

# 第十章 内部排序

## 10.1 概述

1. 排序——将文件或表中的记录，通过某种方法整理成按关键字大小次序排列的处理过程。

假定 $n$ 个记录的文件为

$$(R_1, R_2, \dots, R_n)$$

对应的关键字为

$$(K_1, K_2, \dots, K_n)$$

则排序是确定如下一个排列

$$p_1, p_2, \dots, p_n$$

使得： $K_{p_1} \leq K_{p_2} \leq \dots \leq K_{p_n}$

从而得到一个有序文件

$$(R_{p_1}, R_{p_2}, \dots, R_{p_n})$$

# 学生成绩表

	学 号	姓 名	数 学	外 语
1	20051	刘大海	80	75
2	20042	王 伟	90	83
3	20066	吴晓英	82	88
4	20038	刘 伟	80	70
5	20052	王 洋	60	70

(a) 无序表

	学 号	姓 名	数 学	外 语
1	20038	刘 伟	80	70
2	20042	王 伟	90	83
3	20051	刘大海	80	75
4	20052	王 洋	60	70
5	20066	吴晓英	82	88

(b) 按学号排列的有序表

	学 号	姓 名	数 学	外 语
1	20052	王 洋	60	70
2	20051	刘大海	80	75
3	20038	刘 伟	80	70
4	20066	吴晓英	82	88
5	20042	王 伟	90	83

(c) 按数学成绩排列的有序表

	学 号	姓 名	数 学	外 语	总 分
1	20042	王 伟	90	83	173
2	20066	吴晓英	82	88	170
3	20051	刘大海	80	75	155
4	20038	刘 伟	80	70	150
5	20052	王 洋	60	70	130

(d) 按总分成绩排列的有序表

## 2. 什么是排序的稳定性

假设在待排序的文件中，存在两个具有相同关键字的记录 $R(i)$ 与 $R(j)$ ，其中 $R(i)$ 位于 $R(j)$ 之前。在用某种排序法排序之后， $R(i)$ 仍位于 $R(j)$ 之前，则称这种排序方法是**稳定的**；否则，称这种排序方法是**不稳定的**。

例 数列

$(10, 25, 22, 42, \underline{25}, 30, 18)$     稳定的排序     $(10, 18, 22, 25, \underline{25}, 30, 42)$   
 $(10, 25, 22, 42, \underline{25}, 30, 18)$     不稳定的排序     $(10, 18, 22, \underline{25}, 25, 30, 42)$

	学 号	姓 名	数 学	外 语
1	20051	刘大海	80	75
2	20042	王 伟	90	83
3	20066	吴晓英	82	88
4	20038	刘 伟	80	70
5	20052	王 洋	60	70

不稳定的排序

	学 号	姓 名	数 学	外 语
1	20052	王 洋	60	70
2	20038	刘 伟	80	70
3	20051	刘大海	80	75
4	20066	吴晓英	82	88
5	20042	王 伟	90	83

(e) 按数学成绩排列的有序表

3. **内部排序(内排序)**——指待排序文件不大，一次可以在内存中完成的排序。即在排序进行的过程中不使用计算机外部存储器的排序过程。排序速度快。

**外部排序(外排序)**——指待排序文件较大，文件存放在外存上，不能一次调入内存的排序。即在排序进行的过程中需要对外存进行访问的排序过程。排序速度慢。

本章主要讨论各种内部排序的方法。

#### 4. 待排序的记录和顺序表(文件)的数据类型

```
#define MAXSIZE 20           //最大长度
typedef int KeyType;         //关键字类型
typedef struct               //记录类型
{
    KeyType key;             //关键字
    InfoType otherinfo;     //其它数据类型
} RecType;                  //记录类型名

typedef struct
{
    RecType r[MAXSIZE+1];   //r[0]用作监视哨
    int length;              //实际表长
} SeqList;                  //记录表类型
```

或 表和表长分别定义和说明

```
RecType r[MAXSIZE+1];       //r[0]用作监视哨
int length;                  //实际表长
```

## 5. 排序算法分析

### (1) 时间复杂度

- 对 $n$ 个记录排序，所需**比较关键字的次数**；  
最好情况；最坏情况；平均情况
- 对 $n$ 个记录排序，所需**移动记录的次数**；  
最好情况；最坏情况；平均情况

### (2) 空间复杂度

排序过程中，除文件中的记录所占的空间外，所需的**辅助存储空间**的大小。

## 6. 内排序方法

### (1) 对顺序表的排序

- 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 (Shell) 排序；
- 交换排序：冒泡排序：单向冒泡排序，双向冒泡排序  
快速排序
- 选择排序：简单选择排序；  
树形选择排序；  
堆排序

## ➤ 归并排序

2-路归并排序

k-路归并排序

## ➤ 基数排序

多关键字排序

最高位优先法

最低位优先法

链式基数排序

## (2) 对单链表的排序

直接插入, 简单选择, 冒泡排序, 基数排序





Donald Ervin Knuth  
高德纳（1938-）  
斯坦福大学教授  
1974年图灵奖得主

《计算机程序设计的艺术》

The Art of Computer Programming

第一卷：基本算法 1968

第二卷：半数字化算法 1969

第三卷：排序与搜索 1973

第四卷：组合算法

《计算机与排版》TEX排版软件

《作文式程序设计》

《超现实数》

《具体数学》

## 10.2 插入排序

### 算法基本思想

将待排序的记录插入到已排序的子文件中，使得插入之后得到的子文件仍然是有序子文件。插入一个记录，首先要对有序子文件进行查找，以确定这个记录的插入位置。按查找方式的不同，插入排序又可以分为线性插入排序和折半插入排序，前者使用顺序查找，后者使用折半查找。

## 1. 直接插入排序(线性插入排序)

设待排序的文件为:  $(r[1], r[2], \dots, r[n])$

关键字为:  $(r[1].key, r[2].key, \dots, r[n].key)$

首先, 将初始文件中的记录 $r[1]$ 看作有序子文件;

第1遍: 将 $r[2]$ 插入有序子文件中, 若:

$$r[2].key < r[1].key,$$

则 $r[2]$ 插在 $r[1]$ 之前; 否则, 插在 $r[1]$ 的后面。

第2遍: 将记录 $r[3]$ 插入前面已有2个记录的有序子文件中, 得到3个记录的有序子文件。

以此类推, 依次插入 $r[4], \dots, r[n]$ , 最后得到 $n$ 个记录的递增有序文件。

例. 直接插入排序, 设K0为“监视哨”

	K0	K1	K2	K3	K4	K5	K6
初始关键字:	21	( 43 )	21	89	15	43	28
第1遍排序后: (43后移)	89	( 21 43 )	89	15	43	28	
第2遍排序后: (不后移)	15	( 21 43 89 )	15	43	28		
第3遍排序后: (89, 43, 21后移)	43	( 15 21 43 89 )	43	28			
第4遍排序后: (89后移)	28	( 15 21 43 43 89 )	28				
第5遍排序后: (89, 43, 43后移)	28	( 15 21 28 43 43 89 )					

## 直接插入排序算法

```
void InsertSort(RecType r[], int n)
// 对数组r[1..n]中的n个记录作插入排序
{ int i, j;
  for (i=2; i<=n; i++) {
    r[0]=r[i];           //待插记录r[i]存入监视哨中
    j=i-1;               //已排序的范围1 — i-1
                        //从r[i-1]开始向左扫描
    while(r[0].key<r[j].key)
    { r[j+1]=r[j];        //记录后移
      j--;                //继续向左扫描
    }
    r[j+1]=r[0];          //插入记录r[0], 即原r[i]
  }
}
```

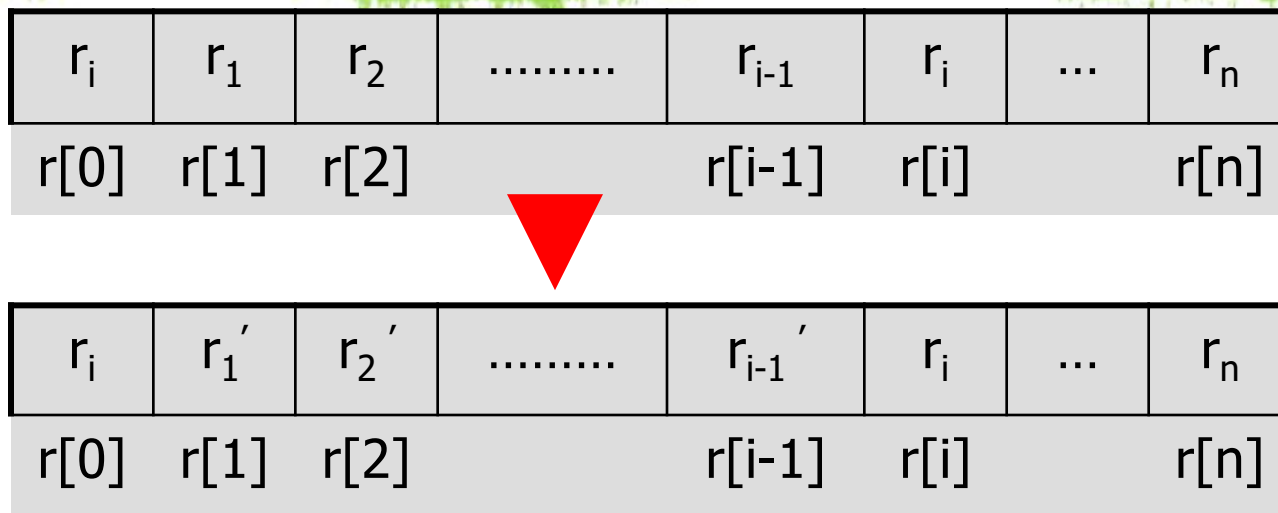
## 直接插入排序算法分析:

(1) **最好情况**, 原 $n$ 个记录递增有序:

比较关键字 $n-1$ 次

移动记录 $2(n-1)$ 次, (将数据复制到 $r[0]$ 后又复制回来)

$r_i$	$r_1$	$r_2$	.....	$r_{i-1}$	$r_i$	...	$r_n$
$r[0]$	$r[1]$	$r[2]$		$r[i-1]$	$r[i]$		$r[n]$



(2) 最坏情况，原 $n$ 个记录递减有序：

比较关键字的次数：

$$\sum_{i=2}^n i = 2+3+\dots+n = (n-1)(n+2)/2 = O(n^2)$$

移动记录的次数(个数)：

$$\begin{aligned} \sum_{i=2}^n (i-1+2) &= 3+4+\dots+(n+1) \\ &= (n-1)(n+4)/2 \text{ 次} = O(n^2) \end{aligned}$$

(3) 平均比较关键字的次数约为:

$$\sum_{i=2}^n (1+2+\dots+i) / i = \sum_{i=2}^n \frac{i+1}{2} = (3+4+\dots+(n+1)) / 2 \\ = (n-1)(n+4) / 4 = O(n^2)$$

平均移动记录的次数约为:

$$\sum_{i=2}^n ((0+2) + (1+2) + \dots + (i-1+2)) / i = \sum_{i=2}^n \frac{(i+3)}{2} = (5+6+\dots+(n+3)) / 2 \\ = (n-1)(n+8) / 2 = O(n^2)$$

故, 时间复杂度为 $O(n^2)$ 。

(4) 只需少量中间变量作为辅助空间。  $O(1)$

(5) 算法是稳定的。



## 2. 折半插入排序（二分插入排序）

- (a) 由于插入排序的基本思想是在一个有序序列中插入一个新的记录，因此可以利用“折半查找”查询插入位置，由此得到的插入排序算法为“折半插入排序”，又被称为二分法插入排序。
- (b) 直接插入排序的算法简单易行，对长度(n)很大的记录序列宜采用性能更好的插入排序算法。
- (c) 折半插入排序过程中的折半查找的目的是查询插入点，因此不论是否存在和给定值相同的关键字，结束查找过程的条件都是 $high < low$ ，并且插入位置为low指示的地方。

# 折半插入排序实例

在序列[1 14 19 23 55 84 92]已排好序的基础上，将元素15插入到序列中，最后还是一个有序序列。

初始序列: [1 14 19 23 55 84 92] 15

↑  
*low*

↑  
*mid*

↑  
*high*

15 < 23, 则  $high = mid - 1$

第一趟排序: [1 14 19 23 55 84 92] 15

↑  
*low*

↑  
*mid*

↑  
*high*

15 > 14, 则  $low = mid + 1$

第二趟排序: [1 14 19 23 55 84 92] 15

↑  
*high*  
↑ ↑  
*low mid*

15 < 19, 则  $high = mid - 1$

第三趟排序: [1 14 19 23 55 84 92] 15

↑ ↑  
*high low*  
↑  
*mid*

此时  $high < low$ , 则 *low* 为插入点

最终结果: [1 14 15 19 23 55 84 92]

# 折半插入排序算法

```
void BInsertSort (SqList &L)
{ // 对顺序表L作折半插入排序
  for ( i=2; i<=L.length; ++i )
  {
    L.r[0] = L.r[i];           //假定第一个记录有序
                                // 将L.r[i]暂存到L.r[0]
    low = 1; high = i-1;
    while (low<=high)
    { // 在r[low..high]中折半查找有序插入的位置
      m = (low+high)/2;         // 折半
      if (L.r[0].key < L.r[m].key)
        high = m-1;             // 插入点在低半区
      else
        low = m+1;              // 插入点在高半区
    } // while
    for ( j=i-1; j>=low; - -j ) L.r[j+1] = L.r[j]; // 记录后移
    L.r[high+1] = L.r[0];       // 插入
  } //for
} // BInsertSort
```

### 3. 希尔(shell)排序（缩小增量排序）

(a) 是插入排序中效率最高的一种排序方法。又称“缩小增量排序”，是由D. L. Shell在1959年提出来的。

(b) 基本思想是，先对待排序列进行“宏观调整”，待序列中的记录“基本有序”时再进行直接插入排序。

注：所谓“基本有序”是指，在序列中的各个关键字之前，只存在少量关键字比它大的记录。

(c) 做法是：先取定一个小于 $n$ 的整数 $d_1$ 作为第一个增量，把文件的全部记录分成 $d_1$ 个组，所有距离为 $d_1$ 的倍数的记录放在同一个组中，在各组内进行直接插入排序；然后，到第二个增量 $d_2 < d_1$ 重复上述分组和排序，直至所取的增量 $d_t = 1$ （ $d_t < d_{t-1} < \dots < d_2 < d_1$ ），即所有记录放在同一组中进行直接插入排序为止。

## 希尔排序的复杂度

空间复杂度：只占一个暂存单元（ $r[0]$ ）即 $S(n)=O(1)$ ；

时间复杂度：希尔排序的时间复杂度和所取增量序列相关，例如已有学者证明，当增量序列为  $2^{t-k+1}-1$  ( $k=0, 1, \dots, t$ ) 时 ( $t$  为排序趟数)，希尔排序的时间复杂度为  $O(n^{3/2})$ ，还有人在大量的实验基础上推出：当  $N$  在某特定范围内，希尔排序所需的比较和移动次数约为  $n^{1.3}$ ，当  $n \rightarrow \infty$  时，一般认为是  $O(n(\log_2 n)^2)$ 。

注：

希尔排序是一个不稳定的排序方法。

## 10.3 交换排序

### 10.3.1 冒泡排序

**基本思想：** 设待排序的文件为 $r[1..n]$

**第1趟(遍)：** 从 $r[1]$ 开始, 依次比较两个相邻记录的关键字 $r[i].key$ 和 $r[i+1].key$ , 若 $r[i].key > r[i+1].key$ , 则交换记录 $r[i]$ 和 $r[i+1]$ 的位置; 否则, 不交换。 ( $i=1, 2, \dots, n-1$ )

第1趟之后,  $n$ 个关键字中最大的记录移到了 $r[n]$ 的位置上。



















**第2趟：** 从 $r[1]$ 开始, 依次比较两个相邻记录的关键字 $r[i].key$ 和 $r[i+1].key$ , 若 $r[i].key > r[i+1].key$ , 则交换记录 $r[i]$ 和 $r[i+1]$ 的位置; 否则, 不交换。 ( $i=1, 2, \dots, n-2$ )

第2趟之后, 前 $n-1$ 个关键字中最大的记录移到了 $r[n-1]$ 的位置上。

.....

作完 $n-1$ 趟, 或者不需再交换记录时为止。

例:                      第1趟                      第2趟                      第3趟                      第4趟

											
k1=	43	21	21	21	21	21	21	21	15	15	15
											
k2=	21	43	43	43	43	43	15	15	21	21	21
											
k3=	89	89	89	15	15	15	15	28	28	28	28
											
k4=	15	15	15	89	28	28	28	43	43	43	43
											
k5=	28	28	28	28	89	43	43	43	43	43	43
											
k6=	43	43	43	43	43	89	89	89	89	89	89

比较次数=5+4+3+2=14

交换记录的次数=4+2+1=7, 移动记录次数=3\*7=21

## 冒泡排序算法(对n个整数按递增次序作冒泡排序)

```
void bubble1(int a[], int n)
{ int i, j, temp;
  for(i=0; i<n-1; i++)          //作n-1趟排序
    for(j=0; j<n-1-i; j++)
      if (a[j]>a[j+1])
        { temp=a[j];           //交换记录
          a[j]=a[j+1];
          a[j+1]=temp;
        }
  for(i=0; i<n; i++)
    printf("%d", a[i]);        //输出排序后的元素
}
```



## 改进的冒泡排序算法

```
void bubblesort(RecType r[], int n)
{
    int i, j, swap;    RecType temp;
    j=1;                //置比较的趟数为1
    do { swap=0;        //置交换标志为0
        for (i=1; i<=n-j; i++)
        {
            if (r[i].key>r[i+1].key)
            {
                temp=r[i];    //交换记录
                r[i]=r[i+1];
                r[i+1]=temp;
                swap=1;        //置交换标志为1
            }
        }
        j++;                //作下一趟排序
    } while (j<n && swap);
}
```

## 算法分析：

- **最好情况：**待排序的文件已是有序文件，只需要进行1趟排序，共计比较关键字的次数为

$$n-1 \quad \text{不交换记录。}$$

- **最坏情况：**要经过 $n-1$ 趟排序，所需总的比较关键字的次数为

$$(n-1) + (n-2) + \dots + 1 = n(n-1)/2$$

交换记录的次数最多为

$$(n-1) + (n-2) + \dots + 1 = n(n-1)/2$$

移动记录次数最多为

$$3n(n-1)/2 \text{ 。}$$

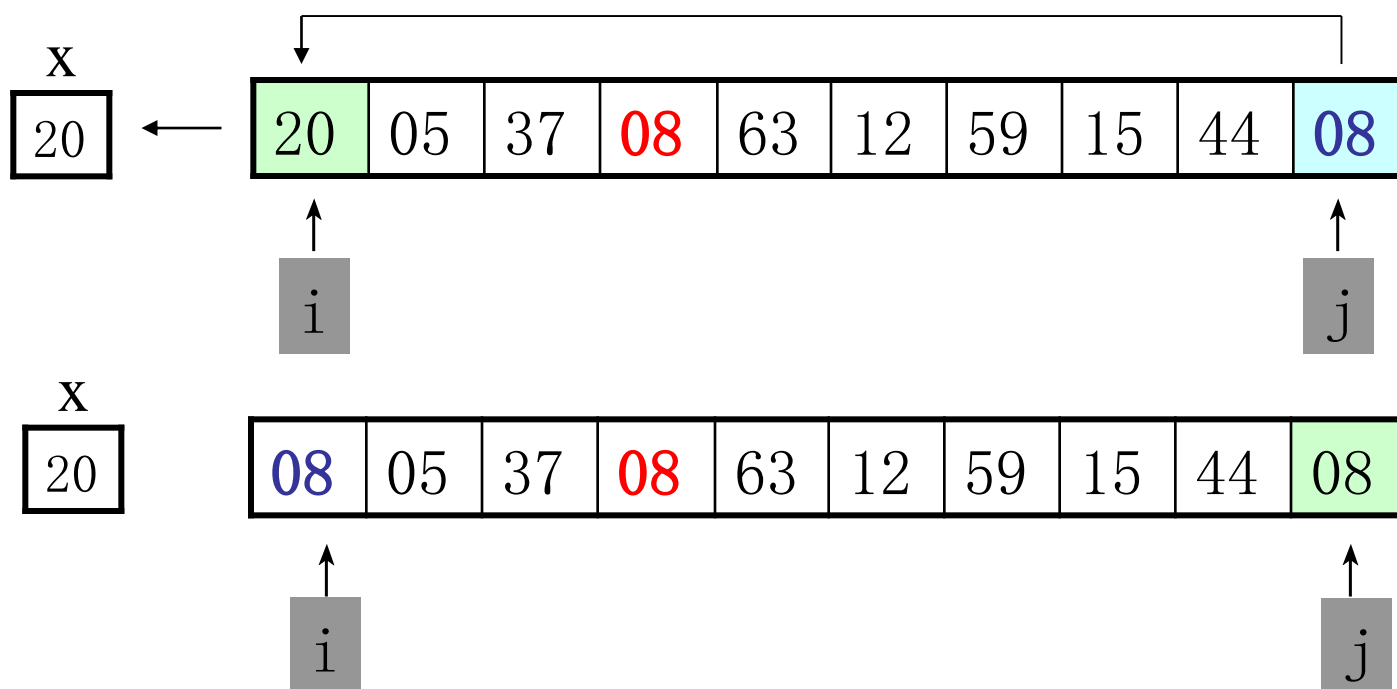
- 只需要少量中间变量作为辅助空间。  $O(1)$
- 算法是稳定的。

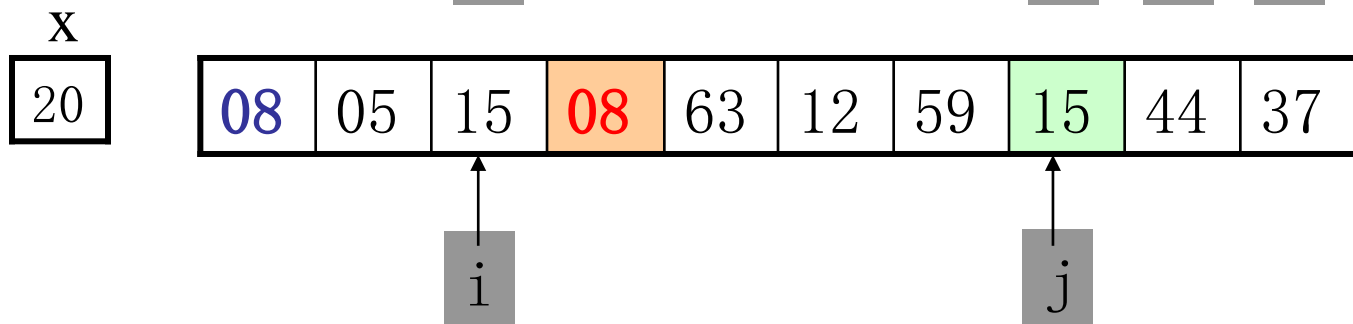
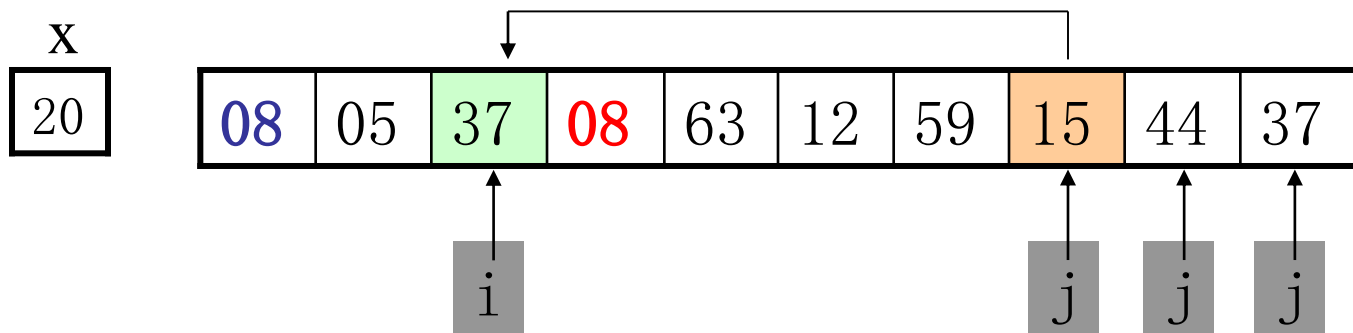
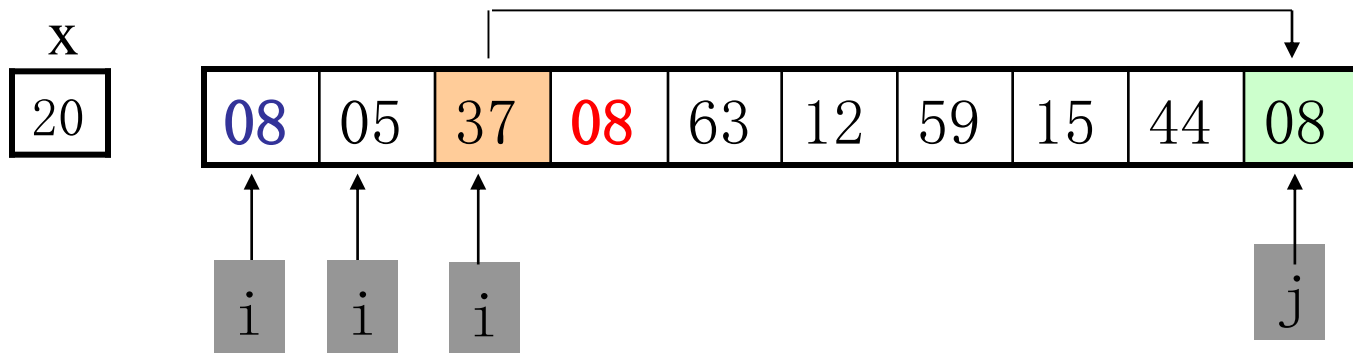
### 10.3.2 快速排序

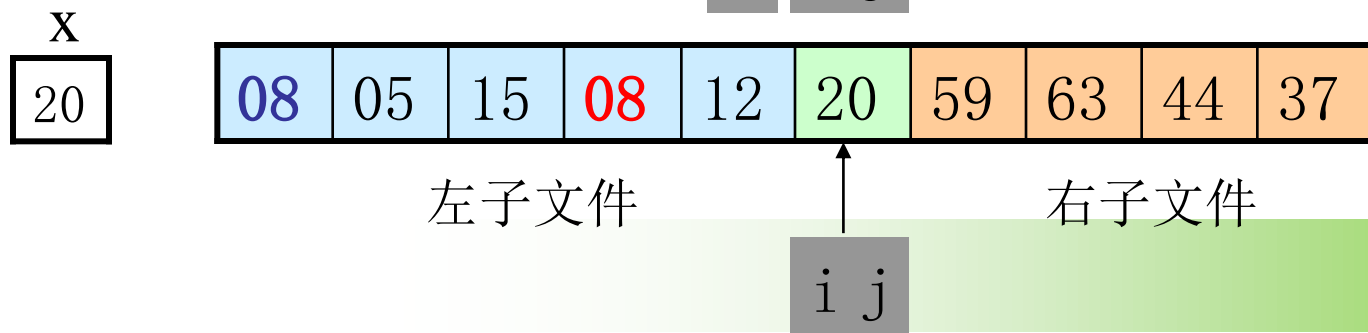
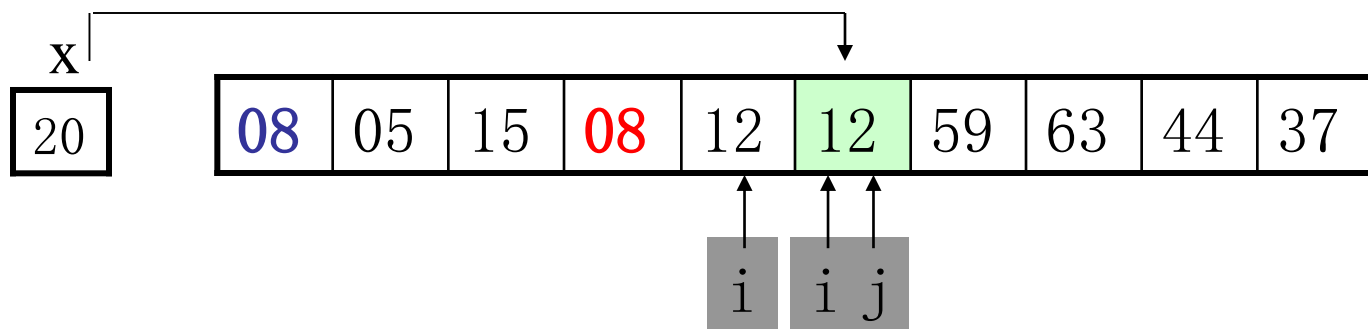
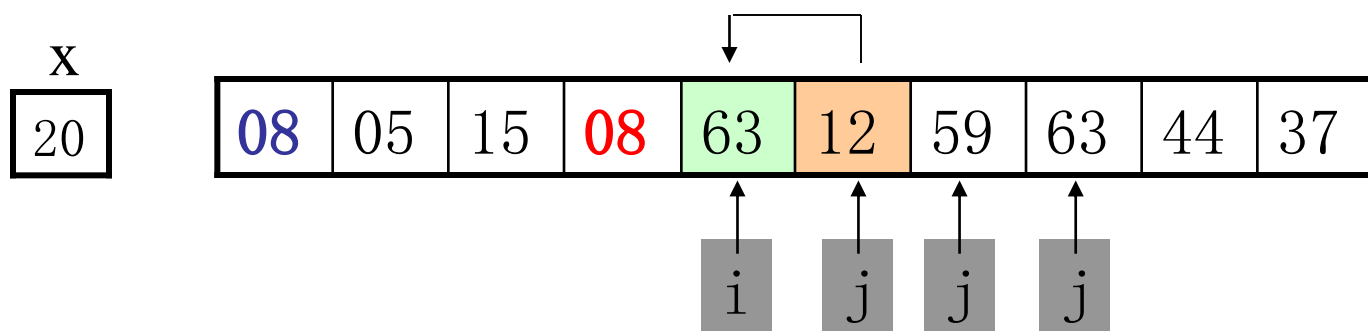
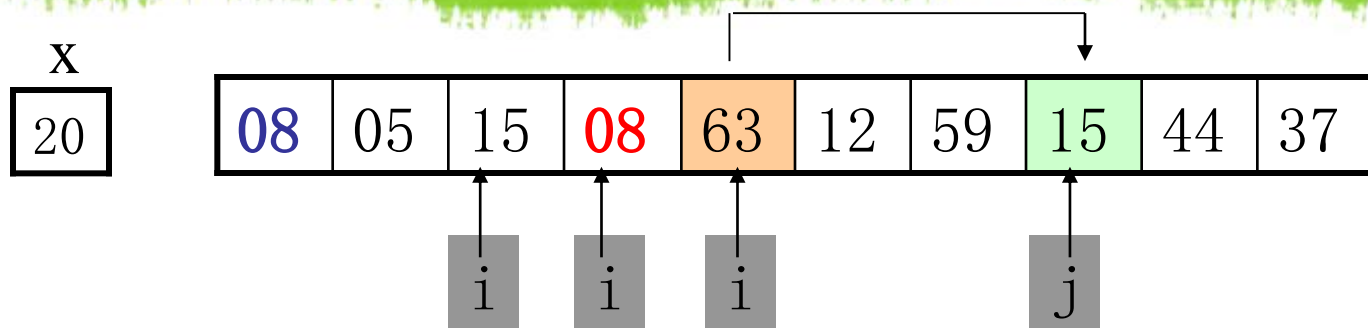
**基本思想：**首先在 $r[1..n]$ 中，确定一个 $r[i]$ ，经过比较和移动，将 $r[i]$ 放到“中间”某个位置上，使得 $r[i]$ 左边所有记录的关键字小于等于 $r[i].key$ ， $r[i]$ 右边所有记录的关键字大于等于 $r[i].key$ 。以 $r[i]$ 为界，将文件划分为左、右两个子文件。

用同样的方法分别对这两个子文件进行划分，得到4个更小的子文件。继续进行下去，使得每个子文件只有一个记录为止，便得到原文件的有序文件。

例. 给定文件 (20, 05, 37, 08, 63, 12, 59, 15, 44, 08),  
选用第1个元素20进行划分:







```

void quksort(RecType r[], int low, int high)
{
    RecType x; int i, j;
    if (low < high)                                //有两个以上记录
    {
        i = low; j = high; x = r[i];              //保存记录到变量x中
        do {                                        //此时i指示位置可用
            while (i < j && r[j].key >= x.key)
                j--;                                //j从右向左端扫描通过key不小于x.key的元素
            if (i < j)                              //i, j未相遇
            {
                r[i] = r[j]; i++;                  //此时j指示位置可用
                while (i < j && r[i].key <= x.key)
                    i++;                            //i从左向右端扫描通过key不大于x.key的元素
                if (i < j)
                {
                    r[j] = r[i]; j--;
                }
            }
        } while (i != j);                          //i, j未相遇
    }
}

```

//划分结束, i经过的是key不大于x. key的元素;  
j经过的是key不小于x. key的元素。  
i, j至少有一个指示的位置可用

```
    r[i]=x;  
    quksort(r, low, i-1);           //递归处理左子文件  
    quksort(r, i+1, high);         //递归处理右子文件  
}
```

对文件r[1..n]快速排序:

```
void quicksort(RecType r[], int n)  
{  
    quksort(r, 1, n);  
}
```



# 算法分析

- 就平均速度而言，快速排序是已知内部排序方法中最好的一种排序方法，其时间复杂度为 $O(n\log_2 n)$ 。
- 但是，在最坏情况下（基本有序时），快速排序所需的比较次数和冒泡排序的比较次数相同，其时间复杂度为 $O(n^2)$ 。
- 快速排序需要一个栈空间来实现递归。若每次划分均能将文件均匀分割为两部分，则栈的最大深度为 $\lfloor \log_2 n \rfloor + 1$ ，所需栈空间为 $O(\log_2 n)$ ，即空间复杂度 $S(n) = O(\log_2 n)$
- 快速排序是不稳定的。

## 练习：

用快速排序法描述出下面序列的排序过程：

关键字序列：

**(52, 49, 80, 36, 14, 75, 58, 97, 23, 61)**

经第1趟快速排序之后为：

(23, 49, 14, 36) 52 (75, 58, 97, 80, 61)

经第2趟快速排序之后为：

(14) 23 (49, 36) 52 (61, 58) 75 (80, 97)

经第3趟快速排序之后为：

(14, 23, 36, 49, 52, 58, 61, 75, 80, 97)

**思考：**试对以下序列( 90, 45, 39, 54, 68, 87, 76 ) 进行快速排序，是否发现什么特殊情况？

由于枢轴记录的关键字“90”大于其它所有记录的关键字，致使一次划分之后得到的子序列(1)的长度为0，由此可以想像，快速排序不适于对原本有序或基本有序的记录序列进行排序。

**注：**为避免出现枢轴记录关键字为“最大”或“最小”的情况，通常进行的快速排序采用“三者取中”的改进方案，即以  $R[s]$ 、 $R[t]$  和  $R[(s+t)/2]$  三者中关键字介于中值者为枢轴。只要将它和  $R[s]$  互换，一次划分的算法仍不变。

## 10.4 选择排序

### 1. 简单选择(选择排序)

算法思想: 设待排序的文件为  $(r[1], r[2], \dots, r[n])$ , 关键字为  $(r[1].key, r[2].key, \dots, r[n].key)$ ,

**第1趟(遍):** 在  $(r[1], r[2], \dots, r[n])$  中, 选出关键字最小的记录  $r[\min].key$ , 若  $\min \neq 1$ , 则交换  $r[1]$  和  $r[\min]$ ;

需要进行  $n-1$  次比较。

**第2趟(遍):** 在  $n-1$  个记录  $(r[2], \dots, r[n])$  中, 选出关键字最小的记录  $r[\min].key$ , 若  $\min \neq 2$ , 则交换  $r[2]$  和  $r[\min]$ ;

需要进行  $n-2$  次比较。

.....

**第  $n-1$  趟(遍):** 在最后的2个记录  $(r[n-1], r[n])$  中, 选出关键字最小的记录  $r[\min].key$ , 若  $\min \neq n-1$ , 则交换  $r[n-1]$  和  $r[\min]$ ; 需要进行1次比较。



简单选择排序算法：（对数组 $r[1..n]$ 中的记录作简单选择排序）

```
void SelectSort(RecType r[], int n)
{
    int i, j, min;
    RecType x;                                //交换记录的中间变量
    for (i=1; i<n; i++)                       //共n-1趟(遍)
    {
        min=i;                                //r[i]为最小记录r[min]
        for (j=i+1; j<=n; j++)
            if (r[j].key<r[min].key)
                min=j;                        //修改min
        if (min!=i)                            //若r[min]不是r[i]
        {
            x=r[min];                          //交换r[min]和r[i]
            r[min]=r[i];
            r[i]=x;
        }
    }
}
```

## 算法分析:

(1) 比较次数, 在任何情况下, 均为

$$\begin{aligned}\sum_{i=1}^{n-1} (n-i) &= (n-1) + (n-2) + \dots + 1 \\ &= n(n-1)/2 = O(n^2)\end{aligned}$$

(2) 交换记录的次数

在最好情况下, 原 $n$ 个记录递增有序:

不移动记录。

在最坏情况下, 每次都要交换数据 (不是递减有序)

共交换记录 $n-1$ 对, 移动记录数 $3(n-1)$ 次。

故, 时间复杂度为 $O(n^2)$ 。

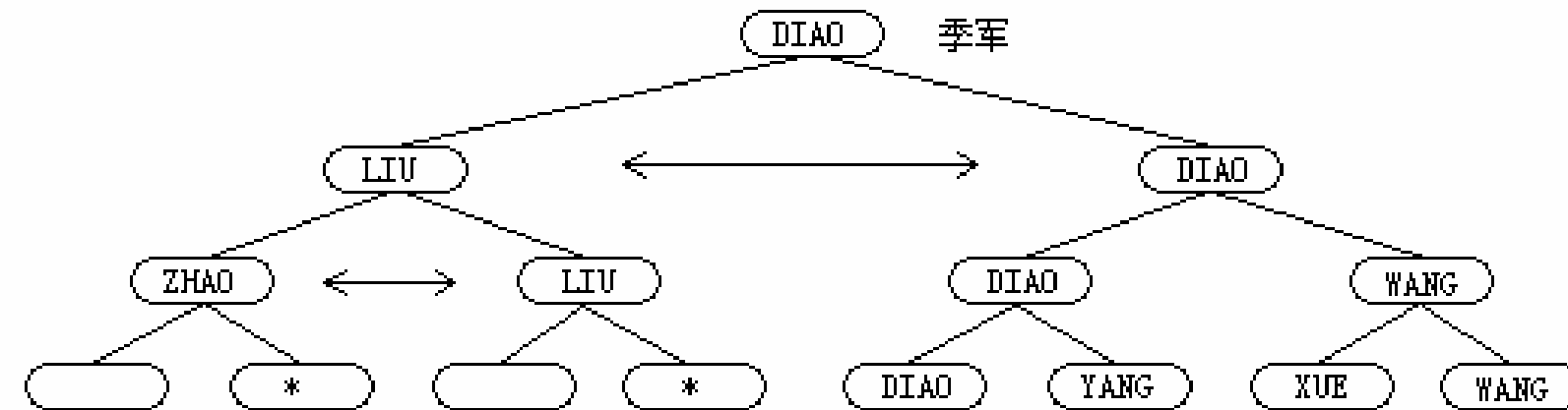
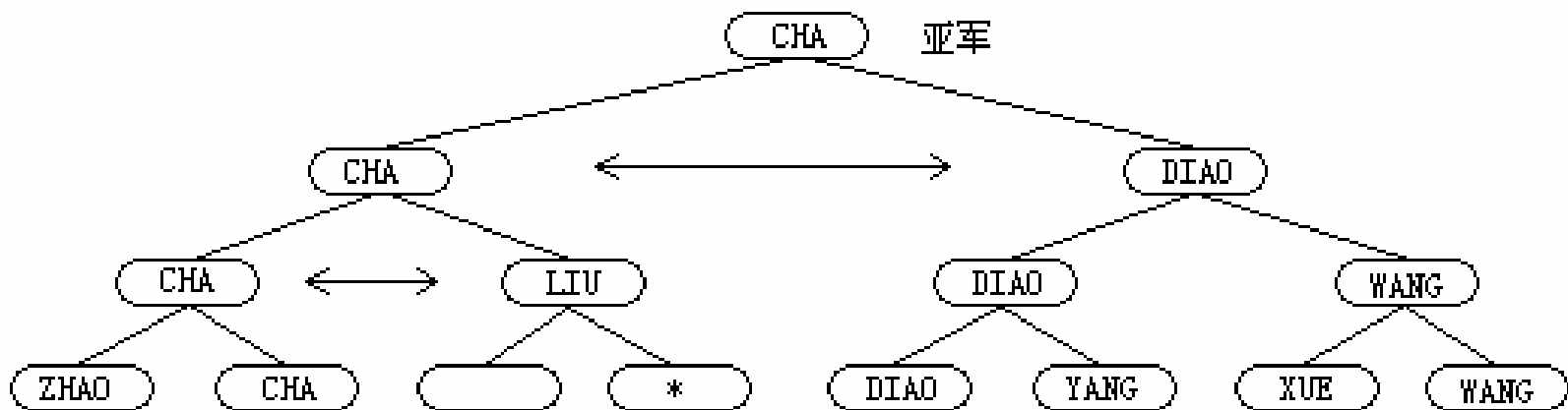
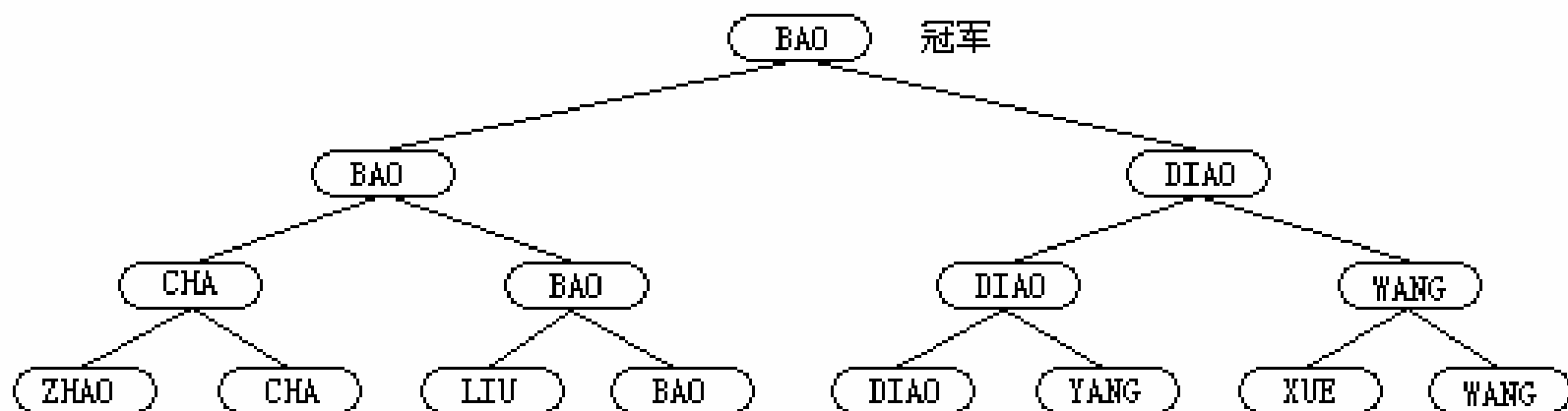
(3) 只需少量中间变量作为辅助空间。  $O(1)$

(4) 算法是不稳定的。

思考：从 **$n$** 个元素中找出最小值，至少进行 **$n-1$** 次比较，然而，继续从余下的 **$n-1$** 个元素中找出次小值是否一定要 **$n-2$** 次比较呢？

若能利用前 **$n-1$** 次比较所得信息，则可减少以后各趟选择排序中所用的比较次数。实际上，体育比赛中的锦标赛就是一种选择排序。





## 2. 树形选择排序

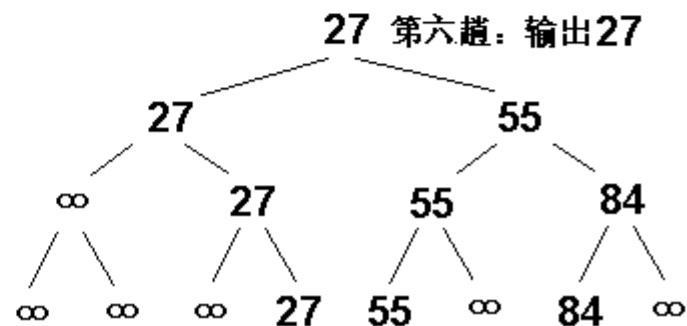
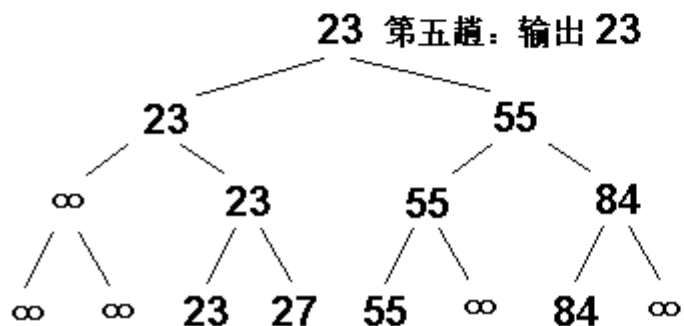
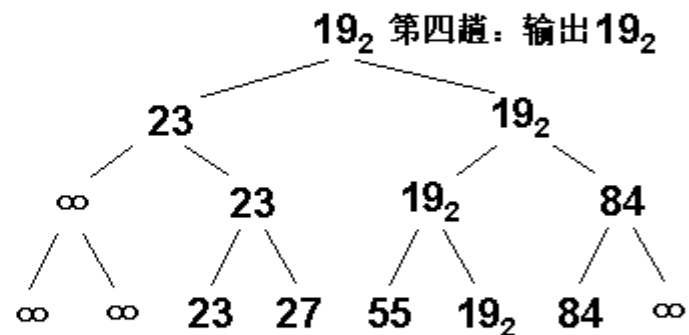
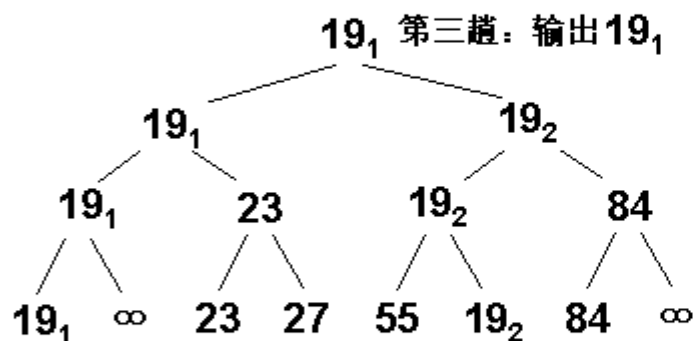
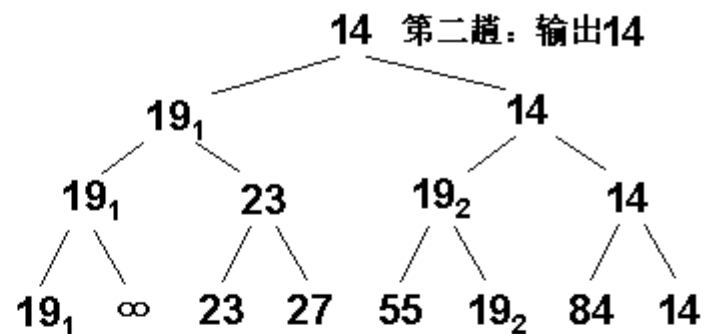
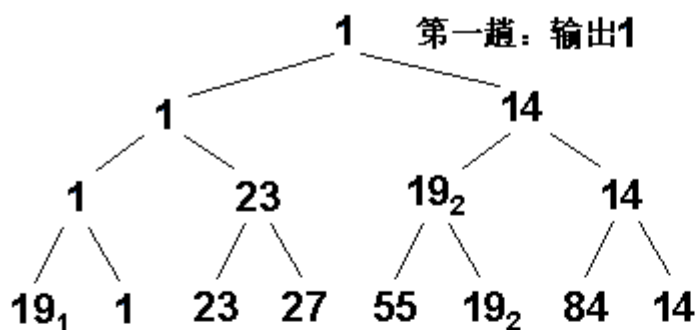
树形选择排序又称锦标赛排序（**Tournament Sort**），是一种按照锦标赛的思想进行选择排序的方法。可用一棵有 **$n$** 个叶子结点的完全二叉树表示。

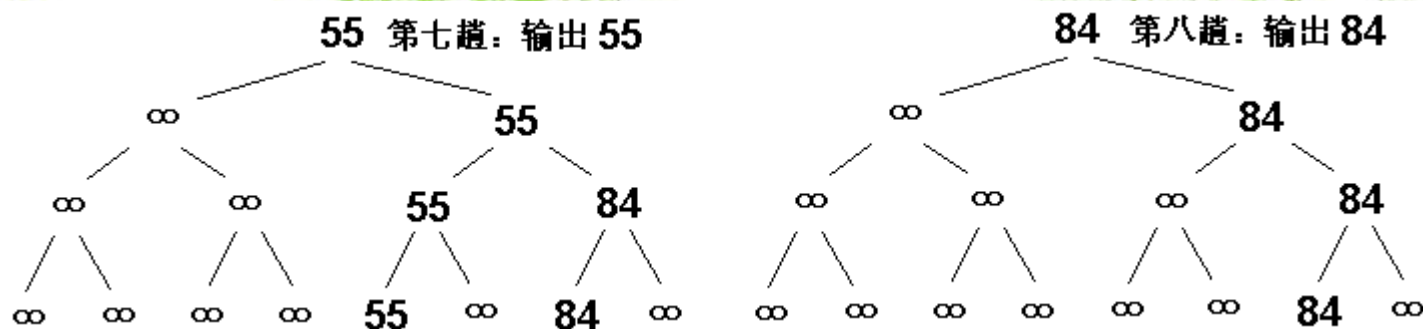
基本思想为：

- ① 首先对 **$n$** 个记录的关键字进行两两比较，然后在其中 **$n/2$** 个较小者之间再进行两两比较，如此重复，直到选出最小关键字的记录为止（树根）。
- ② 将最小关键字输出，且将其原来位置改为极大数，与此位置相关部分重新（向树根方向）进行比较，选出次小关键字，保留结果。
- ③ 如此下去，直至全部排序完成。

# 树形排序实例

待排序记录的关键字为： **$19_1, 1, 23, 27, 55, 19_2, 84, 14$**





最终结果为: **1,14,19<sub>1</sub>,19<sub>2</sub>,23,27,55, 84**

该方法比直接选择排序速度上有很大提高，其缺点有：

- (1) 需要另开储存空间保存排序结果；
- (2)  $n$ 个待排序关键字，需要额外的  $(n-1)$  个内部结点（包括根结点），增加了内存开销。
- (3) 将最小关键字改为极大数，再与兄弟结点比较属于多余。

故：树形选择排序一般不是用来排序而是用来证明某些问题。为了弥补以上缺点，威洛姆斯在1964年提出了另一种形式的选择排序—堆排序。

# 树形选择排序的算法复杂度

空间复杂度：需要额外的（ **$n-1$** ）个辅助空间，即  
 **$S(n)=O(n)$** ;

时间复杂度：

由于含有 **$n$** 个叶子结点的完全二叉树的深度为  $\lceil \log_2 n \rceil + 1$ ，则在树形选择排序中，第一次选最小值时比较 **$n-1$** 次，以后每选一个次小值要比较  $\log_2 n$  次，故总的比较次数为  $(n-1) + (n-1)\log_2 n = n\log_2 n$ 。结点的移动次数不超过比较次数。故总的开销为  **$O(n\log_2 n)$** 。

即：树形选择排序的**平均时间复杂度为**  
 **$O(n\log_2 n)$**

树形选择排序是**不稳定**的。

### 3.堆排序 (Heap Sort)

堆的定义：n个元素的序列 $\{k_1, k_2, \dots, k_n\}$  当且仅当满足下关系时，称之为堆。

$$\left\{ \begin{array}{l} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{array} \right. \quad \text{或} \quad \left\{ \begin{array}{l} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{array} \right.$$

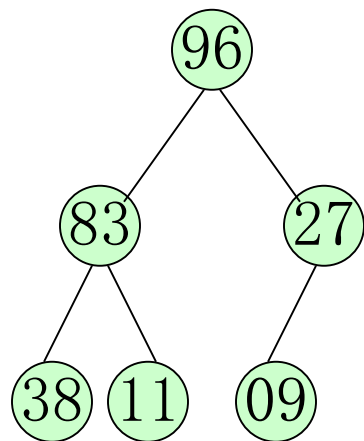
$(i=1, 2, \dots, \lfloor n/2 \rfloor)$

其中： 前面一种称为小顶堆；  
后面一种称为大顶堆。

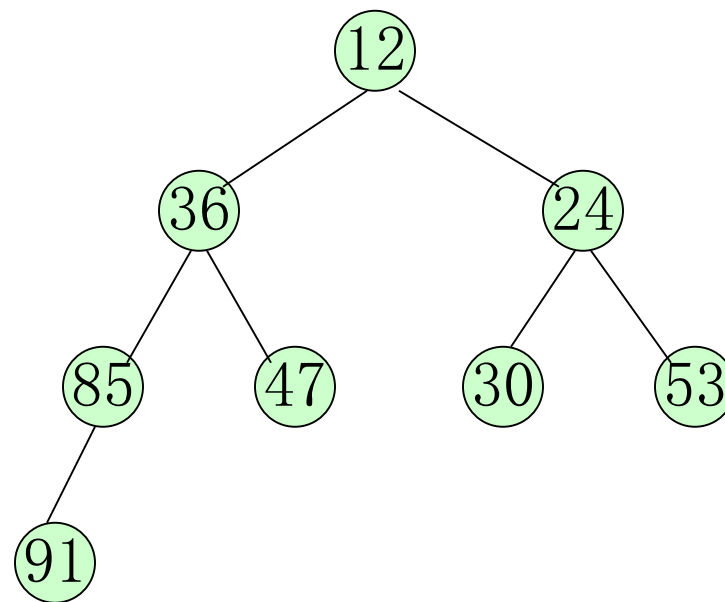
$n$ 个元素的序列 $\{k_1, k_2, \dots, k_n\}$ 可看成是一个结点  
个数为 $n$ 的完全二叉树，若其为大顶堆，则 $k_1$ 最大；  
若其为小顶堆，则 $k_1$ 最小。

例： 序列1：  $\{96, 83, 27, 38, 11, 09\}$

序列2：  $\{12, 36, 24, 85, 47, 30, 53, 91\}$



序列1的二叉树(大顶堆)



序列2的二叉树(小顶堆)

通常， $n$ 个元素的序列  $\{k_1, k_2, \dots, k_n\}$  不符合堆的定义，所以，面临的第一个问题：

**问题1：**如何将序列  $\{k_1, k_2, \dots, k_n\}$  处理成（大顶）堆（初始化）？

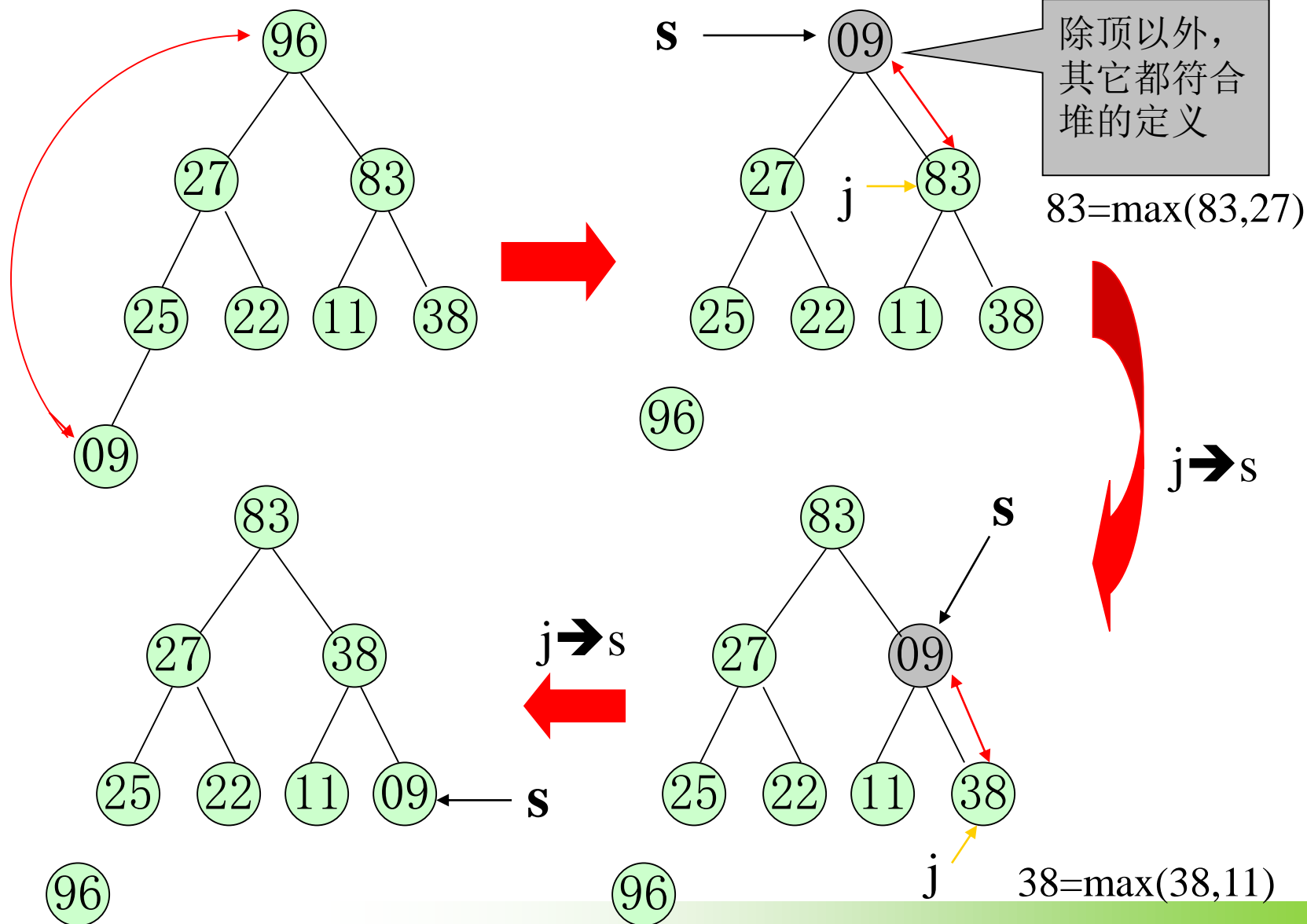
问题1一旦解决，得到规模为 $n$ 的堆，则 $k_1$ 最大，将 $k_1$ 与 $k_n$ 互换，则最大的数已放置到最后，同时，剩下的序列  $\{k_1, k_2, \dots, k_{n-1}\}$  不是堆，如何将其重新处理成规模为 $n-1$ 的堆，求取第二大的数据，以此类推，堆的规模逐步减小，直到求出第 $n-1$ 大的数据，完成递增排序。所以，面临的第二个问题是：



**问题2：**如何在堆顶元素被替换后，调整剩余元素成为一个新的堆。

**提示：**根据上述过程描述，借助大顶堆可实现序列的递增排序；借助小顶堆可实现序列的递减排序。

问题2方法: 某序列的堆形式: {96, 27, 83, 25, 22, 11, 38, 09}



## 堆调整算法：

```
typedef   SqList      HeapType;
```

```
//采用顺序表存储表示
```

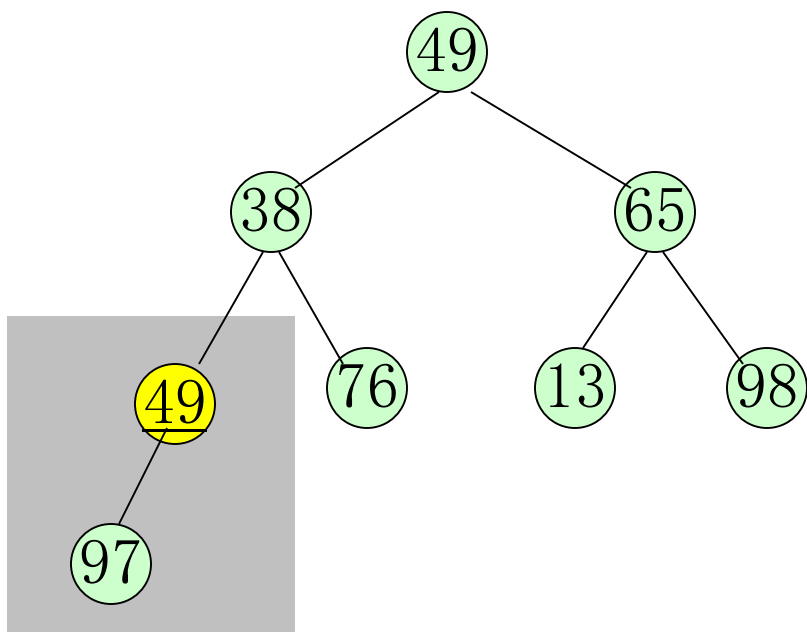
```
void  HeapAdjust (HeapType  &H,
```

```
                int  s, int  m)
```

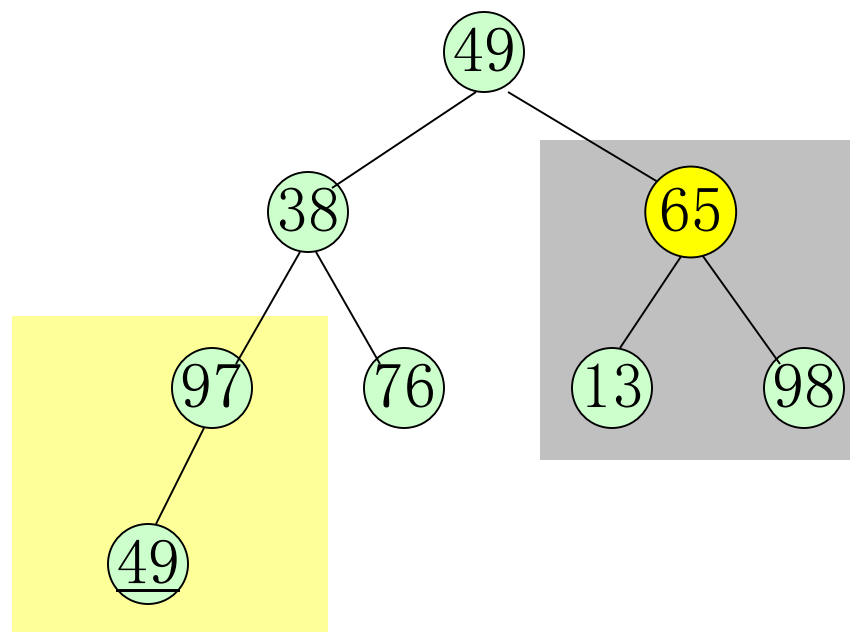
```
//已知H.r[s...m]中记录的关键字除H.r[s].key之外均满足堆的定义，本函数调整H.r[s]的关键字，使H.r[s...m]成大顶堆。
```

[illegible]

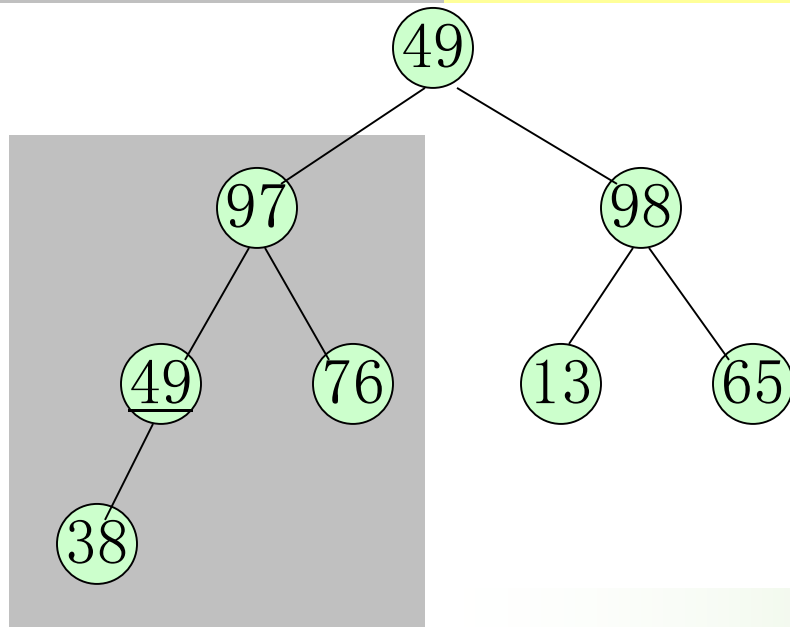
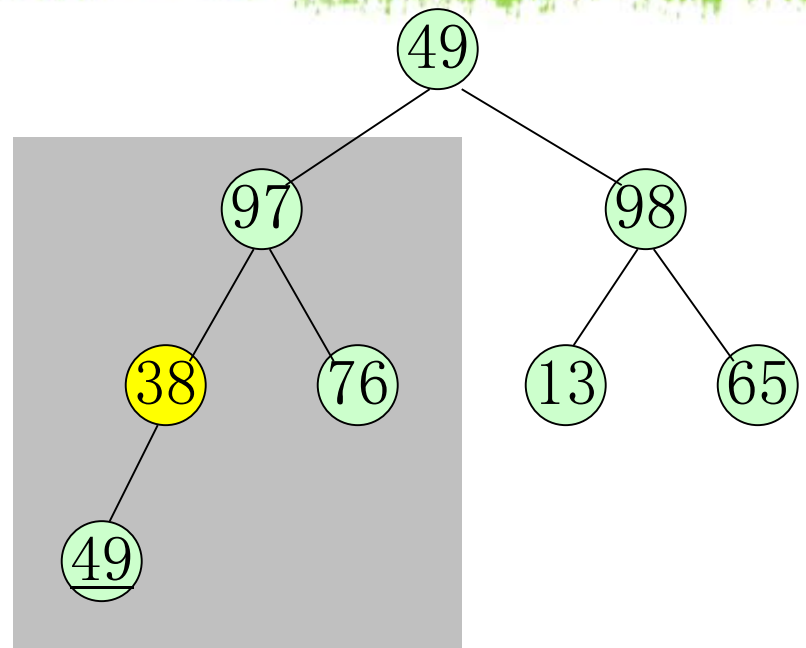
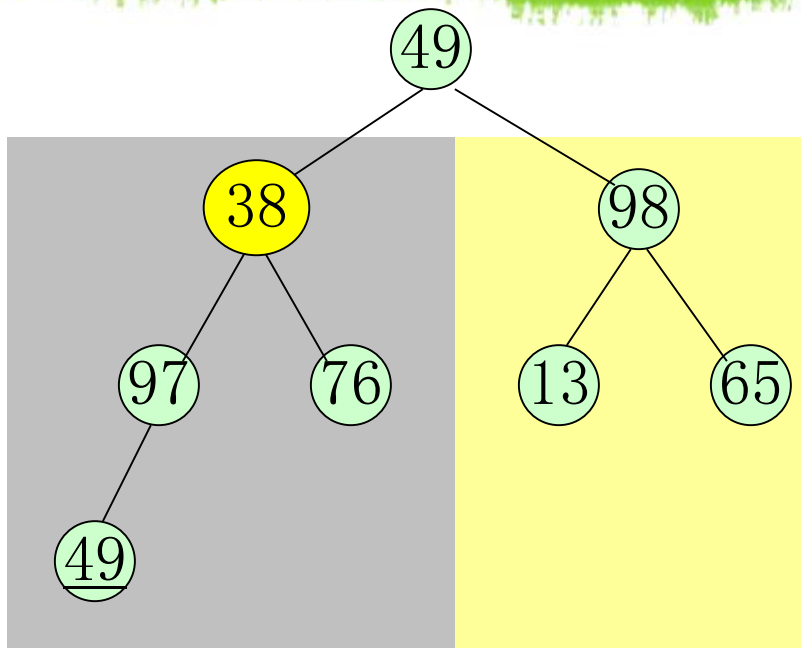
**问题1方法：**建立序列：{49, 38, 65, 49, 76, 13, 98, 97} 的初始大顶堆。



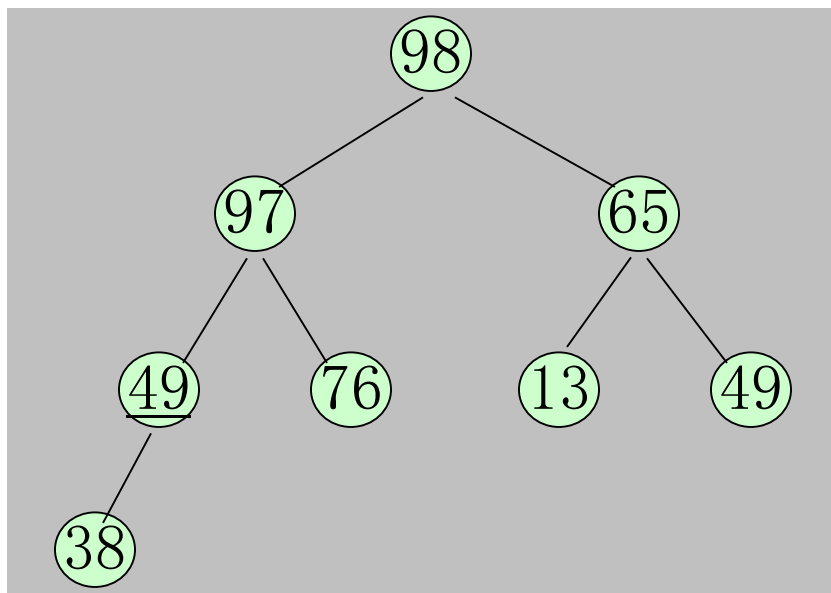
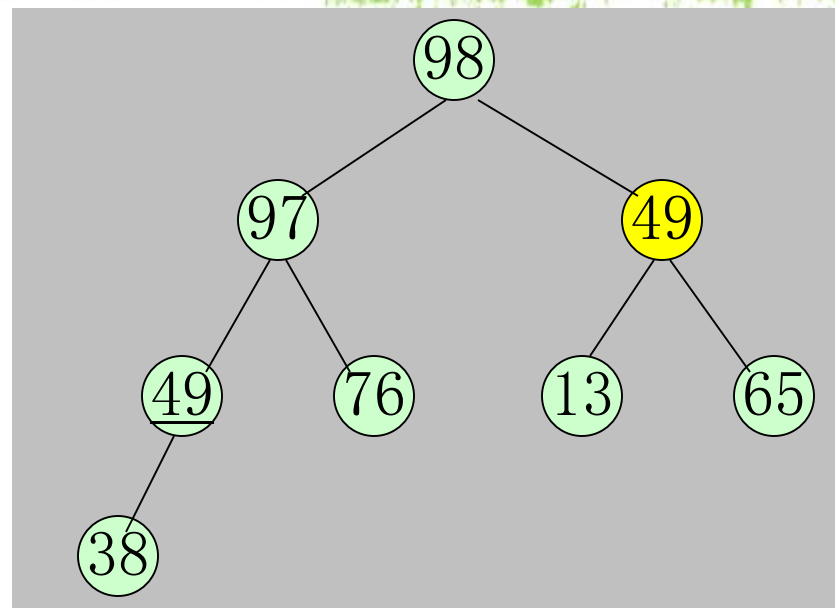
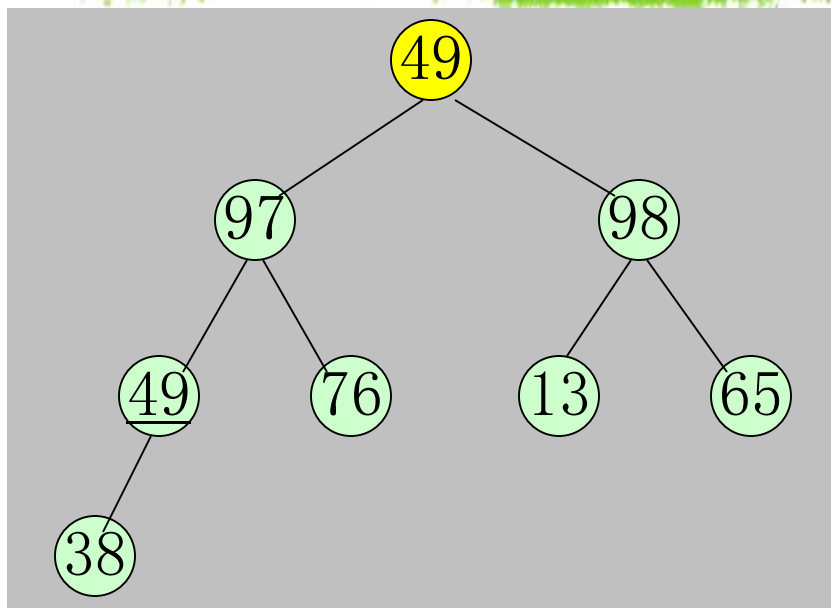
1、对以最后一个结点（序号 $n$ ）的双亲结点（序号 $i=\lfloor n/2 \rfloor$ ）为根的二叉树，进行堆调整。



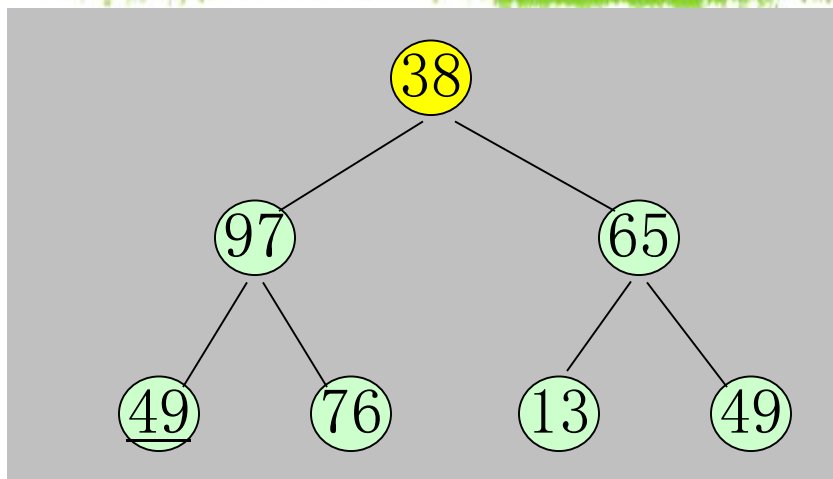
2、对以序号 $i=i-1$ 的结点为双亲结点为根的二叉树，进行堆调整。



3、对以序号 $i=i-1$ 的结点为双亲结点为根的二叉树，进行堆调整。

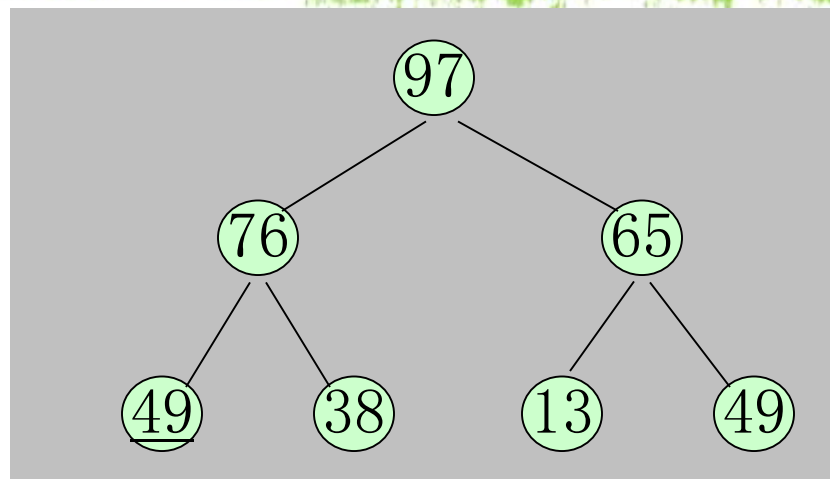


4、对以序号 $i=i-1$ 的结点为双亲结点为根的二叉树，进行堆调整。此时 $i$ 已等于1，调整后，初始大顶堆建成。



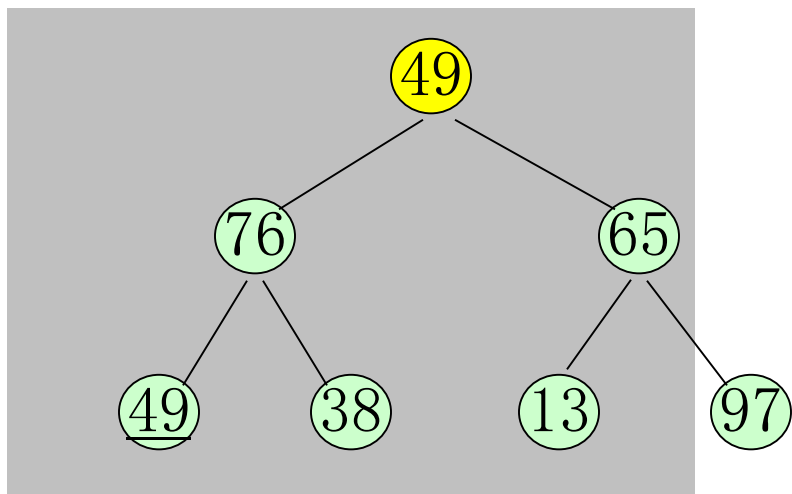
98

选择



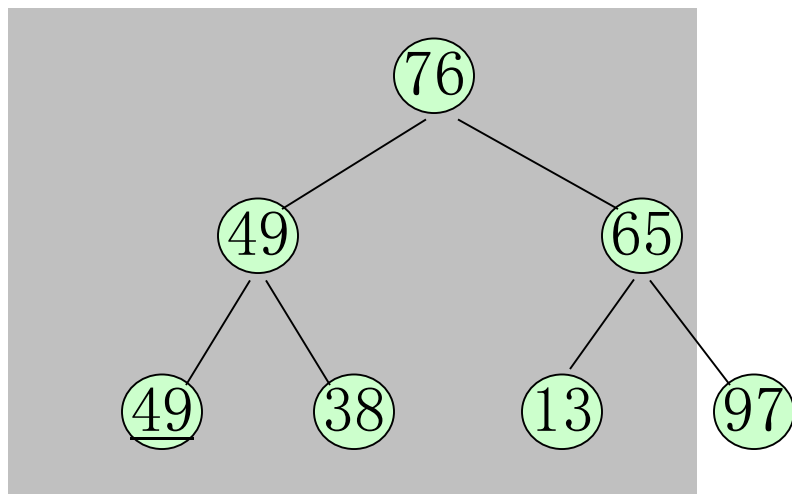
98

调整



98

较小范围选择



98

较小范围调整



堆排序

$\lfloor n/2 \rfloor \rightarrow i$

$i > 0$

假

真

调整以*i*为  
根的二叉树

$i--$

初始化堆

选择、调整*n*-1次

$n \rightarrow i$

$i > 1$

假

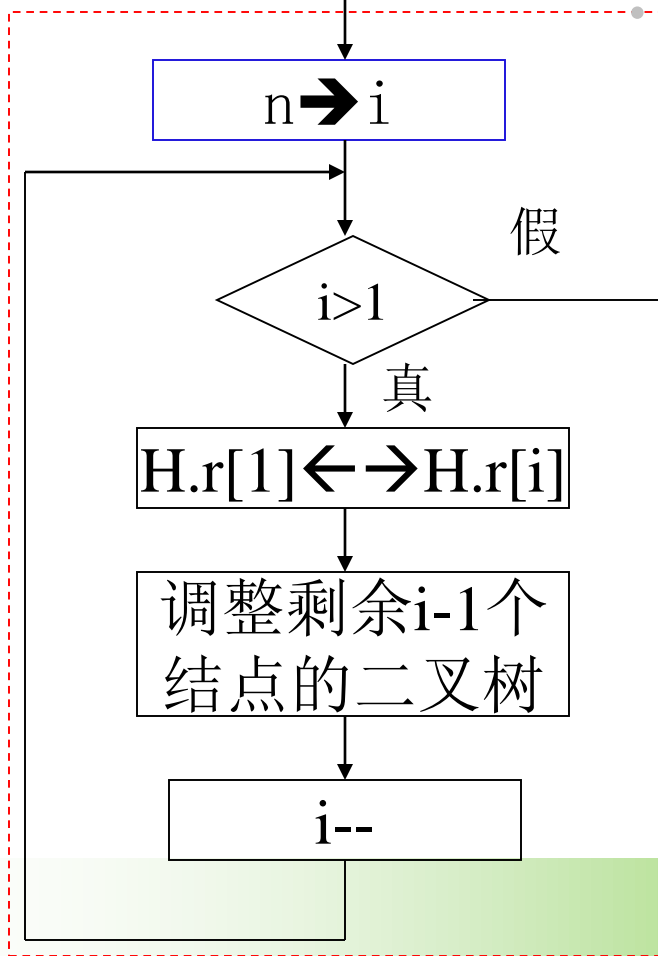
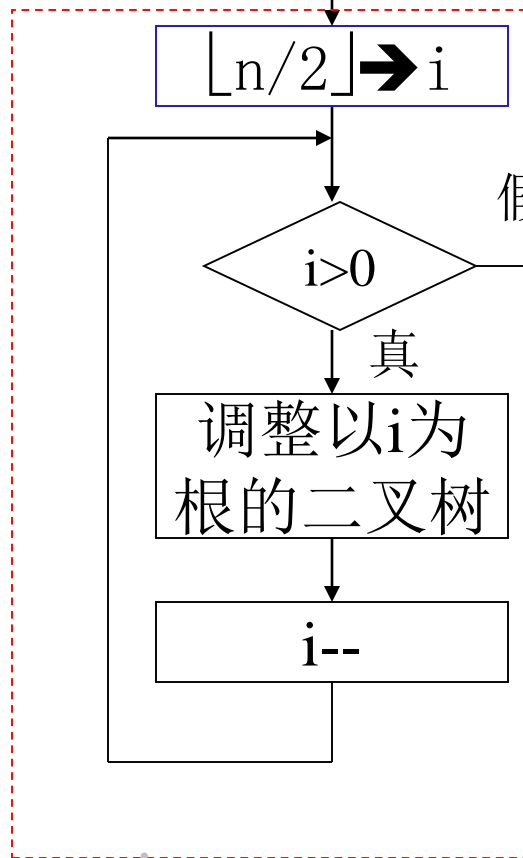
真

$H.r[1] \leftrightarrow H.r[i]$

调整剩余*i*-1个  
结点的二叉树

$i--$

结束



## 算法分析与评价:

➤对深度为k的堆, 调整算法进行的关键字比较次数至多为 $2(k-1)$ 次,

➤建立n个元素的初始堆, 调用HeapAdjust算法  $\lfloor n/2 \rfloor$ 次, 总共进行的关键字比较次数不超过 $4n$ 次。

n个结点的完全二叉树深度为 $h = \lfloor \log_2 n \rfloor + 1$

需要调整的层为 $h-1$ 层至1层, 以第 $i$ 层某结点为根的二叉树深度对应为 $h-i+1$ , 第 $i$ 层结点最多为 $2^{i-1}$ 个, 故调整时比较关键字最多为:

$$\sum_{i=h-1}^1 2^{i-1} * 2 * (h-i+1-1) = \sum_{i=h-1}^1 2^i * (h-i)$$

令  $j = h-i$ , 当  $i=h-1$  时  $j=1$  当  $i=1$  时  $j=h-1$

$$\sum_{j=1}^{h-1} 2^{h-j} * j = 2^{h-1} * 1 + 2^{h-2} * 2 + \dots + 2^1 * (h-1) = 2^{h+1} - 2h - 2$$

$$< 2^{h+1} = 2^{\lfloor \log_2 n \rfloor + 2} < 4 * 2^{\log_2 n} = 4n$$

## 算法分析与评价(续)：

➤  $n$ 个结点的完全二叉树深度为 $\lfloor \log_2 n \rfloor + 1$ ，选择调整过程 $n-1$ 次，总共比较次数至多为：

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

➤ 最坏情况下，算法的时间复杂度为 $O(4n + n \log n) \rightarrow O(n \log n)$ ；

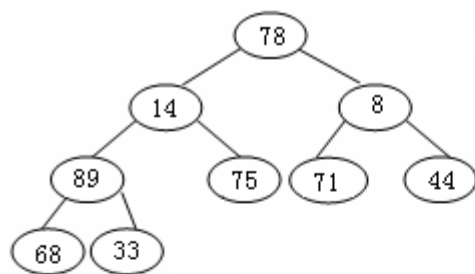
➤ 仅需要一个记录大小供交换用的辅助存储空间；

➤ 堆排序是**不稳定排序**。

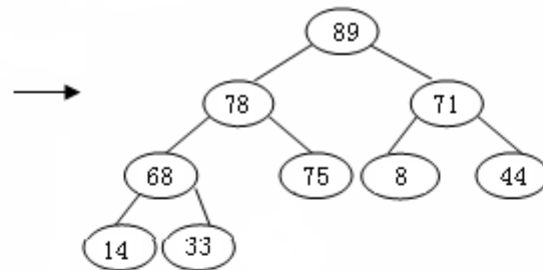
➤ 堆排序对记录较少的文件不提倡，对较大的文件很有效；

# 堆排序实例模拟

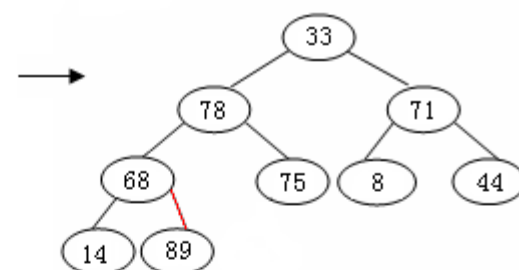
待排序记录的关键字为：（**78**，**14**，**8**，**89**，**75**，**71**，**44**，**68**，**33**），利用“大顶堆”排序法进行排序。



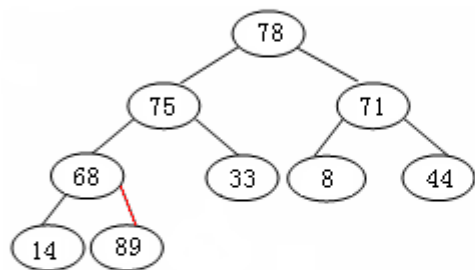
原始数据的完全二叉树



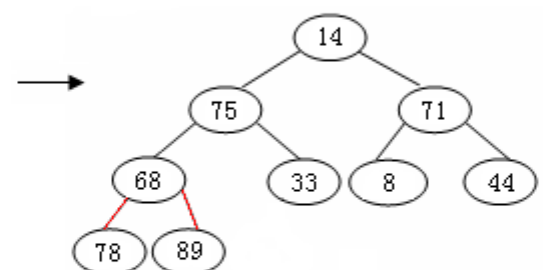
大堆



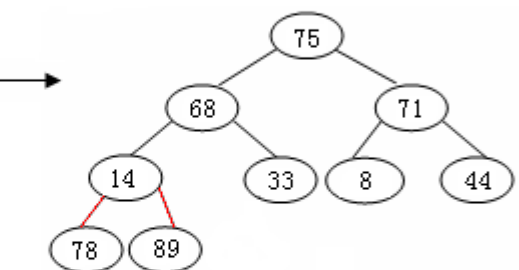
输出结点89



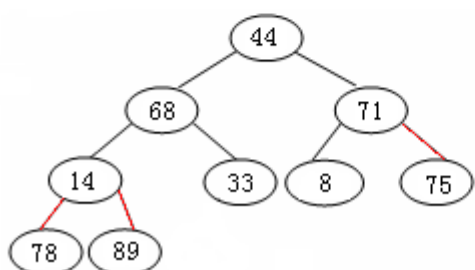
重新调整成大堆



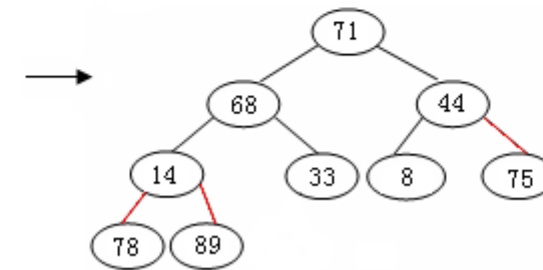
输出结点78



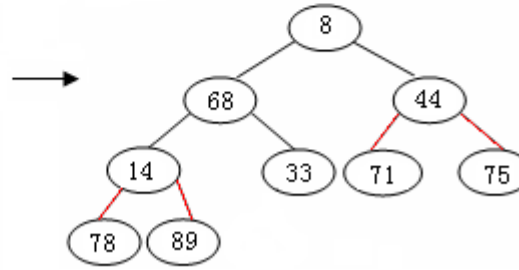
重新调整成大堆



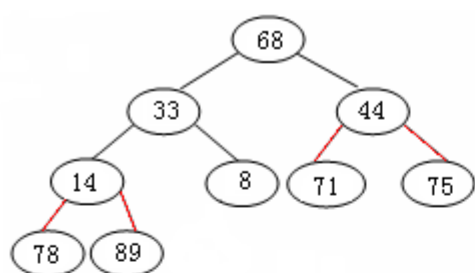
输出结点75



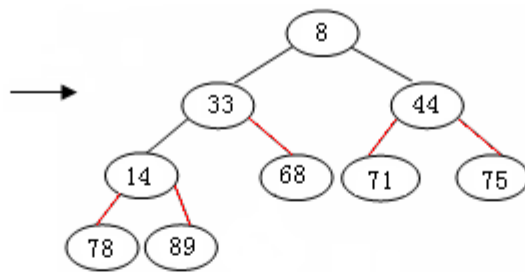
重新调整成大堆



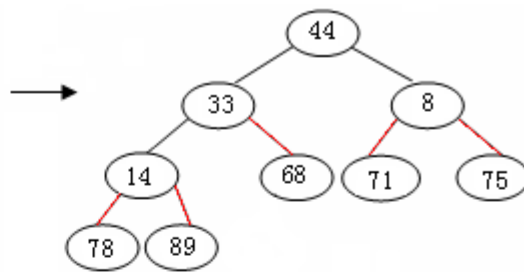
输出结点71



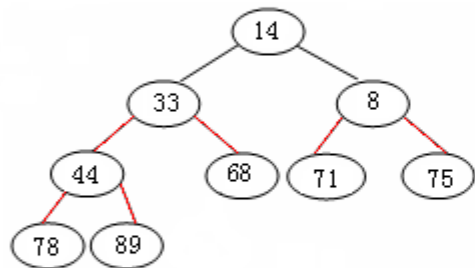
重新调整成大堆



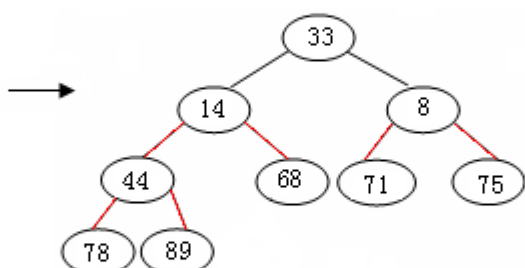
输出结点68



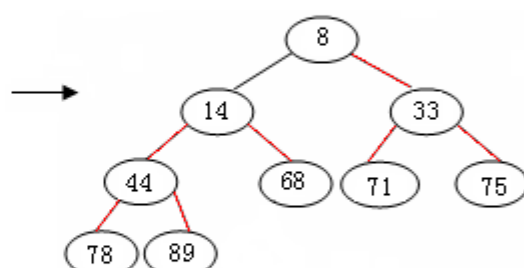
重新调整成大堆



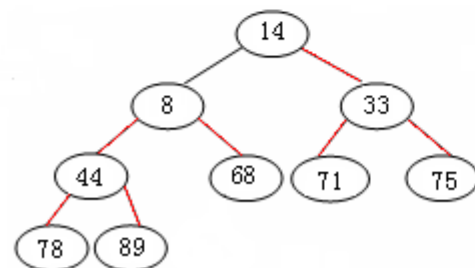
输出结点44



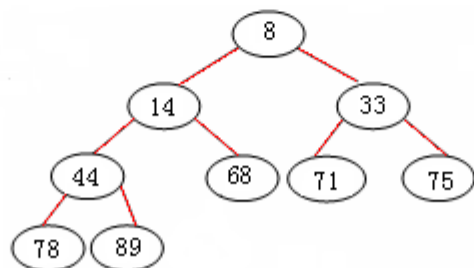
重新调整成大堆



输出结点33



重新调整成大堆



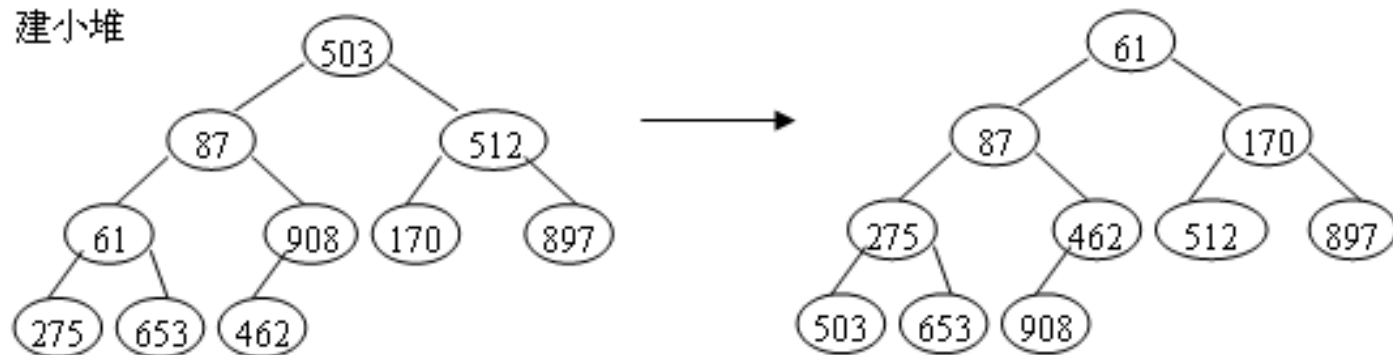
输出结点14

从小到大的序列为：（8， 14， 33， 44， 68， 71， 75， 78， 89）

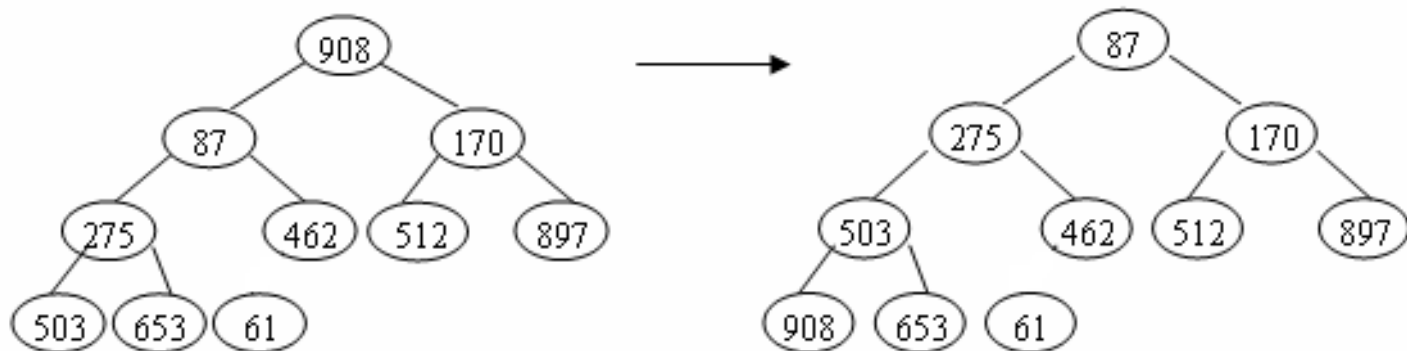
## 练习:

已知待排序的序列为 (**503**, **87**, **512**, **61**, **908**, **170**, **897**, **275**, **653**, **462**)。 **(1)**建立一个堆 (画出第一步和最后堆的结果图), 希望先输出最小值。 **(2)**输出最小值后, 如何得到次小值。 (并画出相应结果图)

建小堆



求次小值



## 10.5 归并排序

基本思想：把 $k$  ( $k \geq 2$ ) 个有序子文件合并在一起，形成一个新的有序文件。同时归并 $k$ 个有序子文件的排序过程称为 $k$ -路归并排序。

**2-路归并排序：** 归并2个有序子文件的排序。

例. 将有序文件A和B归并为有序文件C。

$$A = (2, 10, 15, 18, 21, 30)$$

$$B = (5, 20, 35, 40)$$

按从小至大的次序从A或B中依次取出

$$2, 5, 10, 15, \dots, 40,$$

顺序归并到C中, 得:

$$C = (2, 5, 10, 15, 18, 20, 21, 30, 35, 40)$$

一般地，2-路归并过程为：

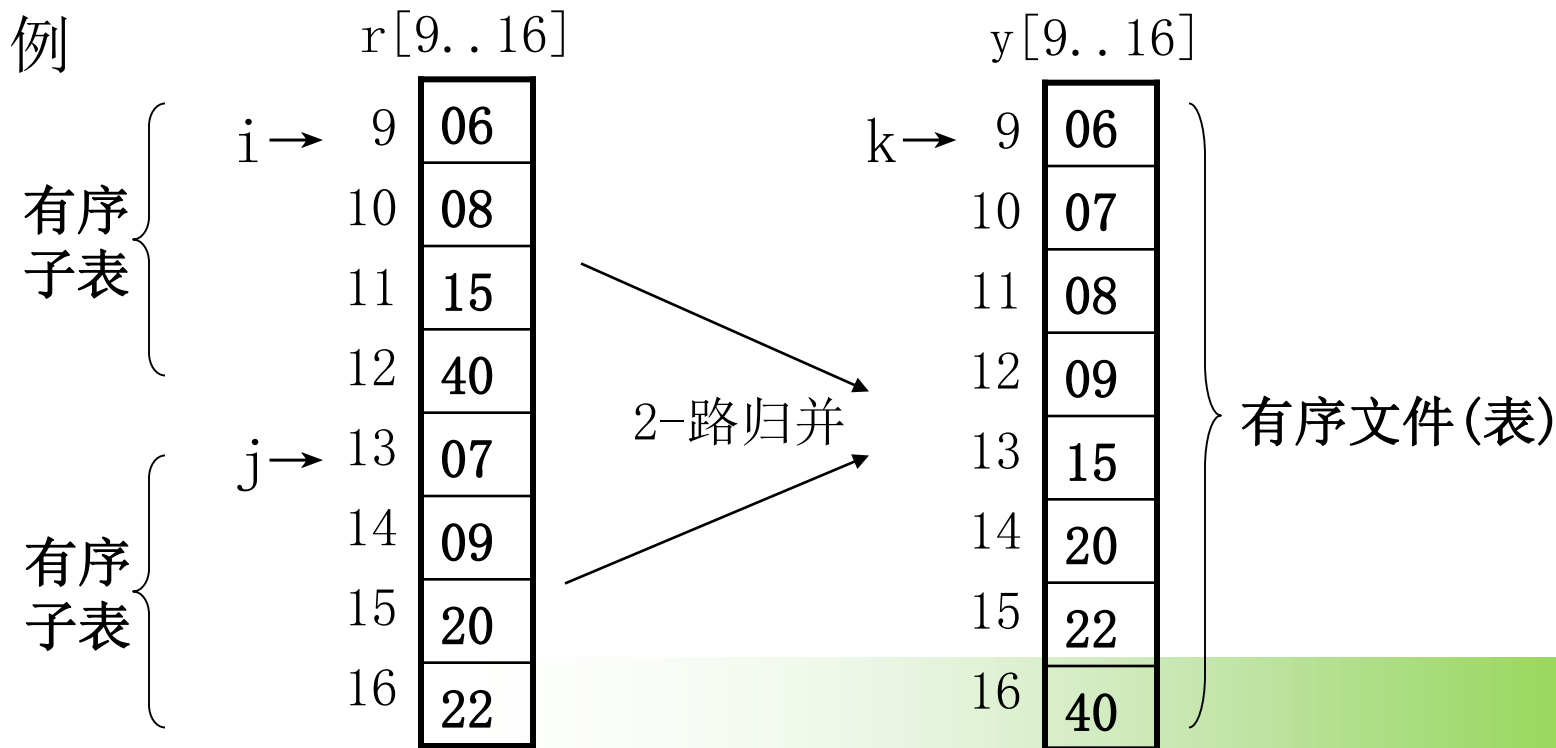
假定文件 $r[\text{low}..\text{high}]$ 中的相邻子文件(子表)

$(r[\text{low}], r[\text{low}+1], \dots, r[\text{mid}])$  和  $(r[\text{mid}+1], \dots, r[\text{high}])$

为有序子文件, 其中:  $\text{low} \leq \text{mid} < \text{high}$  。

将这两个相邻有序子文件归并为有序文件 $y[\text{low}..\text{high}]$ , 即:

$(y[\text{low}], y[\text{low}+1], \dots, y[\text{high}])$





将两个有序子文件归并为有一个有序文件的算法

```
void merge(r, y, low, mid, high)
```

```
RecType r[], y[]; int low, mid, high;
```

```
{ int k=i=low, j=mid+1;
```

```
while (i<=mid && j<=high)
```

```
{ if (r[i].key<=r[j].key)
```

```
    { y[k]=r[i];          //归并前一个子文件的记录  
      i++; } }
```

```
else
```

```
{ y[k]=r[j];          //归并后一个子文件的记录  
  j++; }
```

```
k++;
```

```
}
```

```
while (j<=high)           //归并后一个子文件余下的记录
{ y[k]=r[j];
  j++;  k++;
}
while (i<=mid)            //归并前一个子文件余下的记录
{ y[k]=r[i];
  i++;  k++;
}
} // merge
```

## 2-路归并排序

假定文件( $r[1], r[2], \dots, r[n]$ )中记录是随机排列的, 进行2-路归并排序, 首先把它划分为长度均为1的 $n$ 个有序子文件, 然后对它们逐步进行2-路归并排序。其步骤如下:

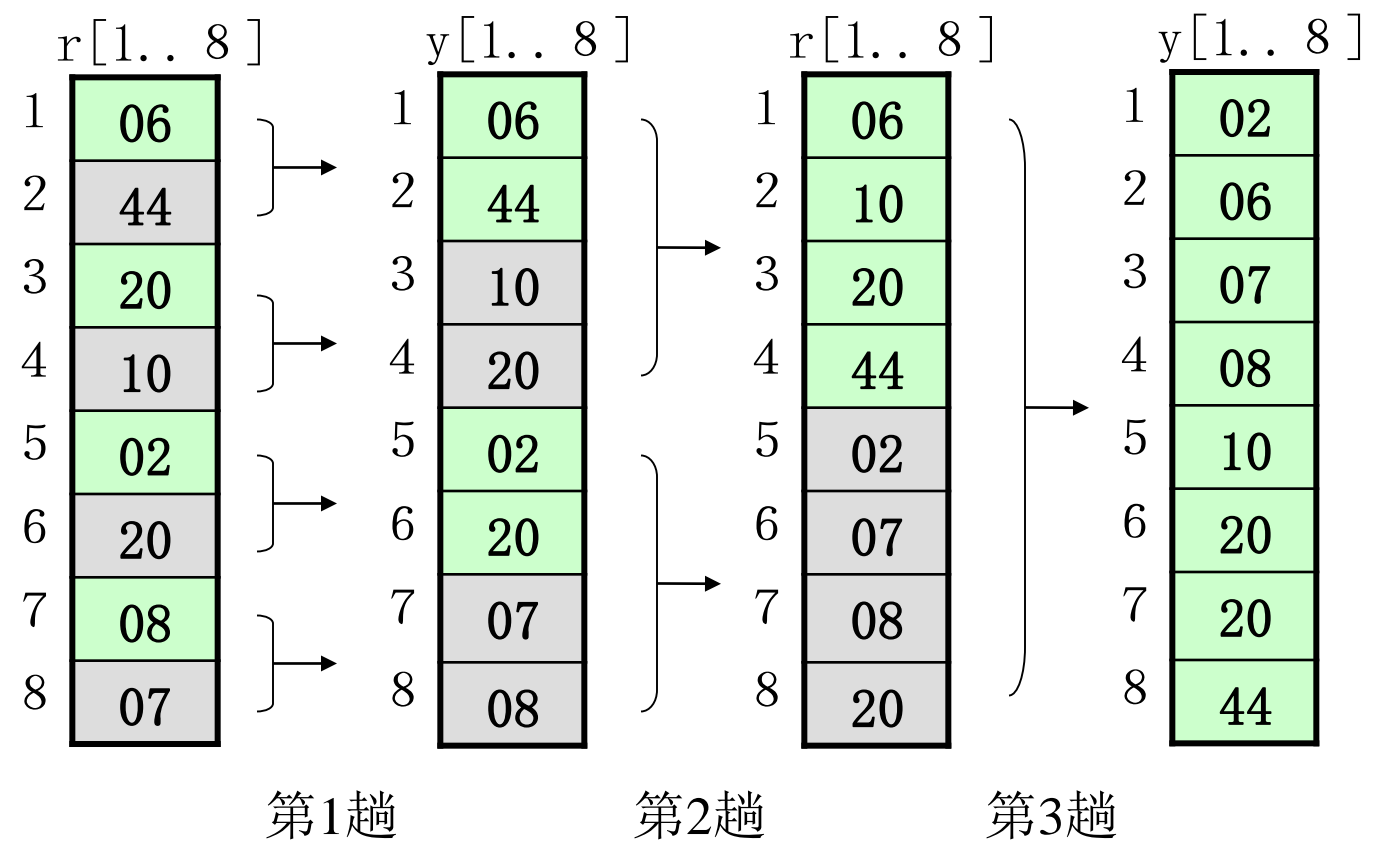
**第1趟:** 从 $r[1..n]$ 中的第1个和第2个有序子文件开始, 调用算法merge, 每次归并两个相邻子文件, 归并结果放到 $y[1..n]$ 中。在 $y$ 中形成  $\lceil n/2 \rceil$  个长度为2的有序子文件。若 $n$ 为奇数, 则 $y$ 中最后一个子文件的长度为1。

**第2趟：**把 $y[1..n]$ 看作输入文件, 将 $\lceil n/2 \rceil$ 个有序子文件两两归并, 归并结果回送到 $r[1..n]$ 中, 在 $r$ 中形成 $\lceil \lceil n/2 \rceil / 2 \rceil$ 个长度为4的有序子文件。若 $y$ 中有奇数个子文件, 则 $r$ 中最后一个子文件的长度为2。

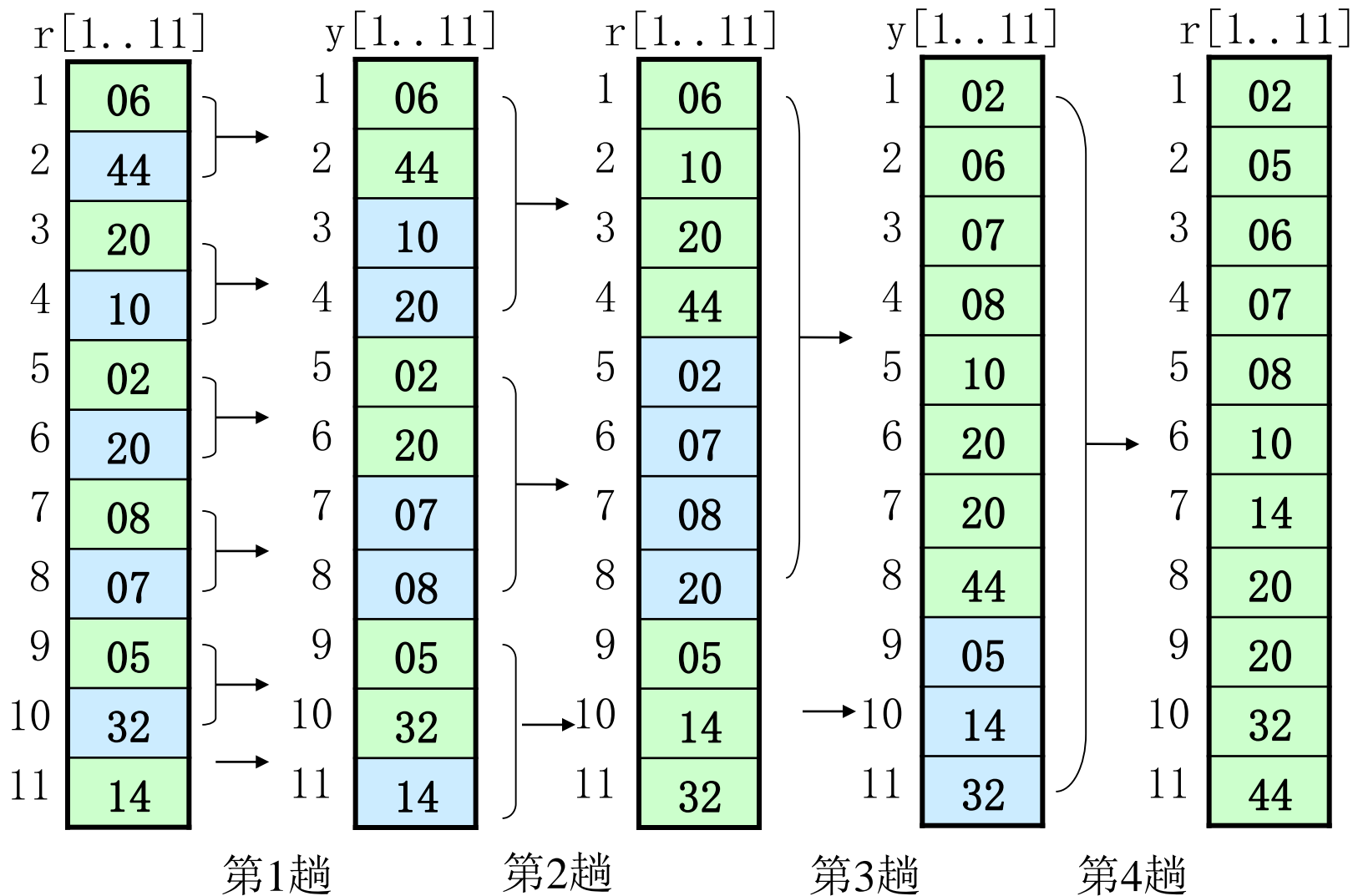
.....

共计经过 $\lceil \log_2 n \rceil$ 趟归并, 最后得到 $n$ 个记录的有序文件。

例1.对8个记录作2路归并排序,共进行 $\lceil \log_2 8 \rceil = 3$  趟归并。



例2. 对11个记录作2-路归并排序, 进行 $\lceil \log_2 11 \rceil = 4$ 趟归并。

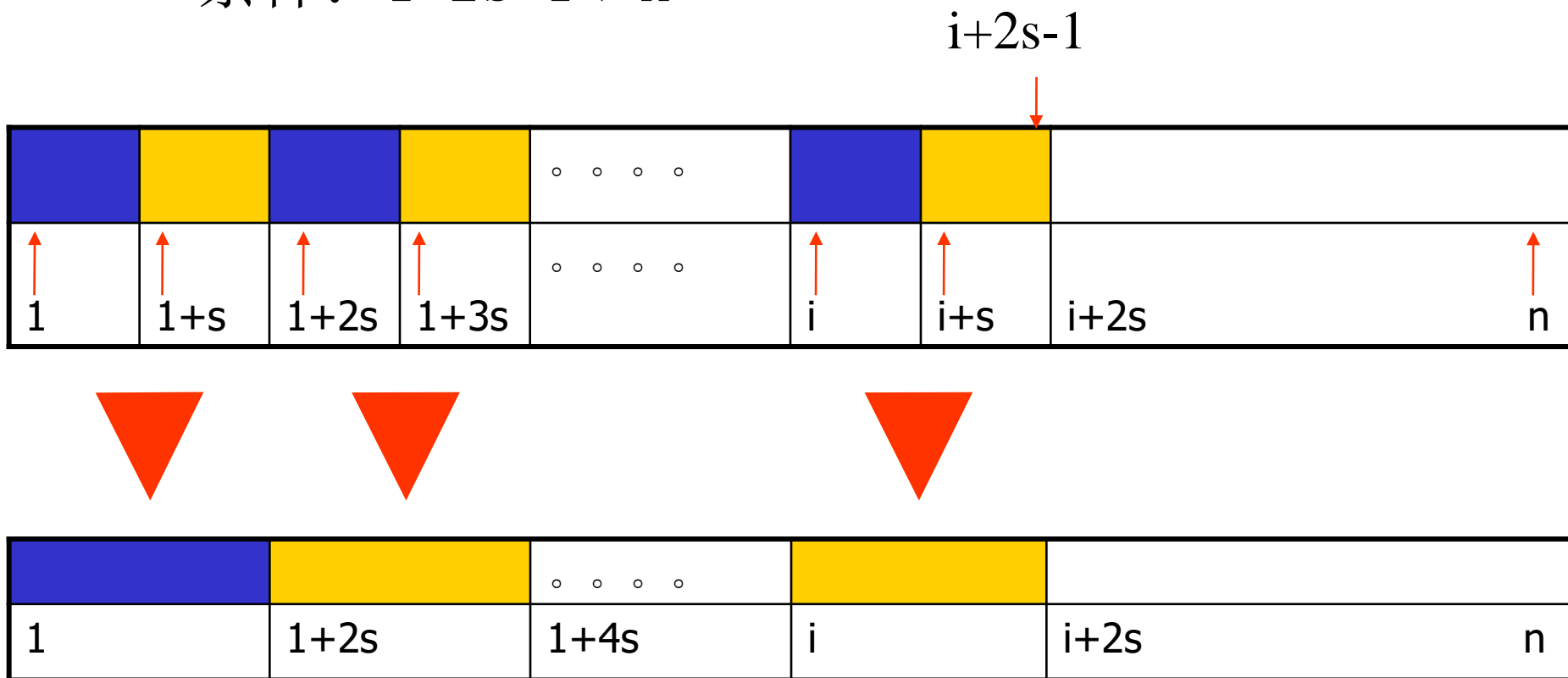


## 一趟归并排序算法:

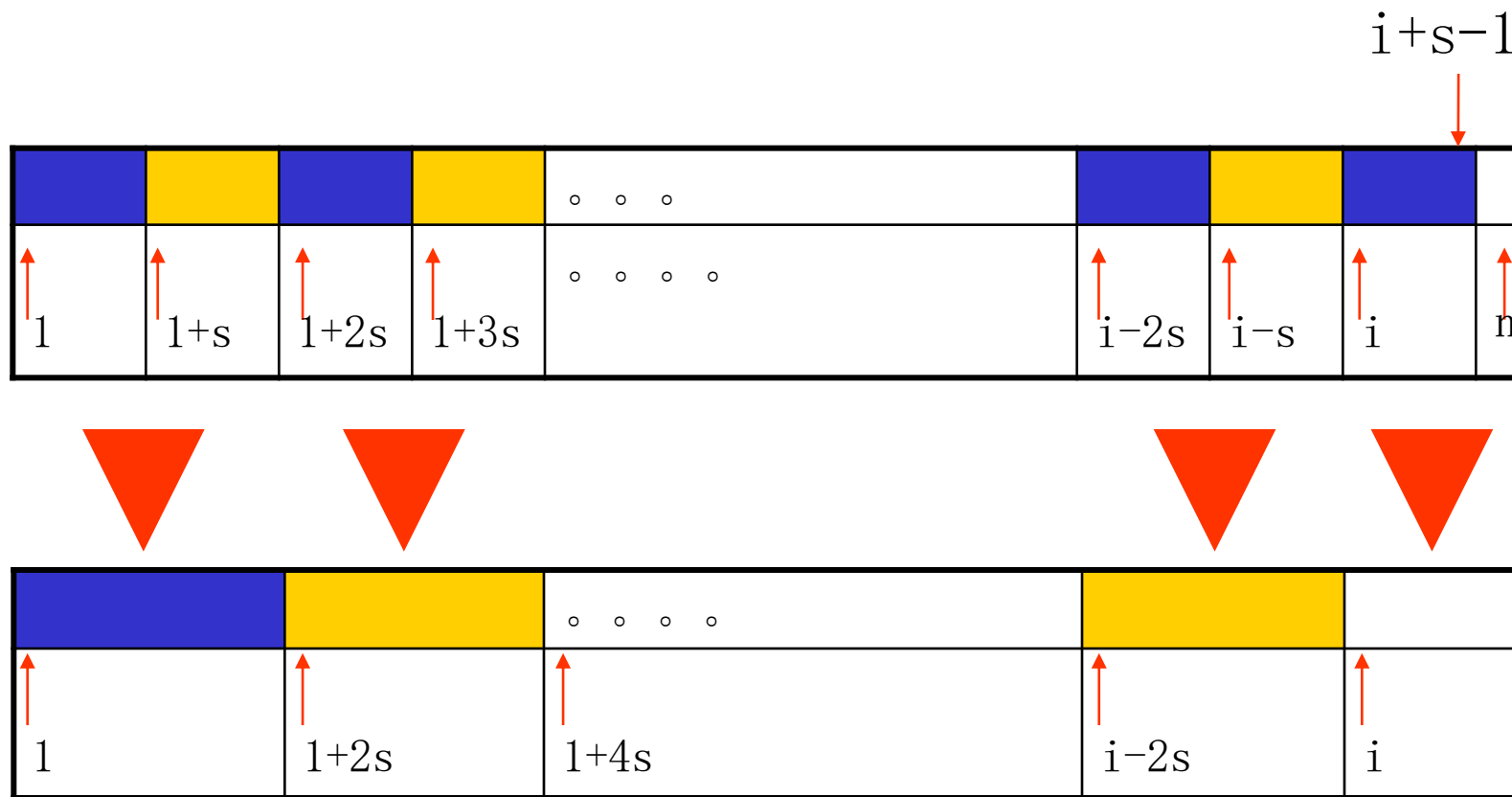
假设:  $s$  为子文件的长度, 将  $r$  中的子文件归并到  $y$  中

(1) 两等长子文件合并

条件:  $i+2s-1 \leq n$



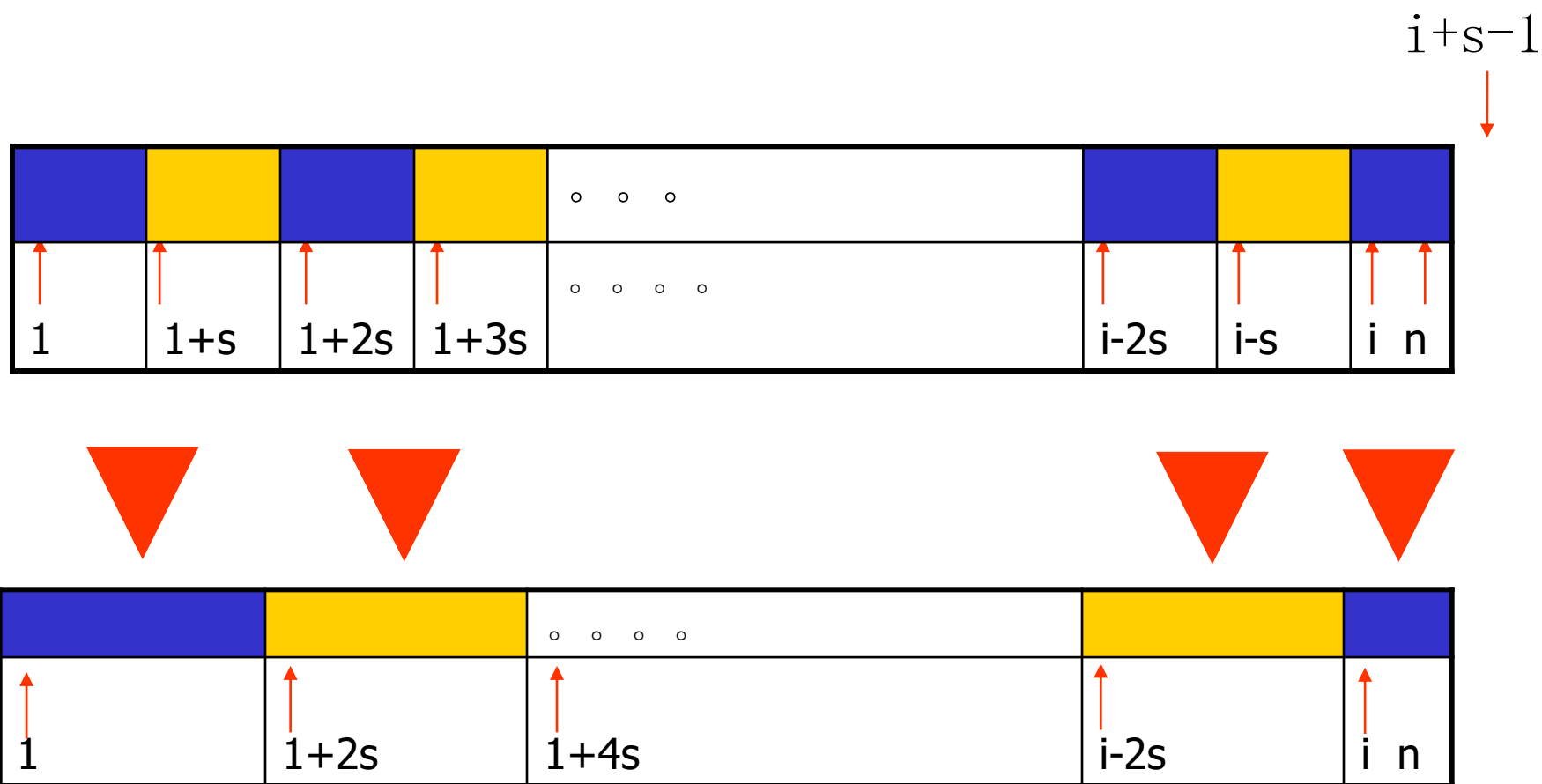
(2) 长度为 $s$ 的子文件与长度取值在  $[1, s)$  内的子文件合并  
 条件:  $i+2s-1 > n$  并且  $i+s-1 < n$





(3) 剩余一个长度为  $[1, s]$  子文件时, 直接复制。

条件:  $i+s-1 > n$



## 一趟归并排序算法:

```
void mergepass(RecType r[], RecType y[], int s)
{ int i=1;
  while(i+2*s-1<= n)           //两两归并长度均为s的子文件
  { merge(r, y, i, i+s-1, i+2*s-1);
    i=i+2*s;
  }
  if (i+s-1<n)                 //最后两个子长度为s和长度不足s的文件
    merge(r, y, i, i+s-1, n);
  else
    while(i<=n)                //复制最后一个子文件, 长度≤s
    { y[i]=r[i];
      i++;
    }
}
```

对文件 $r[1..n]$ 归并排序的算法(调用算法mergepass)

```
void mergesort(RecType r[], int n)
{
    RecType y[n+1] ;
    int s=1;                //子文件初始长度为1
    while (s<n)
    { mergepass(r, y, s);    //将 $r[1..n]$ 归并到 $y[1..n]$ 
      s=2*s;                //修改子文件长度
      mergepass(y, r, s);    //将 $y[1..n]$ 归并到 $r[1..n]$ 
      s=2*s;                //修改子文件长度
    }
}
```

## 算法分析

- 对 $n$ 个记录的文件进行归并排序, 共需  $\lceil \log_2 n \rceil$  趟, 每趟所需比较关键字的次数不超过 $n$ , 共比较  $O(n \log_2 n)$  次。
- 每趟移动 $n$ 个记录, 共移动  $O(n \log_2 n)$  个记录。
- 归并排序需要一个大小为 $n$ 的辅助空间  $y[1..n]$ 。
- 归并排序是稳定的。

## 10.6 基数排序

前面的各种排序算法中，需要进行关键字间的比较。基数排序不同于前面的各种算法，仅分析关键字自身每位的值，通过分配、回收进行处理。

本算法假定记录的关键字为整型（实际并不限制）。

基数排序有两种方法：

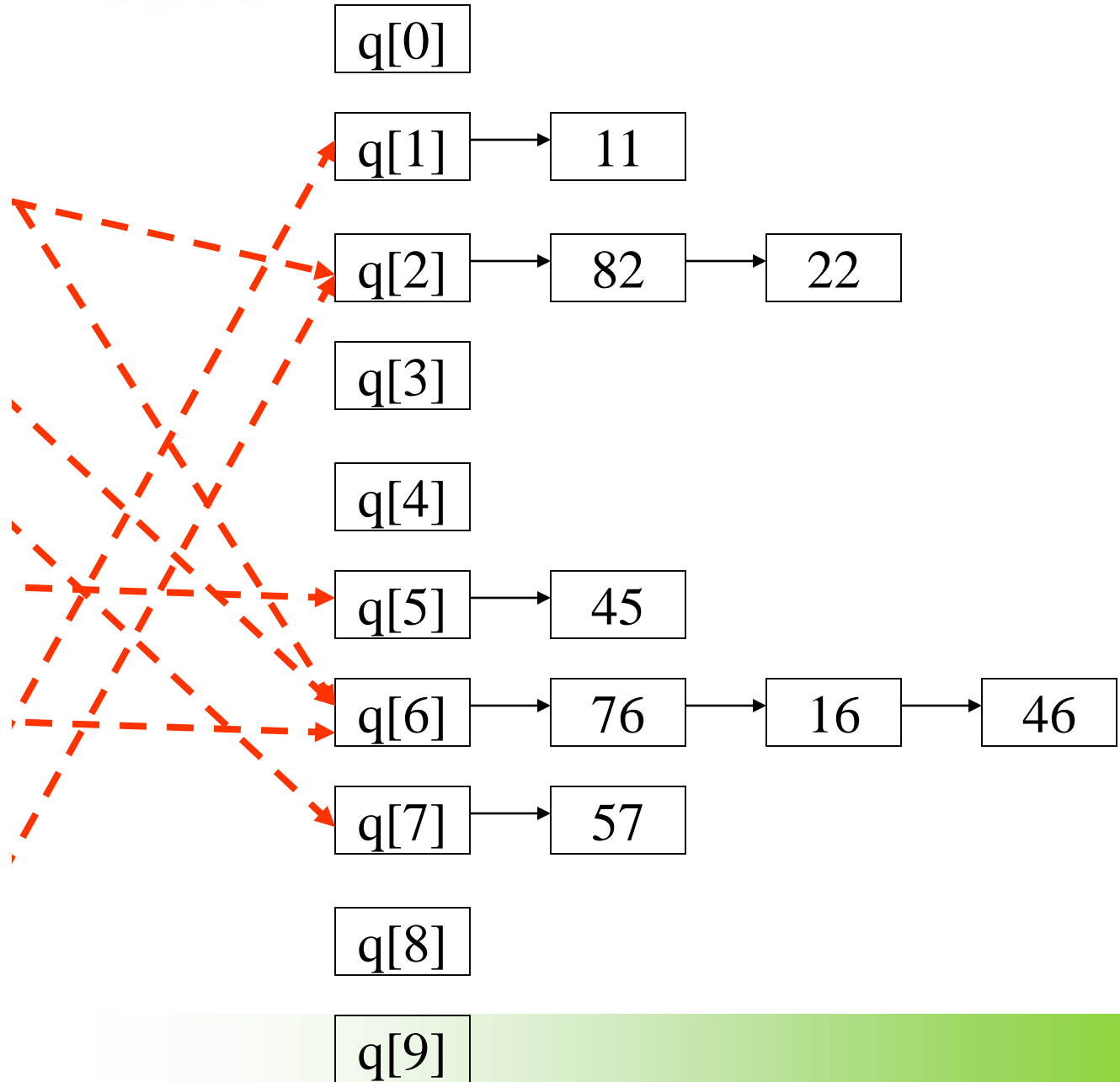
（1）首先根据最高有效位进行排序，然后根据次高有效位进行排序，直到根据最低有效位进行排序，产生一个有序序列。

（2）首先根据最低有效位进行排序，然后根据次低有效位进行排序，直到根据最高有效位进行排序，产生一个有序序列。

10个队列

第一次分配

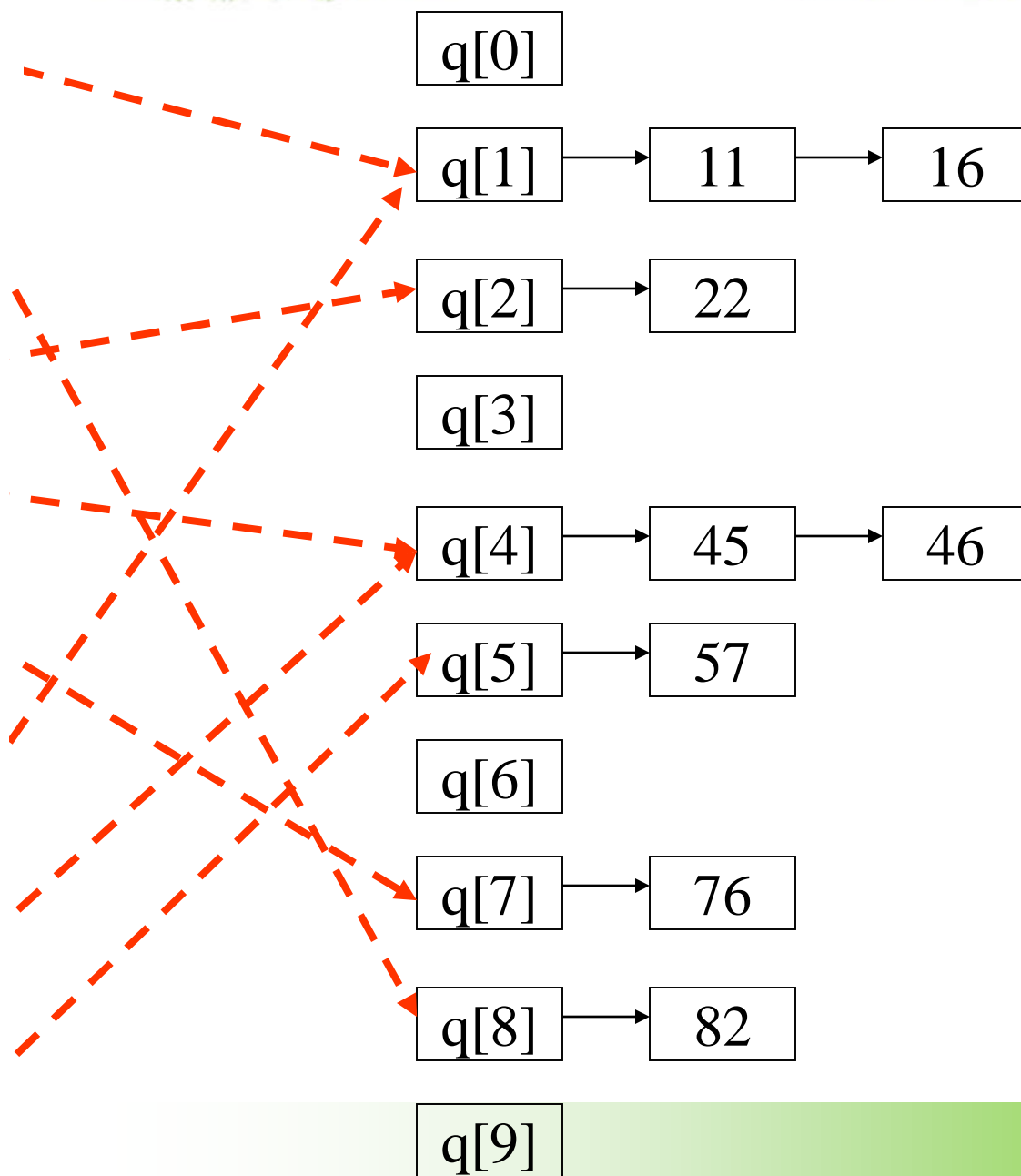
原始序列



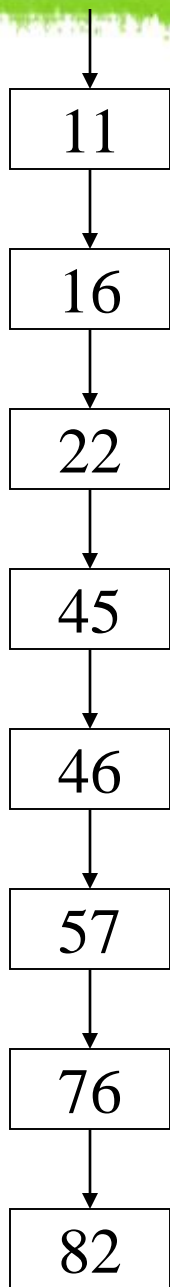
10个队列

第二次分配

第一次收集结果



第二次收集结果



## 算法分析:

- (1) 设数字有效位为 $d$ 位, 需要 $d$ 趟分配、回收。
- (2) 每趟分配运算时间 $O(n)$
- (3) 收集: 基数为 $r_d$ , 即 $r_d$ 个队列。从 $r_d$ 个队列中收集, 运算时间 $O(r_d)$
- (4) 一趟分配、回收运算时间 $O(n+r_d)$ , 时间复杂度 $O(d*(n+r_d))$
- (5) 基数排序是稳定的。
- (6) 辅助空间: 每个队列首尾2个指针, 共 $2r_d$ 个指针。



## 练习：

用基数排序法描述出下面序列的排序过程：  
设待排序文件各记录的关键字为：

288,371,260,531,287,235,56,299,18,23。

第一趟排序后，结果为：

260, 371, 531, 23, 235, 56, 287, 288, 18, 299

第二趟排序后，结果为：

18, 23, 531, 235, 56, 260, 371, 287, 288, 299

第三趟排序后，结果为：

18, 23, 56, 235, 260, 287, 288, 299, 371, 531

## 10.7内部排序方法比较

### 一、 时间性能

按平均的时间性能来分，有三类排序方法：

(1) 时间复杂度为 $O(n\log_2 n)$ 的方法有：快速排序、堆排序和归并排序，其中快速排序目前被认为是最快的一种排序方法。后两者之比较，在  $n$  值较大的情况下，归并排序较堆排序更快。

(2) 时间复杂度为 $O(n^2)$ 的有：插入排序、冒泡排序和选择排序，其中以插入排序最为常用，特别是对于已按关键字基本有序排列的记录序列尤为如此。选择排序过程中记录移动次数最少。

(3) 时间复杂度为 $O(n)$ 的排序方法只有基数排序一种。

(4) 当待排记录序列按关键字顺序有序时，插入排序和冒泡排序能达到 $O(n)$ 的时间复杂度；而对于快速排序而言，这是最不好的情况，此时的时间性能蜕化为 $O(n^2)$ ，因此应尽量避免。

(5) 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。

(6) 以上对排序的时间复杂度的讨论主要考虑排序过程中所需进行的关键字间的比较次数，当待排序记录中其它各数据项比关键字占有更大的数据量时，还应考虑到排序过程中移动记录的操作时间，有时这种操作的时间在整个排序过程中占的比例更大，从这个观点考虑，简单排序的三种排序方法中冒泡排序效率最低。

## 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：插入、冒泡和选择排序)和堆排序的空间复杂度均为 $O(1)$ 。

(2) 快速排序为 $O(\log_2 n)$ ，为递归程序执行过程中栈所需的辅助空间。

(3) 归并排序和基数排序所需辅助空间最多，其空间复杂度为 $O(n)$ 。

### 三、排序方法的稳定性

稳定的排序方法指的是，对于两个关键字相等的记录在经过排序之后，不改变它们在排序之前在序列中的相对位置。

(1) 除希尔排序、快速排序、直接选择和堆排序是不稳定的排序方法外，本章讨论的其它排序方法都是稳定的。

(2) “稳定性”是由方法本身决定的。一般来说，排序过程中所进行的比较操作和交换数据仅发生在相邻的记录之间，没有大步距的数据调整时，则排序方法是稳定的。



类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	Shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	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
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	不稳定
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		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定

注：基数排序的复杂度中， $r$ 代表关键字的基数， $d$ 代表长度， $n$ 代表关键字的个数。

# 本章小结

本章主要讨论各种内部排序的方法。学习本章的目的是了解各种排序方法的原理以及各自的优缺点，以便在编制软件时能按照情况所需合理选用。一般来说，在选择排序方法时，可有下列几种选择：

1) 若待排序的记录个数 $n$ 值较小（例如 $n < 30$ ），则可选用插入排序法，但若记录所含数据项较多，所占存储量大时，应选用选择排序法（降低移动次数）。反之，若待排序的记录个数 $n$ 值较大时，应选用快速排序法。但若待排序记录关键字有“有序”倾向时，就慎用快速排序，而宁可选用堆排序或归并排序，而后两者的最大差别是所需辅助空间不等。

2) 快速排序和归并排序在 $n$ 值较小时的性能不及直接插入排序，因此在实际应用时，可将它们和插入排序“混合”使用。如在快速排序划分子区间的长度小于某值时，转而调用直接插入排序；或者对待排记录序列先逐段进行直接插入排序，然后再利用“归并操作”进行两两归并直至整个序列有序为止。

3) 基数排序的时间复杂度为 $O(d \times n)$ ，因此特别适合于待排记录数  $n$  值很大，而关键字“位数  $d$ ”较小的情况。

4) 一般情况下，对单关键字进行排序时，所用的排序方法是否稳定无关紧要。但当按“最次位优先”进行多关键字排序时(除第一趟外)必须选用稳定的排序方法。