

# 数据结构

## 10 排序

董洪伟 陈聪 周世兵

联系电话：13812529213

E-mail: worldguard@163.com

# 主要内容

- 什么是排序
- 内部排序
  - 插入式排序：直接插入排序法、希尔排序法
  - 交换式排序：起泡法、快速排序法
  - 选择式排序：直接选择排序法、锦标赛排序法、堆排序
  - 归并排序
- 各种内部排序方法的比较

# 什么是排序

- **排序**：按照一定的规则，对一系列数据进行排列
- **数据表**：待排序的数据对象的有限集合

5	0	2	9	4	3	7	1	4	6
---	---	---	---	---	---	---	---	---	---



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---



9	8	7	6	5	4	3	2	1	0
---	---	---	---	---	---	---	---	---	---

# 什么是排序

## • 排序码

- 通常数据对象有多个属性域，即多个数据成员组成，其中有一个属性域可用来区分对象，作为排序依据
- 用哪个属性域作为排序码，要视具体的应用需要而定
- 比如描述学生数据有姓名、学号、年龄、成绩等属性域，可以按照学号排序，也可以按照姓名、年龄等等

# 什么是排序

- 稳定性

– 例：

按学号排序：

姓名	学号	年龄
张三	1	21
李四	2	20
王五	3	20

按年龄排序：

姓名	学号	年龄
王五	3	20
李四	2	20
张三	1	21

# 什么是排序

- 排序算法的稳定性

- 如果有两个数据 $r[i]$ 和 $r[j]$ ，它们的排序码 $k[i] == k[j]$ ，且在排序之前， $r[i]$ 排在 $r[j]$ 前面
- 如果在排序之后， $r[i]$ 仍在对象 $r[j]$ 的前面，则称这个排序方法是稳定的，否则称这个排序方法是不稳定的

# 什么是排序

- 内部排序

- 数据对象全部存放在内存中进行的排序

- 外部排序

- 数据对象个数太多，不能同时存放在内存中，只能存放在外部存储器中，根据排序过程的要求，取一部分到内存中来排序，然后再存回外部存储器

# 什么是排序

- 排序算法优劣的衡量

- 时间复杂度

- 平均情况

- 最好情况和最差情况：有一些算法，其复杂度受最初数据的排列情况影响较大

- 空间复杂度

- 排序时所需的额外的存储空间

- 额外：指除了存放数据以外还需要的



# 什么是排序

- 教材采用的排序表的结构定义

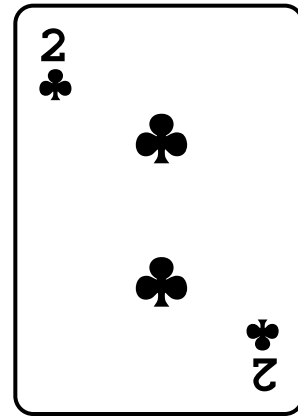
```
#define MAXSIZE 20          // 最多元素个数
typedef int KeyType;        // 关键字类型
typedef struct{
    KeyType key;            // 关键字
    InfoType otherinfo;
}RedType;                  // 一条数据记录
typedef struct{
    RedType r[MAXSIZE+1];  // 数组
    int length;            // 元素个数
}SqlList;
```

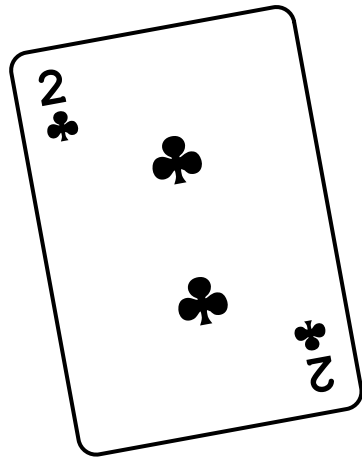
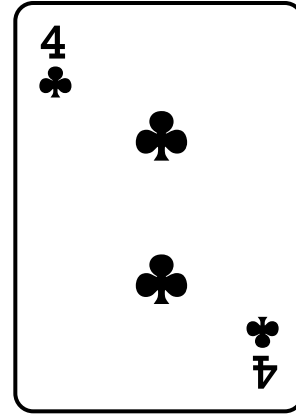
# 插入式排序

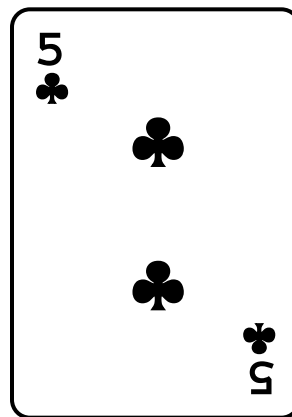
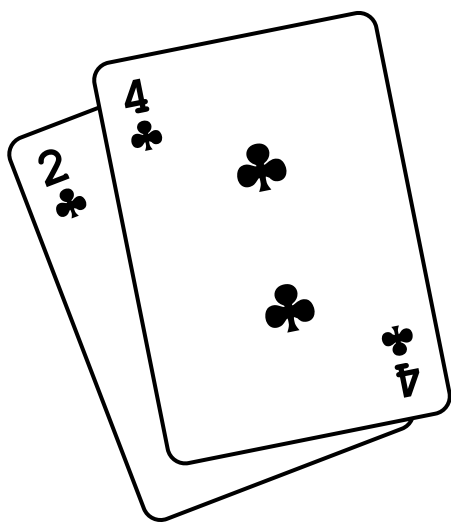
- 基本思想：

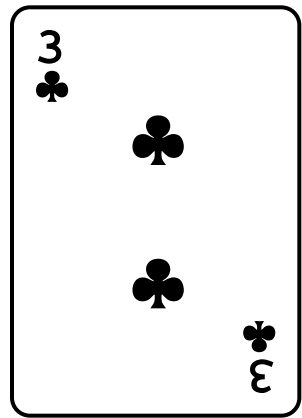
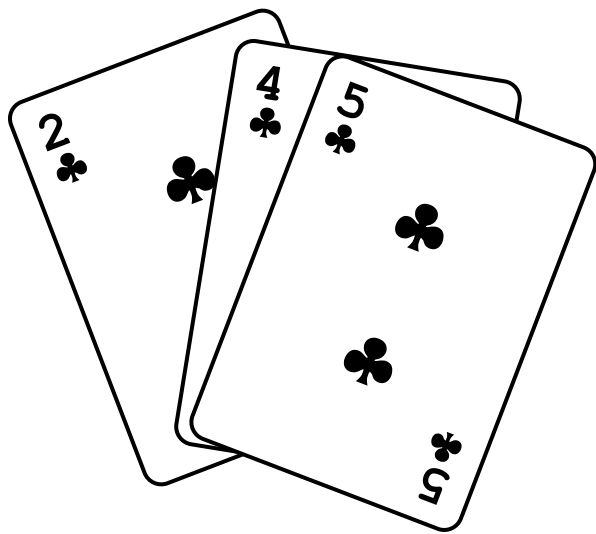
- 每一步将一个待排序的对象，按其排序码大小，插入到前面已经排好序的一组对象的适当位置上，直到全部插入为止
- 类比：扑克牌抓牌











# 直接插入排序

- 基本思想

- 假设当插入第 $i$  ( $i \geq 1$ )个元素时, 前面的 $v[1], v[2], \dots, v[i-1]$ 都已经排好了序
- 这时, 用 $v[i]$ 的排序码与 $v[i-1], v[i-2], \dots, v[1]$ 的排序码进行比较, 找到正确的插入位置, 将 $v[i]$ 插入, 原来位置上的对象向后顺移

# 直接插入排序：基本思想

- 例

$i$   
↓

49	38	65	97	76	13	27
----	----	----	----	----	----	----

--	--	--	--	--	--	--



# 直接插入排序：基本思想

## • 例

$i$   
↓

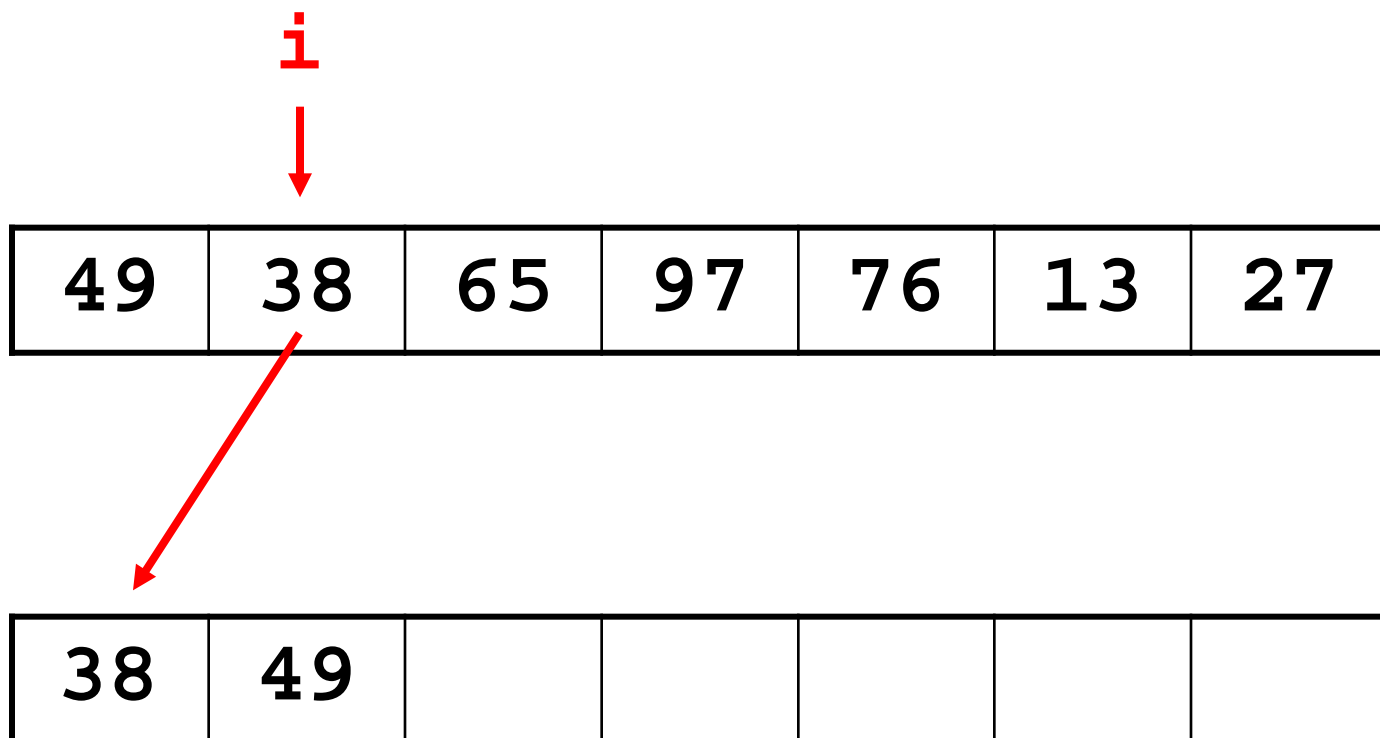
49	38	65	97	76	13	27
----	----	----	----	----	----	----



49						
----	--	--	--	--	--	--

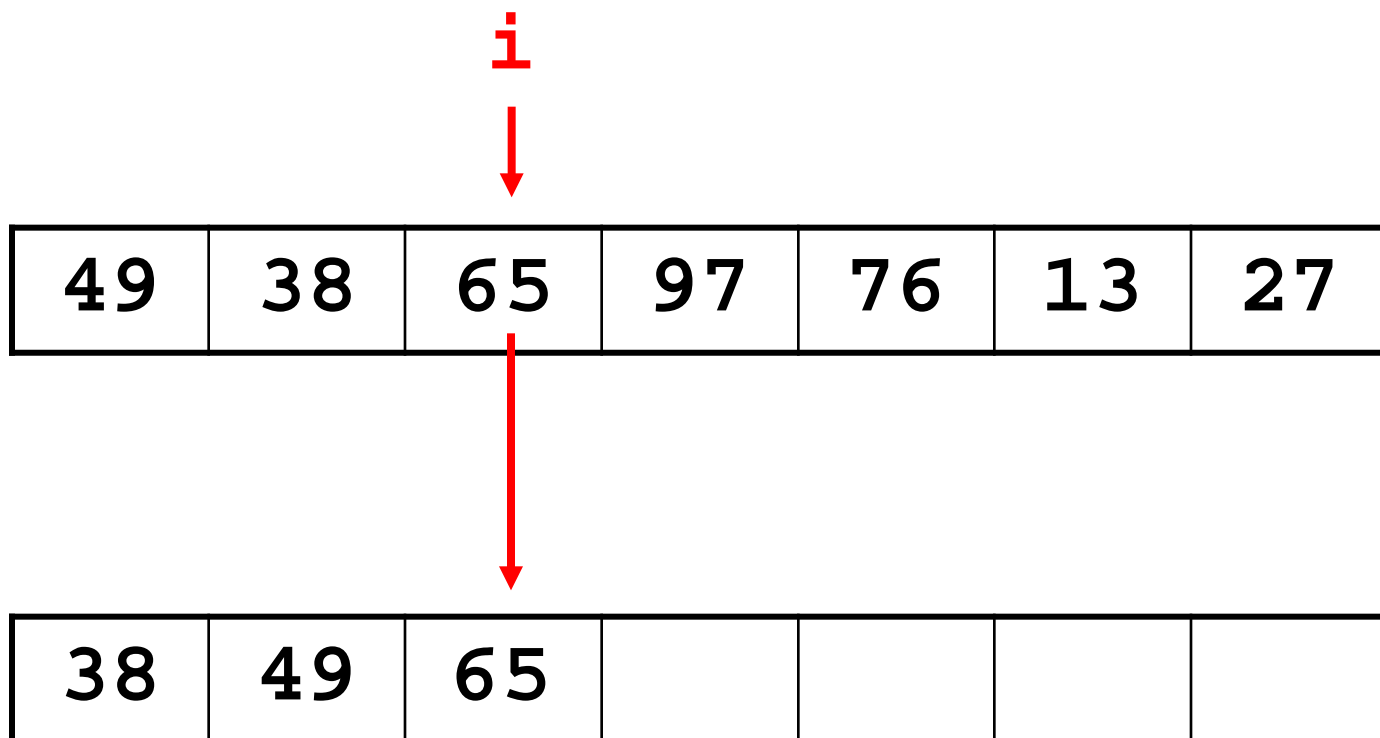
# 直接插入排序：基本思想

## • 例



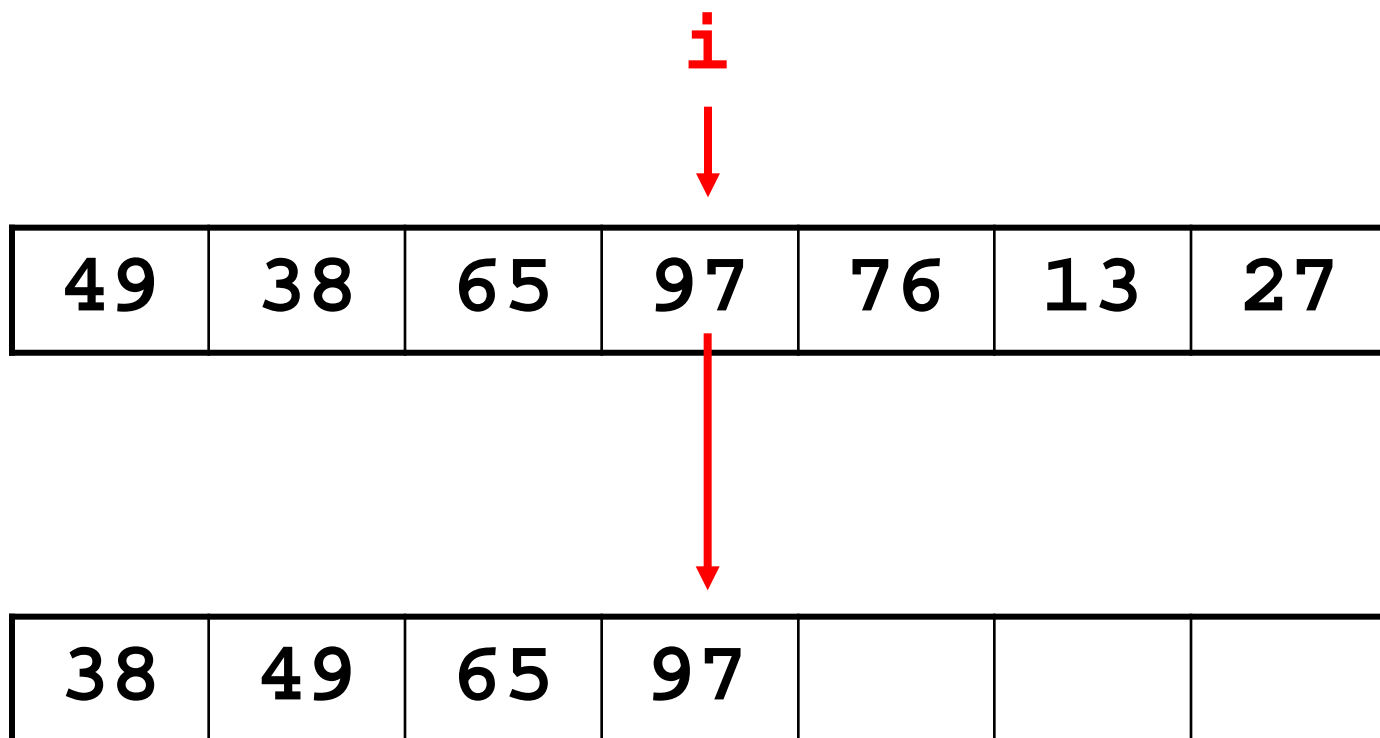
# 直接插入排序：基本思想

## • 例



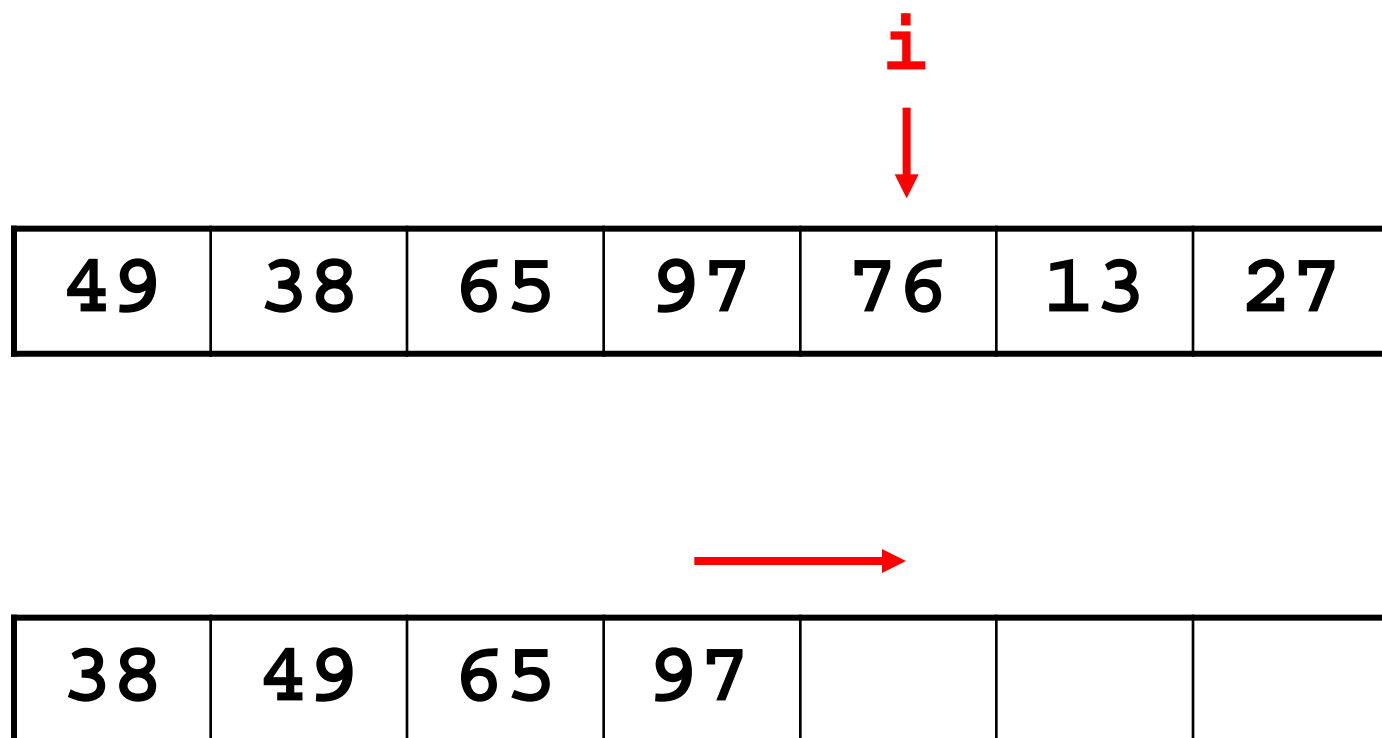
# 直接插入排序：基本思想

## • 例



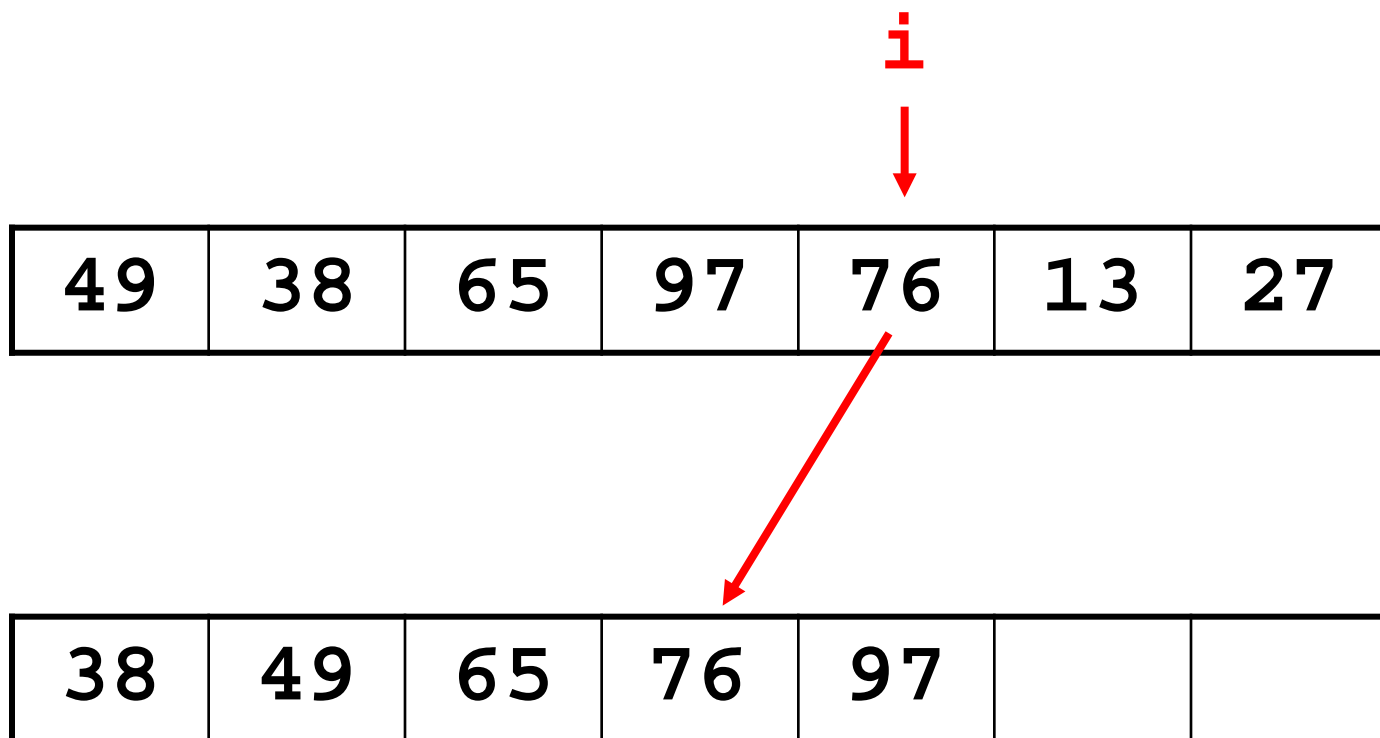
# 直接插入排序：基本思想

## • 例



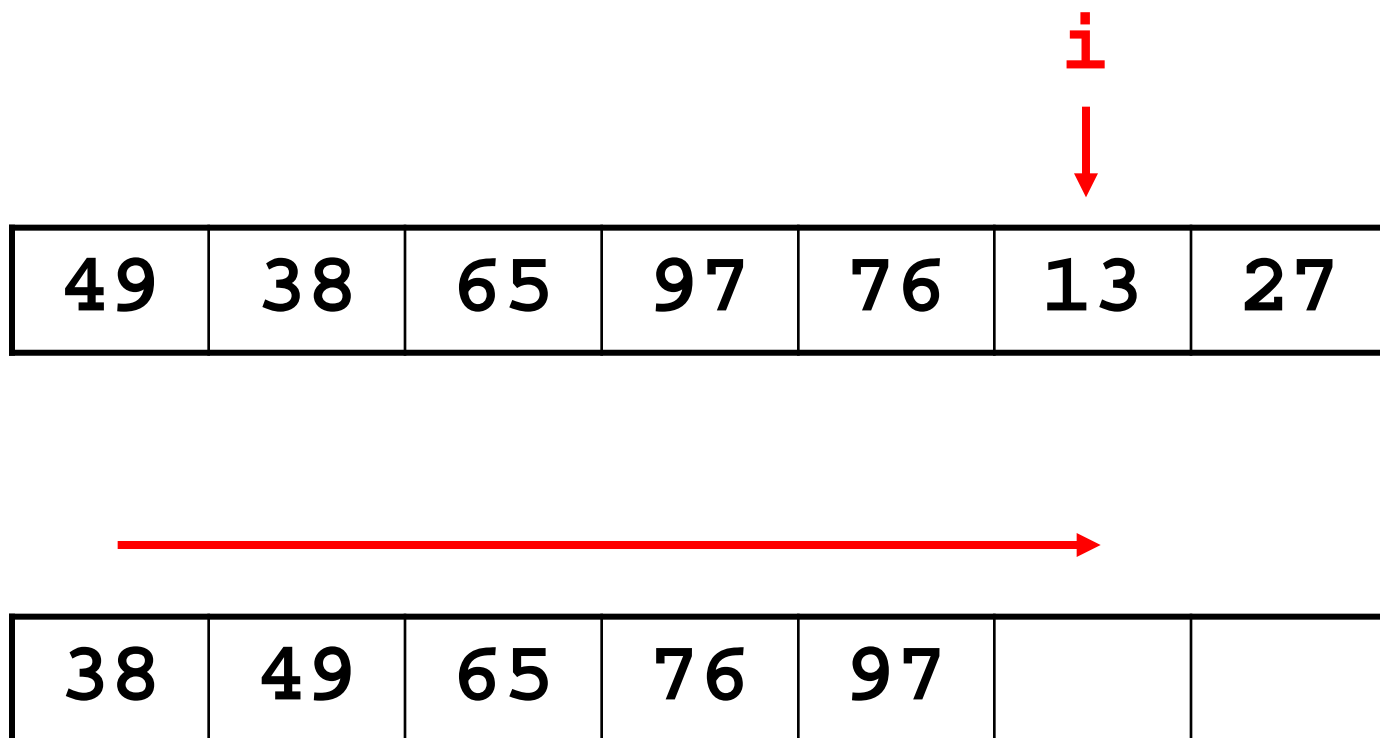
# 直接插入排序：基本思想

## • 例



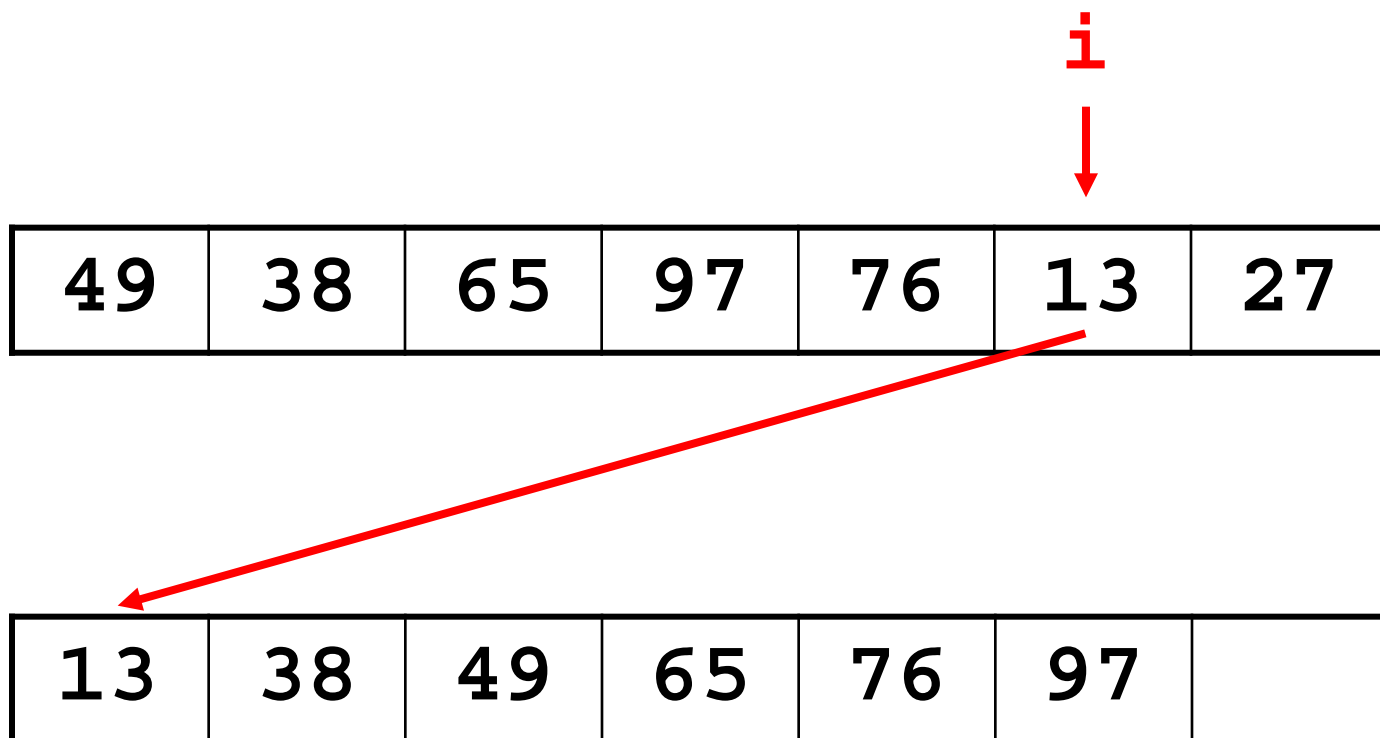
# 直接插入排序：基本思想

## • 例



# 直接插入排序：基本思想

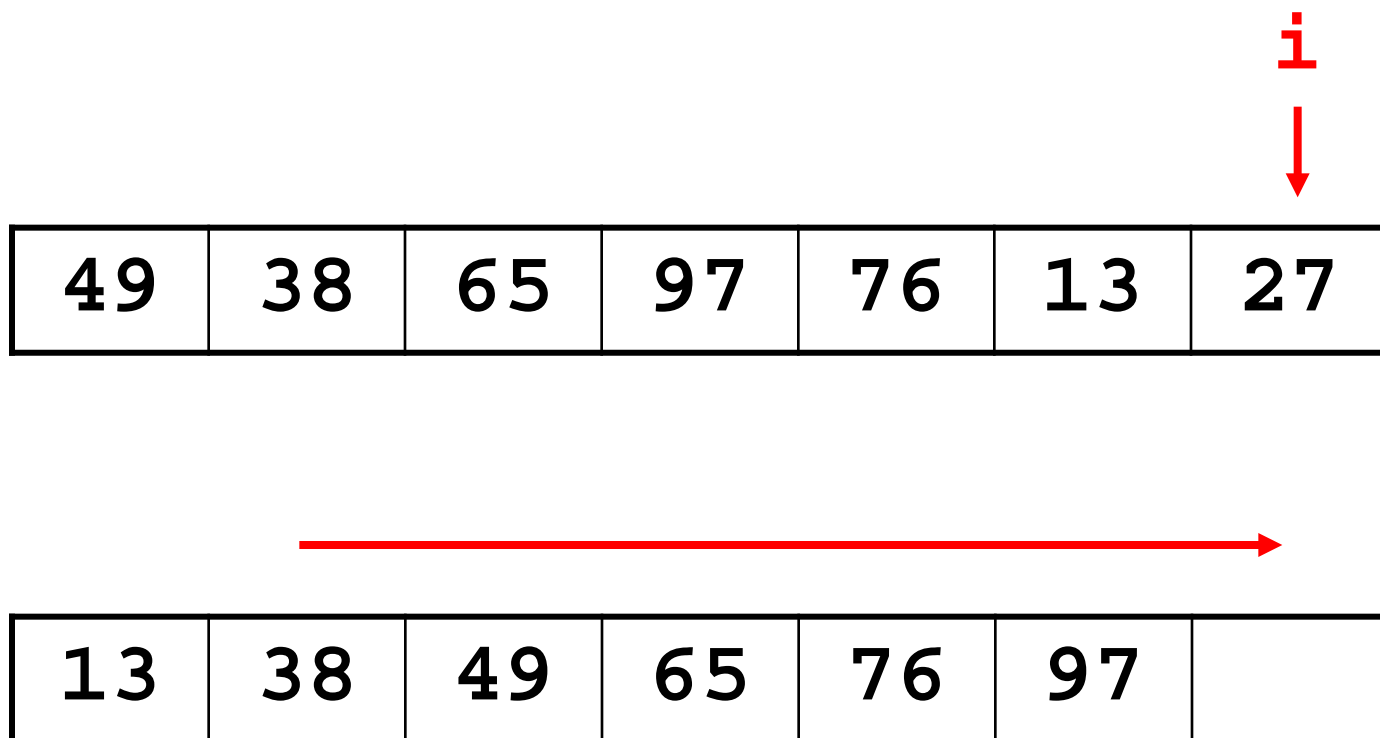
## • 例





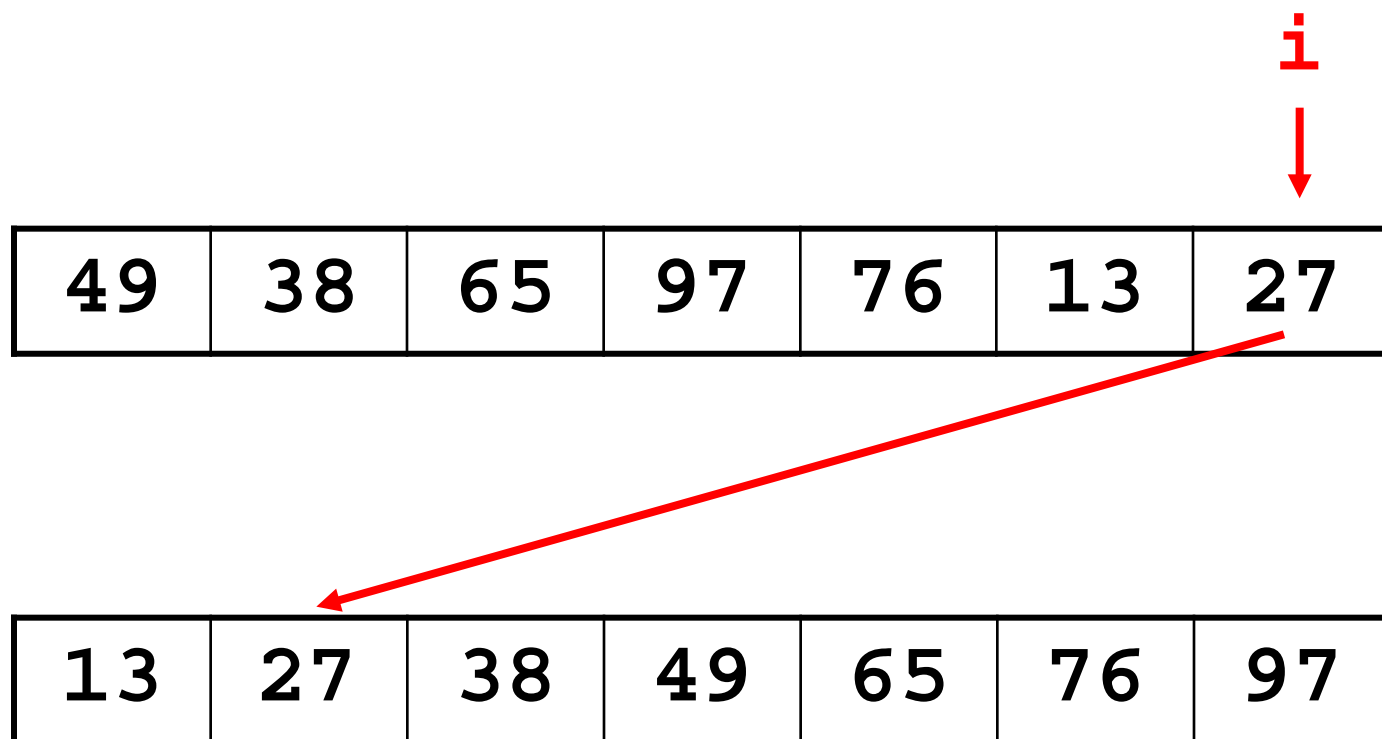
# 直接插入排序：基本思想

## • 例



# 直接插入排序：基本思想

## • 例



## • 算法

```
void InsertSort(SqList &L) {  
    for(i = 2; i <= L.length; i ++)  
        if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\  
        {  
            L.r[0] = L.r[i];           //用r[0]先记录r[i]的值  
            L.r[i] = L.r[i-1];          //r[i-1]后移一个单元  
  
            //从i-2开始, 往左扫描, 直到找到一个<=r[0]的  
            for(j=i-2; LT(L.r[0].key, L.r[j].key); j--)  
                L.r[j+1] = L.r[j];      //每个元素后移  
            L.r[j+1] = L.r[0];           //最后把r[0]写入  
        }  
}
```

```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--){
            L.r[j+1] = L.r[j];      //每个元素后移
        }
        L.r[j+1] = L.r[0];         //最后把r[0]写入
    }

```

		<b>i</b> ↓					
0	1	2	3	4	5	6	7
	38	65	97	76	13	27	49

```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--){
            L.r[j+1] = L.r[j];      //每个元素后移
        }
        L.r[j+1] = L.r[0];         //最后把r[0]写入
    }

```

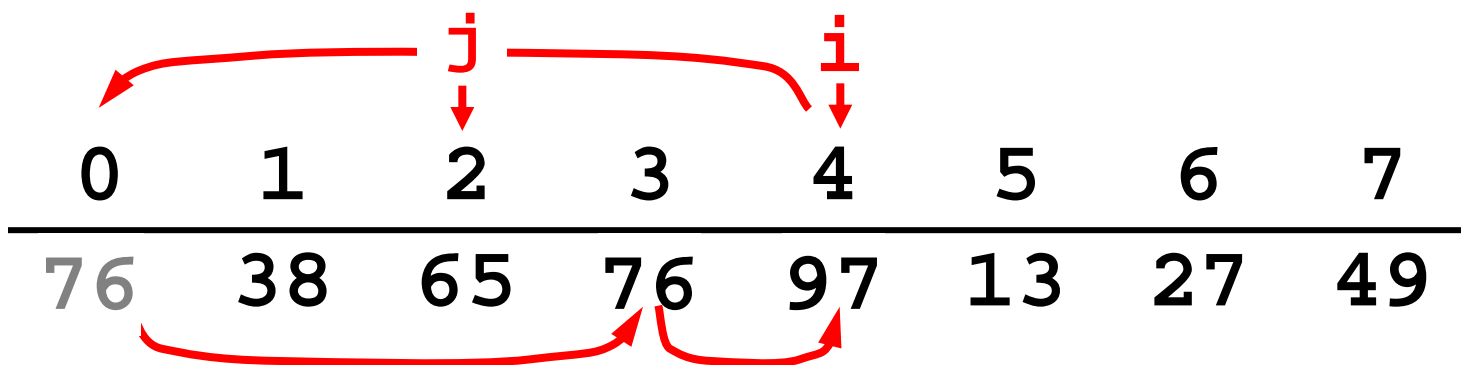
i  
↓

0	1	2	3	4	5	6	7
	38	65	97	76	13	27	49

```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--){
            L.r[j+1] = L.r[j];      //每个元素后移
            L.r[j+1] = L.r[0];      //最后把r[0]写入
        }
    }

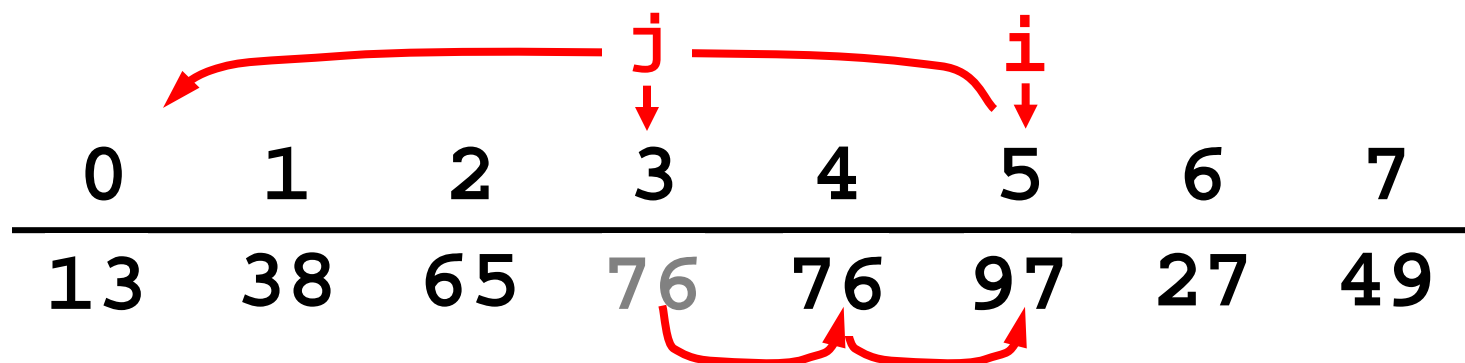
```



```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j-- )
            L.r[j+1] = L.r[j];      //每个元素后移
        L.r[j+1] = L.r[0];          //最后把r[0]写入
    }

```



```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--){
            L.r[j+1] = L.r[j];     //每个元素后移
            L.r[j+1] = L.r[0];     //最后把r[0]写入
        }
    }

```

0	1	2	3	4	5	6	7
13	38	65	65	76	97	27	49



```
for(i = 2; i <= L.length; i ++)  
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\  
    {  
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值  
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元  
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的  
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--)  
            L.r[j+1] = L.r[j];     //每个元素后移  
        L.r[j+1] = L.r[0];         //最后把r[0]写入  
    }
```

0	1	2	3	4	5	6	7
13	38	38	65	76	97	27	49

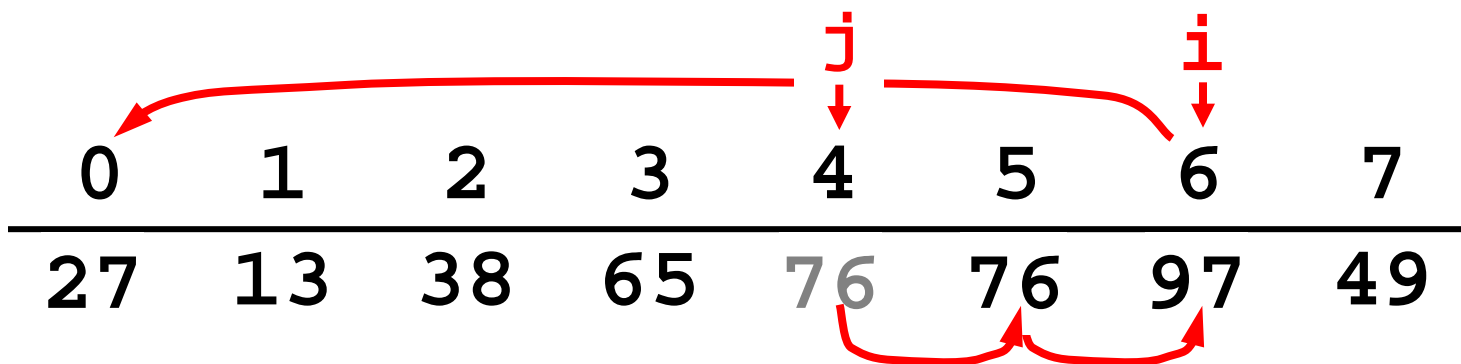
```
for(i = 2; i <= L.length; i ++)  
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小  
    {  
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值  
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元  
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的  
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--)  
            L.r[j+1] = L.r[j];     //每个元素后移  
        L.r[j+1] = L.r[0];         //最后把r[0]写入  
    }
```

0	1	2	3	4	5	6	7
13	13	38	65	76	97	27	49

```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j-- )
            L.r[j+1] = L.r[j];     //每个元素后移
        L.r[j+1] = L.r[0];         //最后把r[0]写入
    }

```



```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--)
            L.r[j+1] = L.r[j];      //每个元素后移
        L.r[j+1] = L.r[0];          //最后把r[0]写入
    }

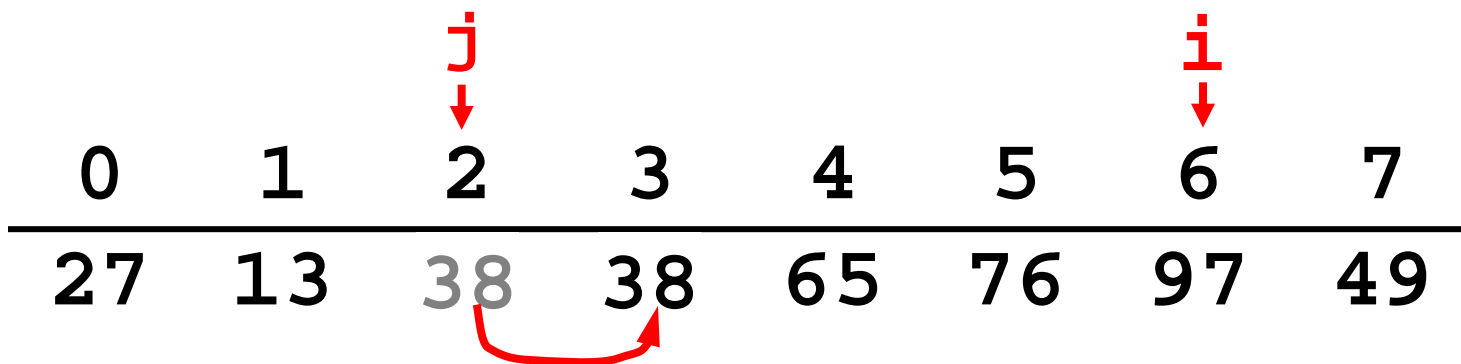
```

0	1	2	3	4	5	6	7
27	13	38	65	65	76	97	49

Diagram illustrating the insertion sort process. Red arrows indicate the movement of elements. A red arrow labeled 'j' points down to index 3. A red arrow labeled 'i' points down to index 6. A red curved arrow shows the element at index 3 (65) being shifted to index 4.

```
for(i = 2; i <= L.length; i ++)  
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小  
    {  
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值  
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元  
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的  
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--)  
            L.r[j+1] = L.r[j];     //每个元素后移  
        L.r[j+1] = L.r[0];         //最后把r[0]写入  
    }
```

0	1	2	3	4	5	6	7
27	13	38	38	65	76	97	49



```

for(i = 2; i <= L.length; i ++)
    if(LT(L.r[i].key, L.r[i-1].key) //i比i-1小\
    {
        L.r[0] = L.r[i];           //用r[0]先记录r[i]的值
        L.r[i] = L.r[i-1];         //r[i-1]后移一个单元
        //从i-2开始, 往左扫描, 直到找到一个<=r[0]的
        for(j=i-2; LT(L.r[0].key, L.r[j].key); j--)
            L.r[j+1] = L.r[j];     //每个元素后移
        L.r[j+1] = L.r[0];         //最后把r[0]写入
    }

```

0	1	2	3	4	5	6	7
27	13	27	38	65	76	97	49

# 直接插入排序

- 时间复杂度

- 最好的情况： $O(n)$

- 数列已经排好序，只需要比较 $n-1$ 次

- 最差的情况： $O(n^2)$

- 排序之前是逆序

- $n-1$ 次插入，每次都应插入到最左

- 而每次插入从最右开始扫描直到最左，需要比较 $i$ 次

- 所以总的比较次数  $= \sum_{i=2}^n i = (n+2)(n-1) / 2$

# 直接插入排序

- 平均情况： $O(n^2)$ 
  - $n-1$ 次插入，每次插入从最右开始扫描直到遇见一个比当前元素小的，平均需要比较  $i/2$  次
- 空间复杂度
  - $O(1)$
  - 第0个数组空间用来存放临时数据
- 稳定性：
  - 稳定



# 折半插入排序

- 基本思想

- 基本思想跟直接插入排序相同
- 不同在于在查找插入位置时
  - 直接插入排序是采用顺序查找法
  - 折半插入排序采用折半查找法（即二分法）

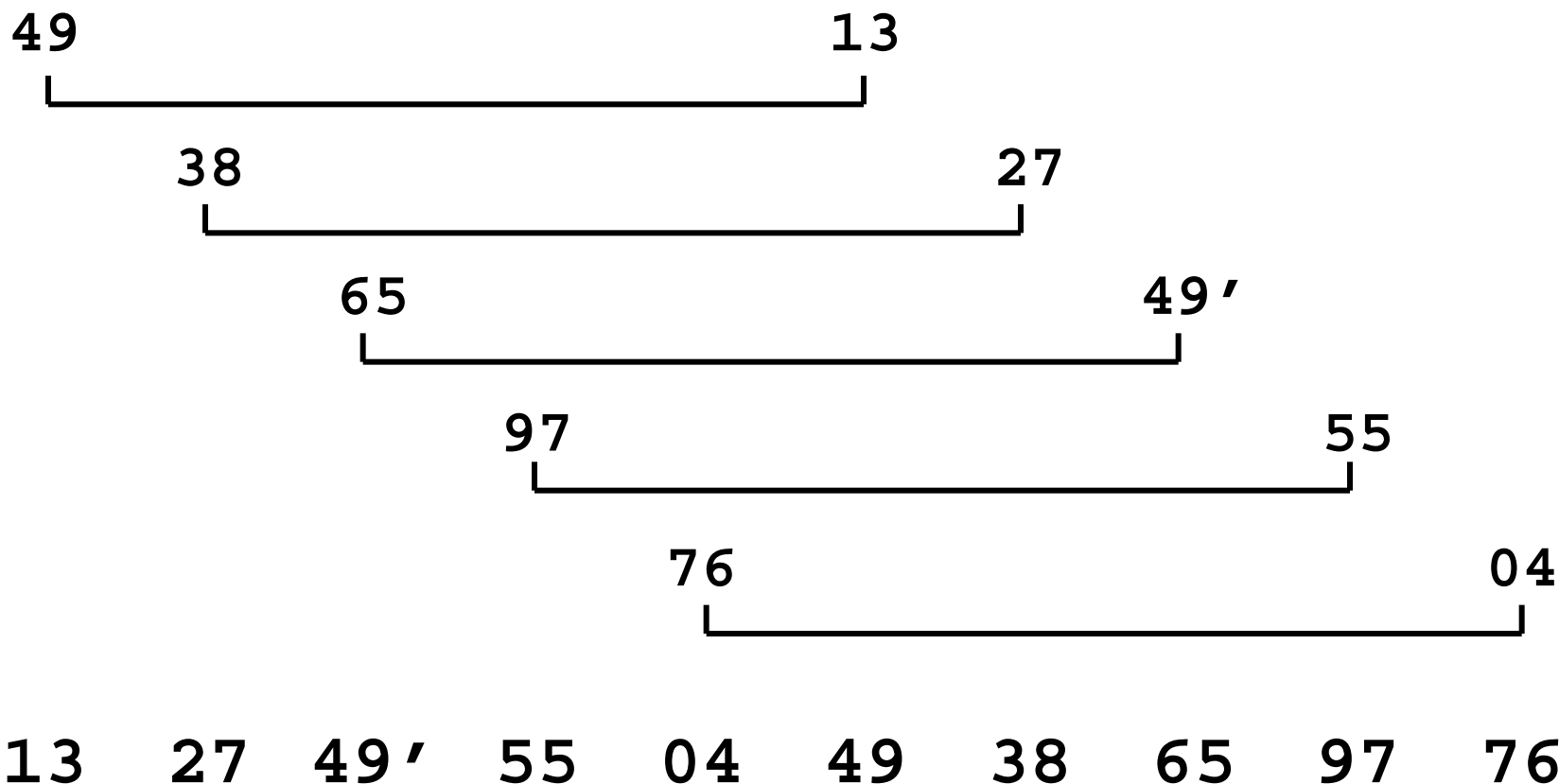
# 希尔排序

- 希尔排序： 又称缩小增量排序
- 基本思想：
  - 设待排序列有 $n$ 个元素，取一整数 $gap$  ( $gap < n$ ) 作为间隔，将全部元素分为 $gap$ 个子序列，所有距离为 $gap$ 的元素放在同一个子序列中
  - 在每一个子序列中分别采用直接插入排序
  - 然后缩小间隔 $gap$ ，例如取 $gap = gap / 2$ ，重复上述的子序列划分和排序工作
  - 直到最后取 $gap = 1$ ，将所有元素放在同一个序列中排序为止

# • 例

– 第一趟排序,  $\text{gap} = 5$

49    38    65    97    76    13    27    49'    55    04



- 第二趟排序, gap = 3

13 27 49' 55 04 49 38 65 97 76

A horizontal number line with four vertical tick marks. Above the first tick mark is the number 13, above the second is 55, above the third is 38, and above the fourth is 76.

A horizontal number line with three points marked by vertical tick marks. Above the first tick mark is the number 27, above the second is 04, and above the third is 65.

[illegible]

13 04 49' 38 27 49 55 65 97 76

-第三趟排序, gap = 1

13    04   49'   38   27   49   55   65   97   76

04   13   27   38   49'   49   55   65   76   97

# 希尔排序

- 回顾直接插入排序的特点：
  - 数据大致有序时最快， $O(n)$
- 希尔排序的原理
  - 开始时gap的值较大，子序列中的元素较少，排序速度较快
  - 随着排序进展，gap值逐渐变小，子序列中元素个数变多，但是由于前面工作的基础，大多数元素已基本有序，所以排序速度仍然很快

# 希尔排序

- Gap的取法

- 最初shell提出取 $\text{gap} = n/2$ ,  $\text{gap} = \text{gap}/2$ , ..., 直到 $\text{gap} = 1$
- Knuth提出取 $\text{gap} = \lfloor \text{gap}/3 \rfloor + 1$
- 还有人提出都取奇数为好
- 也有人提出各gap互质为好

- Knuth利用大量实验统计资料得出:

- 当n很大时, 排序码平均比较次数和对象平均移动次数大约在 $n^{1.25}$ 到 $1.6n^{1.25}$ 的范围内

- 稳定性: 不稳定

# 交换式排序

- 基本思想

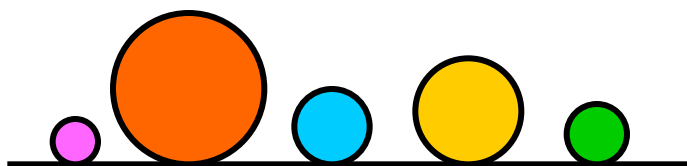
- 两两比较待排序对象的排序码，如果发生逆序（即排列顺序与排序后的次序正好相反），则交换之
- 直到所有对象都排好序为止
- 起泡法
- 快速排序法

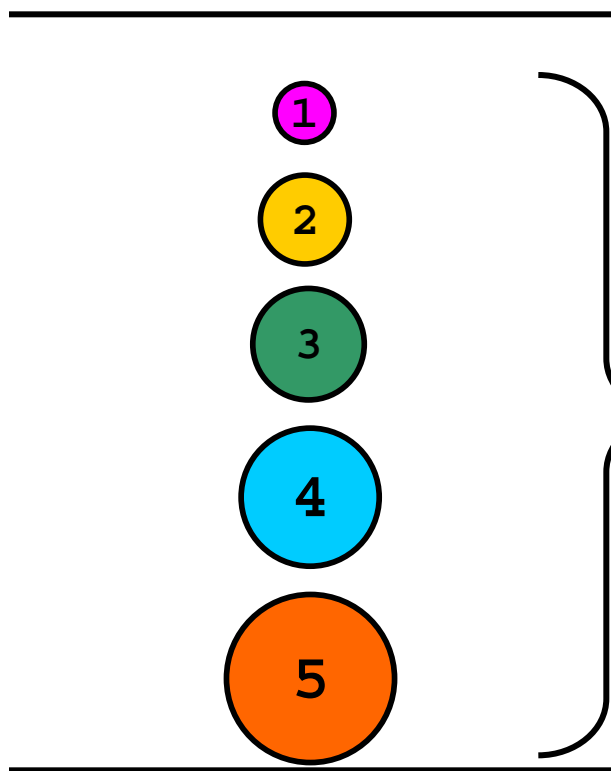


# 起泡排序法

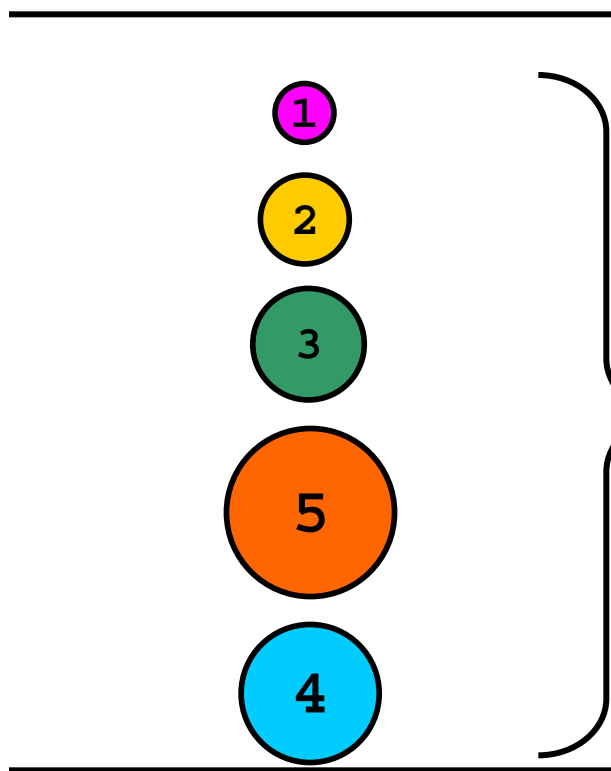
- 基本思想

- 水中气泡，大的还是小的上浮更快？

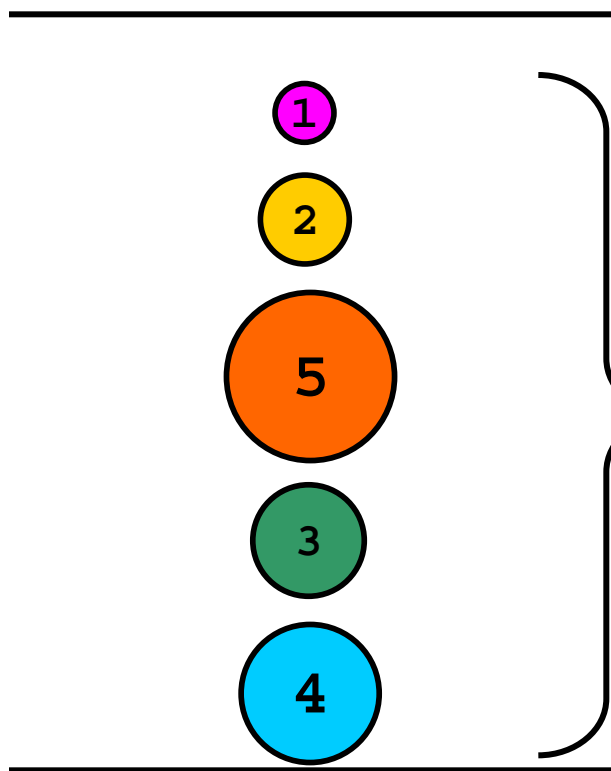




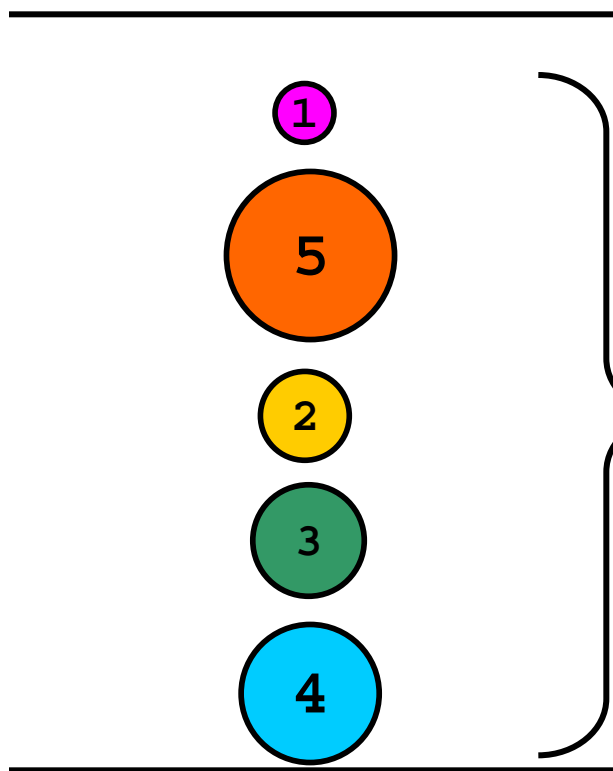
扫描所有气泡：  
上、下两个气泡，  
若大的在下面，则交换



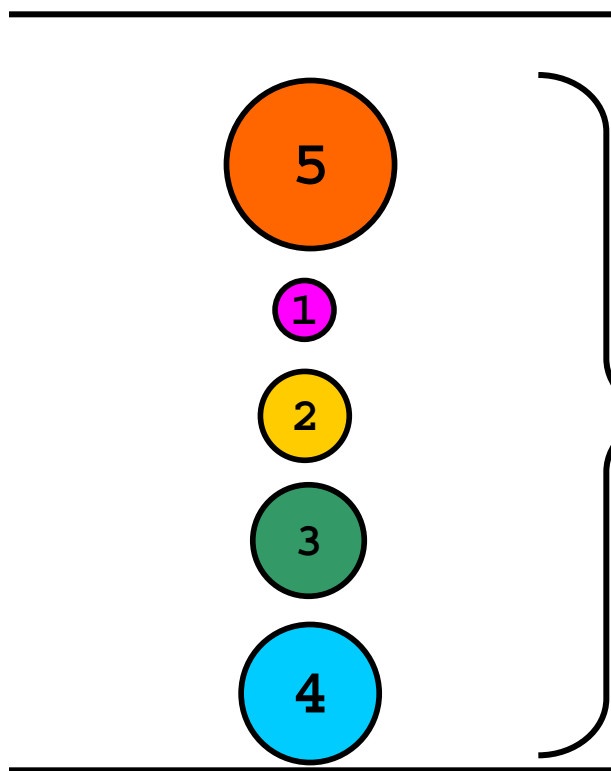
扫描所有气泡：  
上、下两个气泡，  
若大的在下面，则交换



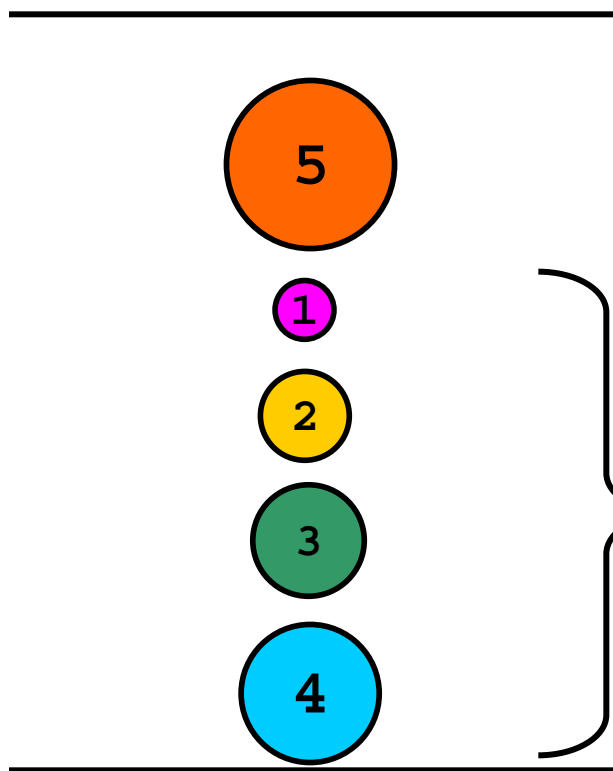
扫描所有气泡：  
上、下两个气泡，  
若大的在下面，则交换



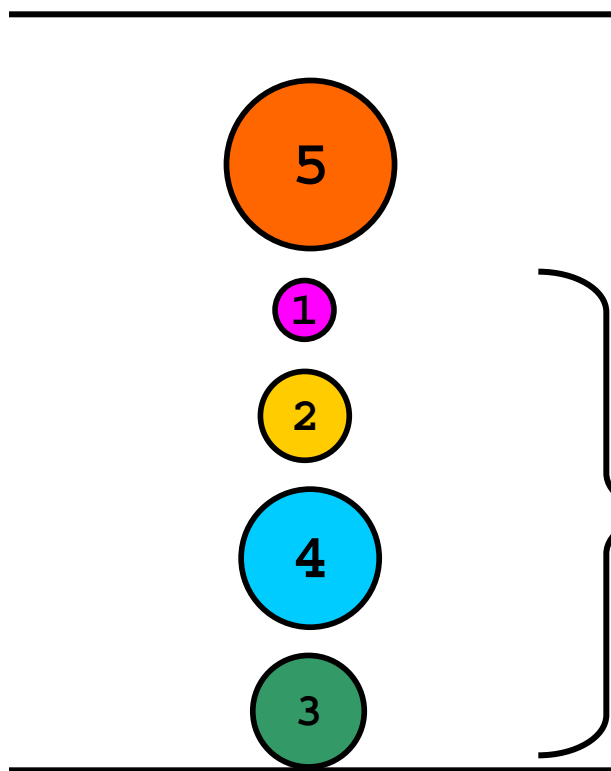
扫描所有气泡：  
上、下两个气泡，  
若大的在下面，则交换



扫描所有气泡：  
上、下两个气泡，  
若大的在下面，则交换

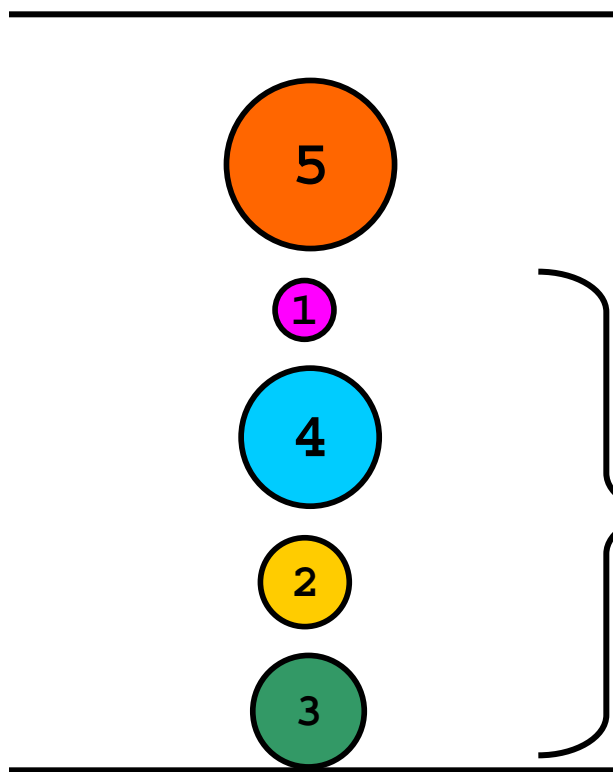


扫描下面4个气泡：  
上、下两个气泡，  
若大的在下面，则交换

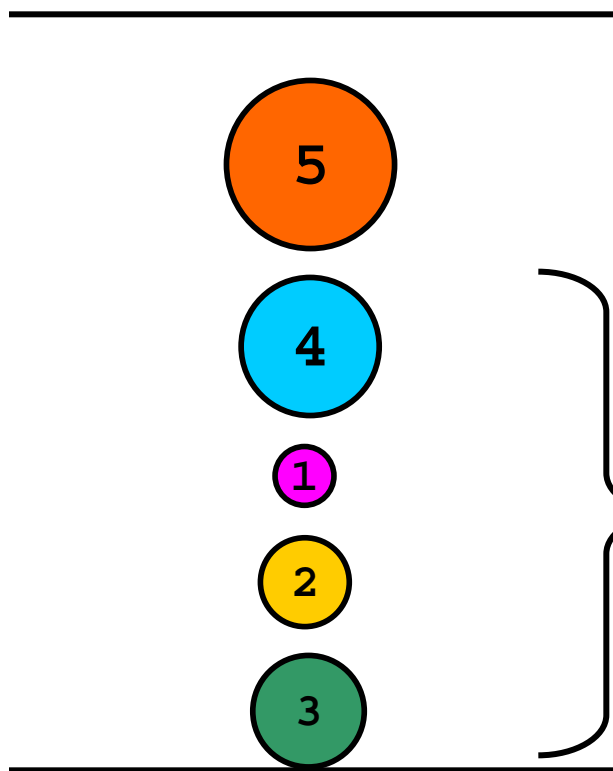


扫描下面4个气泡：  
上、下两个气泡，  
若大的在下面，则交换

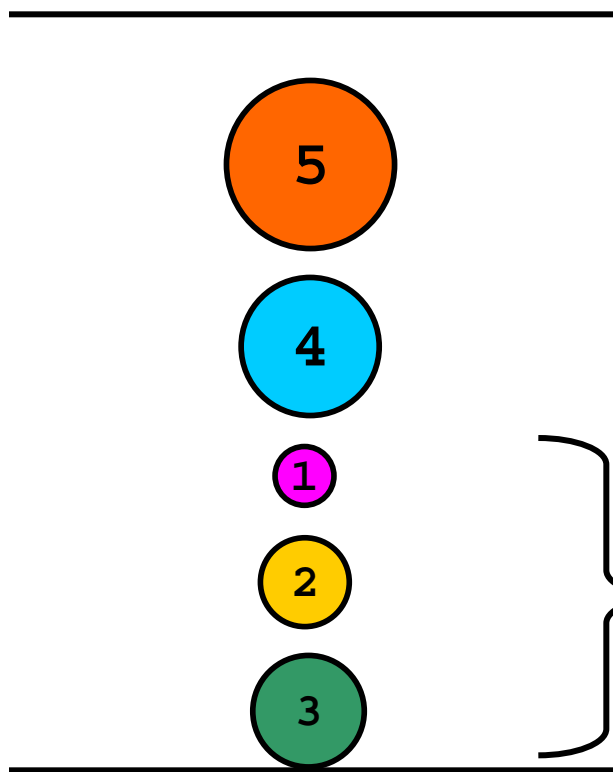




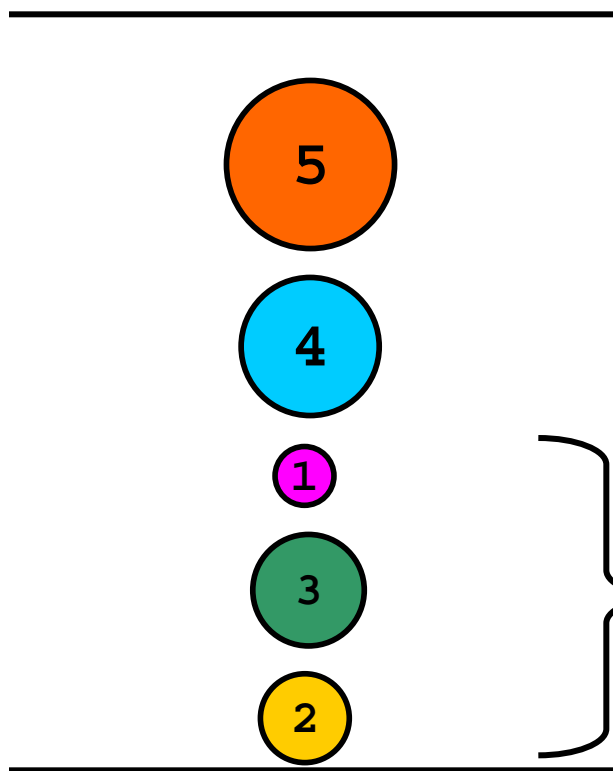
扫描下面4个气泡：  
上、下两个气泡，  
若大的在下面，则交换



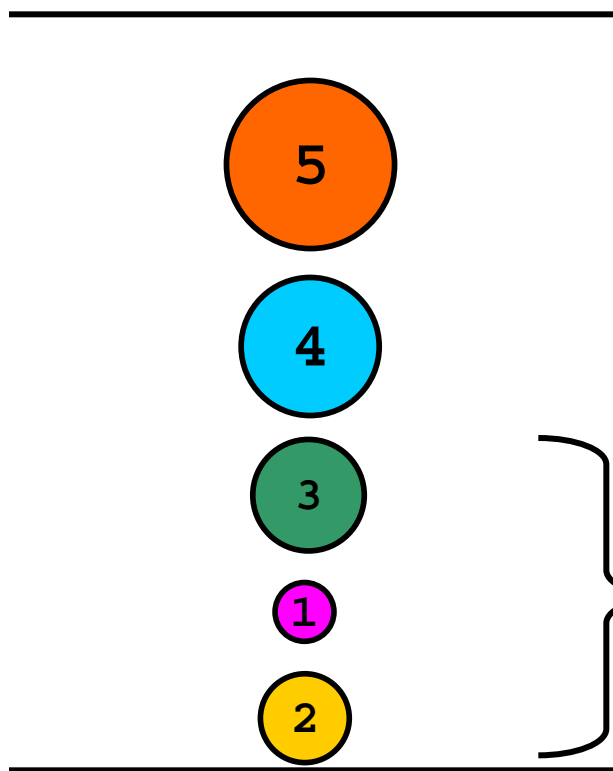
扫描下面4个气泡：  
上、下两个气泡，  
若大的在下面，则交换



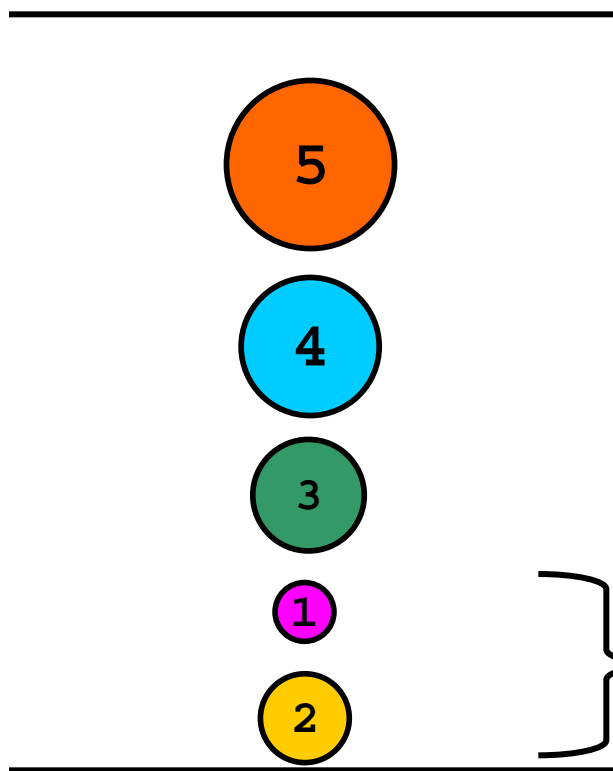
扫描下面3个气泡：  
上、下两个气泡，  
若大的在下面，则交换



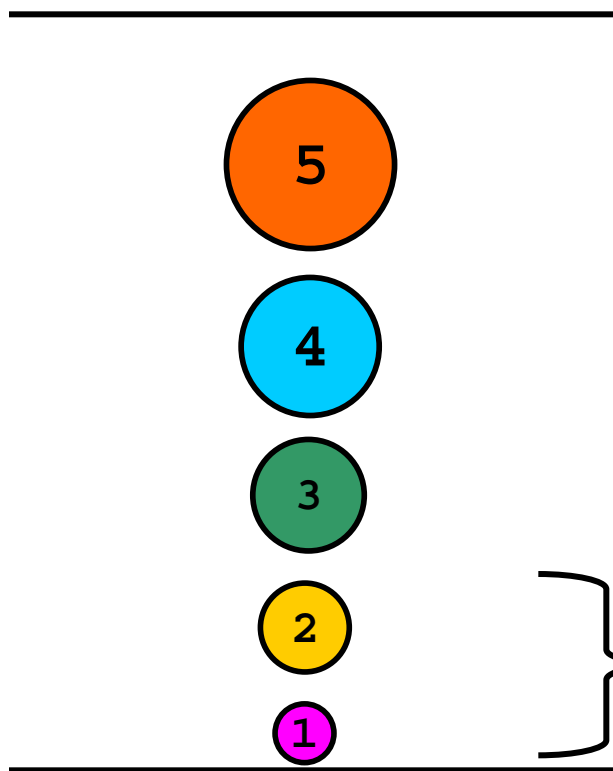
扫描下面3个气泡：  
上、下两个气泡，  
若大的在下面，则交换



扫描下面3个气泡：  
上、下两个气泡，  
若大的在下面，则交换

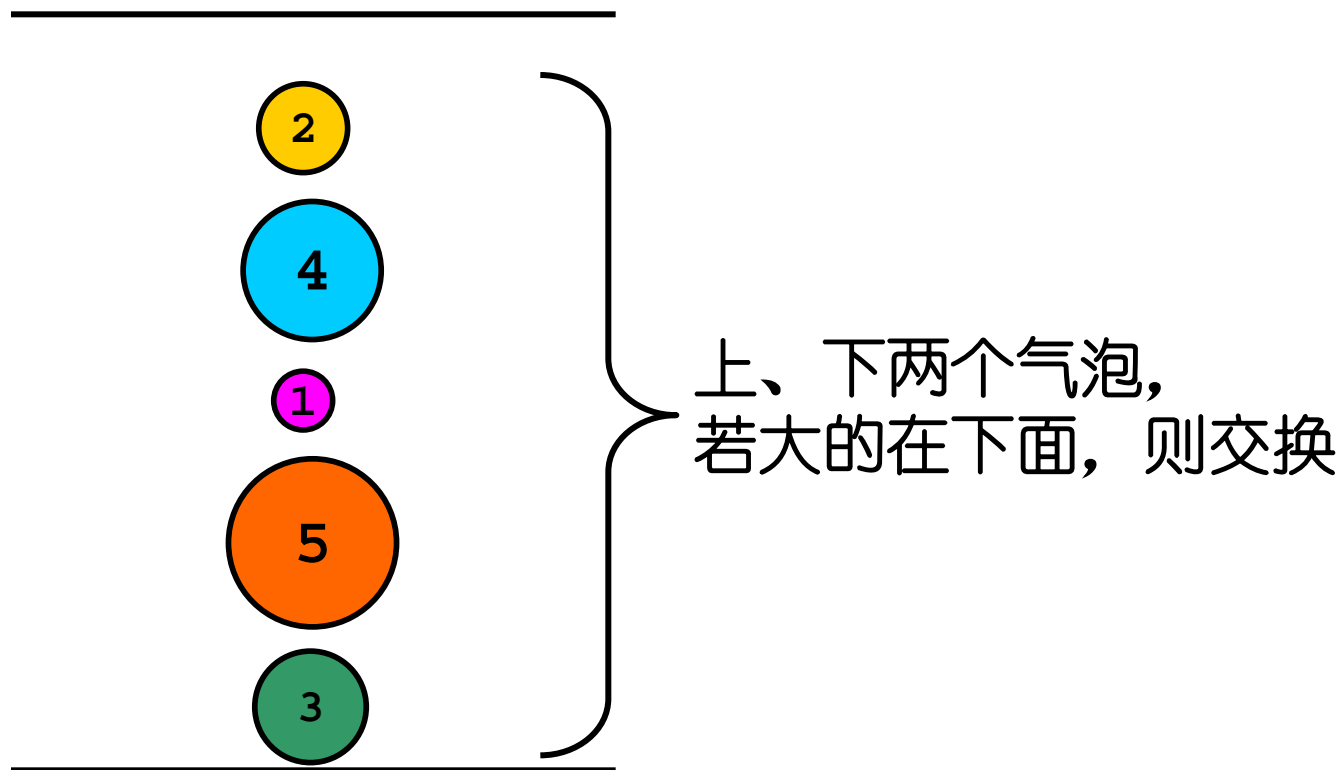


扫描下面2个气泡：  
上、下两个气泡，  
若大的在下面，则交换

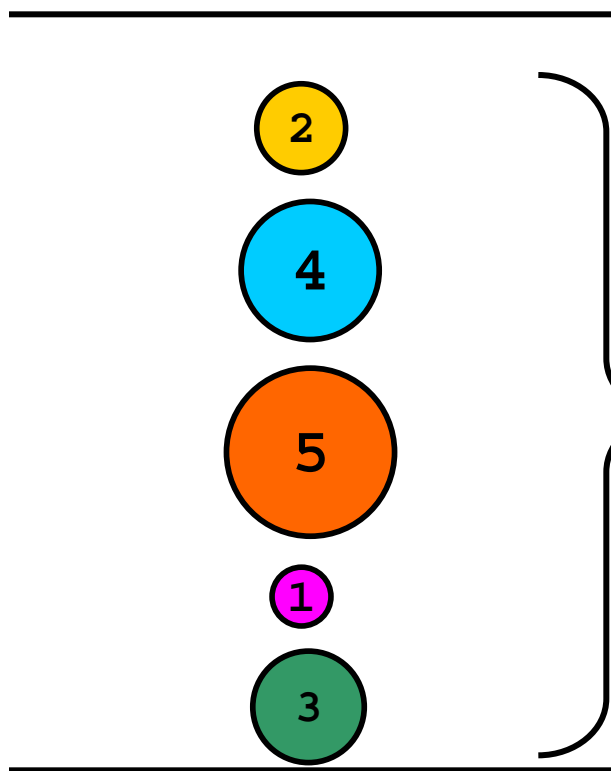


扫描下面2个气泡：  
上、下两个气泡，  
若大的在下面，则交换

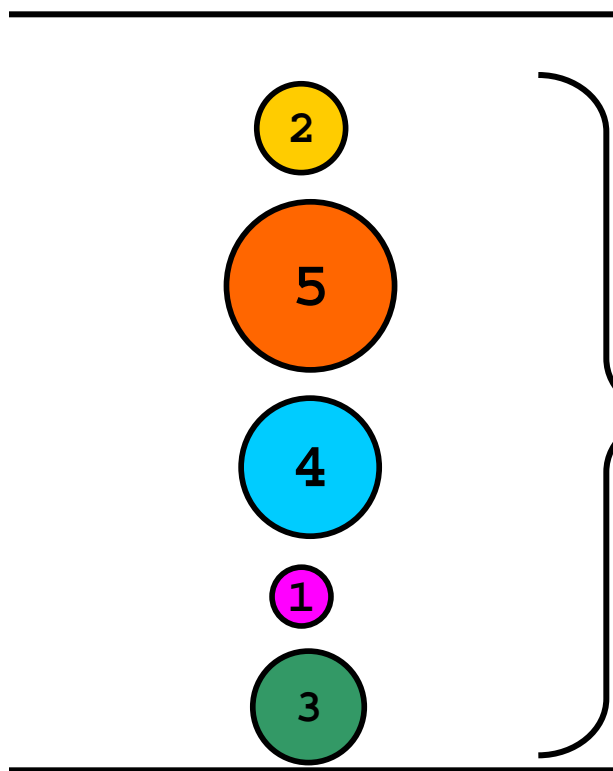
更一般的例子：



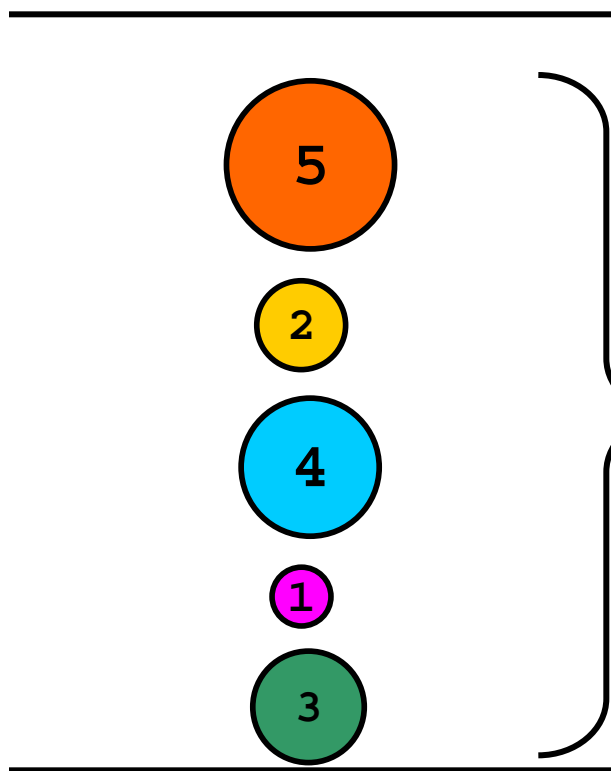




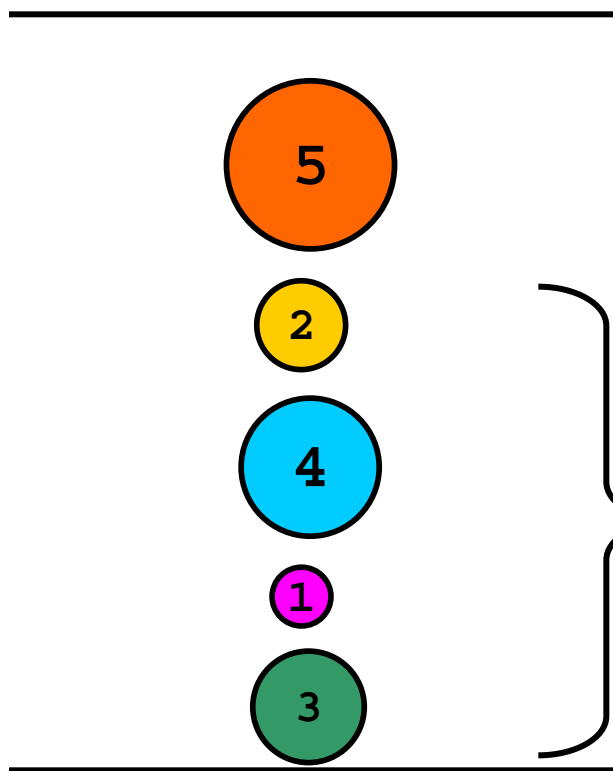
上、下两个气泡，  
若大的在下面，则交换



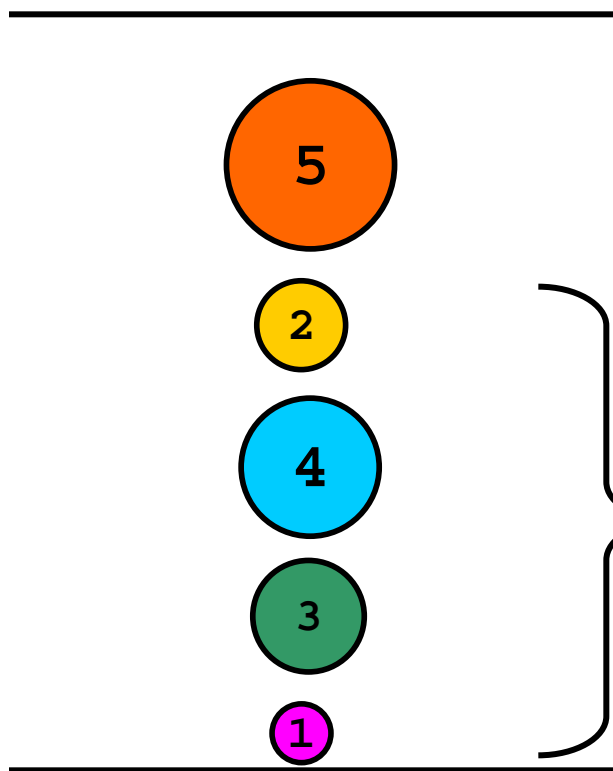
上、下两个气泡，  
若大的在下面，则交换



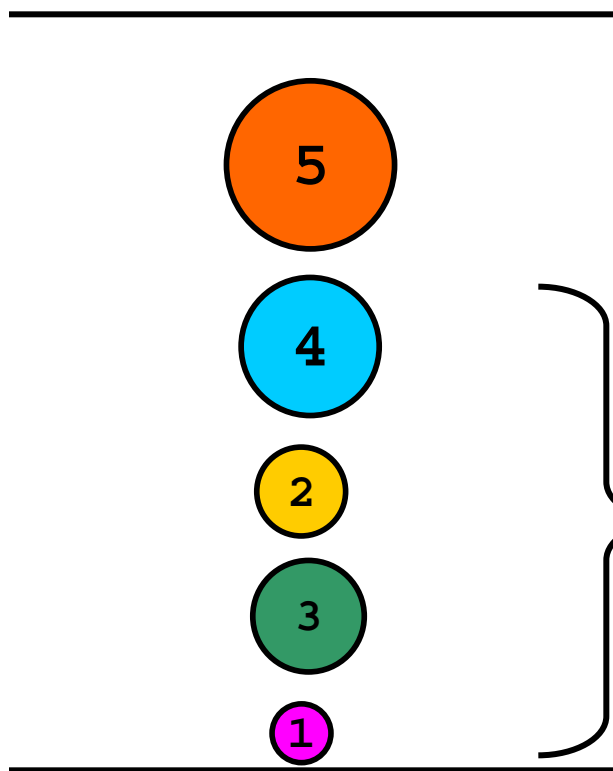
上、下两个气泡，  
若大的在下面，则交换



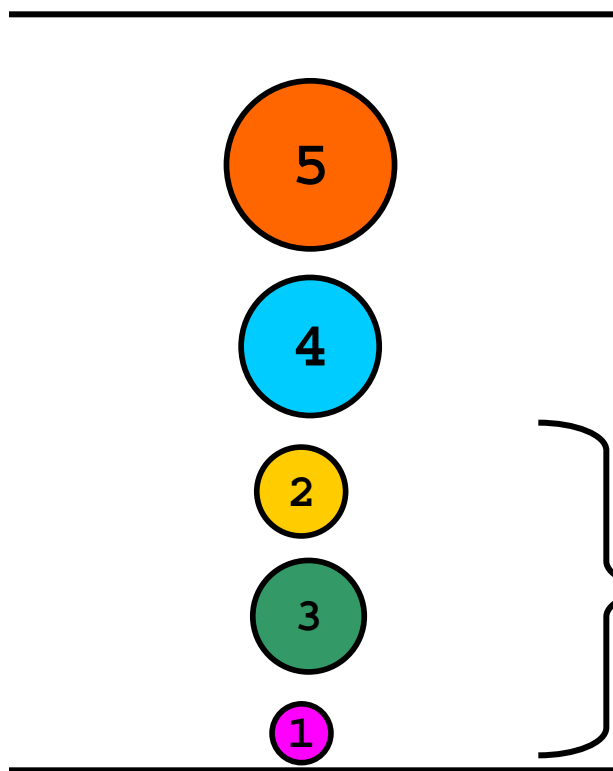
上、下两个气泡，  
若大的在下面，则交换



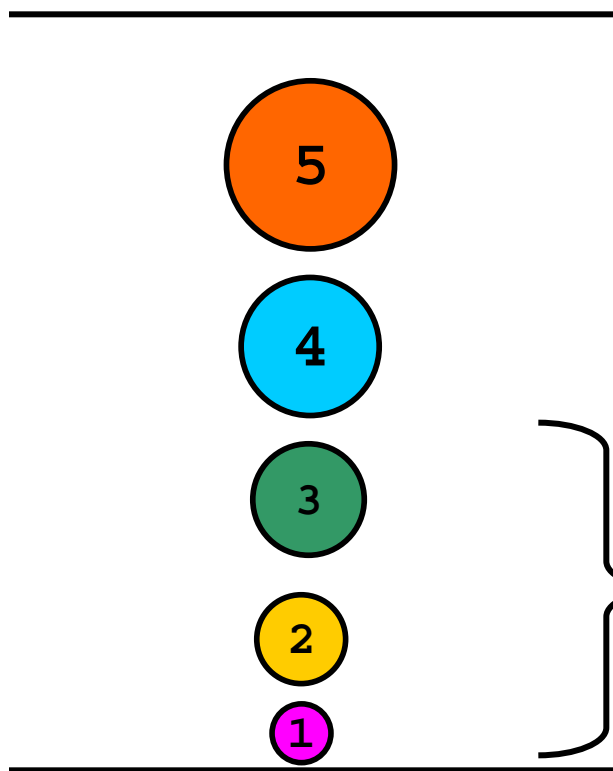
上、下两个气泡，  
若大的在下面，则交换



上、下两个气泡，  
若大的在下面，则交换



上、下两个气泡，  
若大的在下面，则交换



上、下两个气泡，  
若大的在下面，则交换



# 起泡排序法

## • 算法

- 设待排序数据个数为 $n$
- 最多作 $n-1$ 趟扫描： $i = n, \dots, 3, 2$
- 每一趟从前向后： $j = 1 \sim i-1$ 
  - 依次两两比较 $\text{data}[j]$ 和 $\text{data}[j+1]$
  - 若发生逆序，则交换 $\text{data}[j]$ 和 $\text{data}[j+1]$

0	1	2	3	...	n
不用					

# 起泡排序法

- $i=4$

- $j=1$

- 比较data[1]和data[2]

- 不交换

1	2	3	4
37	96	8	54

- $j=2$

- 比较data[2]和data[3]

- 交换

1	2	3	4
37	96	8	54

- $j=3$

- 比较data[3]和data[4]

- 交换

1	2	3	4
37	8	96	54

1	2	3	4
37	8	54	96

# 起泡排序法

- $i=3$

- $j=1$

- 比较data[1]和data[2]

- 交换

1	2	3	4
37	8	54	96

- $j=2$

- 比较data[2]和data[3]

- 不交换

1	2	3	4
8	37	54	96

1	2	3	4
8	37	54	96

# 起泡排序法

- $i=2$

- $j=1$

- 比较 `data[1]` 和 `data[2]`

- 不交换

1	2	3	4
8	37	54	96

1	2	3	4
8	37	54	96

# 起泡排序法

- 时间复杂度

- 最差情况：排序前，所有数据是倒序

- 需要 $n-1$ 趟扫描，即外层循环
    - 第 $i$ 次扫描 ( $i=1, 2, \dots, n-1$ )，需要比较 $n-i$ 次
    - 总共需要比较的次数为

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

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

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

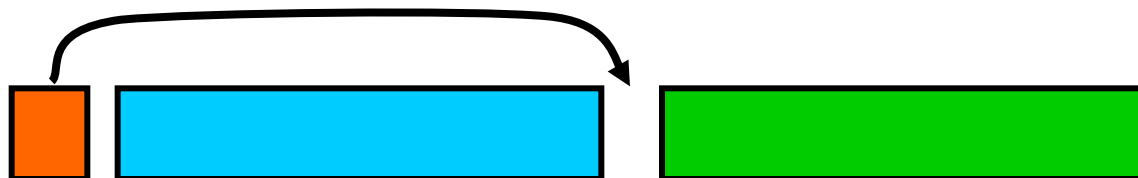
# 起泡排序法

- 最好情况：所有数据已经排好了序
  - 只需要扫描1趟，比较 $n-1$ 次
- 所以起泡排序法的时间复杂度为
  - 最好： $O(n)$
  - 最差： $O(n^2)$
  - 平均： $O(n^2)$
- 空间复杂度
  - $O(1)$ ：交换时需要一个临时变量
- 稳定性：稳定

# 快速排序

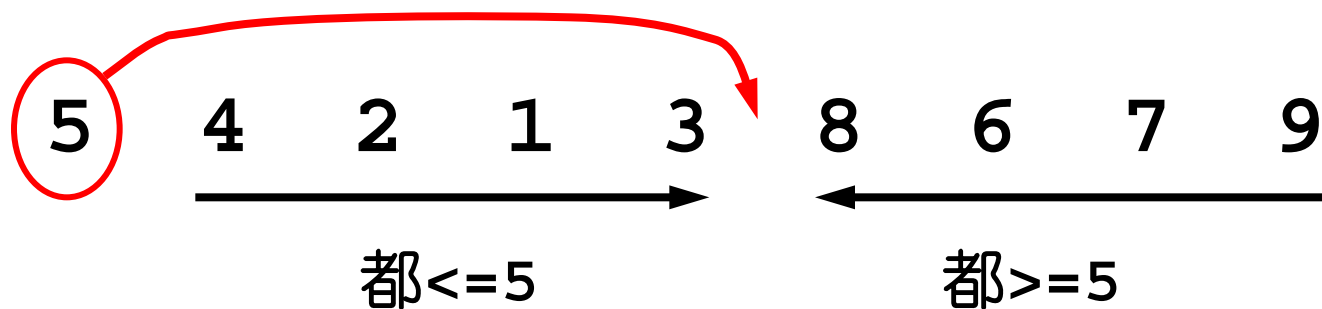
## • 基本思想

- 以某一个数据(例如第一个)作为基准, 将整个序列“划分”为左右两个子序列:
  - 左侧子序列中所有数据都小于等于基准数据
  - 右侧子序列中所有数据都大于等于基准数据
- 这时基准对象就排在这两个子序列中间
- 然后分别对这两个子序列重复施行上述方法, 直到排序完毕



# 快速排序

- 最理想的情况：



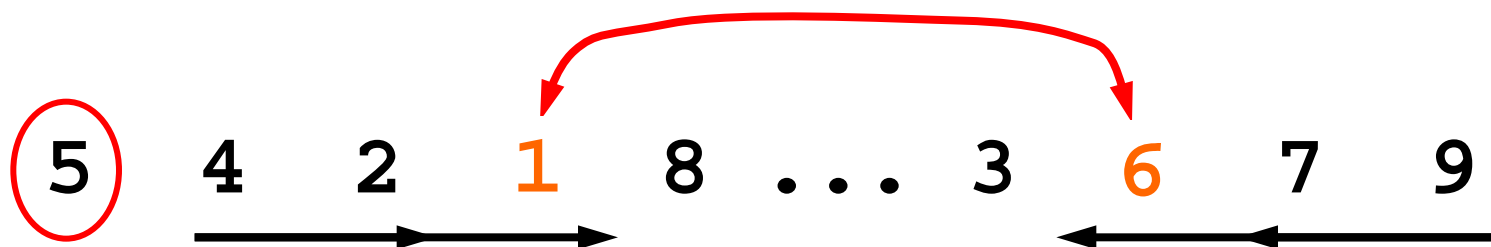
- 疑问

- 运气哪有这么好，总能“一刀切”？
- 为了把一个元素放到正确的位置上，扫描了剩下的所有元素，复杂度仍然是 $O(n^2)$ ，“快速”在哪里？



# 快速排序

- 一般情况：



- 遇到“障碍”交换之，两箭头总能相遇
- 在正确放置一个元素的同时，交换了几对乱序的元素，复杂度肯定  $< O(n^2)$

# 快速排序

- 递归函数

```
void QSort(SqList &L, int low, int high) {  
    if(low < high) {        //待排序数列长度大于1  
        pivotloc = Partition(L, low, high);  
        //对左子序列进行排序  
        QSort(L, low, pivotloc - 1);  
        //对右子序列进行排序  
        QSort(L, pivotloc + 1, high);  
    }  
}
```

## • 分割函数1

```
int Partition(int data[], int low, int high){
    int pivot      = low;          // 以最左元素为中轴
    int pivotvalue = data[low];    // 记录中轴的值
    while(low < high) {
        while(low<high && data[high] >= pivotvalue)
            high --; // high向左, 直到遇见比pivot小的
        while(low<high && data[low] <= pivotvalue)
            low ++;  // low向右, 直到遇见比pivot大的
        // low和high扫描受阻, 交换low和high的值
        Swap(&data[low], &data[high]);
    }
    // 交换中轴和low的值 (也就是把中轴放置到正确的位置上)
    Swap(&data[pivot], &data[low]);
    return low;
}
```

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low								high	
↓								↓	
1	2	3	4	5	6	7	8		
49	38	65	97	76	13	27	49'		

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low		high						
↓		↓						
1	2	3	4	5	6	7	8	
49	38	65	97	76	13	27	49'	

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low		high						
↓		↓						
1	2	3	4	5	6	7	8	
49	38	65	97	76	13	27	49'	

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low			high				
↓			↓				
1	2	3	4	5	6	7	8
49	38	65	97	76	13	27	49'

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low			high				
↓			↓				
1	2	3	4	5	6	7	8
49	38	27	97	76	13	65	49'

↑  
pivot



```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

		low				high			
		↓				↓			
1	2	3	4	5	6	7	8		
49	38	27	97	76	13	65	49'		

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low				high			
↓				↓			
1	2	3	4	5	6	7	8
49	38	27	97	76	13	65	49'

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

		low			high				
		↓			↓				
1	2	3	4	5	6	7	8		
49	38	27	13	76	97	65	49'		

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low				high			
↓				↓			
1	2	3	4	5	6	7	8
49	38	27	13	76	97	65	49'

↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low high							
1	2	3	4	5	6	7	8
49	38	27	13	76	97	65	49'

↑ pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

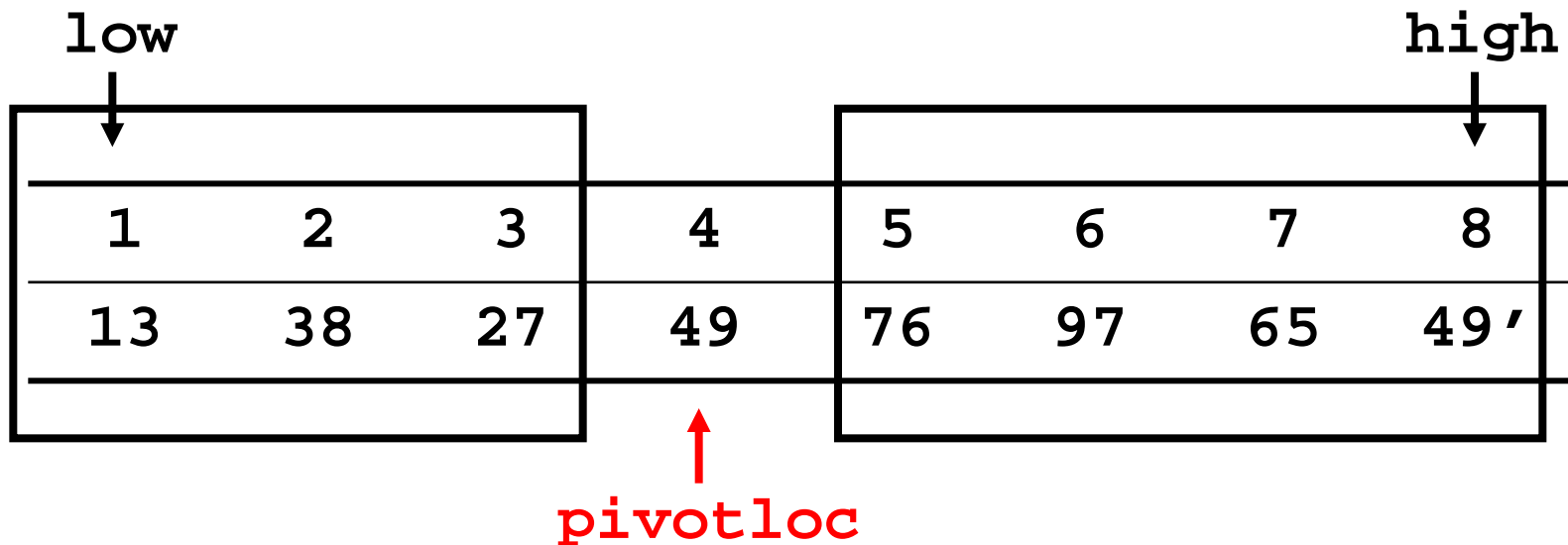
low high							
1	2	3	4	5	6	7	8
13	38	27	49	76	97	65	49'

↑  
pivot

```

void QSort(SqList &L, int low, int high) {
    if(low < high) {        //待排序数列长度大于1
        pivotloc = Partition(L, low, high);
        //对左子序列进行排序
        QSort(L, low, pivotloc - 1);
        //对右子序列进行排序
        QSort(L, pivotloc + 1, high);
    }
}

```



```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low		high
↓		↓
1	2	3
13	38	27

↑  
pivot



```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low	high	
↓	↓	
1	2	3
13	38	27

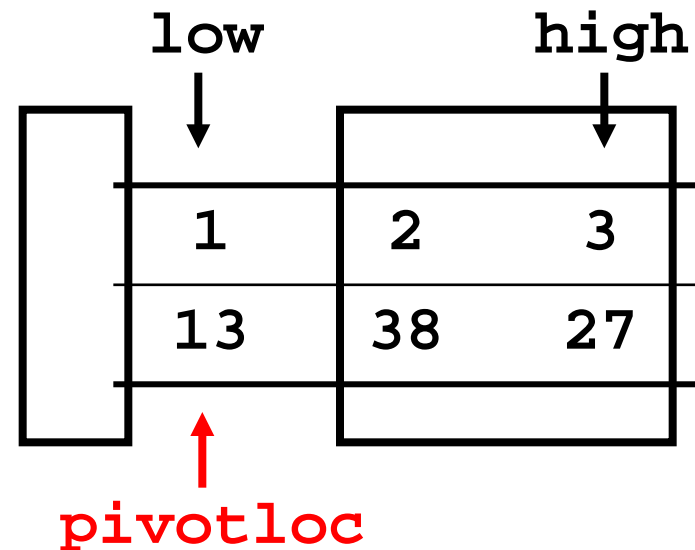
↑  
pivot

```
while(low < high) {  
    while(low<high && data[high] >= pivotvalue)  
        high --;    // high向左, 直到遇见比pivot小的  
    while(low<high && data[low] <= pivotvalue)  
        low ++;    // low向右, 直到遇见比pivot大的  
    // low和high扫描受阻, 交换low和high的值  
    Swap(&data[low], &data[high]);  
}  
// 交换中轴和low的值 (也就是把中轴放置到正确的位置上)  
Swap(&data[pivot], &data[low]);  
return low;
```

low high		
↓	↙	
1	2	3
13	38	27

↑  
pivot

```
void QSort(SqList &L, int low, int high) {  
    if(low < high) { //待排序数列长度大于1  
        pivotloc = Partition(L, low, high);  
        //对左子序列进行排序  
        QSort(L, low, pivotloc - 1);  
        //对右子序列进行排序  
        QSort(L, pivotloc + 1, high);  
    }  
}
```

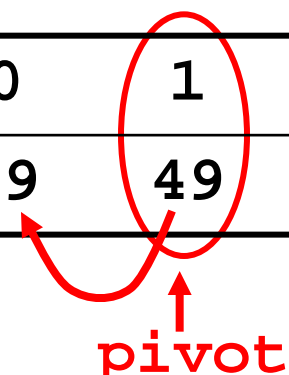


- 分割算法2 (P274, 算法10.6b)

```
L.r[0] = L.r[low];      //把最左元素当作基准
while(low < high){
    //high向左, 直到遇见比pivot小的
    while(low < high && L.r[high].key>=pivotkey)
        high --;
    L.r[low] = L.r[high];
    //low向右, 直到遇见比pivot大的
    while(low<high && L.r[low].key<=pivotkey)
        low ++;
    L.r[high] = L.r[low];
}
L.r[low] = L.r[0];
return low;
```

```
L.r[0] = L.r[low];           //把最左元素当作基准  
while(low < high){  
    while(low < high && L.r[high].key>=pivotkey)  
        high --; //high向左, 直到遇见比pivot小的  
    L.r[low] = L.r[high];  
    while(low<high && L.r[low].key<=pivotkey)  
        low ++; //low向右, 直到遇见比pivot大的  
    L.r[high] = L.r[low];  
}  
L.r[low] = L.r[0];  return low;
```

low									high
↓									↓
0	1	2	3	4	5	6	7	8	
49	49	38	65	97	76	13	27	49'	



pivot

```
L.r[0] = L.r[low];           //把最左元素当作基准
while(low < high){
    while(low < high && L.r[high].key>=pivotkey)
        high --; //high向左, 直到遇见比pivot小的
    L.r[low] = L.r[high];
    while(low<high && L.r[low].key<=pivotkey)
        low ++; //low向右, 直到遇见比pivot大的
    L.r[high] = L.r[low];
}
L.r[low] = L.r[0];  return low;
```

low									high	
↓									↓	
0	1	2	3	4	5	6	7	8		
49	27	38	65	97	76	13	27	49'		

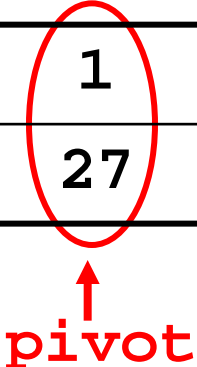
↑  
pivot

```

L.r[0] = L.r[low];           //把最左元素当作基准
while(low < high){
    while(low < high && L.r[high].key>=pivotkey)
        high --; //high向左, 直到遇见比pivot小的
    L.r[low] = L.r[high];
    while(low<high && L.r[low].key<=pivotkey)
        low ++; //low向右, 直到遇见比pivot大的
    L.r[high] = L.r[low];
}
L.r[low] = L.r[0];  return low;

```

		low ↓						high ↓	
0	1	2	3	4	5	6	7	8	
49	27	38	65	97	76	13	27	49'	



```

L.r[0] = L.r[low];           //把最左元素当作基准
while(low < high){
    while(low < high && L.r[high].key>=pivotkey)
        high --; //high向左, 直到遇见比pivot小的
    L.r[low] = L.r[high];
    while(low<high && L.r[low].key<=pivotkey)
        low ++; //low向右, 直到遇见比pivot大的
    L.r[high] = L.r[low];
}
L.r[low] = L.r[0];  return low;

```

			low ↓					high ↓	
0	1	2	3	4	5	6	7	8	
49	27	38	65	97	76	13	65	49'	

↑  
pivot



```

L.r[0] = L.r[low];           //把最左元素当作基准
while(low < high){
    while(low < high && L.r[high].key>=pivotkey)
        high --; //high向左, 直到遇见比pivot小的
    L.r[low] = L.r[high];
    while(low<high && L.r[low].key<=pivotkey)
        low ++; //low向右, 直到遇见比pivot大的
    L.r[high] = L.r[low];
}
L.r[low] = L.r[0];  return low;

```

			low				high		
			↓				↓		
0	1	2	3	4	5	6	7	8	
49	27	38	13	97	76	13	65	49'	

↑  
pivot

```
L.r[0] = L.r[low];           //把最左元素当作基准
while(low < high){
    while(low < high && L.r[high].key>=pivotkey)
        high --; //high向左, 直到遇见比pivot小的
    L.r[low] = L.r[high];
    while(low<high && L.r[low].key<=pivotkey)
        low ++; //low向右, 直到遇见比pivot大的
    L.r[high] = L.r[low];
}
L.r[low] = L.r[0];  return low;
```

				low					high		
				↓					↓		
0	1	2	3	4	5	6	7	8			
49	27	38	13	97	76	97	65	49'			

↑  
pivot

```

L.r[0] = L.r[low];           //把最左元素当作基准
while(low < high){
    while(low < high && L.r[high].key>=pivotkey)
        high --; //high向左, 直到遇见比pivot小的
    L.r[low] = L.r[high];
    while(low<high && L.r[low].key<=pivotkey)
        low ++; //low向右, 直到遇见比pivot大的
    L.r[high] = L.r[low];
}
L.r[low] = L.r[0];  return low;

```

				low	high			
				↓	↓			
0	1	2	3	4	5	6	7	8
49	27	38	13	97	76	97	65	49'

↑  
pivot



# 快速排序

- 时间复杂度

- 最好情况和平均情况： $k n \log_2 n$

- 排序前数据杂乱无章

- 系数 $k$ 是同数量级的排序算法中最小的

- 最差情况： $O(n^2)$

- 排序前，数据已经排好序，或基本排好序

- 简单理解：快速排序之所以快，就在于在正确放置中轴的同时，能够对多对乱序元素作交换，如果都已经排好序，这个优势就发挥不出来了

# 快速排序

- 最差情况： $O(n^2)$

- 排序前，数据已经排好序
- 每次划分只得到一个比上一次少一个对象的子序列，必须经过 $n-1$ 趟才能把所有对象定位
- 而且第 $i$ 趟需要经过 $n-i$ 次比较才能找到第 $i$ 个对象的安放位置
- 所以总的比较次数=

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

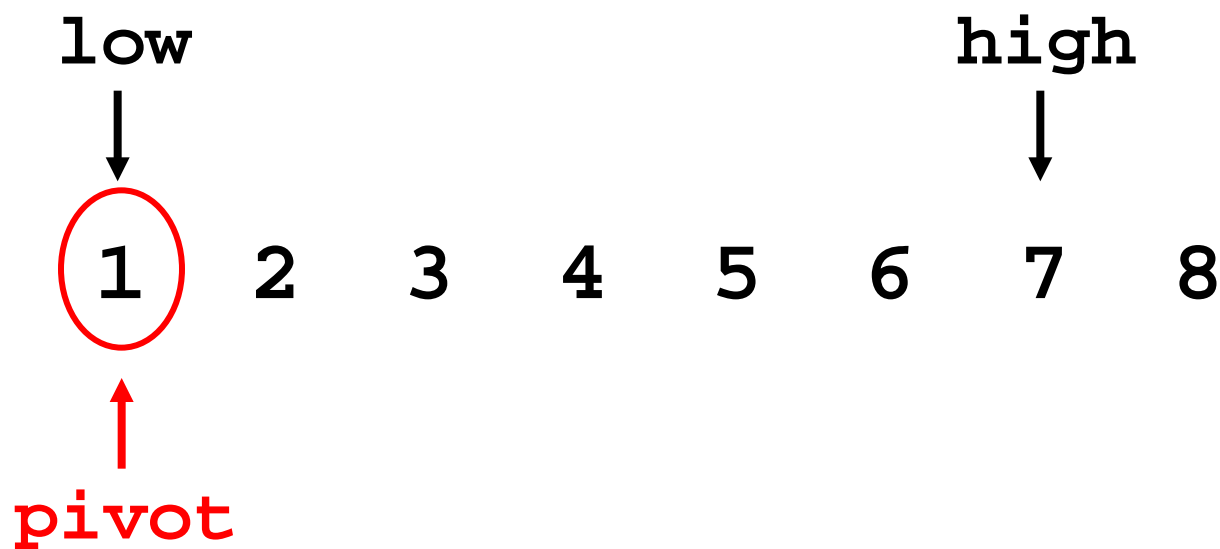
# 快速排序

- 例



# 快速排序

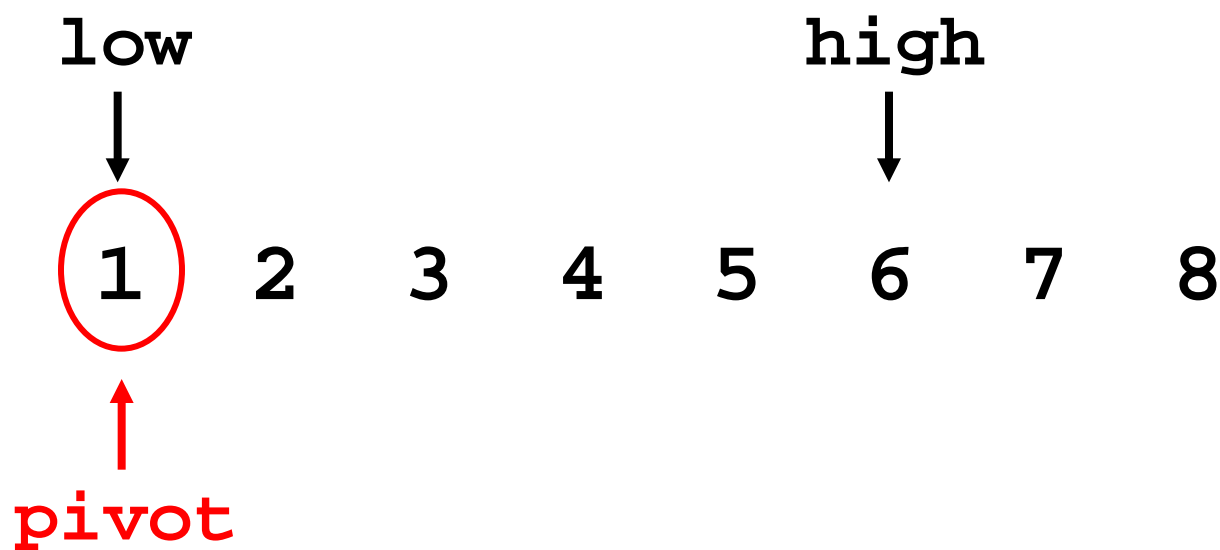
- 例





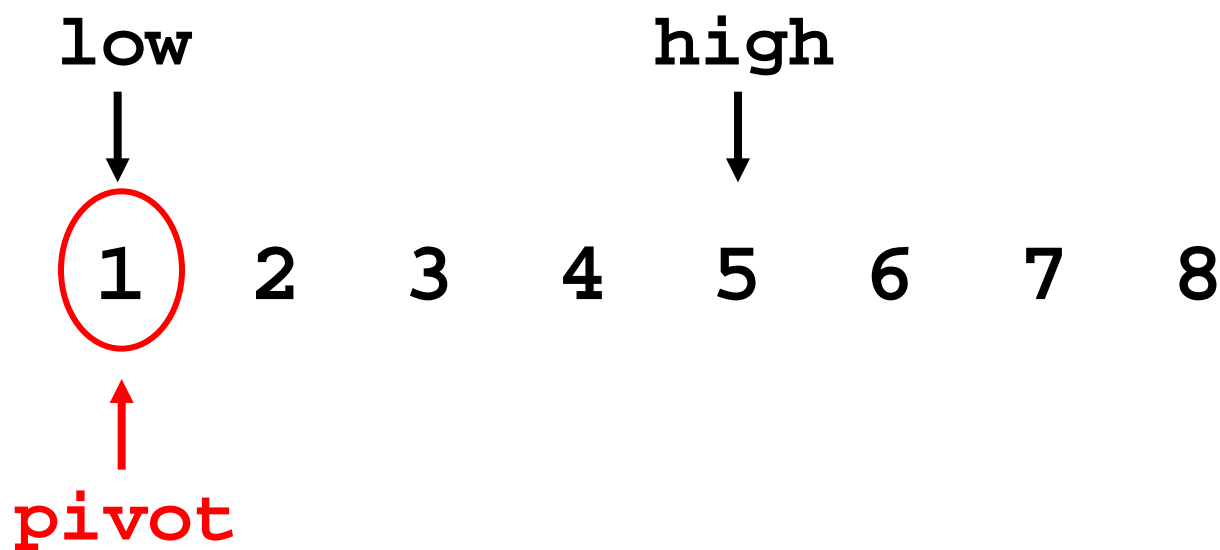
# 快速排序

- 例



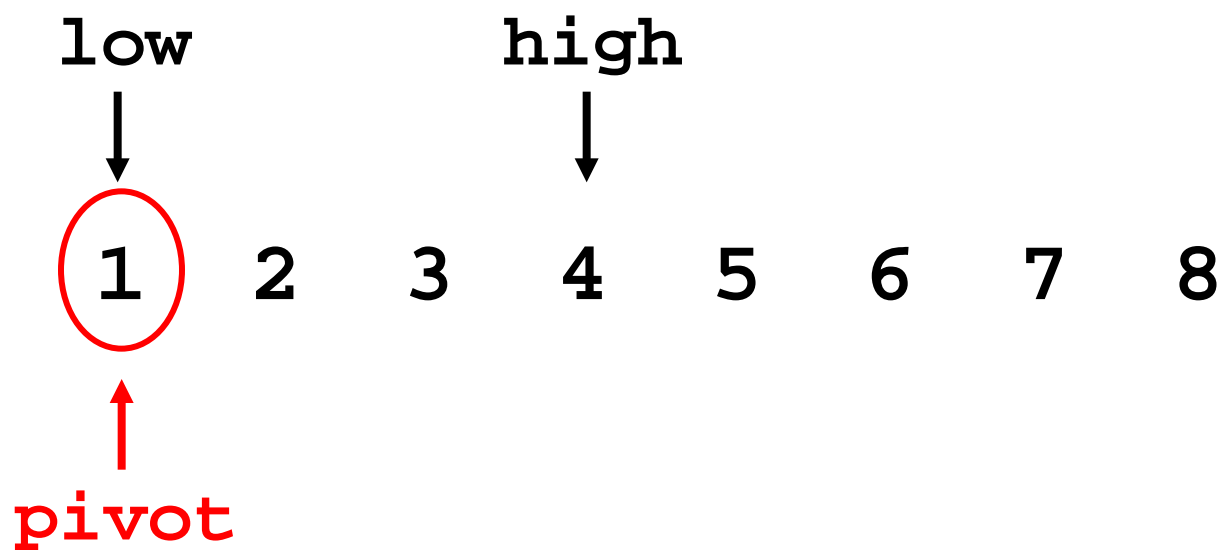
# 快速排序

- 例



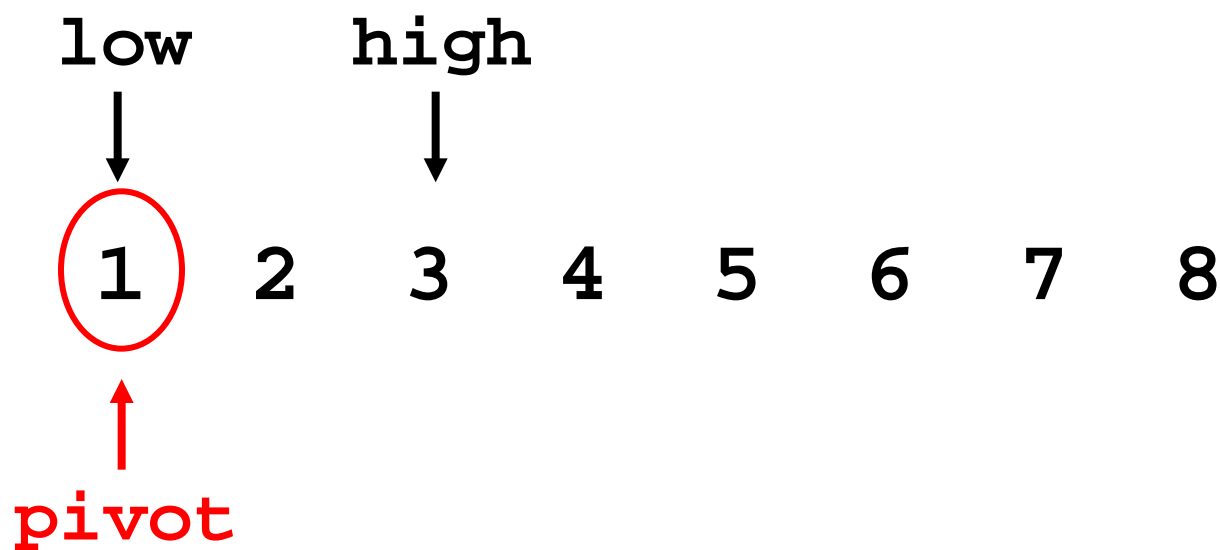
# 快速排序

- 例



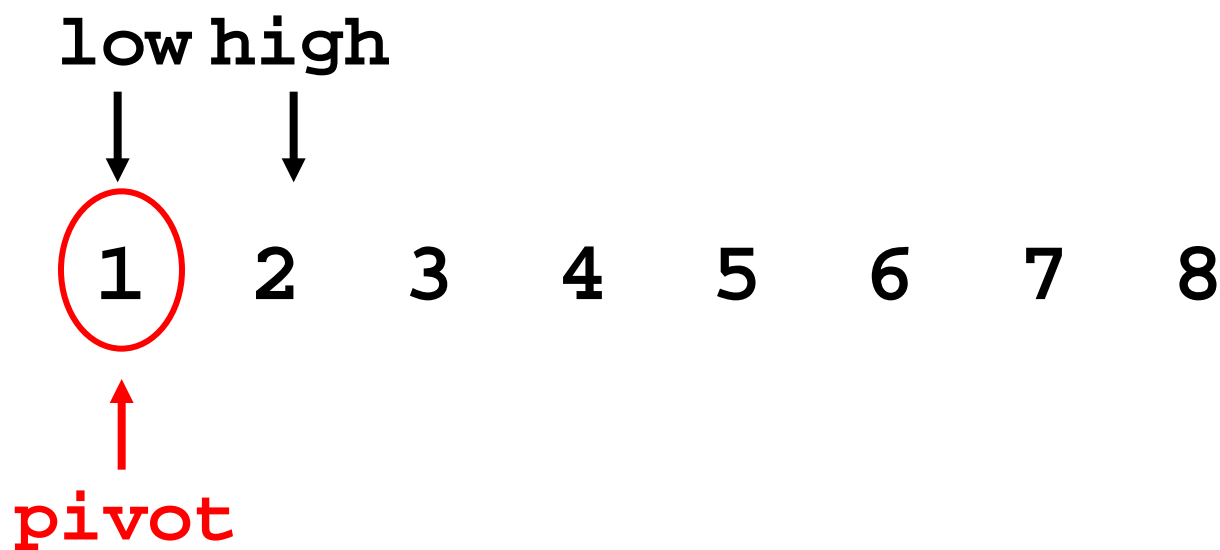
# 快速排序

- 例



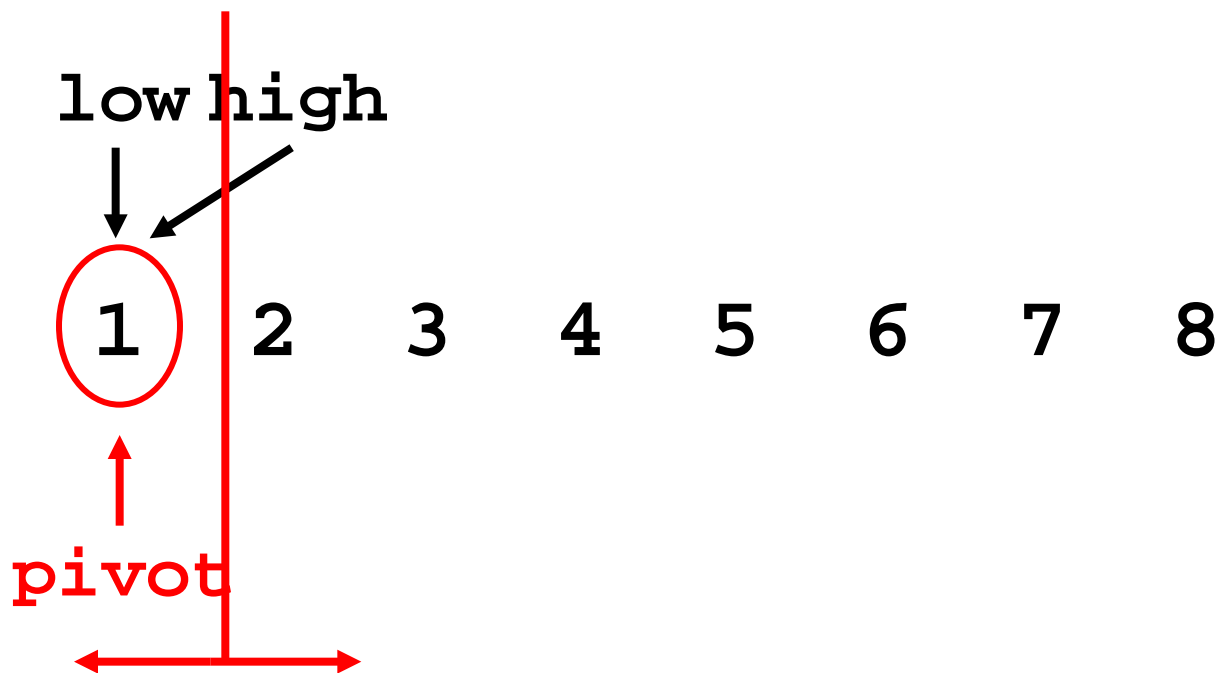
# 快速排序

- 例



# 快速排序

- 例

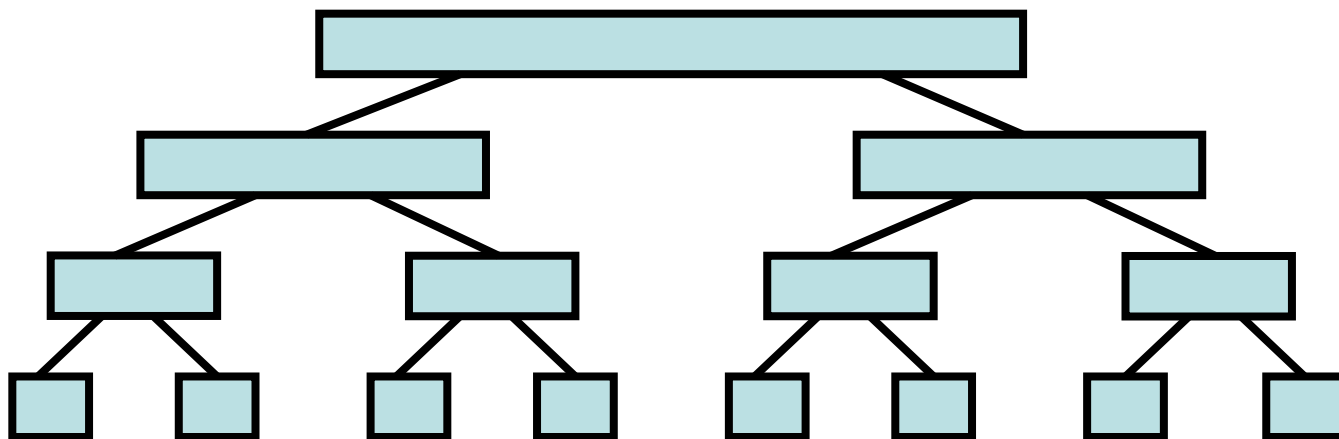


- 另外：在元素数量很少时效果不好

# 快速排序

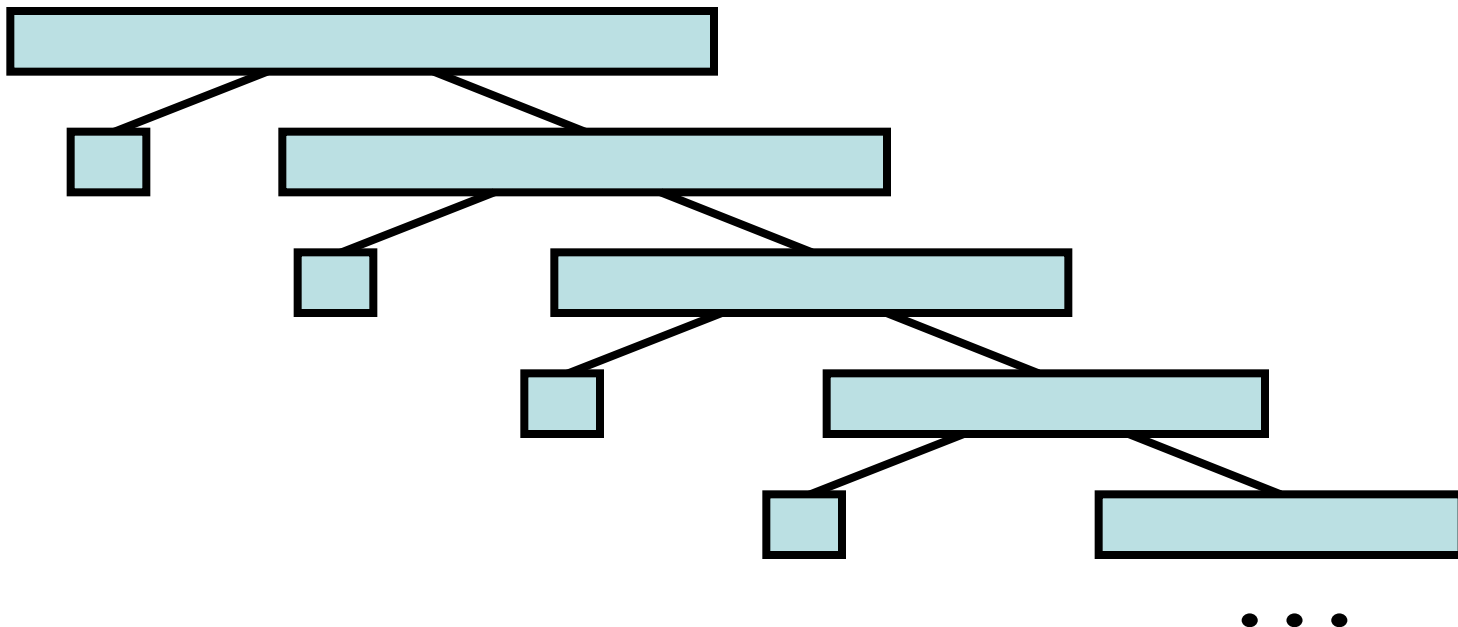
- 空间复杂度

- 使用了递归，相当于增加了一个堆栈
- 堆栈的深度 = 递归的层数
  - 最少 $\log_2 n$ ：每一次都切在中间



# 快速排序

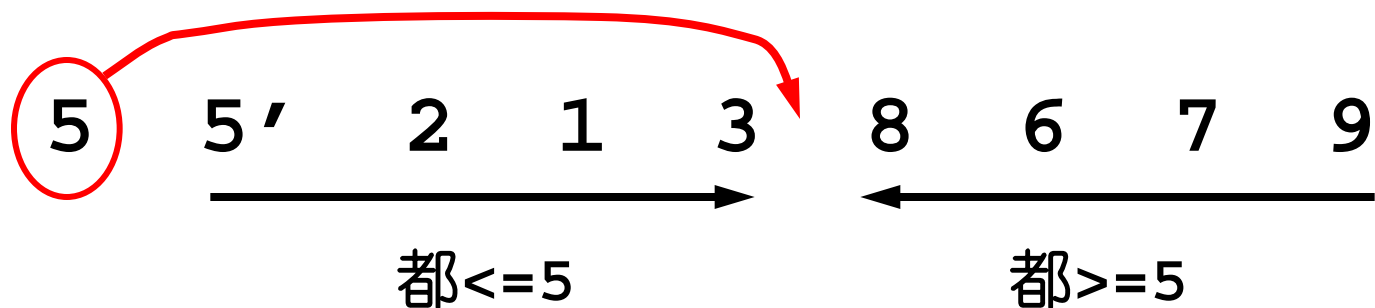
- 最多 $n$ : 每次都只分出一个元素





# 快速排序

- 不稳定:



# 选择排序

- 基本思想

- $n$ 个元素  $(1, 2, 3, \dots, n)$
- 第 $i$ 趟扫描 ( $i=1, \dots, n-1$ )，扫描第 $i$ 到 $n$ 的元素，找到这 $n-i+1$ 个元素中最小的，放到第 $i$ 个位置上
- 简单选择排序
- 堆排序

# 简单选择排序

- 算法

```
void SelectSort(int data[], int number) {  
    for(i = 1; i <= number; i++) {  
        // 找到i~number中最小的一个IndexMin  
        IndexMin = i;  
        for(j = i; j <= number; j++)  
            if(data[j] < data[IndexMin])  
                IndexMin = j;  
        // 把IndexMin和i作交换  
        if(IndexMin != i)  
            Swap(&data[IndexMin], &data[i]);  
    }  
}
```

```
for(i = 1; i <= number; i ++){  
    // 找到i~number中最小的一个IndexMin  
    IndexMin = i;  
    for(j = i; j <= number; j ++)  
        if(data[j] < data[IndexMin])  
            IndexMin = j;  
    // 把IndexMin和i作交换  
    if(IndexMin != i)  
        Swap(&data[IndexMin], &data[i]);  
}
```

<b>i</b> ↓				
1	2	3	4	5
<hr/>				
28	41	36	7	16
↑ <b>j</b>				

MinValue = 28

IndexMin = 1

```
for(i = 1; i <= number; i ++){
    // 找到i~number中最小的一个IndexMin
    IndexMin = i;
    for(j = i; j <= number; j ++){
        if(data[j] < data[IndexMin])
            IndexMin = j;
    }
    // 把IndexMin和i作交换
    if(IndexMin != i)
        Swap(&data[IndexMin], &data[i]);
}
```

<b>i</b> ↓				
1	2	3	4	5
<hr/>				
28	41	36	7	16
	↑ <b>j</b>			

MinValue = 28

IndexMin = 1

```
for(i = 1; i <= number; i ++){
    // 找到i~number中最小的一个IndexMin
    IndexMin = i;
    for(j = i; j <= number; j ++){
        if(data[j] < data[IndexMin])
            IndexMin = j;
    }
    // 把IndexMin和i作交换
    if(IndexMin != i)
        Swap(&data[IndexMin], &data[i]);
}
```

<b>i</b> ↓				
1	2	3	4	5
<hr/>				
28	41	36	7	16
		↑ <b>j</b>		

MinValue = 28

IndexMin = 1

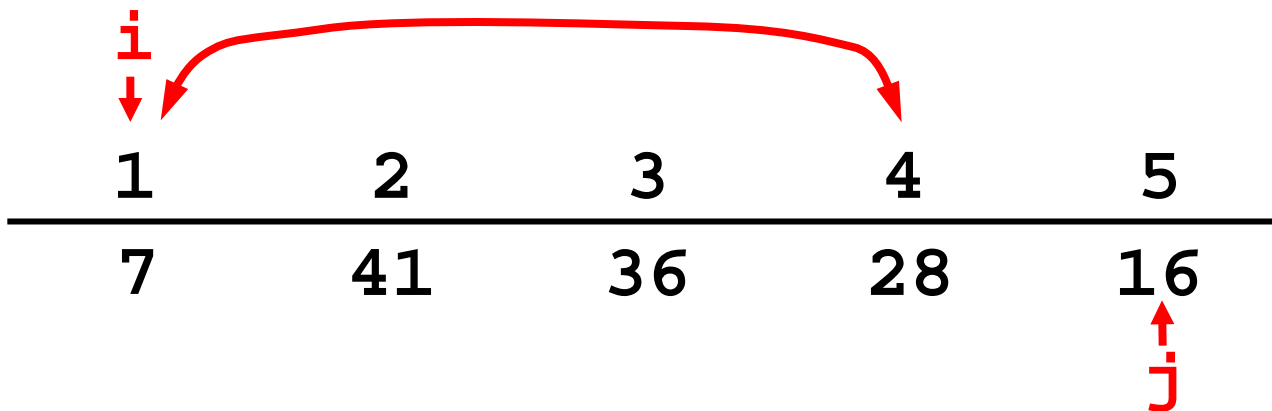
```
for(i = 1; i <= number; i ++) {  
    // 找到i~number中最小的一个IndexMin  
    IndexMin = i;  
    for(j = i; j <= number; j ++)  
        if(data[j] < data[IndexMin])  
            IndexMin = j;  
    // 把IndexMin和i作交换  
    if(IndexMin != i)  
        Swap(&data[IndexMin], &data[i]);  
}
```

<b>i</b> ↓				
1	2	3	4	5
<hr/>				
28	41	36	7	16
			↑ <b>j</b>	

MinValue = 7

IndexMin = 4

```
for(i = 1; i <= number; i ++) {  
    // 找到i~number中最小的一个IndexMin  
    IndexMin = i;  
    for(j = i; j <= number; j ++)  
        if(data[j] < data[IndexMin])  
            IndexMin = j;  
    // 把IndexMin和i作交换  
    if(IndexMin != i)  
        Swap(&data[IndexMin], &data[i]);  
}
```



MinValue = 7

IndexMin = 4



```
for(i = 1; i <= number; i ++) {  
    // 找到i~number中最小的一个IndexMin  
    IndexMin = i;  
    for(j = i; j <= number; j ++)  
        if(data[j] < data[IndexMin])  
            IndexMin = j;  
    // 把IndexMin和i作交换  
    if(IndexMin != i)  
        Swap(&data[IndexMin], &data[i]);  
}
```

	<b>i</b> ↓			
1	2	3	4	5
<hr/>				
7	41	36	28	16
	↑ <b>j</b>			

MinValue = 41

IndexMin = 2

```
for(i = 1; i <= number; i ++) {  
    // 找到i~number中最小的一个IndexMin  
    IndexMin = i;  
    for(j = i; j <= number; j ++)  
        if(data[j] < data[IndexMin])  
            IndexMin = j;  
    // 把IndexMin和i作交换  
    if(IndexMin != i)  
        Swap(&data[IndexMin], &data[i]);  
}
```

	<b>i</b> ↓			
1	2	3	4	5
7	41	36	28	16
		↑ <b>j</b>		

MinValue = 36

IndexMin = 3

```
for(i = 1; i <= number; i ++) {  
    // 找到i~number中最小的一个IndexMin  
    IndexMin = i;  
    for(j = i; j <= number; j ++)  
        if(data[j] < data[IndexMin])  
            IndexMin = j;  
    // 把IndexMin和i作交换  
    if(IndexMin != i)  
        Swap(&data[IndexMin], &data[i]);  
}
```

	<b>i</b> ↓			
1	2	3	4	5
<hr/>				
7	41	36	28	16
			↑ <b>j</b>	

MinValue = 28

IndexMin = 4

```
for(i = 1; i <= number; i ++) {  
    // 找到i~number中最小的一个IndexMin  
    IndexMin = i;  
    for(j = i; j <= number; j ++)  
        if(data[j] < data[IndexMin])  
            IndexMin = j;  
    // 把IndexMin和i作交换  
    if(IndexMin != i)  
        Swap(&data[IndexMin], &data[i]);  
}
```

1	2	3	4	5
7	16	36	28	41

MinValue = 16

IndexMin = 5

# 简单选择排序

- 时间复杂度

- 数据移动的次數：

- 当元素已经排好序时，不需要移动数据

- 当元素是逆序时，需要移动 $3(n-1)$ 次

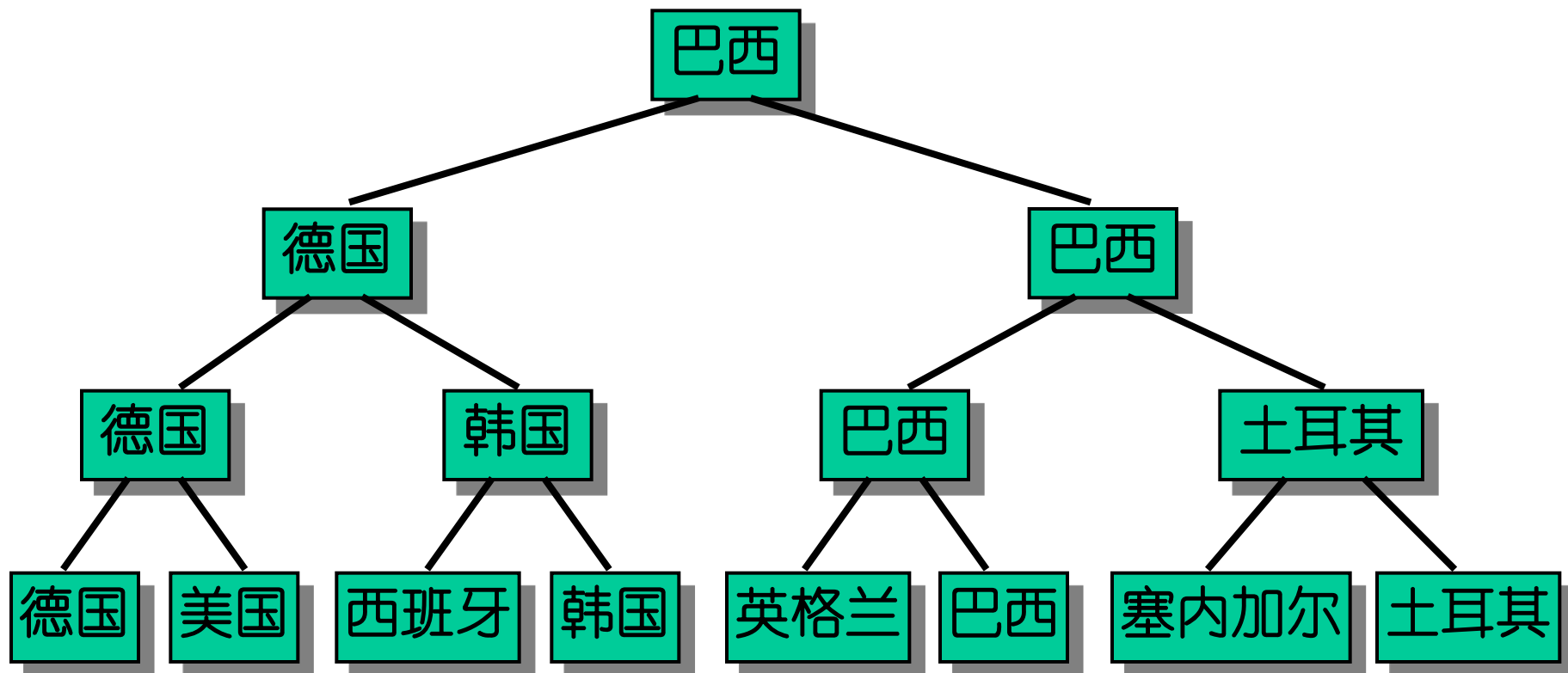
- 数据比较次数： $n(n-1)/2$

- 所以不论什么情况，时间复杂度都是 $O(n^2)$

- 空间复杂度： $O(1)$

- 稳定性：不稳定

# 2002年日韩世界杯淘汰赛8强

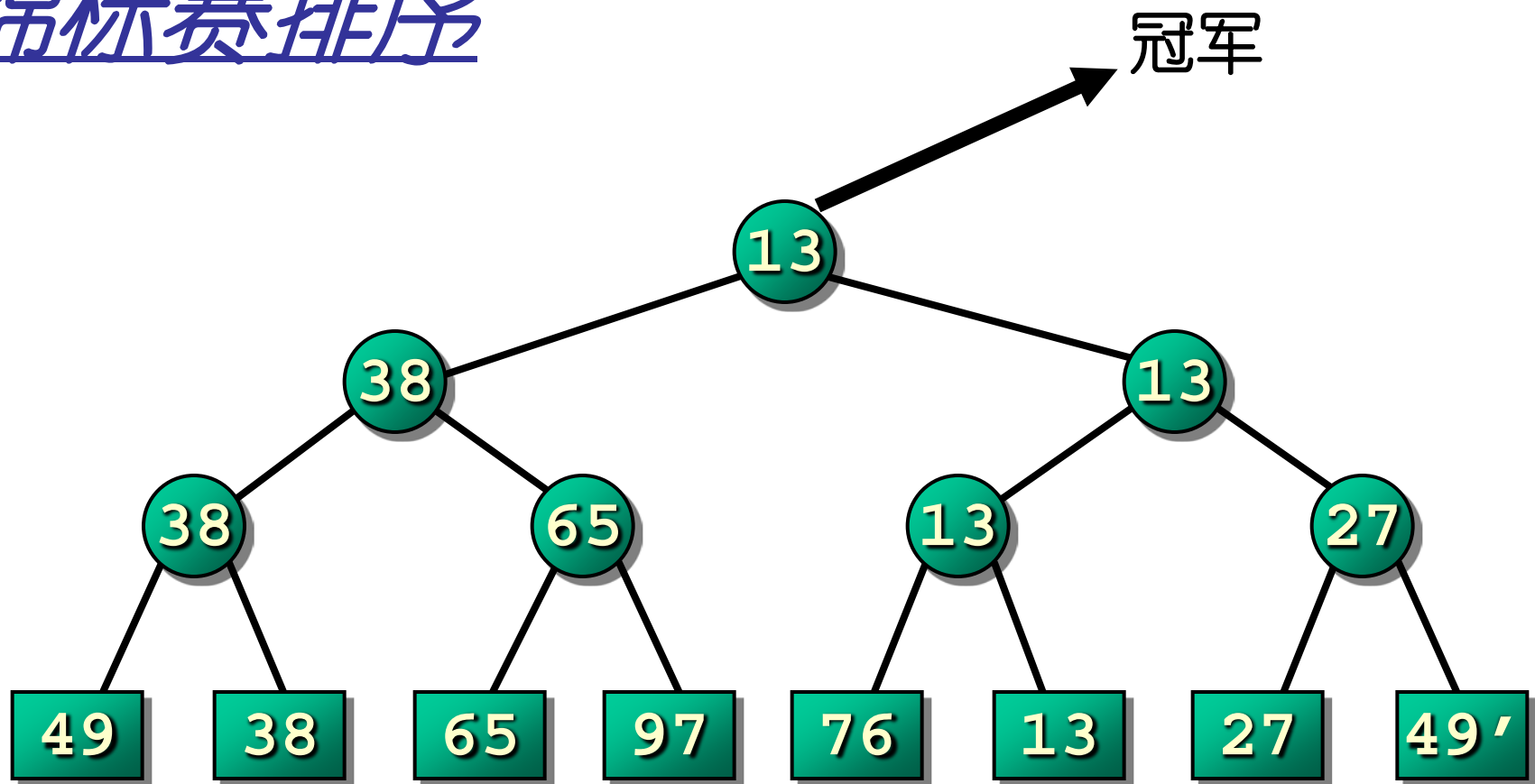


# 锦标赛排序

- 基本思想

- 类似体育比赛时的淘汰赛
- 首先取得 $n$ 个对象的排序码，两两比较，得到  $\lfloor n/2 \rfloor$  个比较的优胜者(排序码小者)，作为第一步比较的结果保留下来
- 然后对这  $\lfloor n/2 \rfloor$  个对象再进行排序码的两两比较，...
- 重复，直到选出一个排序码最小的对象为止

# 锦标赛排序



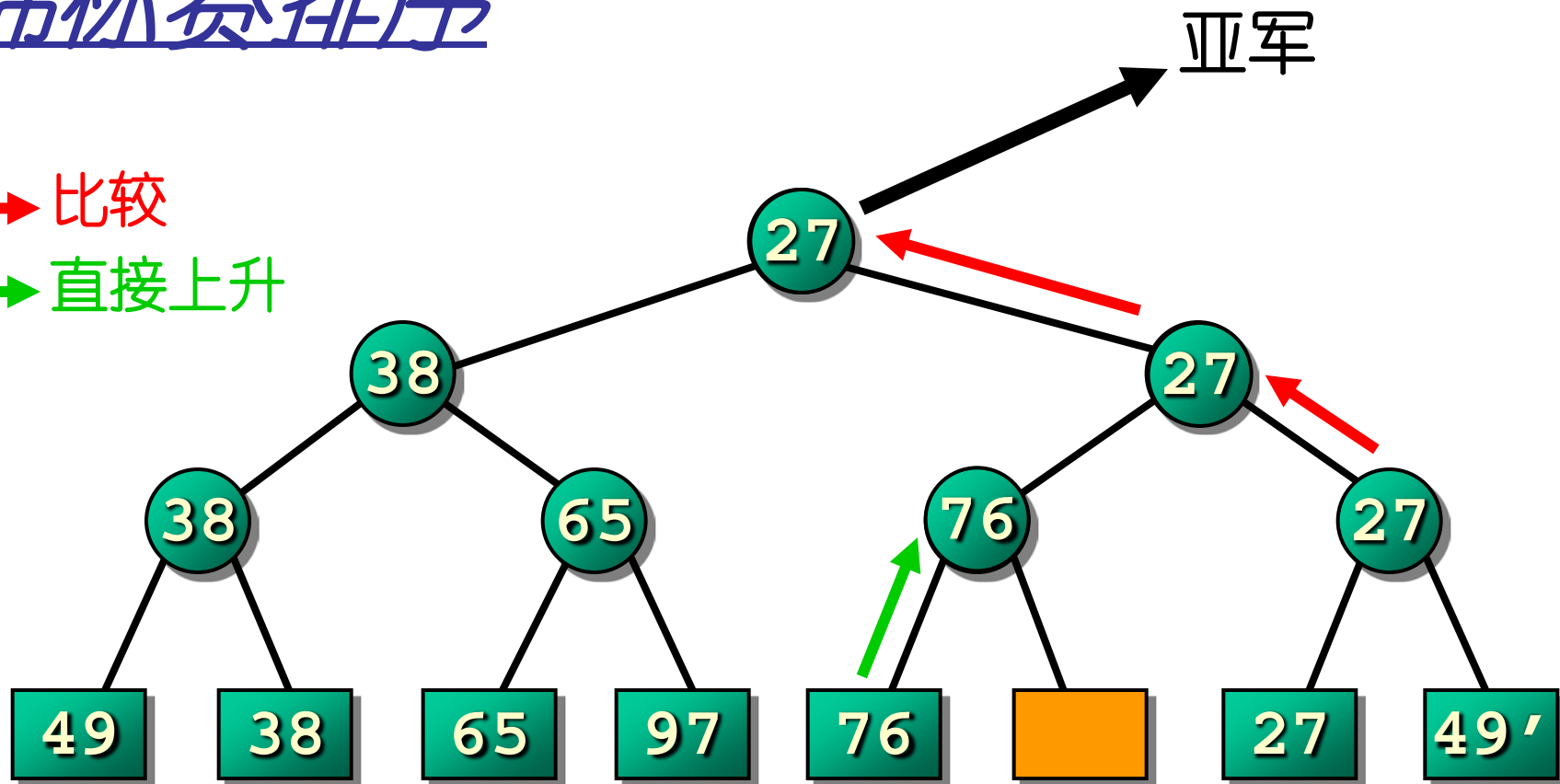
- 比较次数 = 7
- 输出冠军



# 锦标赛排序

→ 比较

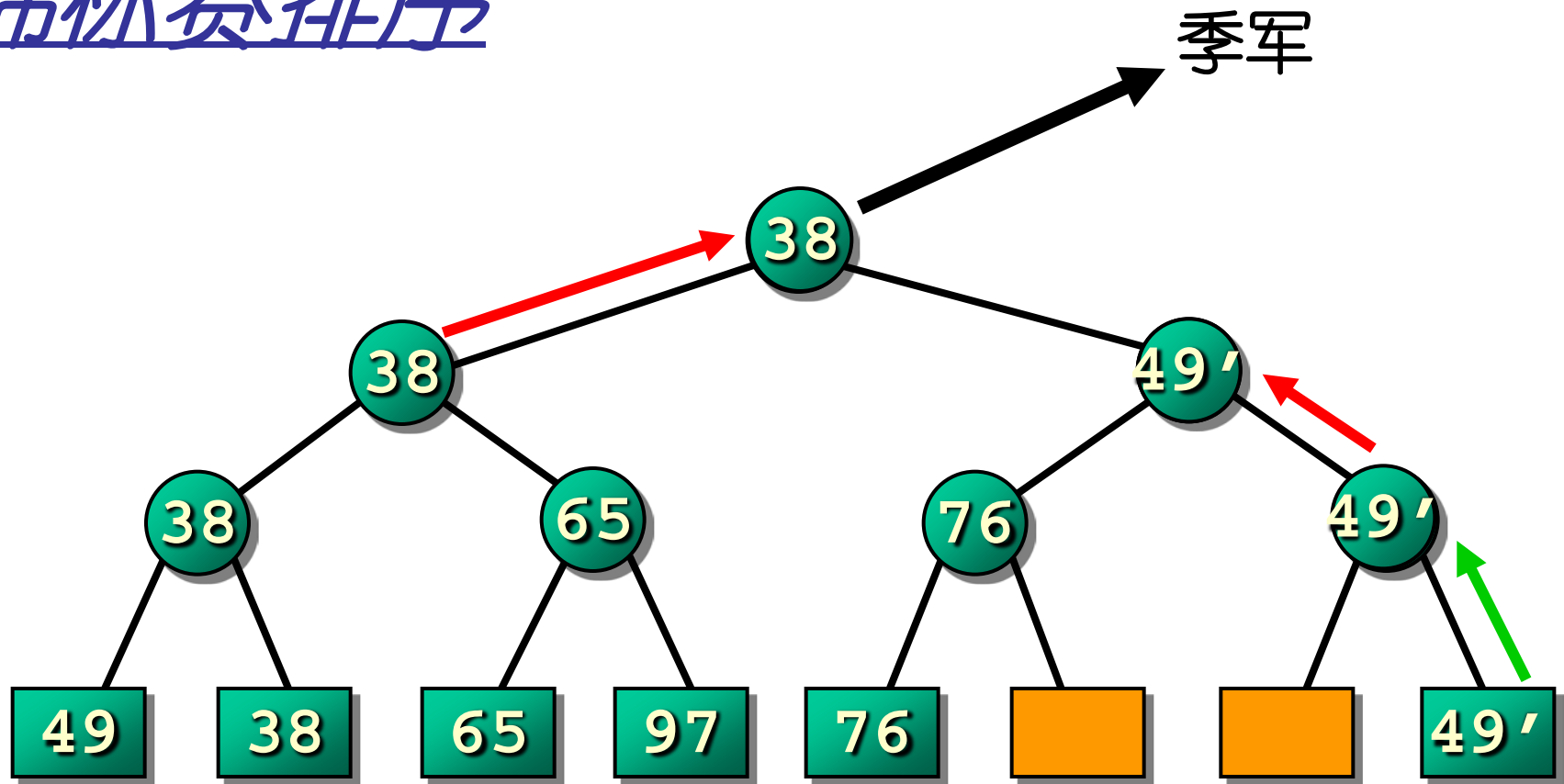
→ 直接上升



- 比较次数 = 2

- 输出亚军

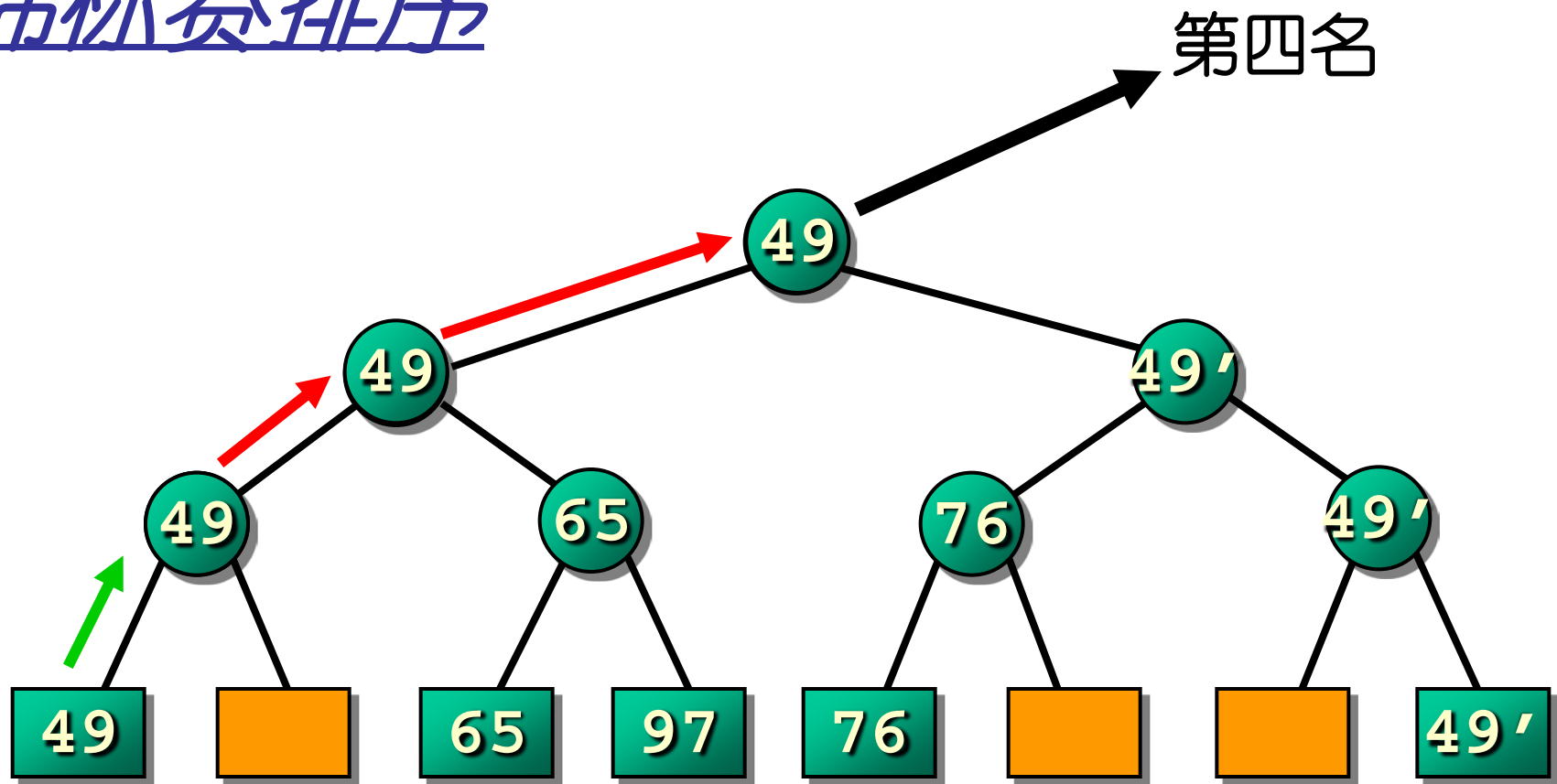
# 锦标赛排序



- 比较次数 = 2

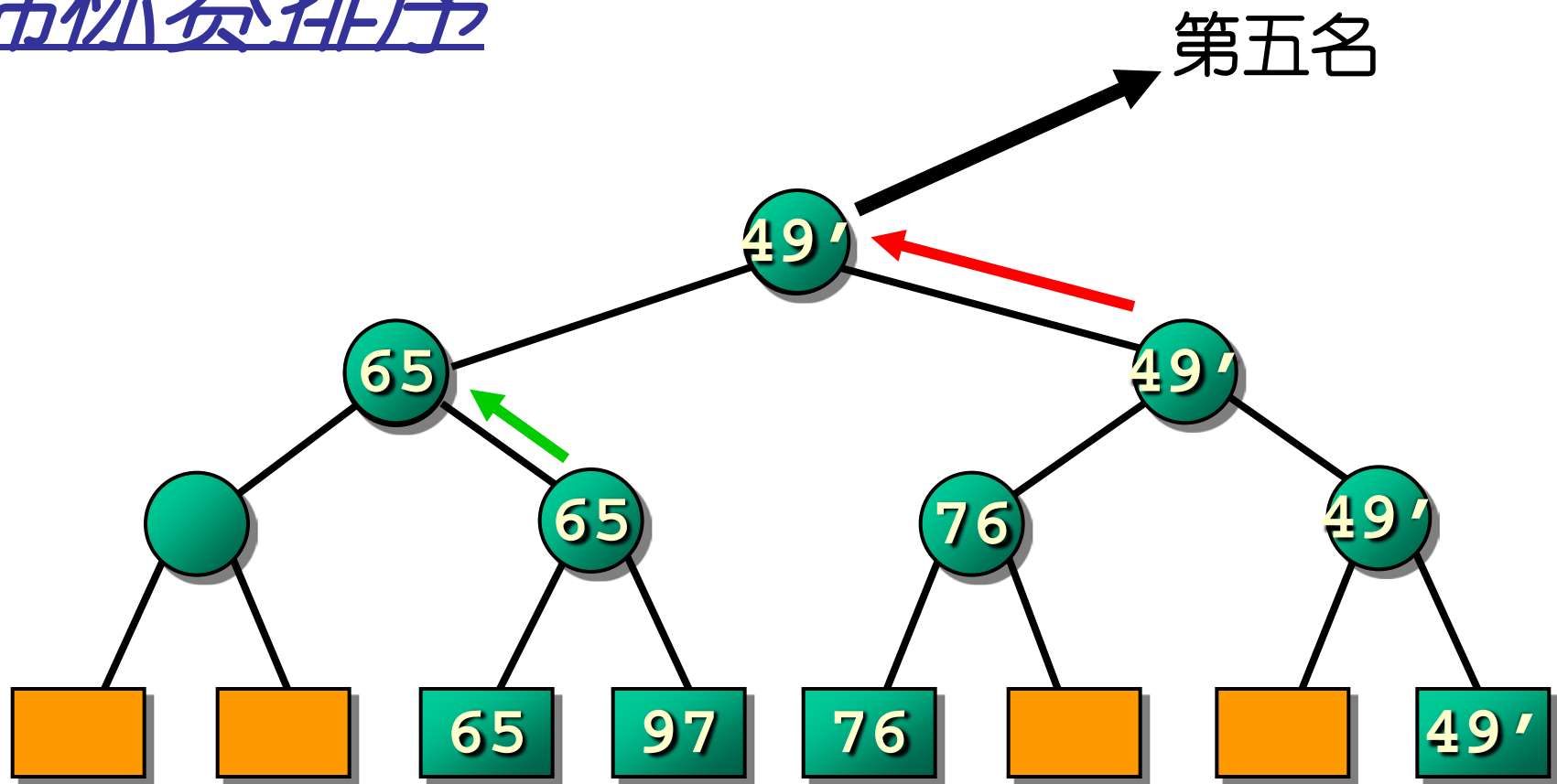
- 输出季军

# 锦标赛排序



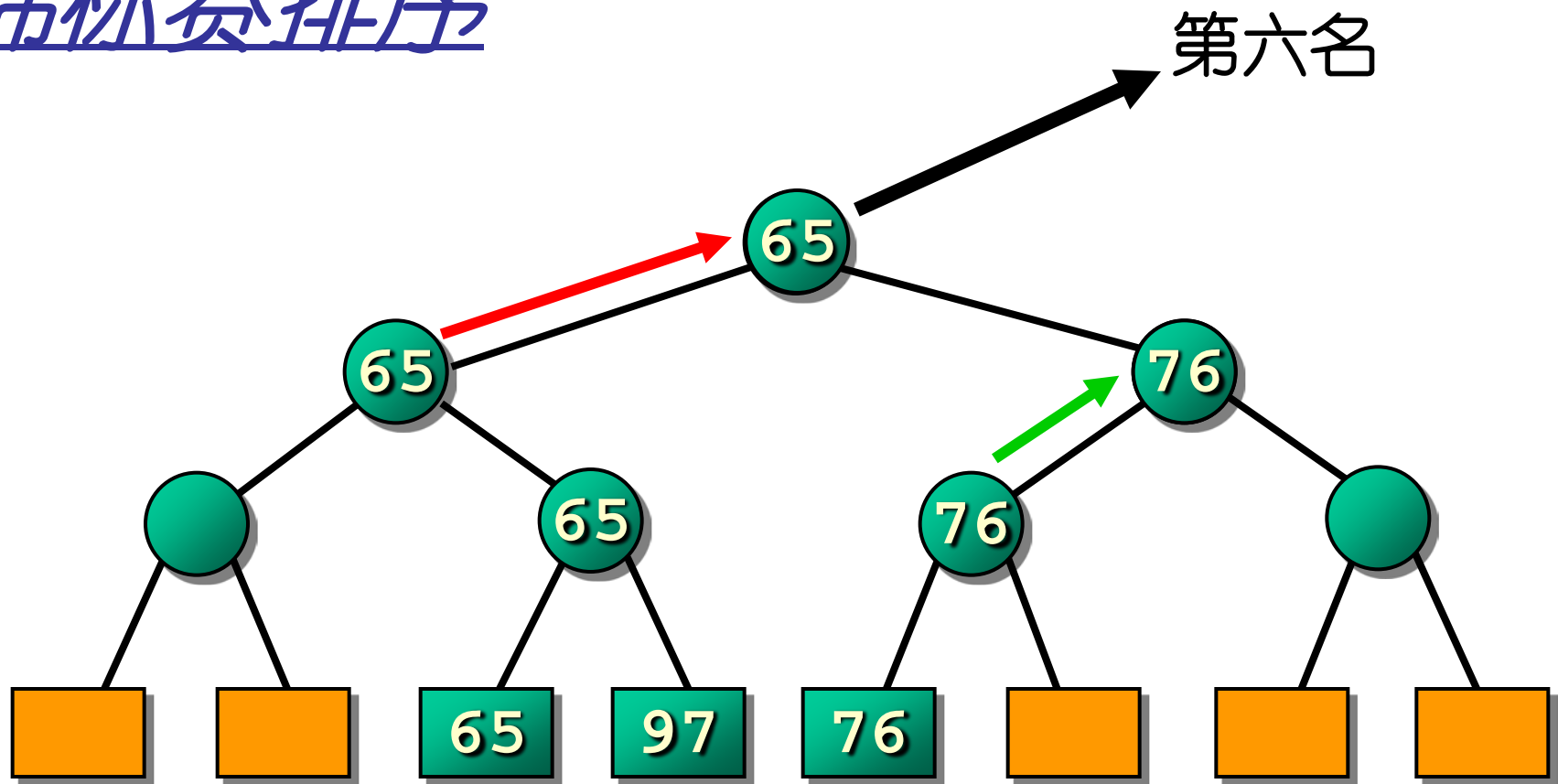
- 比较次数 = 2
- 输出第四名

# 锦标赛排序



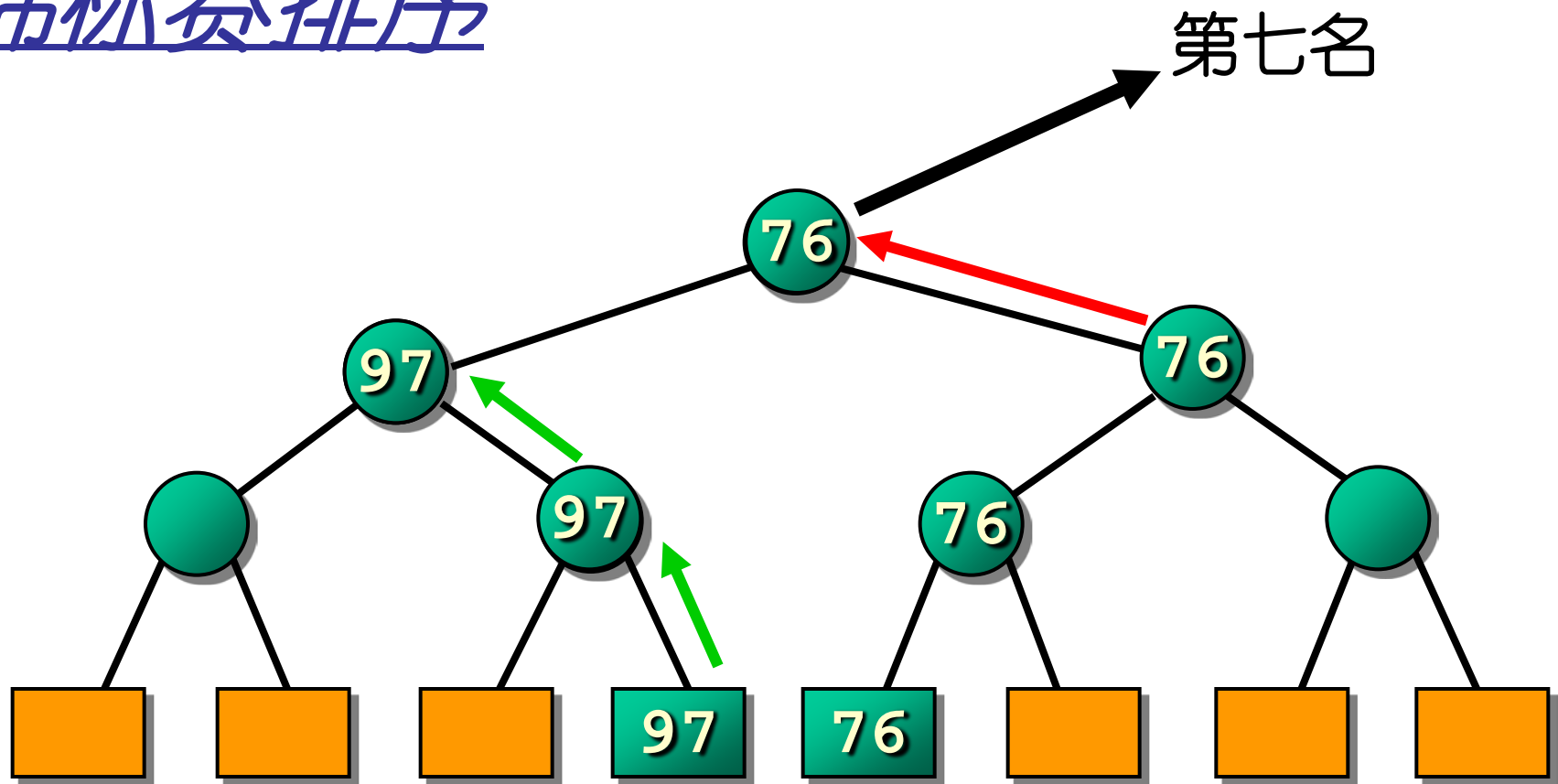
- 比较次数 = 1
- 输出第五名

# 锦标赛排序



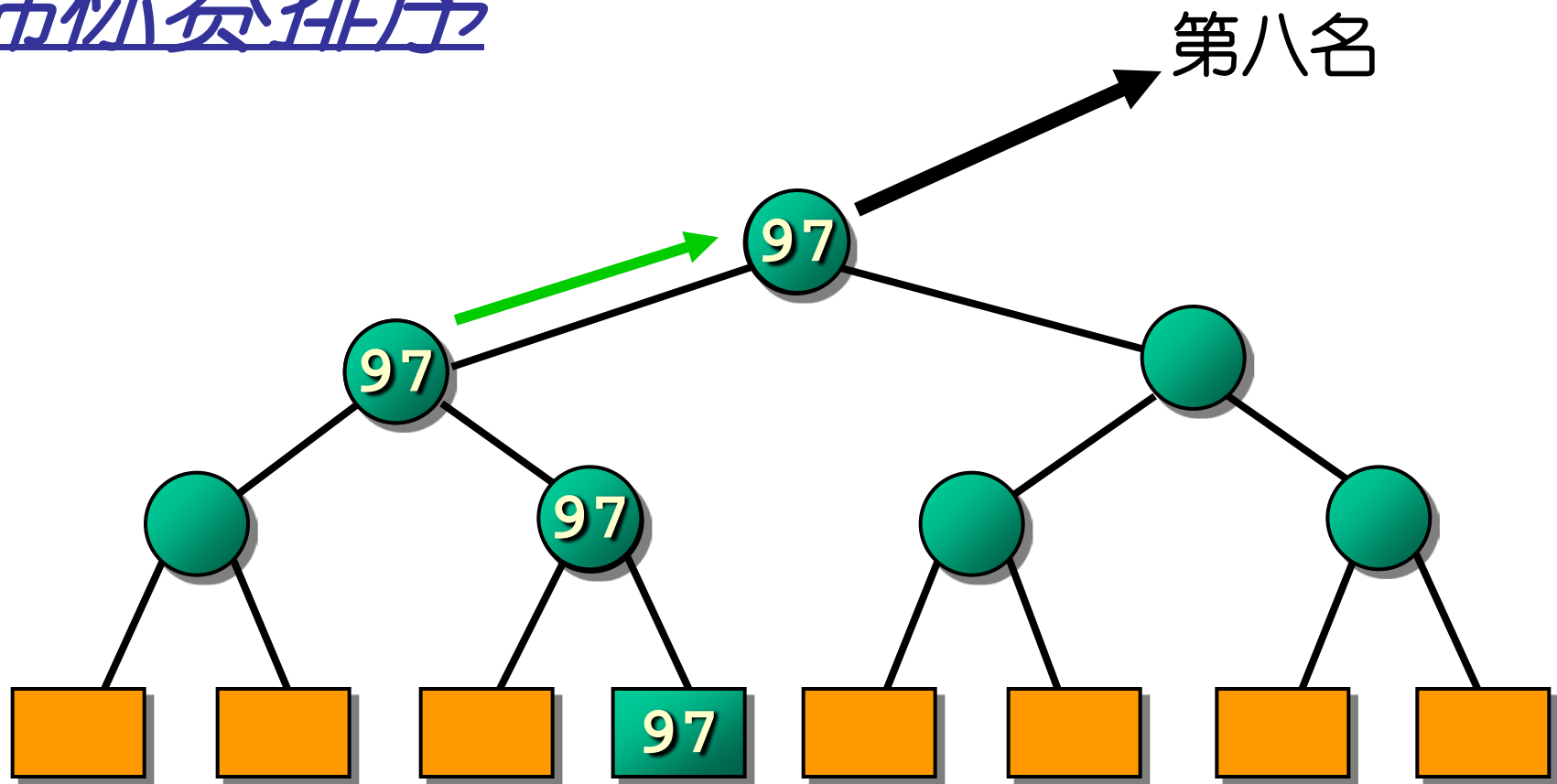
- 比较次数 = 1
- 输出第六名

# 锦标赛排序



- 比较次数 = 1
- 输出第七名

# 锦标赛排序



- 比较次数 = 0
- 输出第八名

# 锦标赛排序

- 时间复杂度

- 锦标赛排序构成的选择树是满二叉树（如果元素不够补充空节点），其深度为  $\lceil \log_2 n \rceil$
- 除第一次选择时需要进行  $n-1$  次比较外，选择其它元素每次只需比较  $O(\log_2 n)$  次，所以总的比较次数为  $O(n \log_2 n)$
- 对象的移动次数不超过排序码的比较次数，所以锦标赛排序总时间复杂度为  $O(n \log_2 n)$



# 锦标赛排序

- 空间复杂度

- 锦标赛排序法虽然减少了许多排序时间，但是使用了较多的附加存储
- 如果有 $n$ 个对象，必须使用至少 $2n-1$ 个结点来存放选择树

- 稳定性：

- 稳定

# 堆排序

## • 堆

–  $n$ 个元素的序列 $\{K_1, K_2, \dots, K_n\}$ , 满足:

$$\begin{cases} k_i \leq k_{2i} \\ k_i \leq k_{2i+1} \end{cases} \quad \text{或} \quad \begin{cases} k_i \geq k_{2i} \\ k_i \geq k_{2i+1} \end{cases}$$

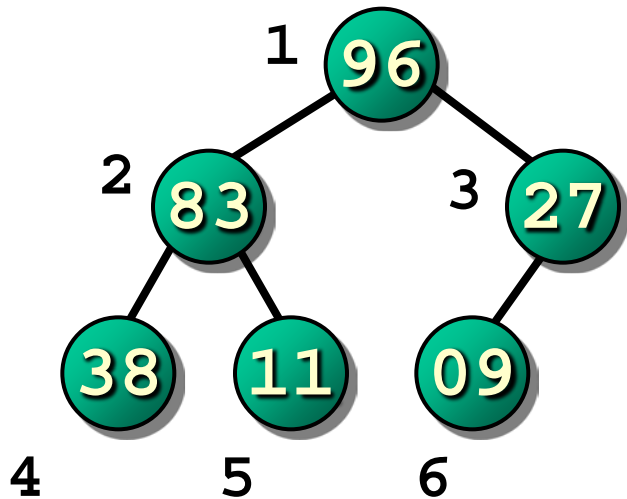
最小堆

最大堆

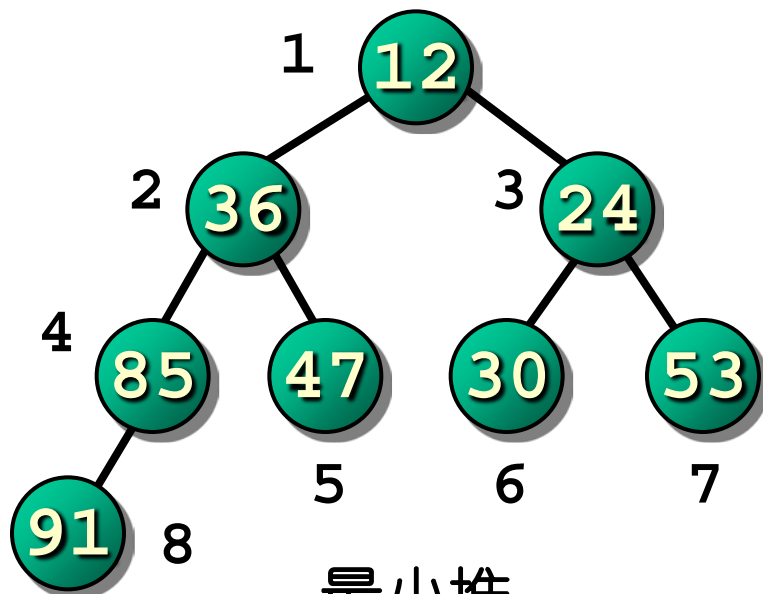
0	1	2	3	4	5	6
	96	83	27	38	11	09

0	1	2	3	4	5	6
	96	83	27	38	11	09

0	1	2	3	4	5	6	7	8
	12	36	24	85	47	30	53	91



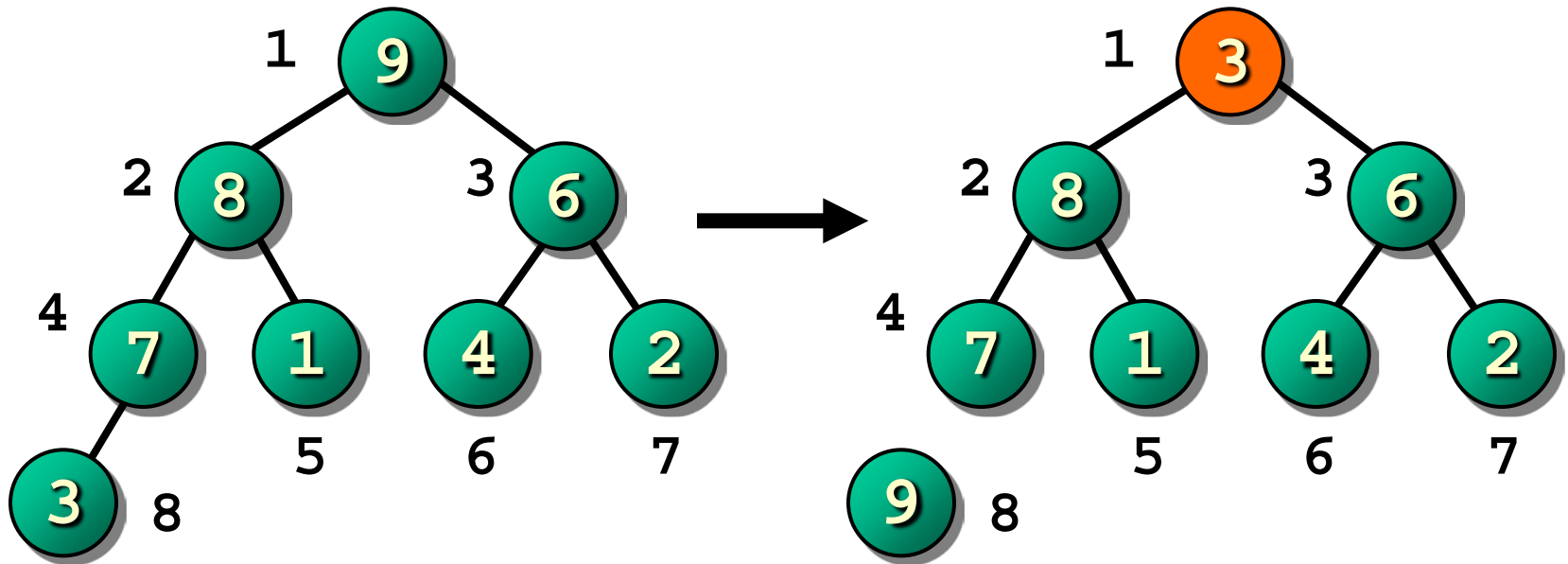
最大堆



最小堆

# 堆排序

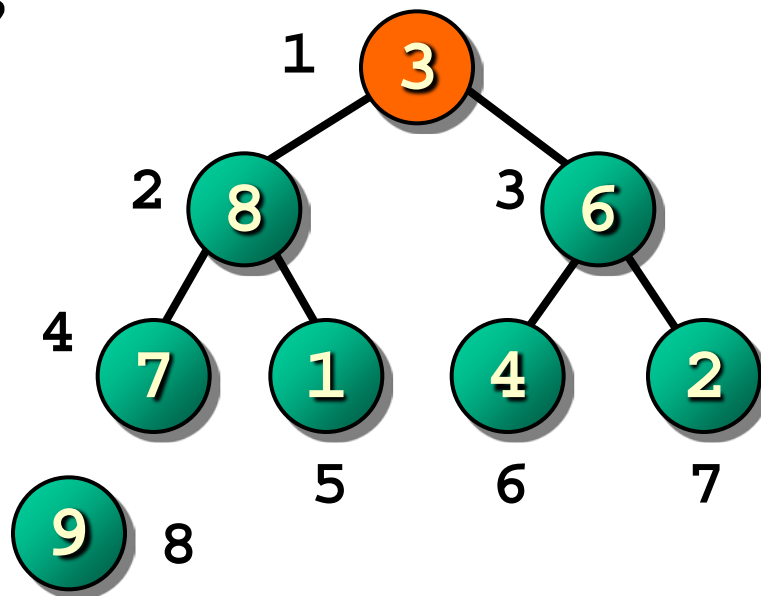
- 假设一组数据已经组织成了最大堆



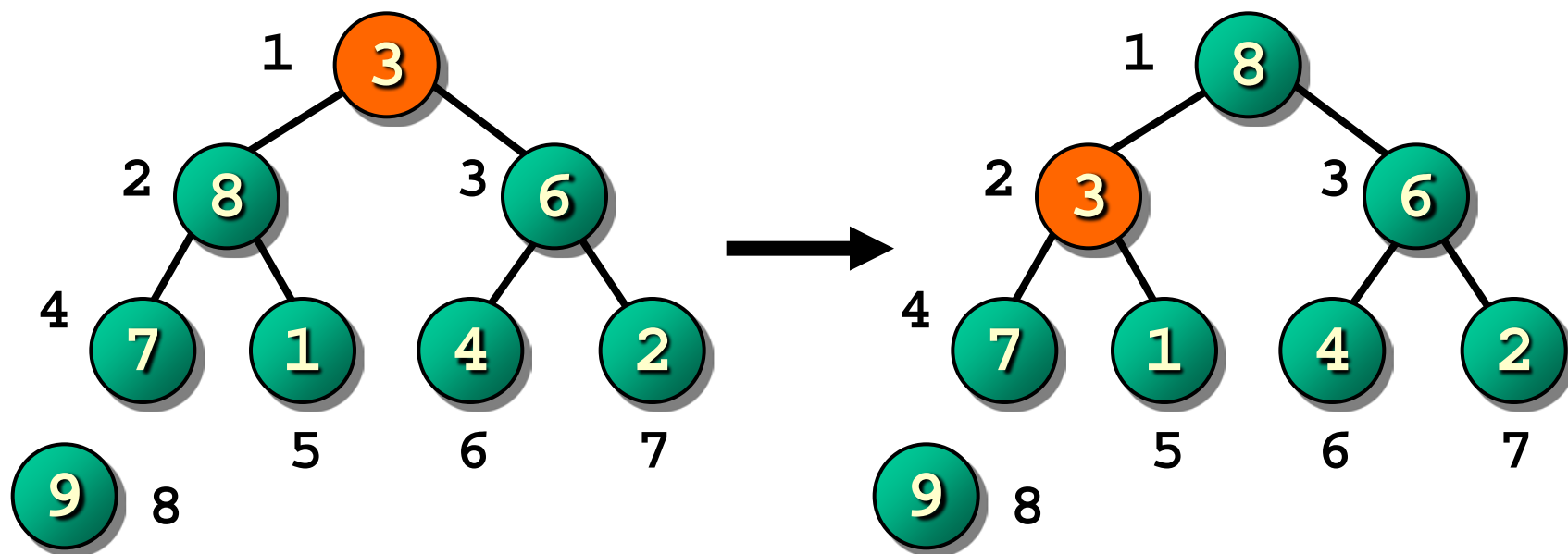
- 堆顶肯定是最小的
- 输出之：这里说的“输出”不是真的删除，而是把它和最后一个元素做交换

# 堆排序

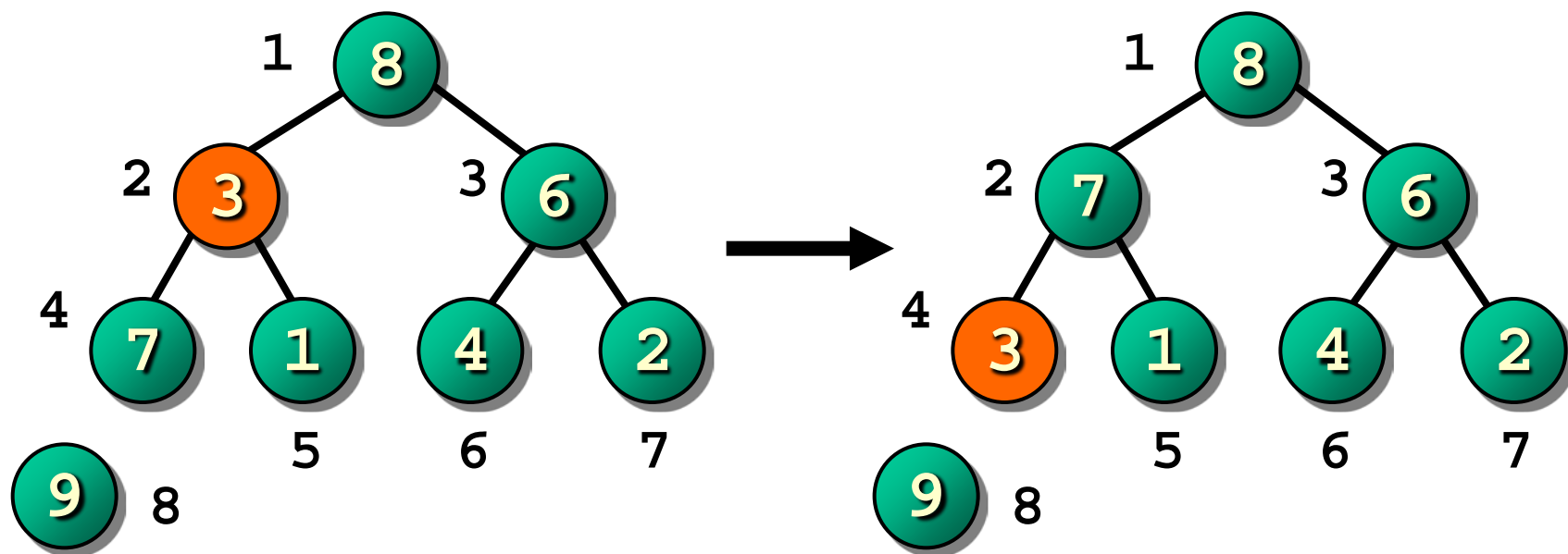
- “输出”后，需要重新调整成最大堆
  - 称作“筛选”



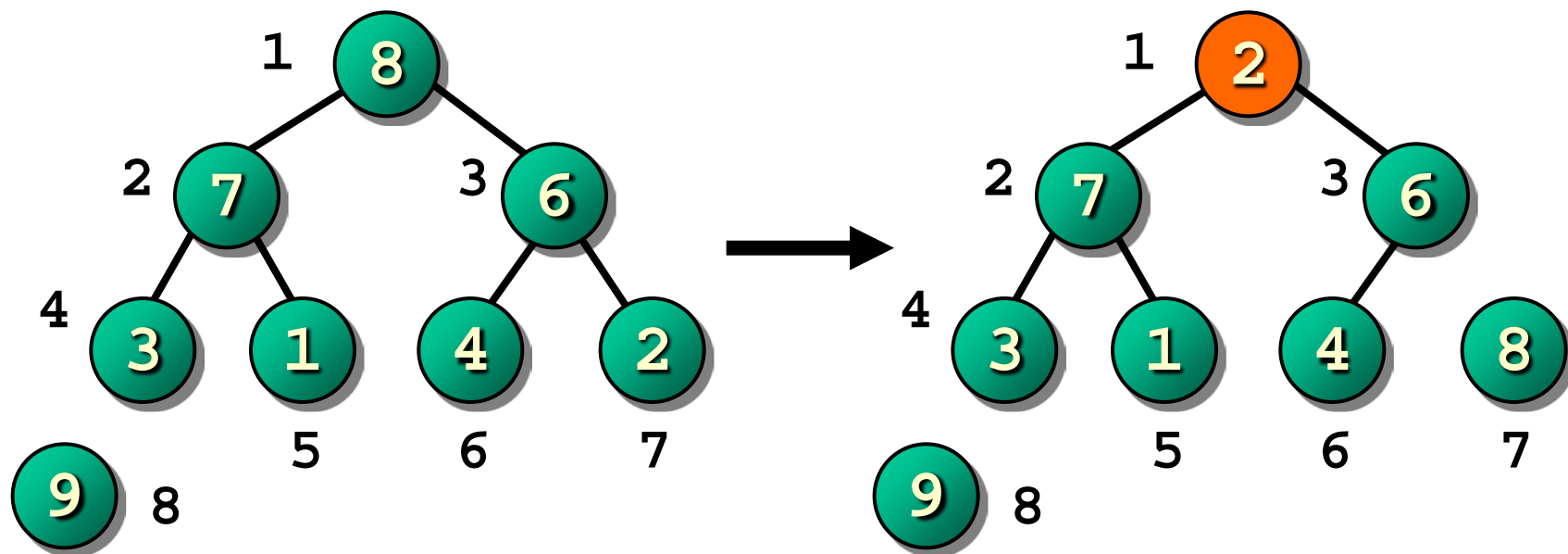
- 3比8、6小，必须下移
- 要从“儿子”中挑选合适的替换人选，因为除了堆顶元素，剩下来最大的就是第2层的元素了



- 3的两个“儿子”中，8更大，应作为“继承人”
- 把3和8做交换

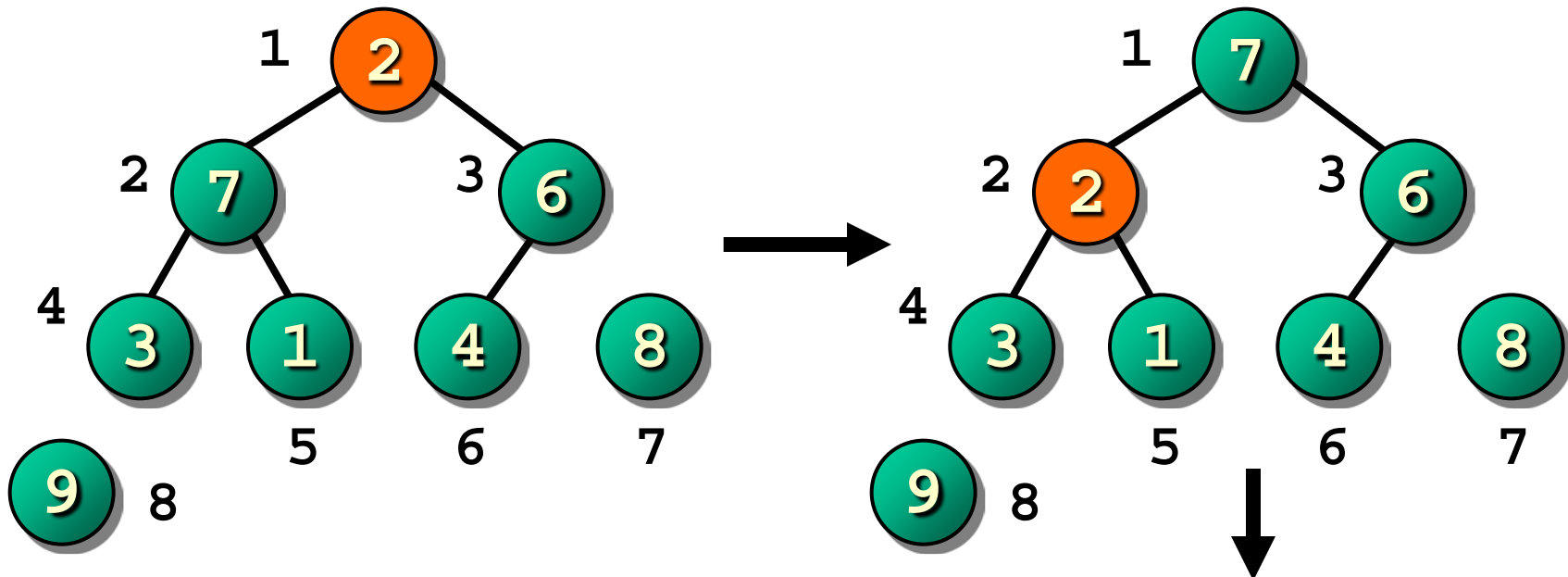


- 现在的3还不满足最大堆的要求，继续调整
- 在现在的3的“儿子”中挑选更大的一个做替换
- 直到3找到一个合适的位置为止

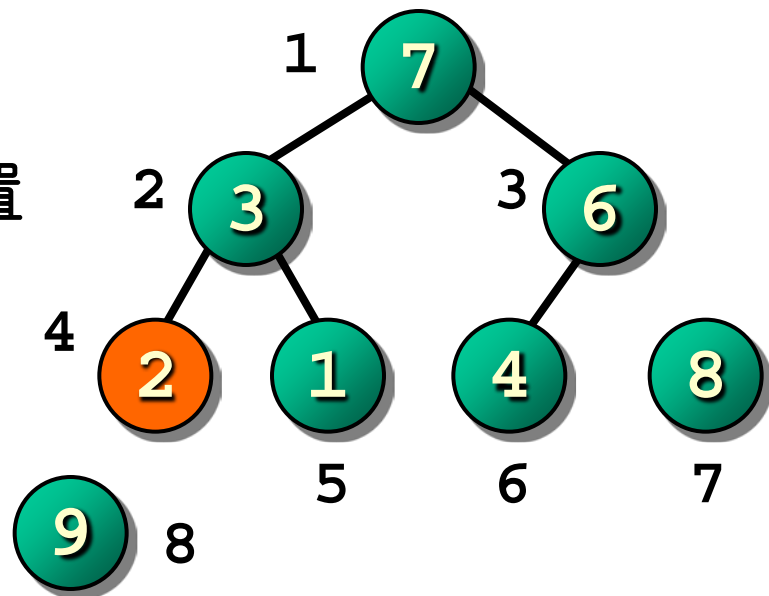


- 经过刚才的“筛选”，又重新整理成了最大堆
- 再次输出堆顶元素，肯定是现在最大的



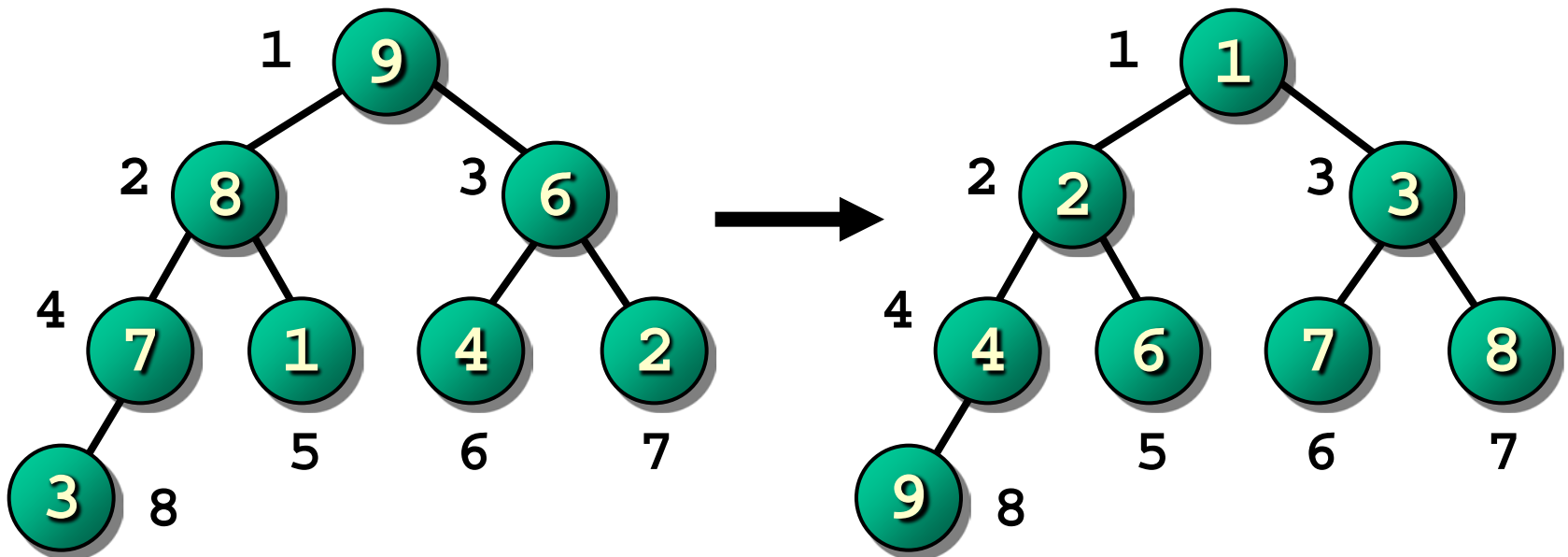


- 类似地，再次进行调整
  - 下移，直到找到合适的位置



# 堆排序

- 总之，如果待排序的数据已经被组织成了最大堆
  - 先输出堆顶元素
  - 再把新的堆顶元素重新整理成最大堆
  - 直到所有元素都输出



# 堆排序

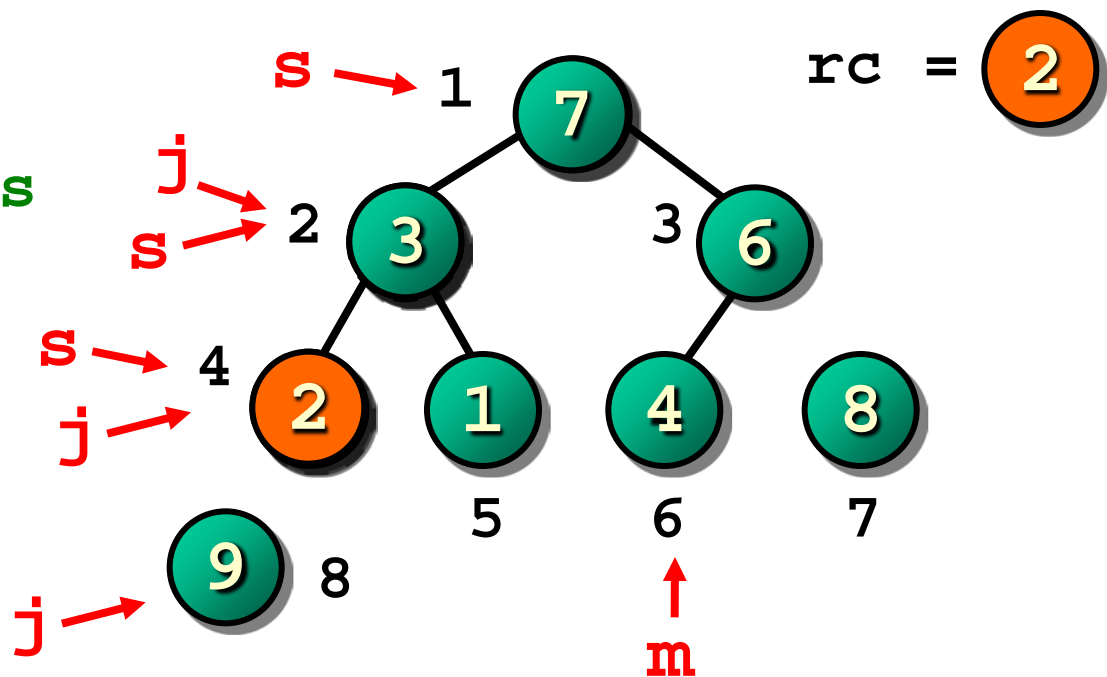
- 调整一个元素的算法

```
void HeapAdjust(HeapType &H, int s, int m) {  
    rc = H.r[s]; //暂时保存待下移的数据  
    for(j = 2 * s; j <= m; j *= 2) {  
        if(j < m && LT(H.r[j].key, H.r[j+1].key))  
            j ++; //j指向s较大的“儿子”  
        if(!LT(rc.key, H.r[j].key))  
            break; //若j的值比rc小, 说明找到了s的位置  
        H.r[s] = H.r[j]; //否则元素j上移  
        s = j;  
    }  
    H.r[s] = rc; //写入s  
}
```

```

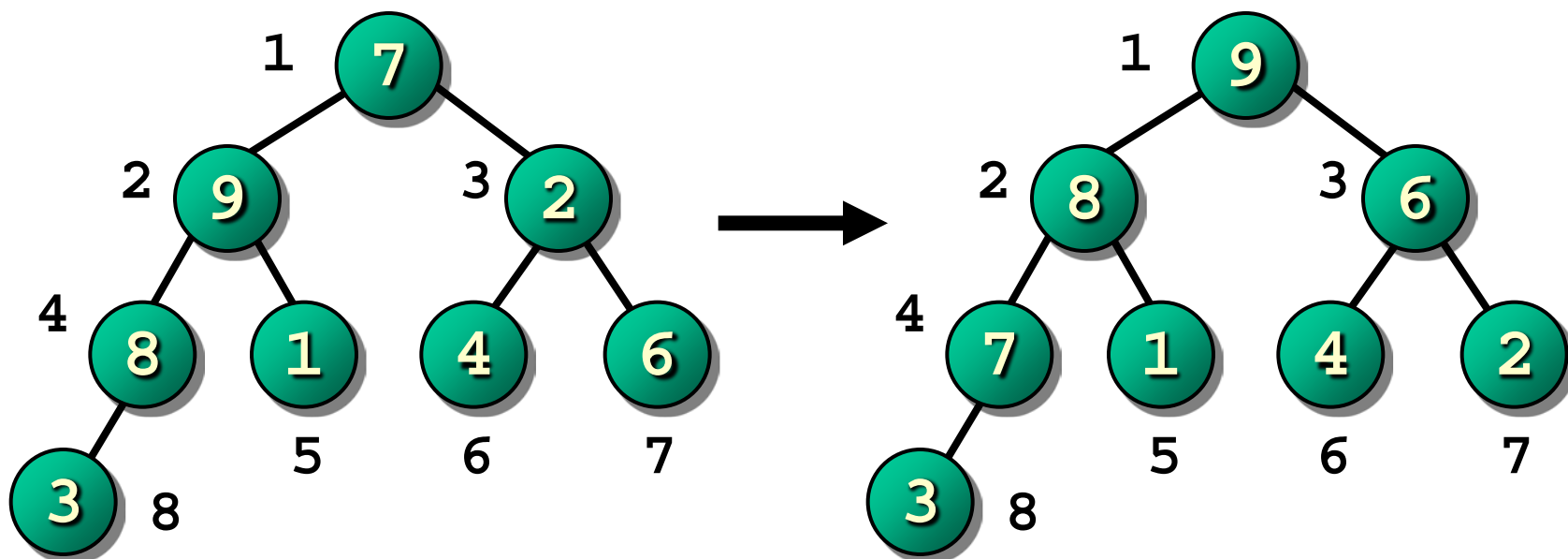
rc = H.r[s]; //暂时保存待下移的数据
for(j = 2 * s; j <= m; j *= 2) {
    if(j < m && LT(H.r[j].key, H.r[j+1].key))
        j ++;    //j指向s较大的“儿子”
    //若j的值比rc小, 说明找到了s的位置
    if(!LT(rc.key, H.r[j].key)) break;
    H.r[s] = H.r[j]; //否则元素j上移
    s = j;
}
H.r[s] = rc; //写入s

```



# 堆排序

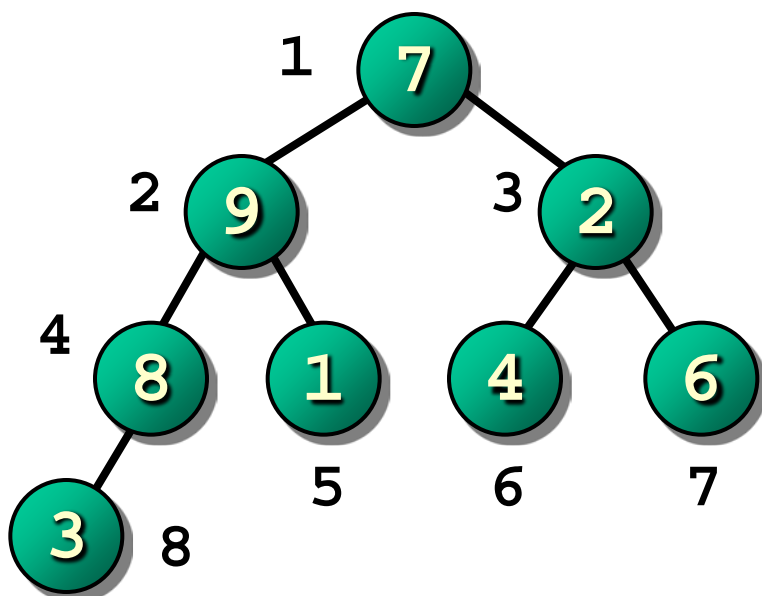
- 杂乱无章的数据怎么整理成最大堆？



# 堆排序

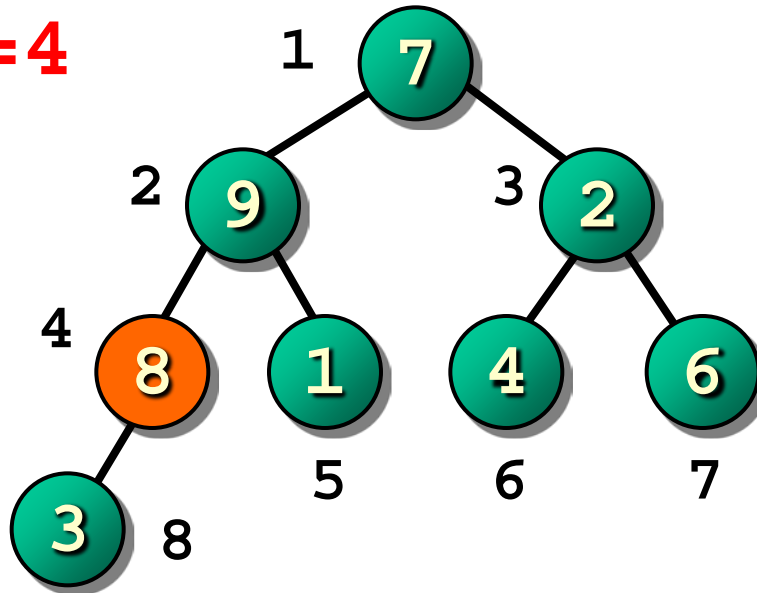
- 把原始数据整理成最大堆

- 对于叶节点来说，已经不可能再下移
- 因此从非叶节点开始调整
  - $n$ 个节点中，非叶节点是  $1 \sim \lfloor n/2 \rfloor$



# 堆排序

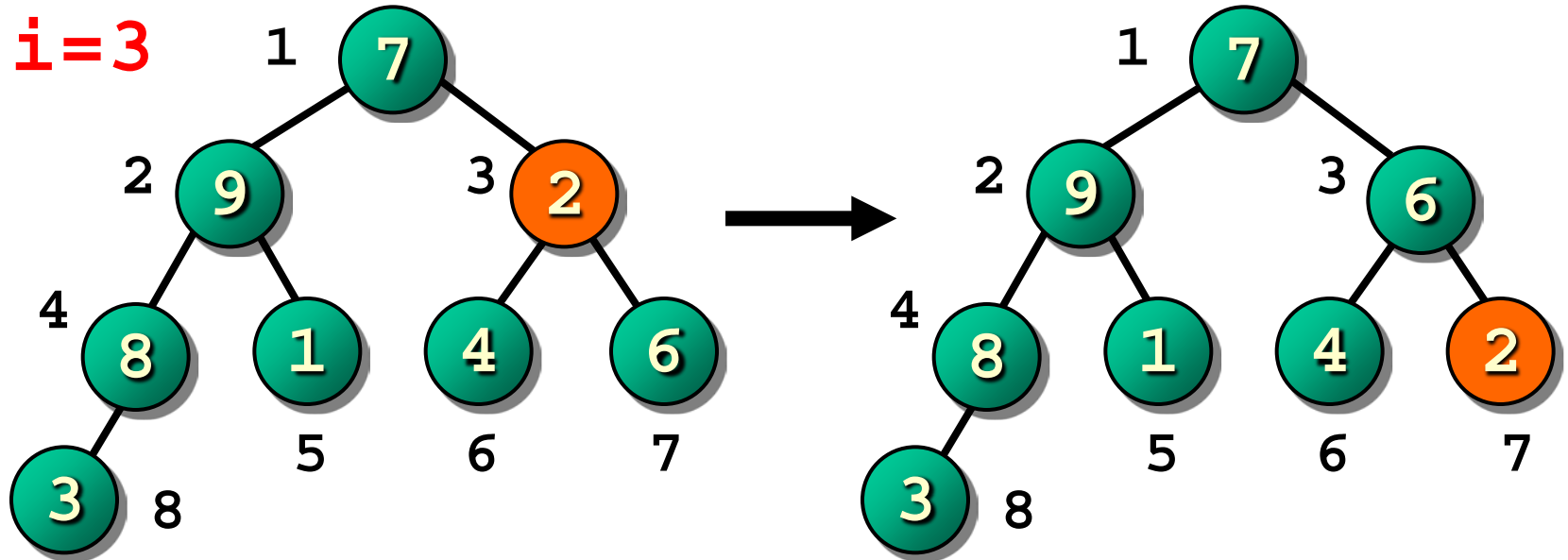
•  $i=4$



- $i$  比它的“儿子”们小么？
- 是，则下移
- 并用最大的那个儿子来替换它
- 直到找到合适的位置

# 堆排序

•  $i=3$

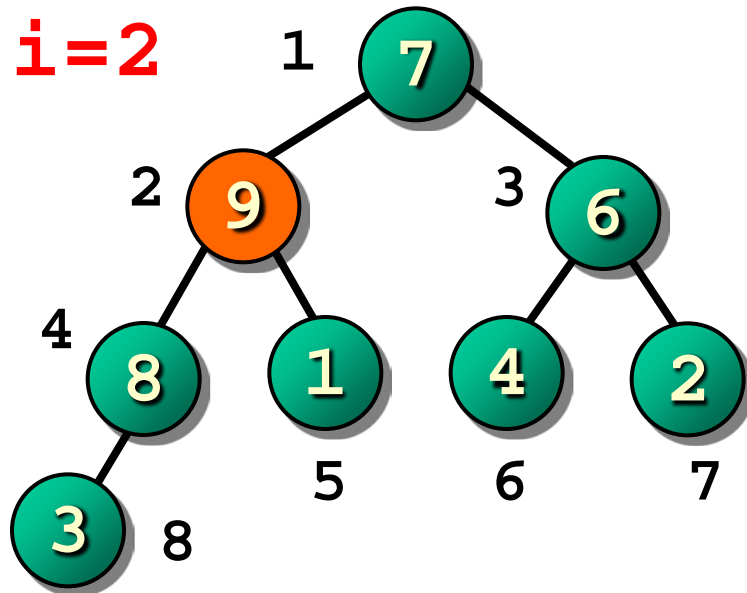


- $i$  比它的“儿子”们小么？
- 是，则下移
- 并用最大的那个儿子来替换它
- 直到找到合适的位置



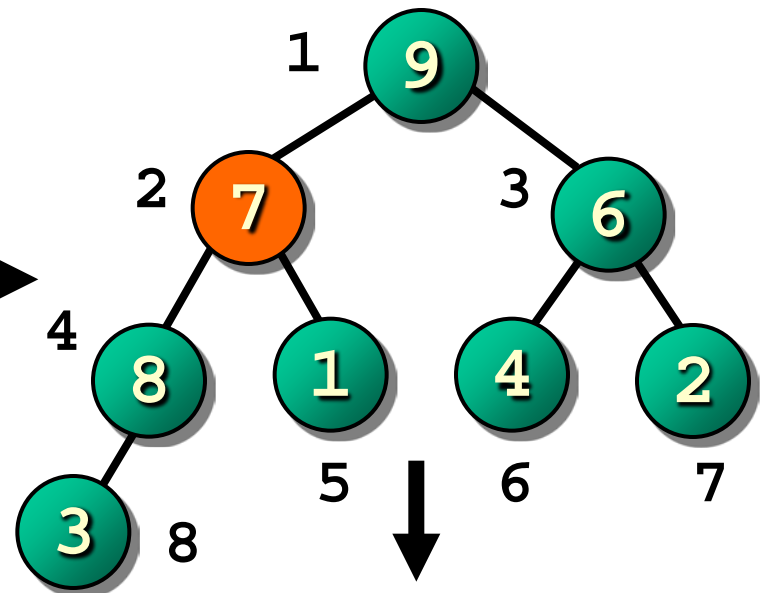
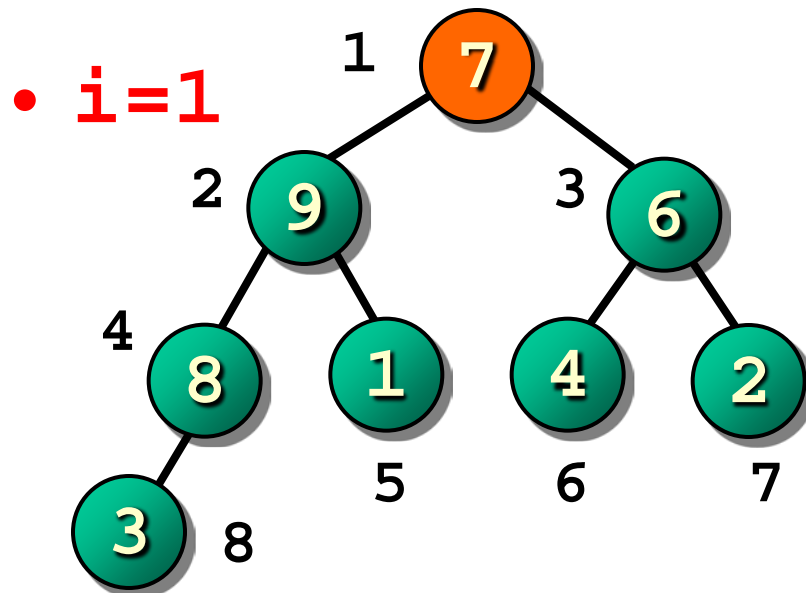
# 堆排序

•  $i=2$

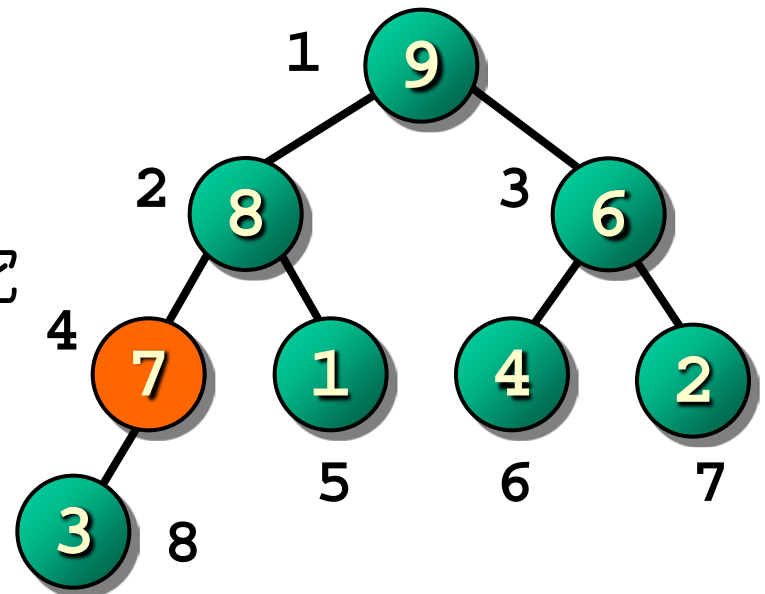


- $i$  比它的“儿子”们小么？
- 是，则下移
- 并用最大的那个儿子来替换它
- 直到找到合适的位置

# 堆排序



- $i$  比它的“儿子”们小么？
- 是，则下移
- 并用最大的那个儿子来替换它
- 直到找到合适的位置



# 堆排序

- 把原始数据整理成最大堆

- $i = \lfloor n/2 \rfloor$  to 1

- 调整第 $i$ 个元素

- 堆排序

- 先把所有原始数据整理成最大堆

- 输出堆顶元素

- 调整新堆顶元素

} 直到所有元素都输出

# 堆排序

- 时间复杂度

- 最差  $O(n) + O(n \log n) = O(n \log n)$
- 这是它相对于快速排序的一个优点
  - 所以特别适合大量数据的排序

- 空间复杂度

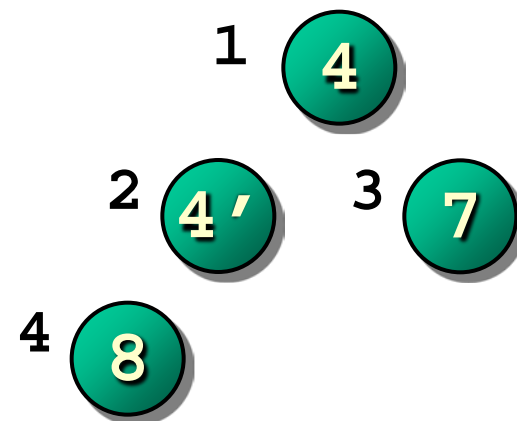
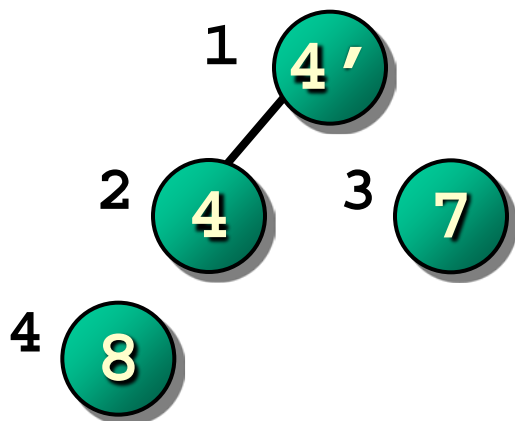
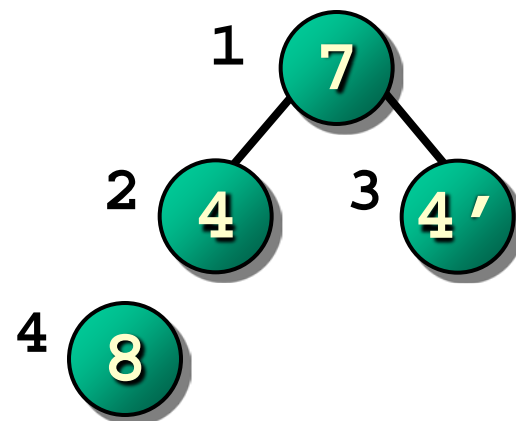
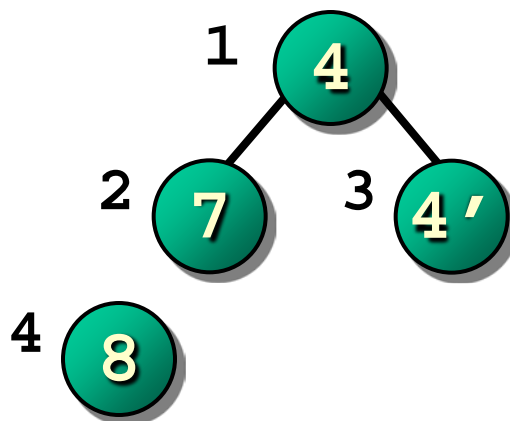
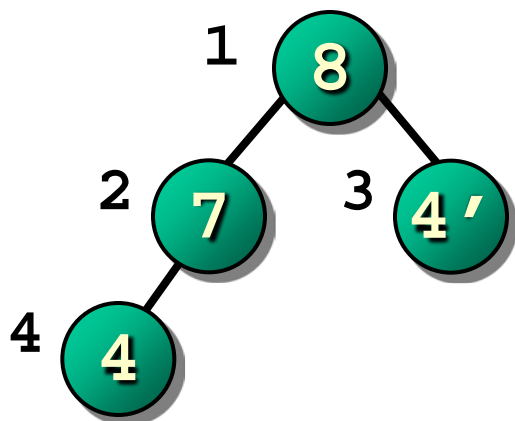
- $O(1)$ : 交换数据时使用了一个临时变量

- 稳定性:

- 不稳定

# 堆排序

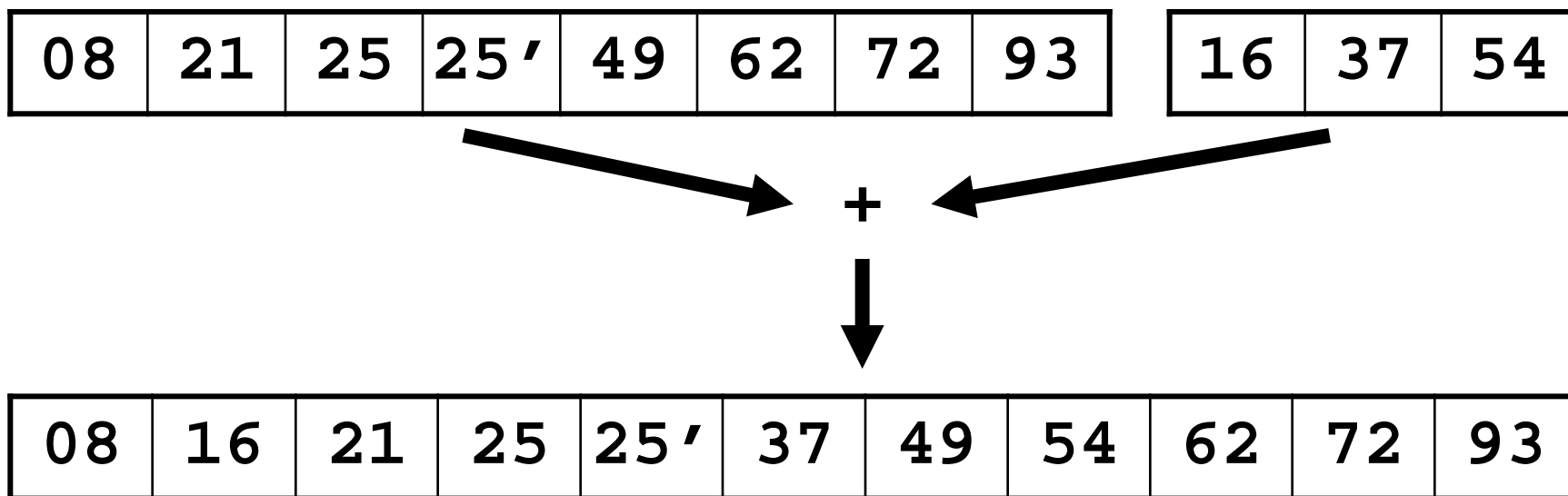
- 不稳定



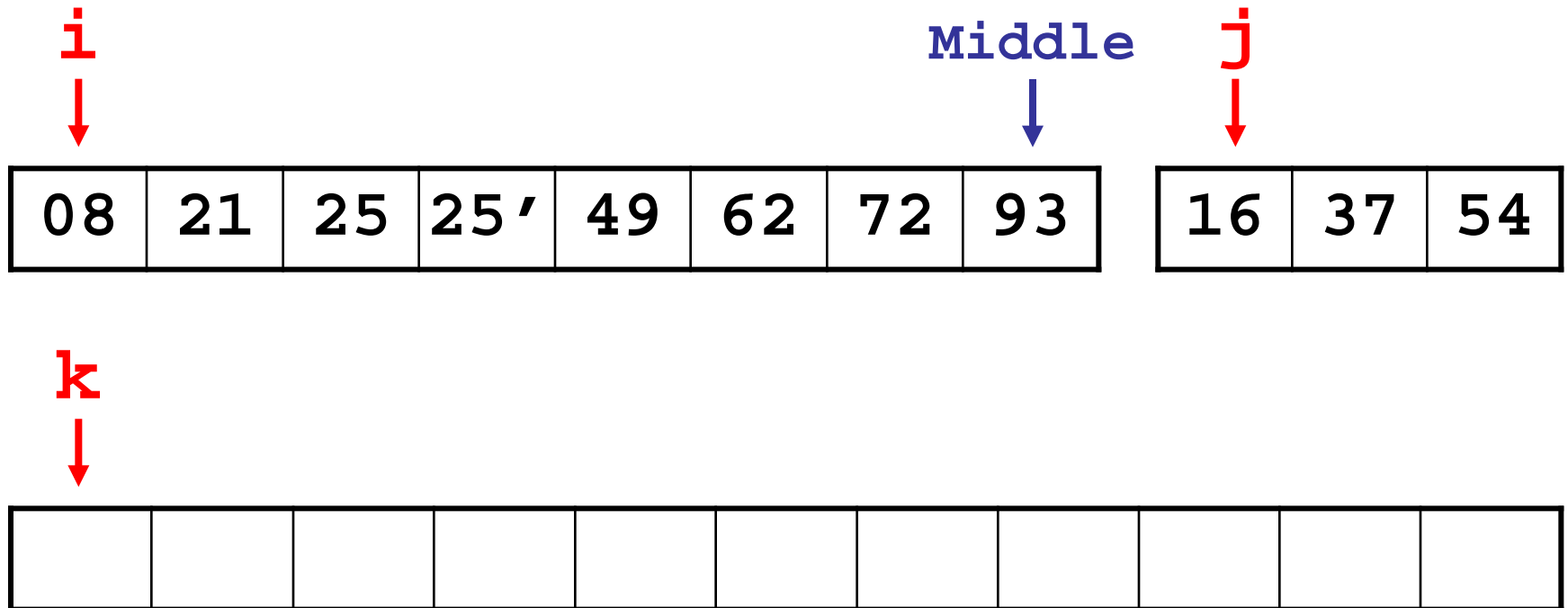
# 归并排序法

## • 归并

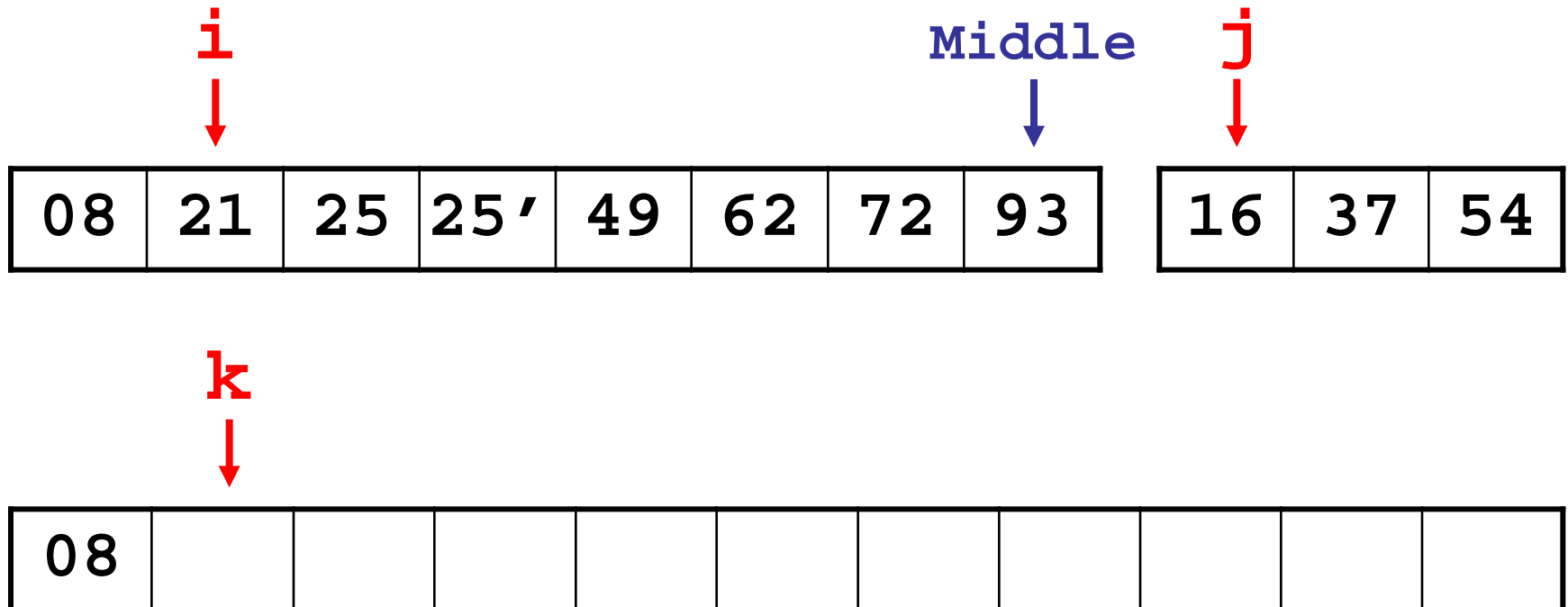
- 将两个或两个以上的有序表合并成一个新的有序表



```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```

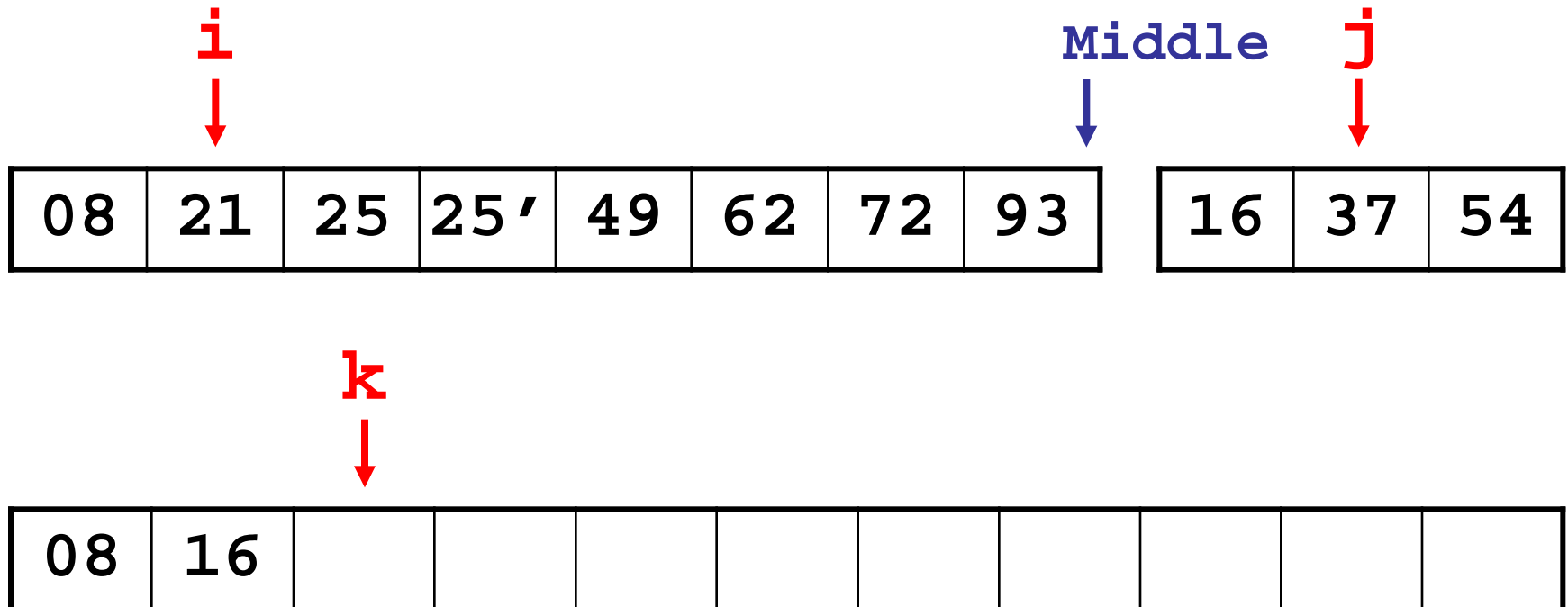


```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```

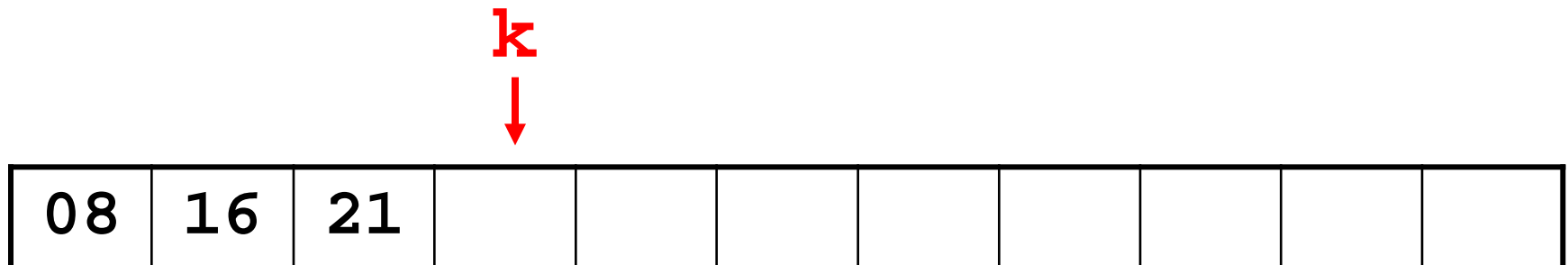
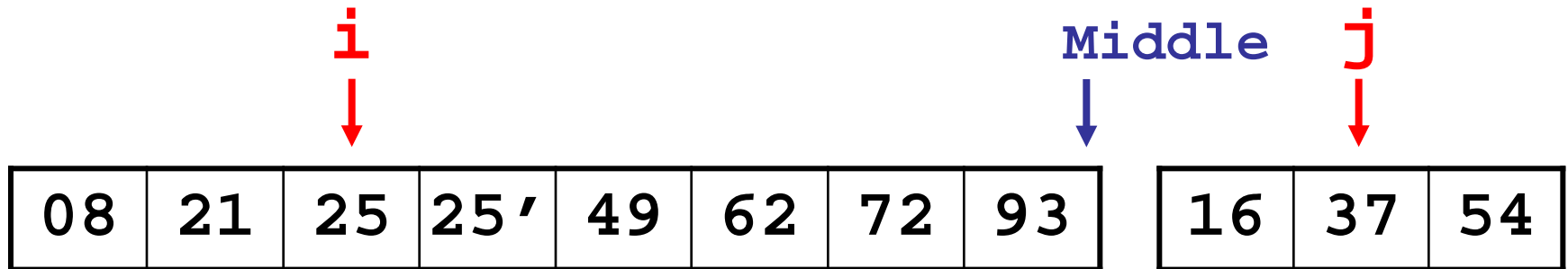




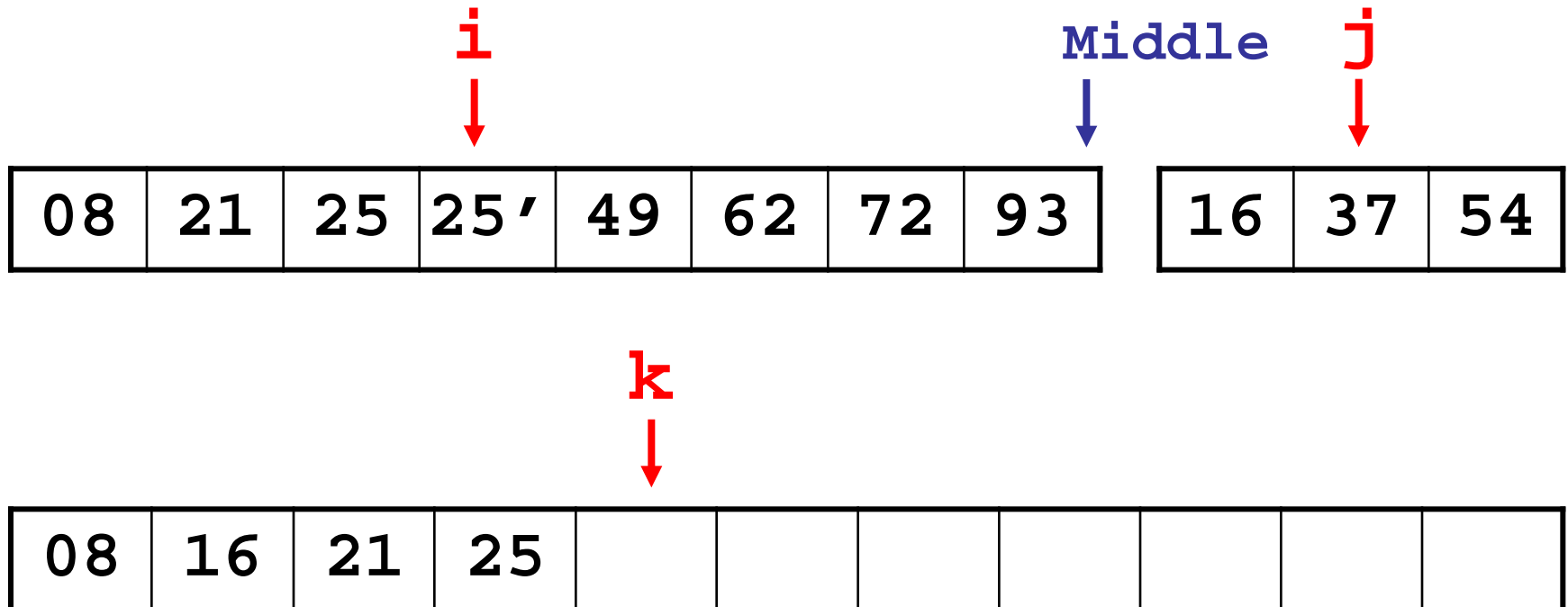
```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```



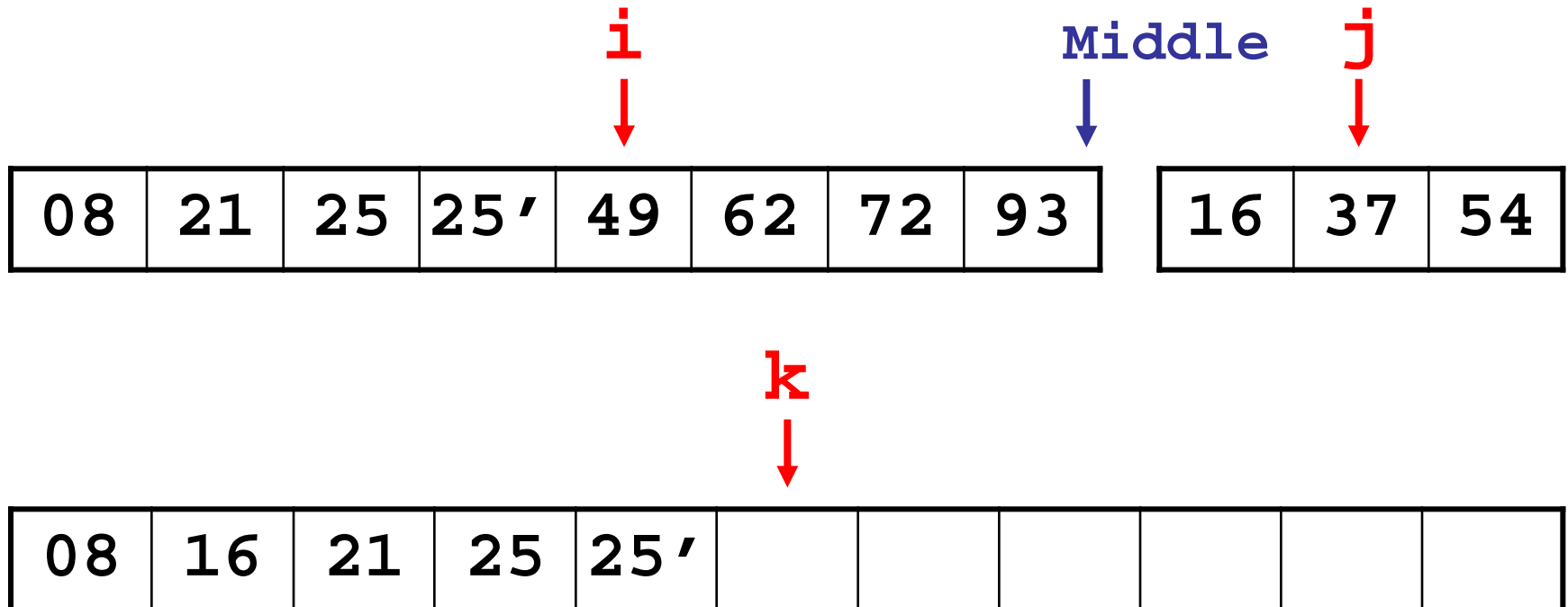
```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```



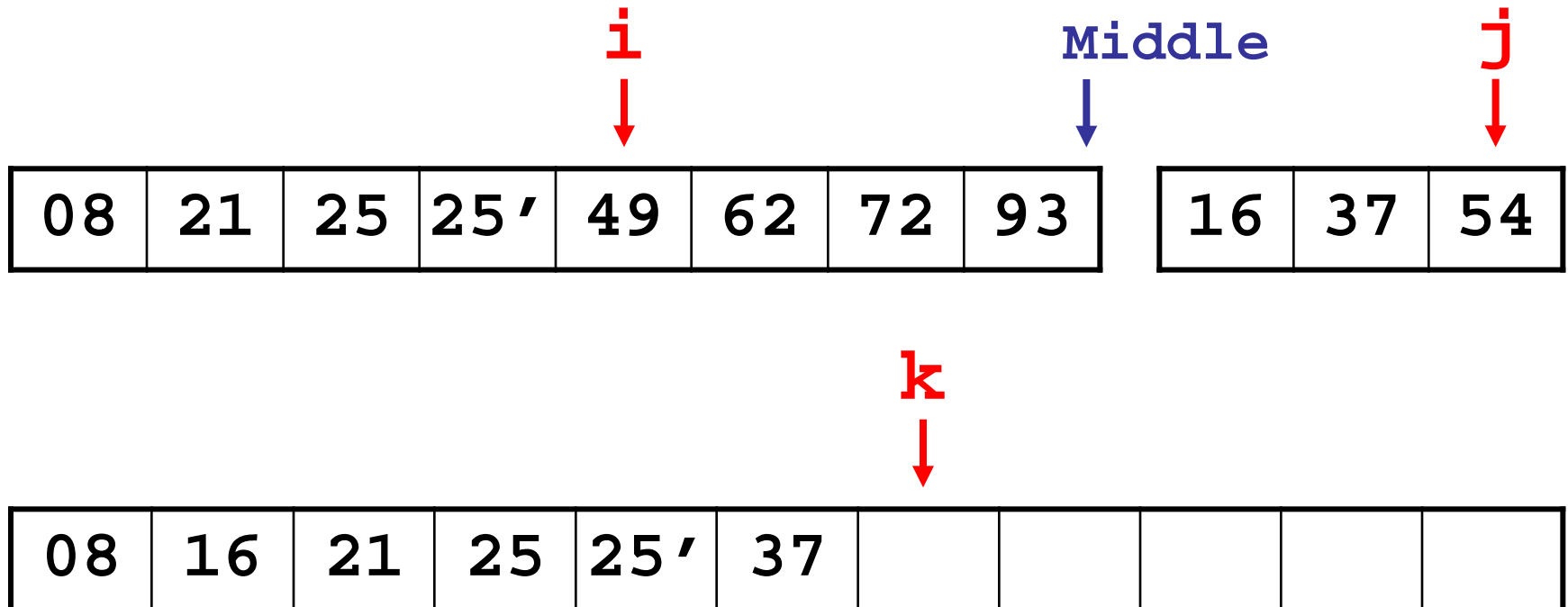
```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```



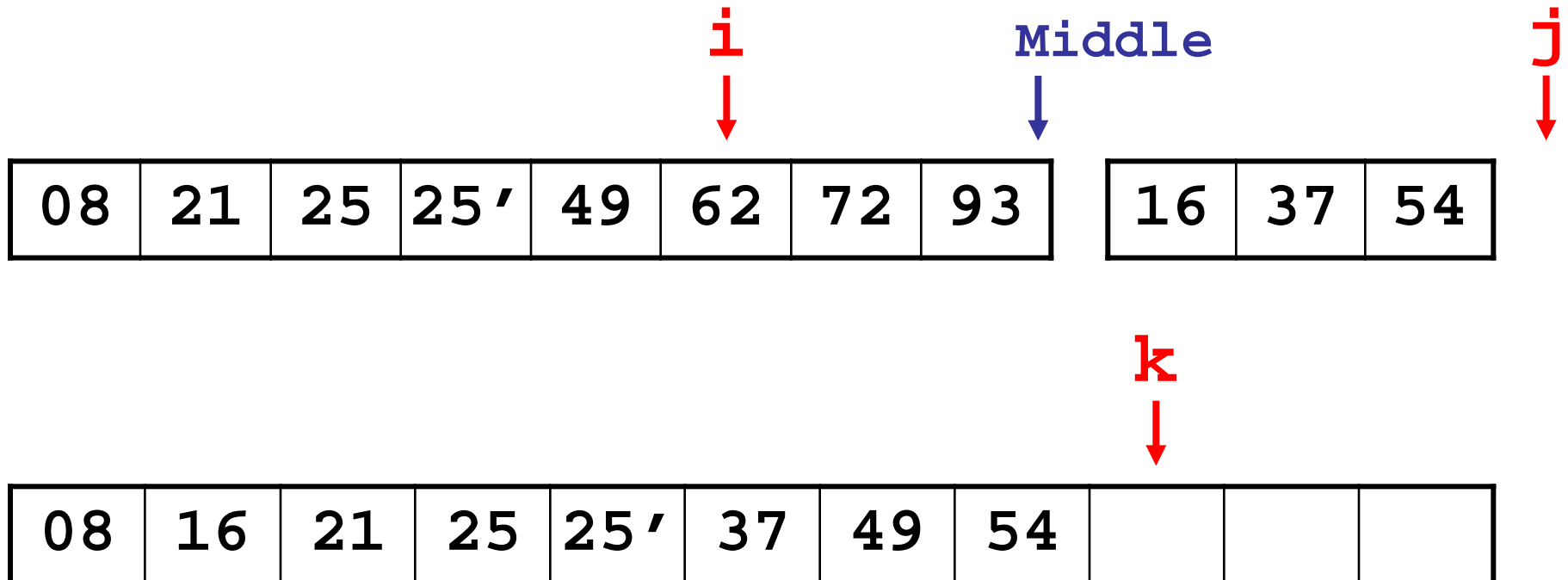
```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```



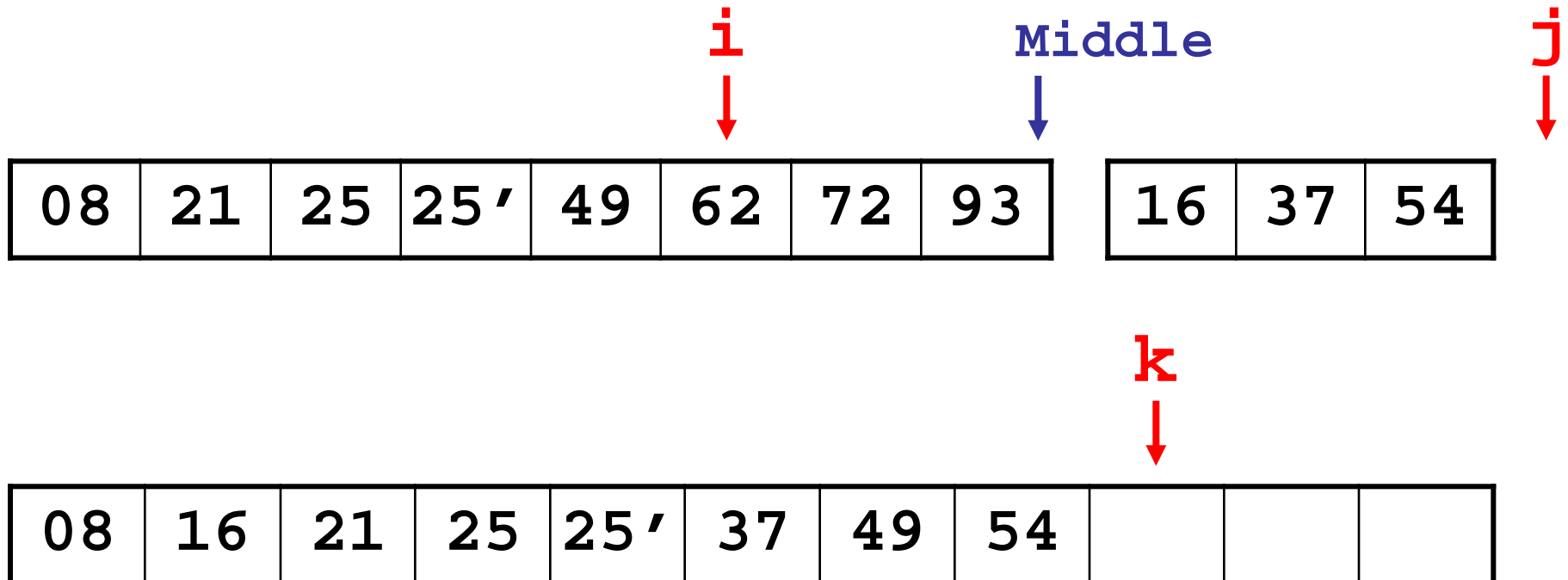
```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```



```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```



```
while(i <= Middle)
    Output[k++] = Data[i++];
while(j <= Middle)
    Output[k++] = Data[j++];
```

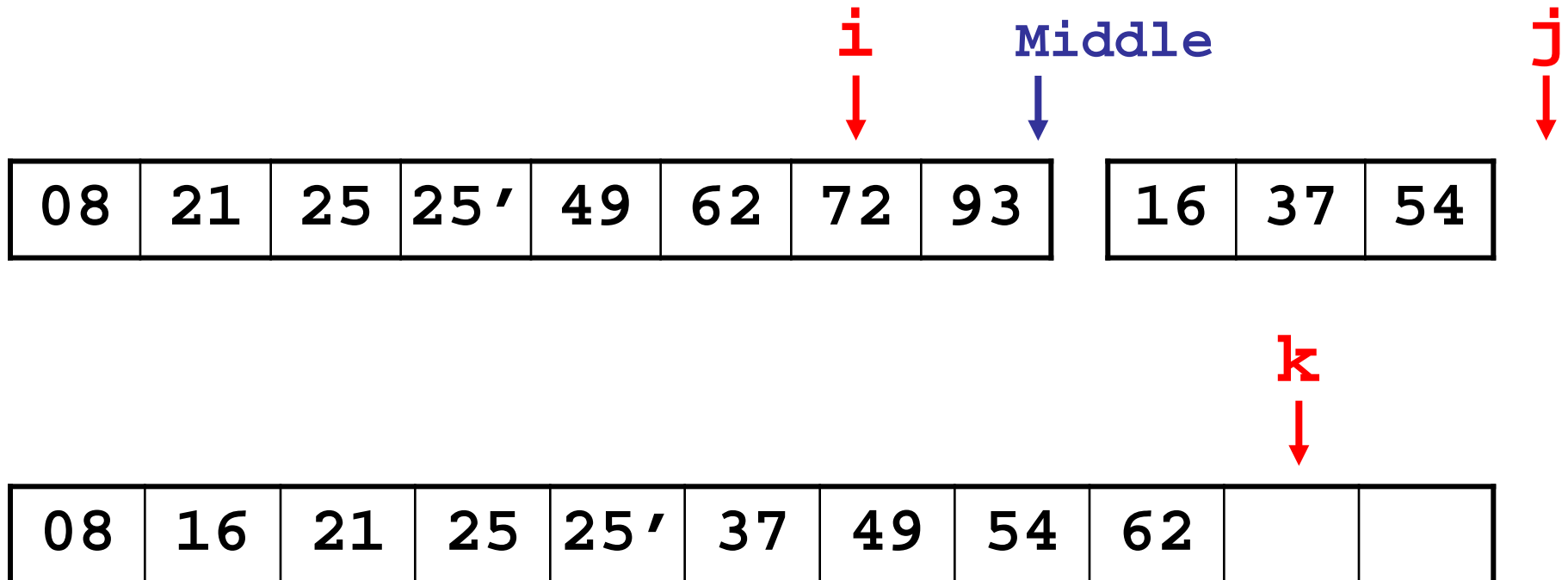


```
while(i <= Middle)
```

```
    Output[k++] = Data[i++];
```

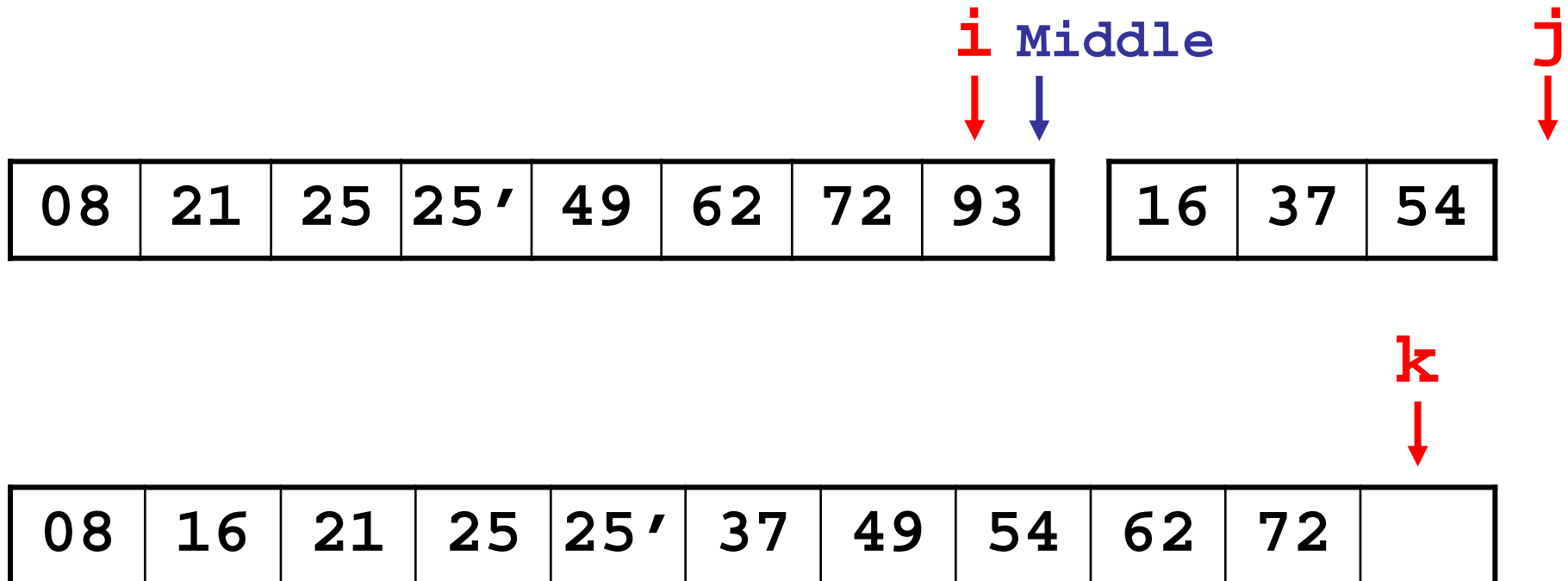
```
while(j <= Middle)
```

```
    Output[k++] = Data[j++];
```

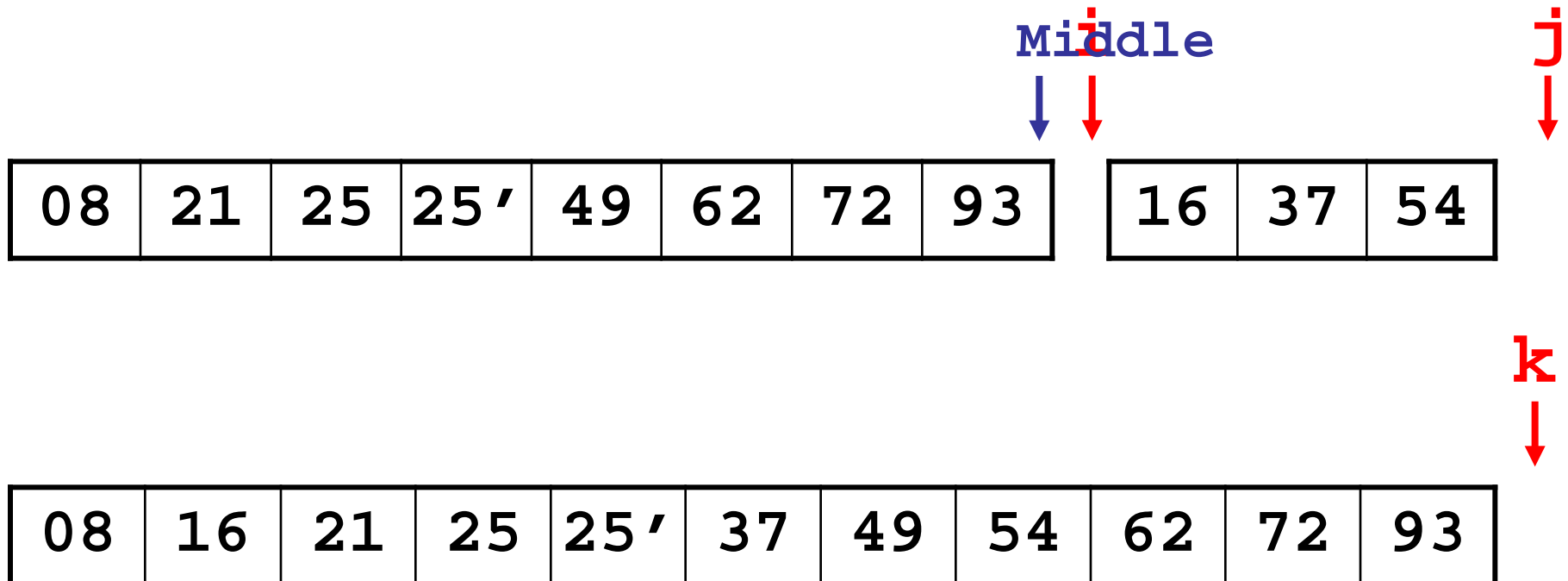




```
while(i <= Middle)
    Output[k++] = Data[i++];
while(j <= Middle)
    Output[k++] = Data[j++];
```



```
while(i <= Middle)
    Output[k++] = Data[i++];
while(j <= Middle)
    Output[k++] = Data[j++];
```



# 归并排序法

- 二路归并算法

- 如果两个序列都还有数据

- 挑选其中更小的一个写入目标序列

```
while(i <= Middle && j <= N)
    if(Data[i] <= Data[j])
        Output[k++] = Data[i++];
    else
        Output[k++] = Data[j++];
```

# 归并排序法

- 如果有一个序列已经扫描完毕
  - 只可能有一个序列扫描完毕
  - 把另一个序列剩余数据依次写入目标序列

```
// i还有剩余
while(i <= Middle)
    Output[k++] = Data[i++];
// j还有剩余
while(j <= Middle)
    Output[k++] = Data[j++];
```

# 归并排序法

- 迭代的归并排序算法

- 假设初始序列有  $n$  个对象
- 首先把它看成是  $n$  个长度为 1 的有序子序列（归并项），先做两两归并
- 得到  $\lceil n/2 \rceil$  个长度为 2 的归并项（如果  $n$  为奇数，则最后一个有序子序列的长度为1）
- 再做两两归并...
- 重复，最后得到一个长度为  $n$  的有序序列

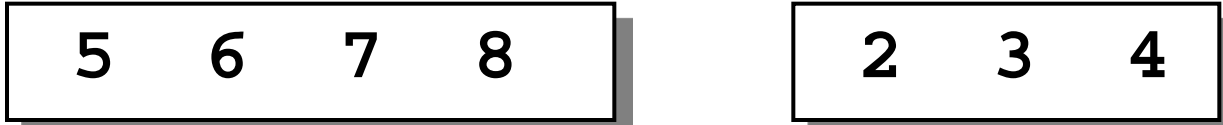
length=1



length=2



length=4



length=8



# 归并排序法

- 时间复杂度

- $O(n \log_2 n)$

- 空间复杂度

- 需要另外一个与原待排序序列同样大小的辅助空间
  - 这是这个算法的缺点

- 稳定性

- 稳定

# 基数排序

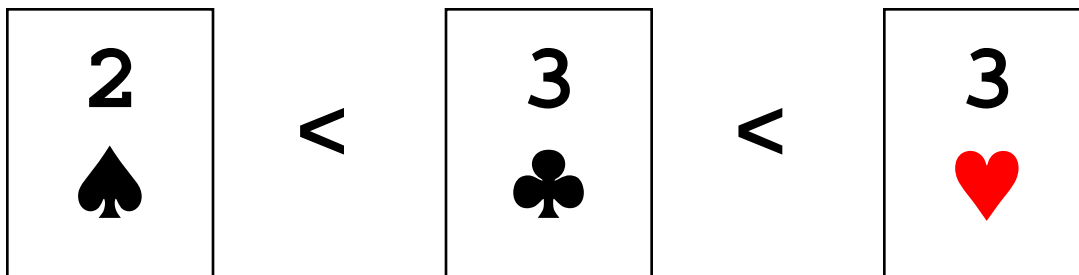
- 多关键字排序

- 前面的排序方法只有一个关键字（排序码）
- 有时候可能存在多个关键字，比如扑克牌

- 关键字1：面值  $2 < 3 < \dots < K < A$

- 关键字2：花色  $\clubsuit < \diamondsuit < \heartsuit < \spadesuit$

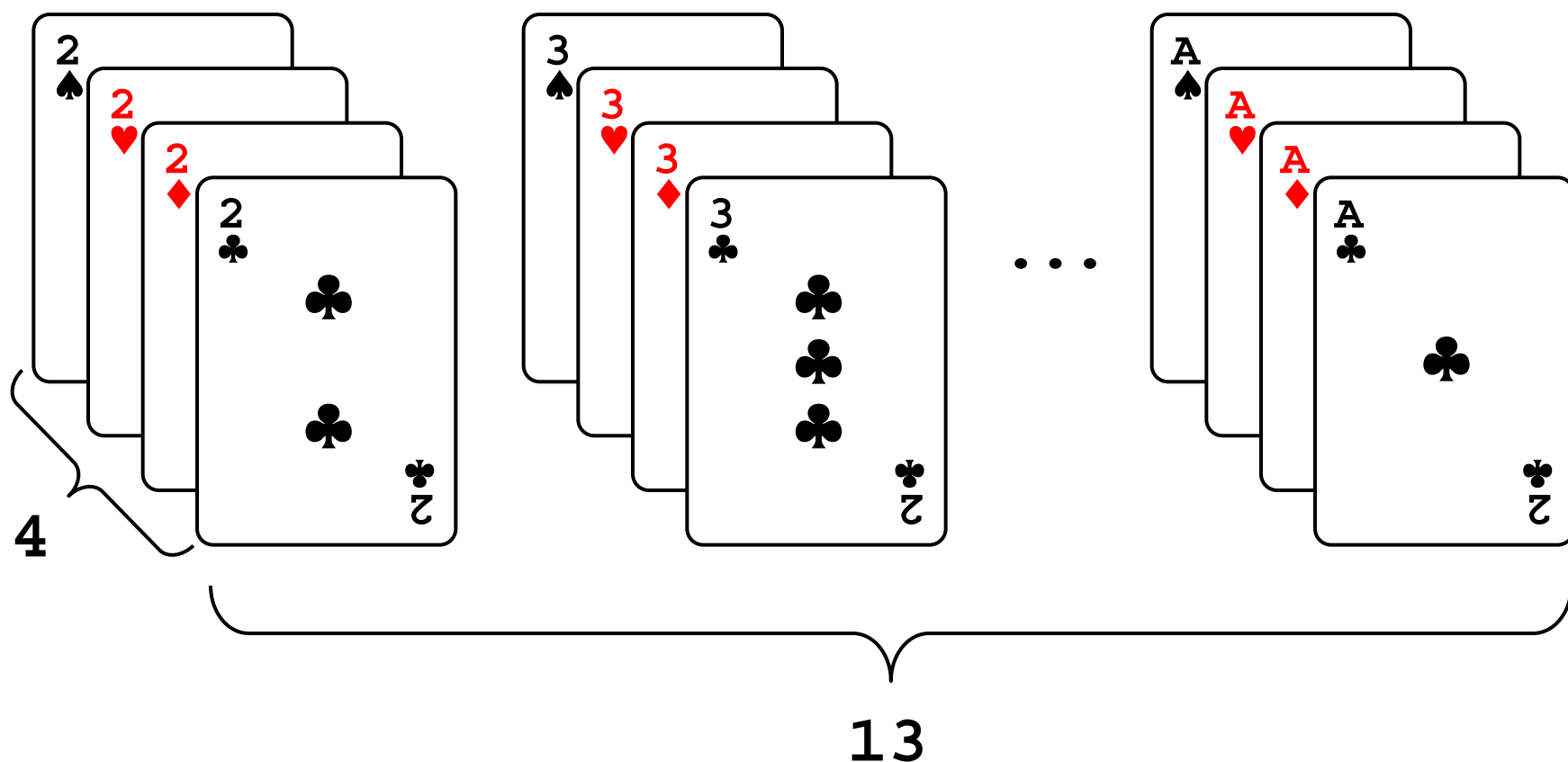
- 在扑克牌中，面值是主关键字，当主关键字相同时再比较次关键字





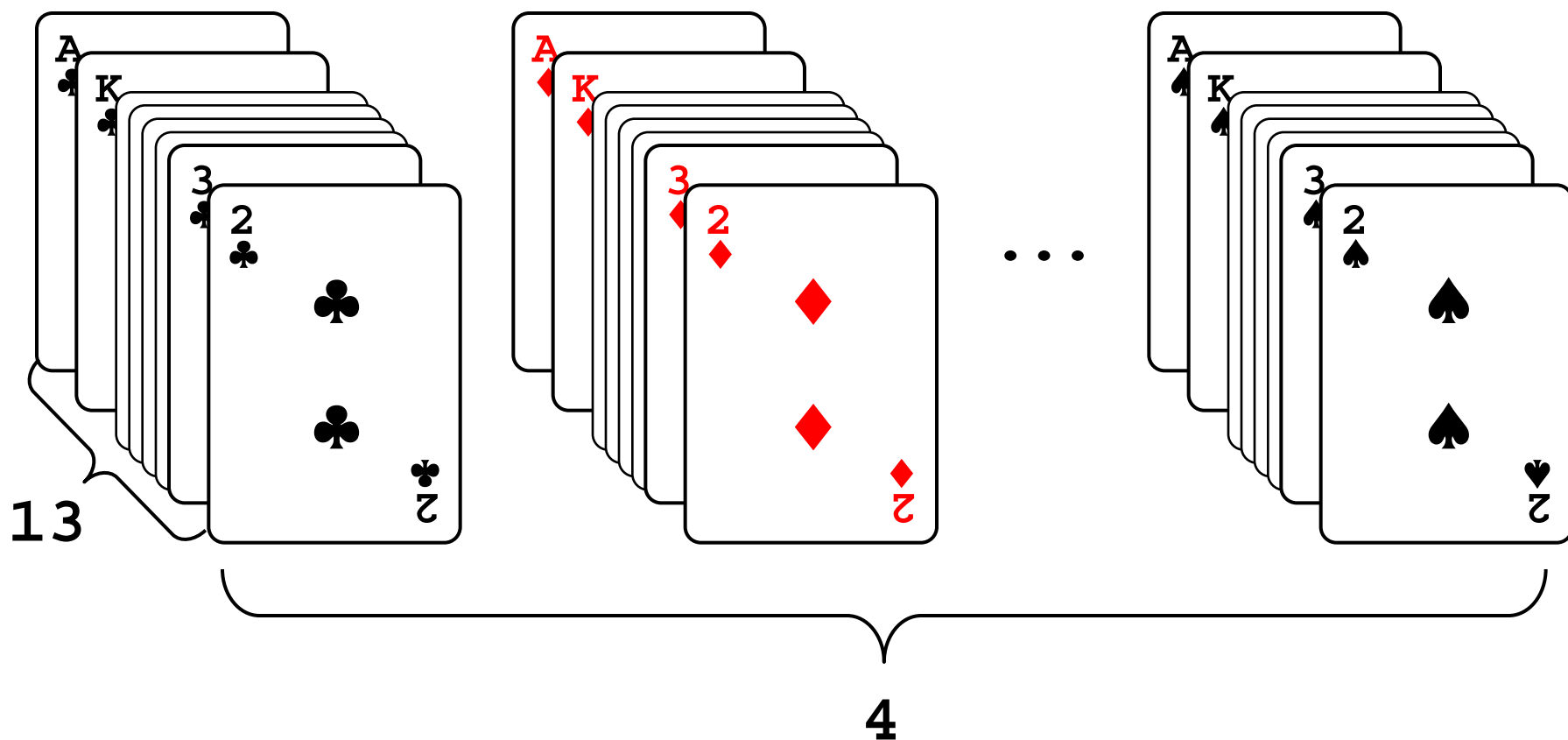
# 基数排序

- 主关键字优先对扑克牌进行排序



# 基数排序

- 次关键字优先对扑克牌进行排序



# 基数排序

## • 扩展一下问题

- 在算术中,  $300 > 299$ , 因为前者的百位数比后者的百位数大
- 也就是我们比较两个数字的大小, 总是先看最高位, 再看次高位, 以此类推
- 因此可以把每一位数看作是一个关键字
- 每个关键字的取值范围是  $0 \sim 9$
- 这里有3个关键字

# 基数排序

- 类似的

- CBAD < CDAB

- 可以把这里的每一位看作一个关键字

- 每一个关键字的取值范围是A~Z

- 这里有4个关键字

# 基数排序

- 链式基数排序算法

- 接下来我们都以关键字为数字来讨论

- 假设对如下记录进行排序：

- 278、109、063、930、589、184、505、  
269、008、083

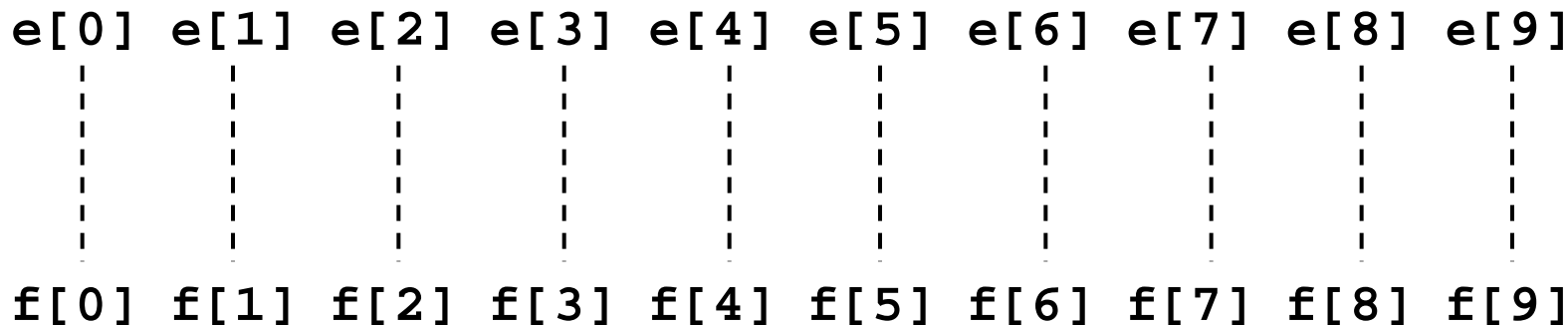
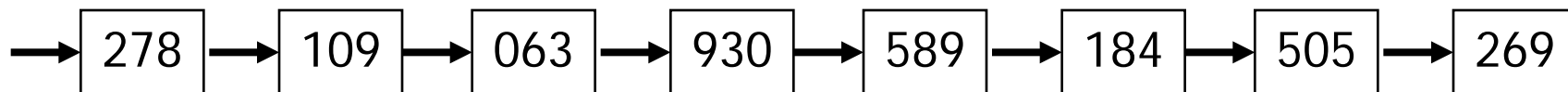
- 注意：

- 这里的一条记录不是一个整数 (`int`)
    - 而是有3个关键字
    - 每个关键字是一个整数而已

# 基数排序

- 基本思想：“分配” + “收集”

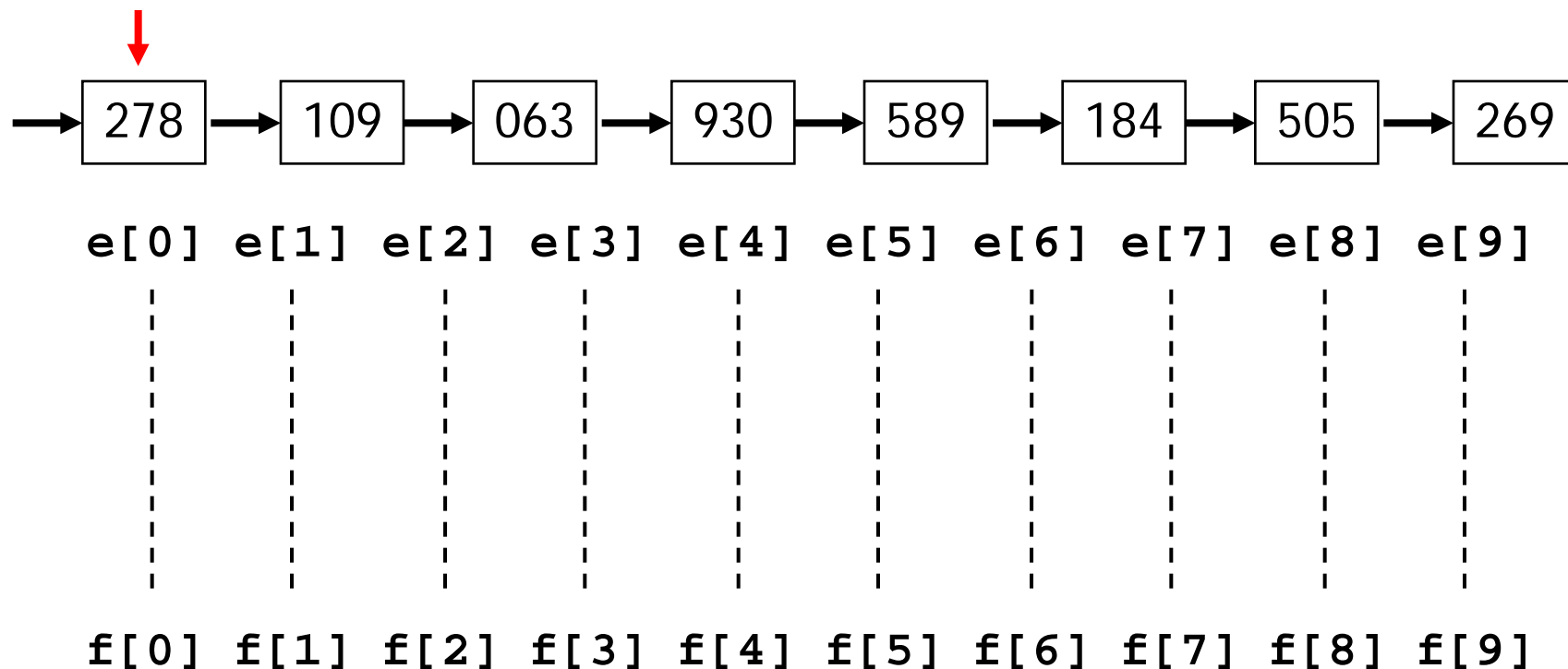
- 首先原始数据被保存在链表中
- 关键字的取值范围是0~9，因此再准备10个队列（也用链表实现）



# 基数排序

- 第1趟：针对个位数

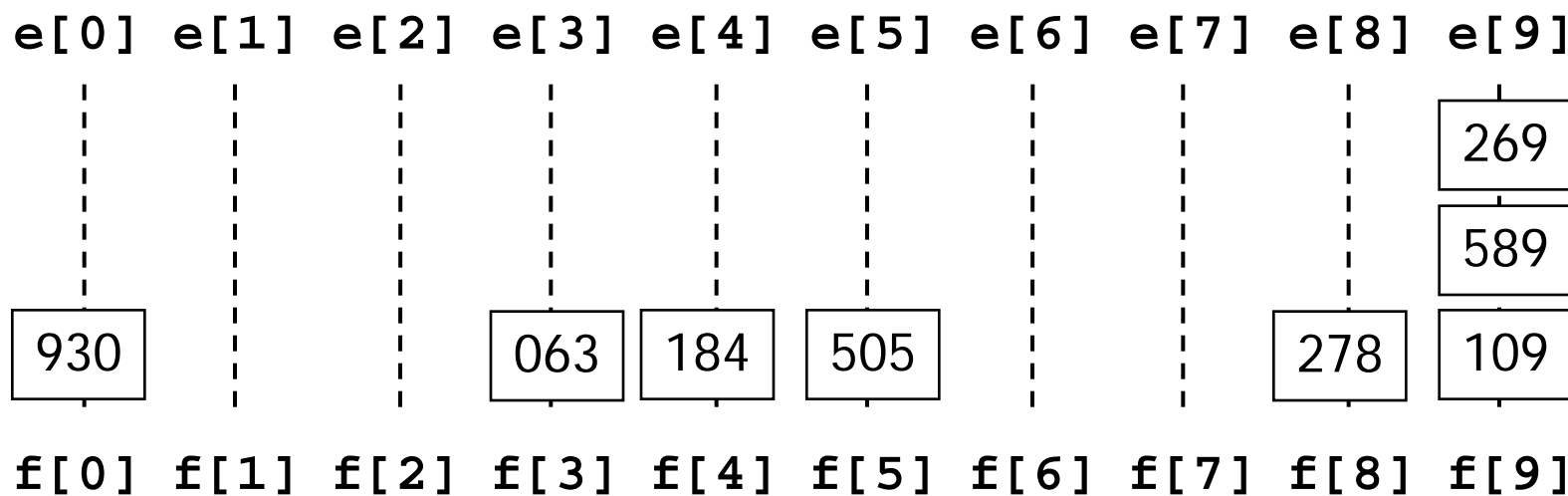
- “分配”：扫描每一个记录，按照个位数分别放到相应的队列中



# 基数排序

- 第1趟：针对个位数

- “收集”：将各队列的记录重新组织成一个链表

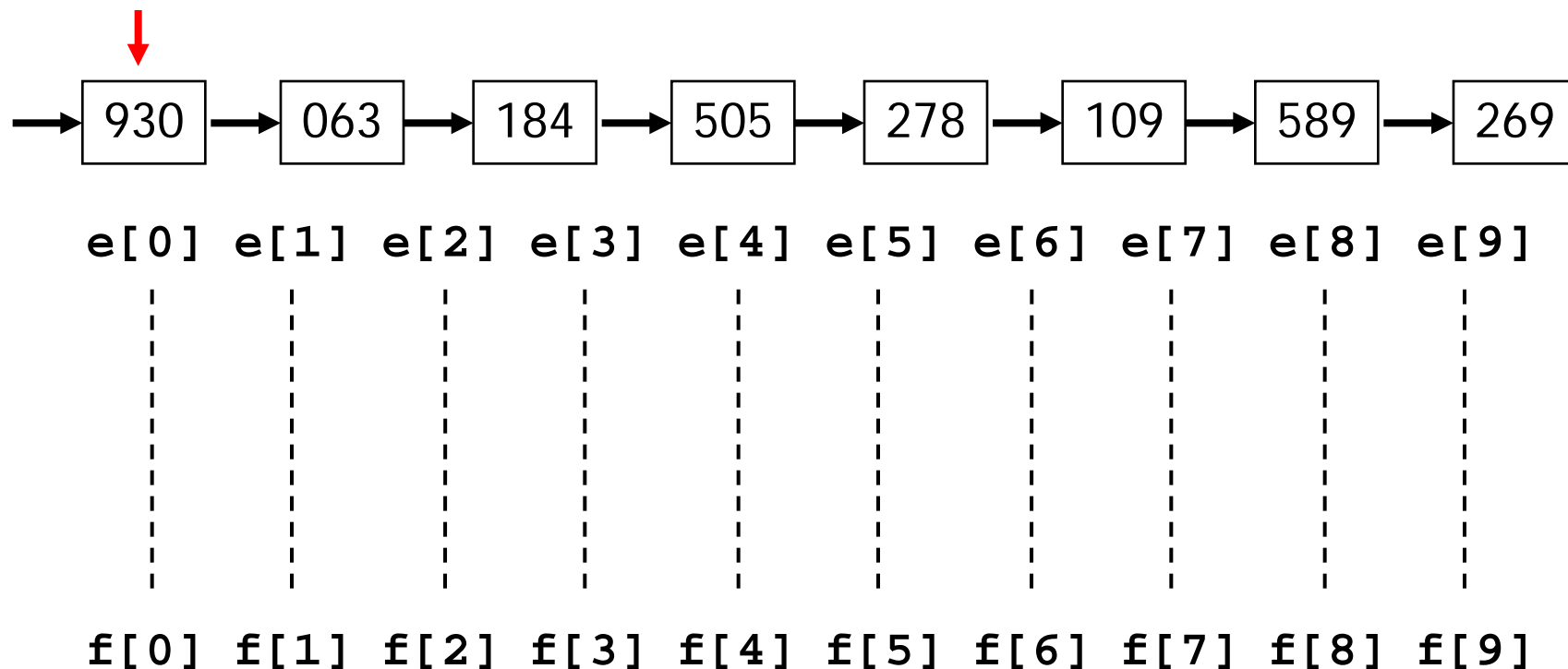




# 基数排序

- 第2趟：针对十位数

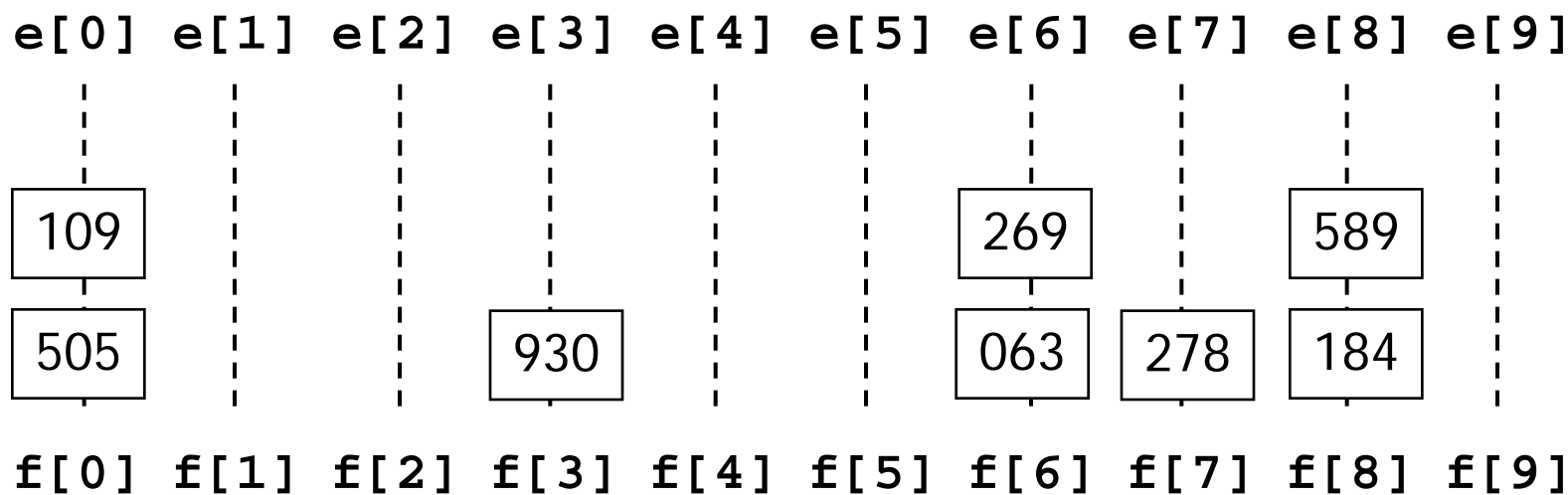
- “分配”：扫描每一个记录，按照十位数分别放到相应的队列中



# 基数排序

- 第2趟：针对十位数

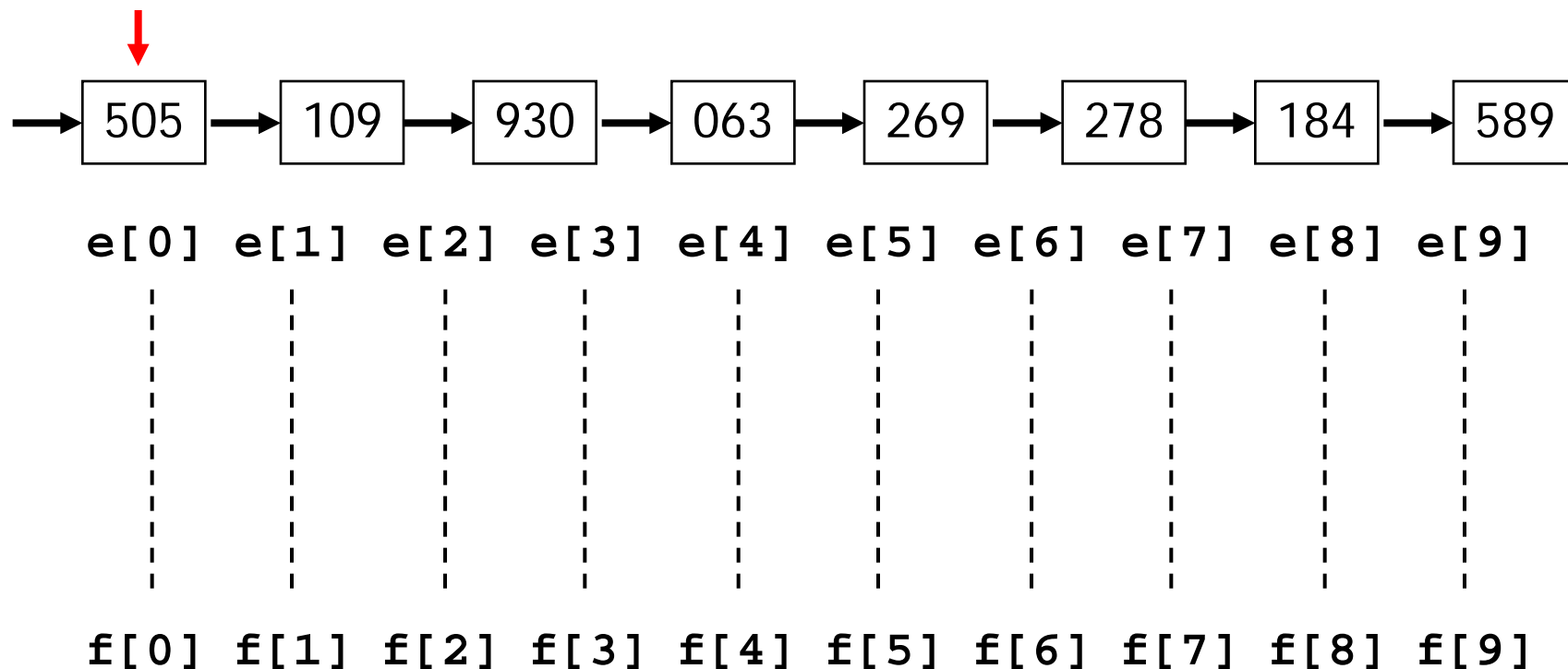
- “收集”：将各队列的记录重新组织成一个链表



# 基数排序

- 第3趟：针对百位数

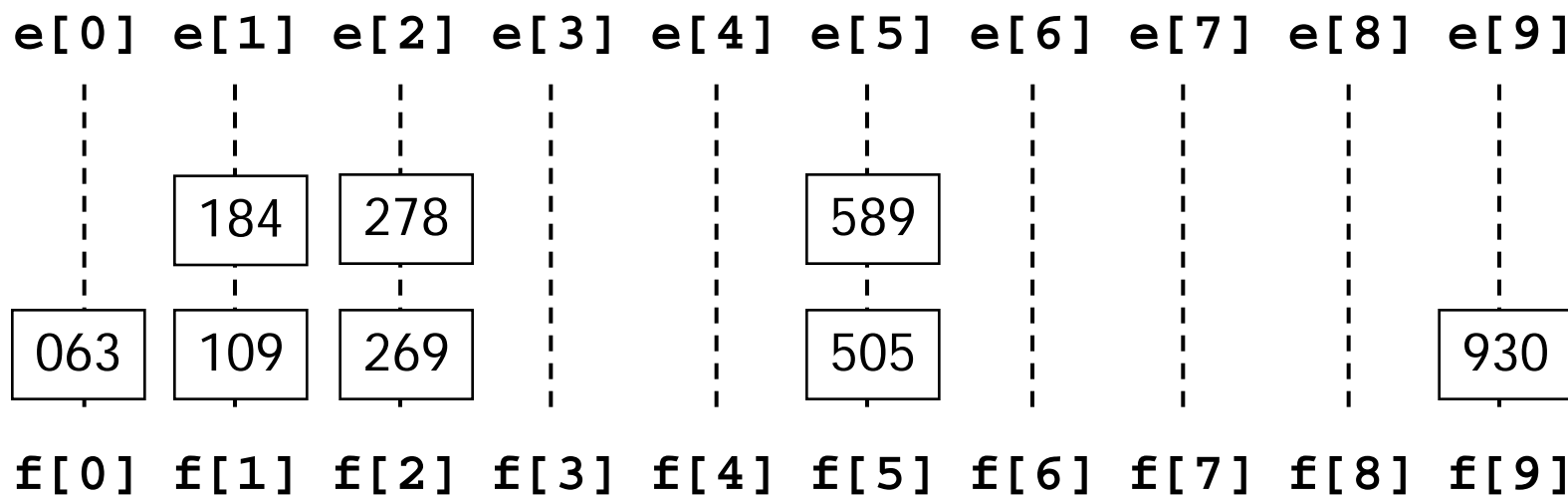
- “分配”：扫描每一个记录，按照百位数分别放到相应的队列中



# 基数排序

- 第3趟：针对百位数

- “收集”：将各队列的记录重新组织成一个链表



# 基数排序

- 具体的算法
  - 详见P288

# 各种内部排序方法的比较

	最好时间	最差时间	平均时间	空间复杂度	稳定性
直接插入	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	√
起泡法	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	√
简单选择	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	×
快速排序	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$	×
堆排序		$O(n \log n)$	$O(n \log n)$	$O(1)$	×
归并排序		$O(n \log n)$	$O(n \log n)$	$O(n)$	√
希尔排序			$O(n^{1.3})$	$O(1)$	×

# 各种内部排序方法的比较

- 从平均时间来看

- 快速排序、堆排序、归并排序处于同一数量级
- 其中以快速排序为最优
  - 但是快速排序最差情况下复杂度较高
- 堆排序和归并排序相比较
  - 归并排序速度较快
  - 但是消耗附加存储空间更多
  - 一般用于外部排序

# 各种内部排序方法的比较

- 简单排序方法

- 包括起泡排序、直接插入、简单选择
- 其中直接插入最简单
  - 当数据基本有序，或 $n$ 很小时最佳
  - 常和其它“复杂排序方法”相结合

- 总之

- 没有一种排序方法是最优的
- 只能根据实际情况进行选择



# 各种内部排序方法的比较

- 算法是否稳定的简单推断

- 通常，“比较”只发生在相邻两个记录之间的排序算法是稳定的
- 有可能对不相邻的两个记录进行交换的算法是不稳定的

# 作业和思考题

- 作业

- 习题集P61: 10.1(1)~(4)
- (2)中的增量 $d[] = \{5, 3, 1\}$