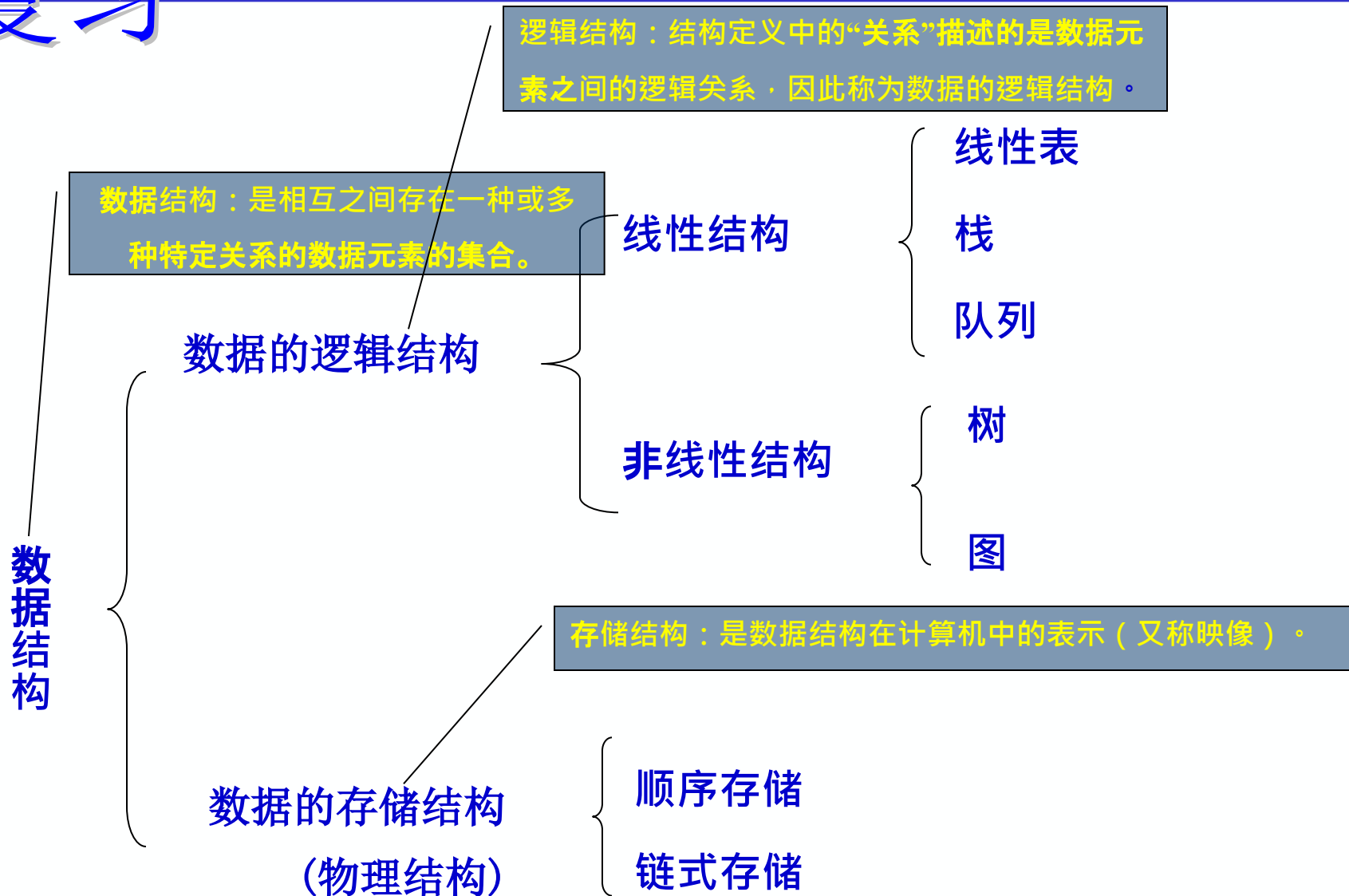


# 算法与数据结构

主讲教师：陈娜

联系方式：chenna@stdu.edu.cn

# 复习



近几周  
上课  
内容

- 第2章 线性表
- 第3章 栈和队列
- 第4章 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**

## 第2章 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、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**

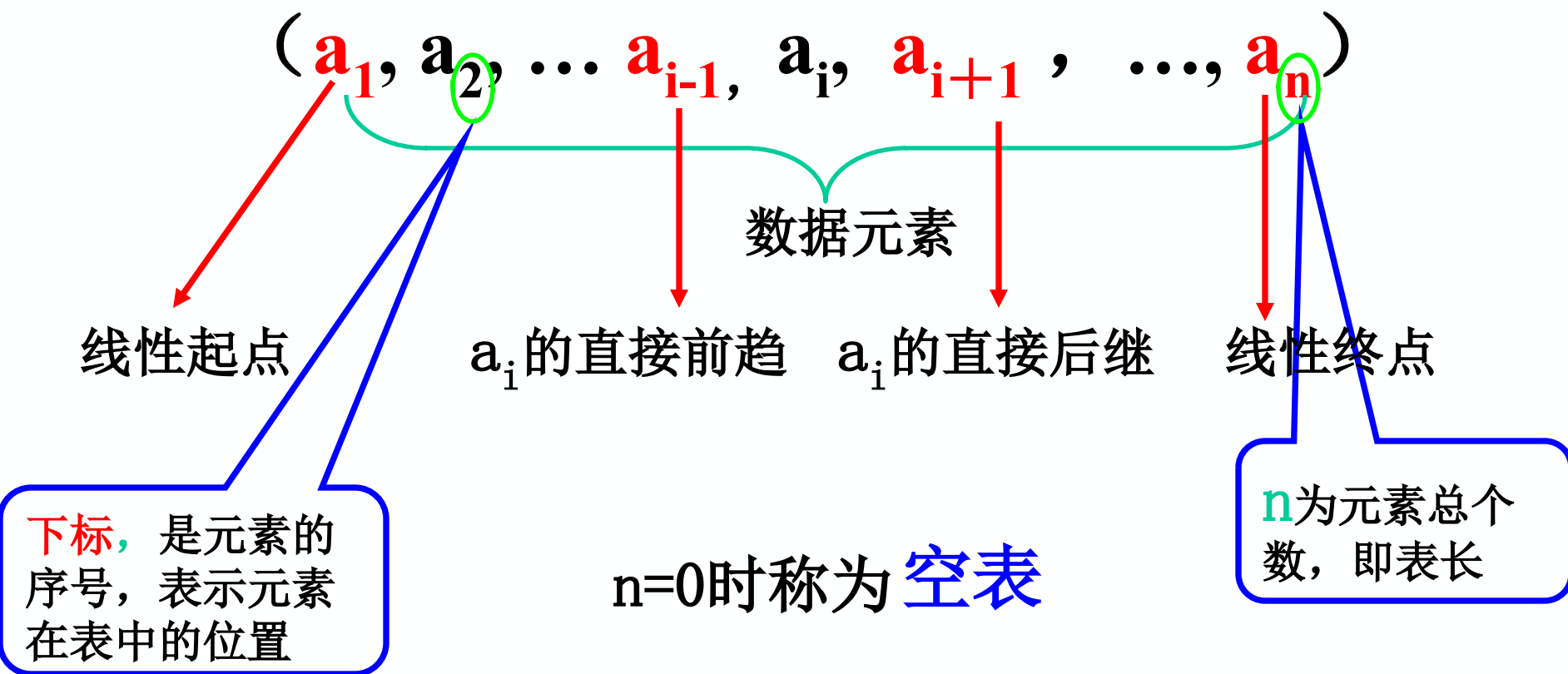
**2.3 线性表的链式表示和实现**

**2.4 线性表的应用**



## 2.1 线性表的类型定义

线性表的定义：用数据元素的有限序列表示



## 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	04级计算机1班
041810260	何仕鹏	男	20	04级计算机2班
041810284	王 爽	女	19	04级计算机3班
041810360	王亚武	男	18	04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



抽象数据类型**线性表**的定义如下：

**ADT List {**

**数据对象：**

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

{称 **n** 为线性表的**表长**;

称 **n=0** 时的线性表为**空表**。}

**数据关系：**

$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \}$

{设线性表为  $(a_1, a_2, \dots, a_i, \dots, a_n)$ ,

称  $i$  为  $a_i$  在线性表中的**位序**。}

基本操作:

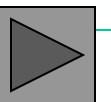
结构初始化操作

结构销毁操作

引用型操作

加工型操作

} ADT List



# 初始化操作

**InitList( &L )**

**操作结果:**

构造一个空的线性表L。

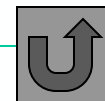


# 结构销毁操作

**DestroyList( &L )**

**初始条件：**线性表 L 已存在。

**操作结果：**销毁线性表 L。



## 引用型操作:

**ListEmpty( L )**

**ListLength( L )**

**PriorElem( L, cur\_e, &pre\_e )**

**NextElem( L, cur\_e, &next\_e )**

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

**LocateElem( L, e)**



# ListEmpty( L )

## ( 线性表判空 )

初始条件：线性表**L**已存在。

操作结果：若**L**为空表，则返回  
TRUE，否则FALSE。



# ListLength( L )

(求线性表的长度)

初始条件：线性表**L**已存在。

操作结果：返回**L**中元素个数。



# PriorElem( L, cur\_e, &pre\_e )

(求数据元素的前驱)

**初始条件：**线性表L已存在。

**操作结果：**若cur\_e是L的元素，但不是第一个，则用pre\_e 返回它的前驱，否则操作失败，pre\_e 无定义。





# NextElem( L, cur\_e, &next\_e )

( 求数据元素的后继 )

**初始条件：**线性表L已存在。

**操作结果：**若cur\_e是L的元素，但不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。



# GetElem( L, i, &e )

(求线性表中某个数据元素)

初始条件: 线性表**L**已存在,  
且  $1 \leq i \leq \text{LengthList}(\text{L})$ 。

操作结果: 用 **e** 返回**L**中第 **i** 个元素的值。



# LocateElem( L, e)

(定位函数)

**初始条件：**线性表**L**已存在，e为给定值

**操作结果：**返回**L**中**第1个**与**e相等**的元素的位序。  
若这样的元素不存在，  
则返回值为0。



# 加工型操作

**ClearList( &L )**

**ListInsert( &L, i, e )**

**ListDelete(&L, i, &e)**



# ClearList( &L )

## (线性表置空)

初始条件： 线性表**L**已存在。

操作结果： 将**L**重置为空表。



# ListInsert( &L, i, e )

(插入数据元素)

初始条件：线性表**L**已存在，  
且  $1 \leq i \leq \text{LengthList}(\text{L}) + 1$ 。

操作结果：在**L**的第**i**个元素之前插入新的元素**e**，**L**的长度增1。



# ListDelete(&L, i, &e)

(删除数据元素)

**初始条件：**线性表L已存在且非空，  
 $1 \leq i \leq \text{LengthList}(L)$ 。

**操作结果：**删除L的第i个元素，并用e返回其值，L的长度减1。





## 2.2 线性表的顺序表示和实现

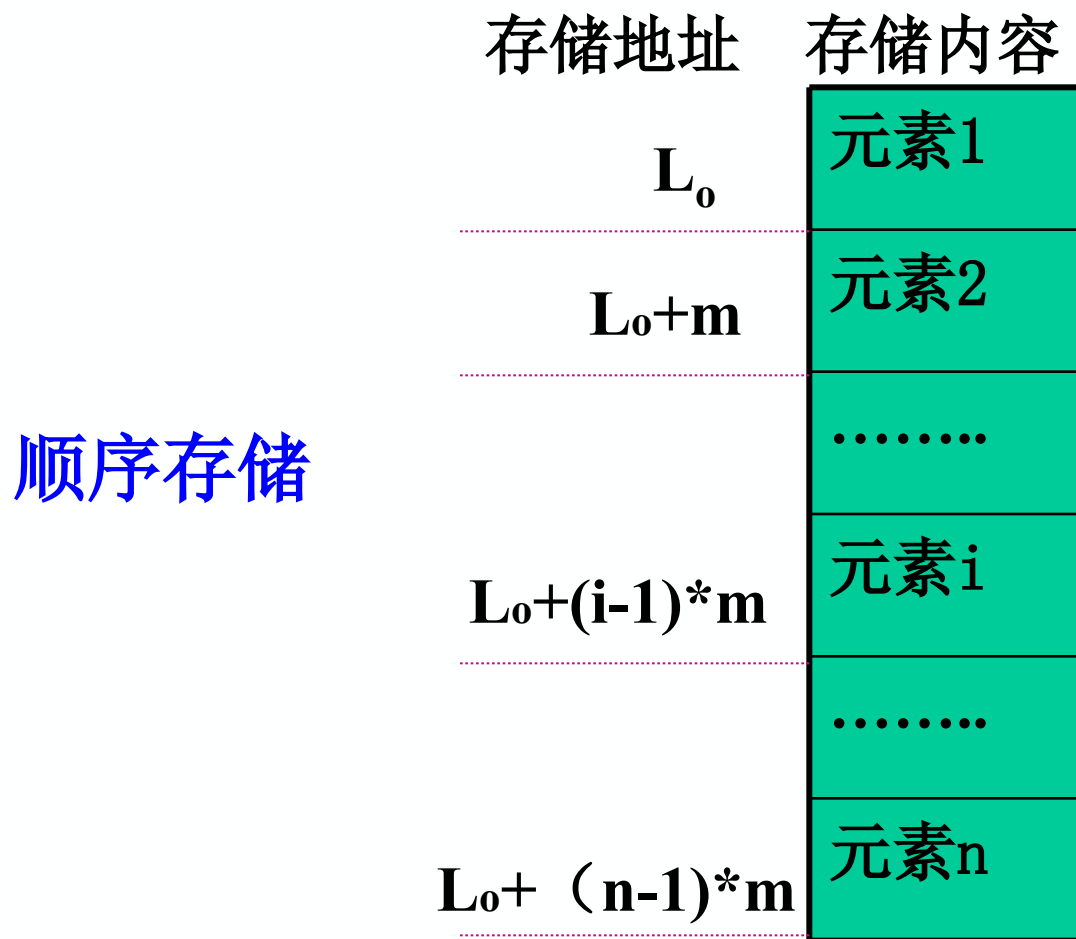
线性表的顺序表示又称为**顺序存储结构**或**顺序映像**。

**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。





$$\text{Loc(元素i)} = L_0 + (i-1)*m$$

```
#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;
```

数据结构基本运算操作有：

查找、插入、删除

## 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

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——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。

# 引用类型作形参的三点说明

(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);
```

```
void main (void )
```

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

```
    sub(a);
```

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

```
}
```

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

```
{ b[ ]="world";}
```

```
#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];
}
```

# 线性表的基本操作在顺序表中的实现

## 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;  
}
```

## 2. 销毁线性表L

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

## 3. 清空线性表L

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

## 4. 求线性表L的长度

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

## 5. 判断线性表L是否为空

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

## 6. 获取线性表L中的某个数据元素的内容

//根据指定位置，获取相应位置数据元素的内容

```
int GetElem(SqList L, int i, ElemType &e)
{
    if (i<1 || i>L.length) return ERROR;

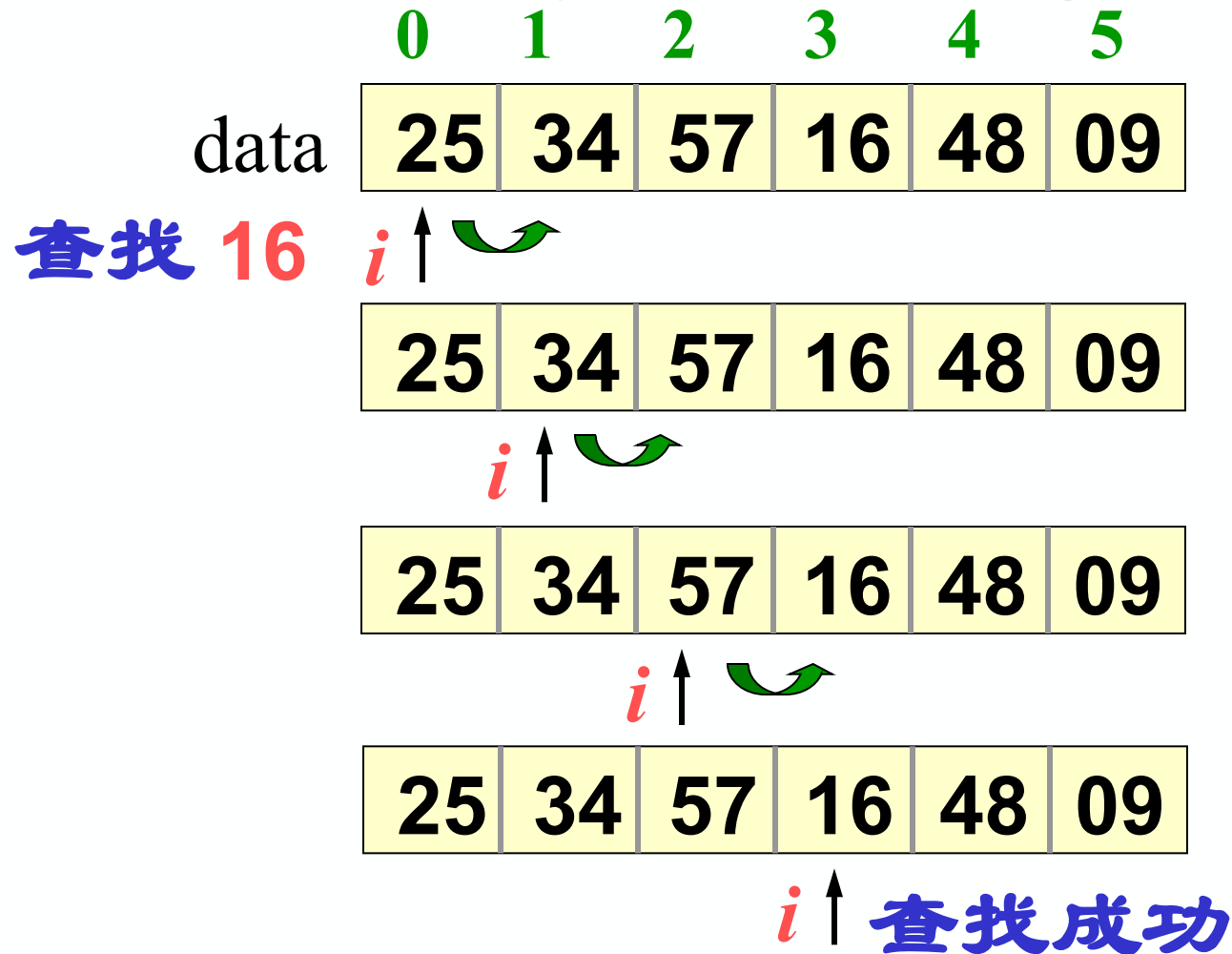
    //判断i值是否合理，若不合理，返回ERROR

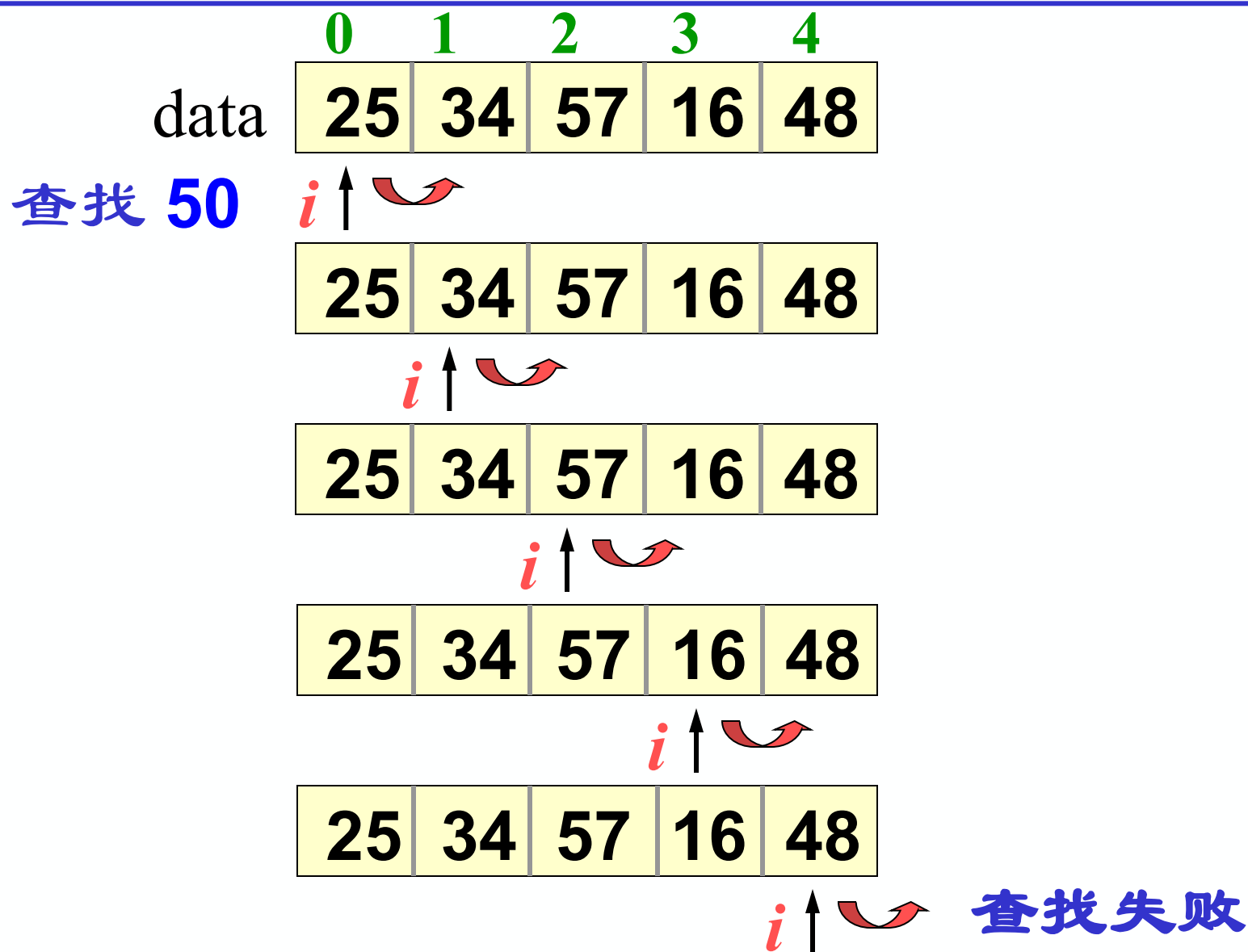
    e=L.elem[i-1];    //第i-1的单元存储着第i个数据

    return OK;
}
```

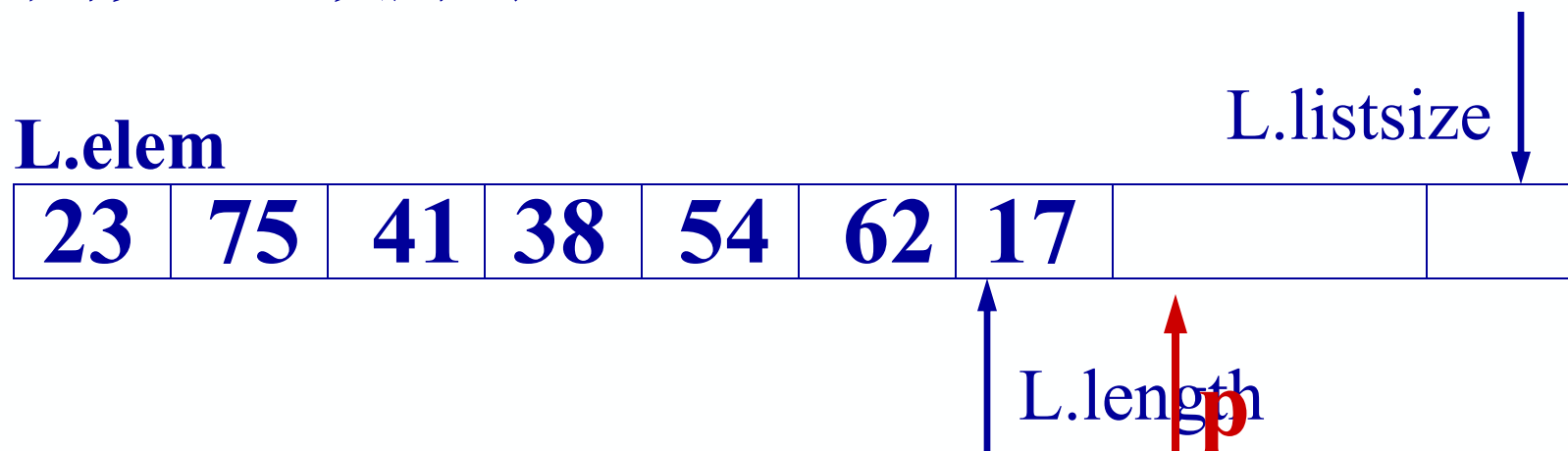
# 查找（根据指定数据查找，返回数据所在的位置）

## 顺序查找图示





例如：顺序表



i 8

e = 38 50

可见，基本操作是：  
将顺序表中的元素  
逐个和给定值 e  
相比较。

## 7. 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 ? ? ?



查找成功的平均比较次数( $p_i$ 为各项的查找概率)

若查找概率相等，则

查找不成功 数据比较  $n$  次

**查找算法的时间复杂度为：**

**$O(\text{ListLength}(L))$**

**$O(n)$**

# 线性表操作

**ListInsert(&L, i, e)的实现:**

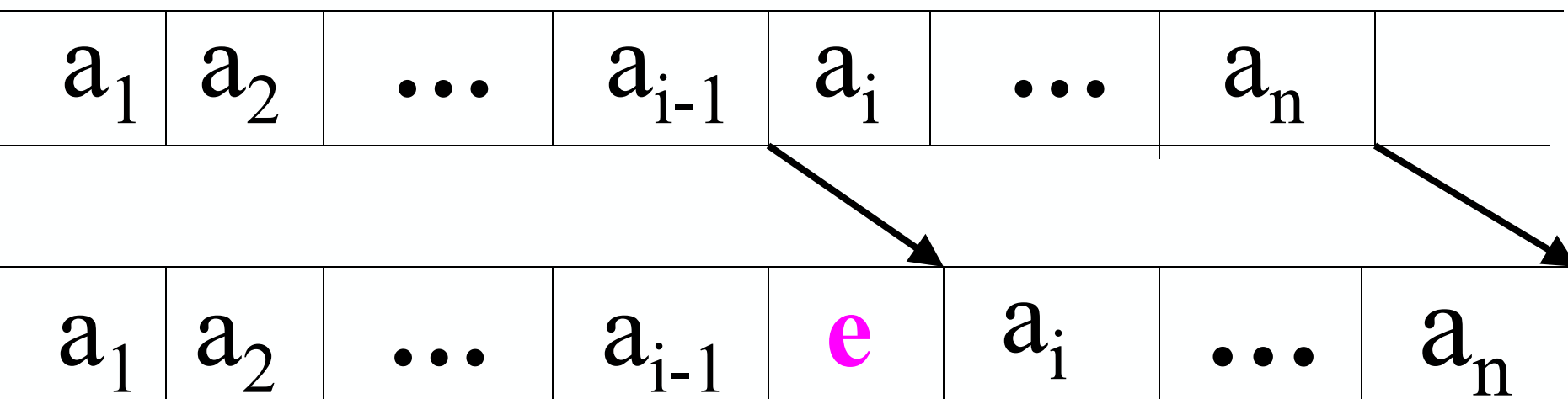
**首先分析:**

插入元素时,

线性表的**逻辑结构**发生什么变化?

$(a_1, \dots, a_{i-1}, a_i, \dots, a_n)$  改变为  
 $(a_1, \dots, a_{i-1}, e, a_i, \dots, a_n)$

$\langle a_{i-1}, a_i \rangle \longrightarrow \langle a_{i-1}, e \rangle, \langle e, a_i \rangle$



表的长度增加

# 算法1：在线性表中指定位置前插入一个元素

插入元素时，线性表的逻辑结构由 $(a_1, \dots, a_{i-1}, a_i, \dots, a_n)$ 改变为 $(a_1, \dots, a_{i-1}, b, a_i, \dots, a_n)$ ，在第 $i-1$ 个数据元素和第 $i$ 个数据元素之间插入新的数据元素。

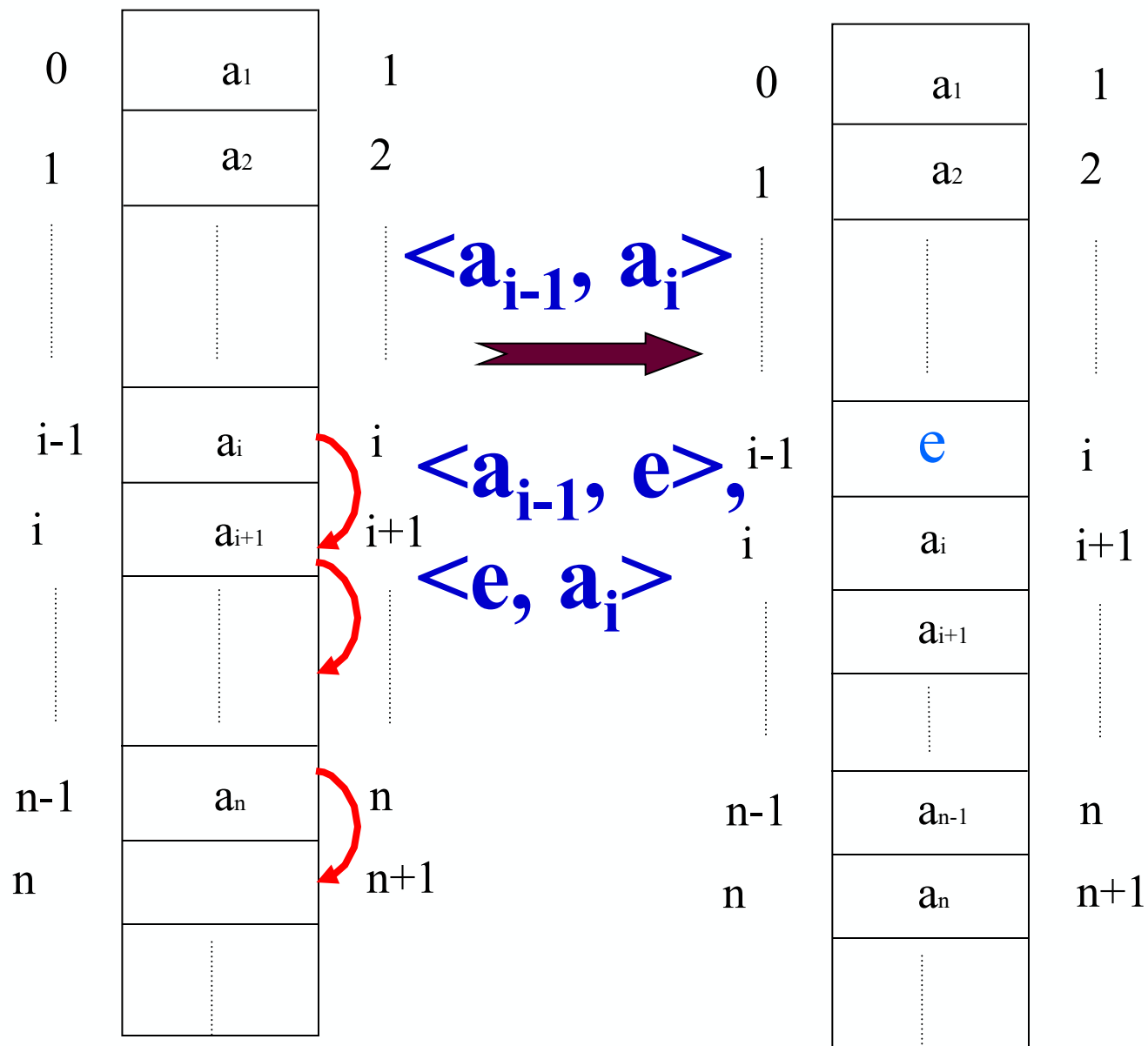
$$(a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$$

变成长度为 $n+1$ 的线性表

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

需将第 $i$ 至第 $n$ 共  $(n-i+1)$  个元素后移

V数组下标    内存    元素序号                  V数组下标    内存    元素序号

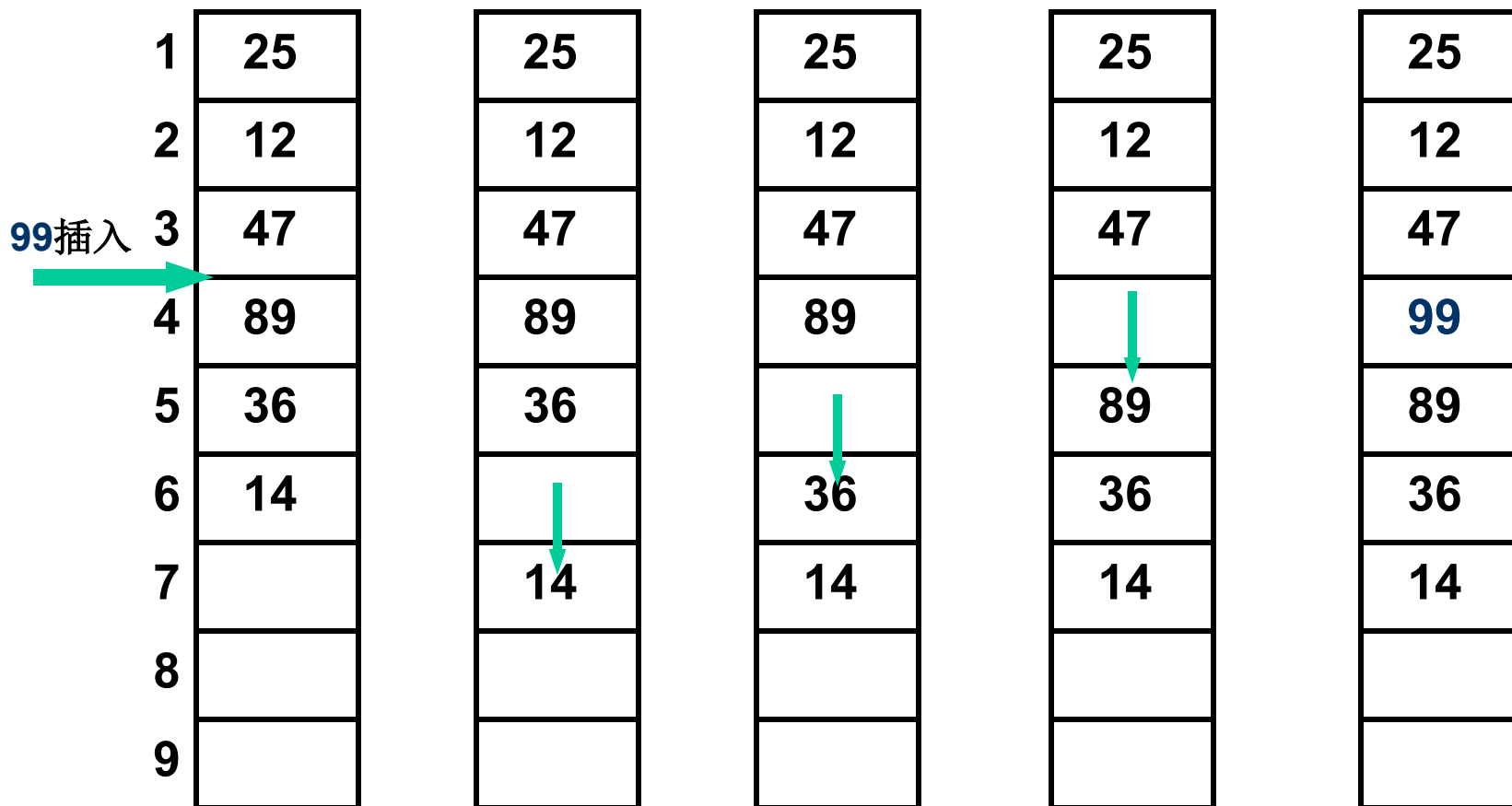




插入25

图 线性表插入前后的状况  
(a) 插入前 $n=8$ ; (b) 插入后 $n=9$

# 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



- (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。

## 8. 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？

## 考虑移动元素的平均情况：

假设在第  $i$  个元素之前插入的概率为  $p_i$ ，则在长度为  $n$  的线性表中插入一个元素所需移动元素次数的期望值为：

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

若假定在线性表中任何一个位置上进行插入的概率都是相等的，则移动元素的期望值为：

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

**插入 算法时间复杂度为：**

**$O(\text{ListLength}(L))$**

**$O(n)$**

# 线性表操作

**ListDelete(&L, i, &e)**的实现:

首先分析:

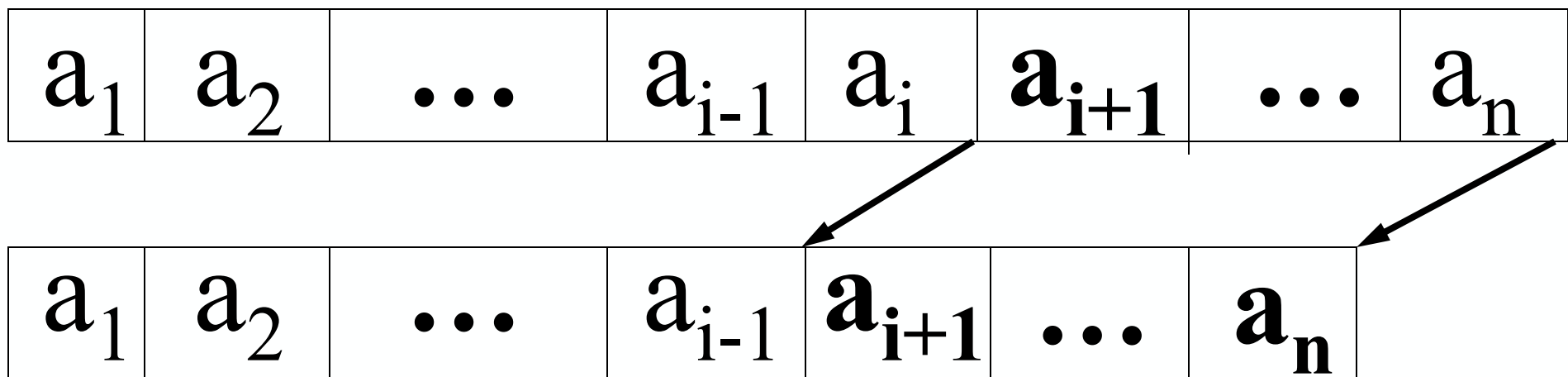
删除元素时,

线性表的逻辑结构发生什么变化?

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

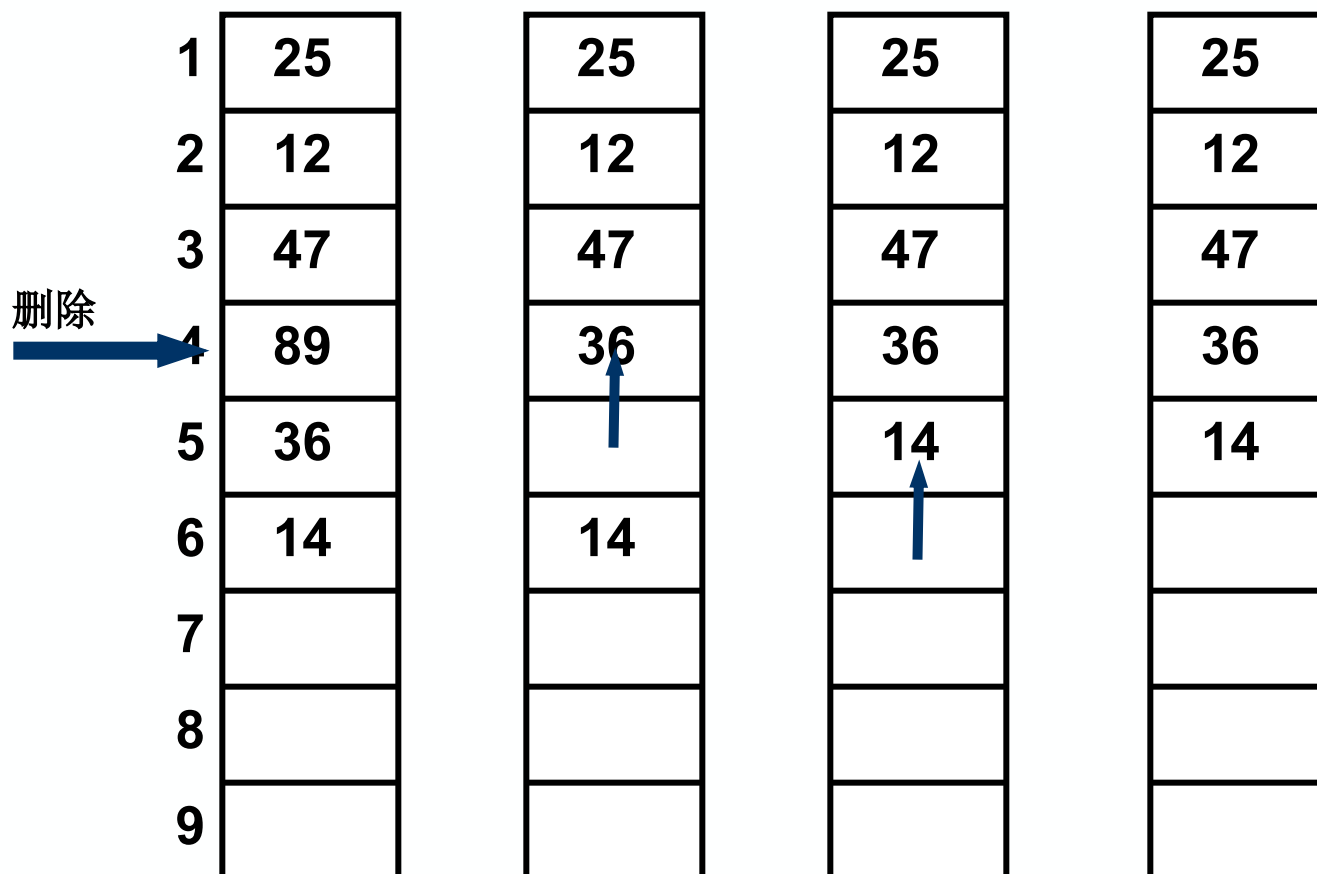
$(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

$\langle a_{i-1}, a_i \rangle, \langle a_i, a_{i+1} \rangle \longrightarrow \langle a_{i-1}, a_{i+1} \rangle$



表的长度减少

# 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



- (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。

## 9. 将线性表L中第i个数据元素删除

```
Status ListDelete_Sq(SqList &L,int i,ElemType &e){  
    if((i<1)||(i>L.length)) return ERROR;    //i值不合法  
    e=L.elem[i-1];                            //将欲删除的元素保留在e中  
    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？

## 考虑移动元素的平均情况：

假设删除第  $i$  个元素的概率为  $q_i$ ，  
则在长度为  $n$  的线性表中删除一个元素所需  
移动元素次数的期望值为：

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

若假定在线性表中任何一个位置上进行删除  
的概率都是相等的，则移动元素的期望值为：

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

**删除 算法时间复杂度为：**

$$O(\text{ListLength}(L))$$

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)

- (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**



# 知识回顾

- 线性表的基本概念
- 线性表的抽象数据类型定义
- 线性表的顺序存储(顺序表)



## 2.3 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

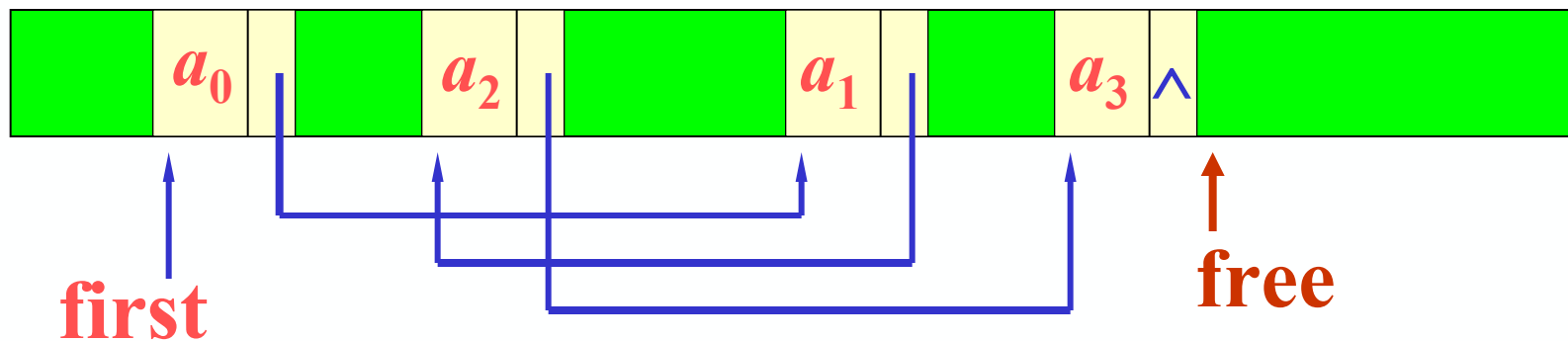
线性表的链式表示又称为非顺序映像或链式映像。

## 单链表的存储映像



↑  
**free**

(a) 可利用存储空间

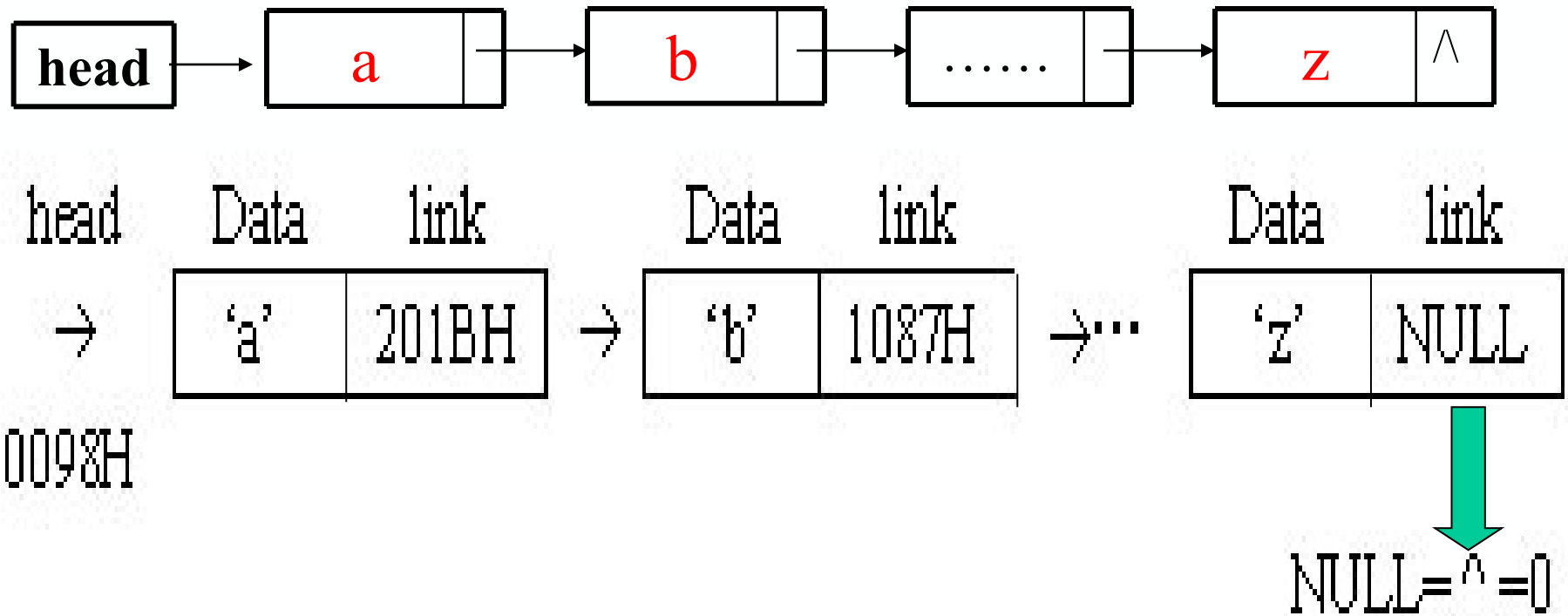


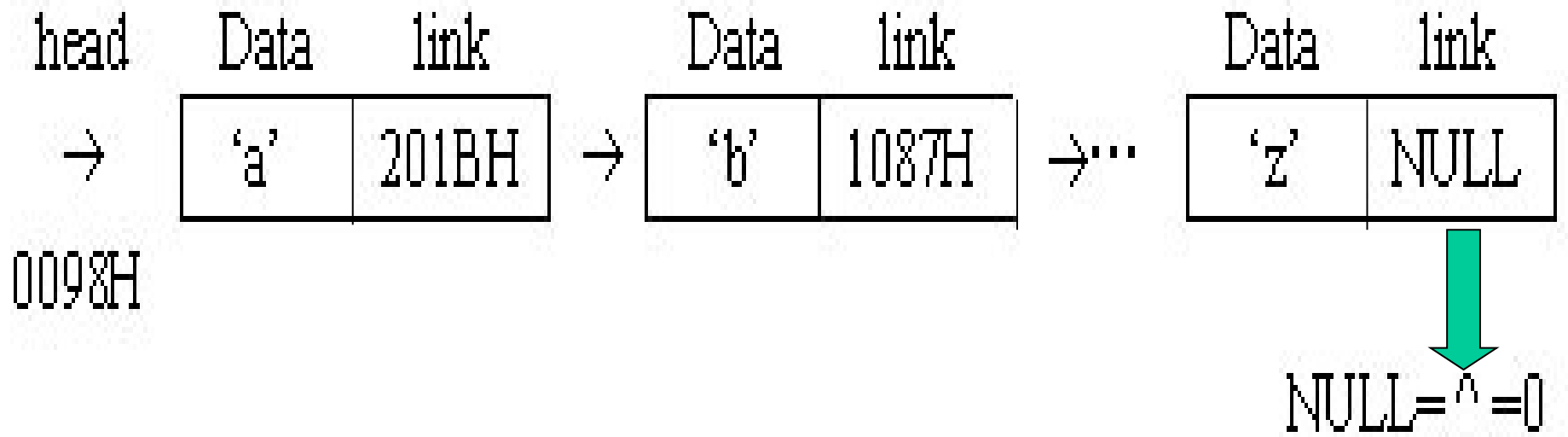
(b) 经过一段运行后的单链表结构

# 例 画出26个英文字母表的链式存储结构

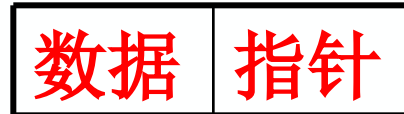
逻辑结构: ( a, b, ... , y, z )

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

用一组**地址任意**的存储单元**存放**线性表中的数据元素。

以**元素**(数据元素的映象)

+ **指针**(指示后继元素存储位置)

= **结点**

(表示数据元素 或 数据元素的存储映象)

以“**结点的序列**”表示线性表

——称作**链表**

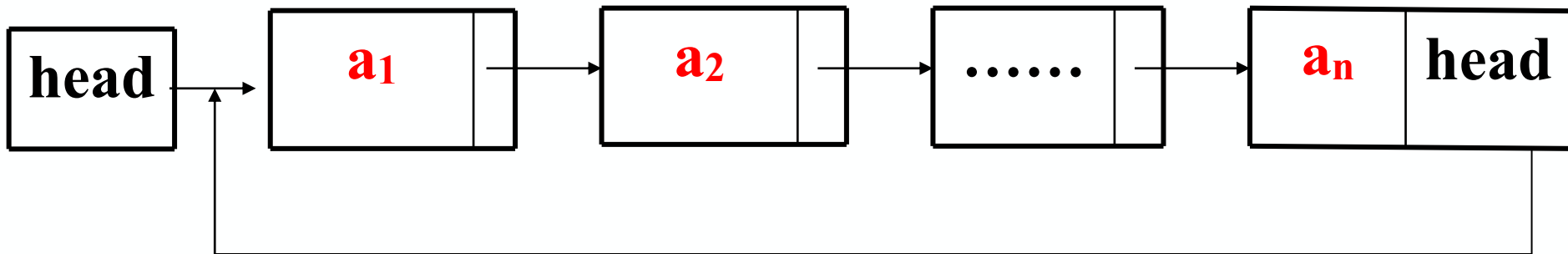
# 与链式存储有关的术语

- 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：

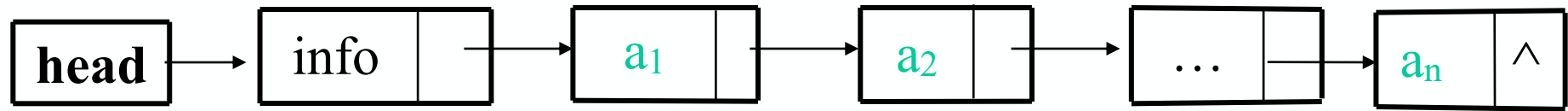
- 结点只有一个指针域的链表，称为**单链表**或**线性链表**
- 有两个指针域的链表，称为**双链表**
- 首尾相接的链表称为**循环链表**

循环链表示意图：





## 4、头指针、头结点和首元结点



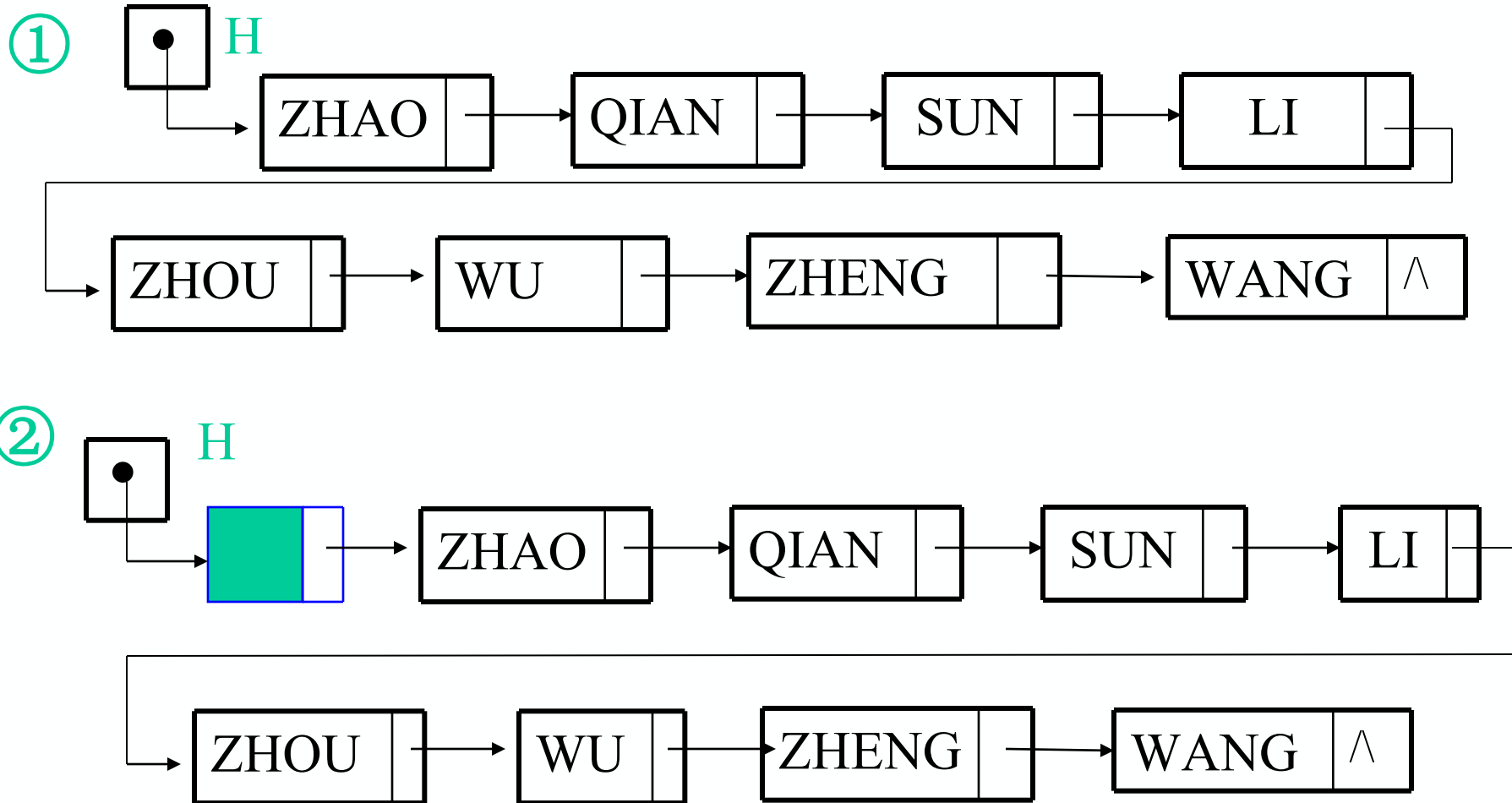
头指针 头结点 首元结点

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

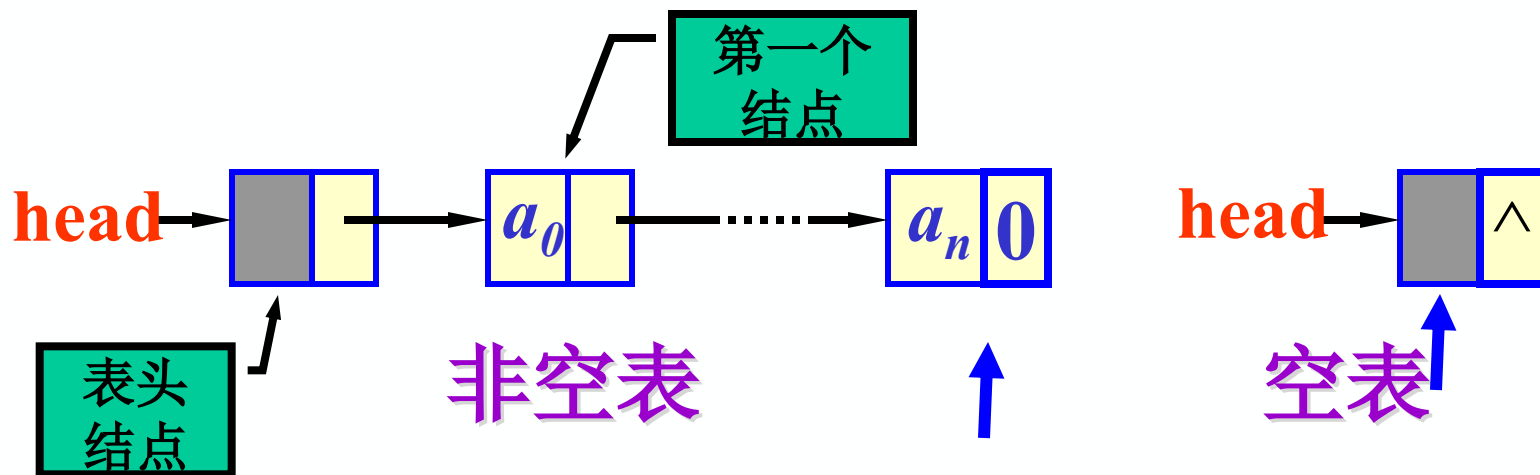
头结点是在链表的首元结点之前附设的一个结点；数据域内只放空表标志和表长等信息

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

有头结点时，当头结点的指针域为空时表示空表



### 1. 便于**首元结点**的处理

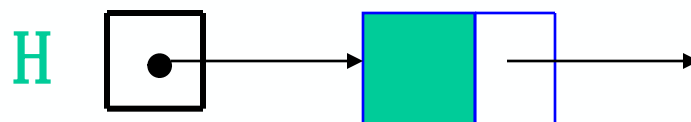
首元结点的地址保存在头结点的指针域中，所以在链表的第一个位置上的操作和其它位置一致，无须进行特殊处理；

### 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

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**

# 链表的优缺点

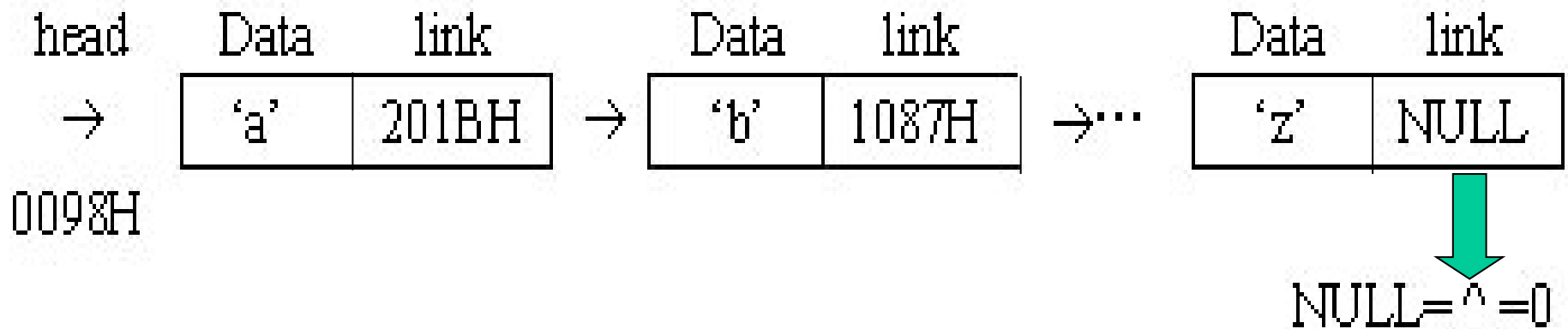
## 优点

- 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**）



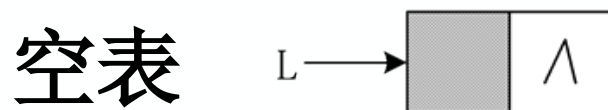
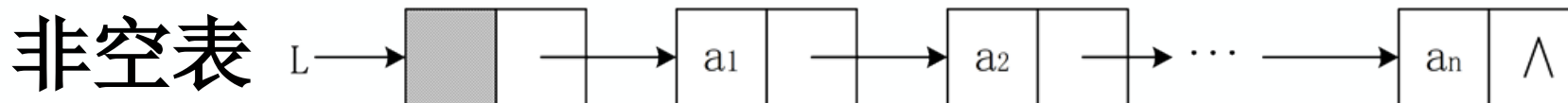


# 练习

- 
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



## 2.3.1 单链表的定义和实现



- ✓ 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**



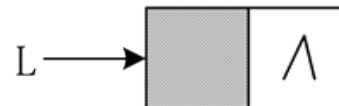
若  $p \rightarrow \text{data} = a_i$ , 则  $p \rightarrow \text{next} \rightarrow \text{data} = a_{i+1}$



## 2.3.2 单链表基本操作的实现

1. 初始化线性表L    `InitList(L)`
2. 销毁线性表L    `DestoryList(L)`
3. 清空线性表L    `ClearList(L)`
4. 求线性表L的长度    `ListLength(L)`
5. 判断线性表L是否为空    `IsEmpty(L)`
6. 获取线性表L中的某个数据元素内容    `GetElem(L, i, e)`
7. 检索值为e的数据元素    `LocateElem(L, e)`
8. 在线性表L中插入一个数据元素    `ListInsert(L, i, e)`
9. 删除线性表L中第i个数据元素    `ListDelete(L, i, e)`

## 1. 初始化(构造一个空表)



### 【算法思想】

- (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;  
}
```

## 2. 销毁

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

### 3. 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;
```

```
}
```



## 4. 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;  
}
```

## 5. 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;  
}
```

- 按序号查找
- 按值查找

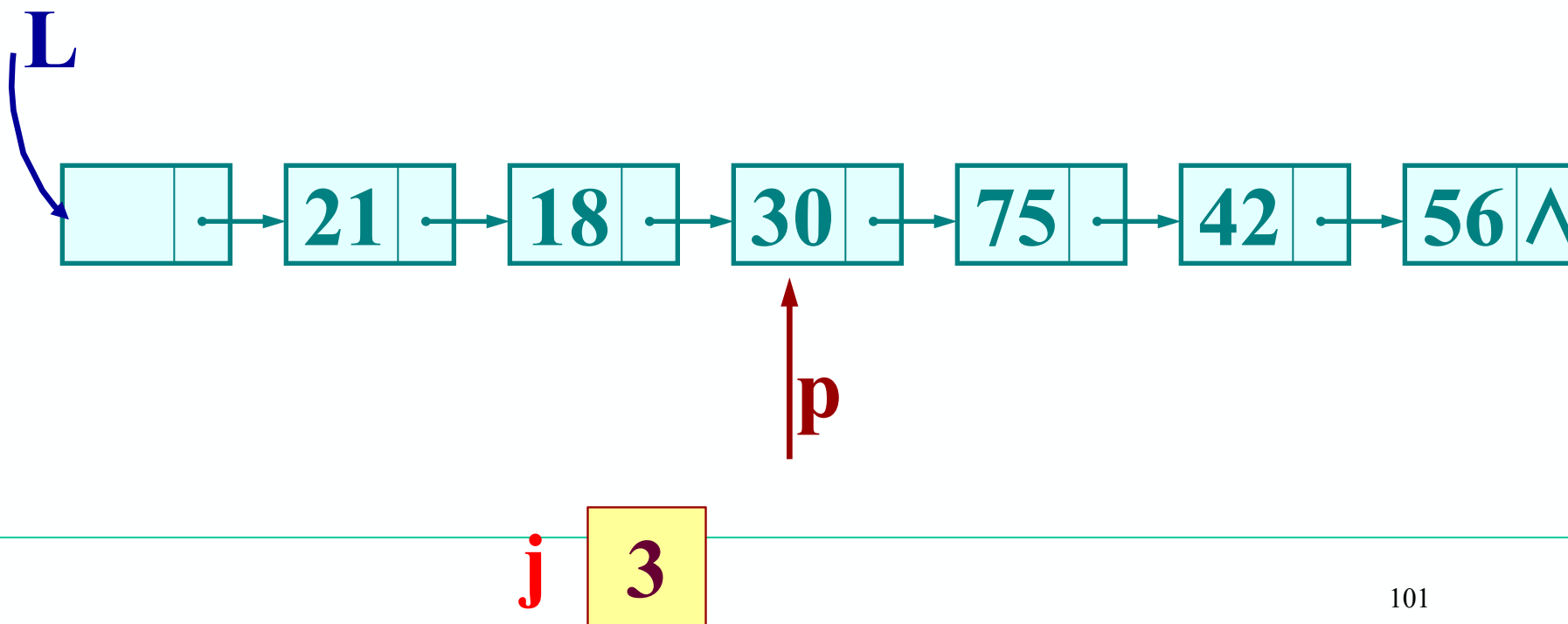
## 按序号查找

- 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

# 线性表的操作

## GetElem(L, i, &e)

在单链表中的实现:



- ✓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 $p = L \rightarrow next$ 。
- ✓用 $j$ 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。

### 6. 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
```

## 复杂度分析

- 查找第  $i$  个数据元素的基本操作为：移动指针，比较  $j$  和  $i$
- 若  $1 \leq i \leq n$ , 度  $\leq i-1$ .
- 若  $i > n$ , 度  $\leq n$

$O(n)$



- ✓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”；
- ✓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“NULL”。

### 7. 在线性表L中查找值为e的数据元素

```
LNode *LocateELem_L (LinkList L, Elemtyp e) {  
    p=L->next;  
    while(p && p->data!=e)  
        p=p->next;  
    return p; //返回L中值为e的数据元素的位置, 查找失败返回NULL  
}
```

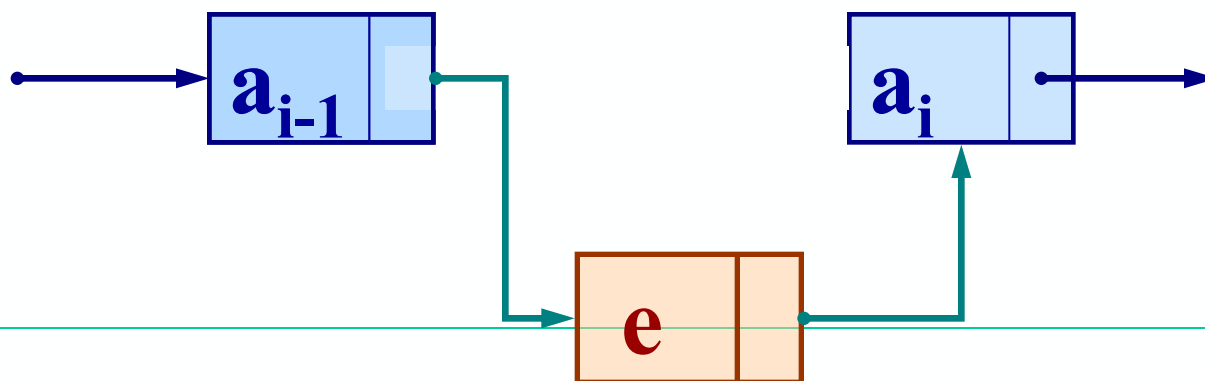
**$O(n)$**

# 线性表的操作 **ListInsert(&L, i, e)**

在单链表中的实现:

有序对  $\langle a_{i-1}, a_i \rangle$

改变为  $\langle a_{i-1}, e \rangle$  和  $\langle e, a_i \rangle$

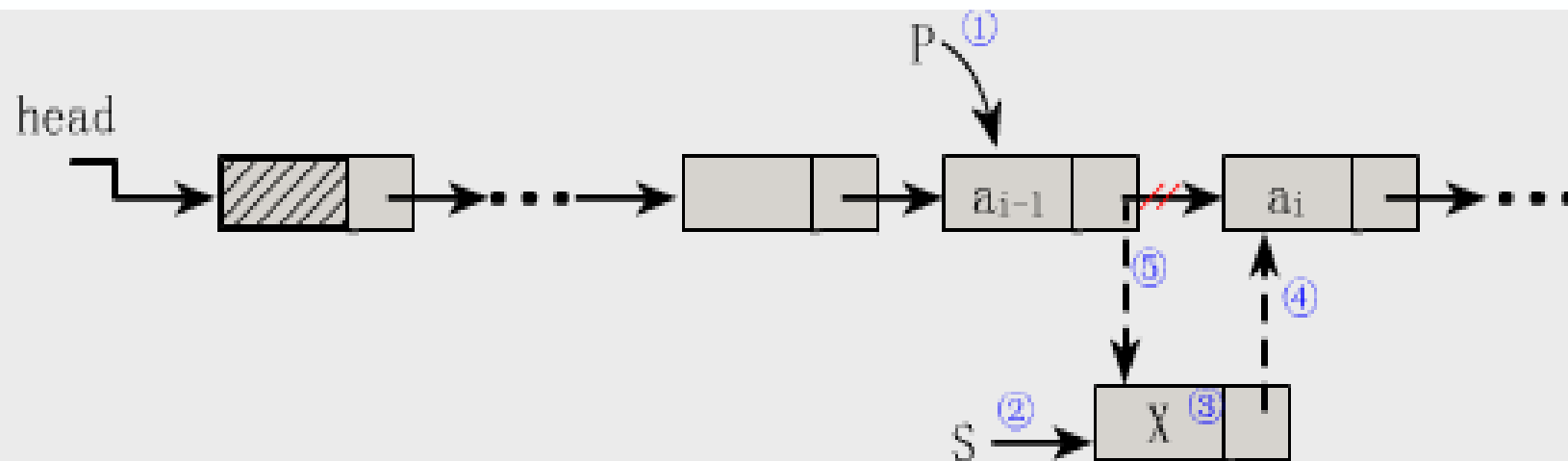


可见，在链表中插入结点只需要修改指针。但同时，若要在第  $i$  个结点之前插入元素，修改的是第  $i-1$  个结点的指针。

因此，在单链表中第  $i$  个结点之前进行插入的基本操作为：

找到线性表中第  $i-1$  个结点，然后修改其指向后继的指针。

- 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

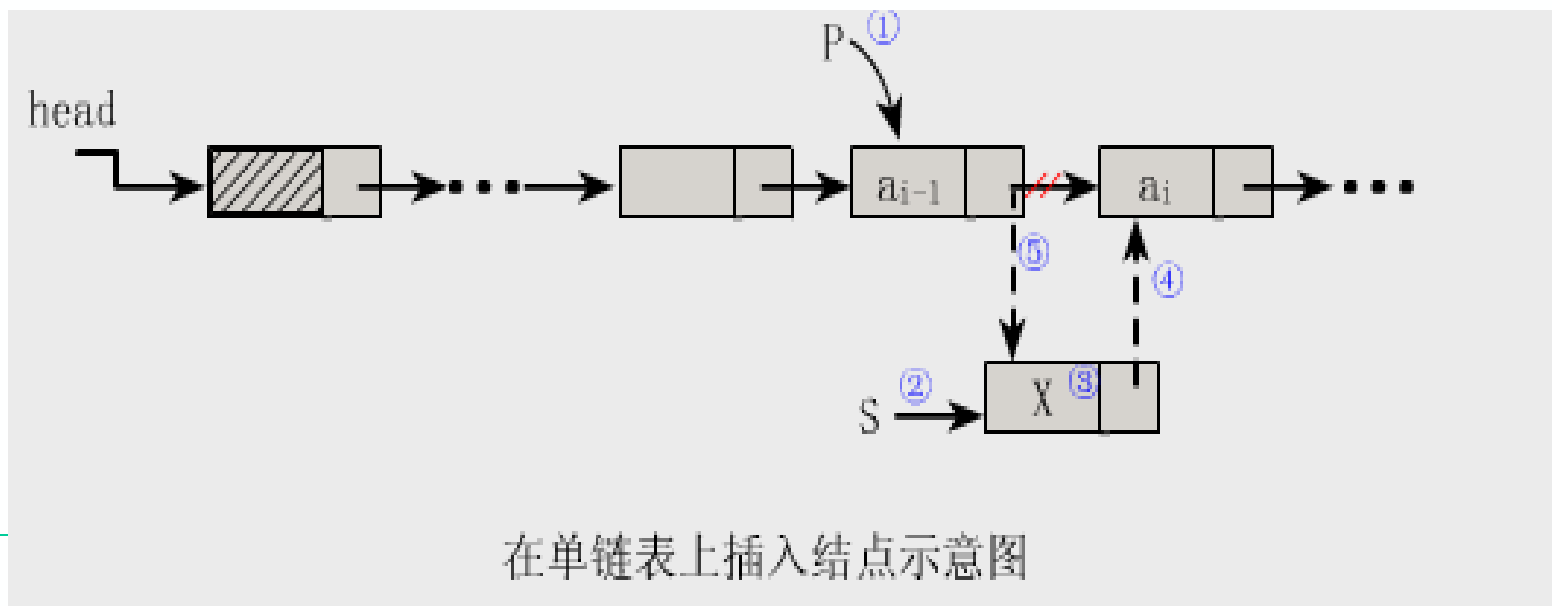


在单链表上插入结点示意图

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

思考：步骤1和2能互换么？

- (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 $*s$



## 8. 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
```

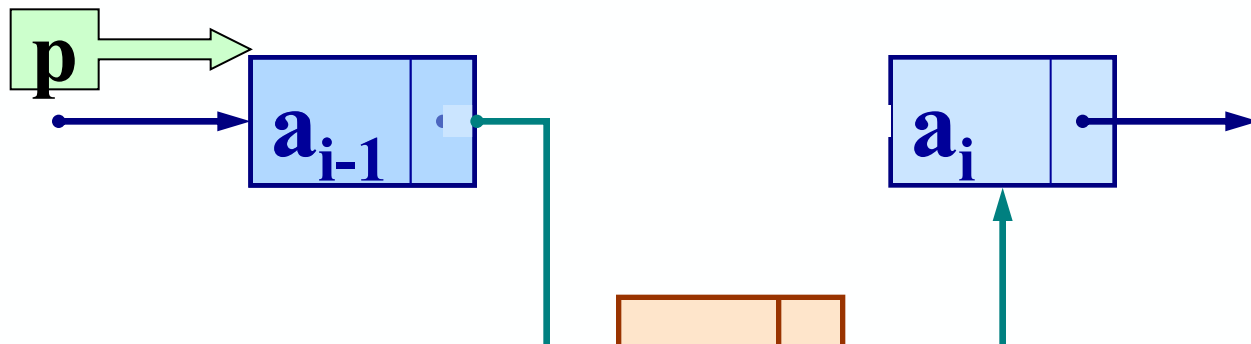
```
    s->data=e;    //将结点s的数据域置为e
```

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

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

```
    return OK;
```

```
}//ListInsert_L
```

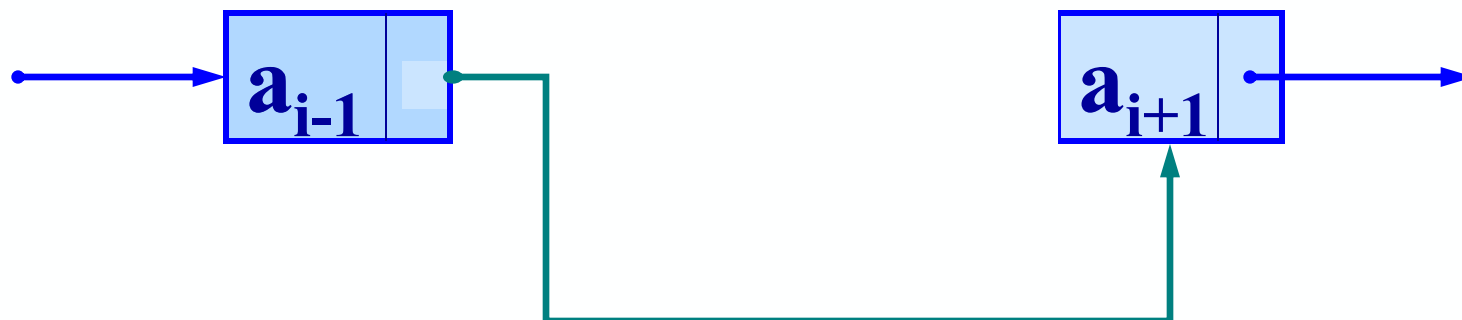


算法的时间复杂度为:  $O(\text{ListLength}(L))$

# 线性表的操作 **ListDelete (&L, i, &e)** 在链表中的实现:

有序对  $\langle a_{i-1}, a_i \rangle$  和  $\langle a_i, a_{i+1} \rangle$

改变为  $\langle a_{i-1}, a_{i+1} \rangle$



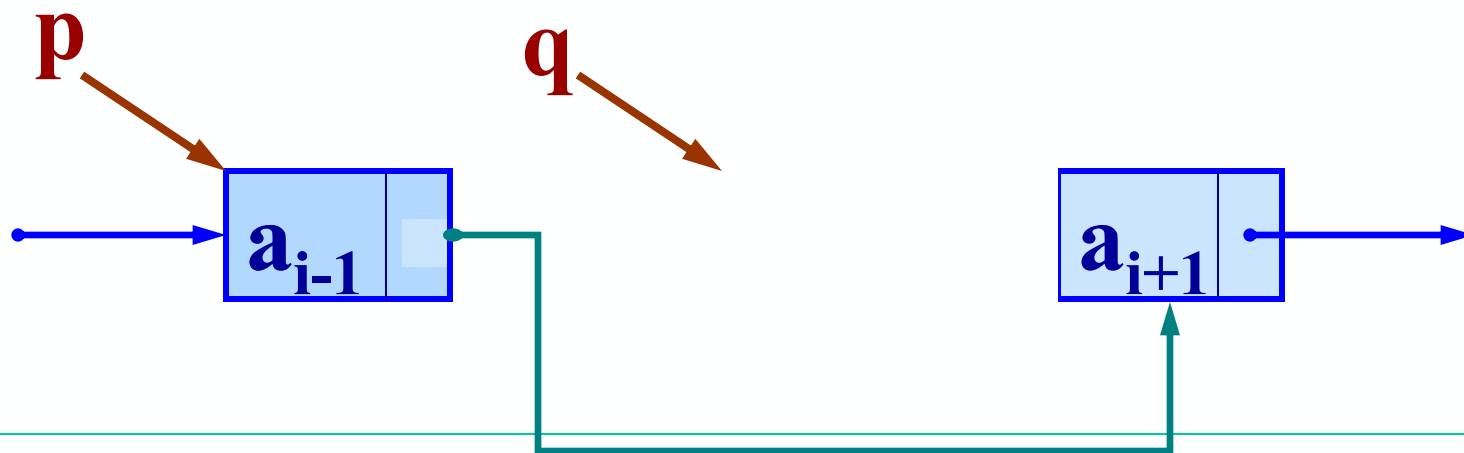


# 在单链表中删除第 $i$ 个结点的基本

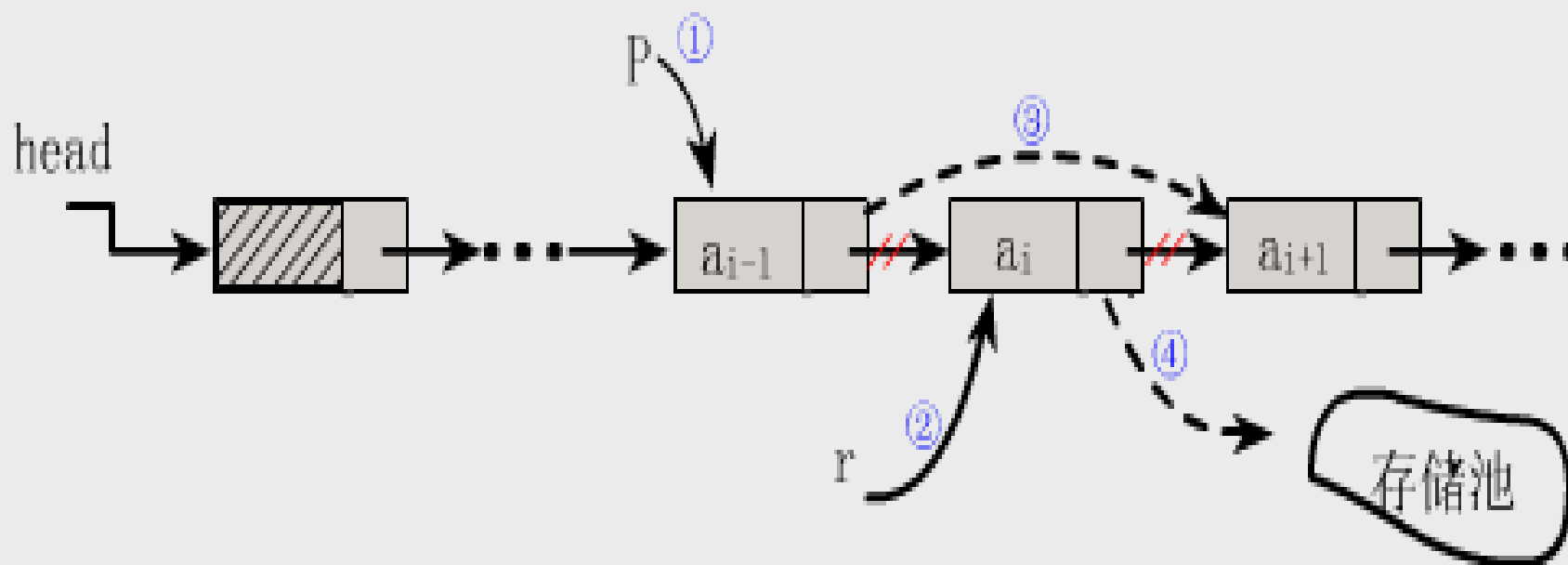
操作为：找到线性表中第  $i-1$  个结点，修改其指向后继的指针。

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

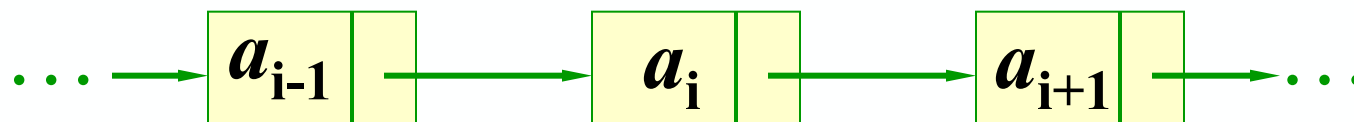
$e = q \rightarrow \text{data};$   $\text{free}(q);$



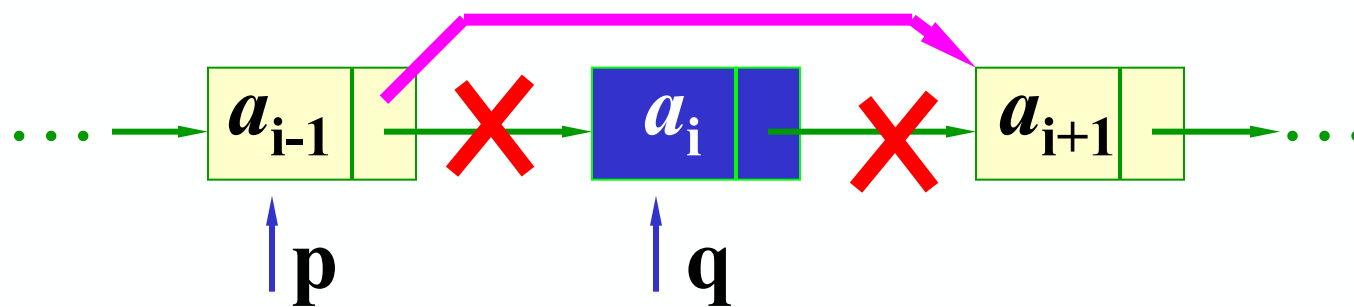
- 将表的第 $i$ 个结点删去
- 步骤：
  - (1) 找到 $a_{i-1}$ 存储位置 $p$
  - (2) 保存要删除的结点的值
  - (3) 令 $p \rightarrow \text{next}$ 指向 $a_i$ 的直接后继结点
  - (4) 释放结点 $a_i$ 的空间



在单链表上删除结点示意图



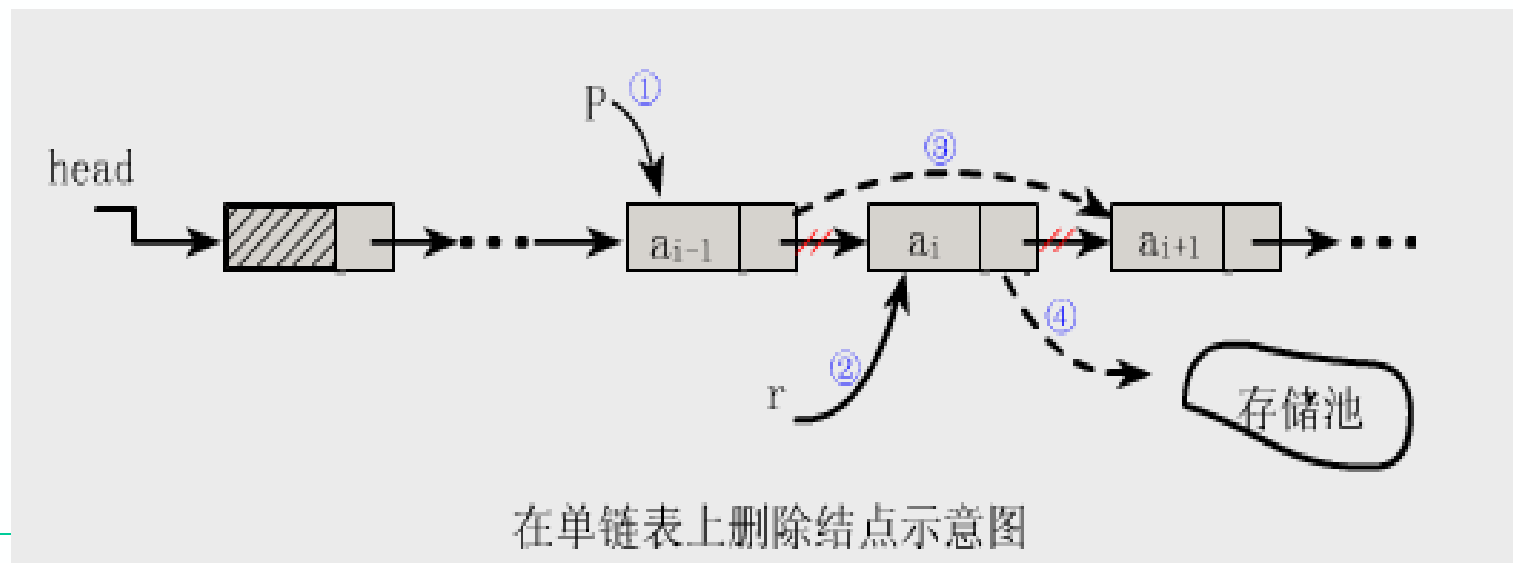
删除前



删除后

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

- (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



## 9. 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-1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
```

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  $O(1)$ 。

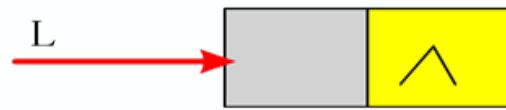
但是，如果要在单链表中进行前插或删除操作，由于要从头查找前驱结点，所耗时间复杂度为  $O(n)$ 。

# 如何从线性表得到单链表？ (创建单链表)

链表是一个动态的结构，它不需要  
予分配空间，因此生成链表的过程  
是一个结点“逐个插入”的过程。

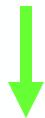
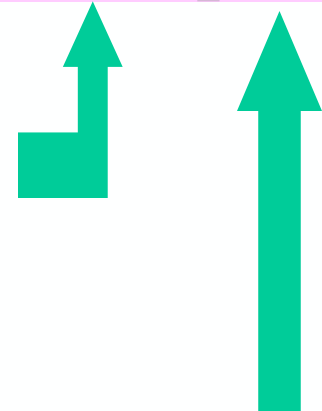


- 从一个空表开始，重复读入数据：
  - ◆ 生成新结点
  - ◆ 将读入数据存放到新结点的数据域中
  - ◆ 将该新结点插入到链表的前端



$p \rightarrow \text{data} = a_n$

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



$p \rightarrow \text{data} = a_{n-1}$

```
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
```

- 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。

```
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
```

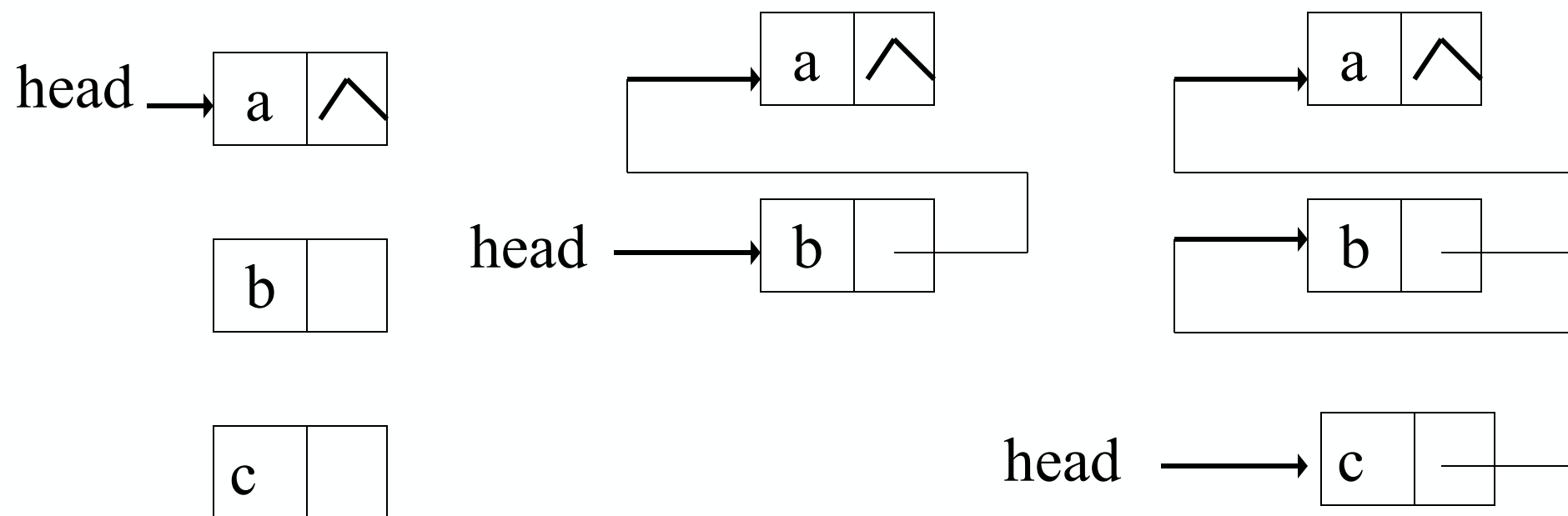
# 思考

- 链表的输出

# 不带头结点的单链表

# 1 建立单链表: 设成员数据域为字符型

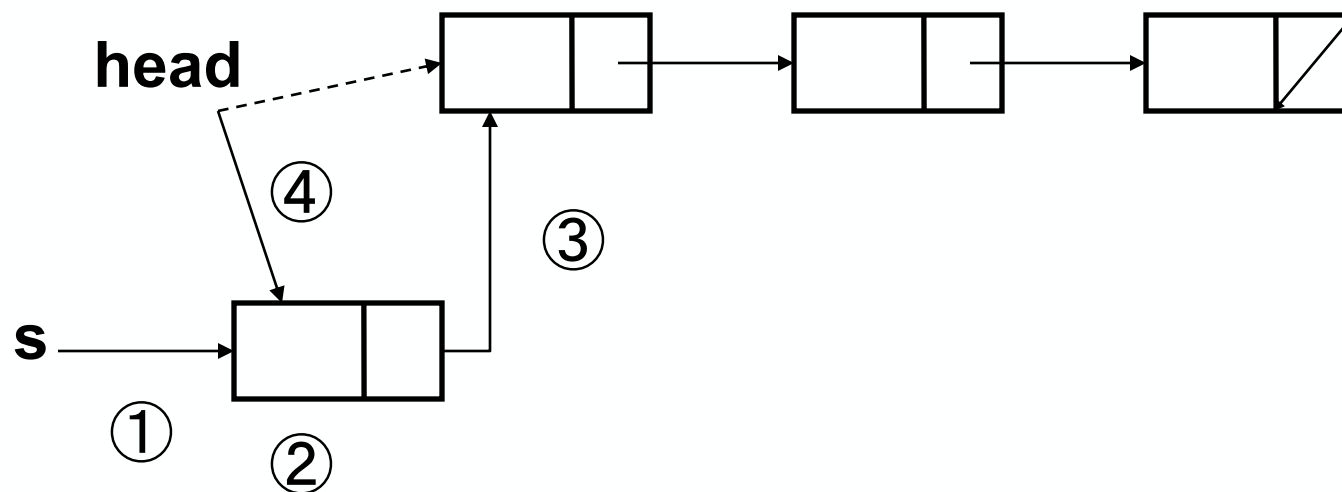
(1) 头插法建表: 依次插入新结点, 并置于表头



**规律:** 新增加的结点指针域指向原`head`指向的结点, 然后将原`head`指向新结点



## 头插法建立单链表



- ① 建立新节点
- ② 向新节点中添入内容
- ③ 使新节点指向链首
- ④ 改变头指针

# 头插法建立单链表

head →

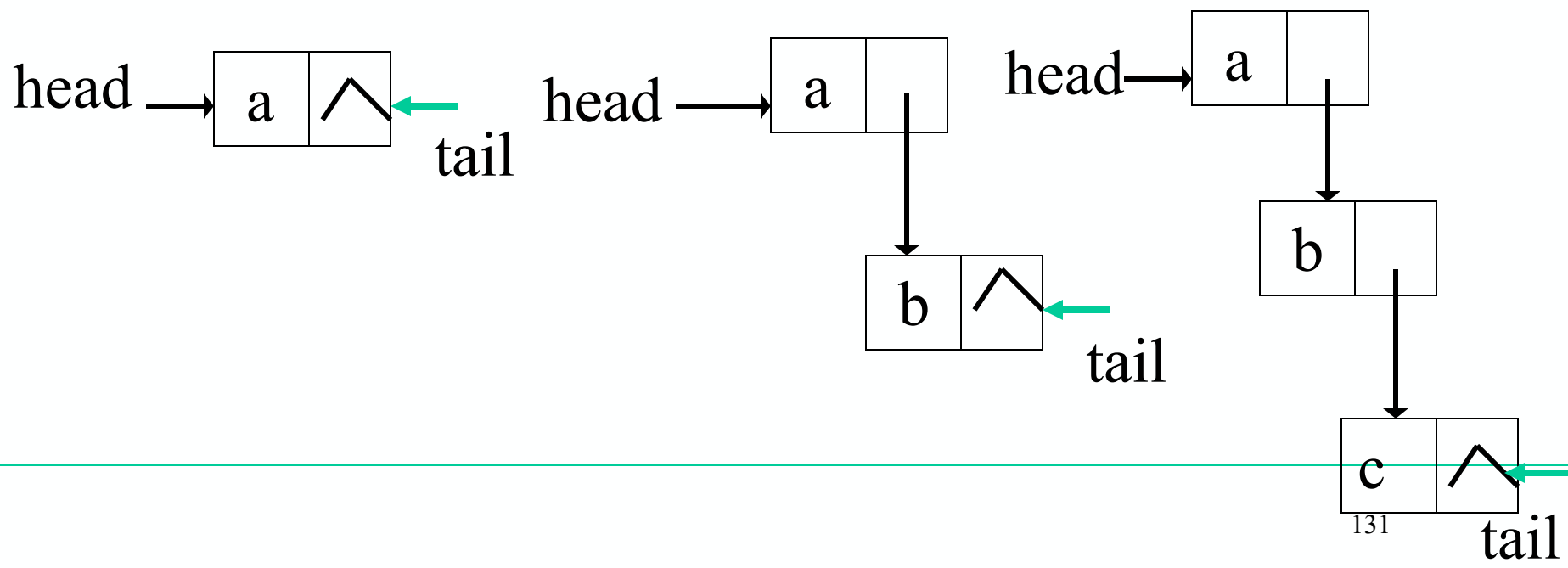
```
linklist *CREATelist()
{ char ch;
  linklist *head,*s;
  head=NULL;
  ch=getchar();
  while (ch!='$')
  { s=malloc(sizeof(linklist));
    s->data=ch;
    s->next=head;
    head=s;
    ch=getchar();
  }
  return(head);}
```

从键盘输入 o  
产生一个指针变量s1  
s1的数据域为‘o’  
head指向s1

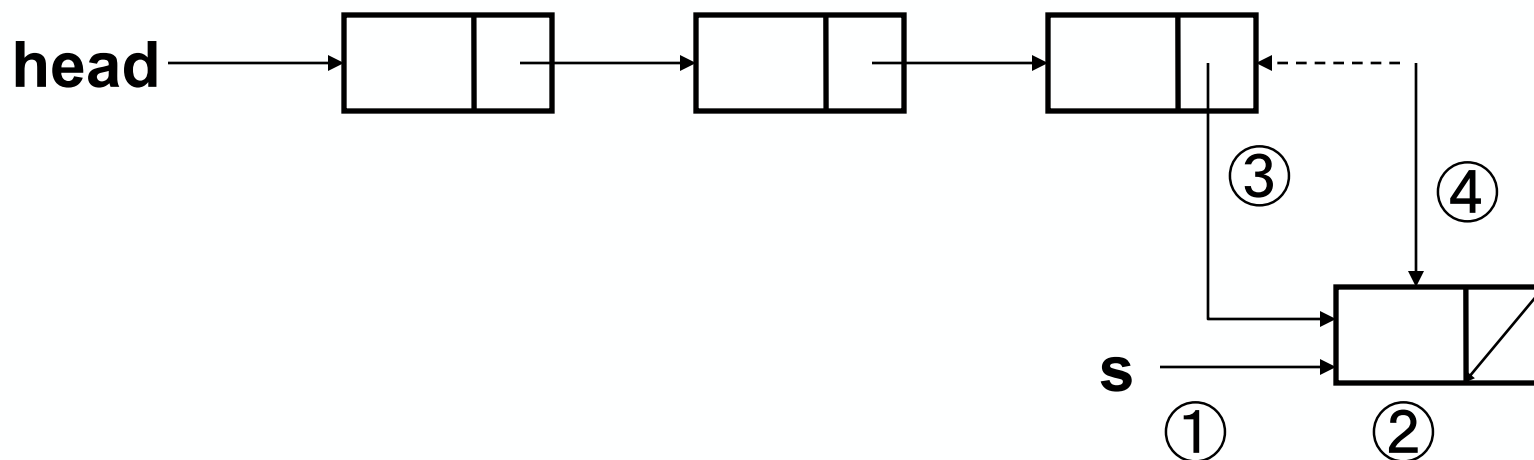
再输入k  
产生指针变量s2  
s2的数据域为‘k’  
s2的指针域为s1  
head指向s2

## (2) 尾插法建表：插入的新结点置于表尾

- 产生一个新结点；
- 新结点的数据域为插入字符；
- 原尾结点（由尾指针指定）的指针域为新结点；
- 尾指针指向新结点；
- 新结点的指针域为空



## 尾插法建立单链表



- ① 建立新节点
- ② 向新节点中添入内容
- ③ 将新节点链入链尾
- ④ 改变尾指针

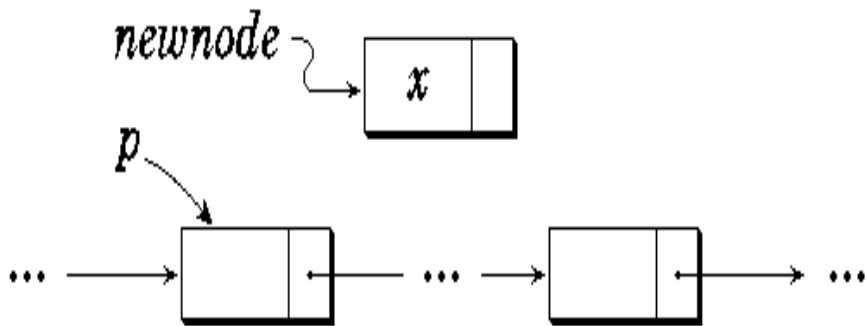
## 尾插法建立单链表

```
linklist *CREATELISTR()
{ char ch;
  linklist *head,*s,*r;
  head=NULL;
  r=NULL;
  ch=getchar();
  while(ch!='$')
  {
    s=malloc(sizeof(linklist));
    s->data=ch;
    if (head==NULL) head=s;
    else r->next=s;
    r=s;
    ch=getchar();
  }
```

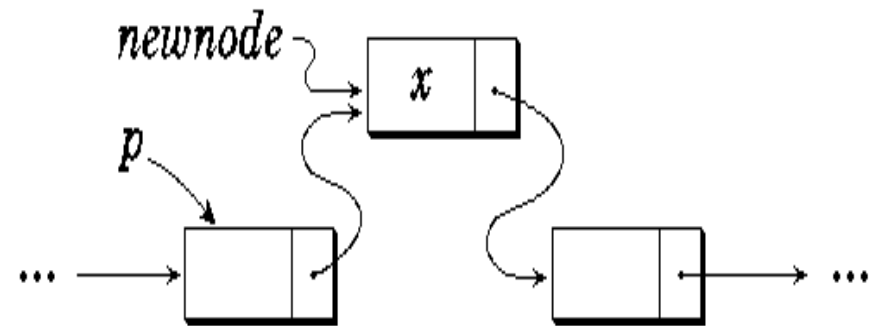
```
if (r!=NULL) r->next=NULL;
return head;
}
```

## • 单链表的插入

- 第一种情况：在第一个结点前插入
- 第二种情况：在链表末尾插入
- 第三种情况：在链表中间插入



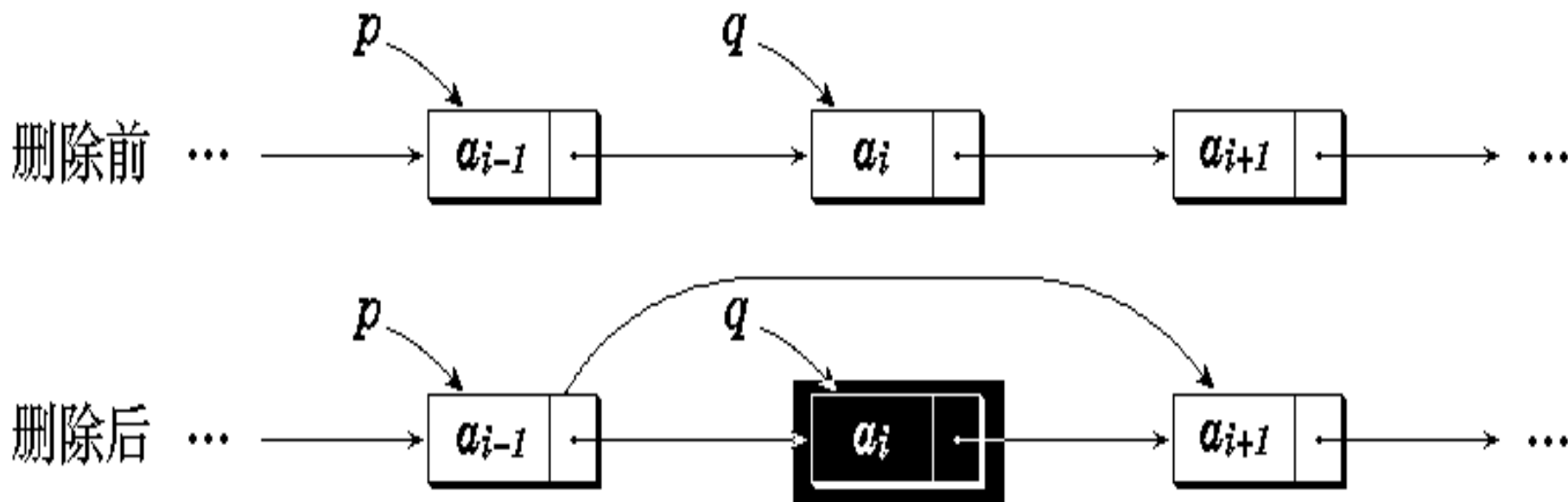
(插入前)



(插入后)

# • 删除

- 第一种情况：删除表中第一个元素
- 第二种情况：删除表中或表尾元素



在单链表中删除含 $a_i$ 的结点

- 单链表特点

- 它是一种动态结构，整个存储空间为多个链表共用
- 不需预先分配空间
- 指针占用额外存储空间
- 不能随机存取，查找速度慢



# 知识回顾

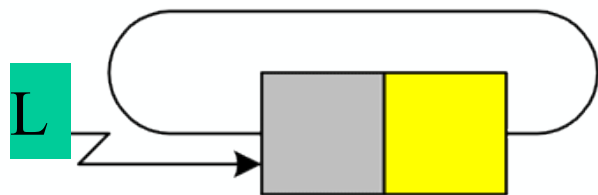
- 线性表的链式存储
- 单链表（带头结点）
  - 单链表的定义
  - 单链表的建立
  - **插入一个节点**
  - 删除一个节点
  - 输出



## 2.3.3 循环链表

L

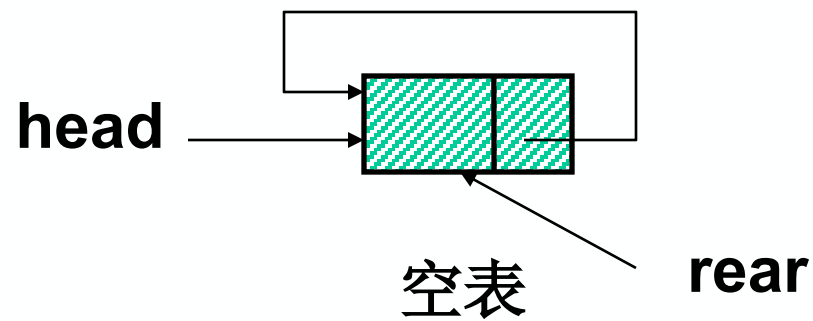
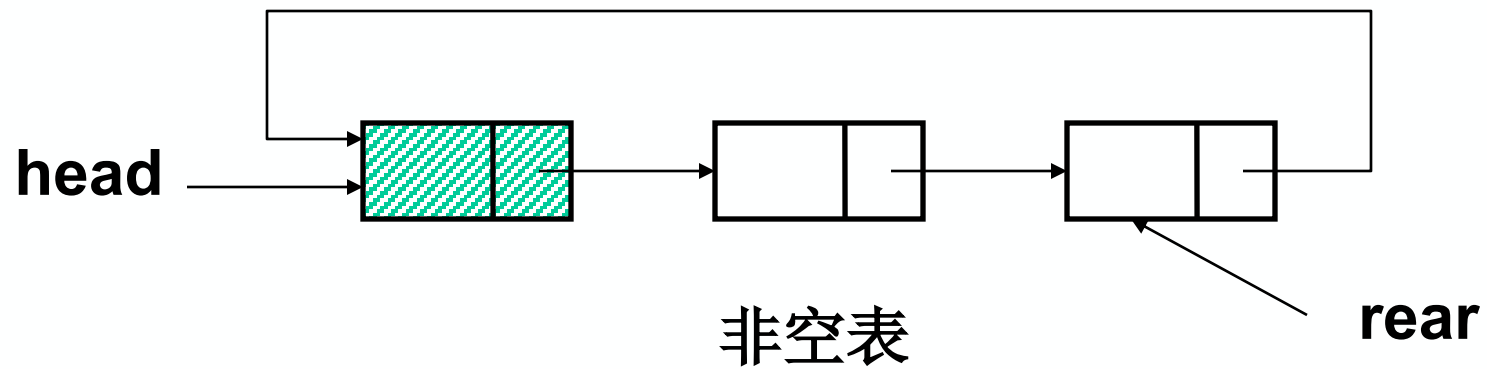
(a) 非空单循环链表



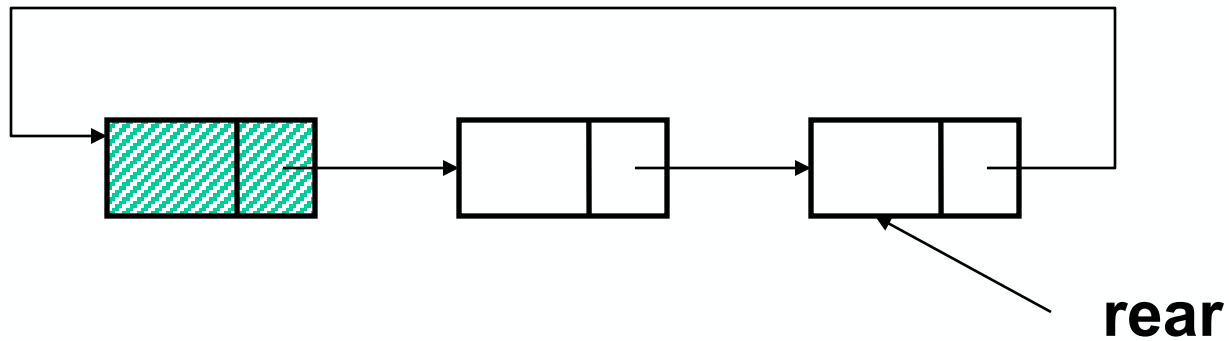
(b) 空表

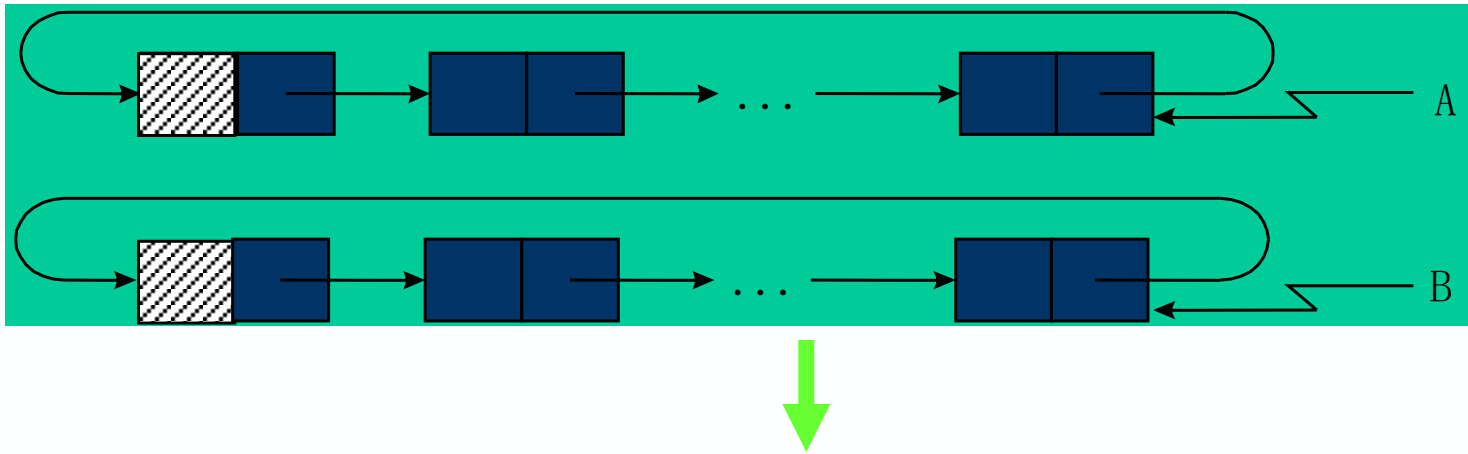
$L \rightarrow \text{next} = L$

## 循环链表



## 采用尾指针的循环链表

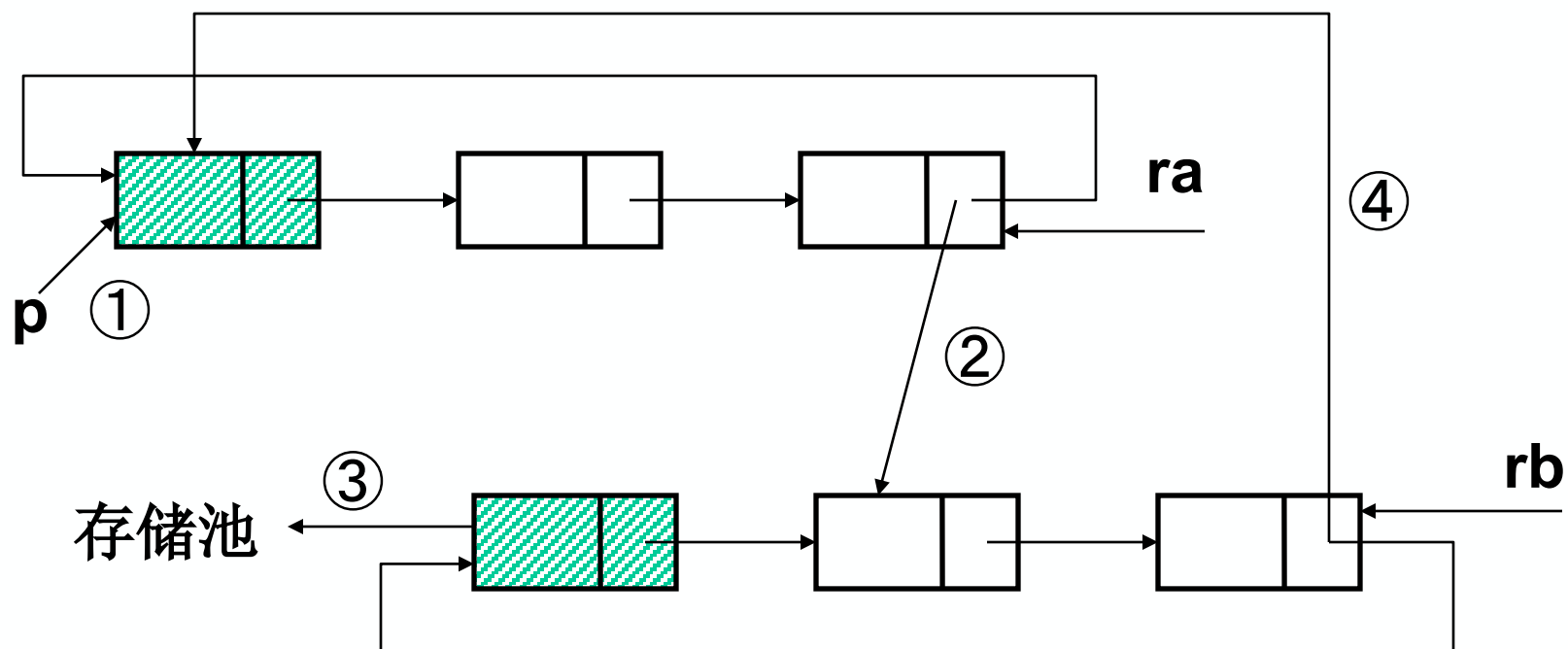




**$p = B \rightarrow next \rightarrow next;$      $B \rightarrow next = A \rightarrow next;$**

**$A \rightarrow next = p$**

## 两循环链表的链接



## 两循环链表的链接

```
linklist *CONNECT(linklist *ra,linklist *rb)
```

```
{ linklist *p;
```

```
    p=ra->next;           ①
```

```
    ra->next=rb->next->next; ②
```

```
    free(rb->next);         ③
```

```
    rb->next=p;             ④
```

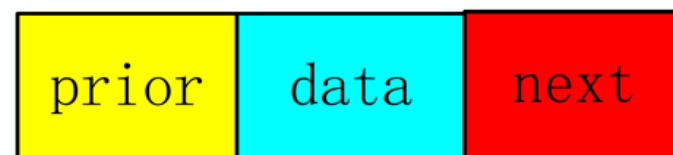
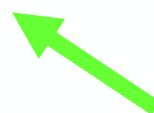
```
    return rb;
```

```
}
```

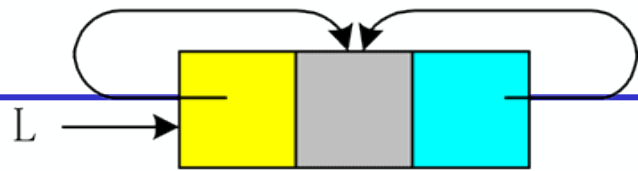


## 2.3.4 双向链表

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





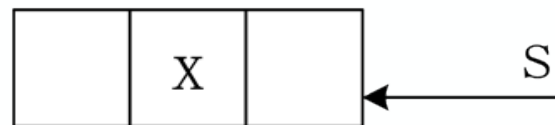


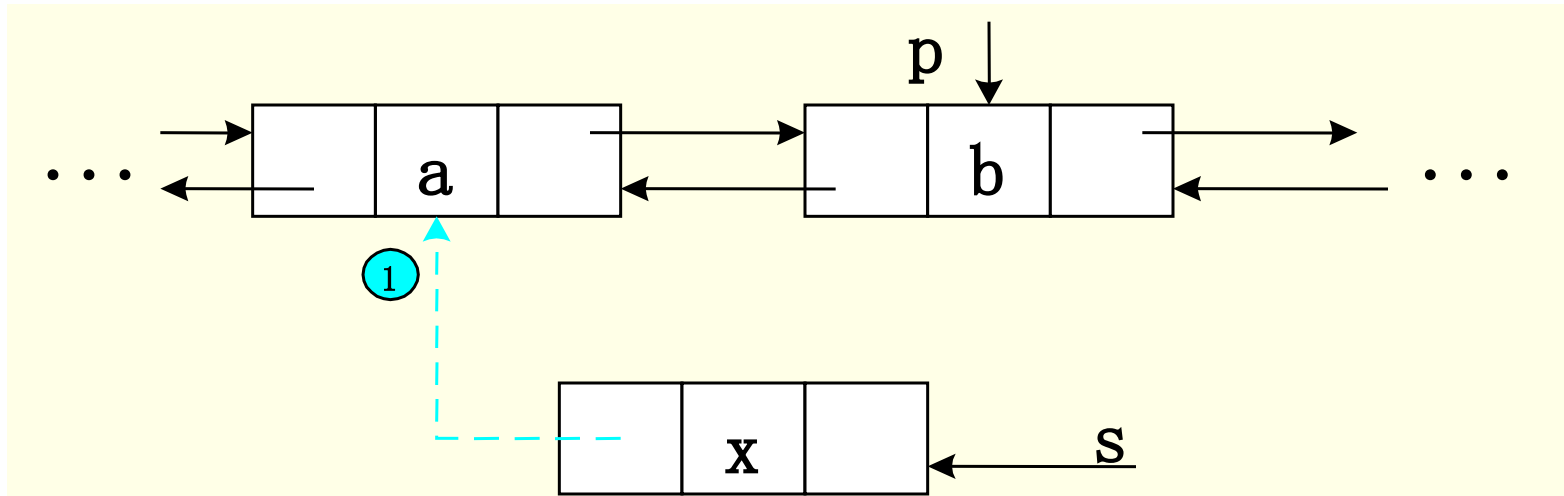
(a) 空双向循环链表

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

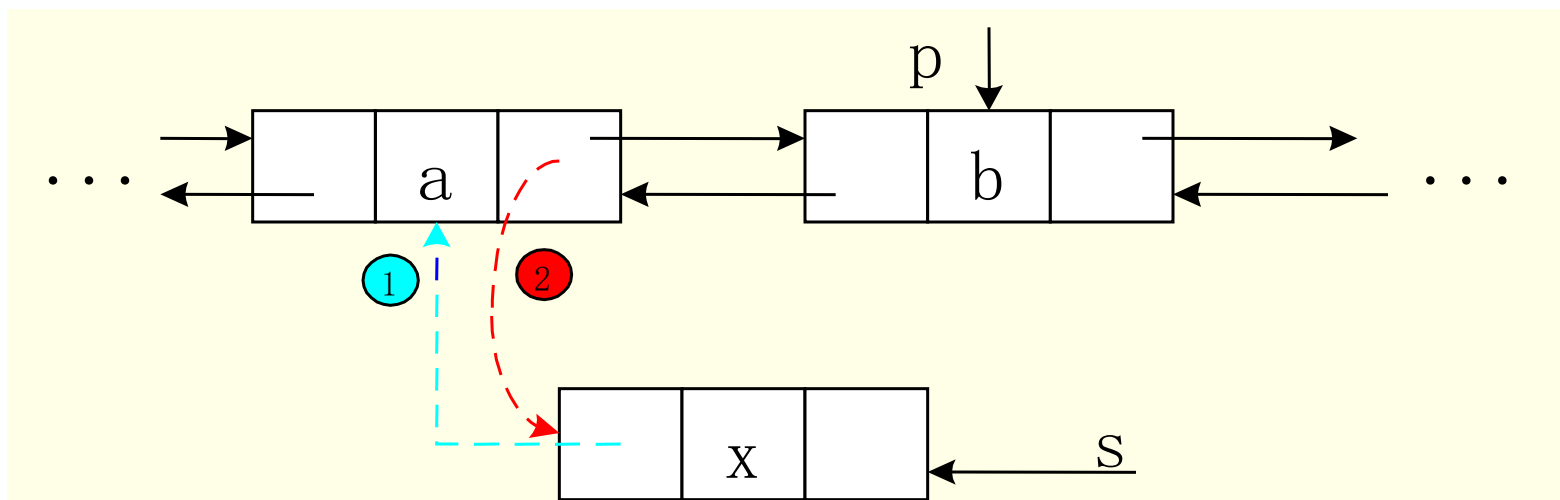
(b) 双向循环链表

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





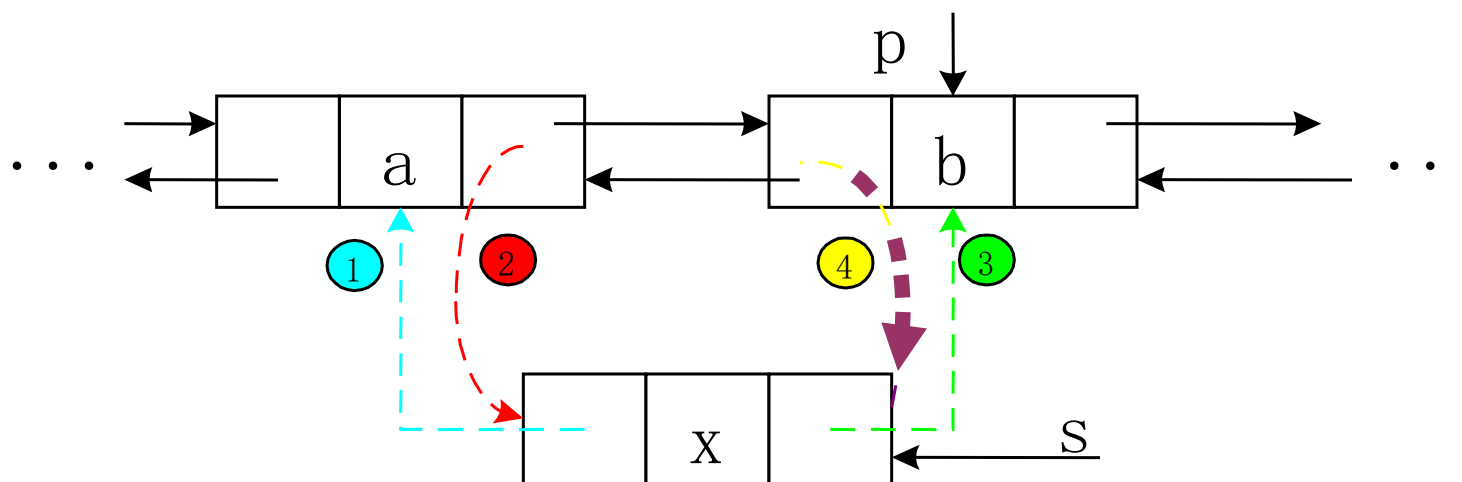
**1.  $s \rightarrow \text{prior} = p \rightarrow \text{prior};$**



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

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

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



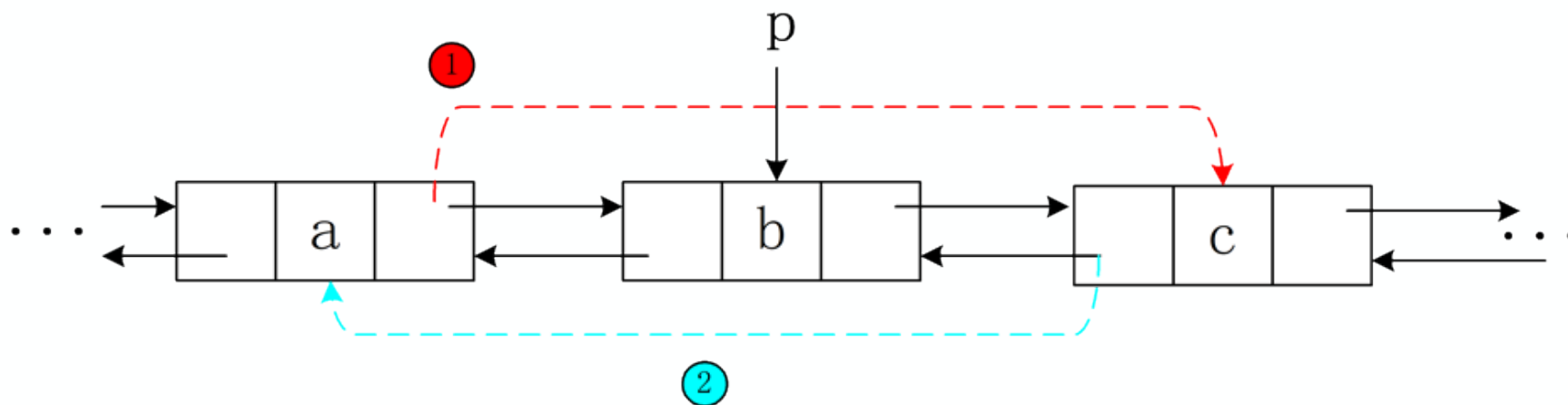
1.  $s \rightarrow \text{prior} = p \rightarrow \text{prior};$
2.  $p \rightarrow \text{prior} \rightarrow \text{next} = s;$
3.  $s \rightarrow \text{next} = p;$
4.  $p \rightarrow \text{prior} = s;$

```
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;  
}
```





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



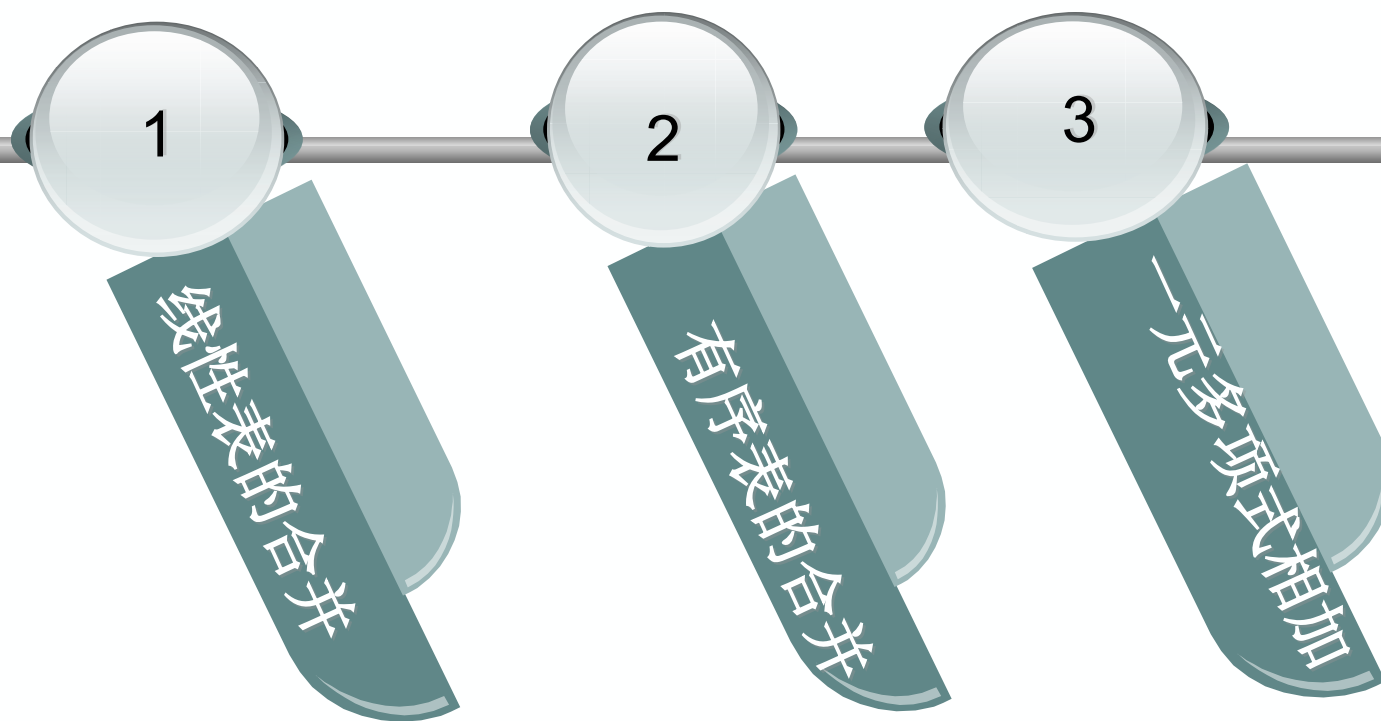
1.  $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next};$

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

```
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;  
}
```



## 2.4 线性表的应用





## 2.4.1 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)**

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

```
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);  
    }  
}
```



## 2.4.2 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)**



# 【算法思想】 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】 一有序的顺序表合并

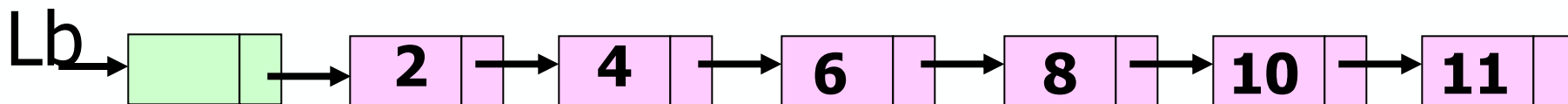
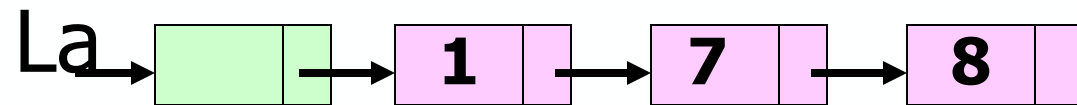
```
void MergeList_Sq(SqList LA, SqList LB, SqList &LC){
    pa=LA.elem; pb=LB.elem; //指针pa和pb的初值分别指向两个表的第一个元素
    LC.length=LA.length+LB.length; //新表长度为待合并两表的长度之和
    LC.elem=new ElemType[LC.length]; //为合并后的新表分配一个数组空间
    pc=LC.elem; //指针pc指向新表的第一个元素
    pa_last=LA.elem+LA.length-1; //指针pa_last指向LA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插入到 LC表的最后
        else *pc++=*pb++;    }
    while(pa<=pa_last) *pc++=*pa++;
        //LB表已到达表尾， 依次将LA的剩余元素插入LC表的最后
    while(pb<=pb_last) *pc++=*pb++;
        //LA表已到达表尾， 依次将LB的剩余元素插入LC表的最后
```

$T(n)=$

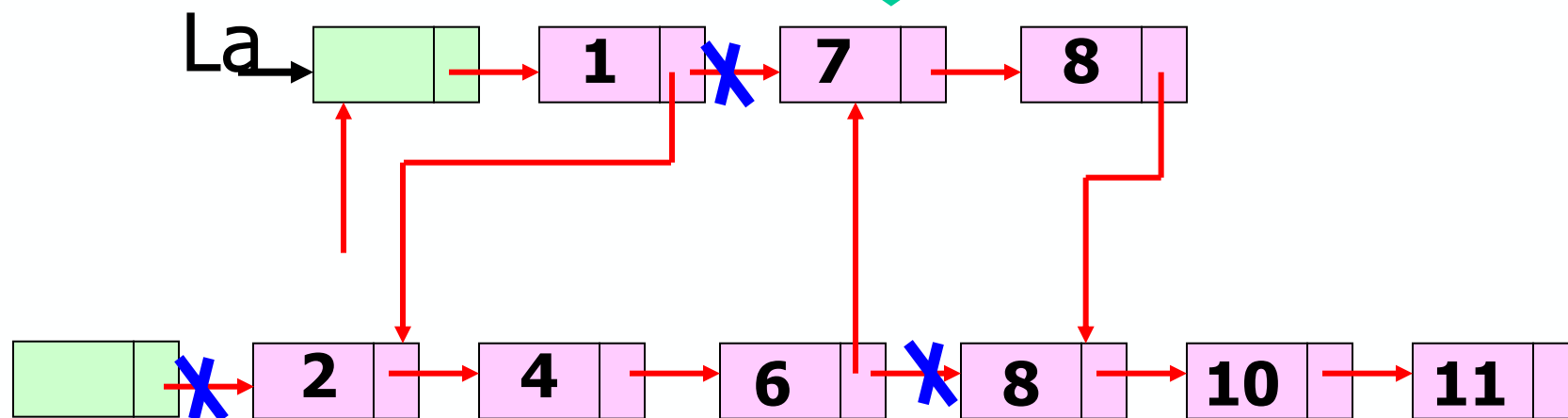
$S(n)=$

# 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。



合并后

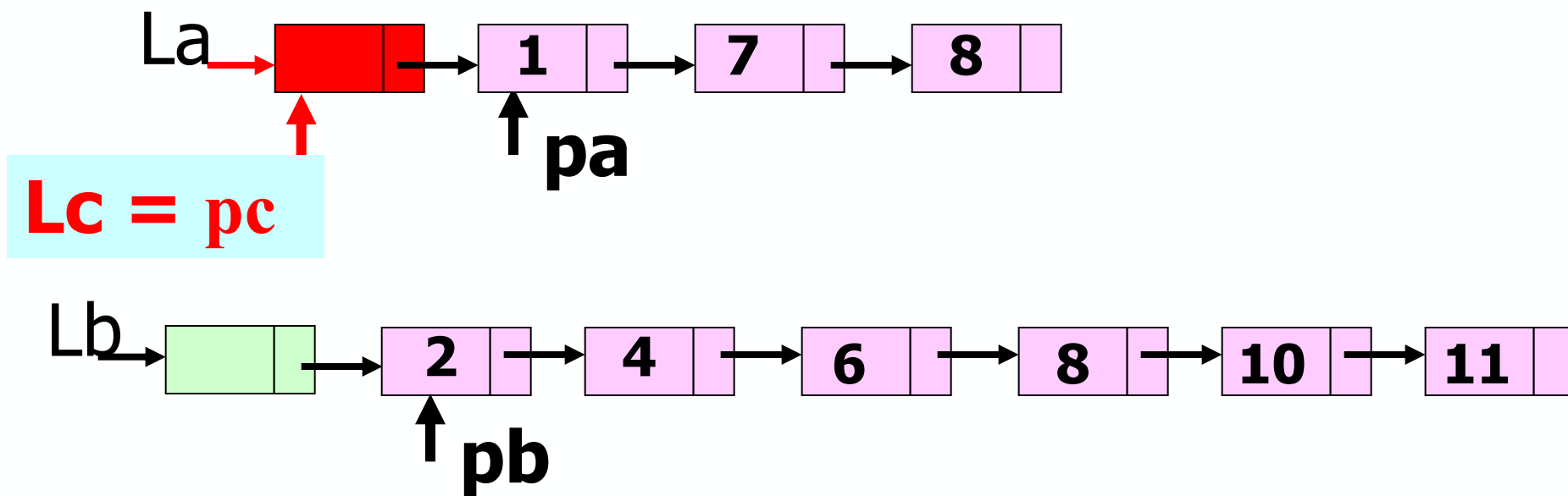


(1)  $L_c$  指向  $L_a$

(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

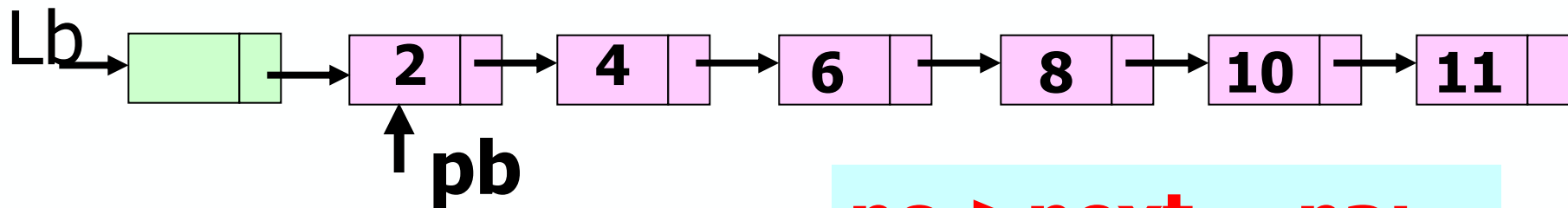
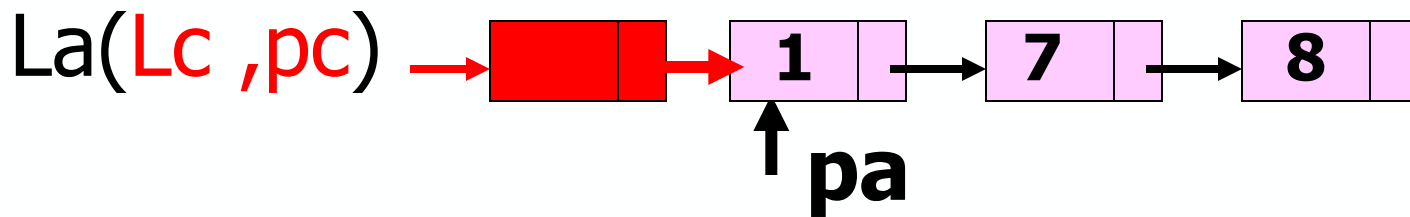
(3) 继续将  $L_a$  或  $L_b$  其中一个表的剩余结点插入在  $L_c$  表的最后

(4) 释放  $L_b$  表的表头结点



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

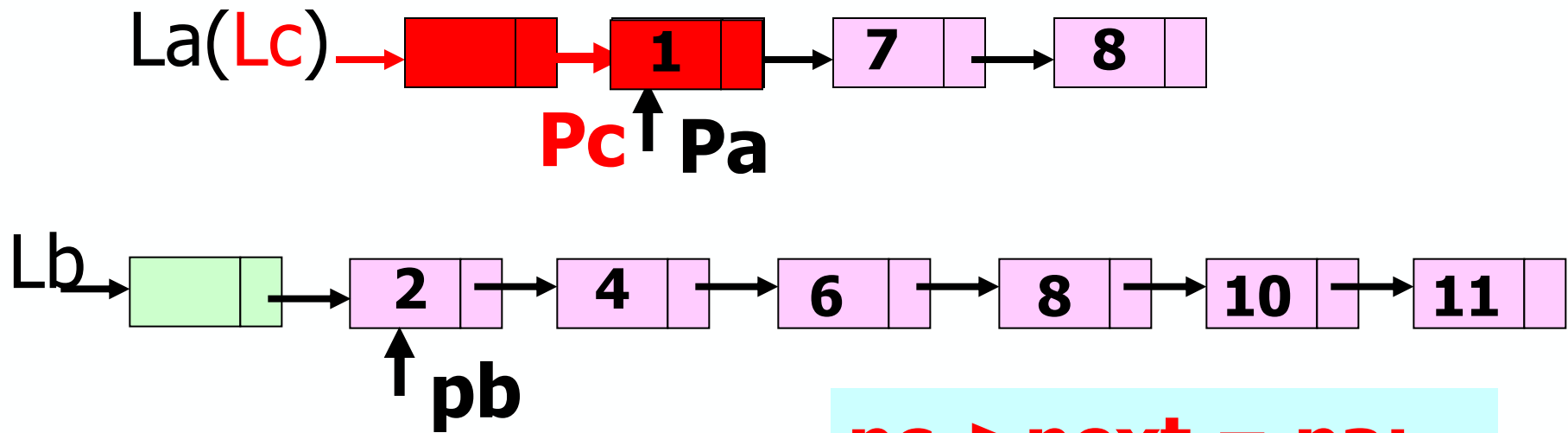
与数据结构



$pc \rightarrow next = pa;$

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

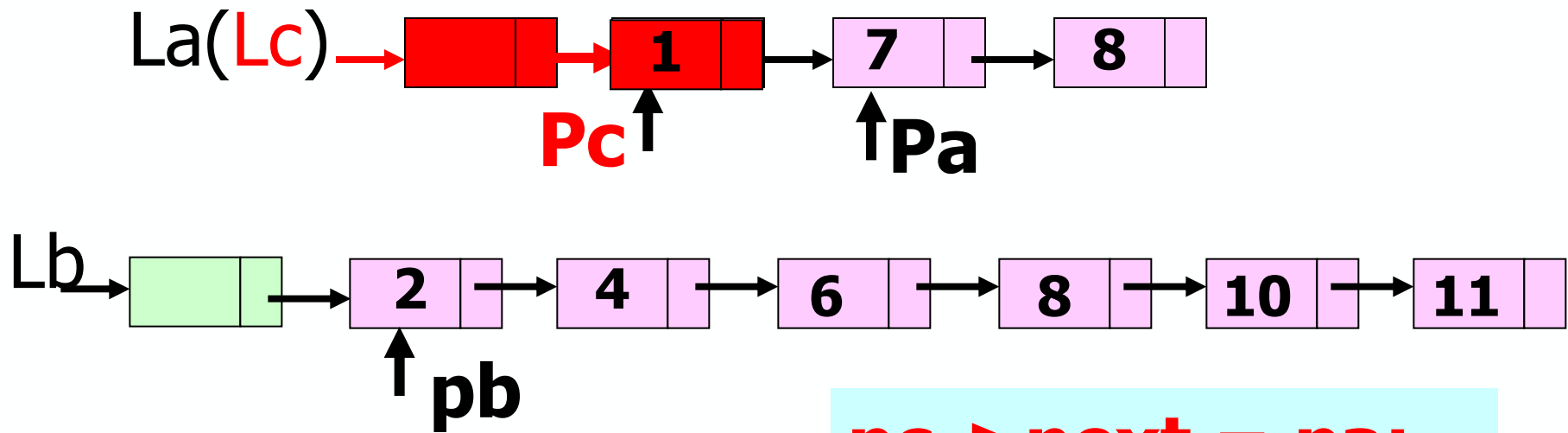
与数据结构



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



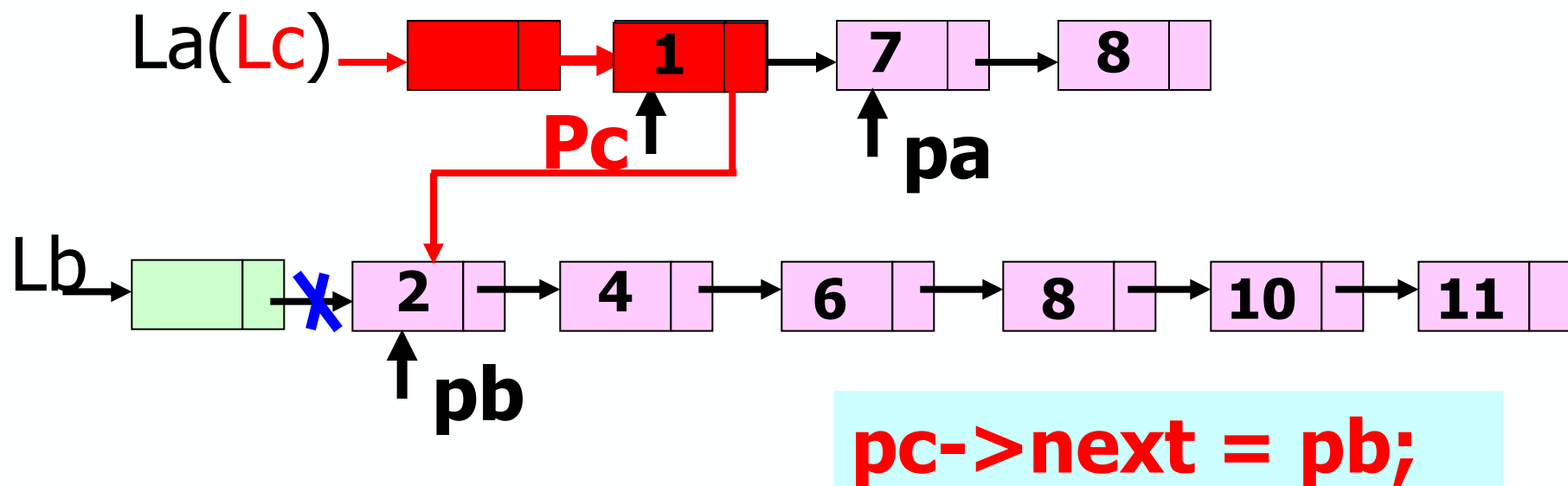
# 有序链表合并( $pa \rightarrow data \leq pb \rightarrow data$ )



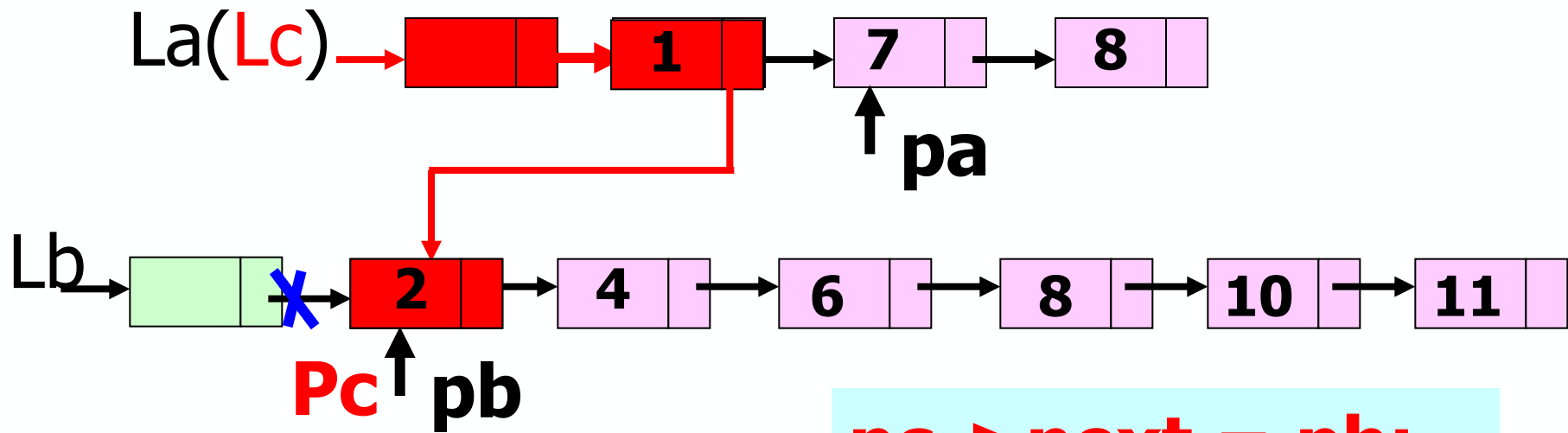
```
pc->next = pa;  
pc = pa;  
pa = pa->next;
```

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

与数据结构

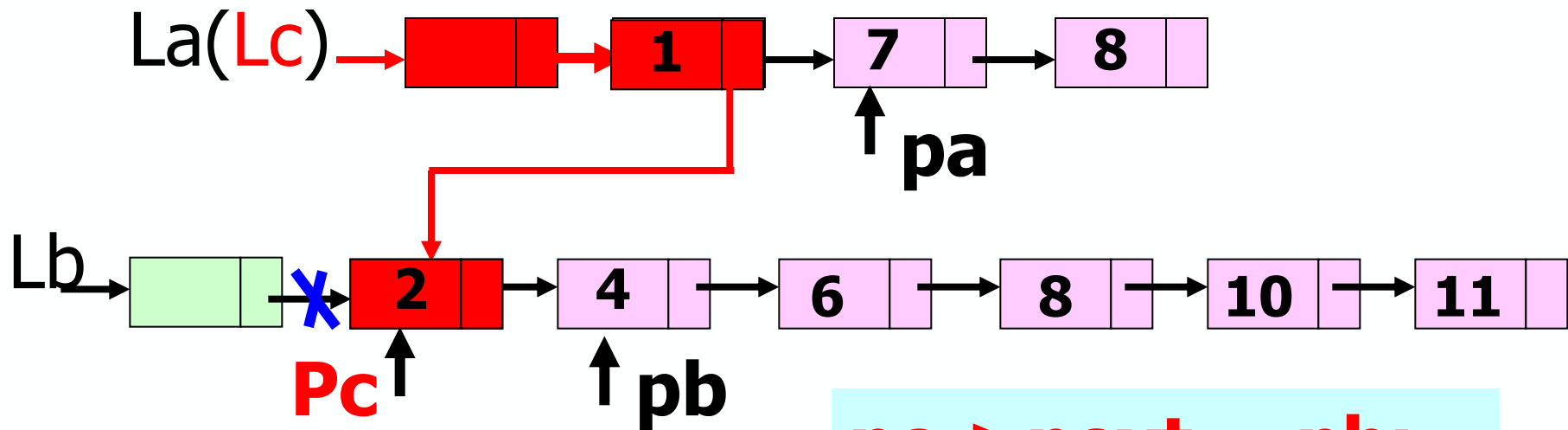


# 有序链表合并( $pa \rightarrow data > pb \rightarrow data$ )

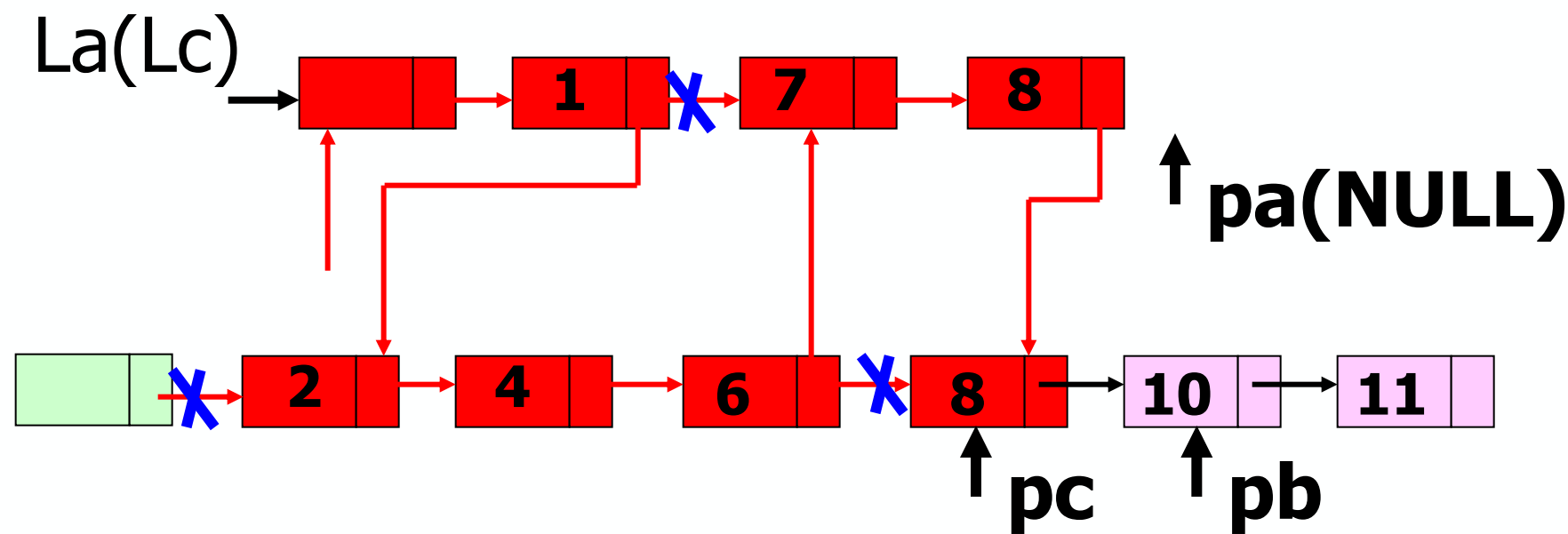


```
pc->next = pb;  
pc = pb;
```

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

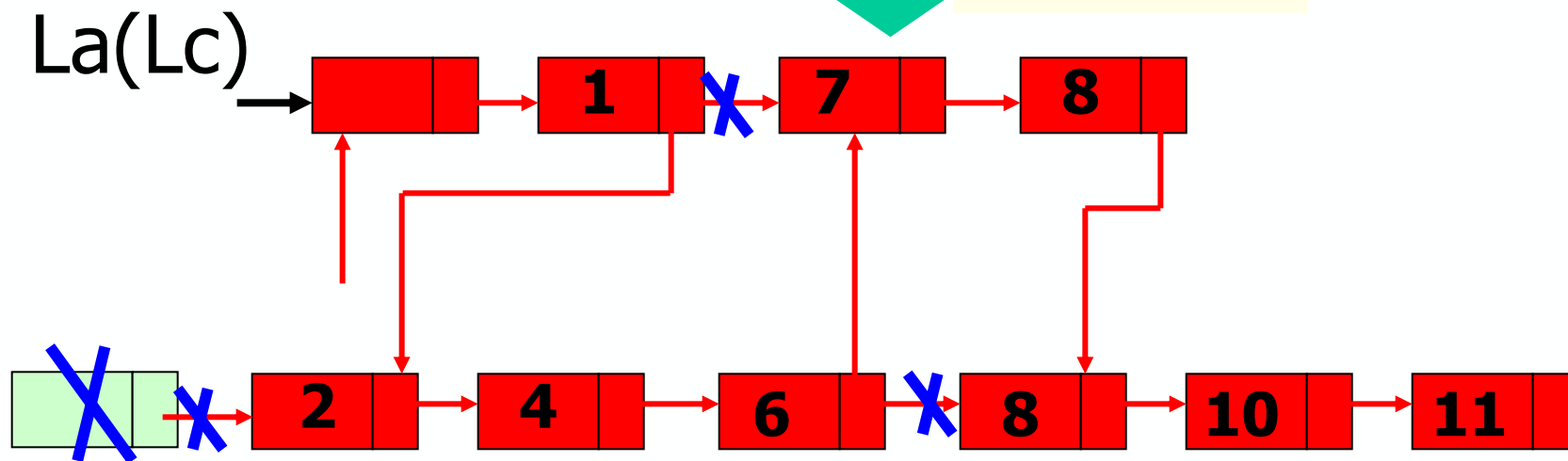


```
pc->next = pb;  
pc = pb;  
pb = pb->next;
```



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

合并后



**delete Lb;**

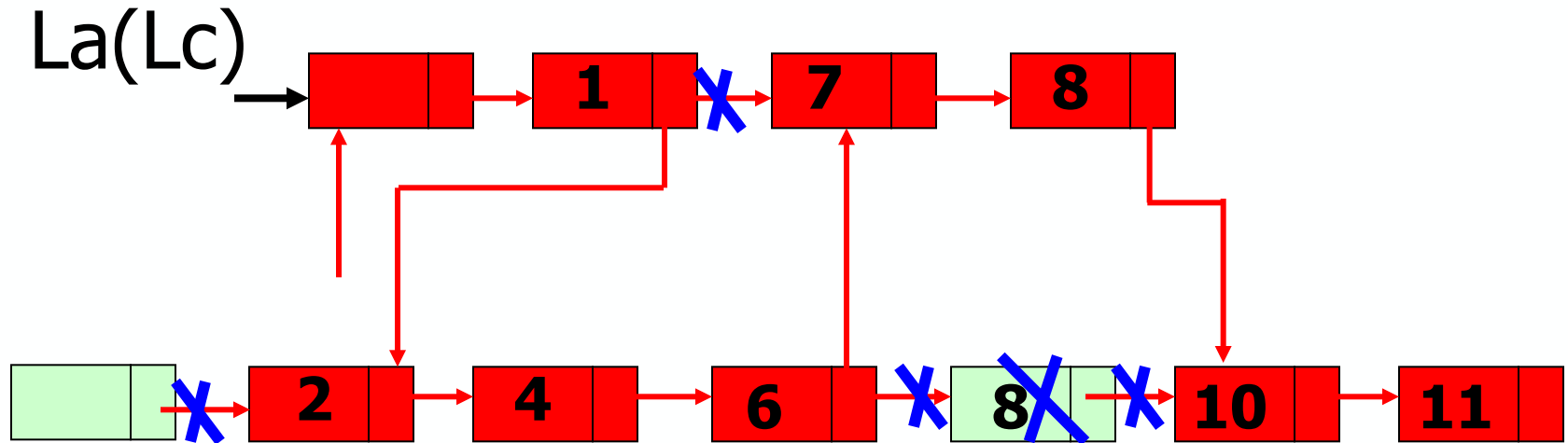
# 【算法描述】 一 有序的链表合并

```
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)=**

**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**

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

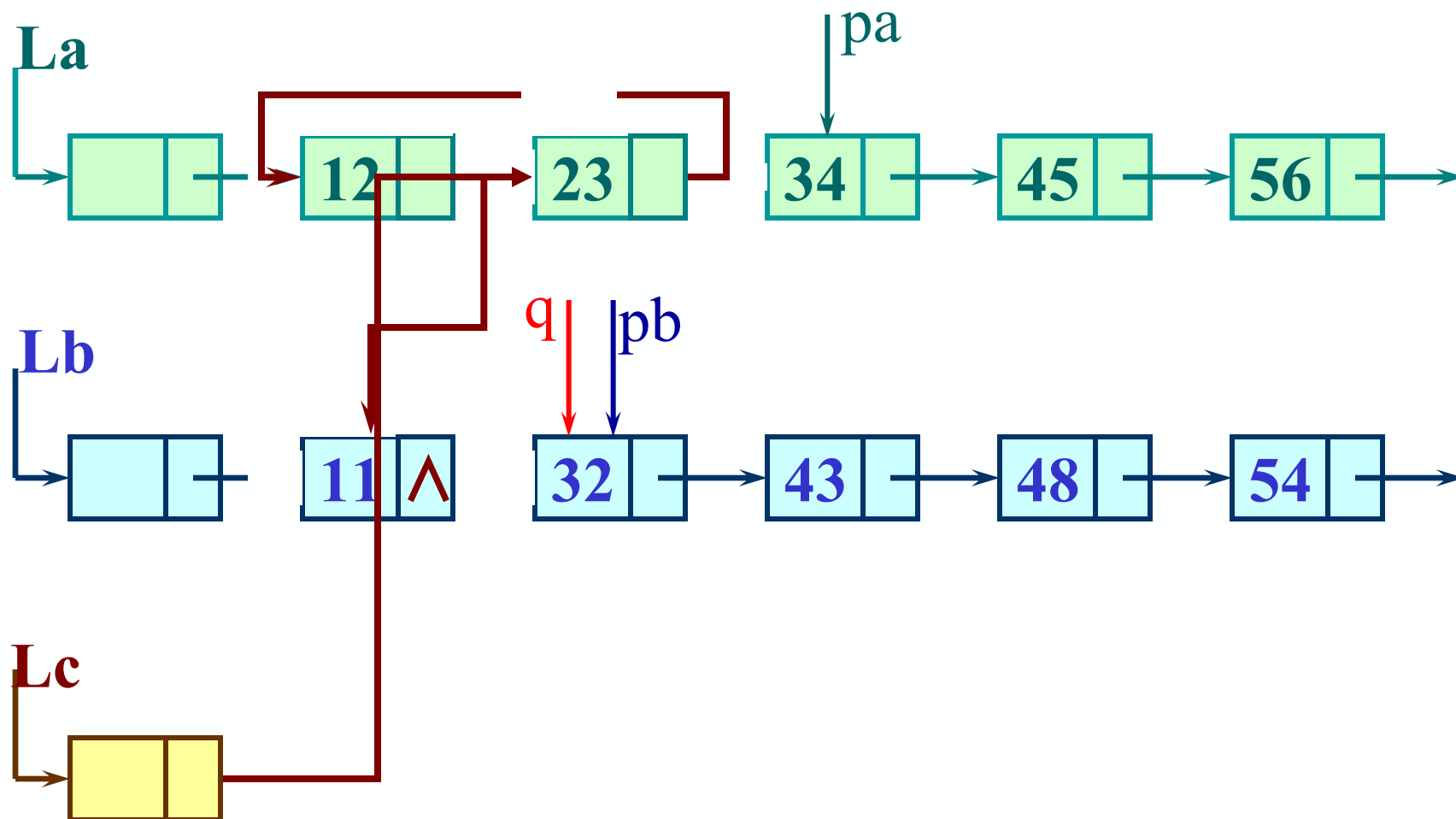


(2) 将两个**非递减**的有序链表合并为一个**非递增**的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。表中**允许有重复**的数据。

## 【算法思想】

- (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

## 第（2）题实现过程动态演示



### 一元多项式

在计算机中，可以用一个线性表来表示：

$$P = (p_0, p_1, \dots, p_n)$$

但是对于形如

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

的多项式，上述表示方法是否合适？

一般情况下的一元n次多项式可写成

$$P_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_mx^{e_m}$$

其中： $p_i$  是指数为 $e_i$  的项的非零系数，

$$0 \leq e_1 < e_2 < \dots < e_m = n$$

可以用下列线性表表示：

$$((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$$

例如:

$$P_{999}(x) = 7x^3 - 2x^{12} - 8x^{999}$$

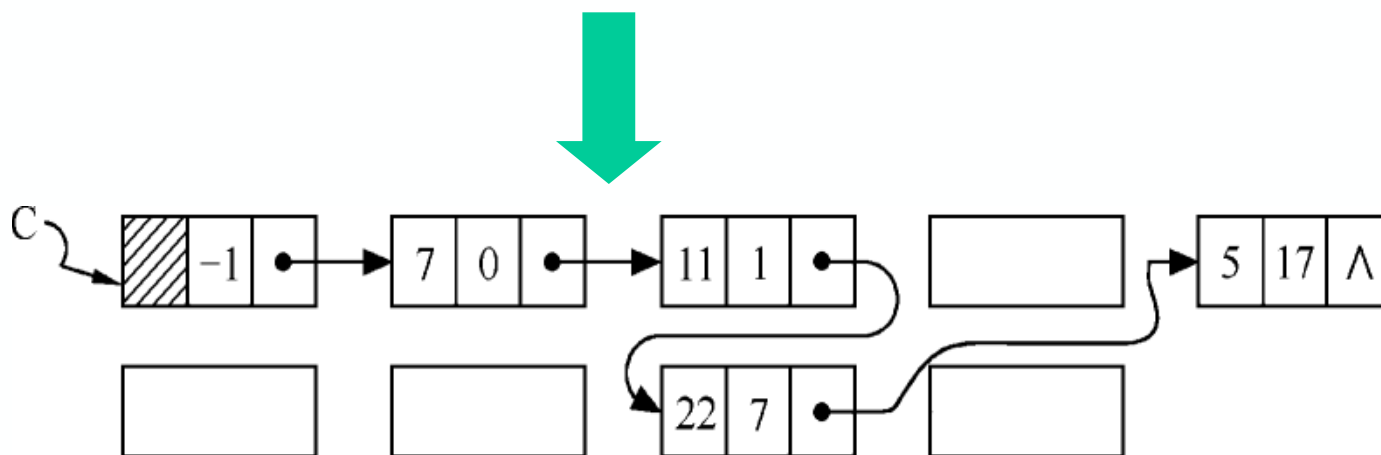
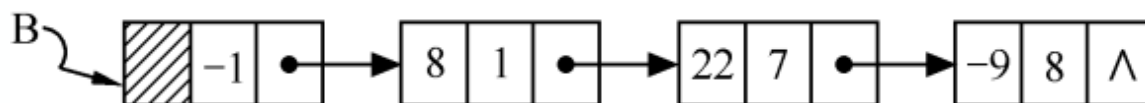
可用线性表

$((7, 3), (-2, 12), (-8, 999))$  表示

## 2.4.3 一元多项式的表示及相加

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

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





抽象数据类型一元多项式的定义如下：

ADT Polynomial {

**数据对象：**

$D = \{ a_i \mid a_i \in \text{TermSet}, i=1,2,\dots,m, m \geq 0$

$\text{TermSet}$  中的每个元素包含一个表示系数的实数和表示指数的整数 }

**数据关系：**

$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D,$

且  $a_{i-1}$  中的指数值  $<$   $a_i$  中的指数值  $i=2,\dots,n\}$

# 基本操作:

## **CreatPolyn ( &P, m )**

**操作结果:** 输入  $m$  项的系数和指数,  
建立一元多项式  $P$ 。

## **DestroyPolyn ( &P )**

**初始条件:** 一元多项式  $P$  已存在。

**操作结果:** 销毁一元多项式  $P$ 。

## **PrintPolyn ( &P )**

**初始条件:** 一元多项式  $P$  已存在。

**操作结果:** 打印输出一元多项式  $P$ 。

## PolynLength( P )

初始条件：一元多项式 P 已存在。

操作结果：返回一元多项式 P 中的项数。

## AddPolyn ( &Pa, &Pb )

初始条件：一元多项式 Pa 和 Pb 已存在。

操作结果：完成多项式相加运算，即：

$Pa = Pa + Pb$ , 并销毁一元多项式 Pb。

## SubtractPolyn ( &Pa, &Pb )

... ..

## MultiplyPolyn(&Pa, &Pb )

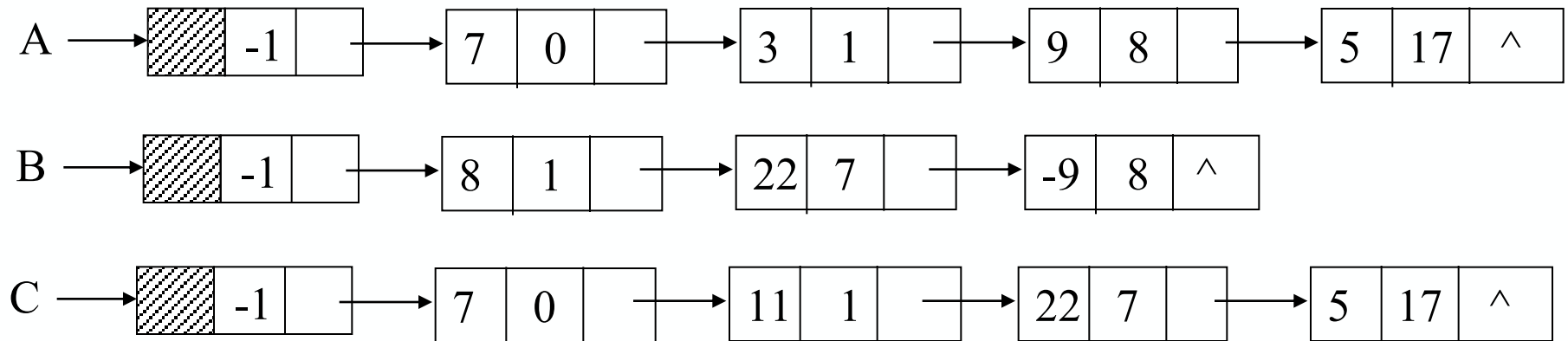
... ..

} ADT Polynomial

## 抽象数据类型一元多项式的实现：

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

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



```
Void CreatPolyn (polynomail &P, int m ) {  
    // 输入m项的系数和指数，建立表示一元多项式的有序链表P  
P=new PNode; P->next=NULL;  
for ( i=1; i<=m; ++i )  
{    s=new PNode; cin>>s->coef>>s->expn;  
        Pre=P;  
        q=P->next;  
        while (q&&q->expn<s->expn )  
            {        pre=q; q=q->next;}  
        s->next=q;  
        Pre->next=s;  
}
```

核心算法AddPolyn是把分别由pa和pb所指的两个多项式相加，结果为pa所指的多项式。相加时，首先设两个指针变量qa和qb分别从多项式的首项开始扫描，比较qa和qb所指结点指数域的值，可能出现下列三种情况之一：

(1)  $qa \rightarrow \text{exp}$  大于  $qb \rightarrow \text{exp}$ , 摘取qb指针所指结点插入到“和多项式”链表中去

(2)  $qa \rightarrow \text{exp}$  等于  $qb \rightarrow \text{exp}$ , 则将其系数相加。

✓若相加结果不为零，修改qa所指结点的系数值，并释放qb所指结点。

✓否则同时释放qa和qb所指结点。然后qa、qb继续向后扫描。

(3)  $qa \rightarrow exp$  小于  $qb \rightarrow exp$ , 摘取  $qa$  指针所指结点插入到“和多项式”链表中去。

扫描过程一直进行到  $qa$  或  $qb$  有一个为空为止, 然后将有余结点的链表接在结果链表上。所得  $pa$  指向的链表即为两个多项式之和。



```
void AddPolyn(Polynomial &pa, Polynomial &pb)
{
p1=pa->next; p2=pb->next;
p3=pa;
while (p1&& p2)
{
if (p1->expn==p2->expn){
sum=p1->coef+p2->coef;
if (sum!=0){
p1->coef=sum;
p3->next=p1; p3=p1;
p1=p1->next;
r=p2; p2=p2->next; delete r;
}
else {
r=p1; p1=p1->next;delete r;
r=p2; p2=p2->next;delete r;
}}
}
```

```
else if (p1->expn < p2->expn)
```

```
{  
    p3->next = p1;  
    p3 = p1;  
    p1 = p1->next;  
}
```

```
else
```

```
{  
    p3->next = p2;  
    p3 = p2;  
    p2 = p2->next;  
}  
}
```

```
p3->next = p1 ? p1 : p2;
```

```
delete pb;
```

```
}
```

- 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。

- 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

# Homework-1

- 习题

2.(10) 要求自己上机调试.

# Homework-2

- 2.算法设计题  
(6)(7)

# Homework-3

(1) 将两个**递增**的有序链表合并为一个**递增**的有序链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。表中**不允许有重复**的数据。



(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。要求结果链表仍使用原来两个链表的存储空间，不另外占用其他的存储空间。表中**允许有重复**的数据。

(3) 设计一个算法，通过一趟遍历在单链表中确定**值最大的**结点。

**【算法思想】**类似于求n个数中的最大数

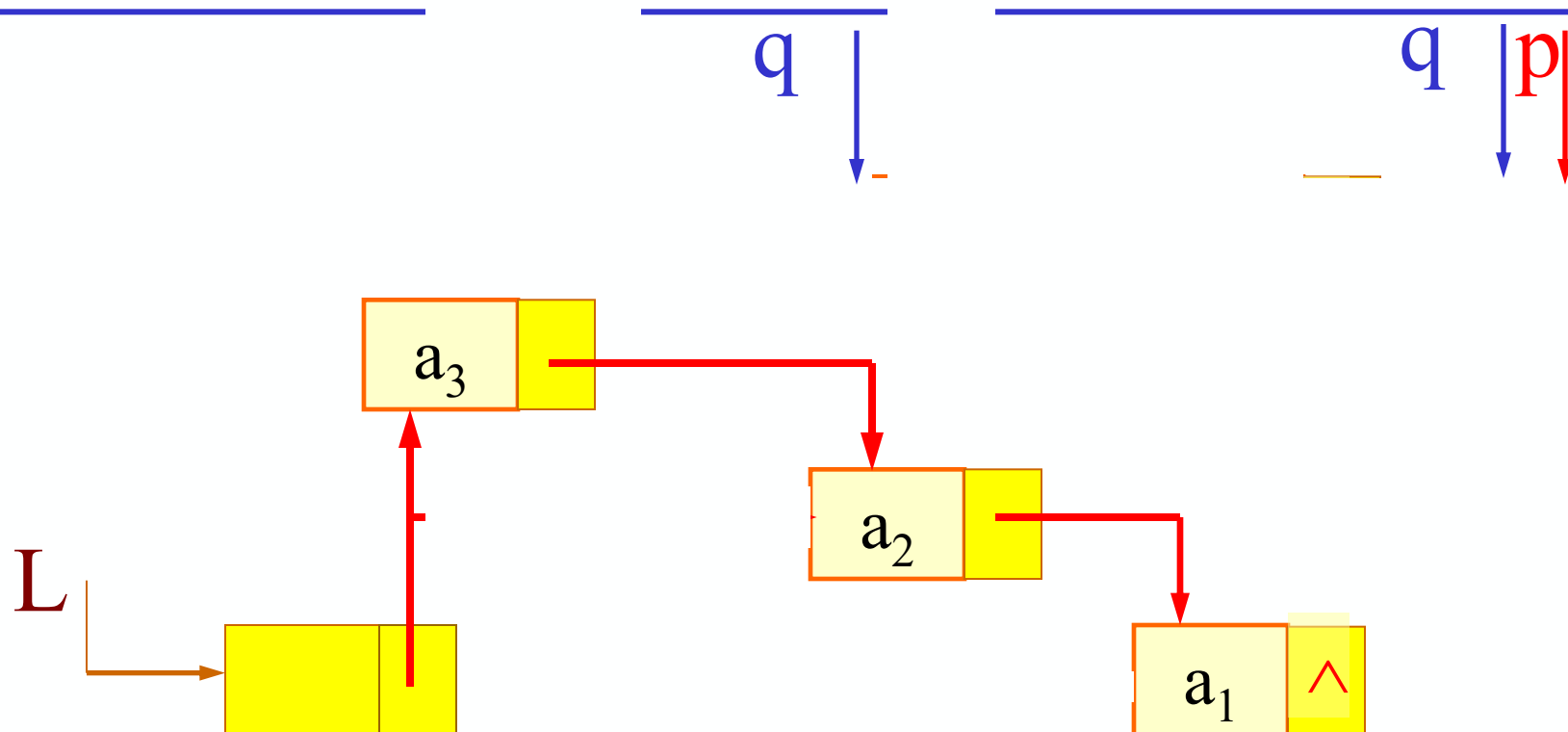
- ✓可假设第一个结点最大，用指针pmax指向。
- ✓然后用pmax依次和后面的结点进行比较，发现大者则用pmax指向该结点。
- ✓这样将链表从头到尾遍历一遍时，pmax所指向的结点就是最大者。

其中的比较语句形式如下：

```
if (p->data > pmax->data) pmax=p;
```

(4) 设计一个算法，通过一趟遍历，将链表中所有结点的链接方向逆转，且仍利用原表的存储空间。

**【算法思想】** 从首元结点开始，逐个地把链表L的当前结点p插入新的链表头部



- ✓标志后继结点 $q$
- ✓修改指针（将 $p$ 插入在头结点之后）
- ✓重置结点 $p$ （ $p$ 重新指向原表中后继）