第九章 排序

一.概论

前面较为系统地介绍了一些基本的数据结构。

本章集中讨论非数值程序设计中两个重要的技术问题——排序、查找中的排序问题.

排序(sorting)是计算机程序设计中的一种重要的运算，其功能是将一个数据元素的任意序列，按照要求重新排列成按一定规则有序的序列。

排序(又称分类)通常可以理解为：根据与项目中所包含的关键字或信息项的有关规则，对信息项目加以排列整理的动作过程。

二.相关概念

1.关键字(key):

目前几乎所有数据结构教科书讨论的排序都是基于关键字的，即讨论的排序关系都是以关键字为“序”组织的，对于其它，例如：基于属性的排序基本没有涉及。

如果数据元素(或结点、记录)中的某个数据项的值可以用它标识一个数据元素，则将该数据项称为关键字。

举例。

2.主、次关键字

主关键字: 可以唯一地标识一个记录的关键字称为主关键字(major key；primary key），实际应用中关键字不加声明时都指的是主关键字。

Major key:在一个记录中的最主要的关键字；

Primary key:在文件组织中,进行大量访问时所用关键字。

次关键字(或辅助关键字,secondary key):可以识别若干记录的关键字称次关键字。

实际使用中，当记录没有主关键字时，可以通过若干次关键字结合来达到主关键字的作用。

举例。2007据新民晚报报道，中国姓名前三位张伟、王伟、王芳的人数分别为29万、28万、26万

3.排序(sorting)

设含有n个记录的集合为$R=\{r_1,r_2,\ldots,r_n\}$,其对应的关键字集合为$K=\{k_1,k_2,\ldots,k_n\}$,给定关系α，按照关系α针对关键字集合K对R进行运算，使得R有如下序列:$(r_{\alpha1}, r_{\alpha2}, \ldots, r_{\alpha n})$，我们将这个操作过程称为排序。

$$R=\{r_1, r_2, \cdots, r_n\}$$

$$\Downarrow \alpha$$

$$R_\alpha=(r_{\alpha1}, r_{\alpha2}, \cdots, r_{\alpha n})$$

本质在关系α下寻找R的一种排列的过程。

但要注意的是目前大多数排序关系α是针对关键字的。

例： 对一组人员进行排序。

α:身高大小关系：

α:血缘关系：

α:性别关系：

α:肤色关系：

α:拥有的电影票与座位的关系：

现实生活中的排序例子：

4.排序的稳定性

在排序关系下，假设排序前r_i在r_j之前，排序之后领先关系不变，则称此排序过程和排序方法是稳定的，否则是不稳定的。

排序前	排序后	排序后
A_1:85	A_2:90	A_2:90
A_2 :90	A_1 :85	A_3 :85
A_3 :85	A_3 :85	A_1 :85
A_4 :78	A_4 :78	A_4 :78
	稳定的	

5.排序的分类

因此，一般数据处理中数据量都很大，数据一般都存放于外部存储器上，并以文件的形式出现。

由于排序过程中数据规模大小不同，可以将排序分为两大类

内(部)排序：将整个待排序的文件装入内存，并在其中进行排序，这种排序过程称为内(部)排序(internal sorting).

外(部)排序：由于数据规模过大，排序过程不能在内存中一次完成，需要不断进行内外存数据交换才能完成排序，这样的排序过程称外(部)排序(external sorting).

6.排序的应用

排序除在日常生活中有广泛应用外，在计算机技术中的用途也非常广泛。例如：

- 作为检索的辅助手段；
- 作为数据项匹配的手段；

另外还广泛地运用于运筹学、作业调度等重要运算中。

7.影响排序的因素

对于数据处理工作，排序是其最基本的运算之一。在现今的计算机系统中，花费在排序上的时间占系统CPU时间的比重很大，据资料统计，在一些商用计算机上，20%~60%的CPU的时间用在排序上——属于系统的核心运算。

但是一个好的排序算法的选择是很困难的，这是因为影响排序的因素很多所造成的。主要的因素有：

- 待排序的记录数目n；
- 记录的大小（规模）；
- 关键字的结构及其分布情况；
- 对排序稳定性的要求；
- 语言工具的条件；
- 存储结构；
- 计算机的硬件性能。

正是由于涉及的实际应用情况比较复杂，因此还没有一个排序算法能做到在各种情况下都是最好的，甚至很难确定在给定情况下用哪一种算法为最好。

与数据结构的评价类似，排序算法的评价标准也主要有两条：

- 算法执行所需时间；
- 算法所需附加空间；

三.存储结构的设计

排序运算可以遵循前面运用的一些存储结构，但有时为了保证高效的排序，可能还需要设计一些特殊的存储方式。

排序常用的存储结构有以下几种：

1.向量

待排序的初始文件各记录依其自然顺序存放在连续的一块地址空间中。

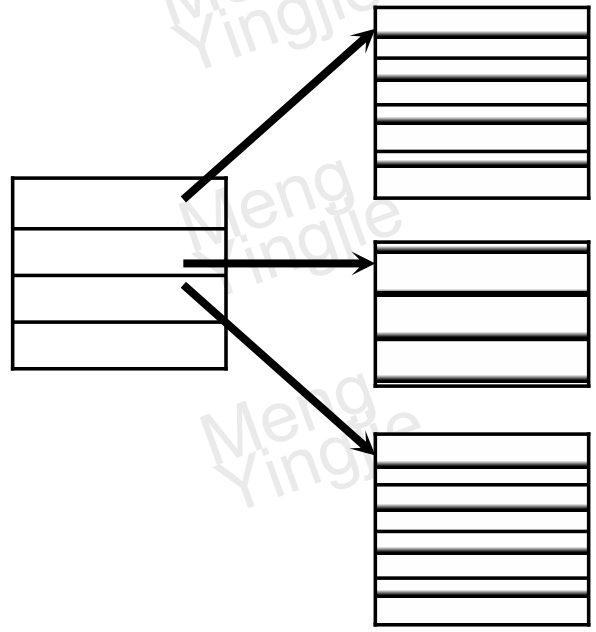
2.链表结构

记录以结点形式按记录原始次序链接起来。

3.地址向量结构

将要排序文件的各个记录存储到内存的各个块中，这些块的地址一般是不连续的。按各记录的原始次序，将这些的块的首地址依次存入内存的一块连续单元中，由各块的首地址组成了一个向量——地址向量。

这样可实现局部连续,整体不连续的组织模式，这种组织方式具有较高的实用性。



本节结束

一.插入类排序基本方法

基本思想：

每次只考虑一个待排记录 r。

将 r 按照排序关系插入到一个已经有序的文件适当位置；

重复上述过程直到全部记录插完为止。

二.直接插入排序 (straight insertion sort)

将要排序的源文件$F=R_1,R_2,\ldots,R_n$,视为两部分：

$F'=R_1$；　　$F''=R_2,\ldots,R_n$

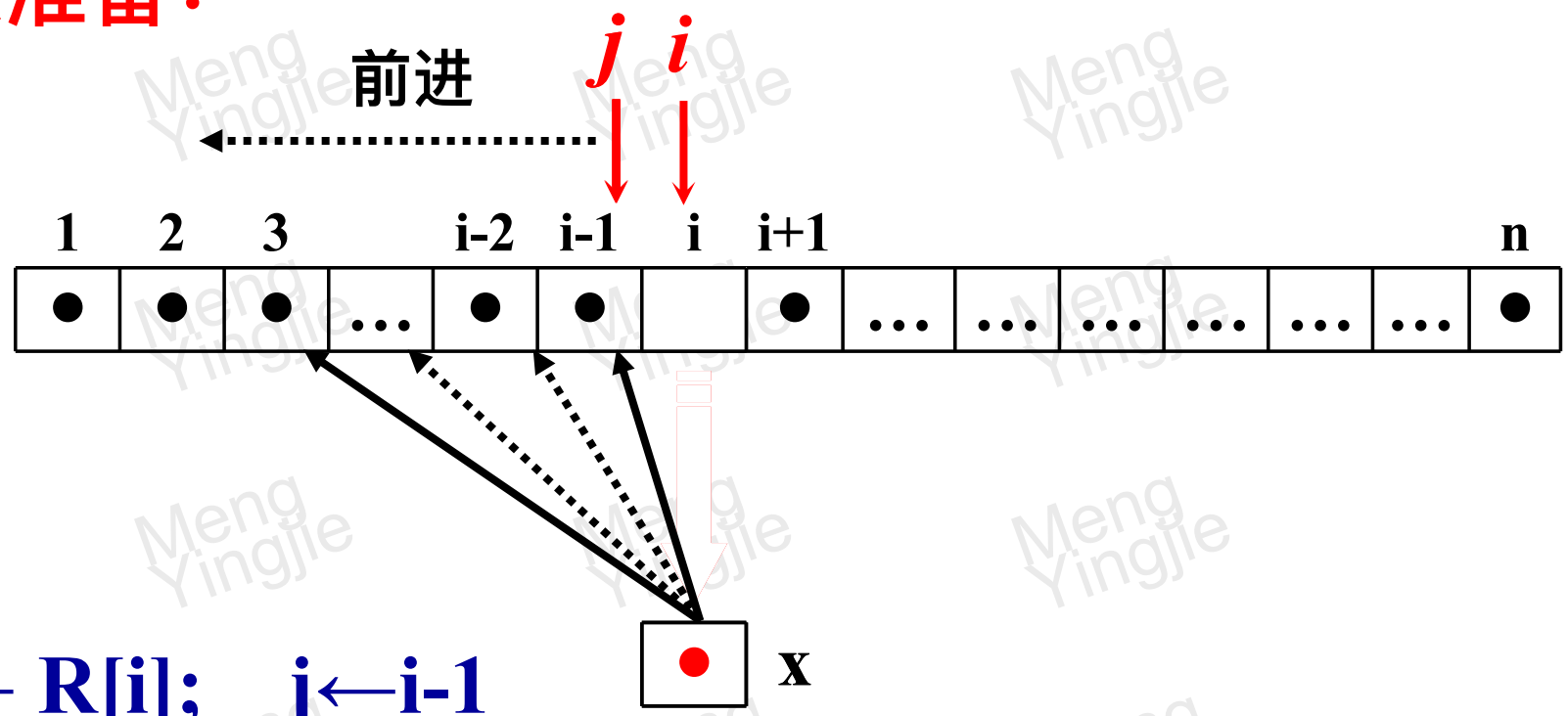
对F '和F ''重复如下工作：

①从F ''中取出一个记录R_i，并将R_i从F ''中删除；

②将R_i插入到F '并使F '线性有序。

第i个记录的插入过程：

(1)进行插入准备：



x ← R[i]; j←i-1

(2)寻找插入位置：向前进行比较，条件：

(x.key<R[j].key) and (1 ≤ j)

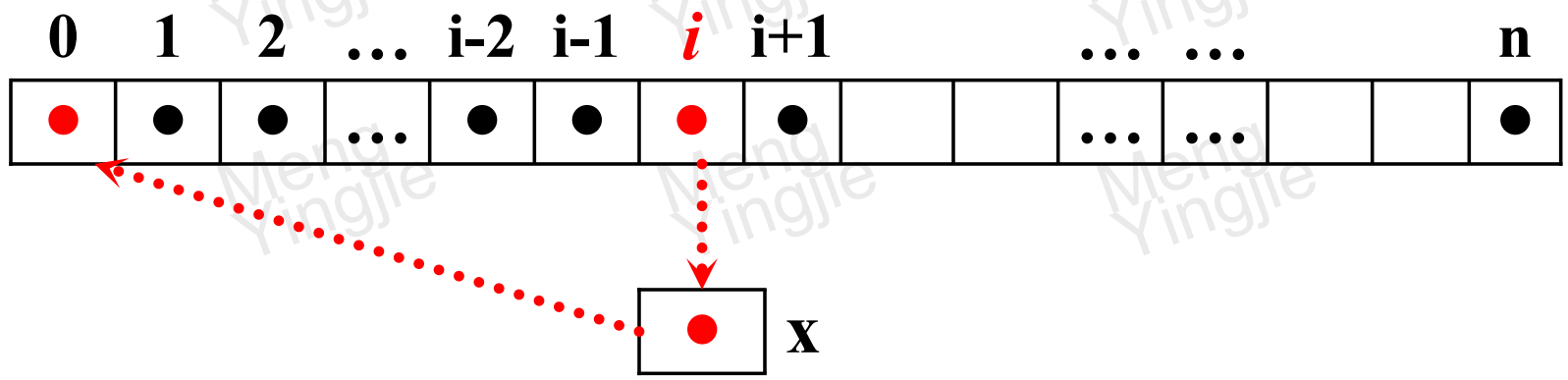
(3)插入元素归位：

R[j+1] ← x

```
PROC  StrSort(VAR R:ARRAY[1..n] OF datatype);
BEGIN   FOR  i←2  TO  n  DO
          【x←R[i];    j←i-1;
            WHILE  x.key＜R[j].key AND (1 ≤ j ) DO
              【 R[j+1]←R[j];    j←j-1】 ;
            R[j+1]←x 】
END;
```

一趟插入

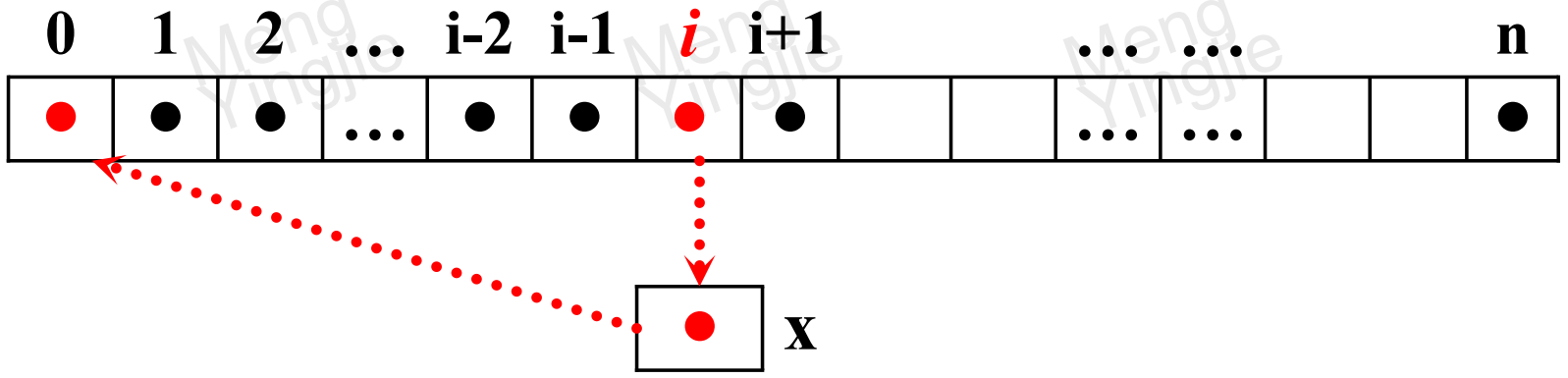
算法改进，设置0号单元，取代X单元:



直接插入算法过程：

```
PROC  StrSort(VAR R:ARRAY[0..n] OF datatype);
BEGIN
    FOR  i←2  TO  n  DO
        【R[0]←R[i];
         j←i-1;
         while  R[0].key＜R[j].key  DO
            【R[j+1]←R[j];    j←j-1】 ;
         R[j+1]←R[0]】
END;
```

一趟插入



例： 对初始关键字集{8,3,2,5,9,1,6}进行直接插入排序。

i从2到7共经过6趟插入就可完成排序。

初始状态: [8] 3 2 5 9 1 6

i=2: [3 8] 2 5 9 1 6 第1趟结果

i=3: [2 3 8] 5 9 1 6 第2趟结果

i=4: [2 3 5 8] 9 1 6 第3趟结果

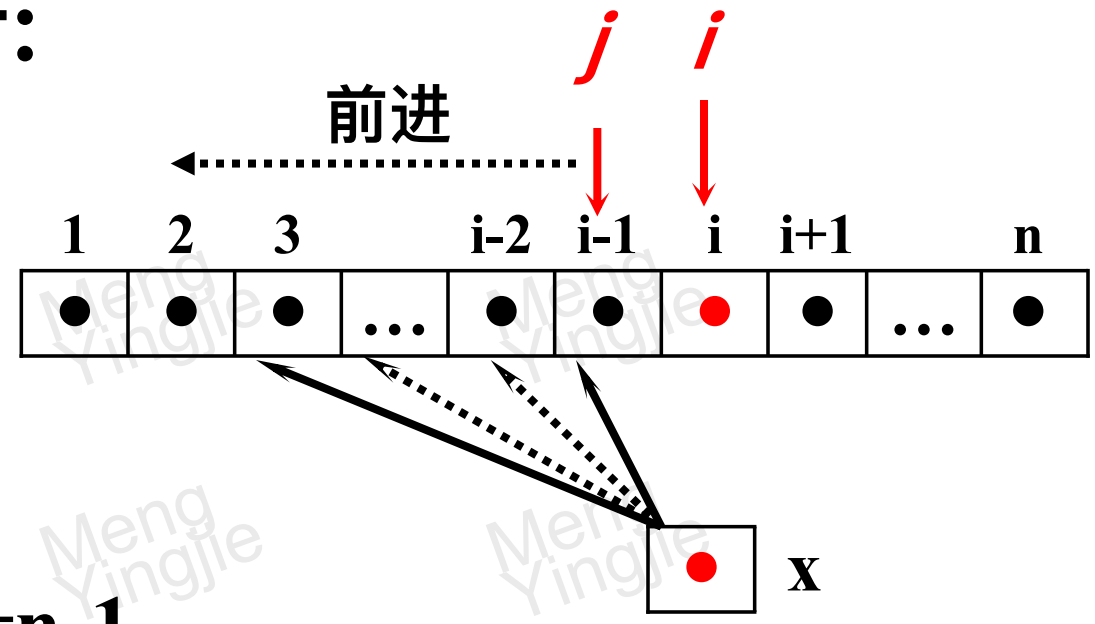
i=5: [2 3 5 8 9] 1 6 第4趟结果

i=6: [1 2 3 5 8 9] 6 第5趟结果

i=7: [1 2 3 5 6 8 9] 第6趟结果

直接插入算法的简单分析:

算法是稳定的;
(x.key<R[j].key);



算法时间复杂性分析:

比较次数:

$C_{min}=n-1$

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

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

移动次数:

$M_{min}=2(n-1)$

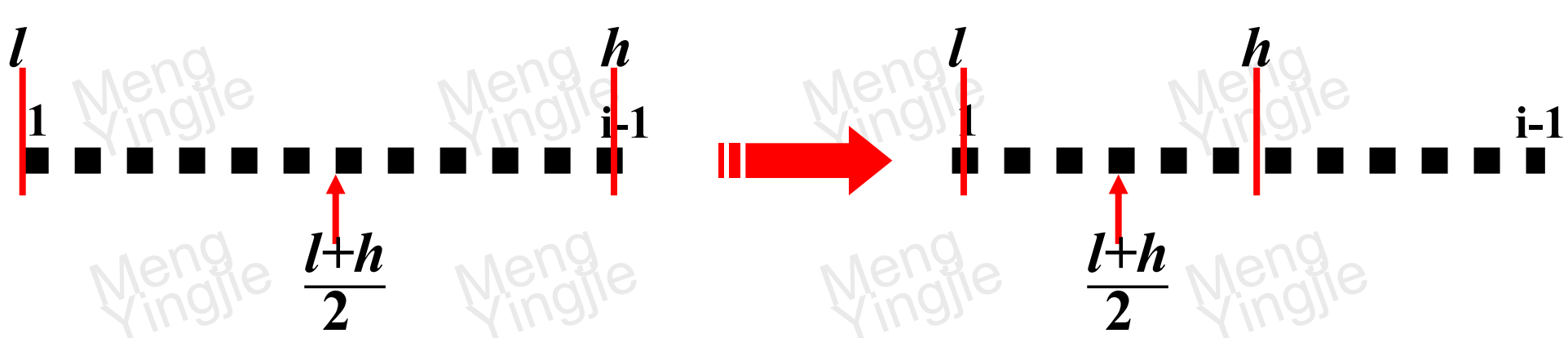
$M_{max}=\sum_{i=2}^{n} (i+1)\approx n^2/2$

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

对直接插入排序改进可以得到以下几种排序算法：

三.二分插入排序 (dichotomising insertion sort)

也称折半插入排序。与直接插入排序的区别：在插入第i个时搜索采用二分策略。

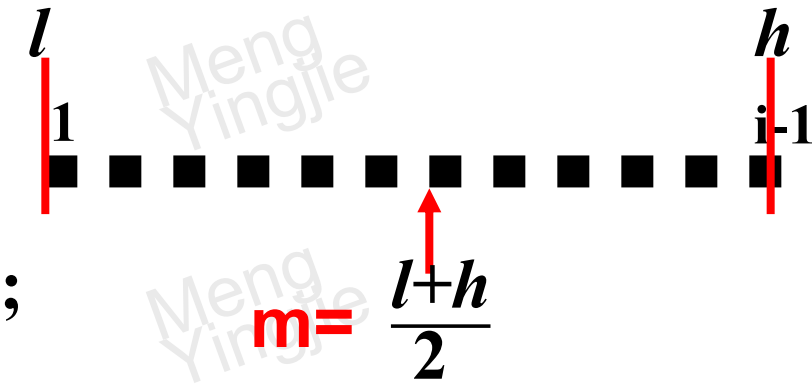


仅对比较次数有改善，移动次数无影响。算法过程如下：

二分插入算法过程：

```
PROC  DichSort(VAR R:ARRAY[1..n] OF datatype);
BEGIN
   FOR  i←2  TO  n  DO
      【x←R[i];   l←1;     h←i-1;
       while  l≤h  DO
          【m←(l+h) DIV 2;
            IF  x.key＜R[m].key  THEN  h←m-1
                                 ELSE  l←m+1】 ;
       FOR  j←i-1  DOWNTO  l  DO  R[j+1]←R[j] ;
       R[l]←x】
END;
```

一趟插入



比较时间$O(n\log_2 n)$

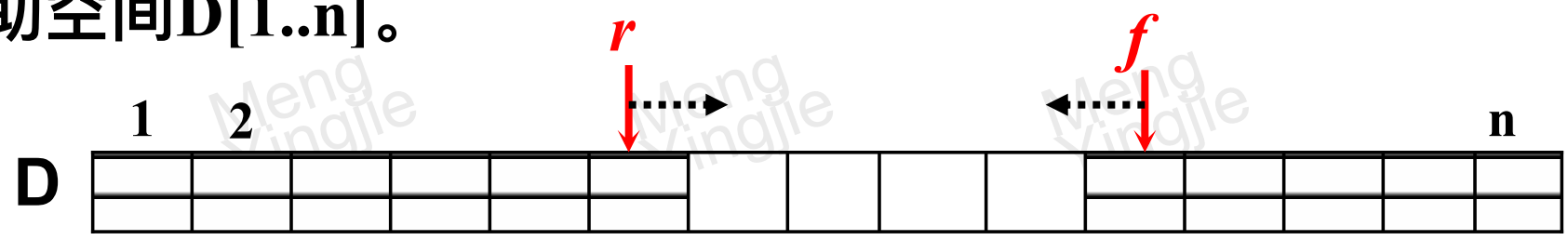
移动时间$O(n^2)$

四.2-路插入排序

在二分插入排序的基础上再改进。

目的：减少排序过程中记录的移动次数。

借助辅助空间D[1..n]。



具体做法：需选取一个轴心元素，例如就取R[1]

①R[1]⇒D[1]

②以D[1]为依据，

若 R[i].key＞D[1].key，在 1..r 之间进行二分插入排序

否则，在 f..n 之间进行二分插入排序

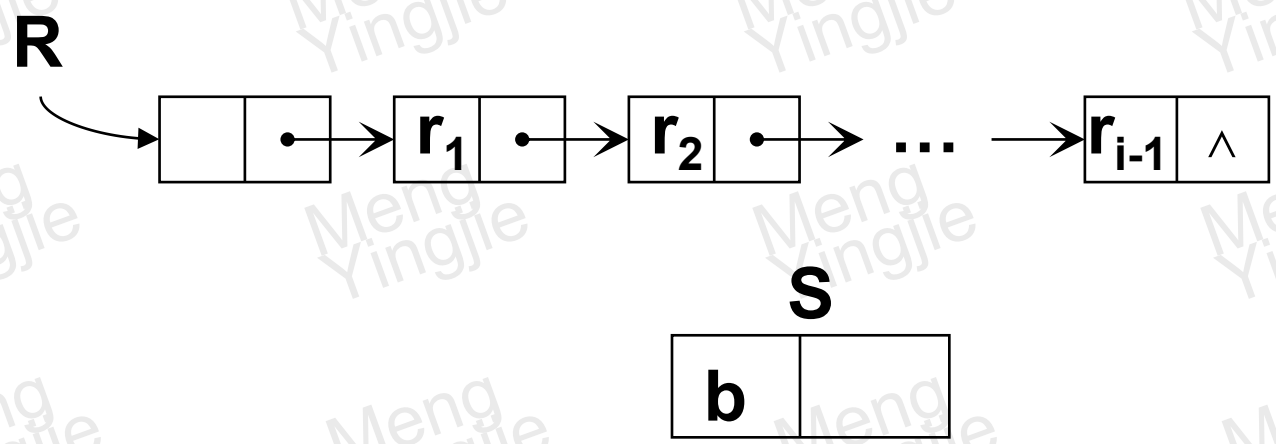
③直到R全部元素都插入到D中为至。

五.表插入排序 (list inserting sort)

采用链式结构来组织待排序的数据元素。

是一种较为古典的算法，仅适用于链结构，下完成分类。

其独到之处是利用链结构的特点，通过变更记录指针来调整记录的逻辑顺序，从而在不发生记录迁移的情况下完成分类。



提示：单链表中曾讨论过的，第1个a前插入一个结点b算法:

将 (P↑.data≠a) 换成 (P↑.data>b)

六.希尔排序 (Shell sort)

基本思想：

又称缩小增量法(diminishing increment sort),是一种快速的排序方法,1959年由D.L.Shell提出。

把对源文件F的排序分成多步来完成，

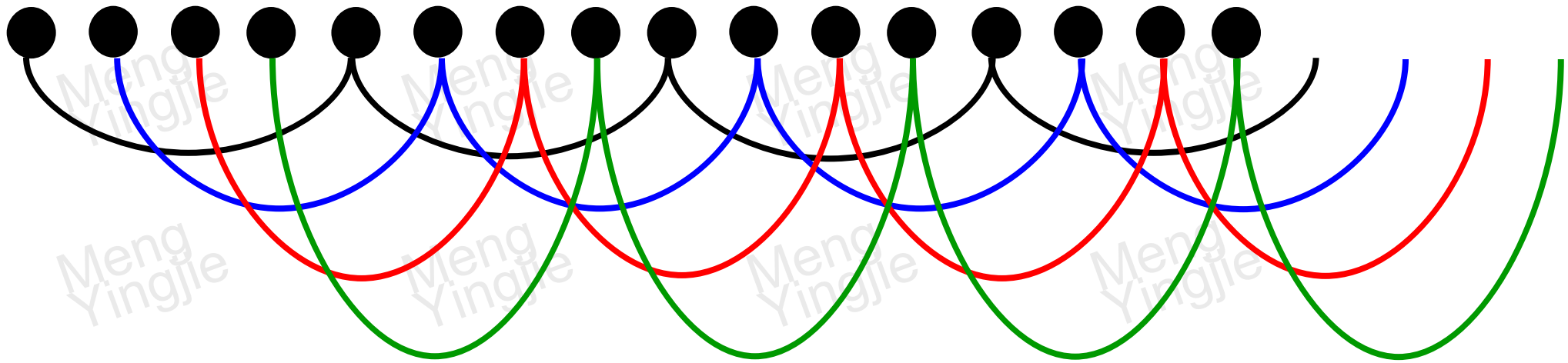
算法在每步中取一个步长 d，将F逻辑上看成 d 个文件；

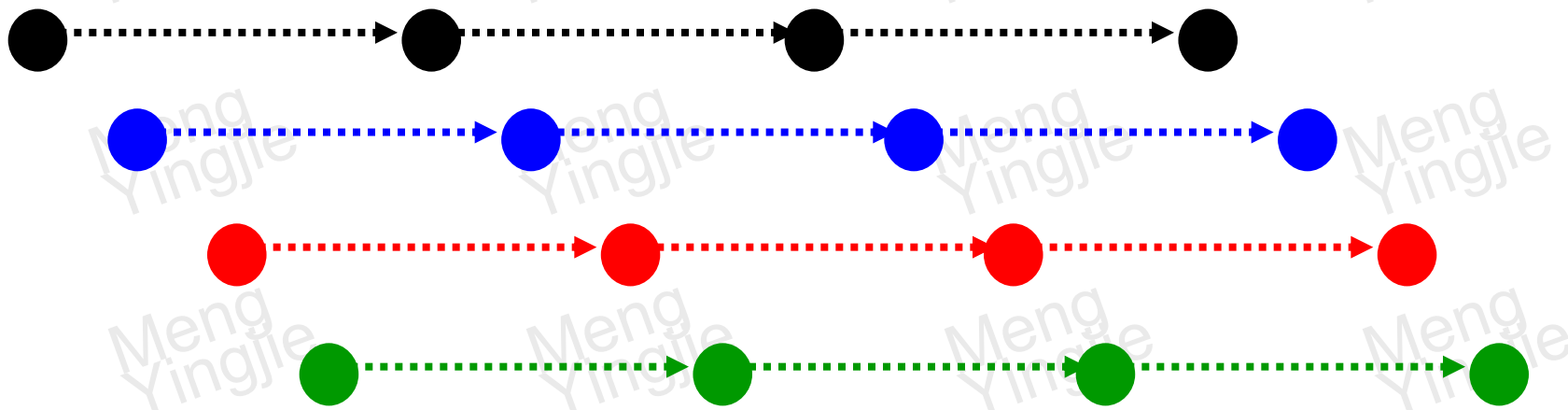
1、按照插入排序的办法把这 d 个文件分别排序；

2、 然后缩小 d ；

3、重复1-2，直到 d=1 的一次排序为止。

假设步长d=4，F被看成4个文件：





希尔排序过程：

```
PROC  ShellSort(VAR R:ARRAY[1..n] OF datatype； d:integer);
BEGIN k←d;
      REPEAT
        FOR  i←k+1 TO  n  DO
             【x←R[i];     j←i-k;
          while  (x.key＜R[j].key) AND (1≤ j)  DO
              【 R[j+k]←R[j];     j←j-k】 ;
           R[j+k]←x】 ;
        k←k DIV  2
      UNTIL  k＜1
END;
```

插入排序

希尔排序说明:

希尔排序的复杂度分析提出了一些非常困难的数学难题，在这些问题中还有许多至今仍未获得解决，尤其是，到目前为止我们还不知道如何去选择能够产生最好结果的渐进序列。该方法是不稳定的。

Shell: $d_1=\lfloor n/2 \rfloor$; $d_{i+1}=\lfloor d_i/2 \rfloor$

Kunth： $d_{i+1}=\lfloor d_{i-1}/3 \rfloor$

其所著《计算机程序设计技巧》第三卷,证明比较和移动次数为$n^{1.3}$左右。

本节结束

一.交换类排序基本方法

基本思想：

每次考虑两个待排序的记录。

依据排序关系两两比较待排序记录，并交换不满足顺序要求的那些偶对，直到全部满足为止。

二.起泡排序 (bubble sort)

[illegible]一下,把较大的结点(或者较小的结点)推到适当位置。

具体做法(以下比较指关键的比较):

先比较R_1与R_2,如果R_1大,则交换R_1和R_2；然后对R_2和R_3做同样的处理,重复此过程直到处理完R_{n-1}和R_n的比较和交换。

这样从(R_1, R_2), (R_2, R_3),直到 (R_{n-1}, R_n)的n-1次比较和交换过程我们称为一趟起泡。

一趟起泡的明显结果是将最大的传到了最后(最终位置)。显然，第二趟起泡只需进行(R_1, R_2), (R_2, R_3),…,(R_{n-2}, R_{n-1})的比较和交换。

这样最多做n-1次起泡即可得到最终有序序列。

(同样的道理也可以采用将最小的传递到最前的做法)

一趟起泡示意图： 42,20,17,13,28,14,23,15

前推最小的	初始	比较	交换
	42	42	13
	20	20	20
	17	17	17
	13	13	28
	28	28	14
	14	14	28
	23	23	15
	15	15	23

起泡算法过程：

```
PROC  BubbleSort(VAR R:ARRAY[1..n] OF datatype);
BEGIN
    FOR  i←1  TO  n-1  DO
      【flag←0;
        FOR  j←1  TO  n-i  DO
            IF  R[j+1].key＜R[j].key  THEN 【flag←1;
                                            X←R[j];
                                            R[j]←R[j+1];
                                            R[j+1]←x】 ;
        IF  flag=0  THEN    exit 】
END;
```

一趟起泡

起泡算法的简单分析:

算法是稳定的;

算法时间复杂性分析:

比较次数:
$$\begin{cases} C_{min}=n-1 \\ C_{max}=\sum_{i=1}^{n-1}(n-i)\approx n^2/2 \end{cases}$$

移动次数:
$$\begin{cases} M_{min}=0 \\ M_{max}=\sum_{i=1}^{n-1}3i\approx 3n^2/2 \end{cases}$$

算法空间复杂性分析：O(1)

三.快速排序 (quick sort)

1.概述

也称分区交换排序，或Hoare排序。1962年首先由霍尔(C.A.R.Hoare)提出。

是至今为止内部(比较)排序中较快的一种，它有广泛的应用，典型的应用是UNIX系统调用库函数例程中的qsort函数。

[illegible]性能而在某些应用中无法采用。

[illegible]主要特点是采取了分治的思想。

编写:蒙应杰

2.快排基本思想:

(以下比较指关键的比较，关键字指排序关键字)

在待排序的n个记录中任取一个记录r(例如就取第1个)，作为轴心元素；

以r为标准将所有记录分为两组；

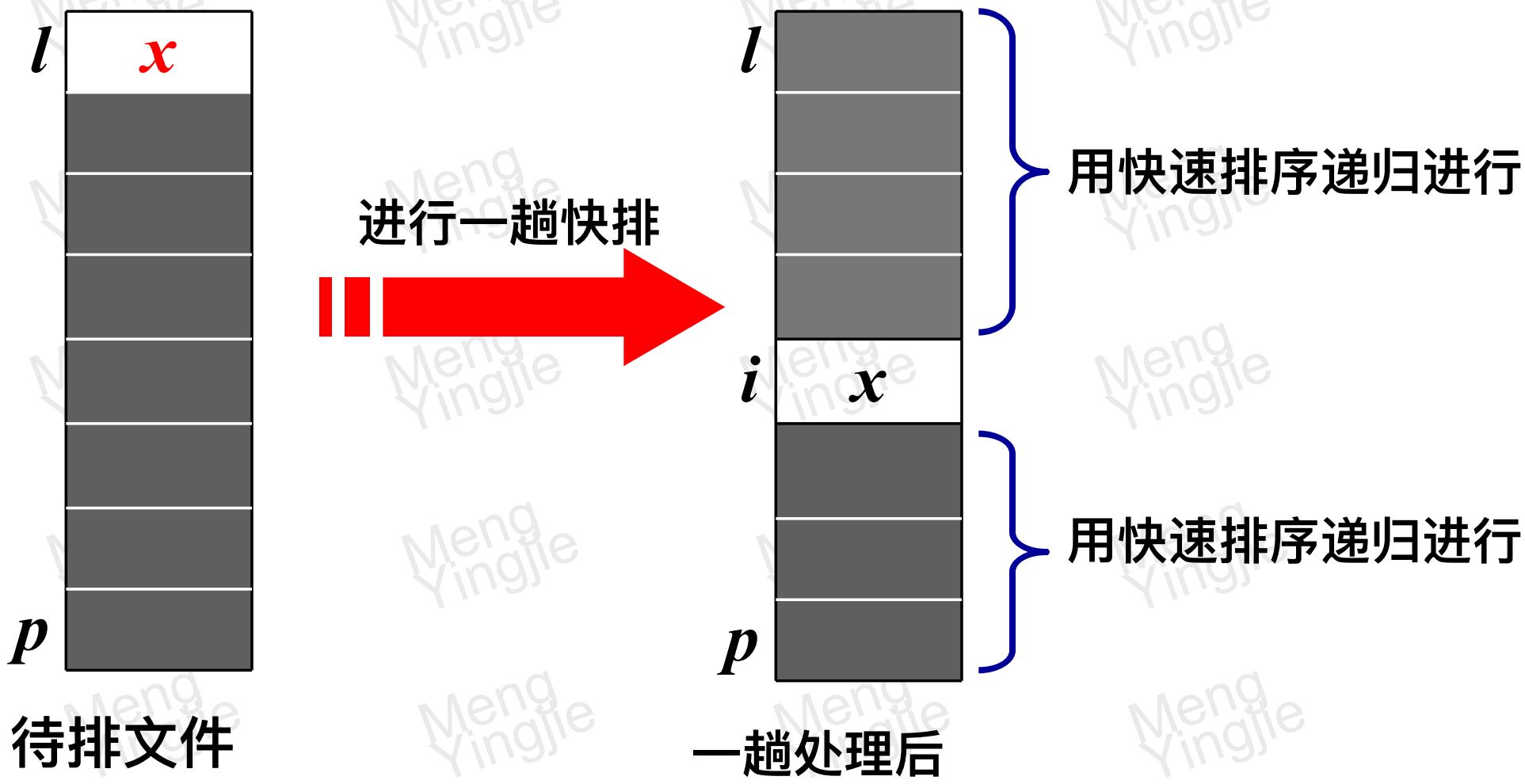
第1组中各记录的关键字都小于r的关键字；

第2组中各记录的关键字都大于r的关键字；

并把r排在这两组中间(最终位置),这一过程称为一趟快排。

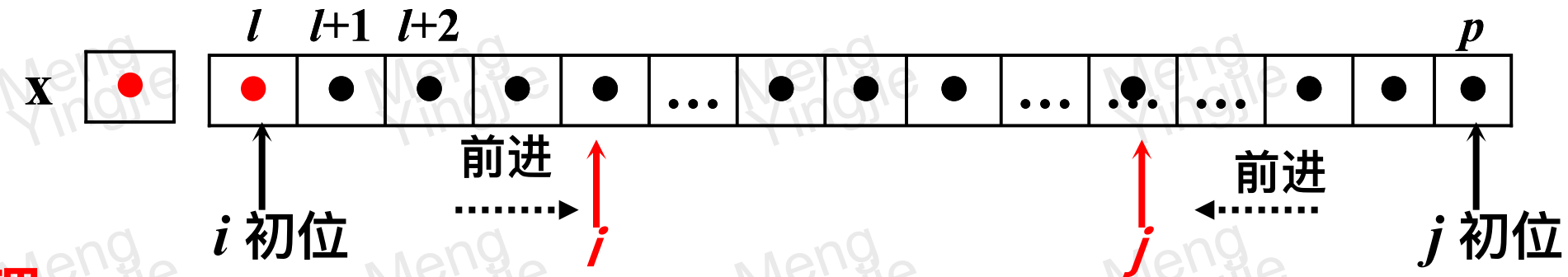
然后对这两组分别重复上述方法，直到所有记录都排在应有位置上。

快速排序示意图:



3.一趟快排描述

初始准备：



处理：

(1).当(i<j)且R[j].key＞x.key时，j指针前进；

(2).如果i＜j，则：

①R[j]⇒R[i]；i+1⇒i

②当(i<j)且R[i].key＜x.key时，i指针前进；

③如果 i＜j，则：R[i]⇒R[j]；j-1⇒j

(3) 如果 i=j，则本趟结束，否则转(1)

(4)以上结束后： x ⇒ R[i]

一趟快排示意图： 46,55,13,42,94,05,17,70

46	55	13	42	94	05	17	70

初始状态：

x

46	17	05	13	42	46	94	55	70

4.快排算法过程：

```
PROC  QuickSort(VAR R:ARRAY[1..n] OF datatype;l,p:integer);
BEGIN IF  l≥p  THEN  exit ;
      i←l;     j←p;      x←R[i];
      REPEAT
          WHILE (x.key≤R[j].key)AND(i＜j) DO  j←j-1;
          IF i＜j THEN 【R[i]←R[j];    i←i+1;
                        WHILE (R[i].key＜x.key)AND(i＜j) DO
                            i←i+1;
                        IF i＜j THEN 【R[j]←R[i];   j←j-1;】】
      UNTIL  i=j;
      R[i]←x;
      i←i+1;      j←j-1;
      IF l＜j THEN  CALL QuickSort(R, l , j);
      IF i＜p THEN  CALL QuickSort(R, i , p);
END;
```

一趟快排

快速排序的简单分析:

算法是不稳定的;

算法空间复杂性:

可将算法改为非递归进行分析;

空间主要是栈的递归调用深度，最多不会超过n;

如果每次都选较长的部分进栈，处理较短的部分，递归深度最多不超过$\log_2 n$，也就是说快速排序需要的附加存储开销为$O(\log_2 n)$.

快速排序的时间复杂性:

最坏情况：待排序文件基本有序。经轴心结点分割得到一个空集，另一个是剩下的其它元素. 总比较次数为：$\sum_{i=1}^{n}(n-i)\approx n^2/2$

最理想情况：经轴心元素分割后得到两个长度基本相等的集合.

总比较次数为：

$$\begin{aligned} C(n) &\le n+2C(n/2) \\ &\le 2n+4C(n/4) \\ &\le 3n+8C(n/8) \\ &\le \cdots \\ &\le n\log_2 n+nC(1) \\ &= O(n\log_2 n) \end{aligned}$$

可以证明平均比较次数也是$O(n\log_2 n)$.快速排序的记录移动次数小于等于比较次数，因此，快速排序的总执行时间为$O(n\log_2 n)$.

本节结束

一.选择类排序基本方法

基本思想：

每次考虑一个数据元素的最终位置。

每步从待排序的记录中挑选一个当前最小(或最大)的结点，将其放在应有位置，直到全部记录排完为止。

二.直接选择排序 (straight selection sort)

是一种简单的排序方法。

具体做法(以下比较指关键的比较)：

首先在所有记录中选出最小的记录；

把它与第一记录交换；

然后在剩下的记录中再送出(全局)次最小的记录与第二个记录进行交换；

依次类推，直到全部记录排完为止。

例： 对初始关键字集{8,3,2,5,9,1,6}进行直接选择排序。

共经过6趟选择就可完成排序。

初始状态：	8	3	2	5	9	1	6	
	[1]	3	2	5	9	8	6	第1趟结果
	[1	2]	3	5	9	8	6	第2趟结果
	[1	2	3]	5	9	8	6	第3趟结果
	[1	2	3	5]	9	8	6	第4趟结果
	[1	2	3	5	6]	8	9	第5趟结果
	[1	2	3	5	6	8]	9	第6趟结果
	[1	2	3	5	6	8	9]	最终结果

直接选择算法过程：

```
PROC  SeleSort(VAR R:ARRAY[1..n] OF datatype);
BEGIN
    FOR  i←1  TO  n-1  DO
        【k←i;    {k为当前最小者的下标编号 }
          FOR  j←i+1  TO  n  DO
              IF  R[j].key＜R[k].key  THEN  k←j;
          IF  i≠k  THEN 【 x←R[k];
                          R[k]←R[i];
                          R[i]←x】】
END;
```

一趟选择

直接选择算法的简单分析:

比较次数与待排序记录初始顺序无关。

算法时间复杂性分析:

比较次数: $C=\sum_{i=1}^{n-1}(n-i)\approx n^2/2$

移动次数:
$$\begin{cases} M_{min}=0 \\ M_{max}=3(n-1) \end{cases}$$

空间复杂性:O(1)

算法是不稳定的。

三.树形选择排序 (tree selection sort)

背景:

充分利用前一趟选择的中间结果,目的是希望减少以后选择的比较次数。

例如，在第一趟选择中进行的许多次比较，在第二趟比较中并不需要再进行。

例如，对(3,7,2,4,1)进行第一趟选择要进行的比较有：

<3,7>,<3,2>,<2,4>,<2,1>， 排序结果:(1,7,2,4,3)

第二趟比较有,<7,2>,<2,4>,<2,3>

树形选择的具体做法:

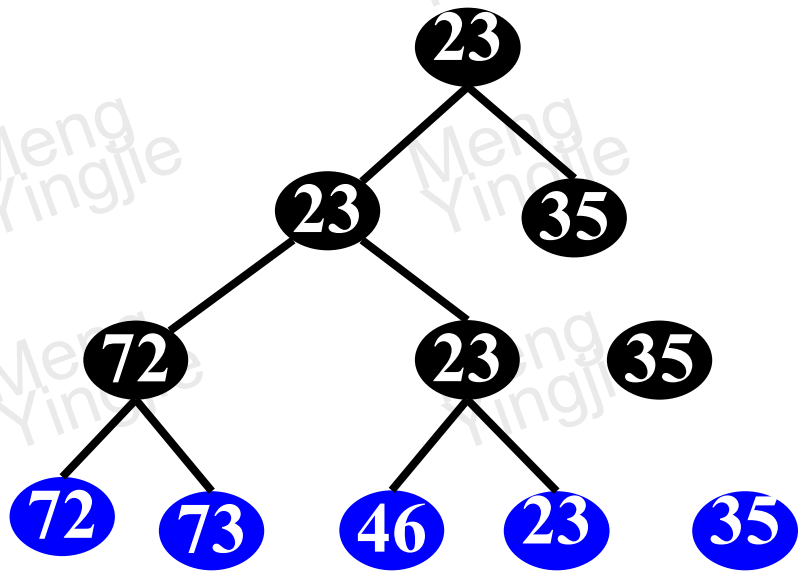
把n个待排序的记录两两比较，取出⌊n/2⌋个较小者作为第一步的结果保留下来。

再把这⌊n/2⌋个较小者两两比较……,如此反复，

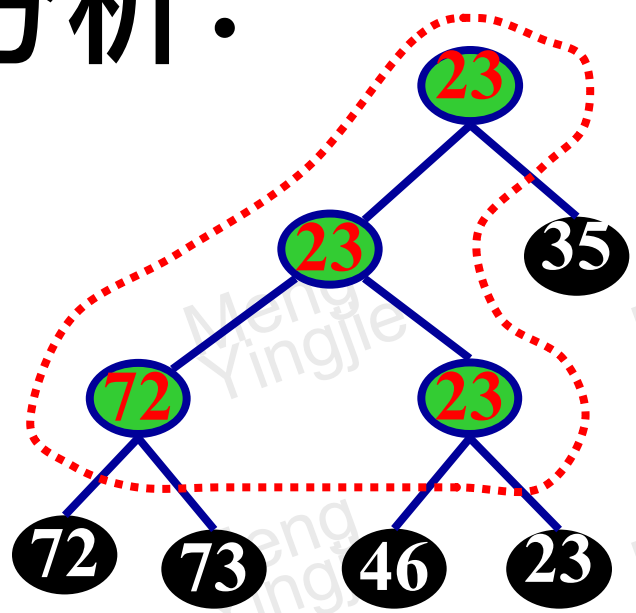
直到比较出最小的为止。

(类似于锦标赛过程,也称锦标赛排序,tournament sort)

例如:



分析：



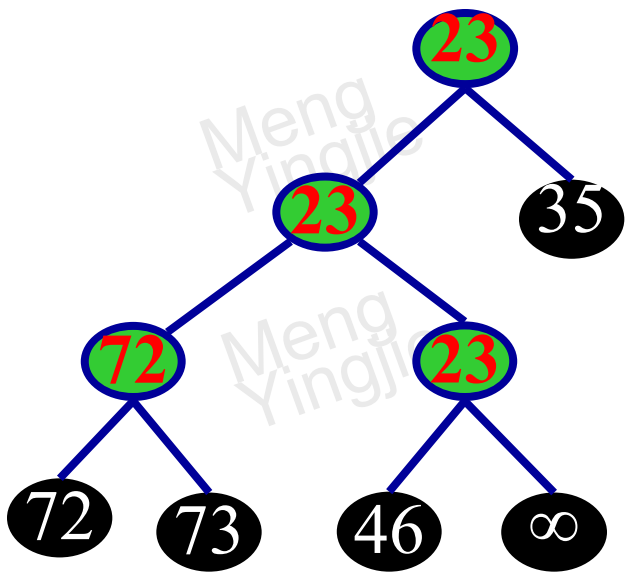
产生最小结点后，将其原来所在位置值改为∞，

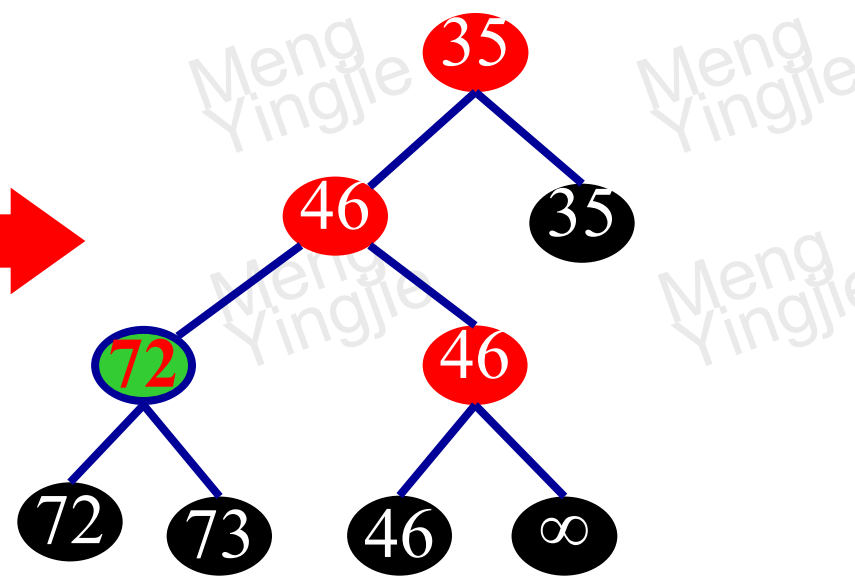
再在延此修改上升结点即可产生第二趟的最小结点。

第二趟最小元的选择只需进行$\log_2 n$次比较。

总比较次数: $(n-1)+(n-2)\log_2 n$

空间需要: n–1(根据二叉树性质)





四.堆排序 (heap sort)

背景:

为了克服树形选择排序的缺陷,威廉姆斯(J.Willioms)和弗洛伊德(Floyd)在1964年对树形选择排序提出了改进，使其比较次数达到了树形选择排序的量级，同时又不需要增加额外的单元。

该方法称为堆(选)排序。

1.定义:

设L是长度为n的表，当1≤i≤⌊n/2⌋时，其数据元素满足：

$$L(i)\le L(2i) \text{ 且 } L(i)\le L(2i+1),$$

则称L是一个**堆**。

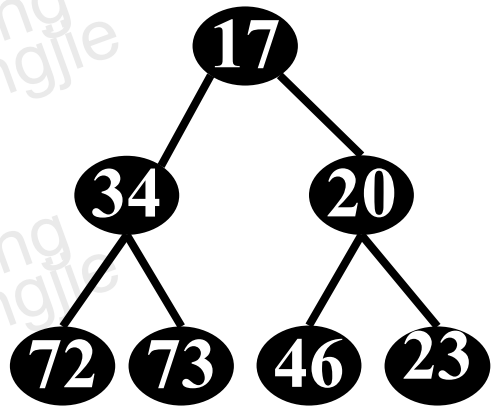
其中：将 L(1)称为**堆顶**

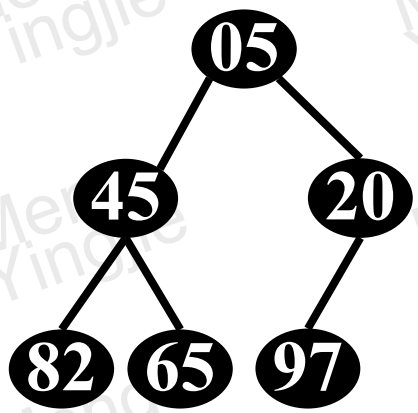
L(n)称为**堆底**

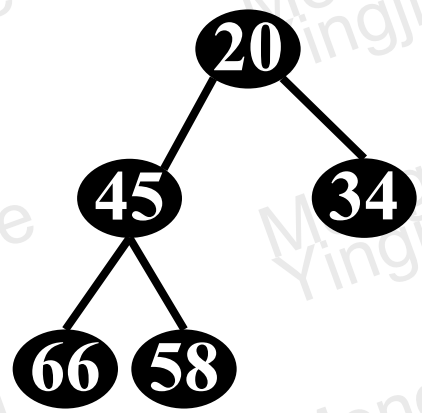
举例:

L[1..7]

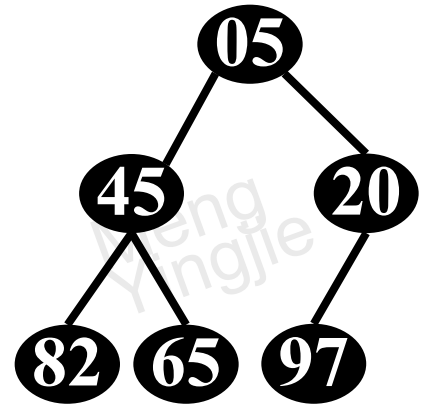
1	2	3	4	5	6	7
17	34	20	72	73	46	23







2.堆的特性:



①堆顶存放的是最小元素(或最大);

②线性表中蕴涵完全二叉树(即顺序二叉树)。具备其所有特性。

例如: 深度为$\lfloor \log_2 n \rfloor$;

每个结点的左右子树深度相差最多为1;

每片叶子位于相邻深度;

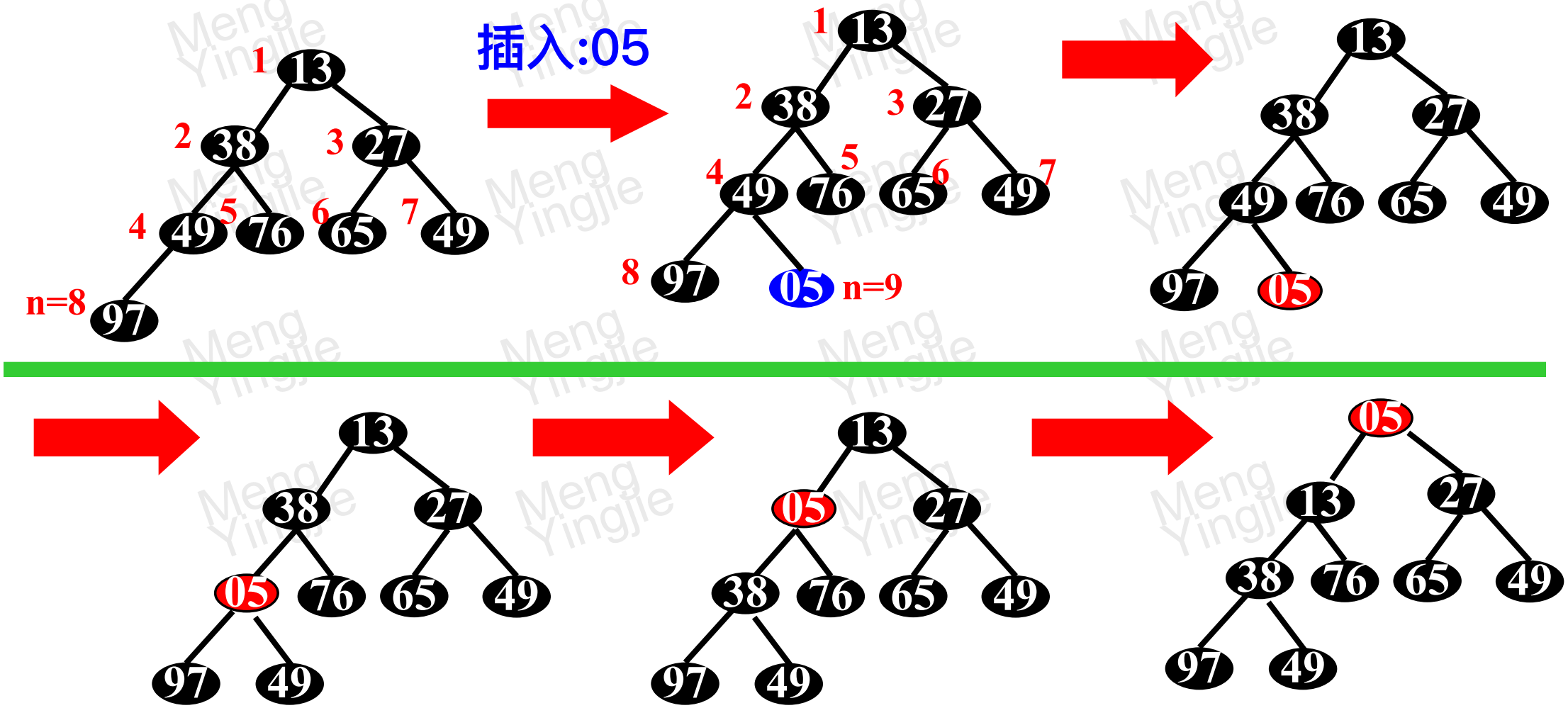
结点关系(父子)可通过计算获取;

实际上是以顺序方式存储的结点具备一定性质的一棵完全二叉树。

3.在堆中插入结点:

基本方法: ①连接: (堆底，数组尾部)

②调整: (父比子大交换)



插入过程的算法设计：

设堆A[1..max]采用小堆顶,当前堆底为 n, 插入数据是 x

(1)连接

n+1=>n;　　x=>A[n]

(2)调整 (往上爬，爬不动就结束)

获取 x初始位置：　n=>i

当 i>1 做:

如果　A(⌊i/2⌋)<A(i)　则: 结束

否则: ①交换: A(⌊i/2⌋), A(i)

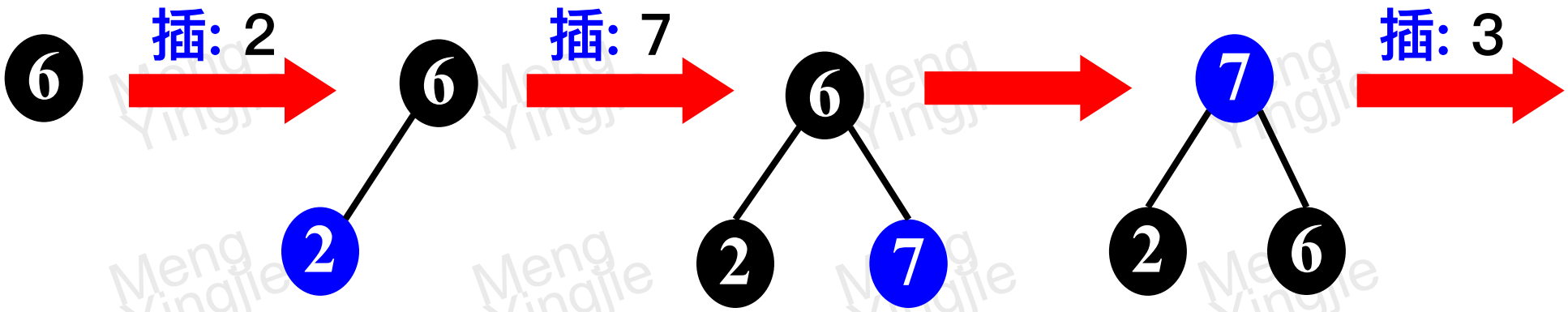
②⌊i/2⌋=>i

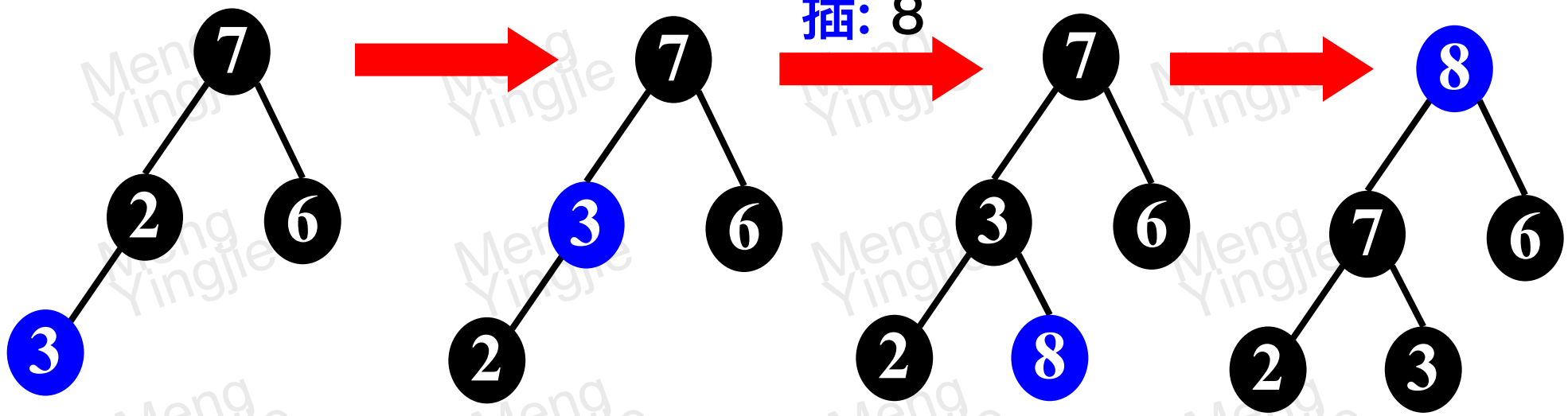
插入时的调整过程：

```
PROC  InsertHeap(VAR A:ARRAY[1..max] OF datatype;n:integer);
BEGIN   i←n;
        WHILE  i>1  DO
            IF  (A[i DIV 2].key<A[i].key)
              THEN  exit;
              ELSE 【swop(A[i DIV 2], A[i])   {交换数组两个元素}
                    i← i DIV  2 】
END;
```

最坏比较和交换次数是走一个二叉树的深度。

利用插入建堆： {6,2,7,3,8,4,5,9},若堆顶放最大元





插入建堆算法过程：

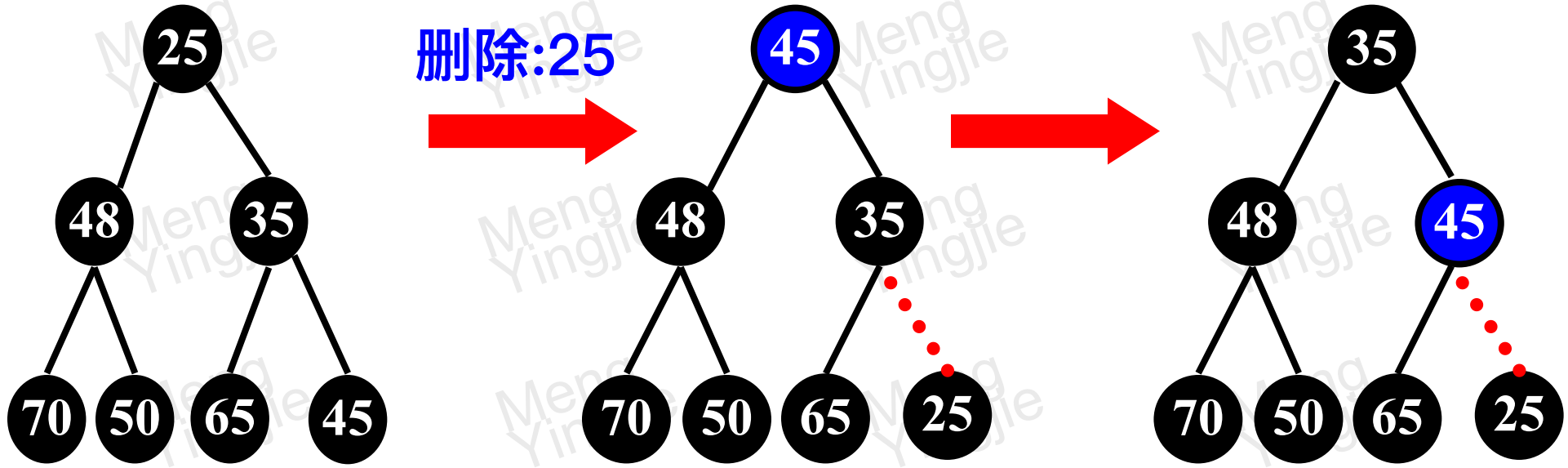
```
PROC  InsBuildHeap(VAR A:ARRAY[1..max] OF datatype;n:integer);
{建立有n个元素的堆}
BEGIN   read(A[1]);
        FOR  i←2  TO  n  DO
        【read(A[i]);
          InsertHeap(A,i)】 ;
END;
```

时间复杂度:$O(n\log_2 n)$

4.在堆中删除结点:

基本方法: ①交换: (与堆底元素)

②调整(筛选): (上升元下降)



删除过程的算法设计：

设堆A[1..max]采用小堆顶,当前堆顶为 i, 堆底为 k

(1)交换

（ A[i]，A[k]）；k-1=>k

(2)筛选 (退位，退不动就结束)

a. (双孩子)当 2i+1＜k 做:

①选 A(2i), A(2i+1) 中较小者的下标为m

②若A(i)＜A(m) ,则不用下降，结束

否则: 【交换: A(i), A(m)； m =>i 】

b. (结束时单孩子判定) 2i=k 时比较A(i)＞A(2i) ,A(i)下降

筛选的处理过程：

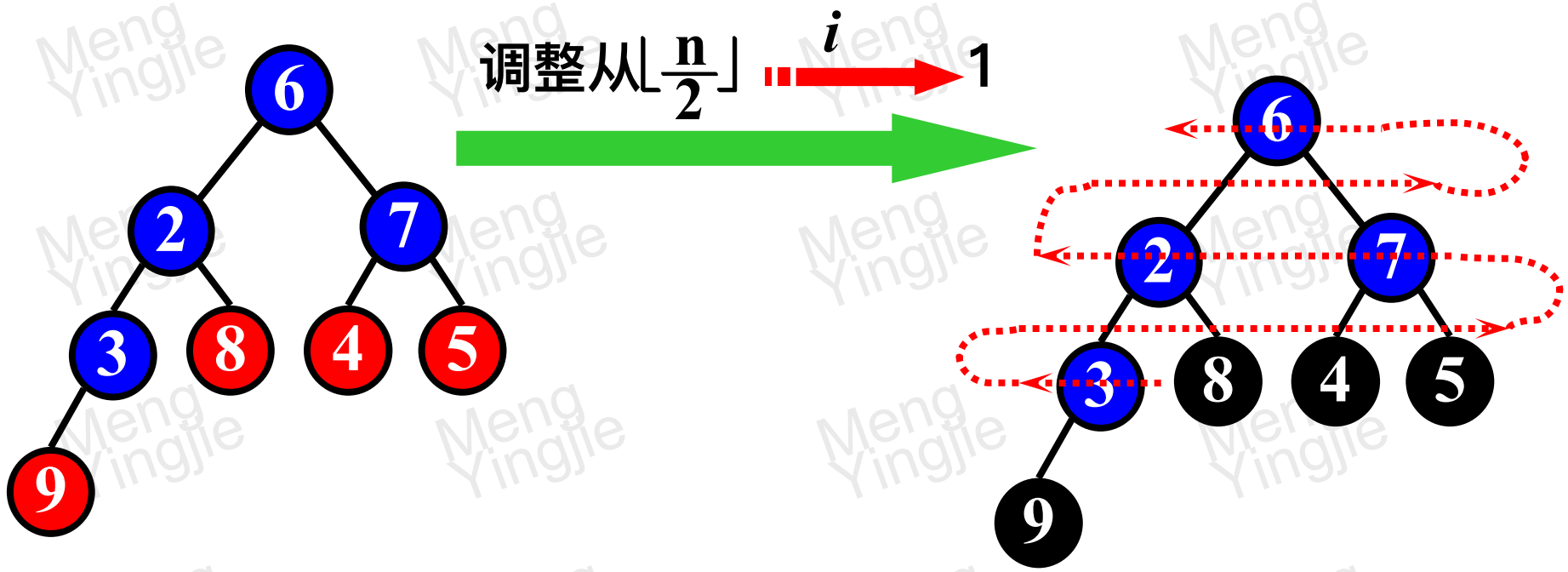
```
PROC  HeapRectify(VAR A:ARRAY[1..max] OF datatype;i,k:integer);
{i位堆顶，k为堆底}
BEGIN
      WHILE  2i+1≤k  DO
          【min(A[2i], A[2i+1], m); {m为A(2i)A(2i+1)中较小者下标}
           IF  (A[i].key＞A[m].key)
                  THEN   swop(A[i], A[m])
                   ELSE   exit ;
          i←m】  ;
     IF  (2i=k) AND (A[i].key＞A[k].key) THEN   swop(A[i],A[k])
END;
```

最坏比较和交换次数是走一个二叉树的深度。

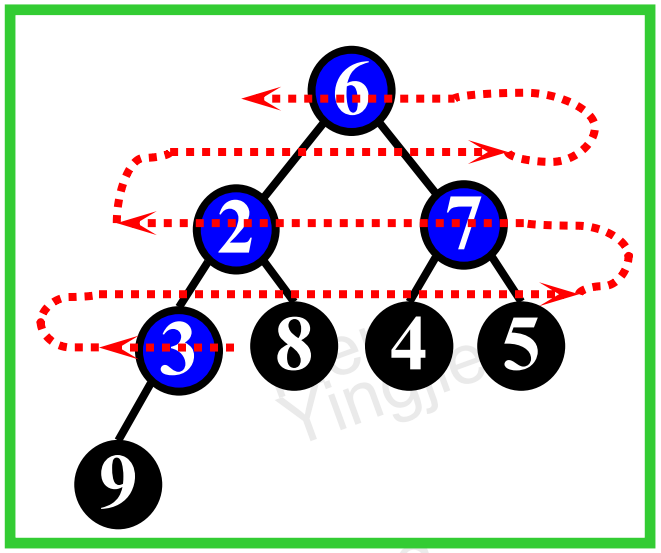
利用删除的调整过程建堆——筛选建堆

{6,2,7,3,8,4,5,9},若堆顶放最小元。

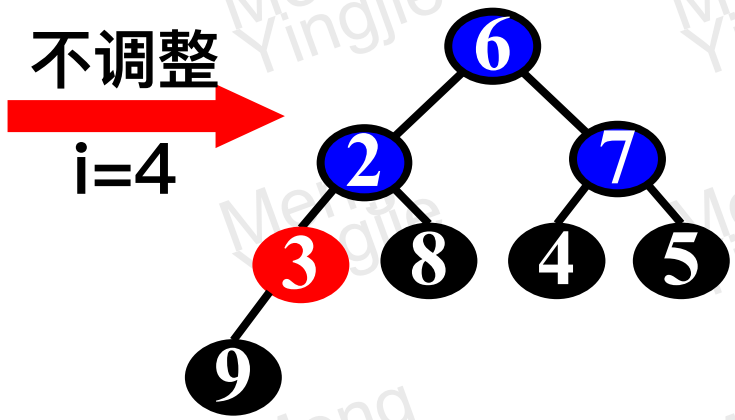
初始状态

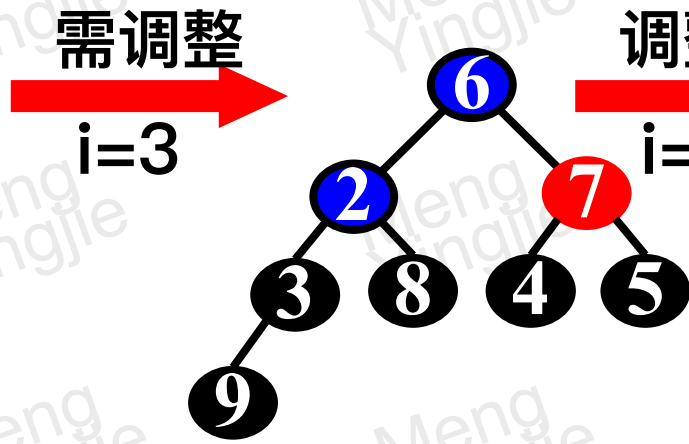


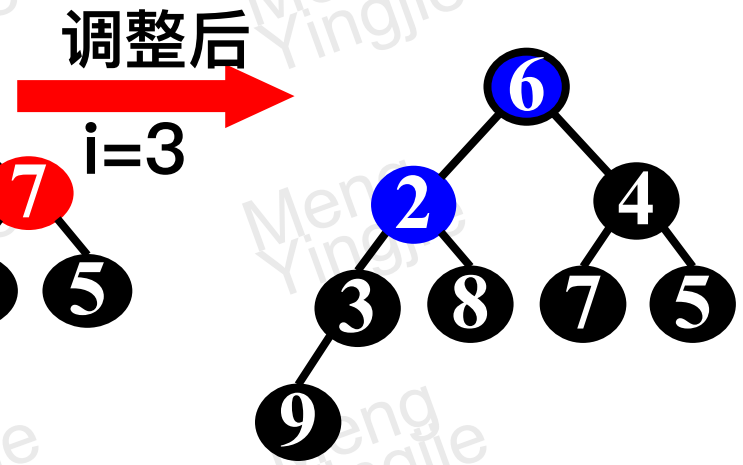
$L(i) \le L(2i)$ 且 $L(i) \le L(2i+1)$, $1 \le i \le \lfloor n/2 \rfloor$

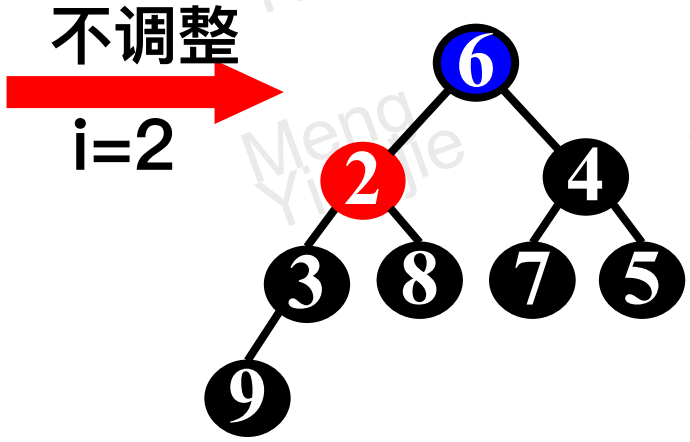


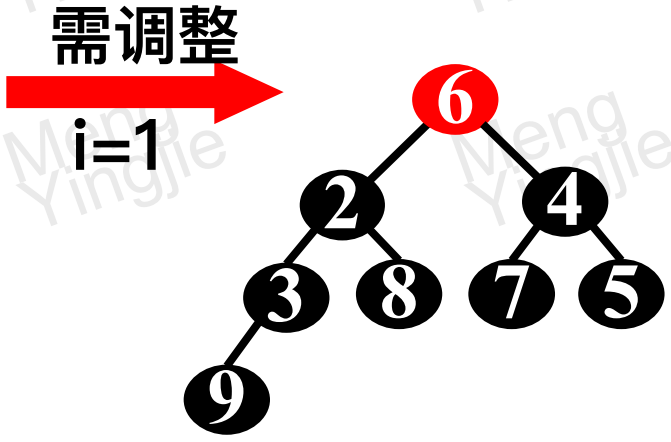
n=8, 故： i 从$\lfloor n/2 \rfloor$=4 DOWNTO 1 DO 筛选

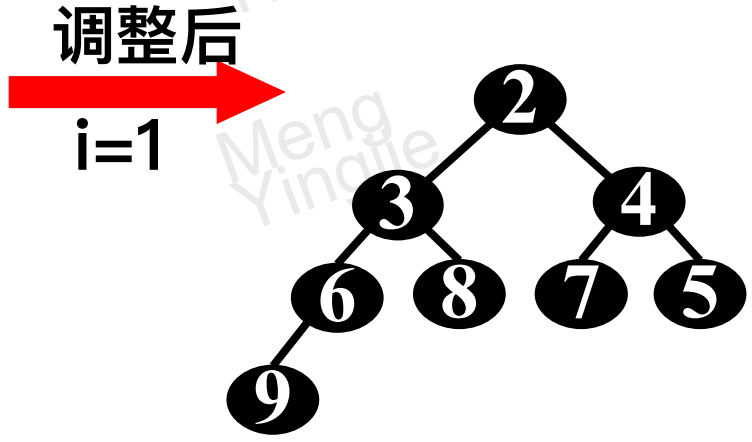












筛选建堆算法过程：

```
PROC  LineBuildHeap(VAR A:ARRAY[1..max] OF datatype;n:integer);
{元素存储于A[1..max],建立一个有n个元素的堆}
BEGIN
      FOR  i←1  TO  n  DO
           read(A[i]);
      FOR  i←n DIV 2  DOWNTO  1  DO
           HeapRectify(A,i,n)
END;
```

时间复杂性O(n).(证明略)

5.堆排序:

基本方法:

(1).建堆

(2).进行如下动作n-1次。

①删除堆顶元素

②执行筛选算法。

堆排序过程:

```
PROC  HeapSort(VAR A:ARRAY[1..max] OF datatype;n:integer);
BEGIN  ?BuildHeap(A,n)
       FOR  i←n  TO  2  DO
            【swop(A(1),A[i]);
              HeapRectify(A,1,i-1)】
END;
```

堆排序是不稳定的. 适合于大量记录情况.

此外，堆排序的空间复杂性为O(1).

堆排序的最坏、平均时间复杂性均为$O(n\log_2 n)$, 平均时间性能比快速排序要差一些(相差系数)。最坏情况比快速排序好的多。(证明，可参阅有关算法设计与分析的资料)

本节结束

一.合并类排序基本方法

前面讨论的各类排序对待排序的初始记录顺序均无任何要求。

合并排序(或归并排序,merge sort)则要求待排序文件已经部分排序。

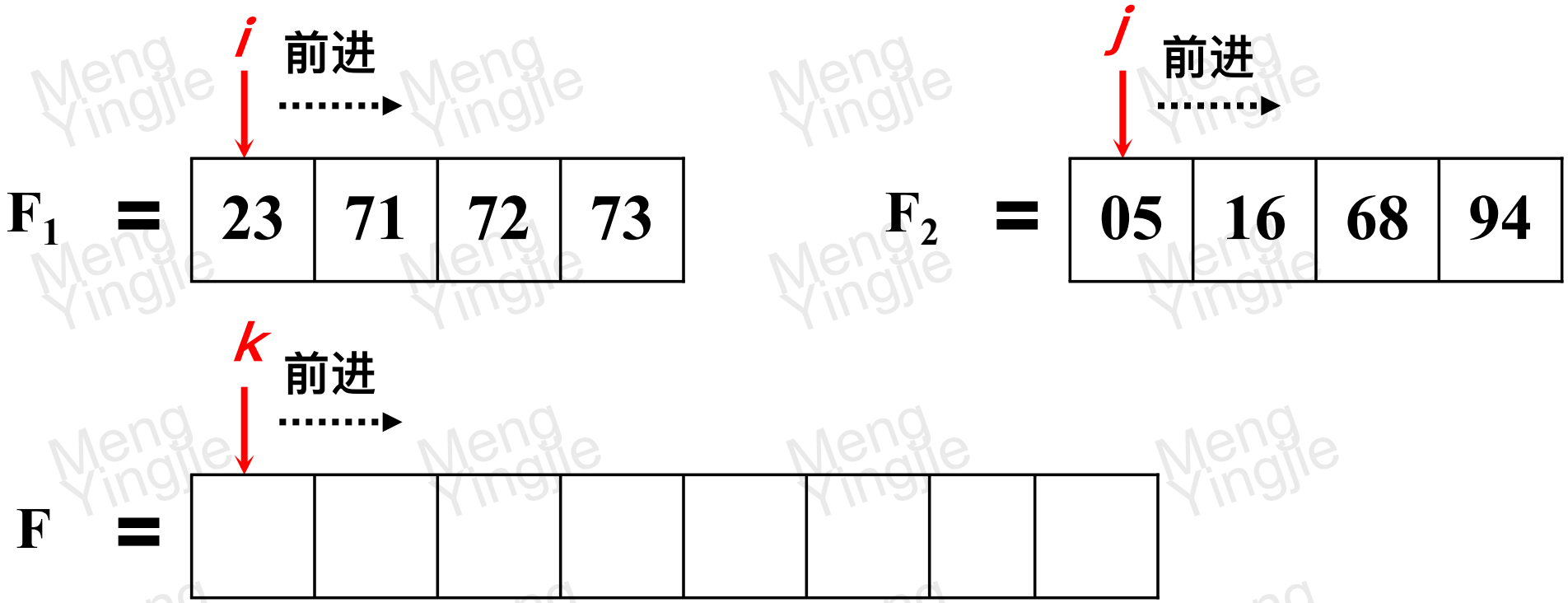
所谓部分排序的含义是：把待排序的文件分成若干个子文件，每个子文件内的记录是排序的。

归并排序的基本思想是：

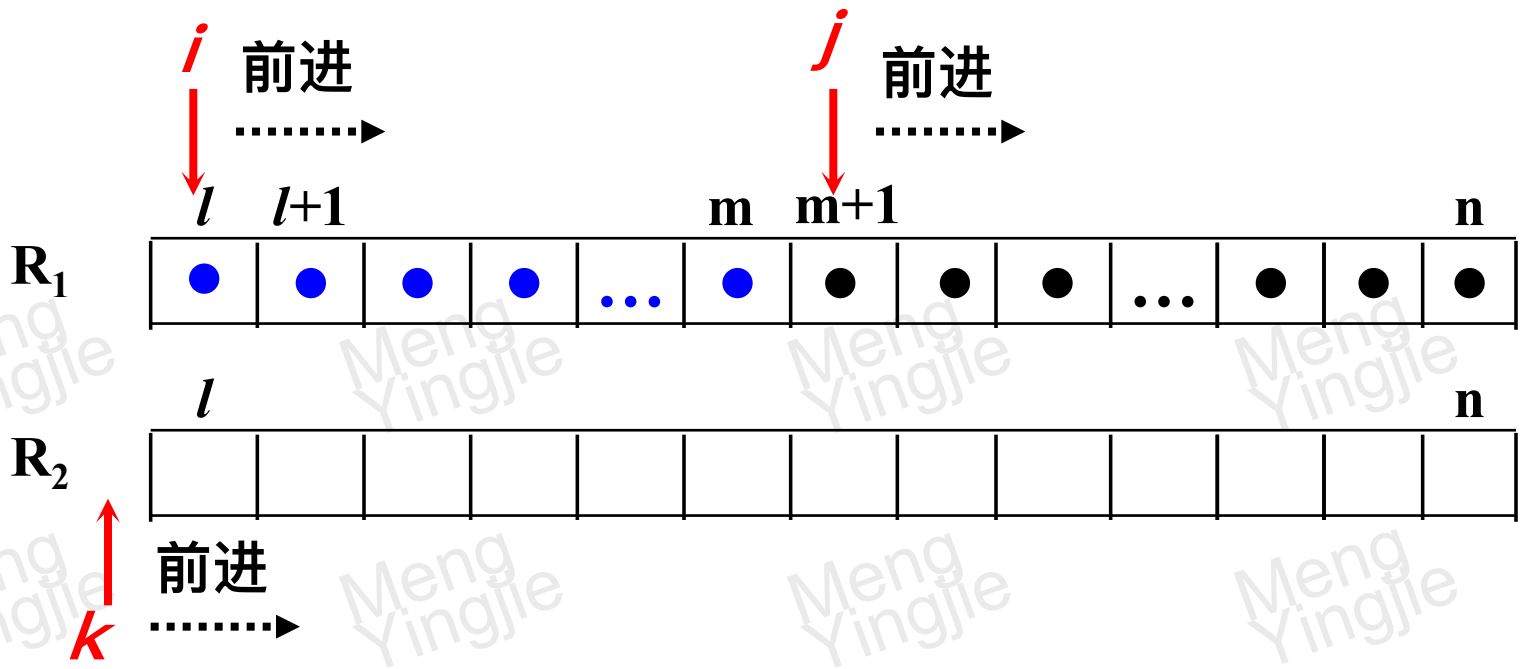
首先把一个集合中的数据元素分成若干个子集，对每个子集中的元素进行排序，再将所得到的有序子集进行合并。

二.两组合并

例，设有两个部分有序的子文件F_1=(23,71,72,73),
F_2=(05,16,68, 94),合并的具体做法如下：



需要的空间复杂度为O(n)




```
PROC  Merge(VAR R1,R2:ARRAY[1..max] OF datatype;l,m,n:integer);
BEGIN  i←l;   j←m+1;    k←l-1;
       WHILE (i≤m)AND(j≤n)  DO
          【k←k+1;
            IF  R[i].key≤R[j].key  THEN 【R2[k]←R1[i]; i←i+1】
                                   ELSE 【R2[k]←R1[j]; j←j+1】 】;
       IF i>m THEN  CALL  COPY(R1,j,n,R2) //*抄写j..n到R2
              ELSE  CALL  COPY(R1,i,m,R2)
END;
```

三.二路归并排序

1.基本思想:

如果文件中有n个记录，可将文件看成n个有序的子文件，这样就出现文件“部分有序”的状态；因而可以使用合并排序。

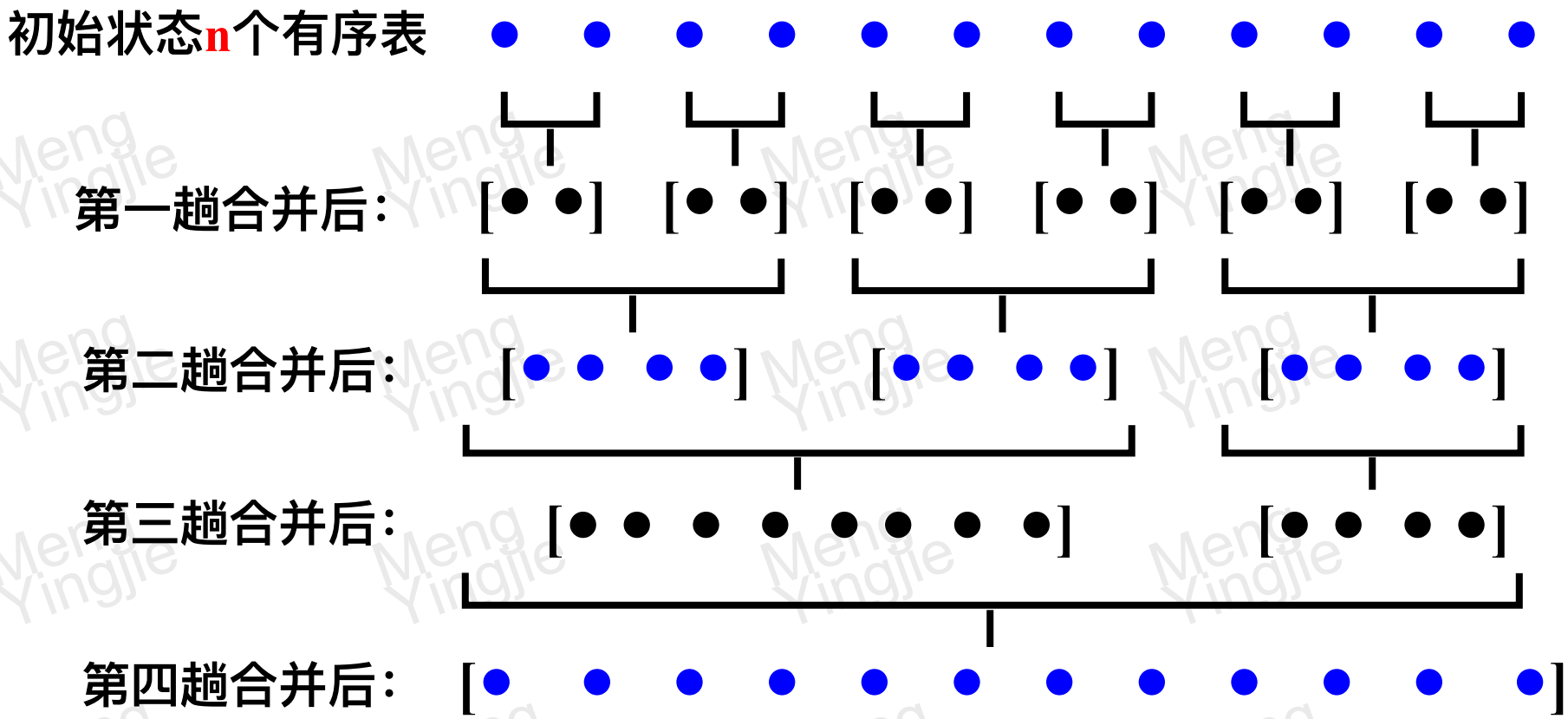
不过合并n个有序段，其操作较为复杂；

通常可以先将每两个子文件进行合并，得到n/2个部分有序的较大子文件，每个子文件含有两个记录；

再将这些子文件合并，如此反复，直到最后合并成一个文件时，排序就完成了；

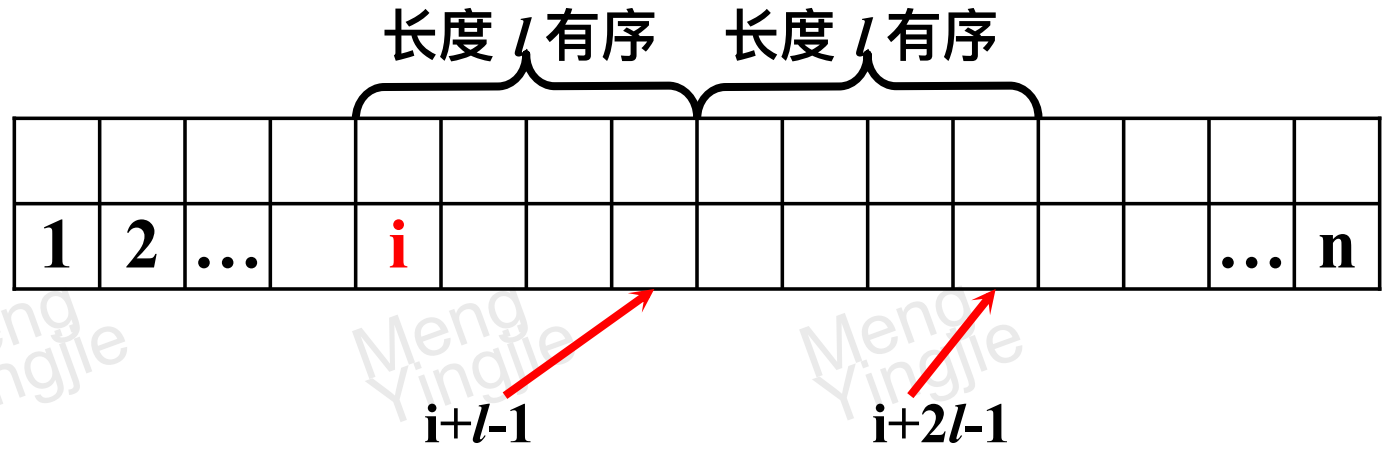
操作过程如下图所示:

合并排序示意图:



上述每步合并都是将两个子文件合并成一个新的子文件，上述做法称为“二路合并排序”，类似地可以有“三路合并排序”、“多路合并排序”。

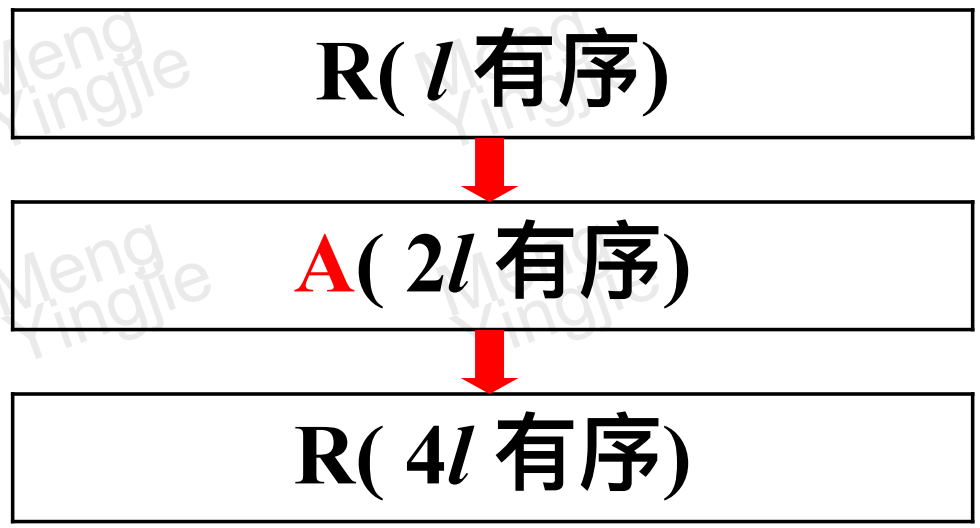
2.一趟合并:




```
PROC  MergePass(VAR R1,R2:ARRAY[1..max] OF datatype;l,n:integer);
{R1[1..n],由长度 l 的有序段构成，进行一趟合并使其变为长度为2l 有序}
BEGIN  i←1;
       WHILE (i≤n-2l+1) DO {对完整2l 段进行合并}
           【CALL  Merge(R1,R2, i , i+l-1 , i+2l-1);
            i ←i +2l 】;
       IF  i+l-1＜n  {两两合并后，最后所剩段的段长是否超过l }
       THEN  Merge(R1,R2, i , i+l-1 , n) {合并最后不足2l 部分 }
       ELSE  CALL  COPY(R1,i,n,R2) {抄写不足l 长度的部分 }
END;
```

3.二路合并排序:

```
PROC  MergeSort(VAR R,A:ARRAY[1..max] OF datatype;n:integer);
{A为临时数组}
BEGIN  l ← 1;
       WHILE ( l < n ) DO
          【CALL  MergePass(R, A , l , n);
            l ←2*l  ;
          CALL  MergePass(A, R , l , n);
            l ←2*l 】
END;
```



3.二路合并排序简单分析:

~~堆排序最不稳定的.~~

$$C(n)=0, \quad n=1$$

$$C(n)=C(\lfloor n/2 \rfloor)+C(\lfloor n/2 \rfloor)+n-1$$

解: $C(n)=\lceil n\log_2 n \rceil+2^{\lfloor \log_2 n \rfloor}) +1$

比较时间复杂性$O(n\log_2 n)$

移动时间复杂性$O(n)$

总时间复杂性$O(n\log_2 n)$

空间复杂性$O(n)$

本节结束

一.计数类排序基本方法

基本思想:

排序后的第j个关键字K，恰好大于j-1个别的关键字。

例如，若一个关键字K恰好大于27个别的关键字，则这个关键字所对应的记录就应排列在第28个位置上。

具体做法：利用统计的方法，即统计小于(或大于)关键字K的个数。

由此可见，计数排序(或枚举排序,merge sort)的基础在于比较关键字和计数。

为此，每个记录需带一个计数器(count,存放小于该记录的排序关键字的记录个数).

二.计数排序的过程

```
PROC  TallySort(VAR R:ARRAY[1..max] OF datatype;n:integer);
BEGIN  FOR  i←1  TO  n  DO
            R[i].count←0 ;
       FOR  i←n  DOWNTO  2  DO
            FOR  j←i-1  TO  1  DO
            IF  R[i].key≤R[j].key
            THEN  R[j].count←R[j].count + 1
            ELSE  R[i].count←R[i].count + 1
END;
```

时间复杂度:$O(n^2)$

本节结束

一.分配排序概述：

分配排序是一种常用的排序方法。

前面介绍的各类排序都是通过比较关键值的大小来进行的，一般只涉及单一关键的的比较。

而现实中经常会碰到排序的关键字是由多个数据项的组合情况，即**多因素排序**。

例如，扑克牌的排序、体育比赛中积分的计算等。都是典型的这种排序类型。

具体做法无非**有两种方法**：

例1： 扑克牌排序具有两个关键字，花色和面值。

其次序定义如下：

(1)面值＜花色，即花色比面值重要。

(2)花色：♣ ＜ ♦ ＜ ♥ ♠

(3)面值：2＜3＜4＜5＜6＜7＜8＜9＜10＜J＜Q＜K＜A

依据上述排序关系,若要将一副牌整理为如下由小到大的次序：

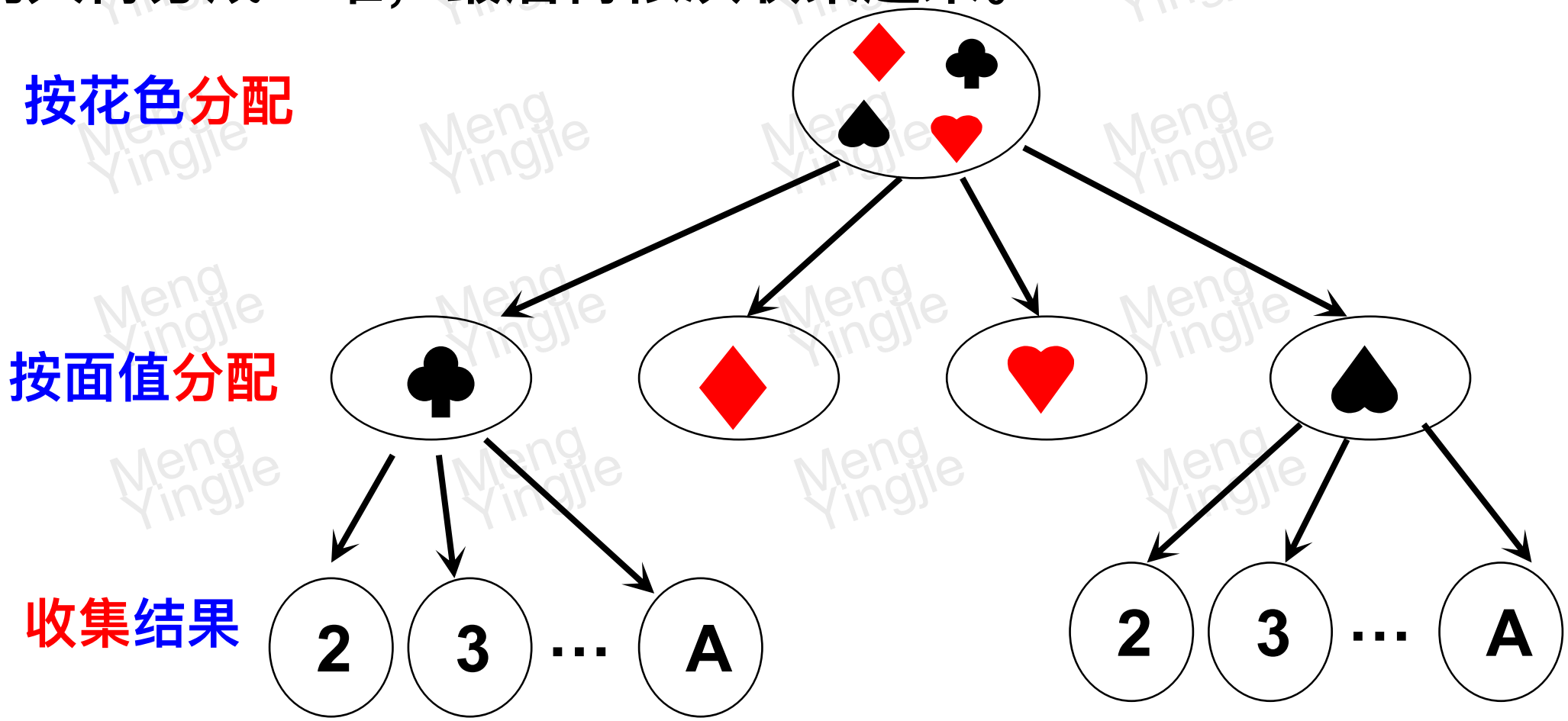
♣2,♣3,…,♣A, ♦2, ♦3,…, ♦A, ♥2, ♥3,…,

♥A, ♠2, ♠3,…, ♠A

要把牌整理有序，可有两种方法：

方法一：

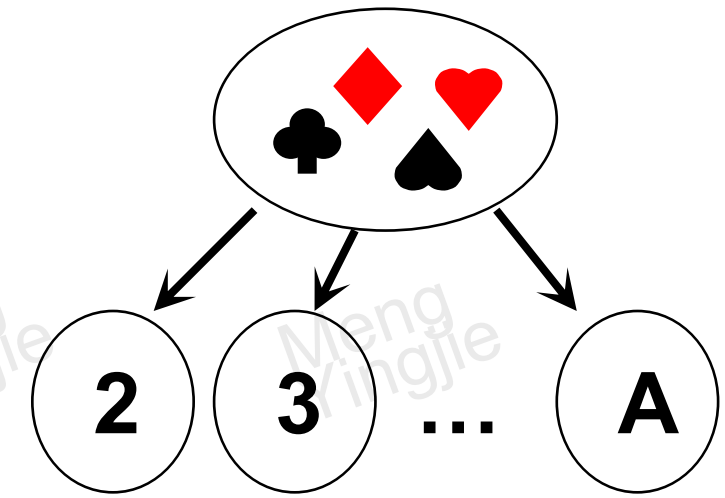
先将牌按花色分成由小到大的四堆,然后每堆按面值由小到大再分成13堆，最后再依次收集起来。



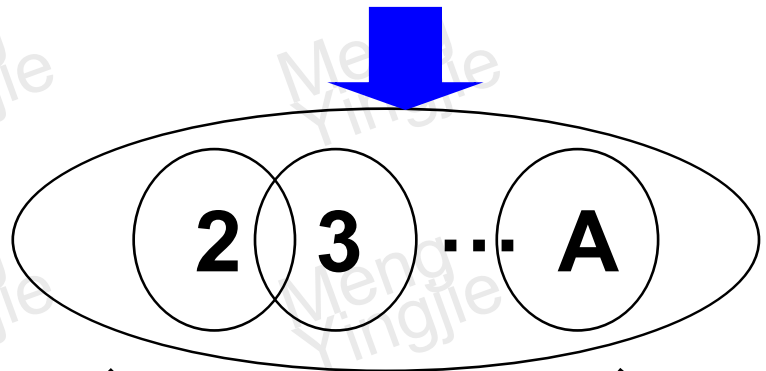
方法二：

按面值分配

先将牌按面值分成13堆,然后由小到大收集起来；

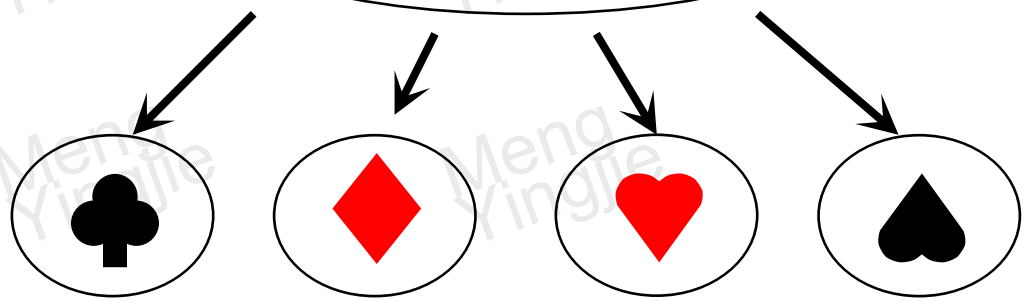


依次收集

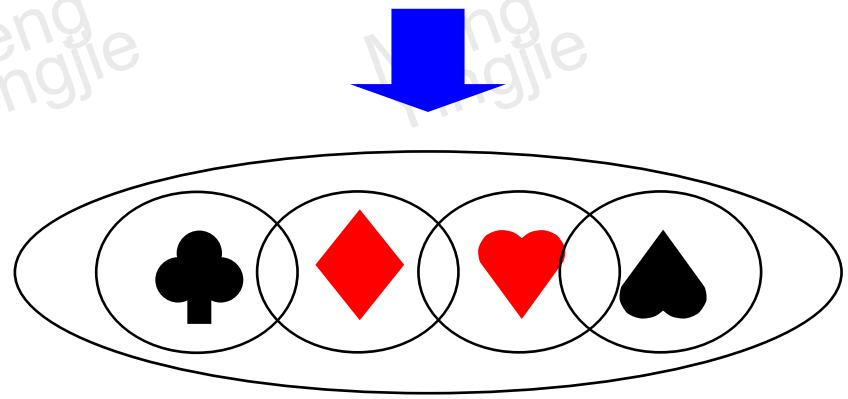


按花色分配

再按花色不同分成四堆,最后顺序的收集起来。



依次收集



例2： 人员卡片排序，以出生日期排序。

排序关键字由年、月、日三个关键字构成。

方法一：

先按年分配成若干堆，再按月分成12干小堆,再对每小堆卡片按日分成31堆，最后依次收集。

例2： 人员卡片排序，以出生日期排序。

排序关键字由年、月、日三个关键字构成。

方法二：

先按日分成31堆，然后依次叠起(31日在最下面)；

再按月分成12堆，然后依次叠起(12月在最下面)；

最后再按年分成若干堆，然后依次叠起(年份最大的在最下面).

总结(d个关键字排序时):

对于方法一:

该方法称为最高位优先法(MSD,most significant digit first).

具体做法:

先对最高位关键字(最重要的)K^1排序，即分成若干堆，每堆中具有相同的K^1值;

然后在各子堆中再按关键字K^2进行排序，又分成若干个子堆,……,直至对各子堆按K^d排序后，将各堆按次序排在一起形成一个有序文件。

总结(d个关键字排序时):

对于方法二:

该方法称为最低位优先法(LSD,least significant digit first).

具体做法:

先从最低位关键字(最次要的)K^d排序，即按K^d值将数据分为若干堆，每堆中的K^d值相同，然后按K^d从小到大的次序收集到一起;

下次再按K^{d-1}值将数据分为若干堆，每堆中的K^{d-1}值相同，然后按K^{d-1}从小到大的次序收集到一起;……;

直至对K^1排序后整个文件有序为止。

总结(d个关键字排序时):

共同特点：采用先分配后收集的方法。

MSD与LSD的比较:

◆MSD排序法必须将文件按关键字位逐层分成若干子堆，而每堆的排序是独立进行的,存在递归处理各组的问题；

◆LSD排序法则不必分成子堆，而是从头到尾进行若干次分配和收集，执行的次数取决于关键字位的个数。

因此一般说来，LSD方法比MSD方法简单。

二.基数排序 (radix sort)

基数排序(radix sort)就是按照LSD法对单逻辑关键字进行排序的一种方法，也称桶排序。

1.基本方法:

把每个关键字看成d元组:

$K_i=(k^1_i, k^2_i, \ldots, k^d_i)$

其中: $C_0 \le k^j_i \le C_{r-1}$ $(1 \le i \le n, 1 \le j \le d)$

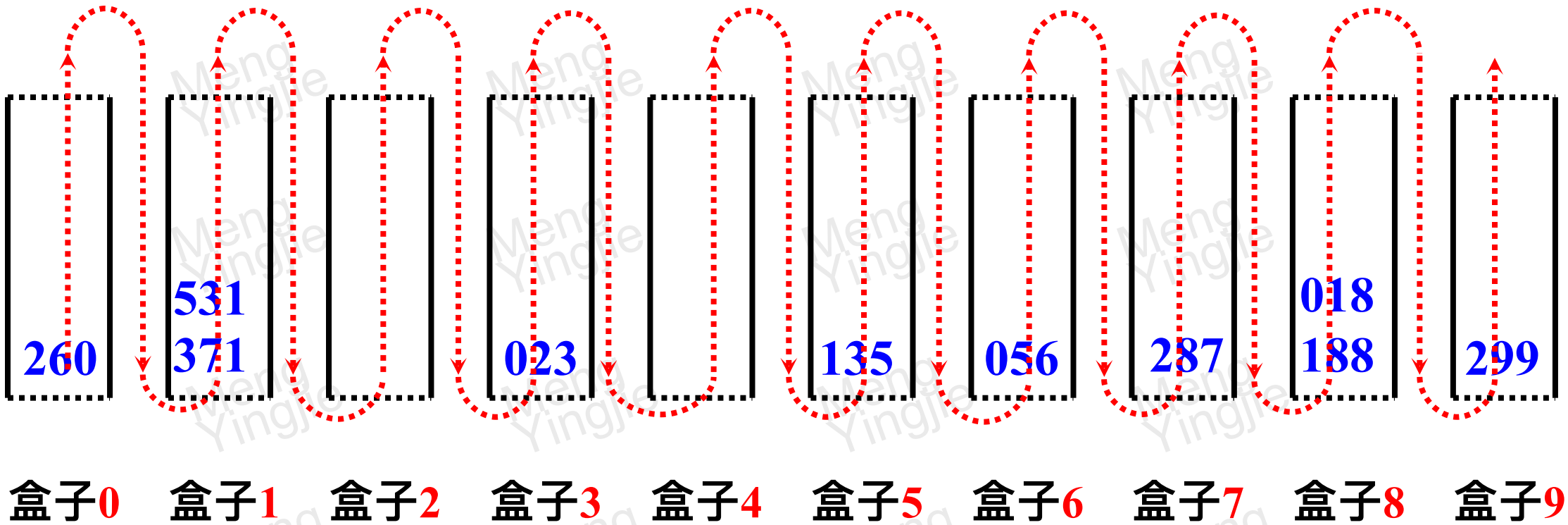
排序时先按k^d_i的值从小到大将记录分配到r个盒子中去,然后依次收集这些记录;

再按k_i^{d-1}的值从小到大将记录分配到r个盒子，如此反复,直到对k^1_i的分配和收集后就得到完全有序的文件。

其中r称为基数, 执行基数排序时, 为了实现分配和收集,需设立r个队列。

例，(188,371,260,531,287,135,56,299,18,23)

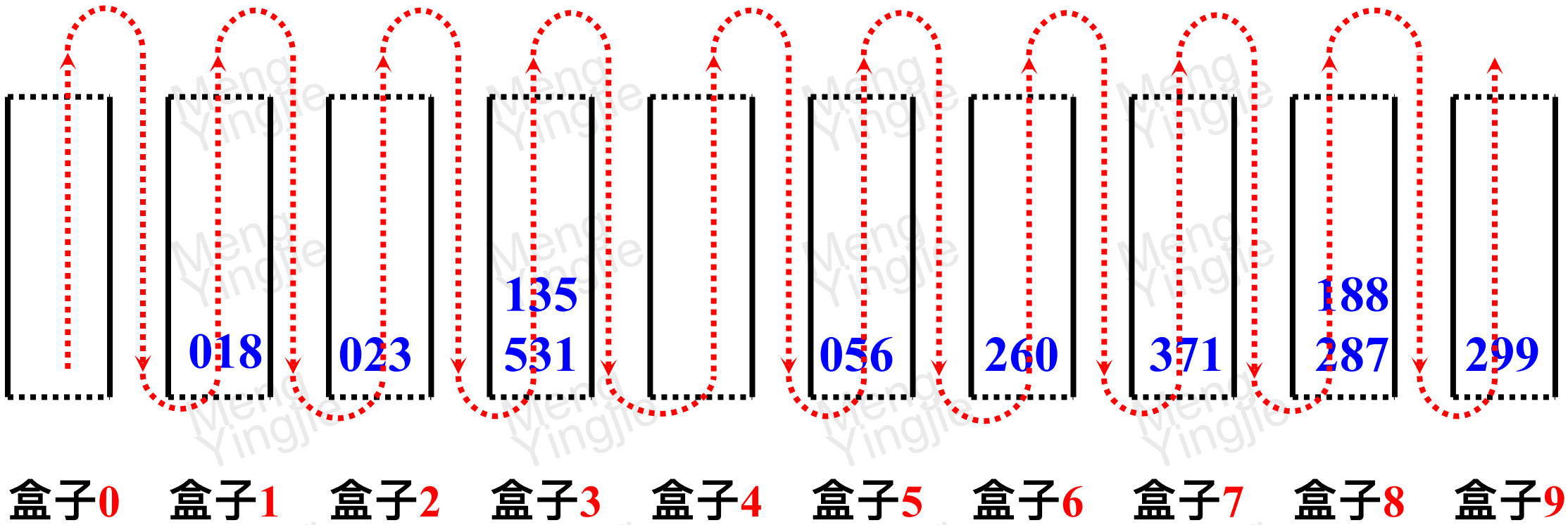
分配按$K^{(3)}$：(188,371,260,531,287,135,056,299,018,023)



依次收集：(260,371,531,023,135,056,287,188,018,299)

第1趟收集结果: (260,371,531,023,135,056.287,188,018,299)

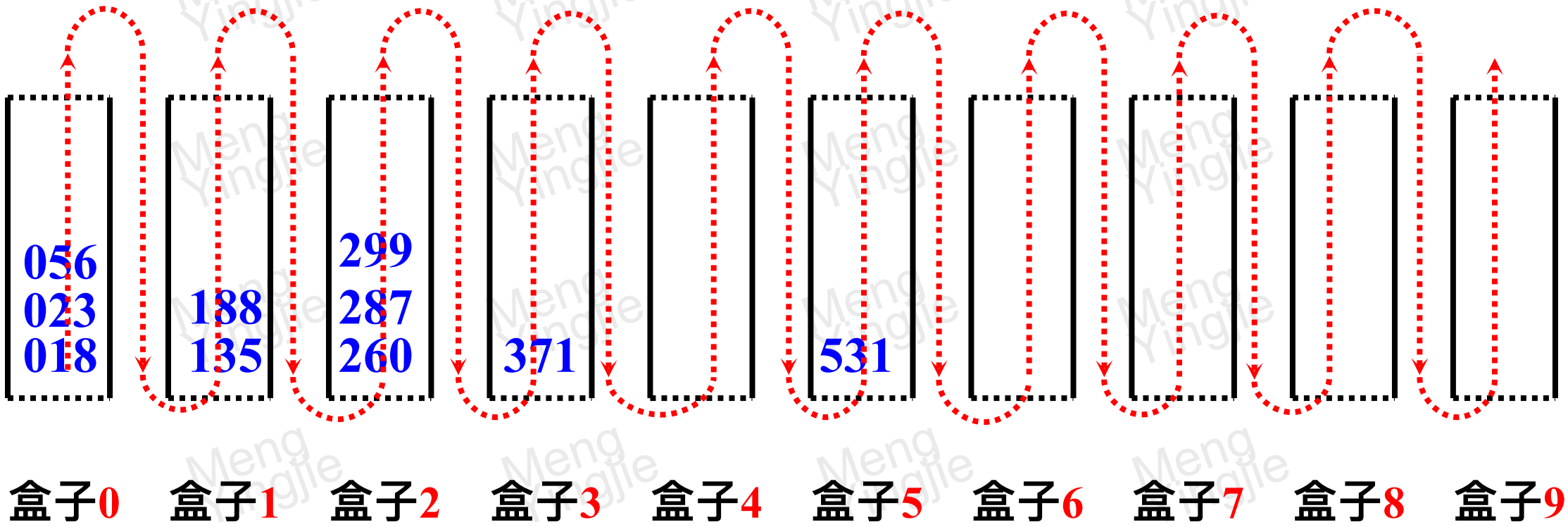
分配按$K^{(2)}$：(260,371,531,023,135,056,287,188,018,299)



依次收集：(018,023,531,135,056,260,371,287,188,299)

第2趟收集结果: (018,023,531,135,056,260,371,287,188,299)

分配按$K^{(1)}$：(018,023,531,135,056,260,371,287,188,299)



第3趟收集结果:(018,023,056,135,188,260,287,299,371,531)

完全有序

2.存储方式的选取:

(1).顺序存储方式:

常规存储：用一维数组存放待排序的记录。

辅助存储：r个一维数组,作为存储分配时的r个队列(每个队列空间大小以最大记录数分配),共需要r×n个结点；

分配、收集一次共需要移动2n个记录；

分配收集共需进行d趟,共需进行2×n×d次移动,代价很高。

2.存储方式的选取:

(2).链式存储方式:

采用链式可以改变顺序存储的缺陷。可以采用静态链或者动态链来组织。

但为了组织分配、收集需要设置r个队列的首、尾指针。

此处我们考虑采用静态链表来实现。

设用数组R存储待排序的数据元素，给R的数据元素增加一个next数据项，用以存放下个数据元素的位置，R及队列的设置如下图所示:

3.基数排序的静态链式组织:

```
TYPE  Ktype=ARRAY[1..d] OF C0..Cr-1;
        node=RECORD
                key:Ktype;
                info:datatype;
                next:0..n
             END;
        elem=RECORD
                f : 0..n;
                e : 0..n
             END;
VAR   R:ARRAY [1..n] OF node;
      Q:ARRAY [C0..Cr-1] OF elem;
```

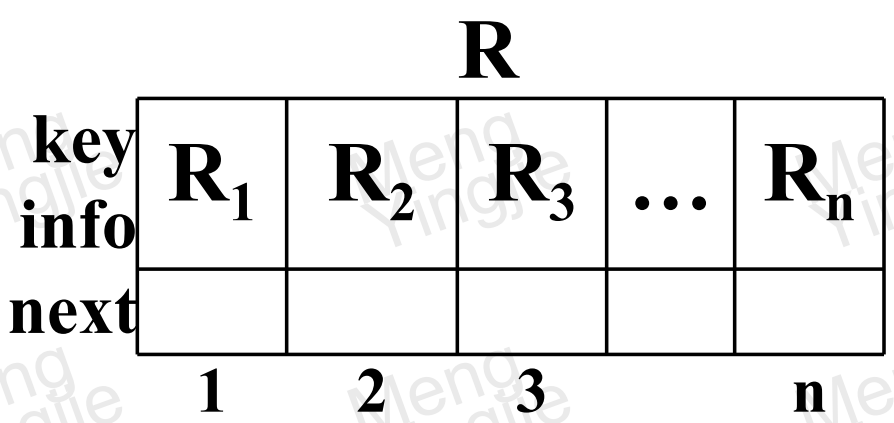
R

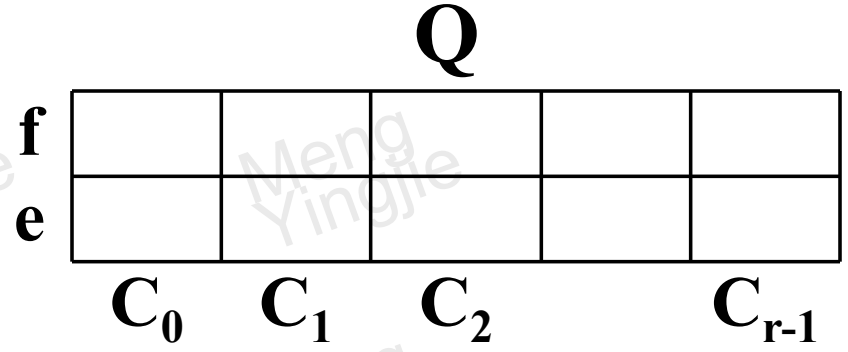
key	R_1	R_2	R_3	…	R_n
info					
next					
	1	2	3		n

Q

f					
e					
	C_0	C_1	C_2		C_{r-1}

4.基数排序算法基本步骤:





(1).准备，P←1 {排序后首元素位置初值}

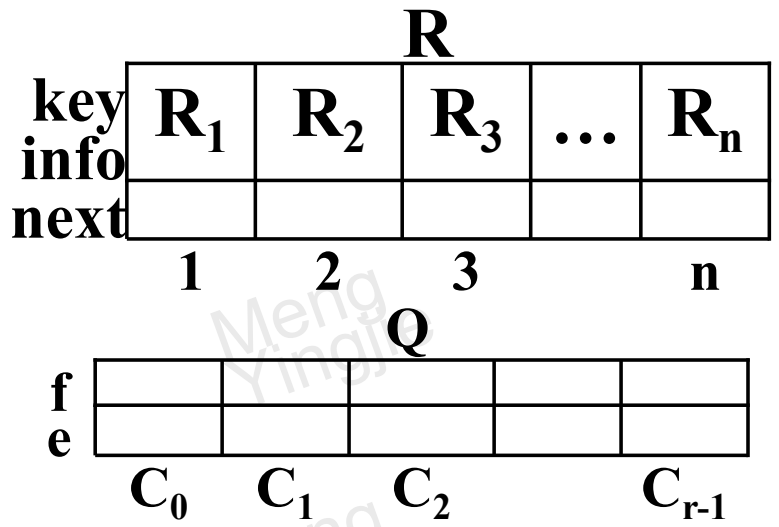
(2).排序： 循环d次

①Q初始化；

②分配；

③收集

算法过程:




```
PROC  RadixSort(R , Q,  P);
BEGIN {准备}
      P←1;
      {排序}
      FOR  i←d  DOWNTO  1  DO
         【 {Q初始化}
            FOR  j←C0  TO  Cr-1  DO 【Q[j].f←0;  Q[j].e←0】 ;
         {分配}
         WHILE  P≠0  DO
            【k←R[p].key[i]; {在第i趟中，要进第K个队列}
             IF  Q[k].f=0 THEN  Q[k].f←p
                          ELSE  R[Q[k].e].next←p;
             Q[k].e←p;
             p←R[p].next】  ;
```

算法过程(续):

```
{以下开始收集}
j←C0;
WHILE Q[j].f=0 DO    {寻找第一个不空队列}
    j←succ(j);    {succ,求x在子界类型C0..Cr-1中的后继函数}
p←Q[j].f;        {确定本趟收集的第一记录位置号}
t←Q[j].e;        {第一个不空队列的尾指针⇒t}
FOR  k←succ(j)  TO Cr-1  DO
    IF Q[k].f≠0 THEN 【 R[t].next←Q[k].f;
                        t←Q[k].e 】 ;
R[t].next←0 ;】
END;
```

算法的简单分析:

时间复杂度:

一趟时间(分配和收集):

Q初始化需要O(r)

分配需要O(n)

收集需要O(r)

总时间代价O(d(n+r)),不需要移动记录。

空间复杂性:

增加了一个next字段,共O(n)辅助空间;

另增加一个Q数组,需要空间O(r)

总空间复杂度O(n+r)

三.本章排序方法比较

排序方法	平均时间	时间最坏	辅助存储	备注
简单排序	$O(n^2)$	$O(n^2)$	O(1)	除shell外的所有插入排序、起泡排序、直接选择排序。
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	
基数排序	O(d(n+r))	O(d(n+r))	O(n+r)	

本节结束

1.设计一个非递归的快速排序算法。

2.如果输入的为有序表，试证明使用快速排序的时间复杂度为$O(n^2)$

3.设关键字集合为{6,2,9,7,8,4,5,0,10}给出用归并排序和快速排序的第1趟排序结果。

4.判断下列序列是否为堆。若不是请调整为堆。

(12,70,33,65,24,56,48,92,86,33).

本章结束