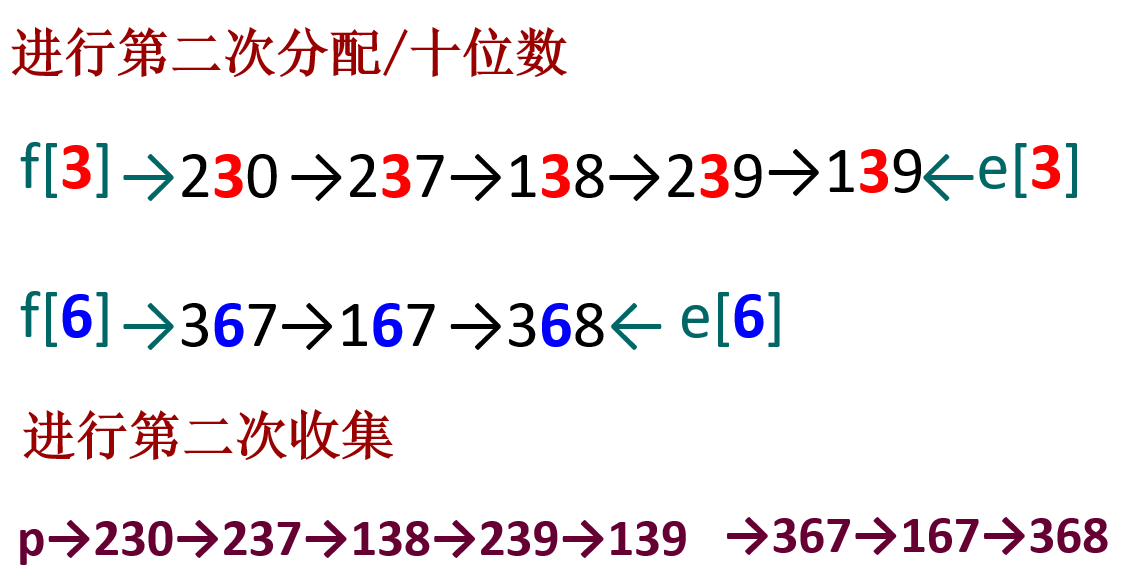
**链式基数排序**

利用“分配”和“收集”对单逻辑关键字进行LSD排序的一种内部排序方法

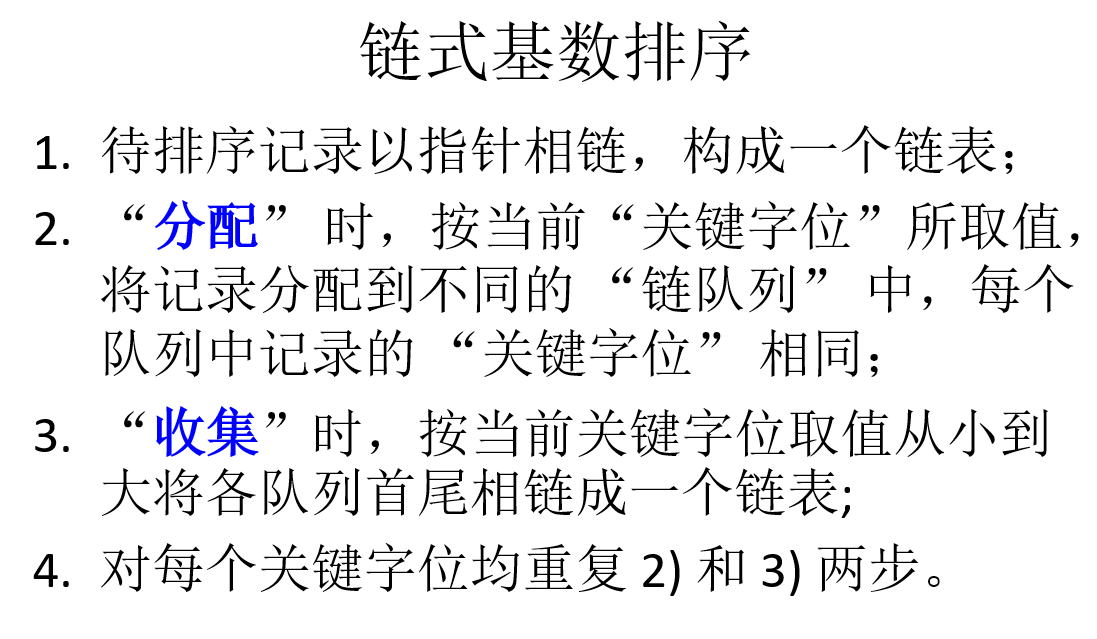
把单关键字𝐾𝑖 分解为一个d元组：

分量𝐾𝑖𝑗，也可看成是一个关键字，有radix种取值，称radix为基数









**数据结构**

**Typedef struct{**

**KeysType keys[MAX\_NUM\_OF\_KEY]; // 一个关键字的排序码序列**

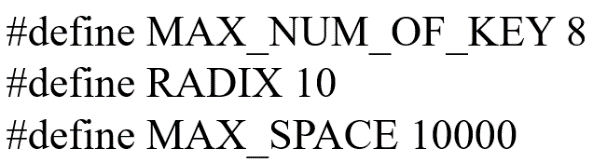
**InfoType otheritems;**

**int next;**

**}SLCell; // 一个关键字结点，next表示链表连接逻辑**

**Typedef struct{**

**SLCell r[MAX\_SPACE]; //静态链表**

 **int keynum; //记录的当前关键字的个数**

**int recnum; //静态链表的当前长度**

**}SLList;**

**Typedef int ArrType[RADIX]; // f数组，即某个关键字位的队列队头**

**// e指向关键字位队列的队尾**

若每个排序码有d位

总时间复杂度为O(d(n+radix))

若基数radix相同, 对于元素个数较多而排序码位数d较少的情况, 使用链式基数排序较好

空间开销：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

**7.各种排序方法的比较**



**基于“比较关键字”进行排序的排序方法时间复杂度的下限**

在最坏情况下达到的最好时间复杂度为 O(nlogn)

**选取排序方法的主要考虑因素**

（1）待排序的记录数目n

n 较小，用简单的排序方法(包括直接插入排序、折半插入排序、简单选择排序、起泡排序)

n 较大，用改进的排序方法(包括快速排序、堆排序、归并排序、基数排序、希尔排序)

（2）每个待排序记录的大小：与元素移动开销相关

（3）关键字的初始分布

若待排序记录基本有序，用简单的排序方法

若待排序记录随机排序，用改进的排序方法

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

（4）是否要求排序的稳定性

（5）存储结构的初始条件和要求

（6）时间复杂度、空间复杂度和开发工作的复杂程度的平衡点