

## 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。

1

### 线性表举例:

例1. 字母表  $L1 = (A, B, C, \dots, Z)$ ,

表长26

例2. 姓名表  $L2 = (\text{李明}, \text{陈小平}, \text{王林}, \text{周爱玲})$

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

2

## 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

3

## 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:  $R_1 = \{\langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i = 2, \dots, n\}$

### 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. ListInsert(&L, i, e) //在元素 $a_i$ 之前插入新元素e
  6. ListDelete(&L, i, &e) //删除第i个元素
  7. LocateElem(L, e, compare()) //比较与定位对应元素
- ..... Scan(L, op, I, &S) //由原线性表产生一个等长线性表

} ADT List

4

## 说明



1. 删除L中第i个数据元素 ( $1 \leq i \leq n$ ), 记作: `ListDelete(&L, i, &e)`

指定序号i, 删除 $a_i$

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

2. 指定元素值x, 删除表L中的值为x的元素, 记作: `DeleteElem(&L, x)`

3. 在 $a_i$ 之前插入元素e ( $1 \leq i \leq n+1$ ),  $L = (a_1, a_2, \dots, a_{i-1}, a_i, \dots, a_n)$

插入e ↑

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

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

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

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

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

6

可利用现有基本操作实现更复杂的操作：

例：将线性表Lb中且不在线性表La的数据元素合并到La中。

**void union(List &La, List Lb)**

```
{
    La_len=ListLength (La); //求La的长度
    Lb_len=ListLength (Lb); //求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
    }
}
```

7

## 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。所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	...
	a <sub>1</sub>	a <sub>2</sub>	a <sub>3</sub>	a <sub>4</sub>	a <sub>5</sub>	a <sub>6</sub>	a <sub>7</sub>	

内存状态

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			

8

### $(a_1, a_2, \dots, a_n)$ 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	$a_2$	$b+l$
$\vdots$	$\vdots$	$\vdots$
$i$	$a_i$	$b+(i-1)*l$
$\vdots$	$\vdots$	$\vdots$
$n$	$a_n$	$b+(n-1)*l$
	//	自由区
	//	
maxleng	//	
		$b+(maxleng-1)*l$

其中： $b$ ——表的首地址/基地址/元素 $a_1$ 的地址。

$l$ ——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。

9

### 2.2.2 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] ---  $a_1$

La.elem[La.length-1] ---  $a_n$

10

□ 顺序表的存储结构是一维数组，如果插入若干元素后元素总个数~~超过~~数组限定的~~长度~~怎么办？

✓ 采用~~动态分配~~的一维数组

□ 在C语言中如何实现~~动态数组~~？

利用C中几个库函数（在<stdlib.h> 中）：

sizeof(x)：C中的~~单目操作符~~，计算变量x的长度（字节数）；

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；

realloc(\*p, newsize)：重新分配一片大小为newsize的连续空间，并把以\*p为首址的原空间数据都复制进去。

free(p)：释放指针p所指变量的存储空间。

11

### 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;    //存储空间或动态数组基地址
  int length;        //表长（表中有多少个元素）
  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] ---  $a_n$

12

### 2.2.3 顺序存储结构的寻址公式

$a_1$	$a_2$	...	$a_i$	...	$a_n$	//	//	//
-------	-------	-----	-------	-----	-------	----	----	----

$i = 0 \quad 1 \quad \dots i-1 \dots n-1$

地址 =  $b \quad b+1 \times l \quad b+(i-1)l \quad b+(n-1)l$

**假设：**线性表的首地址为 $b$ ，每个数据元素占 $l$ 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：

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

**例：**假设  $b=1024$ ,  $l=4$ ,  $i=35$

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

13

线性表的初始化操作（以动态数组表示的顺序结构）

**Status InitList\_Sq( SqList &L ) {**

*// 构造一个空的线性表L，利用动态顺序存储结构*

**L.elem = (ElemType \*)malloc(LIST\_INIT\_SIZE \* sizeof(ElemType));**

**if (! L.elem) exit(OVERFLOW);** *// 存储分配失败*

**L.length = 0;**

**L.listsize = LIST\_INIT\_SIZE;**

**return OK;** *///??*

**} // InitList\_Sq**

```
typedef struct
{ ElemType *elem;
  int length;
  int listsize;
} SqList;
```

**算法时间复杂度：  $O(1)$**  *//在不计入malloc时间复杂度时*

**算法空间复杂度：  $O(1)$**

14

#### 2.2.4 插入算法实现举例

设  $L.elem[0..maxleng-1]$  中有  $length$  个元素，在  
 $L.elem[i-1]$  之前插入新元素  $e$ ， $1 \leq i \leq length$ .

例.  $i=3, e=6, length=6$ ；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

15

### 线性表

线性表的逻辑结构：

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： $L = (a_1, a_2, \dots, a_n)$

线性表顺序存储结构（静态分配）

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

16



## 线性表顺序存储结构（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（表中有多少个元素）
  int listsize;      //当前分配的存储容量
} SqList;
```

SqList L;



$LOC(i) = LOC(1) + (i-1) * l = b + (i-1) * l \quad (1 \leq i \leq n)$

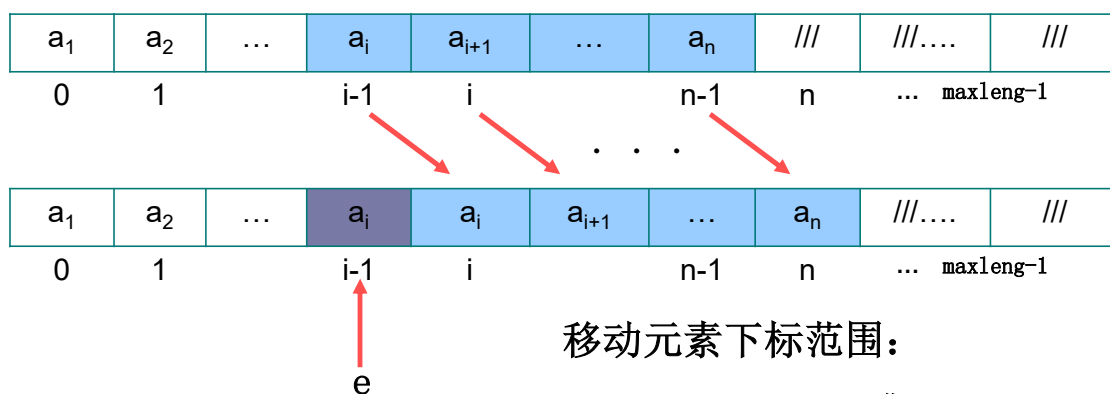
Status InitList\_Sq(SqList &L)

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

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

17

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  $e$



移动元素下标范围:

$i-1 \sim n-1$  或

$i-1 \sim L.length-1$

18

算法1：（用引用参数表示被操作的线性表, 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<=i<=length+1)

Status 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
}
```

```
typedef struct
{ ElemType elem[maxleng];
  int length;
} SqList;
```

19

算法2：（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<=i<=length+1)

int 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
```

```
typedef struct
{ ElemType elem[maxleng];
  int length;
} SqList;
```

$a_1$	$a_2$	...	$a_i$	...	$a_n$	//	//	//
-------	-------	-----	-------	-----	-------	----	----	----

20

#### 算法2.4 (动态分配线性表空间, 用引用参数表示被操作的线性表)

```

int ListInsert_Sq(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)    /*溢出时扩容*/
    {
        ElemType *newbase;
        newbase = (ElemType *) realloc(L.elem,
            (L.listsize + LISTINCREMENT) * sizeof(ElemType));
        if (newbase == NULL) return OVERFLOW;    // 扩容失败
        L.elem = newbase;
        L.listsize += LISTINCREMENT;
    }
}

```

```

typedef struct
{
    ElemType *elem;
    int length;
    int listsize;
} SqList;

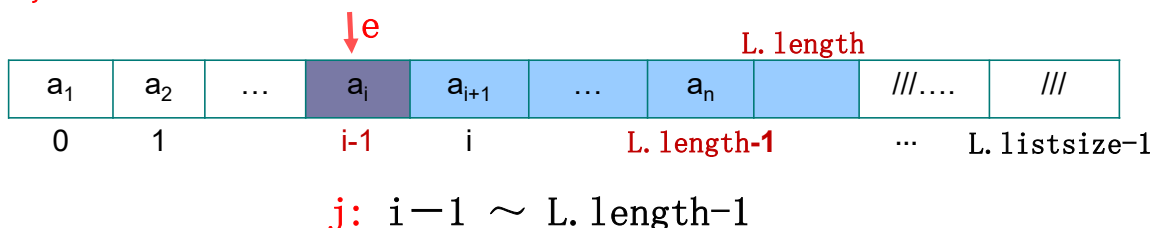
```

21

```

// 向后移动元素, 空出第i个元素的分量elem[i-1]
for (j = L.length - 1; j >= i - 1; j--)
    L.elem[j + 1] = L.elem[j];
L.elem[i - 1] = e;    /*新元素插入*/
L.length++;    /*线性表长度加1*/
return OK;
}

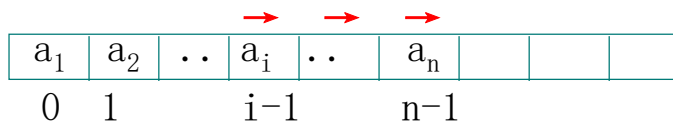
```



22

### 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  $\dots$   $i$   $\dots$   $n-1$   $n$   $n+1$

需移动元素个数:  $n$   $n-1$   $\dots$   $n-i+1$   $\dots$  2 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个数的平均值是:

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

23

### 2.2.6 删除操作及移动元素次数的分析

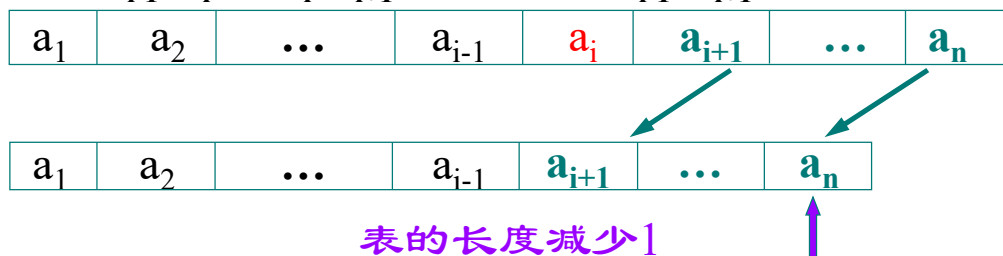
线性表操作 **ListDelete(&L, i, &e)** 的实现:

**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

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

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



24

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

```
    if ((i < 1) || (i > L.length)) return ERROR;    // 删除位置不合法
```

```
    p = &(L.elem[i-1]);    // p 为被删除元素的位置
```

```
    e = *p;    // 被删除元素的值赋给 e
```

```
    q = L.elem+L.length-1;    // 表尾元素的位置
```

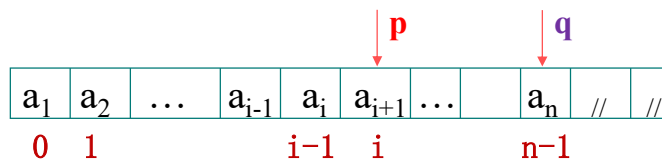
```
    for (++p; p <= q; ++p)
```

```
        *(p-1) = *p;    // 被删除元素之后的元素左移
```

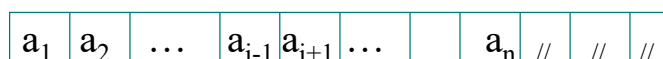
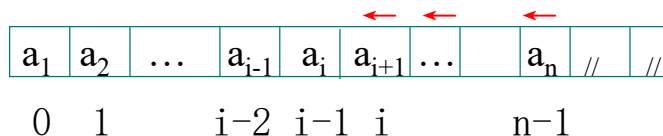
```
    --L.length;    // 表长减1
```

```
    return OK;
```

```
} // ListDelete_Sq
```



25



当 i=1 2 3 ... i ... n  
移动元素个数= n-1 n-2 ... n-i ... 0

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： $E_{dl} = \sum_{i=1}^n q_i (n-i)$

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

算法时间复杂度为： $O(n)$

26

### 2.2.7 顺序存储结构的评价

1. 优点: (1) 一种随机存取结构, 存取任何位序元素时间复杂度是 $O(1)$ ;  
(2) 结构简单, 逻辑上相邻的元素在物理上也相邻;  
(3) 不使用指针, 节省存储空间;  
(4) 适宜于某些并行算法。
2. 缺点: (1) 插入和删除元素要移动元素, 消耗一定时间;  
(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
-----	----	----	----	----	----

27

### 2.3 线性表的链式存储结构

链式存储结构特点:

其结点在存储器中的位置是随意的, 即逻辑上相邻的数据元素在物理上不一定相邻。

如何实现? 通过指针来实现!

每个存储结点包含两部分: 数据域和指针域

链式存储有关的术语:

结点: 数据元素的存储映像。由数据域和指针域两部分组成。

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结构。

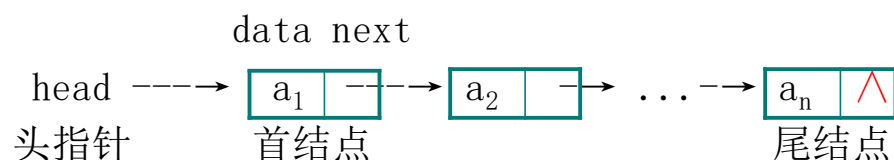
单链表、双链表、多链表、循环链表

28

## 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:



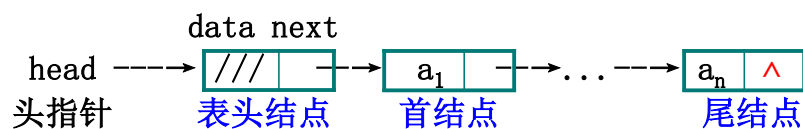
其中: data称为数据域, next称为指针域/链域

当 head==NULL, 为空表; 否则为非空表

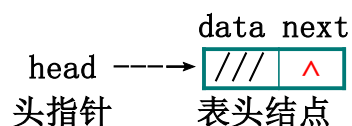
29

#### (2) 带表头结点的单链表:

##### a. 非空表:



##### b. 空表:

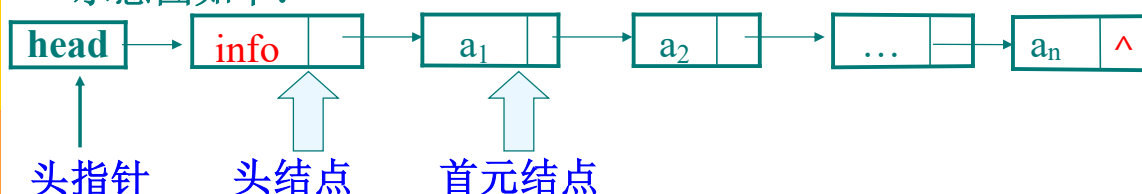


其中: head指向表头结点, head->data不放元素,  
head->next指向首结点a<sub>1</sub>,  
当head->next==NULL, 为空表; 否则为非空表。

30

## 头指针、头结点和首元节点的区别

示意图如下：



头指针是指向链表第一个结点（或为头结点、或为首元结点）的指针。

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，它不计入表长度。

首元结点是指链表中存储线性表第一个数据元素 $a_1$ 的结点。

31

### 讨论1. 在链表中设置头结点有什么好处？

头结点即在链表的首元结点之前附设的一个结点，该结点数据域可以为空，也可存放表长度等附加信息，其作用是为了对链表进行操作时，可对空表、非空表的情况以及对首元结点进行统一处理，编程更方便。

### 讨论2. 如何表示空表？

无头结点时，当头指针的值为空时表示空表；

有头结点时，当头结点的指针域为空时表示空表。

头指针



无头结点

头指针



有头结点

头结点



头结点不计入链表长度！

32



## 2 单链表的结点结构

例1 C语言的“结构”类型：

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

指向结点的指针变量head, p, q说明：

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

例2 用typedef定义指针类型：

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

指针变量head, p, q的说明： Linklist head, p, q;

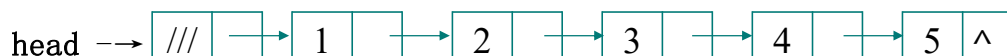
33

## 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 Lnode) //结点所占的字节数
```

```
struct Lnode //定义结点类型
```

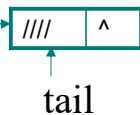
```
{ int data; //data为整型, 数据域
```

```
  struct Lnode *next; //next为指针类型, 指针域
```

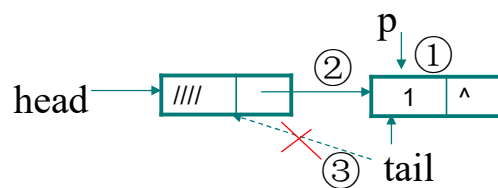
```
};
```

34

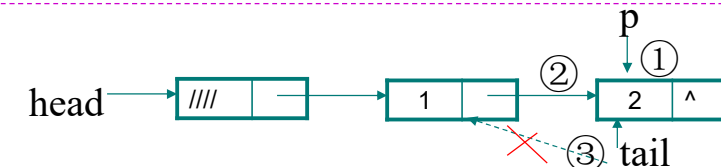
初始化: head



输入1:



输入2:

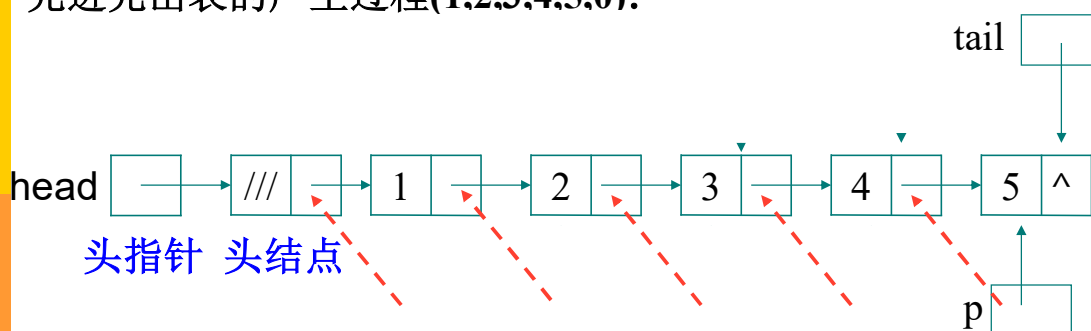


每次输入新元素后:

- 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:  $\text{tail} = p$ ;

35

先进先出表的产生过程(1,2,3,4,5,0):



**head=malloc(sizeof(struct Lnode))**

**tail=head**

**p=malloc(sizeof(struct Lnode))**

**tail->next=p**

**tail=p**

**p=malloc(sizeof(struct Lnode))**

**tail->next=p**

**tail=p**

**tail->next=NULL;**

36

### 算法：生成“先进先出”单链表（链式队列）

```

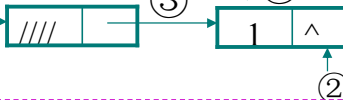
struct Lnode *creat1( )
{
    struct Lnode *head,*tail,*p;           //变量说明
    int e;
    head=(struct Lnode *)malloc(LENG);      //生成表头结点
    tail=head;                              //尾指针指向表头
    scanf("%d",&e);                        //输入第一个数
    while (e!=0)                             //不为0
    {
        p=(struct Lnode *)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
}
    
```

37

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

head → 

初始化: head → 

输入1: head → 

输入2: head → 

每次输入新元素后:

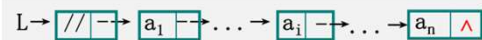
- ① 生成新结点:  $p = \text{malloc}(\text{结点大小}); p \rightarrow \text{data} = e;$
- ② 新结点指针指向“首元素”:  $p \rightarrow \text{next} = \text{head} \rightarrow \text{next};$
- ③ 新结点作为首元素:  $\text{head} \rightarrow \text{next} = p;$

### 例2 生成“后进先出”单链表(链式栈)。

```
struct Lnode *creat2( )           //生成“后进先出”单链表
{ struct Lnode *head, *p;
  head=(struct Lnode *)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 Lnode *)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;                    //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
}
```

39

### 单链表的修改(或存取)



思路：要修改第*i*个数据元素，必须从头指针起一直找到该结点，其指针为

，然后才能执行

**p->data=new\_value**。

读取第*i*个数据元素的操作函数可写为：

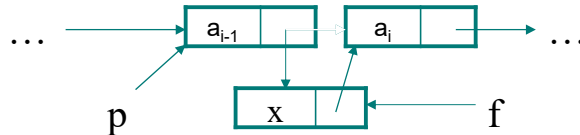
```
Status GetElem_L(LinkList L, int i, ElemType &e) //带头结点的链表L
{ p=L->next; j=1;
  while(p&&j<i) {p=p->next; ++j;}
  if( !p || j>i ) return ERROR;      // if i<1 ?
  e=p->data ;                         //若是修改则为： p->data=e;
  return OK; }                       // GetElem_L
```

缺点：存取单链表第*i*个元素，须从头指针开始逐一查询，无法随机存取。

40

## (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 (使用两个指针)



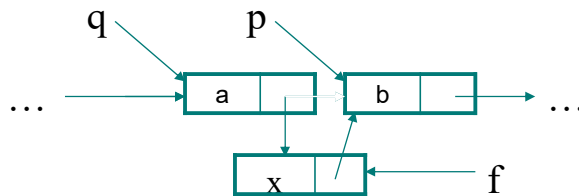
执行:

```
f=(struct Lnode *)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
```

思考: 最后两条语句能互换么?

41

例2: 在已知p指针指向的结点前插入一个元素x (使用三个指针)



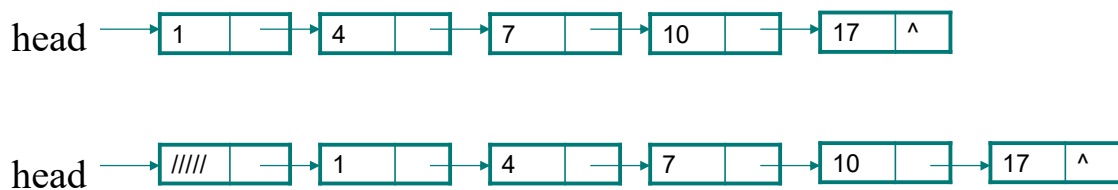
执行:

```
f=(struct Lnode *)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
```

42

**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)

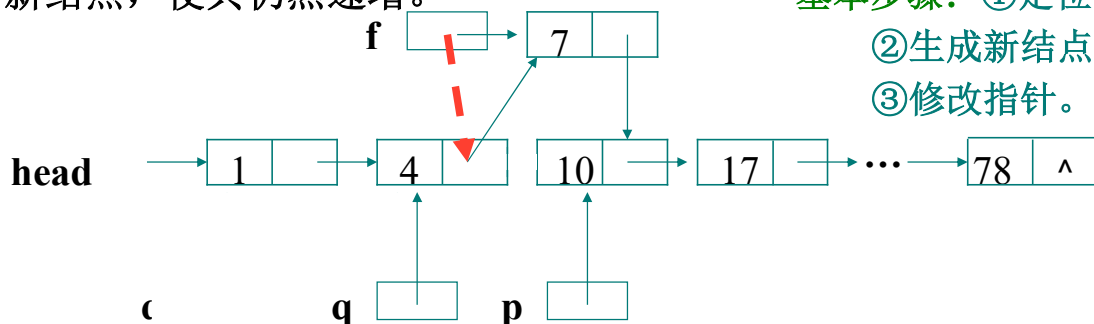
**递增有序单链表的两种形式：**



43

**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。

**基本步骤：**①定位；  
②生成新结点；  
③修改指针。



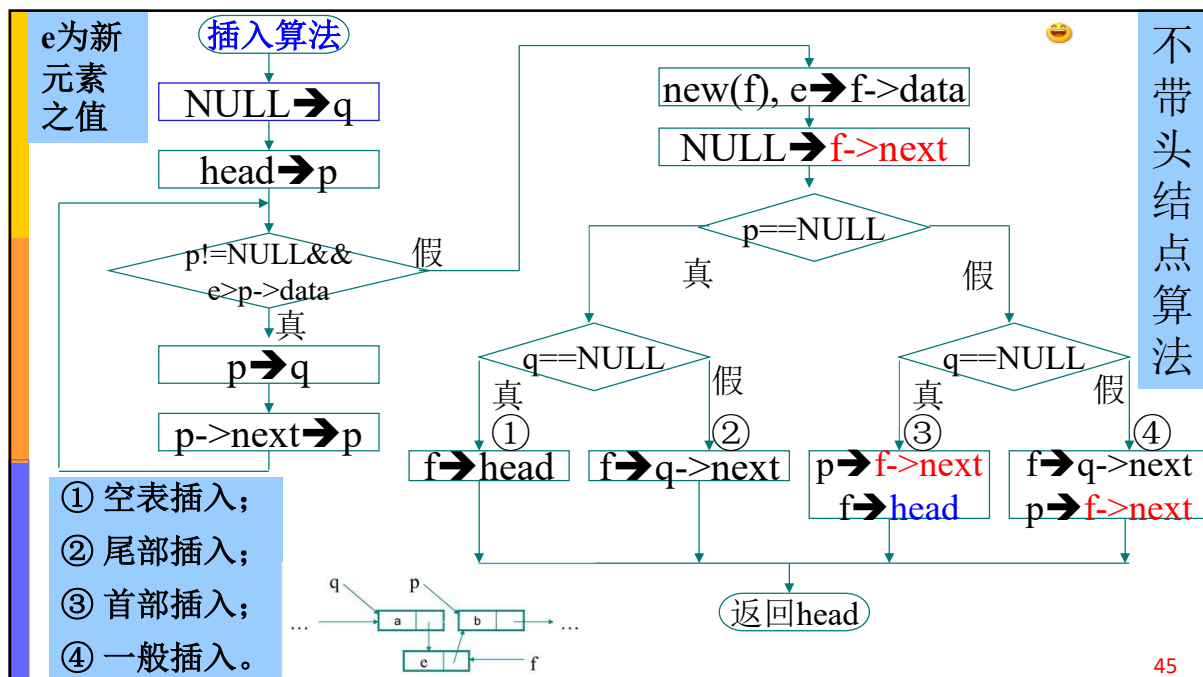
**p, q 可能为NULL**

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

**f->next=p;**

**q->next=f;**

44



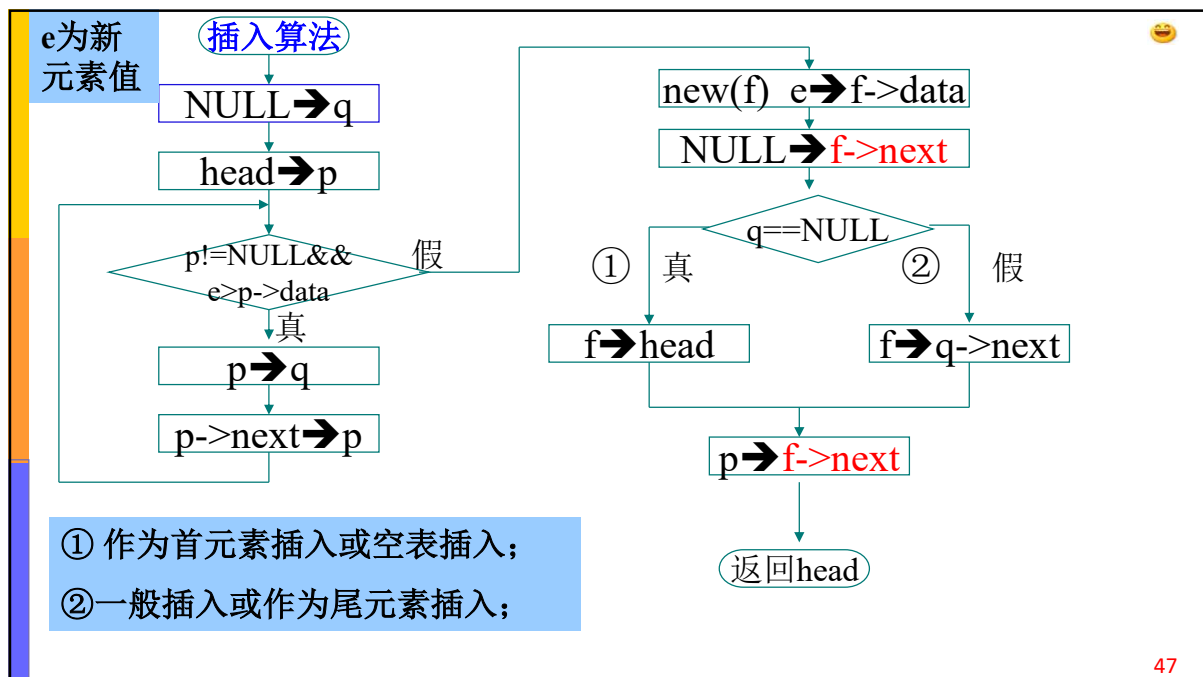
45

**算法1:** 在不带头结点的递增有序单链表中插入元素e。(不包括0)

```

struct Lnode * creat3_1(struct Lnode *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 Lnode *)malloc(LENG); f->data=e; //生成新结点
    if (p==NULL) {
        f->next=NULL;
        if (q==NULL) head=f; //①对空表的插入
        else q->next=f; //②作为最后一个结点插入
    }
    else if (q==NULL) {f->next=p; head=f;} //③作为第一个结点插入
    else {f->next=p; q->next=f;} //④一般情况插入新结点
    return head;
}
  
```

46



```

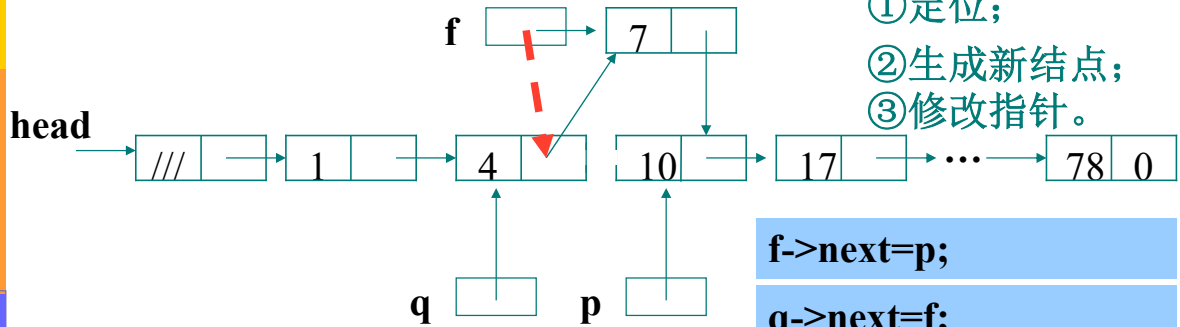
main()
{
    struct Lnode *head;           //定义头指针
    head=NULL;                    //置为空表
    scanf("%d", &e);              //输入整数
    while (e!=0)                  //不为 0，未结束
    {
        head=creat3_1(head, e);  //插入递增有序单链表
        scanf("%d", &e);        //输入整数
    }
}
  
```



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。

基本步骤：

- ①定位；
- ②生成新结点；
- ③修改指针。



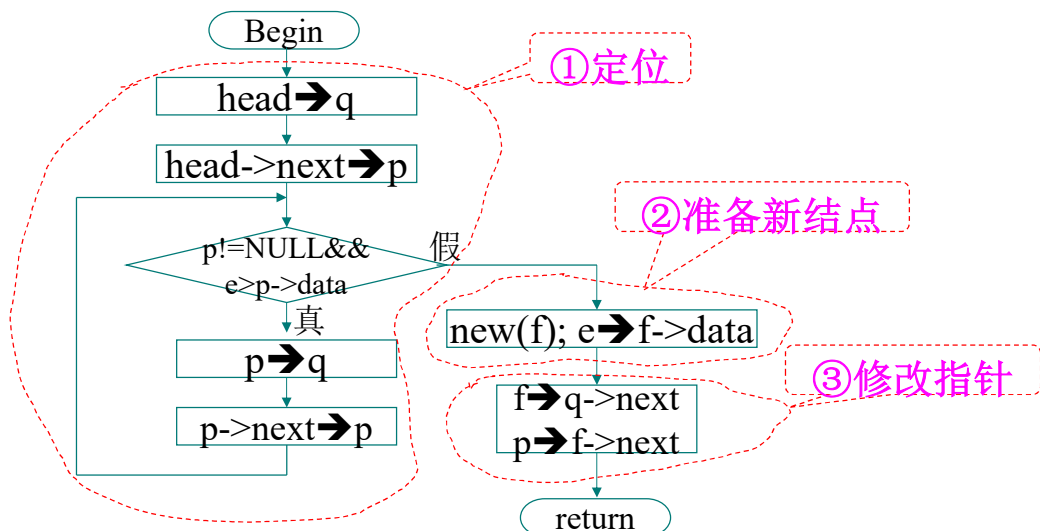
$f \rightarrow next = p;$

$q \rightarrow next = f;$

**q不可能为空**

49

## 带头结点算法 插入算法：

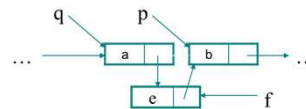


50

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 Lnode *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 Lnode *)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
}
```



51

`main()`

```
{
  head=(struct Lnode *)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
  }
}
```

52

## 线性表

### 线性表的逻辑结构:

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:  $L=(a_1, a_2, \dots, a_n)$

### 线性表顺序存储结构 (静态分配)

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

53

### 线性表顺序存储结构 (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 (表中有多少个元素)
  int listsize;            //当前分配的存储容量
} SqList;
SqList L;
```

a1	a2	...	a <sub>i</sub>	...	a <sub>n</sub>	//	//	//
----	----	-----	----------------	-----	----------------	----	----	----

$LOC(i) = LOC(1) + (i-1) * l = b + (i-1) * l \quad (1 \leq i \leq n)$

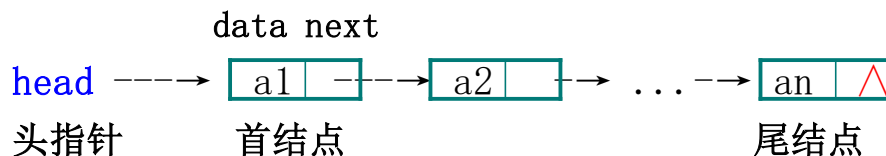
Status ListInsert\_Sq(SqList &L, int i, ElemType e)

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

顺序表(占用连续存储空间)是一种随机存取结构, 存取速度快;  
但插入与删除元素要移动数据元素, 时间复杂度为 $O(n)$ .

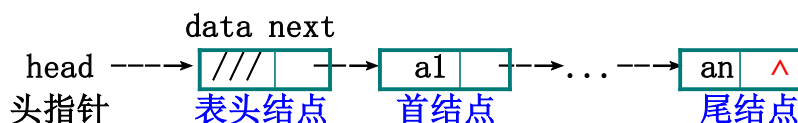
54

### (1) 不带表头结点的单链表:

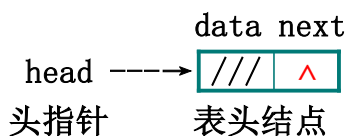


### (2) 带表头结点的单链表:

a. 非空表:



b. 空表:



55

## 单链表的结点结构与典型运算

用typedef定义指针类型:

```
typedef struct Lnode
{ ElemType data;          //data为抽象元素类型
  struct Lnode *next;     //next为指针类型
} Lnode, *Linklist;
Linklist head, p, q;      //定义指针变量head, p, q
```

**Status GetElem\_L**(LinkList L, int i, ElemType &e)

**Status ListInsert\_L**(Linklist L, int i, ElemType e)

**Status ListDelete\_L**(LinkList L, int i, ElemType &e)

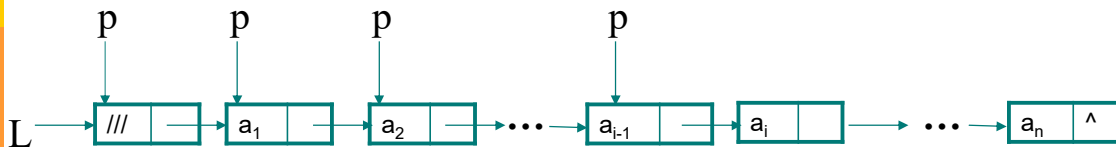
**Void MergeList\_L**(LinkList &La, LinkList &Lb, LinkList &Lc)

56

**算法3:** 在单链表的指定位置(结点*i*之前) 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*->*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*指结点之后

57

**算法3 (2.9) 程序:**

**Status ListInsert\_L(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

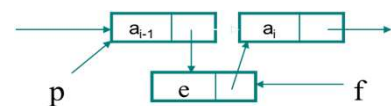
return ERROR;

*f*=(LinkList) malloc(LENG); *f*->*data*=*e*;    //生成新结点

*f*->*next*=*p*->*next*;    *p*->*next*=*f*;    //插入新结点

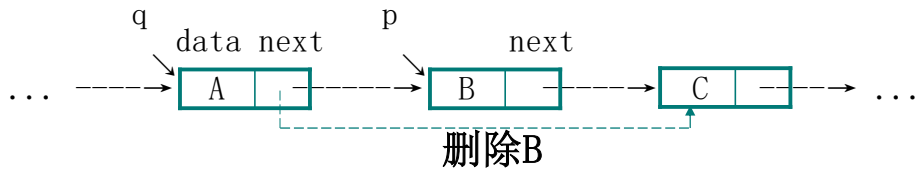
return OK;

}//ListInsert\_L

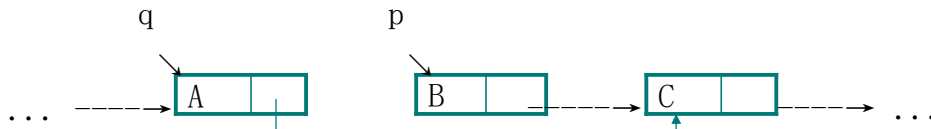


58

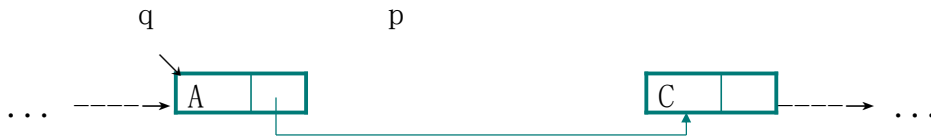
### (3) 在单链表中删除一个结点



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

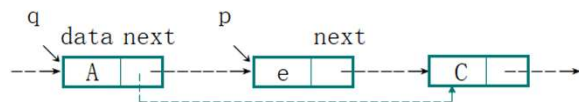


59

算法1: 在带表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 Lnode *q, *p;
  q=head;  p=head->next; //q, p扫描, 与有序链表的插入类似
  while (p && p->data!=e) //查找元素为e的结点
  { q=p; p=p->next;      //查找下一个结点
  }
  if (p) //有元素为e的结点
  { q->next=p->next; free(p); //删除该结点
    return YES;
  }
  else
    return NO;
}
```

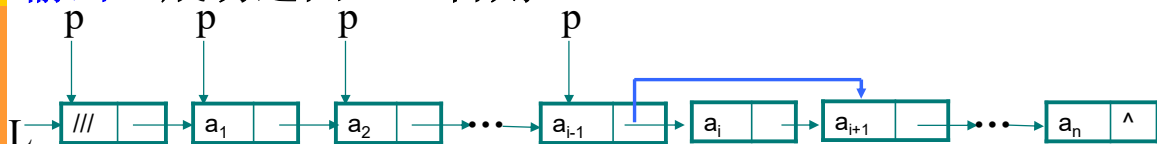


60

算法2 (2.10)：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->next不为空

执行：p=p->next    i-1次

定位到第i-1个结点

当i<1或p->next为空时

删除点错

否则：p结点指针指向该结点

后继的后继，跳过原后继。

61

Status ListDelete\_L(LinkList L, int i, ElemType &e) {

// 删除以 L 为头指针(带头结点)的单链表中第 i 个结点

p = L; j = 0;

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

// 寻找第 i 个结点，并令 p 指向其前趋

if (!(p->next) || j > i-1)

return ERROR;

// 删除位置不合理

q = p->next; p->next = q->next;

// 删除并释放结点

e = q->data; free(q);

return OK;

} // ListDelete\_L

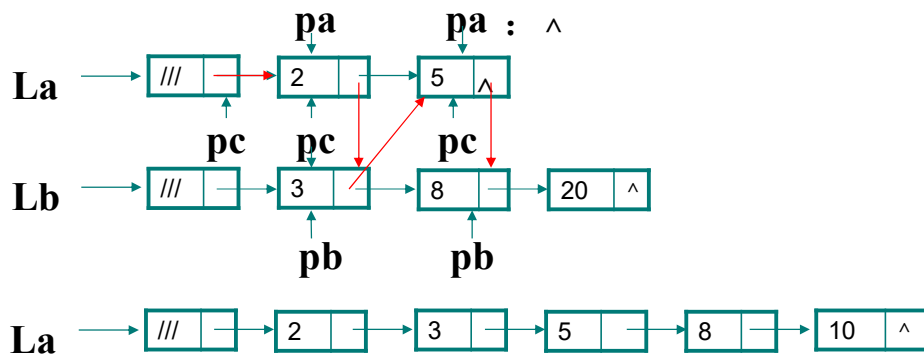
算法2.9的时间复杂度为：

O(ListLength(L))

62

(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)



63

算法设计:

算法主要包括搜索、比较、插入三个操作:

搜索: 需要设立三个指针来指向La、Lb和Lc链表;

比较: 比较La和Lb表中结点数据的大小;

插入: 将La和Lb表中数据较小的结点插入新链表Lc。

仅改变指针便可实现数据的移动, 即  
“数据不动, 指针动”

64



```

Void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc)
{ //单链表La和Lb的元素按值非递减排列, 归并为Lc后亦非递减排列。
  pa=La->next; pb=Lb->next;
  Lc=pc=La;           //用La的头结点作为Lc的头结点
  while(pa&&pb)        //将pa、pb结点按大小依次插入Lc中
  { 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; //插入非空表的剩余段
  free(Lb);              //释放Lb的头结点
} //MergeList_L

```

65

## 算法2.6 (P26): 两个有序顺序表的归并

```

Void MergeList_sq(SqList La, SqList Lb, SqList &Lc) {
  pa = La.elem; pb = Lb.elem;
  Lc.listsize = Lc.length = La.length + Lb.length;
  pc = Lc.elem = (ElemType *) malloc(Lc.listsize * sizeof(ElemType));
  if (!Lc.elem) exit(OVERFLOW); // 存储分配失败
  pa_last = La.elem + La.length - 1;
  pb_last = Lb.elem + Lb.length - 1;
  while (pa <= pa_last && pb <= pb_last) { // 归并
    if (*pa <= *pb) *pc++ = *pa++;
    else *pc++ = *pb++;
  }
  while (pa <= pa_last) *pc++ = *pa++; // 插入La的剩余元素
  while (pb <= pb_last) *pc++ = *pb++; // 插入Lb的剩余元素
} //MergeList_sq

```

66



一线性表  $S = (ZHAO, QIAN, SUN, LI, ZHOU, WU)$ ,

用静态链表如何表示?

i	data	cur
0	头结点	1
1	ZHAO	3
2	LI	5
3	QIAN	6
4	WU	0
5	ZHOU	4
6	SUN	2
...	.....	.....
1000		

**说明1:** 假设S为SLinkList型变量, 则S[MAXSIZE] 为一个静态链表;  
S[0].cur则表示第1个结点在数组中的位置。

**说明2:** 如果数组的第i个分量表示链表的第k个结点, 则:

S[i].data表示第k个结点的数据;

S[i].cur 表示第k+1个结点(即k的直接后继)的位置。

69

**说明:** 静态链表的插入与删除操作与普通链表一样, 不需移动数据元素, 只需修改指示器。

例如: 在线性表  $S = (ZHAO, QIAN, SUN, LI, ZHOU, WU)$  的 QIAN, SUN之间插入新元素LIU, 可以这样实现:

i	data	cur
0	头结点	1
1	ZHAO	3
2	LI	5
3	QIAN	7
4	WU	0
5	ZHOU	4
6	SUN	2
7	LIU	6
1000	...	...

**Step1:** 将QIAN的游标值存入LIU的游标中:  $S[7].cur = S[3].cur;$

**Step2:** 将QIAN的游标换为新元素LIU的下标:  $S[3].cur = 7$

70

## 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

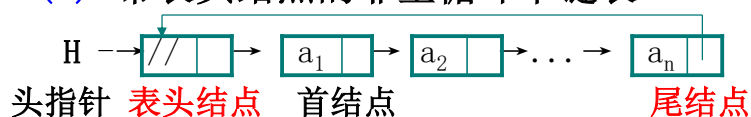
- 1 通过保存直接后继元素的存储位置来表示数据元素之间的逻辑关系；
- 2 插入删除操作通过修改结点的指针实现；
- 3 不能随机存取元素。

71

2.3.2 循环链表：最后一个结点的指针域的指针又指回头结点的链表。

### 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



头指针 表头结点

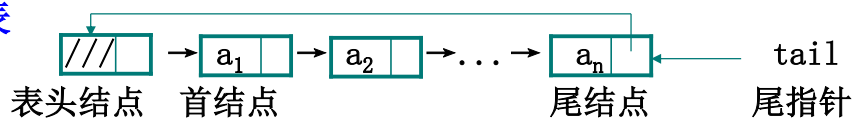
有： $H \rightarrow \text{next} == H$ ,  $H \neq \text{NULL}$

和单链表的差别仅在于：判别链表中尾结点的条件不再是“后继是否为空”，而是“后继是否为头结点”。

72

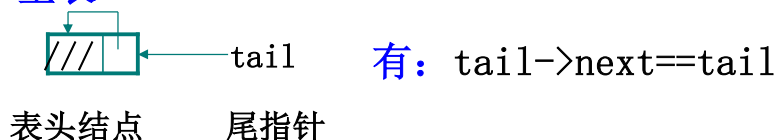
## 2. 只设尾指针的循环链表

### (1) 非空表



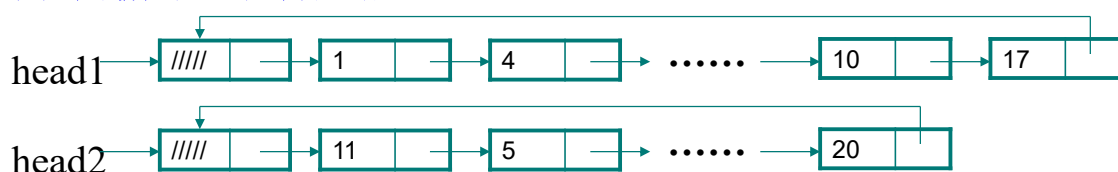
有：  
 tail指向表尾结点  
 tail->data ==  $a_n$   
 tail->next 指向表头结点  
 tail->next->next 指向首结点  
 tail->next->next->data ==  $a_1$

### (2) 空表

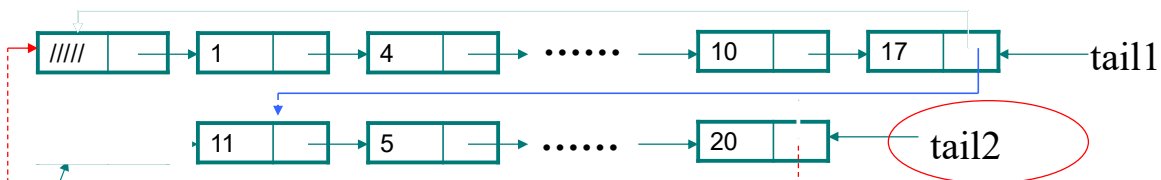


73

## 例：两循环链表首尾相连。



时间复杂度：O(m+n)



- (1)  $p_2 = \text{tail}_2 \rightarrow \text{next};$       (2)  $\text{tail}_2 \rightarrow \text{next} = \text{tail}_1 \rightarrow \text{next};$   
 (3)  $\text{tail}_1 \rightarrow \text{next} = p_2 \rightarrow \text{next};$       (4)  $\text{free}(p_2);$  时间复杂度：O(1)

74

### 3. 循环链表算法举例

例. 求头指针为head的循环单链表的长度, 并依次输出结点的值。

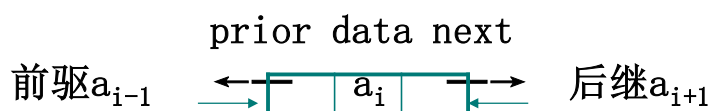
```
int length(struct Lnode *head)
{ int leng=0;
  struct Lnode *p;
  p=head->next;           //p指向首结点
  while (p!=head)         //p未移回到表头结点
  { printf("%d",p->data); //输出
    leng++;               //计数
    p=p->next;            //p移向下一结点
  }
  return leng;            //返回长度值
}
```



75

### 2.3.4 双向链表

1. 双向链表的结点结构:



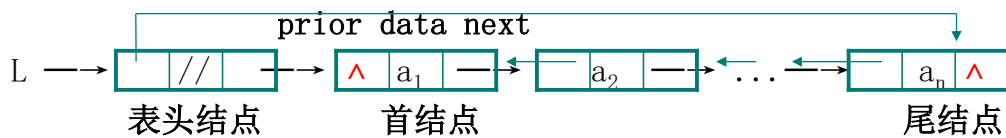
双向链表的存储结构类型定义如下:

```
typedef struct DuLNode{
    ElemType      data;           //数据域
    struct DuLNode *prior;        //前驱指针域
    struct DuLNode *next;        //后继指针域
} DuLNode, *DuLinkList;
```

76

## 2. 双向链表的一般形式

### (1) 非空表



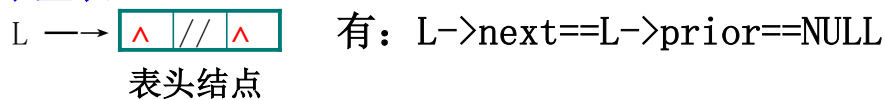
有：L为头指针, L指向表头结点,  $L \rightarrow \text{next}$ 指向首结点

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

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

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

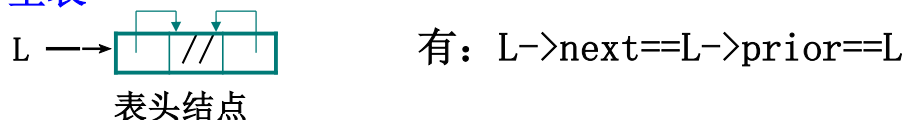
### (2) 空表



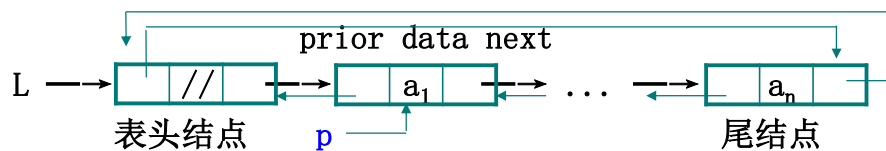
77

## 3. 双向循环链表

### (1) 空表



### (2) 非空表



设p指向 $a_1$ , 有:

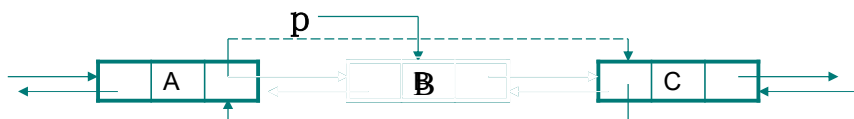
$p \rightarrow \text{next}$ 指向 $a_2$ ,  $p \rightarrow \text{next} \rightarrow \text{prior}$ 指向 $a_1$ ,

所以,  $p == p \rightarrow \text{next} \rightarrow \text{prior}$

同理:  $p == p \rightarrow \text{prior} \rightarrow \text{next}$

78

### (3) 已知指针p指向结点B，删除B

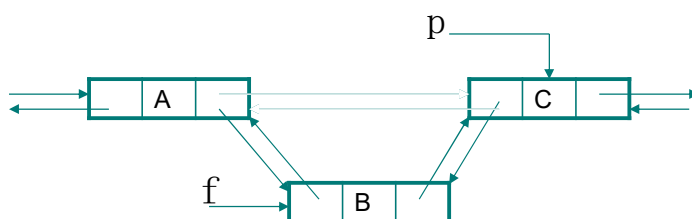


执行:

```
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
```

79

### (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

80



用单链表实现线性表的操作时，存在的**问题**：

1. 单链表的表长是一个隐含的值；
2. 在单链表的最后一个元素之后插入元素时，需遍历整个链表；
3. 在链表中元素的“位序”概念淡化，结点的“位置”加强。

**改进链表的设置**：

1. 增加“表长”、“表尾指针”和“当前位置的指针”三个数据域；
2. 将基本操作中的“位序  $i$ ”改变为“指针  $p$ ”。

81

### 线性表的两种存储结构的比较

具体要求	顺序表	链表
基于空间	表长变化不大，事先易确定其大小时采用。	适于当线性表长度变化大，难以估计其存储规模时采用。
基于时间	一种 <b>随机存取</b> 的存储结构，当线性表的操作以 <b>查找</b> 为主时，宜采用。	对任何位置进行 <b>插入和删除</b> 都只需修改指针，以这类操作为主的线性表宜采用链表实现。  若插入和删除主要发生在表的首尾两端，则宜采用尾指针表示的 <b>单循环链表</b> 。
基于并行	适于并行处理	不适于并行处理

TreeList ?

82

### 讨论如下问题:

- 线性表的两种存储结构各有哪些优缺点?
- 对于线性表的两种存储结构, 如有n个表同时并存, 且处理过程中各表的长度会动态发生变化, 表的总数也可能自动改变, 在此情况下应选用哪种存储表示? 为什么?
- 若表的总数基本稳定, 且很少插入和删除, 但需以最快速度存取表中元素, 则应采用哪种存储表示? 为什么?
- 顺序表与链表进行插入操作时, 有什么不同?

83

## 2.4 一元多项式的表示与相加

### 一元多项式

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

在计算机中, 可以用一个线性表来表示:

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

但是对于如下形式的多项式, 上述表示方法是否合适?

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

84

一般情况下的一元稀疏多项式可写成

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

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

可用线性表表示为

$$((7, 3), (-2, 12), (-8, 999))$$

85

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:

$R_1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n, \text{且} \\ a_{i-1} \text{中的指数值} < a_i \text{中的指数值} \}$

86

### 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。

87

#### **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 )**

... ..

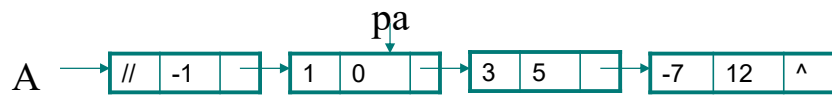
} ADT Polynomial

88

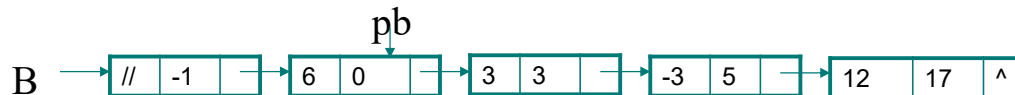
## 多项式的链表表示

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

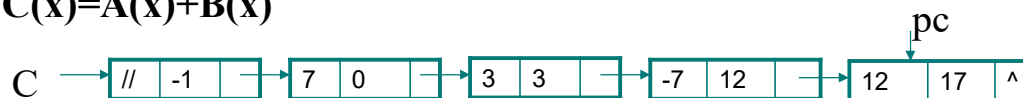
$$A_{12}(x)=1+3x^5-7x^{12}$$



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



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



多项式相加与两个有序单链表的归并有何异同?

89

**Void MergeList\_L(LinkList &La,LinkList &Lb,LinkList &Lc)**

{ //已知单链表La和Lb的元素按值非递减排列, 归并为Lc后也按值非递减排列。

pa=La->next; pb=Lb->next;

Lc=pc=La; //用La的头结点作为Lc的头结点

while(pa&&pb) //将pa、pb结点按大小依次插入Lc中

{ 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; //插入非空表的剩余段

free(Lb); //释放Lb的头结点

} //MergeList\_L

90

### $C(x)=A(x)+B(x)$ 的算法步骤:

1.  $pa$ 、 $pb$ 分别指向首元素结点,生成 $C(x)$ 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 $pa \rightarrow \text{expn}$ 等于 $pb \rightarrow \text{expn}$ :

- (a)  $pa \rightarrow \text{coef} + pb \rightarrow \text{coef}$ 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)  $pa \rightarrow \text{coef} + pb \rightarrow \text{coef}$ 等于零:  $pa$ 、 $pb$ 后移

2-2  $pa \rightarrow \text{expn}$ 小于 $pb \rightarrow \text{expn}$ : 由 $pa$ 产生新结点, 添加到 $pc$ 后,  
 $pc$ 指向新结点,  $pa$ 后移

2-3  $pa \rightarrow \text{expn}$ 大于 $pb \rightarrow \text{expn}$ : 由 $pb$ 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。

91

### 运算效率分析:

#### (1) 系数相加

$0 \leq \text{加法次数} \leq \min(m, n)$

其中  $m$ 和 $n$ 分别表示表 $A(x)$ 和表 $B(x)$ 的结点数。

#### (2) 指数比较

极端情况是表 $A$ 和表 $B$  没有一项指数相同,  
比较次数最多为 $m+n-1$ 。

#### (3) 新结点的创建

极端情况是产生 $m + n$  个新结点

合计时间复杂度为  $O(m+n)$

92

## 本章小结

### 1. 线性结构(包括表、栈、队、数组)的定义和特点:

仅一个首、尾结点, 其余元素仅一个直接前驱和一个直接后继。

### 2. 线性表

- 逻辑结构: “一对一” 或 “1:1”
- 存储结构: 顺序、链式
- 运算: 修改、插入、删除 [查找和排序另述]

### 3. 顺序存储

- 特征: 逻辑上相邻, 物理上也相邻;
- 优点: 随机查找修改快  $O(1)$
- 缺点: 插入、删除慢  $O(n)$

改进方案: 链表存储结构

93

### 4. 链式存储

- 特征: 逻辑上相邻, 物理上未必相邻;
- 优点: 在当前位置插入、删除快  $O(1)$
- 缺点: 随机查找修改慢  $O(n)$

### 5. 几种特殊链表的特点:

**静态链表的特点:** 不用指针也能实现链式存储和运算

**循环链表的特点:** 从任一结点出发均可找到表中其他结点

**双向链表的特点:** 可方便找到任一结点的前驱

94

### 经典问题：

例1：试用C或类C编写一个**高效**算法，将一循环单链表就地逆置。

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

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

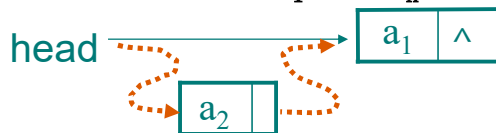
分析：要想让 $a_n$ 指向 $a_{n-1}$ ，..... $a_2$ 指向 $a_1$ ，一般有两种算法：

①替换法：扫描 $a_1 \dots \dots a_n$ ，将每个 $a_{i-1}$ 的指针送入 $a_i$ 的指针域。

思路：前驱变后继

②插入法：扫描 $a_1 \dots \dots a_n$ ，将每个 $a_i$ 插入到链表头部即可。

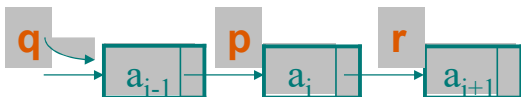
思路：头部变尾部



95

### 替换法的核心语句：

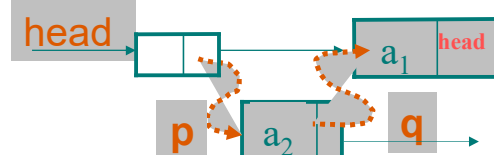
```
q=head;
p=head->next; //有头结点
while(p!=head) //循环单链表
{ r=p->next;
  p->next=q; //p的前驱变后继
  q=p;
  p=r; } //准备处理下一结点
head->next=q; //以 $a_n$ 为首
```



请上机验证并分析效率！

### 插入法的核心语句：

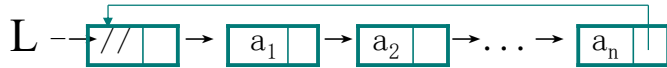
```
p=head->next; //有头结点
if(p!=head){q=p->next;
  p->next=head;p=q}; //处理 $a_1$ 
while(p!=head) //循环单链表
{ q=p->next //保存原后继
  p->next=head->next;
  head->next=p;
  p=q;} //准备处理下一结点
```



96



### ③递归算法（参考袁老师教材2.5.4节）



void reverse (LinkedList L)

{

LinkedList p = L->next; //获得首元素节点

if(L->next == L || p->next==L)

return; //空链表或只剩一个结点时返回

L->next = p->next;

reverse(L); //剩余链递归

p->next->next = p; //a<sub>1</sub>作为a<sub>2</sub>的后继

p->next = L;

}

思路:

当循环链表长度≤1,

递归返回;

当循环链表长度≥2,

摘除首元结点p,

将剩余循环链表逆置,

再将p结点插入之尾部。

97

**例2:** 试用C或类C语言编写一**高效**算法，将一顺序存储的线性表（设元素均为整型量）中所有零元素向表尾集中，其他元素则**顺序**向表头方向集中。

(a<sub>1</sub>, a<sub>2</sub>, ..., a<sub>i-1</sub>, a<sub>i</sub>, a<sub>i+1</sub>, ..., a<sub>n</sub>)

常见做法:

① 从前往后扫描，见到0元素则与尾部非0元素互换; ✗

② 从后往前扫描，见到0元素则后面元素统统前移; ✗

③ 从前往后扫描，见到0元素先计数，再将后续的一个非0元素前移，全部扫完后再把后续部分置0（长度为0元素的个数）。 ✓

98

```

int SortA(SqList &L)
{ int i, zeroNum =0;
if(L.length==0) return(0); //空表则结束
else { for( i=1; i<=L.length; i++)
        {if (L.elem[i] != 0) L.elem[i- zeroNum]= L.elem[i];
        else zeroNum++; }
}
for( i= L.length-zeroNum+1; i<=L.length; i++)
L.elem[i]=0; //表的后部置0
return ok;
} // SortA

```

若表元素全非0,  
也要移动n次?

99

若考虑表完全非空的情况，则程序变长较多。

```

int SortA(SqList &L)
{ int i, zeroNum =0;
if(L.length==0) return(0); //空表则不执行
for ( i=1; i<=L.length; i++)
    {if (L.elem[i] != 0)
        if(zeroNum!=0) L.elem[i- zeroNum]= L.elem[i];
        else zeroNum++; } //适当移动非零元素, 是零则增加计数
for ( i= L.length-zeroNum+1; i<=L.length; i++)
    L.elem[i]=0; //表的后部补0
return ok;
}

```

**讨论1:** 已知带头结点的单链表，其头指针为head, 在不改变链表的情况下，设计一**高效**算法输出链表中倒数第k个结点数据域的值（用C,C++或JAVA语言实现）。



**讨论2:** 一个长度为n ( $n \geq 1$ ) 的升序序列L，处在第  $\lceil n/2 \rceil$  的数称为L的中位数。两个序列的中位数是含它们所有元素的升序序列的中位数。现有两个等长的升序序列A和B，试设计一时空**高效**算法求出两个升序序列A和B的中位数。（用C,C++或JAVA语言实现,2011）。

例: (11, 13, 15, 17, 19)

(2, 4, 6, 8, 20)

这两个序列的中位数是11.

101

41. 用单链表保存 m 个整数，节点的结构为(data,link)，且  $data < n$  (n 为正整数)。现要求设计一个时间复杂度尽可能高效地算法，对于链表中绝对值相等的节点，仅保留第一次出现的节点而删除其余绝对值相等的节点。

例如若给定的单链表 head 如下



删除节点后的 head 为



要求

- (1) 给出算法的基本思想
- (2) 使用 c 或 c++语言，给出单链表节点的数据类型定义。
- (3) 根据设计思想，采用 c 或 c++语言描述算法，关键之处给出注释。
- (4) 说明所涉及算法的时间复杂度和空间复杂度。

可借助辅助数组.

102

**2018. (13分)** 给定一个含 $n$  ( $n \geq 1$ ) 个整数的数组，请设计一个时间上尽可能高效的算法，找出数组中未出现的最小正整数。例如，数组 $[-5, 3, 2, 3]$ 中未出现的最小正整数是1；数组 $[1, 2, 3]$ 中未出现的最小正整数是4。要求：

- (1) 给出算法的基本设计思想。
- (2) 根据设计思想采用C或C++语言描述算法，关键处给出注释。
- (3) 说明你所设计算法的时间复杂度和空间复杂度。

可借助辅助数组.

103

## 课外思考

P45 袁-本章习题  
一 二 三 2, 5-10

### 课外编程练习

先建立一个整型数的单链表，然后统计单链表中元素为0的个数。

104