信息与软件工程学院

程序设计与算法基础II

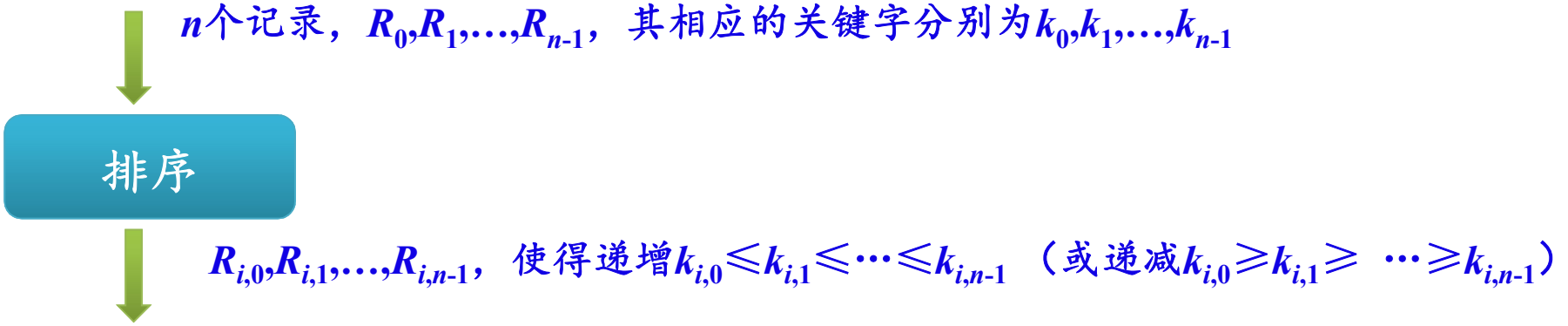
主讲教师：陈安龙

第9章　内 排 序

9.1 排序的概念

1、排序的定义

所谓排序，是整理表中的记录，使之按关键字递增（或递减）有序排列：

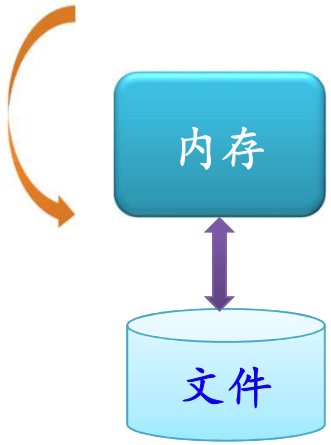


说明：排序数据中可以存在相同关键字的记录。本章仅考虑递增排序。

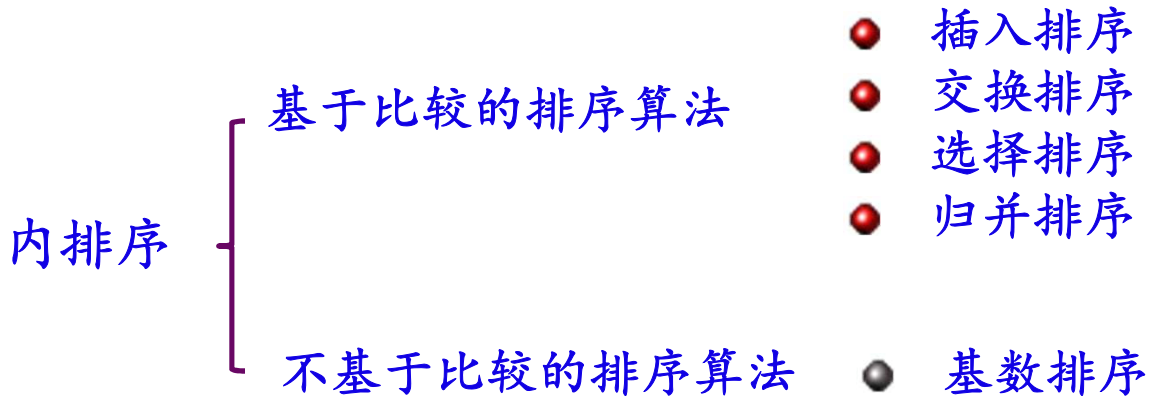
2、内排序和外排序

在排序过程中，若整个表都是放在内存中处理，排序时不涉及数据的内、外存交换，则称之为内排序；

反之，若排序过程中要进行数据的内、外存交换，则称之为外排序。



3、内排序的分类



基于比较的内排序算法最快有多快？

假设有3个记录(R_1,R_2,R_3)，对应的关键字为(k_1,k_2,k_3)。

初始数据序列有 3! = 6 种情况：

- 1，2，3
- 1，3，2
- 2，1，3
- 2，3，1
- 3，1，2
- 3，2，1

📖 *n*个记录，初始数据序列有*n*!种情况

以(R_1,R_2,R_3)=（2,3,1）为例，一种基于比较的排序方法：

$R_1\ R_2\ R_3$
2 3 1

↓ $R_1<R_2$为真，不交换

$R_1\ R_2\ R_3$
2 3 1

↓ $R_2<R_3$为假，R_2、R_2交换

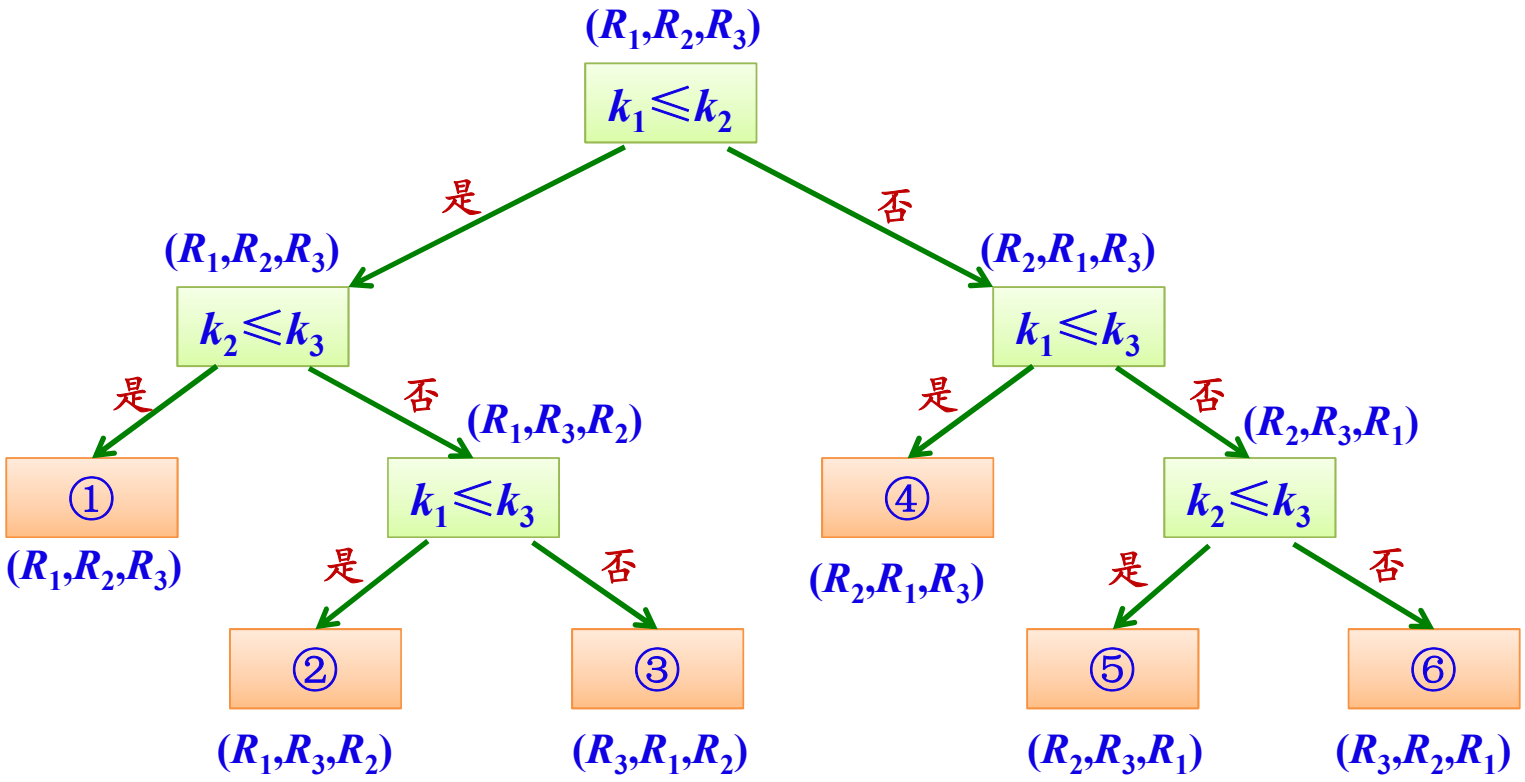
$R_1\ R_3\ R_2$
2 1 3

↓ $R_1<R_3$为假，R_1、R_2交换

$R_3\ R_1\ R_2$
1 2 3

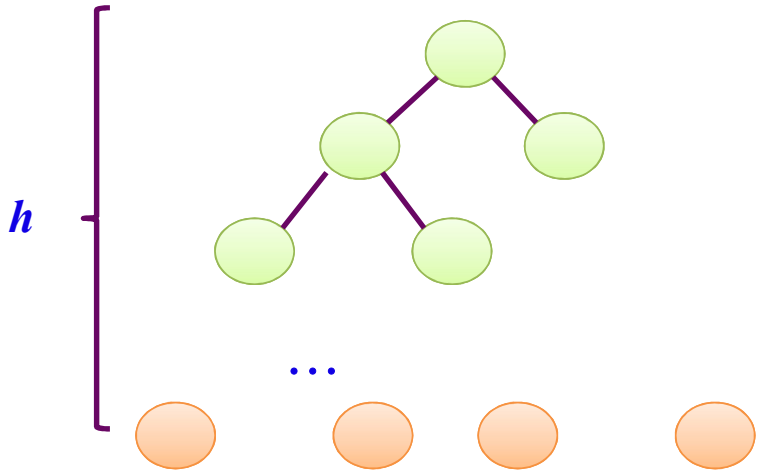
总共3次关键字比较

所有可能的初始序列的排序过程构成一个决策树：



决策树是一棵有$n!$个叶结点的二叉树。

决策树可以近似看成是一颗高度为h，叶结点个数为$n!$的满二叉树。



- 叶结点个数=$n!$
- 总结点个数=$2n!-1$
- $h=\log_2(\text{总结点个数}+1)=\log_2(n!)\approx n\log_2 n$
- 平均关键字比较次数=$h-1$
- 移动次数也是同样的数量级，即这样的算法最坏时间复杂度为$O(n\log_2 n)$。
- 同样可以证明平均时间复杂度也为$O(n\log_2 n)$。

结论：

*n*个记录采用基于比较的排序方法：

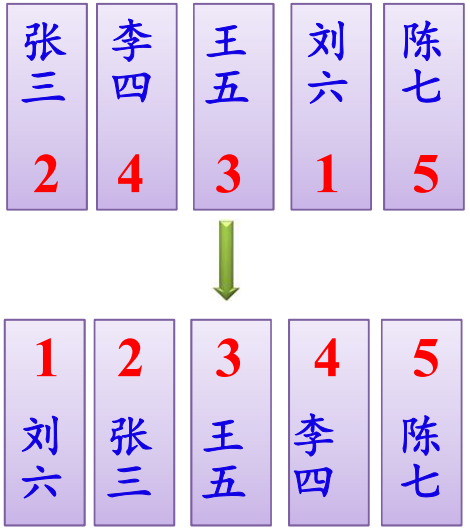
- 最好的平均时间复杂度为$O(n\log_2 n)$。
- 最好情况是排序序列正序，此时的时间复杂度为$O(n)$。

思考题

对n个记录按某个关键字排序，你能够采用基于比较的方法设计出平均时间复杂度好于为$O(n\log_2 n)$的排序算法吗？

4、内排序算法的稳定性

当待排序记录的关键字均不相同时，排序的结果是唯一的。



如果待排序的表中，存在有多个关键字相同的记录，经过排序后这些具有相同关键字的记录之间的相对次序保持不变，则称这种排序方法是稳定的。

3 4 3 1 5

1 2 3 3 5

反之，若具有相同关键字的记录之间的相对次序发生变化，则称这种排序方法是不稳定的。

3 4 3 1 5

1 2 3 3 5

5、正序和反序

若待排序的表中元素已按关键字排好序，称此表中元素为正序；

反之，若待排序的表中元素的关键字顺序正好和排好序的顺序相反，称此表中元素为反序。

有一些排序算法与初始序列的正序或反序有关，另一些排序算法与初始序列的情况无关。

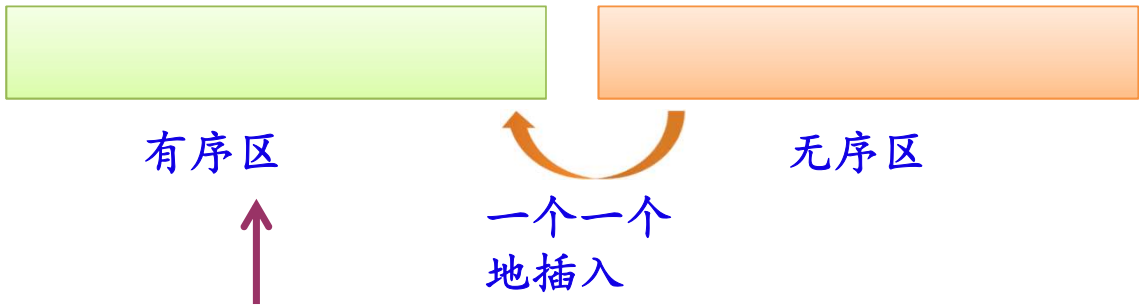
6、 内排序数据的组织

待排序的顺序表的数据元素类型定义如下：

```
typedef int KeyType;        //定义关键字类型
typedef struct              //记录类型
{    KeyType key;           //关键字项
     InfoType data;         //其他数据项,类型为InfoType
} RecType;                  //排序的记录类型定义
```

9.2 插入排序

基本思路



不一定是全局有序（整体有序）　全局有序区
的元素在后面排序中不再发生位置的改变

主要的插入排序方法：

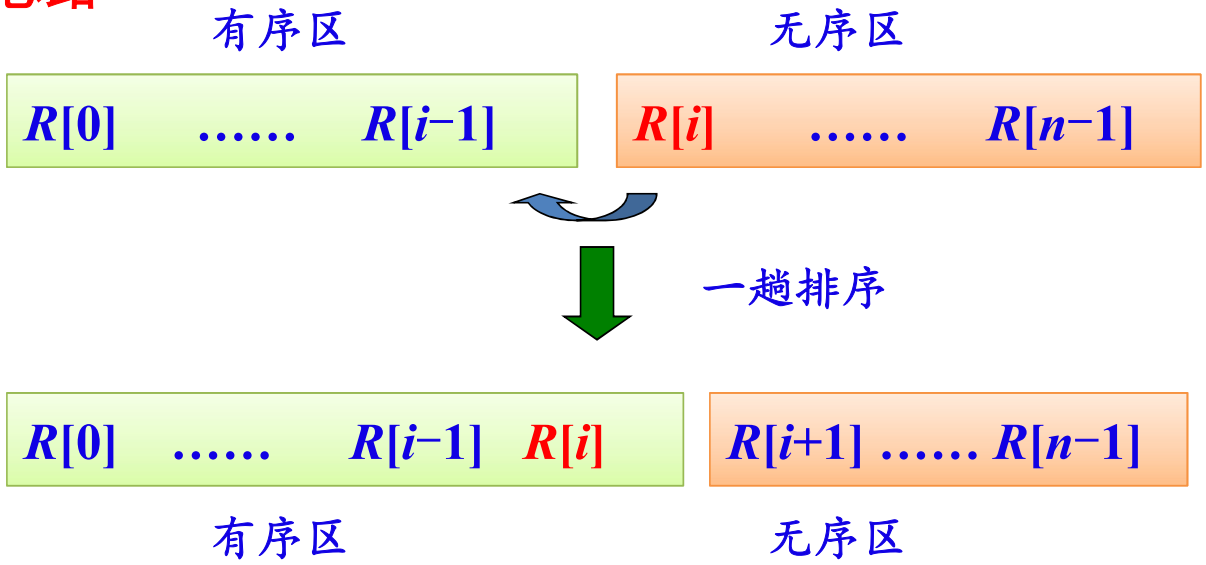
（1）直接插入排序

（2）折半插入排序

（3）希尔排序

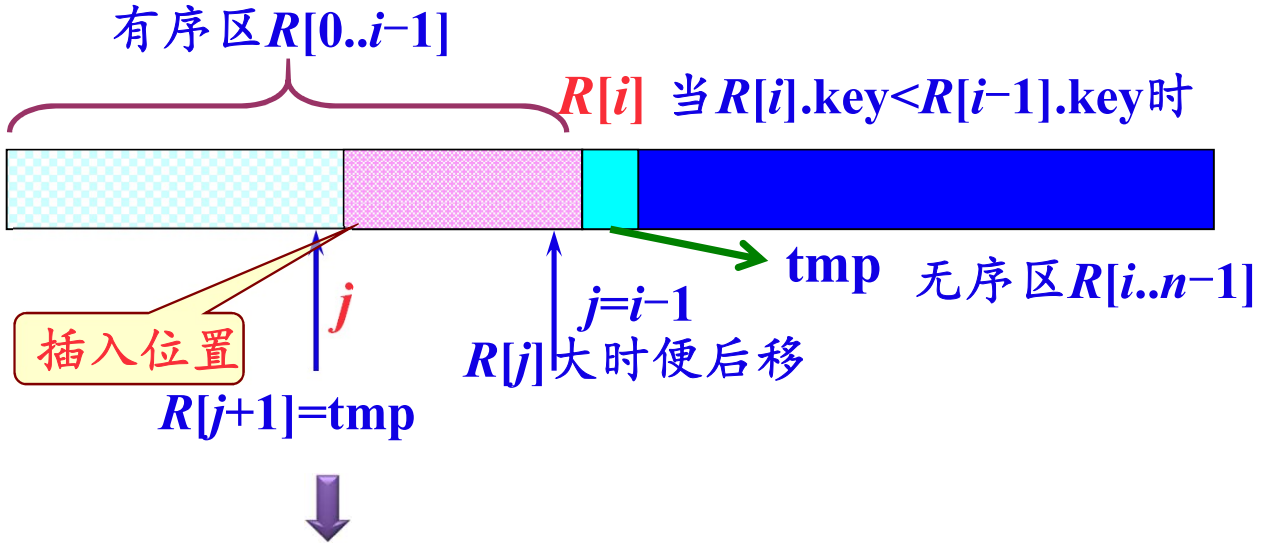
9.2.1 直接插入排序

基本思路



初始时，有序区只有一个元素$R[0]$，$i=1\sim n-1$，共经过$n-1$趟排序

一趟直接插入排序：在有序区中插入$R[i]$的过程。



【例9-1】 设待排序的表有10个元素，其关键字分别为（9，8，7，6，5，4，3，2，1，0）。说明采用直接插入排序方法进行排序的过程。

初始：（9，8，7，6，5，4，3，2，1，0）

i=1：（8，9，7，6，5，4，3，2，1，0）

i=2：（7，8，9，6，5，4，3，2，1，0）

i=3：（6，7，8，9，5，4，3，2，1，0）

i=4：（5，6，7，8，9，4，3，2，1，0）

i=5：（4，5，6，7，8，9，3，2，1，0）

i=6：（3，4，5，6，7，8，9，2，1，0）

i=7：（2，3，4，5，6，7，8，9，1，0）

i=8：（1，2，3，4，5，6，7，8，9，0）

i=9：（0，1，2，3，4，5，6，7，8，9）

直接插入排序的算法：

```
void  InsertSort(RecType R[]，int n)
{    int i，  j;  RecType tmp;
     for (i=1;i<n;i++)
     {     if (R[i].key<R[i-1].key)   //反序时
           {     tmp=R[i];
                 j=i-1;
                 do                          //找R[i]的插入位置
                 {    R[j+1]=R[j];          //将关键字大于R[i].key的记录后移
                      j--;
                 } while  (j>=0 && R[j].key>tmp.key)
                R[j+1]=tmp;                  //在j+1处插入R[i]
           }
     }
}
```

算法分析

最好的情况（关键字在记录序列中正序）：

“比较”的次数：

$$\sum_{i=1}^{n-1} 1 = n-1$$

“移动”的次数：

$$0$$

最好：O(*n*)

最坏的情况（关键字在记录序列中反序）：

“比较”的次数：

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

“移动”的次数：

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

最坏：O(n^2)

总的平均比较和移动次数约为

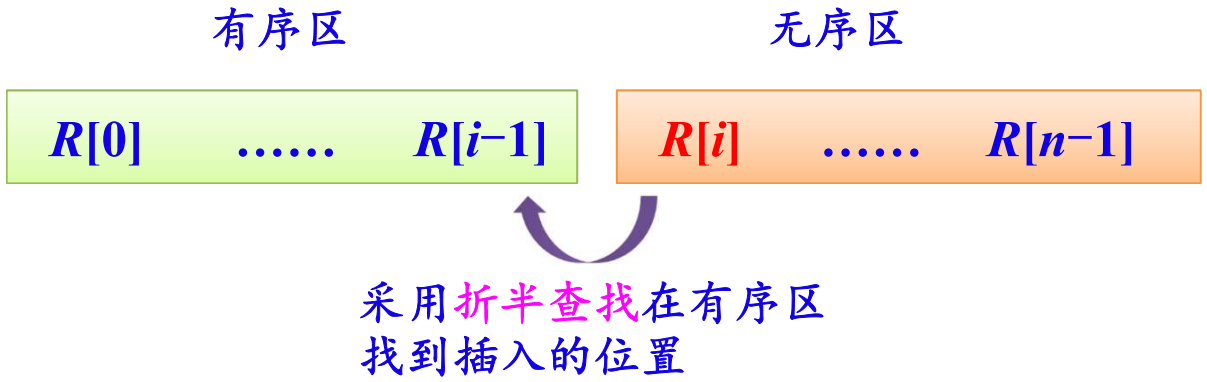
$$\sum_{i=1}^{n-1} \left(\frac{i}{2}+\frac{i}{2}+2\right) = \sum_{i=1}^{n-1} (i+2) = \frac{(n-1)(n+4)}{2} = O(n^2)$$

平均：O(n^2)

9.2.2 折半插入排序

基本思路

查找采用折半查找方法，称为二分插入排序或折半插入排序。



折半插入排序算法：

```
void BinInsertSort(RecType R[], int n)
{   int i, j, low, high, mid;
    RecType tmp;
    for (i=1;i<n;i++)
    {   if (R[i].key<R[i-1].key])                  //反序时
        {   tmp=R[i];                              //将R[i]保存到tmp中
            low=0; high=i-1;
            while (low<=high)                      //在R[low..high]中查找插入的位置
            {   mid=(low+high)/2;                  //取中间位置
                if (tmp.key<R[mid].key)
                    high=mid-1;                    //插入点在左半区
                else
                    low=mid+1;                     //插入点在右半区
            }                                      //找位置high
            for (j=i-1;j>=high+1;j--)              //记录后移
                R[j+1]=R[j];
            R[high+1]=tmp;                         //插入tmp
        }
}
```

算法分析

折半插入排序：在$R[0..i-1]$中查找插入$R[i]$的位置，折半查找的平均关键字比较次数为$\log_2(i+1)-1$，平均移动元素的次数为$i/2+2$，所以平均时间复杂度为：

$$\sum_{i=1}^{n-1}\left(\log_2(i+1)-1+\frac{i}{2}+2\right)=O(n^2)$$

折半插入排序采用折半查找，查找效率提高。但元素移动次数不变，仅仅将分散移动改为集合移动。

【例（补充）】对同一待排序序列分别进行折半插入排序和直接插入排序，两者之间可能的不同之处是_____。

A.排序的总趟数

B.元素的移动次数

C.使用辅助空间的数量

D.元素之间的比较次数

说明：本题为2012年全国考研题

9.2.3 希尔排序

基本思路

① $d=n/2$

② 将排序序列分为d个组，在各组内进行直接插入排序

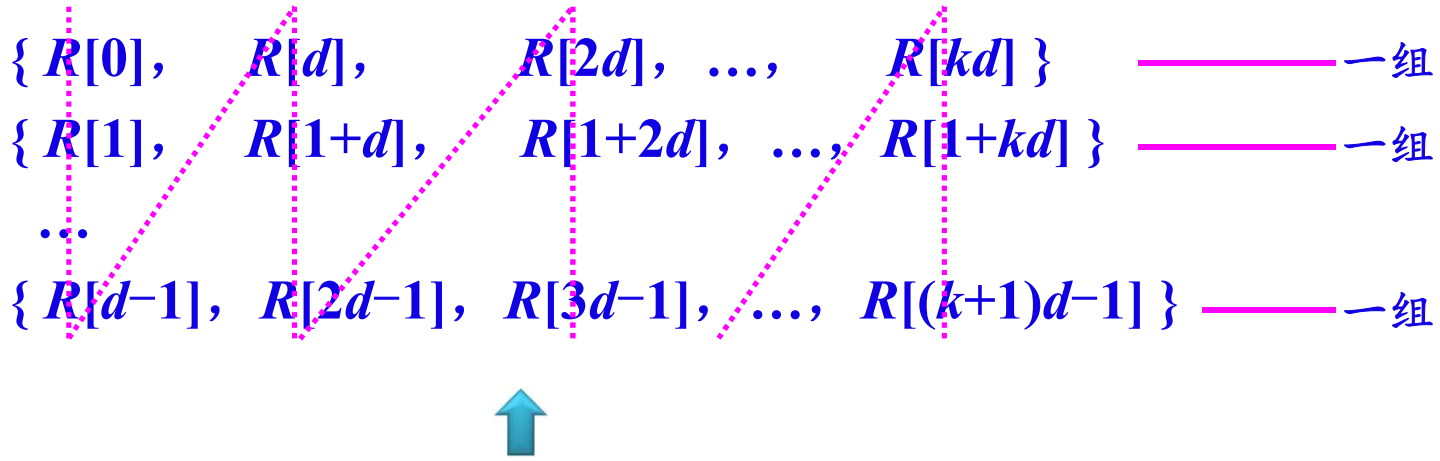
③ 递减$d=d/2$，重复②，直到$d=1$

算法最后一趟对所有数据进行了直接插入排序，所以结果一定是正确的。

一趟希尔排序过程

将记录序列分成若干子序列，分别对每个子序列进行直接插入排序。

例如：将 n 个记录分成 d 个子序列：



相距d个位置的记录分为一组

【例9-2】

初始序列	9	8	7	6	5	4	3	2	1	0
d=5	9	8	7	6	5	4	3	2	1	0
直接插入排序	4	3	2	1	0	9	8	7	6	5
d=*d*/2=2	4	3	2	1	0	9	8	7	6	5
直接插入排序	0	1	2	3	4	5	6	7	8	9
d=*d*/2=1	0	1	2	3	4	5	6	7	8	9
直接插入排序	0	1	2	3	4	5	6	7	8	9

注意：对于*d*=1的一趟，排序前的数据已将近正序！

希尔排序算法：

```
void ShellSort(RecType R[]，int n)
{  int i，  j，  d;
   RecType tmp;
   d=n/2;              //增量置初值
   while (d>0)
   {   for (i=d;i<n;i++)
       {     //对相隔d位置的元素组直接插入排序
             tmp=R[i];
             j=i-d;
             while (j>=0&&tmp.key<R[j].key)
             {   R[j+d]=R[j];
                 j=j-d;
             }
             R[j+d]=tmp;
       }
       d=d/2;          //减小增量
   }
}
```

*d*循环：每个记录都参加排序了

直接插入排序：

```
for (i=1;i<n;i++)
{  tmp=R[i];
   j=i-1;
   while (j>=0 &&
     tmp.key<R[j].key)
   {  R[j+1]=R[j];
      j=j-1;
   }
   R[j+1]=tmp;
}
```

希尔排序的时间复杂度约为$O(n^{1.3})$。

为什么希尔排序比直接插入排序好？

例如：有10个元素要排序。

直接插入排序

大约时间=10^2=100

希尔排序

d=5：分为5组，时间约为5×2^2＝20

＋

d=2：分为2组，时间约为2×5^2＝50

＋

d=1：分为1组，几乎有序，时间约为10

＝ 80

希尔排序算法不稳定的反例：希尔排序法是一种不稳定的排序算法。

【例9-1】希尔排序的组内排序采用的是_____。

A.直接插入排序　　B.折半插入排序

C.快速排序　　D.归并排序

说明：本题为2015年全国考研题

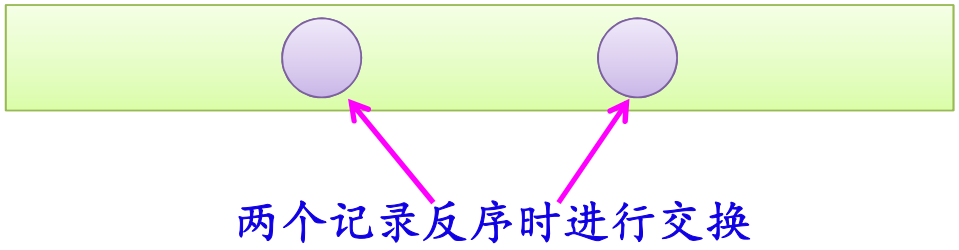
思考题：

插入排序中每趟产生的有序区是全局有序吗？

该区域的元素位置不再改变

9.3 交换排序

基本思路

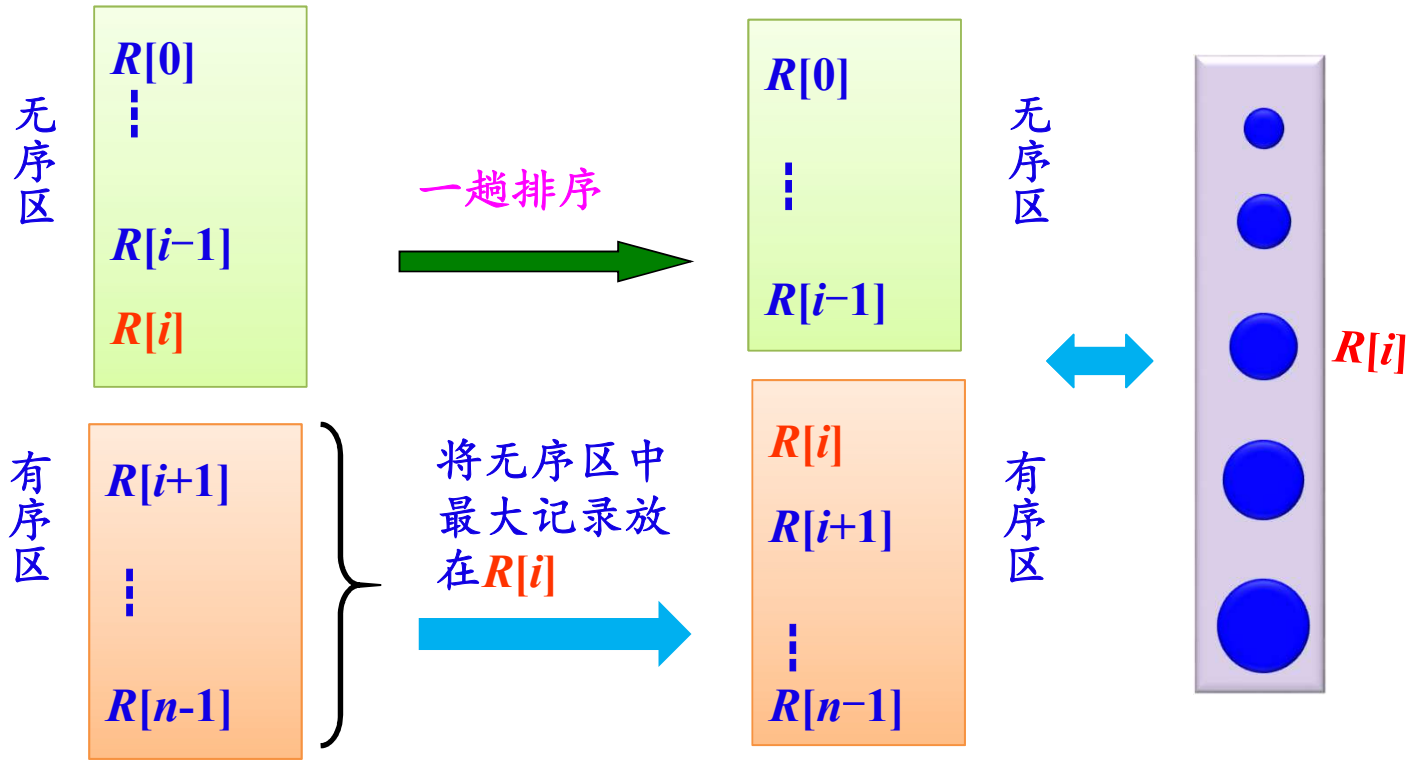


常见的交换排序方法：

（1）冒泡排序（或起泡排序）

（2）快速排序

9.3.1 冒泡排序



初始有序区为空。

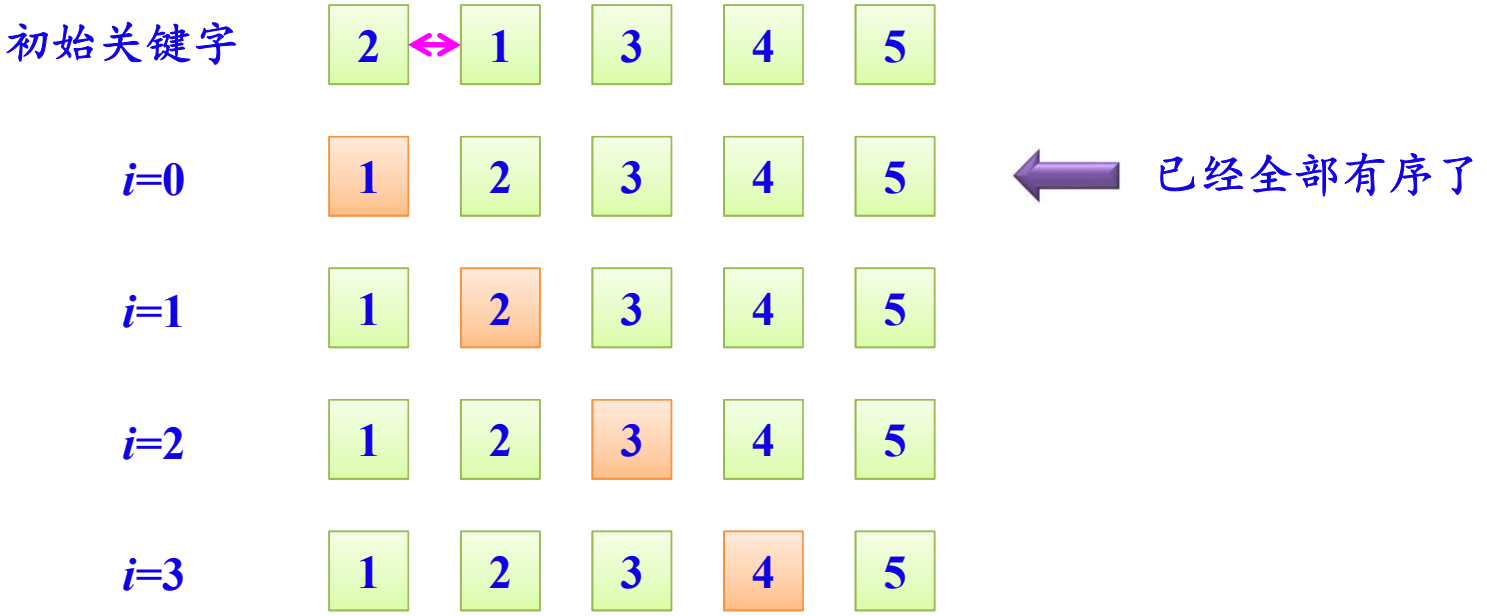
$i=1\sim n-1$，共$n-1$趟使整个数据有序。

有序区总是全局有序的

冒泡排序算法

```
void BubbleSort(RecType R[], int n)
{   int i, j; RecType temp;
    for (i=1;i<=n-1;i++)
    {
        for (j=0;j<n-i;j++)          //比较找本趟最大关键字的记录
            if (R[j].key>R[j+1].key)
            {     temp=R[j];       //R[j]⇔R[j-1]
                  R[j]=R[j+1];
                  R[j+1]=temp;
            }
    }
}
```

采用前面的冒泡排序方法对(2，1，3，4，5) 进行排序



如何提高效率？

一旦某一趟比较时不出现记录交换，说明已排好序了，就可以结束本算法。

改进冒泡排序算法：

```
void   BubbleSort(RecType R[]，int n)
{      int i，  j;  bool exchange;  RecType temp;
       for (i=1;i<=n-1;i++)
       {
            exchange=false;
            for (j=0;j<n-i;j++)      //比较，找出最小关键字的记录
               if (R[j].key>R[j+1].key)
               {   temp=R[j];  R[j]=R[j+1];  R[j+1]=temp;
                   exchange=true;
               }
            if (exchange==false) return;  //中途结束算法
       }
}
```

算法分析

最好的情况（关键字在记录序列中正序）：只需进行一趟冒泡

“比较”的次数：

$n-1$

“移动”的次数：

0

最坏的情况（关键字在记录序列中反序）：需进行$n-1$趟冒泡

“比较”的次数：

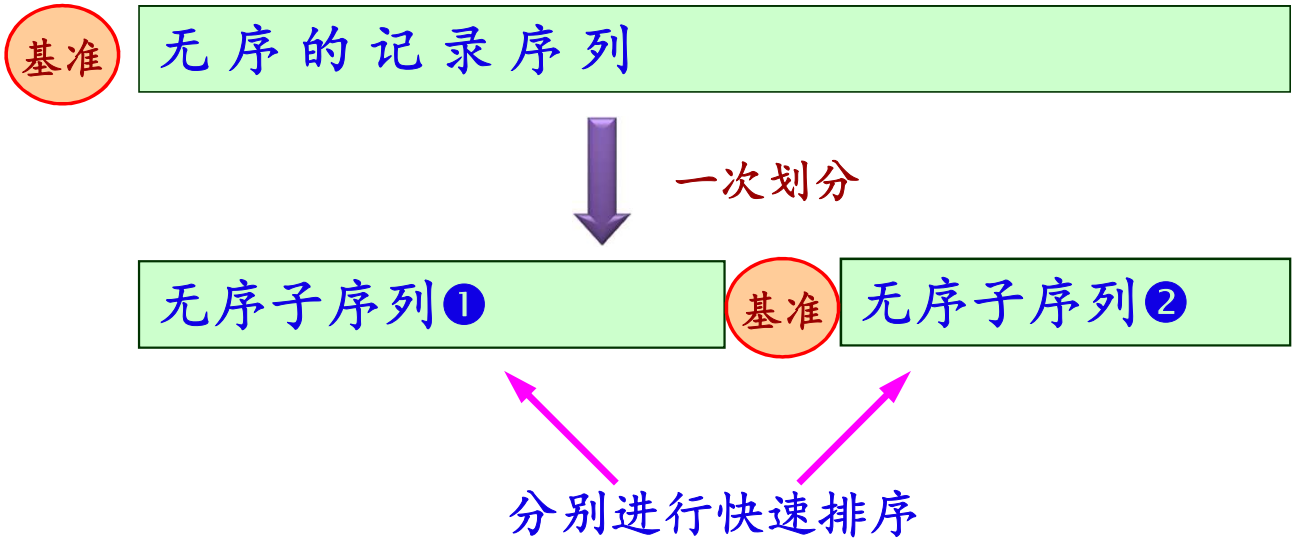
$$\sum_{i=0}^{n-1}(n-i-1)=\frac{n(n-1)}{2}$$

“移动”的次数：

$$\sum_{i=0}^{n-2}3(n-i-1)=\frac{3n(n-1)}{2}$$

所以冒泡排序最好时间复杂度为$O(n)$，最坏和平均为$O(n^2)$。

9.3.2 快速排序



每趟使表的第1个元素放入适当位置（归位），将表一分为二，对子表按递归方式继续这种划分，直至划分的子表长为0或1（递归出口）。

例： 初始关键字：

	x							
	27	38	13	49	76	97	65	50
	i	i	i	i	j	j	j	j

完成一趟排序：(27 38 13) 49 (76 97 65 50)

分别进行快速排序：(13) 27 (38) 49 (50 65) 76 (97)

快速排序结束：13 27 38 49 50 65 76 97

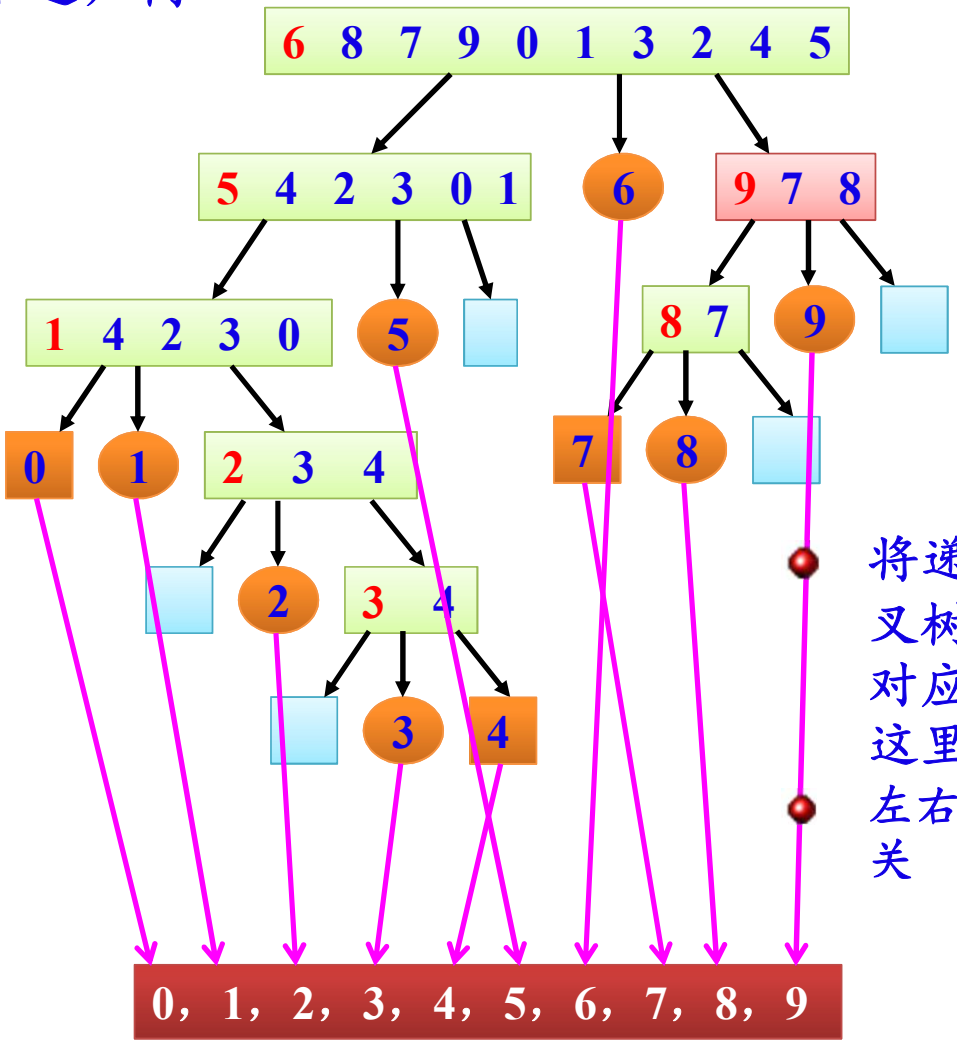
快速排序算法

```
void QuickSort(RecType R[]，int s，int t)
//对R[s]至R[t]的元素进行快速排序
{     int i=s，j=t;          RecType tmp;
      if (s<t)                          //区间内至少存在2个元素的情况
      {   tmp=R[s];                     //用区间的第1个记录作为基准
          while (i!=j)                  //两端交替向中间扫描，直至i=j为止
          {   while (j>i && R[j].key>=tmp.key)  j--;
              R[i]=R[j];
              while (i<j && R[i].key<=tmp.key)  i++;
              R[j]=R[i];
          }
          R[i]=tmp;
          QuickSort(R，s，i-1);   //对左区间递归排序
          QuickSort(R，i+1，t);   //对右区间递归排序
      }
      //递归出口：不需要任何操作
}
```

一次划分

【例9-4】设待排序的表有10个记录，其关键字分别为（6，8，7，9，0，1，3，2，4，5）。说明采用快速排序方法进行排序的过程。

快速排序递归树



将递归树看成一颗3叉树，每个分支结点对应一次递归调用。

这里递归次数：7

左右分区处理的顺序无关

【例（补充）】采用递归方式对顺序表进行快速排序，下列关于递归次数的叙述中，正确的是_____。

A. 递归次数与初始数据的排列次序无关

B. 每次划分后，先处理较长的分区可以减少递归次数

C. 每次划分后，先处理较短的分区可以减少递归次数

D. 递归次数与每次划分后得到的分区处理顺序无关

说明：本题为2010年全国考研题

【例（补充）】为实现快速排序法，待排序序列宜采用存储方式是_____。

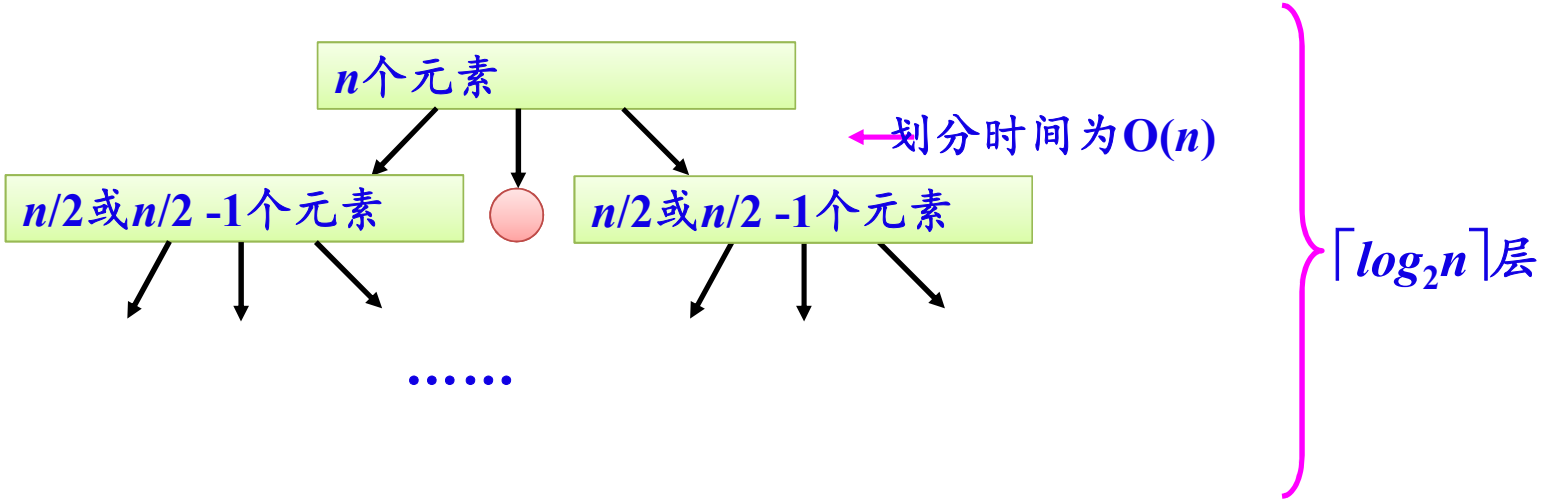
A. 顺序存储　　B. 散列存储

C. 链式存储　　D. 索引存储

说明：本题为2011年全国考研题

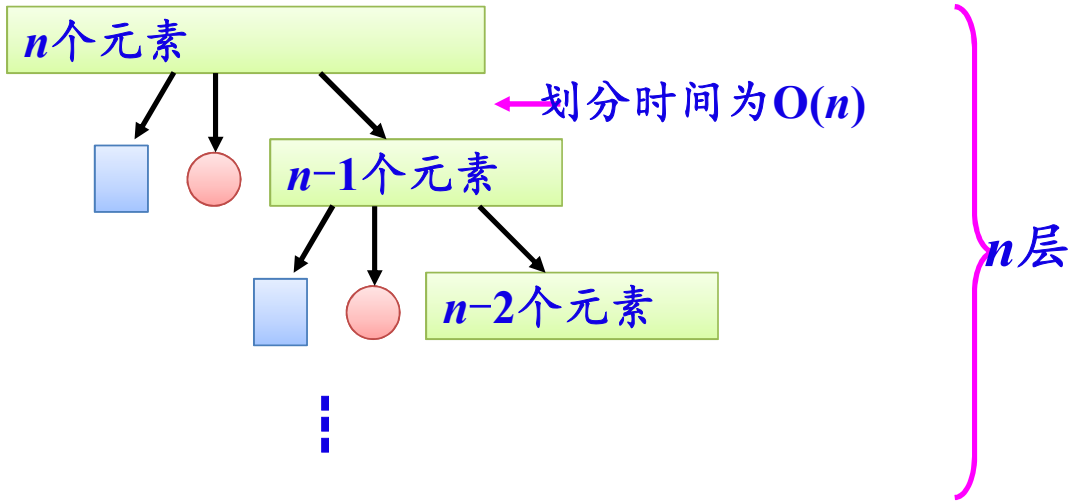
算法分析

最好情况：



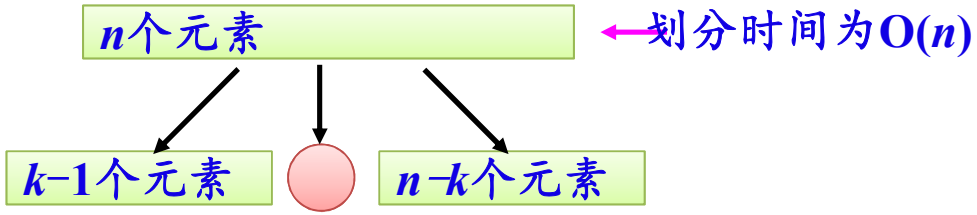
此时时间复杂度为$O(nlog_2n)$，空间复杂度为$O(\log_2 n)$。

最坏情况：



此时时间复杂度为$O(n^2)$，空间复杂度为$O(n)$。

平均情况：



k：1~n，共有n种情况

由此可得快速排序所需时间的平均值为：

$$T_{avg}(n) = Cn + \frac{1}{n}\sum_{k=1}^{n}\left[T_{avg}(k-1) + T_{avg}(n-k)\right]$$

1次划分的时间

则可得结果： $T_{avg}(n)=Cn\log_2 n$。

结论：快速排序的平均时间复杂度为$O(n\log_2 n)$。

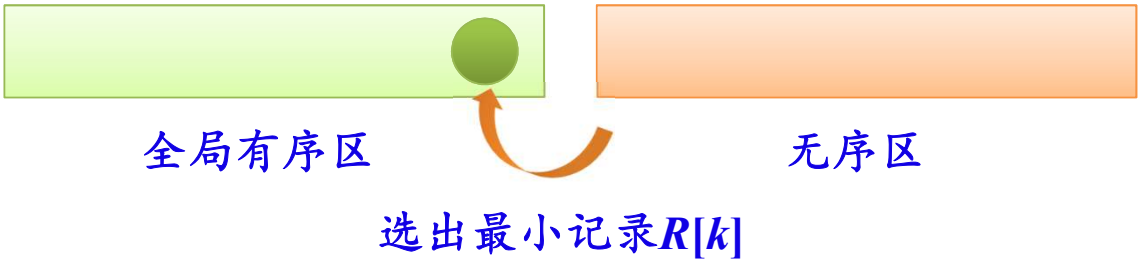
平均所需栈空间为$O(\log_2 n)$。

思考题

快速排序的最坏时间复杂度为$O(n^2)$，与冒泡排序相同。为什么快速排序更好？

9.4 选择排序

基本思路



常见的选择排序方法：

（1）简单选择排序（或称直接选择排序）

（2）堆排序

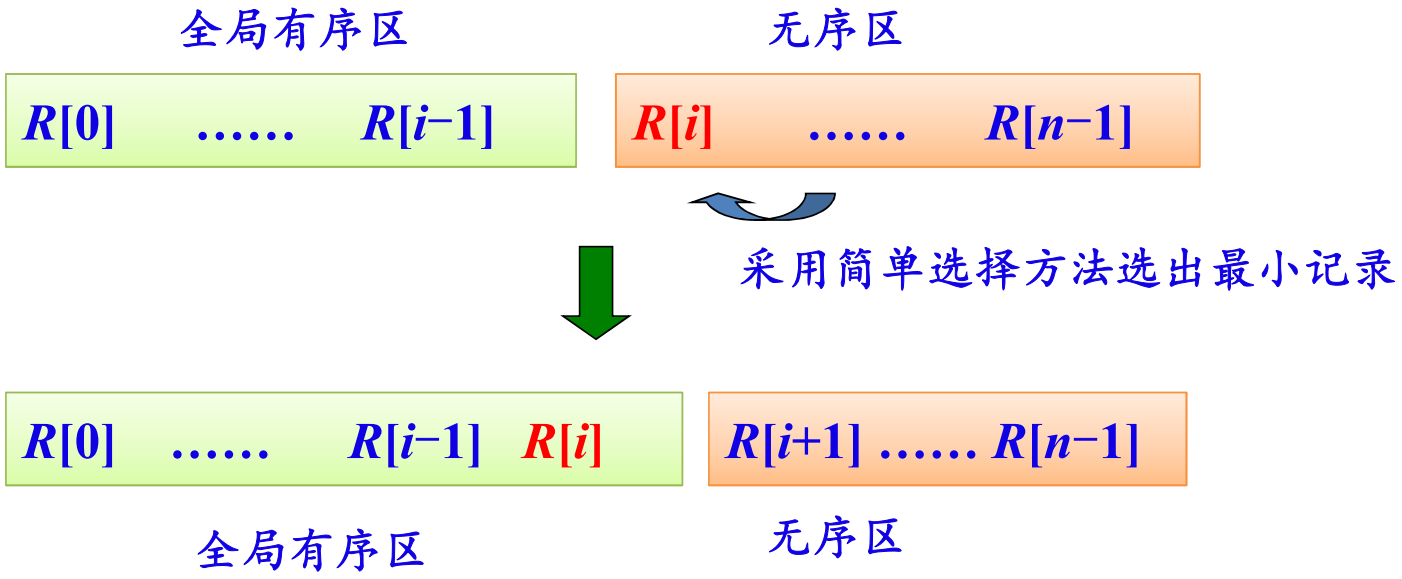
9.4.1 简单选择排序

从*a*[*i*..*n*−1]中选出最小元素：

```
int Min(int a[]，int n，int i)
{     int k=i,   j;            //k保存最小元素的下标
      for (j=i+1;j<n;j++)
            if (a[j]<a[k])  k=j;
      return a[k];
}
```

简单选择　　*n*个记录中找最小记录需要*n*−1次比较

基本思路



初始时，全局有序区为空

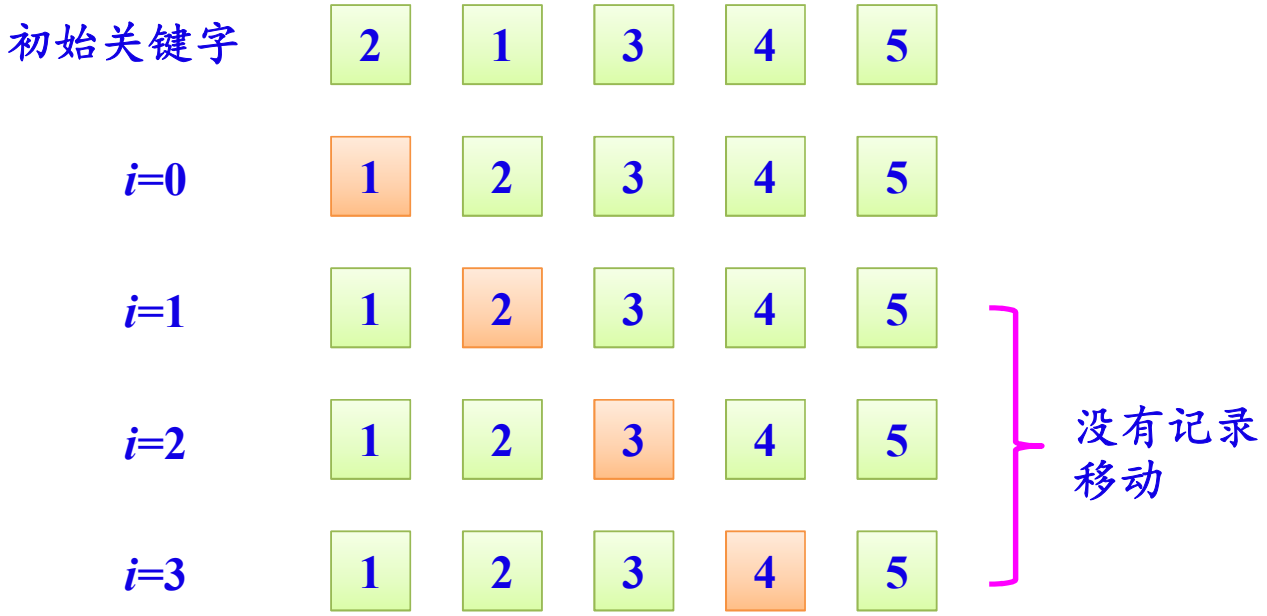
$i=0\sim n-2$，共经过$n-1$趟排序

简单选择排序算法

```
void SelectSort(RecType R[]，int n)
{   int i，j，k; RecType tmp;
    for (i=0;i<n-1;i++)                    //做第i趟排序
    {   k=i;
        for (j=i+1;j<n;j++)
            if (R[j].key<R[k].key)
                k=j;
        if (k!=i)                          //R[i]⇔R[k]
        {   tmp=R[i]; R[i]=R[k]; R[k]=tmp; }
    }
}
```

在*R*[*i*..*n*-1]中采用简单选择方法选出最小的*R*[*k*]

采用简单选择排序方法对(2，1，3，4，5) 进行排序



任何情况下：都有做$n-1$趟

算法分析

从i个记录中挑选最小记录需要比较$i-1$次。

第i（$i=0\sim n-2$）趟从$n-i$记录中挑选最小记录需要比较$n-i-1$。

对 n 个记录进行简单选择排序，所需进行的关键字的比较次数 总计为：

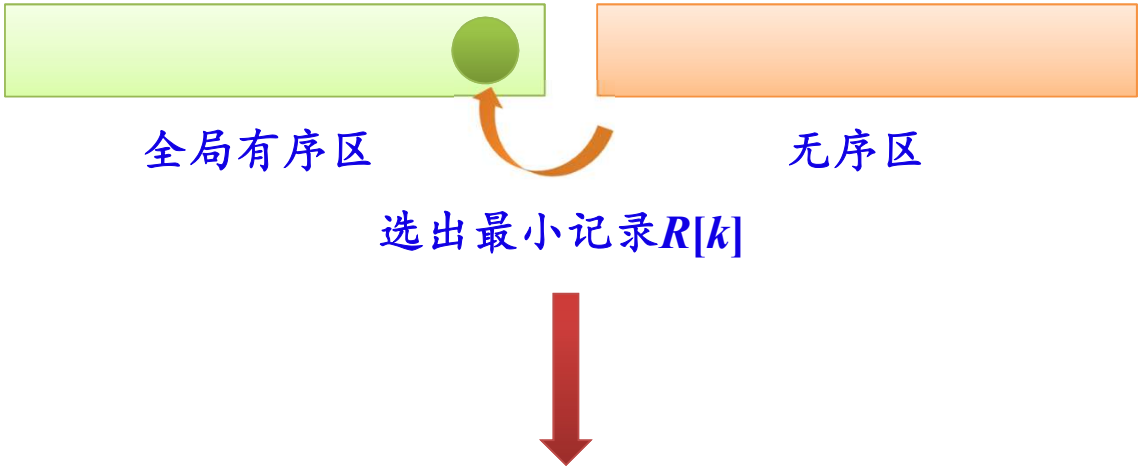
$$\sum_{i=0}^{n-2}(n-i-1)=\frac{n(n-1)}{2}$$

移动记录的次数，正序为最小值 0，反序为最大值$3(n-1)$ 。

简单选择排序的最好、最坏和平均时间复杂度为$O(n^2)$。

9.4.2 堆排序

基本思路



1、堆的定义

一个序列$R[1..n]$，关键字分别为k_1、k_2、…、k_n。

该序列满足如下性质（简称为堆性质）：

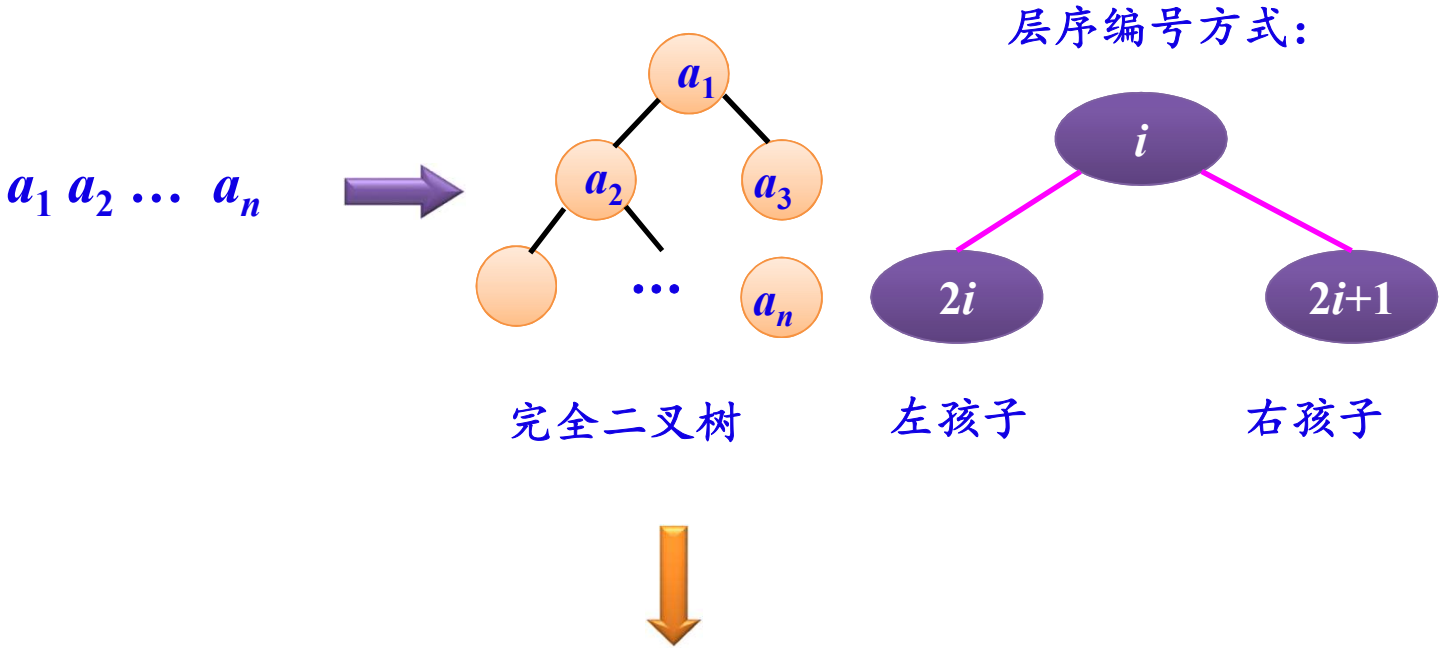
❶ $k_i \leqslant k_{2i}$ 且 $k_i \leqslant k_{2i+1}$

或 ❷ $k_i \geqslant k_{2i}$ 且 $k_i \geqslant k_{2i+1}$　　$(1 \leqslant i \leqslant \lfloor n/2 \rfloor)$

满足第❶种情况的堆称为小根堆，满足第❷种情况的堆称为大根堆。

下面讨论的堆是大根堆。

❶ 将序列$a_1\ a_2\ \cdots\ a_n$看成是一颗完全二叉树



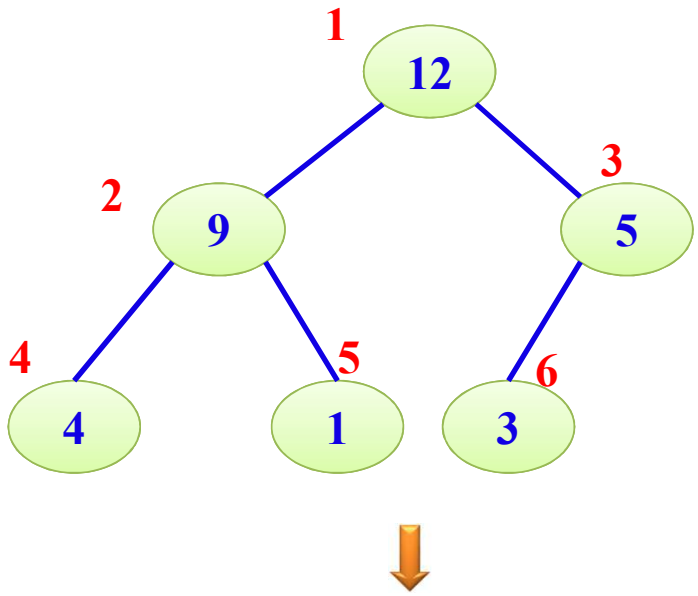
大根堆：对应的完全二叉树中，任意一个结点的关键字都大于或等于它的孩子结点的关键字。

最小关键字的记录一定是某个叶子结点！！！

❷ 如何判断一颗完全二叉树是否为大根堆

n=6

从编号为$n/2$=3的结点开始，
逐一判断所有分支结点

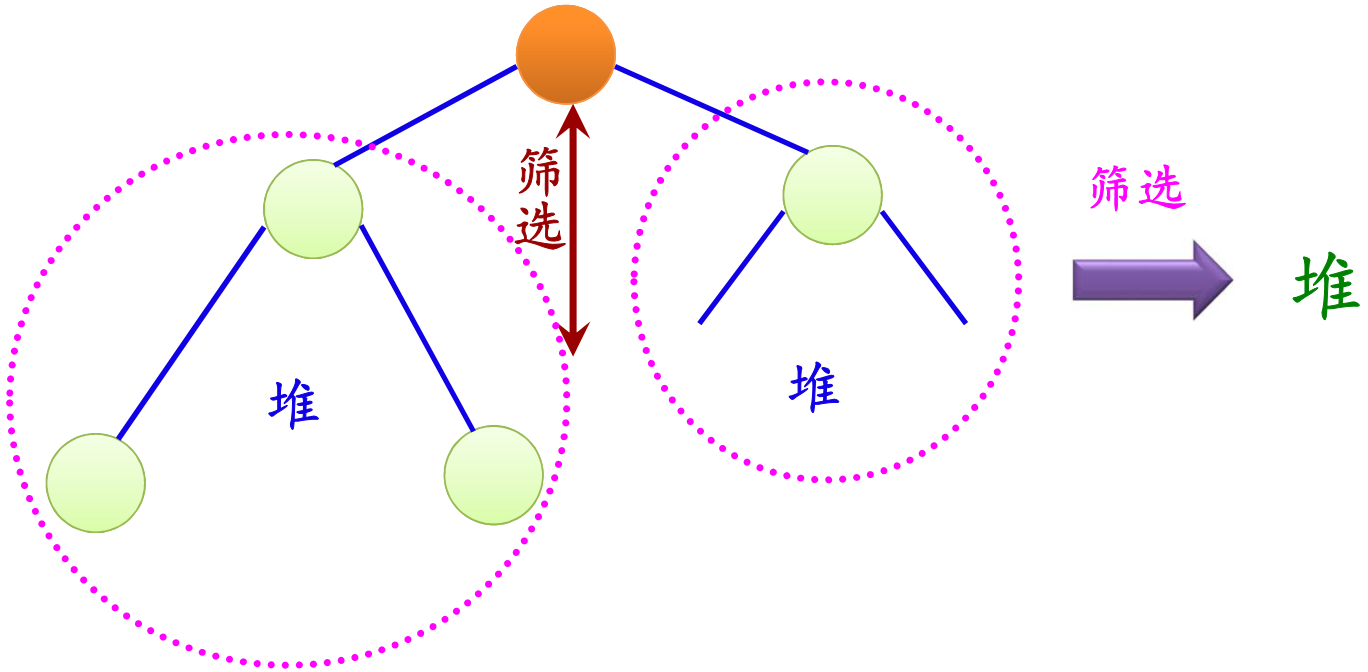


所有分支结点满足定义
⇨ 为大根堆

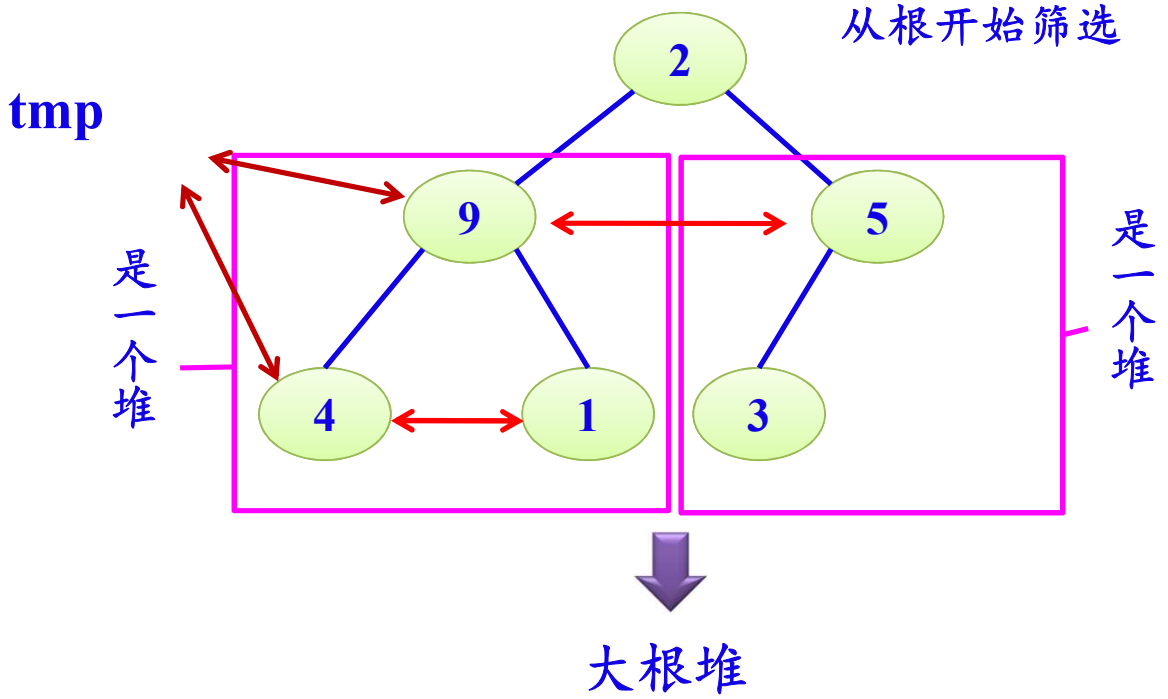
2、堆排序算法设计

堆排序的关键是构造堆，这里采用筛选算法建堆。

所谓“筛选”指的是，对一棵左/右子树均为堆的完全二叉树，“调整”根结点使整个二叉树也成为一个堆。



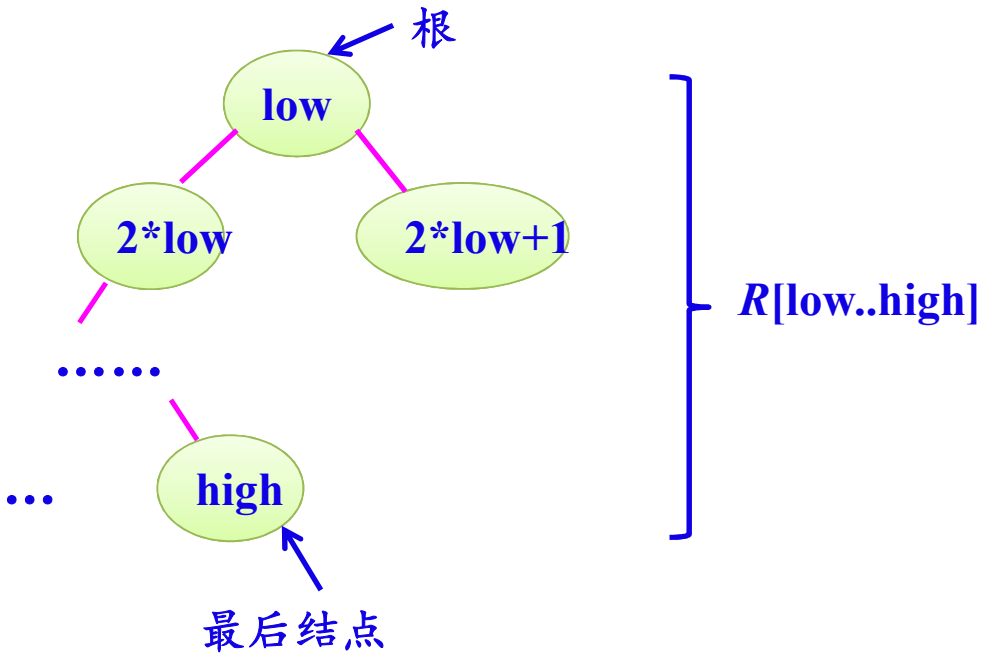
❶ 筛选：不是堆 ⇨ 堆



- 仅仅处理从根结点⇨某个叶子结点路径上的结点
- n个结点的完全二叉树高度为$\lceil \log_2(n+1) \rceil$
- 所有筛选的时间复杂度为$O(\log_2 n)$

筛选算法

sift(RecType R[], int low, int high): *R*[low . . high]



筛选或调整算法：

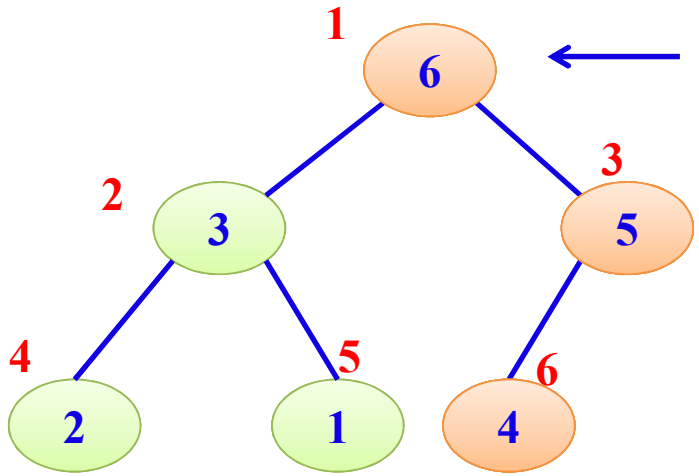
```
void sift(RecType R[]，int low，int high) //调整堆的算法
{    int i=low， j=2*i;                  //R[j]是R[i]的左孩子
     RecType tmp=R[i];
     while (j<=high)
     {
         if (j<high && R[j].key<R[j+1].key) j++;
         if   (tmp.key<R[j].key)         //双亲小
         {    R[i]=R[j];                 //将R[j]调整到双亲结点位置上
              i=j;                       //修改i和j值，以便继续向下筛选
              j=2*i;
         }
         else break;                     //双亲大：不再调整
     }
     R[i]=tmp;
}
```

❷ 一颗完全二叉树 ⇨ 初始堆

例如,序列：（4，3，5，2，1，6）n=6

从编号为$n/2$=3的结点开始，逐一筛选

```
for (i=n/2;i>=1;i--)     //循环建立初始堆
    sift(R，i，n);
```



筛选步骤：

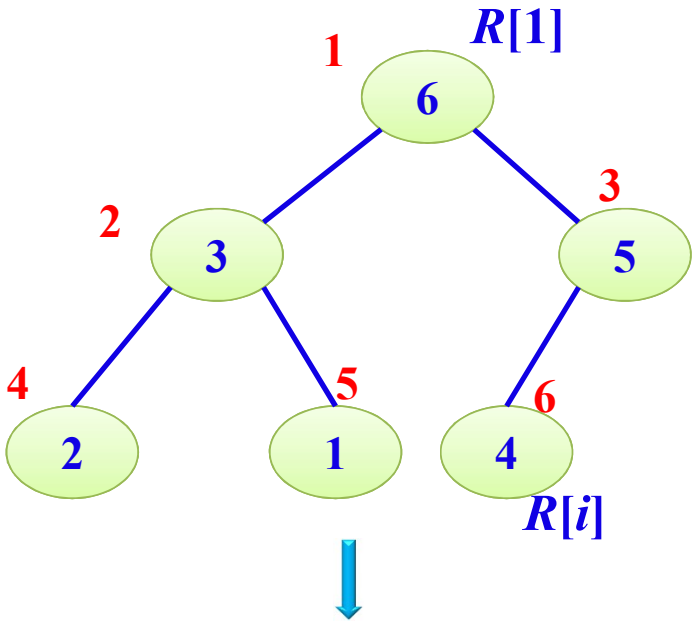
① sift(*R*，3，6)

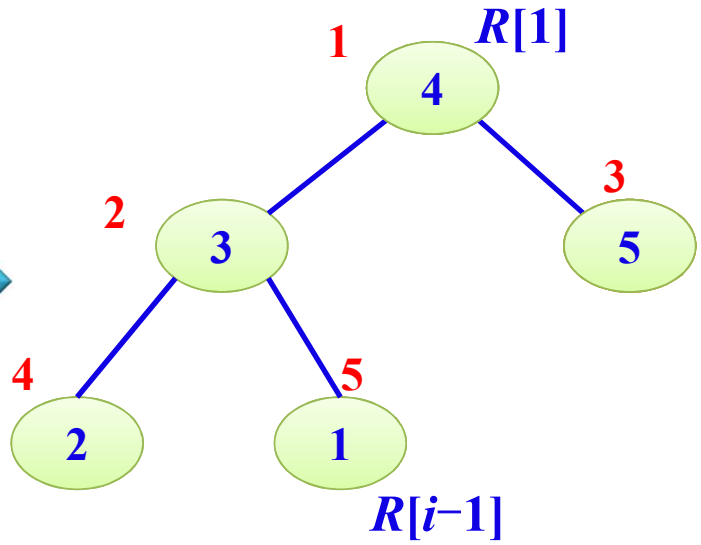
② sift(*R*，2，6)

③ sift(*R*，1，6)

初始堆:（6，3，5，2，1，4）

❸ 最大记录归位





4，3，5，2，1，6

最大记录6归位

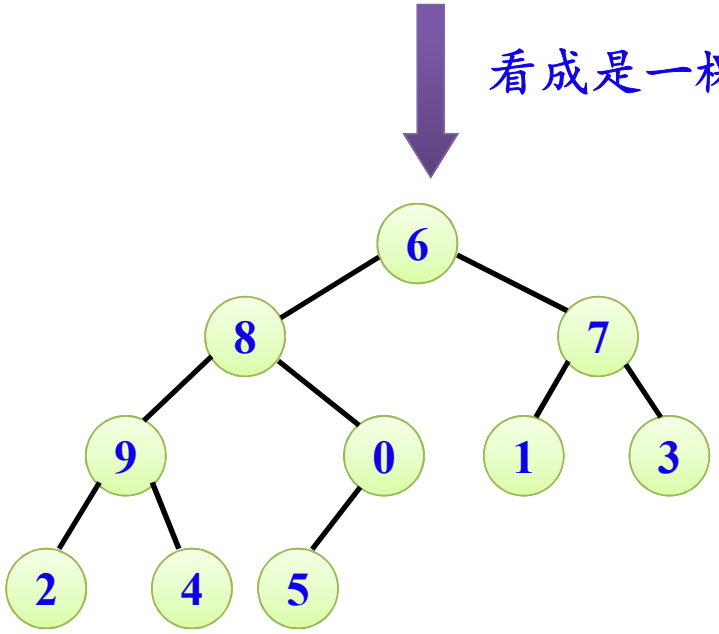
再对$R[1..i-1]$的记录进行筛选

堆排序算法：

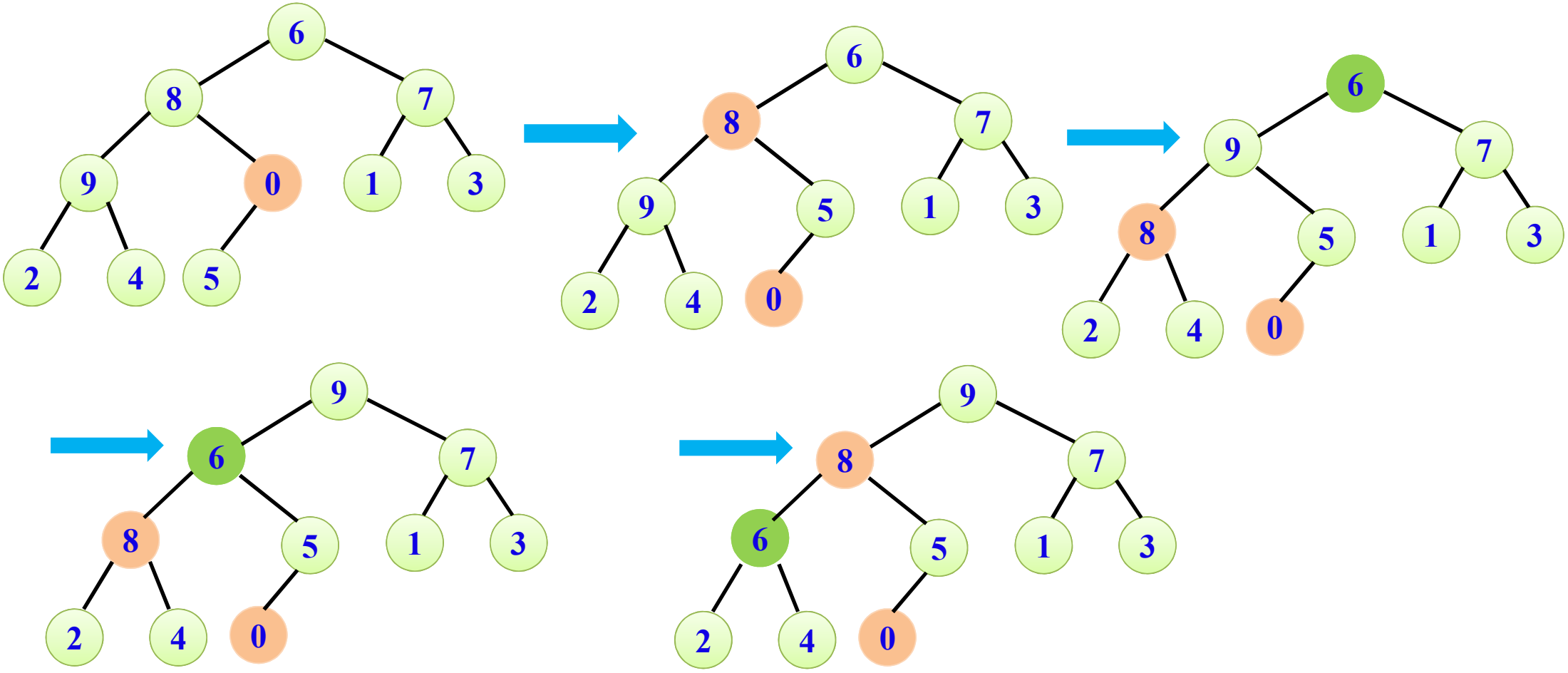
```
void HeapSort(RecType R[]，int n)
{    int i;  RecType tmp;
     for (i=n/2;i>=1;i--)     //循环建立初始堆
         sift(R，i，n);
     for (i=n; i>=2; i--)     //进行n-1次循环，完成推排序
     {
         temp=R[1];          //R[1] ⇔ R[i]
         R[1]=R[i];  R[i]=tmp;
         sift(R，1，i-1);    //筛选R[1]结点，得到i-1个结点的堆
     }
}
```

【例9-6】设待排序的表有10个记录，其关键字分别为{6，8，7，9，0，1，3，2，4，5}。说明采用堆排序方法进行排序的过程。

排序序列：6，8，7，9，0，1，3，2，4，5

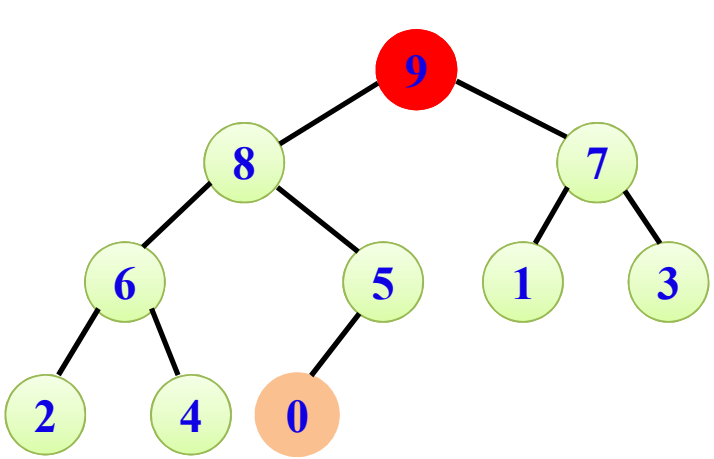


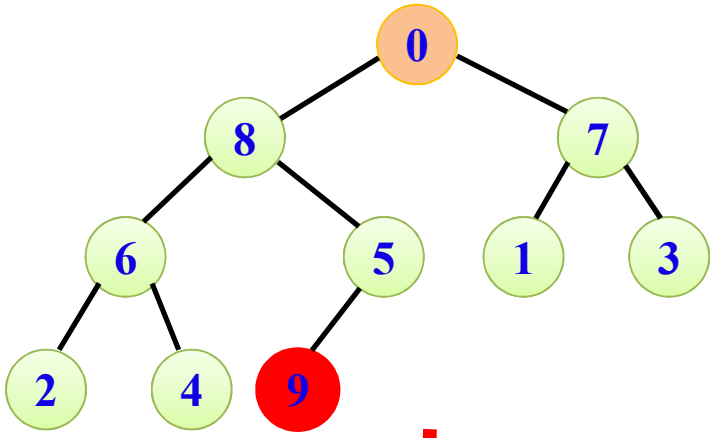
调整成初始大根堆：



调整完毕，成为一个大根堆　9 8 7 6 5 1 3 2 4 0

第1趟排序

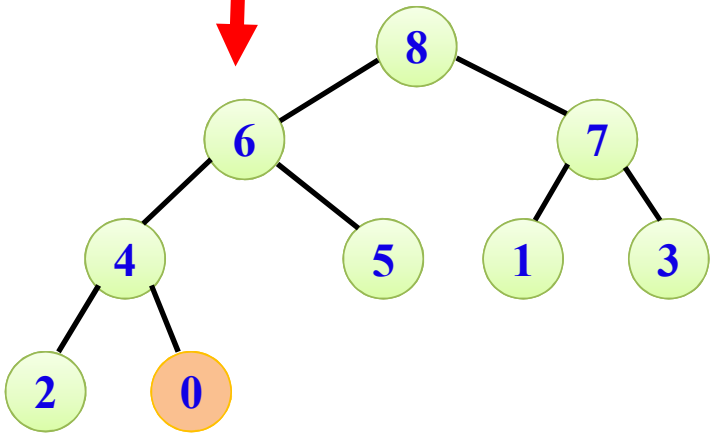




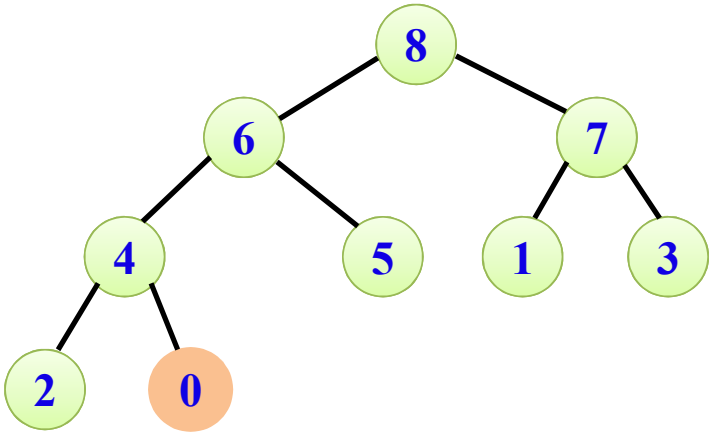
0 8 7 6 5 1 3 2 4 9

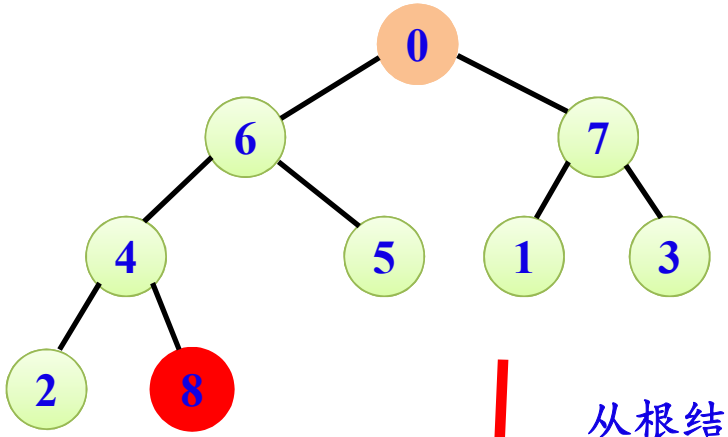
输出9（归位）

从根结点筛选

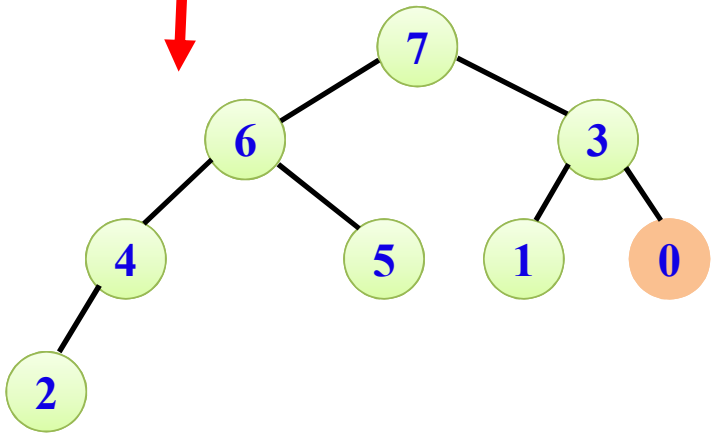


第2趟排序





从根结点筛选



0 6 7 4 5 1 3 2 8 9

输出8（归位）

其他各趟排序依此进行

最终结果：

0	1	2	3	4	5	6	7	8	9

3、堆排序算法分析

❶ 对高度为 h 的堆，一次“筛选”所需进行的关键字比较的次数至多为$2(h-1)$。

❷ 对 n 个关键字，建成高度为h（$=\lfloor \log_2 n \rfloor+1$）的堆，所需进行的关键字比较的次数不超过$4n$。

❸ 调整“堆顶” $n-1$ 次，总共进行的关键字比较的次数不超过：

$$2\left(\lfloor \log_2(n-1) \rfloor+\lfloor \log_2(n-2) \rfloor+\cdots+\log_2 2\right) < 2n(\lfloor \log_2 n \rfloor)$$

因此，堆排序的时间复杂度为$O(n\log n)$。

空间复杂度为$O(1)$，不稳定。

数据结构经典算法的启示

简单选择排序算法

利用了连续多次查找最大记录的特性

堆排序算法

在操作系统中，将多个进程放在一个队列中，每个进程有一个优先级，总是出队优先级最高的进程执行。

采用优先队列，用堆来实现！

【例（补充）】设有1000个无序的整数，希望用最快的速度挑选出其中前10个最大的元素，最好选用_____排序方法。

A.冒泡排序　　B.快速排序

C.堆排序　　D.直接插入排序

n=1000，k=10

- 冒泡排序的大致时间：kn
- 堆排序的大致时间：$4n+k\log_2 n$。

思考题：

选择排序的整体时间性能与初始序列的顺序有关吗？

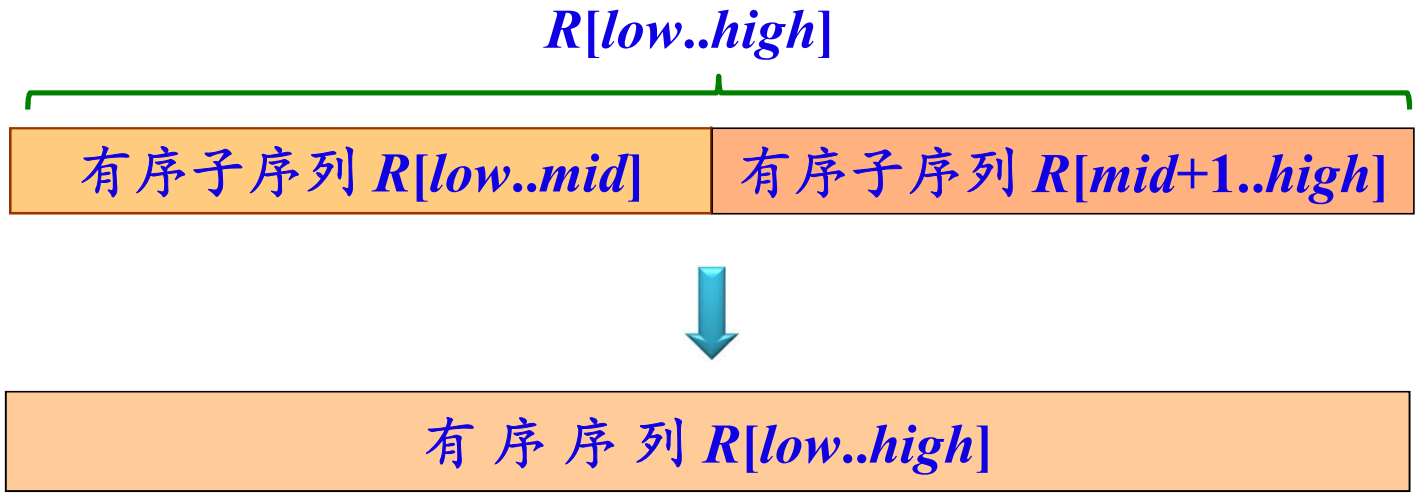
9.5 归并排序

1、归并的思路

归并排序是多次将相邻两个或两个以上的有序表合并成一个新的有序表。

最简单的归并是将相邻两个有序的子表合并成一个有序的表，即二路归并排序。

一次二路归并：将两个位置相邻的记录有序子序列归并为一个记录的有序序列。



2、二路归并算法

❶ Merge()：一次二路归并，将两个相邻的有序子序列归并为一个有序序列。

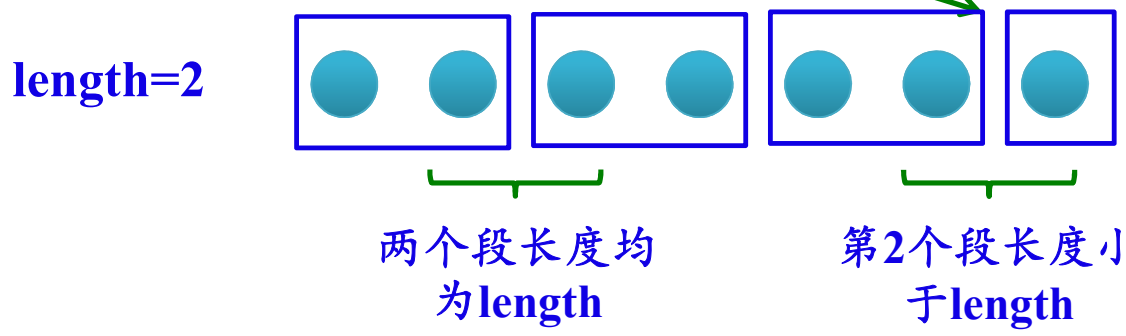
```
void Merge(RecType R[]，int low，int mid，int high)
{    RecType *R1;
     int i=low，  j=mid+1，  k=0;
    //k是R1的下标，i、j分别为第1、2段的下标
     R1=(RecType *)malloc((high-low+1)*sizeof(RecType));
     while (i<=mid && j<=high)
        if (R[i].key<=R[j].key)       //将第1段中的记录放入R1中
        {   R1[k]=R[i]; i++;k++;  }
        else                          //将第2段中的记录放入R1中
        {   R1[k]=R[j]; j++;k++;  }
```

空间复杂度为O(high-low+1)

```
        while (i<=mid)        //将第1段余下部分复制到R1
        {     R1[k]=R[i];  i++;k++;  }
        while (j<=high)       //将第2段余下部分复制到R1
        {     R1[k]=R[j];  j++;k++;  }
        for (k=0，i=low;i<=high;k++，i++) //将R1复制回R中
              R[i]=R1[k];
        free(R1);
}
```

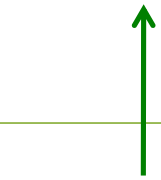
❷ **MergePass()：一趟二路归并（段长度为length ）。**

```
void MergePass(RecType R[]，int length，int n)
{    int i;
     for (i=0;i+2*length-1<n;i=i+2*length)  //归并length长的两相邻子表
         Merge(R，i，i+length-1，i+2*length-1);
     if (i+length-1<n)                      //余下两个子表，后者长度小于length
         Merge(R，i，i+length-1，n-1);     //归并这两个子表
}
```



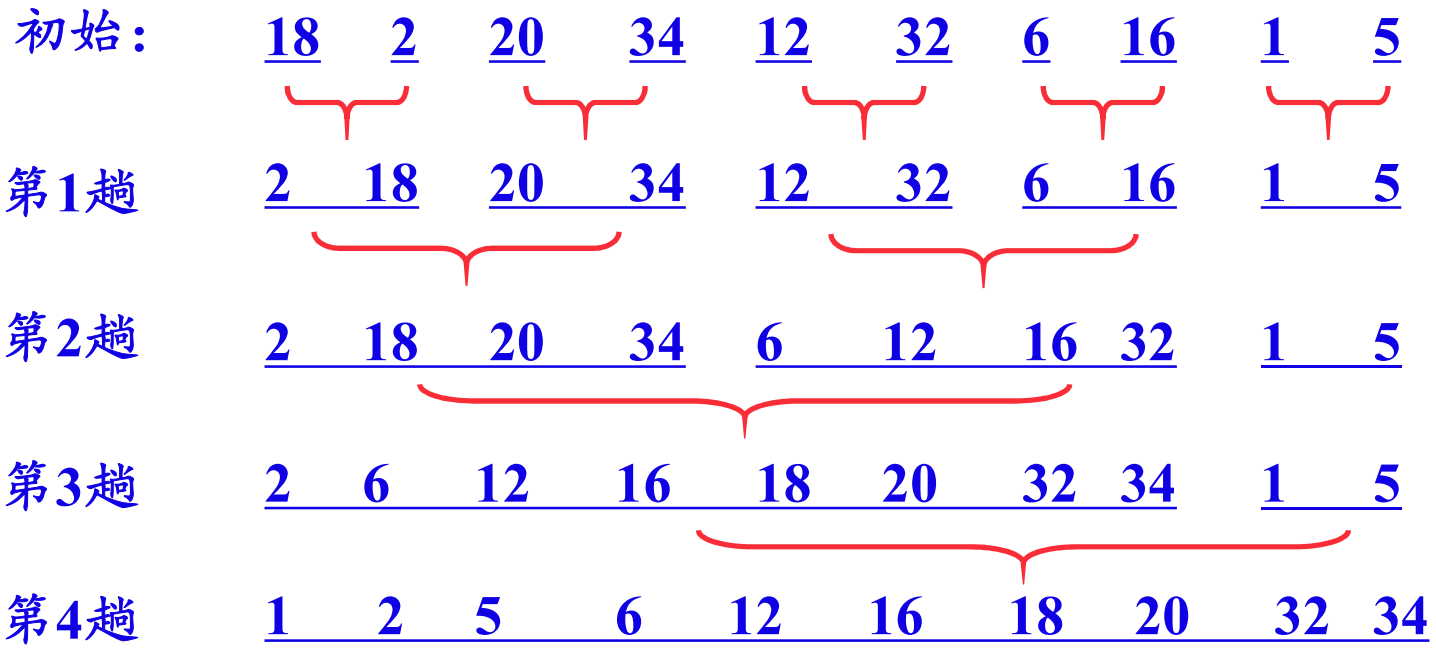
❸ MergeSort()：二路归并排序算法：

```
void MergeSort(RecType R[]，int n)
{     int length;
      for (length=1;length<n;length=2*length)
          MergePass(R，length，n);
}
```



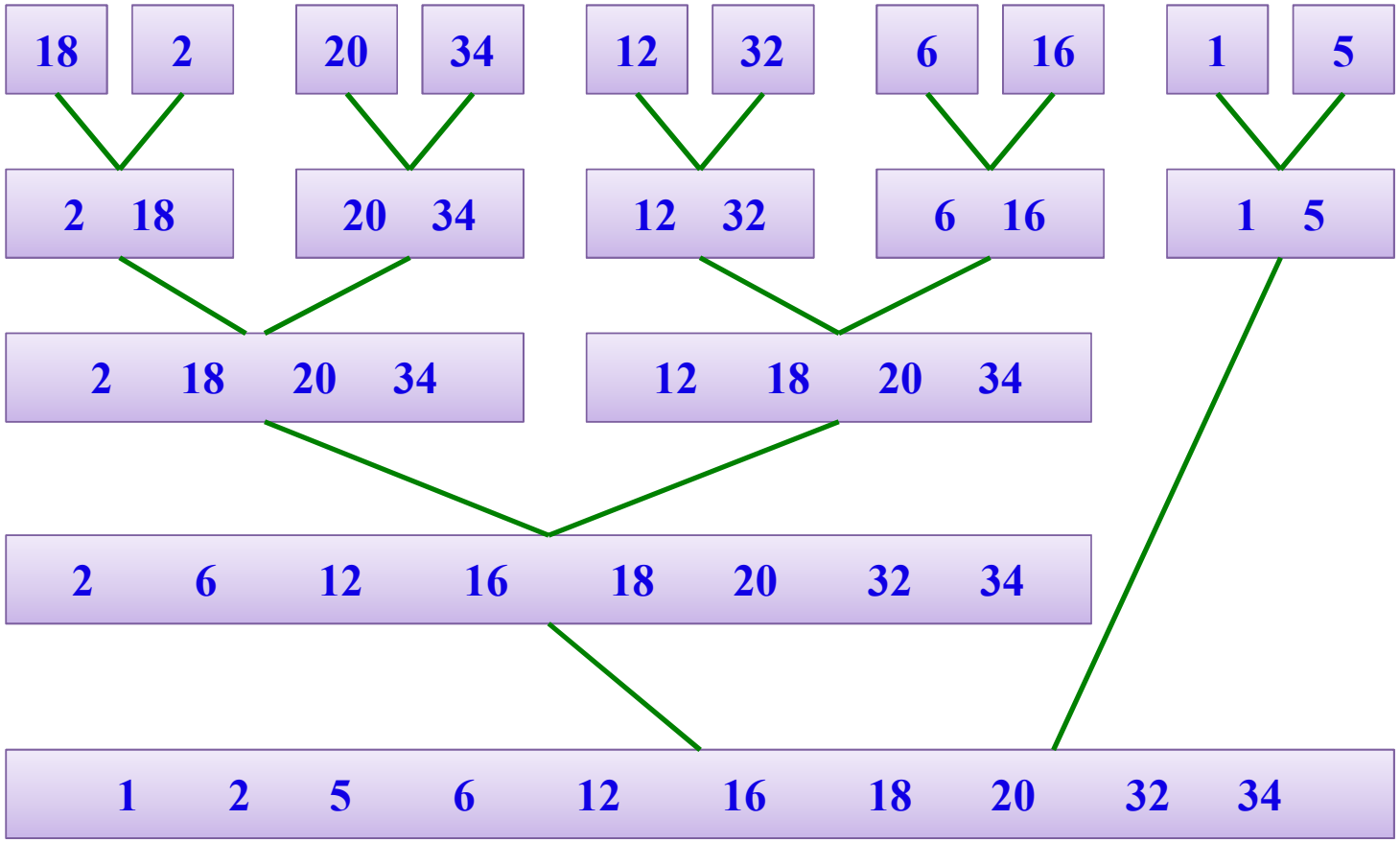
$\lceil \log_2 n \rceil$趟

【例（补充）】 设待排序表有10个记录，其关键字分别为(18，2，20，34，12，32，6，16，1，5)。说明采用归并排序方法进行排序的过程。



需要$\log_2 10$取上界即4趟

更清楚的表示

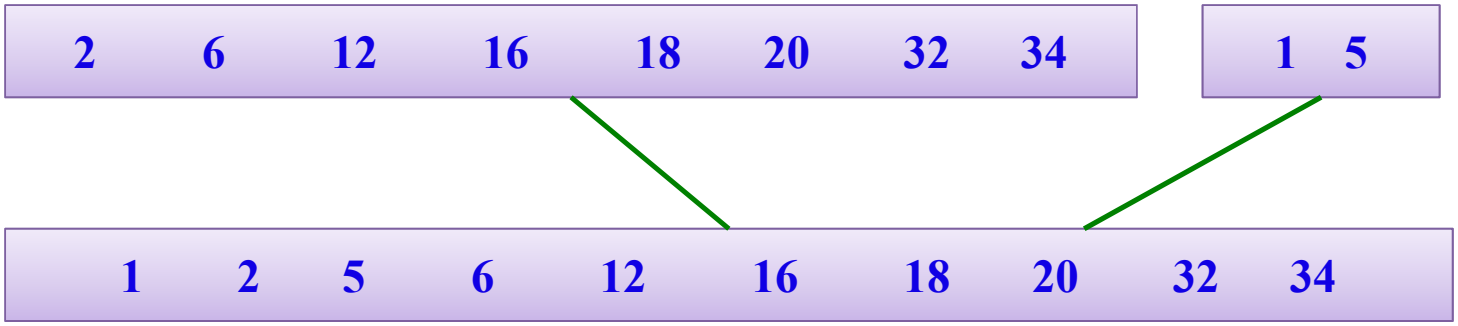


一颗归并树

3、归并算法分析

- 每一趟归并的时间复杂度为 $O(n)$
- 总共需进行$\lceil \log_2 n \rceil$趟。
- 二路归并排序的时间复杂度为$O(n\log_2 n)$。

每一次二路归并后临时空间都会释放。而最后的一次二路归需要全部记录参加归并：



占用临时空间为全部记录个数n

所以空间复杂度为$O(n)$。

【例（补充）】数据序列(5，4，15，10，3，2，9，6，8)是某排序方法第一趟后的结果，该排序算法可能是_____。

A.冒泡排序　　B.二路归并排序

C. 堆排序　　D.简单选择排序

第一趟：{5，4，15，10，3，2，9，6，8}

相邻的两个元素都是递减的

【例（补充）】就排序算法所用的辅助空间而言，堆排序、快速排序和归并排序的关系是_____。

A.堆排序＜快速排序＜归并排序

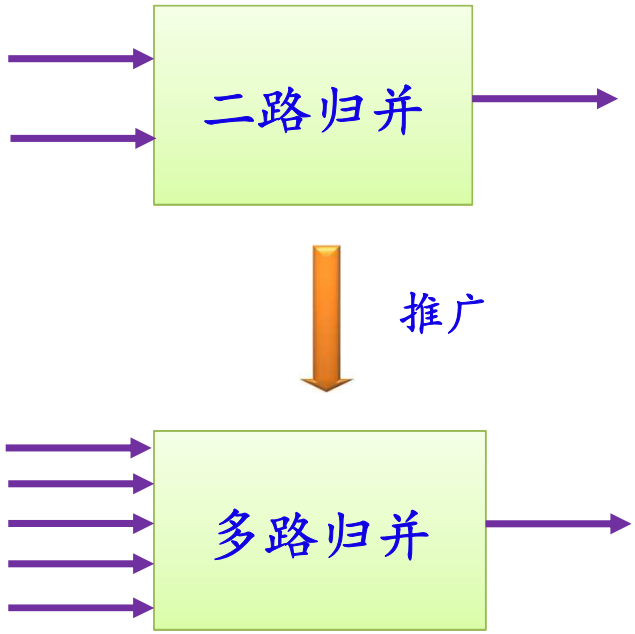
B.堆排序＜归并排序＜快速排序

C.堆排序＞归并排序＞快速排序

D.堆排序＞快速排序＞归并排序

堆排序、快速排序、归并排序

O(1)　　$O(\log_2 n)$　　$O(n)$



思考题：以三路归并为例，多路归并算法设计有哪些难点？

三路归并和二路归并的时间比较

三路归并的时间复杂度为$O(n\log_3 n)$

$n\log_3 n=n\log_2 n/\log_2 3=O(n\log_2 n)$

同一个级别！

9.6 基数排序

1、 基数排序的概念

基数r：对于二进制数r为2，对于十进制数r为10。

示例： 2 3 9 ← r=10

百位　十位　个位

一般地

记录$R[j]$的关键字$R[j].key$ ⇨ $k_j^{d-1}k_j^{d-2}\ldots k_j^1k_j^0$

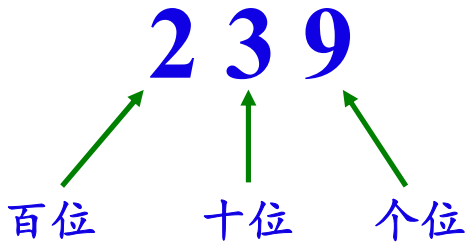
最高位

d位数字或字符组成
每一位的值都在0～r-1范围内，其中r称为基数

最低位

2、基数排序的分类

基数排序有两种：最低位优先（LSD）和最高位优先（MSD）。



- 最低位优先：从个位⇨百位
- 最高位优先：从百位⇨个位

选择哪种基数排序，需要根据数据的特点来定。例如，对整数序列递增排序，选择最低位优先，越重要的位越在后面排序。

最低位优先排序过程如下：

对$i = 0, 1, \cdots, d-1$，依次做一次“分配”和“收集”（使用r个队列Q_0，Q_1，…，Q_{r-1}。）。

- 分配：开始时，把Q_0，Q_1，…，Q_{r-1}各个队列置成空队列，然后依次考察线性表中的每一个结点a_j（$j=0, 1, \cdots, n-1$），如果a_j的关键字$k_j^i=k$，就把a_j放进Q_k队列中。

- 收集：按Q_0，Q_1，…，Q_{r-1}顺序把各个队列中的结点首尾相接，得到新的结点序列，从而组成新的线性表。

由于数据需要放入队列，又要从队列取出来，需要大量元素移动。所以排序数据和队列均采用链表存储更好。

例如(369，367，167，239，237，138，230，139)⇨基数排序

p →369 →367 →167 →239 →237→138 →230 →139

建立10个队列，*f*为队头，*r*为队尾

❶ 进行第1次分配：按个位

f[0]→230 ← *r*[0]

f[7]→367 →167→237 ← *r*[7]

f[8]→138 ← *r*[8]

f[9]→369 →239→139 ← *r*[9]

进行第1次收集

- 分配时是按一个一个元素进行的
- 收集时是按一个一个队列进行的

p →230 →367 →167 →237 →138 →369 →239 →139

第1趟排序完毕

p →230→367 →167 →237 →138 →369 →239 →139

❷ 进行第2次分配：按拾位

f[3] →230 →237 →138 →239 →139 ← *r*[3]

f[6] →367 →167 →369 ← *r*[6]

进行第2次收集

p →230 →237 →138 →239 →139 →367 →167 →369

第2趟排序完毕

p →230 →237 →138 →239 →139 →367 →167 →369

❸ 进行第3次分配：按百位

f[1] →138 →139 →167 ← *r*[1]

f[2] →230 →237 →239 ← *r*[2]

f[3] →367 →369 ← *r*[3]

进行第3次收集

p →138 →139 →167 →230 →237 →239 →367 →369

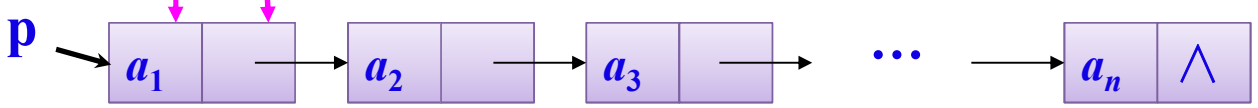
第3趟排序完毕

结论：

基数排序是通过“分配”和“收集”过程来实现排序，不需要关键字的比较。

3、 基数排序算法

```
#define MAXE 20                 //线性表中最多元素个数
#define MAXR 10                 //基数的最大取值
#define MAXD 8                  //关键字位数的最大取值
typedef struct node
{    char data[MAXD];           //记录的关键字定义的字符串
     struct node *next;
} RecType1;                     //单链表中每个结点的类型
```



基数排序数据的存储结构

```
void RadixSort(RecType1 *&p，int r，int d)
//p为待排序序列链表指针，r为基数，d为关键字位数
{    RecType1 *head[MAXR]，  *tail[MAXR]，  *t; //定义各链队的首尾指针
     int i，  j，  k;
     for (i=0;i<d;i--)                    //从低位到高位做d趟排序
     {      for (j=0;j<r;j++)             //初始化各链队首、尾指针
                head[j]=tail[j]=NULL;

     while (p!=NULL)              //对于原链表中每个结点循环
     {     k=p->data[i]-'0';      //找第k个链队
           if (head[k]==NULL)  //进行分配，即采用尾插法建立单链表
           {    head[k]=p;  tail[k]=p;  }
           else
           {    tail[k]->next=p;  tail[k]=p;   }
           p=p->next;                  //取下一个待排序的结点
     }
```

分配

```
    p=NULL;
    for (j=0;j<r;j++)      //对于每一个链队循环进行收集
        if (head[j]!=NULL)
        {       if (p==NULL)
                {     p=head[j];
                      t=tail[j];
                }
                else
                {     t->next=head[j];
                      t=tail[j];
                }
        }
     t->next=NULL;      //最后一个结点的next域置NULL
  }
}
```

收集

排序完成后，*p*指向的是一个有序单链表。

注意：C/C++将数值转换为字符串：

2 1 0

1 2 3　　应该从3 ⇨ 1，即低位到高位

“3 2 1”为了从3 ⇨ 1，应该高位到低位

2 1 0

4、基数排序算法分析

基数排序的时间复杂度为$O(d(n+r))$

其中：分配为$O(n)$

收集为$O(r)$（r为“基数”）

d为“分配-收集”的趟数

基数排序的空间复杂度为$O(r)$

【例（补充）】以下排序方法中，_____不需要进行关键字的比较。

A.快速排序　　B.归并排序

C.基数排序　　D.堆排序

思考题

基数排序中为什么不需要进行关键字的比较？

9.7 各种内排序的比较

各种内排序方法的性能

排序方法	时间复杂度			空间复杂度	稳定性
	平均情况	最坏情况	最好情况		
直接插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
折半插入排序	$O(n^2)$	$O(n^2)$	$O(n\log_2 n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$			$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(\log_2 n)$	不稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
二路归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定

1、按算法平均时间复杂度分类

- 平方阶$O(n^2)$：即简单排序方法，例如直接插入、简单选择和冒泡排序。
- 线性对数阶$O(n\log_2 n)$：例如快速、堆和归并排序。
- 线性阶$O(n)$：例如基数排序（假设r、d为常量）。

2、按算法空间复杂度分类

- O(n)：归并排序，基数排序为O(r)。
- O($\log_2 n$)：快速排序。
- O(1)：其他排序方法。

3、按算法稳定性分类

- 不稳定的：希尔排序、快速排序、堆排序、简单选择排序。
- 稳定的：其他排序方法。

【例9-9】设线性表中每个元素有两个数据项k_1和k_2，现对线性表按以下规则进行排序：先看数据项k_1，k_1值小的在前，大的在后；在k_1值相同的情况下，再看k_2，k_2值，小的在前，大的在后。满足这种要求的排序方法是_____。

A.先按k_1值进行直接插入排序，再按k_2值进行简单选择排序

B.先按k_2值进行直接插入排序，再按k_1值进行简单选择排序

C.先按k_1值进行简单选择排序，再按k_2值进行直接插入排序

D.先按k_2值进行简单选择排序，再按k_1值进行直接插入排序

解：考虑1：排序数据项顺序：k_1、k_2还是k_2、k_1？

越重要的数据项越在后面排序 ⇨ 应为k_2、k_1

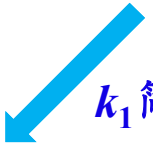
考虑2：k_2选择直接插入排序还是简单选择排序？　稳定性

例如：

标识	k_1	k_2
1	80	90
2	70	90
3	80	72
4	70	72

k_2直接插入排序

标识	k_1	k_2
3	80	72
4	70	72
1	80	90
2	70	90



k_1简单选择排序

标识	k_1	k_2	
2	70	90	相对次序改变
4	70	72	
1	80	90	相对次序改变
3	80	72	

标识	k_1	k_2
1	80	90
2	70	90
3	80	72
4	70	72

k_2简单选择排序

标识	k_1	k_2
3	80	72
4	70	72
1	80	90
2	70	90

k_1直接插入排序

标识	k_1	k_2
4	70	72
2	70	90
3	80	72
1	80	90

相对次序不改变

相对次序不改变

答案为D。

思考题

排序算法的稳定性在多关键字排序中如何使用？

4、如何选择合适的排序算法

因为不同的排序方法适应不同的应用环境和要求，所以选择合适的排序方法应综合考虑下列因素：

- 待排序的元素数目n（问题规模）；
- 元素的大小（每个元素的规模）；
- 关键字的结构及其初始状态；
- 对稳定性的要求；
- 语言工具的条件；
- 排序数据的存储结构；
- 时间和辅助空间复杂度。

【例（补充）】若数据元素序列{11,12,13,7,8,9,23,4,5}是采用下列排序方法之一得到的第二趟排序后的结果，则该排序算法只能是_____。

A. 冒泡排序　　B. 直接插入排序

C. 选择排序　　D. 二路归并排序

说明：本题为2009年全国考研题

【例（补充）】对一组数据(2,12,16,88,5,10)进行排序，若前三趟的结果如下：

第1趟：2，12，16，5，10，88

第2趟：2，12，5，10，16，88

第3趟：2，5，10，12，16，88

则采用的排序方法可能是_____。

A. 冒泡排序　　B. 希尔排序

C. 二路归并排序　　D. 基数排序

说明：本题为2010年全国考研题

——本章完——