



数据结构

第二章 线性表

主讲：陈锦秀

厦门大学信息学院计算机系

第二章 线性表

2.1 线性表的类型定义

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

2.3.1 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

2.1 线性表的类型定义

- **线性表(Linear List)**：由 $n(n \geq 0)$ 个数据元素(结点) a_1, a_2, \dots, a_n 组成的有限序列。其中数据元素的个数 n 定义为表的长度。当 $n=0$ 时称为空表，常常将非空的线性表($n > 0$)记作： (a_1, a_2, \dots, a_n)
- 这里的数据元素 $a_i(1 \leq i \leq n)$ 只是一个抽象的符号，其具体含义在不同的情况下可以不同。
- 例1、26个英文字母组成的字母表
(A, B, C, ..., Z)
- 例2、某校从1978年到1983年各种型号的计算机拥有量的变化情况。
(6, 17, 28, 50, 92, 188)

2.1 线性表的类型定义

•例3、学生健康情况登记表如下：

姓 名	学 号	性 别	年 龄	健康情况
王小林	790631	男	18	健康
陈 红	790632	女	20	一般
刘建平	790633	男	21	健康
张立立	790634	男	17	神经衰弱
.....

■ 例4、一副扑克的点数

(2, 3, 4, ..., J, Q, K, A)

■ 从以上例子可看出线性表的**逻辑特征**是：

- 在非空的线性表，有且仅有一个开始结点 a_1 ，它没有直接前趋，而仅有一个直接后继 a_2 ；
- 有且仅有一个终端结点 a_n ，它没有直接后继，而仅有一个直接前趋 a_{n-1} ；
- 其余的内部结点 $a_i(2 \leq i \leq n-1)$ 都有且仅有一个直接前趋 a_{i-1} 和一个直接后继 a_{i+1} 。

■ **线性表是一种典型的线性结构。**

■ 数据的运算是定义在逻辑结构上的，而运算的具体实现则是在存储结构上进行的。

■ 抽象数据类型的定义为：P19

■ 抽象数据类型线性表的定义如下：

ADT List{

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

数据关系： $R1=\{ \langle a_{i-1}, a_i \rangle \mid a_{i-1} \text{ 和 } a_i \text{ 属于 ElemSet, } i=1,2,\dots,n \}$

基本操作：

InitList(&L)

DestoryList(&L)

ClearList(&L)

ListEmpty (L)

ListLength(L)

GetElem(L, i, &e)

LocateElem(L, e, compare())

PriorElem(L, cur_e, &pre_e)

NextElem(L, cur_e, &next_e)

ListInsert(&L, i, e)

ListDelete(&L, i, &e)

ListTraverse(L, visit())

}ADT List

初始化

销毁

操作

线性表

置空表

判空

求线性表

的长度

求线性表

中某个数

据元素

定位函数

求数据元

素的前驱

求数据元

素的后驱

插入数据

元素

删除数据

元素

遍历线

性表

2.1 线性表的类型定义

- 利用上述定义的线性表可以实现其它更复杂的操作。
- 如：算法2.1和算法2.2

算法2.1

- 例2-1 利用两个线性表**LA**和**LB**分别表示两个集合**A**和**B**，现要求一个新的集合 **$A=A \cup B$** 。

1.[初值]

获取线性表**LA**和**LB**

2.[合并线性表]

对于**LB**中的每一个元素**x**做如下操作：

若 (**x**不属于**LA**) 则 将**x**插入到**LA**的末尾

3.[算法结束]

算法2.1

```
void union(List &La, List Lb) {  
    La_len=Listlength(La);  
    Lb_len=Listlength(Lb);  
    for(i=1;i<=Lb_len;i++) {  
        GetElem(Lb, i, e);  
        if(!LocateElem(La, e, equal))  
            ListInsert(La, ++La_len, e);  
    }  
}
```

求线性表的
长度

取**Lb**中第**i**个
数据赋给**e**

La中不存在和**e**相
同的数据元素，
则插入之

算法2.2

- 例2-2 已知线性表**LA**和线性表**LB**中的数据元素按值非递减有序排列，现要求将**LA**和**LB**归并为一个新的线性表**LC**，且**LC**中的元素仍按值非递减有序排列。
- 例如：设**LA**=(3,5,8,11)
 LB=(2,6,8,9,11,15,20)
 则 **LC**=(2,3,5,6,8,8,9,11,11,15,20)
- 此问题的算法如下：

算法2.2

1.[初值]

获取线性表**LA**和**LB**，并构造空线性表**LC**

2.[选择插入元素]

对于线性表**LA**和**LB**，都从其第一个元素开始做如下操作直到其中一个线性表元素全部遍历完毕：

若 (**LA**的元素 $a \leq$ **LB**的元素 b)

则

将元素 a 插入到**LC**的末尾，并选择**LA**中的下一个元素 a

否则

将元素 b 插入到**LC**的末尾，并选择**LB**中的下一个元素 b

3.[补充剩下的元素]

若 (**LA**还有剩余元素) 则 将**LA**的剩余元素全部插入到**LC**末尾

若 (**LB**还有剩余元素) 则 将**LB**的剩余元素全部插入到**LC**末尾

4.[算法结束]

```
void MergeList(List La, List Lb, List &Lc)
```

```
    InitList(Lc);
```

```
    i=j=1;k=0;
```

```
    La_len=ListLength(La);
```

```
    Lb_len=ListLength(Lb);
```

```
    while((i<=La_len)&&(j<=Lb_len)){
```

```
        GetElem(La, i, ai);
```

```
        GetElem(Lb, j, bj);
```

```
        if(ai<=bj){ListInsert(Lc, ++k, ai);++i;}
```

```
            else{ListInsert(Lc, ++k, bj);++j;}
```

```
    }
```

```
    while(i<=La_len){
```

```
        GetElem(La, i++, ai);
```

```
        ListInsert(Lc, ++k, ai);
```

```
    }
```

```
    while(j<=Lb_len){
```

```
        GetElem(Lb, j++, bj);
```

```
        ListInsert(Lc, ++k, bj);
```

```
    }
```

```
}
```

La和Lb均
非空

算法复杂性分析

■ 算法2.1

- 外重循环为**ListLength(LB)**
- 循环内语句**LocateElem()**的时间复杂度为 **$O(\text{ListLength(LA)})$**
- 总为 **$O(\text{ListLength(LA)} * \text{ListLength(LB)})$**

■ 算法2.2

- 根据算法的执行过程，算法访问**LA**和**LB**的每个元素有仅只有一次
- **$O(\text{ListLength(LA)} + \text{ListLength(LB)})$**

第二章 线性表

2.1 线性表的类型定义

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

2.3.1 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

2.2 线性表的顺序表示和实现

2.2.1 线性表

- 把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称顺序表。
- 假设线性表的每个元素需占用 l 个存储单元，并以所占的第一个单元的存储地址作为数据元素的存储位置。则线性表中第 $i+1$ 个数据元素的存储位置 $LOC(a_{i+1})$ 和第 i 个数据元素的存储位置 $LOC(a_i)$ 之间满足下列关系：

$$LOC(a_{i+1}) = LOC(a_i) + l$$

线性表的第 i 个数据元素 a_i 的存储位置为：

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

基地址

用一组地址连续的存储单元

依次存放线性表中的数据元素



线性表的起始地址
称作线性表的基地址

2.2.1 线性表

- 由于C语言中的一维数组也是采用顺序存储表示，故可以用**数组类型**来描述顺序表。又因为顺序表还应该用一个变量来表示线性表的长度属性，所以我们用**结构类型**来定义顺序表类型。

```
#define LIST_INIT_SIZE 100 //初始分配量
#define LISTINCREMENT 10  //分配增量
typedef struct{
    ElemType *elem; //基址
    int length;      //当前长度
    int listsize;    //当前分配的存储容量，
                    //以sizeof(ElemType)为单位
} Sqlist;           // 俗称 顺序表
```

结构初始化操作:

```
Status InitList_Sq(Sqlist &L){
```

```
    //构造一个空的线性表L。
```

```
    L.elem=(ElemType*)malloc(LIST_INIT_SIZE*sizeof(ElemType));
```

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

```
    L.length=0; //空表长度为0
```

```
    L.listsize=LIST_INIT_SIZE; //初始存储容量
```

```
    return OK;
```

```
}
```

2.2.2 顺序表上实现的基本操作

- 在顺序表存储结构中，很容易实现线性表的一些操作，如线性表的构造、第*i*个元素的访问。

注意：C语言中的数组下标从“0”开始，因此，若L是Sqlist类型的顺序表，则表中第*i*个元素是L.elem[i-1]。

- 以下主要讨论线性表的插入和删除两种运算。

2.2.2 顺序表上实现的基本操作

1、插入

线性表的插入运算是指在表的第 i ($1 \leq i \leq n+1$)个位置上，插入一个新结点 x ，使长度为 n 的线性表

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

变成长度为 $n+1$ 的线性表

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

见P23图2.3

1. [初值]

获取线性表L，插入位置i，插入元素e

2. [检查参数]

若（插入位置超出线性表长度范围） 则 输出错误

3. [检查空间]

若（线性表空间不足）

则

分配新空间

若（分配成功） 则 修改线性表的容量

否则 输出溢出错误

4. [插入元素]

将插入位置i以及其后的L中的元素全部后移一格

将元素e插入位置i

线性表长度增加1

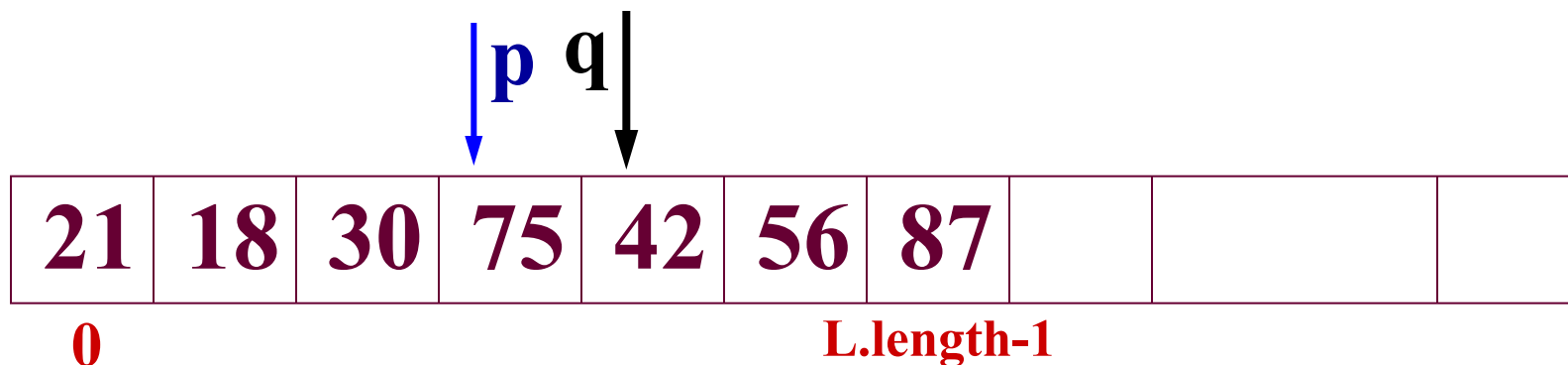
5. [算法结束]

例如: ListInsert_Sq(L, 5, 66)

$q = \&(L.elem[i-1]);$ // q 指示插入位置

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

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



分析“插入”算法的复杂度

- 这里的问题规模是表的长度，设它的值为 n 。该算法的时间**主要花费**在循环的结点后移语句上，该语句的执行次数（即移动结点的次数）是 $n-i+1$ 。由此可看出，所需移动结点的次数不仅依赖于表的长度，而且还与插入位置有关。
- 当 $i=n+1$ 时，由于循环变量的终值大于初值，结点后移语句将不进行；这是最好情况，其时间复杂度 $O(1)$ ；
- 当 $i=1$ 时，结点后移语句将循环执行 n 次，需移动表中所有结点，这是最坏情况，其时间复杂度为 $O(n)$ 。

分析“插入”算法的复杂度:

- 由于插入可能在表中任何位置上进行, 因此需分析算法的平均复杂度。在长度为 n 的线性表中第 i 个位置上插入一个结点, 令 $E_{is}(n)$ 表示移动结点的期望值(即移动的平均次数), 设 p_i 是在第 i 个位置插入元素的概率, 则在第 i 个位置上插入一个结点的移动次数为 $n-i+1$, 故

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

- 不失一般性, 假设在表中任何位置($1 \leq i \leq n+1$)上插入结点的机会是均等的, 则

$$p_1 = p_2 = p_3 = \dots = p_{n+1} = 1/(n+1)$$

因此, 在等概率插入的情况下,

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

分析“插入”算法的复杂度：

- 也就是说，在顺序表上做插入运算，平均要移动表上一半结点。当表长 n 较大时，算法的效率相当低。虽然 $E_{is}(n)$ 中 n 的系数较小，但就数量级而言，它仍然是线性阶的。因此算法的平均时间复杂度为 $O(n)$ 。

2.2.2 顺序表上实现的基本操作

2、删除

线性表的删除运算是指将表的第 i ($1 \leq i \leq n$) 结点删除，使长度为 n 的线性表：

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

变成长度为 $n-1$ 的线性表

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

如P23 图2.4

2、删除

1.[初值]

获取线性表 L ，删除位置 i

2.[检查参数]

若（删除位置超出线性表长度范围） 则输出错误

3.[删除元素]

获取删除位置 i 的元素 e

将删除位置 i 之后的 L 中的元素全部前移一格

线性表长度减1

4.[算法结束]

例如: **ListDelete_Sq(L, 5, e)**

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

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

for ($++p; p \leq q; ++p$) $*(p-1) = *p;$



分析“删除”算法的复杂度：

- 该算法的时间分析与插入算法相似，结点的移动次数也是由表长 n 和位置 i 决定。
- 若 $i=n$ ，则由于循环变量的初值大于终值，前移语句将不执行，无需移动结点；
- 若 $i=1$ ，则前移语句将循环执行 $n-1$ 次，需移动表中除开始结点外的所有结点。这两种情况下算法的时间复杂度分别为 $O(1)$ 和 $O(n)$ 。

分析“删除”算法的复杂度:

- 删除算法的平均性能分析与插入算法相似。在长度为 n 的线性表中删除一个结点, 令 $E_{de}(n)$ 表示所需移动结点的平均次数, 删除表中第 i 个结点的移动次数为 $n-i$, 故

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

式中, q_i 表示删除表中第 i 个结点的概率。

- 在等概率的假设下, $q_1=q_2=q_3=\dots=q_n=1/n$

由此可得:

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

即在顺序表上做删除运算, 平均要移动表中约一半的结点, 平均时间复杂度也是 $O(n)$ 。

第二章 线性表

2.1 线性表的类型定义

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

2.3.1 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

2.3 线性表的链式表示和实现

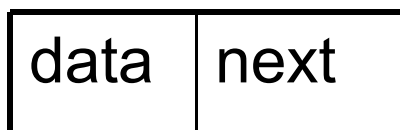
线性表的顺序表示的特点是用物理位置上的邻接关系来表示结点间的逻辑关系，这一特点使我们可以随机存取表中的任一结点，但它也使得插入和删除操作会移动大量的结点。**为避免大量结点的移动**，我们介绍线性表的另一种存储方式，链式存储结构，简称为**链表 (Linked List)**。

2.3.1 线性链表

- 链表是指用一组任意的存储单元来依次存放线性表的结点，这组存储单元即可以是连续的，也可以是不连续的，甚至是零散分布在内存中的任意位置上的。
- 因此，链表中结点的逻辑次序和物理次序不一定相同。

2.3.1 线性链表

- 为了能正确表示结点间的逻辑关系，在存储每个**结点值**的同时，还必须存储指示其后继结点的**地址（或位置）信息**，这个信息称为指针(pointer)或链(link)。这两部分组成了链表中的结点结构：



- 其中：data域是**数据域**，用来存放结点的值。
- next是**指针域（亦称链域）**，用来存放结点的直接后继的地址（或位置）。

2.3.1 线性链表

- 链表正是通过每个结点的链域将线性表的n个结点按其逻辑次序链接在一起。由于上述链表的每一个结点只有一个链域，故将这种链表称为**单链表 (Single Linked)**，或线性链表。
 - 显然，单链表中每个结点的存储地址是存放在其前趋结点next域中，而开始结点无前趋，故**应设头指针head指向开始结点**。
 - 同时，由于终端结点无后继，故**终端结点的指针域为空**，即null（图示中也可用^表示）。
- 例1、线性表: (bat, cat, eat, fat, hat, jat, lat, mat)

单链表示意图如下:

头指针

165

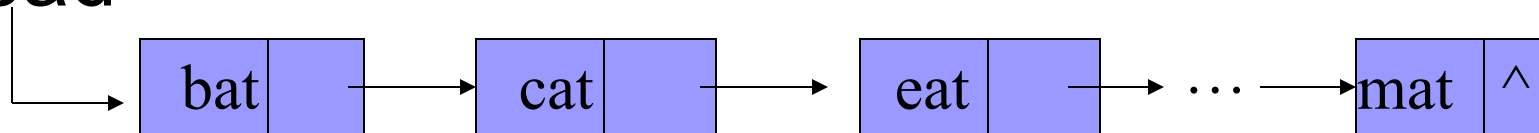
110	...	
	hat	200

130	cat	135
135	eat	170

160	mat	Null
165		
	bat	130
170	fat	110

200	jat	205
205	lat	160

■ head



- 单链表是由表头唯一确定，因此单链表可以用头指针的名字来命名。

□ 例如：若头指针名是**head**，则把链表称为表**head**。

- 用**C**语言描述的单链表如下：

```
typedef struct LNode{
    ElemType data;
    struct LNode *next;
}LNode, *LinkList;
LNode *p;
LinkList head;
```

注意区分指针变量**P**（其值为结点地址）和结点变量***P**这两个不同的概念

2.3.1 线性链表

- p为动态变量，它是通过标准函数生成的，即

p=(LNode*)malloc(sizeof(LNode));

- 函数**malloc**分配了一个类型为**LNode**的结点变量的空间，并将其首地址放入指针变量p中。一旦p所指的结点变量不再需要了，又可通过标准函数

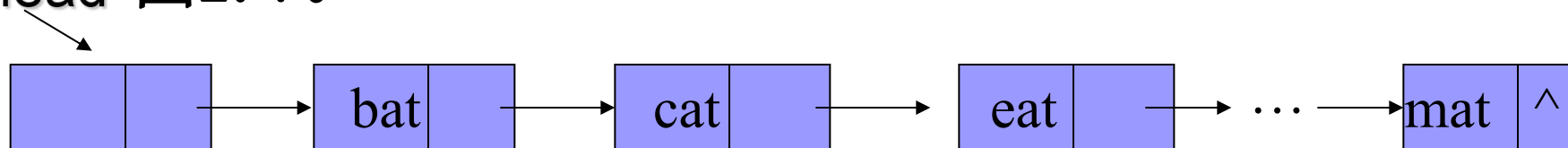
free(p)

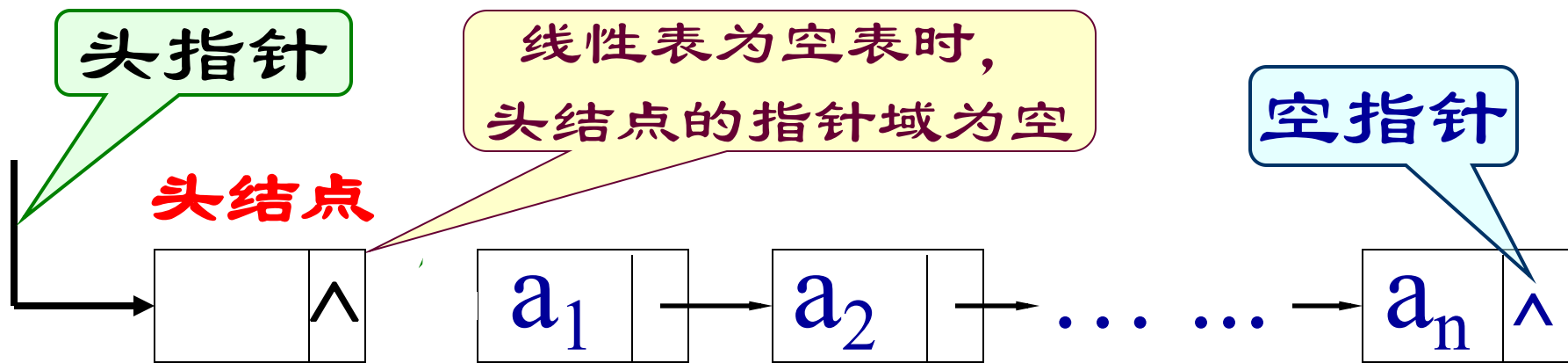
释放所指的结点变量空间。

- 在这样的结构里，第一个结点，有别于其他结点，它的生成与删除都要进行特殊的处理。有时，我们在单链表的第一个结点之前附设一个结点，称之为头结点，那么会带来以下两个优点：

- a、由于开始结点的位置被存放在头结点的指针域中，所以在链表的第一个位置上的操作就和在表的其它位置上的操作一致，无需进行特殊处理；
- b、无论链表是否为空，其头指针是指向头结点的非空指针（空表中头结点的指针域为空），因此空表和非空表的处理也就统一了。

- 头结点的数据域可以不存储任何信息，也可以存放线性表的长度信息。带头结点的单链表见P28页图2.7。





- 以线性表中第一个数据元素 a_1 的存储地址作为线性表的地址，称作线性表的头指针。
- 有时为了操作方便，在第一个结点之前虚加一个“头结点”，以指向头结点的指针为链表的头指针。

2.3.1 线性链表：单链表操作的实现

GetElem_L(L, i, e) // 取第*i*个数据元素

ListInsert_L(&L, i, e) // 插入数据元素

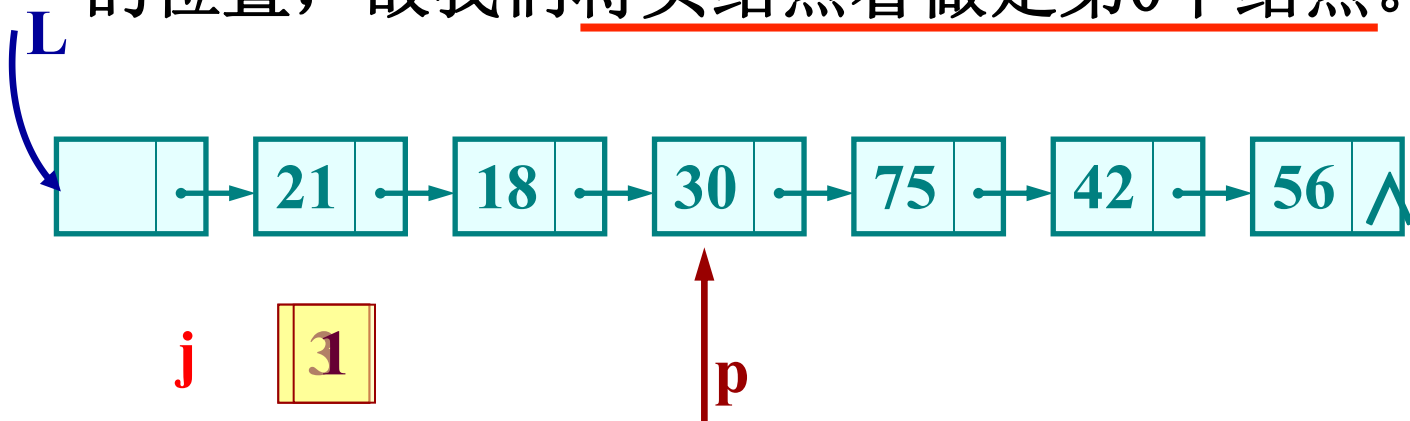
ListDelete_L(&L, i, e) // 删除数据元素

ClearList_L(&L) // 重置线性表为空表

CreateList_L(&L, n) // 生成含 *n* 个数据元素的链表

一、查找运算

- 在链表中，即使知道被访问结点的序号*i*，也不能象顺序表中那样直接按序号*i*访问结点，而只能从链表的头指针出发，顺链域next逐个结点往下搜索，直到搜索到第*i*个结点为止。因此，**链表不是随机存取结构**。
- 设单链表的长度为*n*，要查找表中第*i*个结点，仅当 $1 \leq i \leq n$ 时，*i*的值是合法的。但有时需要找头结点的位置，故我们将头结点看做是第0个结点。



一、查找运算

算法时间复杂度：
 $O(\text{ListLength}(L))$

Status **GetElem_L**(LinkList L , int i, ElemType &e)

{ // L是带头结点的链表的头指针，以e返回第i个元素

p=L->next; j=1; //初始化，p指向第一个结点

妙! **while(p && j<i){** //顺时针向后查找，直到p指向第i个元素或p为空

p=p->next; j++;

}

if (!p || j> i) return ERROR; //第i个元素不存在

e= p->data; //取第i个元素

return OK;

}

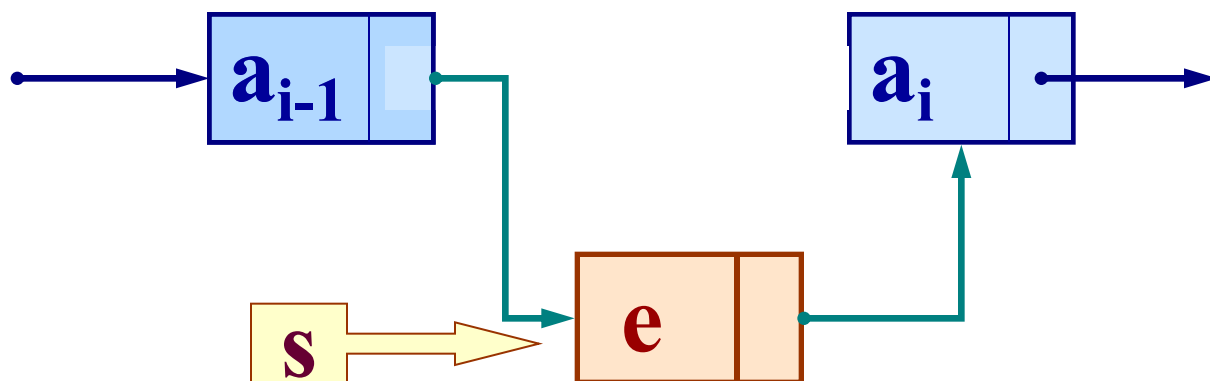
- 算法2.8的基本操作是比较j和i并后移指针，while循环体中的语句频度与被查元素在表中位置有关，若 $1 \leq i \leq n$ ，则频度为i-1，否则频度为n，因此该操作的时间复杂度为 $O(n)$ 。

二、 插入运算

- 插入运算是将值为 e 的新结点插入到表的第 i 个结点的位置上，即插入到 a_{i-1} 与 a_i 之间。

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

- 因此，我们必须首先找到 a_{i-1} 的存储位置，然后生成一个数据域为 e 的新结点 s ，并令新结点 s 的指针域指向结点 a_i ，结点 a_{i-1} 的指针域指向新结点 s 。从而实现三个结点 a_{i-1} ， s 和 a_i 之间的逻辑关系的变化。



二、插入运算

Status ListInsert _L(LinkList &L, int i, ElemType e)

```
{ // L为带头结点的单链表的头指针，本算法在链表中第i个结点之前插入新的元素e
    p=L; j=0;
    while (p && j<i-1) {p=p->next; ++j;} // 寻找第i-1个结点
    if (!p||j>i-1) return ERROR; // i大于表长或者小于1
    s=(LinkList)malloc(sizeof(LNode)); //生成新结点
    s->data=e;
    s->next=p->next;
    p->next=s;
    return OK;
}
```

算法时间复杂度：
 $O(\text{ListLength}(L))$

二、 插入运算

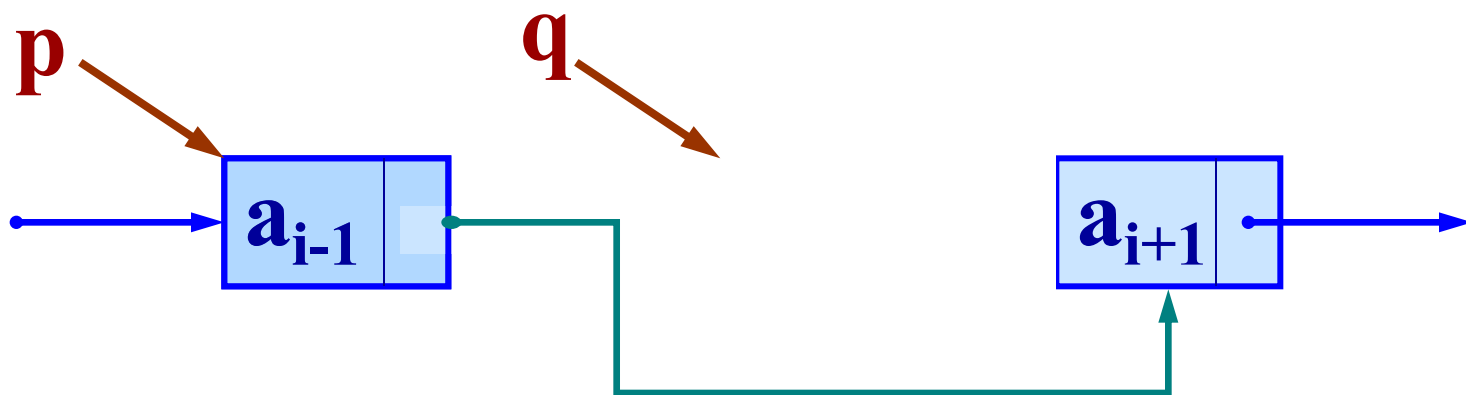
- 设链表的长度为 n ，合法的插入位置是 $1 \leq i \leq n+1$ 。注意当 $i=1$ 时，找到的是头结点，当 $i=n+1$ 时，找到的是结点 a_n 。算法的时间主要耗费在查找操作while语句上，故时间复杂度为 $O(n)$ 。

三、删除运算

- 删除运算是将表的第 i 个结点删去，即：

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

- 因为在单链表中结点 a_i 的存储地址是在其直接前趋结点 a_{i-1} 的指针域next中，所以我们必须首先找到 a_{i-1} 的存储位置 p 。然后令 $p \rightarrow \text{next}$ 指向 a_i 的直接后继结点，即把 a_i 从链上摘下。最后释放结点 a_i 的空间，将其归还给“存储池”。



三、删除运算

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) { //寻找第i个结点, 并令p指向其前趋
p=p->next; ++j;

}

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;

}

算法时间复杂度:
 $O(\text{ListLength}(L))$

三、删除运算

- 设单链表的长度为 n ，则删去第 i 个结点仅当 $1 \leq i \leq n$ 时是合法的。注意， p 是指向待删结点的前一个结点。当 $i=n+1$ 时，虽然被删结点不存在，但其前趋结点却存在，它是终端结点。因此被删结点的直接前趋 $*p$ 存在并不意味着被删结点就一定存在，仅当 $(p \rightarrow \text{next} \neq \text{NULL})$ 时，才能确定待删结点存在。
- 显然，此算法的时间复杂度也是 $O(n)$ 。
- 从上面的讨论可以看出，链表上实现插入和删除运算，无须移动结点，仅需修改指针。

四、建立单链表

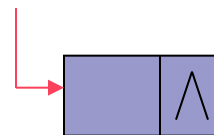
- 动态地建立单链表的常用方法有如下几种：

1、头插法建表

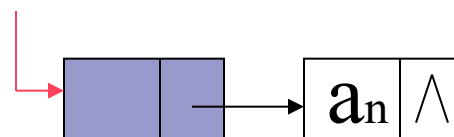
该方法从一个空表开始，重复读入数据，生成新结点，将读入数据存放到新结点的数据域中，然后将新结点插入到当前链表的表头上，直到读入结束标志为止。

创建n个元素的线性链表操作步骤：

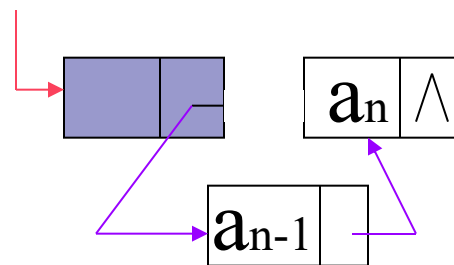
一、建立一个“空表”；



二、输入数据元素 a_n ，
建立结点并插入；



三、输入数据元素 a_{n-1} ，
建立结点并插入；



四、依次类推，直至输入 a_1 为止。

Status CreateList_L(LinkList &L, int n)

{ // 逆序输入n个数据元素，建立带头结点的单链表

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

L->next = NULL; // 先建立一个带头结点的单链表

for (i=n; i>0; --i) {

p=(LinkList)malloc(sizeof(LNode)); //生成新结点

scanf(&p->data); // 输入元素值

p->next=L->next; L ->next=p; // 插入

}

} //CreateList_L

**算法时间复杂度：
O(ListLength(L))**

四、建立单链表

2、尾插法建表（无头结点）

头插法建立链表虽然算法简单，但生成的链表中结点的次序和输入的顺序相反。若希望二者次序一致，可采用尾插法建表。该方法是将新结点插入到当前链表的表尾上，为此必须增加一个尾指针r，使其始终指向当前链表的尾结点。

```
Status CreateList_L(LinkList &L )
```

```
{
```

```
    char ch;  LNode *p, *r;
```

```
    L=NULL;  r=NULL;
```

```
    while( (ch=getchar( )) != '\n'){
```

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

```
        p->data=ch;
```

```
        if (L==NULL) { L=p;  r=p; }  //生成第一个结点
```

```
        else{ r->next=p; r=p; }  //生成其他结点
```

```
    }
```

```
    if (r!=NULL)           //在有生成结点的情况下，为r->next赋空值
```

```
        r->next=NULL;
```

```
    return OK;
```

```
}
```

五、有序链表的合并

- 时间复杂度和算法2.7相同：
 $O(\text{ListLength}(La) + \text{ListLength}(Lb))$
- 空间复杂度不同，不需要另建新表的结点空间。

见书P31，思路与算法2.2同。

Void MergeList_L(LinkList &La, LinkList &Lb, LinkList &Lc)

{ //已知单链线性表La和Lb的元素按值非递减排列。

//归并La和Lb得到新的单链线性表Lc，Lc的元素也按值非递减排列。

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) {

If (pa->data<=pb->data){

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

}

Else {pc->next=pb; pc=pb; pb=pb->next;}

}

Pc->next=pa?pa:pb;

//插入剩余段

Free(Lb);

//释放Lb的头结点

}//MergeList_L

2.3.1 线性链表：静态单链表

- 有时候，也可以用一维数组来描述链表，这种链表称为静态链表。它的形式定义为

```
#define MAXSIZE 1000
typedef struct{
    ElemType data; //数据
    int cur; //指示下一项的数组索引
}component, SLinkList[MAXSIZE];
```

- 如图2.10，数组的第一个分量可以看成头结点，假设S为SLinkList型变量，则 $i=S[i].cur$ 相当于指针的后移。
- 定位函数见算法2.13，类似可以写出插入和删除的操作。所不同的是，用户必须自己实现malloc和free两个函数。
- 为了辨明数组中哪些分量未被使用，解决的办法是建立一个备用结点链表，每当进行插入操作时从备用链表上取得第一个结点作为新结点，在删除时将被删除的结点链接到备用链表上。例子见书P33


```
void InitSpace_SL(SlinkList &space){
```

```
// space为备用链表, space[0].cur为头指针
```

```
    for (i=0; i<MAXSIZE-1; ++i)
```

```
        space[i].cur=i+1;
```

```
    space[MAXSIZE-1].cur=0;
```

```
}
```

```
int Malloc_SL(SlinkList &space){
```

```
//若备用链表非空, 则返回分配的结点头下标, 否则返回0
```

```
    i=space[0].cur;
```

```
    if (space[0].cur) space[0].cur=space[i].cur;
```

```
    return i;
```

```
}
```

```
void Free_SL(SlinkList &space, int k){
```

```
//将下标为k的空闲结点回收至备用链表的头部
```

```
    space[k].cur=space[0].cur; space[0].cur=k;
```

```
}
```

静态单链表举例：

■ 例子2-3 ： 计算 $(A-B) \cup (B-A)$

先由输入建立集合A的静态链表S，然后在输入B元素的同时查找S表，若存在和B相同的元素，则从S表中删除之，否则将此元素插入S表。

1. [初值]

输入集合A元素建立静态链表S

2. [计算 $(A-B) \cup (B-A)$]

对于B中的每一个元素x做如下操作：

若 $(x \text{ 不属于 } S)$ 则 将x插入到S的末尾

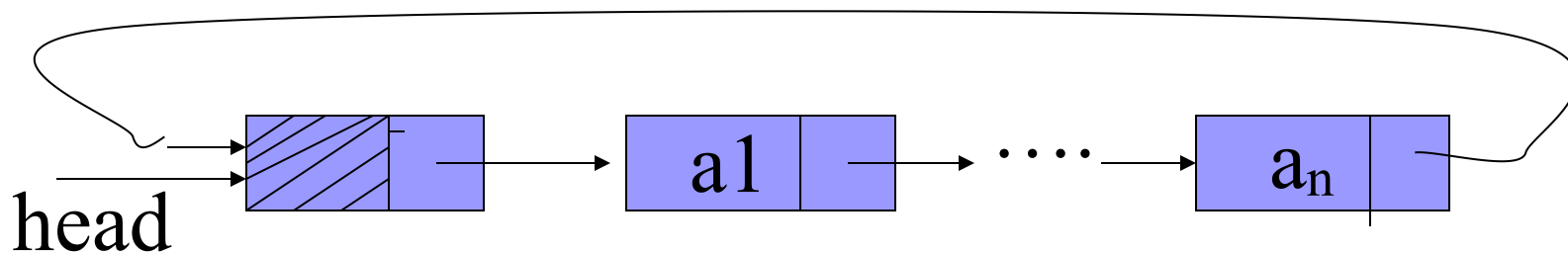
若 $(x \text{ 属于 } S)$ 则 将x从S中删除

3. [算法结束]

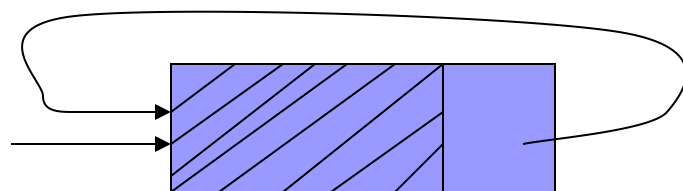
算法复杂度为： $O(m*n)$ ，例子见P34图2.11

2.3.2 循环链表

- 循环链表是一种头尾相接的链表。其特点是无须增加存储量，仅对表的链接方式稍作改变，即可使得表处理更加方便灵活。
- **单循环链表：**在单链表中，将终端结点的指针域 NULL 改为指向表头结点或开始结点，就得到了单链形式的循环链表，并简单称为单循环链表。
- 为了使空表和非空表的处理一致，循环链表中也可设置一个头结点。这样，空循环链表仅有一个自成循环的头结点表示。如下图所示：



(1) 非空表



(2) 空表

- 在用头指针表示的单链表中，找开始结点 a_1 的时间是 $O(1)$ ，然而要找到终端结点 a_n ，则需从头指针开始遍历整个链表，其时间是 $O(n)$

2.3.2 循环链表

- **头指针表示的单循环链表**对表的操作不够方便
 - 因为表的操作常常是在表的首尾位置上进行
- **改用尾指针rear来表示单循环链表**
 - 查找开始结点 a_1 和终端结点 a_n 都会方便，它们的存储位置分别是 $(rear \rightarrow next) \rightarrow next$ 和 $rear$ ，显然，查找时间都是 $O(1)$ 。
- 由于循环链表中没有NULL指针，故涉及遍历操作时，其**终止条件**就不再像非循环链表那样判断 p 或 $p \rightarrow next$ 是否为空，而是判断它们是否等于某一指定指针，如头指针或尾指针等。

2.3.2 循环链表

例：在链表上实现将两个线性表 $(a_1, a_2, a_3, \dots, a_n)$ 和 $(b_1, b_2, b_3, \dots, b_n)$ 链接成一个线性表的运算。

```
LinkedList Connect(LinkedList reara, LinkedList rearb)
{
    LinkedList p=reara—>next;
    reara—>next=(rearb—>next)—>next
    free(rearb—>next);
    rearb—>next=p;
    return (rearb);
}
```

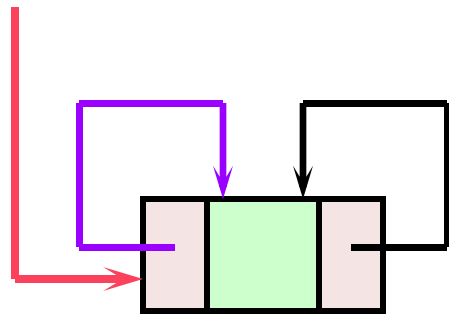
2.3.3 双链表

- **双向链表 (Double linked list)**: 在单链表的每个结点里再增加一个指向其直接前趋的指针域**prior**。这样就形成的链表中有两个方向不同的链，故称为双向链表。形式描述为：

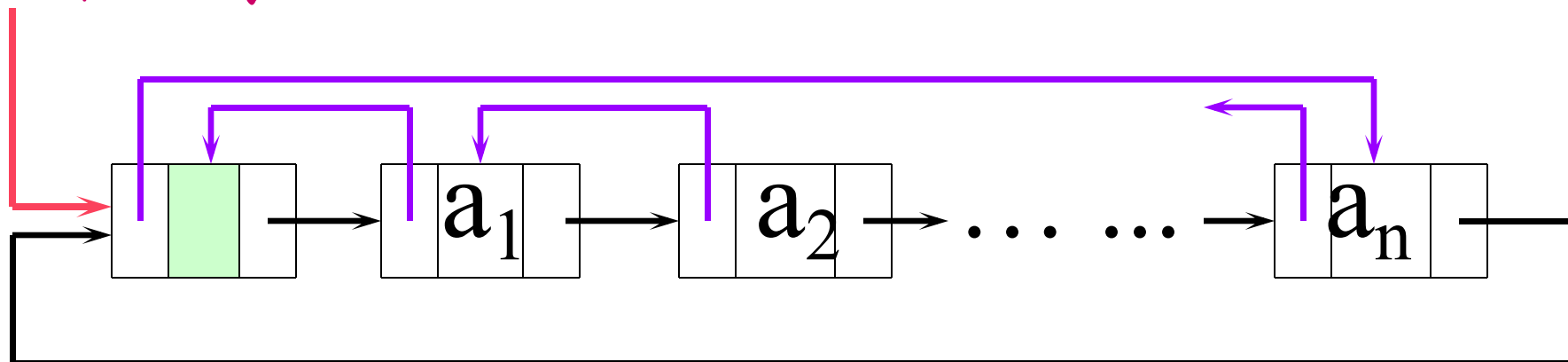
```
typedef struct DuLNode{  
    ElemType data;  
    struct DuLNode *prior, *next;  
}DuLNode, *DuLinkList;
```

双向循环链表

空表



非空表



2.3.3 双链表

- 和单链表类似，双链表一般也是由头指针唯一确定的，增加头指针也能使双链表上的某些运算变得方便。将头结点和尾结点链接起来也能构成循环链表，并称之为双向链表。
- 设指针p指向某一结点，则双向链表结构的对称性可用下式描述：

$$(p \rightarrow \text{prior}) \rightarrow \text{next} = (p \rightarrow \text{next}) \rightarrow \text{prior} = p$$

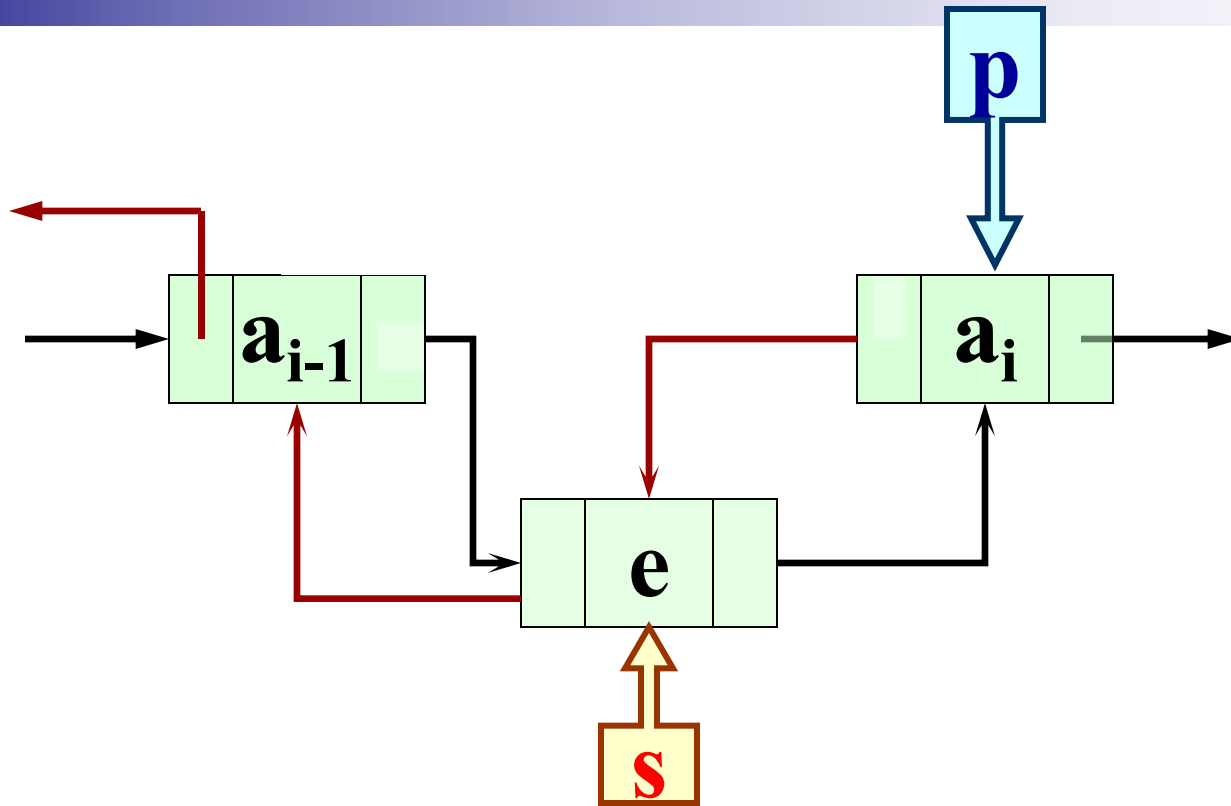
- 即结点*p的存储位置既存放在其前趋结点*(p→prior)的直接后继指针域中
- 也存放在它的后继结点*(p→next)的直接前趋指针域中。

2.3.3 双链表

双向链表的操作特点：

- “查询” 和单链表相同。
- “插入” 和 “删除” 时需要同时修改两个方向上的指针。

插入



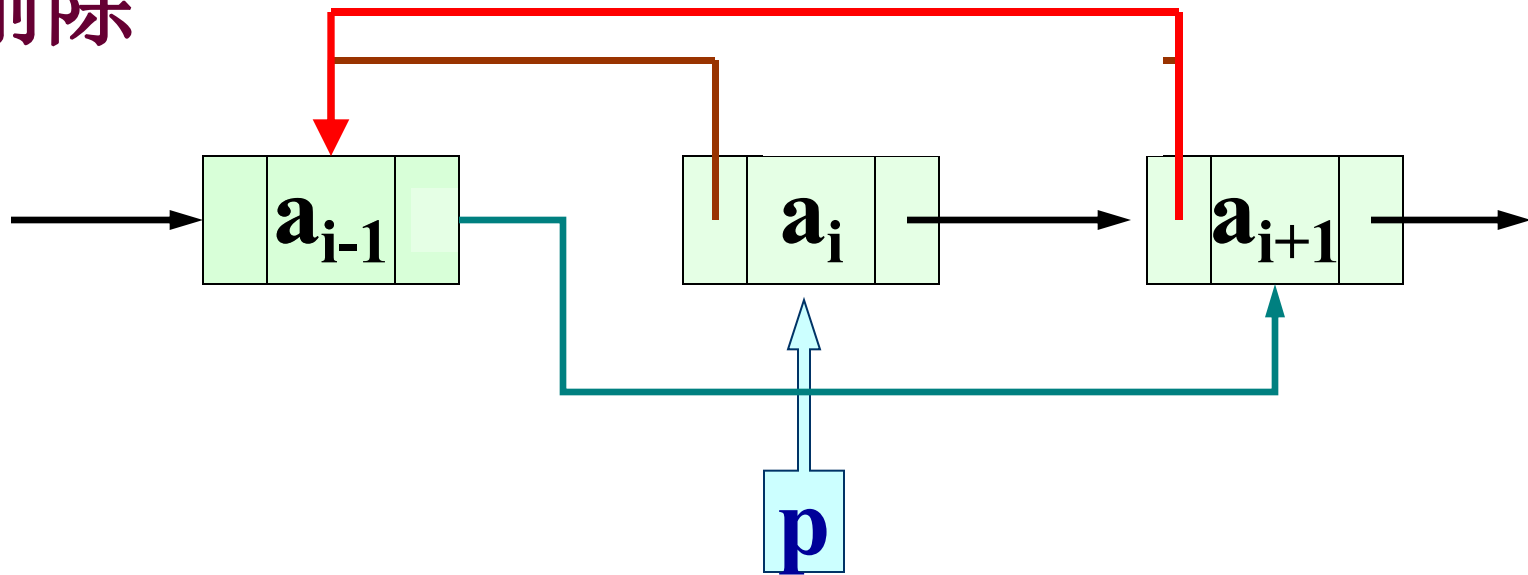
$s \rightarrow \text{prior} = p \rightarrow \text{prior};$ $p \rightarrow \text{prior} \rightarrow \text{next} = s;$

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

双向链表的插入操作算法如下：

```
Status ListInsert_DuL(DuLinkList &L, int i, ElemType e)
{    //在带头结点的双链循环线性表L中第i个位置之前插入元素e
    if (!p=GetElemp_DuL(L,i)))    //在L中确定插入位置
        return ERROR
    if ( !(s=(DuLinkList)malloc(sizeof(DuLNode)))) return ERROR;
    s—>data=e;
    s—>prior=p—>prior;
    p—>prior—>next=s;
    s—>next=p;
    p—>prior=s;
    return OK;
}
```

删除



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

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

删除p指针所指的结点，详见P37算法2.19

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

注意：与单链表的插入和删除操作不同的是，在双链表中插入和删除必须同时修改两个方向上的指针。

小结:

■ 链表的优点:

- 空间的合理利用;
- 插入、删除时不需移动

——线性表的首选存储结构

■ 链表的缺点:

- 求线性表的长度时不如顺序存储结构
- 数据元素在线性表中的“位序”的概念已淡化

■ 为此, 从实际应用出发重新定义一个带头结点的线性链表类型 P37

第二章 线性表

2.1 线性表的类型定义

2.2 线性表的顺序表示和实现

2.3 线性表的链式表示和实现

2.3.1 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

2.4 一元多项式的表示及相加

■ 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：

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

■ 一般情况下的一元多项式

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

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

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:

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

2.4 一元多项式的表示及相加

■ 抽象数据类型一元多项式的定义如下： P40

ADT Polynomial {

数据对象:

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

TermSet 中的每个元素包含一个表示系数的实数和表示指数的整数 }

数据关系:

$R1 = \{ \langle a_{i-1}, a_i \rangle \mid a_{i-1}, a_i \in D, i=2,\dots,n \text{ 且 } a_{i-1} \text{ 中的指数值} < a_i \text{ 中的指数值} \}$

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。

2.4 一元多项式的表示及相加

基本操作:

PrintPolyn (&P)

初始条件: 一元多项式 P 已存在。

操作结果: 打印输出一元多项式 P 。

PolynLength(P)

初始条件: 一元多项式 P 已存在。

操作结果: 返回一元多项式 P 中的项数。

AddPolyn (&Pa, &Pb)

初始条件: 一元多项式 Pa 和 Pb 已存在。

操作结果: 完成多项式相加运算, 即:

$Pa = Pa + Pb$, 并销毁一元多项式 Pb 。

SubtractPolyn (&Pa, &Pb)

... ..

} ADT Polynomial

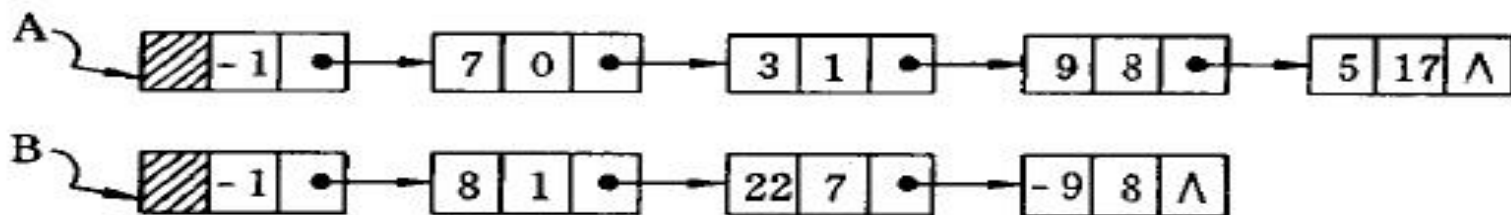
2.4 一元多项式的表示及相加

表达式

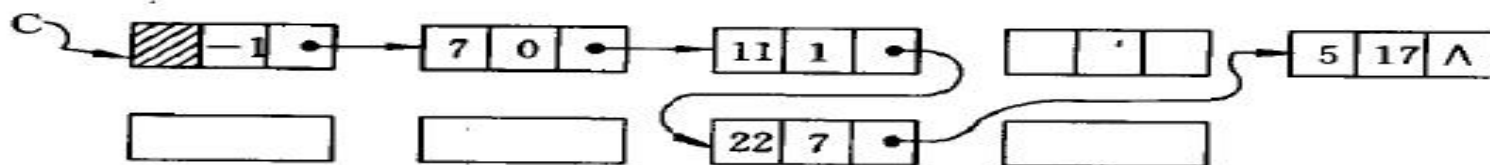
$$A(x) = 7 + 3x + 9x^8 + 5x^{17}$$

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

相加前的链表状态



相加后的链表状态



1.[初值]

获取线性表**LA**和**LB**

2.[多项式相加]

对于线性表**LA**和**LB**，都从其第一个元素开始做如下操作直到其中一个线性表元素全部遍历完毕：

比较**LA**中的元素**a**与**LB**中的元素**b**的多项式指数

(1) 若 (**LA**的元素**a**的指数 < **LB**的元素**b**的指数)

则 选择**LA**中的下一个元素**a**

(2) 若 (**LA**的元素**a**的指数 = **LB**的元素**b**的指数)

则

将元素**a**的多项式系数加上元素**b**的多项式系数

若 (系数和为0) 则 从**LA**中删除元素**a**

选择**LA**的下一个元素**a**和**LB**的下一个元素**b**

(3) 若 (**LA**的元素**a**的指数 > **LB**的元素**b**的指数)

则 将元素**b**插入到**LA**的元素**a**之前，并选择**LB**的下一个元素**b**



3.[补充剩下的元素]

若 (**LB**还有剩余元素) 则 将**LB**的剩余元素全部插入到**LA**末尾

4.[算法结束]

本章小结

1. 了解线性表的逻辑结构特性是数据元素之间存在着线性关系，在计算机中表示这种关系的两类不同的存储结构是顺序存储结构和链式存储结构。用前者表示的线性表简称为顺序表，用后者表示的线性表简称为链表。
2. 熟练掌握这两类存储结构的描述方法，以及线性表的各种基本操作的实现。
3. 能够从时间和空间复杂度的角度综合比较线性表两种存储结构的不同特点及其适用场合。

- 
- 
- 作业
 - 第一次: 2.21,2.29
 - 第二次: 书上的填空题自己做
2.30,2.38,2.39