

第二章 线性表

第二章 线性表

线性表是最简单常用的数据结构，顺序存储结构链式存储结构也是应用中最常用的存储方法，这一部分内容和方法掌握了，有助于理解和掌握后续章节的内容，如栈队列串是特殊的线性表，数组和广义表是线性表的扩展；有助于理解和掌握树和图等复杂的数据结构存储结构和图等复杂结构的操作算法，因为树和图的存储结构大多或是这两种存储结构的扩充，或是它们的组合，因此这一章的内容非常重要，同学们要很好地学习理解和掌握。



第二章 线性表

- 2.1 线性表概念及基本操作
- 2.2 线性表的顺序存储和实现
- 2.3 线性表的链式存储和实现
 - 2.3.1 线性链表
 - 2.3.2 循环链表
 - 2.3.3 双向链表
- 2.4 一元多项式的表示及相加



第二章 线性表

在本课程介绍的几种数据结构中，线性表是最简单的，也是最常用的数据结构，线性表在实际应用大量使用，并不是一个陌生的概念。

2.1 线性表的概念

一 线性表的逻辑结构

线性表是 n 个类型相同数据元素的有限序列，
通常记作 $(a_1, a_2, a_3, \dots, a_n)$ 。

例1、数学中的数列（11，13，15，17，19，21）

例2、英文字母表（A, B, C, D, E... Z）。

例3、某单位的电话号码簿。

姓名	电话号码
蔡颖	63214444
陈红	63217777
刘建平	63216666
王小林	63218888
张力	63215555

...

2.1 线性表的概念

说明：设 $A = (a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 是一线性表

- 1) 线性表的数据元素可以是各种各样的，但同一线性表中的元素必须是同一类型的；
- 2) 在表中 a_{i-1} 领先于 a_i ， a_i 领先于 a_{i+1} ，称 a_{i-1} 是 a_i 的直接前驱， a_{i+1} 是 a_i 的直接后继；
- 3) 在线性表中，除第一个元素和最后一个元素之外，其他元素都有且仅有一个直接前驱，有且仅有一个直接后继，具有这种结构特征的数据结构称为线性结构。线性表是一种线性数据结构；
- 4) 线性表中元素的个数 n 称为线性表的长度， $n=0$ 时称为空表；
- 5) a_i 是线性表的第 i 个元素，称 i 为数据元素 a_i 的序号，每一个元素在线性表中的位置，仅取决于它的序号；

2.1 线性表的概念

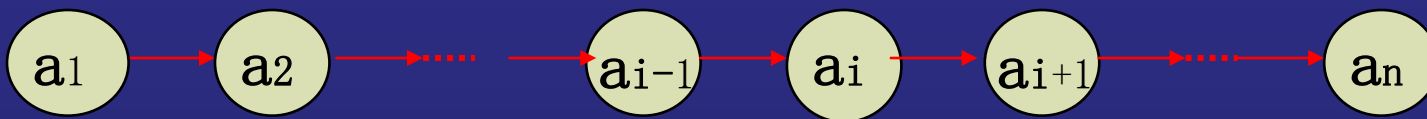
线性表的其他表示方式

二元组表示

$L = \langle D, S \rangle$, 其中 $D = \{ a_1, a_2, a_3, \dots, a_n \}$

$S = \{ R \}$ $R = \{ \langle a_1, a_2 \rangle, \langle a_2, a_3 \rangle, \langle a_3, a_4 \rangle \dots \langle a_{n-1}, a_n \rangle \}$

图示表示



顶点：表示数据

边：表示是数据间的顺序结构关系

2.1 线性表的概念

二 线性表的基本操作

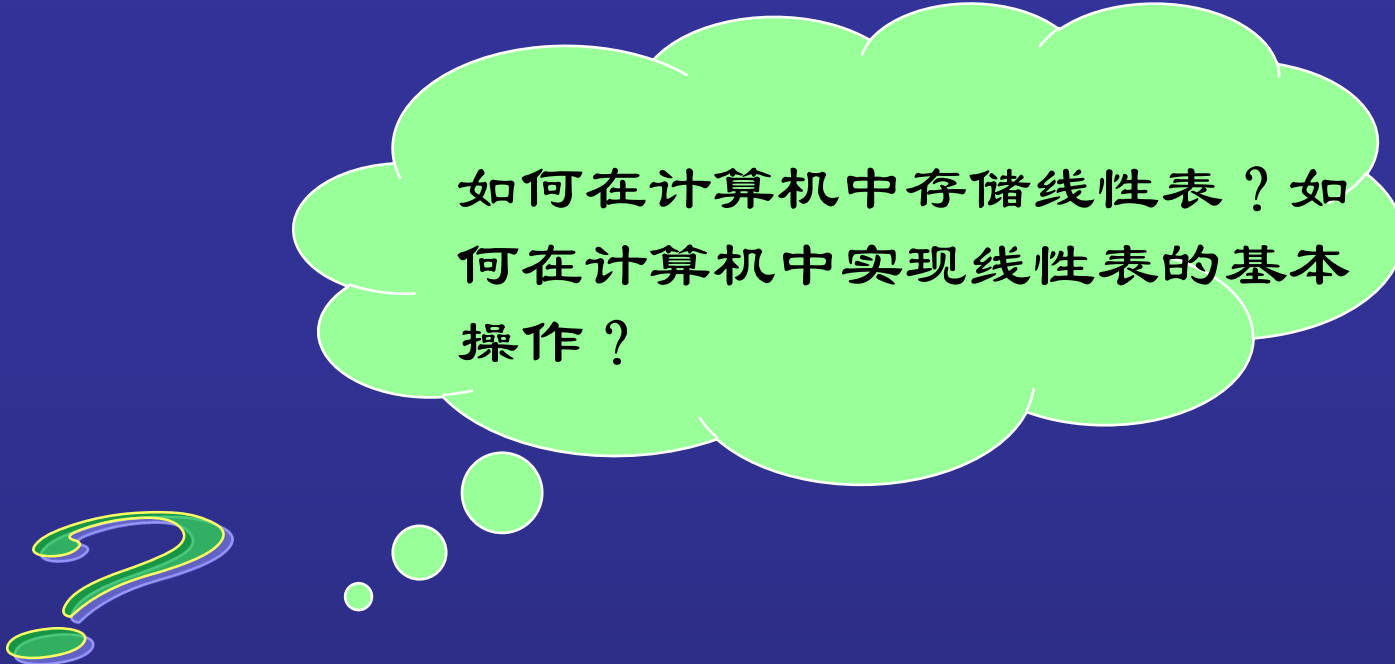

- 1 存取操作：存取线性表中第 i 个数据元素；
- 2 查找操作：在线性表中查找满足条件元素；
- 3 插入操作：在线性表的第 i 个元素之前插入一个新元素；
- 4 删除操作：删除线性表的第 i 个元素；
- 5 分解操作：将一个线性表拆分为多个线性表；
- 6 合并操作：
- 7 排序



2.1 线性表的概念

说明

- 1 上面列出的操作，只是线性表的一些常用的基本操作；
- 2 不同的应用，基本操作可能是不同的；
- 3 线性表的复杂操作可通过基本操作实现；



如何在计算机中存储线性表？如何在计算机中实现线性表的基本操作？

为了存储线性表，至少要保存两类信息：

- 1) 线性表中的数据元素；
- 2) 线性表中数据元素的顺序关系；

2.2 线性表的顺序存储和实现

- 一 线性表的顺序存储结构——顺序表
- 二 顺序表的基本操作算法
- 三 效率分析

2.2 线性表的顺序存储和实现

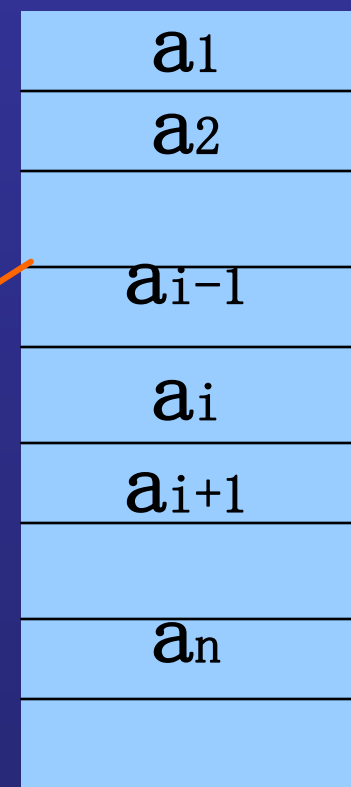
一 线性表的顺序存储结构——顺序表

线性表的顺序存储结构

线性表的顺序存储结构，就是用一组连续的内存单元依次存放线性表的数据元素。

用顺序存储结构存储的线性表——称为顺序表

线性表 ($a_1, a_2, a_3, \dots, a_n$) 的顺序存储结构



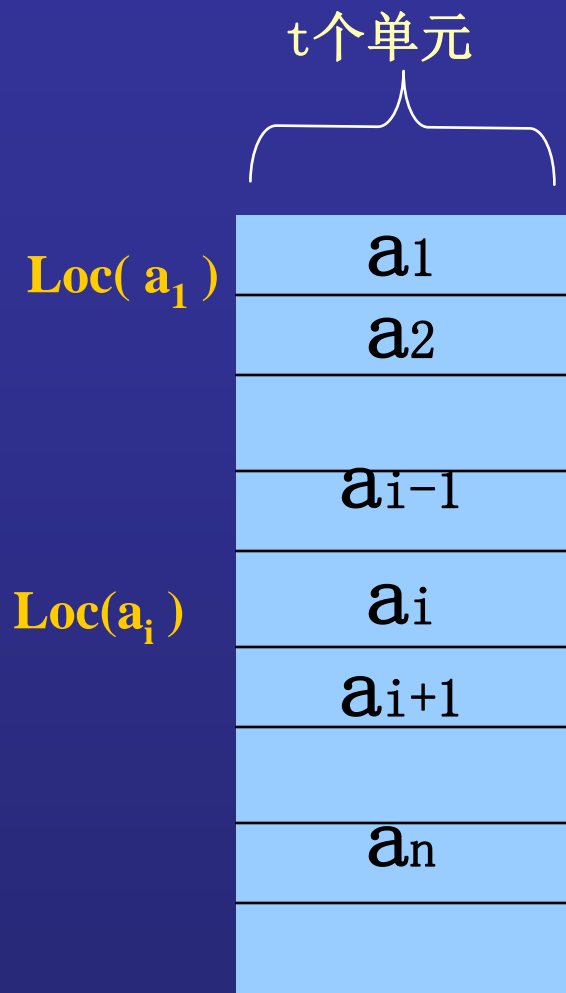
2.2 线性表的顺序存储和实现

说明:

- 在顺序存储结构下, 线性表元素之间的逻辑关系, 通过元素的存储顺序反映 (表示) 出来;

- 假设线性表中每个数据元素占用 t 个存储单元, 那么, 在顺序存储结构中, 线性表的第 i 个元素的存储位置与第 1 个元素的存储位置的关系是:

$$\text{Loc}(a_i) = \text{Loc}(a_1) + (i - 1) t$$



2.2 线性表的顺序存储和实现

可用C语言中的一维数组来表示,但数组不是线性表,数组存放的是线性表,数组的类型由线性表的性质决定.

如:

```
#define MAX 100
```

```
int v[MAX];
```

2.2 线性表的顺序存储和实现

顺序表的类型定义

```
#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 struct{
    ElemType *elem; //线性表存储空间基址
    int length; //当前线性表长度
    int listsize; //当前分配的线性表存储空间大小
                //（以sizeof(ElemType)为单位）
}SqList;
```

SqList：类型名，

SqList类型的变量是结构变量，它的三个域分别是：

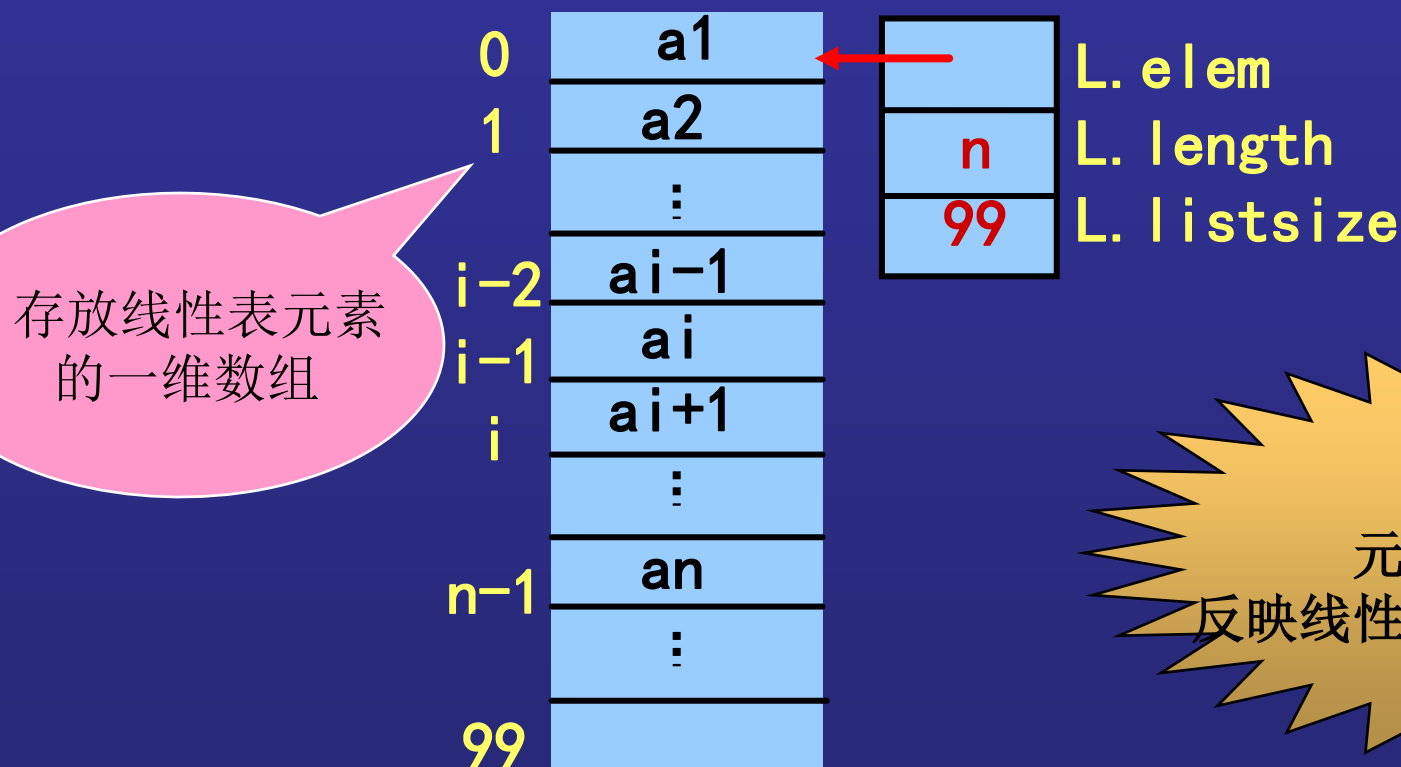
***elem**：存放线性表元素的一维数组基址；其存储空间在初始化操作（建空表）时动态分配；

length：存放线性表的表长；

listsize：用于存放当前分配（存放线性表元素）的存储空间的大小。

2.2 线性表的顺序存储和实现

设 $A = (a_1, a_2, a_3, \dots, a_n)$ 是一线性表， L 是 `SqList` 类型的结构变量，用于存放线性表 A ，则 L 在内存中的状态如图所示：



二、顺序表的基本操作算法

1. 插入 $\text{insert}(v, x, i)$

功能：在顺序表 v 中的第 i ($1 \leq i \leq n+1$)个数据元素之前插入一个新元素 x , 插入前线性表为

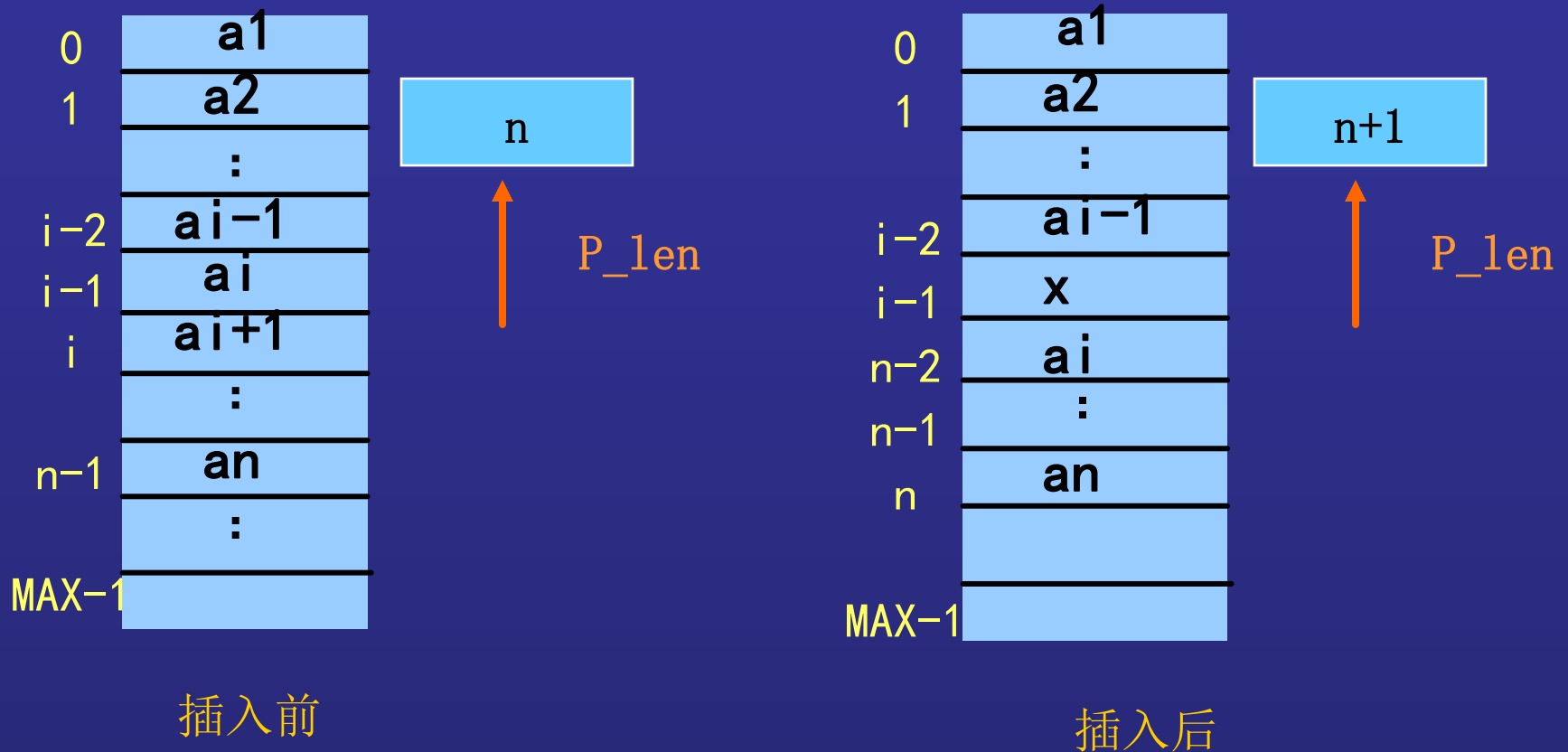
$(a_1, a_2, a_3, \dots, a_{i-1}, a_i, \dots a_n)$

插入后, 线性表长度为 $n+1$, 线性表为

$(a_1, a_2, a_3, \dots, a_{i-1}, x, a_i, \dots a_n)$

2.2 线性表的顺序存储和实现

插入操作示意图：



插入操作算法

```
int insert ( int v[ ], int i, int x, int * p_len)
{
    int j;
    if (i<=0|| i>*p_len) return ( 0 ); /* i 值不合法
    for ( j=*p_len , j>= i; j--)
        v[j]= v[ j-1];
    v[i-1]=x ; *p_len++; /* 插入x , 表长增1
    return (1);
}
```



2.2 线性表的顺序存储和实现

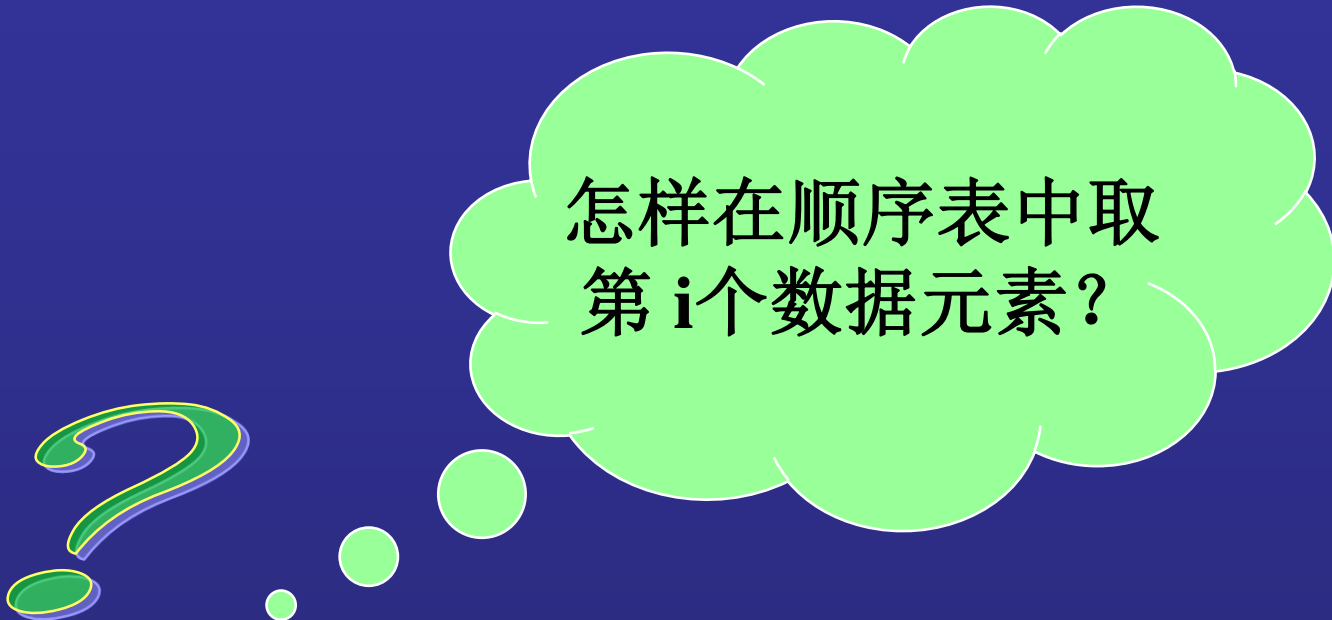
删除算法的主要步骤

- 1) 若 i 不合法或表 L 空, 算法结束, 并返回 0; 否则转2)
- 2) 将第 i 个元素之后的元素 (不包括第 i 个元素) 依次向前移动一个位置;
- 3) 表长-1

删除操作算法

```
int delete ( int v[ ], int i, int * p_len)
{
    int j;
    if (i<=0|| i>*p_len) return ( 0 ); /* i 值不合法
    for ( j=i , j<*p_len; j++)
        v[j-1]= v[ j];
        *p_len - - ;
    return (1);
}
```

2.2 线性表的顺序存储和实现



怎样在顺序表中取
第 i 个数据元素？

2.2 线性表的顺序存储和实现

算法时间复杂度分析

算法时间复杂度取决于移动元素的个数，移动元素的个数不仅与表长有关，而且与插入位置有关。

插入位置	移动元素个数
1	n
2	$n-1$
i	$n-i+1$
n	1
$n+1$	0

1	a_1
	a_2
...	
$i-1$	a_{i-1}
i	a_i
.	a_{i+1}
n	a_n

p_i : 在第*i*个元素之前插入元素的概率, 在长度为*n*的顺序表中插入一个元素, 所需移动元素个数数学期望值为:

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

假设在线性表的任何位置插入元素的概率相同, 即

$$p_i = 1/(n+1)$$

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

由此可见

- 在顺序表中插入一个元素, 平均要移动表的一半元素。
- 表长为*n*的顺序表, 插入算法的时间复杂度为 $O(n)$

2.2 线性表的顺序存储和实现

顺序表是线性表最简单的一种存储结构

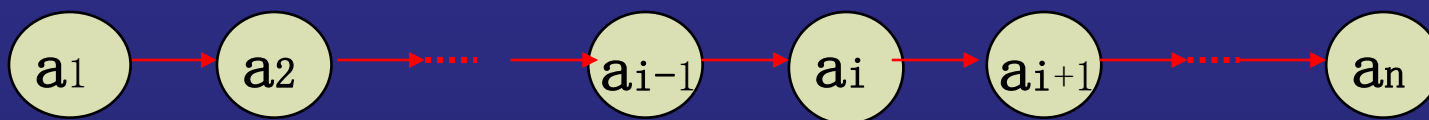
小结

顺序表的特点：

- 1 通过元素的存储顺序反映 线性表中数据元素之间的逻辑关系；
- 2 可随机存取顺序表的元素；
- 3 顺序表的插入、删除操作要通过移动元素实现；

2.3 线性表的链式存储和实现

线性表的链式存储结构是用一组任意的存储单元存储线性表的各个数据元素。为了表示线性表中元素的先后关系，每个元素除了需要存储自身的信息外还需保存直接前趋元素或直接后继元素的存储位置。



2.3 线性表的链式存储和实现

2.3.1 线性链表

- 一 线性链表的概念
- 二 线性链表的基本操作算法
- 三 线性链表的其它操作

2.3.2 循环链表

2.3.3 双向链表

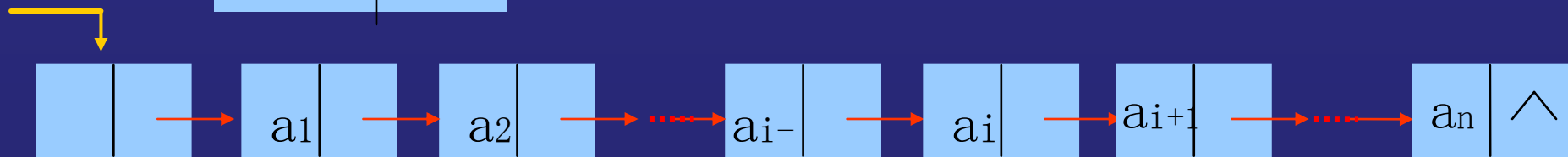
- 一 双向链表的概念
- 二 双向链表的基本操作算法

1 线性链表

用一组任意的存储单元存储线性表中的数据元素，对每个数据元素除了保存自身信息外，还保存了直接后继元素的存储位置。

1010	a4	0
1012		
1014	a3	1010
1016		
1018		
1020	a1	1024
1022		
1024	a2	1014
1026		

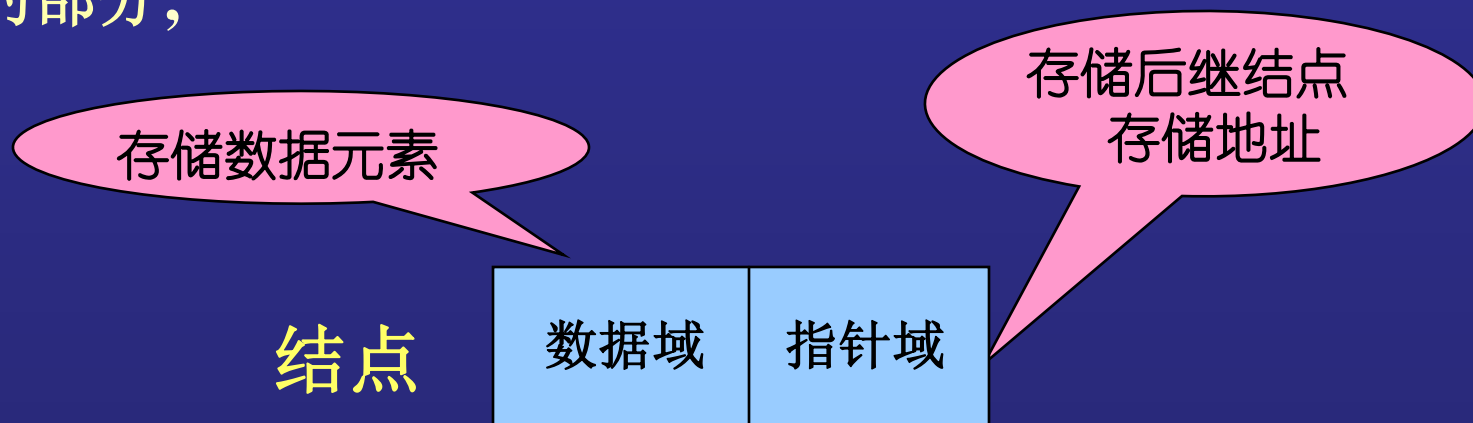
用线性链表存储线性表时，数据元素之间的关系是通过保存直接后继元素的存储位置来表示的



结点：数据元素及直接后继的存储位置（地址）组成一个数据元素的存储结构，称为一个结点；

结点的数据域：结点中用于保存数据元素的部分；

结点的指针域：结点中用于保存数据元素直接后继存储地址的部分；



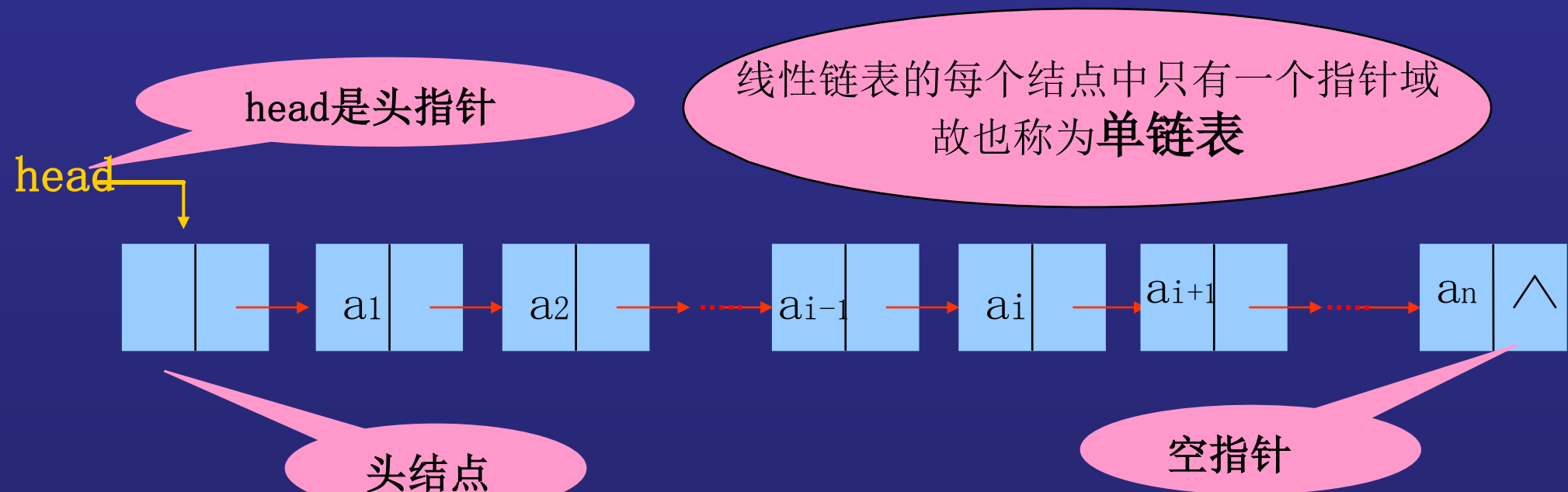
2.3.1 线性链表

头指针：用于存放线性链表中第一个结点的存储地址；

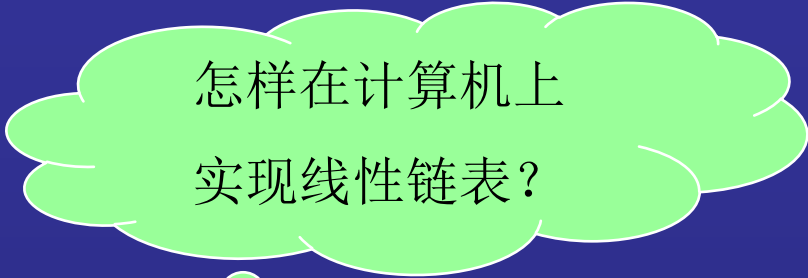
空指针：不指向任何结点，线性链表最后一个结点的指针通常是空指针；

头结点：线性链表的第一元素结点前面的一个附加结点，称为头结点；

带头结点的线性链表：第一元素结点前面增加一个附加结点的线性链表称为带头结点的线性链表；



2.3.1 线性链表



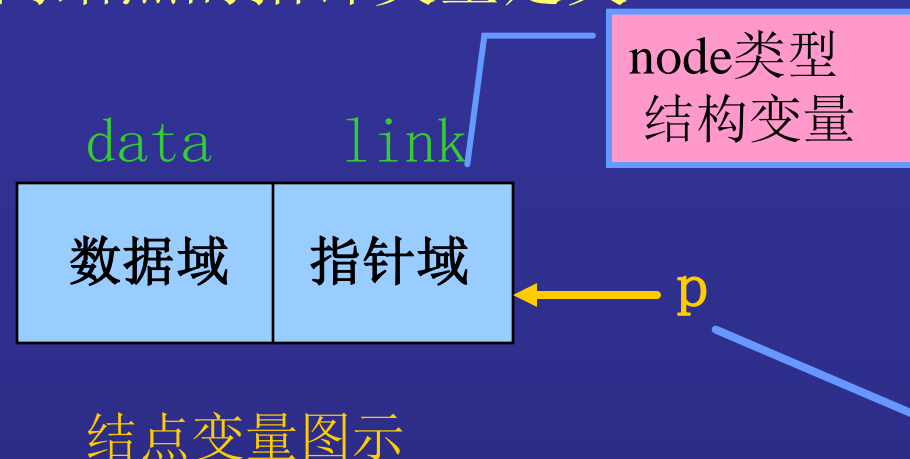
怎样在计算机上
实现线性链表?



2.3.1 线性链表

线性链表的结点类型定义及指向结点的指针类型定义

```
struct node{  
    int data;  
    Struct node *link;  
}typedef struct node NODE;
```



Node: 结构类型名;

Node类型结构变量有两个域:

data: 用于存放线性表的数据元素,

link: 用于存放元素直接后继结点的地址;

该类型结构变量用于表示线性链表中的一个结点;

NODE *p: p为指向结点(结构)的指针变量;

2.3 线性链表

两个C 函数

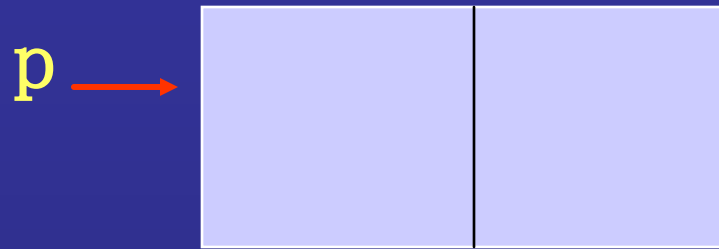
1) **malloc(int size)**

功能： 在系统内存中分配size个的存储单元，并返回该空间的基址。

使用方法：

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

为p申请一个结点



`p = (NODE *)malloc(sizeof(NODE));` 图
示

2.2 线性链表

2) free (p)

功能：将指针变量p所指示的存储空间，回收至系统内存空间中去

使用方法：

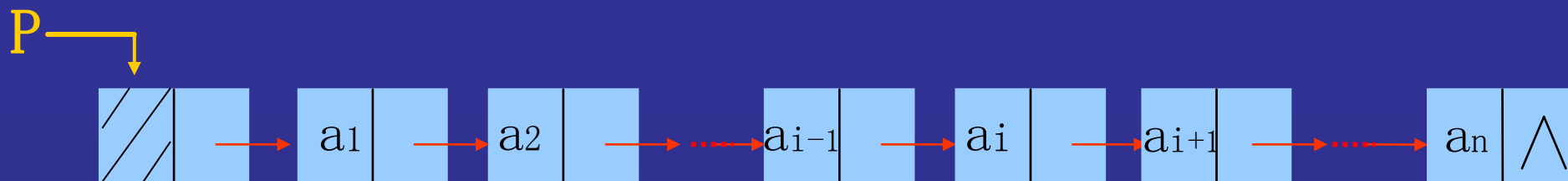
调用free (p)

```
...  
NODE *p;           p →  
p = (NODE *) malloc (sizeof (NODE));  
// 一旦p所指示的内存空间不再使用,  
//调用free( ) 回收之  
free(p);
```

调用free (p) 图示

2.3.1 线性链表

二 线性链表基本操作的算法



如何在线性链表
上实现线性表的基本操作？
如何建表？如何插入？删除？

约定用带头结点的线性链表
存储线性表

2.3.1 线性链表

1) 初始化操作

功能： 建空线性链表

参数： head 为线性链表的头指针

主要步骤： 调用malloc ()分配一结点的空间,并将其地址赋值给head;

算法：

```
NODE *create_head ( )
```

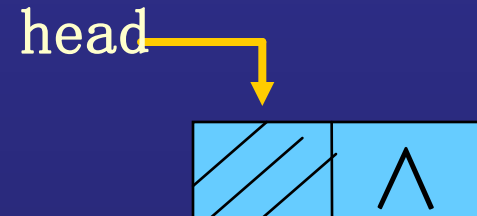
```
{NODE *head
```

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

```
    head->link = null;
```

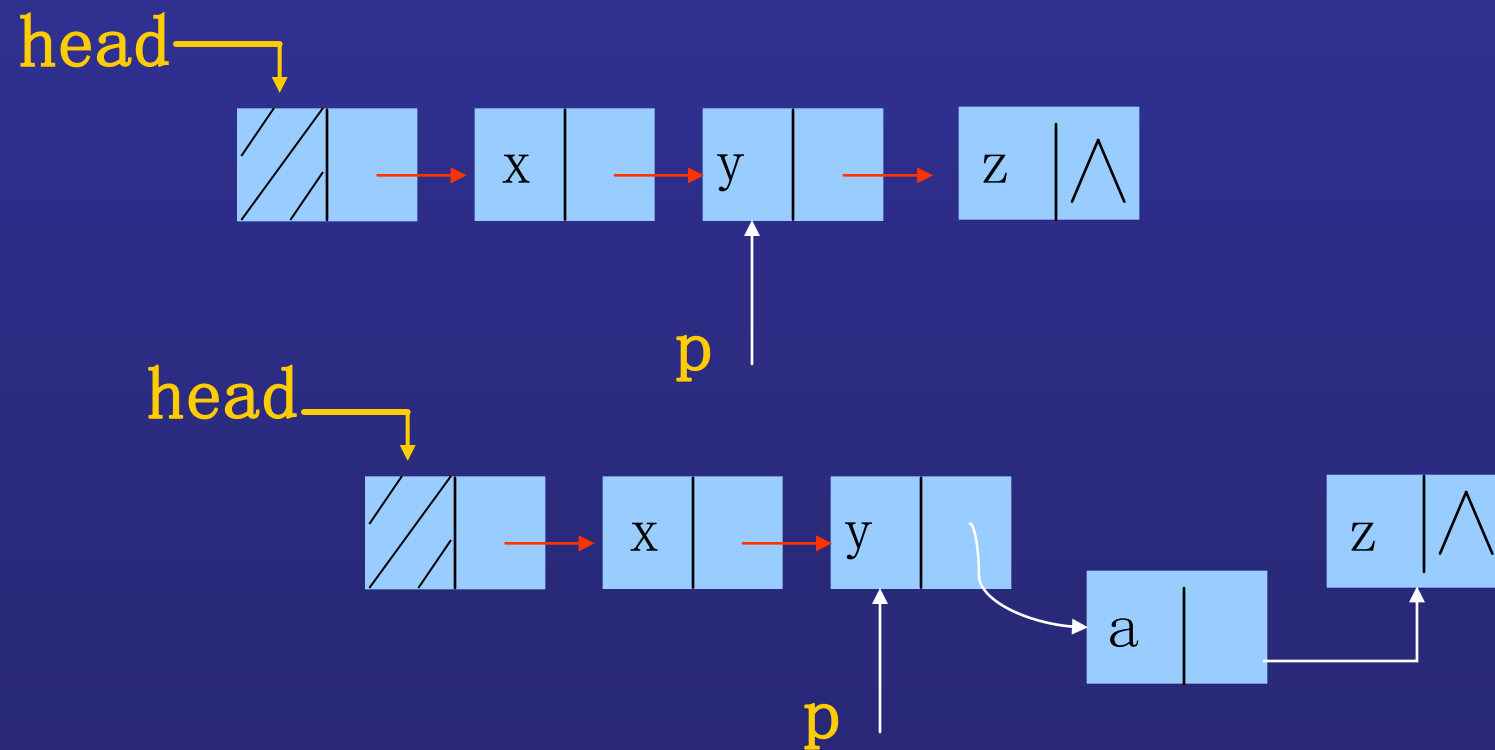
```
    Return (head);
```

```
}
```



- **NODE *create_link(n) /*建立有n个结点的线性单链表的算法**
{ NODE *head, *p, *q;
int i;
p=(NODE *)malloc(sizeof(NODE));head=p;
for(i=1;i<=n;i++)
{q= (NODE *)malloc(sizeof(NODE));
q->data=i; q->link=NULL;
p->link=q;p=q;
}
return(head);
}

```
q=(NODE *)malloc(sizeof(NODE));  
q->deta='a';  
q->link=p->link;  
p->link=q;
```

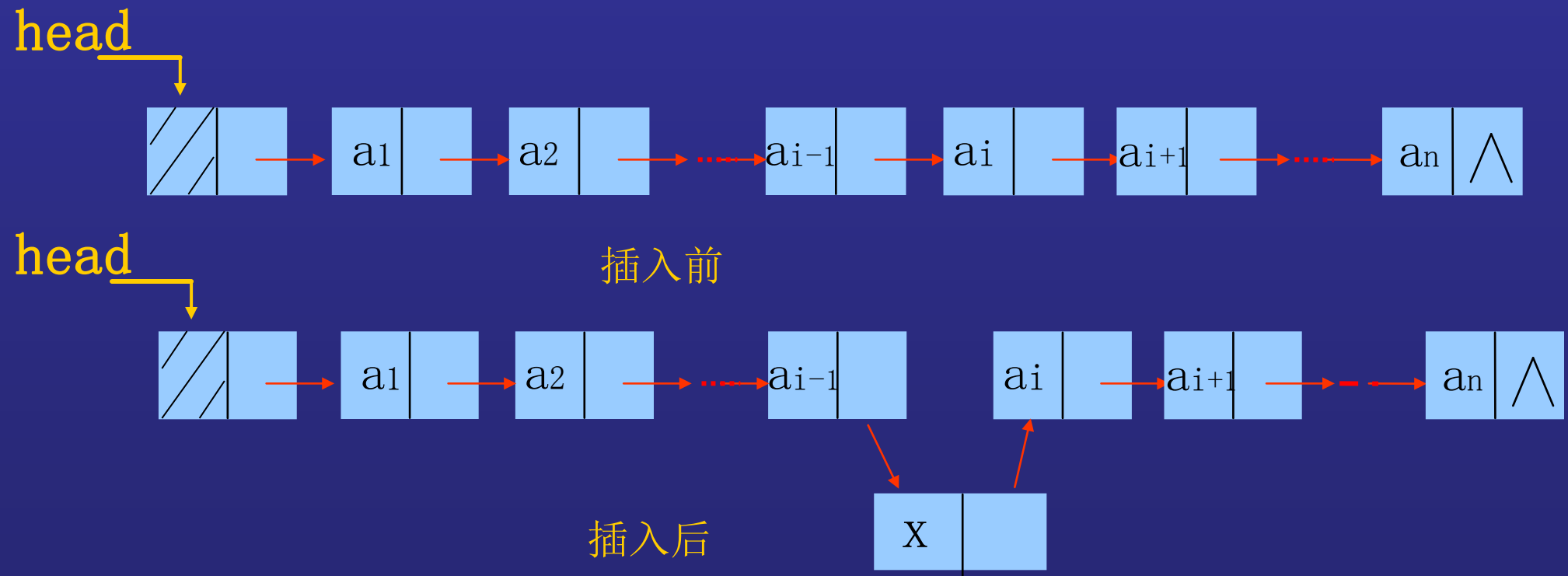


2.3.1 线性链表

1 3 插入操作 `Insert_link(NODE *head, int x, int i)`

功能：在线性链表的第 i 个元素结点之前插入一个新元素结点 x ；

插入操作图示：






2.3.1 线性链表

插入操作主要步骤:

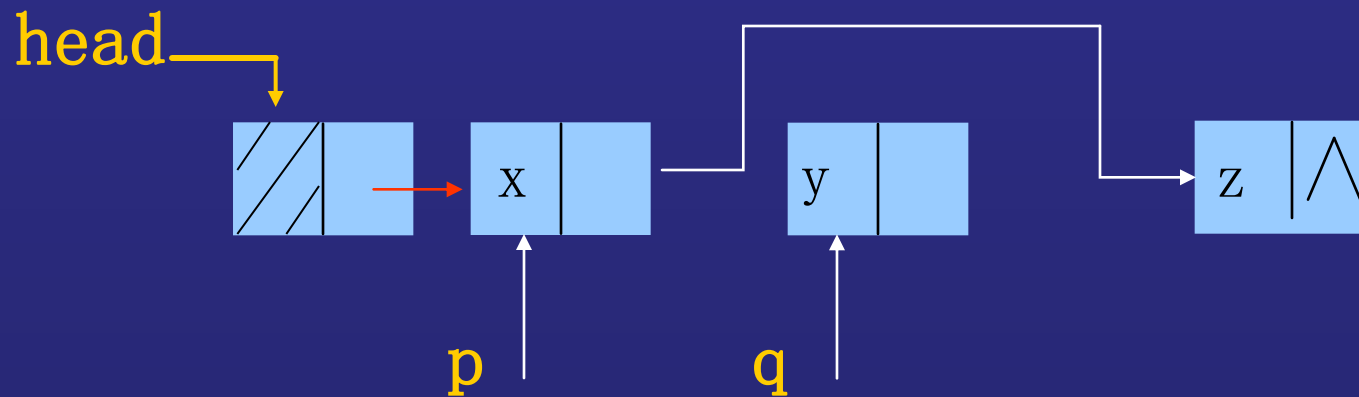
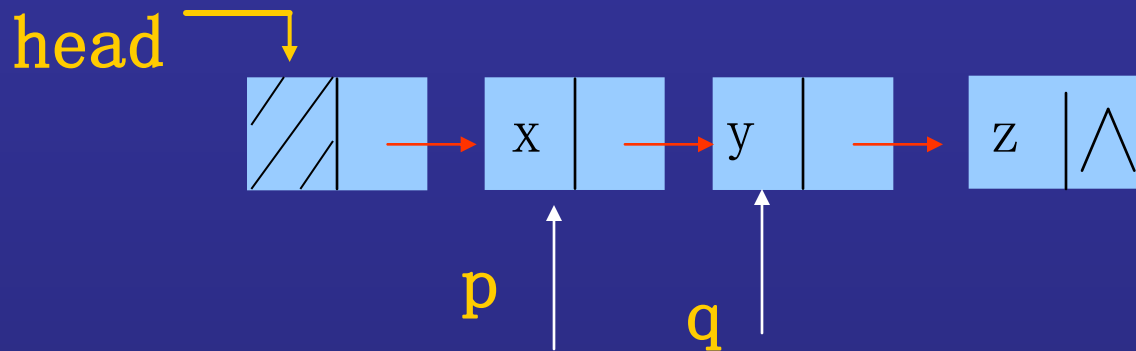
- 1) 查找链表L的第 $i-1$ 个元素结点;
- 2) 为新元素建立结点;
- 3) 修改第 $i-1$ 个元素结点的指针和新元素结点指针完成插入;



- Int insert_link(NODE *head,int x, int i)
{ NODE *p,*s;int j;
p=head;j=0;
while ((p!=NULL)&&(j<i-1))
{p=p->link;j++;}
if(p==NULL)return(0);
s=(NODE *(malloc(sizeof(NODE)));
s->data=x;
s->link=p->link;
p->link=s;
return(1);
}

删除结点

- $q = p \rightarrow \text{link};$
 $p \rightarrow \text{link} = q \rightarrow \text{link};$
 $\text{free}(q);$

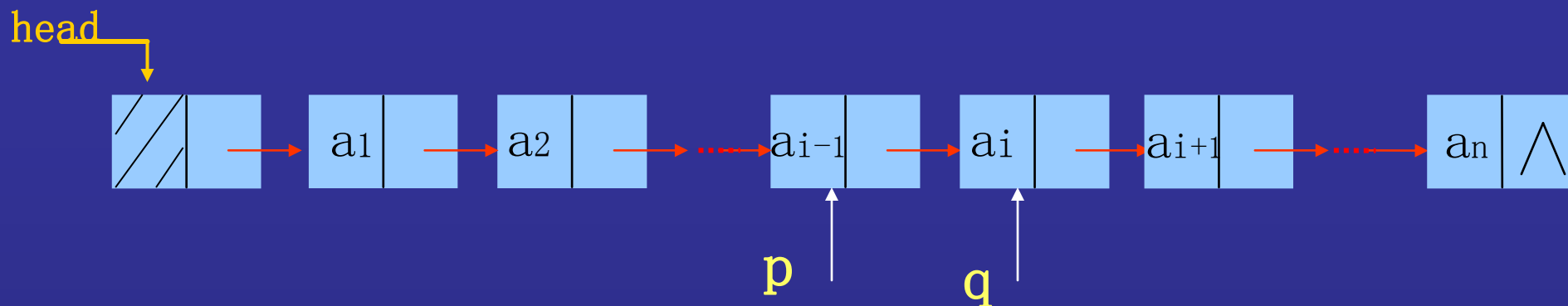


2.3.1 线性链表

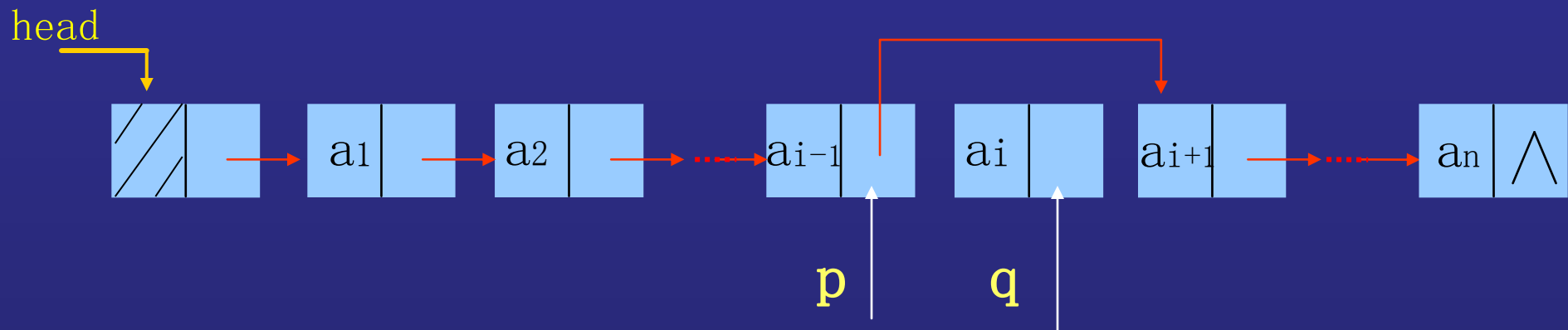
删除操作主要步骤:

- 1) 查找链表的第 $i-1$ 个元素结点,使指针 p 指向它, 将指针 q 指向第 i 个结点;
- 2) 修改第 $i-1$ 个元素结点指针, 使其指向第 i 个元素结点的后继;
- 3) 回收 q 指针所指的第 i 个结点空间;

2.3.1 线性链表



删除前



删除后

- 在线性链表中删除第i个结点
- ```
Int delete_link(NODE *head, int i)
{ NODE *p,*q;int j;
 p=head;j=0;
 while ((p!=NULL)&&(j<i-1))
 {p=p->link;j++;}
 if(p==NULL)return(0);
 q=p->link;p->link=q->link;free(q);
 return(1);
}
```

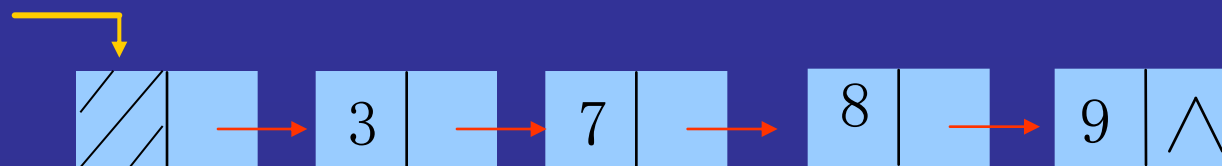
### 三 线性链表的其他操作的实现

例：将两个递增有序线性链表归并成一个递减有序表。

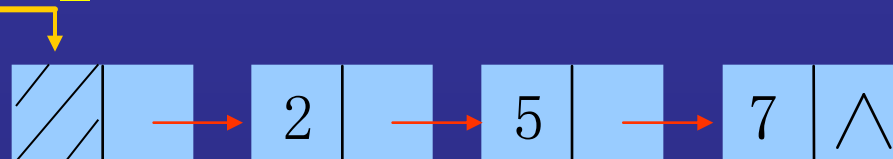
设线性表A、B分别用头指针为head\_a、head\_b的两个带头结点的线性链表存储, 归并后的递减有序表头指针为head, 将两表数据较小的结点依次取出插入到新表



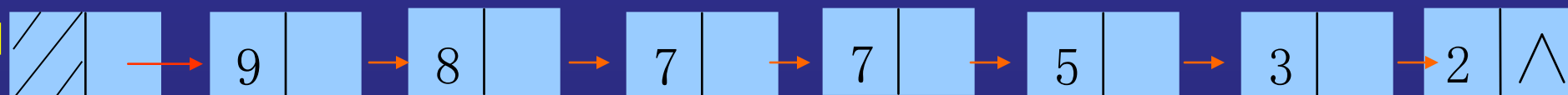
head\_a



head\_b




head





```
NODE * merge_link(NODE *head_a, NODE *head_b)
{ NODE *p, *q, *head;
 head=(NODE *)malloc(sizeof (NODE));
 head->link=NULL;
 p=head_a->link; q=head_b->link;
 while((p!=NULL) &&(q!=NULL))
 If (p->data<q->data)
 {head_a->link=p->link;p->link=head->link;
 head->link=p; p=head_a->link;}
 else {head_b->link=q->link;q->link=head->link;
 head->link=q; q=head_b->link;}
```



```
while(p!=NULL)
 {head_a->link=p->link;p->link=head->link;
 head->link=p; p=head_a->link;}
while(q!=NULL)
 {head_b->link=q->link;q->link=head->link;
 head->link=q; q=head_b->link;}
free(head_a); free(head_b);
return(head);
}
```

## 2.3.1 线性链表小结

线性链表是线性表的一种链式存储结构

线性链表的特点

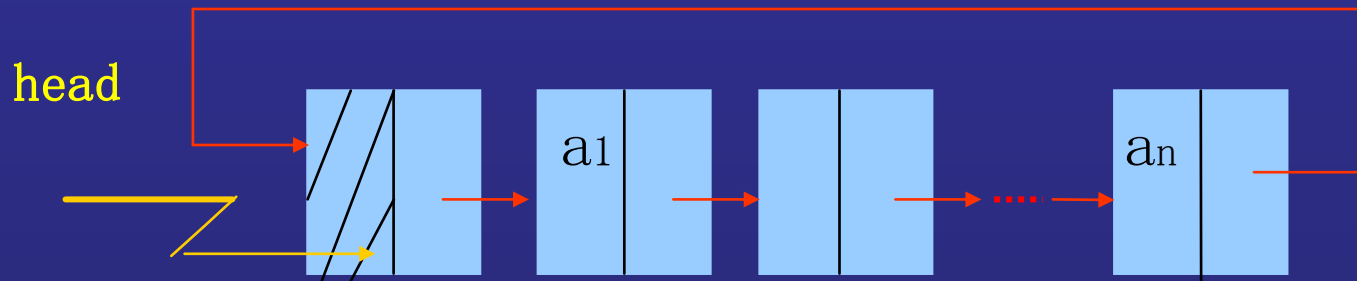
- 1 通过保存直接后继元素的存储位置来表示数据元素之间的逻辑关系；
- 2 插入删除操作通过修改结点的指针实现；
- 3 不能随机存取元素；

## 2.4 单向循环链表

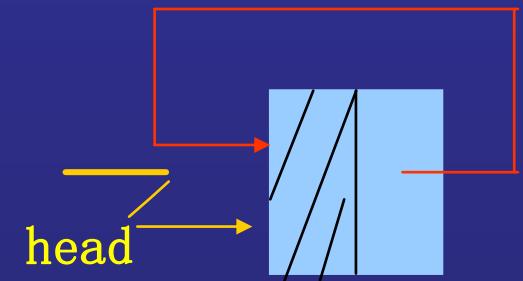
### 1 循环链表的概念

循环链表是线性表的另一种链式存储结构，它的特点是将线性链表的最后一个结点的指针指向链表的第一个结点

### 2 循环链表图示



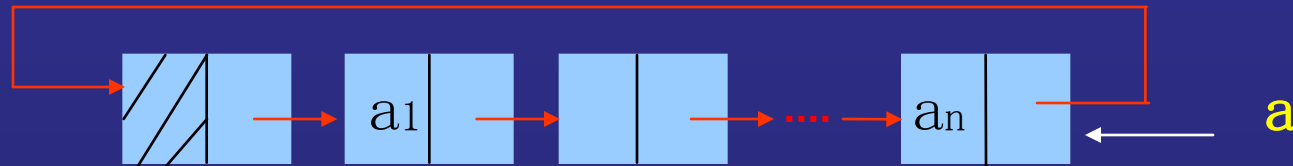
(a) 非空表



(b) 空表

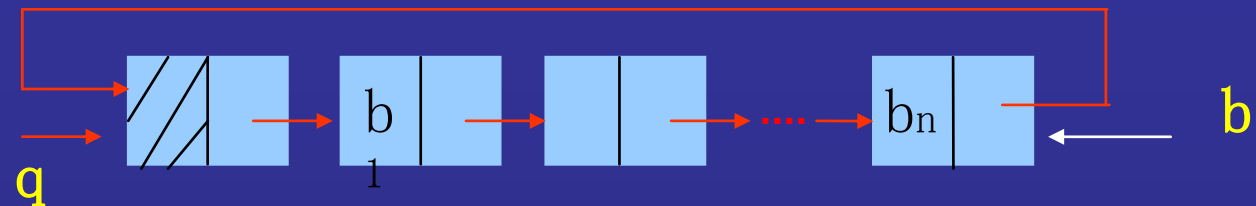
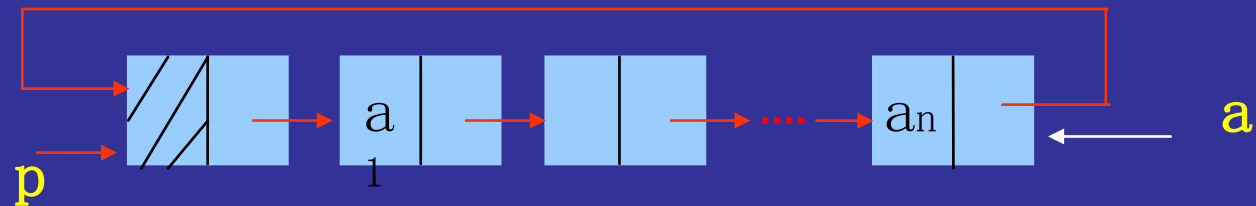
说明

- 在解决某些实际问题时循环链表可能要比线性链表方便些。如将一个链表链在另一个链表的后面；
- 循环链表与线性链表操作的主要差别是算法中循环结束的条件不是 $p$ 或 $p \rightarrow \text{link}$ 是否为NULL,而是它们是否等于首指针；
- 对循环链表，有时不给出头指针，而是给出尾指针

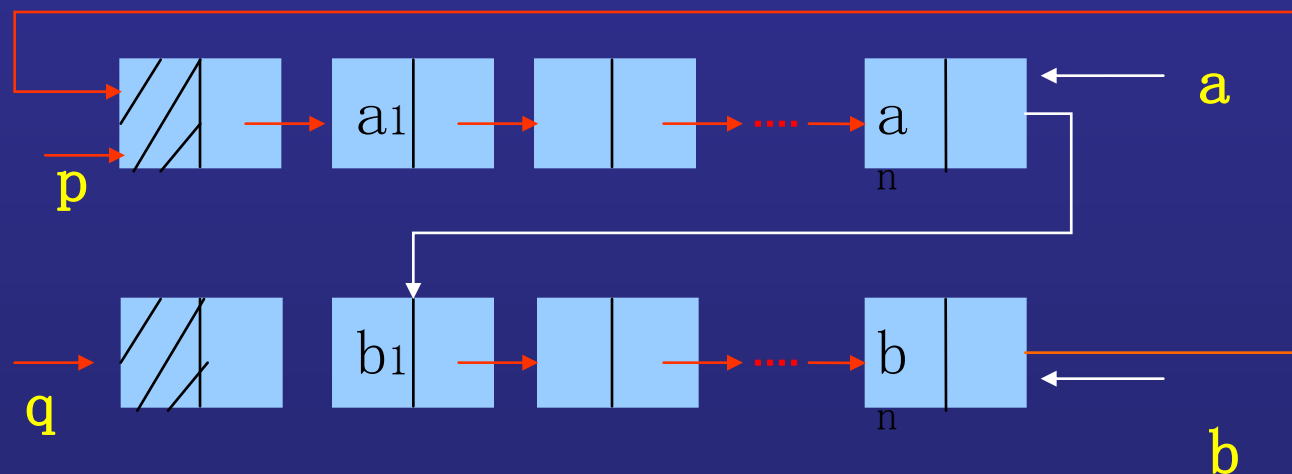


给出尾指针的循环链表

## 2.4 单向 循环链表



```
p=a->link;
q=b->link;
a->link=q->link;
b->link=p;
free(q);
```



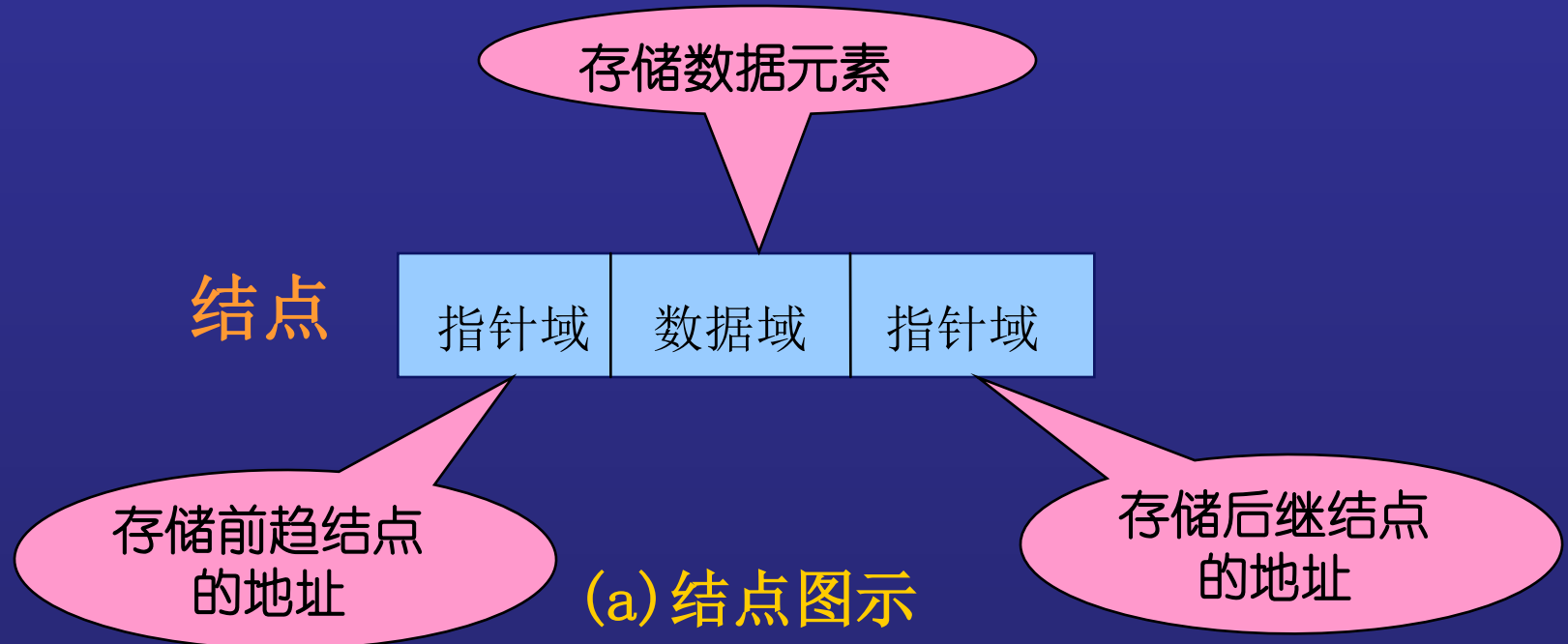
## 约瑟夫回环问题

```
Void lead (NODE * head, int m)
{ NODE *p, *q; int i;
 p=head; q=NULL; i=1;
 while(p->link!=p)
 { while((i<m)&&(p->link!=p))
 { i++; q=p; p=p->link;}
 if (p->link!=p)
 { i=1; printf(“%d”, p->data);
 q->link=p->link; q=p; p=p->link; free(q);}
 else printf(“%d”, p->data);
 }
}
```

## 2.5 双向链表

### 1 双向链表的概念

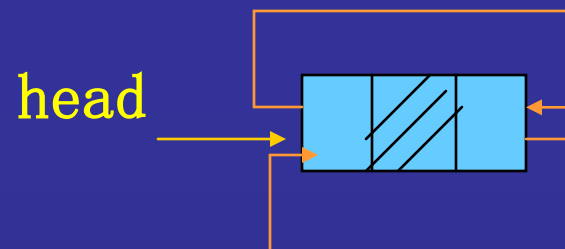
双向链表中，每个结点有两个指针域，一个指向直接后继元素结点，另一个指向直接前趋元素结点。



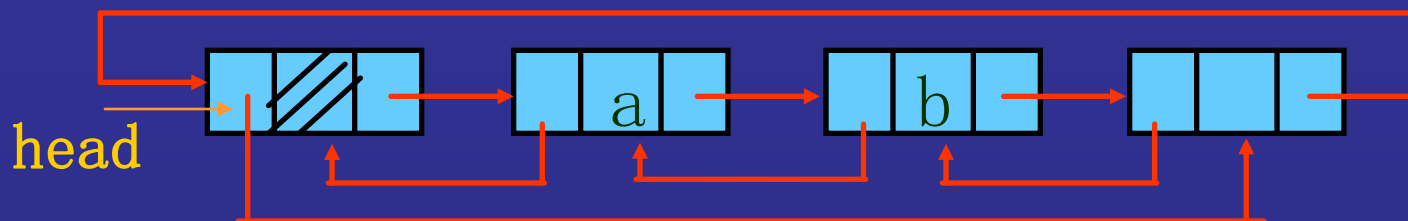


## 2.5 双向链表

### 2 双向链表图示



(b) 空的双向循环链表



(c) 非空的双向循环链表

**Struct node**

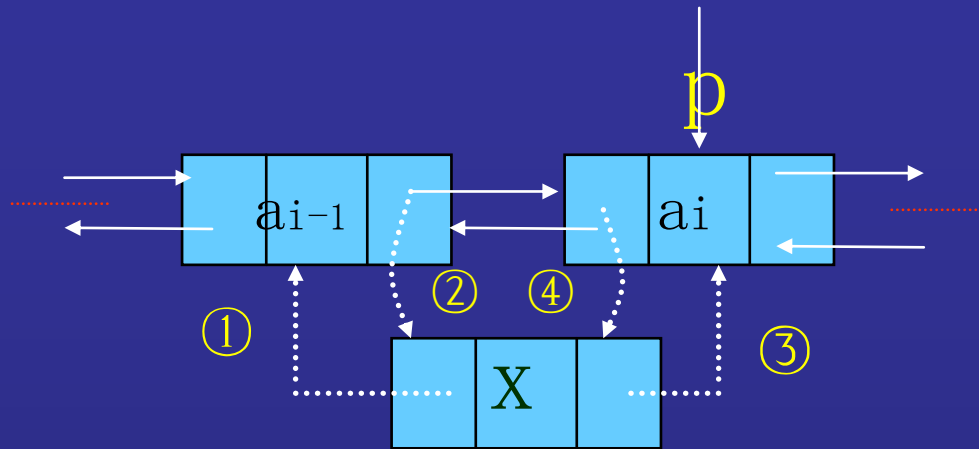
```
{ int data
```

```
 struct node * llink, *rlink;
```

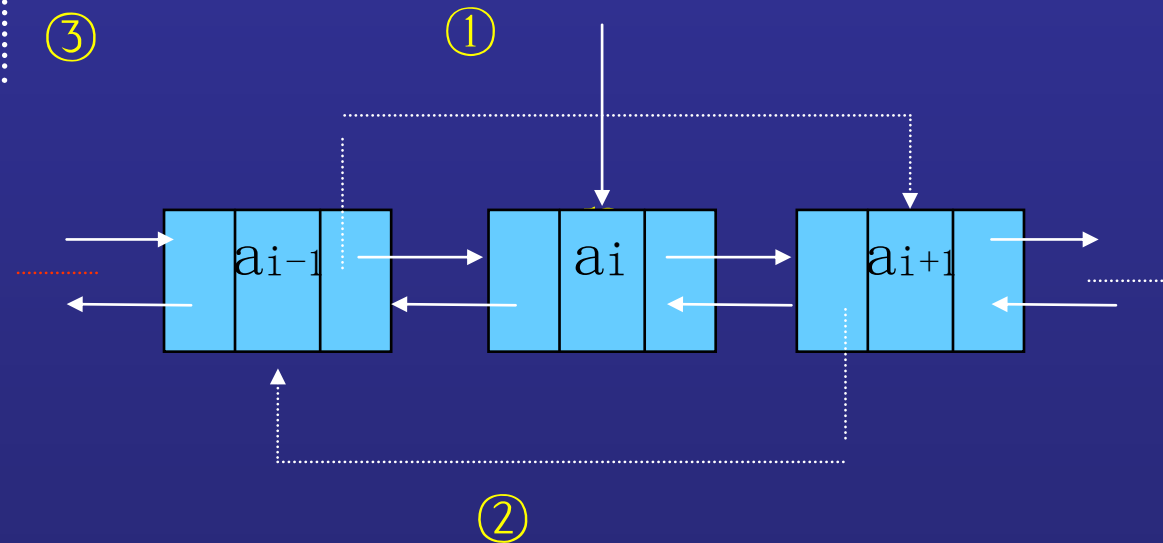
```
}
```

## 2.5 双向链表

### 3、双向链表的基本操作算法



在双向链表中插入一个  
结点时指针的变化情况



在双向链表中删除结点时指针变  
化情况

1) 插入操作算法 (在p 所指结点之后插入q 结点的过程)

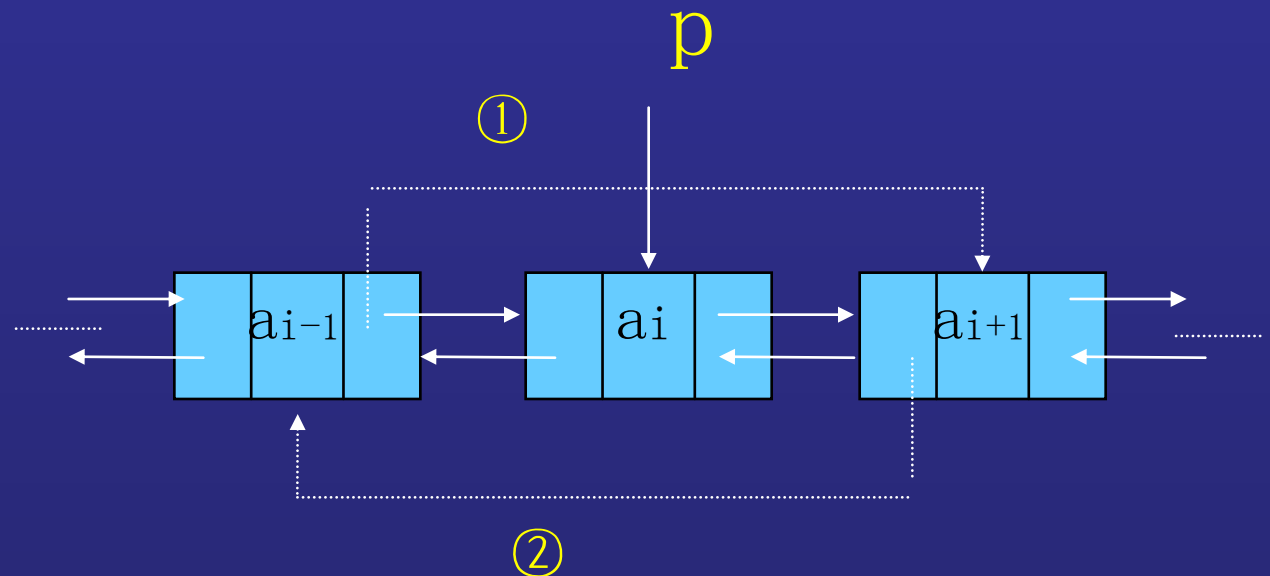
```
q=(NODE *)malloc(sizeof (NODE));
```

```
q->data=x;
```

```
q->rlink=p->rlink; q->llink=p;
```

```
p->rlink=q; (q->rlink)->llink=q;
```

## 2) 删除操作算法

$$(p \rightarrow \text{llink}) \rightarrow \text{rlink} = p \rightarrow \text{rlink};$$
$$(p \rightarrow \text{rlink}) \rightarrow \text{llink} = p \rightarrow \text{llink};$$
$$\text{free}(p);$$


## 2.6 一元多项式的表示及相加

### 一、一元多项式的表示

数学上:

$$P(x) = p_0 + p_1 x + p_2 x^2 + \dots + p_n x^n$$

$$Q(x) = q_0 + q_1 x + q_2 x^2 + \dots + q_m x^m$$

不失一般性, 设  $m < n$  则两个多项式相加可表为

$$R(x) = (p_0 + q_0) + (p_1 + q_1)x + \dots + (p_m + q_m)x^m + p_{m+1}x^{m+1} + \dots + p_n x^n$$

计算机 领域:

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

$$Q = (q_0, q_1, q_2, \dots, q_m)$$

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

为用计算机实现多项式运算,  
如何表示一元多项式?



## 2. 4 一元多项式的表示及相加

例:  $S(x) = 1 + 3x^{1000} + 2x^{2000}$

系数线性表 ( 1, 0, ..., 0, 3, 0, ..., 0, 2 )

非零系数指数线性表: ((1,0), (3,1000), (2,2000))

例: 求两多项式的和多项式

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

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

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

$$A = ((7, 0), (3, 1), (9, 8), (5, 17))$$

$$B = ((8, 1), (22, 7), (-9, 8))$$

$$C = ((7, 0), (11, 1), (5, 17))$$

如何存储一元多项式?



## 2 一元多项式链式存储结构

为用线性链表表示一元多项式，我们要给出数据元素的类型定义

## 1) 元素的类型定义

```
struct node{
int coef; //存储项系数
int index; //存储项指数
struct node * next

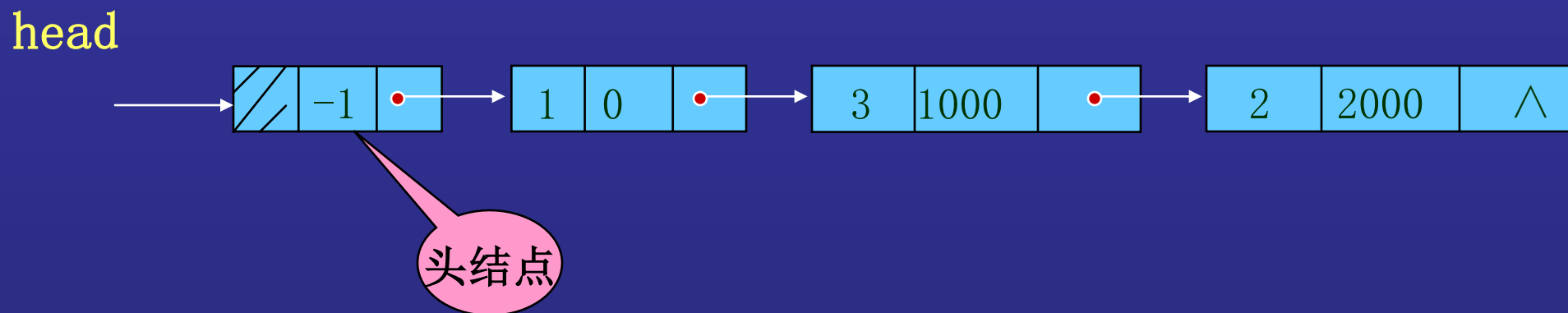
} ;typedef struct node NODE;
```

coef    index    next



## 2. 6 一元多项式的表示及相加

### 一元多项式链式存储结构图示





## 2. 4 一元多项式的表示及相加

### 三、一元多项式的相加算法

#### 一元多项式的相加

例：求两多项式的和多项式

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

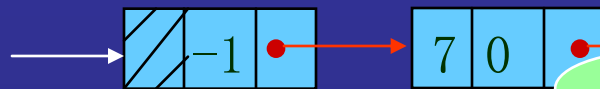
$$B(x) = 8x + 22x^7 - 9x^8$$

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

## 2. 4 一元多项式的表示及相加

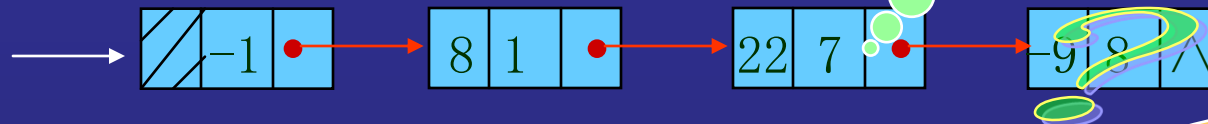
设多项式 $A(x)$ ， $B(x)$ 分别用带表头结点的线性链表 $ah$ ， $bh$ 表示，和多项式 $C(x)$ 用带表头结点的线性链表 $ch$ 表示

$ah$



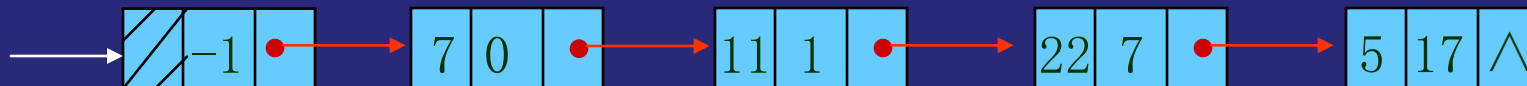
如何实现用这种线性链表表示的多项的加法运算？

$bh$



和多项式链表

$ch$



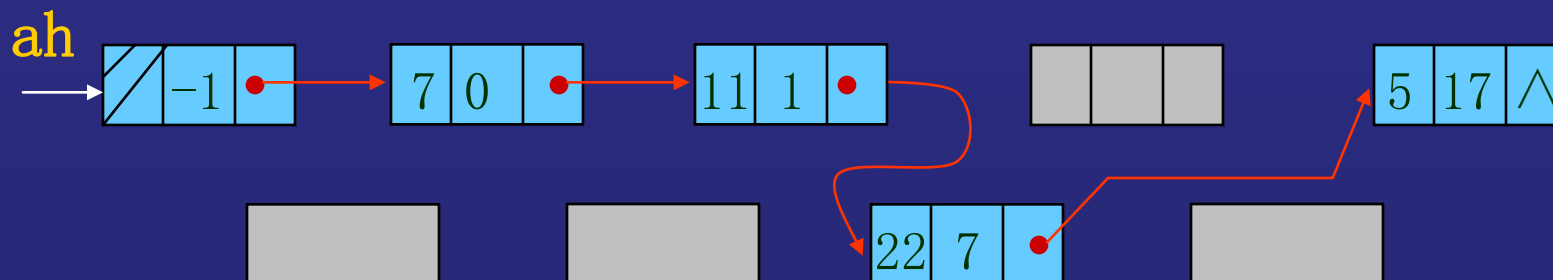
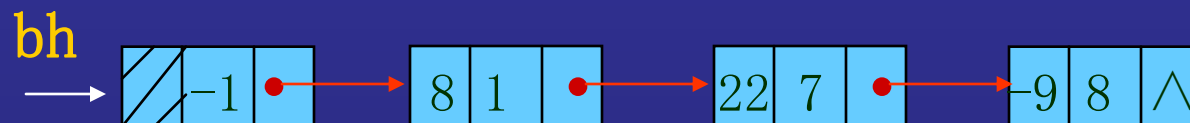
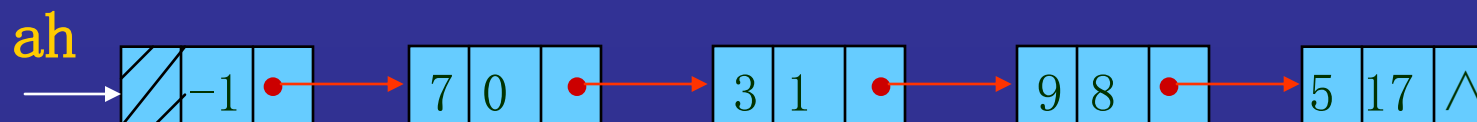
$ch=ah+bb$

$ah=ah+bb$

$bh=ah+bh$

## 2. 6 一元多项式的表示及相加

一元多项式的相加算法图示



## 2. 6 一元多项式的表示及相加

### 一元多项式加法算法主要步骤

分别对两个链表ah、bh进行扫描，设p、q分别指向线性链表ah、bh的当前进行比较的某个结点：

$p \rightarrow \text{index} < q \rightarrow \text{index}$  : p所指结点应为和多项式中的结点


$p \rightarrow \text{index} = q \rightarrow \text{index}$  : 将p所指结点的系数“加”到q所指结点的系数上相加；

$p \rightarrow \text{index} > q \rightarrow \text{index}$  : 从表bh中删除q所指结点，并将其插入到ah表p所指结点之前；


## 一元多项式的相加算法（算法2.8）

```
void polyadd(NODE *ah, NODE * bh)
{ NODE *pre_p, *p, *q, *temp;
 char comp;
 pre_p=ah; p=ah->next; q=bh->next;
 while ((p!=NULL)&&(q!=NULL))
 { comp=compare(p->index, q->index)
 switch(comp)
```

（接下页）



续 { case '<': // 多项式A中当前结点的指数值小  
pre\_p=p; p=p->next; break;  
case '=': //两者的指数值相等  
p->coef+=q->coef ;  
if (p->coef ==0.0) { // 合并后系数为0  
pre\_p->next=p->next; free(p); }  
else pre\_p=p;  
p=pre\_p->next; temp=q; q=q->next;  
free(temp); break;  
case '>': //多项式A中当前结点的指数值大  
temp=q->next; q->next=p;  
pre\_p->next=q; pre\_p=q; q=temp;  
}  
}



```
if (q!=NULL) pre_p->next=q;
free(bh);
}
```

```
char compare (int n1, int n2)
{ if (n1<n2) return(<);
 else if (n1==n2)return(=);
 else return(>);
}
```

## 第二章 线性表小结


本章学习了线性表的顺序存储结构——顺序表，链式存储结构，线性链表，循环链表，双向链表，以及在这两种存储结构下如何实现线性表的基本操作。这里再一次需要强调：本课程不仅要从概念和方法上了解每一种数据结构的逻辑结构和基本操作，更重要的是要学习如何在计算机上实现，即如何在计算机上存储线性表，如何在计算机上实现线性表的操作。我们已经看到，在不同的存储结构下，线性表的同一操作的算法是不同的，在顺序表存储结构下，线性表的插入删除操作，通过移动元素实现，在线性链表存储结构下，线性表的插入删除操作，通过修改指针实现。对于某一实际问题，如何选择合适的存储结构，如何在某种存储结构下实现对数据对象的操作，我们要通过数据结构课程的学习，很好地理解各种存储结构是如何存储和表达数据对象的有关信息的，各种存储结构下操作的特点。为实际问题的程序设计打下坚实的基础。




# 上机实验题

## 上机验证第一、二章的 算法例子和习题

- `#define NULL 0`
- `struct node`
- `{int data;`
- `struct node *link;`
- `};typedef struct node NODE;`
- `NODE *create_link(n)`
- `{ NODE *head,*p,*q;`
- `int i;`
- `p=(NODE *)malloc(sizeof(NODE));`
- `head=p;`
- `for (i=1;i<=n;i++)`
- `{q=(NODE *)malloc(sizeof(NODE));`
- `q->data=i;q->link=NULL;`
- `p->link=q;p=q;`
- `}`
- `return(head);`
- `}`



- `int insert_link(NODE *head,int x,int i)`
- `{ NODE *p,*s;int j;`
- `p=head;j=0;`
- `while ((p!=NULL)&&(j<i-1))`
- `{p=p->link;j++;}`
- `if(p==NULL)return(0);`
- `s=(NODE *)malloc(sizeof(NODE));`
- `s->data=x;`
- `s->link=p->link;`
- `p->link=s;`
- `return(1);`
- `}`



- `main()`
- `{`
- `int j;`
- `NODE *t;`
- `t=create_link(10);`
- `for (j=1;j<=10;j++)`
- `{t=t->link;printf("%3d",t->data);`
- `}`
- `insert_link(t,99,6);`
- `for (j=1;j<=11;j++)`
- `{t=t->link;printf("%3d",t->data);`
- `}`
- `}`