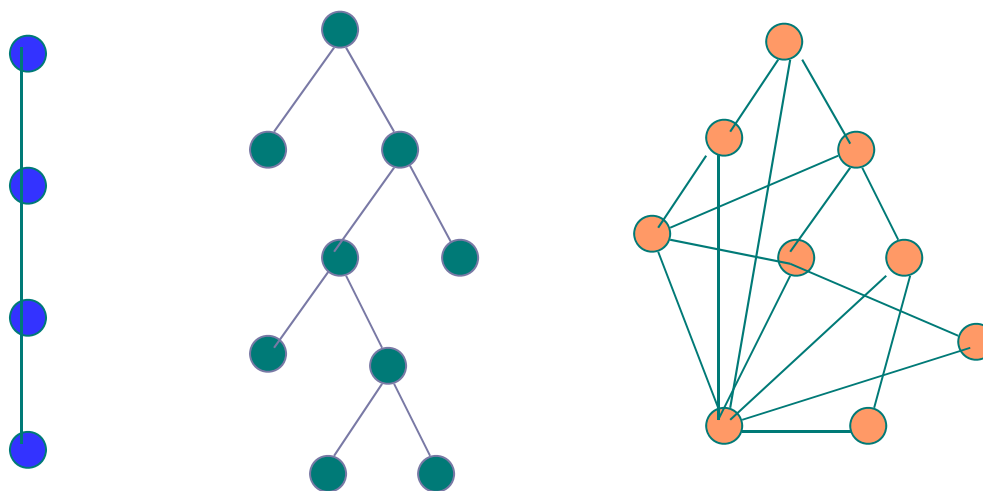


数据结构

华中科技大学计算机学院



第二章 线性表

2.1 线性表的定义

2.1.1 线性表的逻辑结构

1. 线性表:

- 由 n ($n \geq 0$) 个数据元素 (a_1, a_2, \dots, a_n) 构成的有限序列。
- 记作: $L = (a_1, a_2, \dots, a_n)$
 - a_1 —— 首元素
 - a_n —— 尾元素

2. 表的长度(表长)——线性表中数据元素的数目。

3. 空表——不含数据元素的线性表。

线性表举例：

例1. 字母表 $L1=(A, B, C, \dots, Z)$

表长26


例2. 姓名表 $L2=(李明, 陈小平, 王林, 周爱玲)$

表长4

例3. 图书登记表

序号	书 名	作 者	单价(元)	数量(册)
1	程序设计语言	李 明	10. 50	500
2	数据结构	陈小平	9. 80	450
...
n	DOS使用手册	周爱玲	20. 50	945

表长n



线性表的特征:

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

1. a_{i-1} 在 a_i 之前, 称 a_{i-1} 是 a_i 的 **直接前驱**
($1 < i \leq n$)
2. a_{i+1} 在 a_i 之后, 称 a_{i+1} 是 a_i 的 **直接后继**
($1 \leq i < n$)
3. a_1 没有前驱
4. a_n 没有后继
5. a_i ($1 < i < n$) 有且仅有一个直接前驱和一个直接后继

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

基本操作:

1. InitList (&L) //构造空表L。
2. ListLength (L) //求表L的长度
3. GetElem(L, i, &e) //取元素 a_i , 由e返回 a_i
4. PriorElem(L, ce, &pre_e) //求ce的前驱, 由pre_e返回
5. InsertElem(&L, i, e) //在元素 a_i 之前插入新元素e
6. DeleteElem(&L, i) //删除第i个元素
7. EmptyList (L) //判断L是否为空表

.....

}ADT List

说 明

1. 删除表L中第i个数据元素 ($1 \leq i \leq n$), 记作:

$\text{DeleteElem}(\&L, i)$

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

指定序号i, 删除 a_i ↑

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

2. 指定元素值x, 删除表L中的值为x的元素, 记作:

$\text{DeleteElem}(\&L, x);$

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

若 $a_i = x$, 删除 a_i ↑

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

3. 在元素 a_i 之前插入新元素 e ($1 \leq i \leq n+1$), 记作:

$\text{InsertElem}(\&L, i, e)$

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

插入 e \uparrow


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

4. 查找——确定元素值(或数据项的值)为 e 的元素。

给定: $L = (a_1, a_2, \dots, a_i, \dots, a_n)$ 和元素 e

若有一个 $a_i = e$, 则称“查找成功”, $i = 1, 2, \dots, n$

否则, 称“查找失败”。



5. 排序——按元素值或某个数据项值的递增（或递减）次序重新排列表中各元素的位置。

例. 排序前：L=(90, 60, 80, 10, 20, 30)

排序后：L=(10, 20, 30, 60, 80, 90)

L变为有序表

6. 将表La和表Lb合并为表Lc

例. 设有序表：

La=(2, 14, 20, 45, 80)

Lb=(8, 10, 19, 20, 22, 85, 90)

合并为表

Lc=(2, 8, 10, 14, 19, 20, 20, 22, 45, 80, 85, 90)

7. 将表La复制为表Lb

La= (2, 14, 20, 45, 80)

Lb= (2, 14, 20, 45, 80)

- 可利用现有操作组成更复杂的操作：

例：将线性表Lb中的，且不在线性表La数据元素合并到线性La中。

```
void union(List &La, List &Lb)
{
    La_len=ListLength(La); //求线性表La的长度
    Lb_len=ListLength(Lb); //求线性表Lb的长度
    for (i=1;i<= Lb_len;i++)
    {
        GetElem(Lb, i, e); //取Lb的的第i个数据元素赋给e
                           //即依次取出Lb中的所有元素
        if (!LocateElem(La,e,equal)) //判断e在La中是否存在
            ListInsert(La, ++La_len, e); //不存在则插入
    }
}
```

2.2 线性表的顺序表示（顺序存储结构）

2.2.1. 顺序分配——将线性表中的数据元素依次存放到计算机存储器中一组地址连续的存储单元中，这种分配方式称为顺序分配或顺序映像。由此得到的存储结构称为顺序存储结构或向量（一维数组）。

例. 线性表：a=(30, 40, 10, 55, 24, 80, 66)

...	30	40	10	55	24	80	66	...	内存状态
	a_1	a_2	a_3	a_4	a_5	a_6	a_7		

C语言中静态一维数组的定义：

int a[11]; //两种存储方式

30	40	10	55	24	80	66				
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]
	30	40	10	55	24	80	66			

(a_1, a_2, \dots, a_n) 顺序存储结构的一般形式

序号	存储状态	存储地址
1	a_1	b
2	a_2	$b+p$
	\dots	
i	a_i	$b+(i-1)*p$
	\dots	
n	a_n	$b+(n-1)*p$
	//	
	//	自由区
maxleng	//	$b+(\text{maxleng}-1)*p$

其中： b ----表的首地址/基地址/元素 a_1 的地址。

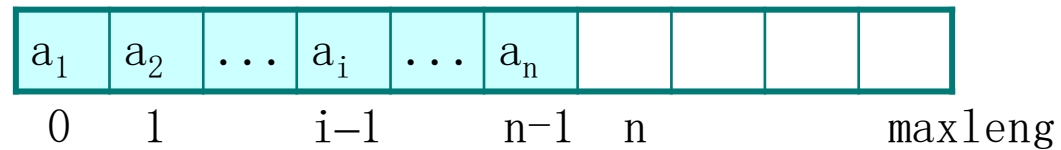
p ----1个数据元素所占存储单元的数目。

maxleng ----最大长度, 为某个常数。

2.2.2 线性表顺序结构在C语言中的定义

例1. 分别定义元素所占空间、表长、尾元素的位置

```
#define maxleng 100
{ ElemType la[maxleng+1]; //下标:0,1,...,maxleng
  int length;              //当前长度
  int last;                //an的位置
}
```



$\text{last} = \text{length} - 1 = n - 1$

或:

A horizontal array of 10 cells. The first six cells are light blue and contain a_1 , a_2 , \dots , a_i , \dots , a_n respectively. The last four cells are white. Below the array, indices are marked: 0 under the first white cell, 1 under the cell containing a_1 , 2 under the cell containing a_2 , i under the cell containing a_i , n under the cell containing a_n , and maxleng under the last cell.

$\text{last} = \text{length} = n$ 结论: last 和 length 二取一

线性表顺序结构在C语言中的定义（静态分配）

例2. 元素所占空间和表长合并为C语言的一个结构类型：

```
#define maxleng 100
typedef struct
{ ElemType elem[maxleng]; //下标:0,1,...,maxleng-1
  int length;              //表长
} SqList;
SqList La;
.....
```

其中：typedef---别名定义，SqList---结构类型名

La---结构类型变量名

La.length---表长

La.elem[0]---a1

La.elem[La.length-1]---an

线性表顺序结构在C语言中的定义（动态分配）

```
#define LIST_INIT_SIZE 100
#define LISTINCREMENT 10
typedef struct
{ ElemType *elem; //存储空间基地址
  int length;      //表长
  int listsize;    //当前分配的存储容量
                  //（以sizeof(ElemType)为单位）
} SqList;
SqList Lb;
```

其中：typedef---别名定义，SqList---结构类型名

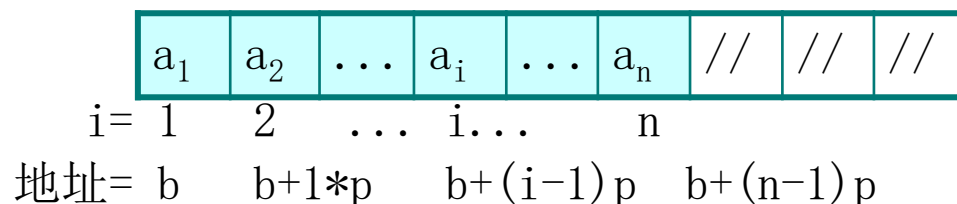
Lb---结构类型变量名

Lb.length---表长

Lb.elem[0]--- a_1

Lb.elem[Lb.length-1]--- a_n

2.2.3 顺序存储结构的寻址公式



假设：线性表的首地址为 b ，每个数据元素占 p 个存储单元，则表中任意元素 a_i ($1 \leq i \leq n$) 的存储地址是：

$$\begin{aligned} \text{LOC}(i) &= \text{LOC}(1) + (i-1)*p \\ &= b + (i-1)*p \quad (1 \leq i \leq n) \end{aligned}$$

例：假设 $b=1024$, $p=4$, $i=35$

$$\begin{aligned} \text{LOC}(i) &= b + (i-1)*p \\ &= 1024 + (35-1)*4 \\ &= 1024 + 34*4 \\ &= 1160 \end{aligned}$$

2.2.4 插入算法实现举例

设 $L.\text{elem}[0..\text{maxleng}-1]$ 中有 length 个元素，在
 $L.\text{elem}[i-1]$ 之前插入新元素 e ， $1 \leq i \leq \text{length}$
例. $i=3, e=6, \text{length}=6$

插入6之前：

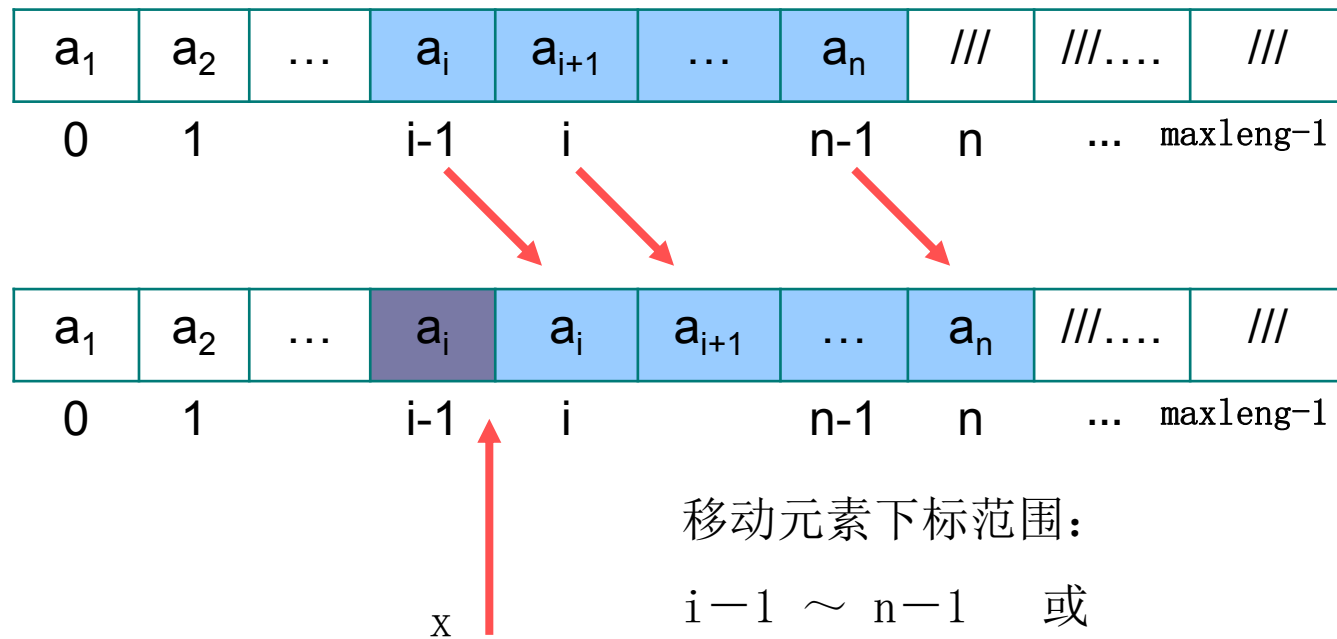
→ → → →								
2	5	8	20	30	35	//	//	//
0	1	2	3	4	5	6	7	8

35, 30, 20, 8 依次后移一个位置

插入6之后：

2	5	6	8	20	30	35	//	//
0	1	2	3	4	5	6	7	8

在线性表 $L=(a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n)$ 中的第 i 个元素前插入元素 x



移动元素下标范围:

$i-1 \sim n-1$ 或

$i-1 \sim L.\text{length}-1$

算法1：（用指针指向被操作的线性表, 静态分配）：

//设 L.elem[0..maxleng-1]中有length个元素，在
L.elem[i-1]之前插入新元素e，($1 \leq i \leq \text{length} + 1$)

```
int Insert1(SqList *L, int i, ElemType e)
{ if (i < 1 || i > L->length + 1) return ERROR;    //i值不合法
  if (L->length >= maxleng) return OVERFLOW;      //溢出
  for (j = L->length - 1; j >= i - 1; j--)
      L->elem[j + 1] = L->elem[j];    //向后移动元素
  L->elem[i - 1] = e;                  //插入新元素
  L->length++;                          //长度变量增1
  return OK ;                          //插入成功
}
```

算法2：（用引用参数表示被操作的线性表）：

//设 L.elem[0..maxleng-1]中有length个元素，在
L.elem[i-1]之前插入新元素e，($1 \leq i \leq \text{length} + 1$)

```
Status Insert2(SqList &L, int i, ElemType e)
{ if (i < 1 || i > L.length + 1) return ERROR;    //i值不合法
  if (L.length >= maxleng) return OVERFLOW;    //溢出
  for (j = L.length - 1; j >= i - 1; j--)
      L.elem[j + 1] = L.elem[j];                //向后移动元素
  L.elem[i - 1] = e;                             //插入新元素
  L.length++;                                    //长度变量增1
  return OK;                                     //插入成功
}
```

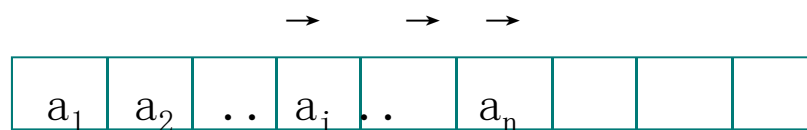


算法3: (动态分配线性表空间, 用引用参数表示被操作的线性表)

```
int Insert3(SqList &L, int i, ElemType e)
{
    int j;
    if (i < 1 || i > L.length + 1)    // i的合法取值为1至n+1
        return ERROR;
    if (L.length >= L.listsize)    /* 溢出时扩充 */
    {
        ElemType *newbase;
        newbase = (ElemType *) realloc(L.elem,
            (L.listsize + LISTINCREMENT) * sizeof(ElemType));
        if (newbase == NULL) return OVERFLOW;    // 扩充失败
        L.elem = newbase;
        L.listsize += LISTINCREMENT;
    }
}
```


2.2.5 插入操作移动元素次数的分析

在 $(a_1, a_2, \dots, a_i, \dots, a_n)$ 中 a_i 之前插入新元素 e
 ($1 \leq i \leq n+1$)



当插入点为:	1	2	\dots	i	\dots	n	$n+1$
需移动元素个数:	n	$n-1$	\dots	$n-i+1$	\dots	1	0

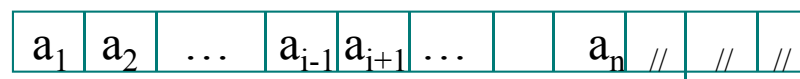
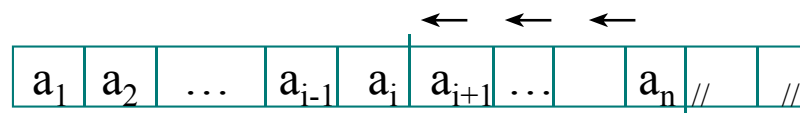
假定 p_i 是在各位置插入元素的概率, 且

$$p_1 = p_2 = \dots = p_n = p_{n+1} = 1/(n+1),$$

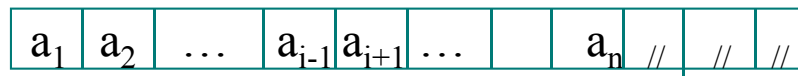
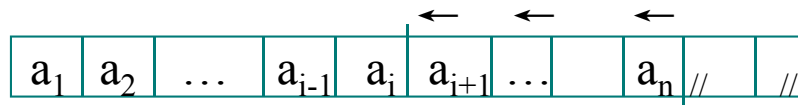
则插入一个元素时移动元素的平均值是:

$$E_{is} = \sum_{i=1}^{n+1} p_i (n-i+1) = 1/(n+1) * (n + (n-1) + \dots + 1 + 0) = n/2$$

2.2.6 删除操作及移动元素次数的分析



```
int delete(SqList *L, int i)
{if (i<1 || i>L->length)
    { printf("not exist");
      return ERROR;}
  else { for(j=i; j<=L->length-1; j++)
          L->elem[j-1]=L->elem[j];
        L->length--; return OK;}
}
```



$$\begin{array}{ccccccccccc} \text{当 } i= & 1 & 2 & 3 & \dots & i & \dots & n \\ \text{移动元素个数} = & n-1 & n-2 & & \dots & n-i & \dots & 0 \end{array}$$

假定 q_i 是在各位置删除元素的概率，且：

$$q_1 = q_2 = \dots = q_n = 1/n$$

则删除一个元素时移动元素的平均值是：

$$E_{d1} = \sum_{i=1}^n q_i (n-i) = 1/n * ((n-1) + \dots + 1 + 0) = (n-1)/2$$

2.2.7 顺序存储结构的评价

1. 优点:

- (1) 是一种随机存取结构, 存取任何元素的时间是一个常数, 速度快;
- (2) 结构简单, 逻辑上相邻的元素在物理上也是相邻的;
- (3) 不使用指针, 节省存储空间。

2. 缺点:

- (1) 插入和删除元素要移动大量元素, 消耗大量时间;
- (2) 需要一个连续的存储空间;
- (3) 插入元素可能发生“溢出”;
- (4) 自由区中的存储空间不能被其它数据占用(共享)。

内存:

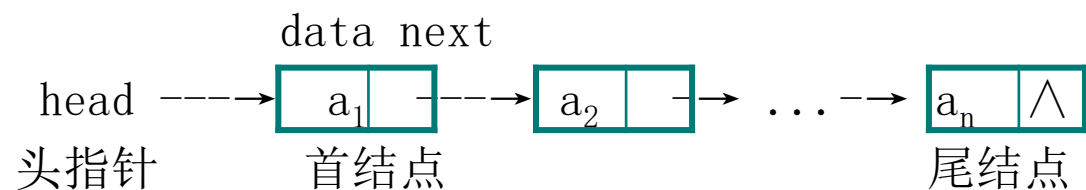
2k	占用	5k	占用	3k
----	----	----	----	----

2.3 线性表的链式存储结构

2.3.1 单链表

1. 单链表的一般形式:

(1) 不带表头结点的单链表:

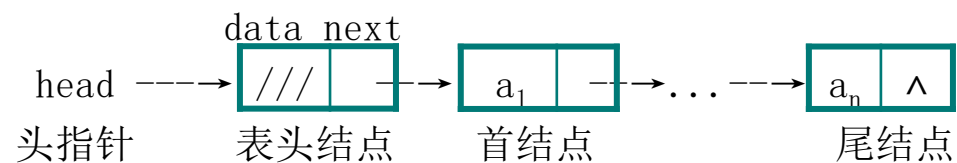


其中: data称为数据域, next称为指针域/链域

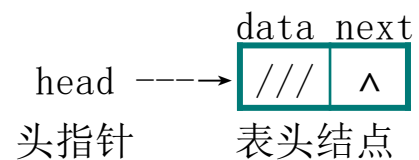
当 head==NULL, 为空表; 否则为非空表

(2) 带表头结点的单链表:

a. 非空表:



b. 空表:



其中: head指向表头结点,

head->data不放元素,

head->next指向首结点a₁,

当head->next==NULL, 为空表; 否则为非空表。

2. 单链表的结点结构

例1 C语言的“结构”类型：

```
struct node
{ ElemType data;      //data为抽象元素类型
  struct node *next;  //next为指针类型
};
```

指向结点的指针变量head, p, q说明：

```
struct node *head, *p, *q;
```

例2 用typedef定义指针类型：

```
typedef struct node
{ ElemType data;      //data为抽象元素类型
  struct node *next;  //next为指针类型
} node, *Linklist;
```

指针变量head, p, q的说明：

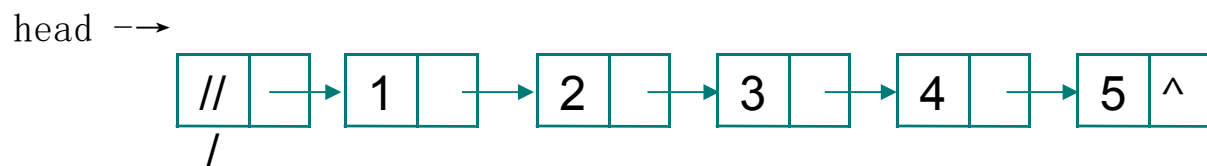
```
Linklist head, p, q;
```

3. 单链表操作和算法举例：

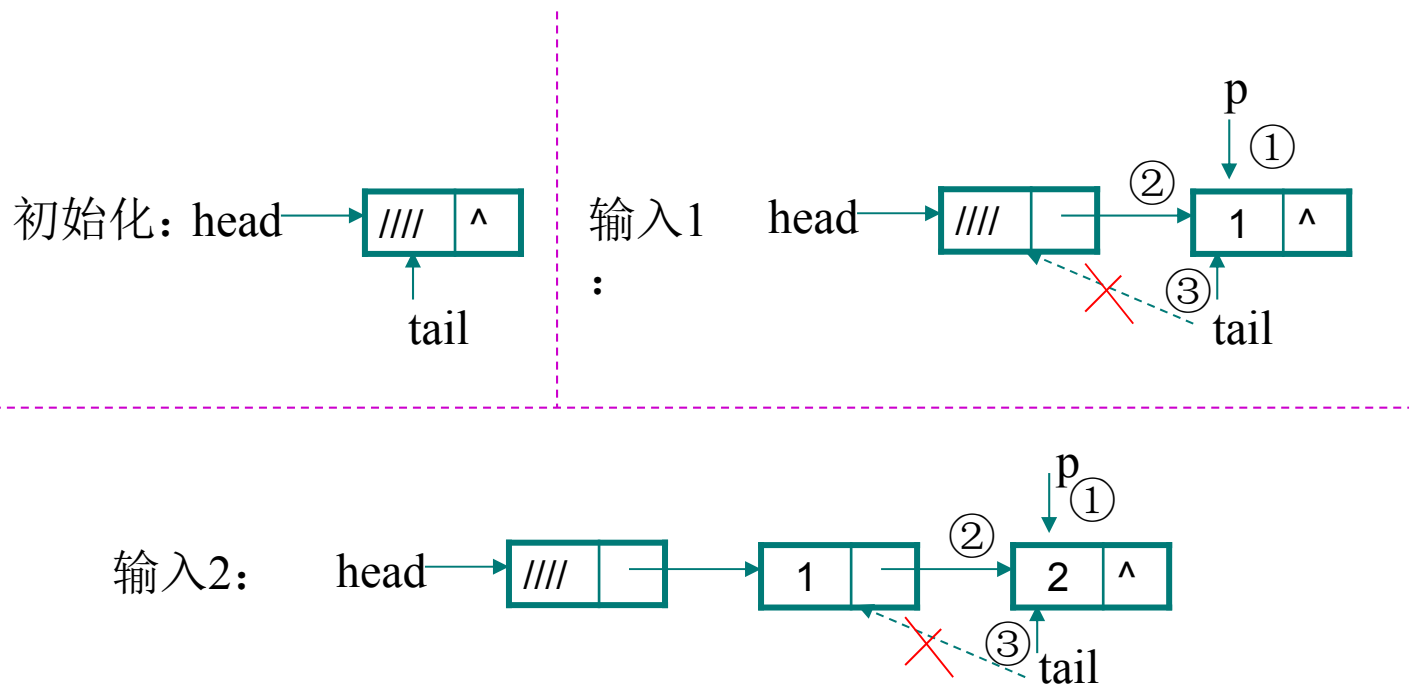
(1) 生成单链表

例1 输入一系列整数，以0为结束标志，生成“先进先出”单链表。

若输入：1, 2, 3, 4, 5则生成：



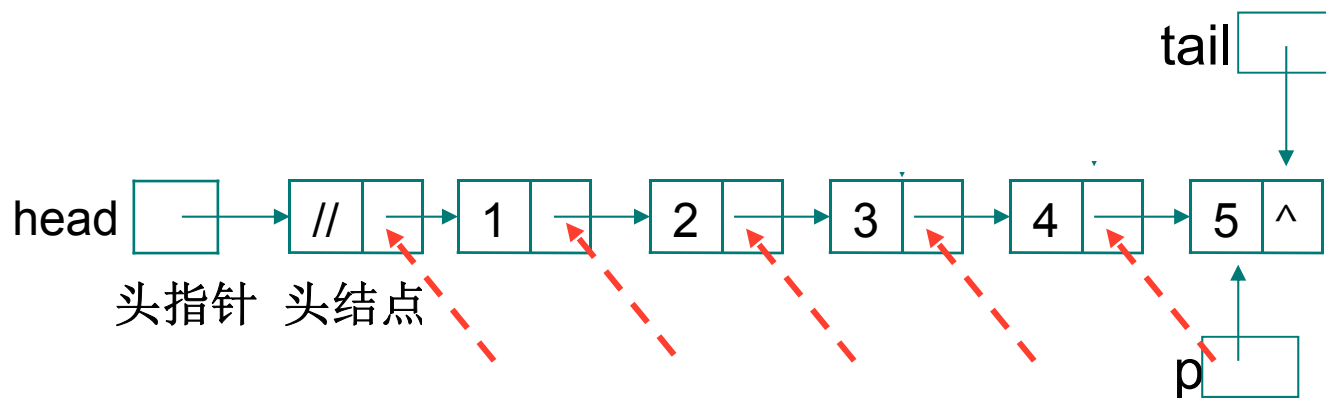
```
#define LENG sizeof(struct node) //结点所占的单元数
struct node                      //定义结点类型
{ int data;                      //data为整型
  struct node *next;            //next为指针类型
};
```



每次输入新元素后：

- ① 生成新结点； $p = \text{malloc}(\text{结点大小})$ ； $p \rightarrow \text{data} = e$ ； $p \rightarrow \text{next} = \text{NULL}$ ；
- ② 添加到表尾； $\text{tail} \rightarrow \text{next} = p$ ；
- ③ 设置新表尾。 $\text{tail} = p$ ；

先进先出表的产生过程(1,2,3,4,5,0):



```
head=malloc(sizeof(struct
```

```
tail=head
```

```
p=malloc(sizeof(struct node))
```

```
tail->next=p
```

```
tail=p
```

```
p=malloc(sizeof(struct node))
```

```
tail->next=p
```

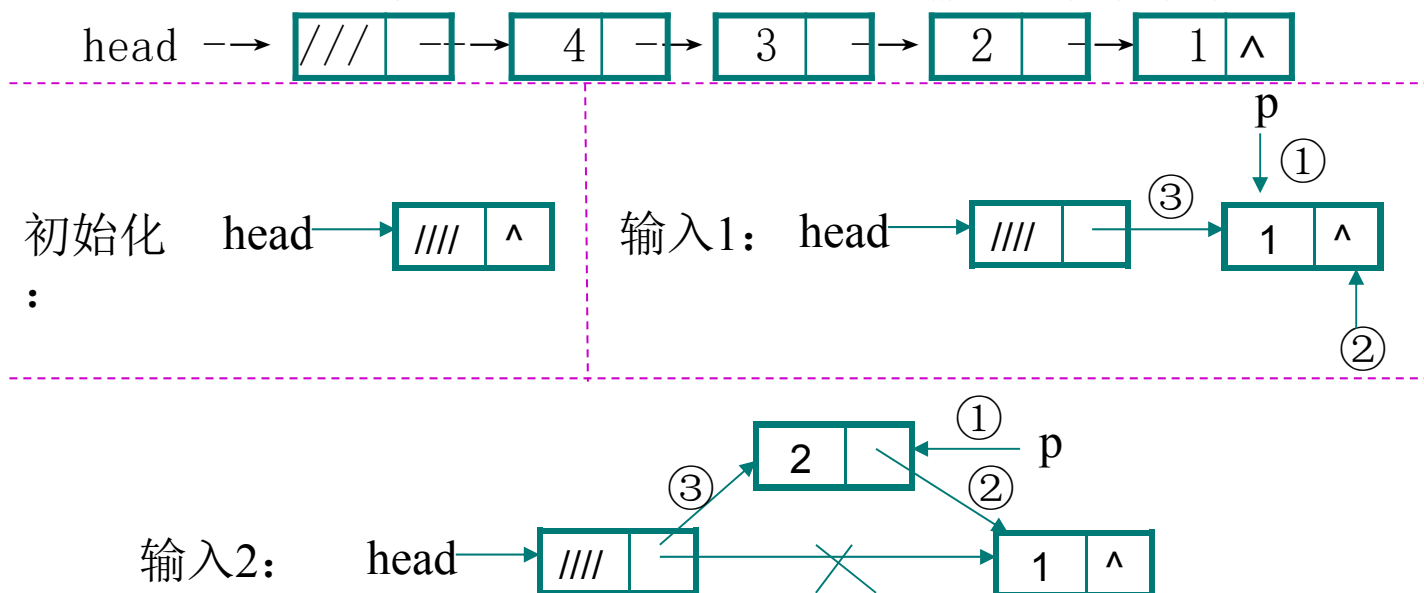
```
tail=p
```

```
tail->next=NULL;
```

算法：生成“先进先出”单链表（链式队列）

```
struct node *creat1( )
{ struct node *head,*tail,*p;           //变量说明
  int e;
  head=(struct node *)malloc(LENG); //生成表头结点
  tail=head;                          //尾指针指向表头
  scanf("%d",&e);                      //输入第一个数
  while (e!=0)                         //不为0
  { p=(struct node *)malloc(LENG); //生成新结点
    p->data=e;                        //装入输入的元素e
    tail->next=p;                     //新结点链接到表尾
    tail=p;                          //尾指针指向新结点
    scanf("%d",&e);                  //再输入一个数
  }
  tail->next=NULL;                    //尾结点的next置为空指针
  return head;                        //返回头指针
}
```


例2 生成“后进先出”单链表(链式栈)。输入:1, 2, 3, 4, 0生成:



每次输入新元素后:

- ① 生成新结点; $p = \text{malloc}(\text{结点大小})$; $p \rightarrow \text{data} = e$;
- ② 新结点指针指向原首结点; $p \rightarrow \text{next} = \text{head} \rightarrow \text{next}$;
- ③ 新结点作为首元素: $\text{head} \rightarrow \text{next} = p$;

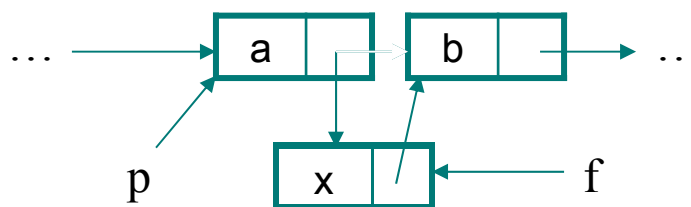
例2 生成“后进先出”单链表(链式栈)。

算法：

```
struct node *creat2( )                //生成“后进先出”单链表
{ struct node *head,*p;
  head=(struct node *)malloc(LENG);   //生成表头结点
  head->next=NULL;                    //置为空表
  scanf("%d",&e);                     //输入第一个数
  while (e!=0)                        //不为0
  { p=(struct node *)malloc(LENG);    //生成新结点
    p->data=e;                         //输入数送新结点的data
    p->next=head->next;                //新结点指针指向原首结点
    head->next=p;                      //表头结点的指针指向新结点
    scanf("%d",&e);                   //再输入一个数
  }
  return head;                        //返回头指针
}
```

(2) 插入一个结点

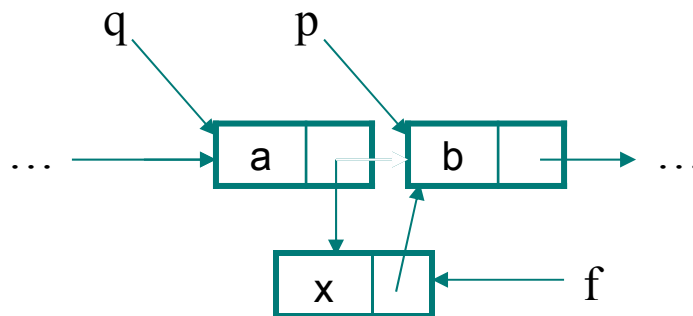
例1：在已知p指针指向的结点后插入一个元素x



执行：

```
f=(struct node *)malloc(LENG);    //生成
f->data=x;                        //装入元素x
f->next=p->next;                   //新结点指向p的后继
p->next=f;                        //新结点成为p的后继
```

例2：在已知p指针指向的结点前插入一个元素x



执行：

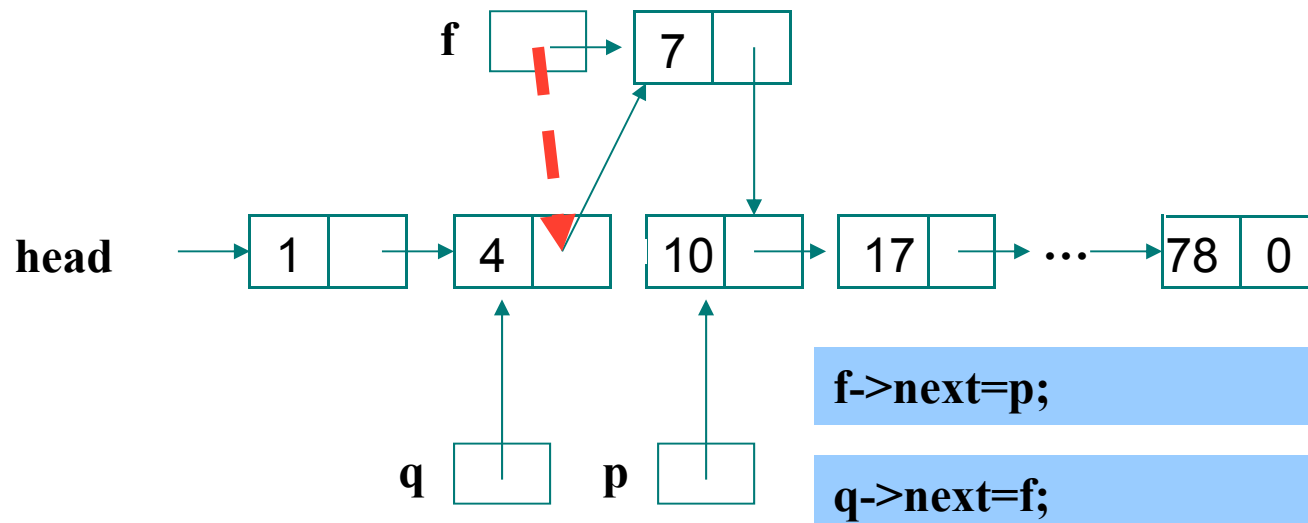
```
f=(struct node *)malloc(LENG); //生成
f->data=x;                      //装入元素x
f->next=p;                       //新结点成为p的前趋
q->next=f;                       //新结点成为p的
                                //前趋结点的后继
```

例3：输入一系列整数, 以0为结束标志, 生成递增有序单链表。（不包括0）

递增有序单链表的两种形式：

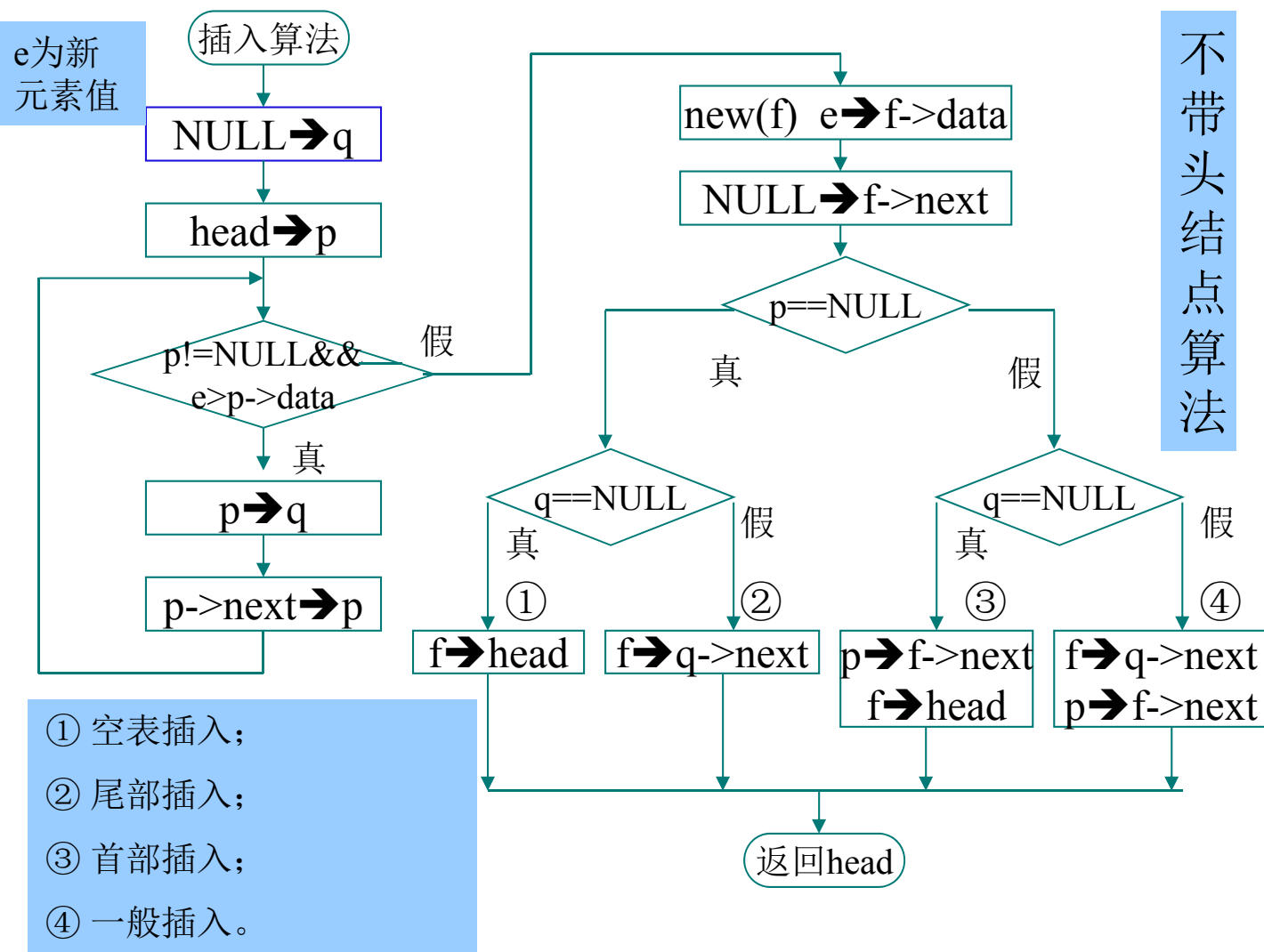


分析：假定有一个无表头结点的递增排序的链表，插入一个新结点，使其仍然递增。



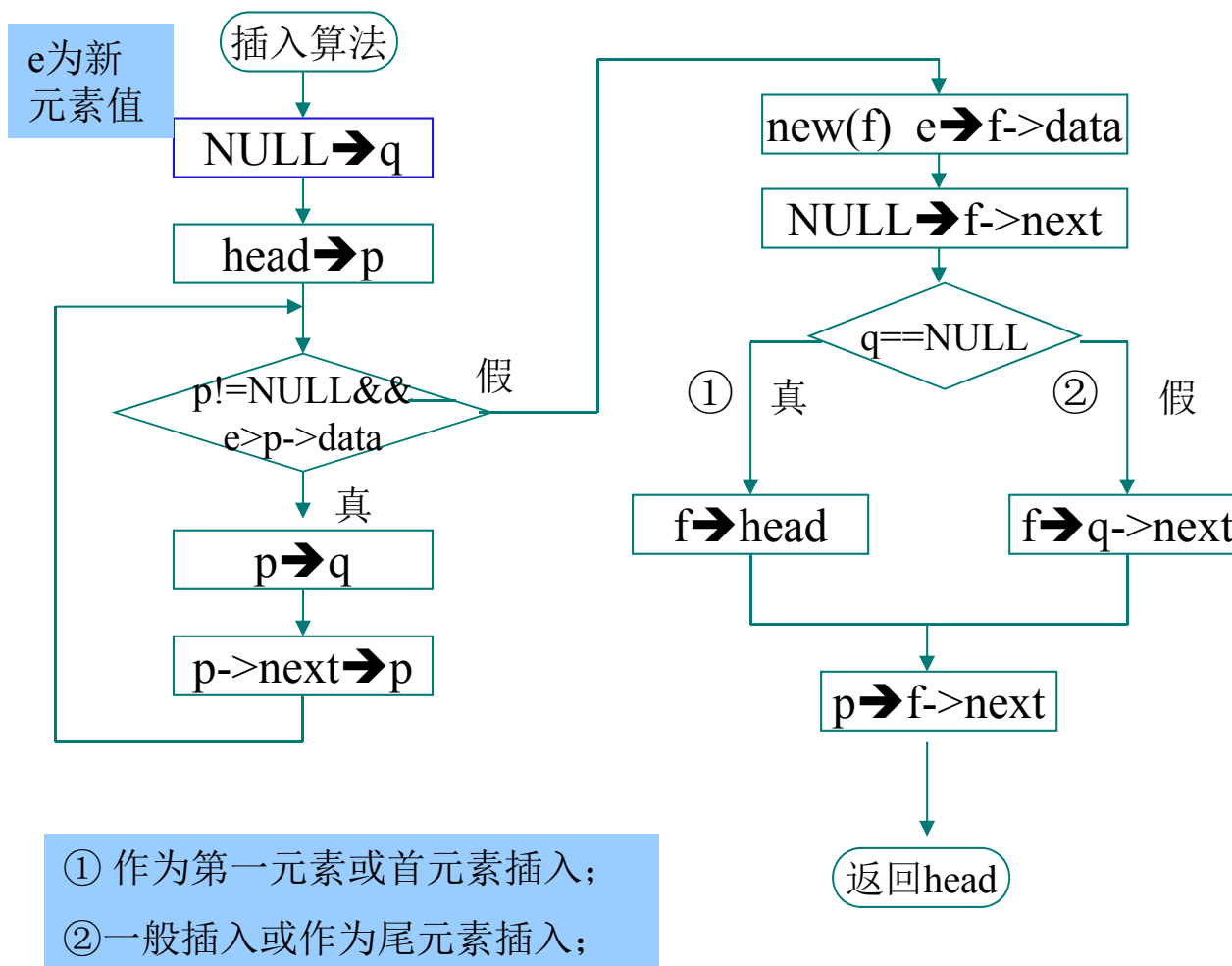
p, q 可能为NULL


- ① (p, q 同时空) 空表插入;
- ② (仅p为空) 尾部插入;
- ③ (仅q 为空) 首部插入;



算法1：生成不带头结点的递增有序单链表。（不包括0）

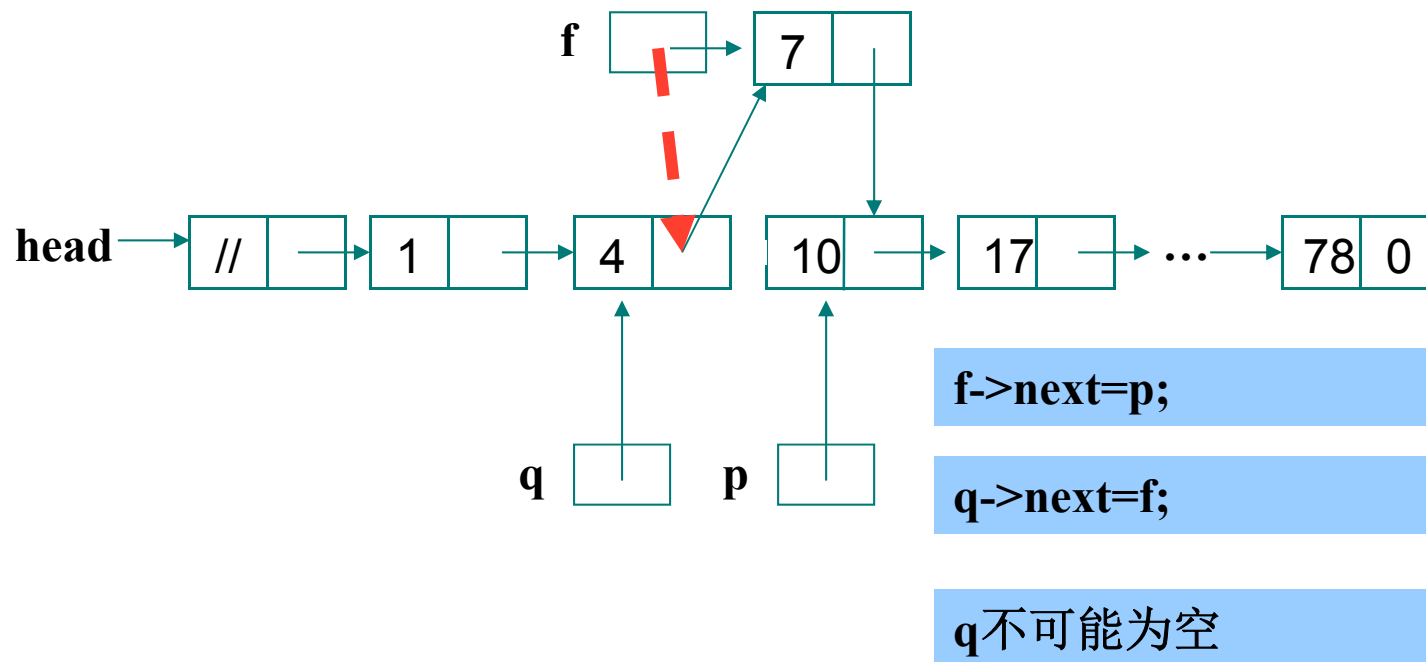
```
struct node * creat3_1(struct node *head, int e)
{ q=NULL;  p=head;                //q, p扫描, 查找插入位置
  while (p && e>p->data)            //未扫描完, 且e大于当前结点
  { q=p;  p=p->next; }             //q, p后移, 查下一个位置
  f=(struct node *)malloc(LENG);    //生成新结点
  f->data=e;                        //装入元素e
  if (p==NULL) {
      f->next=NULL;
      if (q==NULL)                //(1)对空表的插入
          head=f;
      else q->next=f;}             //(2)作为最后一个结点插入
  else if (q==NULL)                //(3)作为第一个结点插入
      {f->next=p;  head=f;}
  else
      {f->next=p;  q->next=f; }    //(4)一般情况插入新结点
  return head;
}
```

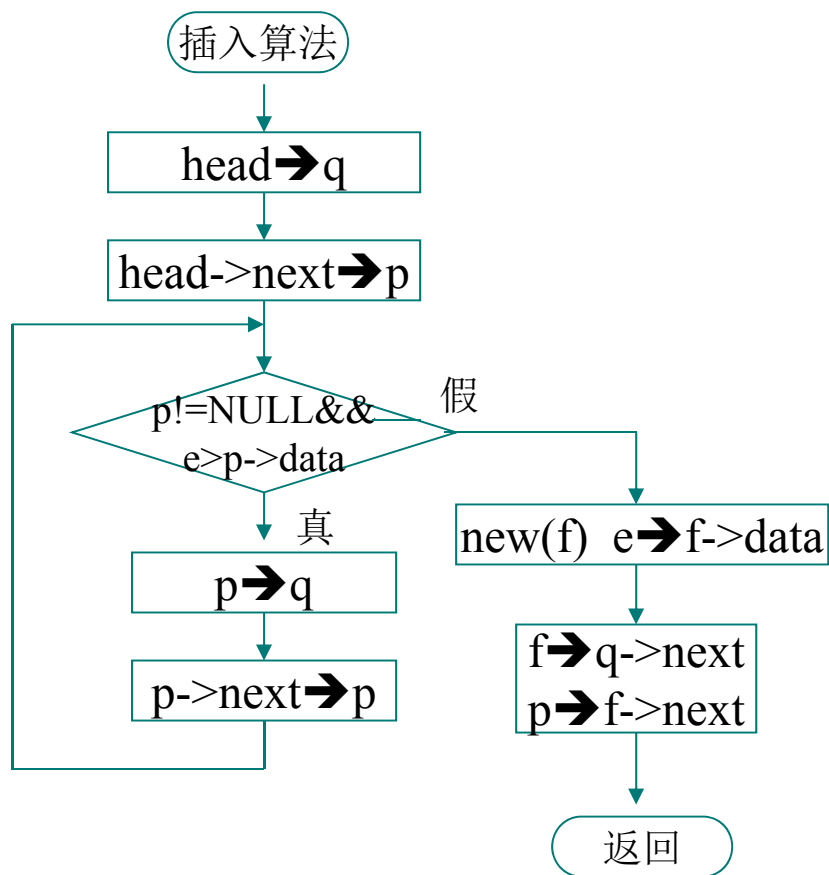


```
main()
{
    struct node *head;           //定义头指针
    head=NULL;                   //置为空表
    scanf("%d", &e);             //输入整数
    while (e!=0)                 //不为 0，未结束
    {
        head=creat3_1(head, e); //插入递增有序单链表
        scanf("%d", &e);        //输入整数
    }
}
```

分析：假定有一个有表头结点的递增排序的链表，插入一个新结点，使其仍然递增。




带头结点算法



算法2：生成带头结点的递增有序单链表。（不包括0）

```
void creat3_2(struct node *head, int e)
{ q=head;
  p=head->next;           //q, p扫描, 查找插入位置
  while (p && e>p->data) //未扫描完, 且e大于当前结点
  { q=p;
    p=p->next;           //q, p后移, 查下一个位置
  }
  f=(struct node *)malloc (LENG); //生成新结点
  f->data=e;                  //装入元素e
  f->next=p;   q->next=f;     //插入新结点
}
```

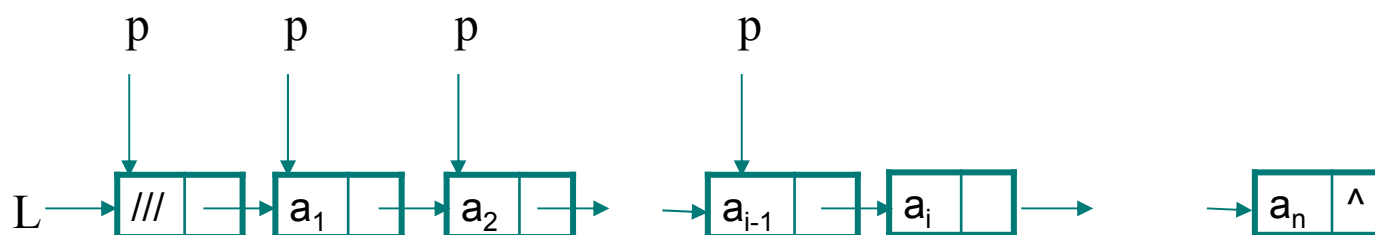


```
main()
{
    head=(struct node *)malloc(LENG);    //生成表头结点
    head->next=NULL;                      //置为空表
    scanf("%d",&e);                       //输入整数
    while (e!=0)                          //不为 0，未结束
    {
        creat3_2(head,e);    //插入递增有序单链表head
        scanf("%d",&e);      //输入整数
    }
}
```

算法3：在单链表的指定位置插入新元素

输入：头指针L、位置i、数据元素e

输出：成功返回OK，否则ERROR



执行： $p=L$

当 p 不为空

执行： $p=p \rightarrow \text{next}$ $i-1$ 次

定位到第 $i-1$ 个结点

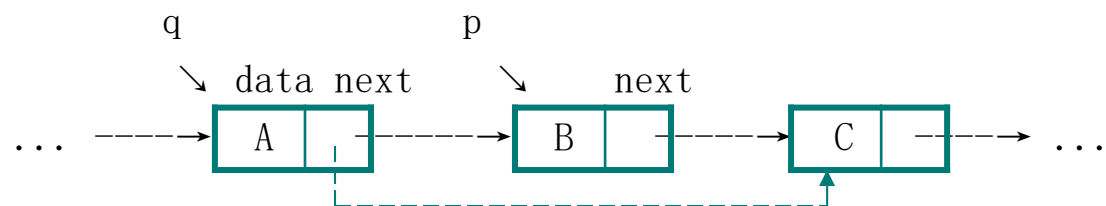
当 $i < 1$ 或 p 为空时插入点错

否则新结点加到 p 指向结点之后

算法3程序:

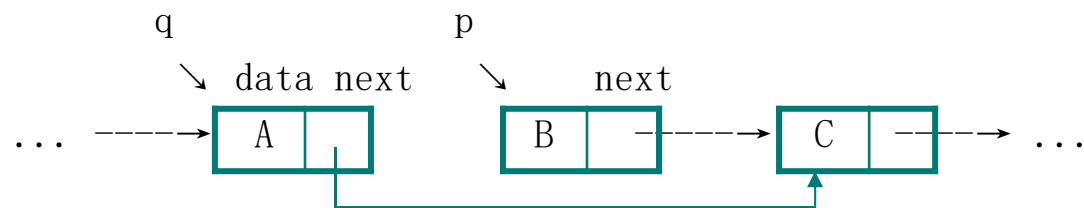
```
int insert( Linklist &L, int i, ElemType e)
{
    p=L;
    j=1;
    while (p && j<i)
        { p=p->next;      //p后移, 指向下一个位置
          j++; }
    if (i<1 || p==NULL)    //插入点错误
        return ERROR;
    f=(LinkList) malloc (LENG); //生成新结点
    f->data=e;                //装入元素e
    f->next=p->next;  p->next=f; //插入新结点
    return OK;
}
```


(3) 在单链表中删除一个结点

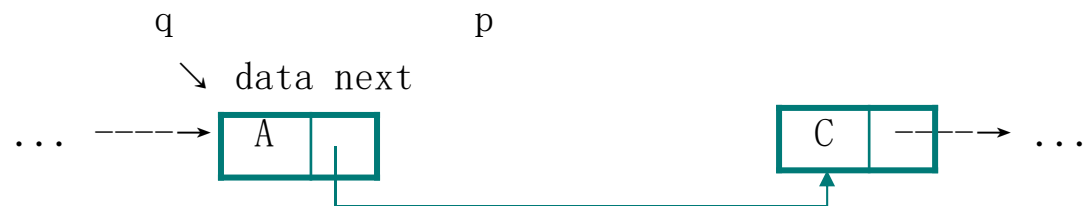


删除B

① 执行: $q \rightarrow \text{next} = p \rightarrow \text{next}$; //A的next域=C的地址 (B的next域)



② 执行: $\text{free}(p)$; //释放p所指向的结点空间



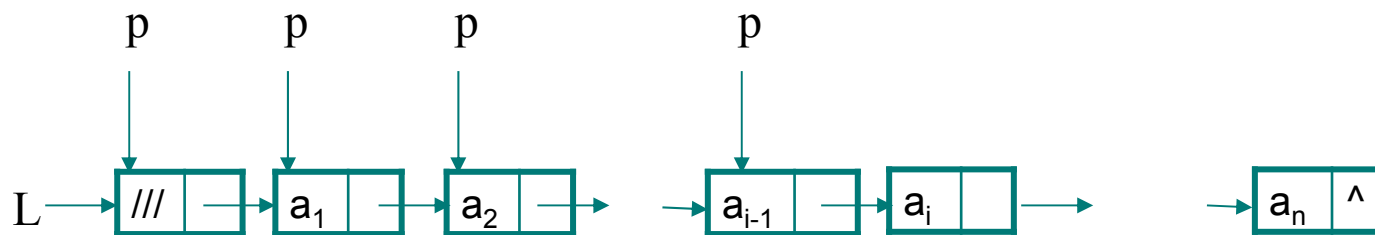
算法1：在带头结点的单链表中删除元素值为e的结点

```
int Deletel(Linklist head, ElemType e)
{ struct node *q,*p;
  q=head;  p=head->next;  //q,p扫描
  while (p && p->data!=e) //查找元素为e的结点
  { q=p;                //记住前一个结点
    p=p->next;           //查找下一个结点
  }
  if (p)                //有元素为e的结点
  { q->next=p->next;     //删除该结点
    free(p);            //释放结点所占的空间
    return YES;
  }
  else
    return NO;          //没有删除结点
}
```

算法2：在单链表中删除指定位置的元素

输入：头指针L、位置i

输出：成功返回OK，否则ERROR



执行： $p=L$


当 $p \rightarrow \text{next}$ 不为空

执行： $p=p \rightarrow \text{next}$ $i-1$ 次

定位到第 $i-1$ 个结点

当 $i < 1$ 或 $p \rightarrow \text{next}$ 为空时删除点错

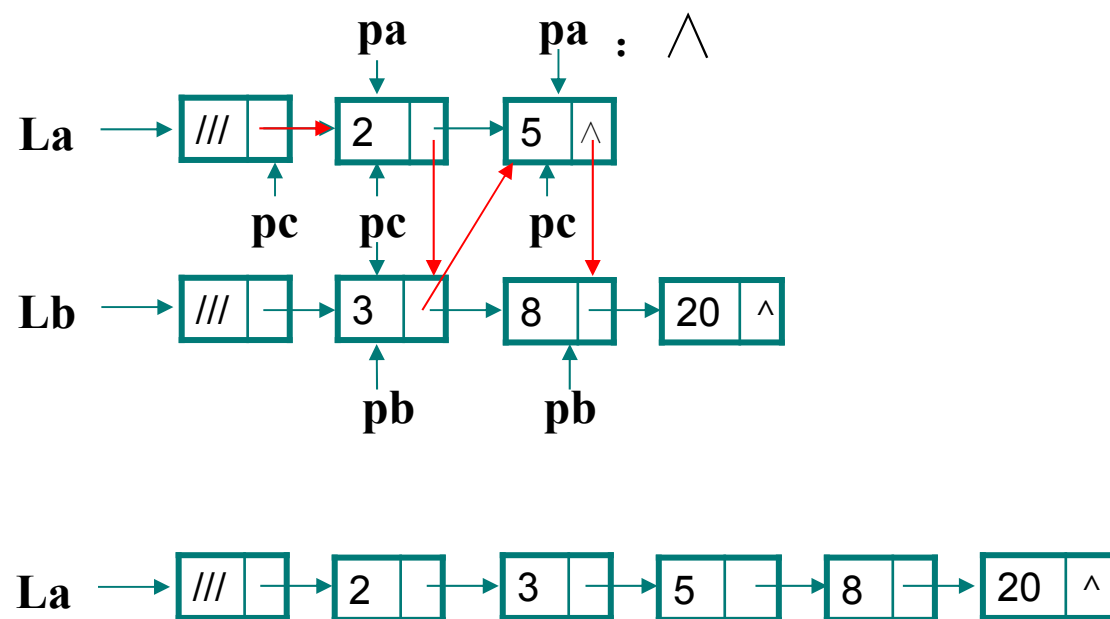
否则 p 指向结点的后继跳过原后继。



算法2程序:

```
int Delete2( Linklist &L, int i, ElemType &e)
{
    p=L;
    j=1;
    while (p->next && j<i)          //循环结束时p不可能为空
    {
        p=p->next;                  //p后移, 指向下一个位置
        j++;
    }
    if (i<1 || p->next==NULL) //删除点错误
        return ERROR;
    q=p->next;                      //q指向删除结点
    p->next=q->next;                //从链表中摘出
    e=q->data;                      //取走数据元素值
    free(q);                       //释放结点空间
    return OK;
}
```

(4) 将两个有序单链表La和Lb合并为有序单链表Lc:
(该算法利用原单链表的结点)






算法:

输入: 两单链表的头指针

输出: 合并后的单链表的头指针

```
struct node *merge(struct node *La, struct node *Lb)
{
    struct node *pa, *pb, *pc;
    pa=La->next;           //pa指向表La的首结点
    pb=Lb->next;           //pb指向表Lb的首结点
    pc=La;                 //使用表La的头结点, pc为尾指针
    free(Lb);              //释放表Lb的头结点
```

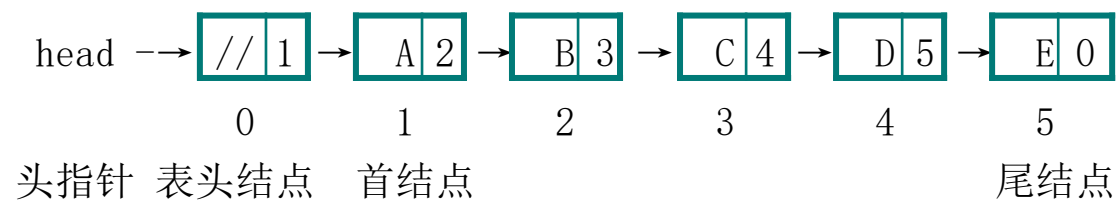


```
while (pa && pb)                //表La表Lb均有结点
    if (pa->data<=pb->data)      //取表La的一个结点
    { pc->next=pa;                //插在表Lc的尾结点之后
      pc=pa;                     //变为表Lc新的尾结点
      pa=pa->next;               //移向表La下一个结点
    }
    else                         //取表Lb的一个结点
    { pc->next=pb;                //插在表Lc的尾结点之后
      pc=pb;                     //变为表Lc新的尾结点
      pb=pb->next;               //移向表Lb下一个结点
    }

if (pa) pc->next=pa;             //插入表La的剩余段
else pc->next=pb;                //插入表Lb的剩余段
return La;
}
```

例

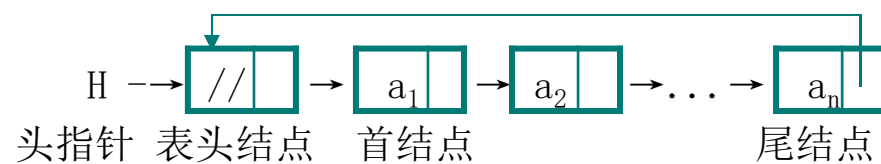
	data	next
0	//	1
1	A	2
2	B	3
3	C	4
4	D	5
5	E	0
6	//	//
7	//	//



2.3.2 循环链表

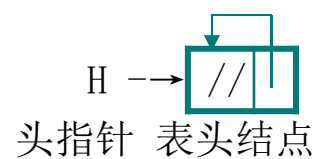
1. 一般形式

(1) 带表头结点的非空循环单链表



有: $H \rightarrow \text{next} \neq H$, $H \neq \text{NULL}$

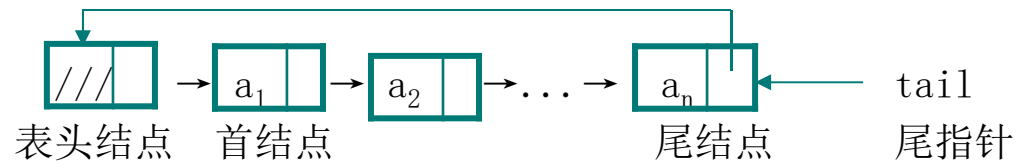
(2) 带表头结点的空循环单链表



有: $H \rightarrow \text{next} == H$, $H \neq \text{NULL}$

2. 只设尾指针的循环链表

(1) 非空表



有: tail指向表尾结点

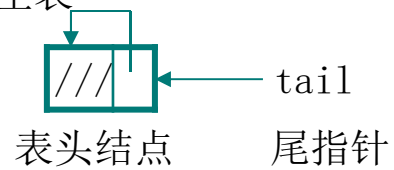
$\text{tail} \rightarrow \text{data} == a_n$

$\text{tail} \rightarrow \text{next}$ 指向表头结点

$\text{tail} \rightarrow \text{next} \rightarrow \text{next}$ 指向首结点

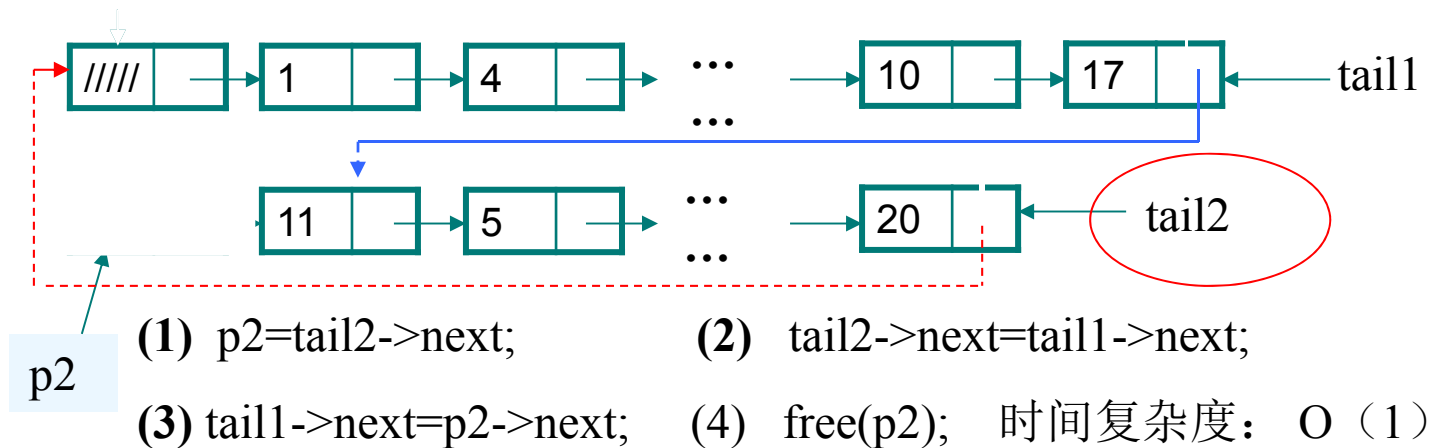
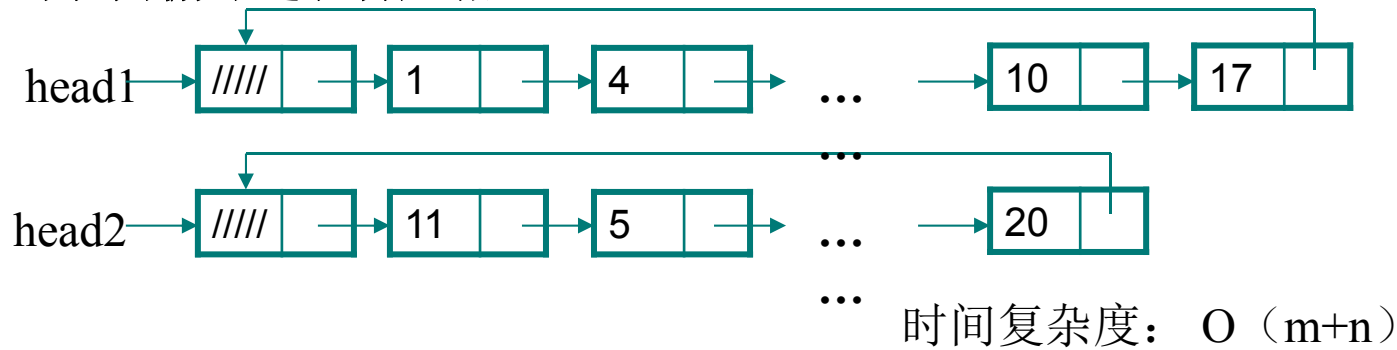
$\text{tail} \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{data} == a_1$

(2) 空表



有: $\text{tail} \rightarrow \text{next} == \text{tail}$

例：两循环链表首尾相连。



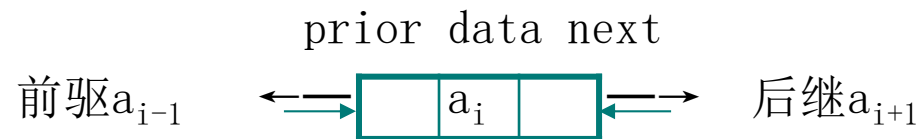
3. 循环链表算法举例

例. 求以head为头指针的循环单链表的长度,
并依次输出结点的值。 算法如下:

```
int length(struct node *head)
{ int leng=0;           //长度变量初值为0
  struct node *p;
  p=head->next;         //p指向首结点
  while (p!=head)       //p未移回到表头结点
  { printf("%d",p->data); //输出
    leng++;              //计数
    p=p->next; }         //p移向下一结点
  return leng;          //返回长度值
}
```

2.3.4 双向链表

1. 双向链表的结点结构:



结点类型定义

```
struct Dnode
```

```
{ ElemType data;           //data为抽象元素类型
  struct Dnode *prior,*next; //prior,next为指针类型
};
```

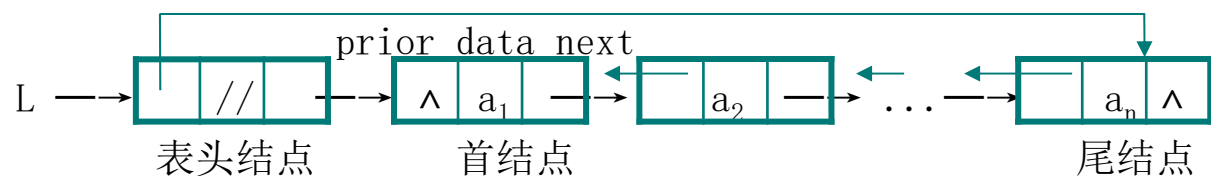
或者

```
typedef struct Dnode
```

```
{ ElemType data;           //data为抽象元素类型
  struct Dnode *prior,*next; //prior,next为指针类型
}*DLList                  //DLList为指针类型
```

2. 双向链表的一般形式

(1) 非空表



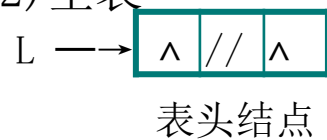
有：L为头指针, L指向表头结点, $L \rightarrow next$ 指向首结点

$L \rightarrow next \rightarrow data == a_1$

$L \rightarrow prior$ 指向尾结点, $L \rightarrow prior \rightarrow data == a_n$

$L \rightarrow next \rightarrow prior == L \rightarrow prior \rightarrow next == NULL$

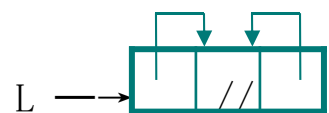
(2) 空表



有： $L \rightarrow next == L \rightarrow prior == NULL$

3. 双向循环链表

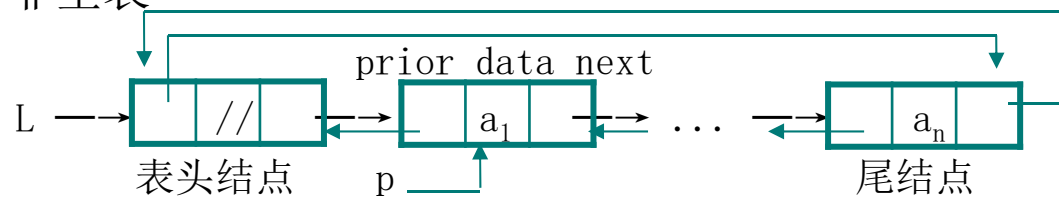
(1) 空表



表头结点

有: $L \rightarrow next == L \rightarrow prior == L$

(2) 非空表



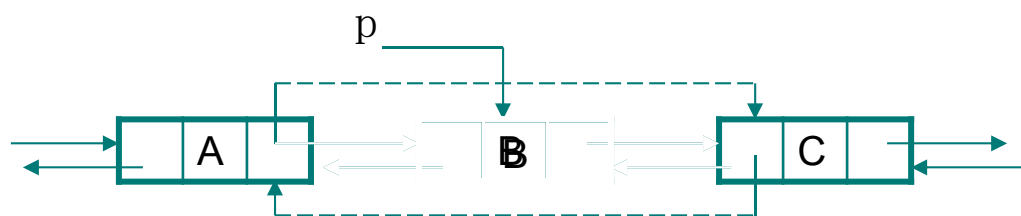
设p指向a1, 有:

$p \rightarrow next$ 指向 a_2 , $p \rightarrow next \rightarrow prior$ 指向 a_1 ,

所以, $p == p \rightarrow next \rightarrow prior$

同理: $p == p \rightarrow prior \rightarrow next$

(3) 已知指针p指向结点B，删除B



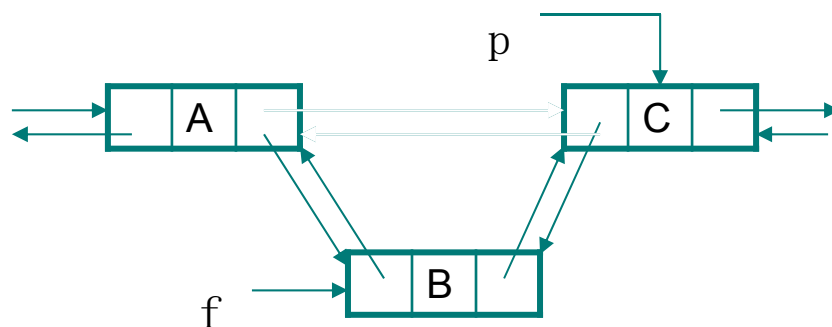
执行：

`p->prior->next=p->next;` //结点A的next指向结点C

`p->next->prior=p->prior;` //结点C的prior指向结点A

`free(p);` //释放结点B占有的空间

(4) 已知指针p指向结点C，在A、C之间插入结点B



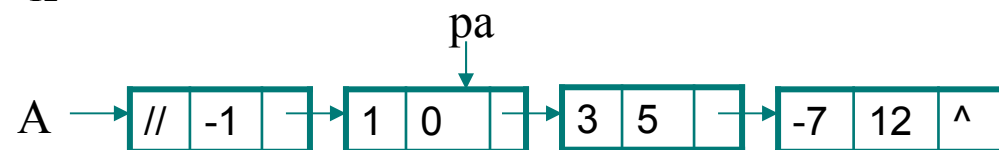
执行：

- ① $f \rightarrow \text{prior} = p \rightarrow \text{prior};$ //结点B的prior指向结点A
- ② $f \rightarrow \text{next} = p;$ //结点B的next指向结点C
- ③ $p \rightarrow \text{prior} \rightarrow \text{next} = f;$ //结点A的next指向结点B
- ④ $p \rightarrow \text{prior} = f;$ //结点C的prior指向结点B

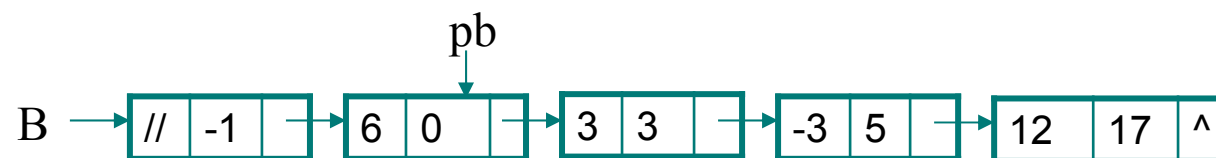
2.4 多项式的链表表示

coef	expn	next
------	------	------

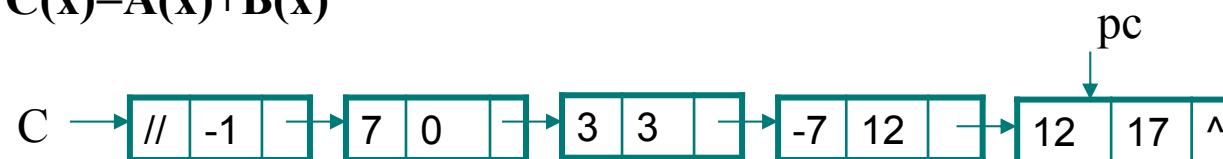
$$A_{12}(x)=1+3x^5-7x^{12}$$



$$B_{17}(x)=6+3x^3-3x^5+12x^{17}$$



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



$C(x)=A(x)+B(x)$ 的算法步骤:

1. pa、pb分别指向首元素结点,产生 $C(x)$ 的空链表, pc指向头结点;
2. pa不为空并且pb不为空,重复下列操作:
 - 2-1 pa->expn等于pb->expn
 - (a) pa->coef+pb->coef不等于零: 产生新结点, 添加到pc后, pc指向新结点。 pa、pb后移
 - (b) pa->coef+pb->coef等于零: pa、pb后移
 - 2-2 pa->expn小于pb->expn: 根据pa产生新结点, 添加到pc后, pc指向新结点pa后移
 - 2-3 pa->expn大于pb->expn: 根据pb产生新结点, 添加到pc后, pc指向新结点pb后移
3. pa为空, 取pb剩余结点产生新结点, pb为空, 取pa剩余结点产生新结点, 依次添加到pc的后面。