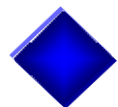
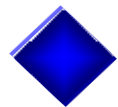


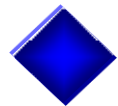
第二章 线性表



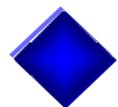
实例



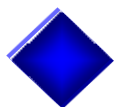
线性表的逻辑结构



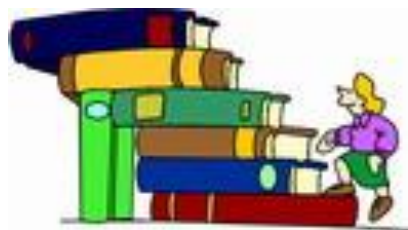
线性表的顺序存储结构



线性表的链式存储结构



线性表的应用举例



本章小结

线性表实例



图书馆的书目自动检索系统

001	高等数学	樊映川	S01
002	理论力学	罗远祥	L01
003	高等数学	华罗庚	S01
004	线性代数	栾汝书	S02
.....

书目文件

索引表

高等数学	001, 003.....
理论力学	002,
线性代数	004,
.....

樊映川	001, ...
华罗庚	002,
栾汝书	004,
.....

✧ 各个数据(书目信息)是**一对一的线性关系**

✧ 适于使用**线性表**作为数学模型

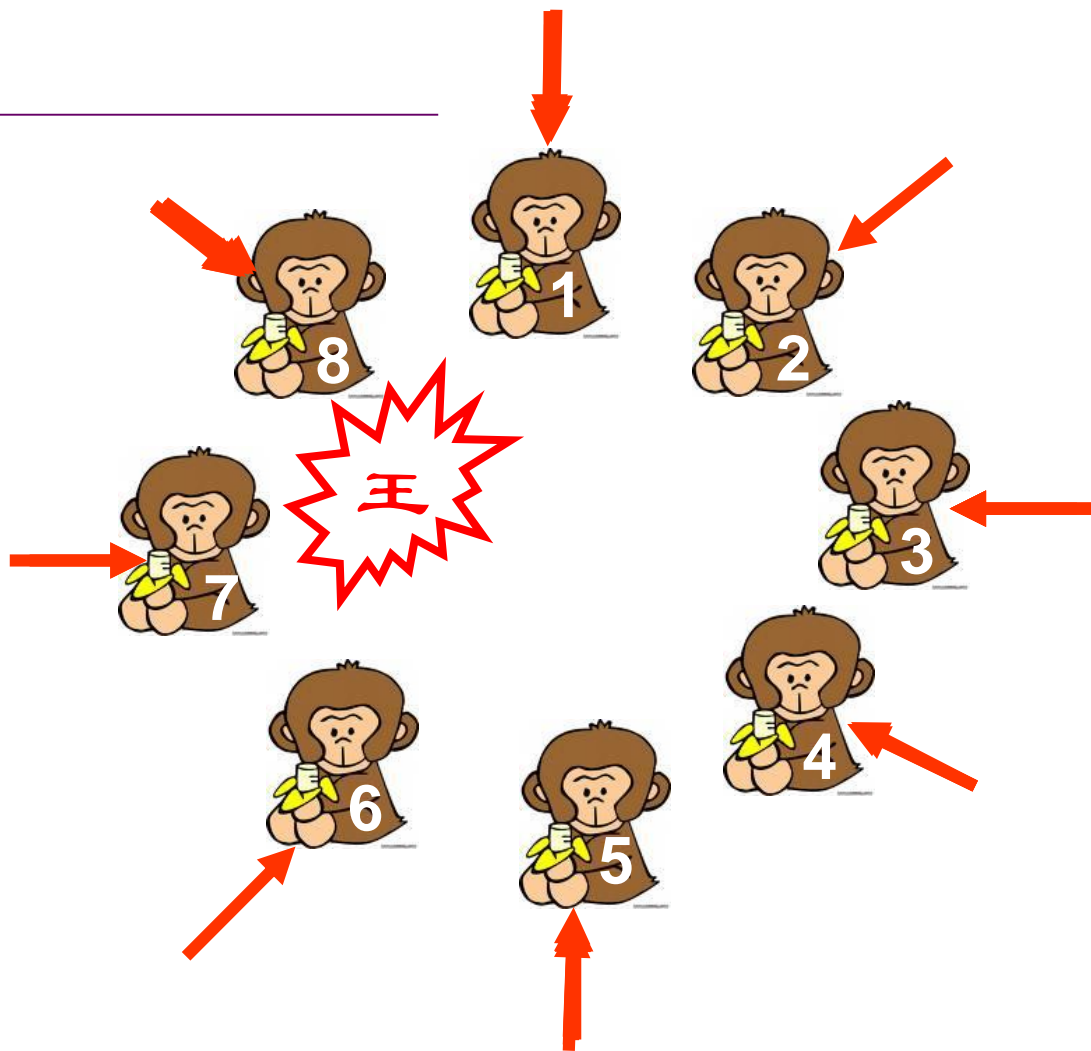
线性表实例

猴子选大王

$n=8$

$s=3$

$m=4$



✧ 各个数据(猴子)是**一对一的线性关系**

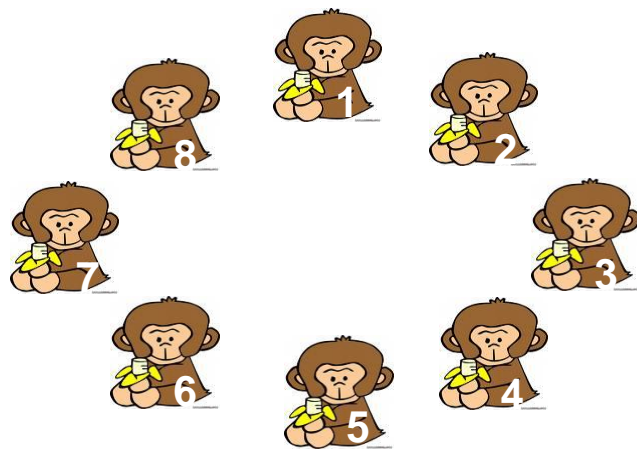
✧ 适于使用**线性表**作为数学模型

线性表特点

在数据元素的非空有限集中——

- ① 存在**唯一**的一个被称作“**第一个**”的数据元素
- ② 存在**唯一**的一个被称作“**最后一个**”的数据元素
- ③ 除第一个外，每个数据元素**只有一个前驱**
- ④ 除最后一个外，每个数据元素均**只有一个后继**

001	高等数学	樊映川	S01
002	理论力学	罗远祥	L01
003	高等数学	华罗庚	S01
004	线性代数	栾汝书	S02
.....



Back

线性表的逻辑结构

🕸 线性表—— n 个数据元素的有限序列。

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

🍪 英文字母表(A,B,C,...,Z)是一个线性表

🍪 一组学生信息是一个线性表

学号	姓名	年龄
001	张三	18
002	李四	19
.....

🕸 特征：有限、序列、同构

✓ 元素个数 n ——表长度， $n=0$ 空表

✓ 当 $1 < i < n$ 时，

a_i 的直接前驱是 a_{i-1} ， a_1 无直接前驱

a_i 的直接后继是 a_{i+1} ， a_n 无直接后继

✓ 元素同构，且不能出现缺项

线性表的逻辑结构

✧ 抽象数据类型线性表的定义

ADT List

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

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

基本操作:

InitList(&L);

构造空的线性表L

DestroyList(&L);

销毁已有线性表L

ListLength(L);

求线性表L中的元素个数

GetElem(L, i, &e);

用e返回线性表L中第i个元素值

ListInsert(&L, i, e);

在线性表L第i个元素之前插入e

ListDelete(&L, i, &e);

删除线性表L中第i个元素

.....

} ADT List

Back

线性表的顺序存储结构

蜘蛛网图标 顺序表：

蜘蛛图标 用一组地址连续的存储单元存放的线性表称为~。

蜘蛛图标 元素地址计算方法：

- $LOC(a_i) = LOC(a_1) + (i-1) * L$

- 其中：L——每个元素占用的存储单元个数

$LOC(a_i)$ ——线性表第*i*个元素的地址

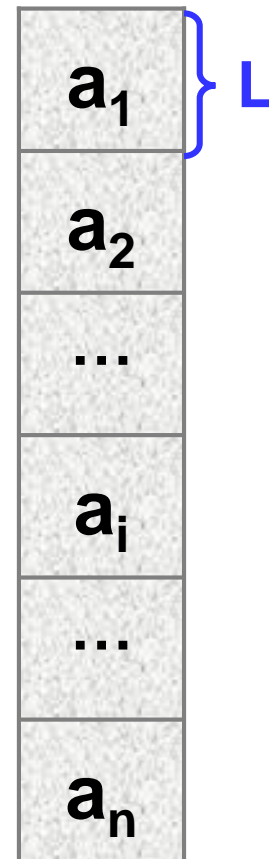
蜘蛛图标 特点：

- 逻辑相邻的数据元素存储位置也相邻

- 任一数据元素均可随机存取

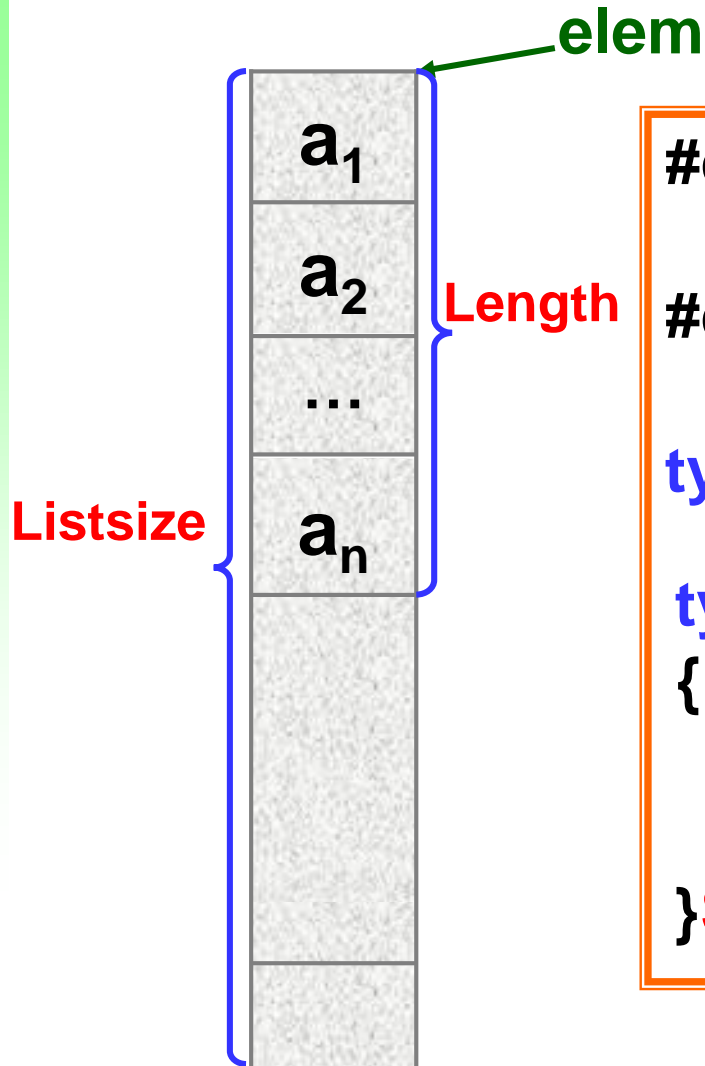
随机存取——对第*i*个元素，可计算其地址 $LOC(a_i)$ ，直接对其操作，无需从表头依次查找

蜘蛛图标 实现：一维数组或动态分配顺序存储结构



线性表的顺序存储结构

🕷 线性表的 动态分配 顺序存储结构




```
#define LIST_INIT_SIZE  100
                        //线性表存储空间的初始分配量
#define LISTINCREMENT  10
                        //线性表存储空间的分配增量
typedef int ElemType;  //元素类型

typedef struct
{
    ElemType *elem; //存储空间基址
    int length;      //长度
    int listsize;    //容量
} SqList;
```


线性表的顺序存储结构

`/*stdio.h中*/
#define NULL 0`

 动态分配相关库函数 (包含在 **stdlib.h** 或 **malloc.h** 中)

◆ 申请内存函数

void *malloc(unsigned size)

功能：申请长度为 **size** 个字节的内存空间

返回值：申请成功——返回分配内存空间的首地址；申请失败——返回 **NULL**

说明：需将返回值进行 强制转换

◆ 释放内存函数

void free(void *p)

功能：释放以 **p** 为首地址的内存空间

返回值：无

说明：释放的内存空间必须是 **malloc()** 申请的，否则可能破坏系统

例：申请长度为 **n** 的整型数组空间

```
int *p, n;
```

```
scanf("%d",&n);
```

```
p = (int *) malloc( n * sizeof(int) );
```

```
.....
```

```
free(p);
```

整型数组名为 **p**,
元素为 **p[0],....., p[n-1]**

malloc() 申请的存储空间
在程序中可以改大或改小

线性表的顺序存储结构

✍ 构造一个空顺序表L



```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef int ElemType;
typedef struct
{ ElemType *elem;
  int length;
  int listsize;
} SqList;
```

实现思路——

- ① 申请一段连续的存储空间
- ② L.length=0;
- ③ L.listsize= LIST_INIT_SIZE ;

线性表的顺序存储结构

✎ 构造一个空顺序表L

使用引用型参数的原因——
在子函数中改变了L的成员

```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef int ElemType;
typedef struct
{ ElemType *elem;
  int length;
  int listsize;
} SqList;
```

//常用宏定义及类型

```
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
```

```
typedef int Status;
```

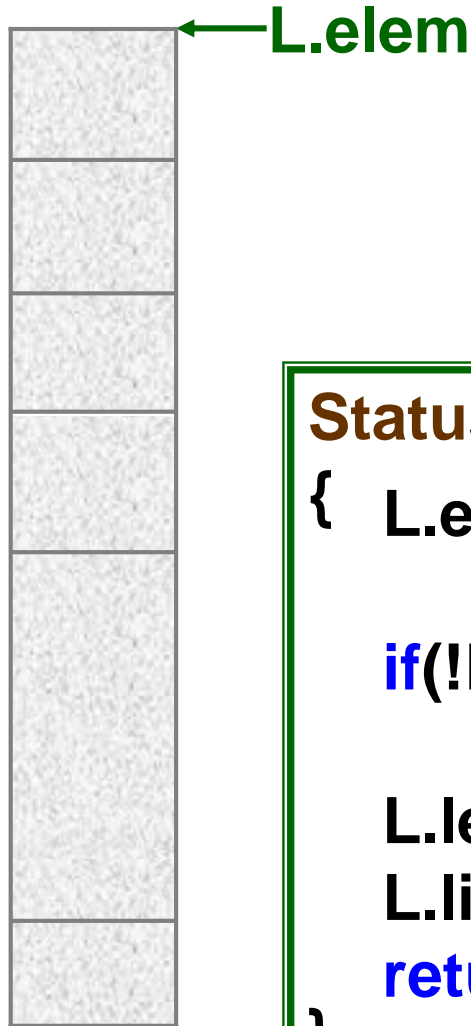
```
Status InitList_Sq(SqList &L)
```

```
{
```

```
}
```

线性表的顺序存储结构

构造一个空顺序表L



```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef int ElemType;
typedef struct
{ ElemType *elem;
  int length;
  int listsize;
} SqList;
```

Status InitList_Sq(SqList &L)

```
{ L.elem = (ElemType*) malloc (LIST_INIT_SIZE
                             *sizeof (ElemType));
  if(!L.elem)
    exit(OVERFLOW);
  L.length = 0;
  L.listsize = LIST_INIT_SIZE;
  return OK;
}
```

动态申请顺序表

判断申请是否成功

设置表长
及容量

$T(n)=O(1)$

线性表的顺序存储结构

查找

问题：在线性表L中查找是否存在数据元素e，返回第1个值与e相同的元素的位序，若不存在则返回0



查找成功

位序i

4

```
while( *p!=e && i<=L.length )  
{ i++; p++; }
```

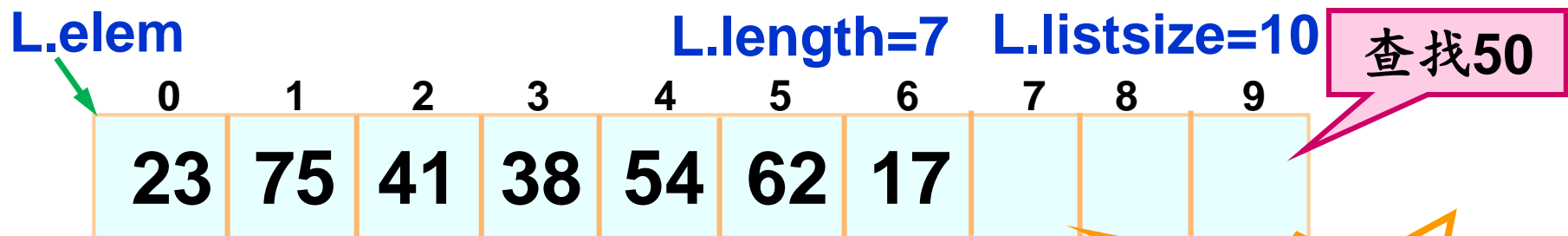
初始—— $p = L.elem;$
 $i = 1;$

查找成功时，
i为元素e的位序

线性表的顺序存储结构

查找

问题：在线性表L中查找是否存在数据元素e，返回第1个值与e相同的元素的位序，若不存在则返回0



查找失败

位序i

8

```
while( *p!=e && i<=L.length )  
{ i++; p++; }
```

初始—— p=L.elem;
i = 1;

查找失败
条件

$i > L.length$

线性表的顺序存储结构

查找

问题：在线性表L中查找是否存在数据元素e，返回第1个值与e相同的元素的位序，若不存在则返回0

```
int LocationElem( SqList L, ElemType e )
{
    int i=1;      位序，初始为1
    ElemType *p=L.elem;  从第1个元素开始查找
    while( *p !=e && i<=L.length )
    { i++; p++; }      查找
    if(i<=L.length)   判断是否存在
        return i;
    else
        return 0;
}
```

$T(n)=O(n)$

线性表的顺序存储结构

✎ 插入

问题：在线性表L的第 i ($1 \leq i \leq n+1$)个元素之前插入一个新的数据元素 e



插入一个元素后，线性表L的逻辑结构发生了什么变化？

使长度为 n 的线性表

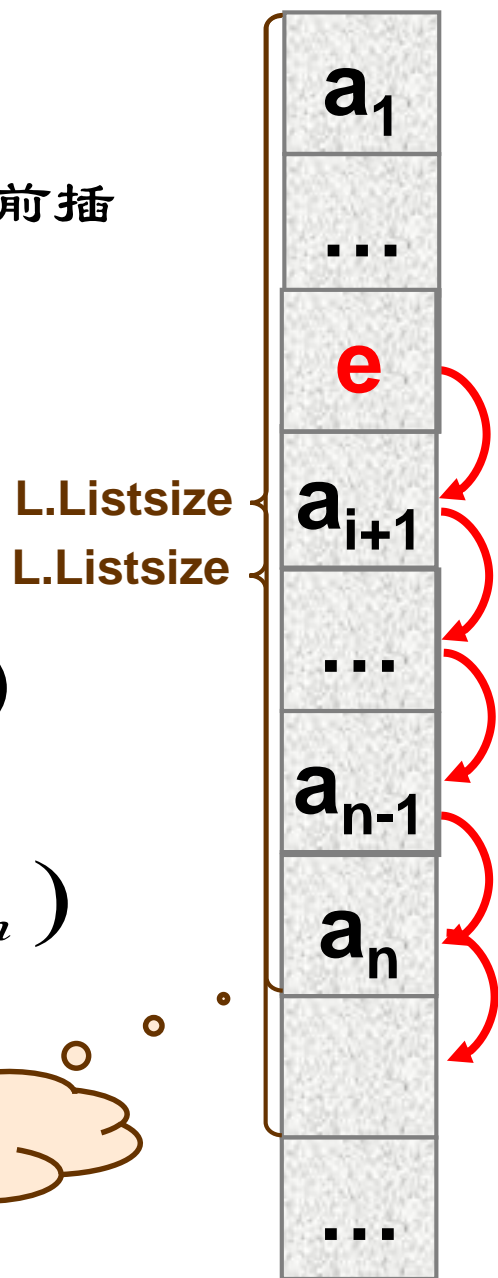
$(a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$

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 a_n 至 a_i 依次后移，再进行插入 e

移动 $n-i+1$ 次



线性表的顺序存储结构

✎ 插入

问题：在线性表 L 的第 i ($1 \leq i \leq n+1$) 个元素之前插入一个新的数据元素 e

元素后移——

先移动 a_n, \dots , 最后移动 a_i

$p = \&(L.elem[L.length-1]);$

$q = \&(L.elem[i-1]);$

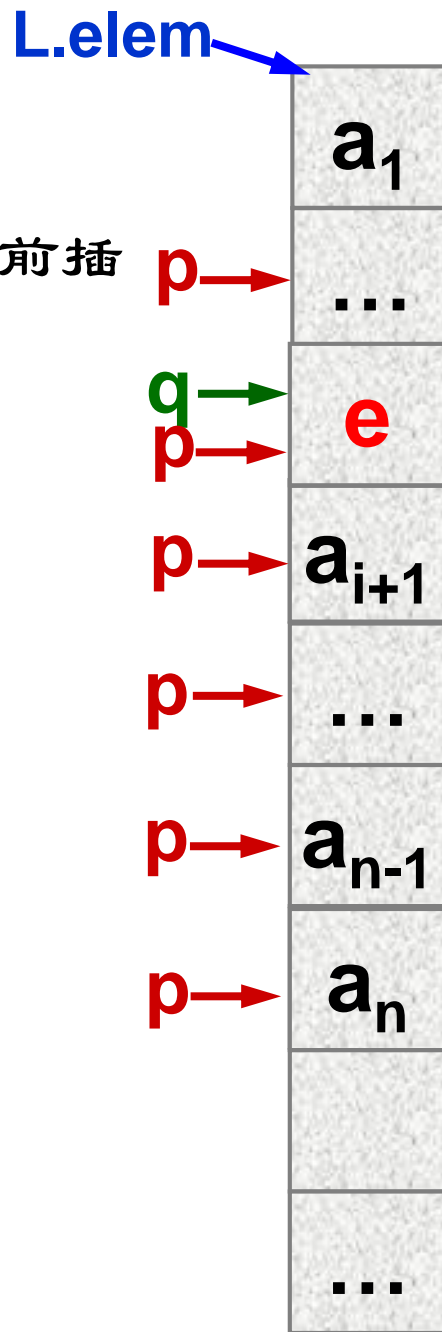
for(; $p \geq q$; $p--$)

元素后移

$*(p+1) = *p ;$

$*q = e ;$

插入



线性表的顺序存储结构

插入

问题：在线性表L的第 i ($1 \leq i \leq n+1$)个元素之前插入一个新的数据元素 e

```
Status ListInsert( SqList &L, int i, ElemType e )
```

```
{  ElemType *p, *q, *newbase;
```

```
  if( (i<1) || (i>L.length+1) )
```

判断 i 是否合法

```
    return (ERROR);
```

```
    .....
```

若存储空间已满，则需增大

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

```
    q=&(L.elem[i-1]);
```

```
    for( ; p>=q ; p-- )
```

元素后移

```
        *(p+1)=*p ;
```

```
    *q = e ;
```

插入

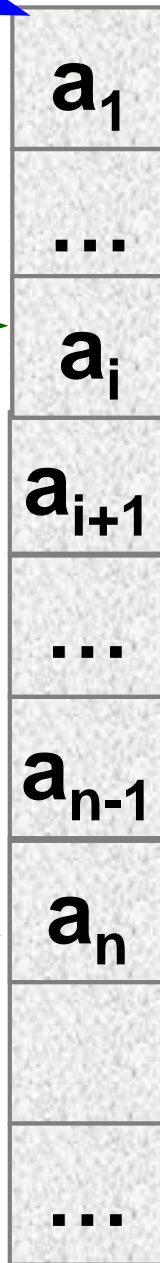
```
    ++L.length;
```

```
    return (OK);
```


线性表长度+1

```
}
```

L.elem



线性表的顺序存储结构

 动态分配相关库函数 (包含在 **stdlib.h** 或 **malloc.h** 中)

◆ 修改内存大小函数

void * realloc(void ***p**, unsigned **size**)

功能：将**p**所指向的存储空间大小改为**size**，**size**可以比原内存空间大或小。

- ① 若原存储空间后有足够空间可供扩展，则在**原存储区位置向高地址方向扩展**；
- ② 若原存储区后没有足够的空间，则**分配另一个足够大的存储空间**，并将原存储内容**复制**到新的存储空间中。

返回值：若申请成功，返回分配内存的**首地址**；
若申请失败，返回NULL

说明：需将返回值进行强制转换

线性表的顺序存储结构

插入

问题：在线性表L的第 i ($1 \leq i \leq n+1$)个元素之前插入一个新的数据元素 e

```
Status ListInsert( SqList &L, int i, ElemType e )
```

```
{  ElemType *p,*q, *newbase;
```

```
  if( (i<1) || (i>L.length+1) )
```

判断 i 是否合法

```
    return (ERROR);
```

```
  .....
```

若存储空间已满，则需增大

```
if (L.length>=L.listsize)
```

```
{  newbase=(ElemType *)realloc(L.elem, (L.listsize+  
    LISTINCREMENT)*sizeof(ElemType));
```

```
  if (!newbase) exit(OVERFLOW);
```

```
  L.elem=newbase;
```

```
  L.listsize+=LISTINCREMENT;
```

```
}
```

L.elem →

a_1

...

a_i

a_{i+1}

...

a_{n-1}

a_n

...

线性表的顺序存储结构

插入

问题：在线性表L的第 i ($1 \leq i \leq n+1$)个元素之前插入一个新的数据元素 e

```
Status ListInsert( SqList &L, int i, ElemType e )
```

```
{ ElemType *p, *q, *newbase;
```

```
if( (i<1) || (i>L.length+1) )
```

判断 i 是否合法

```
return (ERROR);
```

```
.....
```

若存储空间已满，则需增大

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

```
q=&(L.elem[i-1]);
```

```
for( ; p>=q ; p-- )
```

元素后移

```
*(p+1)=*p ;
```

```
*q = e ;
```

插入

$T(n)=O(n)$

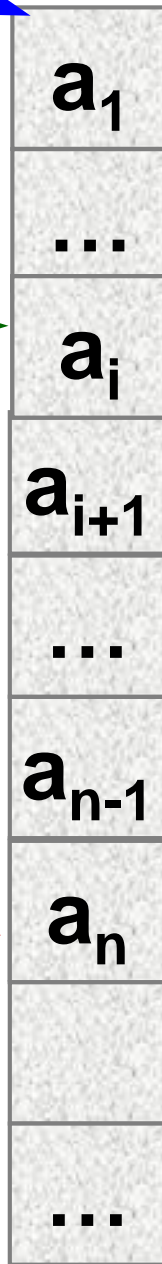
```
++L.length;
```

```
return (OK);
```

线性表长度+1

```
}
```

L.elem



线性表的顺序存储结构

✎ 插入

问题：在线性表 **L** 的第 **i** ($1 \leq i \leq n+1$) 个元素之前插入一个新的数据元素 **e**

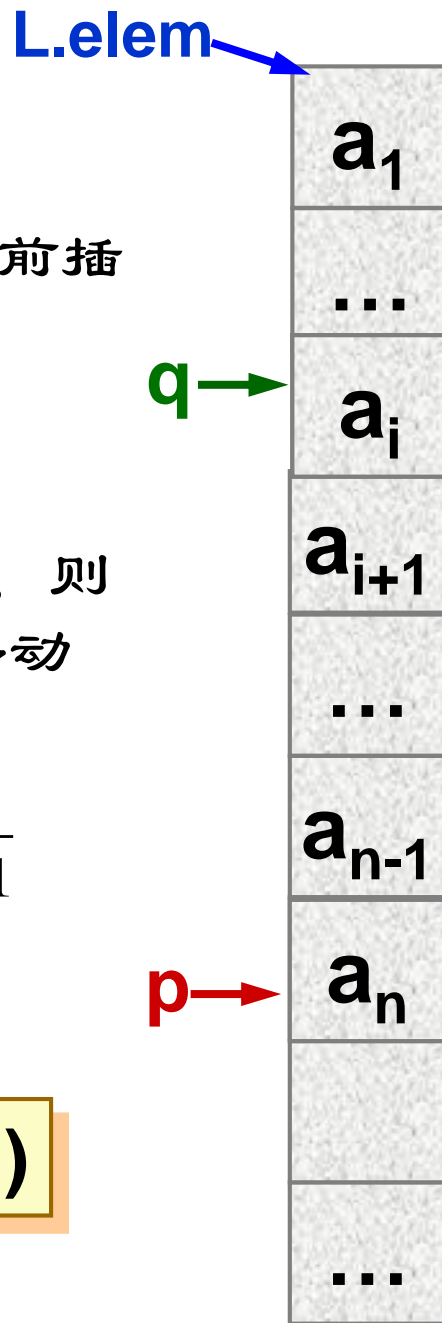
T(n) 的另一种分析方法

☺ 设 **P_i** 是在第 **i** 个元素之前插入新元素的概率，则在长度为 **n** 的线性表中插入元素时，需要移动元素的 **平均次数** 为：

$$E_{is} = \sum_{i=1}^{n+1} P_i (n - i + 1) \quad \text{若认为 } P_i = \frac{1}{n+1}$$

$$\begin{aligned} \text{则 } E_{is} &= \frac{1}{n+1} \sum_{i=1}^{n+1} (n - i + 1) = \frac{1}{n+1} \sum_{i=1}^n i \\ &= \frac{1}{n+1} \frac{n(n+1)}{2} = \frac{n}{2} \end{aligned}$$

$$T(n) = O(n)$$



线性表的顺序存储结构

✂ 删除

问题：将线性表L的第 i ($1 \leq i \leq n$)个元素删除



删除一个元素后，线性表L的逻辑结构发生了什么变化？

使长度为 n 的线性表

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

变成长度为 $n-1$ 的线性表

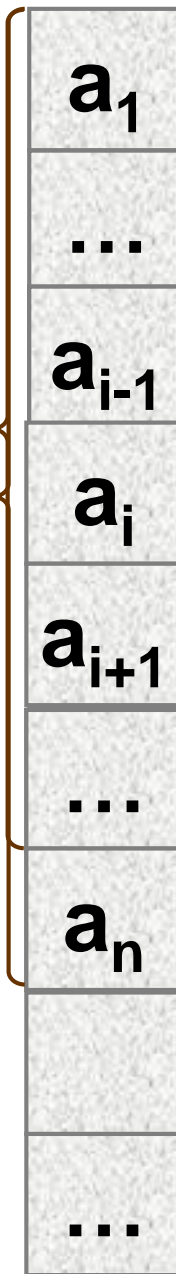
$(a_1, a_2, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$

需将 a_{i+1} 至 a_n 依次前移

移动 $n-i$ 次

L.Listsize

L.Listsize



线性表的顺序存储结构

删除

问题：将线性表L的第 i ($1 \leq i \leq n$)个元素删除

元素前移——

先移动 a_{i+1}, \dots , 最后移动 a_n

$p = \&(L.elem[i-1]);$ 记录删除位置

$e = *p;$ 被删除元素 $\rightarrow e$

$q = \&(L.elem[L.length-1]);$

或 $q = L.elem + L.length - 1;$

for($p++;$ $p \leq q;$ $p++$)

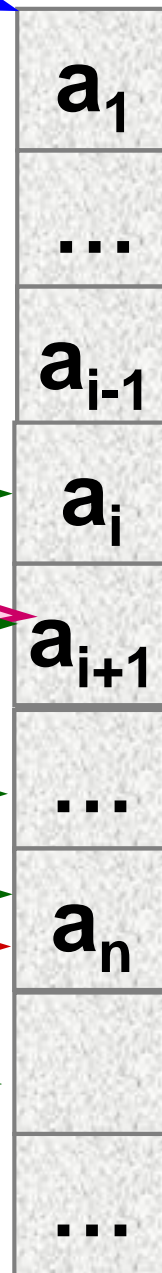
$*(p-1) = *p;$

记录
表尾位置

元素前移

从此处开始前移

L.elem



线性表的顺序存储结构

L.elem

✂ 删除

问题：将线性表L的第 i ($1 \leq i \leq n$)个元素删除

```
Status ListDelete( SqList &L, int i, ElemType &e )
```

```
{ ElemType *p,*q;
```

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

判断i是否合法

```
return (ERROR);
```

```
p=&(L.elem[i-1]);
```

记录删除位置

```
e=*p;
```

被删除元素→e

```
q=L.elem+L.length-1;
```

记录表尾位置

```
for(++p; p<=q; ++p)
```

```
*(p-1)=*p;
```

元素前移

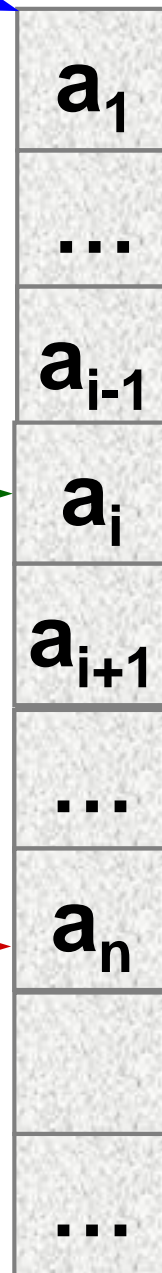
```
--L.length;
```

线性表长度-1

```
return (OK);
```

$T(n)=O(n)$

```
}
```



线性表的顺序存储结构

L.elem

删除

问题：将线性表L的第*i* ($1 \leq i \leq n$)个元素删除

$T(n)$ 的另一种分析方法

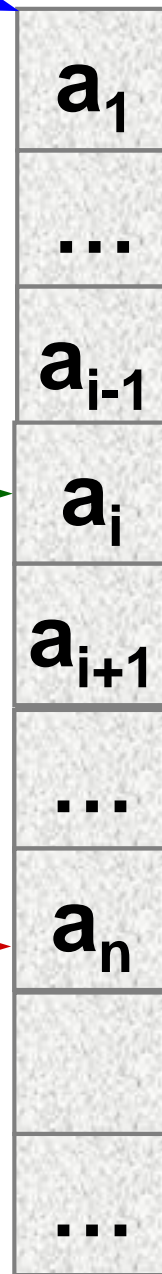
☺ 设 Q_i 是删除第*i*个元素的概率，则在长度为*n*的线性表中删除一个元素时，需要移动元素的平均次数为：

$$E_{de} = \sum_{i=1}^n Q_i (n-i) \quad \text{若认为 } Q_i = \frac{1}{n}$$

$$\begin{aligned} \text{则 } E_{de} &= \frac{1}{n} \sum_{i=1}^n (n-i) = \frac{1}{n} \sum_{i=1}^{n-1} i \\ &= \frac{1}{n} \frac{(n-1)n}{2} = \frac{n-1}{2} \end{aligned}$$

$$T(n) = O(n)$$

☹ 因此，在顺序表中插入或删除一个元素时，平均移动表的一半元素，当*n*很大时，效率很低。



线性表的顺序存储结构

🏠 顺序存储结构的优缺点

😊 优点

- 逻辑相邻的数据元素存储位置也相邻
- 任一数据元素均可随机存取
- 存储空间使用紧凑

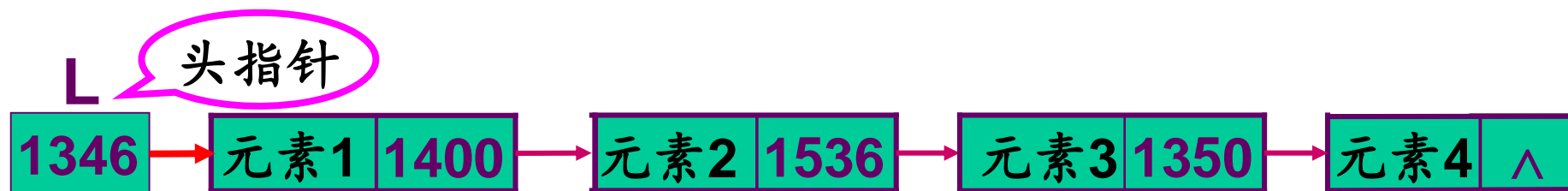
😞 缺点

- 插入、删除操作需要移动大量的元素

顺序存储结构
适用于何种问题？

- ✓ 适用于要求存储空间紧凑、经常对数据进行随机存取的问题；
- ✓ 不适用于经常对数据进行插入/删除操作的问题。

线性表的链式存储结构

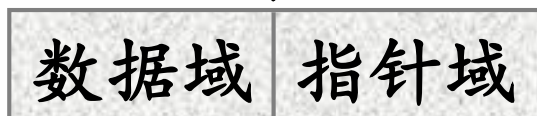


🕸 特点：

🕸 用一组任意的存储单元存储线性表的数据元素

🕸 利用指针反映数据元素的逻辑相邻关系

结点



- 数据域：元素本身信息
- 指针域：存储结点直接后继的地址

存储地址	元素值	指针
1346	元素1	1400
1350	元素4	^
.....		
1400	元素2	1536
.....		
1536	元素3	1350

线性表的链式存储结构

🕸 单链表：

🕷 结点中只含一个指针域的链表称为~

单链表结点结构

data

link

```
typedef struct node
{   ElemType data;           //数据域
    struct node *link;       //指针域
} Lnode, *LinkList;
```

LinkList p;

p指向结点的**数据域**为: **p->data**



指针域为: **p->link**

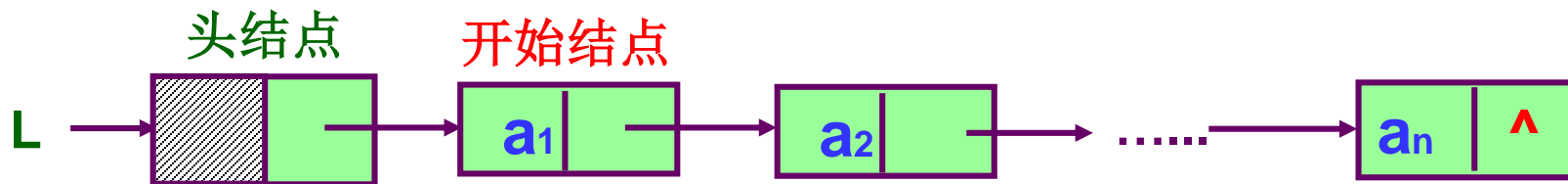
生成新结点: **p=(LinkList)malloc(sizeof(Lnode));**

释放结点p: **free(p);**

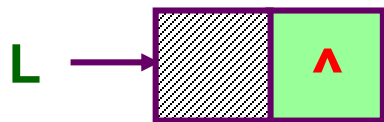
线性表的链式存储结构

🕸 单链表：

🕸 头结点：在单链表第一个结点前附设的结点称为~



🕸 空的单链表中，头结点的指针域为空。



🕸 设置头结点的原因：

- ① 设置头结点后，头指针指向头结点，不论链表否为空，**头指针总是非空**。
- ② 头结点的设置使得对链表**开始结点**的操作与对表中**其它结点**的操作一致（都在某一结点之后）。

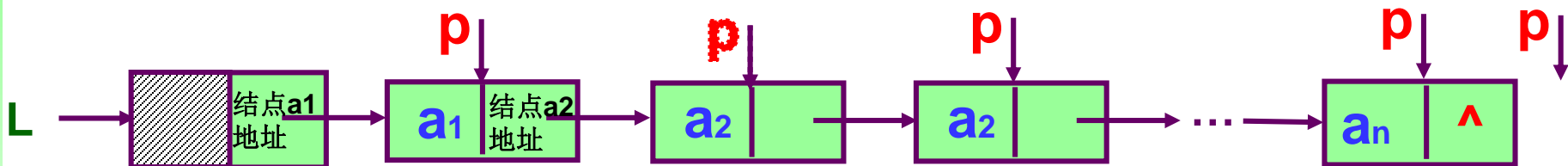
线性表的链式存储结构



单链表：

✍️ 基本语句

```
typedef struct node
{ ElemType data;      //数据域
  struct node *link;  //指针域
} Lnode, *LinkList;
```



一般，从开始结点操作

```
LinkList p;
p=L->link;
```



指针 p 后移如何实现？

```
p=p->link;
```



链表未结束的循环条件为何？

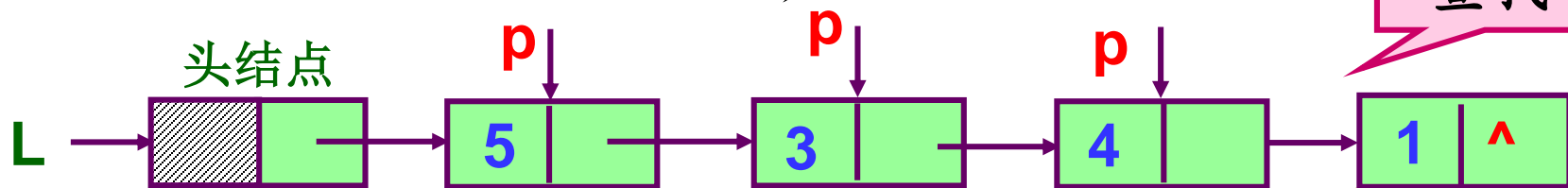
```
while(p!=NULL) 或 while(p)
{ ... }         { ... }
```

单链表的基本操作

查找

```
# define NULL 0
```

问题：查找单链表中是否存在数据域为 e 的结点，若有则返回该结点的指针；否则返回**NULL**



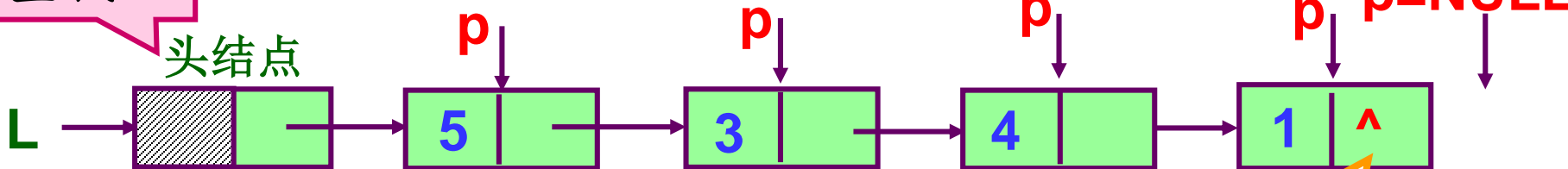
从开始结点查找 $p=L->link;$

```
while(  $p->data \neq e \ \&\& \ p \neq \text{NULL}$  )
```

```
     $p=p->link;$ 
```

★ 判断链表未结束的条件

查找7



查找失败时，
 $p=\text{NULL}$

查找失败

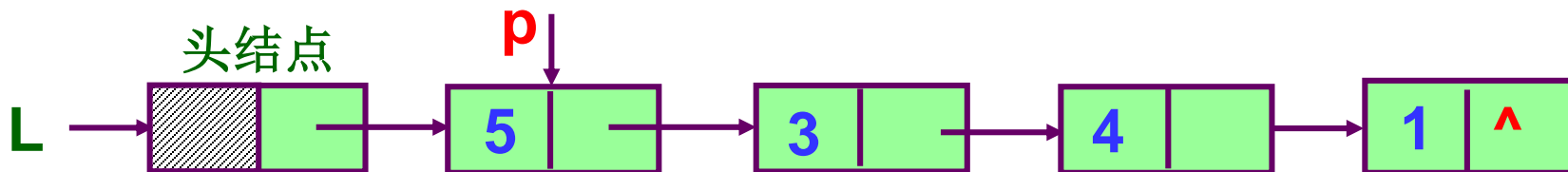
查找成功

查找成功时，
 p 指向结点 e

单链表的基本操作

查找

问题：查找单链表中是否存在数据域为 e 的结点，若有则返回该结点的指针；否则返回**NULL**



```
LinkList dlbcz( LinkList L, ElemType e )
{
    LinkList p=L->link;
    while( p->data!=e && p!=NULL )
        p=p->link;
    return (p);
}
```

$T(n)=O(n)$

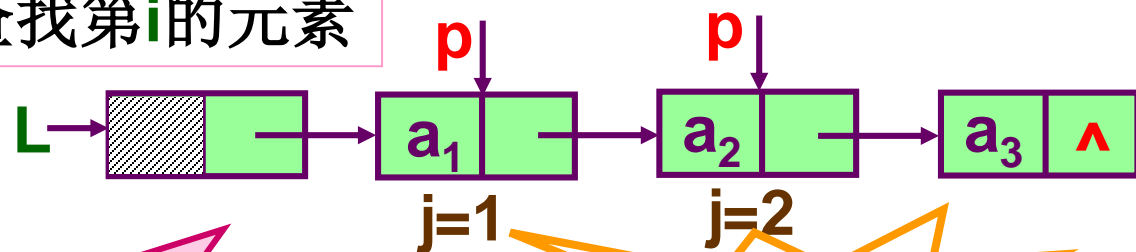
单链表的基本操作

取元素值

问题：取单链表中第*i*个元素的值，若存在该元素，用*e*返回其值；否则返回**ERROR**

初始—— $p = L \rightarrow \text{link};$
 $j = 1;$

查找第*i*的元素



查找第2个元素

查找成功

查找成功时，
 p 指向第*i*个结点

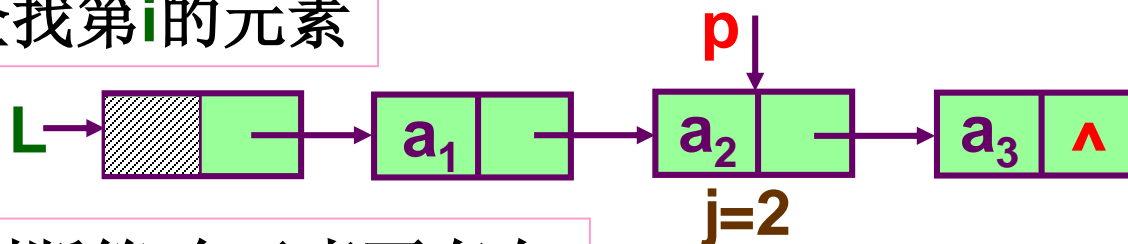
```
while(  $p \ \&\& \ j < i$  )  
{  $p = p \rightarrow \text{link};$   
   $j++;$   
}
```

单链表的基本操作

取元素值

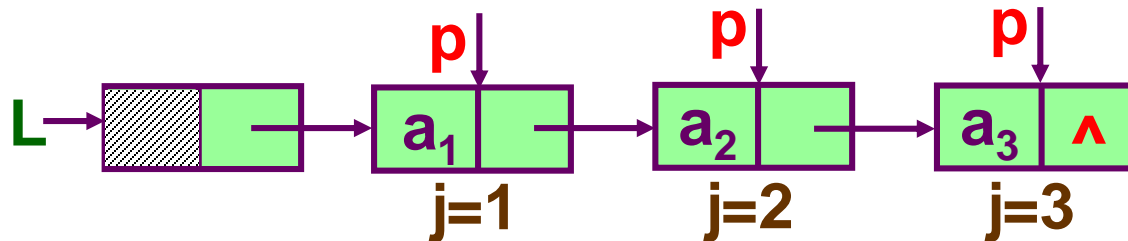
问题：取单链表中第*i*个元素的值，若存在该元素，用*e*返回其值；否则返回**ERROR**

查找第*i*的元素

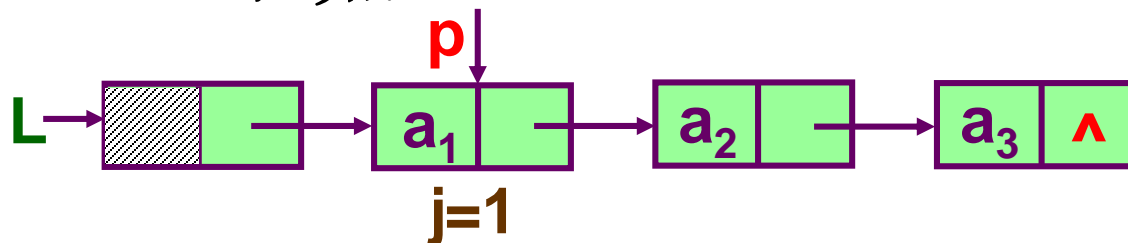


判断第*i*个元素不存在

① *i* > 表长，如： *i* = 4



② *i* < 1，如： *i* = 0



初始—— $p = L \rightarrow \text{link};$
 $j = 1;$

```
while(  $p \ \&\& \ j < i$  )  
{  
     $p = p \rightarrow \text{link};$   
     $j++;$   
}
```

$p = \text{NULL}$

$j > i$

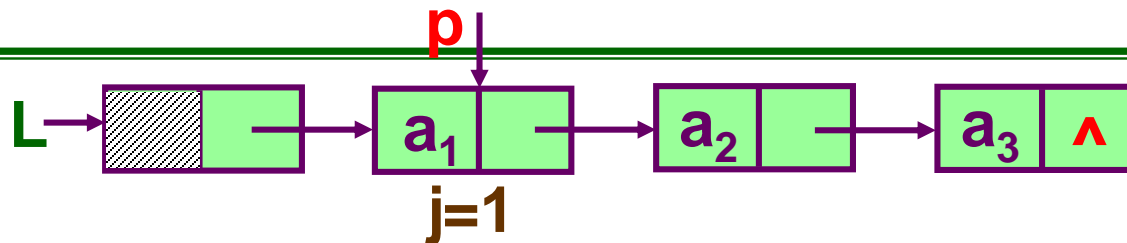
单链表的基本操作

取元素值

问题：取单链表中第*i*个元素的值，若存在该元素，用*e*返回其值；否则返回**ERROR**

```
Status GetElem_L( LinkList L, int i, ElemType &e )
{
    LinkList p=L->link;  int j=1;
    while(p && j<i)
    {  p=p->link;  j++;  }
    if(p==NULL || j>i) return ERROR;
    e=p->data;
    return OK;
}
```

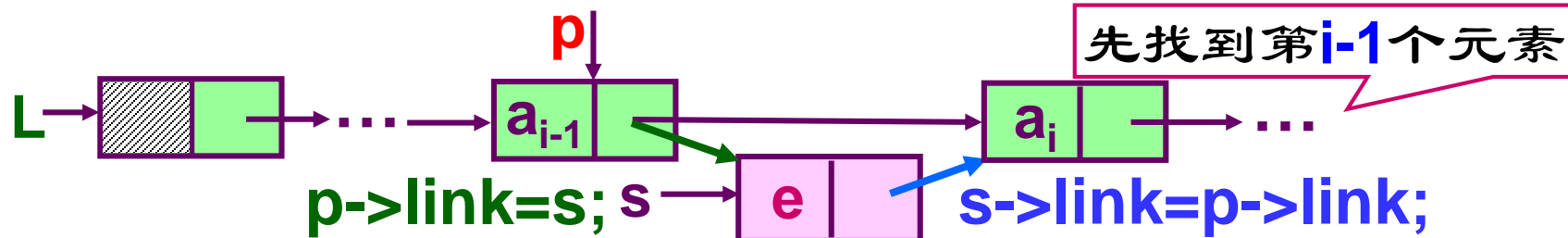
$T(n)=O(n)$



单链表的基本操作

插入

问题：在单链表L的第i个元素之前插入新元素e



Status ListInsert_L(LinkList L, int i, ElemType e)

{ LinkList p = L, s; int j = 0; 查找第i-1个元素

while (p && j < i-1) { p = p->link; ++j; }

if (p==NULL || j > i-1) 判断插入位置是否合法

return **ERROR**;

s = (LinkList) malloc (sizeof (Lnode)); **T(n)=O(n)**

s->data = e;

s->link = p->link; p->link = s; 生成新结点并插入

return **OK**;

}



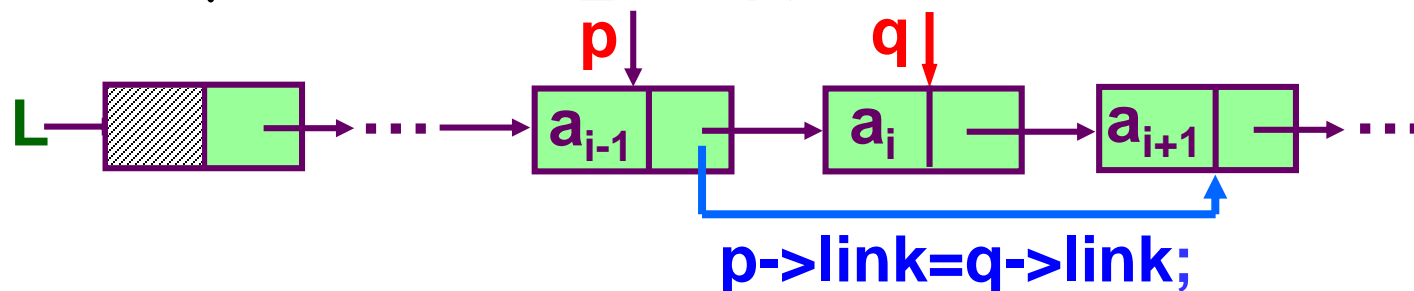
修改链的顺序能否改变?



单链表的基本操作

删除

问题：删除单链表L的第i个结点



```
Status ListDelete_L(LinkList L, int i, ElemType &e)
{
    LinkList p = L, q;    int j = 0;    查找第i-1个元素
    while (p->link && j < i-1) { p = p->link; ++j; }
    if (p->link == NULL || j > i-1)    判断删除位置是否合法
        return ERROR;
    q = p->link;    p->link = q->link;    删除结点并释放
    e = q->data;    free(q);
    return OK;
}
```

$T(n) = O(n)$

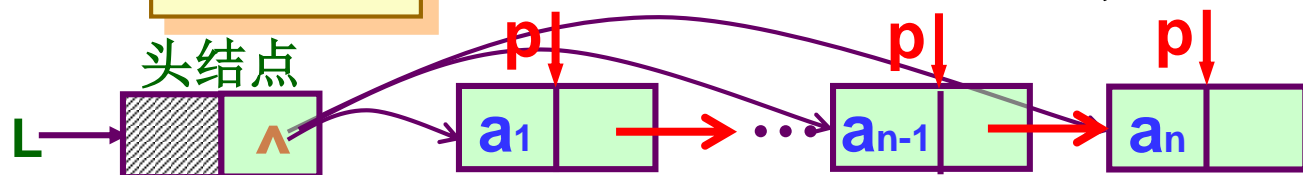
单链表的基本操作

❌ 动态建立单链表

问题：输入 n 个数据，建立单链表 L

头插法

在红结点后插入新结点，逆序输入 n 个数据



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

```
void Create_L1(LinkList &L, int n)
```

```
{  LinkList p;  int i;
```

```
  L=(LinkList)malloc(sizeof(LNode));
```

```
  L->data=0;  L->link=NULL;
```

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

```
  {  p=(LinkList)malloc(sizeof(LNode));
```

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

```
    p->link=L->link;  L->link=p;
```

```
  }
```

```
}
```

建立头结点

建立新结点

将 p 插入为 L 的后继

$T(n)=O(n)$

单链表的基本操作

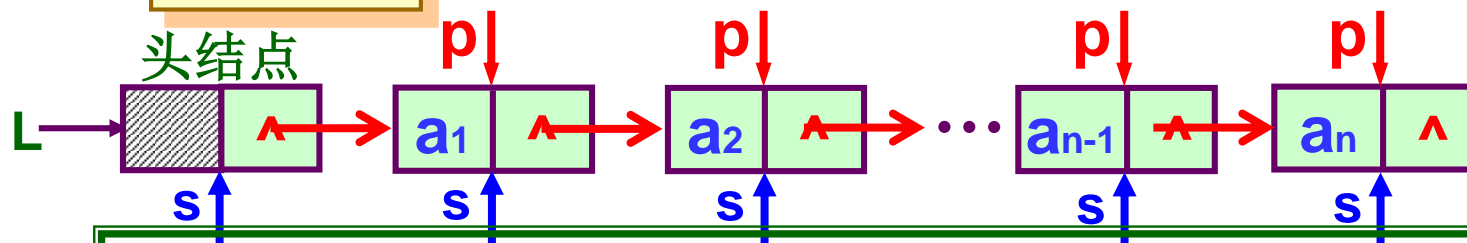
❌ 动态建立单链表

问题：输入 n 个数据，建立单链表 L

s 指向表尾结点，初始 $s=L$

尾插法

在单链表尾插入新结点，顺序输入 n 个数据



$s \rightarrow \text{link} = p;$
 $s = p;$

```
void Create_L2(LinkList &L, int n)
```

```
{  LinkList p, s=L;  int i;  
  L=(LinkList)malloc(sizeof(LNode));  
  L->data=0;  L->link=NULL;
```

建立头结点

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

```
  {  p=(LinkList)malloc(sizeof(LNode));
```

建立新结点

```
    scanf("%d",&p->data);  p->link=NULL;
```

```
    s->link=p;  s=p;
```

将 p 插入为 s 的后继

$T(n)=O(n)$

```
}
```


单链表存储结构

单链表存储结构的优缺点

😊 优点：

- 单链表是一种**动态结构**，不需预先分配存储空间
- 插入/删除**不需要移动大量数据元素**

😞 缺点：

- 指针占用额外存储空间
- 单链表的表长是一个隐含的值
- 不能随机存取

链表未结束条件

`while(p!=NULL)`

.....

单链表存储结构
适用于何种问题？

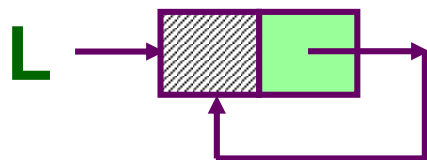
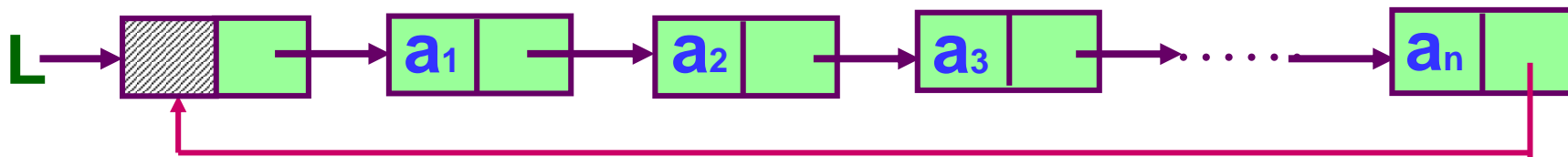
- ✓ 适用于经常对数据进行**插入/删除**操作的问题；
- ✓ 而不适用于经常对数据进行**随机存取**的问题。



线性表的链式存储结构

🕸 循环链表

🕷 表中最后一个结点的指针域指回头结点的链表。



空循环链表

单向性

🕷 特点：从表中任一结点出发均可找到表中其它结点，
使某些运算在链表上更易实现。

🕷 与单链表的区别：算法的循环条件不同

✓ 单链表

while(p!=NULL)

.....

✓ 循环链表

while(p!=L)

.....

线性表的链式存储结构

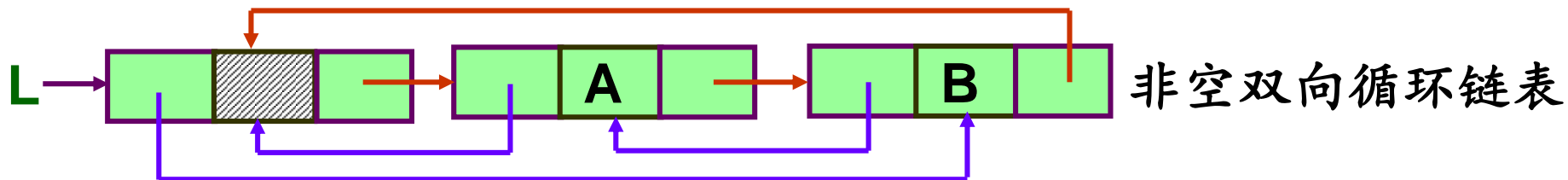
🕸️ 双向链表

🕷️ 表中结点有前驱指针域和后继指针域的链表。

结点结构

prior	element	next
-------	---------	------

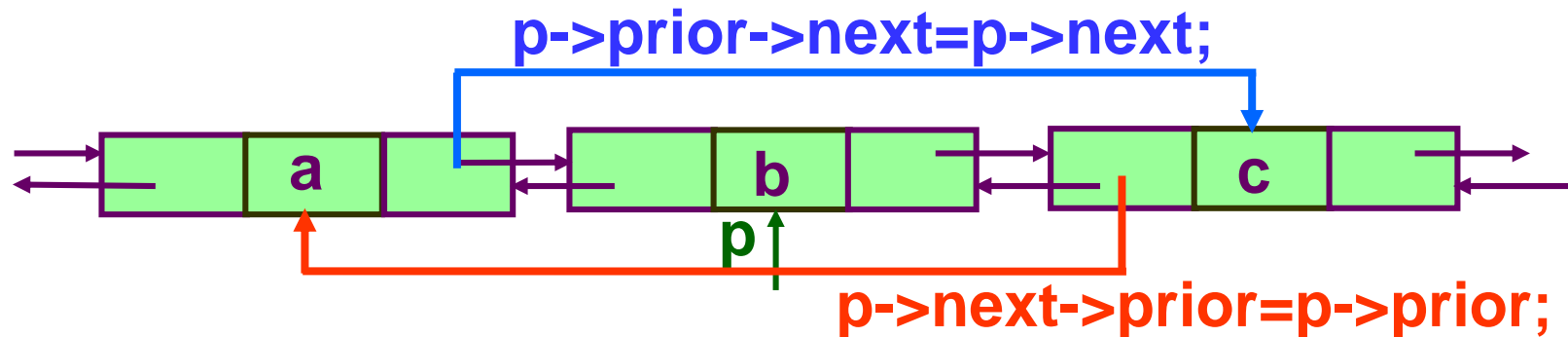
```
typedef struct Dunode
{
    ElemType element;           //数据域
    struct Dunode *prior,*next; //指针域
} DuLnode, *DuLinkList;
```



双向链表的基本操作

删除

问题：删除双向链表中指针 p 指向的结点。



```
void del_dulist(DuLinkList p)
{
    p->prior->next=p->next;
    p->next->prior=p->prior;
    free(p);
}
```

$T(n)=O(1)$

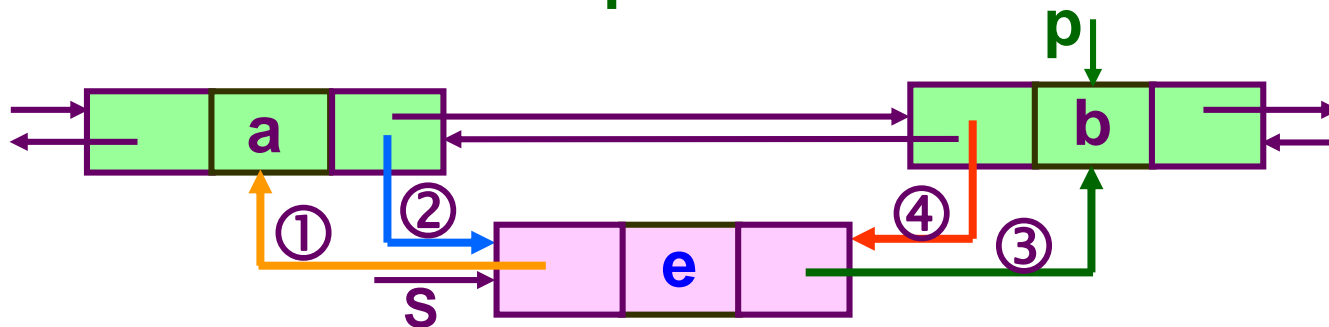


删除第 i 个结点呢？

双向链表的基本操作

插入

问题：在双向链表中指针 **p** 指向的结点前插入新结点。



```
void ins_dulist(DuListLink p, ElemType e)
{
    DuListLink s;
    s=(DuListLink)malloc(sizeof(DuLnode));
    s->element=e;
    ① s->prior=p->prior;
    ② p->prior->next=s;
    ③ s->next=p;
    ④ p->prior=s;
}
```

$T(n)=O(1)$

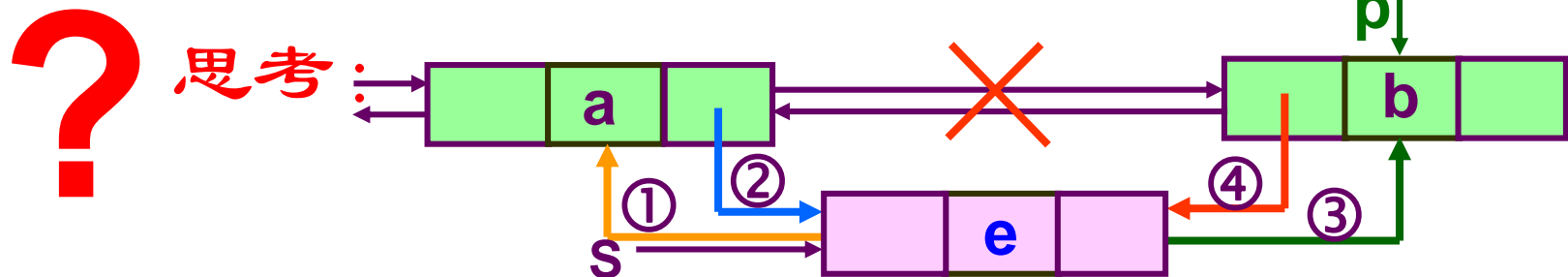


在第i个结点前插入呢？

双向链表的基本操作

✂ 插入

问题：在双向链表中指针 **p** 指向的结点前插入新结点。



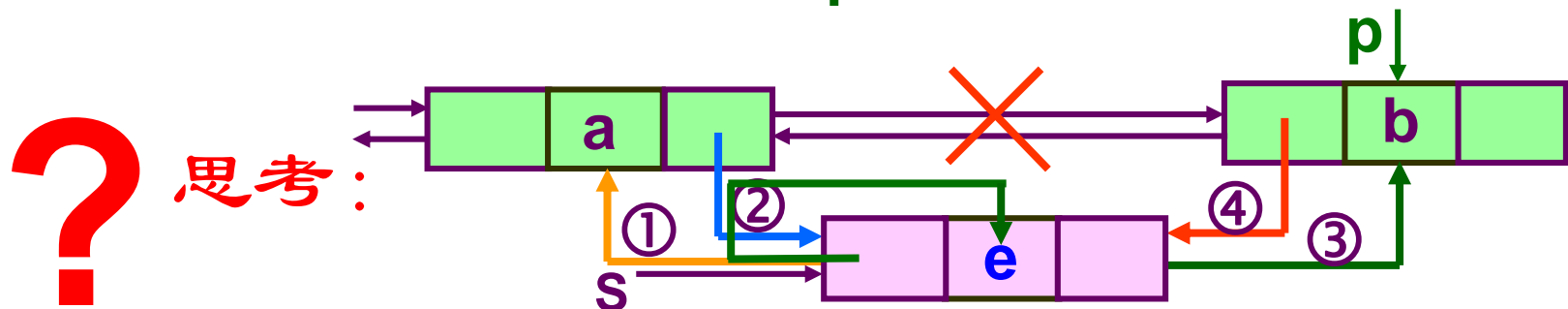
③ **s->next=p;**
② **p->prior->next=s;**
① **s->prior=p->prior;**
④ **p->prior=s;**

④ **p->prior=s;**
① **s->prior=p->prior;**
② **p->prior->next=s;**
③ **s->next=p;**

双向链表的基本操作

✂️ 插入

问题：在双向链表中指针 **p** 指向的结点前插入新结点。



- ③ $s \rightarrow next = p;$
- ② $p \rightarrow prior \rightarrow next = s;$
- ① $s \rightarrow prior = p \rightarrow prior;$
- ④ $p \rightarrow prior = s;$

- ④ $p \rightarrow prior = s;$
- ① $s \rightarrow prior = p \rightarrow prior;$
- ② $p \rightarrow prior \rightarrow next = s;$
- ③ $s \rightarrow next = p;$

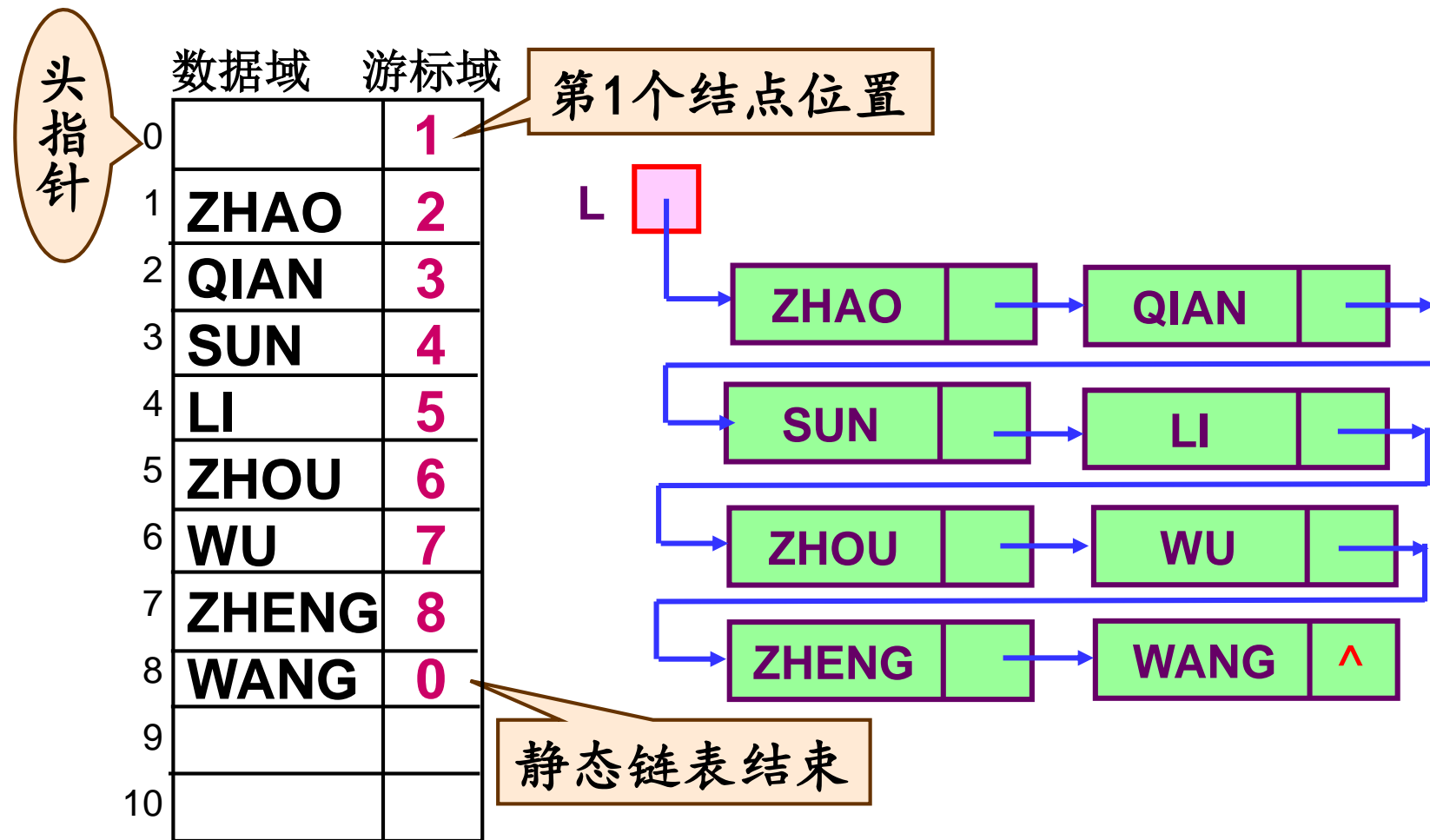


对链表进行操作时，必须**后**断开原来结点之间的关系

线性表的链式存储结构

静态链表

借助一维数组描述链式结构，使用游标模拟指针。



线性表的链式存储结构

静态链表

借助一维数组描述链式结构，使用游标模拟指针。



静态链表的插入和删除操作不需要移动元素，仅需要修改“游标”，因而仍具有链式存储结构的优点。

静态链表可用于在无指针类型的程序设计语言中使用链式存储结构。

线性表的应用举例

✍️ 问题：一元多项式的表示及相加

🏠 一元多项式的表示：

表示方法1： 所有项的升幂表示

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

可存储每一项的系数： $(p_0, p_1, p_2, \dots, p_n)$

但对类似 $S(x) = 1 + 3x^{1000} + 2x^{20000}$ 的多项式浪费空间

✓ 表示方法2： 非零系数项的升幂表示

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

其中： $p_i \neq 0 (i = 1, \dots, m)$ $0 \leq e_1 < e_2 < \dots < e_m = n$

则需存储每个非零系数项的系数和指数： (p_i, e_i)

线性表的应用举例

✍️ 问题：一元多项式的表示及相加

🏠 一元多项式的相加：

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

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



适合采用哪种
存储方式？

适合链式存储

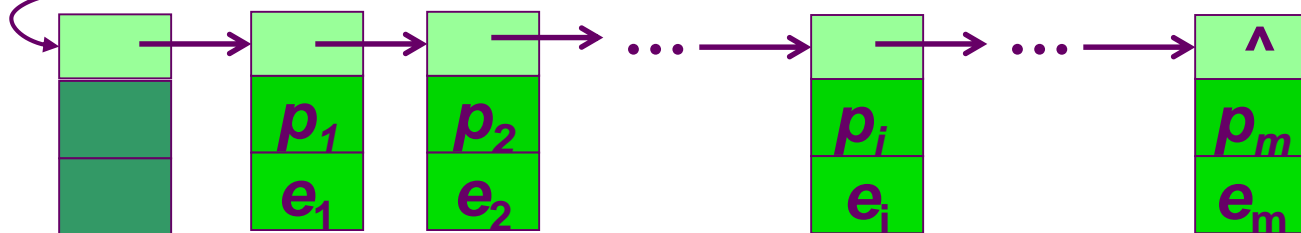
为方便实现并节省存储空间，常在多项式A(x)上进行修改、插入、删除操作得到和多项式。

结点结构



```
typedef struct node
{
    int coef, exp;
    struct node *next;
} Lnode, *LinkList;
```

Head



究竟用什么样的数据结构好呢？
数组？链表？



线性表的应用举例

✍️ 问题：一元多项式的表示及相加

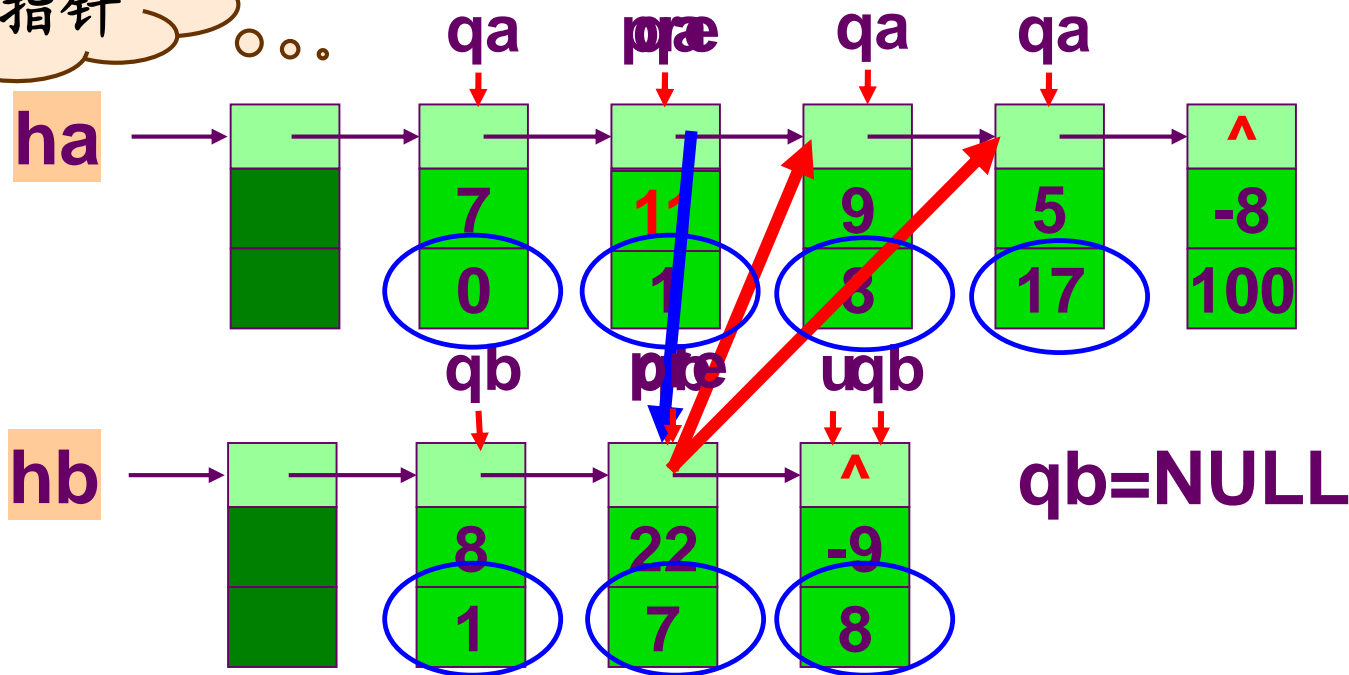
🏠 一元多项式的相加：

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

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

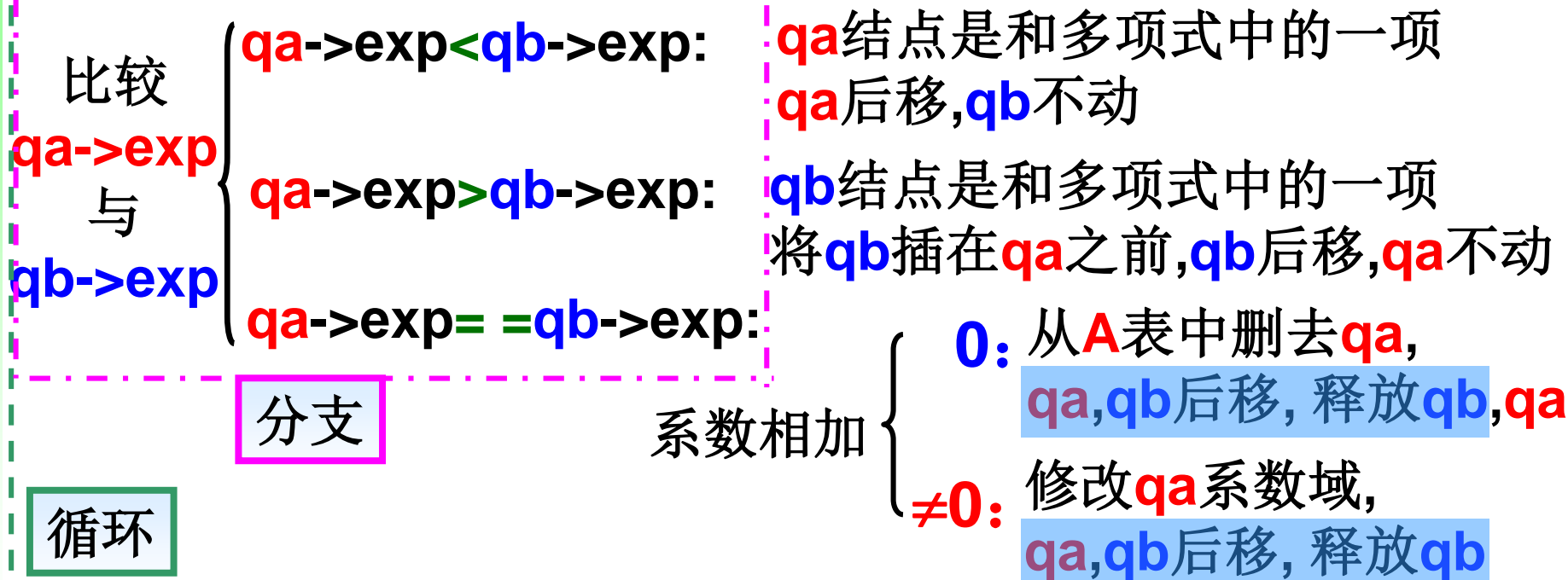
为方便实现并节省存储空间，常在多项式A(x)上进行修改、插入、删除操作得到和多项式。

跟踪指针



运算规则

设 qa, qb 分别指向A,B中某一结点, 其初值是开始结点



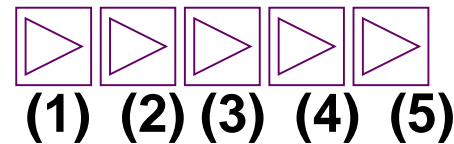
直到 qa 或 qb 为NULL {
若 $qb == NULL$, 结束
若 $qa == NULL$, 将B中剩余部分连到A上

分支



分析

线性表的应用举例



 问题：一元多项式的表示及相加

```
void AddPoly(LinkList ha, LinkList hb)
```

```
{ LinkList qa, qb, u, pre; int x;
```

```
qa=ha->next; qb=hb->next; pre=
```

```
while((qa!=NULL) &&(qb!=NULL))
```

```
{ ① if(qa->exp<qb->exp)
```

```
{ pre=qa; qa=qa->next;}
```

```
②④else if(qa->exp==qb->exp)
```

```
{ ..... }
```

```
③else
```

```
{ u=qb->next;qb->next=qa;pre=
```

```
pre=qb; qb=u; }
```

```
}
```

```
⑤if(qa==NULL) pre->next=qb;
```

```
free(hb);
```

```
}
```

```
x=qa->coef+qb->coef;
```

```
if(x!=0)
```

```
{ qa->coef=x;
```

```
pre=qa;
```

```
}
```

```
else
```

```
{ pre->next=qa->next;
```

```
free(qa);
```

```
}
```

```
qa=pre->next;
```

```
u=qb;
```

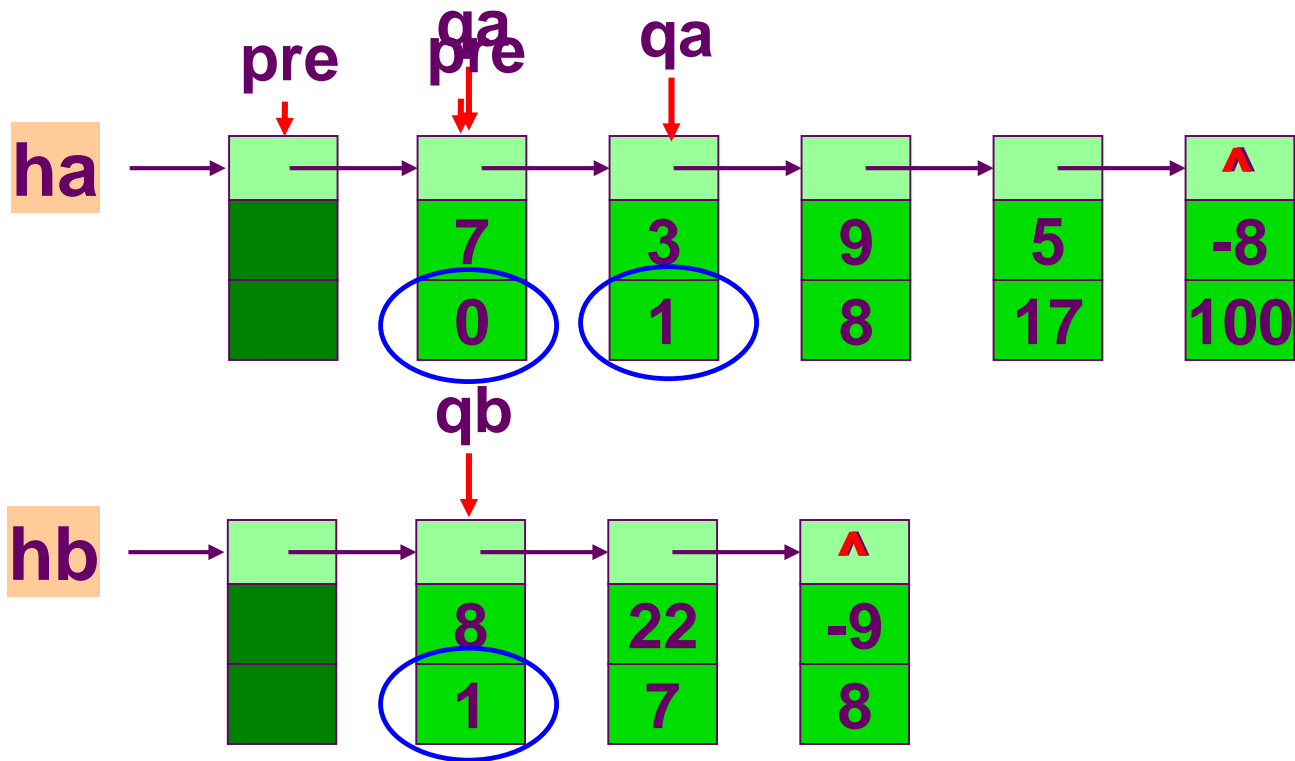
```
qb=qb->next;
```

```
free(u);
```

Back

线性表的应用举例

问题：一元多项式的表示及相加



qa->exp < **qb**->exp:

qa 结点是和多项式中的一项

qa 后移, **qb** 不动

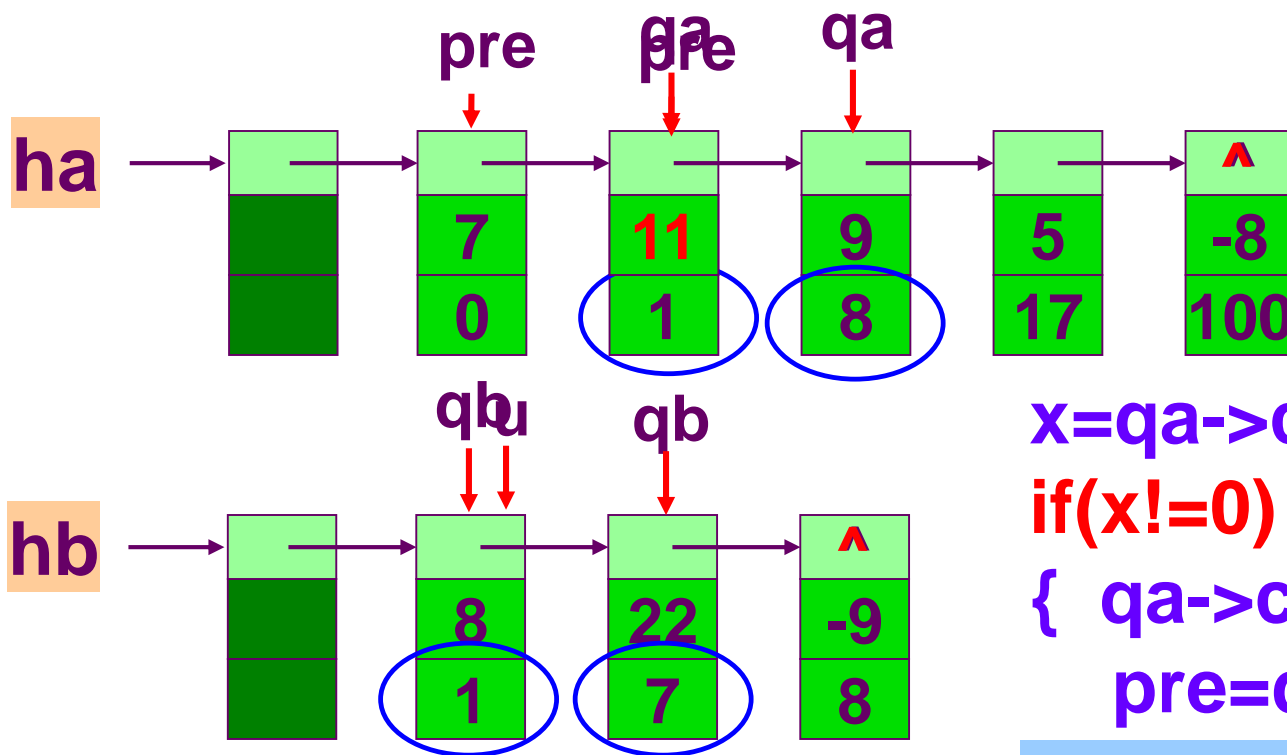
pre=qa;

qa=qa->next;



线性表的应用举例

问题：一元多项式的表示及相加



```
x=qa->coef+qb->coef;  
if(x!=0)  
{ qa->coef=x;  
  pre=qa;  
}
```

qa->exp = qb->exp:

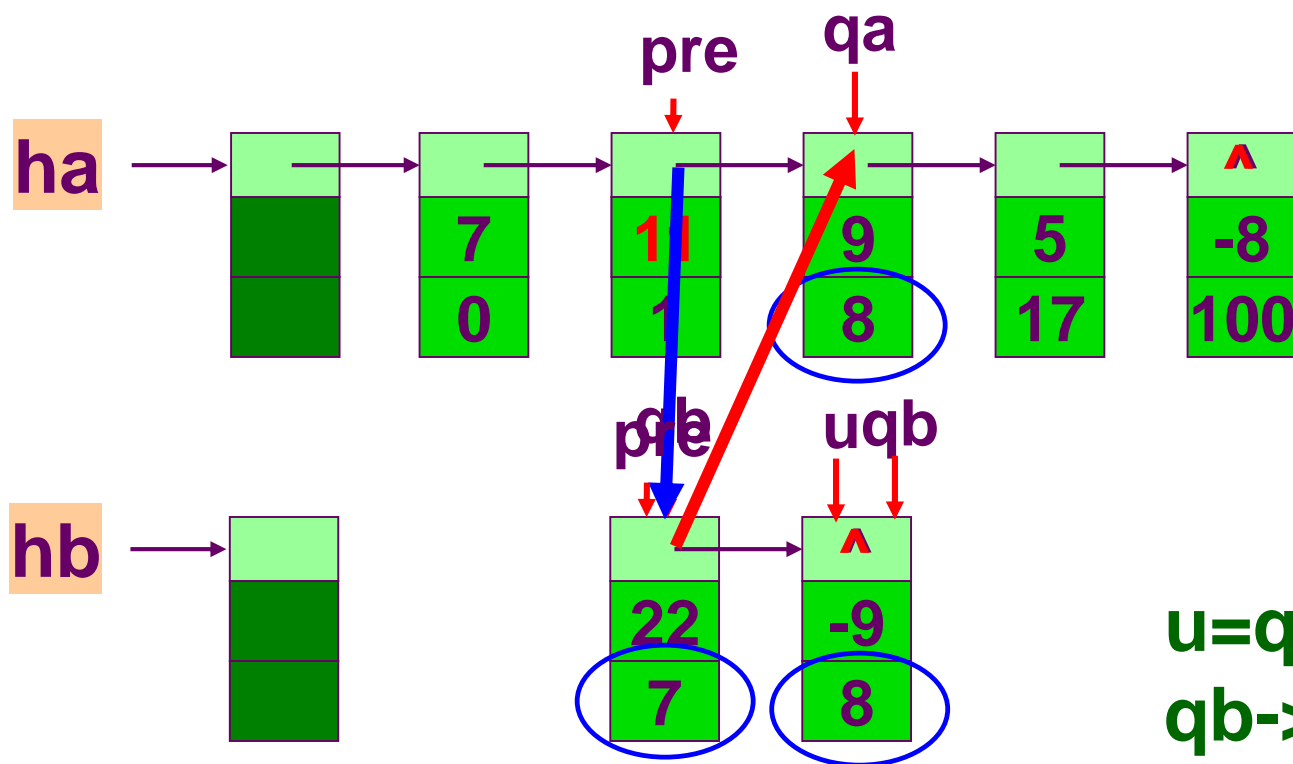
系数相加 $\neq 0$: 修改**qa**系数域,
qa, qb后移, 释放**qb**

```
qa=pre->next;  
u=qb;  
qb=qb->next;  
free(u);
```



线性表的应用举例

问题：一元多项式的表示及相加



qa->exp > **qb**->exp:

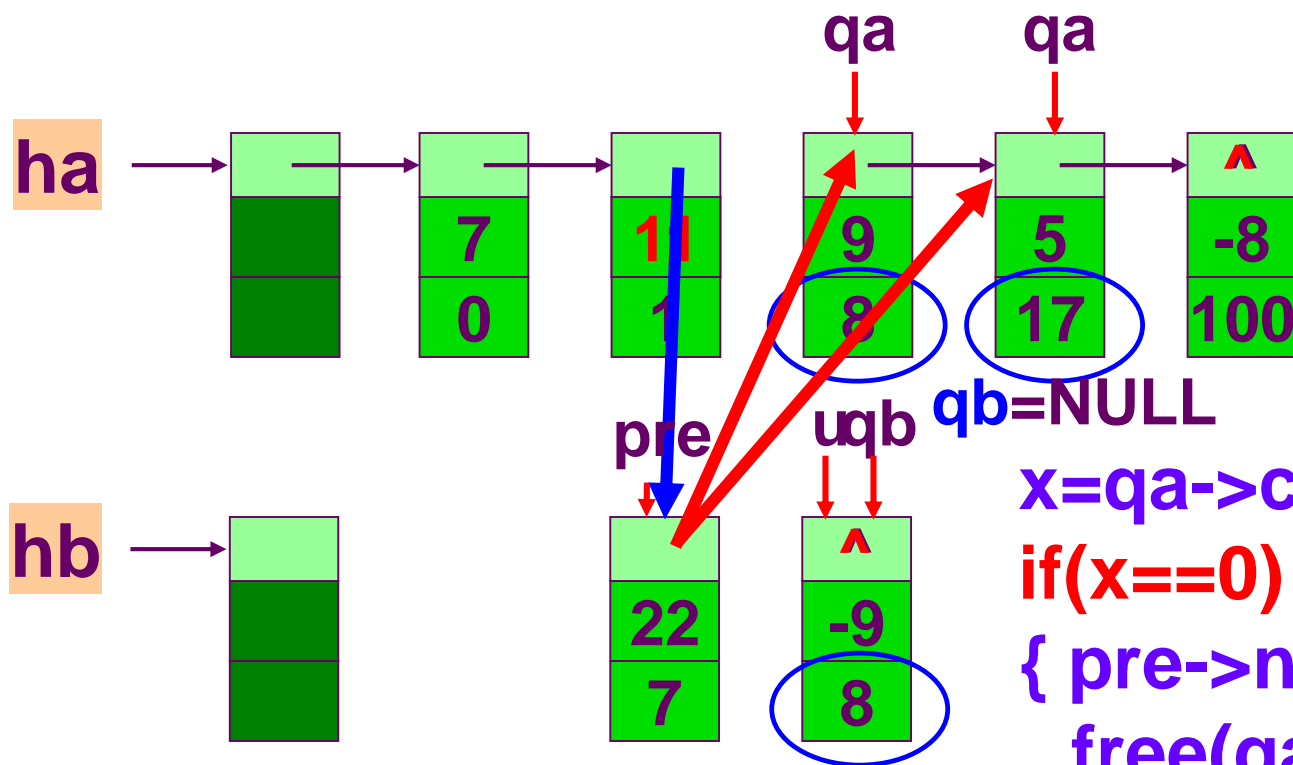
qb 结点是和多项式中的一项
将 **qb** 插在 **qa** 之前, **qb** 后移, **qa** 不动
需要记录 **qb** 的直接后继

```
u=qb->next;  
qb->next=qa;  
pre->next=qb;  
pre=qb;  
qb=u;
```



线性表的应用举例

问题：一元多项式的表示及相加



$qa \rightarrow \text{exp} = qb \rightarrow \text{exp}$:

系数相加 = 0: 从A表中删去 **qa**,
qa, qb 后移, 释放 **qb**, **qa**

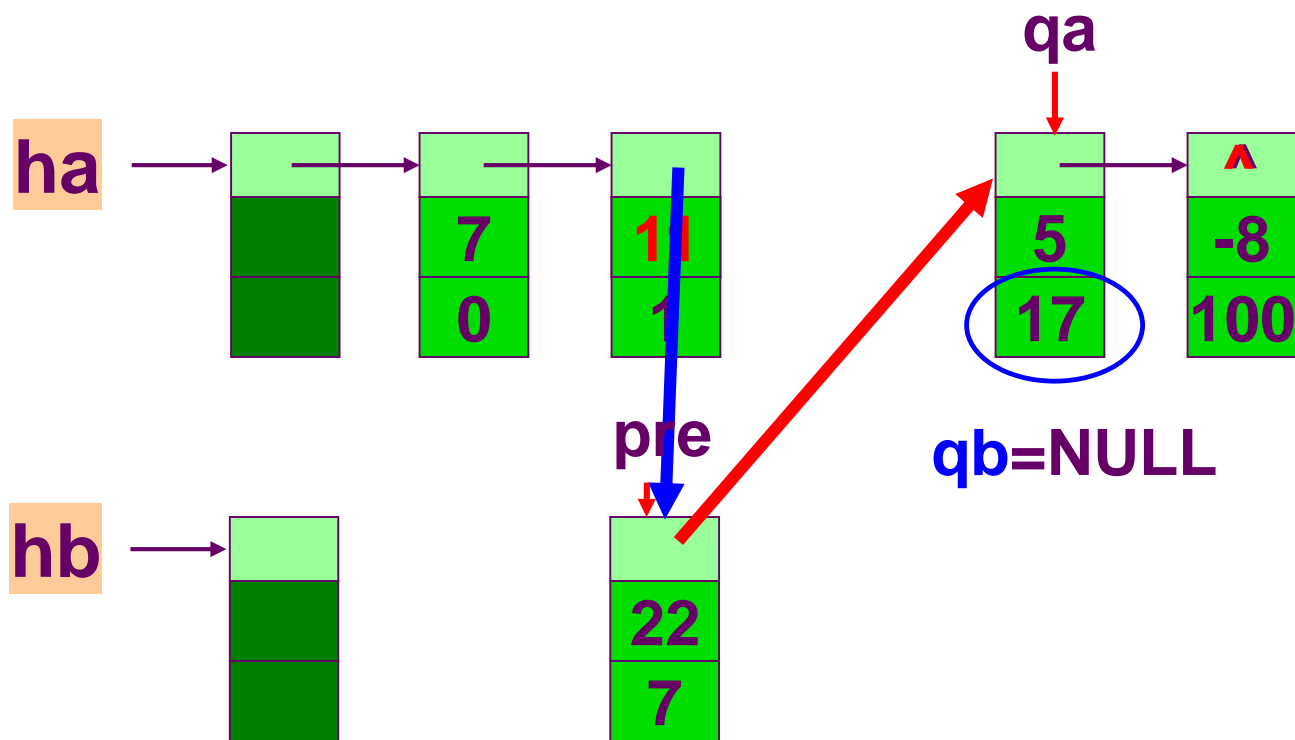
```
x=qa->coef+qb->coef;  
if(x==0)  
{ pre->next=qa->next;  
  free(qa); }
```

```
qa=pre->next;  
u=qb;  
qb=qb->next;  
free(u);
```



线性表的应用举例

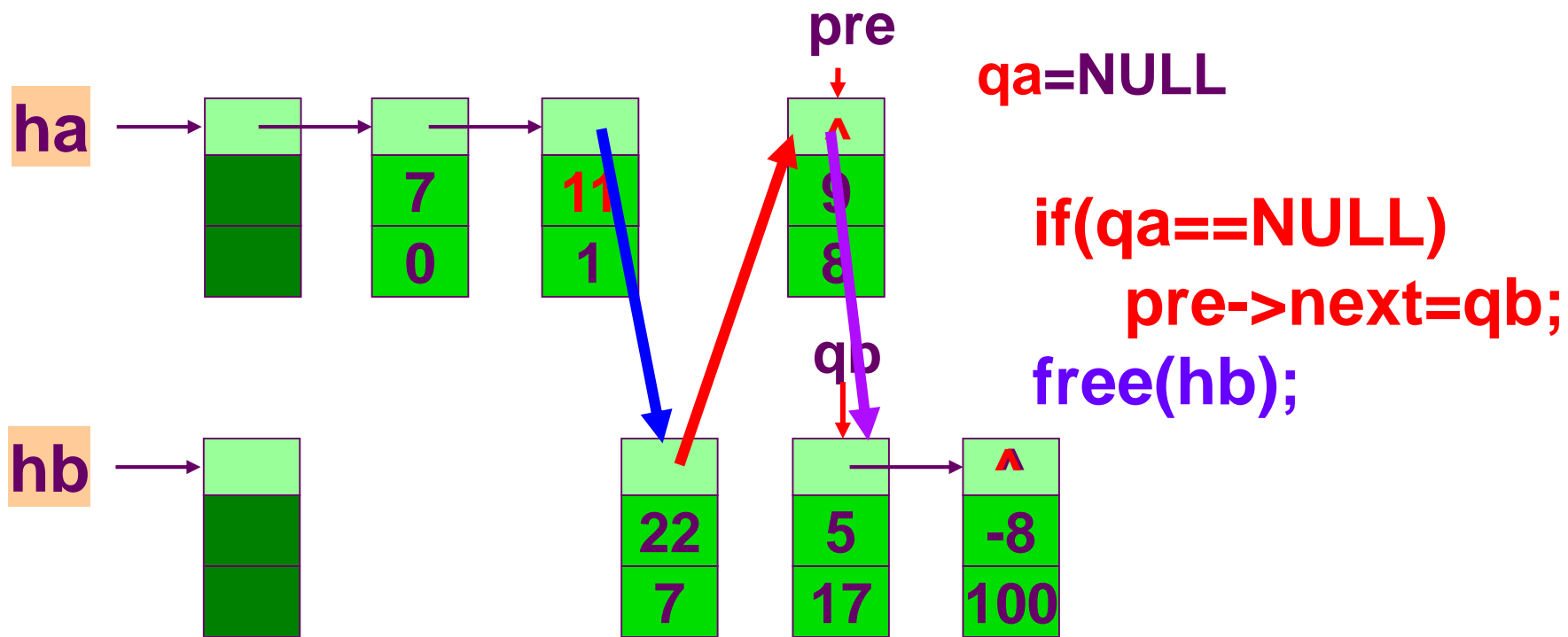
问题：一元多项式的表示及相加



直到 **qa** 或 **qb** 为 **NULL** { 若 **qb** == **NULL**, 结束

线性表的应用举例

问题：一元多项式的表示及相加



直到 **qa** 或 **qb** 为 **NULL** {
 若 **qb** == **NULL**, 结束
 若 **qa** == **NULL**, 将 **B** 中剩余部分连到 **A** 上



本章小结

- 掌握线性表的定义，顺序和链式存储结构和基本运算的实现
- 掌握循环链表、双向链表的特点，及基本运算的实现
- 了解静态链表
- 掌握一元多项式的表示和相加