

# 数据结构

# Data Structure

**Linear List**



## 第二章 线性表

### 本章内容

- 2.1 线性表的类型定义**
- 2.2 线性表的顺序表示和实现**
- 2.3 线性表的链式表示和实现**
  - 2.3.1 线性链表**
  - 2.3.2 循环链表**
  - 2.3.3 双向链表**
- 2.4 一元多项式的表示及相加**



## 2.1 线性表的类型定义

定义：一个线性表是有 **$n$** 个数据元素的有限序列：

$$(a_1, a_2, \dots, a_i, \dots, a_n)$$

- ✿ 线性表中元素之间的关系是线性关系：
  - 存在惟一的第一个元素；
  - 存在惟一的最后一个元素；
  - 每个元素均只有一个直接前驱（第一个元素除外）；
  - 每个元素均只有一个直接后继（最后一个元素除外）。
- 一个线性表中元素的个数 **$n(n \geq 0)$** 定义为线性表的长度， **$n=0$** 时称为空表。非空线性表中的每个元素都有一个确定的位序。



## 2.1 线性表的类型定义

### ■ 线性表的抽象数据类型定义:

**ADT List {**

数据对象:  $D = \{ a_i \mid a_i \in \text{ElemSet}, i=1,2,\dots,n, n \geq 0 \}$

数据关系:  $R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

基本运算:

**InitList(&L);**

**DestroyList(&L);**

**Length(L);**

**GetElem(L,i,&e);**

**LocateElem(L,e,compare());** //返回L中第1个与e满足

//关系**compare()**的数据元素的位序

//若这样的元素不存在, 则返回值为**0**

**InsertElem(&L,i,e);** //在L的第i个元素之前插入新元素e

**DeleteElem(&L,i,&e);**

.....

**} ADT List**



## 2.1 线性表的类型定义

### ■ 抽象运算（算法2.1）

**例2-1** 利用两个线性表**LA**和**LB**分别表示集合**A**和**B**，求一个新的集合 **$A = A \cup B$** 。

**算法思路：**集合中元素间是松散的关系（对元素没有次序要求），对于**LB**中的每个元素，在**LA**中查找是否存在与其相同者，若不存在，则将其加入到**LA**中。

**算法：**

```
void union(List &La, List &Lb) {  
    La_len = Length(La); Lb_len = Length(Lb);  
    for(i = 1; i <= Lb_len; i++) {  
        GetElem(Lb, i, e);  
        if (!LocateElem(La, e, equal))  
            InsertElem(La, ++La_len, e);  
    }  
} //union
```



## 第二章 线性表

### 本章内容

**2.1** 线性表的类型定义

**2.2** 线性表的顺序表示和实现

**2.3** 线性表的链式表示和实现

**2.3.1** 线性链表

**2.3.2** 循环链表

**2.3.3** 双向链表

**2.4** 一元多项式的表示及相加



## 2.2 线性表的顺序表示和实现

### ■ 顺序表---线性表的顺序存储

内涵:

线性表的顺序存储指用**一组地址连续的存储单元**依次存储线性表的数据元素。这称为**顺序表**。

特点:

- 存储单元地址连续（需要一段连续空间）
- 逻辑上相邻的数据元素其物理位置也相邻
- 存储密度大（存储空间全部用于存储元素）
- 随机存取

元素序号与存储位置存在如下关系:

$a_1$	}d
$a_2$	
$\vdots$	
$a_i$	
$\vdots$	
$a_n$	

$$\text{Loc}(a_i) = \text{Loc}(a_1) + (i - 1) * d \quad (1 \leq i \leq n)$$



//-----静态分配（静态数组）-----

```
typedef struct {  
    ElemType  elem[INIT_SIZE]; //存储空间，容量为100  
    int  length;           //表长(元素个数)  
} SqList;
```

//-----数组空间动态分配-----

```
#define INIT_SIZE 100 //空间初始分配量  
#define INCREMENT 10 //增量  
typedef struct{  
    ElemType *elem; //存储空间首地址  
    int      length; //表长(元素个数)  
    int      listsize; //存储容量  
}SqList;
```

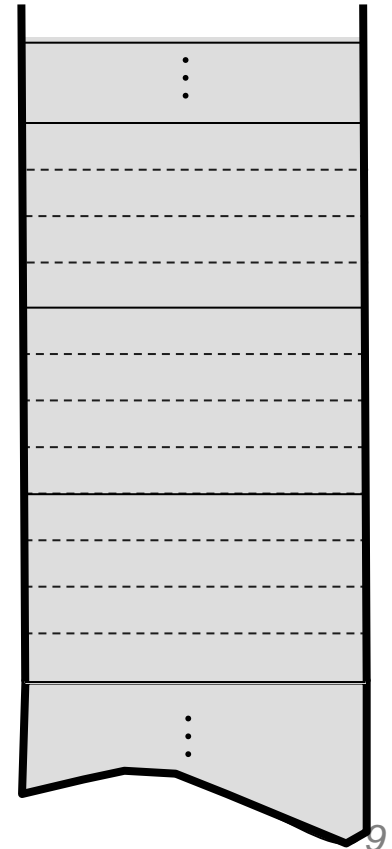




## 2.2 线性表的顺序表示和实现

### ■ 创建一个空的顺序表（算法2.3）

```
Status InitList_sq(SqList &L)  {  
    L.elem = (ElemType *)malloc(INIT_SIZE*sizeof(ElemType));  
    if (!L.elem)  
        return OVERFLOW;  
    L.length = 0;  
    L.listsize = INIT_SIZE;  
    return OK;  
} // InitList_sq
```





## 2.2 线性表的顺序表示和实现

### ■ 线性表上的基本运算

#### ※ 插入运算

将元素 $e$ 插入到线性表:  $(a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$ 中, 构成新的线性表 $(a_1, a_2, \dots, a_{i-1}, e, a_i, \dots, a_n)$ , 满足 $a_{i-1} \leq e \leq a_i$ , (其中 $\leq$ 为比较关系), 即不破坏原线性关系。

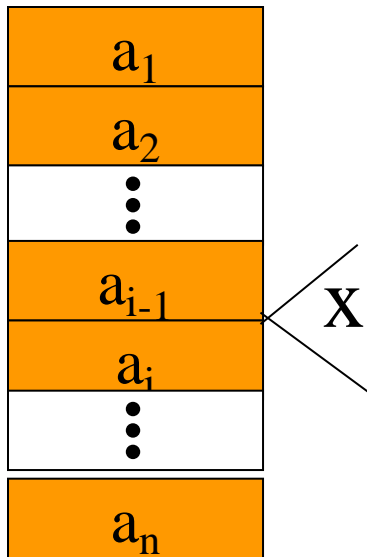
#### ※ 表的长度为 $n+1$

※ 将元素 $e$ 插入到元素 $a_{i-1}$ 之后,  $a_{i-1}$ 的直接后继和 $a_i$ 的直接前驱就改变了, 需要在顺序表的存储结构上反映出来。

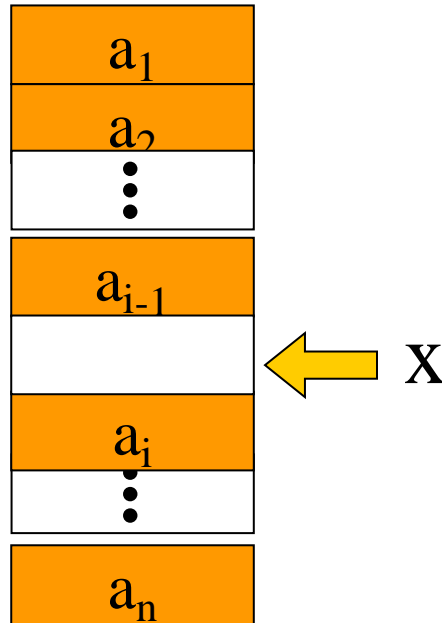


## 2.2 线性表的顺序表示和实现

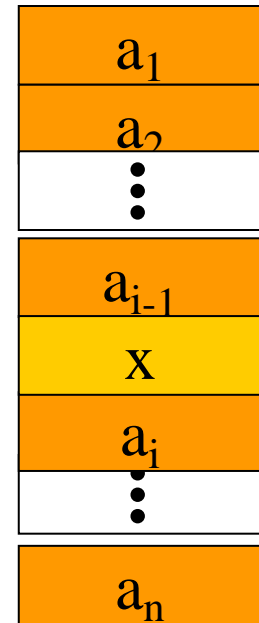
- 顺序表上插入运算的实现:



在线性表L的第i个位置上插入元素x。



将元素 $a_i \sim a_n$  ( $n-i+1$ 个元素) 依次向后移动一个位置, 腾出要放置x的单元。



将元素x放在腾出的第i个空位上。



## 2.2 线性表的顺序表示和实现

### ■ 顺序表上的插入运算

```
Status ListInsert_sq(SqList &L, int i, ElemType x) {
```

判断参数是否合法合理，是则继续，否则终止算法；

判断存储空间是否够用，若不足则进行扩充，扩充成功后继续，否则终止算法；

在物理空间中查找插入位置；

移动相关元素（相关元素依次后移）并插入新元素；

善后工作（修改表长），结束算法

```
} // ListInsert_sq
```



## 2.2 线性表的顺序表示和实现

- 用类C语言实现顺序表的插入

```
Status ListInsert_sq(SqList &L, int i, ElemType x) {  
    //将元素x插入到线性表L的第i个元素位置上  
    if (i < 1 || i > L.length+1) return ERROR; //判定输入是否合法  
    if (L.length >= L.listsize) { //判定空间是否够用  
        newbase = (ElemType *)realloc(L.elem,  
            (L.listsize+INCREMENT)*sizeof(ElemType));  
        if (!newbase) return OVERFLOW;  
        L.elem = newbase;  
        L.listsize += INCREMENT; //修改顺序表的参数  
    }  
    插入元素的相关处理; // 见下页  
}//ListInsert_sq
```



## 2.2 线性表的顺序表示和实现

- 用类C语言实现顺序表的插入

```
Status ListInsert_sq(SqList &L, int i, ElemType x){
```

```
//将元素x插入在线性表L的第i个元素位置上
```

```
参数的合法性检查及存储空间的处理; //上一页
```

```
q = &(L.elem[i-1]); //直接定位插入位置
```

```
//q为插入位置，数组元素的下标从0开始
```

```
for(p = &(L.elem[L.length-1]); p >= q; --p)
```

```
    *(p+1)=*p; //移动元素并进行插入处理，可改用数组下标实现
```

```
*q = x;
```

```
L.length++;
```

```
return OK;
```

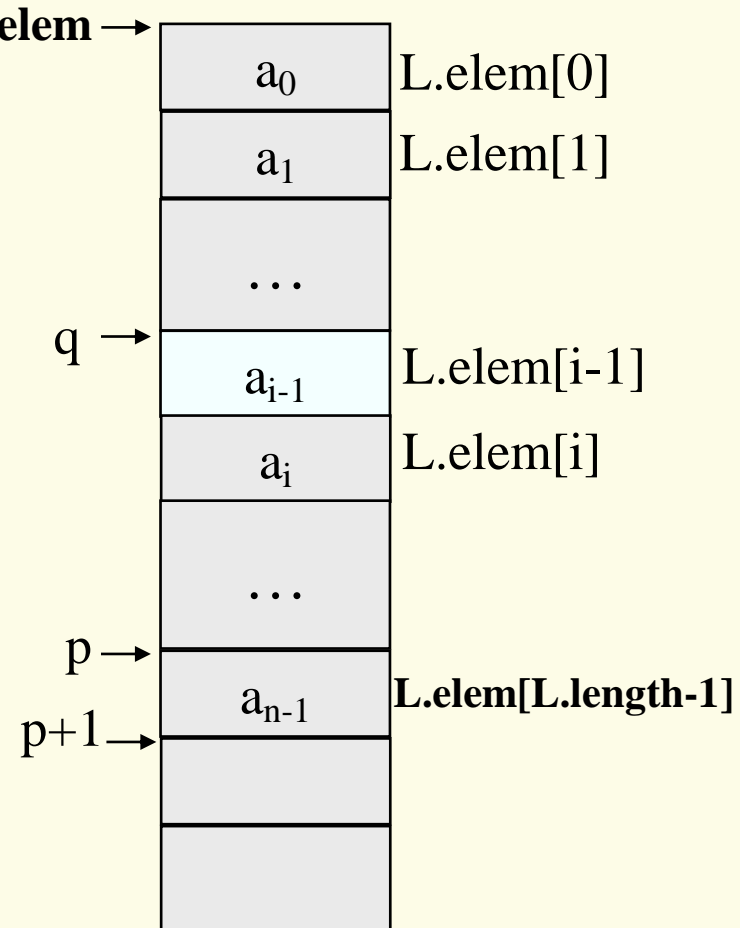
```
}//ListInsert_sq
```



## 2.2 线性表的顺序表示和实现

### ■ 用类C语言实现顺序表的插入

```
Status ListInsert_sq(SqList &L, int i, ElemType x) {  
    if (i<1 || i>L.length+1) return ERROR;  
    if (L.length>=L.listsize) {  
        newbase = (ElemType *)re  
            (L.listsize+INCREMEN  
        if (!newbase) return OVERI  
        L.elem = newbase;  
        L.listsize += INCREMENT;  
    }  
    q = &(L.elem[i-1]); //q为插入位置,  
    for(p=&(L.elem[L.length-1]); p>=  
        *(p+1)=*p;  
        L.length++;  
    return OK;  
} //ListInsert_sq
```





## 2.2 线性表的顺序表示和实现

### ■ 顺序表上插入运算效率分析:

#### 时间复杂度

从算法流程上看, 判断参数的合法性、查找插入位置等消耗的是常数时间, 主要的时间消耗在插入新元素时对相关元素的移动上, 显然, 需移动的元素个数与插入位置有关。

- 最好情况: 在表尾插入, 不移动元素,  $T(n)=O(1)$
- 最坏情况: 在表头插入, 移动 $n$ 个元素,  $T(n)=O(n)$
- 平均情况:

设 $p_i$ 为在第 $i$ 个元素之前插入一个元素的概率, 且

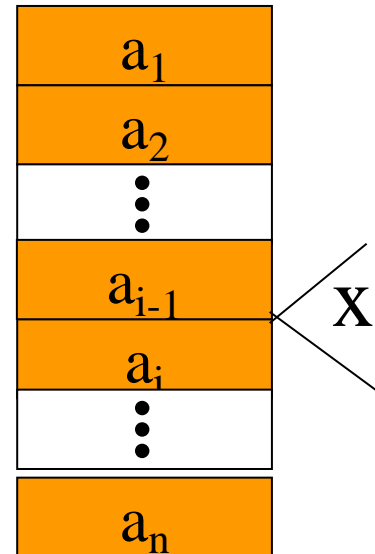
$P_1=P_2=\dots=P_i=\dots=P_n=1/(n+1)$ , 则平均移动次数为:

$$E_{is} = \sum_{i=1}^{n+1} P_i (n - i + 1) = 1/(n+1) \sum_{i=1}^{n+1} (n - i + 1) = n/2$$

即 $T(n) = O(n)$ 。

#### 空间复杂度

需要的额外单元数为常量,  $S(n) = O(1)$







## 2.2 线性表的顺序表示和实现

### ■ 线性表上的基本运算

#### ※ 删除运算

将元素 $a_i$ 从线性表:  $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 中删除, 构成新的线性表 $(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$ , 删除后不破坏原线性关系。

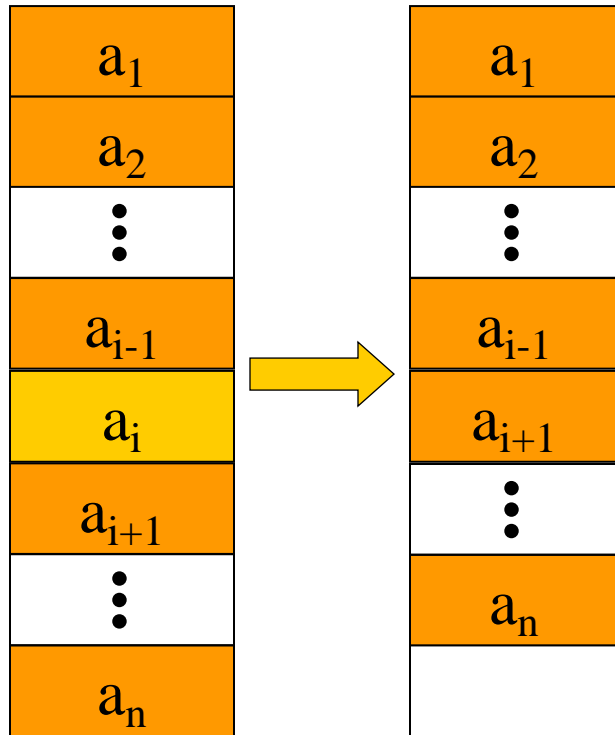
#### ※ 表的长度为 $n-1$

※ 将元素 $a_i$ 删除之后,  $a_{i-1}$ 的直接后继和 $a_{i+1}$ 的直接前驱就改变了, 需要在顺序表的存储结构上反映出来。



## 2.2 线性表的顺序表示和实现

### ■ 顺序表上删除运算的实现:



```
Status ListDelete_Sq(SqList &L, int i,
ElemType &e) {
    判断参数是否合法合理,
    是则继续, 否则终止算法;
    在物理空间中查找待删
    除的元素;
    删除处理 (将后续元素
    依次前移);
    善后工作 (修改表长),
    结束算法
} // ListDelete_Sq
```

请同学们自行分析顺序表的删除算法及时间、空间复杂度。



# 顺序表小结

## ■ 顺序表的优缺点

### ■ 优点:

- 不需要额外的存储空间来表示元素间的逻辑关系
- 可以随机地存取表中的任意一个元素

### ■ 缺点:

- 插入和删除元素时要移动大量的元素
- 必须事先分配足够的空间

## ■ 顺序表的基本运算的复杂度

### ■ 插入

- $T(n)=O(n)$  ,  $S(n)=O(1)$

### ■ 删除

- $T(n)=O(n)$  ,  $S(n)=O(1)$

$a_1$
$a_2$
$\vdots$
$a_{i-1}$
$a_i$
$a_{i+1}$
$\vdots$
$a_n$



## 小结

### ■ 用类C语言实现顺序表的插入

```
Status ListInsert_sq(Sqlist &L, int i, ElemType x) {  
    if (i<1 || i>L.length+1) return ERROR;  
    if (L.length>=L.listsize) {  
        newbase = (ElemType *) malloc((L.listsize+INCREMENT)*sizeof(ElemType));  
        if (!newbase) return OVERFLOW;  
        L.elem = newbase;  
        L.listsize += INCREMENT;  
    }  
    q = &(L.elem[i-1]); //q为插入位置  
    for(p=&(L.elem[L.length-1]); p>q; p--) *p = *q;  
    *q = x; L.length++;  
    return OK;  
} //ListInsert_sq
```

### 类C语言书写的算法与C程序之间的差别:

- (1) 算法中除形式参数外,变量不作定义, 在程序中必须定义;
- (2) 算法中使用的元素类型(ElemType)没有定义, 程序中必须定义, 符号OK、ERROR、OVERFLOW等需要定义;
- (3) 算法中的比较运算符(equal、less)未作定义, 程序中必须定义;
- (4) 必要的头文件(例如stdio.h、stdlib.h), 在程序中必须包含。



## 第二章 线性表

### 本章内容

**2.1** 线性表的类型定义

**2.2** 线性表的顺序表示和实现

**2.3** 线性表的链式表示和实现

**2.3.1** 线性链表

**2.3.2** 循环链表

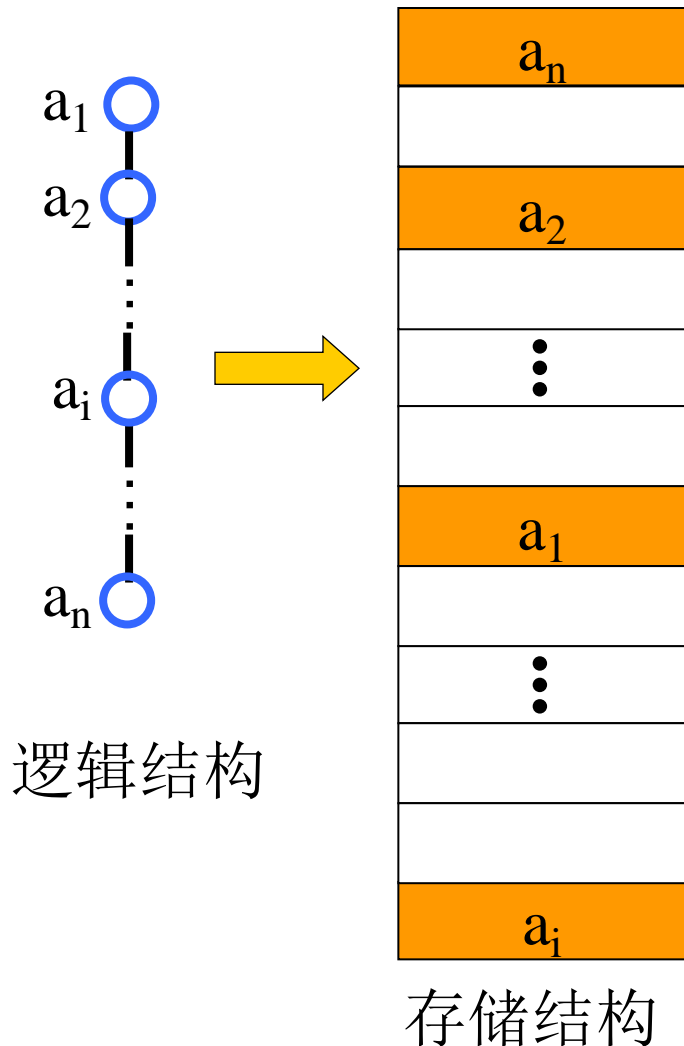
**2.3.3** 双向链表

**2.4** 一元多项式的表示及相加



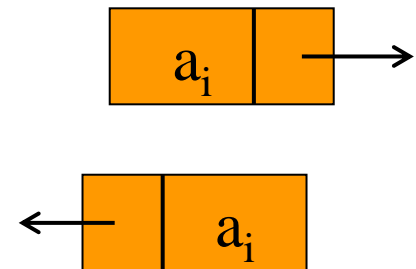
## 2.3 线性表的链式表示和实现

- 线性表的链式存储：元素的存储位置不要求相邻



### 实现方式

- ✿ 对每一个元素的存储单位进行扩充，在存储元素值的同时，还在结点中存储与当前元素有直接关系的其他元素(直接前趋 / 直接后继)的存储单位的地址(指针)。





## 2.3 线性表的链式表示和实现

### ■ 链表---线性表的链式存储

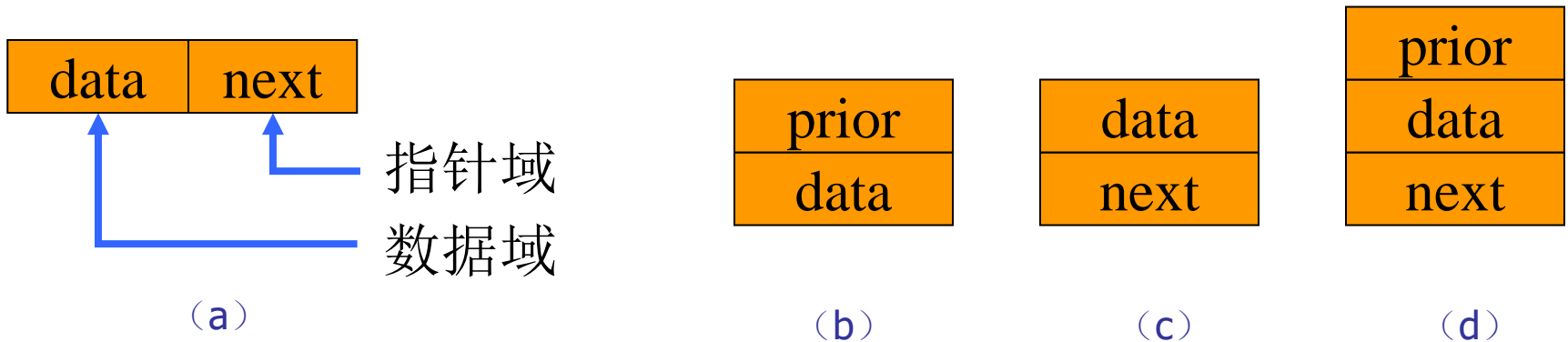
#### ■ 内涵：

线性表的链式存储指用任意的存储单元存放线性表中的元素，每个元素与其直接前驱和（或）直接后继之间的关系用指针来存储。这称为**链表**。

#### ■ 术语

结点：数据元素及与其有直接关系的元素的地址构成的存储单位。

指针：指示出当前结点的直接前驱（或后继）结点



链表结点结构示意图



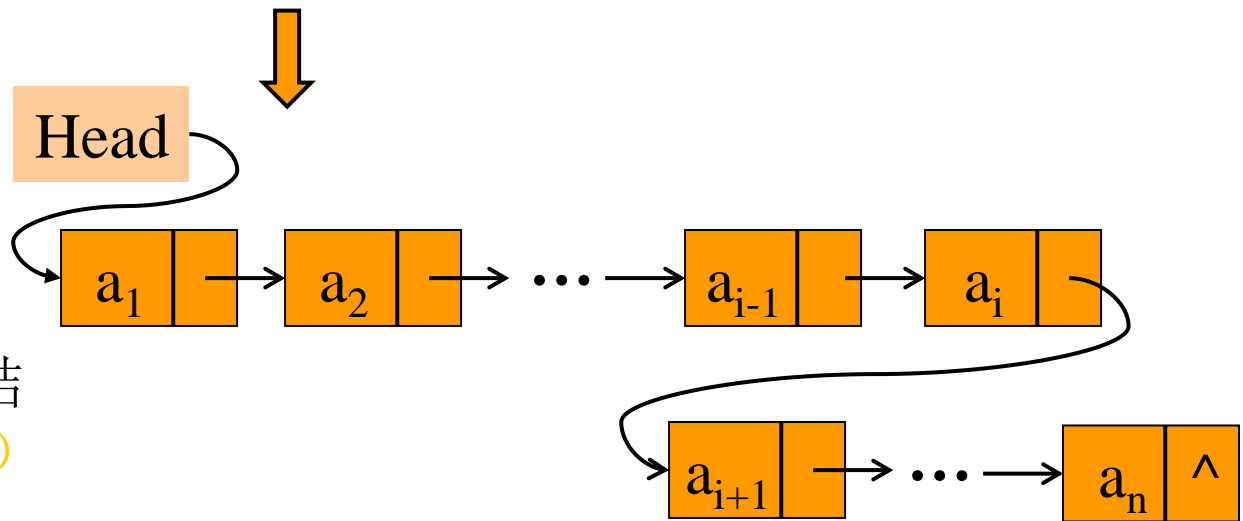
## 2.3.1 单链表的表示和实现

### ■ 单链表

链表中，如果每个结点中只包含一个指针域，则称之为线性链表或单链表。

\* 一般情况下，单链表中的每个结点存储其直接后继结点的位置信息。

线性表  $(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$



指向单链表第一个结点的头指针（Head）可确定一个单链表。

单链表示意图





# 线性表的顺序存储和单链表存储

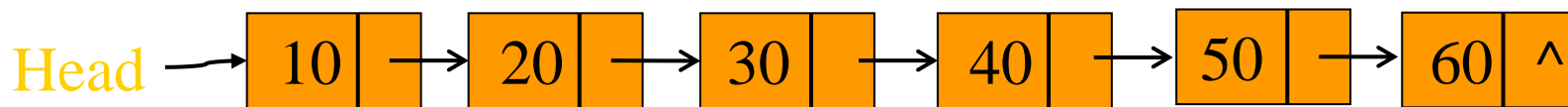
☞ 顺序表：线性表的顺序存储

☞ 单链表：线性表的（一种）链式存储

例如， 线性表  $L = (10, 20, 30, 40, 50, 60)$

10	La.elem[0]
20	La.elem[1]
30	
40	
50	La.elem[4]
60	La.elem[5]

顺序表



单链表

作业



## 2.3.1 单链表的表示和实现

### ■ 单链表的C语言实现

//-----**结点类型定义**-----

```
typedef struct LNode{  
    ElemType data;  
    struct LNode *next;  
} LNode, *LinkList;
```

访问变量a:

```
a.data    // ElemType类型变量  
a.next    // 指针变量
```

令p指向变量a:

```
p = &a;  
p->data    // 等同于a.data, (*p).data  
p->next    // 等同于a.next, (*p).next
```

令p指向动态申请的结点:

```
p = (LNode *)malloc(sizeof(LNode));
```

若申请成功，p得到新结点的地址（指针，或称指向新生成的结点）；否则p得到空指针值（NULL）。

```
LNode a; //定义结构体变量a  
LinkList p; //定义指向结构体变量  
//等同于 LNode *p;
```



## 第二章 线性表

### 本章内容

**2.1 线性表的类型定义**

**2.2 线性表的顺序表示和实现**

**2.3 线性表的链式表示和实现**

**2.3.1 线性链表**

**2.3.2 循环链表**

**2.3.3 双向链表**

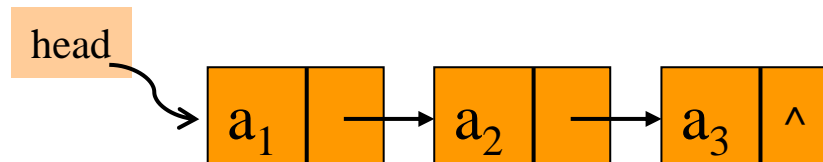
**2.4 一元多项式的表示及相加**



# 内容回顾

## ■ 主要概念

- 结点、指针
- 头指针
- 线性链表（单链表）



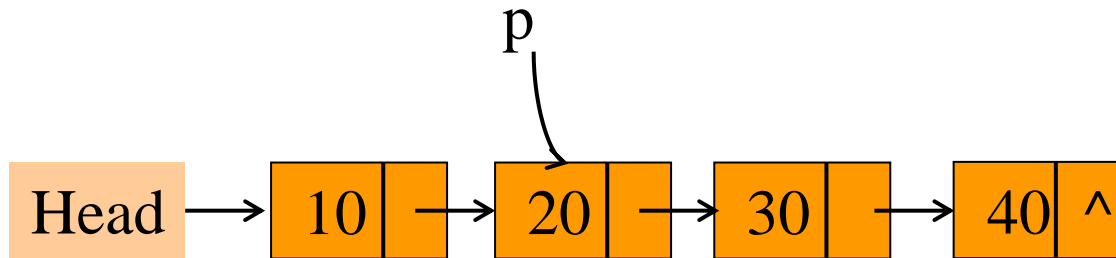
☞ 单链表的C语言实现

//-----**结点类型定义**-----

```
typedef struct LNode{  
    ElemType data;    //数据域  
    struct LNode *next; //指针域  
} LNode, *LinkList;
```



## 2.3.1 单链表的表示和实现



长度为4的单链表

$p$ 、 $p \rightarrow next$ 是两个不同的指针变量

$p = p \rightarrow next$ ; //  $p$ 改为指向下一个结点

$p \rightarrow next = p$ ; //结点的指针域指向结点自己

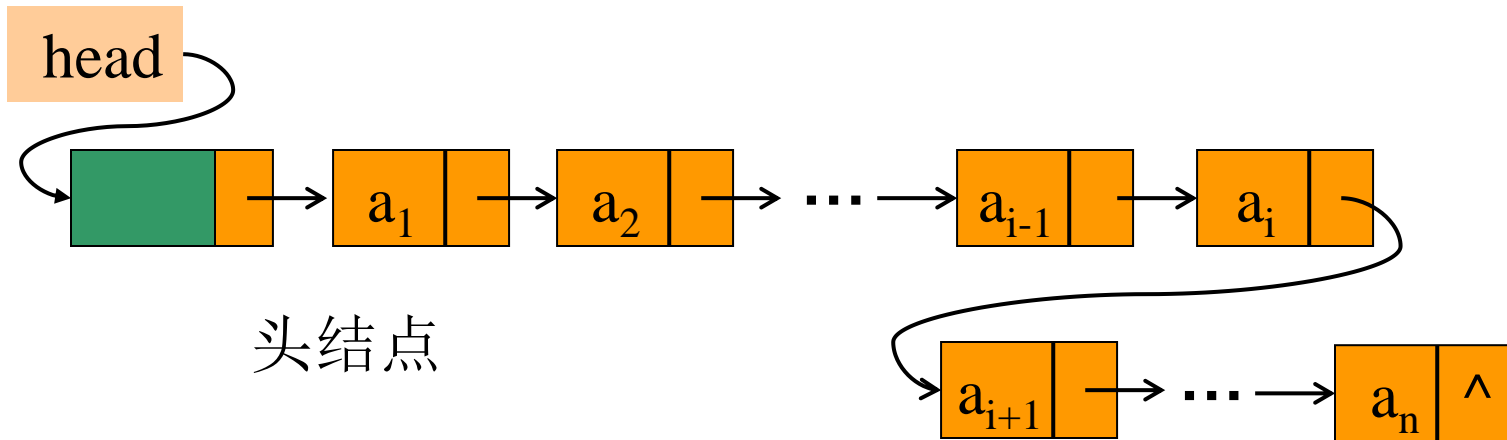
$p = p \rightarrow next \rightarrow next$ ;

$p \rightarrow next = p \rightarrow next \rightarrow next$ ;



## 2.3.1 单链表的表示和实现

### ■ 带头结点的单链表

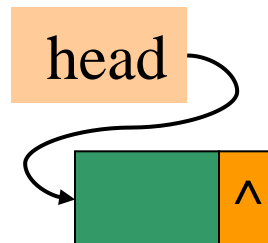


头结点

单链表示意图

**头结点**是链表的第一个结点，其指针域指向第一个元素结点，其数据域可以存储链表的其他信息，如链表的长度、链表的说明等信息，也可以不存储任何信息。

### ☞ 带头结点的空链表

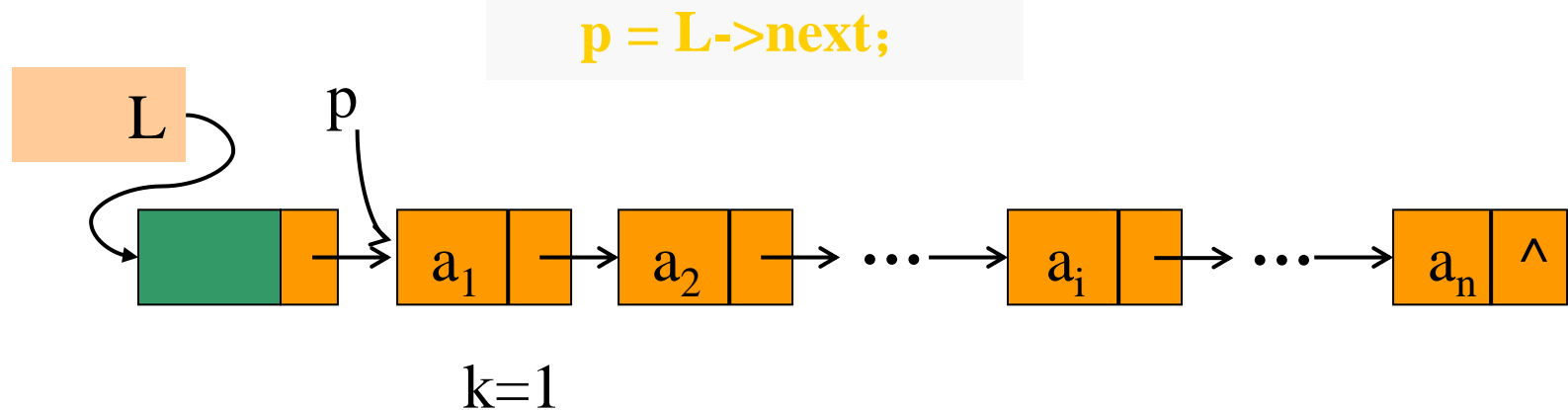




## 2.3.1 单链表的表示和实现

### ■ 单链表上的查找运算

- ※ 在单链表中，必须从头指针出发进行查找，可以是查找第 $i$ 个元素，或者是查找指定的元素是否在表中，等等
- ※ 在单链表上查找第 $i$ 个元素的过程为：设置一个指针 $p$ ，使其按逻辑关系确定的顺序，依次指向第一个元素、第二个元素、...，直到第 $i$ 个元素为止



单链表示意图



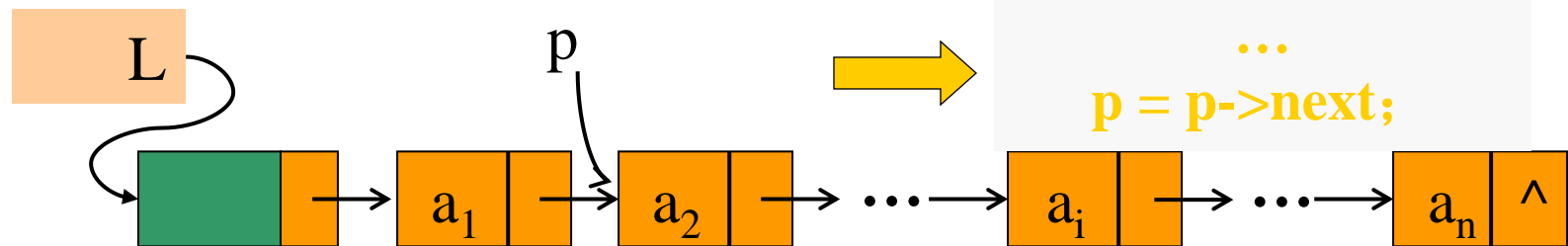
## 2.3.1 单链表的表示和实现

### ■ 单链表上的查找运算

- 在单链表中，必须从头指针出发进行查找，可以是查找第*i*个元素，或者是查找指定的元素是否在表中，等等
- 在单链表上查找第*i*个元素的过程为：设置一个指针

**p**

，使其按逻辑关系确定的顺序，依次指向第一个元素、第二个元素、...，直到第*i*个元素为止



$k=2$

单链表示意图

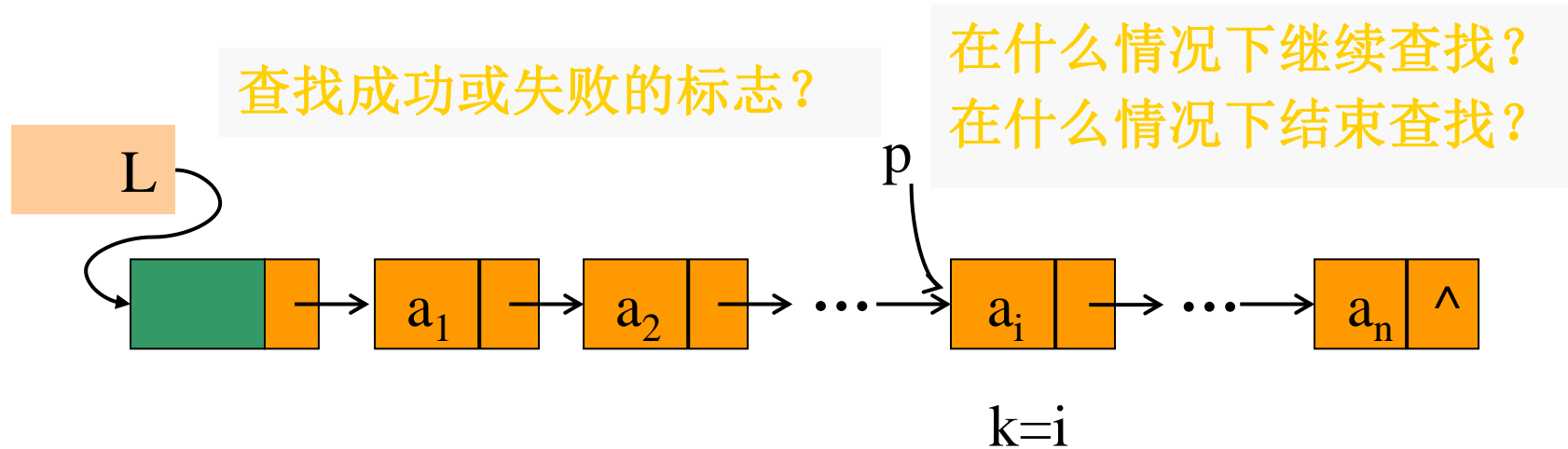




## 2.3.1 单链表的表示和实现

### ■ 单链表上的查找运算

- 在单链表中，必须从头指针出发进行查找，可以是查找第 $i$ 个元素，或者是查找指定的元素是否在表中，等等
- 在单链表上查找第 $i$ 个元素的过程为：设置一个指针 $p$ ，使其按逻辑关系确定的顺序，依次指向第一个元素、第二个元素、...，直到第 $i$ 个元素为止，结果为查找成功或失败。

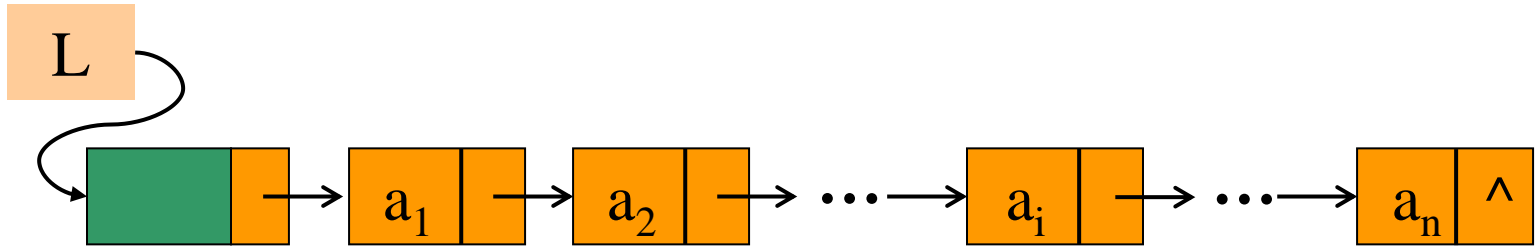


单链表示意图



## 2.3.1 单链表的表示和实现

- 用类C语言实现单链表上的查找第i个元素的运算:



```
Status GetElem_L(LinkList L, int i, ElemType &e)
```

```
{//L是带头结点的单链表的头指针，查找第i个元素
```

```
//若第i个元素存在，则该元素的值由参数e带回，函数返回OK，否则返回ERROR
```

```
    p = L->next; k = 1; //初始化，p指向第一个元素结点，k用于计数
```

```
    while (p && k < i) { //借助p查找，直到p指向第i个元素结点或p变为空指针
```

```
        p = p->next; ++k;
```

```
    } //循环的退出条件为k=i或到达单链表结尾(P为空)
```

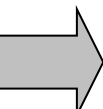
```
    if (!p || k > i) return ERROR; //第i个元素不存在
```

```
    e = p->data;
```

```
    return OK;
```

```
}//GetElem_L
```

问题：算法的时间复杂度？空间复杂度？





## 2.3.1 单链表的表示和实现

### ■ 单链表上的插入运算(在第 $i$ 个位置上插入新的结点)

在单链表中插入一个新元素的处理步骤:

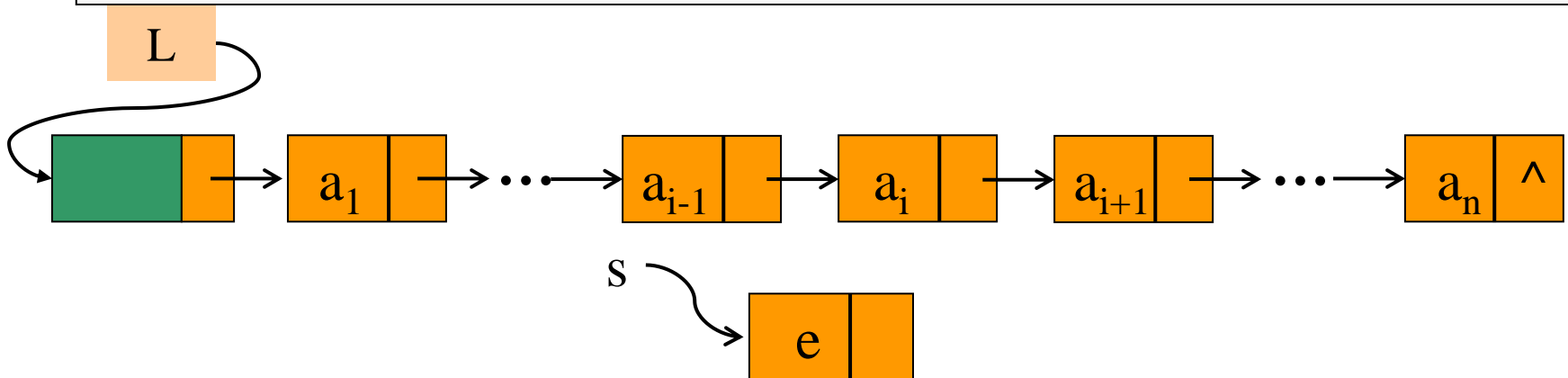
第一步 构造一个新结点, 存入元素的值 $e$ ;

第二步 将新结点插入链表中(即修改相关结点的链接关系):

2.1 查找第 $i$ 个元素( $a_i$ )的直接前驱结点(即 $a_{i-1}$ 所在结点);

2.2 修改结点中的指针域, 使 $e$ 的直接后继为 $a_i$ ;

2.3 使 $a_{i-1}$ 的直接后继为 $e$ 。



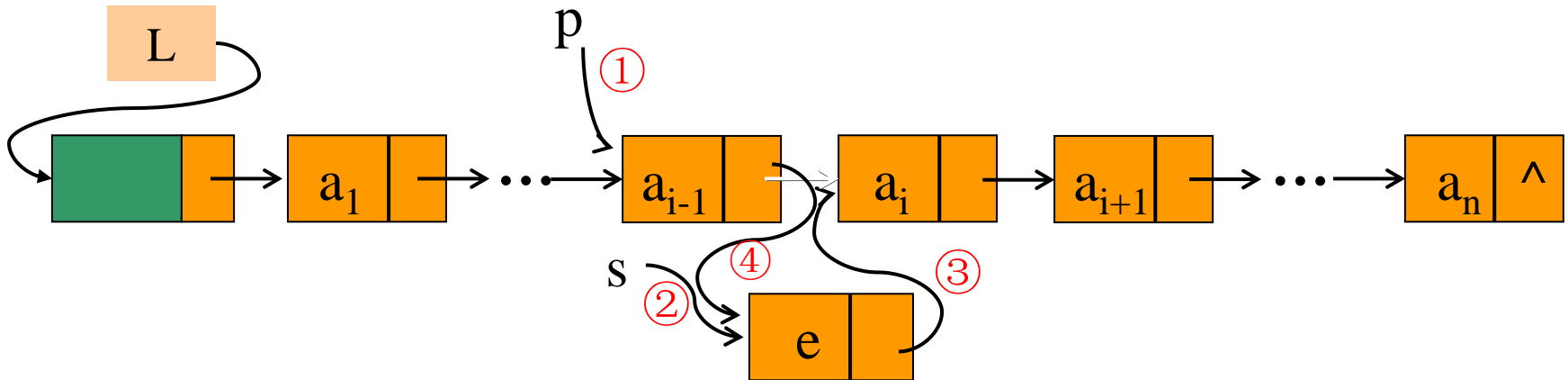


## 2.3.1 单链表的表示和实现

### ■ 单链表上的插入运算(在第 $i$ 个位置上插入新的结点)

在单链表中插入一个新元素的处理步骤:

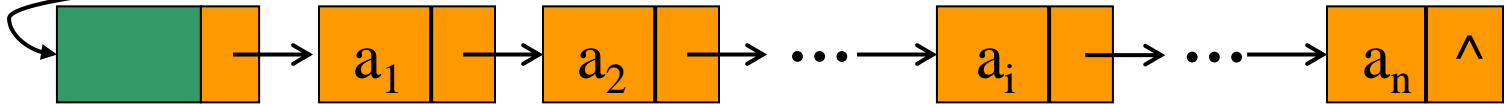
- ① 查找第 $i$ 个元素 ( $a_i$ ) 的直接前驱结点 (即 $a_{i-1}$ 所在结点) ;
- ② 构造一个新结点, 存入元素的值 $e$ ;
- ③ 修改结点中的指针域, 使 $e$ 的直接后继为 $a_i$ ;
- ④ 使 $a_{i-1}$ 的直接后继为 $e$ 。





## 2.3.1 单链表的表示和实现

### ■ 插入的类C语言实现



Status ListInsert\_L(LinkList &L, int i, ElemType e)

{ //在带头结点的单链表L中第i个元素之前插入元素e

**p = L; k = 0; //初始化, p指向头结点, k为计数器**

**while (p && k < i-1) { //逐步移动指针p, 直到p指向第i-1个元素或p为空**

**p = p->next; ++ k;**

**}** // ①

**if (!p || k > i-1) return ERROR; //无法插入**

**if (!(s = (LinkList) malloc(sizeof(LNode)))) //申请元素e的结点空间**

**return OVERFLOW; //无法申请到结点空间**

**s->data = e;** // ②

**s->next = p->next;** // 插入元素e的结点 // ③

**p->next = s;** // ④

**return OK;**

**} //ListInsert\_L**

复杂度: ①查找插入位置:  $T_1(n)=O(n)$

②构造和插入新结点:  $T_2(n)=O(1)$

$\therefore T(n)=O(n)$



## 2.3.1 单链表的表示和实现

- 单链表的基本运算
  - 查找：找第 $i$ 个元素
  - 插入：在第 $i$ 个元素之前加入新元素
  - 删除：去掉第 $i$ 个元素
  - ...



## 2.3.1 单链表的表示和实现

- 单链表的基本运算
  - 查找：找第 $i$ 个元素
  - 插入：在第 $i$ 个元素之前加入新元素
  - 删除：去掉第 $i$ 个元素
  - ...



## 2.3.1 单链表的表示和实现

### ■ 单链表上的删除运算(将第 $i$ 个元素的结点删除)

#### ✱ 逻辑运算

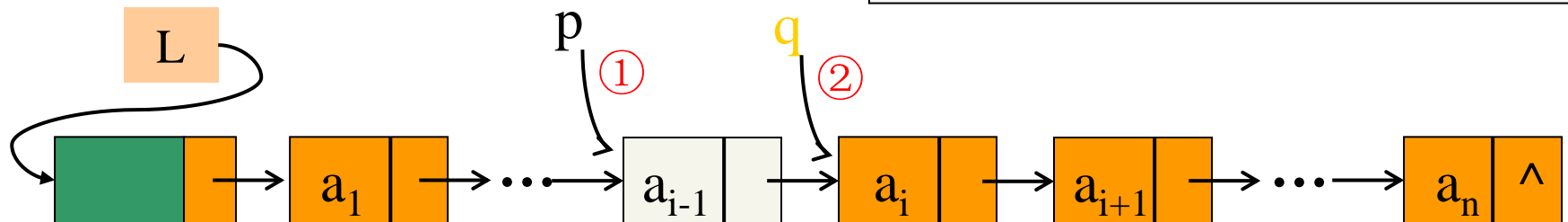
$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots)$



删除结点 $a_i$

$(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

#### ✱ 实现过程示意图



基本步骤:

- ① 找到第 $i-1$ 个元素所在结点;
- ② 增设临时指针 $q$ 指向 $p$ 的下一结点(待删除的结点);

处理思路:

第一步: 找到第 $i$ 个元素( $a_i$ )的直接前驱结点(即 $a_{i-1}$ 所在结点);  
第二步: 修改结点中的指针域, 使 $a_{i-1}$ 的直接后继为 $a_{i+1}$ 。





## 2.3.1 单链表的表示和实现

### ■ 单链表上的删除运算(把第*i*个位置的结点删除)

#### ✱ 逻辑运算

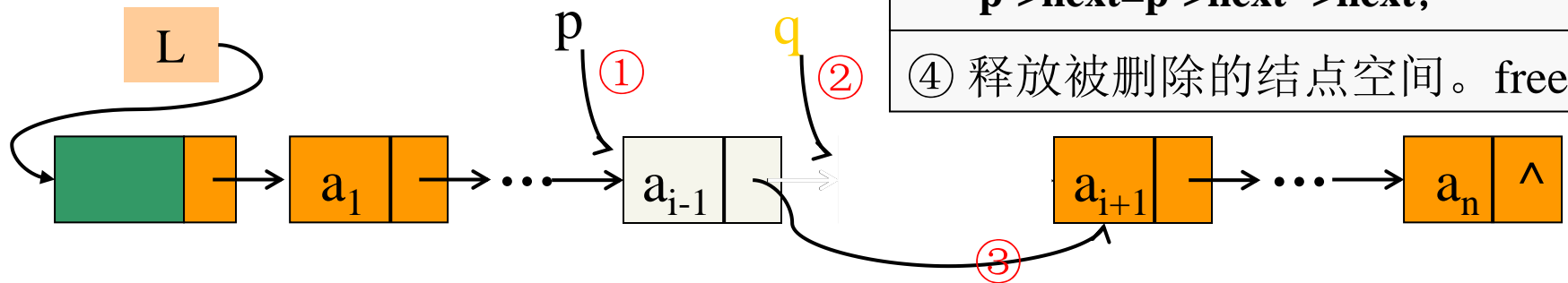
$(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots)$



删除结点 $a_i$

$(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

#### ✱ 单链表上的实现示意图



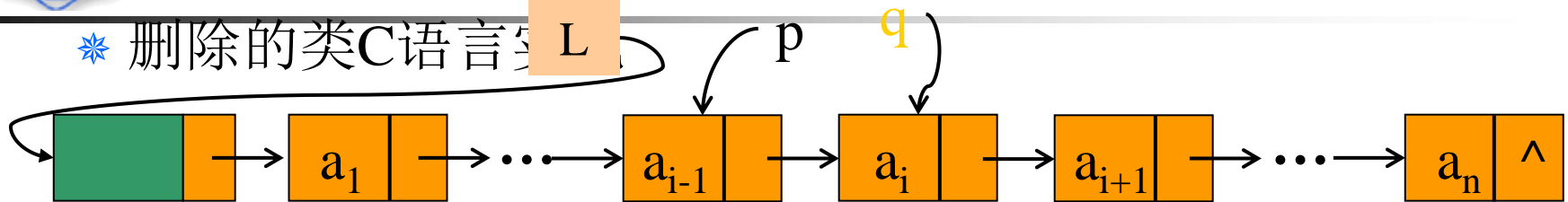
处理步骤:

- ① 查找第*i*个元素 ( $a_i$ ) 的直接前驱结点 (即 $a_{i-1}$ 所在结点);
- ② 增设临时指针 $q$ , 使其指向待删除的结点;
- ③ 修改结点中的指针域, 使 $a_{i-1}$ 的直接后继为 $a_{i+1}$ , 即  
 $p \rightarrow next = p \rightarrow next \rightarrow next;$
- ④ 释放被删除的结点空间。  $free(q)$



## 2.3.1 单链表的表示和实现

\* 删除的类C语言实现



```
Status ListDelete_L(LinkList &L, int i, ElemType &e) {
```

```
//在带头结点的单链表L中，删除第i个元素，并由e带回其值
```

```
    p = L;  k = 0;      //初始化，p指向头结点，k为计数器
```

```
    while (p -> next && k < i-1) { //查找被删除结点的前驱，即第i-1个元素结点
```

```
        p = p -> next; ++k;
```

```
    }
```

```
    if (!(p -> next) || k > i-1) return ERROR; //不存在第i
```

```
    q = p -> next;      // 令q指向待删除的元素结点
```

```
    p -> next = q -> next; // 从链表中去除结点
```

```
    e = q -> data;      // 参数e带回被删除结点的数据
```

```
    free(q);
```

```
    return OK;
```

```
} //ListDelete_L
```

思考：

算法的时间复杂度 $T(n)$ 和空间复杂度 $S(n)$ ？



# 单链表小结

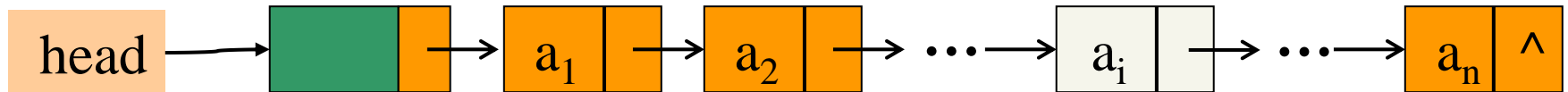
## ■ 链表的优缺点

### ■ 优点:

- 插入、删除时无须移动元素，只需修改指针
- 根据需要申请存储空间，且不要求连续的存储空间

### ■ 缺点:

- 对表中的元素只能进行顺序访问，查找第*i*个元素的时间复杂度为 $O(n)$
- 用指针指示元素之间的逻辑关系（直接前驱、后继），存储空间利用率低

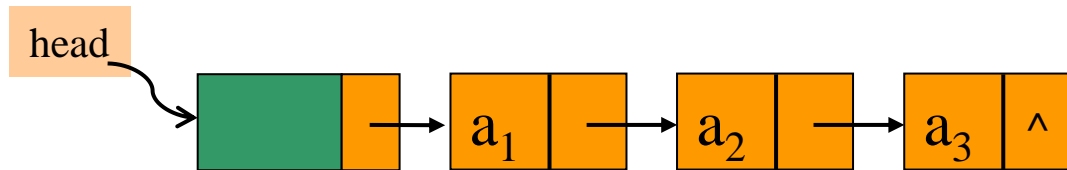


带头结点的单链表示意图

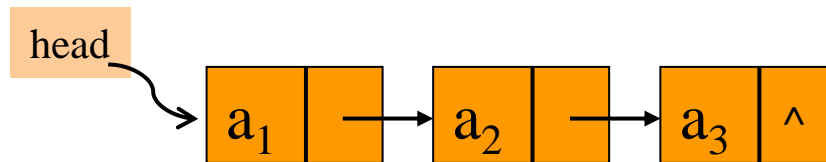


# 单链表小结

- 含头结点的单链表的优点
  - 对空表和非空表的处理是统一的
  - 对第一个元素结点的指针操作与表中其它元素结点的指针操作是一致的



(b) 含头结点的单链表

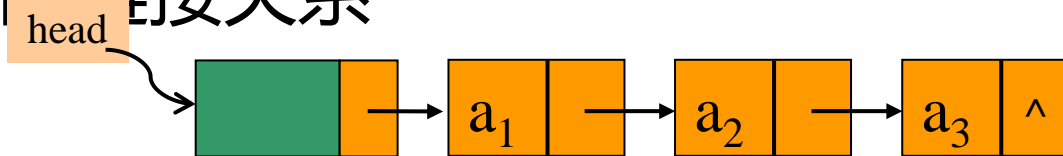


(a) 不含头结点的单链表

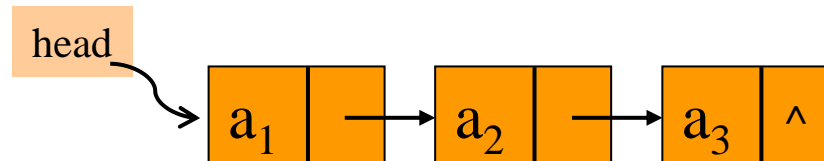


# 单链表（线性链表）要点回顾

- 结点中只设一个指针域（指向当前元素的后继）
- 用**头指针**标识一个单链表，或许含有**头结点**
- 在单链表中只能按顺序访问元素（**顺序存取**）
- 顺序查找是最基本操作，插入、删除操作仅需修改结点间的链接关系



(b) 含头结点的单链表



(a) 不含头结点的单链表



## 第二章 线性表

### 本章内容

**2.1 线性表的类型定义**

**2.2 线性表的顺序表示和实现**

**2.3 线性表的链式表示和实现**

**2.3.1 线性链表**

**2.3.2 循环链表**

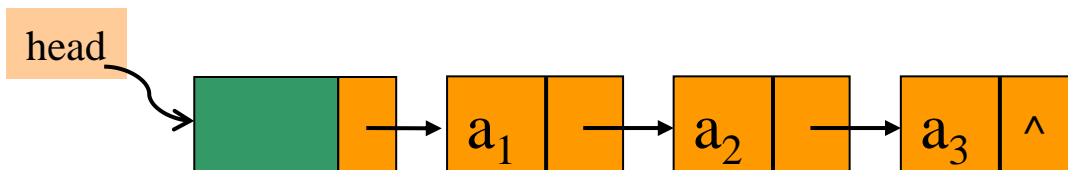
**2.3.3 双向链表**

**2.4 一元多项式的表示及相加**

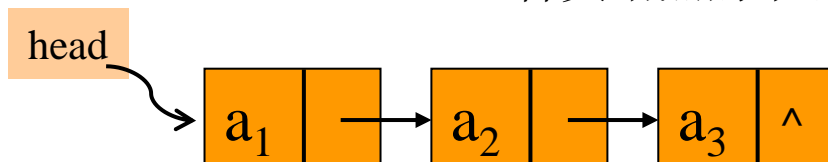


# 头结点对单链表运算的影响？

以创建单链表为例。



(a) 含头结点的单链表



(b) 不含头结点的单链表



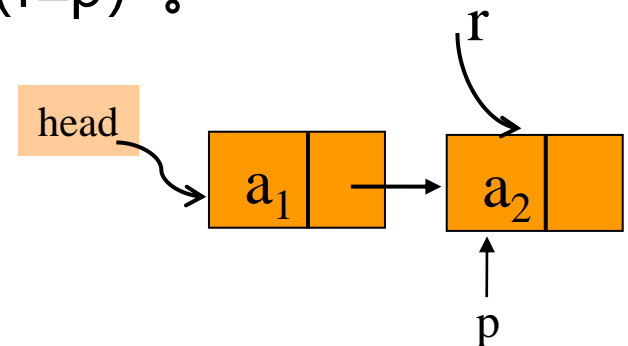
# 不含头结点的单链表

## 用尾指针插入法 (使用尾结点的指针) 建立链表, 没

### 点与插入后续其他结点

```
LinkList head, r, p;  
head = r = NULL; //空表  
p = (LNode *)malloc(sizeof(LNode));  
scanf(p->data);  
p->next = NULL;  
head = r = p;  
  
while () {  
    p = (LNode *)malloc(sizeof(LNode));  
    scanf(p->data);  
    r->next = p;  
    r = p;  
};  
r->next = NULL; //设置表尾标志
```

head的指向; 由于表尾指  
处理, 而是直接设置尾结点  
; r = p; )  
接处理 (即r->next = p; ) ,  
(r=p) 。







# 含头结点的单链表

- **用表尾插入法**（使用尾结点的指针）建立链表（有头结点）  
**插入第一个结点与插入后续其他结点的区别**

```
LinkedList head, r, p;
```

```
//空表（仅有头结点）
```

```
head = (LNode *)malloc(sizeof(LNode));
```

```
head->next = NULL;
```

```
r = head;
```

```
while () {
```

```
    p = (LNode *)malloc(sizeof(LNode));
```

```
    scanf(p->data);
```

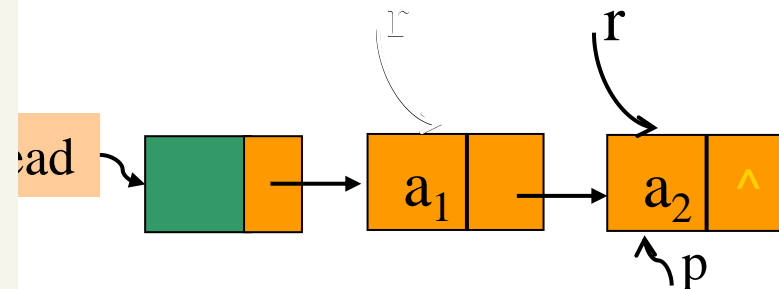
```
    r->next = p;
```

```
    r = p;
```

```
};
```

```
r->next = NULL; //设置表尾标志
```

之后，然后再设置尾结点  
 $r = p;$





# 表尾插入法建立单链表

```
LinkedList head, r, p;  
head = r = NULL; //空表（无头结点）  
p = malloc(sizeof(LNode));  
scanf(p->data);  
p->next = NULL;  
head = r = p;
```

```
while () {  
    p = malloc(sizeof(LNode));  
    scanf(p->data);  
    r->next = p;  
    r = p;  
};  
r->next = NULL;
```

```
LinkedList head, r, p;  
//空表（仅有头结点）  
head = malloc(sizeof(LNode));  
head->next = NULL;  
r = head;
```

```
while () {  
    p = malloc(sizeof(LNode));  
    scanf(p->data);  
    r->next = p;  
    r = p;  
};  
r->next = NULL;
```

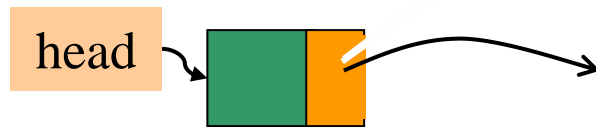
☞ 用表尾插入法（使用尾结点的指针）建立单链表（有头结点），插入第一个结点与插入后续其他结点的处理无区别：



# 含头结点的单链表

- 用表头插入法建立链表（有头结点），插入第一个结点与插入后续其他结点的处理无区别：
- 插入结点时，新结点链接在头结点之后：

$p \rightarrow next = head \rightarrow next;$        $head \rightarrow next = p;$



```
LinkedList head, p;
```

```
//空表（仅有头结点）
```

```
head = (LNode *)malloc(sizeof(LNode));
```

```
head->next = NULL;
```

```
while () {
```

```
    p = (LNode *)malloc(sizeof(LNode));
```

```
    scanf(p->data);
```

```
    p->next = head->next;
```

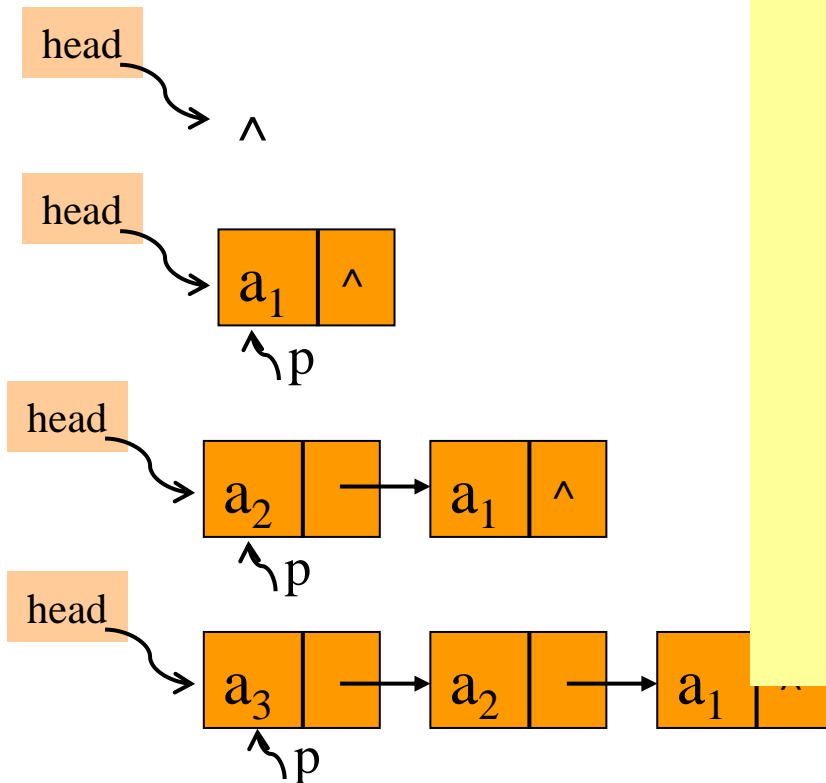
```
    head->next = p;
```

```
};
```



# 不含头结点的单链表

- 单链表不含头结点时，用**表头插入法**（使用头指针）建立链表，**插入第一个结点与插入后续其他结点的处理没有区别**：初始时head为空，每插入一个结点，都是设置新结点的指针域指向head所指的目标结点（即 $p \rightarrow \text{next} = \text{head}$ ；），然后将head更新为指向新生成的结点（即 $\text{head} = p$ ；）。



```
LinkedList head, p;  
head = NULL; //空表
```

```
while () {  
    p = (LNode *)malloc(sizeof(LNode));  
    scanf(p->data);  
    p->next = head;  
    head = p;  
};
```



# 表头插入法建立单链表

- 用表头插入法建立链表（有头结点），插入第一个结点与插入后续其他结点的处理没有区别

```
LinkedList head, p;  
head = NULL; //空表（无头结点）
```

```
while () {  
    p = malloc(sizeof(LNode));  
    scanf(p->data);  
    p->next = head;  
    head = p;  
};
```

```
LinkedList head, p;  
//空表（仅有头结点）  
head = malloc(sizeof(LNode));  
head->next = NULL;
```

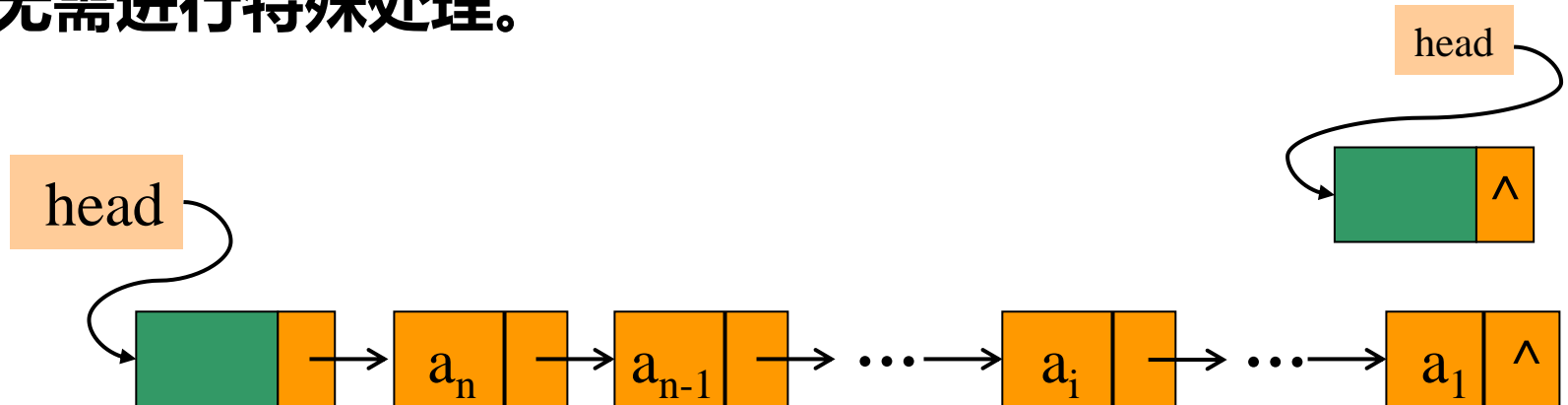
```
while () {  
    p = malloc(sizeof(LNode));  
    scanf(p->data);  
    p->next = head->next;  
    head->next = p;  
};
```

- ☞ 单链表不含头结点时，用表头插入法（使用头指针）建立链表，插入第一个结点与插入后续其他结点的处理没有区别



# 头结点的作用

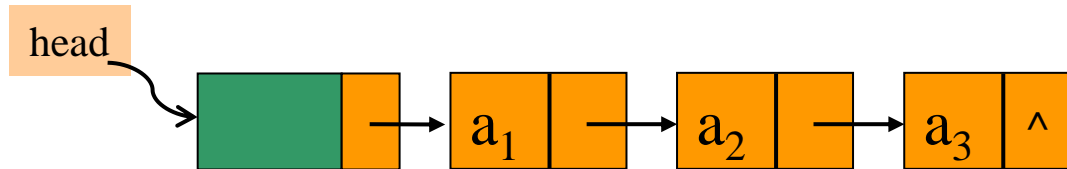
- 在链表的首元结点之前附加一个**头结点**，会带来以下两个优点：
  - a. 含有头结点时，无论链表是否为空，其头指针（链表的标识）都指向其头结点，运算过程中不需要修改头指针，头指针始终不变，这样对空表和非空表的处理是统一的。
  - b. 由于第一个元素结点的指针被存放在头结点的指针域中，这与其他结点的指针存储方式是相同的，所以对第一个元素结点的操作与表中其它元素结点的操作是一致的，无需进行特殊处理。



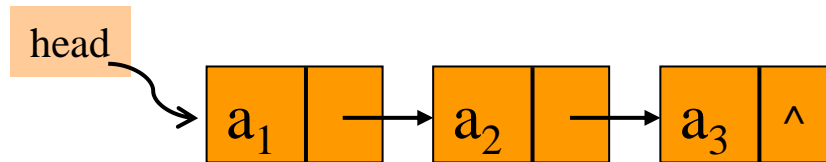


# 单链表的头结点

- 含头结点的单链表的优点
  - 对空表和非空表的处理是统一的
  - 对第一个元素结点的操作与表中其它元素结点的操作是一致的(不存在特殊位置)
  - 运算过程中不修改头结点，因此头指针不变



(b) 含头结点的单链表



(a) 不含头结点的单链表



# 单链表的运算示例：

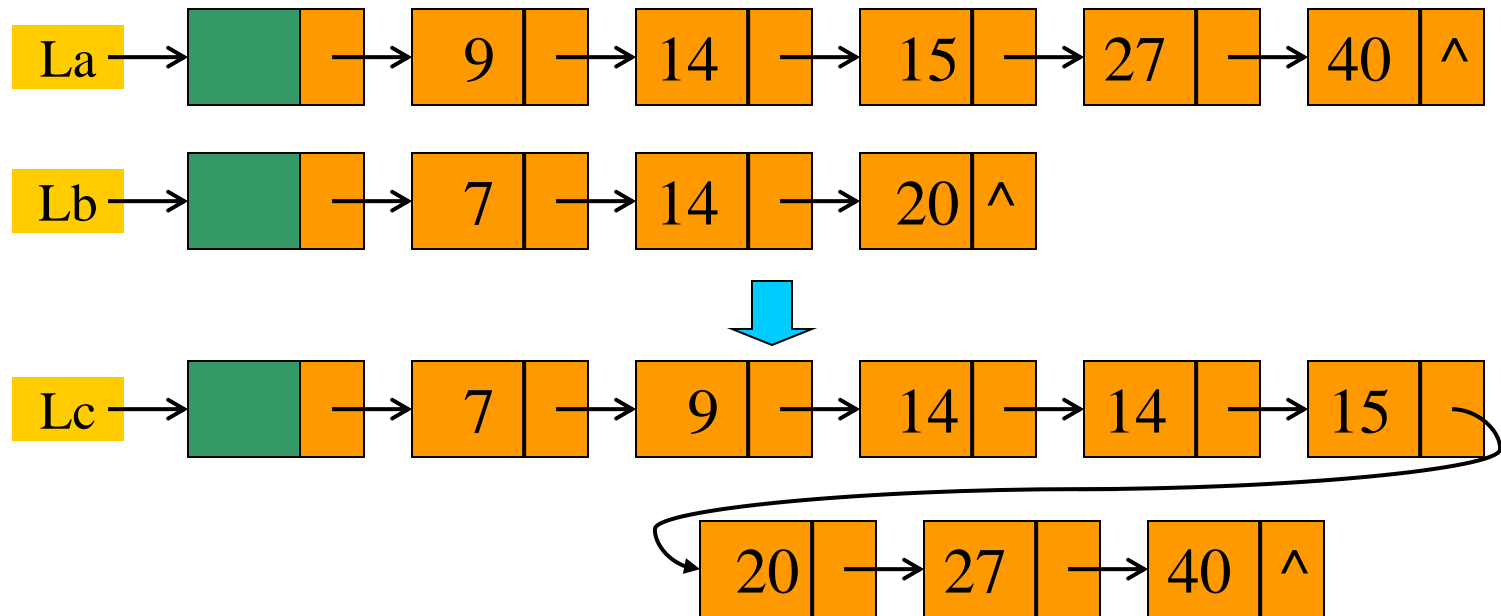
两个单链表合并。





## 2.3.1 单链表的表示和实现

### ■ 两个有序单链表的合并( $L_c = L_a \cup L_b$ )

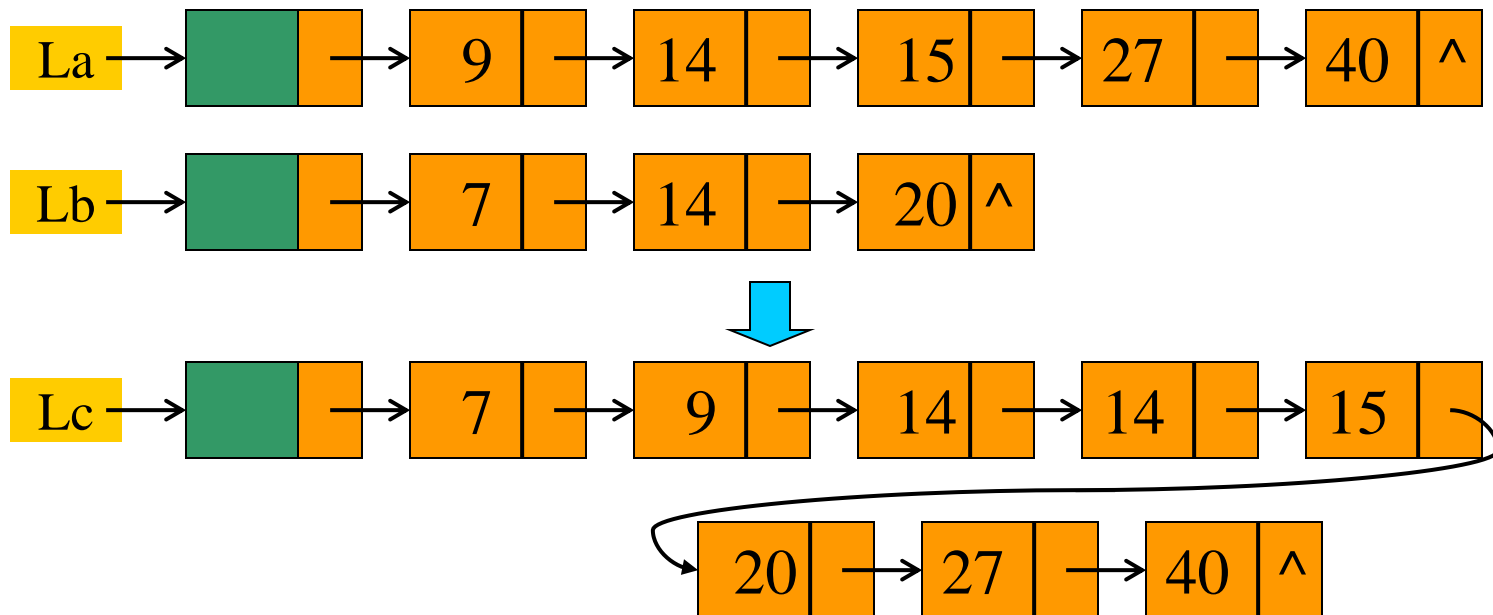


- ☞  $L_c$ 链表中的结点可重新申请（不销毁 $L_a$ 、 $L_b$ 链表），或取自 $L_a$ 、 $L_b$ 链表（销毁 $L_a$ 、 $L_b$ 链表），差别在于 $L_a$ 和 $L_b$ 链表是否进行删除结点的操作。



## 2.3.1 单链表的表示和实现

### ■ 两个有序单链表的合并( $L_c = L_a \cup L_b$ )



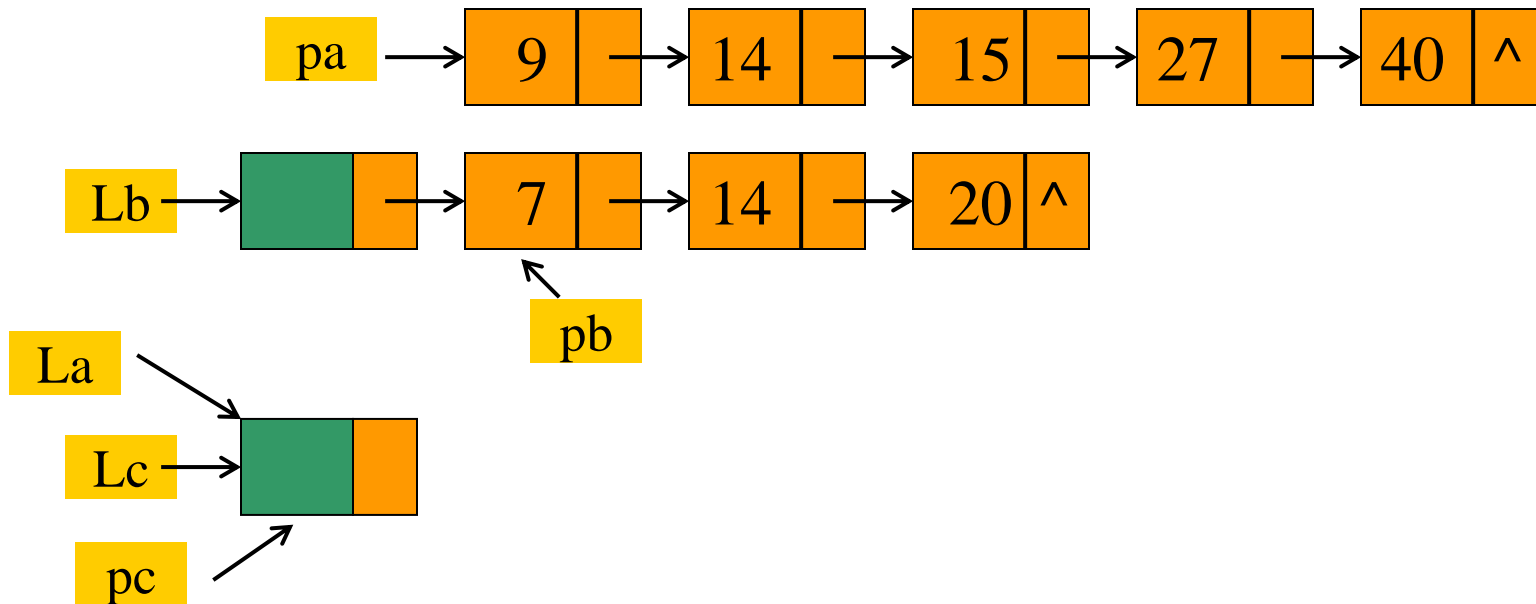
处理思路（销毁 $L_a$ 和 $L_b$ 链表）：

- (1) 设置三个指针  $p_a$ 、 $p_b$ 、 $p_c$  分别指向  $L_a$ 、 $L_b$ 、 $L_c$  的当前结点；
- (2) 比较  $p_a$ 、 $p_b$  所指结点的元素值大小，将元素值较小的结点链接到  $p_c$  所指结点之后，更新  $p_a$ （或  $p_b$ ）和  $p_c$ ；
- (3) 重复(2)，直到  $p_a$ 、 $p_b$  中某一指针为空，将另一非空指针链追加到  $p_c$  之后，算法结束。



## 2.3.1 单链表的表示和实现

### ■ 两个有序单链表的合并( $L_c = L_a \cup L_b$ )



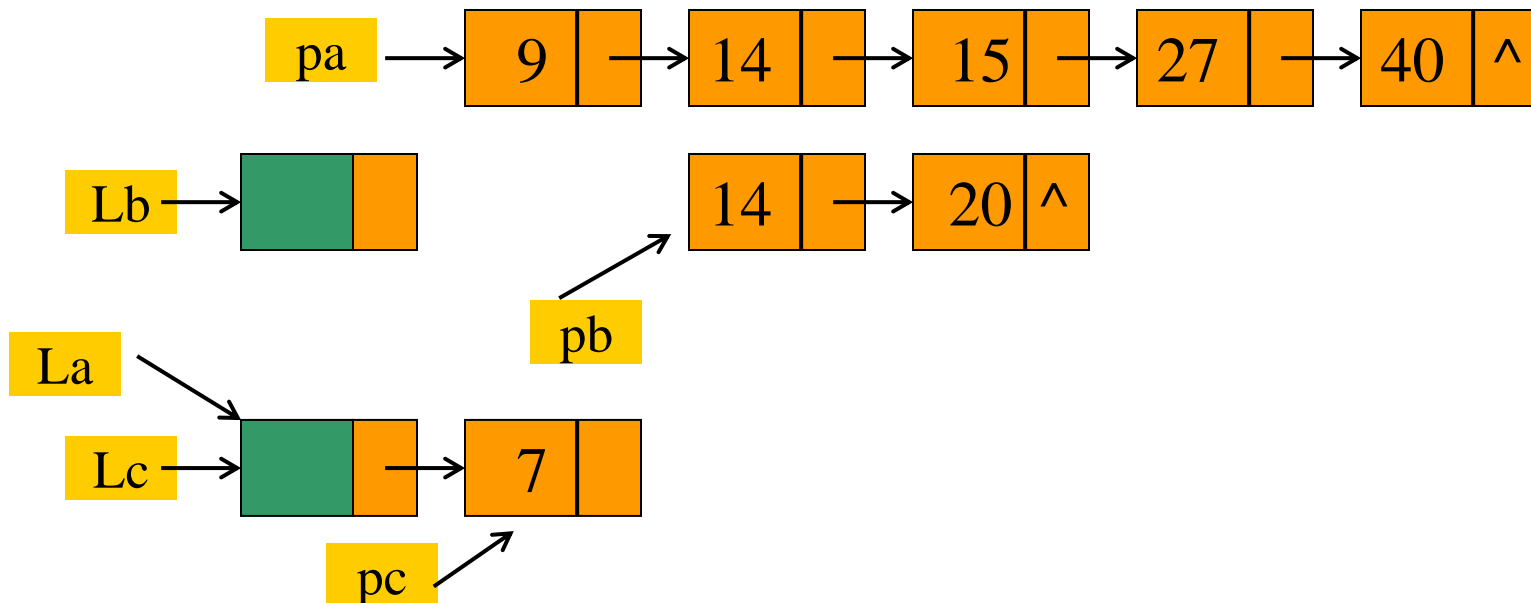
处理思路（销毁 $L_a$ 和 $L_b$ 链表）：

- (1) 设置三个指针  $p_a$ 、 $p_b$ 、 $p_c$  分别指向  $L_a$ 、 $L_b$ 、 $L_c$  的当前结点；
- (2) 比较  $p_a$ 、 $p_b$  所指结点的元素值大小，将元素值较小的结点链接到  $p_c$  所指结点之后，更新  $p_a$ （或  $p_b$ ）和  $p_c$ ；
- (3) 重复(2)，直到  $p_a$ 、 $p_b$  中某一指针为空，将另一非空指针链追加到  $p_c$  之后，算法结束。



## 2.3.1 单链表的表示和实现

### ■ 两个有序单链表的合并( $L_c = L_a \cup L_b$ )



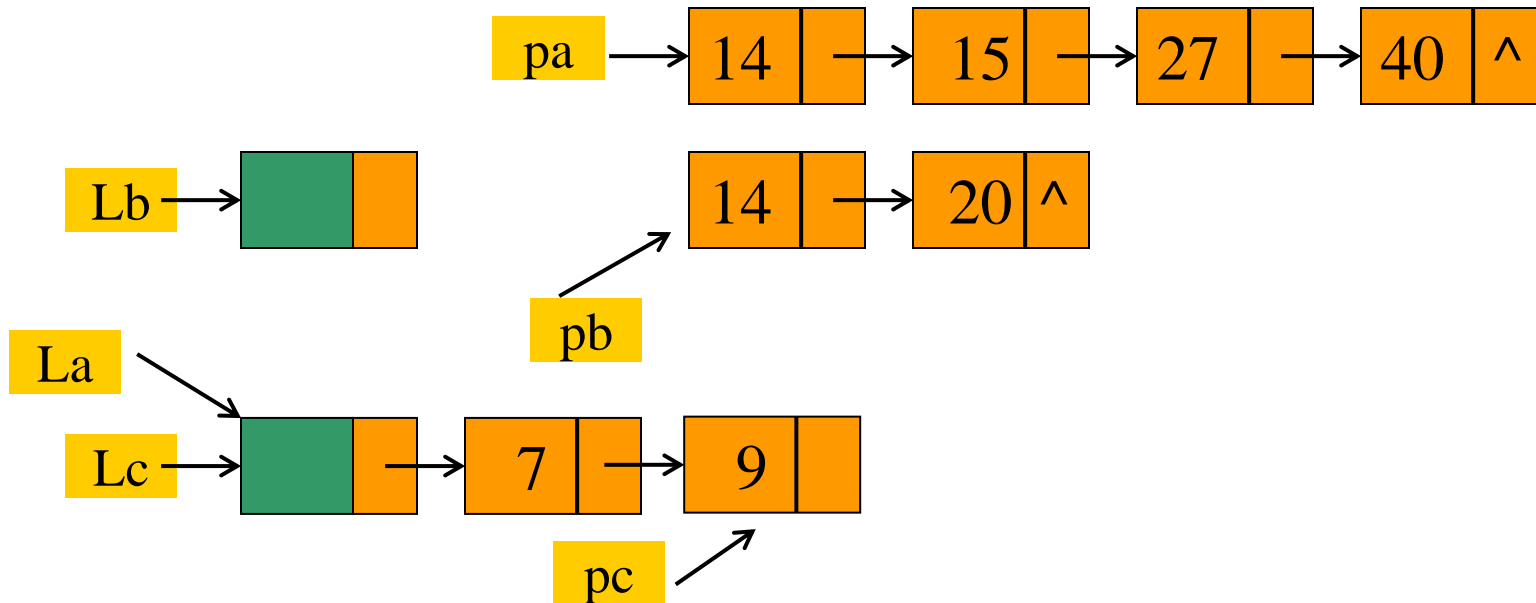
处理思路（销毁 $L_a$ 和 $L_b$ 链表）：

- (1) 设置三个指针  $p_a$ 、 $p_b$ 、 $p_c$  分别指向  $L_a$ 、 $L_b$ 、 $L_c$  的当前结点；
- (2) 比较  $p_a$ 、 $p_b$  所指结点的元素值大小，将元素值较小的结点链接到  $p_c$  所指结点之后，更新  $p_a$ （或  $p_b$ ）和  $p_c$ ；
- (3) 重复(2)，直到  $p_a$ 、 $p_b$  中某一指针为空，将另一非空指针链追加到  $p_c$  之后，算法结束。



## 2.3.1 单链表的表示和实现

### ■ 两个有序单链表的合并( $L_c = L_a \cup L_b$ )



处理思路（销毁 $L_a$ 和 $L_b$ 链表）：

- (1) 设置三个指针  $p_a$ 、 $p_b$ 、 $p_c$  分别指向  $L_a$ 、 $L_b$ 、 $L_c$  的当前结点；
- (2) 比较  $p_a$ 、 $p_b$  所指结点的元素值大小，将元素值较小的结点链接到  $p_c$  所指结点之后，更新  $p_a$ （或  $p_b$ ）和  $p_c$ ；
- (3) 重复(2)，直到  $p_a$ 、 $p_b$  中某一指针为空，将另一非空指针链追加到  $p_c$  之后，算法结束。



## 2.3.1 单链表的表示和实现

- 用类C语言实现归并算法:

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc){  
//归并两个非递减单链表La、Lb到Lc中，且使Lc非递减  
pa = La->next; pb = Lb->next;  
Lc = pc = La;  
while (pa && pb) {  
    if (pa->data <= pb->data) {  
        pc->next = pa; pa = pa->next;  
    }  
    else{  
        pc->next = pb; pb = pb->next;  
    }  
    pc = pc->next;  
} //while  
pc->next = pa? pa : pb;  
free(Lb);  
// MergeList_L
```

时间复杂度?

$$T(n)=O(n_a)+O(n_b)=O(n_{La}+n_{Lb})$$



## 第二章 线性表

### 本章内容

**2.1** 线性表的类型定义

**2.2** 线性表的顺序表示和实现

**2.3** 线性表的链式表示和实现

**2.3.1** 线性链表

**2.3.2** 循环链表

**2.3.3** 双向链表

**2.4** 一元多项式的表示及相加



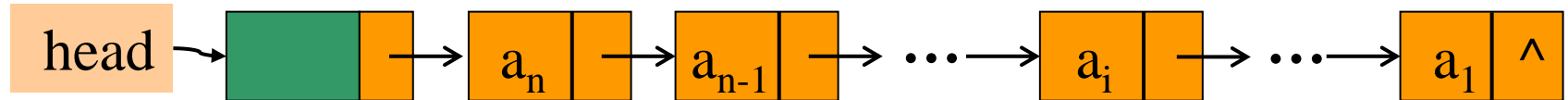
## 2.3.2 循环单链表



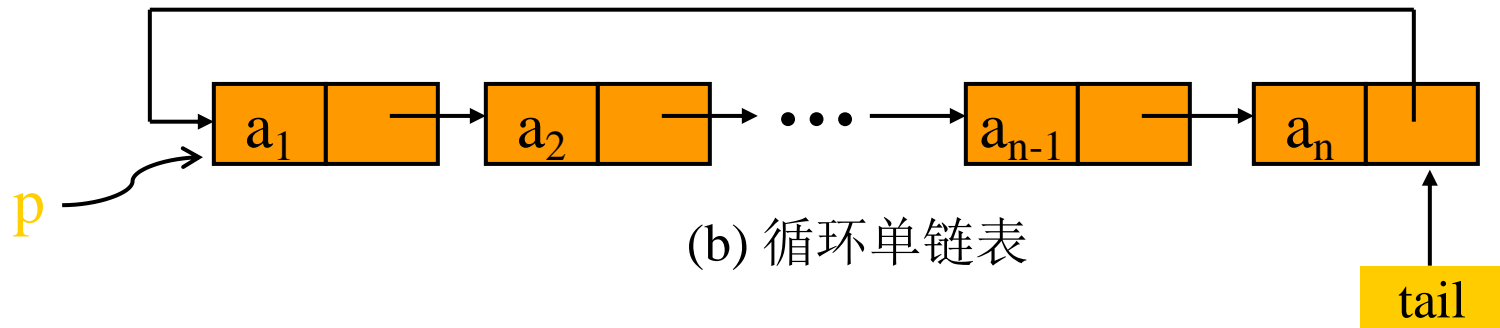


## 2.3.2 循环单链表和双向链表

### ■ 线性表: $(a_1, a_2, \dots, a_i, \dots, a_n)$



(a) 单链表



(b) 循环单链表

☞ 一般将标识循环单链表的指针设置在尾结点，优点如下：

(1) 可以快速得到表头指针 ( $\text{tail} \rightarrow \text{next}$ )；

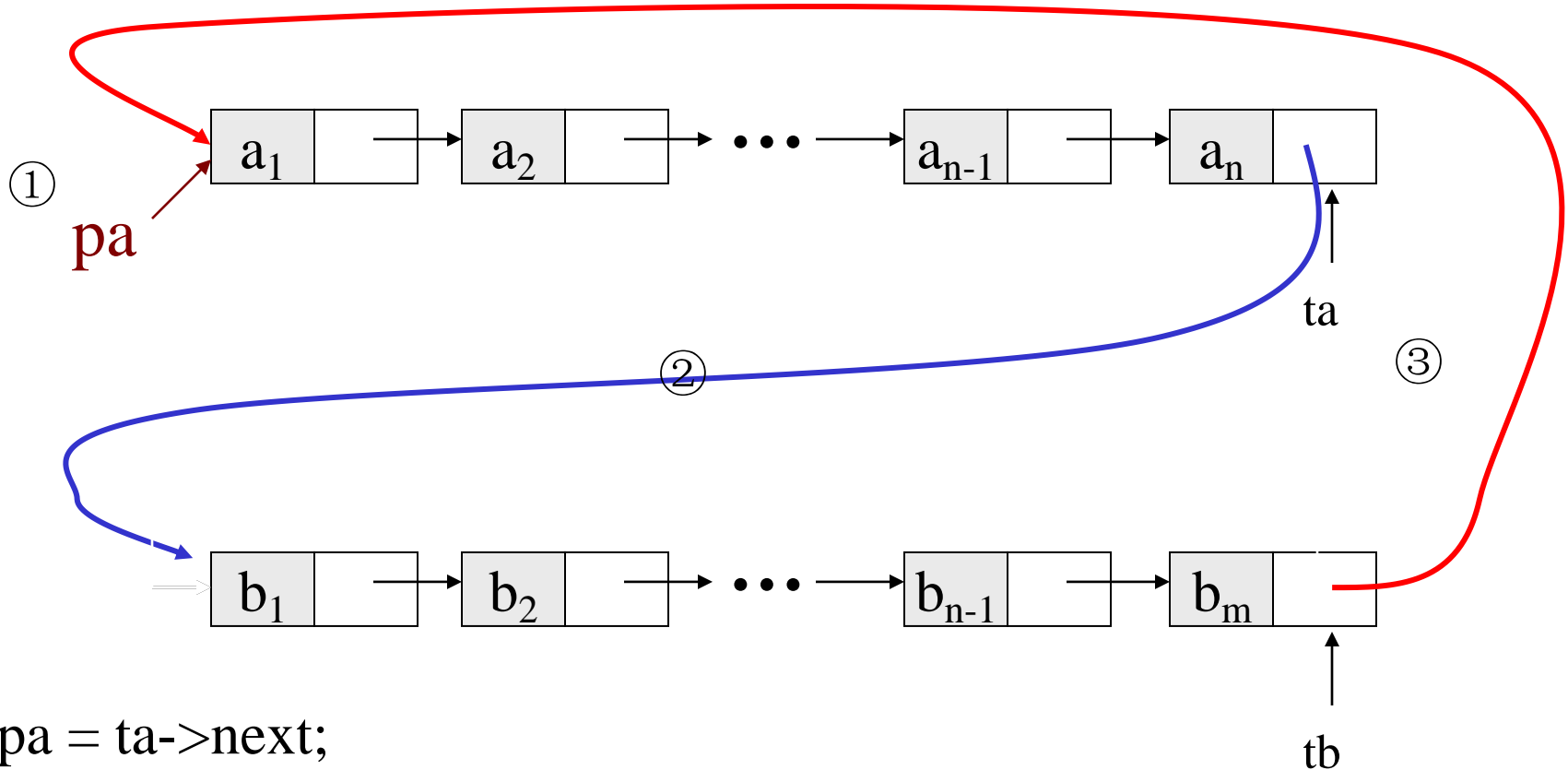
(2) 对于两个表的简单合并，可以通过简单操作完成 ( $T(n)=O(1)$ )。





## 2.3.2 循环单链表和双向链表

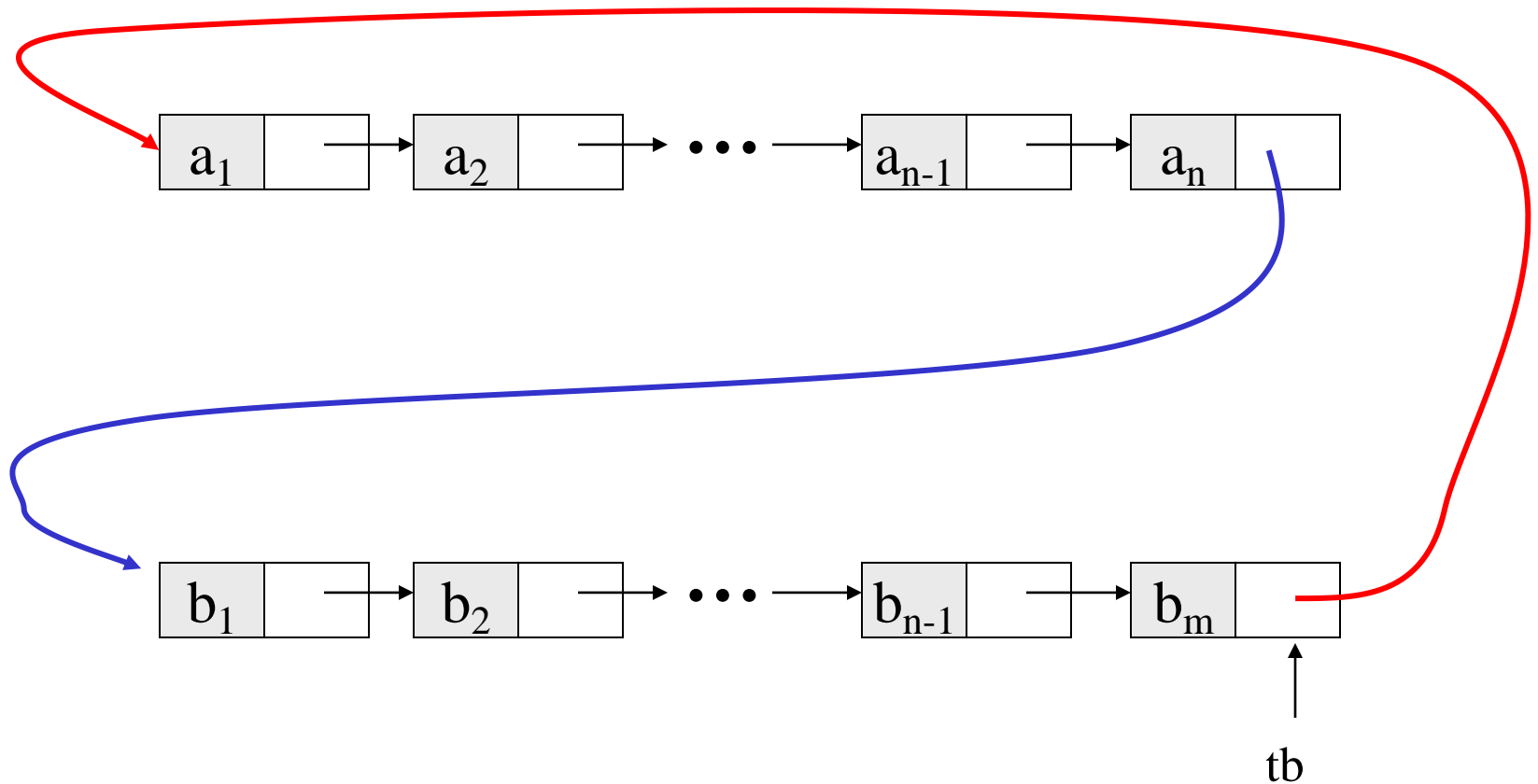
### ■ 循环单链表的合并





## 2.3.2 循环单链表和双向链表

### ■ 循环单链表的合并





# 循环单链表应用示例

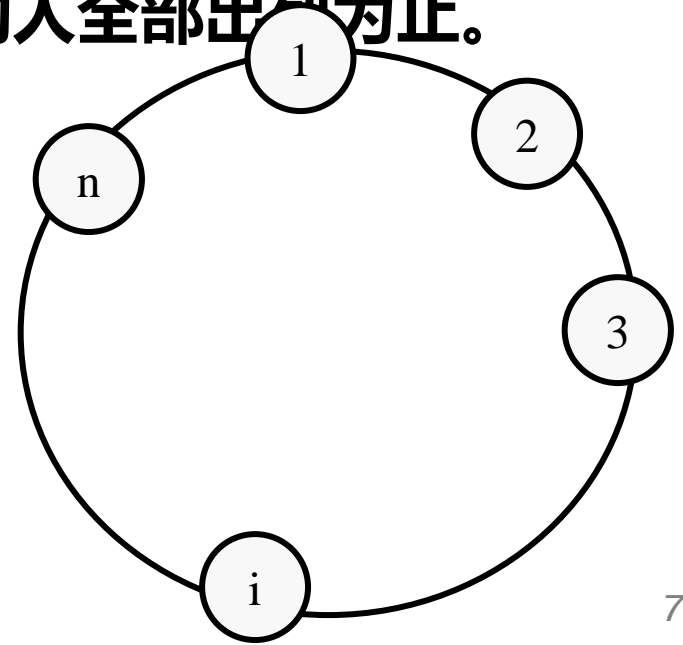


# 约瑟夫环(Joseph Circle)

## ■ 问题描述:

编号为 $1, 2, \dots, n$ 的 $n$ 个人按顺时针方向围坐一圈，每人持有一个密码（正整数）。现在给定一个随机数 $m > 0$ ，从编号为1的人开始，按顺时针方向，从1开始顺序报数，报到 $m$ 时暂停。报 $m$ 的人出圈，同时留下他的密码作为新的 $m$ 值，从他在顺时针方向上的下一个人开始，重新从1开始报数，如此下去，直至所有的人全部出列为止。

请编写程序求出圈的顺序。

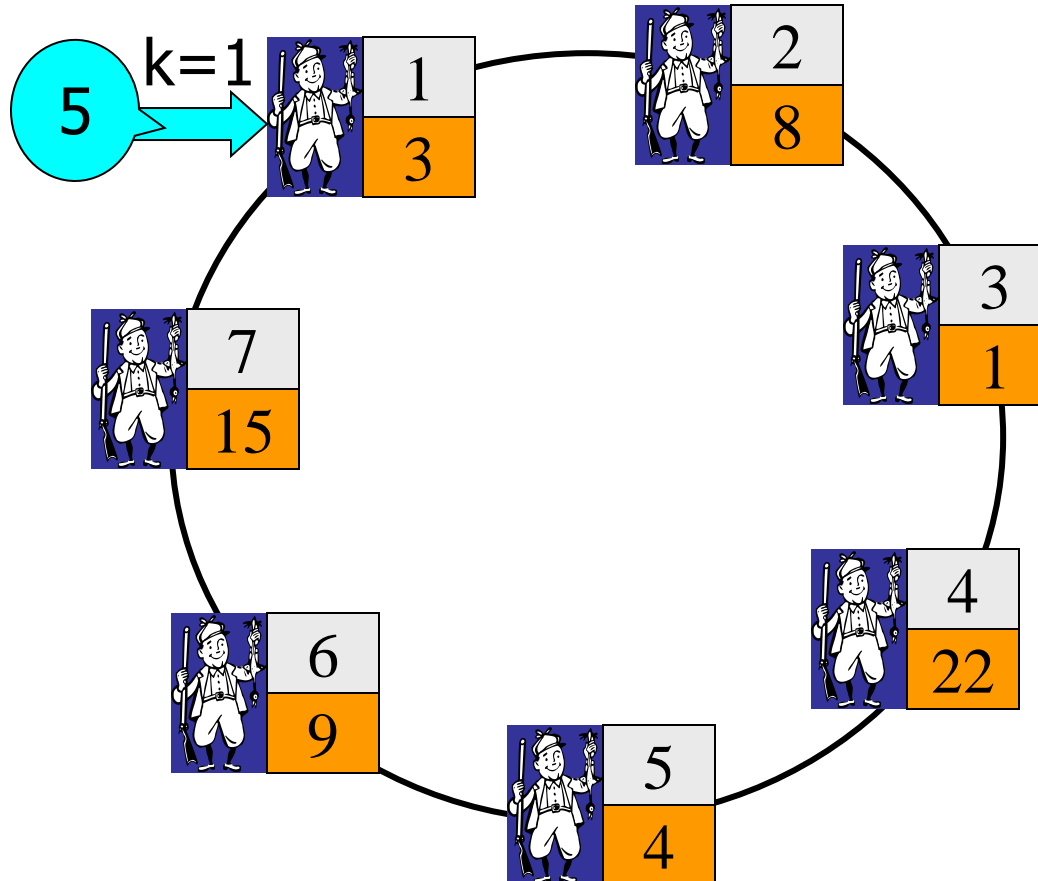




# 约瑟夫环(Joseph Circle)

■ 例:

start



k: 计数  
m: 密码

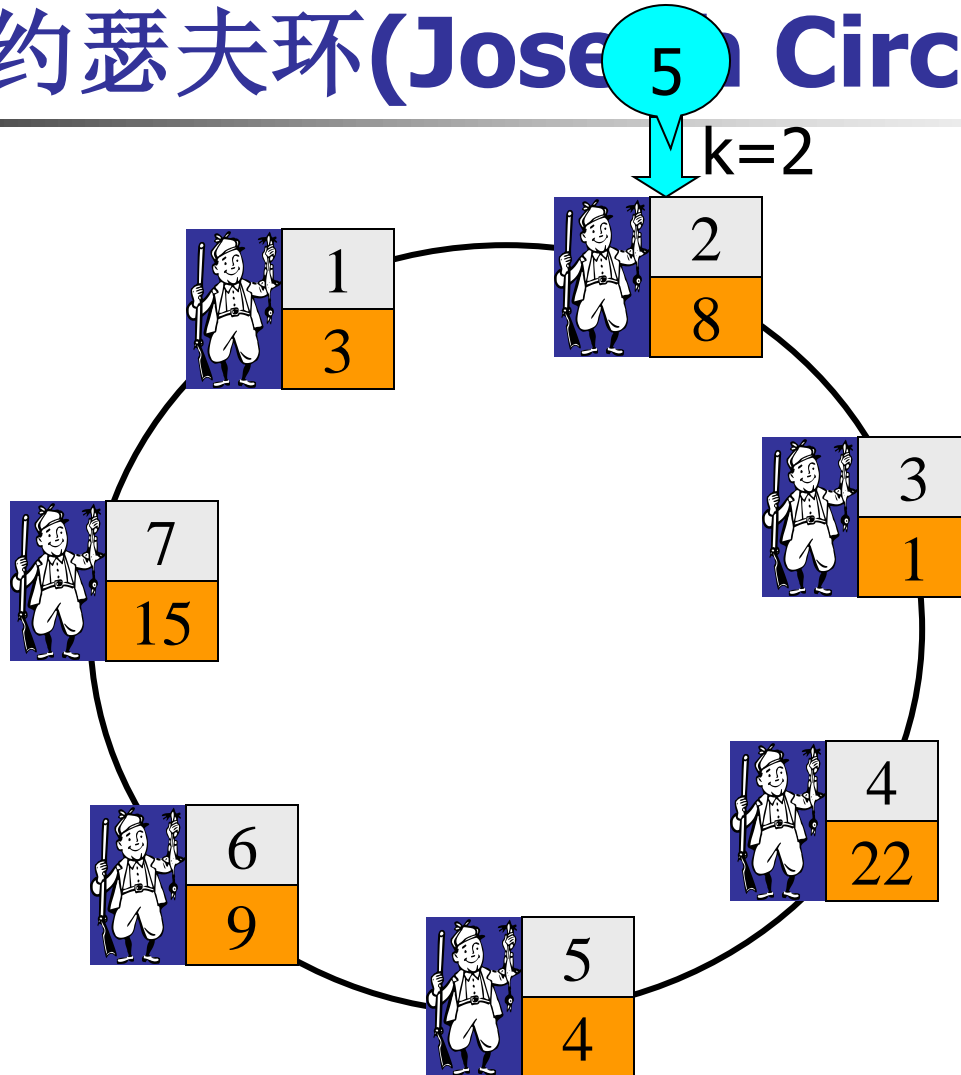
➡ 出队序列:



# 约瑟夫环(Joseph Circle)

■ 例:

start



k: 计数  
m: 密码

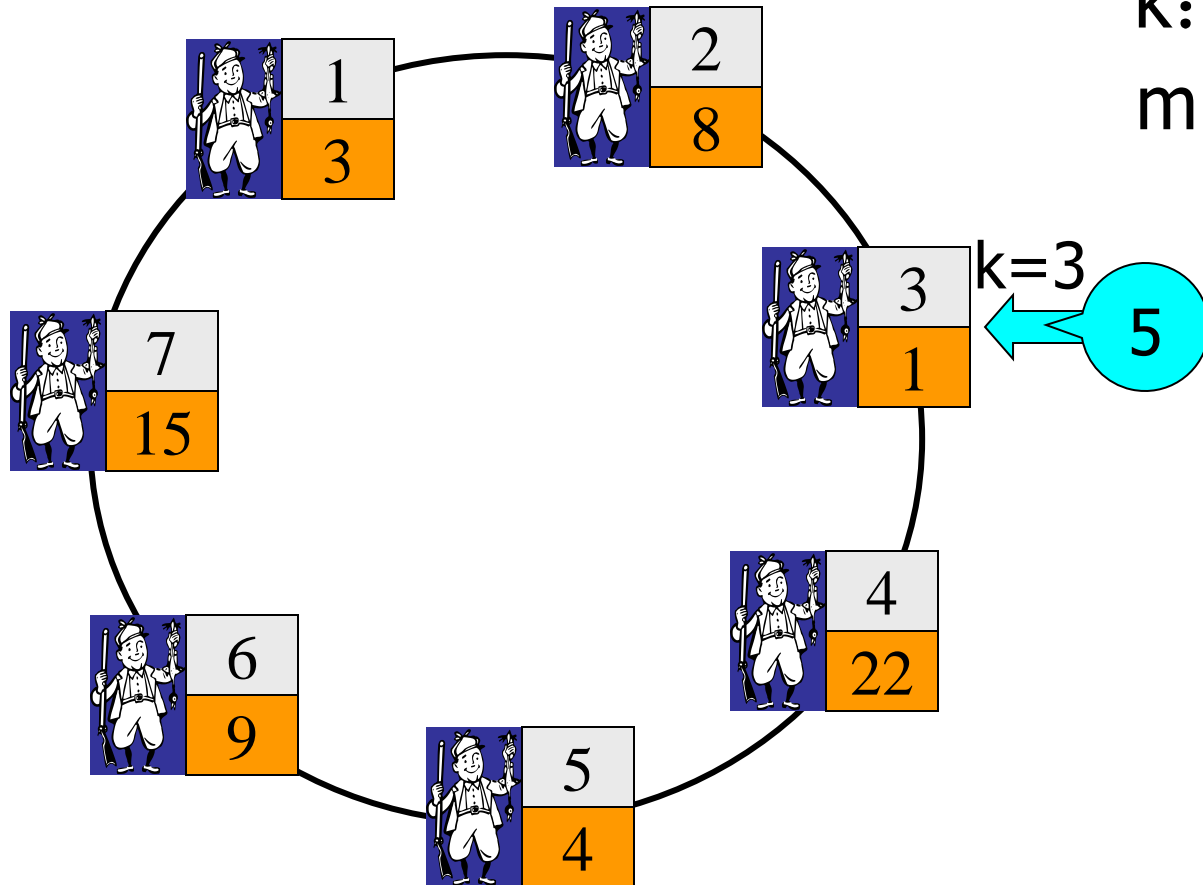
➡ 出队序列:





# 约瑟夫环(Joseph Circle)

■ 例:



➡ 出队序列:

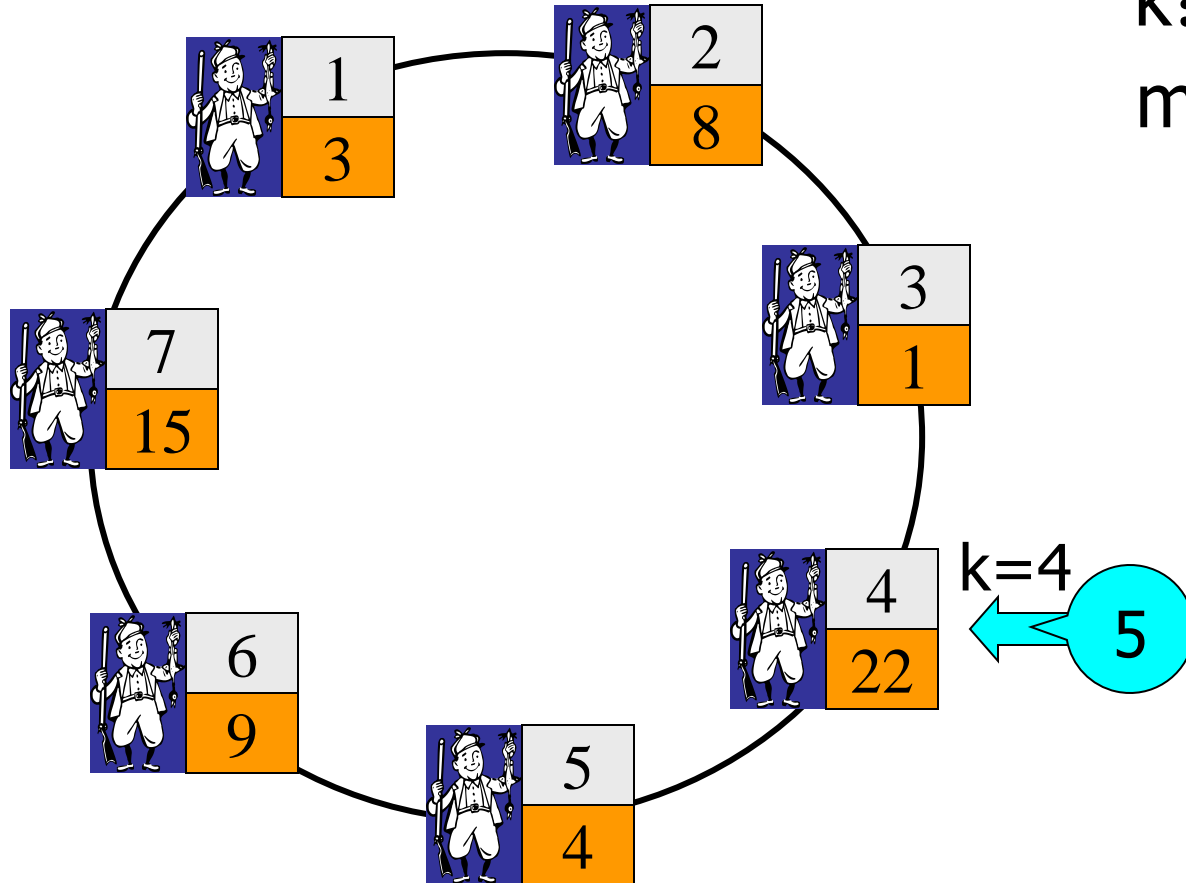


# 约瑟夫环(Joseph Circle)

■ 例:

start

k: 计数  
m: 密码



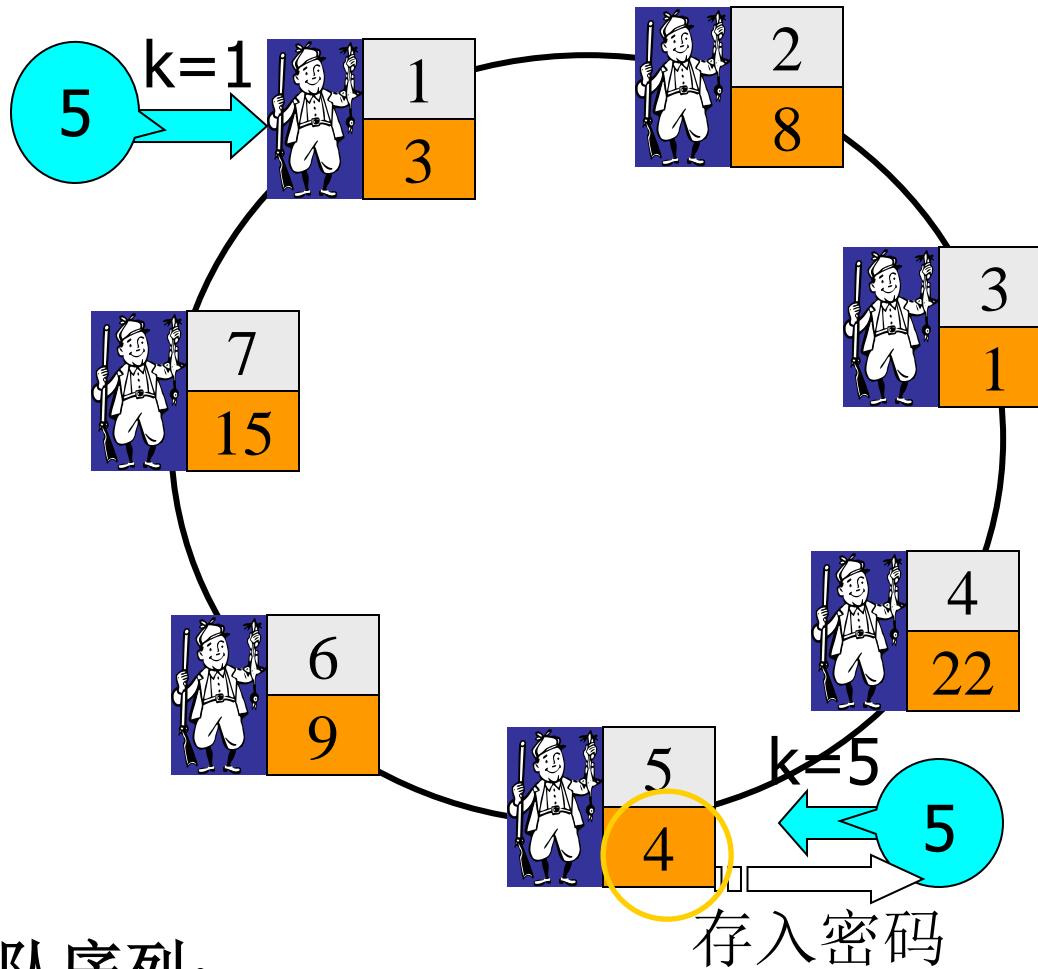
➡ 出队序列:



# 约瑟夫环(Joseph Circle)

■ 例:

k: 计数  
m: 密码



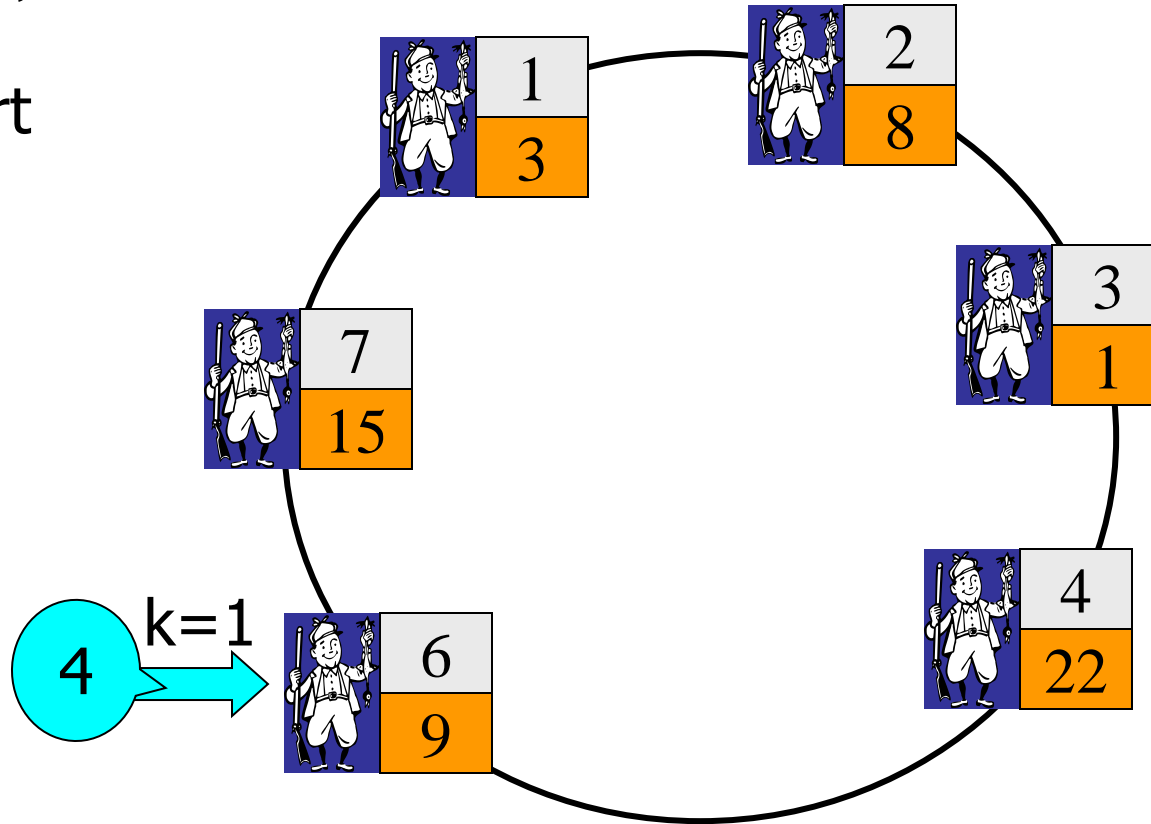
➡ 出队序列:




# 约瑟夫环(Joseph Circle)

■ 例:

start



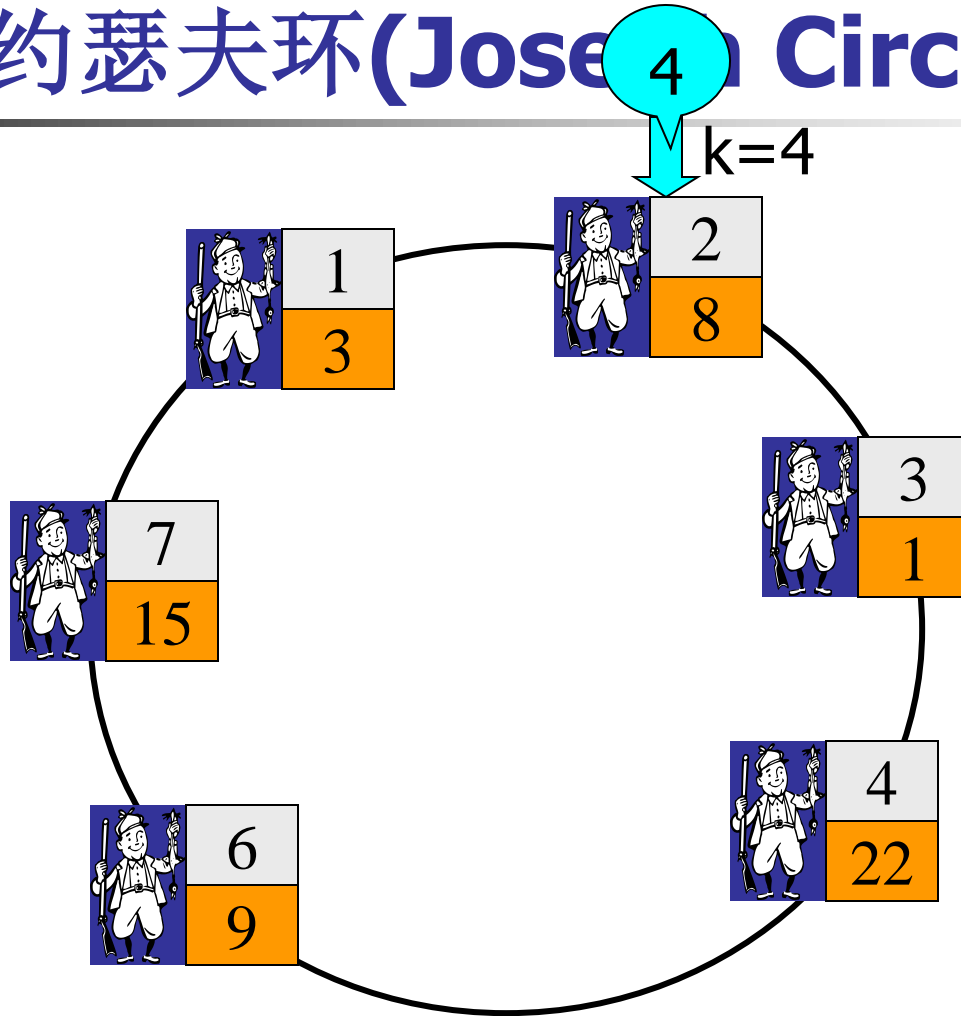
k: 计数  
m: 密码

➡ 出队序列: 




# 约瑟夫环(Joseph Circle)

■ 例:



k: 计数  
m: 密码

➡ 出队序列:  5

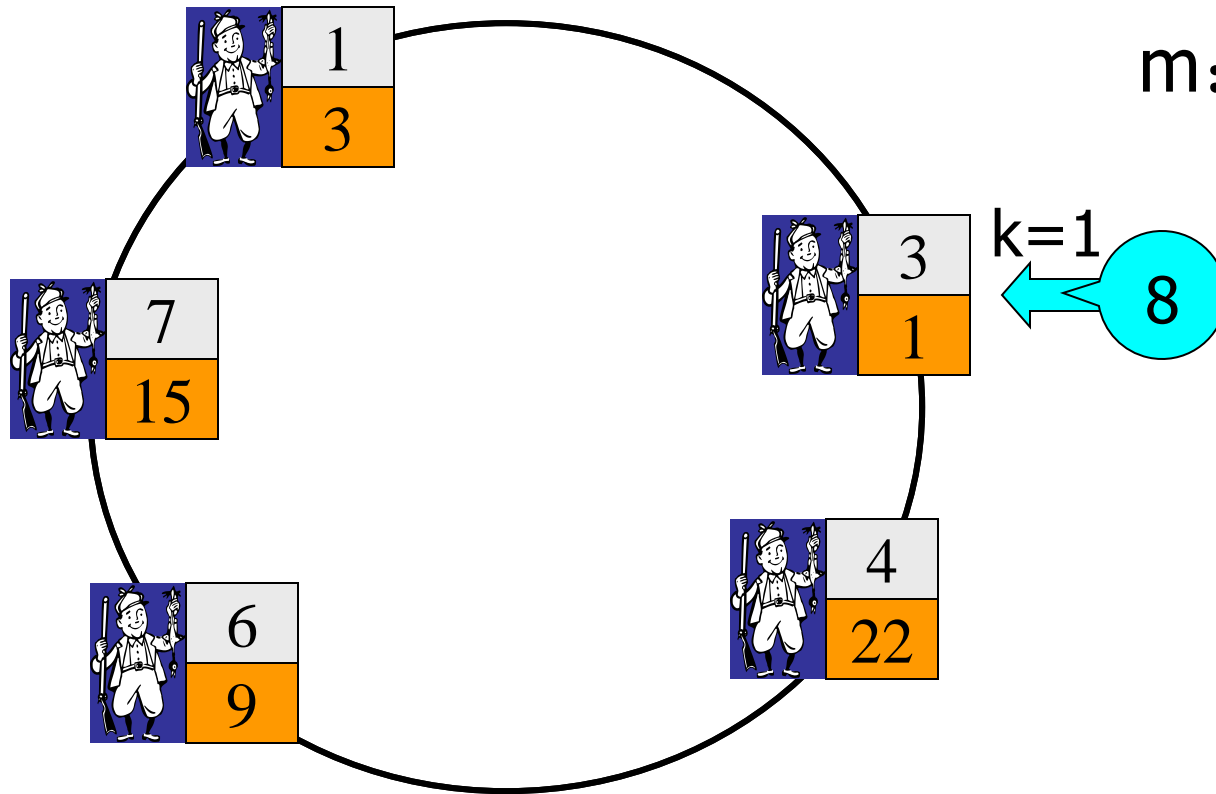


# 约瑟夫环(Joseph Circle)

■ 例:

start

k: 计数  
m: 密码



➡ 出队序列: 

Person 1	5
----------	---

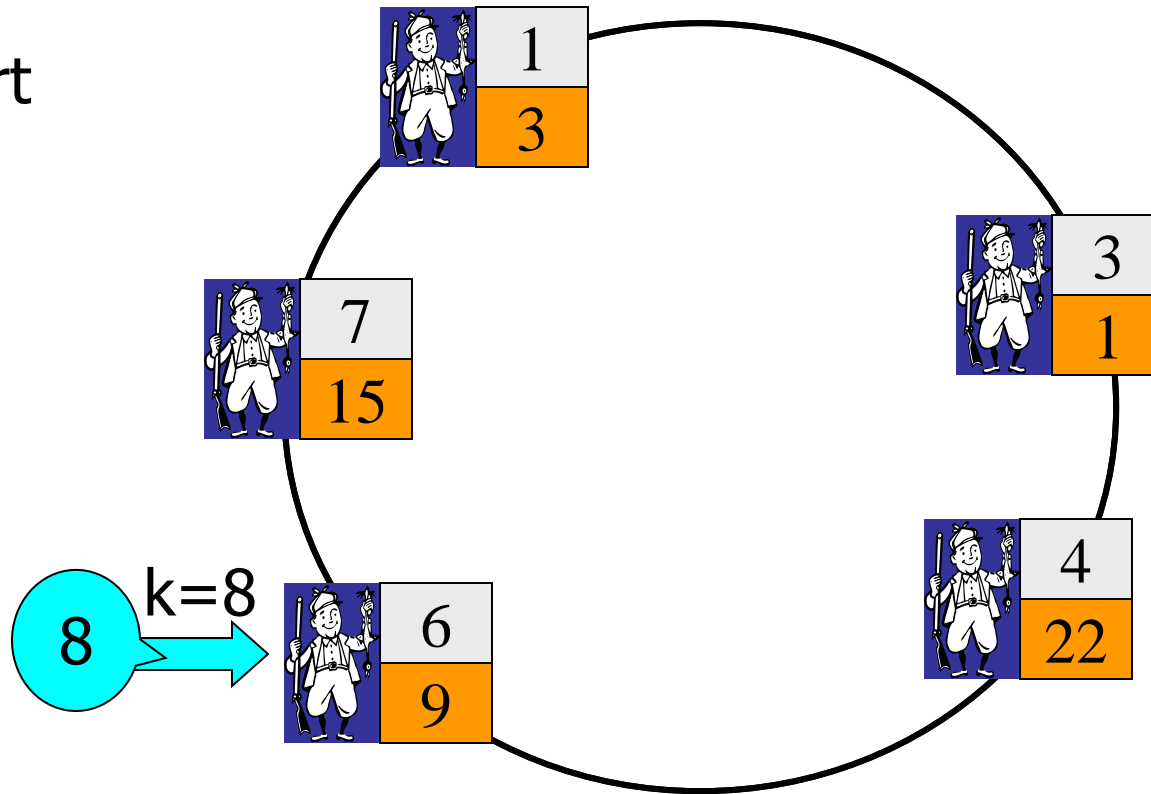
Person 2	2
----------	---



# 约瑟夫环(Joseph Circle)

■ 例:

start



k: 计数

m: 密码

➡ 出队序列: 

Character 1	5
-------------	---

Character 2	2
-------------	---



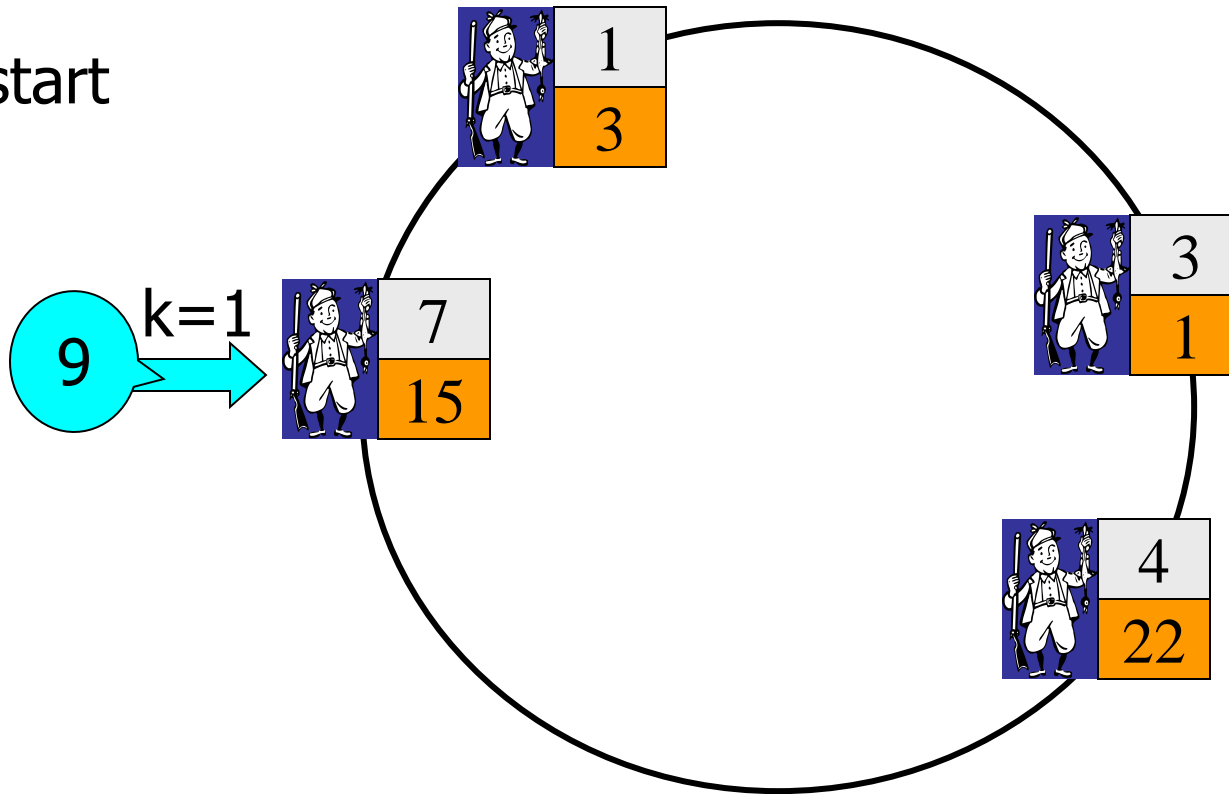
# 约瑟夫环(Joseph Circle)

■ 例:

start

k: 计数

m: 密码



👉 出队序列:

Person	Top Number (Grey)	Bottom Number (Orange)
1	5	
2	2	
3	6	





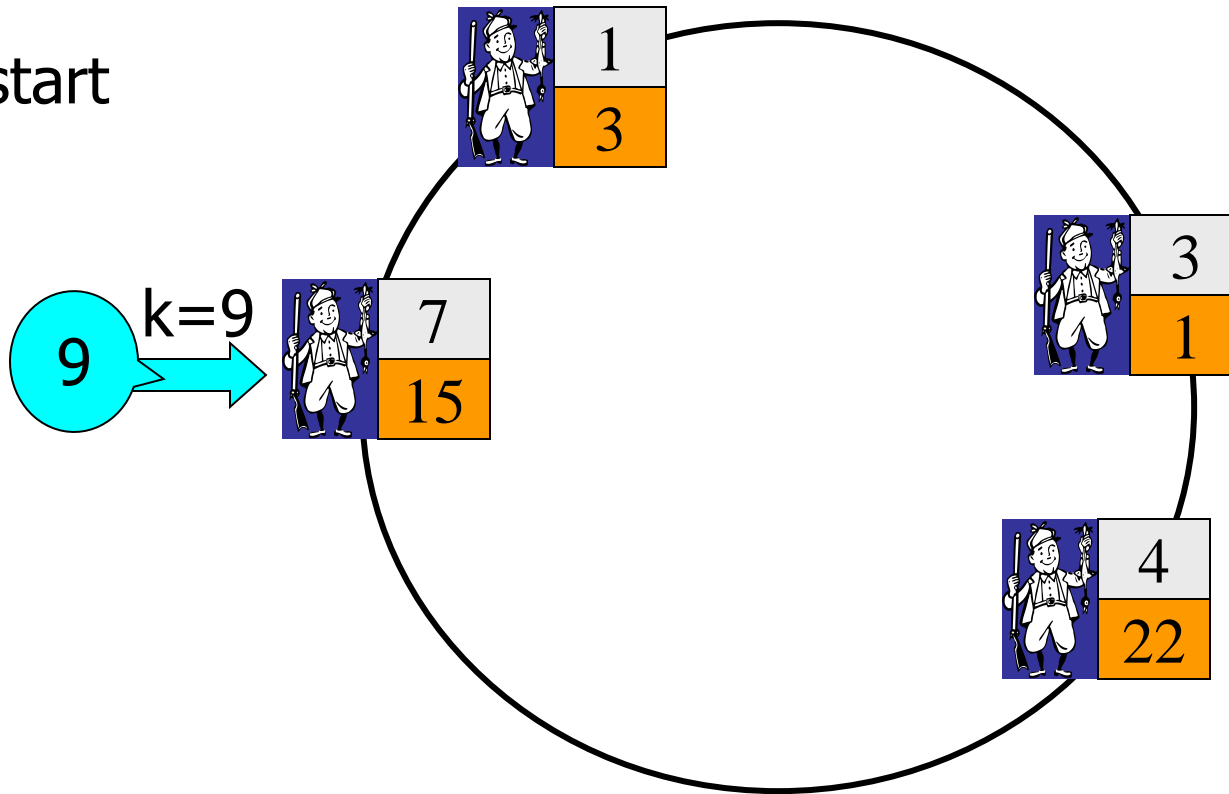
# 约瑟夫环(Joseph Circle)

■ 例:

start

k: 计数

m: 密码



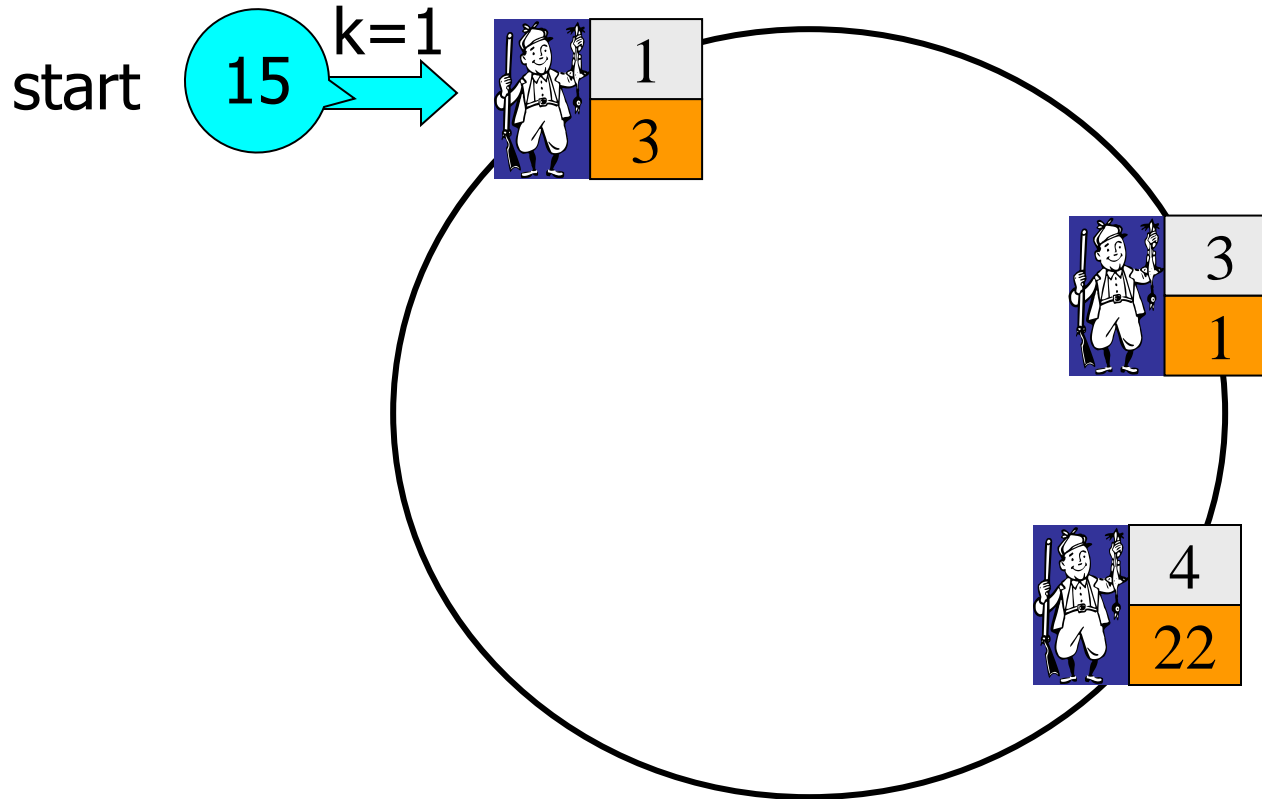
➡ 出队序列:

	5
	2
	6







# 约瑟夫环(Joseph Circle)

■ 例:



k: 计数  
m: 密码

➡ 出队序列:  5  2  6  7



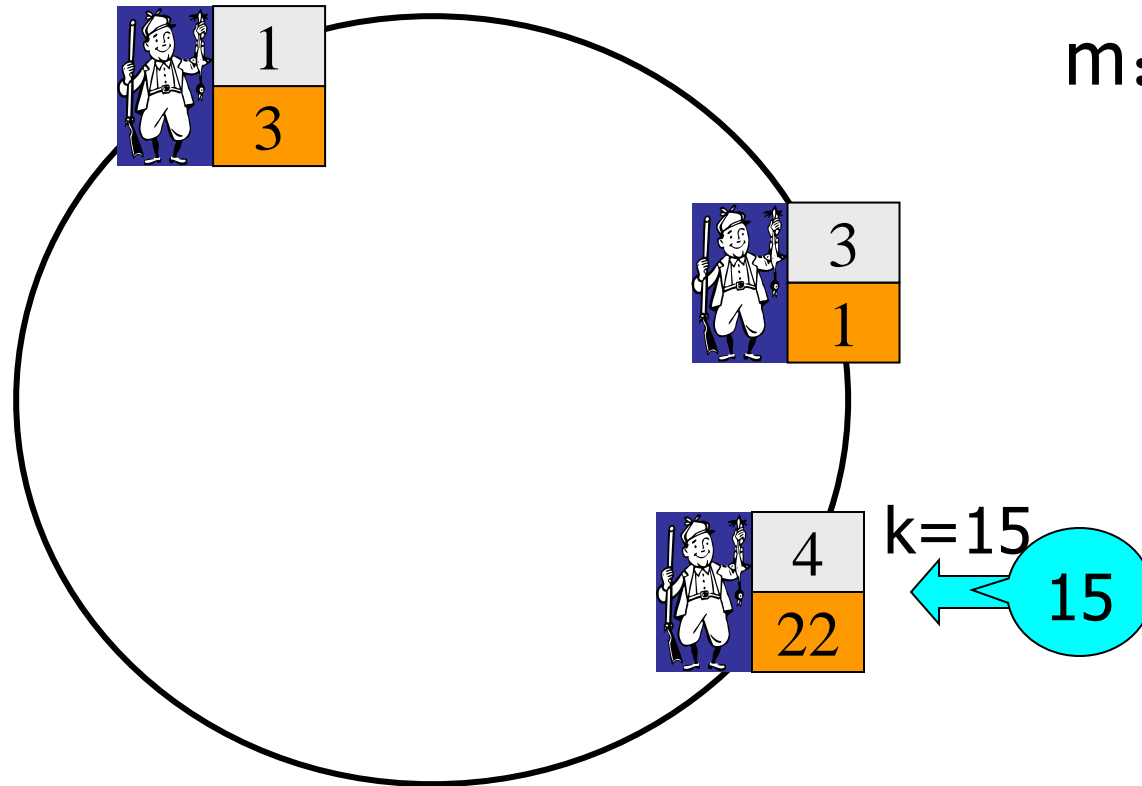
# 约瑟夫环(Joseph Circle)

■ 例:

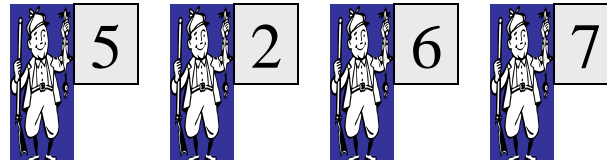
start

k: 计数

m: 密码



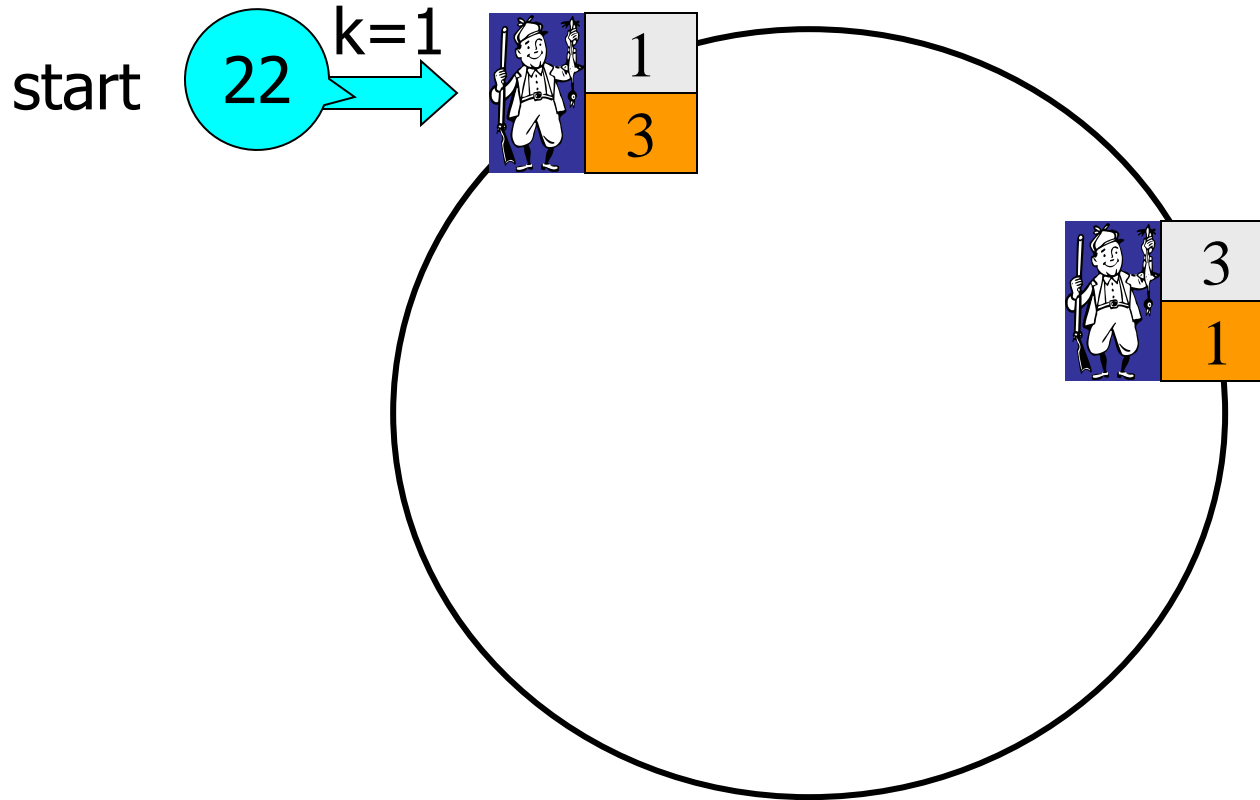
➡ 出队序列:





# 约瑟夫环(Joseph Circle)

■ 例:



k: 计数  
m: 密码

➡ 出队序列:

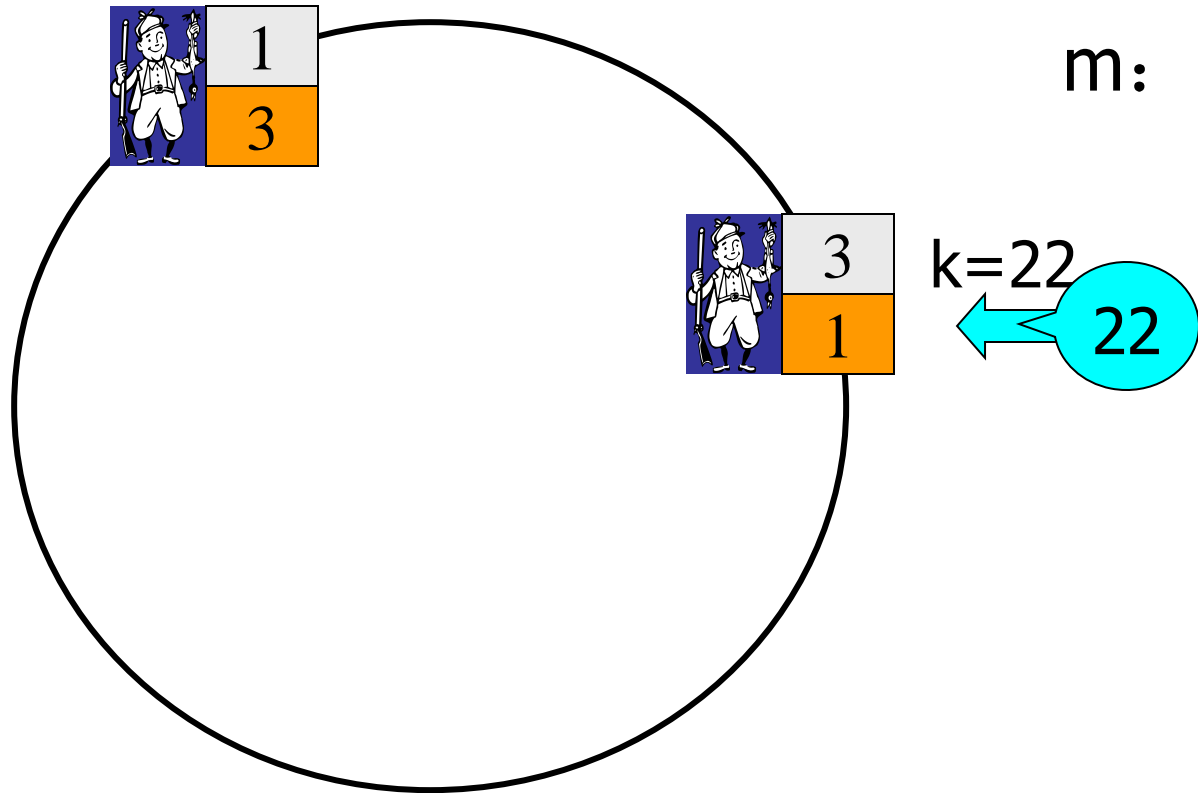
The sequence shows five nodes, each with a soldier icon and a box. The boxes contain the numbers 5, 2, 6, 7, and 4 in order from left to right.



# 约瑟夫环(Joseph Circle)

■ 例:

start



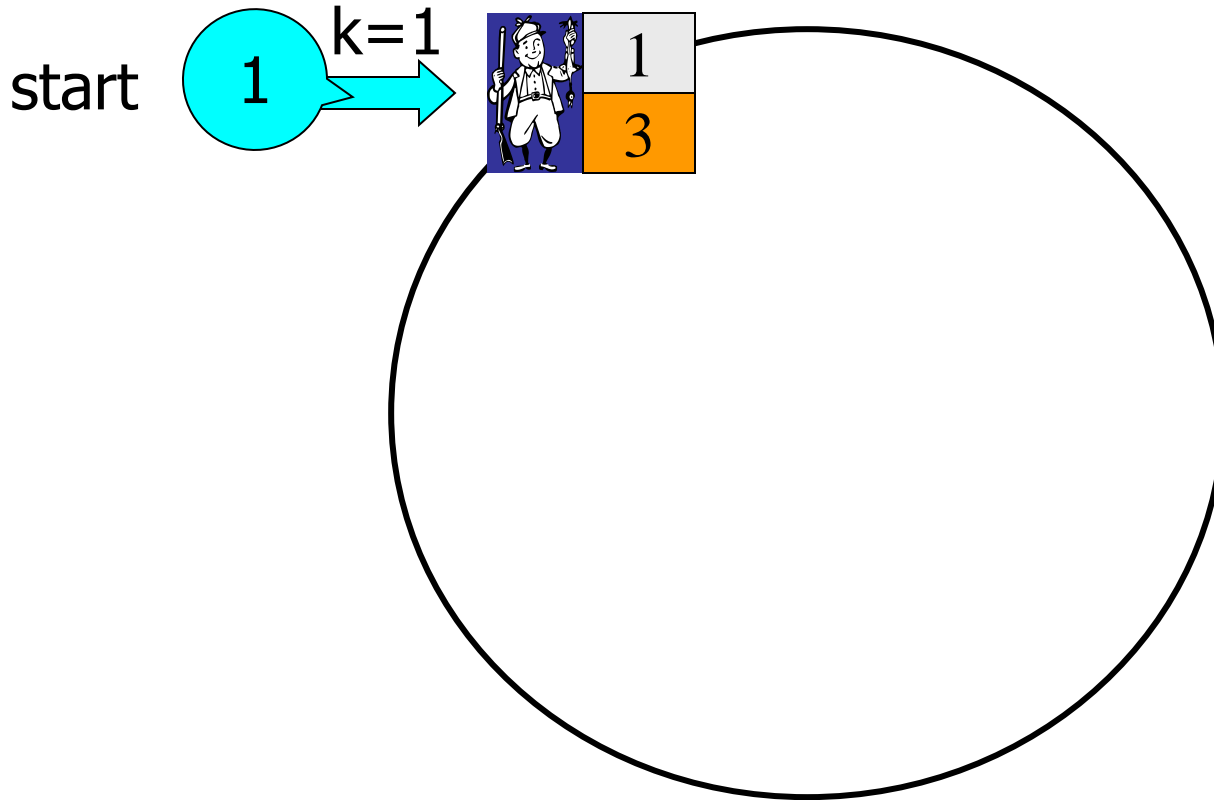
➡ 出队序列:





# 约瑟夫环(Joseph Circle)

■ 例:



k: 计数  
m: 密码

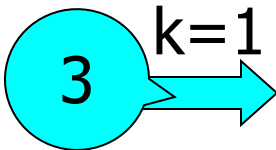
➡ 出队序列:

	5		2		6		7		4		3
--	---	--	---	--	---	--	---	--	---	--	---

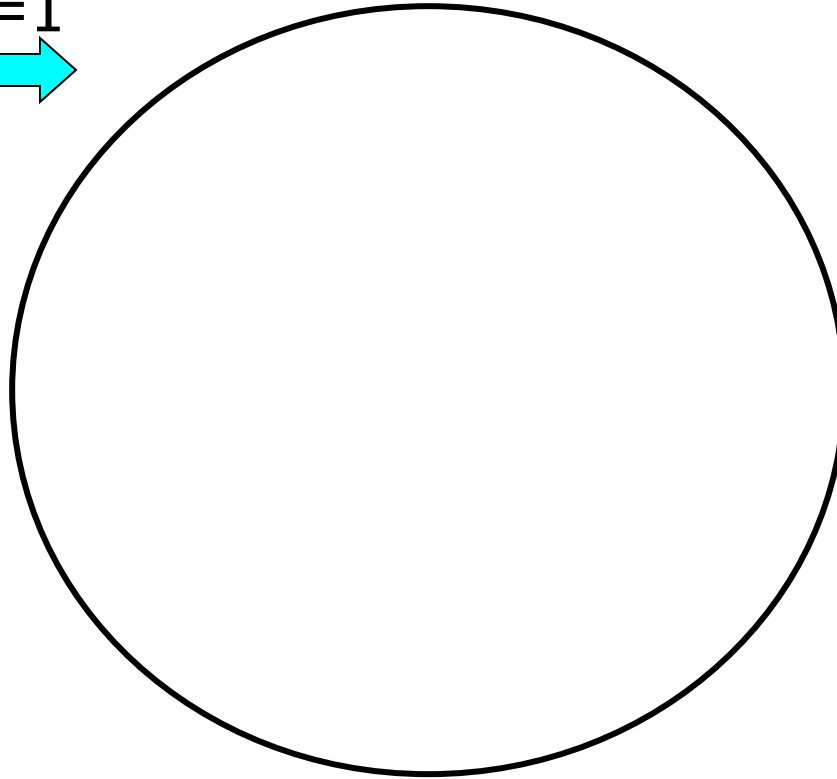


# 约瑟夫环(Joseph Circle)

■ 例:

start 

k: 计数  
m: 密码



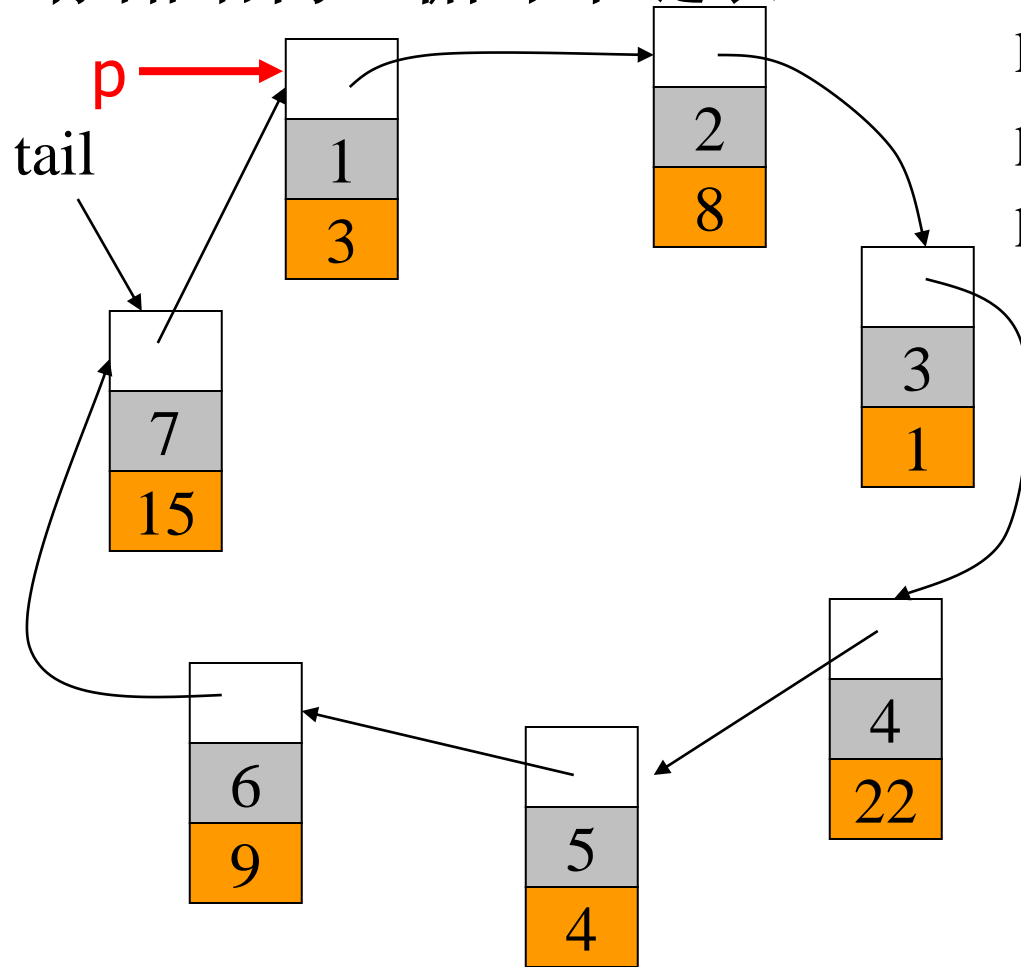
➡ 出队序列:





# 约瑟夫环(Joseph Circle)

## ■ 存储结构（循环单链表） 变量：



k: 计数

p: 指向计数时的当前结点

pre: 指向p的前驱结点

上机实现提示：

- ① 建立循环链表
- ② 测试链表
- ③ 编写游戏过程代码





# Joseph问题代码

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
typedef struct node{
```

```
    int no;
```

```
    /*游戏者的编号*/
```

```
    unsigned int pwd;  
    /*游戏者持有的密  
    码*/
```

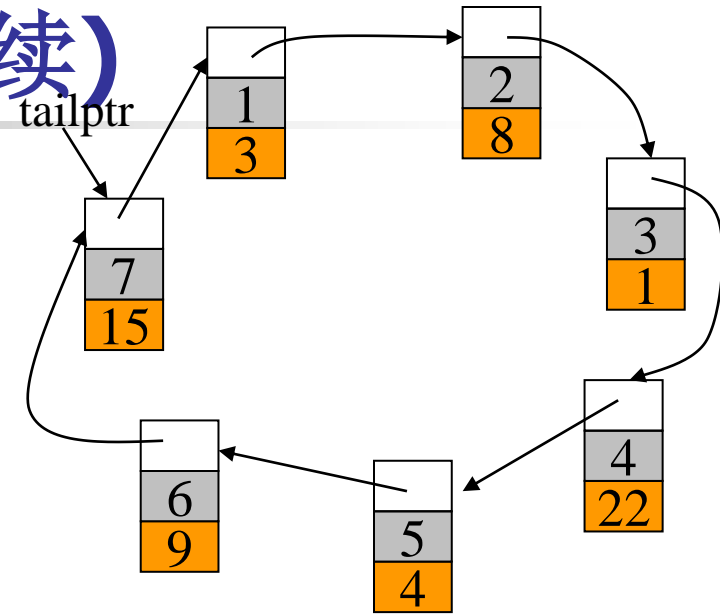
```
    /*游戏者持有的密
```

```
    struct node *next;
```

```
}Node,*LinkList;
```



# Joseph问题代码(续)



```
int main(void) {  
    LinkList tailptr;  
    int n, initial_code;  
    printf("input the number of players and initial password:");  
    scanf("%d %d",&n,&initial_code);  
    tailptr = create_list(n);    /*创建含有n个结点的单循环链表*/  
    if (tailptr) {  
        output_list(tailptr->next); /*输出循环链表中结点的信息*/  
        Joseph(tailptr, n, initial_code);  
    }  
    return 0;  
}
```



## Joseph问题代码(续)

/\*用尾插法创建一个结点数为n的单循环链表，返回值为尾结点的指针\*/

```
LinkedList create_list(int n){
```

```
LinkedList p, rear;  int i;
```

```
    /*先创建循环链表的第一个结点*/
```

```
    p = (Node *)malloc(sizeof(Node));
```

```
    if (!p) {
```

```
        printf("memory allocation error!\n");  return NULL;
```

```
    }
```

```
    printf("input password:");
```

```
    scanf("%d",&(p->pwd));
```

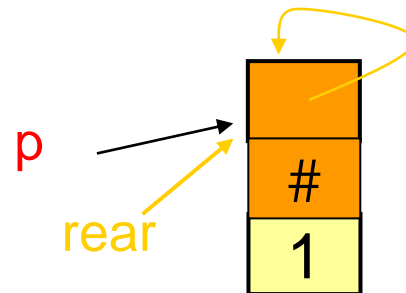
```
    p->no = 1;
```

```
    p->next = p;    rear = p;
```

```
    创建循环链表的其余n-1个结点;
```

```
    返回表尾结点指针;
```

```
}/* create_list */
```





## Joseph问题代码(续)

```
LinkList create_list(int n)
```

```
{ LinkList p, rear; int i;
```

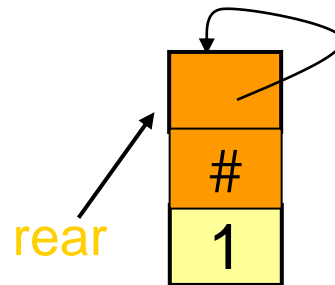
创建循环链表的第一个结点，并令**rear**指向该结点;

```
for(i = 2; i <= n; i++) /*创建循环链表的其余n-1个结点*/  
{
```

```
 }/*for*/
```

```
 return rear;
```

```
 }/* create_list */
```





## Joseph问题代码(续)

```
LinkedList create_list(int n)
```

```
{ LinkedList p, rear; int i;
```

创建循环链表的第一个结点，并令**rear**指向该结点

```
for(i = 2; i <= n; i++) /*创建循环链表的其余n-1个结点*/
```

```
{ p = (Node *)malloc(sizeof(Node));
```

```
if (!p) {
```

```
    printf("memory allocation error!\n"); return NULL;
```

```
}
```

```
printf("input password:"); scanf("%d",&(p->pwd));
```

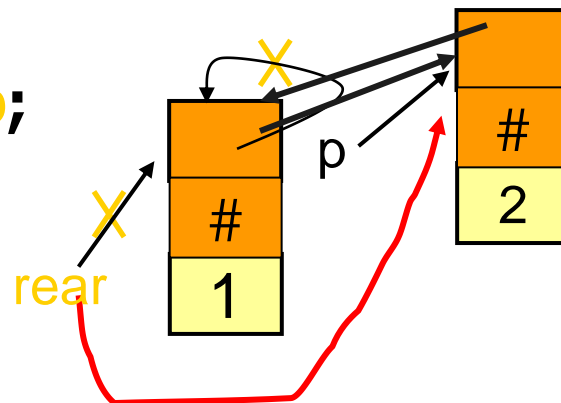
```
p->no = i;
```

```
rear -> next = p; rear = p;
```

```
}/*for*/
```

```
return rear;
```

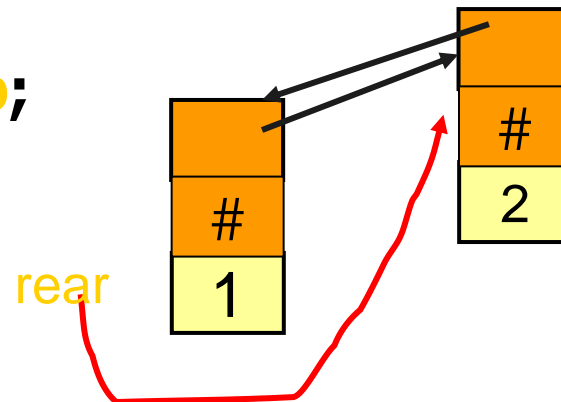
```
}/* create_list */
```





## Joseph问题代码(续)

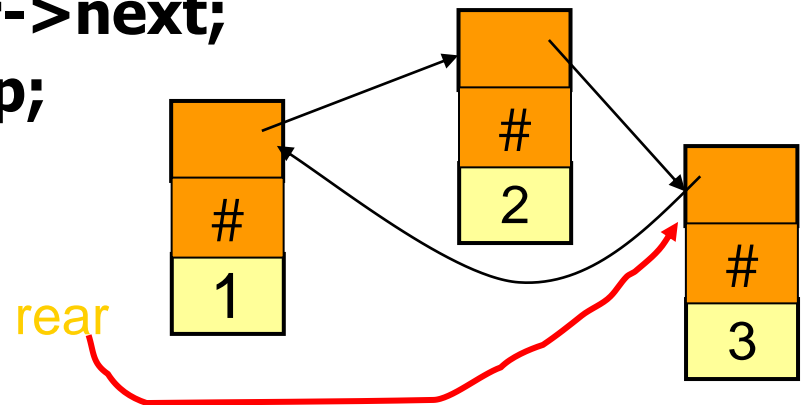
```
LinkedList create_list(int n)
{
    LinkedList p, rear;  int i;
    创建循环链表的第一个结点，并令rear指向该结点
    for(i = 2; i <= n; i++) /*创建循环链表的其余n-1个结点*/
    {
        p = (Node *)malloc(sizeof(Node));
        if (!p) {
            printf("memory allocation error!\n"); return NULL;
        }
        printf("input password:");  scanf("%d",&(p->pwd));
        p->no = i;
        rear -> next = p;    rear = p;
    }/*for*/
    return rear;
}/* create_list */
```





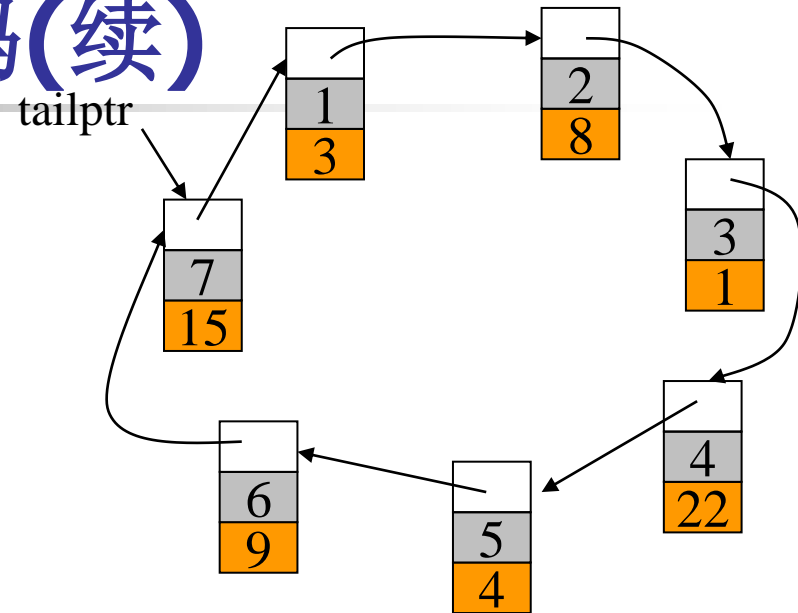
## Joseph问题代码(续)

```
LinkedList create_list(int n)
{
    LinkedList p, rear;  int i;
    创建循环链表的第一个结点，并令rear指向该结点;
    for(i = 2; i <= n; i++) /*创建循环链表的其余n-1个结点*/
    {
        p = (Node *)malloc(sizeof(Node));
        if (!p) {
            printf("memory allocation error!\n"); return NULL;
        }
        printf("input password:");  scanf("%d",&(p->pwd));
        p->no = i;  p->next = rear->next;
        rear->next = p;  rear = p;
    }/*for*/
    return rear;
}/* create_list */
```





# Joseph问题代码(续)



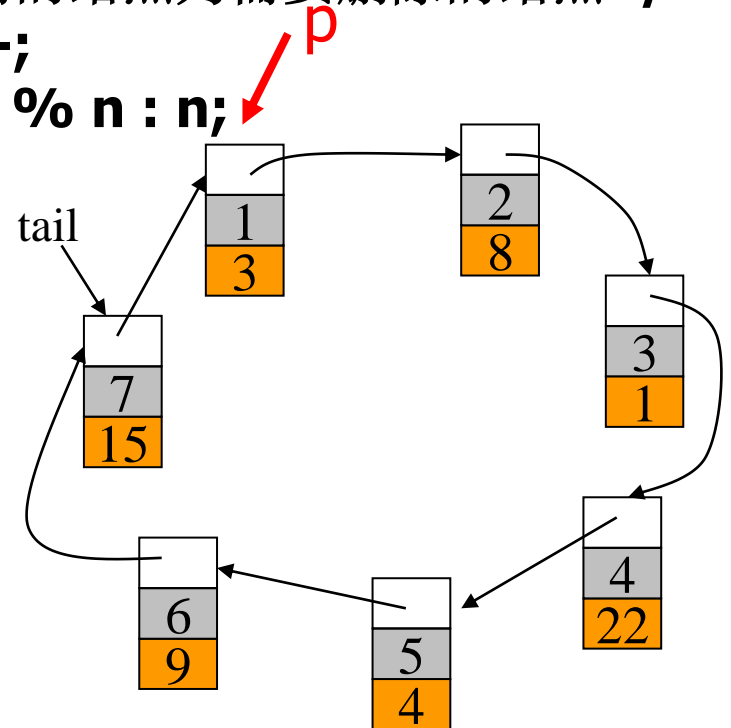
```
int main(void) {  
    LinkList tailptr;  
    int n, initial_code;  
    printf("input the number of players and initial password:");  
    scanf("%d %d",&n,&initial_code);  
    tailptr = create_list(n);          /*创建含有n个结点的单循环链表*/  
    if (tailptr) {  
        output_list(tailptr->next); /*输出循环链表中结点的信息*/  
        Joseph(tailptr, n, initial_code ;  
    }  
    return 0;  
}
```





## Joseph问题代码(续)

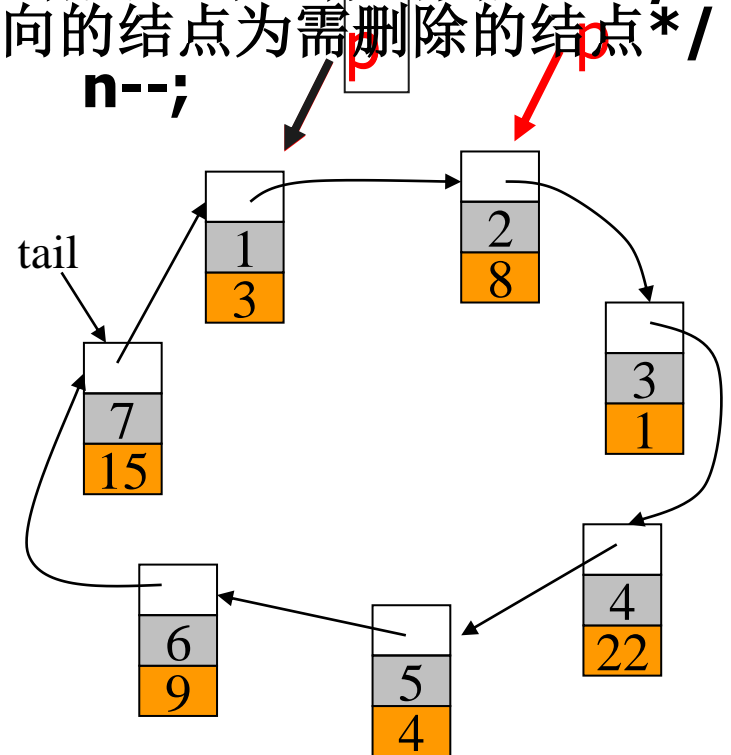
```
void Joseph(LinkList tail, int n, int m)
{ LinkList pre, p;
  int k;    m = m % n ? m % n : n;
  pre = tail; p = tail -> next; k = 1;
  while (n > 1) { /*圈中多于1个人时循环*/
    if (k == m) { /*数到需要出去的人(结点)时进行出圈处理*/
      printf("%4d", p->no); /*p指向的结点为需要删除的结点*/
      pre -> next = p -> next;    n--;
      m = p -> pwd % n ? p -> pwd % n : n;
      free(p);
      p = pre -> next;
      k = 1;
    }
    else {
      k++; pre = p; p = p->next;
    }
  } /*while*/
  printf("%4d\n", p->no);
} /*Joseph*/
```





## Joseph问题代码(续)

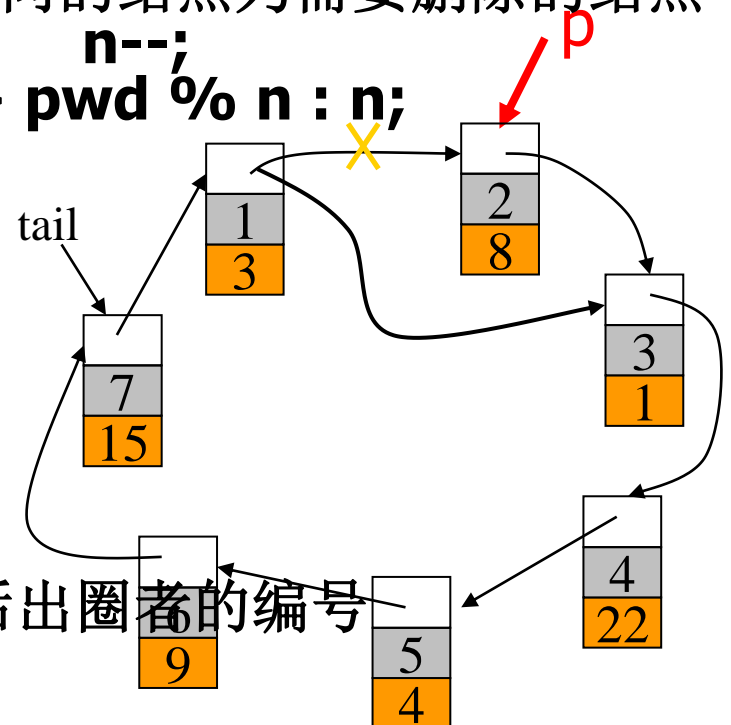
```
void Joseph(LinkList tail, int n, int m) {  
    LinkList pre, p;  
    int k;    // m = m % n ? m % n : n;  
    pre = tail; p = tail -> next; k = 1;  
    while (n > 1) { /*圈中多于1个人时循环*/  
        if (k == m) { /*数到需要出去的人(结点)时进行出圈处理*/  
            printf("%4d", p->no); /*p指向的结点为需删除的结点*/  
            pre -> next = p -> next;    n--;  
            m = p -> pwd;  
            free(p);  
            p = pre -> next;  
            k = 1;  
        }  
        else {  
            k++; pre = p; p = p->next;  
        }  
    } /*while*/  
    printf("%4d\n", p->no);  
} /*Joseph*/
```





# Joseph问题代码(续)

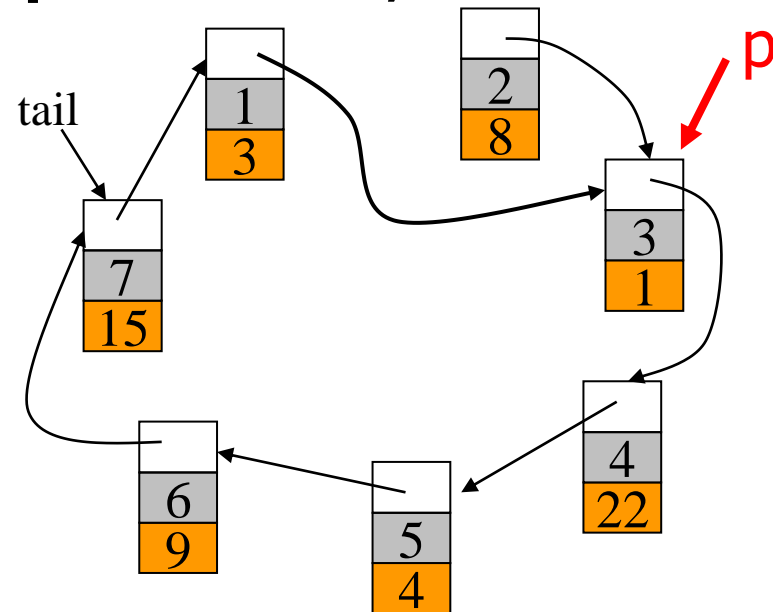
```
void Joseph(LinkList tail, int n, int m)
{ LinkList pre, p;
  int k;    m = m % n ? m % n : n;
  pre = tail; p = tail -> next; k = 1;
  while (n > 1) { /*圈中多于1个人时循环*/
    if (k == m) { /*数到需要出去的人(结点)时进行出圈处理*/
      printf("%4d", p->no); /*p指向的结点为需要删除的结点*/
      pre -> next = p -> next;    n--;
      m = p -> pwd % n ? p -> pwd % n : n;
      free(p);
      p = pre -> next;
      k = 1;
    }
    else {
      k++; pre = p; p = p->next;
    }
  } /*while*/
  printf("%4d\n", p->no); //输出最后出圈者的编号
} /*Joseph*/
```





## Joseph问题代码(续)

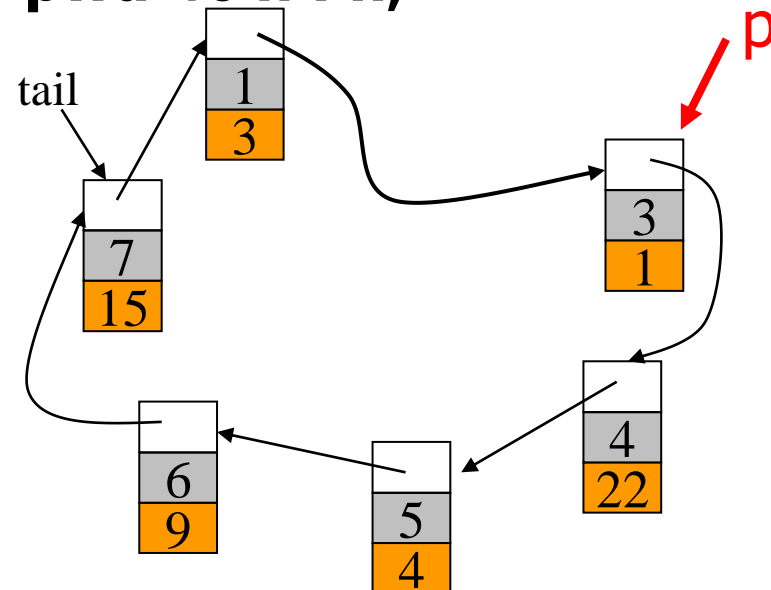
```
void Joseph(LinkList tail, int n, int m)
{
    LinkList pre, p;
    int k; m = m % n ? m % n : n;
    pre = tail; p = tail->next; k = 1;
    while (n > 1) { /* 圈中多于1个人时循环 */
        if (k == m) { /* 数到需要出去的人(结点)时进行出圈处理 */
            printf("%4d", p->no); /* p指向的结点为需要删除的结点 */
            pre->next = p->next; n--;
            m = p->pwd % n ? p->pwd % n : n;
            free(p);
            p = pre->next;
            k = 1;
        }
        else {
            k++; pre = p; p = p->next;
        }
    } /* while */
    printf("%4d\n", p->no); free(p);
} /* Joseph */
```





## Joseph问题代码(续)

```
void Joseph(LinkList tail, int n, int m) {  
    LinkList pre, p;  
    int k; m = m % n ? m % n : n;  
    pre = tail; p = tail -> next; k = 1;  
    while (n > 1) { /*圈中多于1个人时循环*/  
        if (k == m) { /*数到需要出去的人(结点)时进行出圈处理*/  
            printf("%4d", p->no); /*p指向的结点为需要删除的结点*/  
            pre -> next = p -> next; n--;  
            m = p -> pwd % n ? p -> pwd % n : n;  
            free(p);  
            p = pre -> next;  
            k = 1;  
        }  
        else {  
            k++; pre = p; p = p->next;  
        }  
    } /*while*/  
    printf("%4d\n", p->no); free(p);  
} /*Joseph*/
```





## Joseph 问题

```
int main(void){
    LinkList tail;
    int n, it;
    printf("input the number of players and initial
password:");
    scanf("%d %d",&n,&it);
    tail = create_list(n); /*创建单循环链表*/
    if (tail) {
        output_list(tail->next); /*输出循环链表中结点的信息*/
        Joseph(tail, n, it);
    }
    return 0;
}
```

```
void output_list(LinkList head)
{ LinkList p;
  p = head;
  do {
      printf("(%d,%d)\t",p->no,p->pwd);
      p = p->next;
  }while (p != head);
  printf("\n");
}/*output_list*/
```



## 第二章 线性表

### 本章内容

**2.1** 线性表的类型定义

**2.2** 线性表的顺序表示和实现

**2.3** 线性表的链式表示和实现

**2.3.1** 线性链表

**2.3.2** 循环链表

**2.3.3** 双向链表

**2.4** 一元多项式的表示及相加



## 2.3.3 双向链表

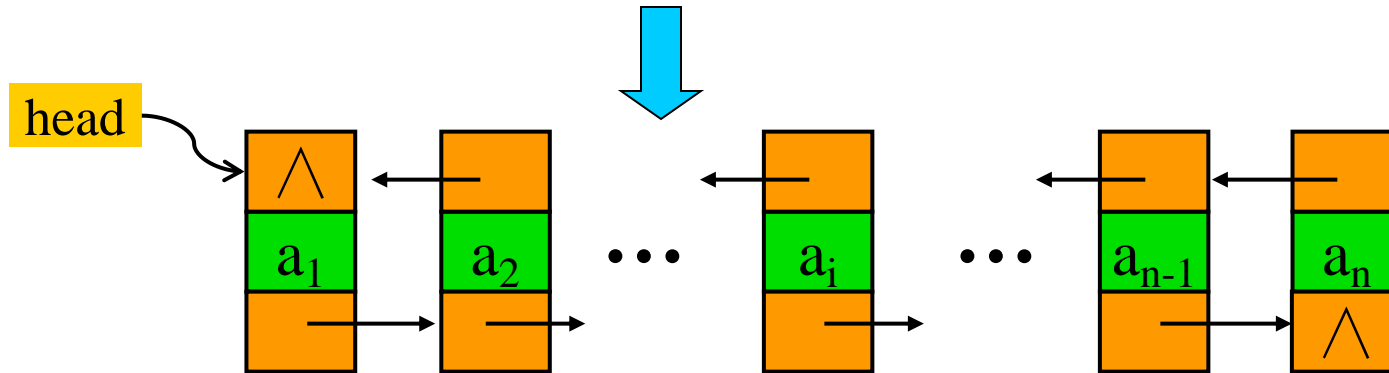




## 2.3.2 循环单链表和双向链表

### ■ 双向链表

线性表:  $(a_1, a_2, \dots, a_i, \dots, a_n)$



双向链表示意图

※ 双向链表的结点结构:



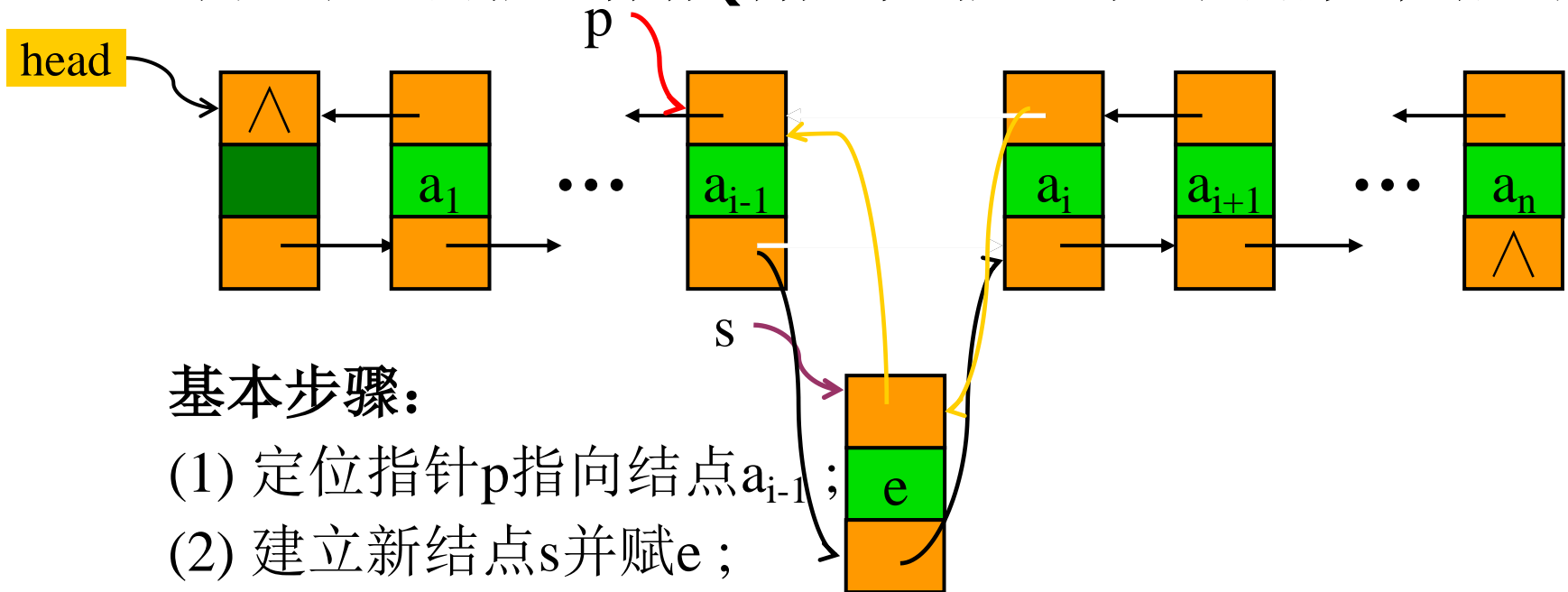
※ 双向链表的C语言描述:

```
typedef struct DuLNode{  
    ElemType data;  
    struct DuLNode *prior;  
    struct DuLNode *next;  
} DuLNode, *DulinkList;
```



## 2.3.2 循环单链表和双向链表

- 双向链表上的插入操作(将元素 $e$ 插入到链表的第 $i$ 个结点前)



基本步骤:

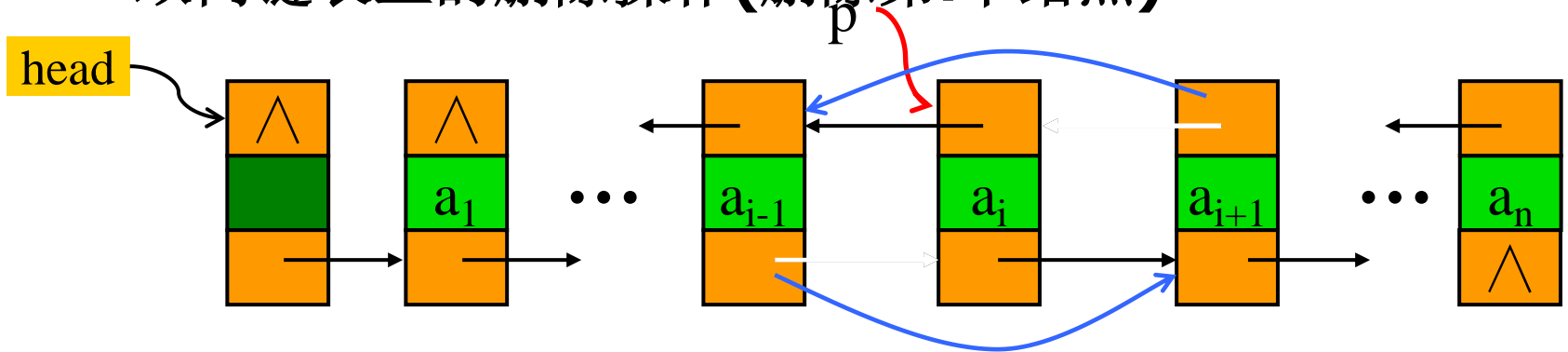
- (1) 定位指针 $p$ 指向结点 $a_{i-1}$ ;
- (2) 建立新结点 $s$ 并赋 $e$ ;
- (3) 修改 $s$ 的 $next$ 指针域指向 $p$ 下一结点:  $s \rightarrow next = p \rightarrow next$ ;
- (4) 修改 $s$ 的 $prior$ 指针域指向 $p$ 结点:  $s \rightarrow prior = p$ ;
- (5) 修改 $p$ 的 $next$ 指针域指向 $s$ 结点:  $p \rightarrow next = s$ ;
- (6) 修改 $s$ 下一结点的 $prior$ 指针域指向 $s$ :  $s \rightarrow next \rightarrow prior = s$ ;

**问题:** ①修改指针的步骤是否可改变? ②头结点作用?



## 2.3.2 循环单链表和双向链表

### ■ 双向链表上的删除操作(删除第*i*个结点)



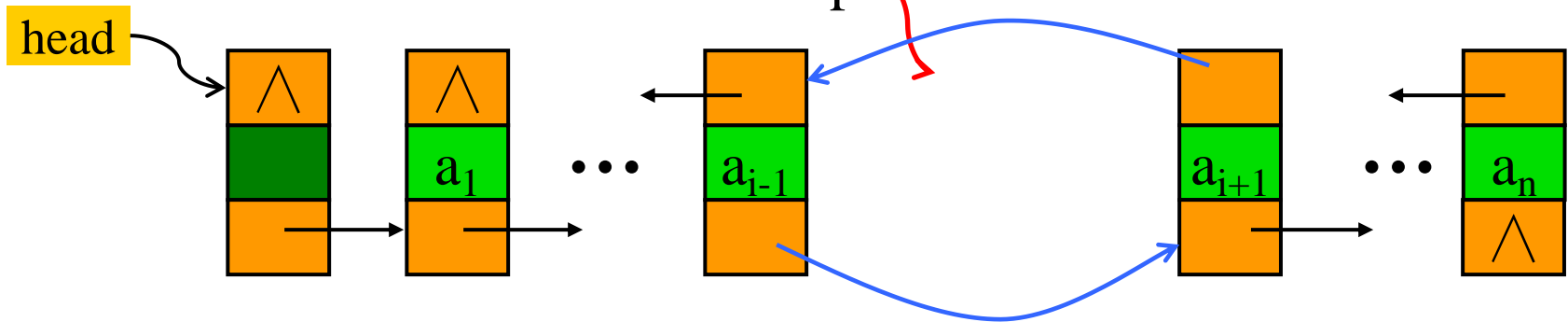
基本步骤:

- (1) 定位指针 $p$ 指向结点 $a_i$ ;
- (2) 修改 $p$ 的前一结点的 $next$ 指针域指向 $p$ 下一结点:  
 $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$
- (3) 修改 $p$ 的下一结点的 $prior$ 指针域指向 $p$ 前一结点:  
 $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$
- (3) 释放结点 $p$ 。



## 2.3.2 循环单链表和双向链表

### ■ 双向链表上的删除操作(删除第 $i$ 个结点)



基本步骤:

- (1) 定位指针 $p$ 指向结点 $a_i$ ;
- (2) 修改 $p$ 的前一结点的next指针域指向 $p$ 下一结点:  
 $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$
- (3) 修改 $p$ 的下一结点的prior指针域指向 $p$ 前一结点:  
 $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$
- (4) 释放结点 $p$ :  $\text{free}(p);$



## 第二章 线性表

### 本章内容

**2.1 线性表的类型定义**

**2.2 线性表的顺序表示和实现**

**2.3 线性表的链式表示和实现**

**2.3.1 线性链表**

**2.3.2 循环链表**

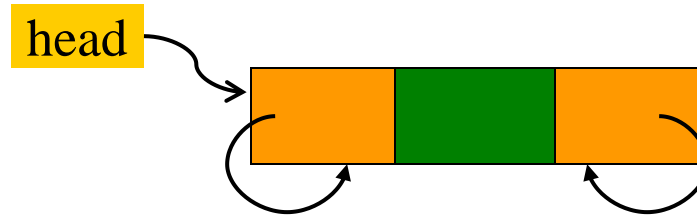
**2.3.3 双向链表**

**2.4 一元多项式的表示及相加**

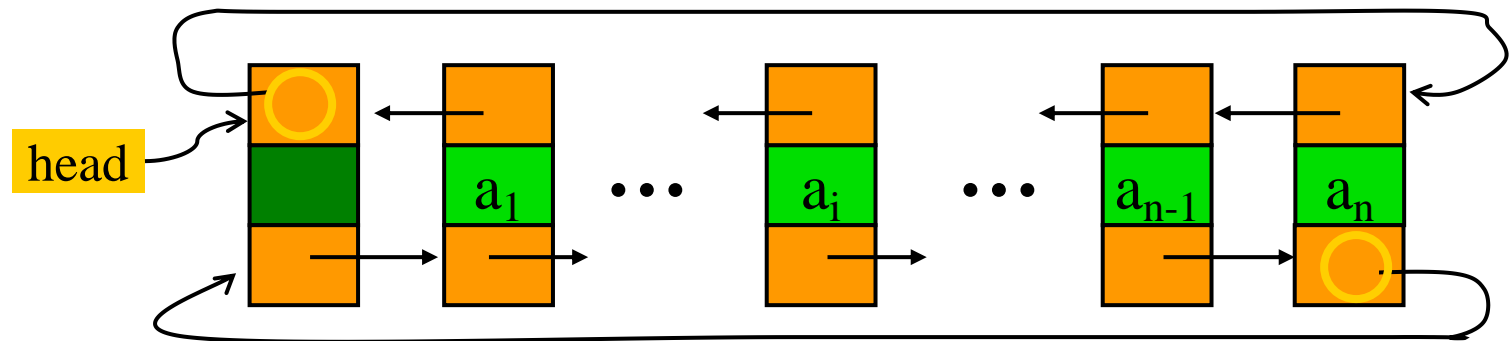


## 2.3.2 循环单链表和双向链表

### ■ 双向循环链表



带头结点的空双向循环链表



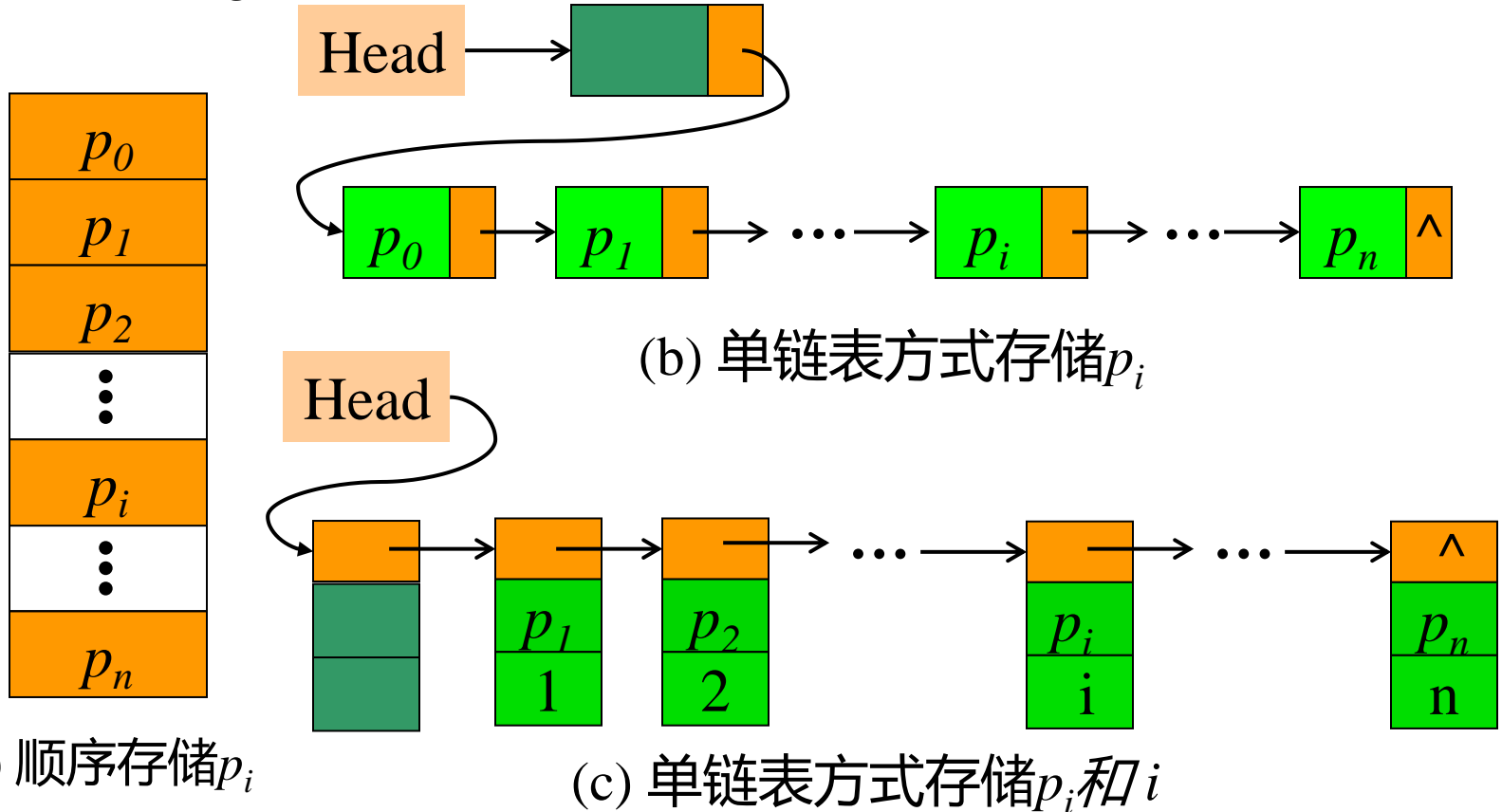
带头结点的非空双向循环链表



## 2.4 一元多项式的表示及相加

### ■ 一元多项式的形式:

$$A(x) = p_0 + p_1x + p_2x^2 + \dots + p_ix^i + \dots + p_nx^n$$



**问题：** 各存储方式的优缺点和适用场合？



## 2.4 一元多项式的

■ 例

$$A(x) = 7 + 3x + 9x^8 + 5x^{17}$$

$$B(x) = 8x + 22x^7 - 9x^8$$

$$C(x) = A(x) + B(x)$$

$$= 7 + 11x + 22x^7 + 5x^{17} - 9x^8$$

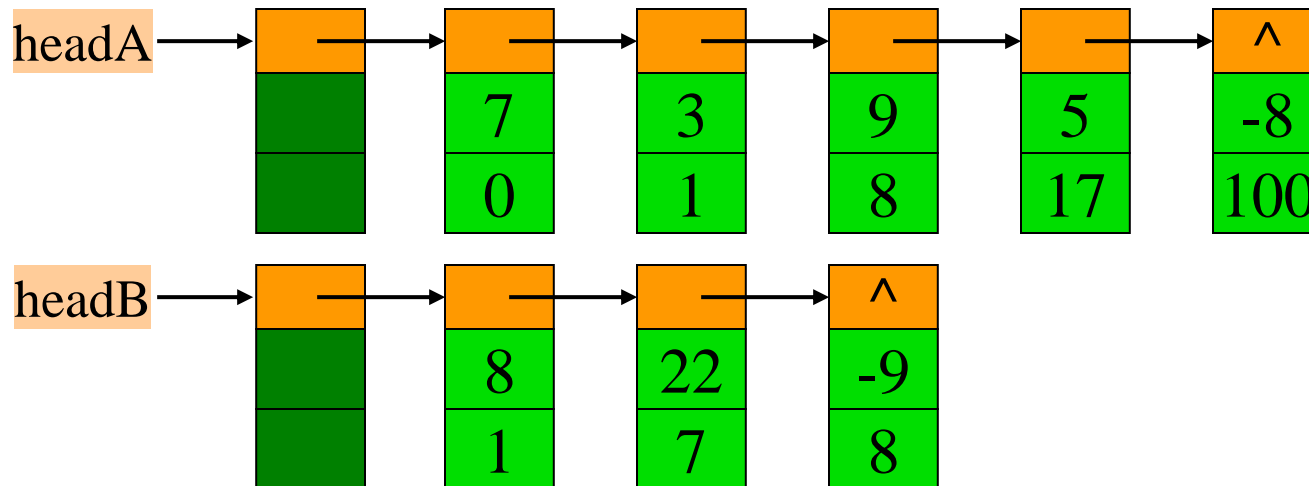
//-----多项式的项-----

```
typedef struct {
    float coef; //系数
    int expn; //指数
} term, ElemType;
```

//-----多项式结点-----

```
typedef struct LNode{
    ElemType data; //数据域
    struct LNode *next; //指针域
} *polynomial;
```

✱ 存储方式的选择——以单链表存储多项式系数和指数







## 2.4 一元多项式的表示及相加

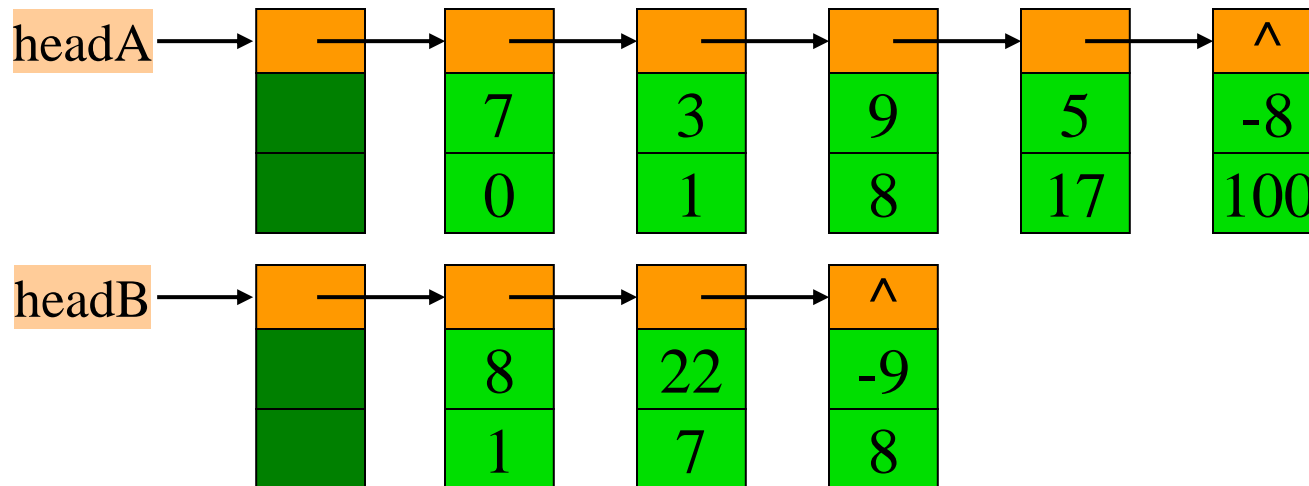
■ 例

$$A(x) = 7 + 3x + 9x^8 + 5x^{17} - 8x^{100}$$

$$B(x) = 8x + 22x^7 - 9x^8$$

$$C(x) = A(x) + B(x) = 7 + 11x + 22x^7 + 5x^{17} - 8x^{100}$$

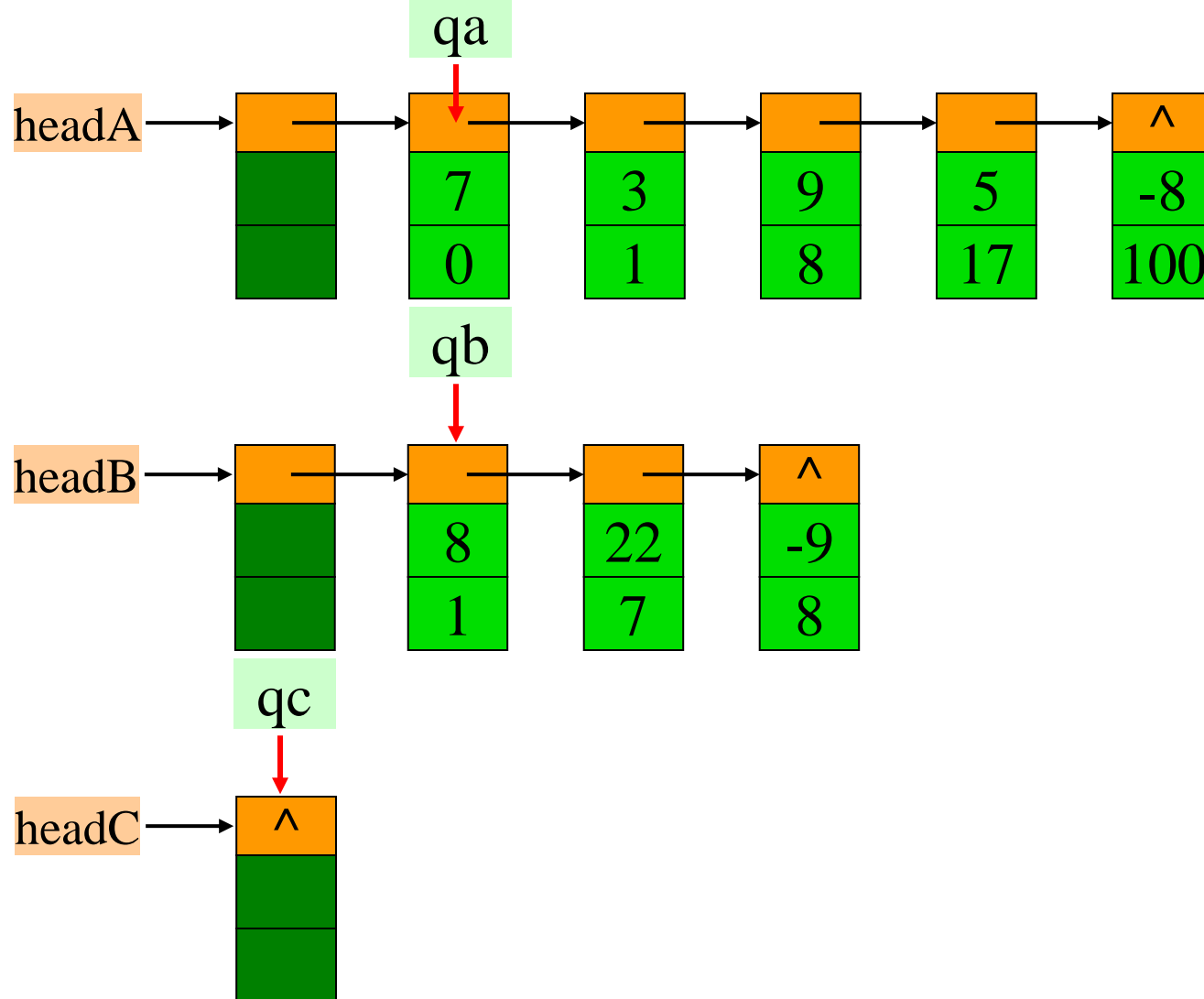
✱ 实现方式：是否保留A、B的结点？





## 2.4 一元多项式的表示及相加

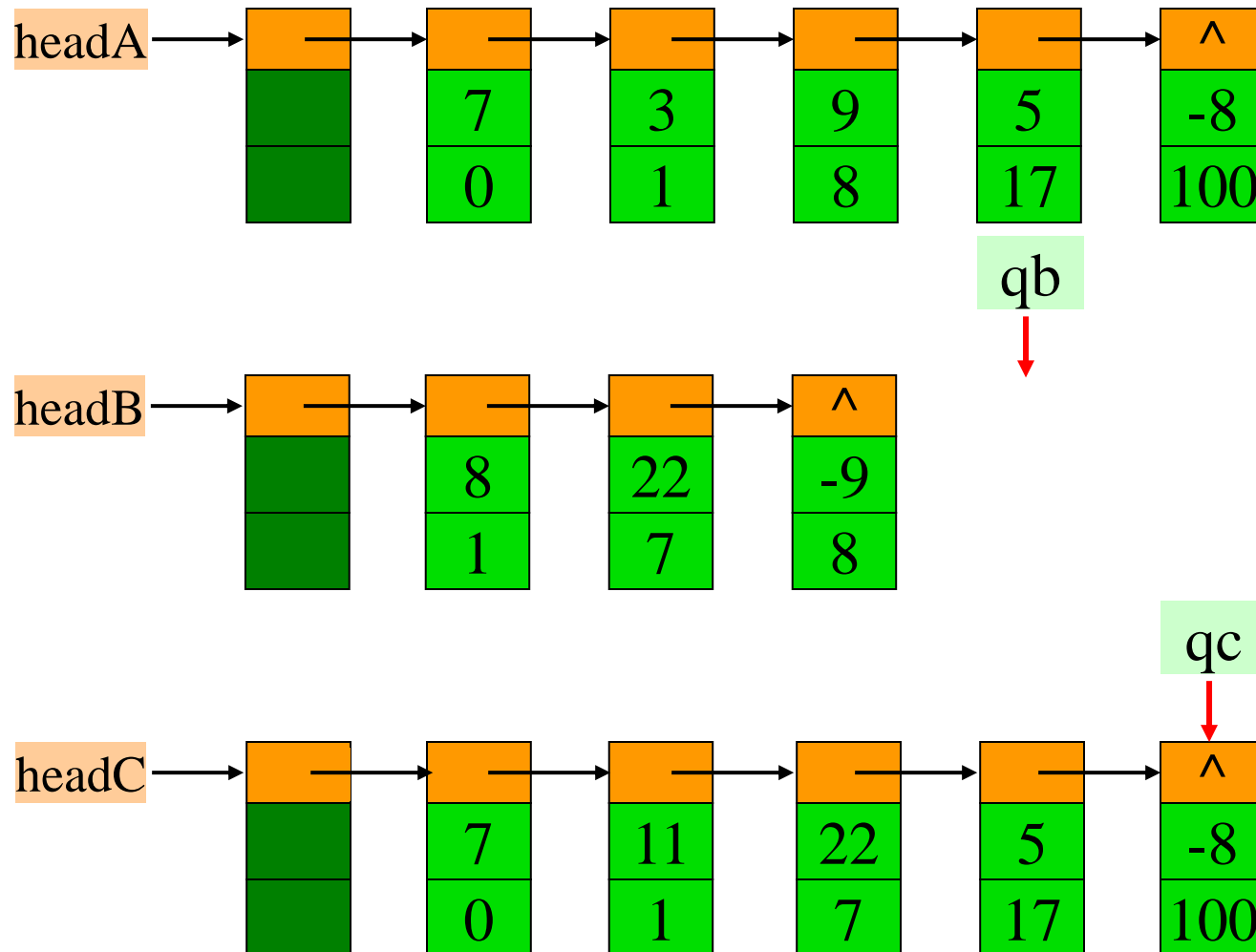
### ■ 一元多项式的相加实现（保留原多项式链表）





## 2.4 一元多项式的表示及相加

## ■ 一元多项式的相加实现（保留原多项式链表）





## 2.4 一元多项式的表示及相加

### ■ 一元多项式 $A(x)$ 、 $B(x)$ 的相加过程

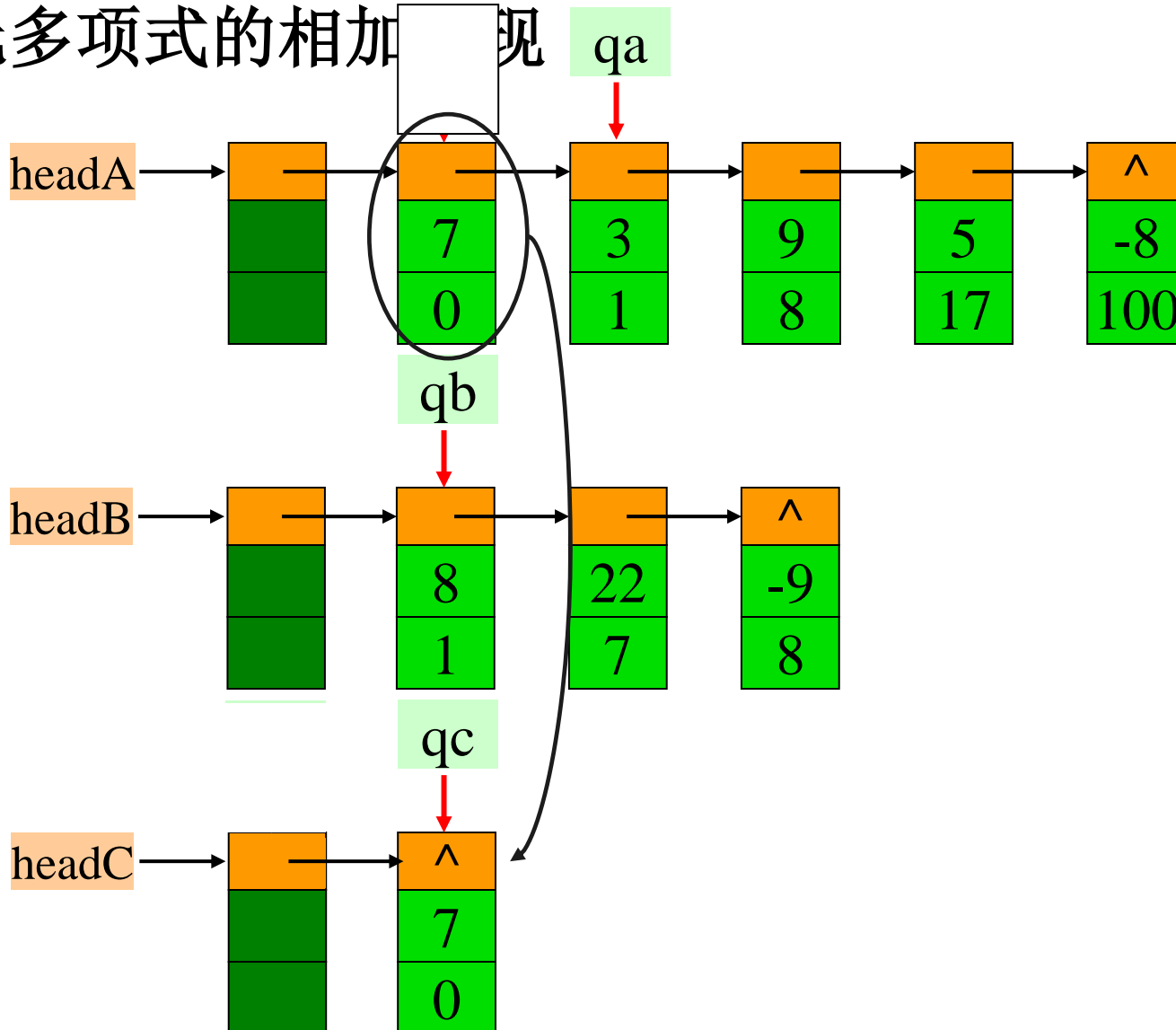
基本步骤:

- (1) 设置指针 $qa$ 和 $qb$ 分别指向  $A$ 、 $B$  多项式中的某一项;
- (2) 比较 $qa$ 和 $qb$ 所指向的项的指数:
  - a. 指数相等, 进行处理 (系数相加)
  - b.  $qa$ 指向的项的指数小, 复制 $qa$ 指向的项, 进行处理
  - c.  $qb$ 指向的项的指数小, 复制 $qb$ 指向的项, 进行处理
- (3) 重复(1), 直到某个多项式结束;
- (4) 将未结束多项式的剩余项拼接上和多项式后。



## 2.4 一元多项式的表示及相加

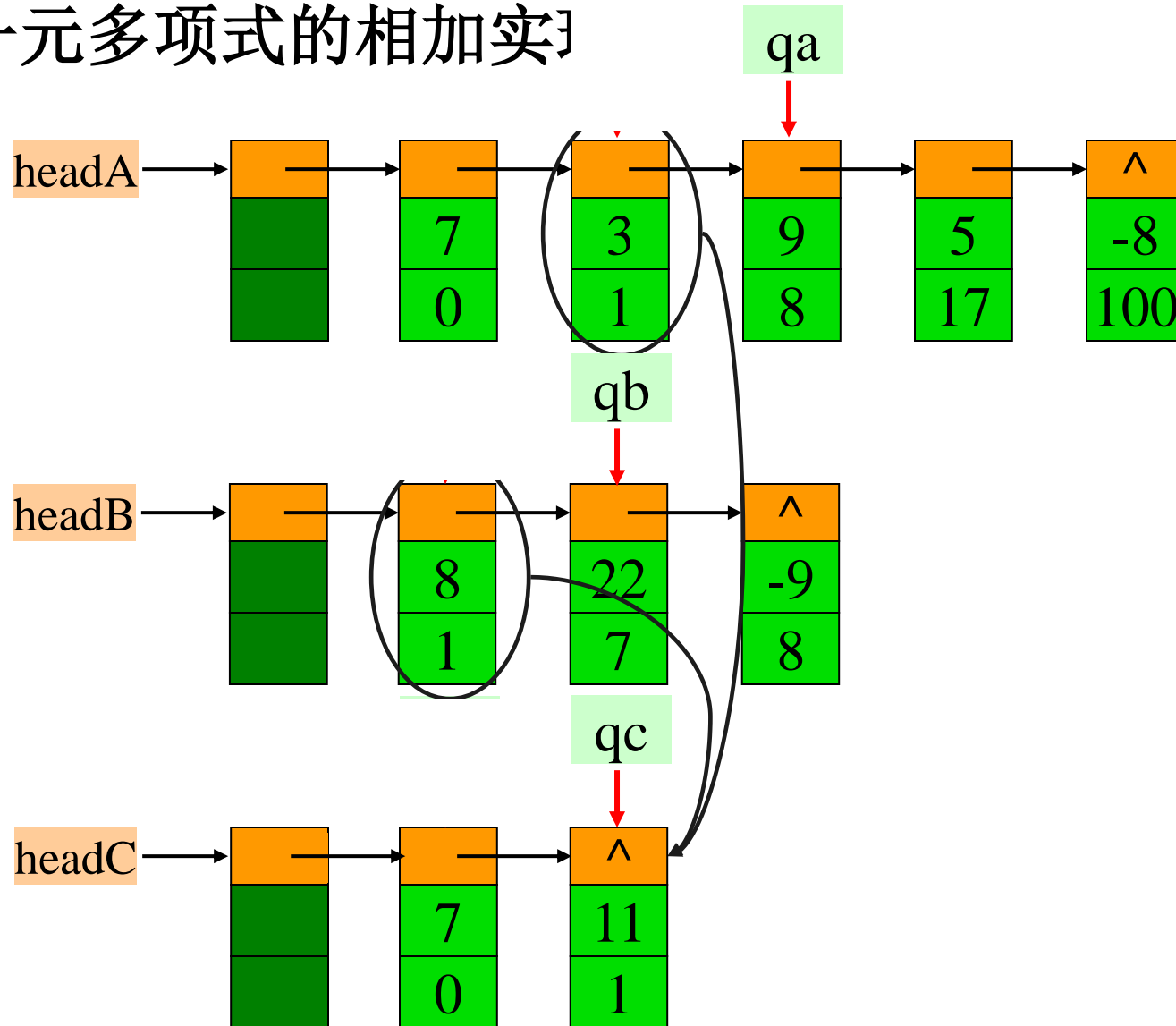
### ■ 一元多项式的相加实现





## 2.4 一元多项式的表示及相加

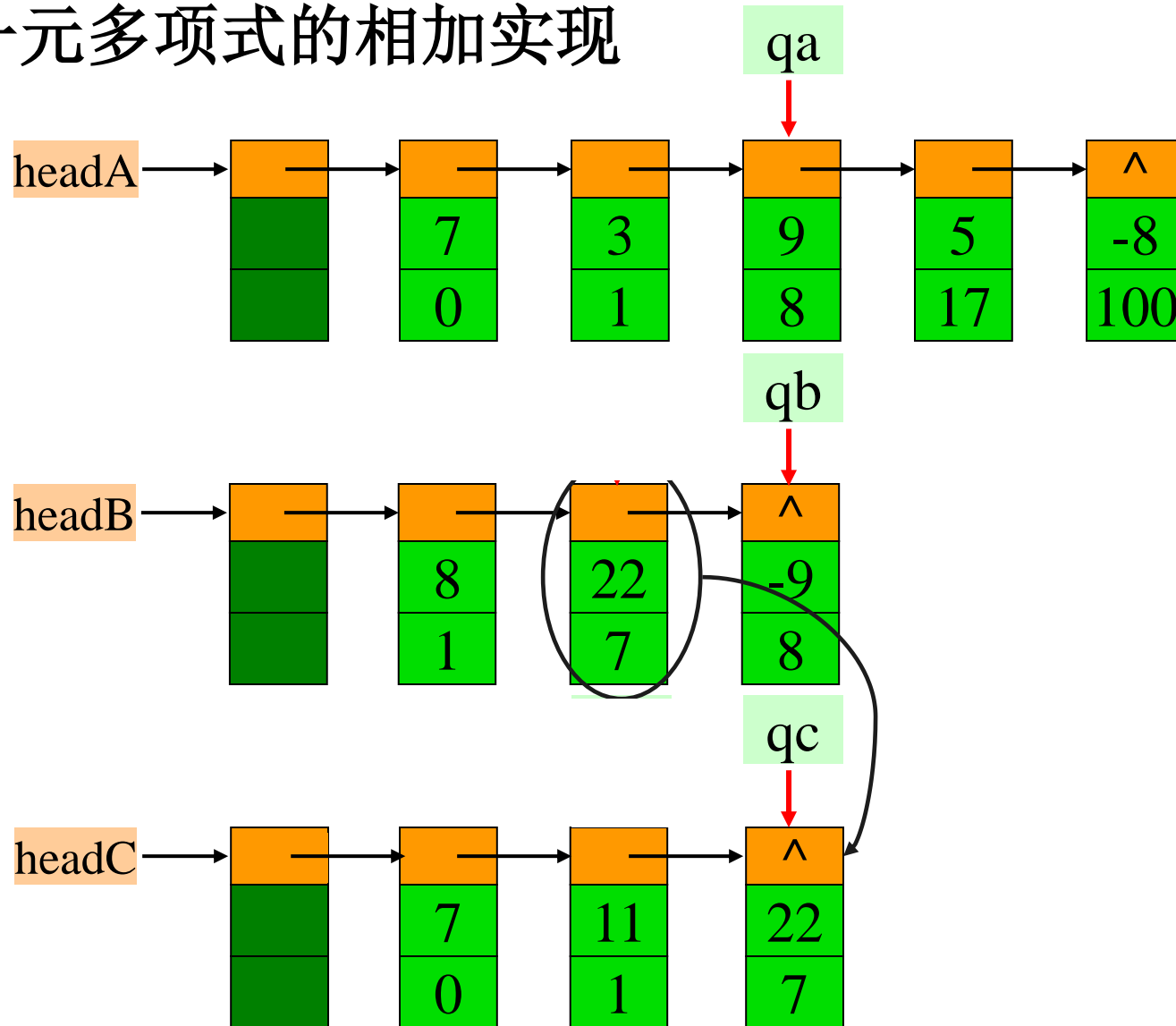
### ■ 一元多项式的相加实现





## 2.4 一元多项式的表示及相加

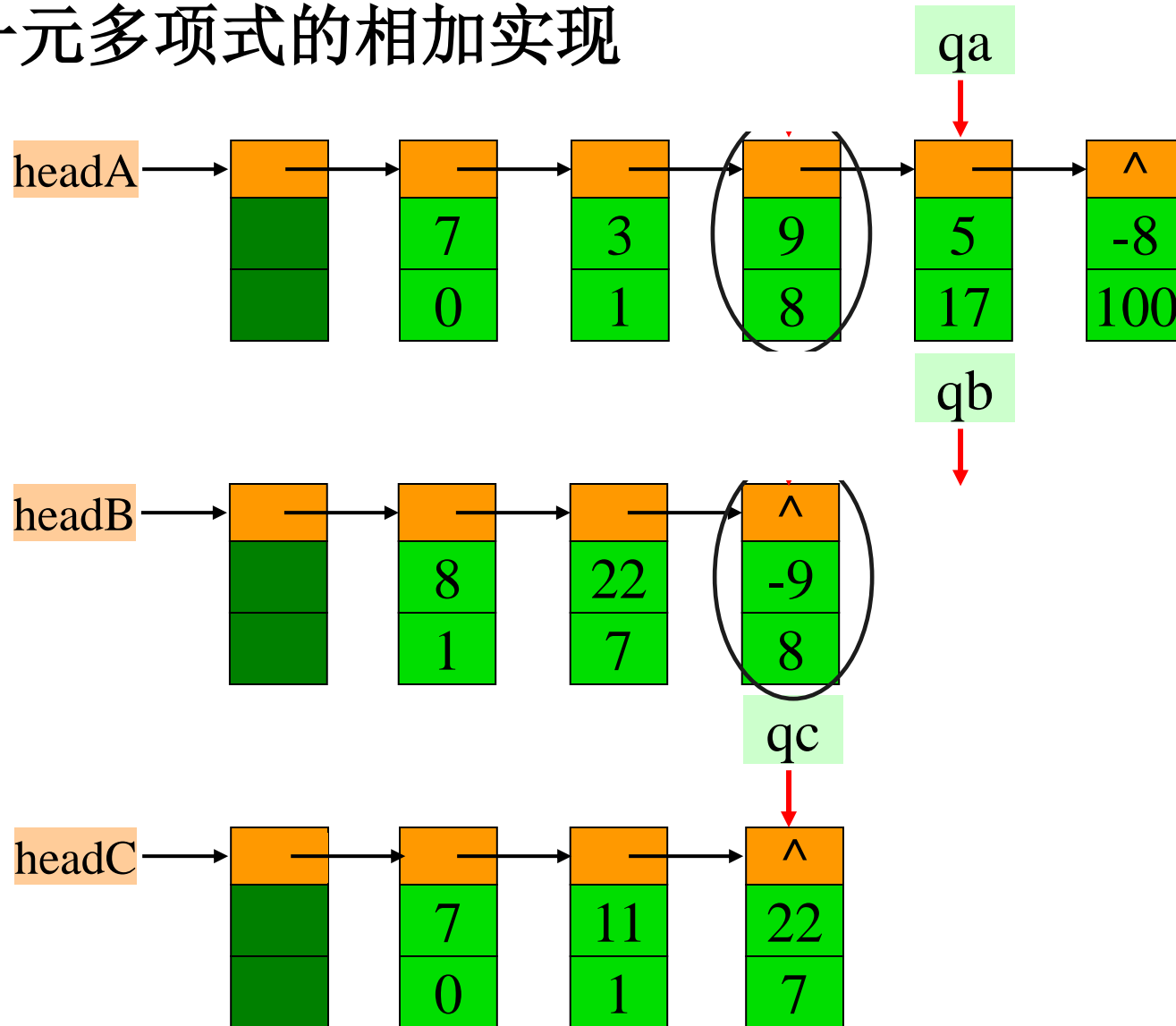
### ■ 一元多项式的相加实现





## 2.4 一元多项式的表示及相加

### ■ 一元多项式的相加实现

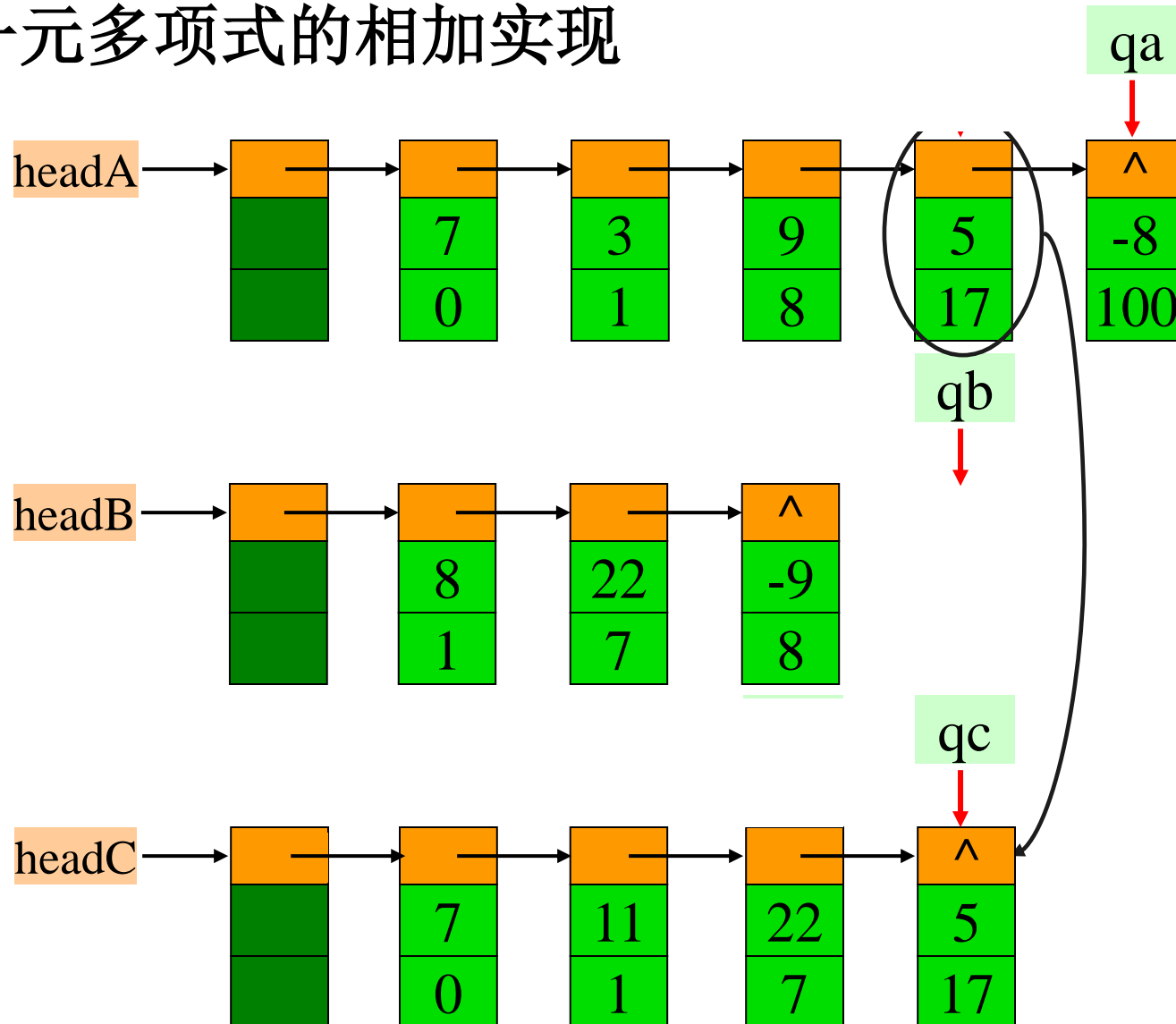






## 2.4 一元多项式的表示及相加

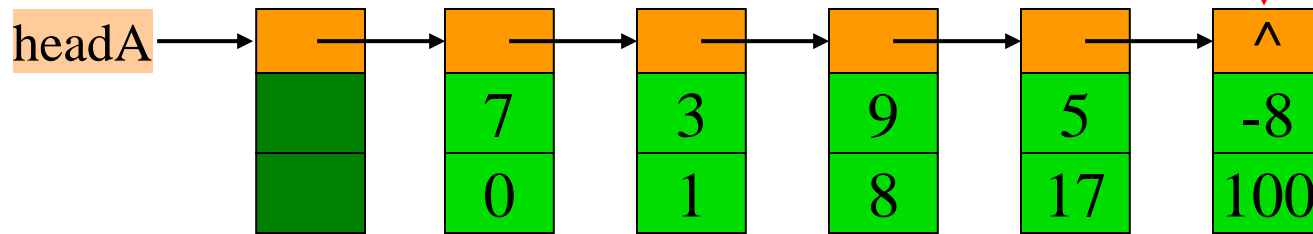
### ■ 一元多项式的相加实现





## 2.4 一元多项式的表示及相加

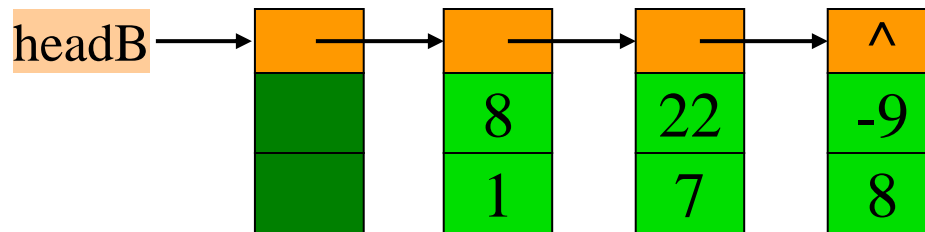
- 一元多项式的相加实现（保留原多项式）



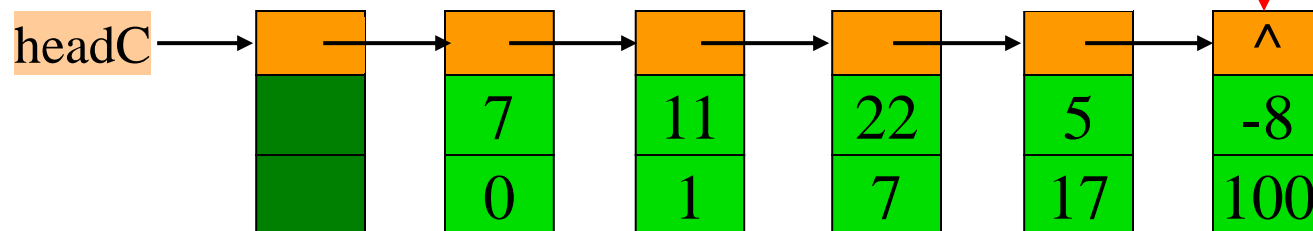
qa



qb



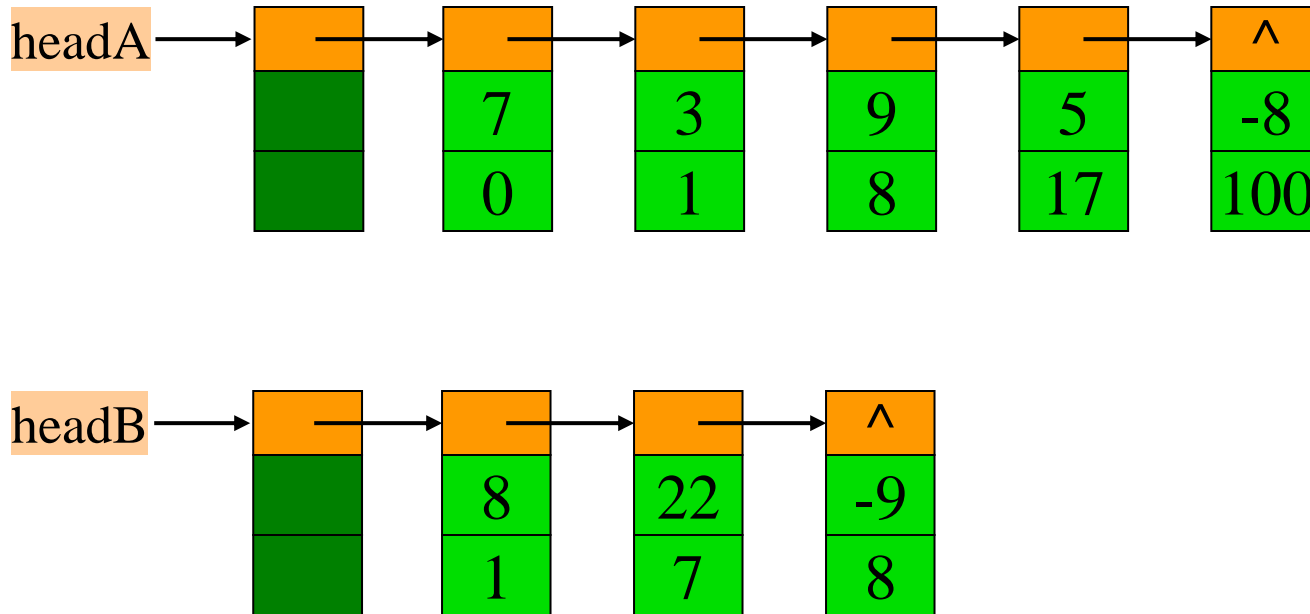
qc





## 2.4 一元多项式的表示及相加

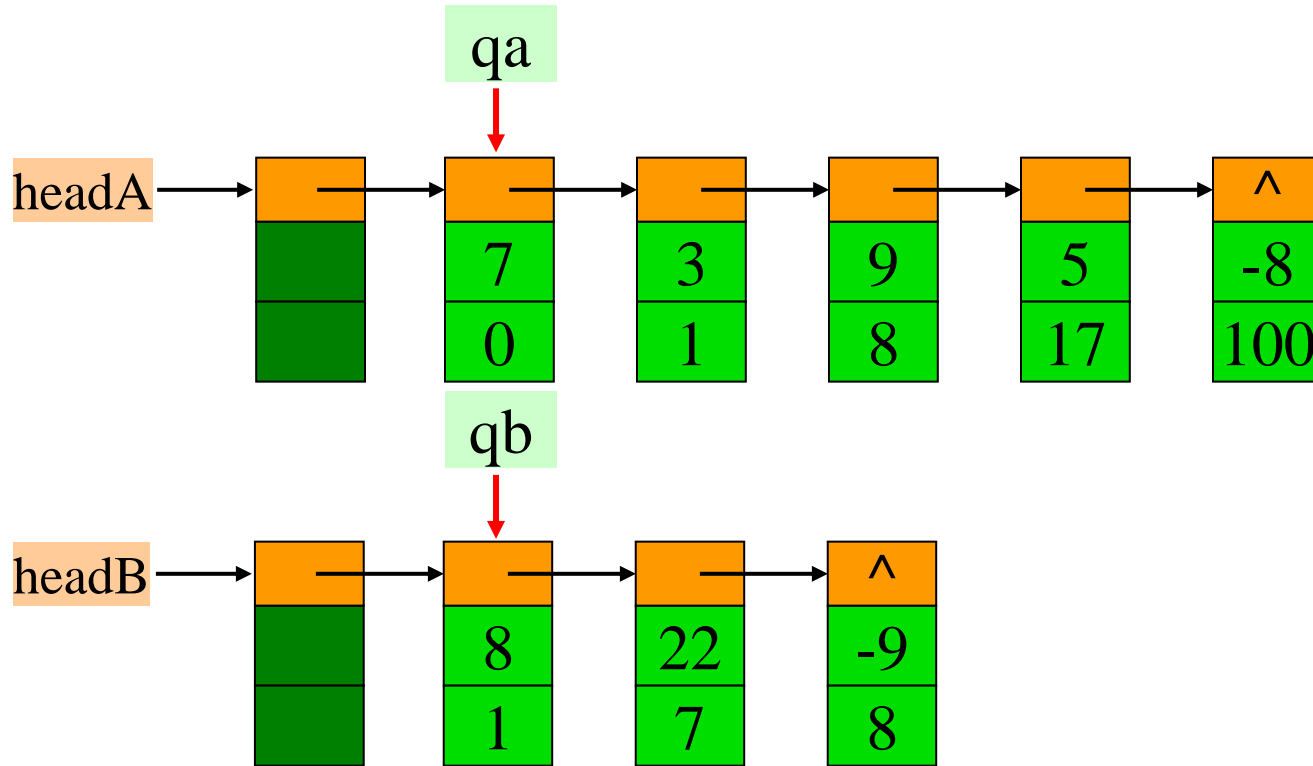
- 一元多项式的相加实现（删除原多项式链表，教材算法2.23）





## 2.4 一元多项式的表示及相加

### ■ 一元多项式的相加实现（删除原多项式链表）



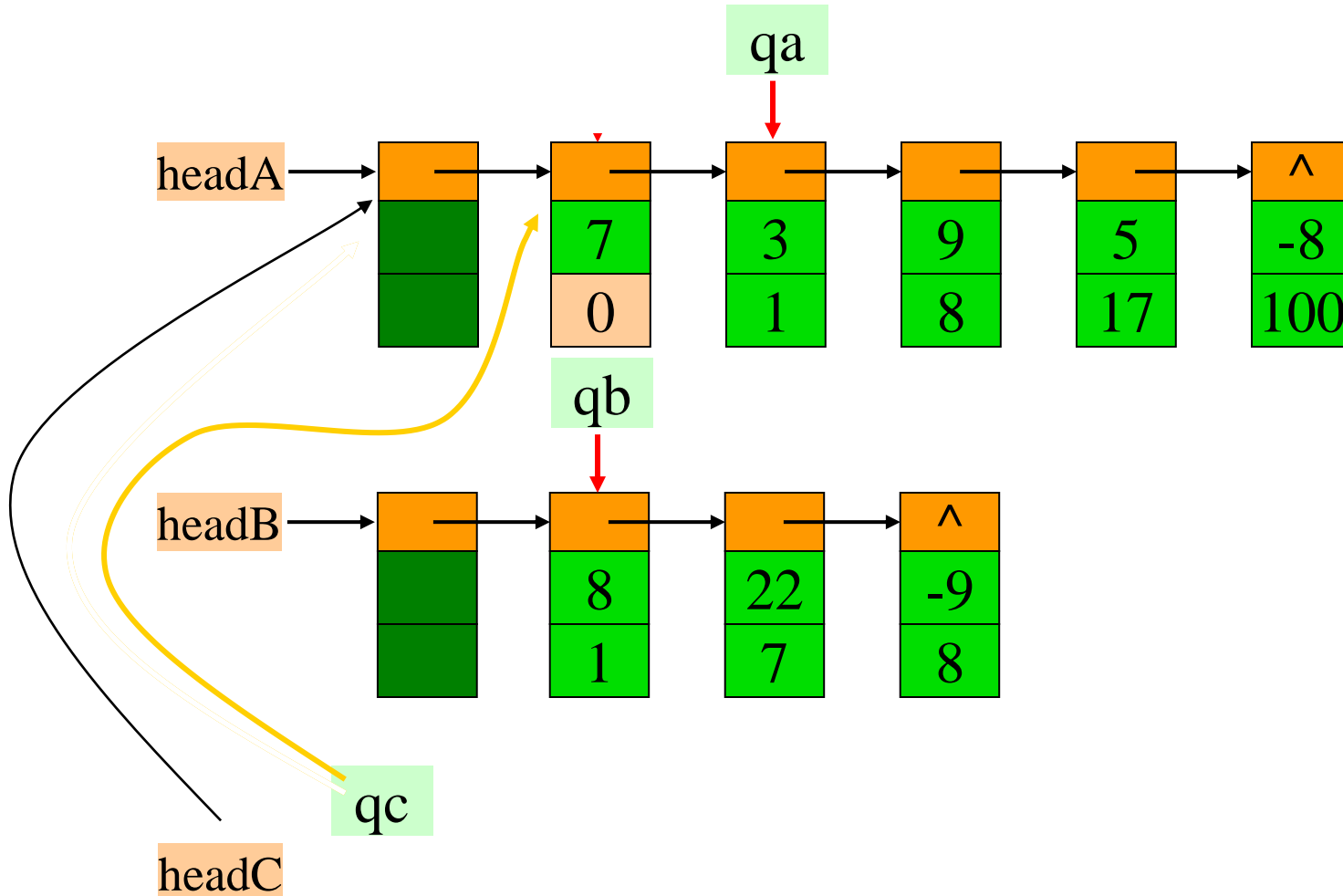
headC: 和多项式链表的头指针

qc: 和多项式链表的尾指针



## 2.4 一元多项式的表示及相加

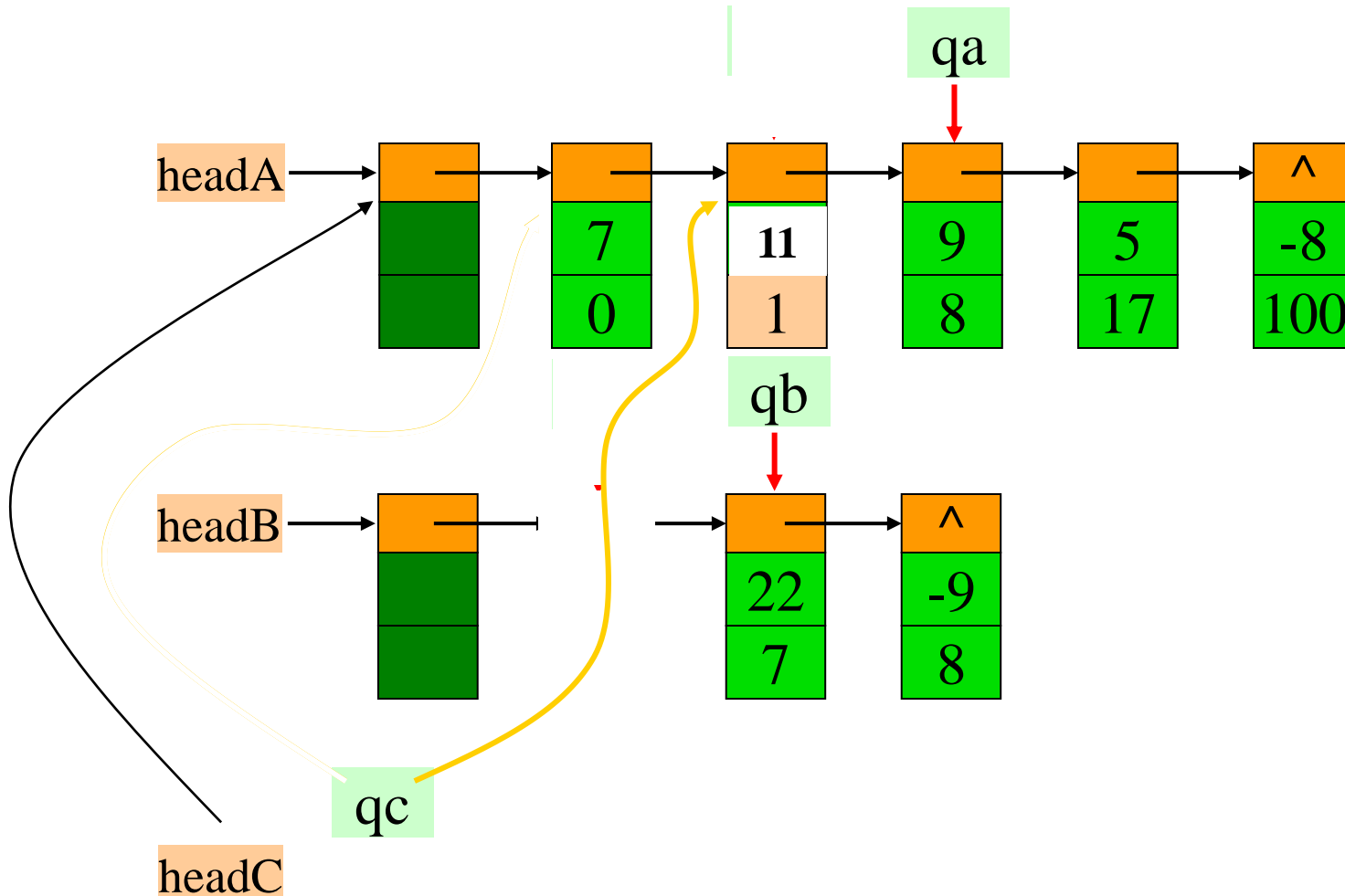
- 一元多项式的相加实现（删除原多项式链表，尽量利用已有结点）





## 2.4 一元多项式的表示及相加

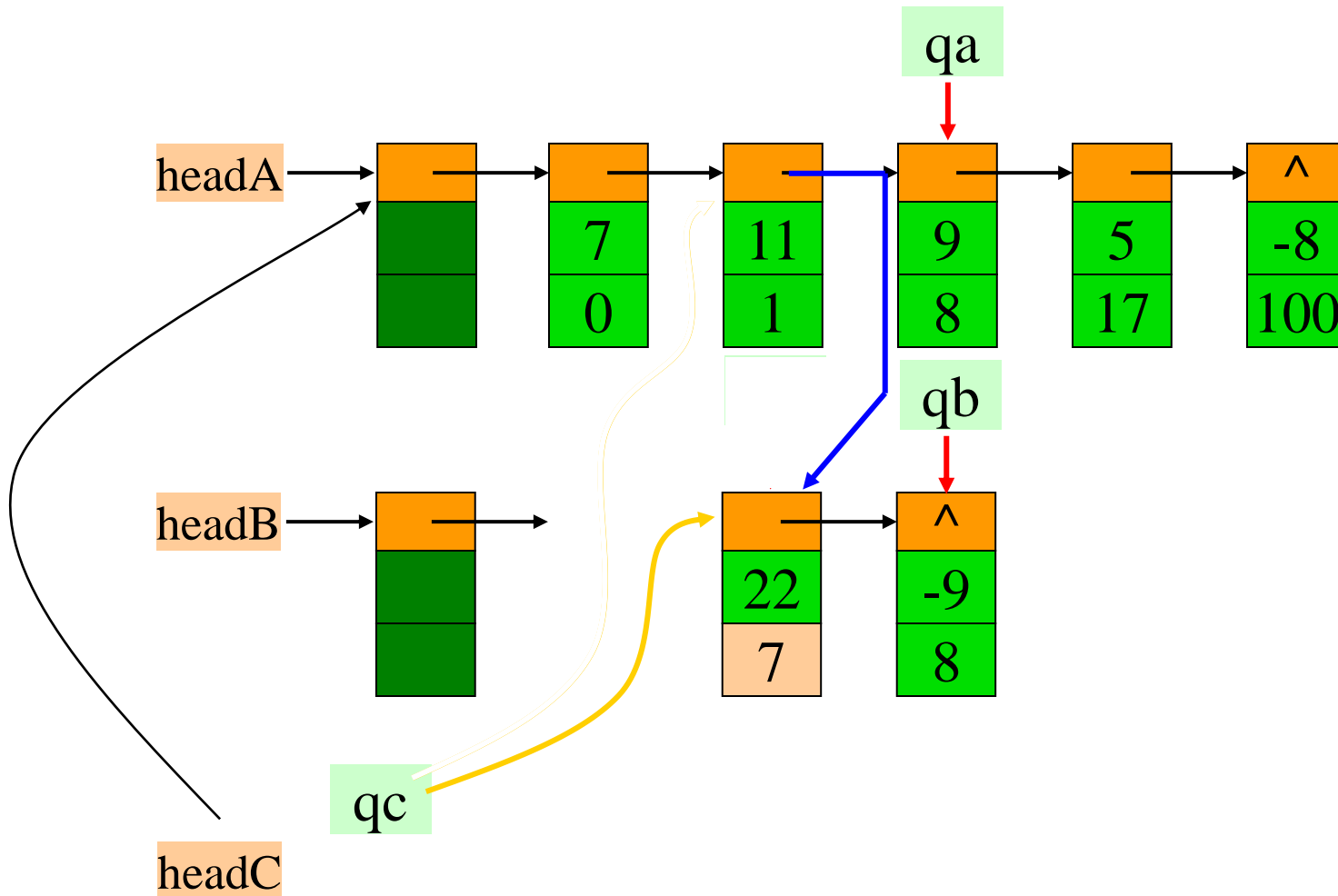
- 一元多项式的相加实现（删除原多项式链表，尽量利用已有结点）





## 2.4 一元多项式的表示及相加

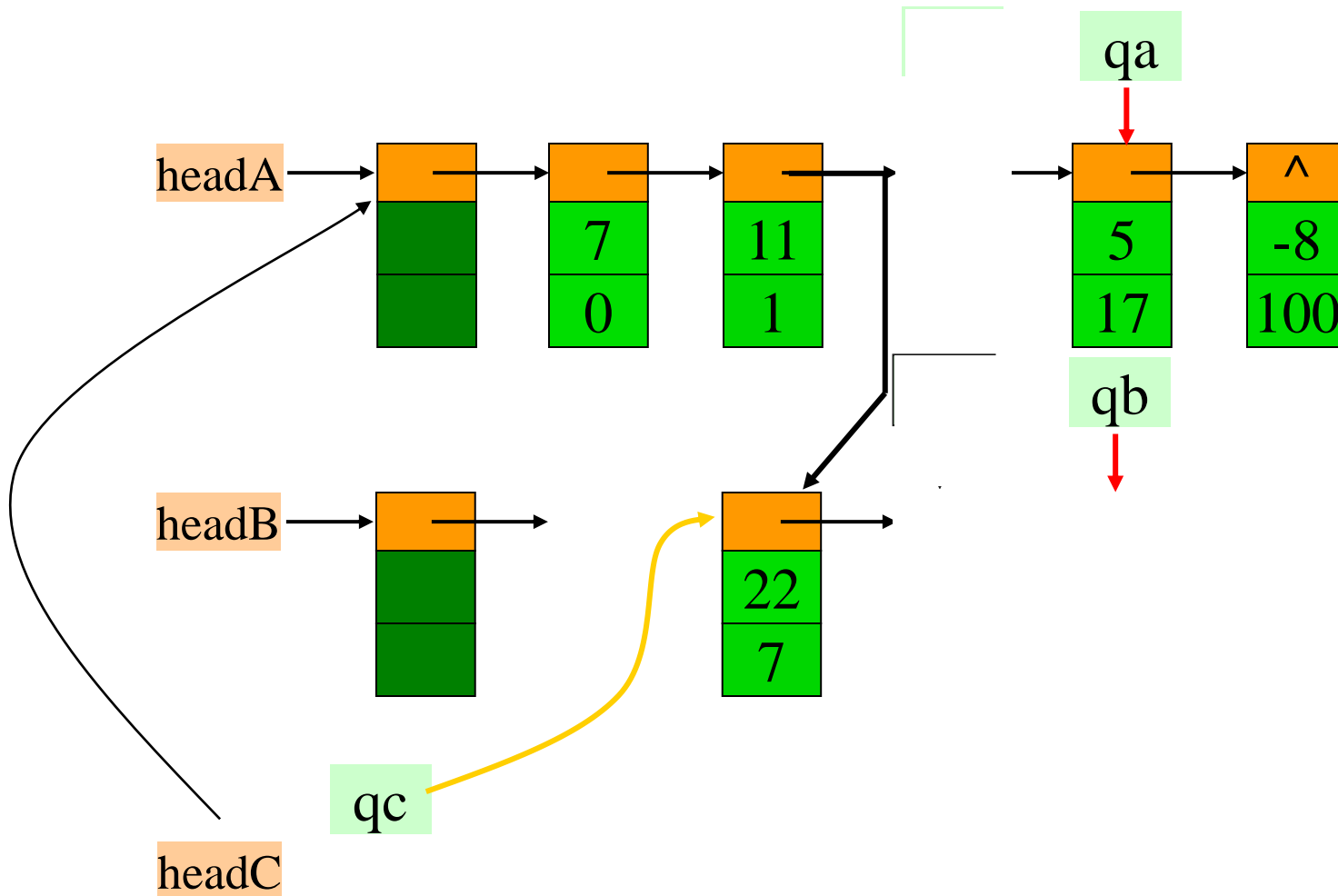
- 一元多项式的相加实现（删除原多项式链表，尽量利用已有结点）





## 2.4 一元多项式的表示及相加

- 一元多项式的相加实现（删除原多项式链表，尽量利用已有结点）

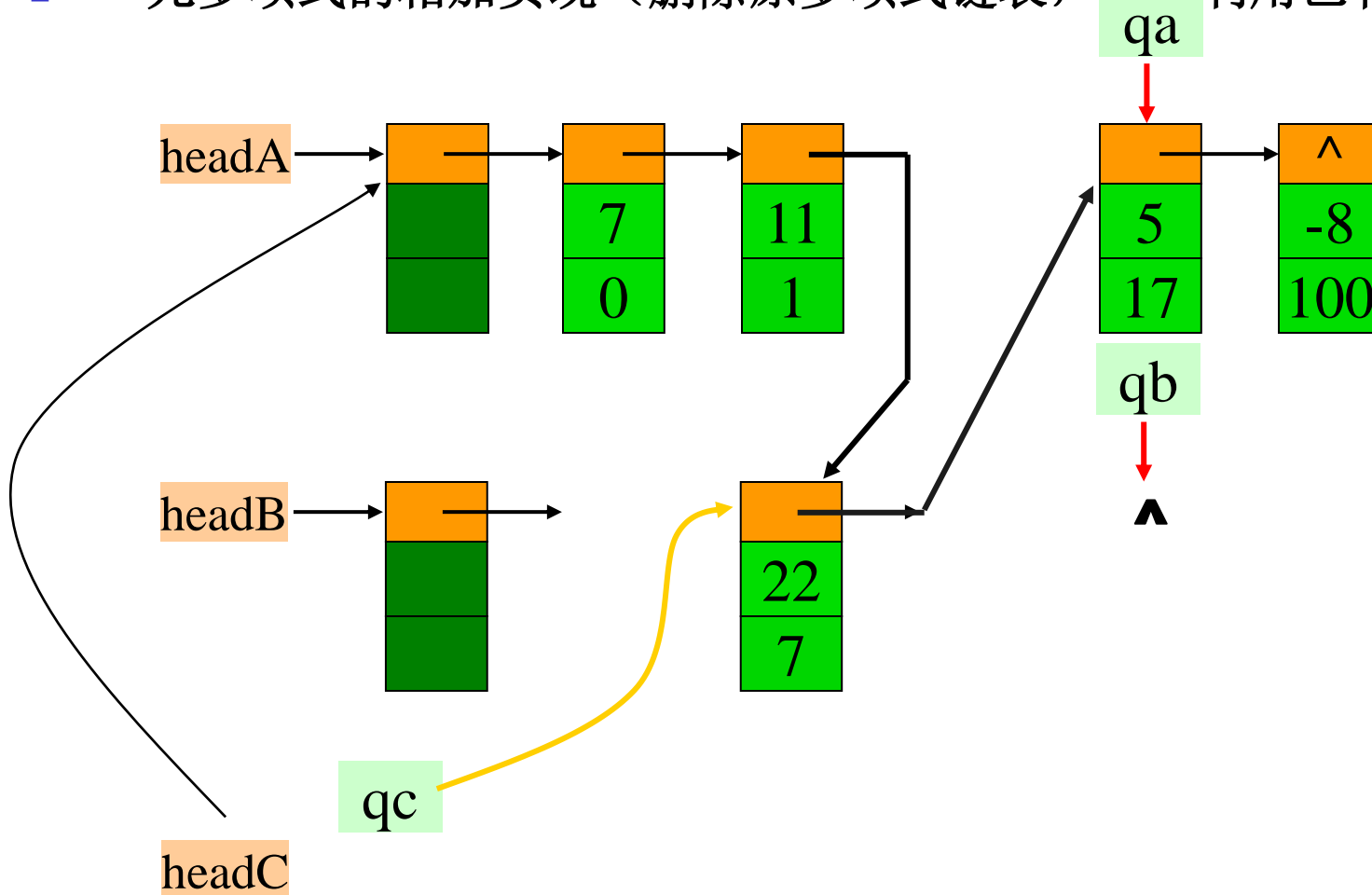






## 2.4 一元多项式的表示及相加

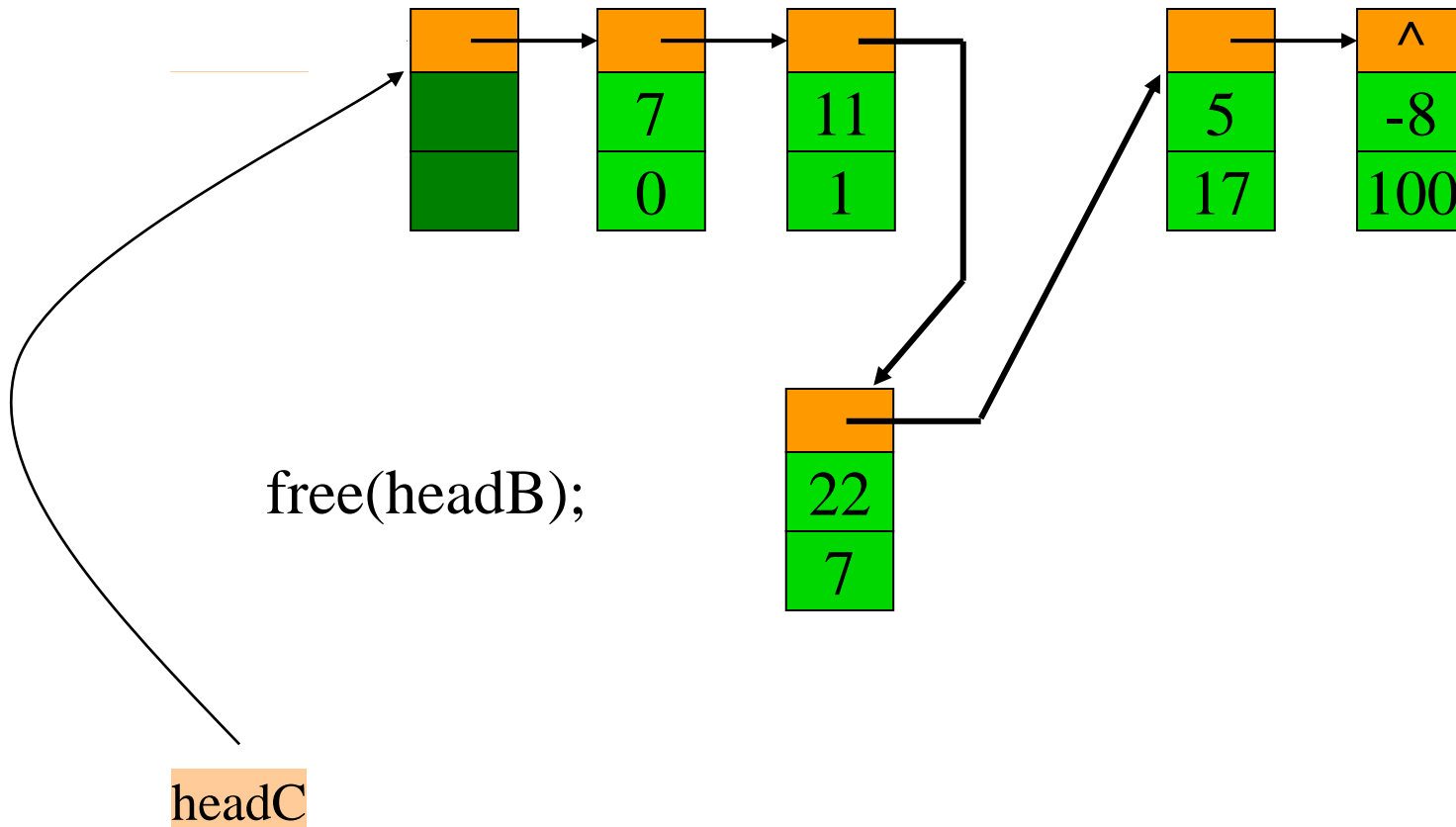
- 一元多项式的相加实现（删除原多项式链表，~~只是~~利用已有结点）





## 2.4 一元多项式的表示及相加

- 一元多项式的相加实现（删除原多项式链表，尽量利用已有结点）





## 2.4 一元多项式的表示及相加

### ■ 一元多项式 $A(x)$ 、 $B(x)$ 的相加过程

基本步骤:

- (1) 设置指针 $qa$ 和 $qb$ 分别指向  $A$ 、 $B$  多项式中的某一项;
- (2) 比较 $qa$ 和 $qb$ 所指向的项的指数:
  - a. 指数相等, 进行处理 (系数相加)
  - b.  $qa$ 指向的项的指数小, 进行处理
  - c.  $qb$ 指向的项的指数小, 进行处理
- (3) 重复(1), 直到某个多项式结束;
- (4) 将未结束多项式的剩余项拼接到和多项式后。



## 一元多项式相加运算（算法2.23）

```
void AddPolyn(polynomialt &Pa, polynomial &Pb)
{
    //多项式Pa和Pb相加Pa=Pa+Pb
    ha = GetHead(Pa); hb = GetHead(Pb);
    qa = NextPos(Pa,ha); qb = NextPos(Pb,hb);
    while (qa && qb) {
        a = GetCurElem(qa); b = GetCurElem(qb);
        switch(*cmp(a,b)) {
            case -1: //多项式PA中当前结点的指数小
                ha = qa; qa = NextPos(Pa,ha); break;
            case 0: //指数相等
                sum = a.coef + b.coef;
                if (sum!=0.0) { SetCurElem(qa,sum); ha = qa; }
                else { DelFirst(ha,qa); FreeNode(qa);
                    DelFirst(hb,qb); FreeNode(qb);
                    qb = NextPos(Pb,hb); qa = NextPos(Pa,ha);
                    break;
                }
            case 1: //多项式PA中当前结点的指数小
```



## 一元多项式相加运算（算法2.23）

```
void AddPolyn(polynomialt &Pa, polynomial &Pb)
```

```
{ //多项式Pa和Pb相加Pa=Pa+Pb
```

```
...
```

```
while (qa && qb) {
```

```
    a = GetCurElem(qa); b = GetCurElem(qb);
```

```
    switch(*cmp(a,b)) {
```

```
        case -1: //多项式PA中当前结点的指数小
```

```
            ha = qa; qa = NextPos(Pa,ha); break;
```

```
        case 0: //指数相等
```

```
            ...
```

```
        case 1: //多项式PB中当前结点的指数小
```

```
            DelFirst(hb,qb); InsFirst(ha,qb);
```

```
            qb = NextPos(Pb,hb); ha = NextPos(Pa,ha);
```

```
        break;
```

```
    } //end of switch
```

```
} //end of while
```

```
if (!ListEmpty(Pb)) Append(Pa,qb);
```

```
FreeNode(hb);
```

```
} //end of AddPolyn
```



# 一元多项式的相加运算

```
void AddPolyn(polynomial &Pa, polynomial &Pb)
{ //多项式Pa和Pb相加Pa=Pa+Pb
  qa = Pa->next; qb = Pb->next;
  Pc = qc = Pa;
  while (qa && qb) {
    if (qa->data.expn < qb->data.expn) {
      .....
    }
    else if (qa->data.expn == qb->data.expn){
      .....
    }
    else {
      .....
    }
  } //while
  .....
} //AddPolyn
```



# 一元多项式的相加运算

```
void AddPolyn(polynomial &Pa, polynomial &Pb)
{    //多项式Pa和Pb相加Pa=Pa+Pb
    qa = Pa->next; qb = Pb->next;
    Pc = qc = Pa;
    while (qa && qb) {
        if (qa->data.expn < qb->data.expn) {
            qc->next = qa; qc = qa; qa = qa->next;
        }
        else if (qa->data.expn == qb->data.expn){
            if (qa->data.coef + qb->data.coef == 0)
                {s = qa; qa = qa->next; free(s);
                 s = qb; qb = qb->next; free(s);}
            else { qa->data.coef = qa->data.coef + qb-
                >data.coef ;

                    qc->next = qa; qc = qa; qa = qa->next;
                    s = qb; qb = qb->next; free(s);}
        }
    }
```



# 一元多项式的相加运算

```
void AddPolyn(polynomial &Pa, polynomial &Pb)
{    //多项式Pa和Pb相加Pa=Pa+Pb
    .....
    while (qa && qb) {
        if (qa->data.expn < qb->data.expn) {
            qc->next = qa; qc = qa; qa = qa->next;
        }
        else if (qa->data.expn == qb->data.expn){    .....
            }
        else {
            qc->next = qb; qc = qb; qb = qb->next;
        }
    } //while
    qc->next = qa? qa : qb;
    free(Pb);
} //AddPolyn
```





## 2.4 一元多项式的表示及相加

### ■ 上机实现提示

- (1)** 编写单链表创建算法(函数);
- (2)** 编写多项式加、减法算法(函数);
- (3)** 编写多项式表达式的输出算法(函数);
- (4)** 编写主函数，两次调用建表函数建立多项式单链表；调用加法函数创建结果链表；调用减法函数创建结果链表；调用输出函数将结果链表转为表达式输出。



# 本章小结

## ■ 本章应掌握的内容

- 什么是线性表？线性结构的含义是什么？
- 线性表采用顺序存储有何特点？查找、插入和删除运算如何实现？
- 线性表采用链式存储有何特点？查找、插入和删除运算如何实现？
  - 链表、结点、指针（链）
  - 单链表、双向链表、循环链表
  - 头指针、头结点

## ☞ 作业

✧ 必做： 2.10 2.11 2.2 2.4 2.5 2.6 2.7 2.22

✧ 选做： 2.9 2.8 2.19 2.24



# ElemType

- 根据需要定义
- **typedef int ElemType; //ElemType是int的别名**

```
typedef struct  
{  
    int num;  
    char name[20];  
    char gender;  
    float score;  
}STUDENT;  
  
typedef STUDENT ElemType;
```

```
typedef struct {  
    double real;  
    double imag;  
}COMPLEX;  
  
typedef COMPLEX ElemType;
```



# 线性表的实现

//-----顺序表：数组空间动态分配-----

```
#define INIT_SIZE 100 //空间初始分配量  
#define INCREMENT 10 //增量  
typedef struct {  
    ElemType *elem; //存储空间首地址  
    int length; //表长(元素个数)  
    int listsize; //存储容量  
}
```

//-----单链表：结点类型定义-----

```
typedef struct LNode{  
    ElemType data; //数据域  
    struct LNode *next; //指针域  
} LNode, *LinkList;
```



# 单链表的C语言实现

## ■ 单链表的C语言实现

**//-----结点类型定义-----**

```
typedef struct LNode{  
    ElemType data;      //数据域  
    struct LNode *next; //指针域  
} LNode, *LinkList;
```



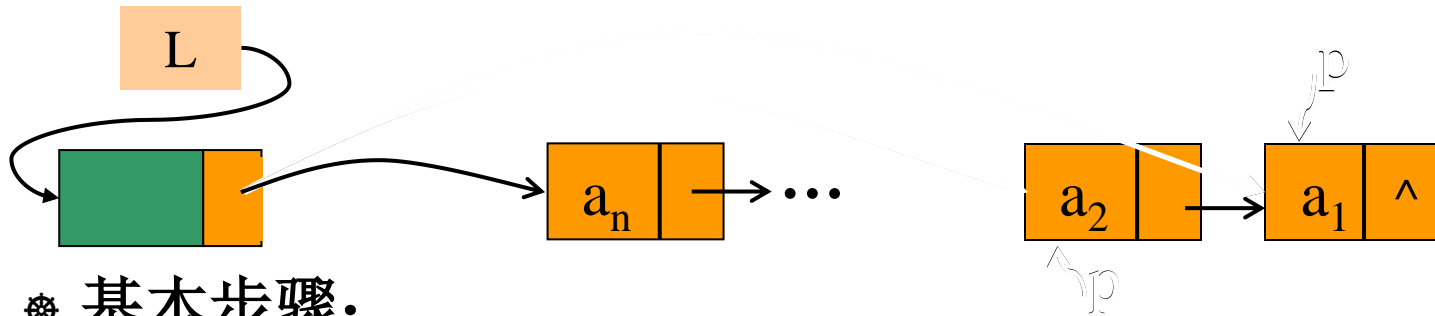
# 如何创建长度为 $n$ 的链表?

- ✧ 表头插入法
- ✧ 表尾插入法



## 2.3.1 单链表的表示和实现

### ■ 用表头插入法建立带头结点的单链表



#### ✱ 基本步骤:

(1) 建立头结点;

$L = (\text{LinkList}) \text{ malloc}(\text{sizeof}(\text{LNode})); L \rightarrow \text{next} = \text{NULL};$

(2) 建立新结点 $p$ , 填入元素;

$p = (\text{LinkList}) \text{ malloc}(\text{sizeof}(\text{LNode})); p \rightarrow \text{data} = a_i$

(3) 将新结点 $p$ 插入到头结点之后, 修改头结点指针域指向 $p$ ;

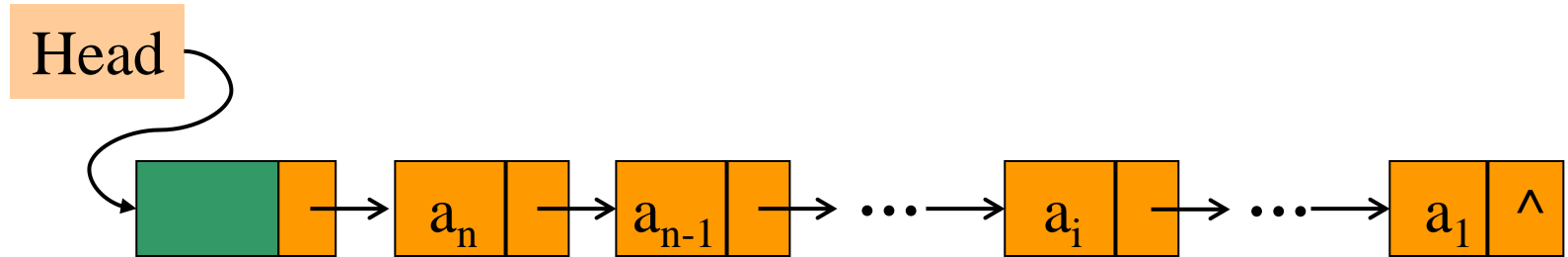
$p \rightarrow \text{next} = L \rightarrow \text{next}; L \rightarrow \text{next} = p;$

(4) 重复(2)(3)步直到插入所有结点, 结束。



## 2.3.1 单链表的表示和实现

- 用类C语言实现表头插入算法:



```
Status CreateList_L(LinkList &L, int n)
```

```
{ //用表头插入法逆序建立带头结点的单链表
```

```
    L = (LinkList) malloc(sizeof(LNode));           //未做溢出判定, 应加入
```

```
    L->next = NULL; //建立头结点, 假定与元素结点结构相同
```

```
    for(i=n; i>0; --i) { //从后向前输入元素
```

```
        p = (LinkList) malloc(sizeof(LNode)); //生成新结点
```

```
        if (!p) return OVERFLOW;
```

```
        scanf("%d",&p->data); //从键盘输入元素值, 存入新结点中
```

```
        p->next = L->next; L->next = p; //新结点插入到表头
```

```
    }
```

```
    return OK;
```

```
}
```

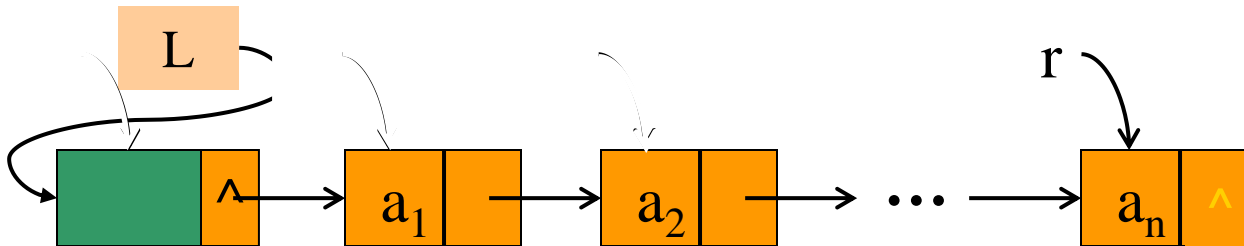
算法复杂度:  $T(n)=O(n)$ ,  $S(n)=O(1)$





## 2.3.1 单链表的表示和实现

### ■ 用表尾插入法建立带头结点的单链表



#### ✱ 基本步骤:

(1) 建立头结点，并附设一个指针 $r$ 一直指向最后一个结点；

$L = (\text{LinkedList}) \text{ malloc}(\text{sizeof}(\text{LNode})); L \rightarrow \text{next} = \text{NULL}; r = L;$

(2) 申请一个新的结点空间，存入元素值；

$p = (\text{LinkedList}) \text{ malloc}(\text{sizeof}(\text{LNode}));$

$p \rightarrow \text{data} = a_i;$

(3) 将新结点接入链表尾部（链接）， $r$ 指向新的表尾结点；

$r \rightarrow \text{next} = p; r = p;$

(4) 重复(2)(3)直到最后一个结点，结束， $r \rightarrow \text{next} = \text{NULL};$



---

**Thank you!**