

# 第二章线性表

- 2. 线性表
- 3. 栈和队列
- 4. 串、数组和广义表

线性结构

(逻辑、存储和运算)

### 线性结构的定义:

若结构是非空有限集, 则有且仅有一个开始结点和一个终端结点, 并且所有结点都最多只有一个直接前趋和一个直接后继。

可表示为:  $(a_1, a_2, \dots, a_n)$

线性结构表达式：  $(a_1, a_2, \dots, a_n)$

## 线性结构的特点：

- ① 只有一个首结点和尾结点；
- ② 除首尾结点外，其他结点只有一个直接前驱和一个直接后继。

简言之，线性结构反映结点间的逻辑关系是一对一的

线性结构包括线性表、堆栈、队列、字符串、数组等等，其中，最典型、最常用的是



线性表

## 2. 线性表



### 教学目标

1. 了解线性结构的特点
2. 掌握顺序表的定义、查找、插入和删除
3. 掌握链表的定义、创建、查找、插入和删除
4. 能够从时间和空间复杂度的角度比较两种存储结构的不同特点及其适用场合

# 教学内容

2.1 线性表的定义和特点

2.2 案例引入

2.3 线性表的类型定义

2.4 线性表的顺序表示和实现

2.5 线性表的链式表示和实现

2.6 顺序表和链表的比较

2.7 线性表的应用

2.8 案例分析与实现



## 2.1 线性表的定义和特点

线性表的定义：用数据元素的有限序列表示

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

数据元素

线性起点

$a_i$  的直接前趋

$a_i$  的直接后继

线性终点

下标，是元素的序号，表示元素在表中的位置

$n=0$ 时称为空表

$n$ 为元素总个数，即表长

## 例1 分析26个英文字母组成的英文表

( **A, B, C, D, .....** , **Z** )

数据元素都是字母； 元素间关系是线性

## 例2 分析学生情况登记表

| 学号        | 姓名  | 性别 | 年龄 | 班级       |
|-----------|-----|----|----|----------|
| 041810205 | 于春梅 | 女  | 18 | 14级计算机1班 |
| 041810260 | 何仕鹏 | 男  | 20 | 14级计算机2班 |
| 041810284 | 王 爽 | 女  | 19 | 14级计算机3班 |
| 041810360 | 王亚武 | 男  | 18 | 14级计算机4班 |
| :         | :   | :  | :  | :        |

数据元素都是记录； 元素间关系是线性

同一线性表中的元素必定具有相同特性



## 2.2 案例引入

### 案例2.1：一元多项式的运算

$$P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$

线性表  $P = (p_0, p_1, p_2, \dots, p_n)$

$$P(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$$

**数组表示**

（每一项的指数*i*隐含在其系数 $p_i$ 的序号中）

| 指数<br>(下标 <i>i</i> ) | 0  | 1 | 2  | 3 | 4 |
|----------------------|----|---|----|---|---|
| 系数 $p[i]$            | 10 | 5 | -4 | 3 | 2 |



$$R_n(x) = P_n(x) + Q_m(x)$$



线性表  $R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \dots, p_m + q_m, p_{m+1}, \dots, p_n)$

稀疏多项式

$$S(x) = 1 + 3x^{10000} + 2x^{20000}$$



## 案例2.2：稀疏多项式的运算

多项式**非零项**的数组表示

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

| 下标i        | 0 | 1 | 2 | 3  |
|------------|---|---|---|----|
| 系数<br>a[i] | 7 | 3 | 9 | 5  |
| 指数         | 0 | 1 | 8 | 17 |

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

| 下标i        | 0 | 1  | 2  |
|------------|---|----|----|
| 系数<br>b[i] | 8 | 22 | -9 |
| 指数         | 1 | 7  | 8  |

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + \dots + p_m x^{e_m}$$



线性表  $P = ((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$

- 创建一个**新数组c**
- 分别从头遍历比较a和b的每一项
  - ✓ **指数相同**，对应系数相加，若其和不为零，则在c中增加一个新项
  - ✓ **指数不相同**，则将指数较小的项复制到c中
- 一个多项式已遍历**完毕**时，将另一个剩余项依次复制到c中即可

●顺序存储结构存在问题

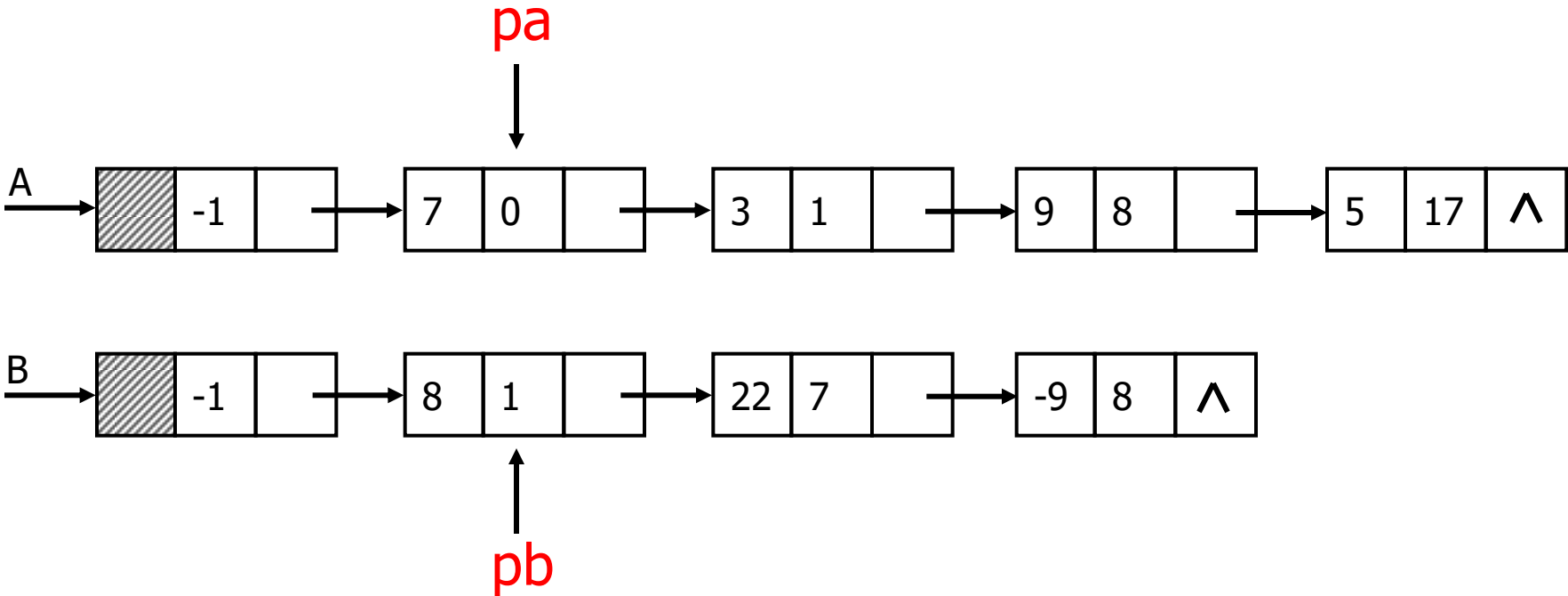
- ✓存储空间分配不灵活
- ✓运算的空间复杂度高



链式存储结构

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

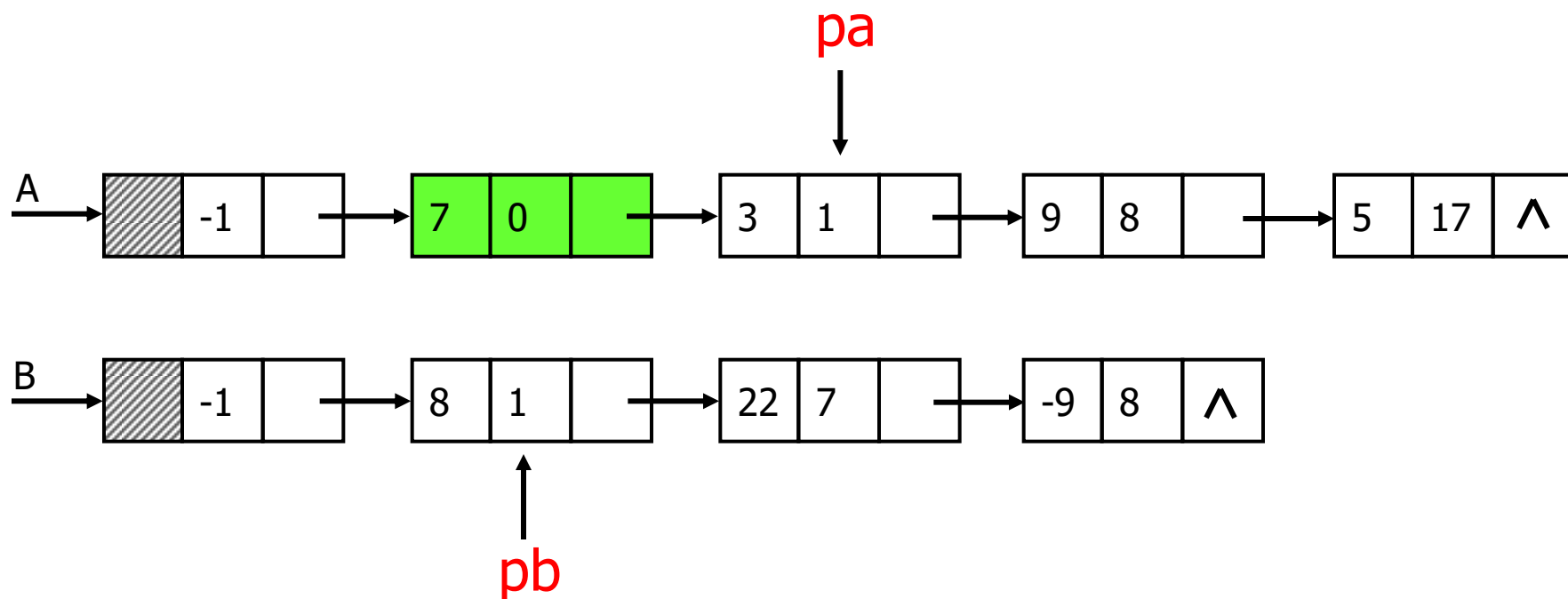
$$B_8(x)=8x+22x^7-9x^8$$



# 多项式相加

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

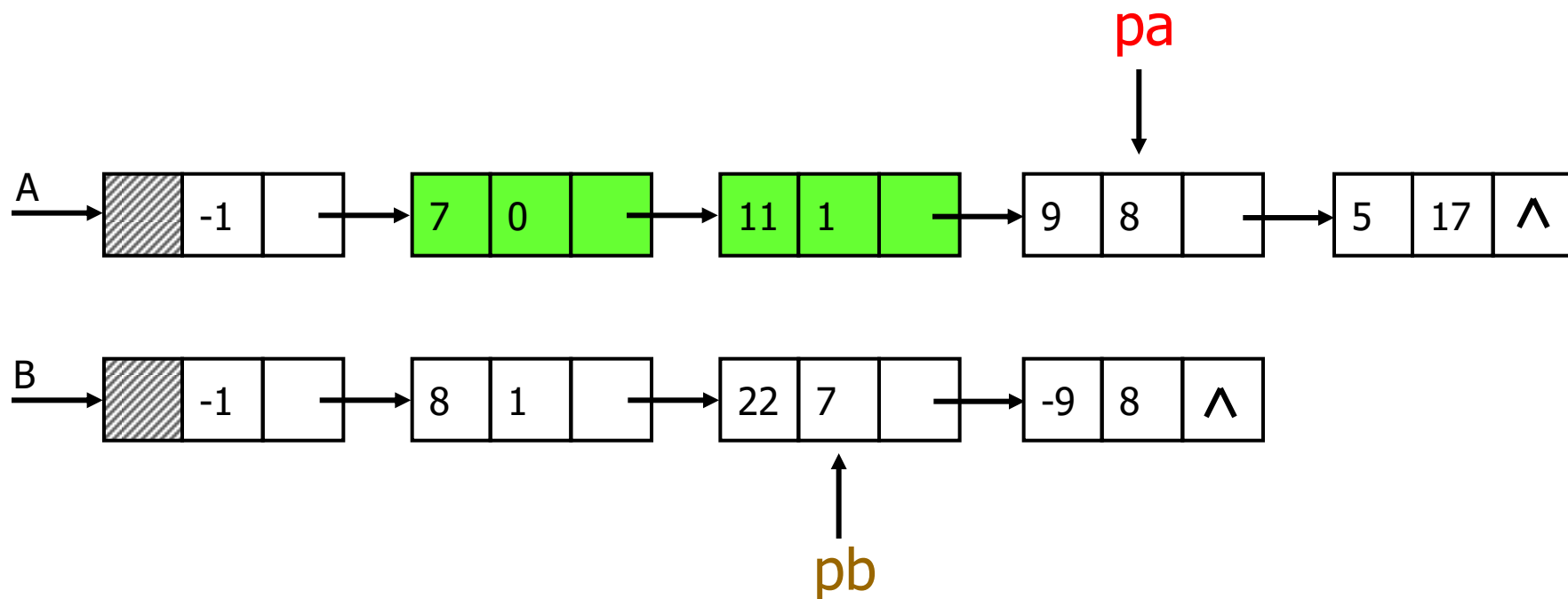
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

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

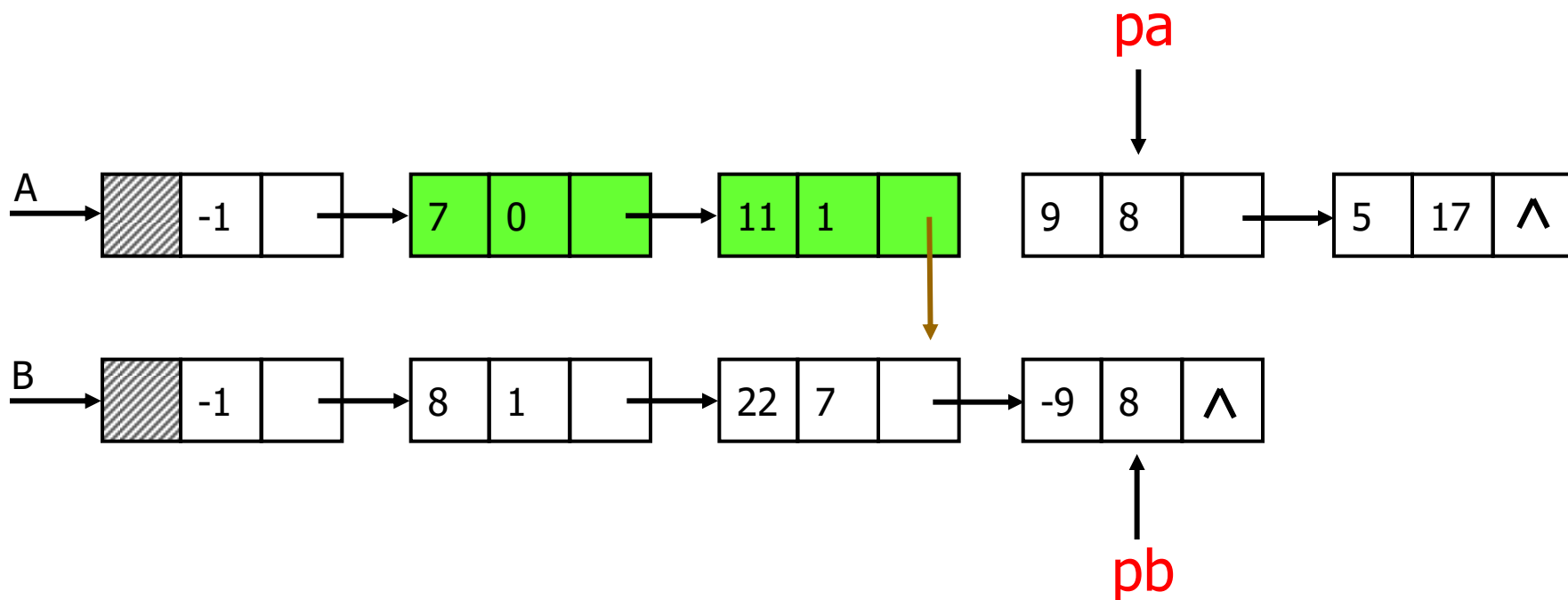
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

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

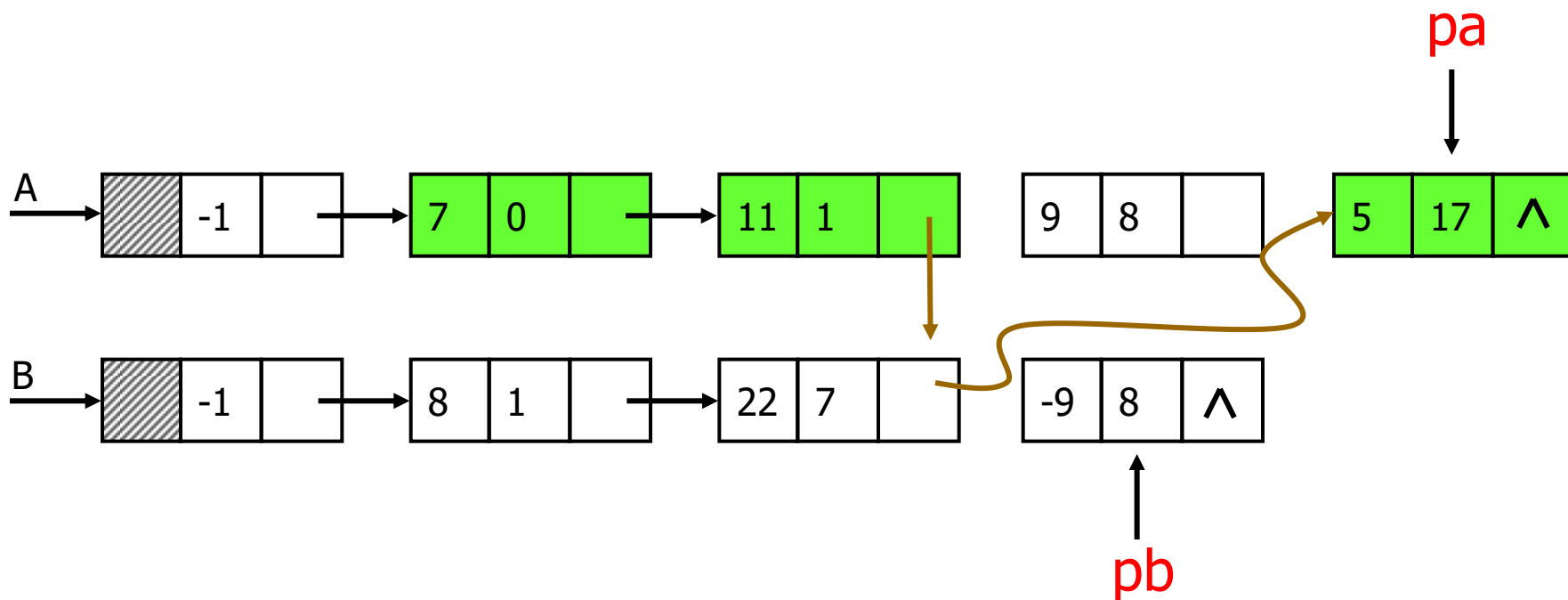
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

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

$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 案例2.3：图书信息管理系统

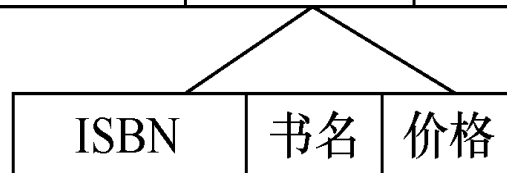
- (1) 查找
- (2) 插入
- (3) 删除
- (4) 修改
- (5) 排序
- (6) 计数

| book.txt - 记事本                |               |    |
|-------------------------------|---------------|----|
| 文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H) |               |    |
| ISBN                          | 书名            | 定价 |
| 9787302257646                 | 程序设计基础        | 25 |
| 9787302219972                 | 单片机技术及应用      | 32 |
| 9787302203513                 | 编译原理          | 46 |
| 9787811234923                 | 汇编语言程序设计教程    | 21 |
| 9787512100831                 | 计算机操作系统       | 17 |
| 9787302265436                 | 计算机导论实验指导     | 18 |
| 9787302180630                 | 实用数据结构        | 29 |
| 9787302225065                 | 数据结构（C语言版）    | 38 |
| 9787302171676                 | C#面向对象程序设计    | 39 |
| 9787302250692                 | C语言程序设计       | 42 |
| 9787302150664                 | 数据库原理         | 35 |
| 9787302260806                 | Java编程与实践     | 56 |
| 9787302252887                 | Java程序设计与应用教程 | 39 |
| 9787302198505                 | 嵌入式操作系统及编程    | 25 |
| 9787302169666                 | 软件测试          | 24 |
| 9787811231557                 | Eclipse基础与应用  | 35 |

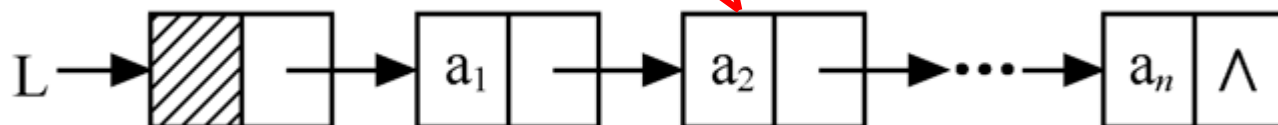


## 图书顺序表

|         |         |         |     |                     |     |  |
|---------|---------|---------|-----|---------------------|-----|--|
| elem[0] | elem[1] | elem[2] | ... | elem[length-1]      | 空闲区 |  |
| $a_1$   | $a_2$   | $a_3$   | ... | $a_{\text{length}}$ |     |  |



## 图书链表





## 总结

- 线性表中数据元素的类型可以为简单类型，也可以为复杂类型。
- 许多实际应用问题所涉的基本操作有很大相似性，不应为每个具体应用单独编写一个程序。
- 从具体应用中**抽象出共性的逻辑结构和基本操作**（**抽象数据类型**），然后实现其存储结构和基本操作。



## 2.3 线性表的类型定义

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:  $R = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,3,\dots,n \}$

基本操作:

InitList( &L )

操作结果: 构造一个空的线性表L;

GetElem( L, i, &e )

初始条件: 线性表L已存在,  $1 \leq i \leq \text{ListLength}(L)$ ;

操作结果: 用e返回L中第i个数据元素的值;

ListInsert( L, i, &e )

初始条件: 线性表L已存在,  $1 \leq i \leq \text{ListLength}(L)$  ;

操作结果: 在线性表L中的第i个位置插入元素e;

...

} ADT List

## 2.4 线性表的顺序表示和实现



线性表的顺序表示又称为**顺序存储结构**或**顺序映像**。

**顺序存储定义：**把逻辑上相邻的数据元素存储在物理上相邻的存储单元中的存储结构。

简言之，逻辑上相邻，物理上也相邻

**顺序存储方法：**用**一组地址连续**的存储单元依次存储线性表的元素，可通过**数组 $V[n]$** 来实现。

## 顺序存储

| 存储地址          | 存储内容  |
|---------------|-------|
| $L_0$         | 元素1   |
| $L_0+m$       | 元素2   |
|               | ..... |
| $L_0+(i-1)*m$ | 元素i   |
|               | ..... |
| $L_0+(n-1)*m$ | 元素n   |

$$\text{Loc(元素i)} = L_0 + (i-1)*m$$

# 顺序表的类型定义

---

```
#define MAXSIZE 100      //最大长度
typedef struct {
    ElemType *elem;      //指向数据元素的基地址
    int length;          //线性表的当前长度
} SqList;
```

---

# 图书表的顺序存储结构类型定义

```
#define MAXSIZE 10000    //图书表可能达到的最大长度
typedef struct           //图书信息定义
{
    char no[20];          //图书ISBN
    char name[50];        //图书名字
    float price;          //图书价格
}Book;
typedef struct
{
    Book *elem;           //存储空间的基地址
    int length;           //图书表中当前图书个数
}SqList;                 //图书表的顺序存储结构类型为SqList
```

## 补充：C语言的动态分配函数（<stdlib.h>）

`malloc(m)`：开辟`m`字节长度的地址空间，  
并返回这段空间的首地址

`sizeof(x)`：计算变量`x`的长度

`free(p)`：释放指针`p`所指变量的存储空间  
，即彻底删除一个变量

---



# 补充：C++的动态存储分配

**new** 类型名T（初值列表）

功能：

```
int *p1 = new int;  
或 int *p1 = new int(10);
```

申请用于存放T类型对象的内存空间，并依初值列表赋以初值

结果值：

成功：T类型的指针，指向新分配的内存

失败：0（NULL）

**delete** 指针P

```
delete p1;
```

功能：

释放指针P所指向的内存。P必须是new操作的返回值

# 补充：C++中的参数传递

- 函数调用时传送给形参表的实参必须与形参在类型、个数、顺序上保持一致
- 参数传递有两种方式
  - 传值方式（参数为整型、实型、字符型等）
  - 传地址
    - 参数为指针变量
    - 参数为引用类型
    - 参数为数组名

# 传值方式

把实参的值传送给函数局部工作区相应的副本中，函数使用这个副本执行必要的功能。  
函数修改的是副本的值，实参的值不变

```
#include <iostream.h>

void swap(float m,float n)
{float temp;

temp=m;

m=n;

n=temp;

}
```

```
void main()
{float a,b;

cin>>a>>b;

swap(a,b);

cout<<a<<endl<<b<<endl;

}
```

# 传地址方式——指针变量作参数

## 形参变化影响实参

```
#include <iostream.h>

void swap(float *m,float *n)
{float t;

 t=*m;

 *m=*n;

 *n=t;

}
```

```
void main()
{float a,b,*p1,*p2;

 cin>>a>>b;

 p1=&a; p2=&b;
 swap(p1, p2);

 cout<<a<<endl<<b<<endl;

}
```

# 传地址方式——指针变量作参数

形参变化不影响实参？？

```
#include <iostream.h>

void swap(float *m,float *n)
{float *t;

 t=m;

 m=n;

 n=t;

}
```

```
void main()
{float a,b,*p1,*p2;

 cin>>a>>b;

 p1=&a; p2=&b;
 swap(p1, p2);

 cout<<a<<endl<<b<<endl;

}
```

Back

# 传地址方式——引用类型作参数

什么是引用???

引用：它用来给一个对象提供一个替代的名字。

```
#include<iostream.h>
void main(){
    int i=5;
    int &j=i;
    i=7;
    cout<<"i="<<i<<" j="<<j;
}
```

- ✓ **j** 是一个引用类型，代表**i**的一个替代名
- ✓ **i** 值改变时，**j** 值也跟着改变，所以会输出  
**i=7 j=7**

# 传地址方式——引用类型作参数

```
#include <iostream.h>

void swap(float& m,float& n)
{float temp;
 temp=m;
 m=n;
 n=temp;
}
```

```
void main()
{float a,b;
 cin>>a>>b;
 swap(a,b);
 cout<<a<<endl<<b<<endl;
}
```

# 引用类型作形参的三点说明

- (1) 传递引用给函数与传递指针的效果是一样的，**形参变化实参也发生变化**。
- (2) 引用类型作形参，在内存中并没有产生实参的副本，它**直接对实参操作**；而一般变量作参数，形参与实参就占用不同的存储单元，所以形参变量的值是实参变量的副本。因此，当**参数传递的数据量较大**时，用引用比用一般变量传递参数的时间和空间效率都好。
- (3) 指针参数虽然也能达到与使用引用的效果，但在被调函数中需要重复使用“\*指针变量名”的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。



# 传地址方式——数组名作参数

传递的是数组的首地址

对形参数组所做的任何改变都将反映到实参数组中

```
#include<iostream.h>
```

```
void sub(char b[]);
```

```
void main(void )
```

```
{  char a[10]="hollo";
```

```
    sub(a);
```

```
    cout<<a<<endl;
```

```
}
```

```
void sub(char b[])
```

```
{  b[1]='e';}
```

## 用数组作函数的参数，求10个整数的最大数

```
#include <iostream.h>
```

```
#define N 10
```

```
int max(int a[]);
```

```
void main ( ) {
```

```
    int a[10];
```

```
    int i,m;
```

```
    for(i=0;i<N;i++)
```

```
        cin>>a[i];
```

```
    m=max(a);
```

```
    cout<<"the max number is:"<<m;
```

```
}
```

```
int max(int b[]){
```

```
    int i,n;
```

```
    n=b[0];
```

```
    for(i=1;i<N;i++)
```

```
        if(n<b[i]) n=b[i];
```

```
    return n;
```

```
}
```

**练习：**用数组作为函数的参数，将数组中n个整数按相反的顺序存放，要求输入和输出在主函数中完成

```
#include <iostream.h>

#define N 10

void sub(int b[ ]){
    int i,j,temp,m;
    m=N/2;
    for(i=0;i<m;i++){
        j=N-1-i;
        temp=b[i];
        b[i]= b[j];
        b[j]=temp; }
    return ;}
```

```
void main ( ) {
    int a[10],i;
    for(i=0;i<N;i++)
        cin>>a[i];
    sub(a);
    for(i=0;i<N;i++)
        cout<<a[i];
}
```

# 线性表的重要基本操作

---

1. 初始化
2. 取值
3. 查找
4. 插入
5. 删除

# 重要基本操作的算法实现

## 1. 初始化线性表L （参数用引用）

```
Status InitList_Sq(SqList &L){  //构造一个空的顺序表L  
    L.elem=new ElemType[MAXSIZE]; //为顺序表分配空间  
    if(!L.elem) exit(OVERFLOW);    //存储分配失败  
    L.length=0;                      //空表长度为0  
    return OK;  
}
```

## 1. 初始化线性表L （参数用指针）

```
Status InitList_Sq(SqList *L){  //构造一个空的顺序表L  
    L-> elem=new ElemType[MAXSIZE];  //为顺序表分配空间  
    if(! L-> elem) exit(OVERFLOW);    //存储分配失败  
    L-> length=0;                      //空表长度为0  
    return OK;  
}
```

# 补充：几个简单基本操作的算法实现

## 销毁线性表L

```
void DestroyList(SqList &L)
{
    if (L.elem) delete[] L.elem;    //释放存储空间
}
```

## 清空线性表L

```
void ClearList(SqList &L)
{
    L.length=0;    //将线性表的长度置为0
}
```

# 补充：几个简单基本操作的算法实现

## 求线性表L的长度

```
int GetLength(SqList L)
{
    return (L.length);
}
```

## 判断线性表L是否为空

```
int IsEmpty(SqList L)
{
    if (L.length==0) return 1;
    else return 0;
}
```



# 线性表的重要基本操作

---

1. 初始化

2. 取值

3. 查找

4. 插入

5. 删除

---

## 2. 取值 (根据位置i获取相应位置数据元素的内容)

获取线性表L中的某个数据元素的内容

```
int GetElem(SqList L, int i, ElemType &e)
```

```
{
```

```
    if (i<1 || i>L.length) return ERROR;
```

```
    //判断i值是否合理，若不合理，返回ERROR
```

```
    e=L.elem[i-1];    //第i-1的单元存储着第i个数据
```

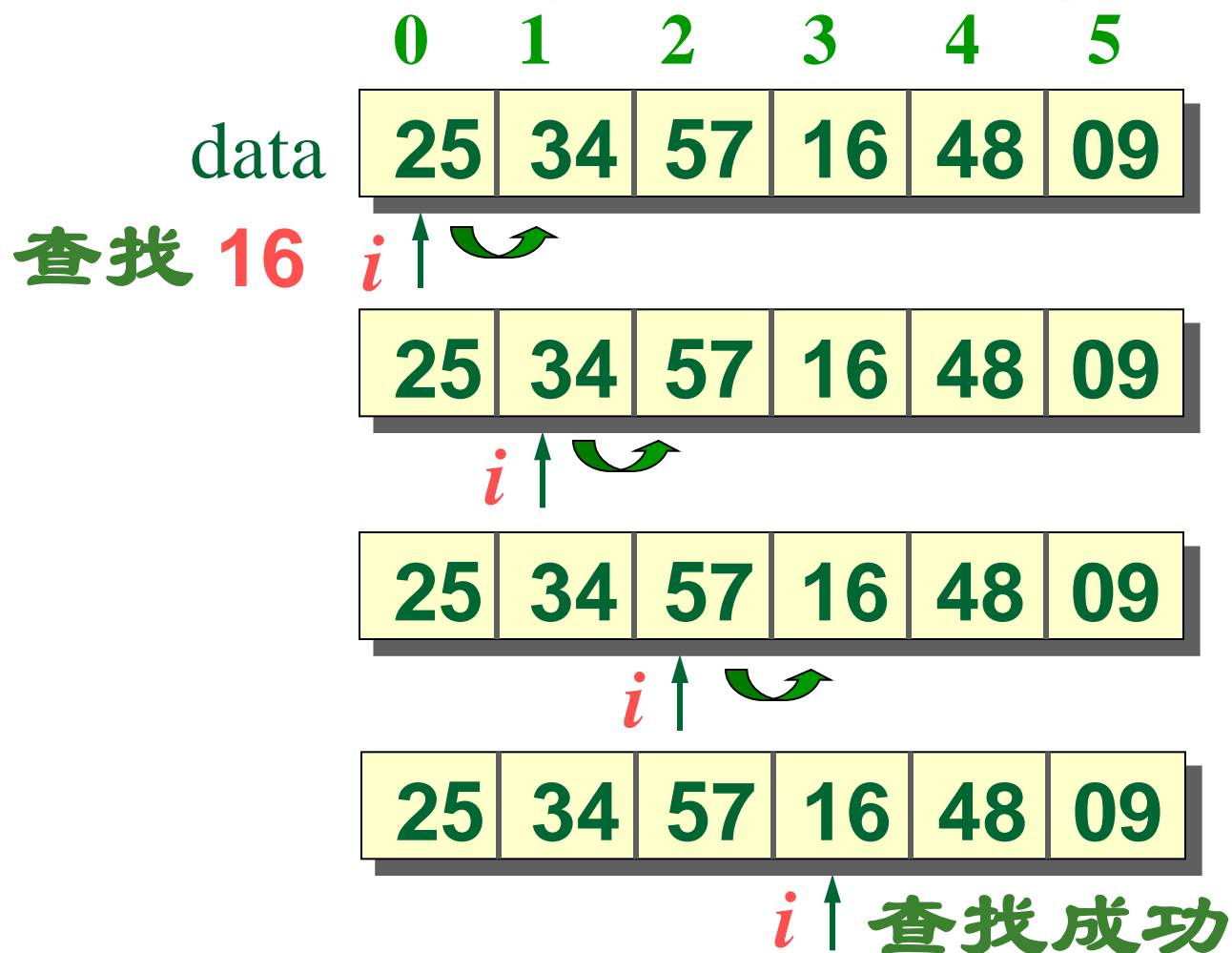
```
    return OK;
```

```
}
```

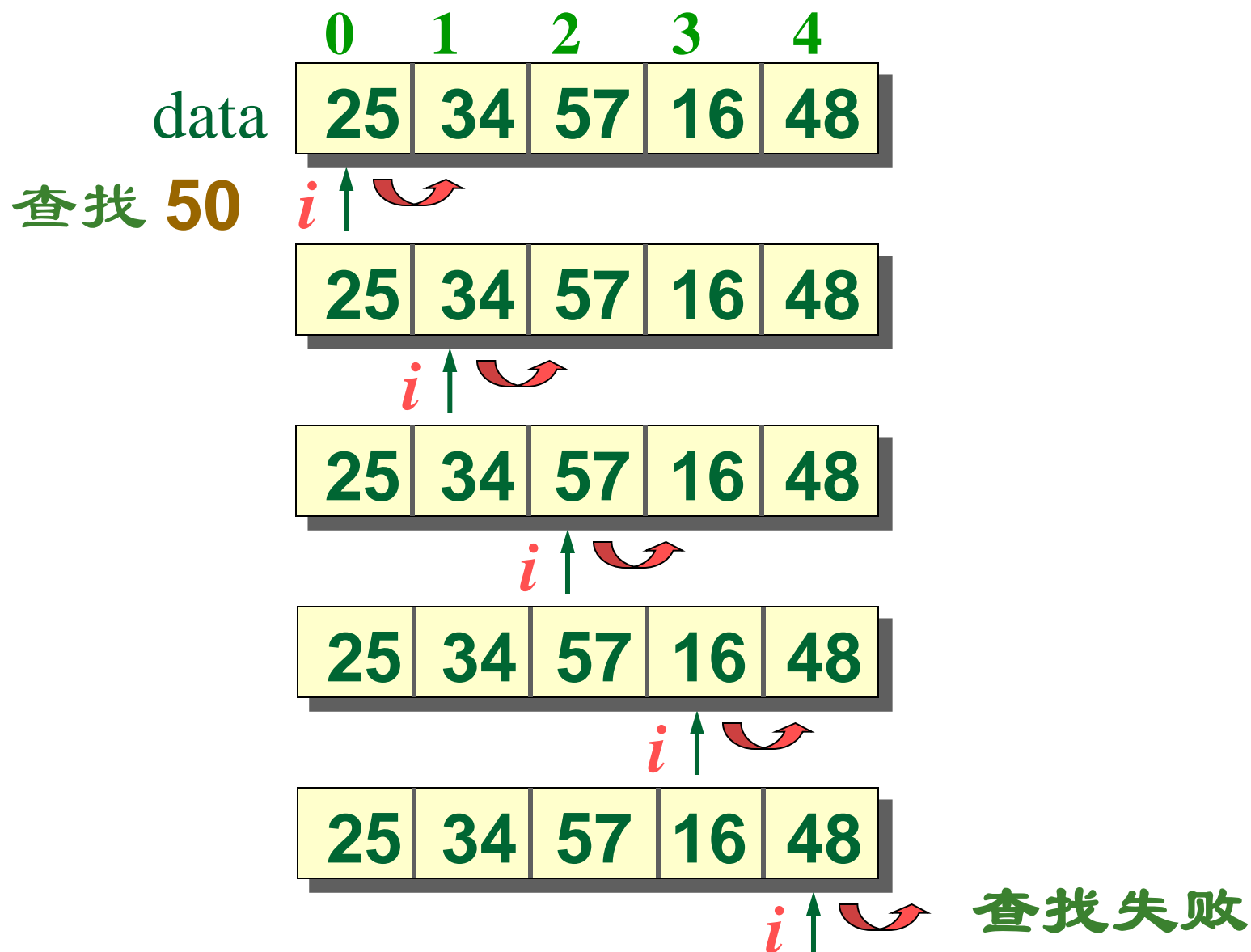
随机存取

### 3. 查找 (根据指定数据获取数据所在的位置)

#### 顺序查找图示



### 3. 查找 (根据指定数据获取数据所在的位置)



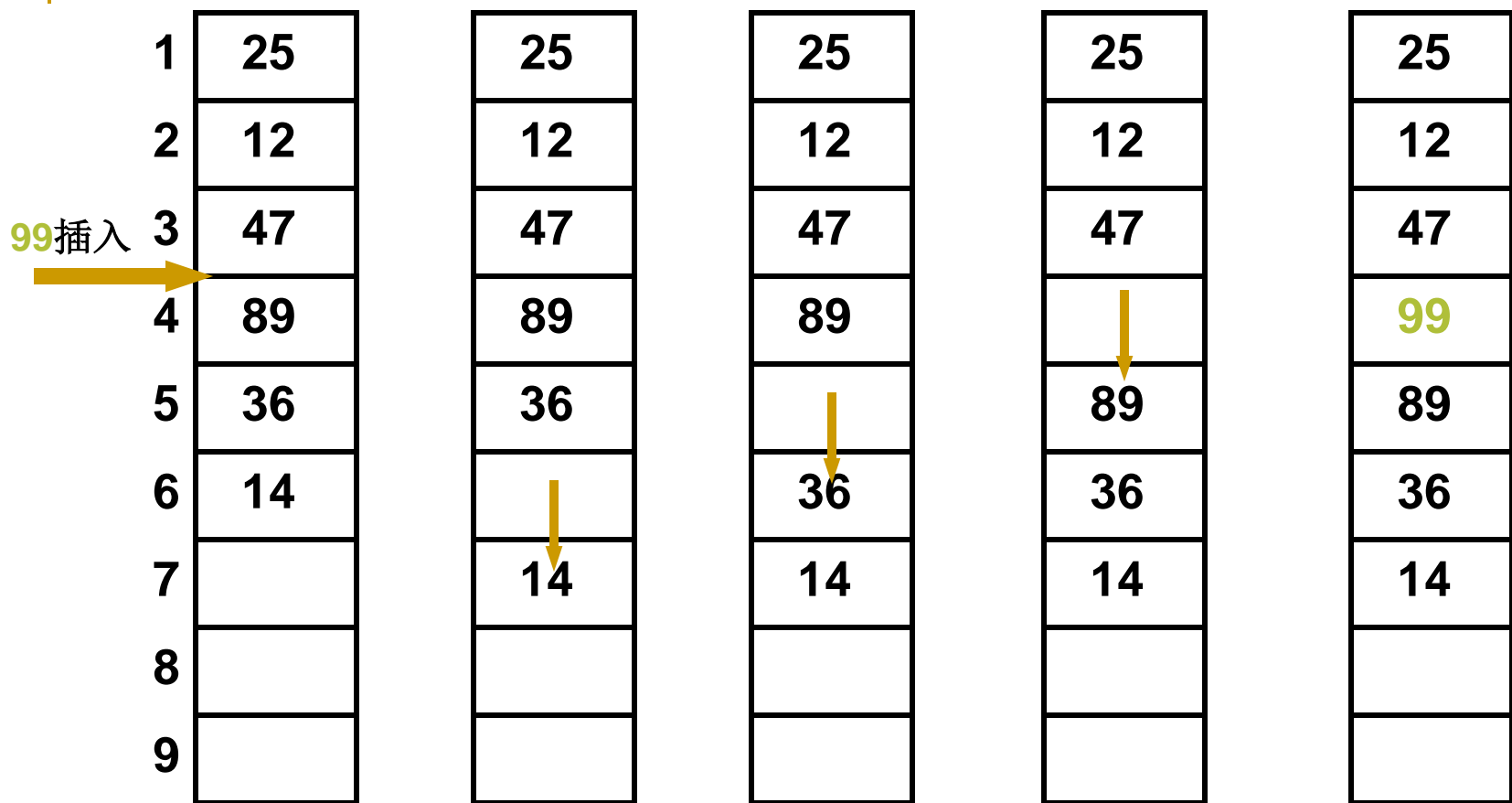
### 3. 查找 (根据指定数据获取数据所在的位置)

在线性表L中查找值为e的数据元素

```
int LocateELem(SqList L,ElemType e)
{
    for (i=0;i< L.length;i++)
        if (L.elem[i]==e) return i+1;
    return 0;
}
```

查找算法时间效率分析 ? ? ?

## 4. 插入（插在第 $i$ 个结点之前）



插第 4 个结点之前，移动  $6-4+1$  次

插在第  $i$  个结点之前，移动  $n-i+1$  次

# 【算法步骤】

---

- (1) 判断插入位置 $i$  是否合法。
  - (2) 判断顺序表的存储空间是否已满。
  - (3) 将第 $n$ 至第 $i$  位的元素依次向后移动一个位置，空出第 $i$ 个位置。
  - (4) 将要插入的新元素 $e$ 放入第 $i$ 个位置。
  - (5) 表长加1，插入成功返回OK。
-

# 【算法描述】

4. 在线性表L中第i个数据元素之前插入数据元素e

```
Status ListInsert_Sq(SqList &L,int i ,ElemType e){  
    if(i<1 || i>L.length+1) return ERROR;           //i值不合法  
    if(L.length==MAXSIZE) return ERROR;  //当前存储空间已满  
    for(j=L.length-1;j>=i-1;j--)  
        L.elem[j+1]=L.elem[j];  //插入位置及之后的元素后移  
    L.elem[i-1]=e;               //将新元素e放入第i个位置  
    ++L.length;                  //表长增1  
    return OK;  
}
```



# 【算法分析】

算法时间主要耗费在移动元素的操作上

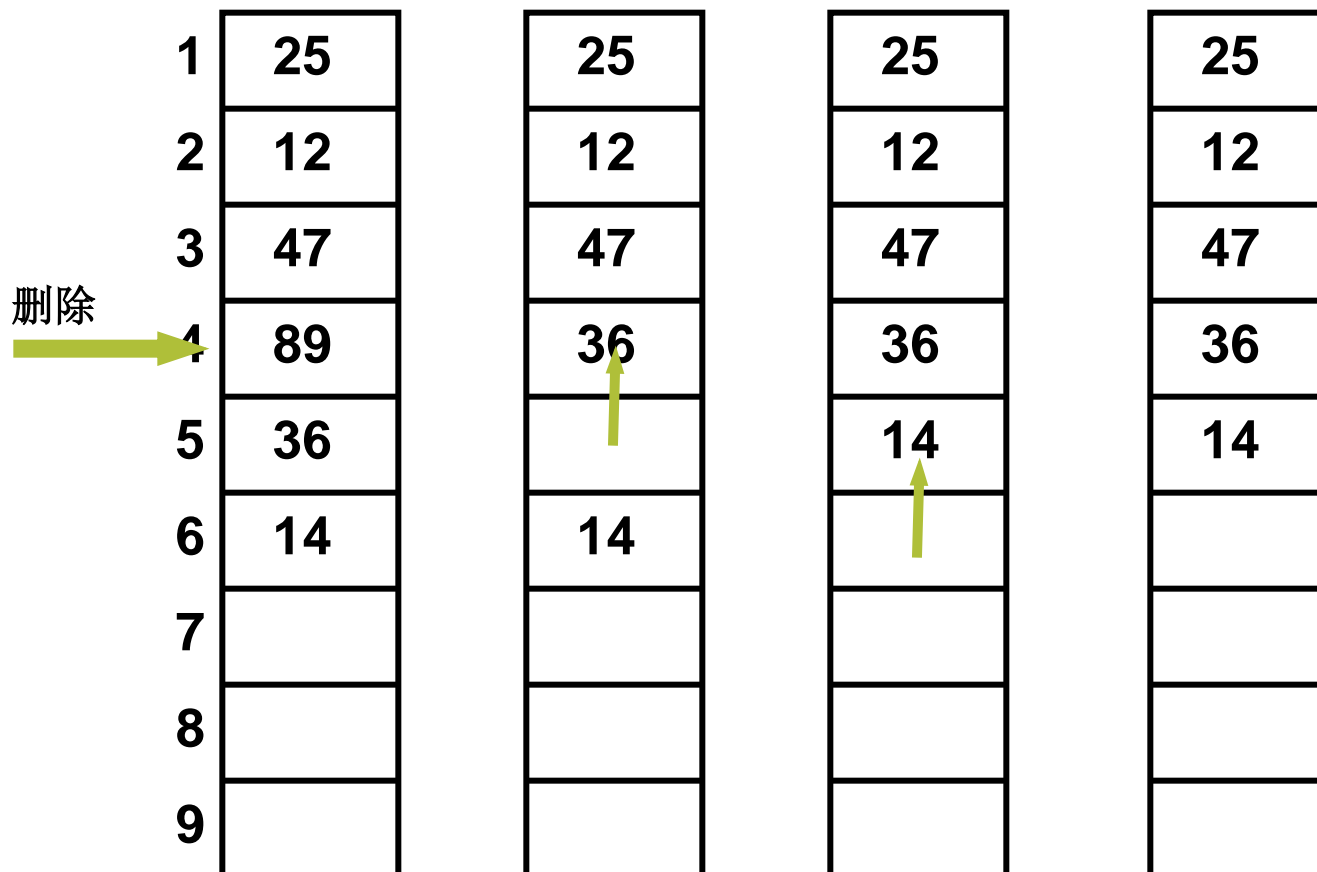
若插入在尾结点之后，则根本无需移动（特别快）；

若插入在首结点之前，则表中元素全部后移（特别慢）；

若要考虑在各种位置插入（共 $n+1$ 种可能）的平均移动次数，该如何计算？

$$\begin{aligned} \text{AMN} &= \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} (n + \cdots + 1 + 0) \\ &= \frac{1}{(n+1)} \frac{n(n+1)}{2} = \frac{n}{2} \end{aligned}$$

## 5. 删除 (删除第 $i$ 个结点)



删除第 4 个结点，移动  $6-4$  次

删除第  $i$  个结点，移动  $n-i$  次

# 【算法步骤】

---

- (1) 判断删除位置 $i$  是否合法（合法值为 $1 \leq i \leq n$ ）。
  - (2) 将欲删除的元素保留在 $e$ 中。
  - (3) 将第 $i+1$ 至第 $n$  位的元素依次向前移动一个位置。
  - (4) 表长减1，删除成功返回OK。
-

# 【算法描述】

---

5. 将线性表L中第i个数据元素删除

```
Status ListDelete_Sq(SqList &L,int i){  
    if((i<1)||(i>L.length)) return ERROR;    //i值不合法  
    for (j=i;j<=L.length-1;j++)  
        L.elem[j-1]=L.elem[j];    //被删除元素之后的元素前移  
    --L.length;    //表长减1  
    return OK;  
}
```

---

# 【算法分析】

算法时间主要耗费在移动元素的操作上

若删除尾结点，则根本无需移动（特别快）；

若删除首结点，则表中 $n-1$ 个元素全部前移（特别慢）；

若要考虑在各种位置删除（共 $n$ 种可能）的平均移动次数，该如何计算？

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

---

查找、插入、删除算法的平均时间复杂度为  
 $O(n)$

显然，顺序表的空间复杂度 $S(n)=O(1)$   
(没有占用辅助空间)

---

# 顺序表（顺序存储结构）的特点

- (1) 利用数据元素的存储位置表示线性表中相邻数据元素之间的前后关系，即线性表的**逻辑结构与存储结构一致**
- (2) 在访问线性表时，可以快速地计算出任何一个数据元素的存储地址。因此可以粗略地认为，**访问每个元素所花时间相等**

这种存取元素的方法被称为**随机存取法**

# 顺序表的优缺点

## 优点:

- ✓ **存储密度大** (结点本身所占存储量/结点结构所占存储量)
- ✓ 可以**随机存取**表中任一元素

## 缺点:

- ✓ 在插入、删除某一元素时, 需要移动大量元素
- ✓ 浪费存储空间
- ✓ 属于静态存储形式, 数据元素的个数不能自由扩充

为克服这一缺点



**链表**



## 2.5 线性表的链式表示和实现



### 链式存储结构

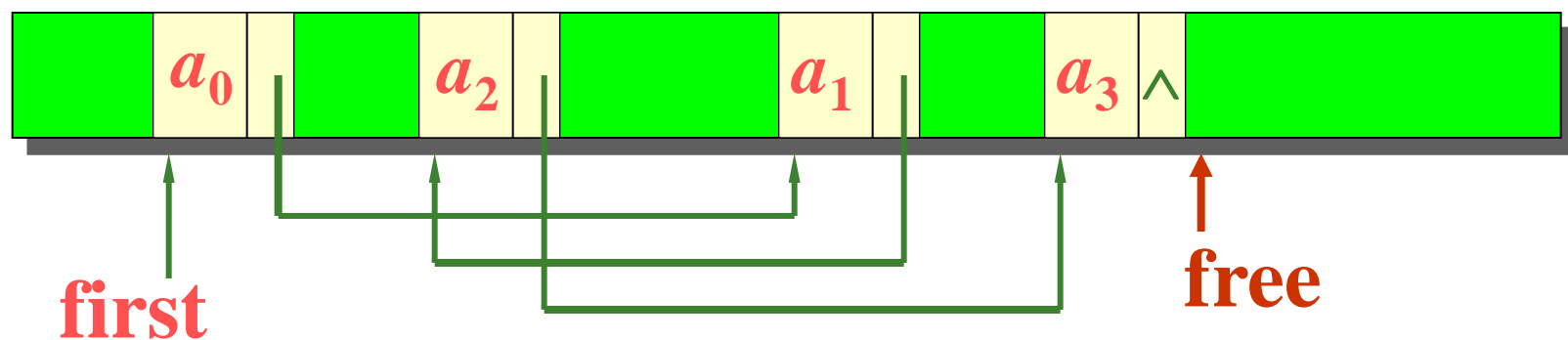
结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻

线性表的链式表示又称为非顺序映像或链式映像。

如何实现？

通过**指针**来实现

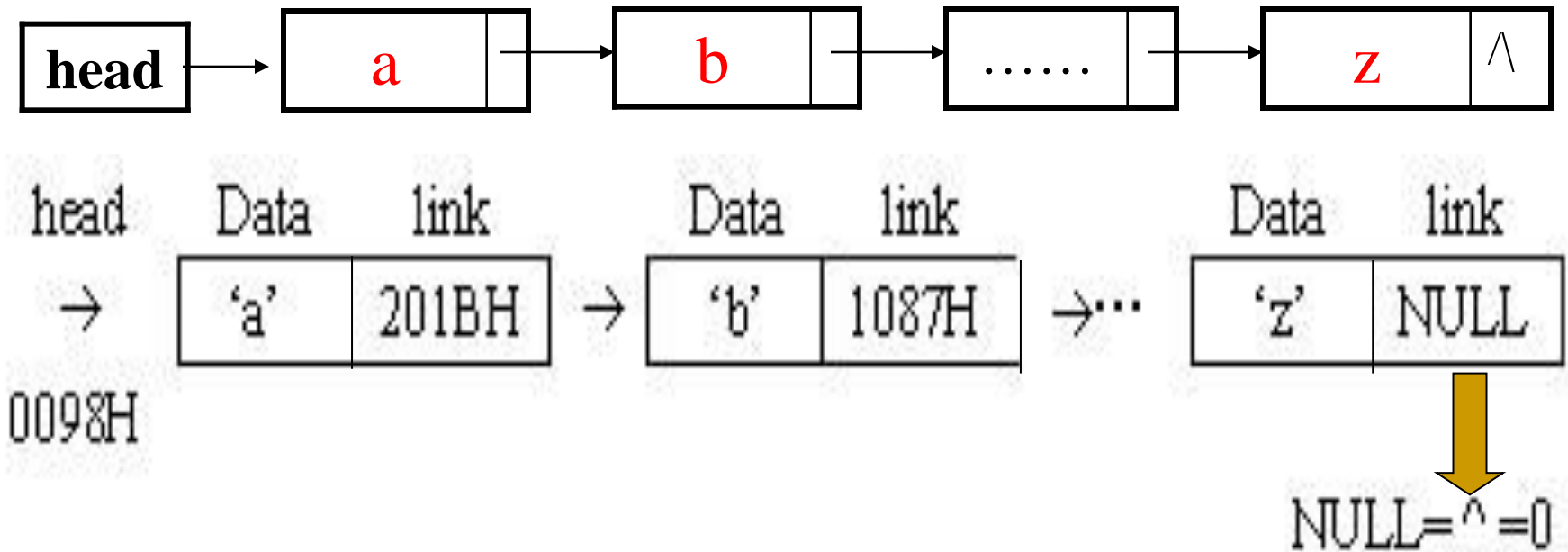
## 单链表的存储映像

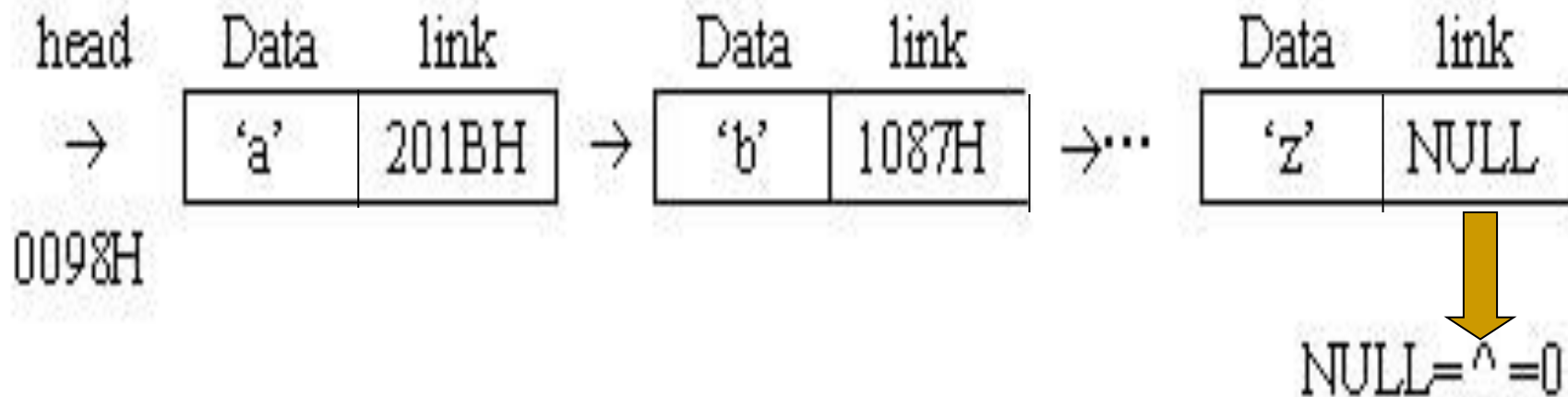


# 例 画出26个英文字母表的链式存储结构

逻辑结构: ( a, b, ..., y, z )

链式存储结构:





各结点由两个域组成：

|    |    |
|----|----|
| 数据 | 指针 |
|----|----|

**数据域：** 存储元素数值数据

**指针域：** 存储直接后继结点的存储位置

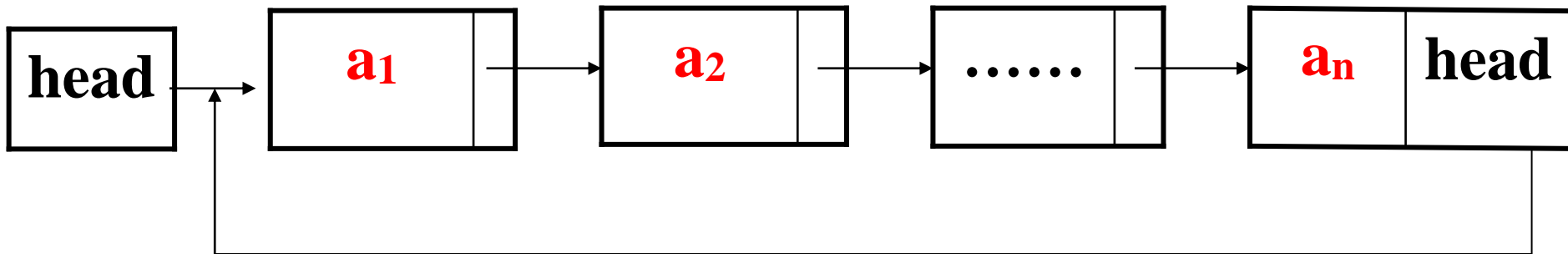
# 与链式存储有关的术语

- 1、**结点**：数据元素的存储映像。由数据域和指针域两部分组成
- 2、**链表**： $n$  个结点由**指针链**组成一个链表。它是线性表的链式存储映像，称为线性表的链式存储结构

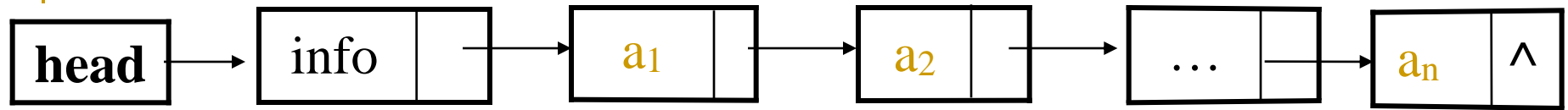
### 3、单链表、双链表、循环链表：

- 结点只有一个指针域的链表，称为**单链表**或**线性链表**
- 有两个指针域的链表，称为**双链表**
- 首尾相接的链表称为**循环链表**

循环链表示意图：



## 4、头指针、头结点和首元结点



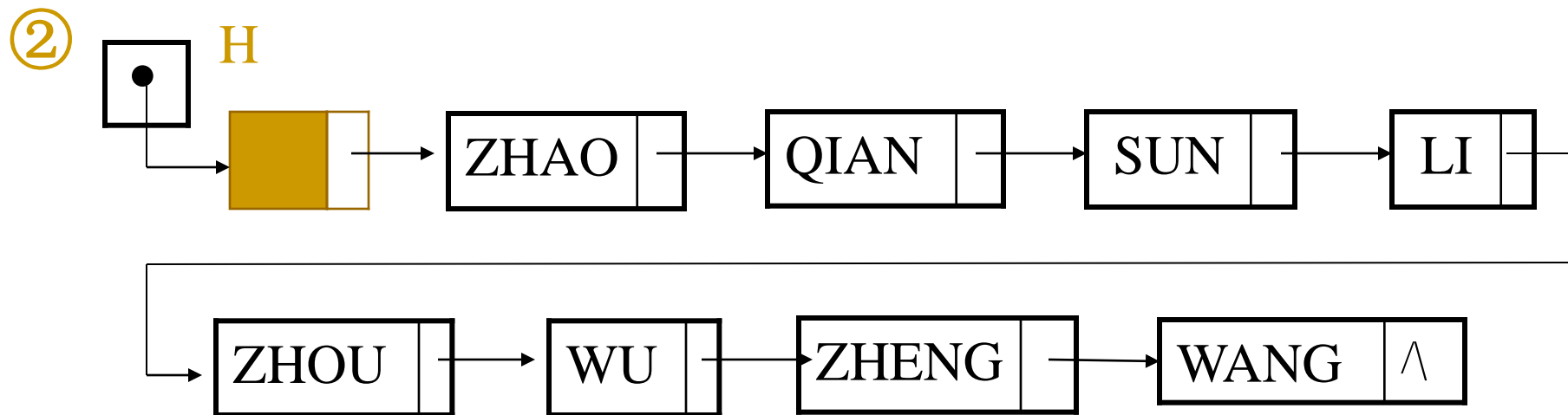
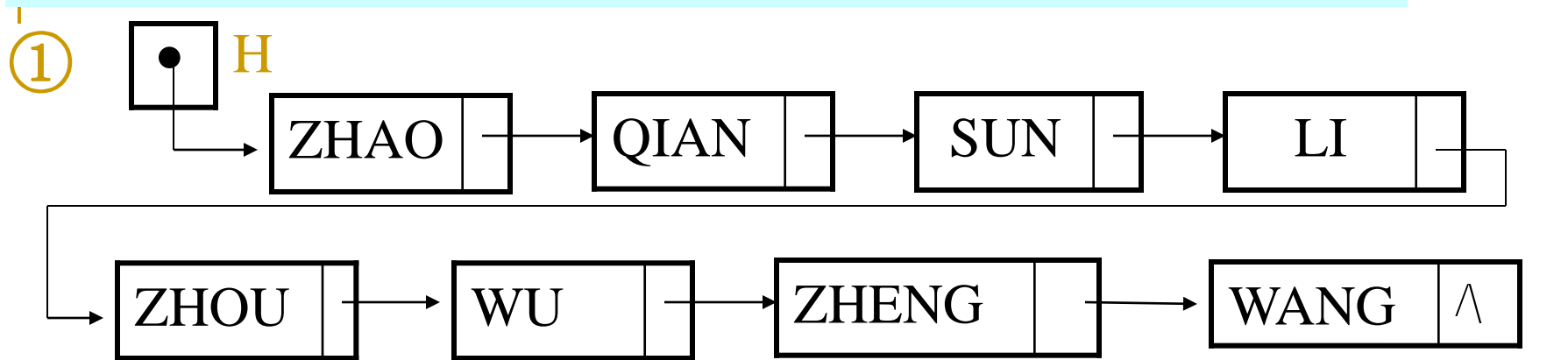
头指针 头结点 首元结点

头指针是指向链表中第一个结点的指针

首元结点是指链表中存储第一个数据元素 $a_1$ 的结点

头结点是在链表的首元结点之前附设的一个结点；数据域内只放空表标志和表长等信息

上例链表的逻辑结构示意图有以下两种形式：

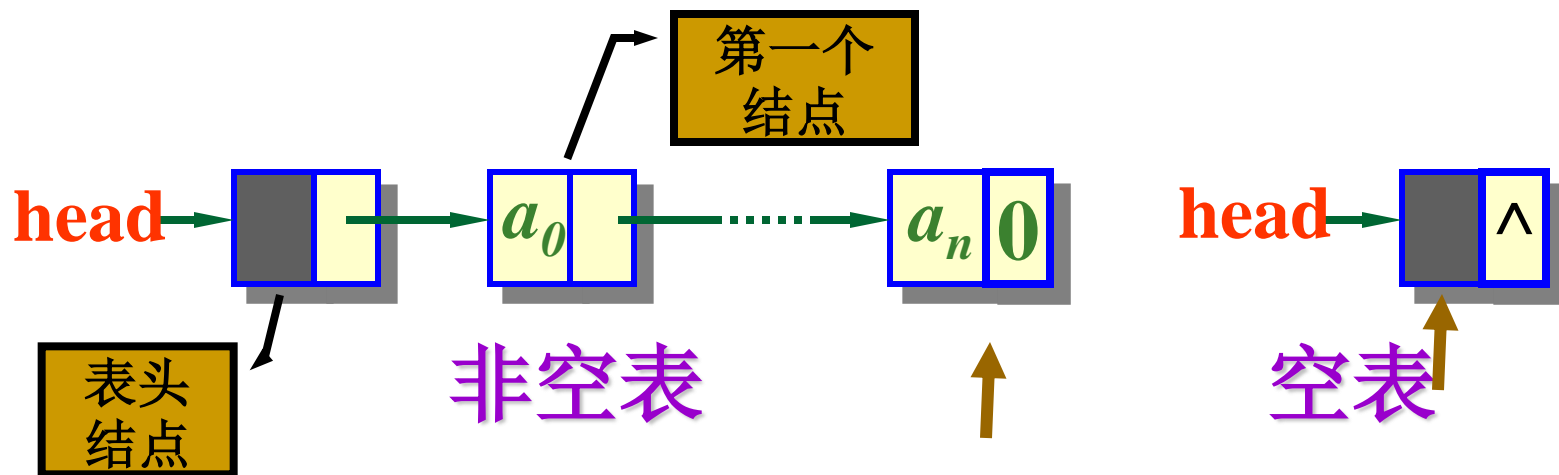


区别：① 无头结点 ② 有头结点



## 讨论1. 如何表示空表?

有头结点时, 当头结点的指针域为空时表示空表



## 讨论2. 在链表中设置头结点有什么好处？

### 1. 便于首元结点的处理

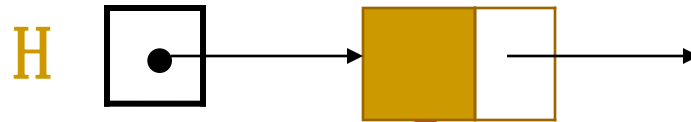
首元结点的地址保存在头结点的指针域中，所以在链表的第一个位置上的操作和其它位置一致，无须进行特殊处理；

### 2. 便于空表和非空表的统一处理

无论链表是否为空，头指针都是指向头结点的非空指针，因此空表和非空表的处理也就统一了。

### 讨论3. 头结点的**数据域**内装的是什么？

头结点的**数据域**可以为空，也可存放线性表**长度**等附加信息，但此结点不能计入链表长度值。



头结点的数据域

# 链表（链式存储结构）的特点

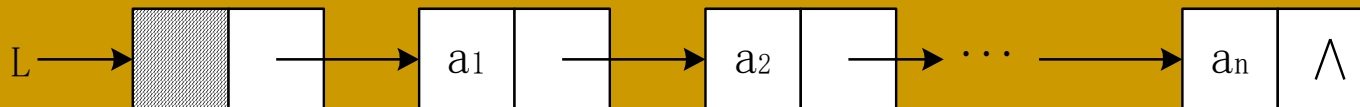
- （1）结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻
- （2）访问时只能通过头指针进入链表，并通过每个结点的指针域向后扫描其余结点，所以寻找第一个结点和最后一个结点所花费的时间不等

这种存取元素的方法被称为顺序存取法

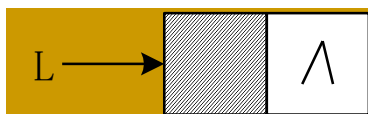
## 2.5.1 单链表的定义和实现



非空表



空表



- ✓ 单链表是由表头唯一确定，因此单链表可以用头指针的名字来命名
- ✓ 若头指针名是L，则把链表称为表L

# 单链表的存储结构定义

```
typedef struct LNode{  
    ElemType    data;           //数据域  
    struct LNode *next;        //指针域  
} LNode, *LinkList;  
    // *LinkList为Lnode类型的指针
```

**LNode \*p**

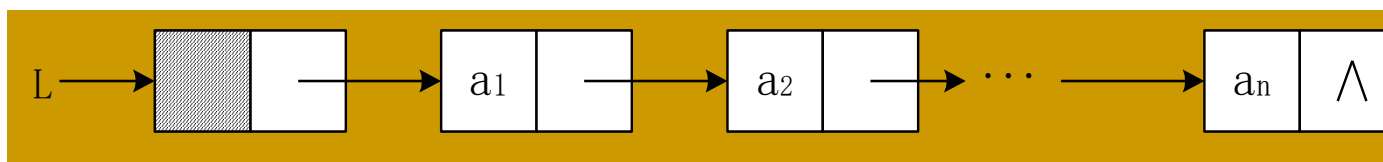


**LinkList p**

# 注意区分指针变量和结点变量两个不同的概念

- 指针变量p: 表示结点地址
- 结点变量\*p: 表示一个结点

**LNode \*p**



若  $p \rightarrow \text{data} = a_i$ , 则  $p \rightarrow \text{next} \rightarrow \text{data} = a_{i+1}$

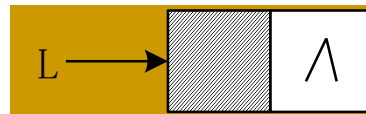
## 2.5.2 单链表基本操作的实现



1. 初始化
2. 取值
3. 查找
4. 插入
5. 删除



## 1. 初始化(构造一个空表)



### 【算法步骤】

- (1) 生成新结点作头结点，用头指针L指向头结点。
- (2) 头结点的指针域置空。

### 【算法描述】

```
Status InitList_L(LinkList &L){  
    L=new LNode;  
    L->next=NULL;  
    return OK;  
}
```

# 补充：几个简单基本操作的算法实现

## 销毁

```
Status DestroyList_L(LinkList &L){  
    LinkList p;  
    while(L)  
    {  
        p=L;  
        L=L->next;  
        delete p;  
    }  
    return OK;  
}
```

---

# 补充：几个简单基本操作的算法实现

## 清空

```
Status ClearList(LinkList & L){
```

```
    // 将L重置为空表
```

```
    LinkList p,q;
```

```
    p=L->next;    //p指向第一个结点
```

```
    while(p)        //没到表尾
```

```
        { q=p->next; delete p;    p=q;  }
```

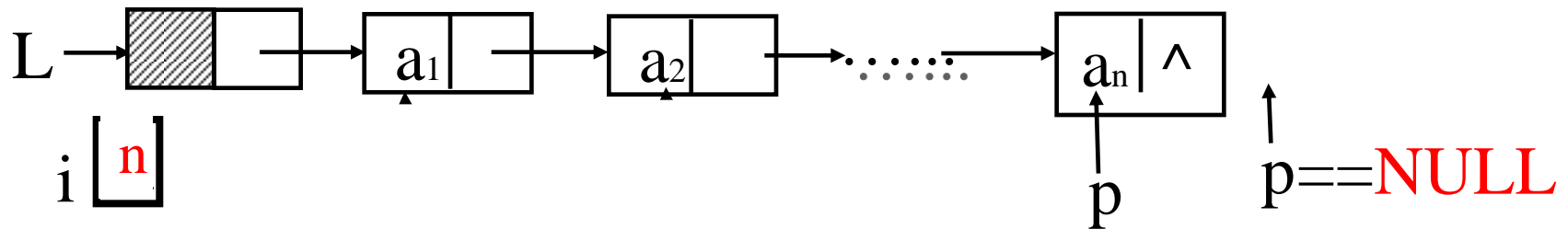
```
    L->next=NULL;    //头结点指针域为空
```

```
    return OK;
```

```
}
```

## 补充：几个简单基本操作的实现

### 求表长



### “数”结点：

- 指针  $p$  依次指向各个结点
- 从第一个元素开始“数”
- 一直“数”到最后一个结点

```
p=L->next;
```

```
i=0;
```

```
while(p){i++;p=p->next;}
```

## 求表长

```
int ListLength_L(LinkList L){
```

```
//返回L中数据元素个数
```

```
    LinkList p;
```

```
    p=L->next; //p指向第一个结点
```

```
    i=0;
```

```
    while(p){//遍历单链表,统计结点数
```

```
        i++;
```

```
        p=p->next;    }
```

```
    return i;
```

```
}
```

## “数”结点:

- 指针p依次指向各个结点
- 从第一个元素开始“数”
- 一直“数”到最后一个结点

---

判断表是否为空

```
int ListEmpty(LinkList L){  
//若L为空表，则返回1，否则返回0  
    if(L->next) //非空  
        return 0;  
    else  
        return 1;  
}
```

---

# 线性表的重要基本操作

---

1. 初始化

2. 取值

3. 查找

4. 插入

5. 删除

---

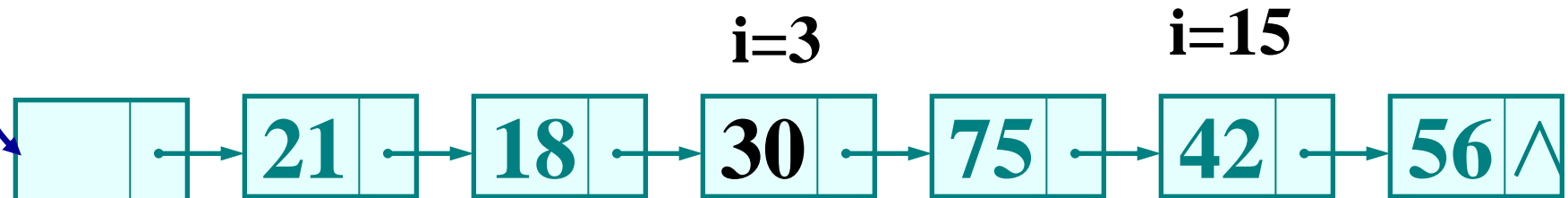
## 2. 取值 (根据位置*i*获取相应位置数据元素的内容)

- 思考：顺序表里如何找到第*i*个元素？
- 链表的查找：要从链表的头指针出发，顺着链域next逐个结点往下搜索，直至搜索到第*i*个结点为止。因此，链表不是随机存取结构



例：分别取出表中 $i=3$ 和 $i=15$ 的元素

**L**



**j**

7

**p**

## 【算法步骤】

- ✓ 从第1个结点 ( $L \rightarrow next$ ) 顺链扫描，用指针 $p$ 指向当前扫描到的结点， $p$ 初值 $p = L \rightarrow next$ 。
- ✓  $j$ 做计数器，累计当前扫描过的结点数， $j$ 初值为1。
- ✓ 当 $p$ 指向扫描到的下一结点时，计数器 $j$ 加1。
- ✓ 当 $j = i$ 时， $p$ 所指的结点就是要找的第 $i$ 个结点。

## 2. 取值 (根据位置i获取相应位置数据元素的内容)

//获取线性表L中的某个数据元素的内容

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

```
    p=L->next;j=1; //初始化
```

```
    while(p&& j<i){ //向后扫描, 直到p指向第i个元素或p为空
```

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

```
    }
```

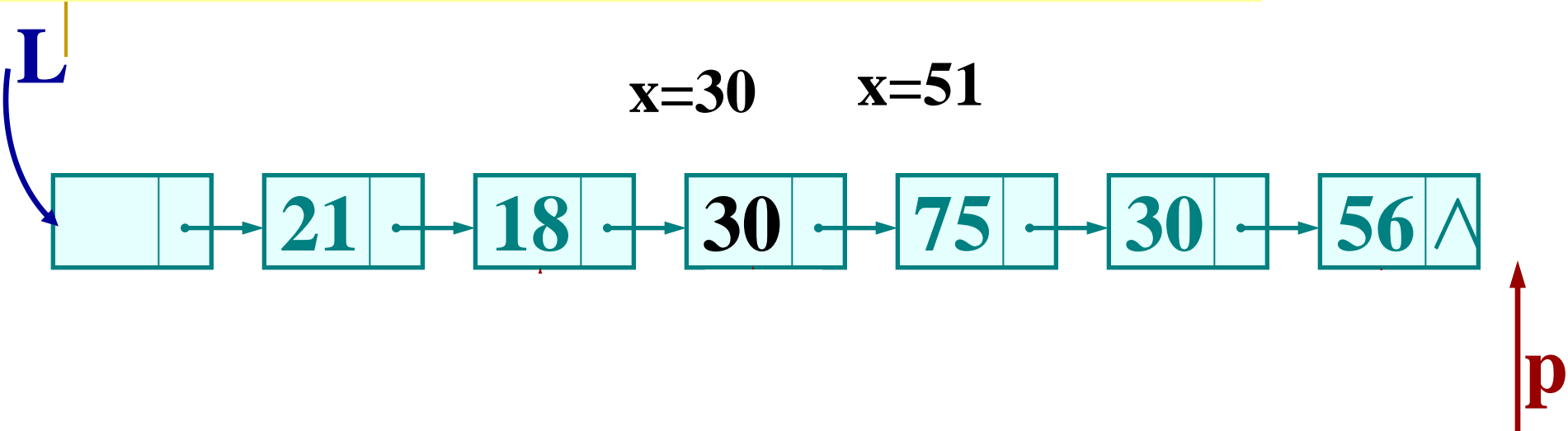
```
    if(!p || j>i) return ERROR; //第i个元素不存在
```

```
    e=p->data; //取第i个元素
```

```
    return OK;
```

```
}//GetElem_L
```

### 3. 查找 (根据指定数据获取数据所在的位置)



**j** **7** 未找到，返回0

- ✓从第一个结点起，依次和e相比较。
- ✓如果找到一个其值与e相等的数据元素，则返回其在链表中的“位置”或地址；
- ✓如果查遍整个链表都没有找到其值和e相等的元素，则返回0或“NULL”。

# 【算法描述】

---

//在线性表L中查找值为e的数据元素

**LNode \*LocateElem\_L (LinkList L, Elemtype e) {**

**//返回L中值为e的数据元素的地址，查找失败返回NULL**

**p=L->next;**

**while(p && p->data!=e)**

**p=p->next;**

**return p;**

**}**

---

# 【算法描述】

---

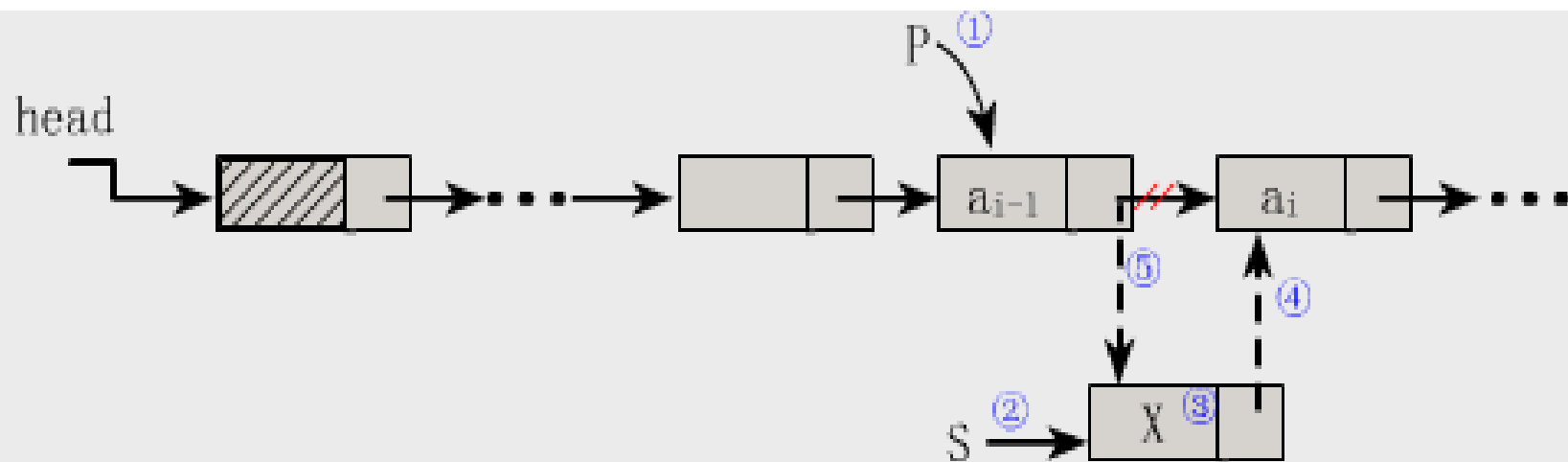
//在线性表L中查找值为e的数据元素

```
int LocateELem_L (LinkList L, Elemtype e) {  
    //返回L中值为e的数据元素的位置序号，查找失败返回0  
    p=L->next; j=1;  
    while(p && p->data!=e)  
        {p=p->next; j++;}  
    if(p) return j;  
    else return 0;  
}
```

---

#### 4. 插入（插在第 $i$ 个结点之前）

- 将值为 $x$ 的新结点插入到表的第 $i$ 个结点的位置上，即插入到 $a_{i-1}$ 与 $a_i$ 之间



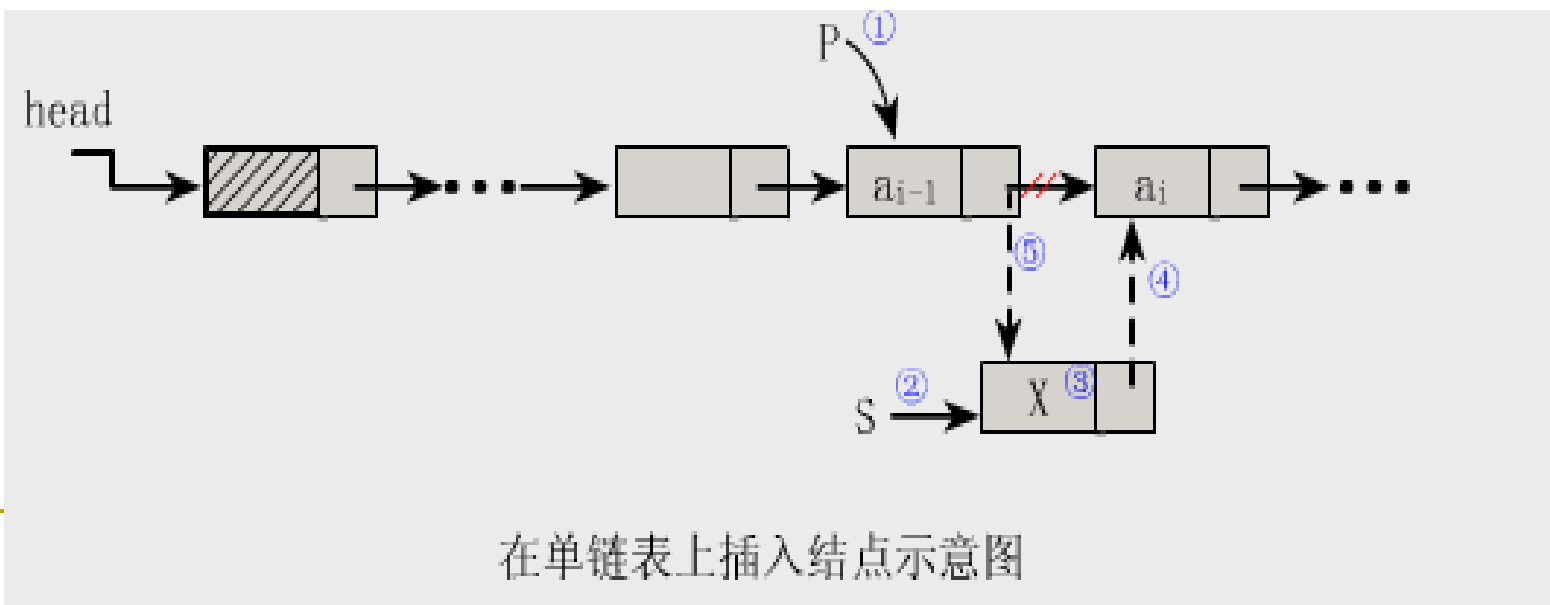
在单链表上插入结点示意图

```
s->next=p->next;    p->next=s
```

思考：步骤1和2能互换么？

# 【算法步骤】

- (1) 找到 $a_{i-1}$ 存储位置 $p$
- (2) 生成一个新结点 $*s$
- (3) 将新结点 $*s$ 的数据域置为 $x$
- (4) 新结点 $*s$ 的指针域指向结点 $a_i$
- (5) 令结点 $*p$ 的指针域指向新结点 $*s$



# 【算法描述】

---

//在L中第i个元素之前插入数据元素e

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

```
    p=L;j=0;
```

```
    while(p&& j<i-1){p=p->next;++j;}    //寻找第i-1个结点
```

```
    if(!p||j>i-1)return ERROR;    //i大于表长 + 1或者小于1
```

```
    s=new LNode;    //生成新结点s
```

```
    s->data=e;    //将结点s的数据域置为e
```

```
    s->next=p->next;    //将结点s插入L中
```

```
    p->next=s;
```

```
    return OK;
```

```
}//ListInsert_L
```

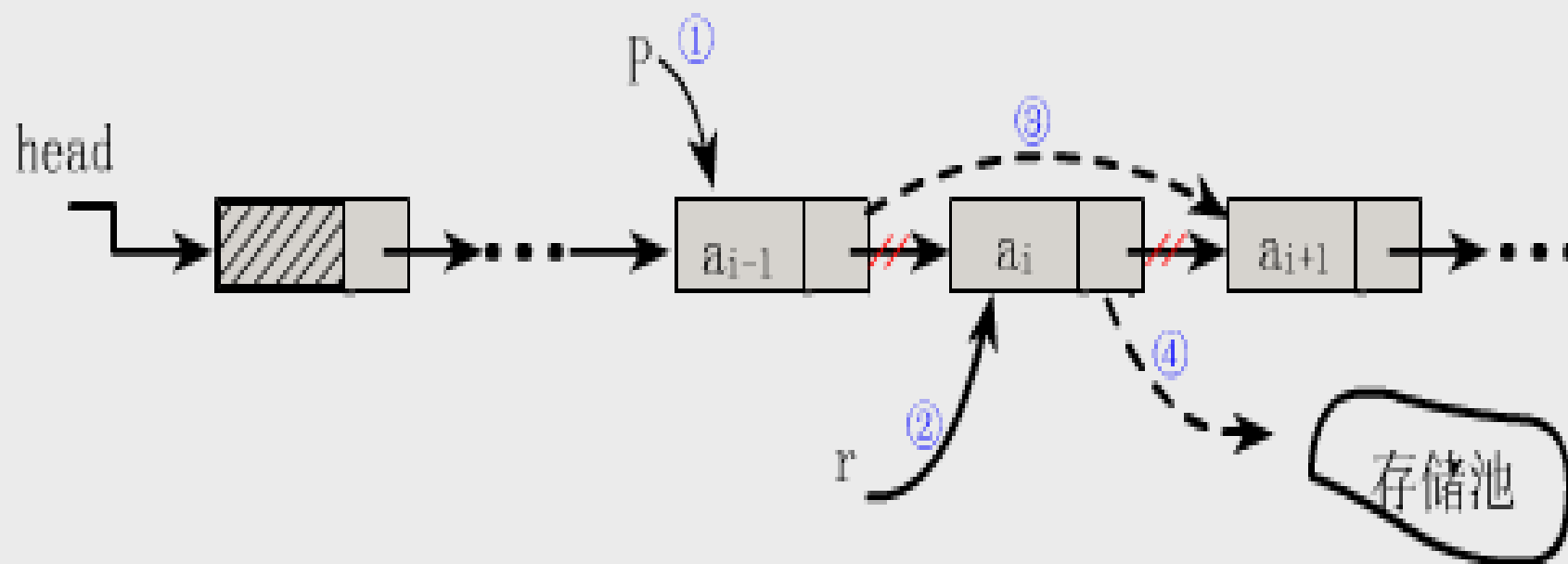
---



## 5. 删除 (删除第 $i$ 个结点)

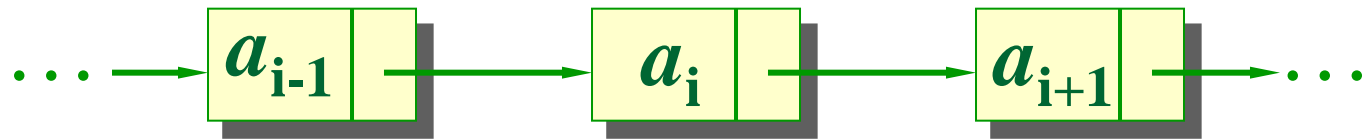
- 将表的第 $i$ 个结点删去
- 步骤:
  - (1) 找到 $a_{i-1}$ 存储位置 $p$
  - (2) 保存要删除的结点的值
  - (3) 令 $p \rightarrow \text{next}$ 指向 $a_i$ 的直接后继结点
  - (4) 释放结点 $a_i$ 的空间

## 5. 删除 (删除第 $i$ 个结点)

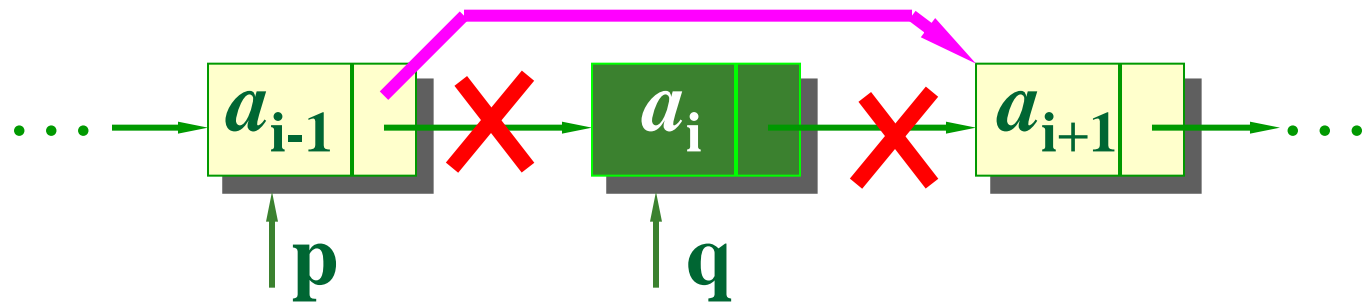


在单链表上删除结点示意图

## 5. 删除 (删除第 $i$ 个结点)



删除前

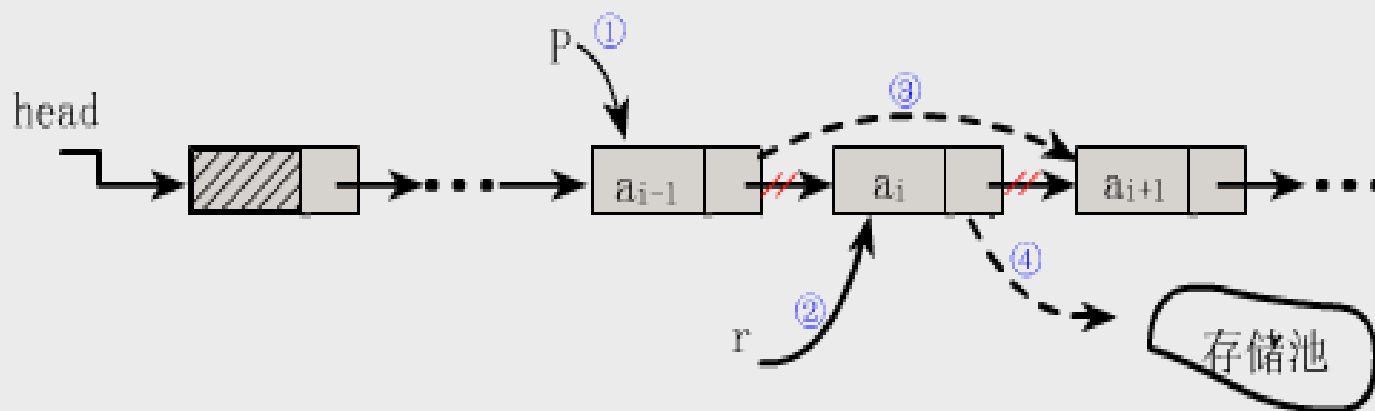


删除后

$p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next} \quad ???$

# 【算法步骤】

- (1) 找到 $a_{i-1}$ 存储位置 $p$
- (2) 临时保存结点 $a_i$ 的地址在 $q$ 中，以备释放
- (3) 令 $p \rightarrow \text{next}$ 指向 $a_i$ 的直接后继结点
- (4) 将 $a_i$ 的值保留在 $e$ 中
- (5) 释放 $a_i$ 的空间



在单链表上删除结点示意图

# 【算法描述】

//将线性表L中第i个数据元素删除

```
Status ListDelete_L(LinkList &L,int i,ElemType &e){  
    p=L;j=0;  
    while(p->next &&j<i-1){//寻找第i个结点，并令p指向其前驱  
        p=p->next; ++j;  
    }  
    if(!(p->next)||j>i-1) return ERROR; //删除位置不合理  
    q=p->next; //临时保存被删结点的地址以备释放  
    p->next=q->next; //改变删除结点前驱结点的指针域  
    e=q->data; //保存删除结点的数据域  
    delete q;    //释放删除结点的空间  
    return OK;  
}  
//ListDelete_L
```

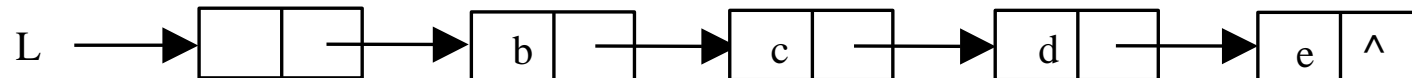
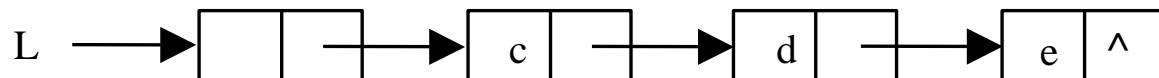
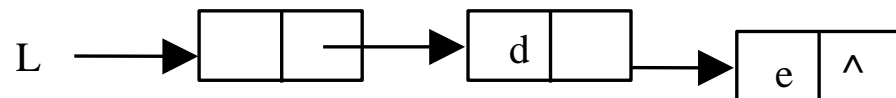
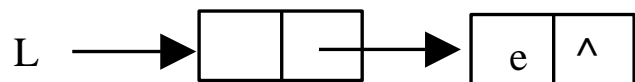
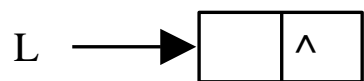
# 链表的运算时间效率分析

1. **查找**：因线性链表只能顺序存取，即在查找时要从头指针找起，查找的时间复杂度为  $O(n)$ 。
2. **插入和删除**：因线性链表不需要移动元素，只要修改指针，一般情况下时间复杂度为  $O(1)$ 。

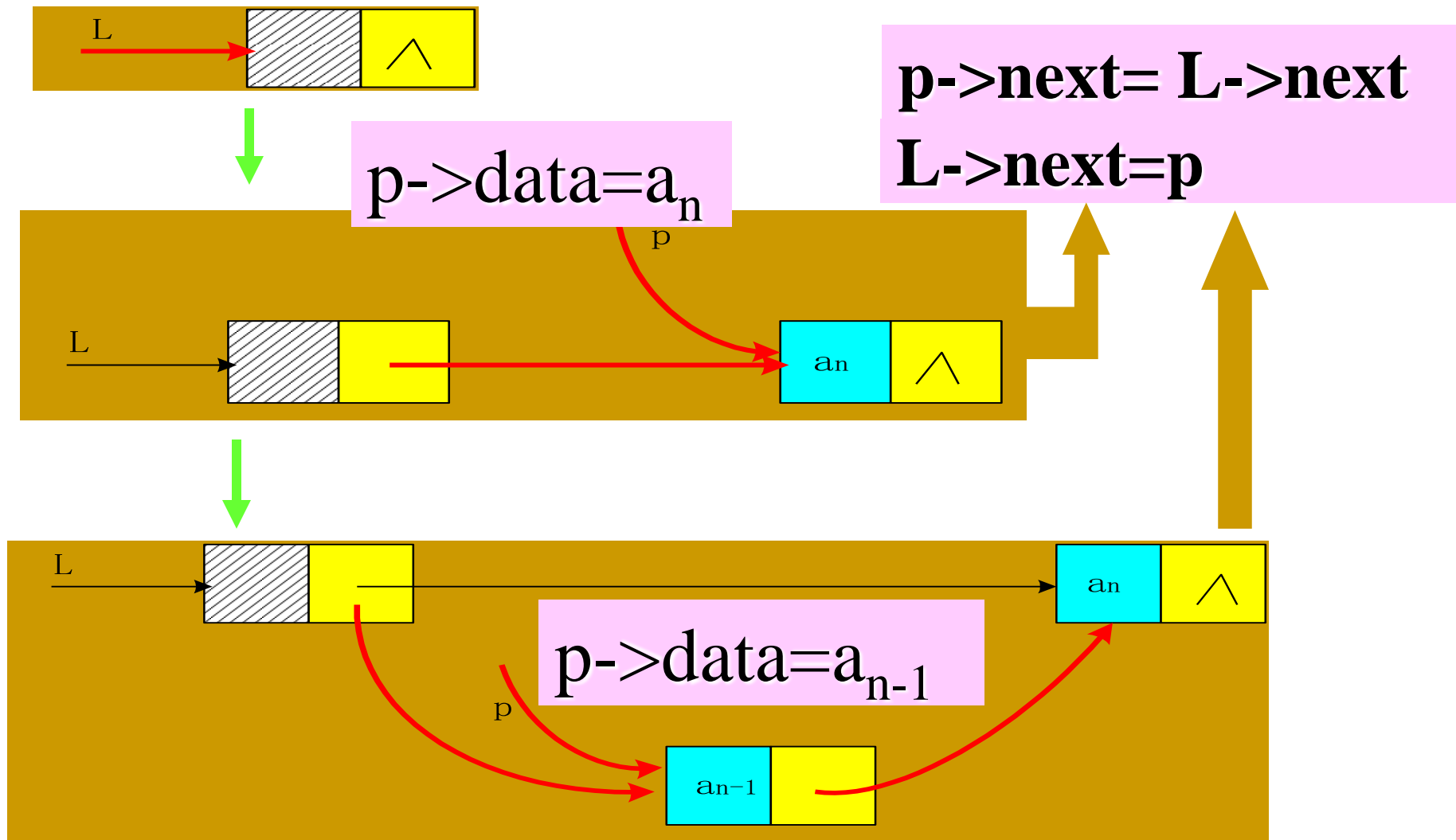
但是，如果要在单链表中进行前插或删除操作，由于要从头查找前驱结点，所耗时间复杂度为  $O(n)$ 。

# 单链表的建立（前插法）

- 从一个空表开始，重复读入数据：
  - ◆ 生成新结点
  - ◆ 将读入数据存放到新结点的数据域中
  - ◆ 将该新结点插入到链表的前端



# 单链表的建立（前插法）



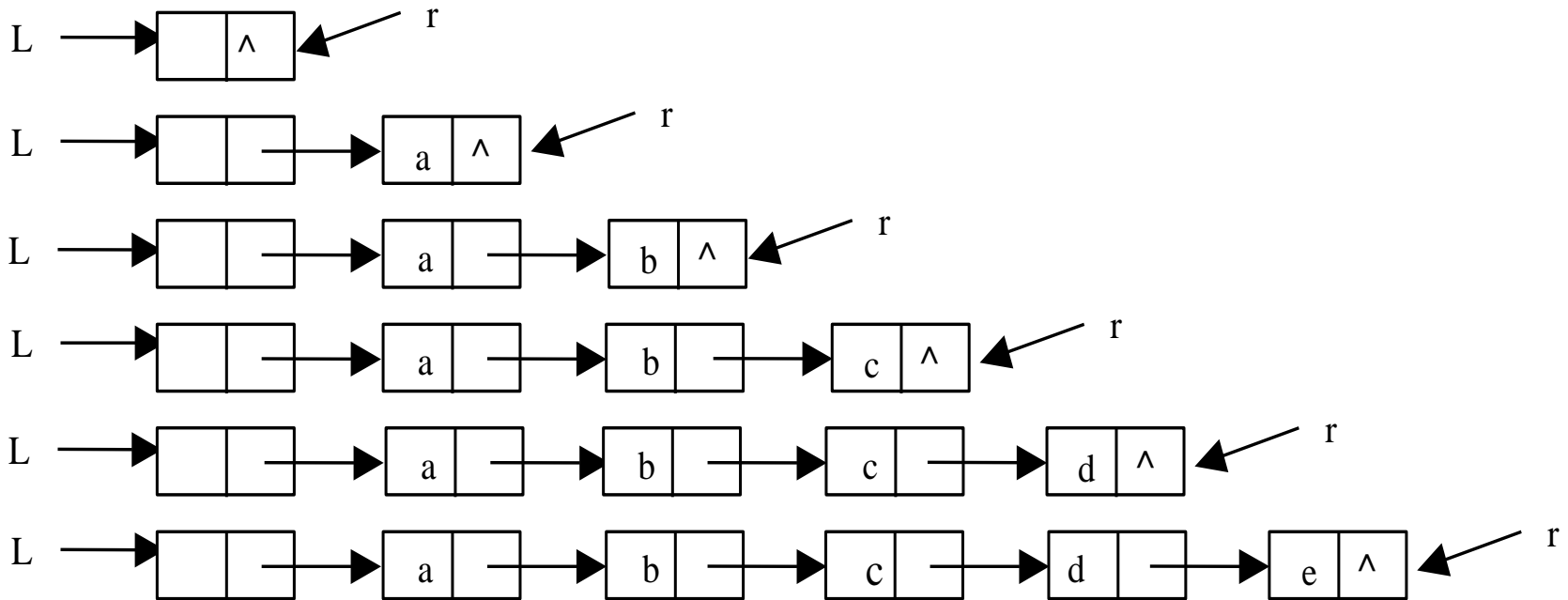


# 【算法描述】

```
void CreateList_F(LinkList &L,int n){  
    L=new LNode;  
    L->next=NULL; //先建立一个带头结点的单链表  
    for(i=n;i>0;--i){  
        p=new LNode; //生成新结点  
        cin>>p->data; //输入元素值  
        p->next=L->next;L->next=p; //插入到表头  
    }  
} //CreateList_F
```

# 单链表的建立（尾插法）

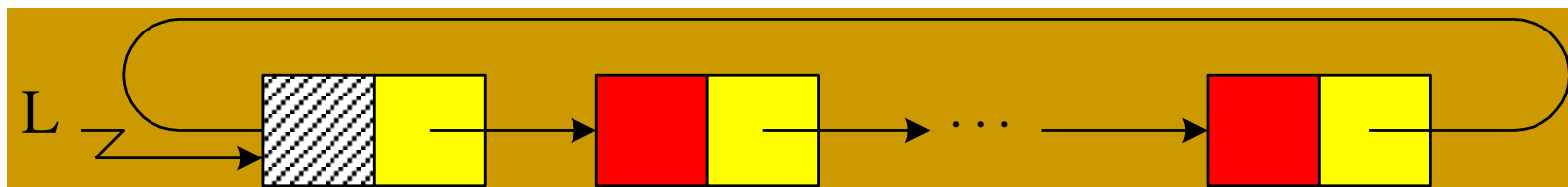
- 从一个空表L开始，将新结点逐个插入到链表的尾部，尾指针r指向链表的尾结点。
- 初始时，r同L均指向头结点。每读入一个数据元素则申请一个新结点，将新结点插入到尾结点后，r指向新结点。



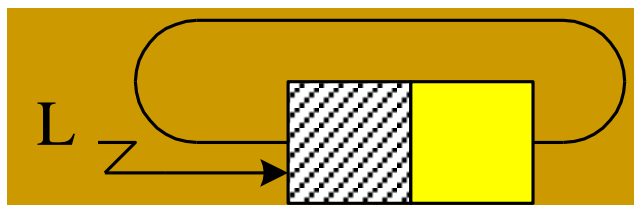
# 【算法描述】

```
void CreateList_L(LinkList &L,int n){  
    //正位序输入n个元素的值，建立带表头结点的单链表L  
    L=new LNode;  
    L->next=NULL;  
    r=L;    //尾指针r指向头结点  
    for(i=0;i<n;++i){  
        p=new LNode;    //生成新结点  
        cin>>p->data;    //输入元素值  
        p->next=NULL; r->next=p;    //插入到表尾  
        r=p;    //r指向新的尾结点  
    }  
} //CreateList_L
```

## 2.5.3 循环链表

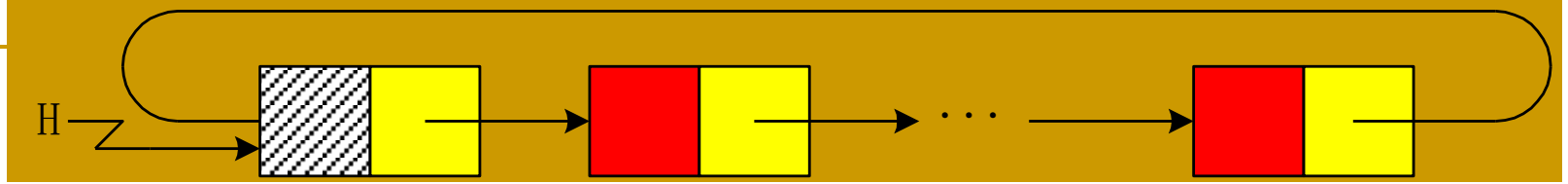


(a) 非空单循环链表



(b) 空表

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



从循环链表中的任何一个结点的位置都可以找到其他所有结点，而单链表做不到；

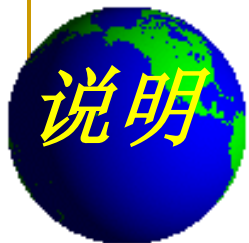


循环链表中没有明显的尾端

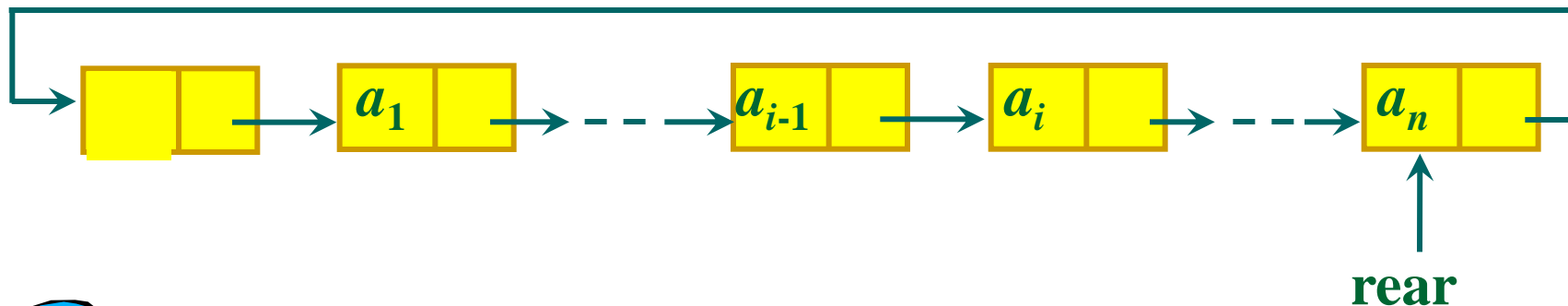


如何避免死循环

循环条件： $p \rightarrow next \neq NULL \rightarrow p \rightarrow next \neq L$



对循环链表，有时不给出头指针，而给出尾指针  
可以更方便的找到第一个和最后一个结点

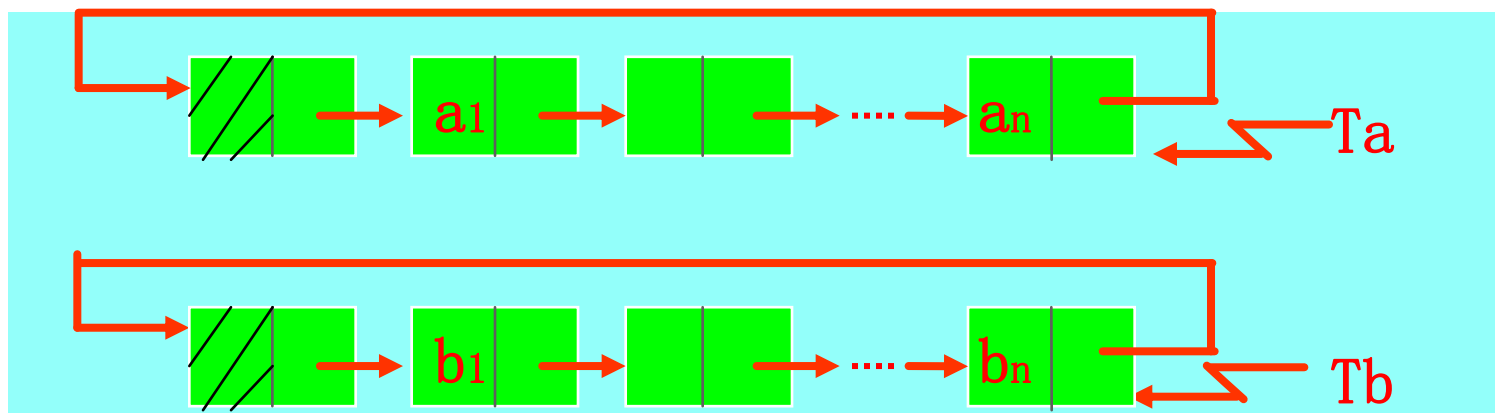
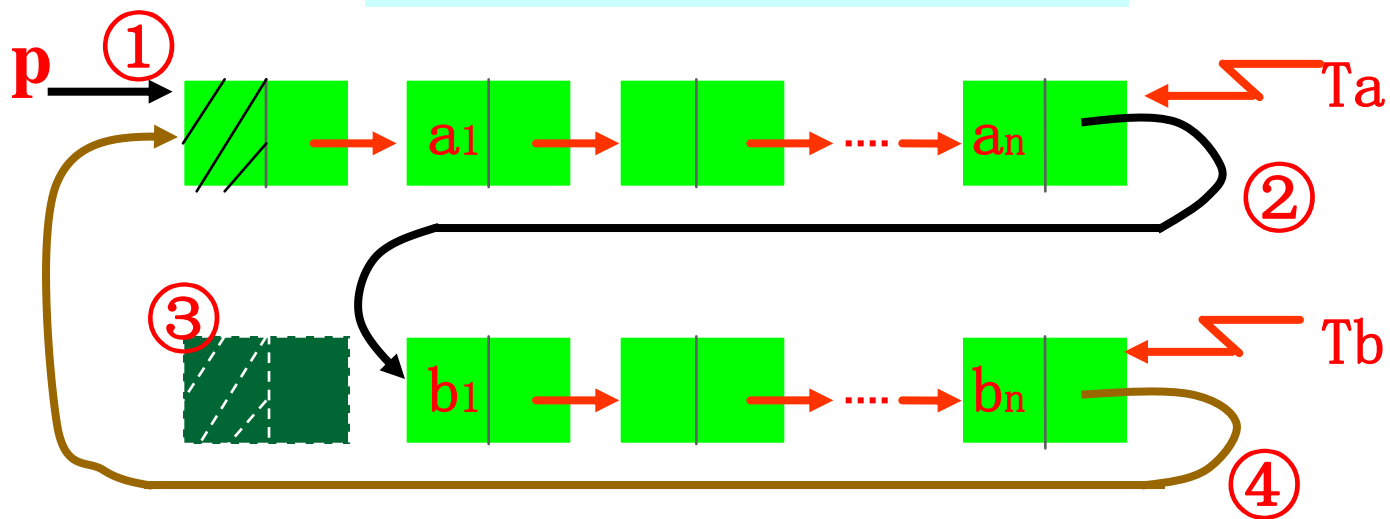


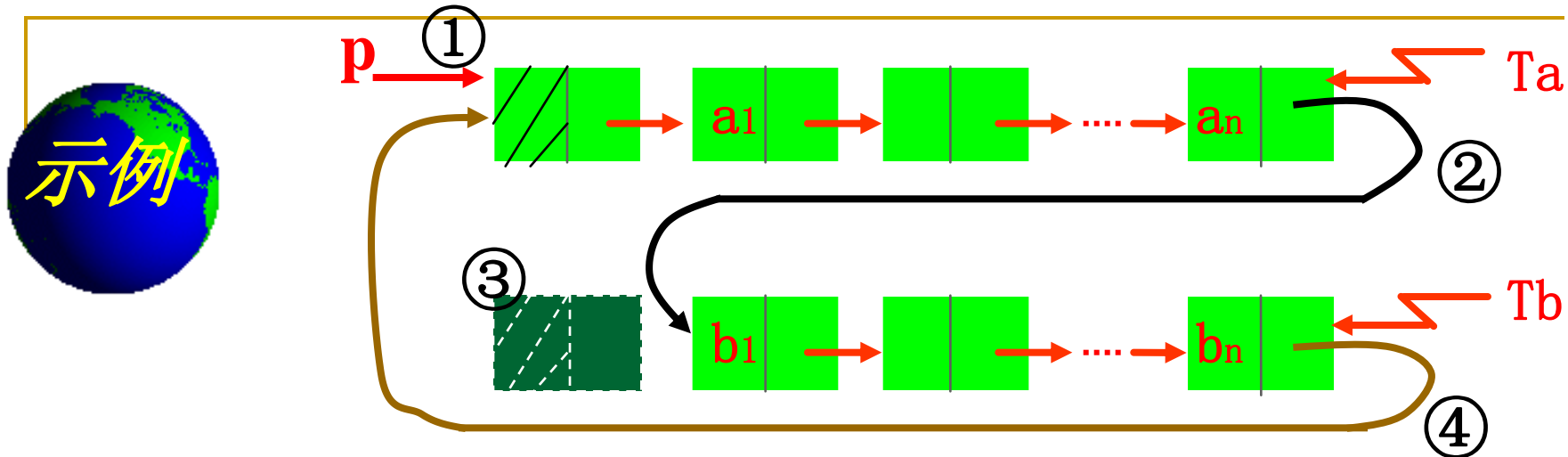
如何查找开始结点和终端结点？

开始结点: `rear->next->next`

终端结点: `rear`

# 循环链表的合并





**LinkedList Connect(LinkedList Ta,LinkedList Tb)**

{//假设Ta、Tb都是非空的单循环链表

**$p = Ta \rightarrow next;$**  //①p存表头结点

**$Ta \rightarrow next = Tb \rightarrow next \rightarrow next;$** //②Tb表头连结Ta表尾

**$delete Tb \rightarrow next;$**  //③释放Tb表头结点

**$Tb \rightarrow next = p;$**  //④修改指针

**$return Tb;$**

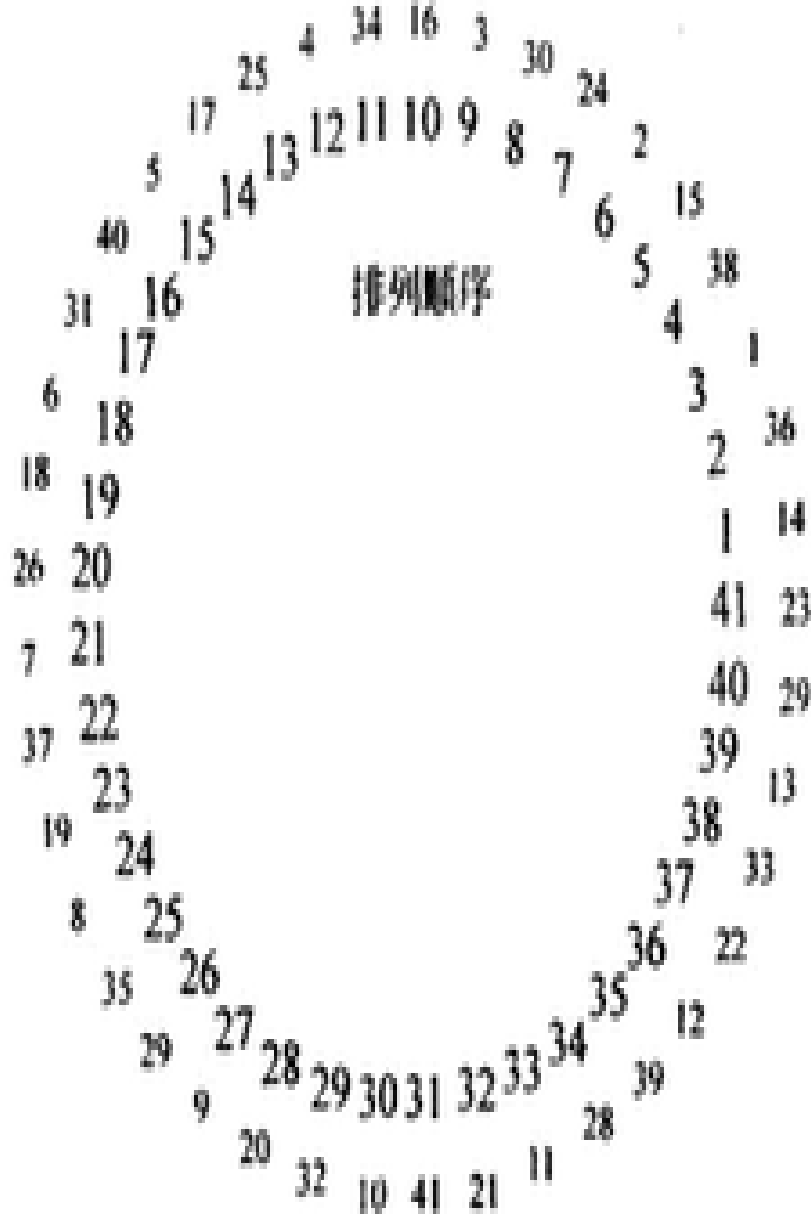
}



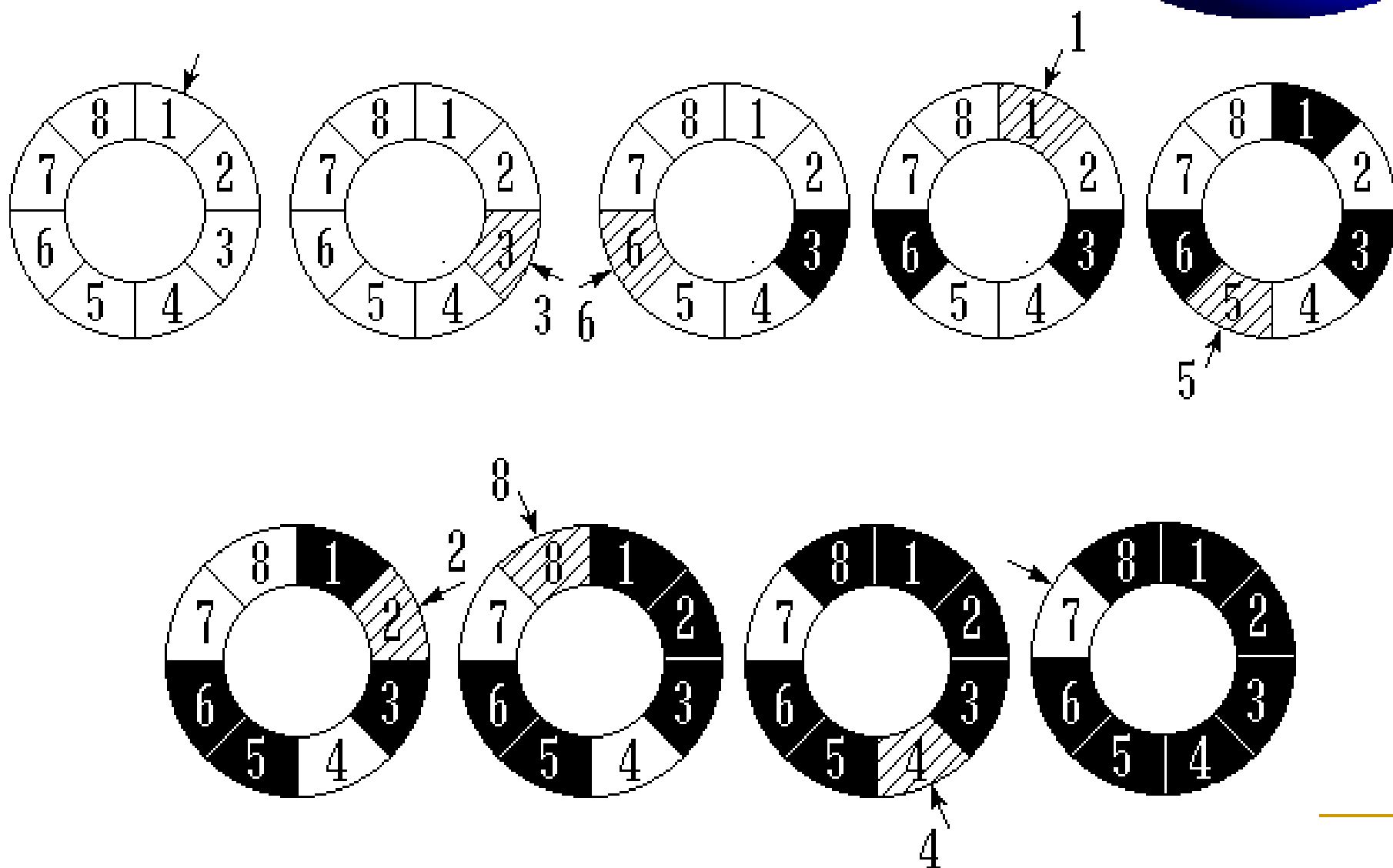
# 约瑟夫问题

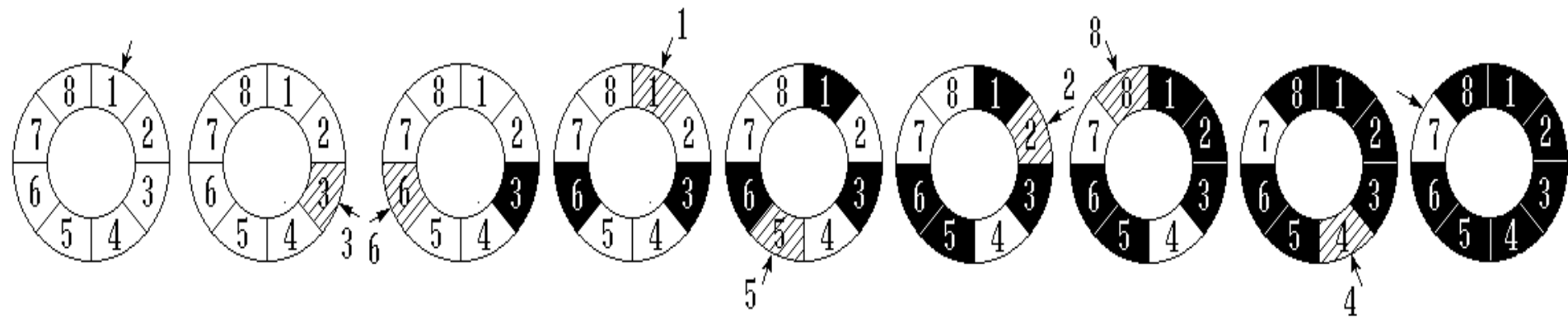
著名犹太历史学家 **Josephus**

- 在罗马人占领乔塔帕特后
- **39** 个犹太人与**Josephus**及他的朋友躲到一个洞中
- **39**个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式
- **41**个人排成一个圆圈，由第**1**个人开始报数，每报数到第**3**人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止
- 然而**Josephus** 和他的朋友并不想遵从，**Josephus**要他的朋友先假装遵从，他将朋友与自己安排在第**16**个与第**31**个位置，于是逃过了这场死亡游戏



■ 例如  $n = 8$   $m = 3$





## 约瑟夫问题的解法

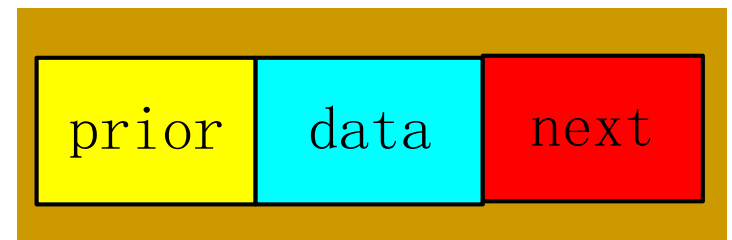
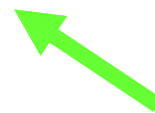
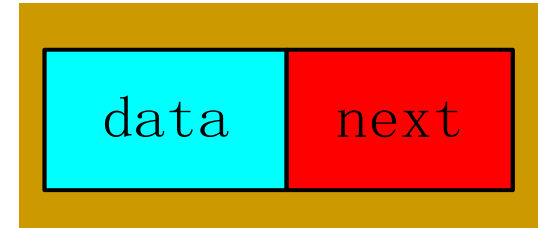
```

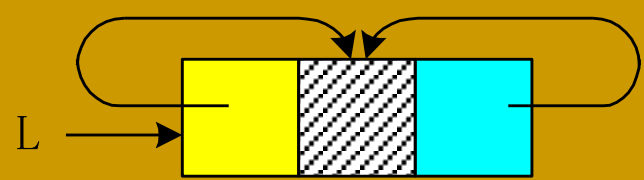
void Josephus ( int n, int m ) {
    Firster (); //检验指针指向第一个结点
    for ( int i = 0; i < n-1; i++ ) { //执行n-1次
        for ( int j = 0; j < m-1; j++ ) Next ();
        //循环m次使current指向被删除结点
        cout << “出列的人是” << GetElem_L () << endl;
        //出列人员的数据
        ListDelete (); //删去每一趟的第m结点
    }
}
    
```

## 2.5.4 双向链表



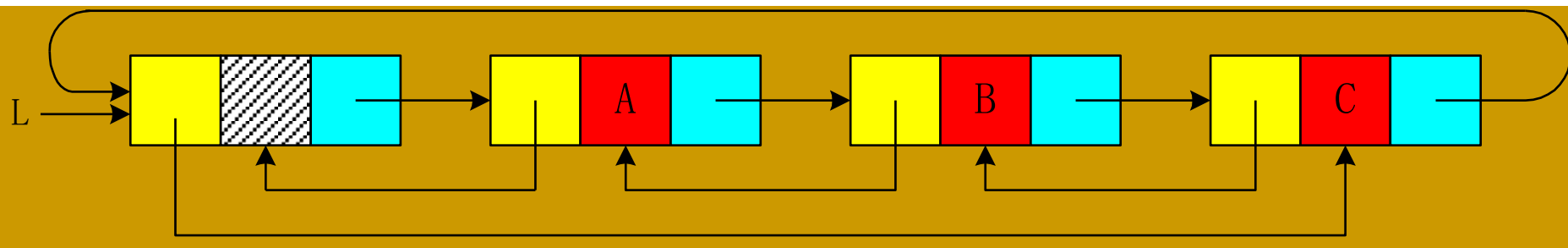
```
typedef struct DuLNode{  
    ElemType  data;  
    struct DuLNode *prior;  
    struct DuLNode *next;  
}DuLNode, *DuLinkList
```





(a) 空双向循环链表

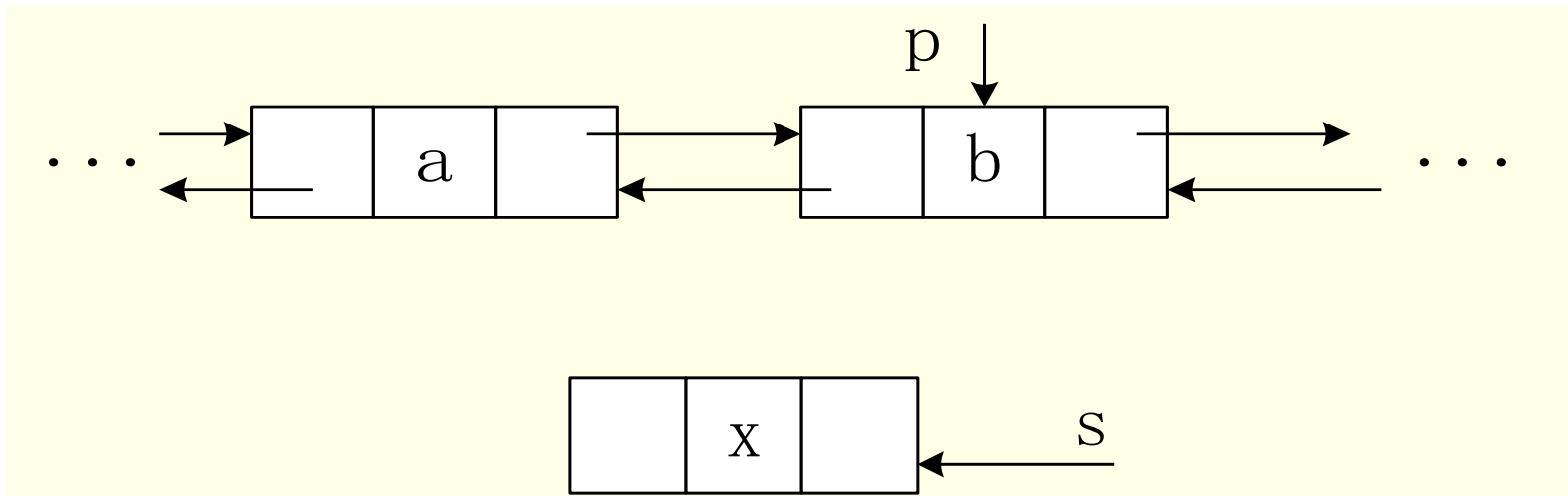
$L \rightarrow \text{next} = L$



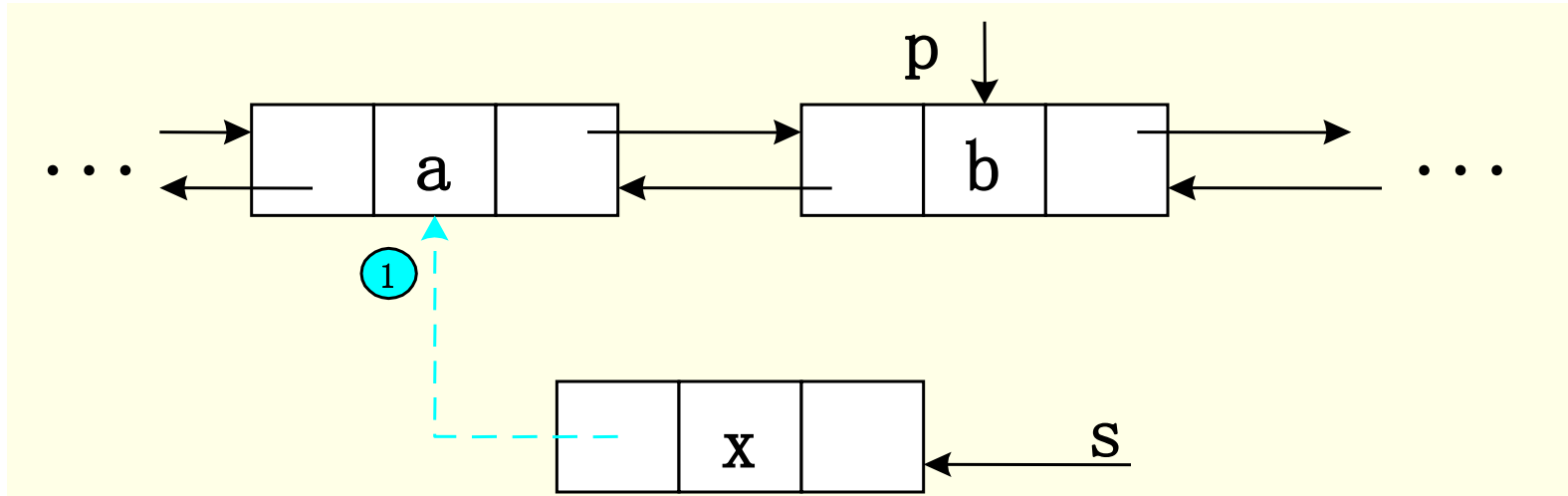
(b) 双向循环链表

$d \rightarrow \text{next} \rightarrow \text{prior} = d \rightarrow \text{prior} \rightarrow \text{next} = d$

# 双向链表的插入

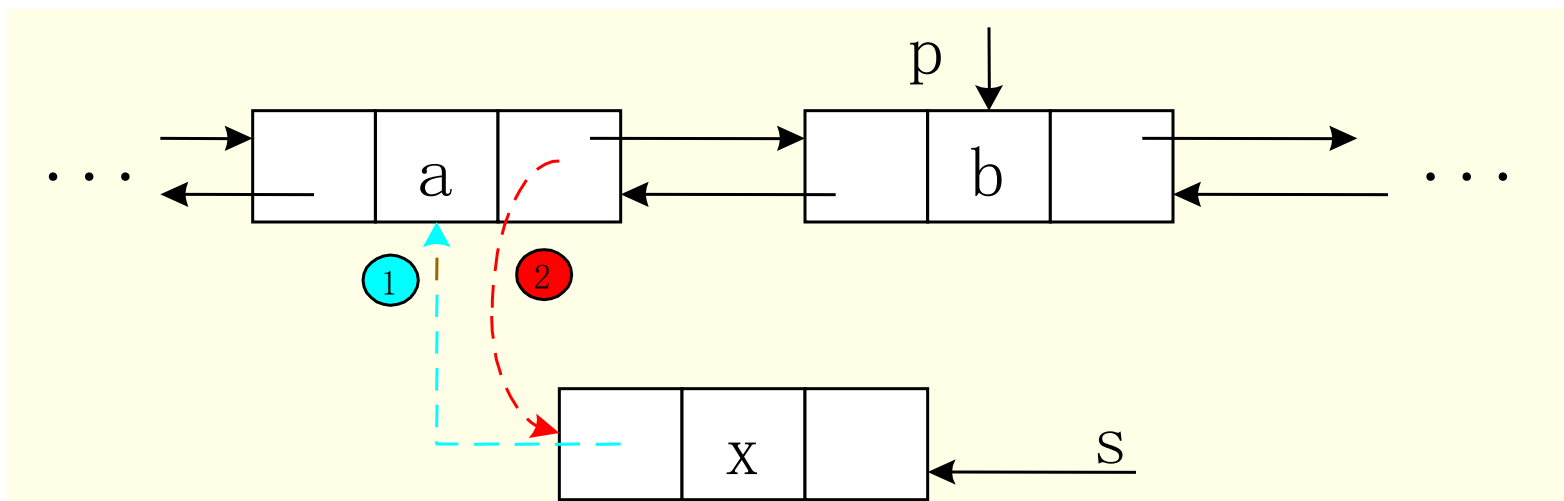


# 双向链表的插入



**1.  $s \rightarrow \text{prior} = p \rightarrow \text{prior};$**

# 双向链表的插入

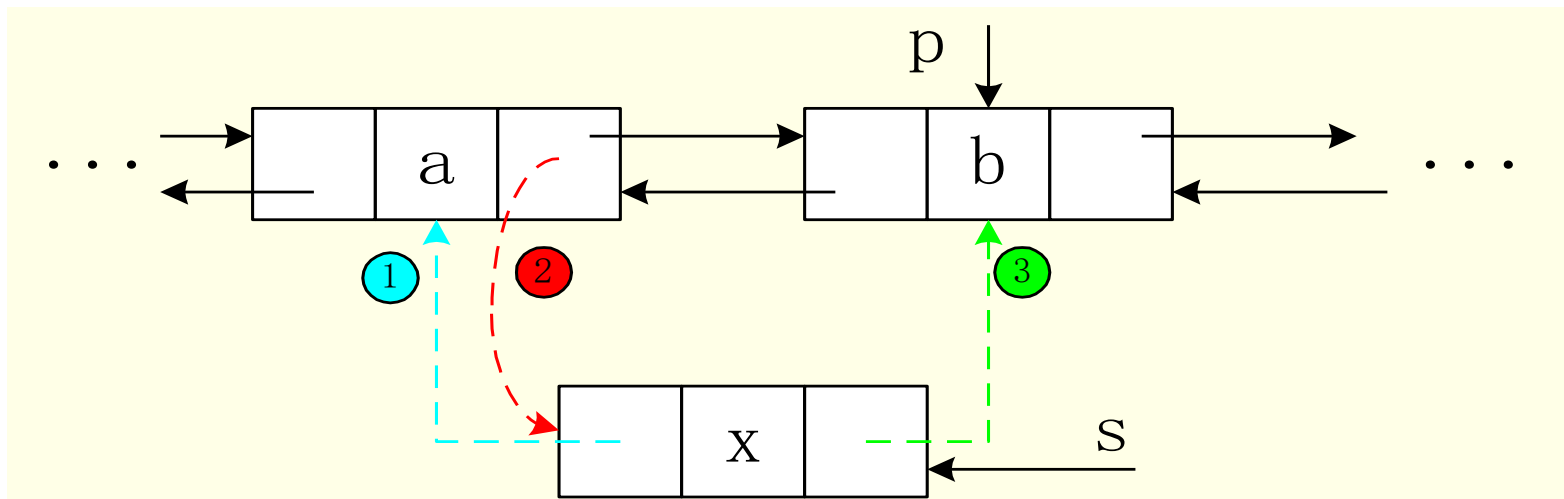


**1.  $s \rightarrow \text{prior} = p \rightarrow \text{prior};$**

**2.  $p \rightarrow \text{prior} \rightarrow \text{next} = s;$**

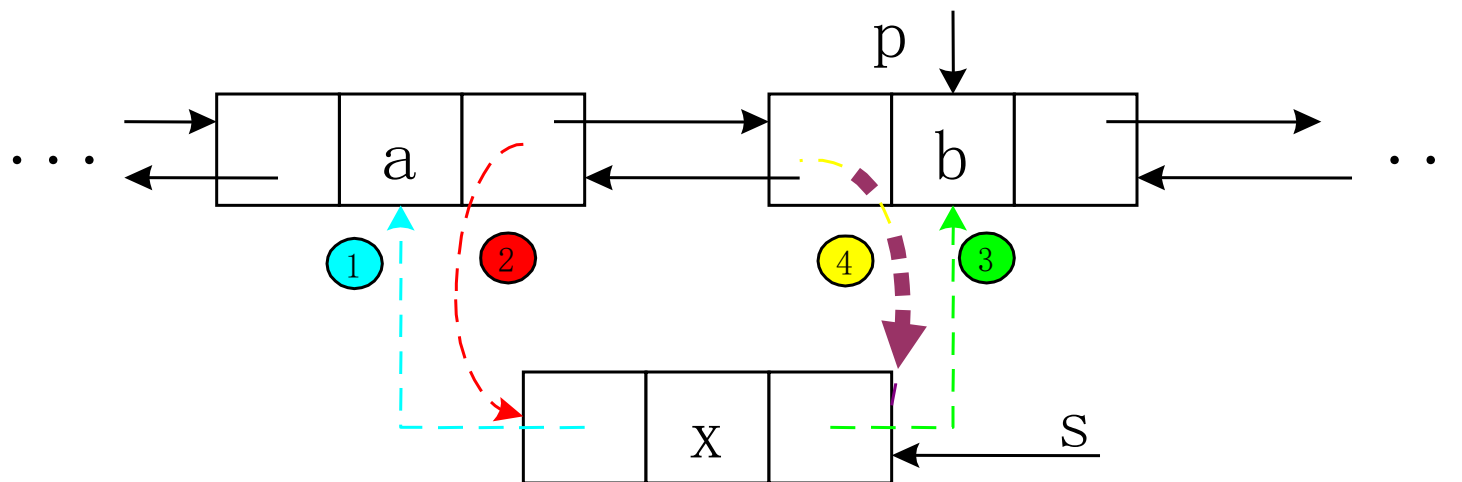


# 双向链表的插入



1.  $s \rightarrow \text{prior} = p \rightarrow \text{prior};$
2.  $p \rightarrow \text{prior} \rightarrow \text{next} = s;$
3.  $s \rightarrow \text{next} = p;$

# 双向链表的插入



1.  $s \rightarrow \text{prior} = p \rightarrow \text{prior};$

2.  $p \rightarrow \text{prior} \rightarrow \text{next} = s;$

3.  $s \rightarrow \text{next} = p;$

4.  $p \rightarrow \text{prior} = s;$

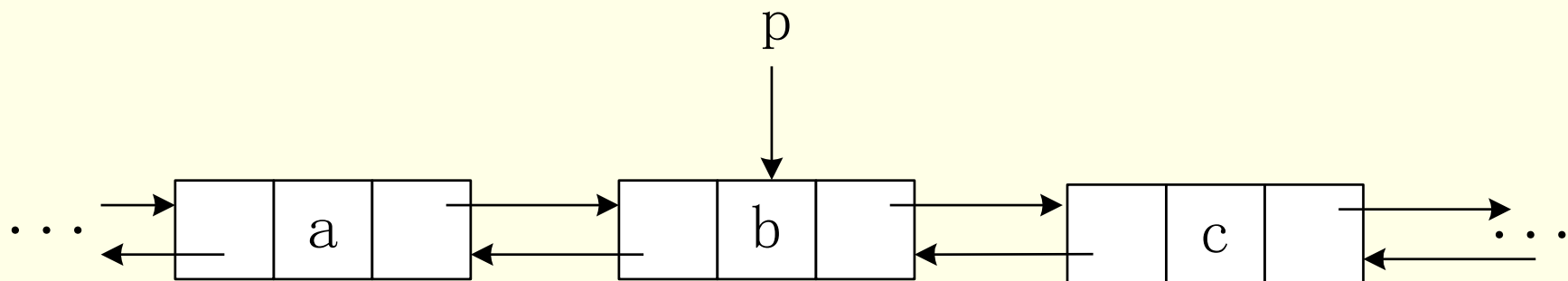
# 双向链表的插入

---

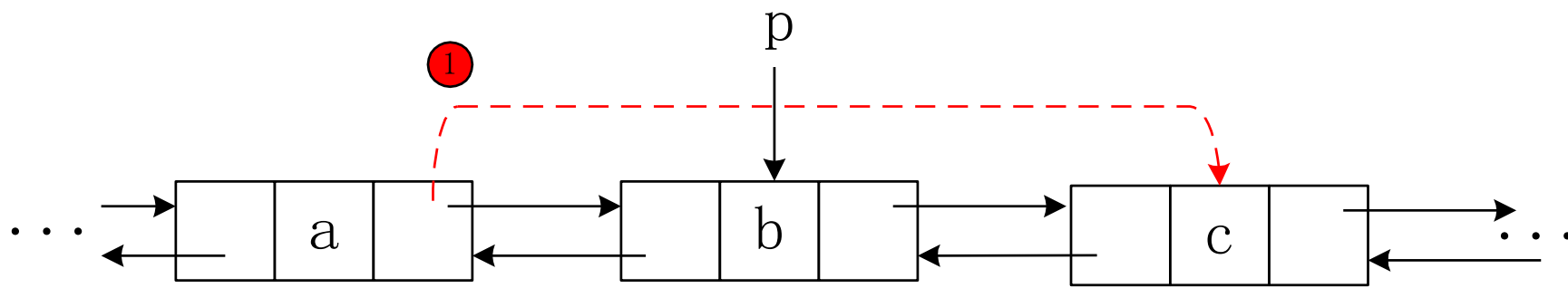
```
Status ListInsert_DuL(DuLinkList &L,int i,ElemType e){  
    if(!(p=GetElemP_DuL(L,i))) return ERROR;  
    s=new DuLNode;  
    s->data=e;  
    s->prior=p->prior;  
    p->prior->next=s;  
    s->next=p;  
    p->prior=s;  
    return OK;  
}
```

---

# 双向链表的删除

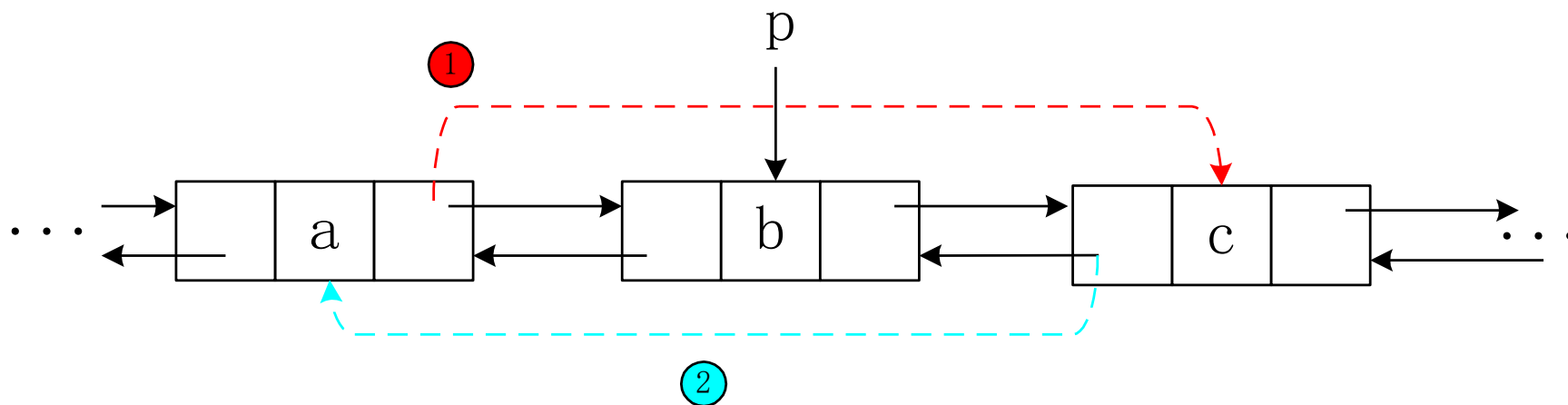


# 双向链表的删除



**1.  $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$**

# 双向链表的删除



1.  $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$

2.  $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior};$

# 双向链表的删除

---

```
Status ListDelete_DuL(DuLinkList &L,int i,ElemType &e){  
    if(!(p=GetElemP_DuL(L,i)))    return ERROR;  
    e=p->data;  
    p->prior->next=p->next;  
    p->next->prior=p->prior;  
    delete p;  
    return OK;  
}
```

---

# 链表的优缺点

---

## 优点

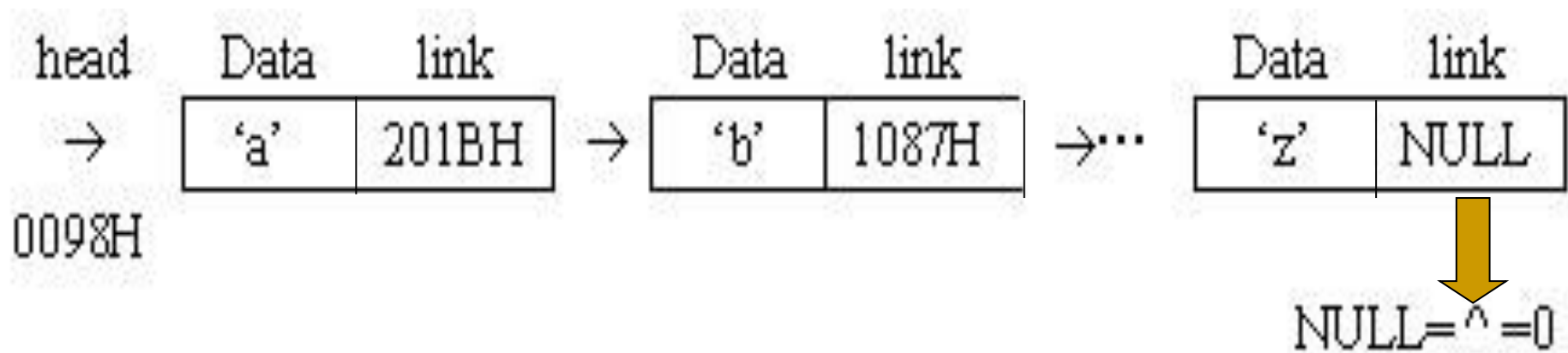
- 数据元素的个数可以自由扩充
  - 插入、删除等操作不必移动数据，只需修改链接指针，修改效率较高
-



# 链表的优缺点

## 缺点

- 存储密度小
- 存取效率不高，必须采用**顺序存取**，即存取数据元素时，只能按链表的顺序进行访问（**顺藤摸瓜**）



# 练习

1. 链表的每个结点中都恰好包含一个指针。
2. 顺序表结构适宜于进行顺序存取，而链表适宜于进行随机存取。
3. 顺序存储方式的优点是存储密度大，且插入、删除运算效率高。
4. 线性表若采用链式存储时，结点之间和结点内部的存储空间都是可以不连续的。
5. 线性表的每个结点只能是一个简单类型，而链表的每个结点可以是一个复杂类型



## 2.6 顺序表和链表的比较

| 存储结构<br>比较项目 |       | 顺序表   | 链表                              |
|--------------|-------|---|---------------------------------|
| 空间           | 存储空间  | 预先分配，会导致空间闲置或溢出现象                                     | 动态分配，不会出现存储空间闲置或溢出现象            |
|              | 存储密度  | 不用为表示结点间的逻辑关系而增加额外的存储开销，存储密度等于1                       | 需要借助指针来体现元素间的逻辑关系，存储密度小于1       |
| 时间           | 存取元素  | 随机存取，按位置访问元素的时间复杂度为 $O(1)$                            | 顺序存取，按位置访问元素时间复杂度为 $O(n)$       |
|              | 插入、删除 | 平均移动约表中一半元素，时间复杂度为 $O(n)$                             | 不需移动元素，确定插入、删除位置后，时间复杂度为 $O(1)$ |
| 适用情况         |       | ① 表长变化不大，且能事先确定变化的范围<br>② 很少进行插入或删除操作，经常按元素位置序号访问数据元素 | ① 长度变化较大<br>② 频繁进行插入或删除操作       |

## 2.7 线性表的应用



1

线性表的合并

2

有序表的合并

## 2.7.1 线性表的合并



问题描述:

假设利用两个线性表La和Lb分别表示两个集合A和B, 现要求一个新的集合

$$A=A \cup B$$

**La=(7, 5, 3, 11)**

**Lb=(2, 6, 3)**

**La=(7, 5, 3, 11, 2, 6)**

# 【算法步骤】

---

依次取出Lb 中的每个元素，执行以下操作：

在La中查找该元素

如果找不到，则将其插入La的最后

---

# 【算法描述】

---

```
void union(List &La, List Lb){
```

```
    La_len=ListLength(La);
```

```
    Lb_len=ListLength(Lb);
```

```
    for(i=1;i<=Lb_len;i++){
```

```
        GetElem(Lb,i,e);
```

```
        if(!LocateElem(La,e))
```

```
            ListInsert(&La,++La_len,e);
```

```
    }
```

```
}
```

$O(ListLength(LA) \times ListLength(LB))$

---

## 2.7.2 有序表的合并



### 问题描述:

已知线性表 **La** 和 **Lb** 中的数据元素按值 **非递减** 有序排列, 现要求将 **La** 和 **Lb** 归并为一个新的线性表 **Lc**, 且 **Lc** 中的数据元素仍按值 **非递减** 有序排列.

**La=(1, 7, 8)**

**Lb=(2, 4, 6, 8, 10, 11)**

**Lc=(1, 2, 4, 6, 7, 8, 8, 10, 11)**



## 【算法步骤】 一有序的顺序表合并

- (1) 创建一个空表 $L_c$
  - (2) 依次从  $L_a$  或  $L_b$  中“摘取”元素值较小的结点插入到  $L_c$  表的最后，直至其中一个表变空为止
  - (3) 继续将  $L_a$  或  $L_b$  其中一个表的剩余结点插入在  $L_c$  表的最后
-

# 【算法描述】 — 有序的顺序表合并

```
void MergeList_Sq(SqList LA, SqList LB, SqList &LC){
    pa=LA.elem; pb=LB.elem;           //指针pa和pb的初值分别指向两个表的第一个元素
    LC.length=LA.length+LB.length;    //新表长度为待合并两表的长度之和
    LC.elem=new ElemType[LC.length];  //为合并后的新表分配一个数组空间
    pc=LC.elem;                        //指针pc指向新表的第一个元素
    pa_last=LA.elem+LA.length-1;      //指针pa_last指向LA表的最后一个元素
    pb_last=LB.elem+LB.length-1;      //指针pb_last指向LB表的最后一个元素
    while(pa<=pa_last && pb<=pb_last){ //两个表都非空

        if(*pa<=*pb) *pc++=*pa++;    //依次“摘取”两表中值较小的结点
        else *pc++=*pb++;            }
    while(pa<=pa_last) *pc++=*pa++;  //LB表已到达表尾
    while(pb<=pb_last) *pc++=*pb++;  //LA表已到达表尾
} //MergeList_Sq
```

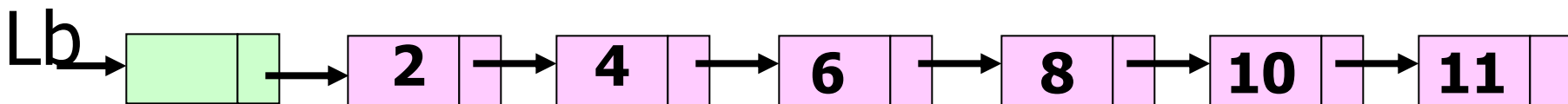
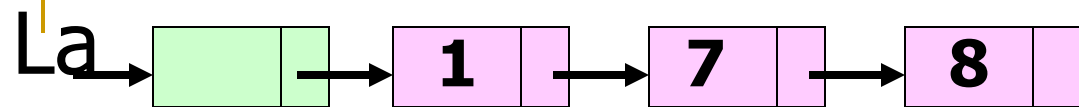
$$T(n) = O(ListLength(LA) + ListLength(LB))$$

$$S(n) = O(n)$$

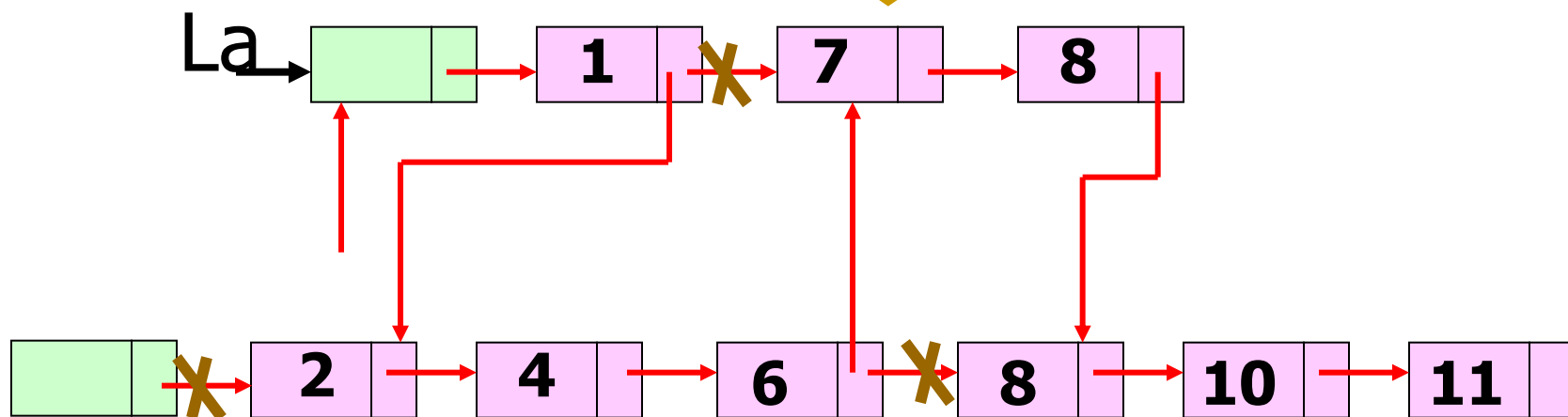
## 有序链表合并——重点掌握

- ✓ 将这两个有序链表合并成一个有序的单链表。
  - ✓ 要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。
  - ✓ 表中允许有重复的数据。
-

# 有序链表合并——重点掌握



合并后



## 【算法步骤】 一 有序的链表合并

(1)  $L_c$  指向  $L_a$

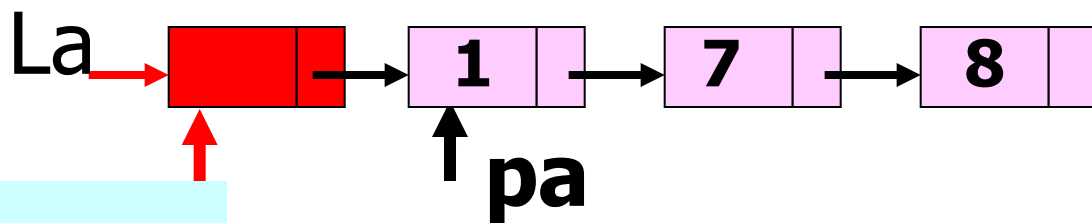
(2) 依次从  $L_a$  或  $L_b$  中“摘取”元素值较小的结点插入到  $L_c$  表的最后，直至其中一个表变空为止

(3) 继续将  $L_a$  或  $L_b$  其中一个表的剩余结点插入在  $L_c$  表的最后

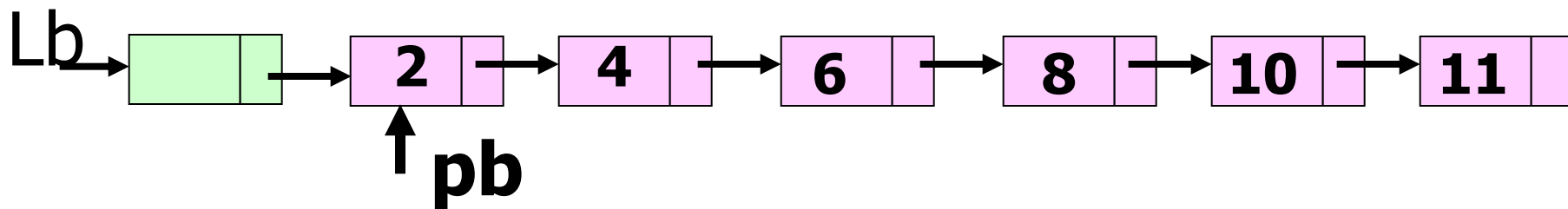
(4) 释放  $L_b$  表的表头结点

---

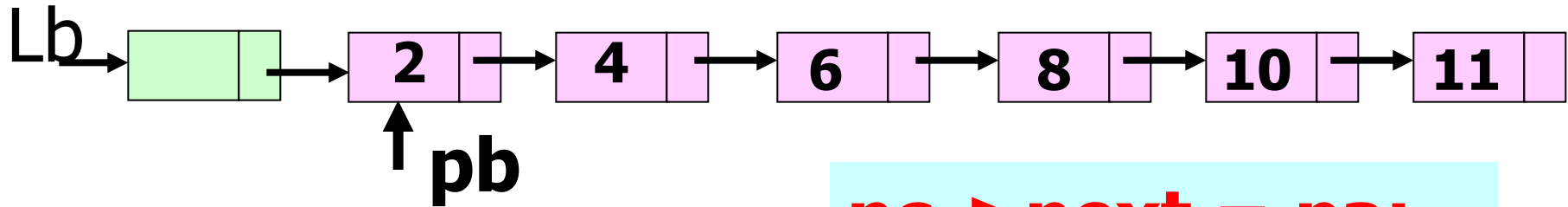
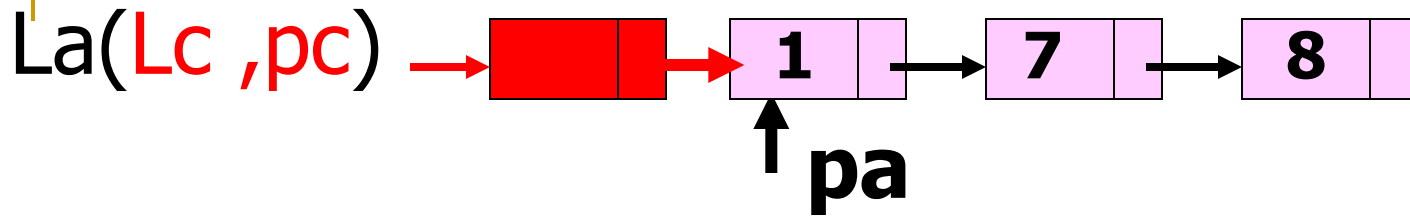
# 有序链表合并（初始化）



**Lc = pc**

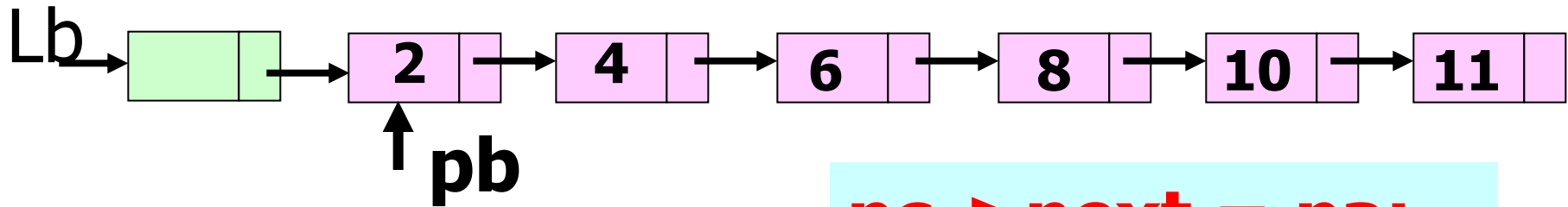
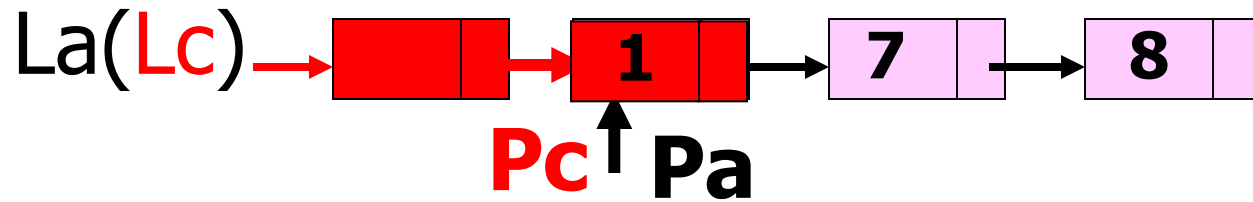


# 有序链表合并( $pa \rightarrow data \leq pb \rightarrow data$ )



$pc \rightarrow next = pa;$

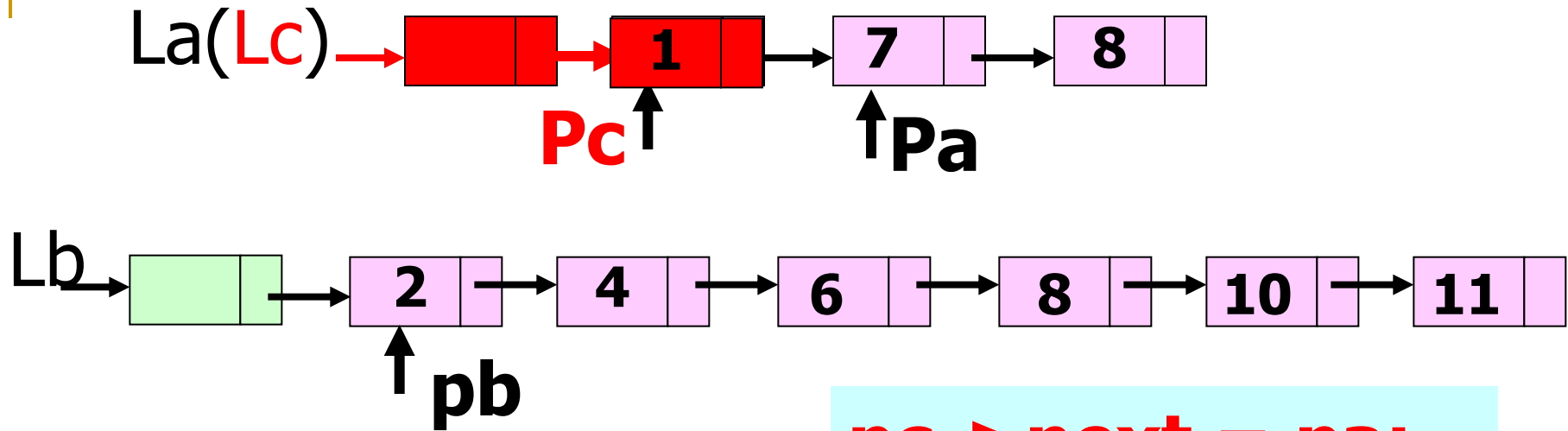
# 有序链表合并( $pa \rightarrow data \leq pb \rightarrow data$ )



```
pc->next = pa;  
pc = pa;
```

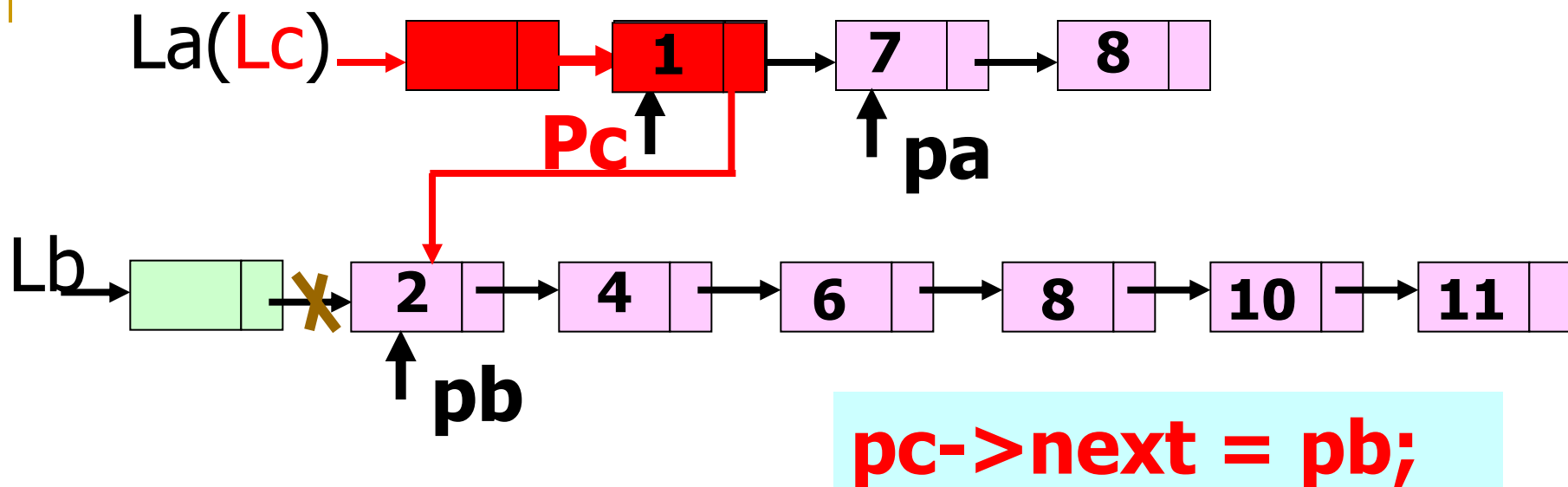


# 有序链表合并( $pa \rightarrow data \leq pb \rightarrow data$ )

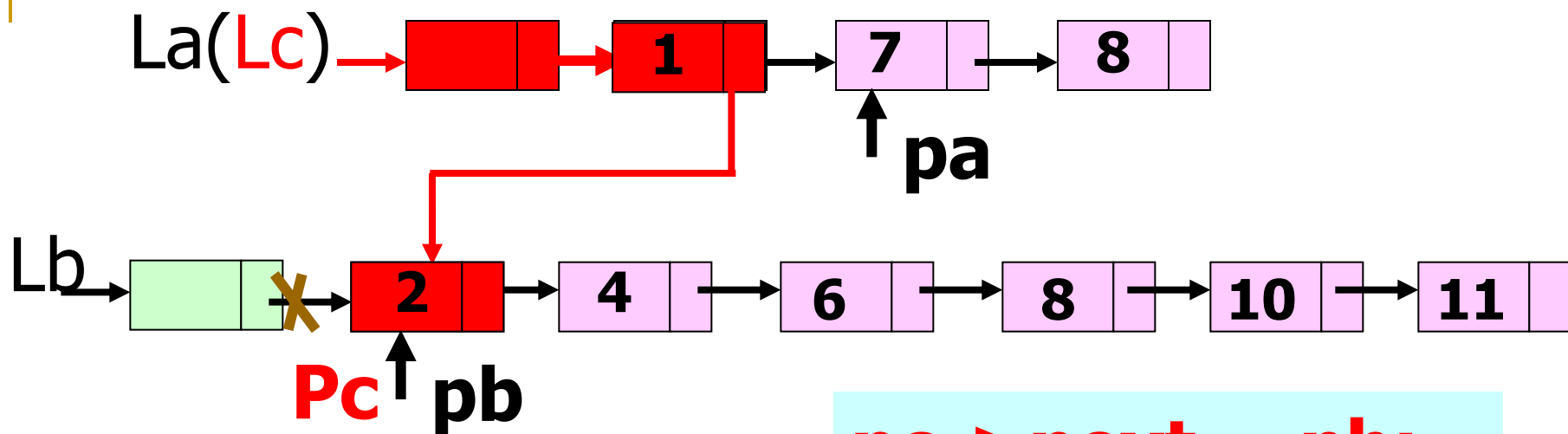


```
pc->next = pa;  
pc = pa;  
pa = pa->next;
```

# 有序链表合并( $pa \rightarrow data > pb \rightarrow data$ )

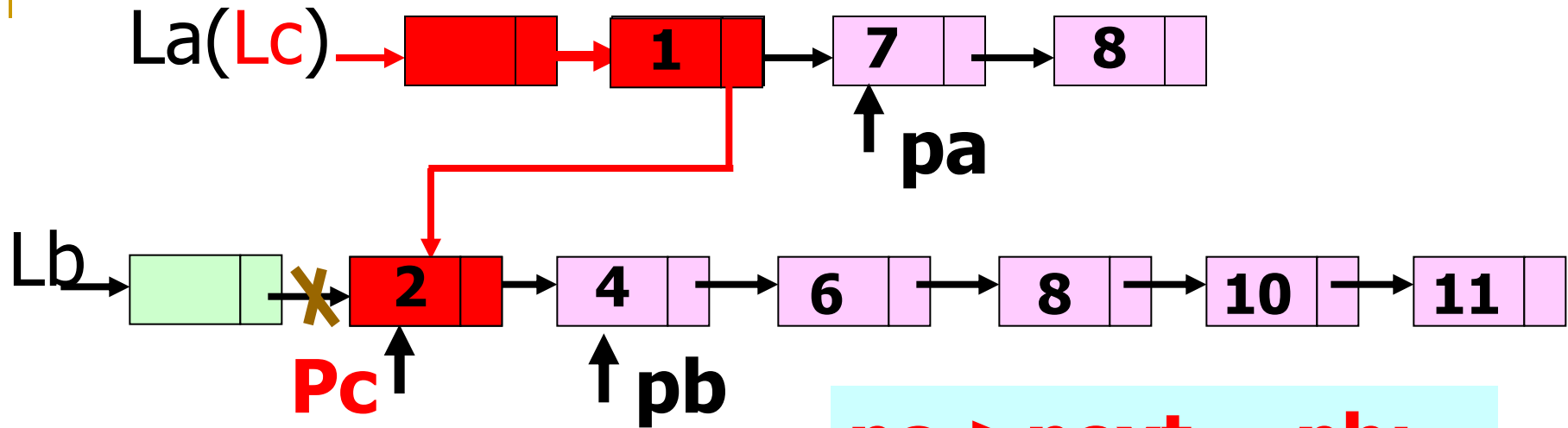


# 有序链表合并( $pa \rightarrow data > pb \rightarrow data$ )



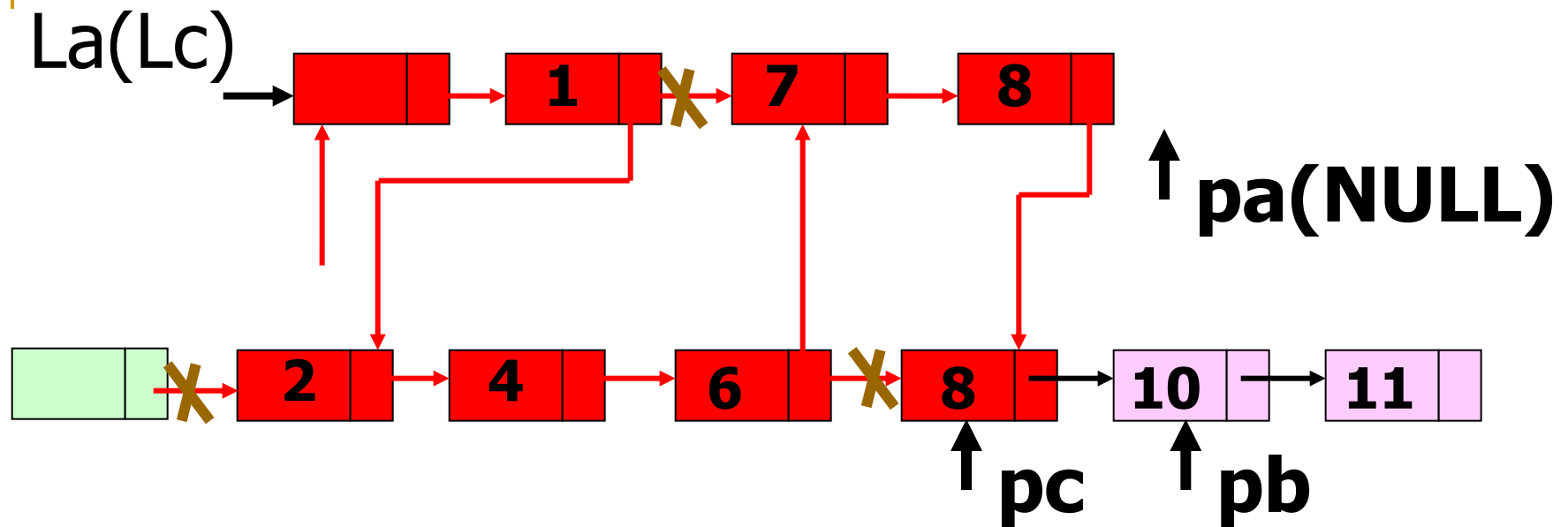
$pc \rightarrow next = pb;$   
 $pc = pb;$

# 有序链表合并( $pa \rightarrow data > pb \rightarrow data$ )



```
pc->next = pb;  
pc = pb;  
pb = pb->next;
```

# 有序链表合并

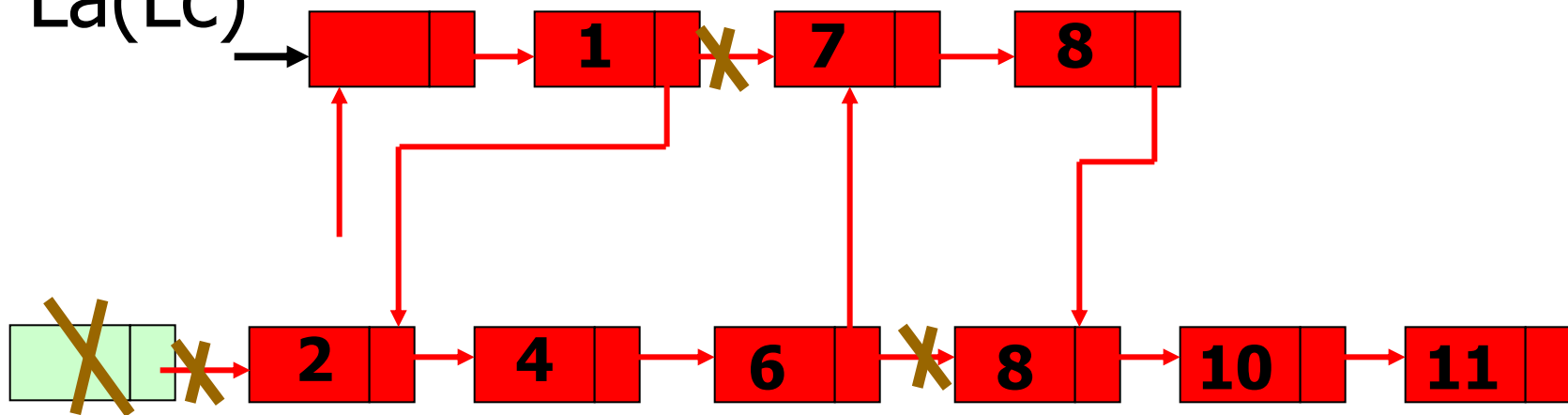


```
pc-> next=pa?pa:pb;
```

# 有序链表合并

合并后

La(Lc)



**delete Lb;**

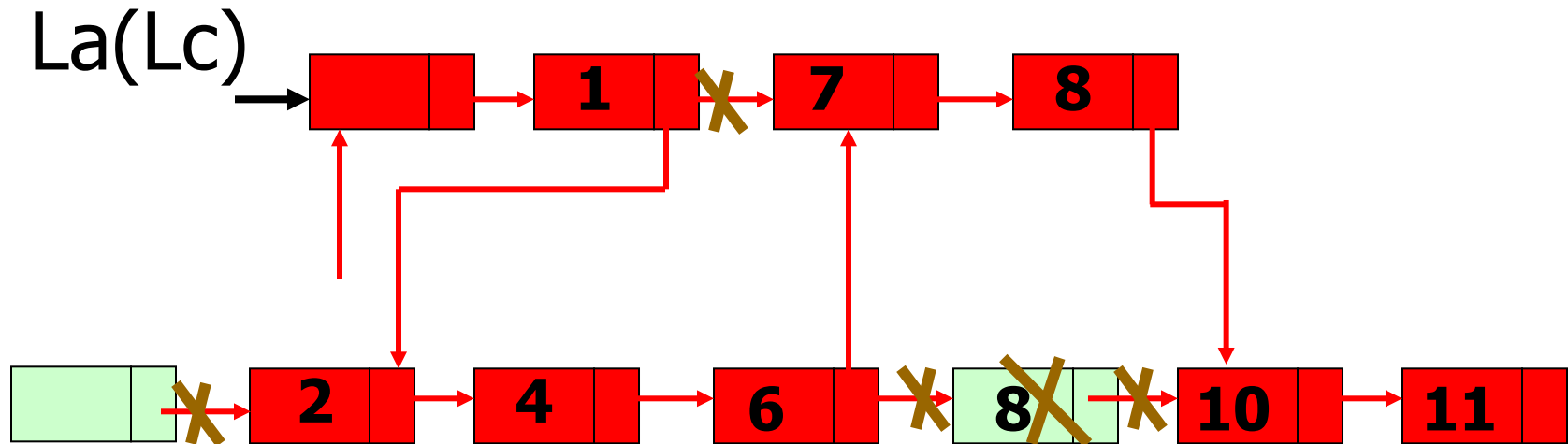
# 【算法描述】 — 有序的链表合并

```
void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc){  
    pa=La->next; pb=Lb->next;  
    pc=Lc=La;          //用La的头结点作为Lc的头结点  
    while(pa && pb){  
        if(pa->data<=pb->data){ pc->next=pa;pc=pa;pa=pa->next;}  
        else{pc->next=pb; pc=pb; pb=pb->next;}  
    }  
    pc->next=pa?pa:pb;  //插入剩余段  
    delete Lb;         //释放Lb的头结点}
```

**T(n)=**  $O(ListLength(LA) + ListLength(LB))$

**S(n)=**  $O(1)$

思考1：要求合并后的表无重复数据，如何实现？



提示：要单独考虑

**$pa \rightarrow data == pb \rightarrow data$**



思考2：将两个非递减的有序链表合并为一个非递增的有序链表，如何实现？

- ✓ 要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。
  - ✓ 表中允许有重复的数据。
-

# 【算法步骤】

(1)  $L_c$  指向  $L_a$

(2) 依次从  $L_a$  或  $L_b$  中“摘取”元素值较小的结点插入到  $L_c$  表的表头结点之后，直至其中一个表变空为止

(3) 继续将  $L_a$  或  $L_b$  其中一个表的剩余结点插入在  $L_c$  表的表头结点之后

(4) 释放  $L_b$  表的表头结点



## 2.8 案例分析与实现

### 案例2.1：一元多项式的运算

$$P(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$$

数组表示

(每一项的指数 $i$ 隐含在其系数 $p_i$ 的序号中)

| 指数<br>(下标 $i$ ) | 0  | 1 | 2  | 3 | 4 |
|-----------------|----|---|----|---|---|
| 系数 $p[i]$       | 10 | 5 | -4 | 3 | 2 |

$$R_n(x) = P_n(x) + Q_m(x)$$

easy

线性表 $R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \dots, p_m + q_m, p_{m+1}, \dots, p_n)$

## 案例2.2：稀疏多项式的运算

多项式**非零项**的数组表示

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


|            |   |   |   |    |
|------------|---|---|---|----|
| 下标i        | 0 | 1 | 2 | 3  |
| 系数<br>a[i] | 7 | 3 | 9 | 5  |
| 指数         | 0 | 1 | 8 | 17 |

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

|            |   |    |    |
|------------|---|----|----|
| 下标i        | 0 | 1  | 2  |
| 系数<br>b[i] | 8 | 22 | -9 |
| 指数         | 1 | 7  | 8  |

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + \dots + p_m x^{e_m}$$

线性表  $P = ((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$

- 
- 创建一个**新数组c**
  - 分别从头遍历比较a和b的每一项
    - ✓ **指数相同**，对应系数相加，若其和不为零，则在c中增加一个新项
    - ✓ **指数不相同**，则将指数较小的项复制到c中
  - 一个多项式已遍历**完毕**时，将另一个剩余项依次复制到c中即可

●顺序存储结构存在问题

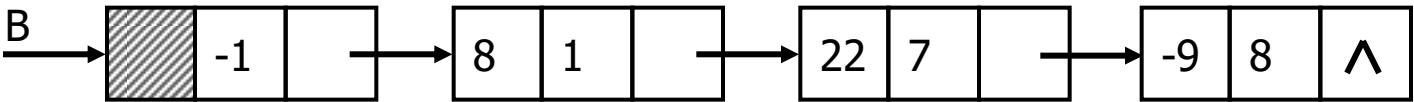
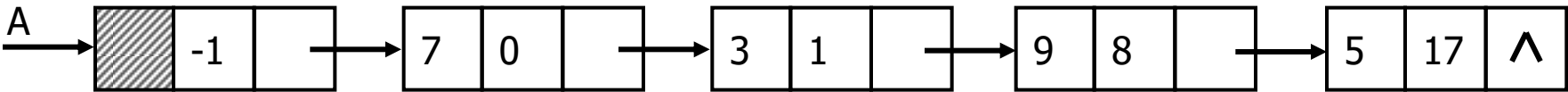
- ✓存储空间分配不灵活
- ✓运算的空间复杂度高



链式存储结构

```
typedef struct PNode
{
    float coef;//系数
    int  expn;    //指数
    struct PNode *next;  //指针域
}PNode,*Polynomial;
```

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



$B_8(x)=8x+22x^7-9x^8$

# 多项式创建--- 【算法步骤】

- ① 创建一个只有头结点的空链表。
- ② 根据多项式的项的个数 $n$ ，循环 $n$ 次执行以下操作：
  - 生成一个新结点\*s;
  - 输入多项式当前项的系数和指数赋给新结点\*s的数据域;
  - 设置一前驱指针pre，用于指向待找到的第一个大于输入项指数的结点的前驱，pre初值指向头结点;
  - 指针q初始化，指向首元结点;
  - 循链向下逐个比较链表中当前结点与输入项指数，找到第一个大于输入项指数的结点\*q;
  - 将输入项结点\*s插入到结点\*q之前。

# 多项式创建--- 【算法描述】

```
void CreatePolyn(Polynomial &P,int n)
```

```
{//输入m项的系数和指数，建立表示多项式的有序链表P
```

```
  P=new PNode;
```

```
  P->next=NULL;
```

```
//先建立一个带头结点的单链表
```

```
  for(i=1;i<=n;++i)
```

```
//依次输入n个非零项
```

```
  {
```

```
    s=new PNode;
```

```
//生成新结点
```

```
    cin>>s->coef>>s->expn;
```

```
//输入系数和指数
```

```
    pre=P;
```

```
//pre用于保存q的前驱，初值为头结点
```

```
    q=P->next;
```

```
//q初始化，指向首元结点
```

```
    while(q&&q->expn<s->expn)
```

```
//找到第一个大于输入项指数的项*q
```

```
    {
```

```
      pre=q;
```

```
      q=q->next;
```

```
    }
```

```
//while
```

```
    s->next=q;
```

```
//将输入项s插入到q和其前驱结点pre之间
```

```
    pre->next=s;
```

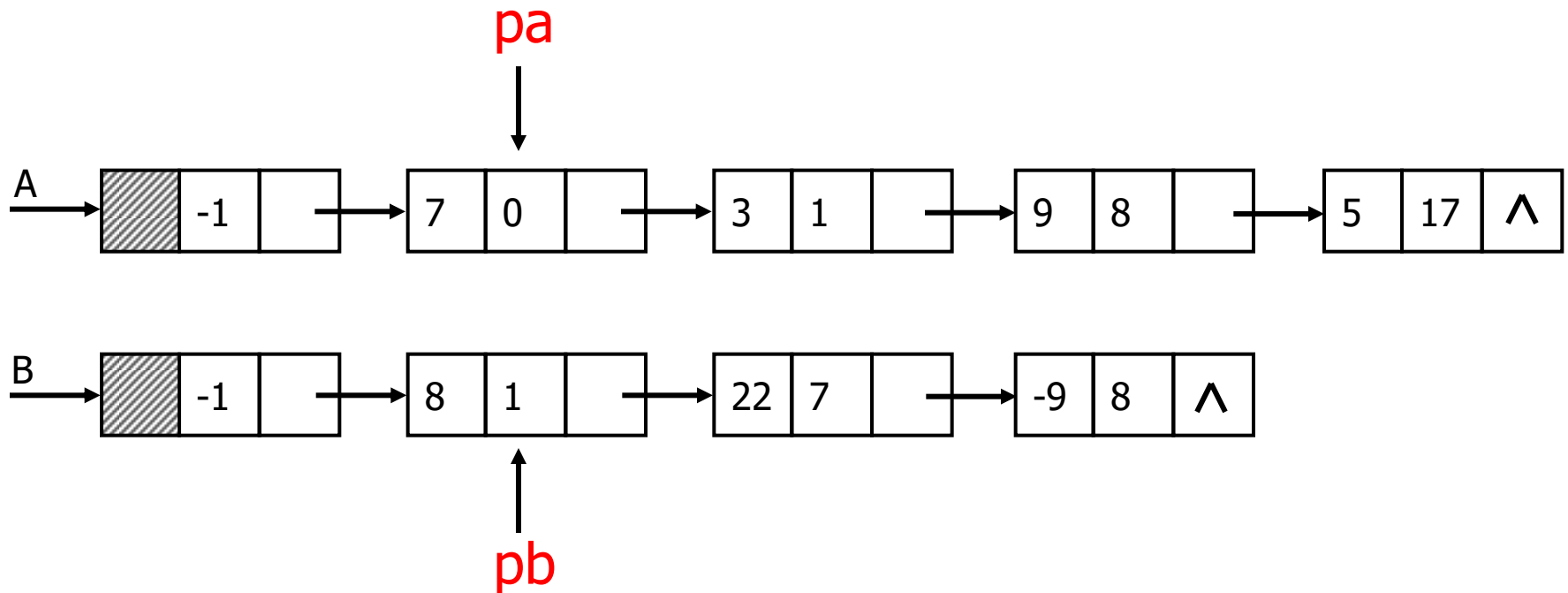
```
  }
```

```
//for
```

```
}
```

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

$$B_8(x) = 8x + 22x^7 - 9x^8$$

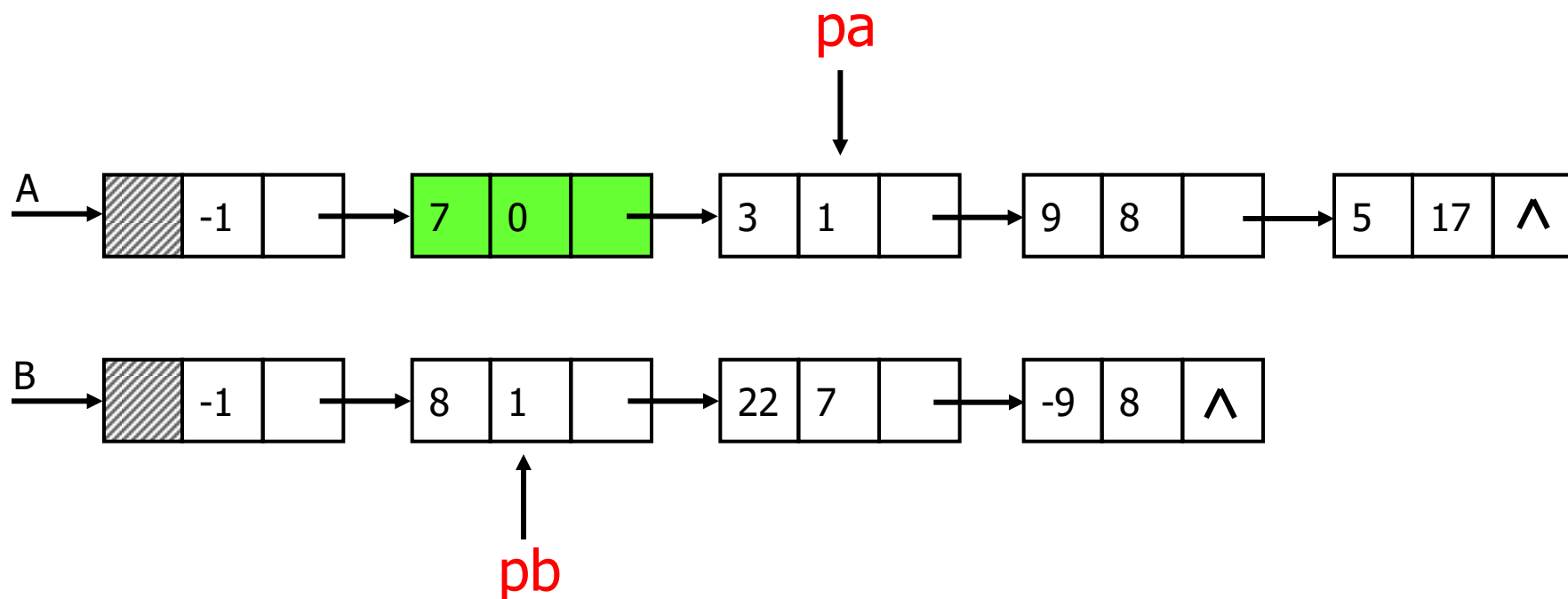




# 多项式相加

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

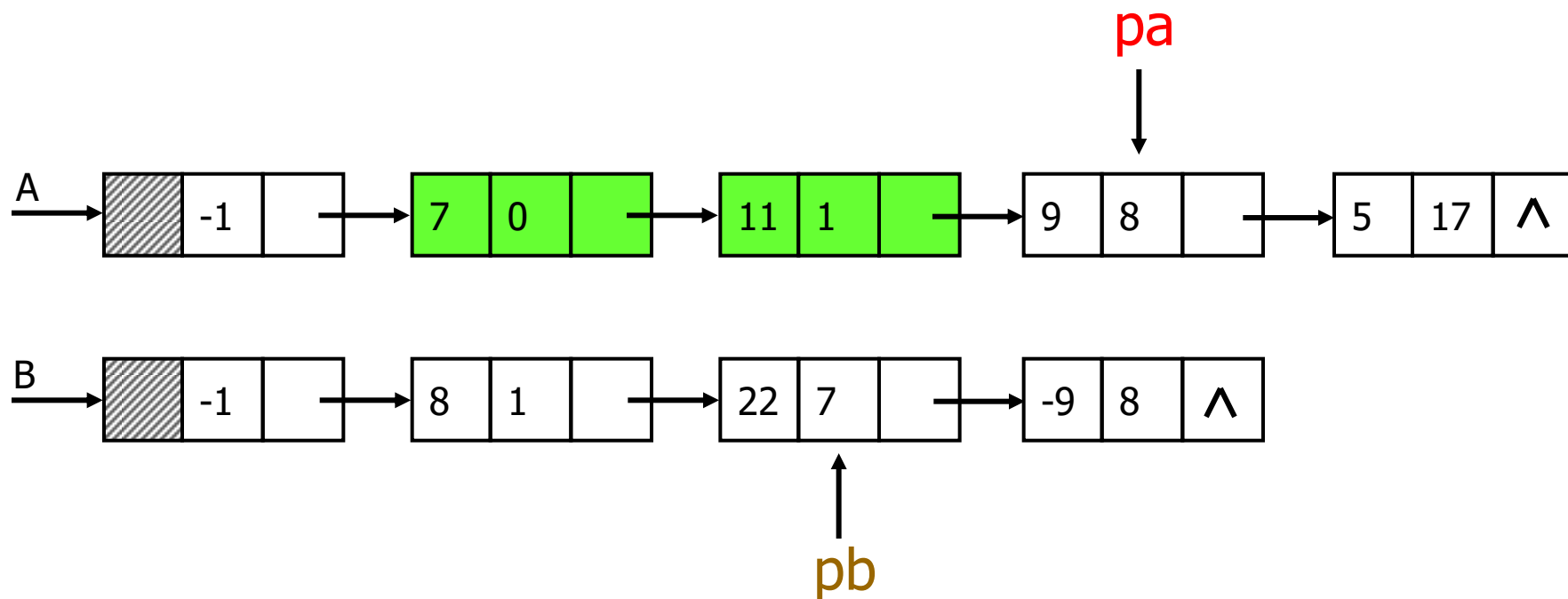
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

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

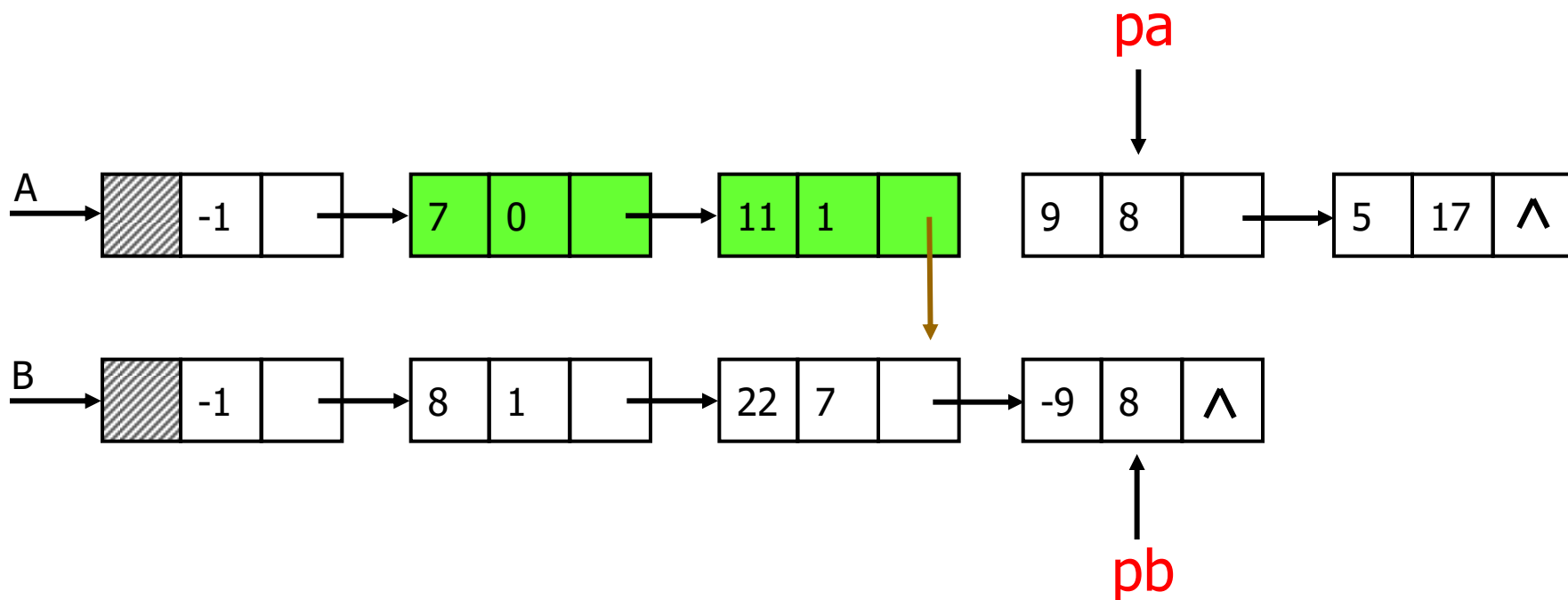
$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加

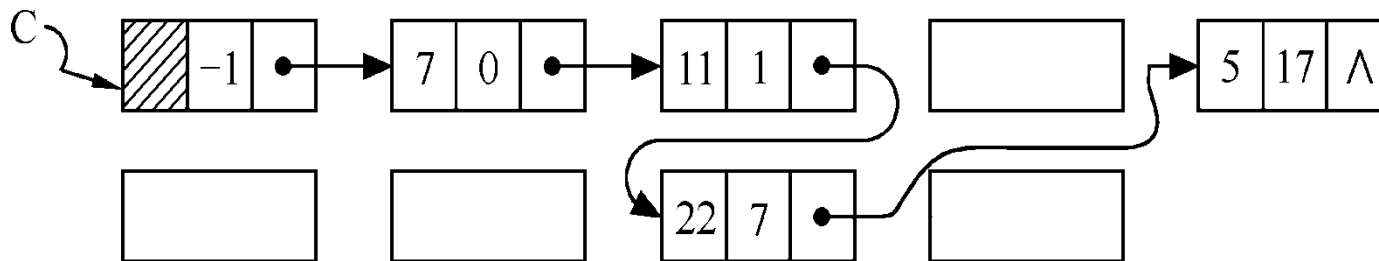
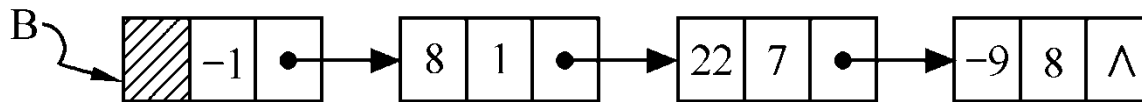
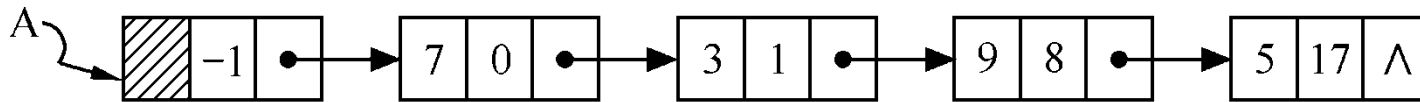
$$A_{17}(x) = 7 + 3x + 9x^8 + 5x^{17}$$

$$B_8(x) = 8x + 22x^7 - 9x^8$$



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

$$B_8(x) = 8x + 22x^7 - 9x^8$$



# 多项式相加--- 【算法步骤】

- ① 指针p1和p2初始化，分别指向Pa和Pb的首元结点。
- ② p3指向和多项式的当前结点，初值为Pa的头结点。
- ③ 当指针p1和p2均未到达相应表尾时，则循环比较p1和p2所指结点对应的指数值（ $p1 \rightarrow \text{expn}$ 与 $p2 \rightarrow \text{expn}$ ），有下列3种情况：
  - 当 $p1 \rightarrow \text{expn}$ 等于 $p2 \rightarrow \text{expn}$ 时，则将两个结点中的系数相加，若和不为零，则修改p1所指结点的系数值，同时删除p2所指结点，若和为零，则删除p1和p2所指结点；
  - 当 $p1 \rightarrow \text{expn}$ 小于 $p2 \rightarrow \text{expn}$ 时，则应摘取p1所指结点插入到“和多项式”链表中去；
  - 当 $p1 \rightarrow \text{expn}$ 大于 $p2 \rightarrow \text{expn}$ 时，则应摘取p2所指结点插入到“和多项式”链表中去。
- ④ 将非空多项式的剩余段插入到p3所指结点之后。
- ⑤ 释放Pb的头结点。

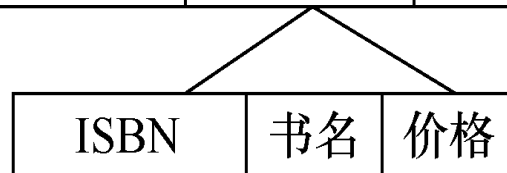
# 案例2.3：图书信息管理系统

| book.txt - 记事本                |               |    |
|-------------------------------|---------------|----|
| 文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H) |               |    |
| ISBN                          | 书名            |    |
| 9787302257646                 | 程序设计基础        |    |
| 9787302219972                 | 单片机技术及应       |    |
| 9787302203513                 | 编译原理          |    |
| 9787811234923                 | 汇编语言程序设计教程    | 21 |
| 9787512100831                 | 计算机操作系统       | 17 |
| 9787302265436                 | 计算机导论实验指导     | 18 |
| 9787302180630                 | 实用数据结构        | 29 |
| 9787302225065                 | 数据结构（C语言版）    | 38 |
| 9787302171676                 | C#面向对象程序设计    | 39 |
| 9787302250692                 | C语言程序设计       | 42 |
| 9787302150664                 | 数据库原理         | 35 |
| 9787302260806                 | Java编程与实践     | 56 |
| 9787302252887                 | Java程序设计与应用教程 | 39 |
| 9787302198505                 | 嵌入式操作系统及编程    | 25 |
| 9787302169666                 | 软件测试          | 24 |
| 9787811231557                 | Eclipse基础与应用  | 35 |

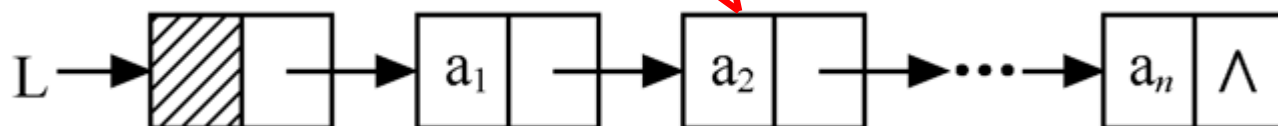
## 实验1---基于线性表的图书信息管理系统

## 图书顺序表

|         |         |         |     |                     |     |  |
|---------|---------|---------|-----|---------------------|-----|--|
| elem[0] | elem[1] | elem[2] | ... | elem[length-1]      | 空闲区 |  |
| $a_1$   | $a_2$   | $a_3$   | ... | $a_{\text{length}}$ |     |  |



## 图书链表



```
struct Book
{
    char id[20];//ISBN
    char name[50];//书名
    int price;//定价
};
```

```
typedef struct
{ //顺序表
    Book *elem;
    int length;
} SqList;
```

```
typedef struct LNode
{ //链表
    Book data;
    struct LNode *next;
} LNode, *LinkList;
```



## 小结

- 1、掌握线性表的逻辑结构特性是数据元素之间存在着线性关系，在计算机中表示这种关系的两类不同的存储结构是顺序存储结构（**顺序表**）和链式存储结构（**链表**）。
- 2、熟练掌握这两类存储结构的描述方法，掌握链表中的**头结点、头指针和首元结点**的区别及**循环链表、双向链表**的特点等。

## 小结

- 3、熟练掌握顺序表的查找、插入和删除算法
- 4、熟练掌握链表的查找、插入和删除算法
- 5、能够从时间和空间复杂度的角度比较两种存储结构的不同特点及其适用场合

|      |       | 顺 序 表  | 链 表  |
|------|-------|--|--|
| 空 间  | 存储空间  | 预先分配，会导致空间闲置或溢出现象  | 动态分配，不会出现闲置或溢出现象                             |
|      | 存储密度  | 不用为表示结点间的逻辑关系而增加额外的存储开销，存储密度等于1  | 需要借助指针来体现元素间的逻辑关系，存储密度小于1                    |
| 时 间  | 存取元素  | 随机存取，时间复杂度为 $O(1)$   | 顺序存取，时间复杂度为 $O(n)$                           |
|      | 插入、删除 | 平均移动约表中一半元素，时间复杂度为 $O(n)$  | 不需移动元素，确定插入、删除位置后，时间复杂度为 $O(1)$              |
| 适用情况 |       | <div>① 表长变化不大，且能事先确定变化的范围</div> <div>② 很少进行插入或删除操作，经常按元素序号访问数据元素</div> | <div>① 长度变化较大</div> <div>② 频繁进行插入或删除操作</div> |

---

# 谢谢大家！

