

第2章 线性表

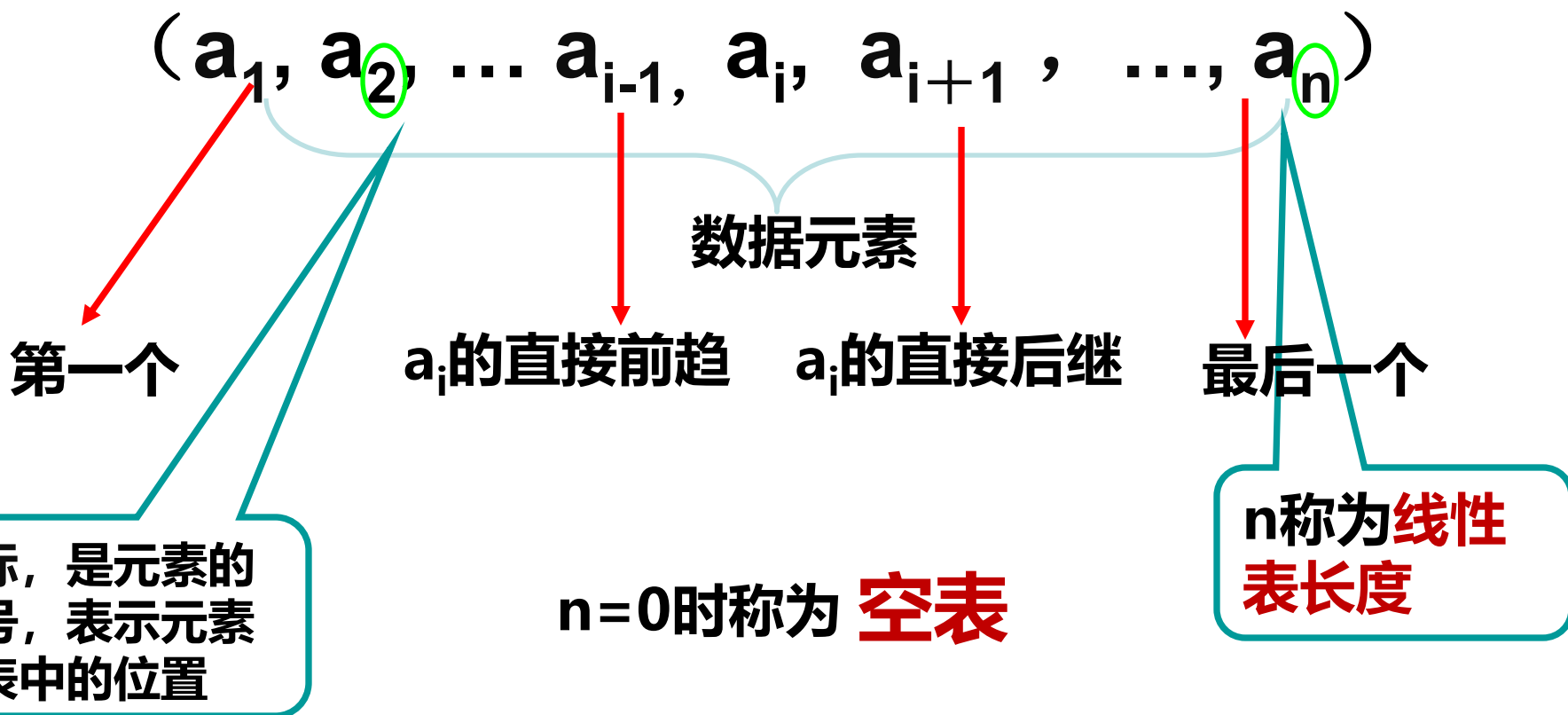
教学目标

1. 了解线性表的定义和特点
2. 掌握顺序表的定义及查找、插入和删除等基本操作
3. 掌握链表的定义及查找、插入和删除等基本操作
4. 能够从时间和空间复杂度的角度比较两种存储结构的不同特点及其适用场合



2.1 线性表的定义和特点

线性表的定义： 线性表是 $n(n \geq 0)$ 个相同类型数据元素 a_1, a_2, \dots, a_n 构成的有限序列。



例1 分析26个英文字母组成的英文表

(A, B, C, D, , Z)

数据元素都是字母; 元素间关系是线性

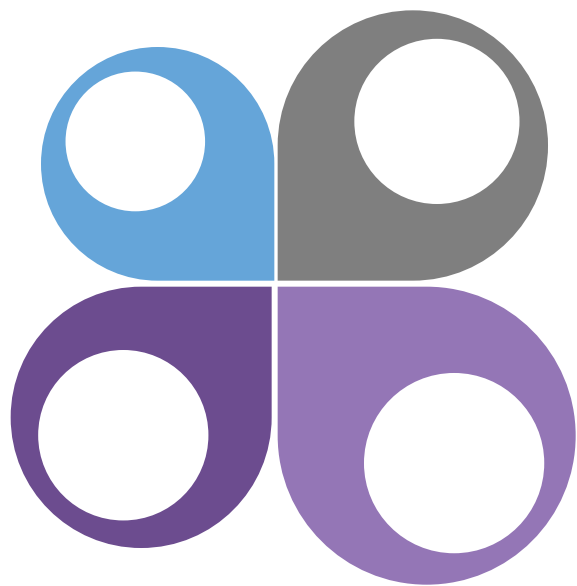
例2 分析学生情况登记表

学号	姓名	性别	年龄	班级
041810205	于春梅	女	18	04级计算机1班
041810260	何仕鹏	男	20	04级计算机2班
041810284	王 爽	女	19	04级计算机3班
041810360	王亚武	男	18	04级计算机4班
:	:	:	:	:

数据元素都是记录; 元素间关系是线性

同一线性表中的元素必定具有相同特性

2.2 案例引入



案例1：一元多项式的运算



案例2：稀疏多项式的运算



案例3：图书信息管理系统

案例1：一元多项式的运算

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

线性表 $P = (p_0, p_1, p_2, \dots, p_n)$

$$P(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$$

指数 (下标 <i>i</i>)	0	1	2	3	4
系数 $p[i]$	10	5	-4	3	2

数组表示

(每一项的指数
 i 隐含在其系数
 p_i 的序号中)

案例1：一元多项式的运算

$$R_n(x) = P_n(x) + Q_m(x)$$

线性表 $R = (p_0 + q_0, p_1 + q_1, p_2 + q_2, \dots, p_m + q_m, p_{m+1}, \dots, p_n)$

稀疏多项式

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



案例2：稀疏多项式的运算

多项式非零项的数组表示

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

下标i	0	1	2	3
系数	7	3	9	5
指数	0	1	8	17

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

下标i	0	1	2
系数	8	22	-9
指数	1	7	8

$$P_n(x) = p_1 x^{e_1} + p_2 x^{e_2} + \dots + p_m x^{e_m}$$

线性表 $P = ((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$



案例2：稀疏多项式的运算

创建一个新数组c



一个多项式已遍历**完毕**时，将另一个剩余项依次复制到c中即可

分别从头遍历比较a和b的每一项

- **指数相同**，对应系数相加，若其和不为零，则在c中增加一个新项。
- **指数不相同**，则将指数较小的项复制到c中。

案例2：稀疏多项式的运算

顺序存储结构存在问题

- ✓存储空间分配不灵活
- ✓运算的空间复杂度高

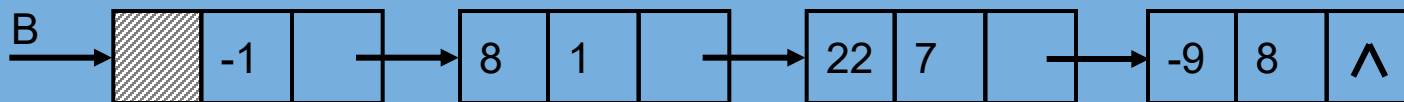
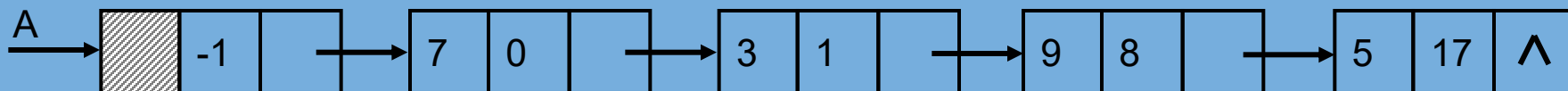


链式存储结构

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

$$B_8(x) = 8x + 22x^7 - 9x^8$$

pa



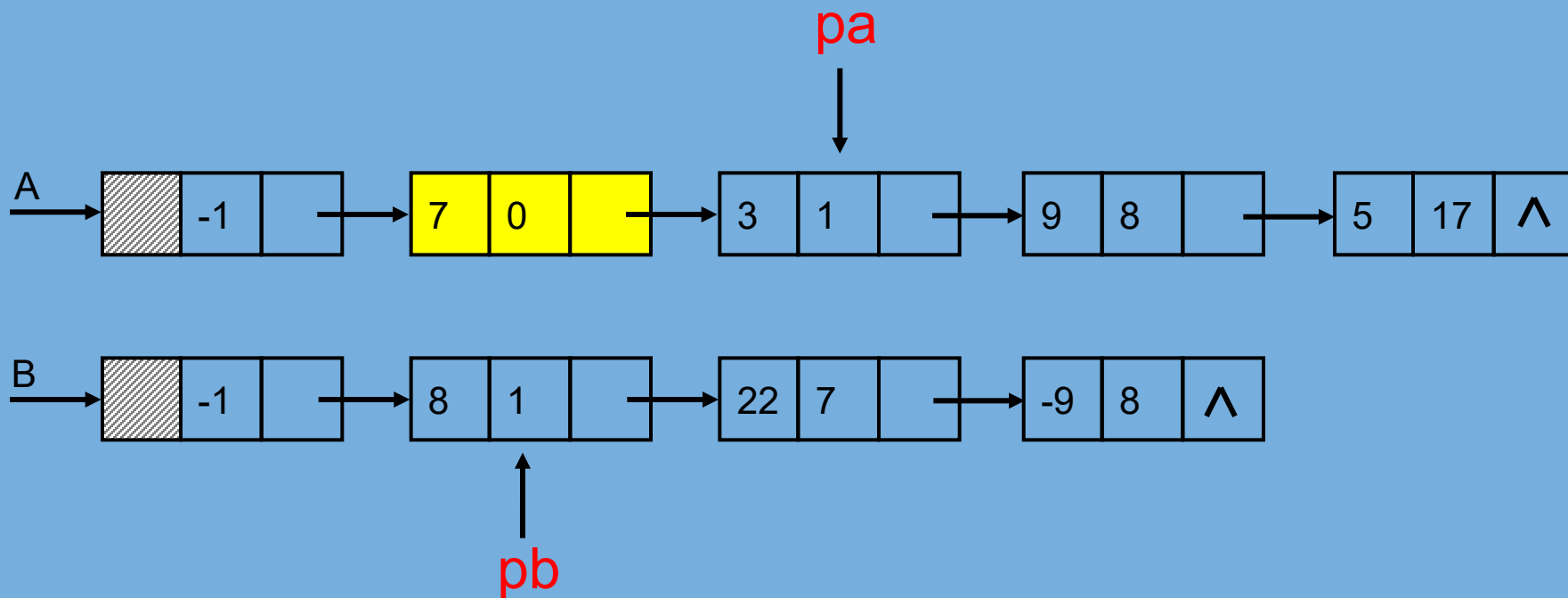
pb



多项式相加

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

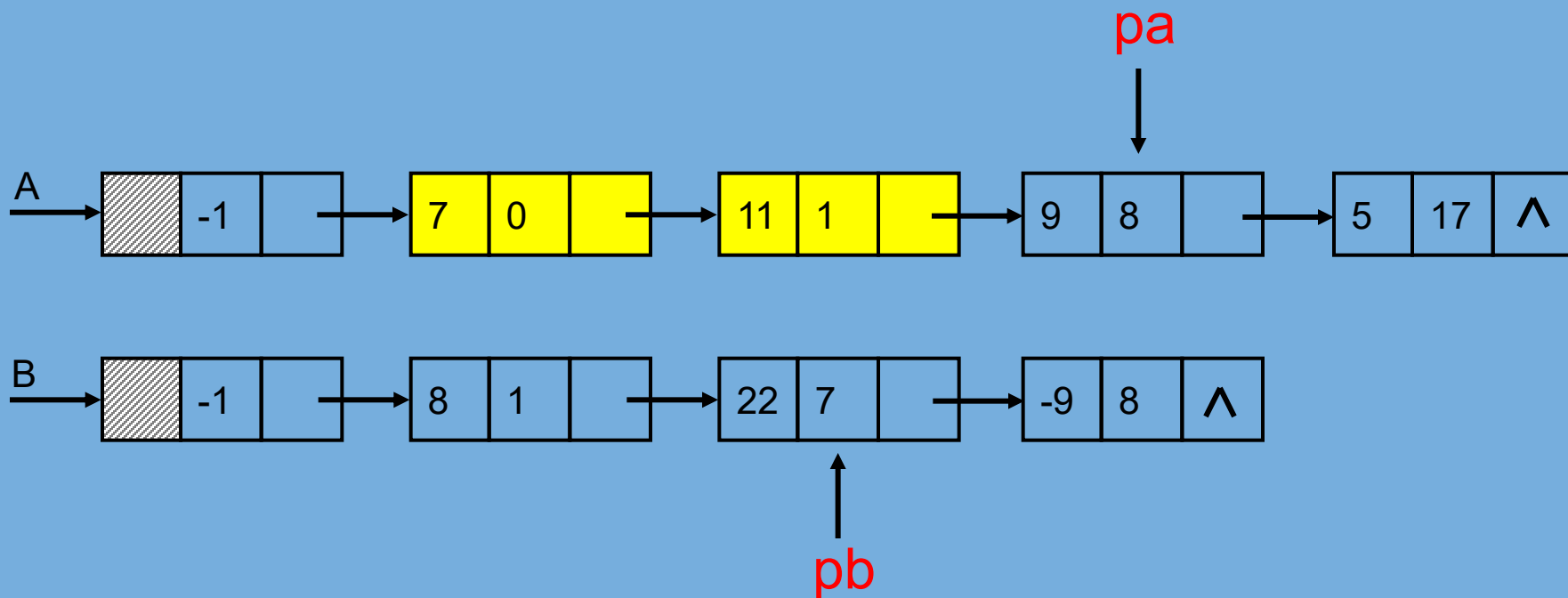
$$B_8(x)=8x+22x^7-9x^8$$



多项式相加

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

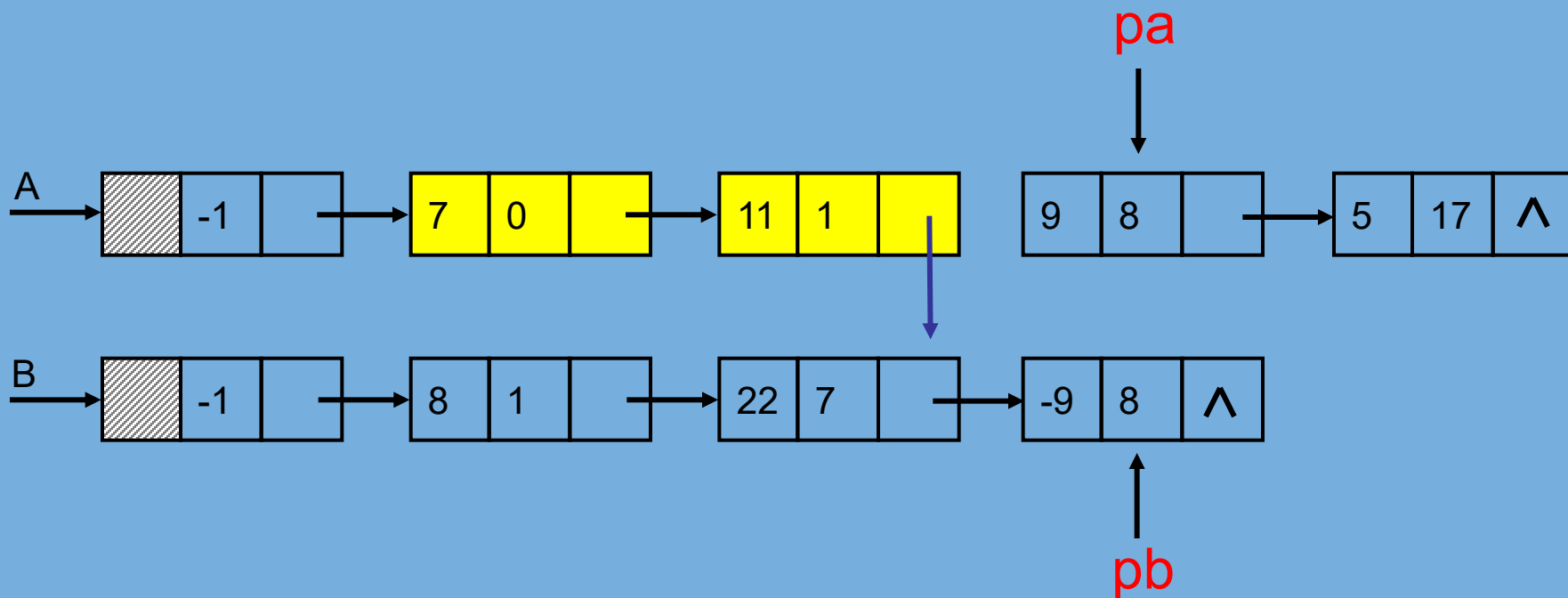
$$B_8(x)=8x+22x^7-9x^8$$



多项式相加

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

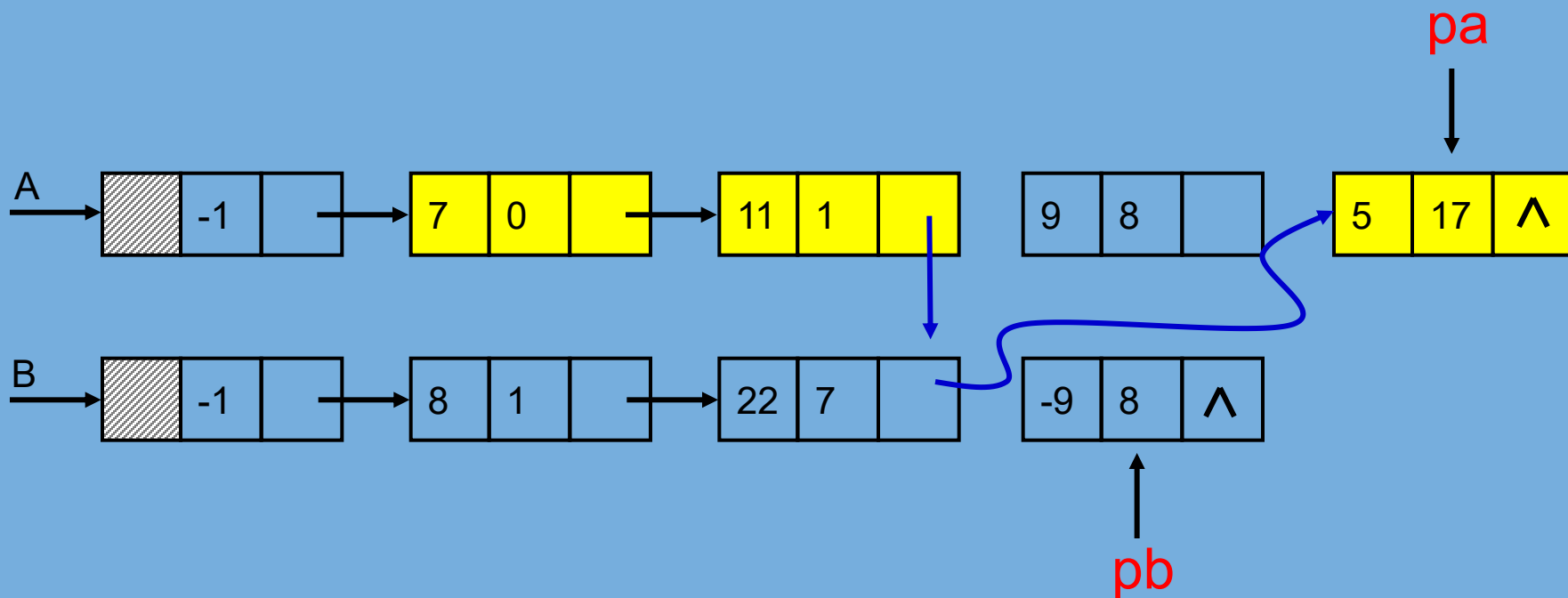
$$B_8(x) = 8x + 22x^7 - 9x^8$$



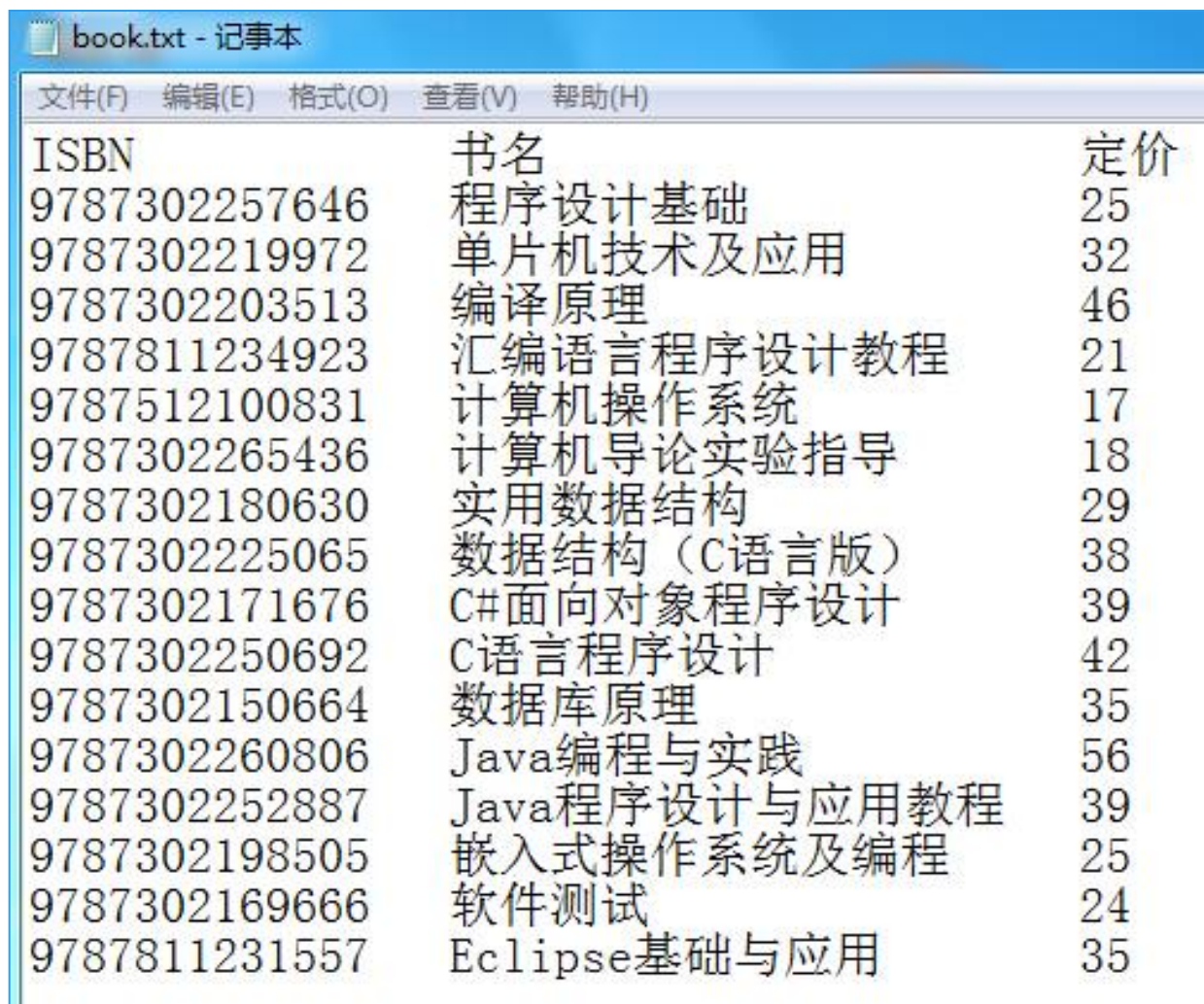
多项式相加

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

$$B_8(x) = 8x + 22x^7 - 9x^8$$



案例3：图书信息管理系统



The screenshot shows a Notepad window with the title 'book.txt - 记事本'. The menu bar includes '文件(F)', '编辑(E)', '格式(O)', '查看(V)', and '帮助(H)'. The text content is a table with three columns: ISBN, 书名 (Book Title), and 定价 (Price). The table contains 15 rows of book data.

ISBN	书名	定价
9787302257646	程序设计基础	25
9787302219972	单片机技术及应用	32
9787302203513	编译原理	46
9787811234923	汇编语言程序设计教程	21
9787512100831	计算机操作系统	17
9787302265436	计算机导论实验指导	18
9787302180630	实用数据结构	29
9787302225065	数据结构 (C语言版)	38
9787302171676	C#面向对象程序设计	39
9787302250692	C语言程序设计	42
9787302150664	数据库原理	35
9787302260806	Java编程与实践	56
9787302252887	Java程序设计与应用教程	39
9787302198505	嵌入式操作系统及编程	25
9787302169666	软件测试	24
9787811231557	Eclipse基础与应用	35

- (1) 查找
- (2) 插入
- (3) 删除
- (4) 修改
- (5) 排序
- (6) 计数

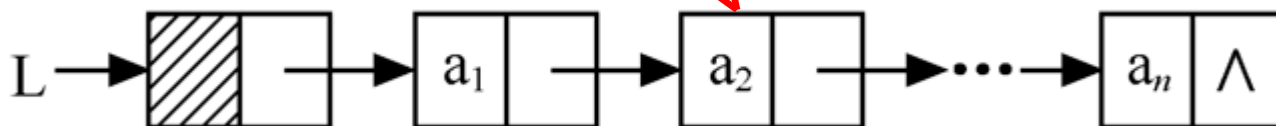
案例3：图书信息管理系统

图书顺序表

elem[0]	elem[1]	elem[2]	...	elem[length-1]	空闲区	
a ₁	a ₂	a ₃	...	a _{length}		



图书链表



2.3 线性表的类型定义

ADT List {

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

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

基本操作:

InitList(&L)

DestoryList(&L)

ClearList(&L)

ListEmpty(L)

ListLength(L)

GetElem(L,i,&e)

LocateElem(L,e)

PriorElem(L,cur_e,&pre_e)

NextElem(L,cur_e,&next_e)

ListInsert(&L,i,e)

ListDelete(&L,i)

TraverseList (L)

}ADT List

利用抽象数据类型所定义的操作可实现更复杂的操作

例：两个线性表的合并

```
Void union( list &La , List Lb) {  
    // 将所有在线性表Lb中但不在La中的数据元素插入到La中  
    La_len = ListLength( La );  
    Lb_len = ListLength( Lb );    //求线性表的长度  
    for ( i = 1 ; i <= Lb_len ; i++) {  
        GetElem( Lb , i , e );      // 取Lb中第i个数据元素赋给e  
        if ( !LocateElem( La , e)) ListInsert( La, ++La_len, e );  
        // La中不存在与e相同的数据元素，插入e  
    }  
} // Union O(La_len*Lb_len)
```

2.4 线性表的顺序表示和实现



线性表的顺序表示：用一组地址连续的存储单元依次存储线性表的各个数据元素。又称为顺序存储结构或顺序映像。称这种存储结构的线性表为**顺序表**。

线性表中逻辑上相邻的元素 a_i 和 a_{i+1} 的存储位置 $LOC(a_i)$ 和 $LOC(a_{i+1})$ 也是相邻(连)的。即：

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

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

整个线性表的所占空间为： $n * m$

可以使用数组实现。

线性表: $(a_1, a_2, \dots, a_i, \dots, a_n)$

顺序存储

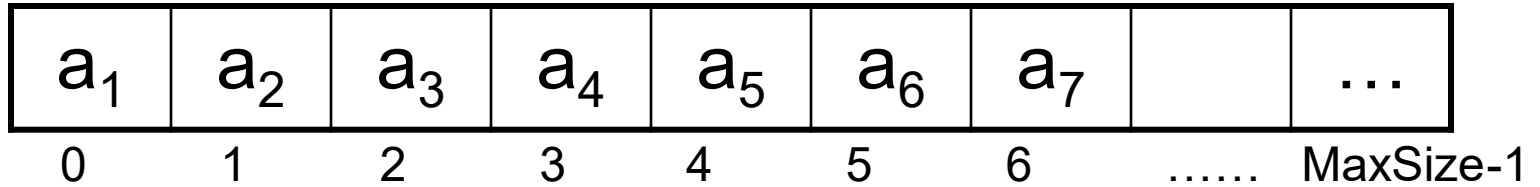
存储地址	存储内容
L_0	a_1
L_0+m	a_2

$L_0+(i-1)*m$	a_i

$L_0+(n-1)*m$	a_n

$$\text{Loc}(a_i) = L_0 + (i-1)*m$$

顺序存储结构的线性表称作**顺序表**，示意图如下：



用固定大小数组实现：

```
typedef struct
{
    ElemType elem[MaxSize];    //数组名为elem
    int length;    //当前元素的个数，最后一个下标为length-1
} SqList;
```

例： `#define MaxSize 10`
`typedef int ElemType;`

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

`void main()`
`{`
 `SqList L;`
 `L.elem[0] = 15;`
 `L.elem[1] = 5;`
 `L.elem[2] = 22;`
 `L.length = 3;`
 `.....`
`}`

用动态数组实现：

```
#define MAXSIZE 100    //最大长度

typedef struct {
    ElemType *elem;    //动态存储空间的首地址，即数组名
    int length;        //当前元素的个数，最后一个下标为length-1
} SqList;
```

本课程接下来均采用动态数组

例

```
typedef int ElemType;
typedef struct {
    ElemType *elem;
    int length;
} SqList;
void main() {
    SqList L;
    L.elem = (ElemType*)
        malloc(MAXSIZE * sizeof(ElemType));
    //L.elem=new ElemType[MAXSIZE];
    L.elem[0] = 15;   L.elem[1] = 5;   L.elem[2] = 22;
    L.length = 3;

    .....
    free(L.elem); //delete []L.elem;
}
```


图书表的顺序存储结构类型定义

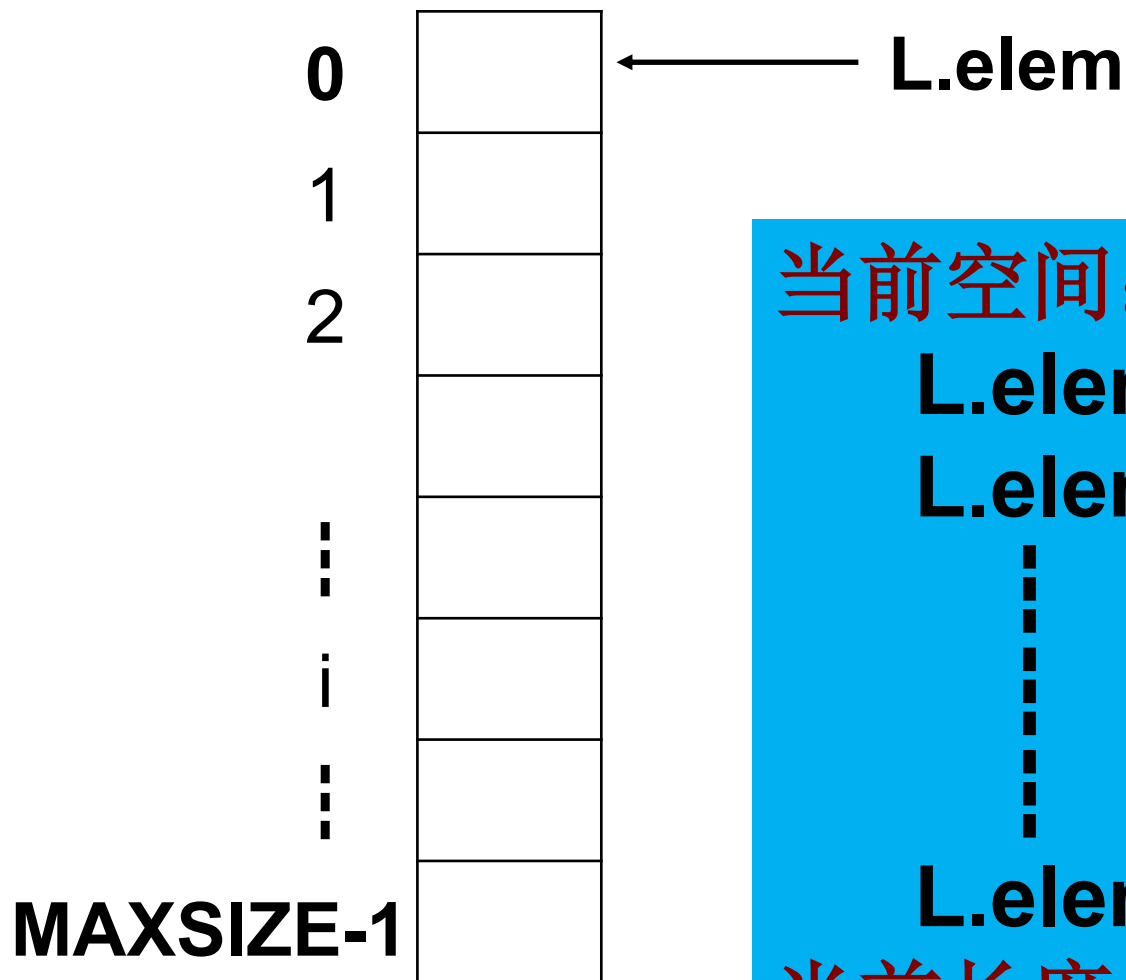
```
#define MAXSIZE 10000 //图书表可能达到的最大长度
typedef struct          //图书信息定义
{
    char no[20];         //图书ISBN
    char name[50];       //图书名字
    float price;         //图书价格
}Book;
typedef struct
{
    Book *elem;           //存储空间的基地址
    int length;           //图书表中当前图书个数
}SqList;                 //图书表的顺序存储结构类型为SqList
```

顺序表基本操作的实现

1. 初始化线性表L (参数用引用)

```
Status InitList(SqList &L){ //构造一个空的顺序表L
    //为顺序表分配空间
    L.elem= (ElemType*)
                malloc(MAXSIZE * sizeof(ElemType));
    // L.elem=new ElemType[MAXSIZE];
    if(!L.elem) exit(OVERFLOW);    //存储分配失败
    L.length=0;                    //空表长度为0
    return OK;
} //InitList
```

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



当前空间:

L.elem[0]

L.elem[1]

⋮

L.elem[$\text{MAXSIZE}-1$]

当前长度:

L.length=0

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

2. 销毁线性表L

```
void DestroyList(SqList &L)  
{  
    if (L.elem) free(L.elem); //释放存储空间  
    // if (L.elem) delete[] L.elem;  
    L.elem=NULL;  
    L.length=0;  
}
```

3. 清空线性表L

```
void ClearList(SqList &L)  
{  
    L.length=0; //将线性表的长度置为0  
}
```

4.取值（获取线性表第i个元素）

获取线性表L中的某个位置(i)数据元素的内容

```
Status GetElem(SqList L,int i,ElemType &e)
{
    if (i<1||i>L.length) return ERROR;
    //判断i值是否合理，若不合理，返回ERROR
    e=L.elem[i-1]; // 下标为i-1的单元存储着第i个数据
    return OK;
}
```



随机存取
 $O(1)$

5. 在线性表L中查找值为e的数据元素

```
int LocateELem(SqList L, ElemType e)
{ //在顺序表L中顺序查找值为e的元素，返回其序号
  for (i=0; i< L.length; i++)
    if (L.elem[i]==e) return i+1;
  return 0; //查找失败
}
```

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

查找算法时间效率分析:

查找成功时: 最坏情况与平均情况均是 $O(n)$

查找不成功时: $O(n)$

查找算法时间效率分析

【定义】平均查找长度(**Average Search Length, ASL**)
在查找成功时，查找给定值与数据元素比较次数的期望值。该值越小说明查找算法越优。

设 p_i 为各项的查找概率， C_i 为查找第 i 个元素时比较的的次数，则在长度为 n 的顺序表中，**ASL**为：

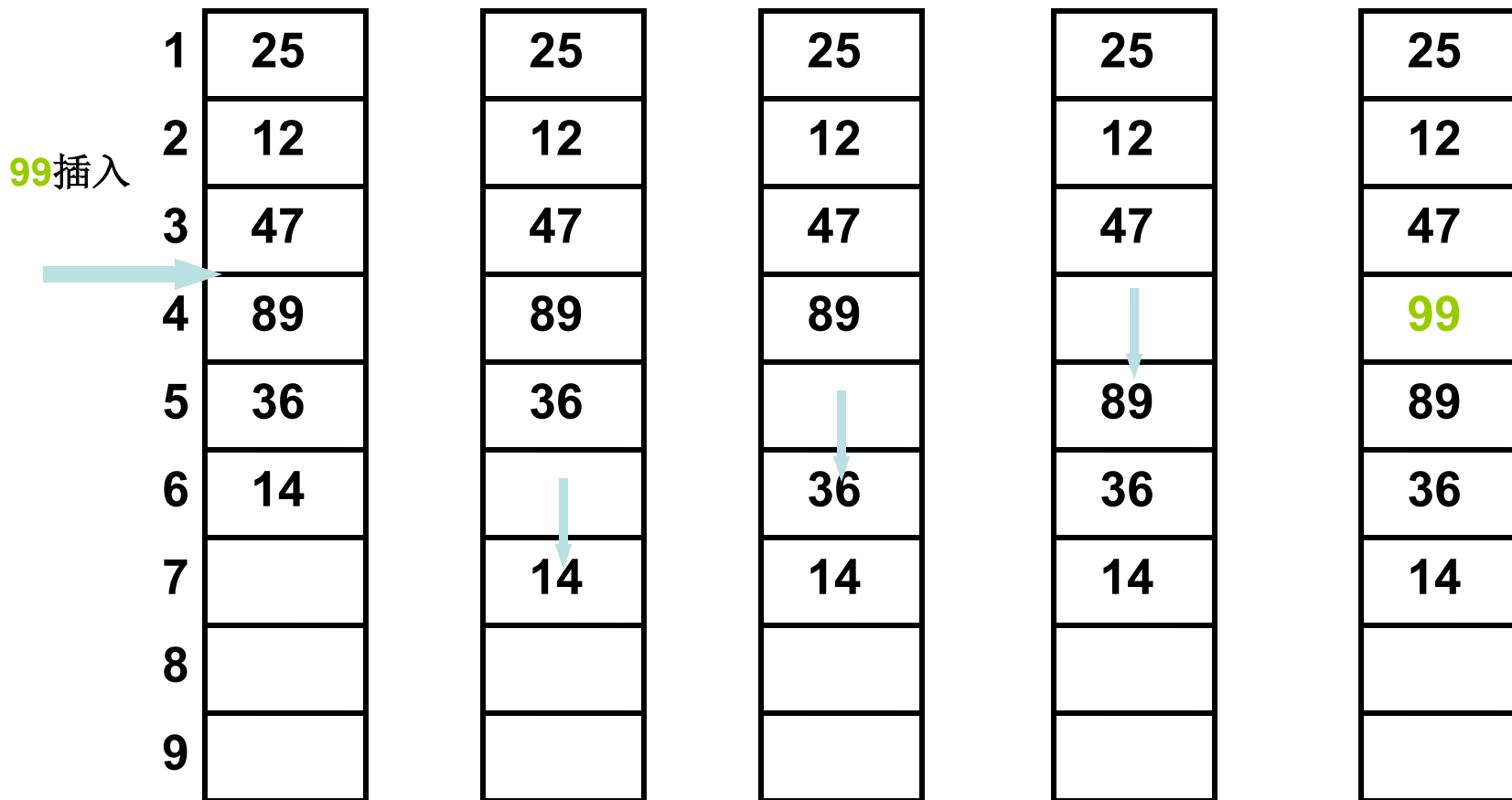
$$ASL = \sum_{i=1}^n p_i \times c_i$$

若每个元素的查找概率相等，即 $p_i=1/n$ ，则：

$$\begin{aligned} ASL &= \frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} (1 + 2 + \cdots + n) \\ &= \frac{1}{n} * \frac{(1+n) * n}{2} = \frac{1+n}{2} \end{aligned}$$

可见顺序查找的时间复杂度为 **$O(n)$** 。

6.在线性表L中第i个数据元素之前插入数据元素e



插第 4 个结点之前，移动 $6-4+1$ 次

插在第 i 个结点之前，移动 $n-i+1$ 次

【算法思想】

- (1) 判断**插入位置 i** 是否合法 ($1 \leq i \leq n+1$) 。
- (2) 判断顺序表的存储空间是否已满, **若满, 则出错。**
- (3) 将线性表第 n 至第 i 位的元素依次**向后移动一个位置**, 空出第 i 个位置。
- (4) 将要插入的新元素 **e 放入第 i 个位置。**
- (5) **表长加1**, 插入成功返回OK。

在线性表L中第i个数据元素之前插入数据元素e

```
Status ListInsert(SqList &L, int i , ElemType e) {  
    if (i<1 || i>L.length+1) return ERROR;           //i值不合法  
    if ( L.length ==MAXSIZE ) return ERROR; //空间已满  
    for (j=L.length-1; j>=i-1; j--)  
        L.elem[j+1]=L.elem[j];    //插入位置及之后的元素后移  
  
    L.elem[i-1]=e;                //将新元素e放入第i个位置  
    ++L.length;                   //表长增1  
    return OK;  
}
```

【算法分析】

按位置插入时： 共有 $n+1$ 个位置可插入（ $1 \sim n+1$ ），
只需考虑移动次数

最好情况： 0次 -----表尾

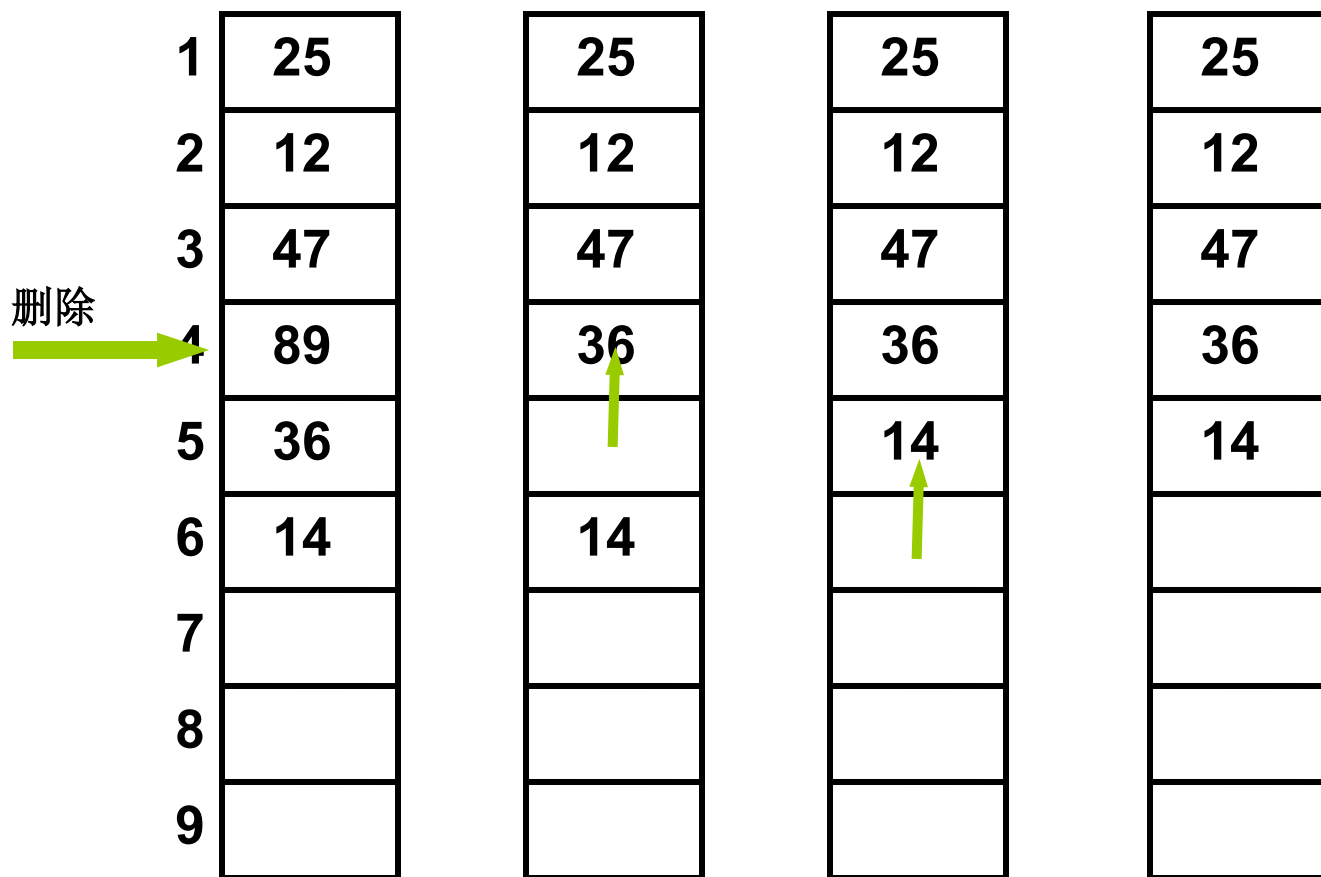
最坏情况： n 次 -----表头

平均情况：平均数据移动次数 E 在
任何位置插入概率相等时：

$$\begin{aligned} E &= \frac{1}{n+1} \sum_{i=1}^{n+1} (n-i+1) = \frac{1}{n+1} (n + \dots + 1 + 0) \\ &= \frac{1}{(n+1)} \frac{n(n+1)}{2} = \frac{n}{2} \end{aligned}$$

平均情况与最坏情况： $O(n)$

7. 将线性表L中第i个数据元素删除



删除第 4 个结点，移动 6-4 次

删除第 i 个结点，移动 $n-i$ 次

【算法思想】

- (1) 判断**删除位置i 是否合法**（合法值为 $1 \leq i \leq n$ ）。
- (2) 将第i+1至第n 位的元素依次**向前移动一个位置**。
- (3) **表长减1**，删除成功返回OK。

将线性表L中第i个数据元素删除（可用e返回其值）

```
Status ListDelete(SqList &L, int i, ElemType &e) {  
    if ((i<1)|| (i>L.length)) return ERROR;    //i值不合法  
    e=L.elem[i-1];    //被删元素值赋给e，带回  
    for (j=i; j<=L.length-1; j++)  
        L.elem[j-1]=L.elem[j];    //被删除元素之后的元素前移  
    --L.length;    //表长减1  
    return OK;  
}
```

【算法分析】

按位置删除时:

共有 n 个位置可删除 ($1 \sim n$) , 考虑移动次数:

最好情况: $O(1)$ -----表尾

最坏情况: $O(n)$ -----表头

平均情况: 平均移动次数 E 在各位置删除
概率相等 ($1/n$) 时:

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

平均情况与最坏情况: $O(n)$

顺序表（顺序存储结构）的特点

- (1) 逻辑关系上相邻的两个元素在物理位置上也相邻。即线性表的**逻辑结构与存储结构一致**
- (2) 在访问线性表时，可以快速地计算出任何一个数据元素的存储地址。因此可以粗略地认为，**访问每个元素所花时间相等**

这种存取元素的方法被称为**随机存取法**

顺序表的优缺点

优点:

- ✓ **存储密度大**（即空间单元利用率高）
（结点本身所占存储量/结点结构所占存储量）
- ✓ 可以**随机存取**表中任一元素

缺点:

- ✓ **时间**: 在插入、删除某一元素时，需要移动大量元素
- ✓ **空间**: 浪费存储空间，属于静态存储形式，数据元素的个数不能自由扩充

为克服这一缺点



链式存储

2.5 线性表的链式表示和实现

特点：1、链式存储结构不要求逻辑上相邻的元素在物理位置上也相邻，可以用任意的存储单元存储数据元素，元素之间的逻辑关系通过指针指向后继结点来表示。

2、插入或删除非常方便，但不能随机存取，只能沿着链一个个搜索，即**顺序存取**。

概念：结点、数据域、指针域、头指针、**头结点****

分类：根据链表指针的设置，可分为：

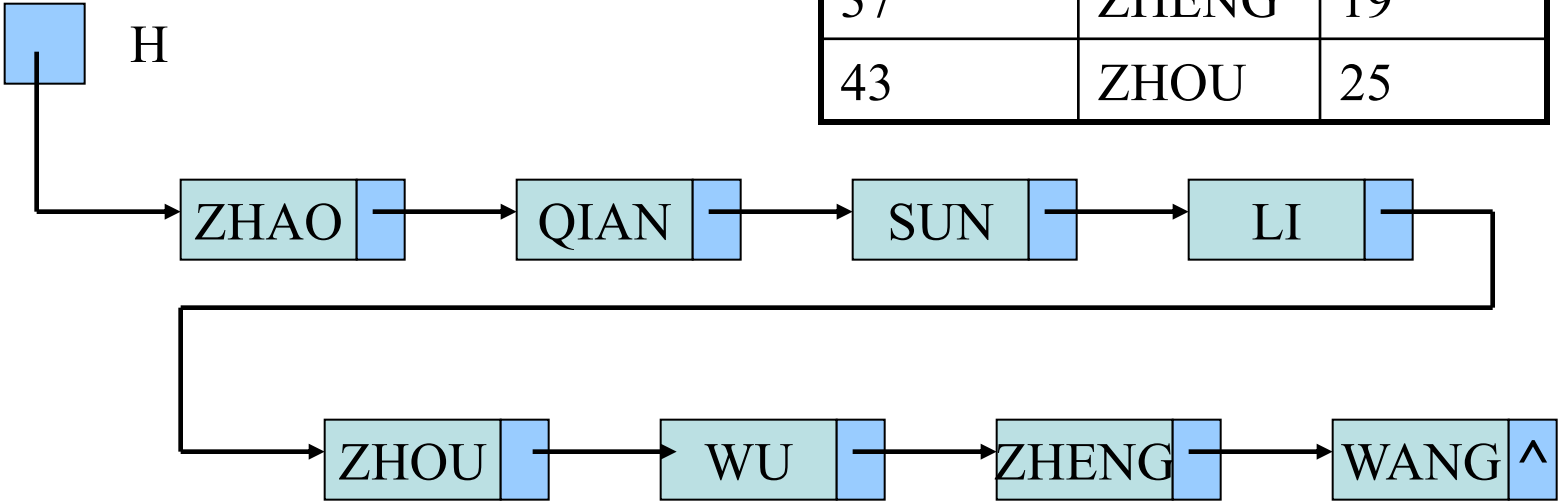
- 1、单链表（线性链表）
- 2、循环链表
- 3、双向链表

例:线性表(ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG)的
链式存储结构

存储地址	数据域	指针域
1	LI	43
7	QIAN	13
13	SUN	1
19	WANG	NULL
25	WU	37
31	ZHAO	7
37	ZHENG	19
43	ZHOU	25

头指针H

31



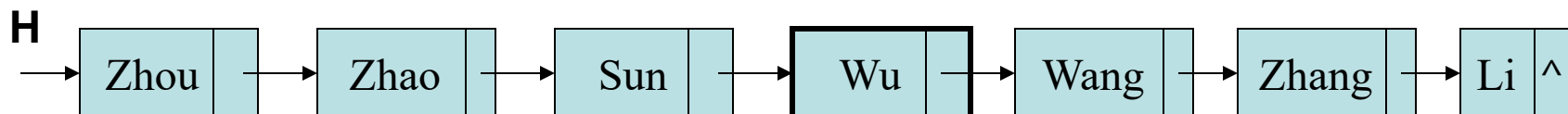
1.单链表：构成链表的结点只有一个指针域

data	next
------	------

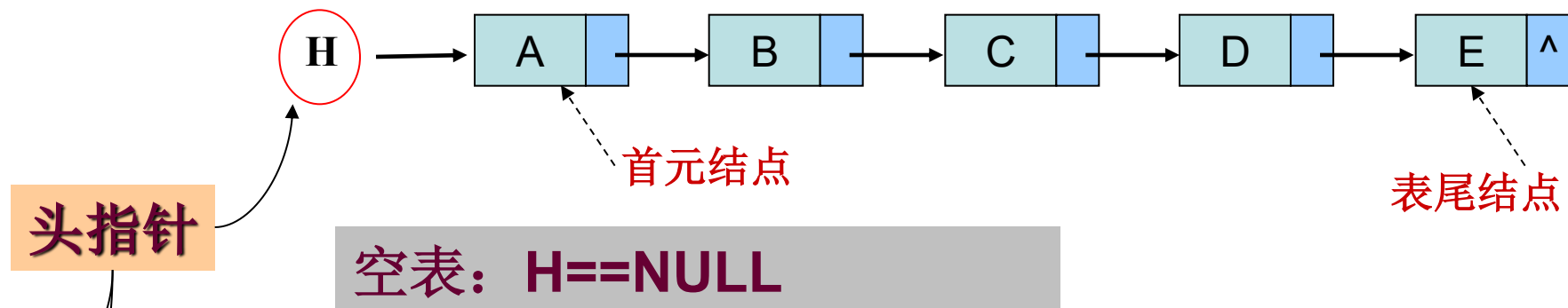
头指针H

31

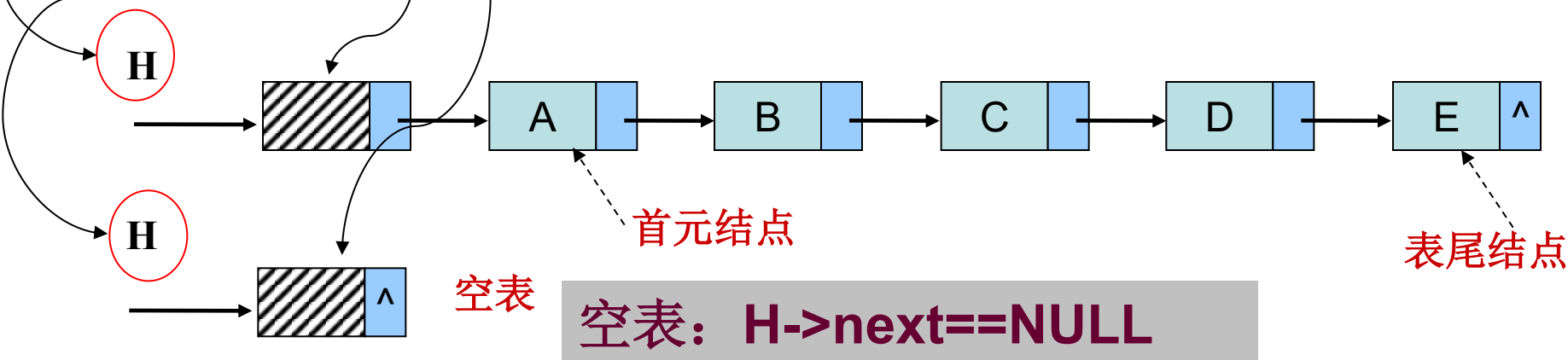
存储地址	数据域	指针域
1	ZHANG	13
7	WANG	1
13	LI	null
19	ZHAO	37
25	WU	7
31	ZHOU	19
37	SUN	25



不带头结点的单链表



带头结点的单链表



设置头结点的目的: 简化链表操作的实现。

讨论. 在链表中设置**头结点**有什么好处?

1 便于**首元结点**的处理

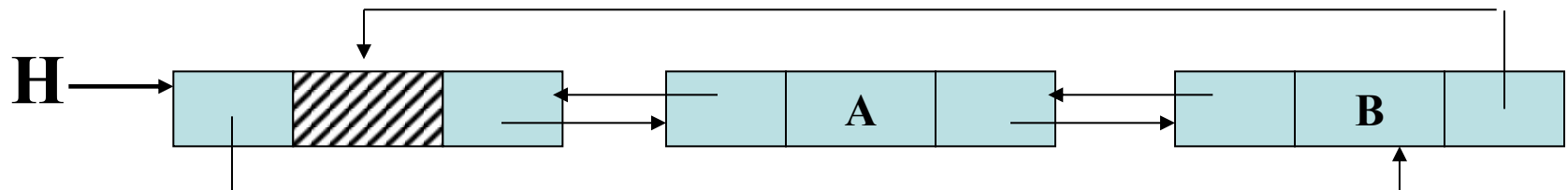
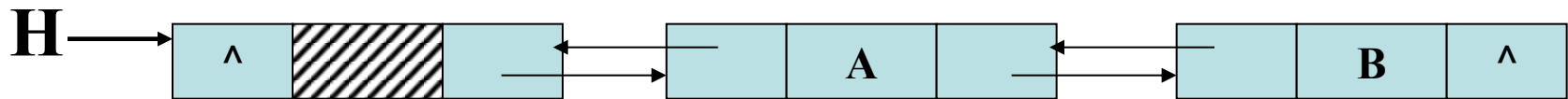
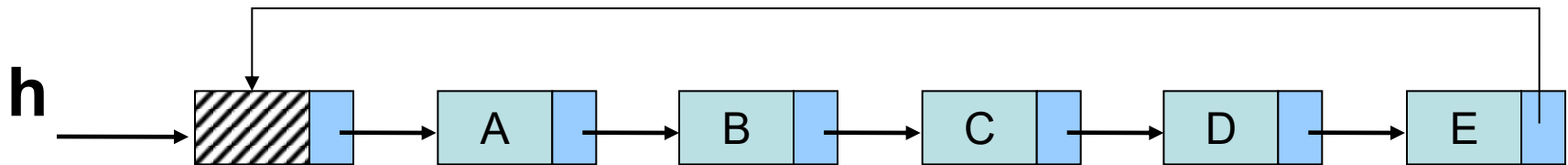
首元结点的地址保存在头结点的指针域中, 所以在链表的第一个位置上的操作和其它位置一致, 无须进行特殊处理;

2 便于**空表和非空表**的统一处理

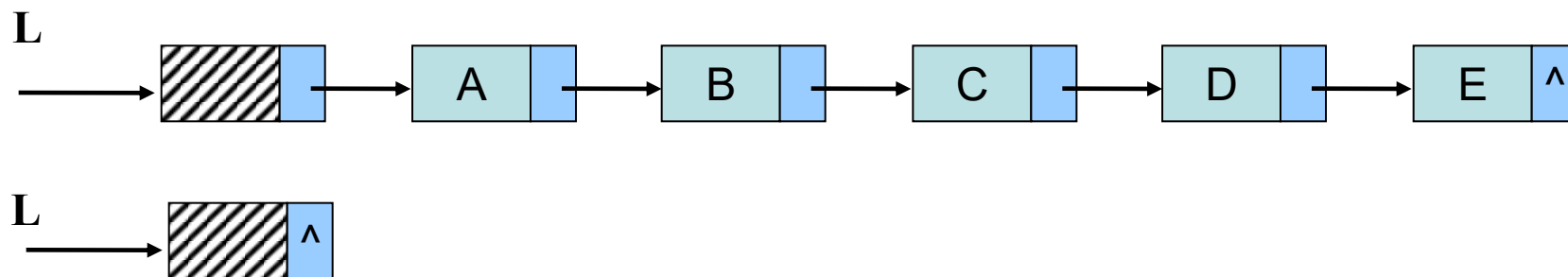
无论链表是否为空, 头指针都是指向头结点的非空指针, 因此空表和非空表的处理也就统一了。即头指针H始终不为空。

2. 循环链表、双向链表示例图：

- 首尾相接的链表称为**循环链表**
- 有两个指针域的链表，称为**双向链表**



2.5.1 单链表的定义和表示



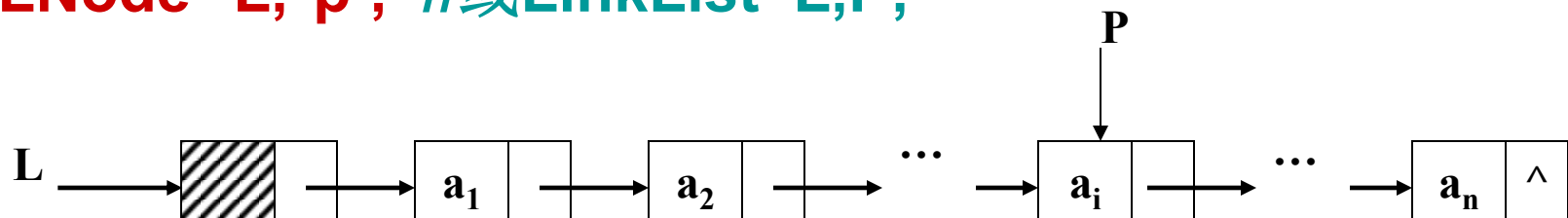
单链表存储结构定义如下：

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

LNode *p;
↕ 一样
LinkList p;

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

LNode *L,*p ; //或LinkList L,P;

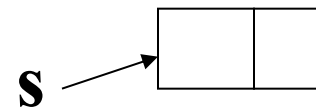


则有: $L \rightarrow next \rightarrow data == a_1$;

$P \rightarrow data == a_i$ $P \rightarrow next \rightarrow data == a_{i+1}$

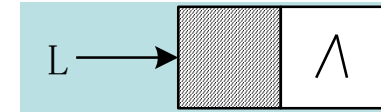
新建结点: $LNode *s$;

$s = (LNode *)malloc(sizeof(LNode))$



2.5.2 单链表基本操作的实现

1.初始化(构造一个空表)



【算法思想】

- (1) 生成新结点作头结点，用头指针 L 指向头结点。
- (2) 头结点的指针域置空。

【算法描述】

```
Status InitList(LinkList &L){  
    L=(LinkList)malloc(sizeof(LNode));  
    //L=new Lnode;  
    L->next=NULL;  
    return OK;  
}
```

2.清空单链表

```
Status ClearList(LinkList & L){ // 将L重置为空表
    LinkList p,q;
    p=L->next; //p指向第一个结点
    while(p)    //没到表尾
    {
        q=p->next;
        free(p);
        p=q;
    }
    L->next=NULL; //头结点指针域为空
    return OK;
}
```

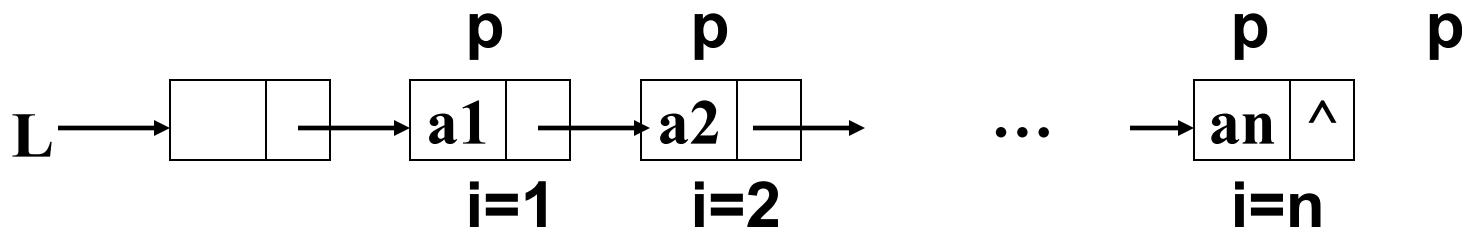


若要销毁单链表：
delete L;
L=NULL;

3.求表长

$O(n)$

```
int ListLength(LinkList L) {  
    //返回L中数据元素个数  
    LinkList p; int i=0;  
    p=L->next;           //p指向第一个结点  
    while(p) {           //遍历单链表,统计结点数  
        i++;  
        p=p->next;  
    }  
    return i;  
}
```



4.查找

- * 按序号查找: 找第 i 个元素
- * 按值查找: 找值为 e 的元素

思考:

- 顺序表里如何找到第 i 个元素?
- 链表的查找: 要从链表的头指针出发, 顺着链域`next`逐个结点往下搜索, 直至搜索到第 i 个结点为止。因此, 链表不是随机存取结构, 而是顺序存取。

4.1取线性表L中第i个元素（算法2.8）

```
Status GetElem(LinkList L,int i,ElemType &e){  
    //取带头结点单链表L中第i个元素值，由e带回  
    p=L->next; //初始化，p指向首元结点  
    j=1;        //初始化，计数器j为1，当前第1个  
    while( p&& j<i ){ //直到p为空或指向第i个元素  
        p=p->next; //p往后走，指向下一个  
        ++j;  
    }  
    if(!p || j>i) return ERROR; // i值不合法，不存在  
    e=p->data; //取第i个元素  
    return OK;  
} //GetElem
```

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

4.2.查找（查找值为e的数据元素）

```
LNode *LocateElem(LinkList L, Elemtype e) {  
    //在带头结点单链表L中查找值为e的元素  
    //若找到返回指向该结点的指针，否则返回NULL  
    LNode *p=L->next; //初始化p指向首元结点  
    while(p && p->data!=e) //遍历单链表找值为e结点  
        p=p->next;  
    return p; //p要么指向所找结点，要么为NULL  
}
```

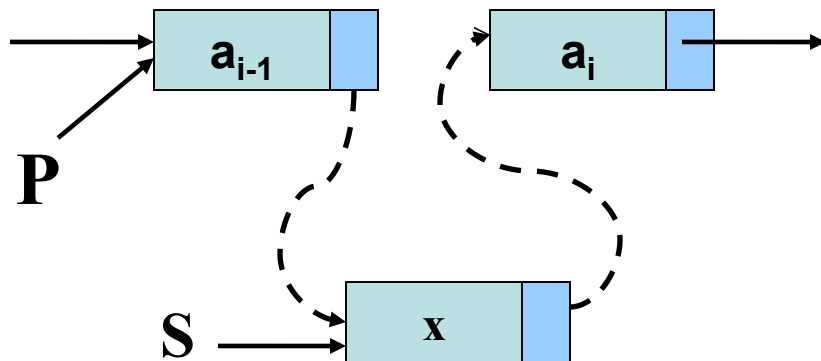
$$T(n)=O(n)$$

5. 单链表上的插入与删除

对于顺序表来说，由于是存储地址连续的，插入和删除都需要移动元素。

对于链表来说，由于存储位置通过指针来链接，插入和删除操作只需修改结点间的指针关系。

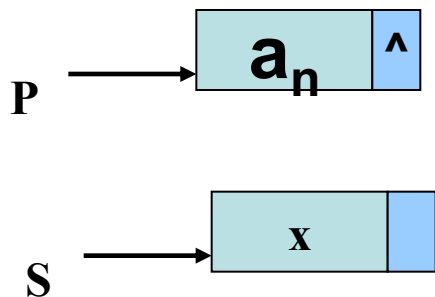
插入 (在单链表第 i 个位置插入一个结点)



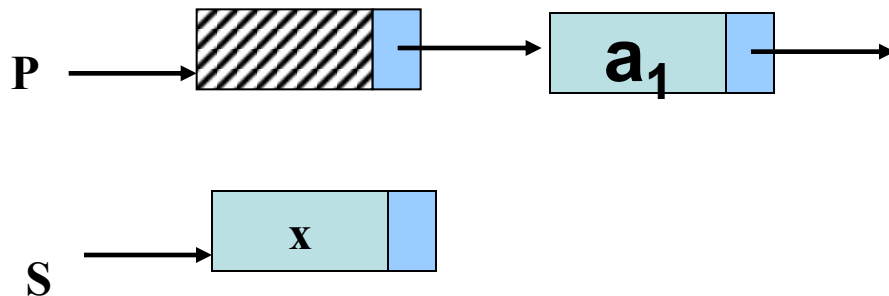
任意位置
均适用

```
s->next=p->next ;    p->next=s;
```

表尾插入

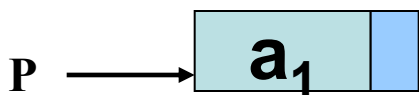


表头插入 (带头结点优势)

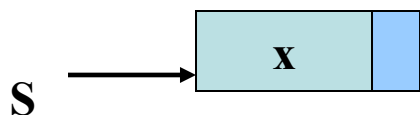



```
s->next=p->next ;    p->next=s;
```

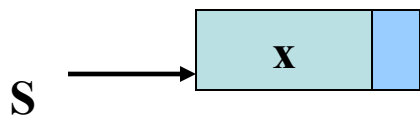
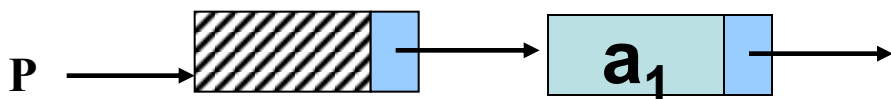
表头插入(不带头结点, 特殊处理)



```
s->next=p ;    p=s;
```



表头插入(带头结点, 优势)

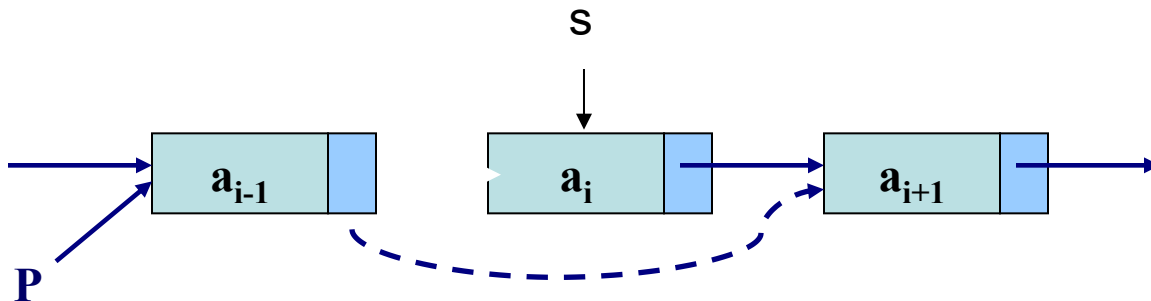


5.在L中第i个位置插入值为e的新结点 (算法2.9)

```
Status ListInsert(LinkList &L,int i,ElemType e){  
    p=L;   j=0;  
    while( p && j < i-1) {  
        p=p->next;  
        ++j;  
    } //寻找第 i-1 个结点,即要插入结点的前驱结点  
    if( !p || j > i-1)  
        return ERROR; // i>n + 1或者i<1, 不合法  
    s=(LinkList)malloc(sizeof(LNode)); //生成新结点s  
    s->data=e; //将结点s的数据域置为e  
    s->next=p->next; //将结点s插入L中  
    p->next=s;  
    return OK;  
} //ListInsert
```

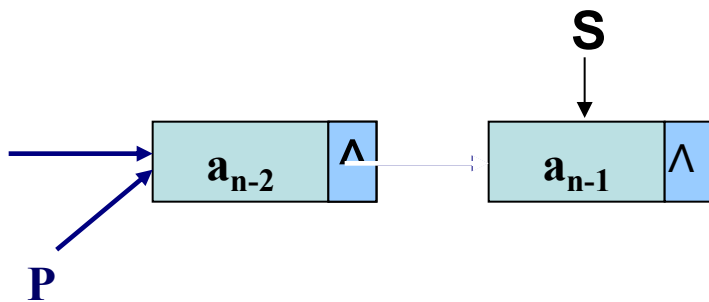
$$T(n)=O(n)$$

6. 单链表的删除

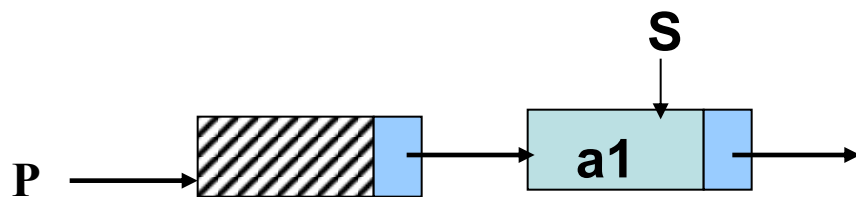


$p \rightarrow \text{next} = p \rightarrow \text{next} \rightarrow \text{next}$

$\text{free}(s)$



表尾删除



表头删除

6.删除单链表L的第i个元素,并由e返回值(算法2.10)

```
Status ListDelete( LinkList &L,int i,ElemType &e) {  
    p=L;j=0;  
    while( p->next && j<i-1 ){ //第i个结点前驱结点p  
        p=p->next;  
        ++j;  
    }  
    if( !(p->next) || j>i-1 ) return ERROR; //删除位置不合理  
    q=p->next; //q指向待删除结点，以备释放  
    p->next=q->next; //改变删除结点前驱结点的指针域  
    e=q->data; //可保存被删结点的值，并带回  
    free(q); //释放删除结点的空间  
    return OK;  
} //ListDelete
```

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

7.创建单链表

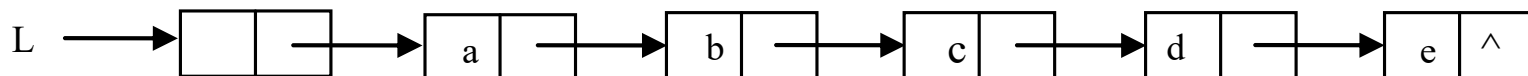
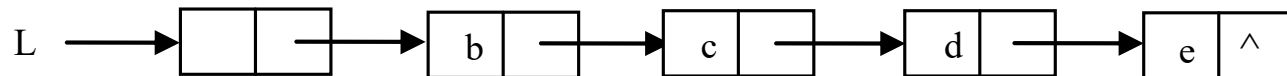
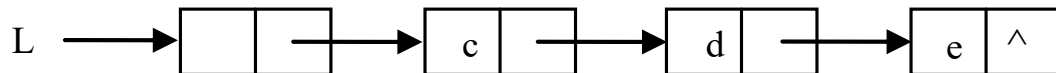
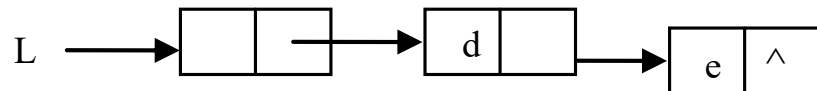
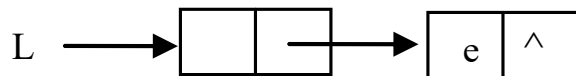
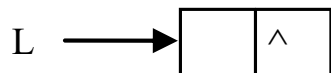
- **InitList()**是创建一个只有头结点的空链表。
- 链表中各个元素（结点）是动态生成并插入到链表中。即从空表开设，逐个生成结点并插入。
- **s=(LinkedList)malloc(sizeof(LNode))**的作用是生成一个结点，**s**指向该结点。反之，**free(s)**的作用是释放结点**s**的空间。
- 可多次通过调用函数 **ListInsert(LinkedList &L, int i, ElemType e)** 实现，也可专门编写创建函数
- 链表可根据需要有不同的生成方式，包括：
头(前)插法、尾(后)插法、有序插入法，其中，最方便的是**头插法**。

头（前）插法建立单链表

头(前)插法：把每次新生成的结点插入到头部，即新结点均为第一个结点，步骤如下：

■ **从一个空表开始，重复读入数据：**

- ◆ **创建头结点，构成空链表**
- ◆ **重复（循环）以下操作：**
 - ◆ **生成新结点p**
 - ◆ **输入新结点p的元素值 (p->data)**
 - ◆ **将新结点p插入到链表的头结点之后**



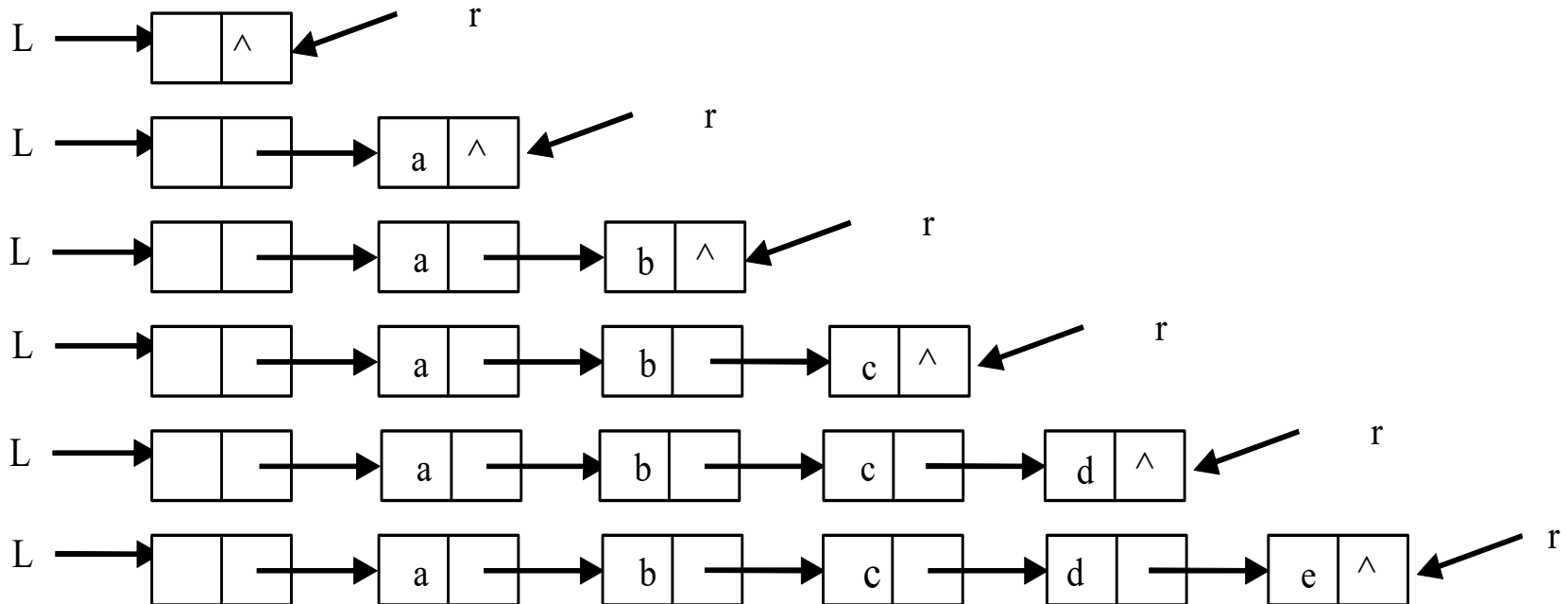
**链表结点
顺序与输入
顺序相反**

头(前)插法创建单链表(算法2.11)

```
void CreateList_L(LinkList &L,int n) {  
    //逆序输入n个元素的值，头插法构建带头结点的单链表L  
    L= (LinkList)malloc(sizeof(LNode));  
    //L=new 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  
        cin>>p->data;//scanf(&p->data); //输入值  
        p->next=L->next;  
        L->next=p;      //插入到表头  
    }  
} //CreateList_L
```

单链表的建立（尾插法）

- 从空表L开始，将新结点逐个插入到链表的尾部，**尾指针r指向链表的尾结点。**
- 初始时，r同L均指向头结点。
- 每读入一个数据元素则申请一个新结点，将新结点插入到尾结点后，r指向新结点。



头(前)插法创建单链表(算法2.12)

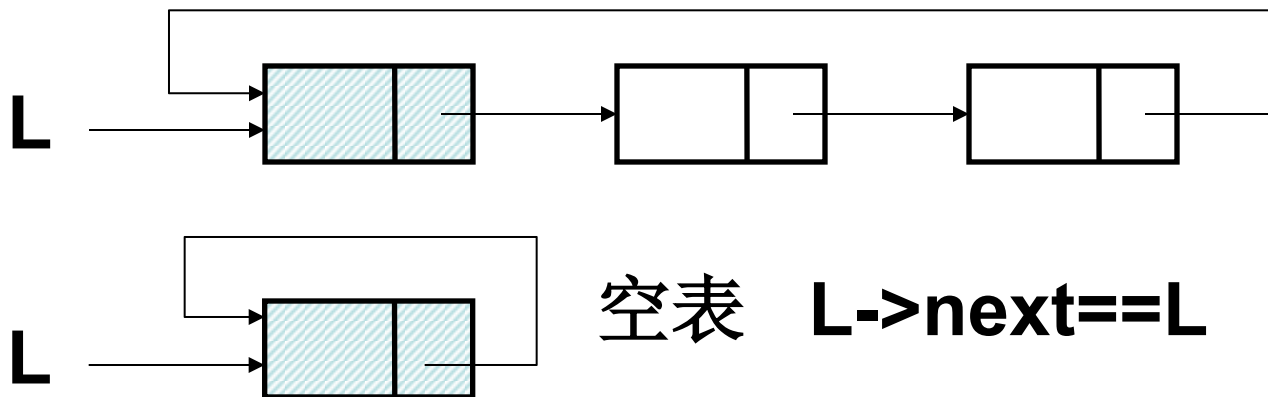
```
void CreateList_L(LinkList &L,int n){  
    //正位序输入n个元素的值，建立带表头结点的单链表L  
    L=new LNode; // L= (LinkList)malloc(sizeof(LNode));  
    L->next=NULL;  
    r=L;           //尾指针r指向头结点  
    for(i=0;i<n;++i){  
        p=new LNode;           //生成新结点  
        cin>>p->data;           //输入元素值  
        p->next=NULL; r->next=p; //插入到表尾  
        r=p;                   //r指向新的尾结点  
    }  
} //CreateList_L
```

2.5.3 循环链表



与单链表的最大区别在于：循环链表的最后一个结点的指针指向头结点，整个链表形成一个环。

最大优势在于：从表中任意一个结点出发都可找到表中其他结点。



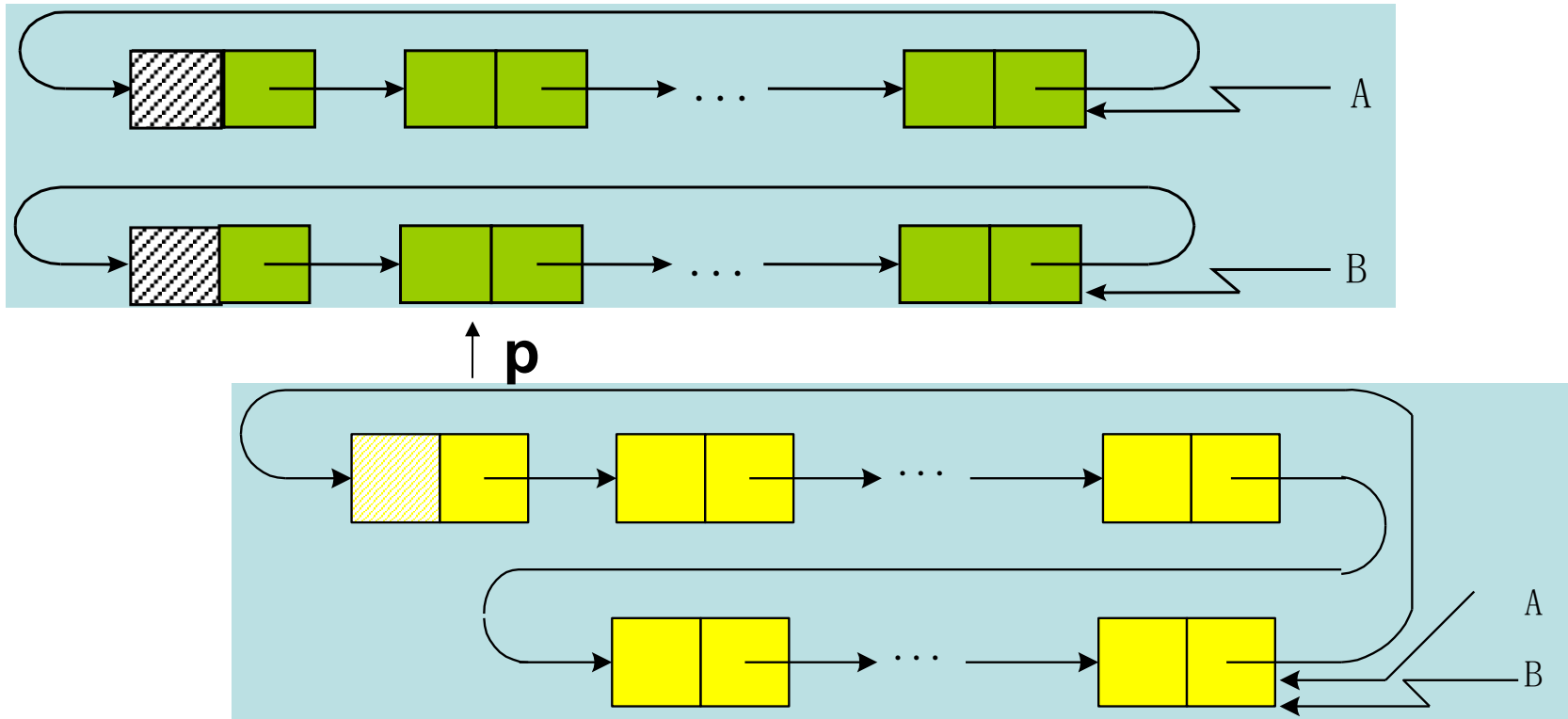
循环单链表的操作与单链表基本一致，但由于循环链表结点中不存在为**NULL**的指针域，故只需改变两处判断：

- 1.空表： $L \rightarrow next == L$ (单： $L \rightarrow next == NULL$)
- 2.表尾： $p \rightarrow next == L$ (单： $p \rightarrow next == NULL$)

```
int ListLength(CLinkList L)
{ //求循环链表L的长度
  p=L->next; k=0
  while(p!=L) {
    k++;
    p=p->next;
  }
  return k;
}
```

```
int ListLength(LinkList L)
{ //求单链表L的长度
  p=L->next; k=0
  while(p!=NULL) {
    k++;
    p=p->next;
  }
  return k;
}
```

某些情况下，循环链表只设置尾指针更合理，既方便找尾结点，又方便找首结点，如循环链表的合并。



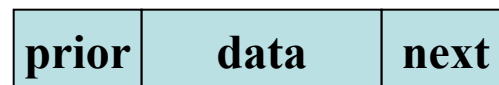
```
p=B->next->next;  
B->next=A->next;  
A->next=p
```

2.5.4 双向链表

对于单链表来说，由于结点结构中只设置一根指向后继的指针next，即只能通过next指针按顺序往后找，不能立即找某一结点的前驱。

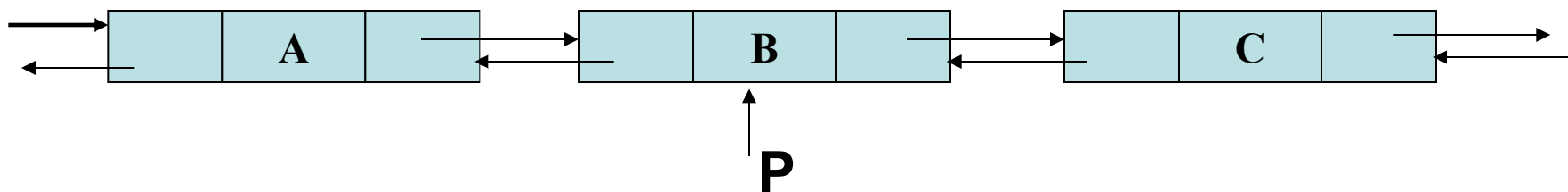
对于双向链表来说，结点结构中除设置一根指向后继的指针外，再设置一根指向前驱的指针，故既能往后找后继，又能往前找前驱。

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



结点结构

双向链表的指针关系



p->data == 'B'

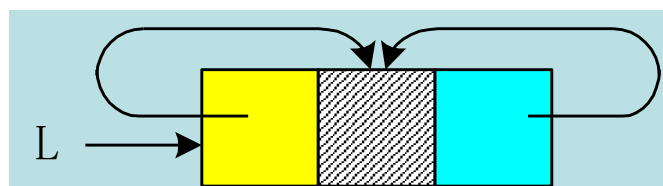
p->next->data == 'C'

p->prior->data == 'A'

p->prior->next == p

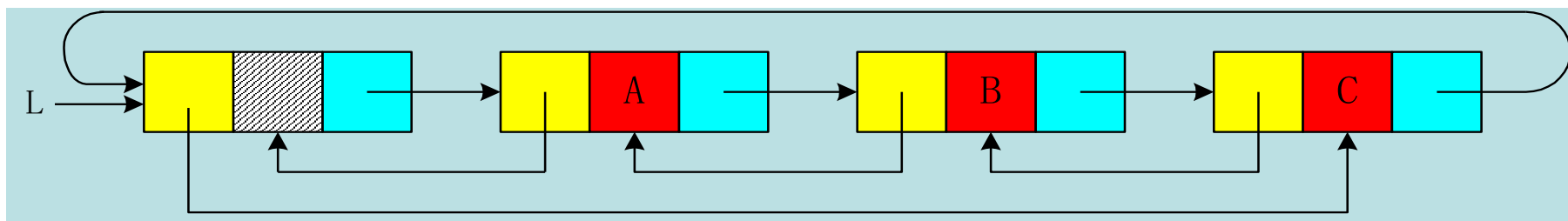
p->next->prior == p

双向链表也可以有循环，称双向循环链表。此时存在两个环，一个通过**next**，另一个通过**prior**。



(a) 空双向循环链表

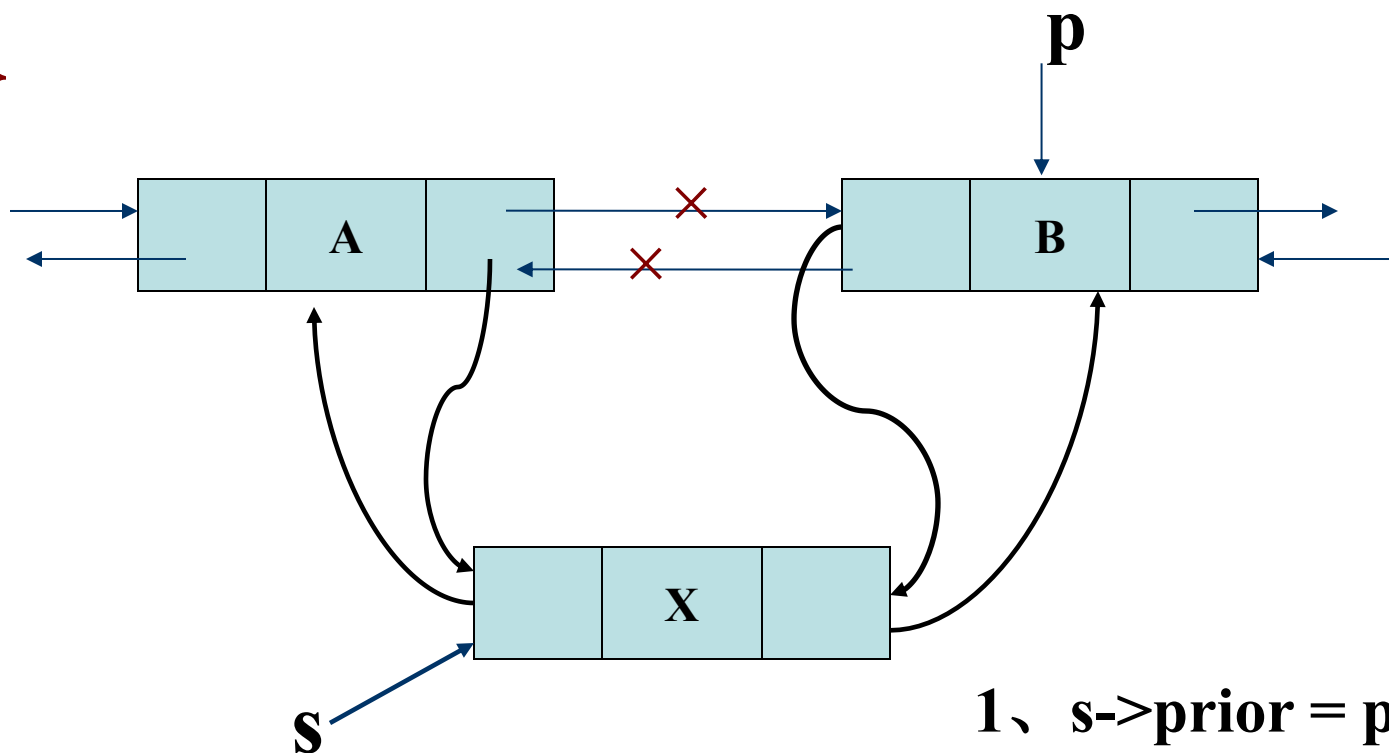
```
L->next==L;  
L->prior==L;
```



(b) 非空双向循环链表

双向链表的插入 (p结点前插入s)

插入



1、 $s \rightarrow \text{prior} = p \rightarrow \text{prior}$

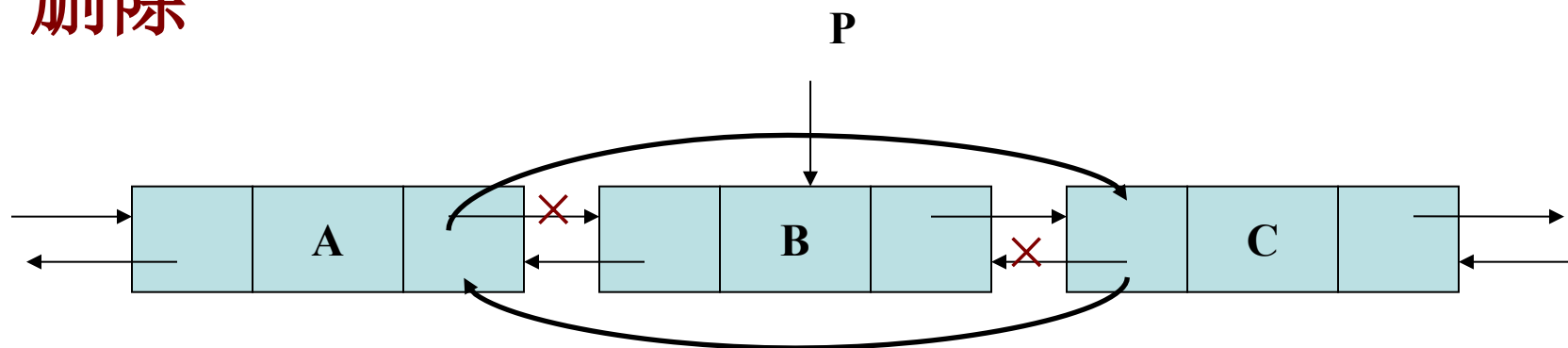
2、 $p \rightarrow \text{prior} \rightarrow \text{next} = s$

3、 $s \rightarrow \text{next} = p$

4、 $p \rightarrow \text{prior} = s$

双向链表的删除操作

删除



1. $p \rightarrow \text{prior} \rightarrow \text{next} = p \rightarrow \text{next}$
2. $p \rightarrow \text{next} \rightarrow \text{prior} = p \rightarrow \text{prior}$
3. `free(p) //delete p`

链表（链式存储结构）的特点

- (1) 结点在存储器中的位置是任意的，即逻辑上相邻的数据元素在物理上不一定相邻
- (2) 访问时只能通过头指针进入链表，并通过每个结点的指针域向后扫描其余结点，所以寻找第一个结点和最后一个结点所花费的时间不等

这种存取元素的方法被称为**顺序存取法**

链表的优缺点

优点

- 数据元素的个数可以自由扩充
- 插入、删除等操作不必移动数据，只需修改链接指针，修改效率较高

缺点

- 元素逻辑关系需要添加一指针域，需较多内存，存储密度低
- 算法相对复杂些，顺序存储

顺序表和链表的比较

存储结构		顺序表	链表
比较项目			
空间	存储空间	预先分配，会导致空间闲置或溢出现象	动态分配，不会出现存储空间闲置或溢出现象
	存储密度	不用为表示结点间的逻辑关系而增加额外的存储开销，存储密度等于1	需要借助指针来体现元素间的逻辑关系，存储密度小于1
时间	存取元素	随机存取 ，按位置访问元素的时间复杂度为 $O(1)$	顺序存取 ，按位置访问元素时间复杂度为 $O(n)$
	插入、删除	平均 移动 约表中一半元素，时间复杂度为 $O(n)$	不需移动 元素，确定插入、删除位置后，时间复杂度为 $O(1)$
适用情况		<div>① 表长变化不大，且能事先确定变化的范围</div> <div>② 很少进行插入或删除操作，经常按元素位置序号访问数据元素</div>	<div>① 长度变化较大</div> <div>② 频繁进行插入或删除操作</div>

2.7 线性表的应用



1. 一般线性表的合并

假设利用两个线性表La和Lb分别表示两个集合A和B,现要求一个新的集合 **$A=A \cup B$**

```
Void MergeList( List &La , List Lb) {  
    // 将所有在线性表Lb中但不在La中的数据元素插入到La中  
    La_len = ListLength( La );  
    Lb_len = ListLength( Lb );    //求线性表的长度  
    for ( i = 1 ; i <= Lb_len ; i++) {  
        GetElem( Lb , i , e );      // 取Lb中第i个数据元素赋给e  
        if ( !LocateElem( La , e )) ListInsert( La, ++La_len, e );  
            // La中不存在与e相同的数据元素，插入e  
    }  
} // Union O(La_len*Lb_len)
```

2. 有序线性表的合并

已知两个线性表La和Lb中的数据元素按值非递减有序排列（有序集），现要求一个新的有序集合Lc， $Lc=La \cup Lb$ ，且Lc中的数据元素仍按值非递减有序排列。

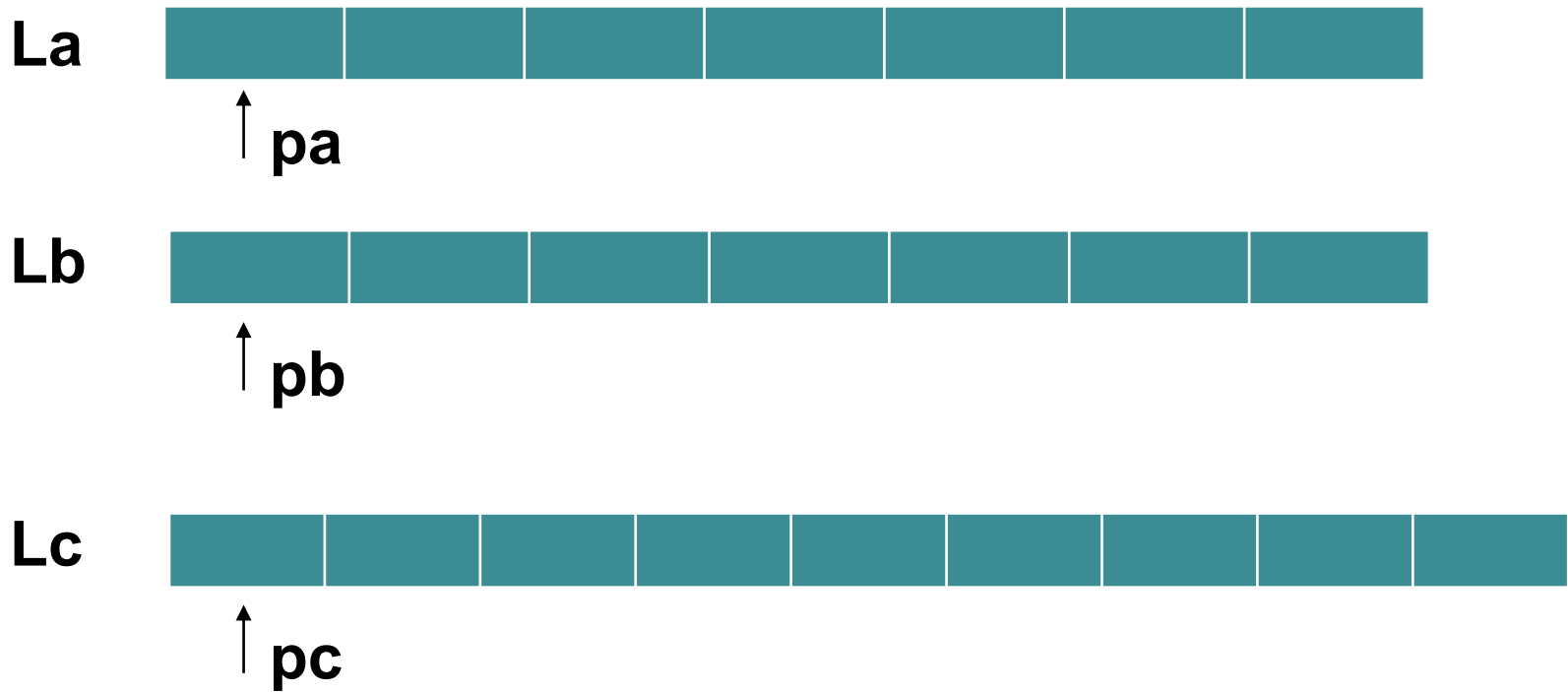
La=(1, 7, 8)

Lb=(2, 4, 6, 8, 10, 11)

Lc=(1, 2, 4, 6, 7, 8, 8, 10, 11)

顺序表结构和链表结构实现方法不一致

(1) 有序顺序表的合并



```
if(*pa<=*pb) *pc++=*pa++;  
else *pc++=*pb++;
```

```

void MergeList_Sq(SqList LA, SqList LB, SqList &LC){
    pa=LA.elem; pb=LB.elem;    //指针pa和pb的初值分别指向两个表的第一个元素
    LC.length=LA.length+LB.length;    //新表长度为待合并两表的长度之和
    LC.elem=new ElemType[LC.length];    //为合并后的新表分配一个数组空间
    pc=LC.elem;    //指针pc指向新表的第一个元素
    pa_last=LA.elem+LA.length-1;    //指针pa_last指向LA表的最后一个元素
    pb_last=LB.elem+LB.length-1;    //指针pb_last指向LB表的最后一个元素
    while(pa<=pa_last && pb<=pb_last){    //两个表都非空
        if(*pa<=*pb) *pc++=*pa++;    //依次“摘取”两表中值较小的结点
        else *pc++=*pb++;    }
    while(pa<=pa_last) *pc++=*pa++;    //LB表已到达表尾
    while(pb<=pb_last) *pc++=*pb++;    //LA表已到达表尾
} //MergeList_Sq

```

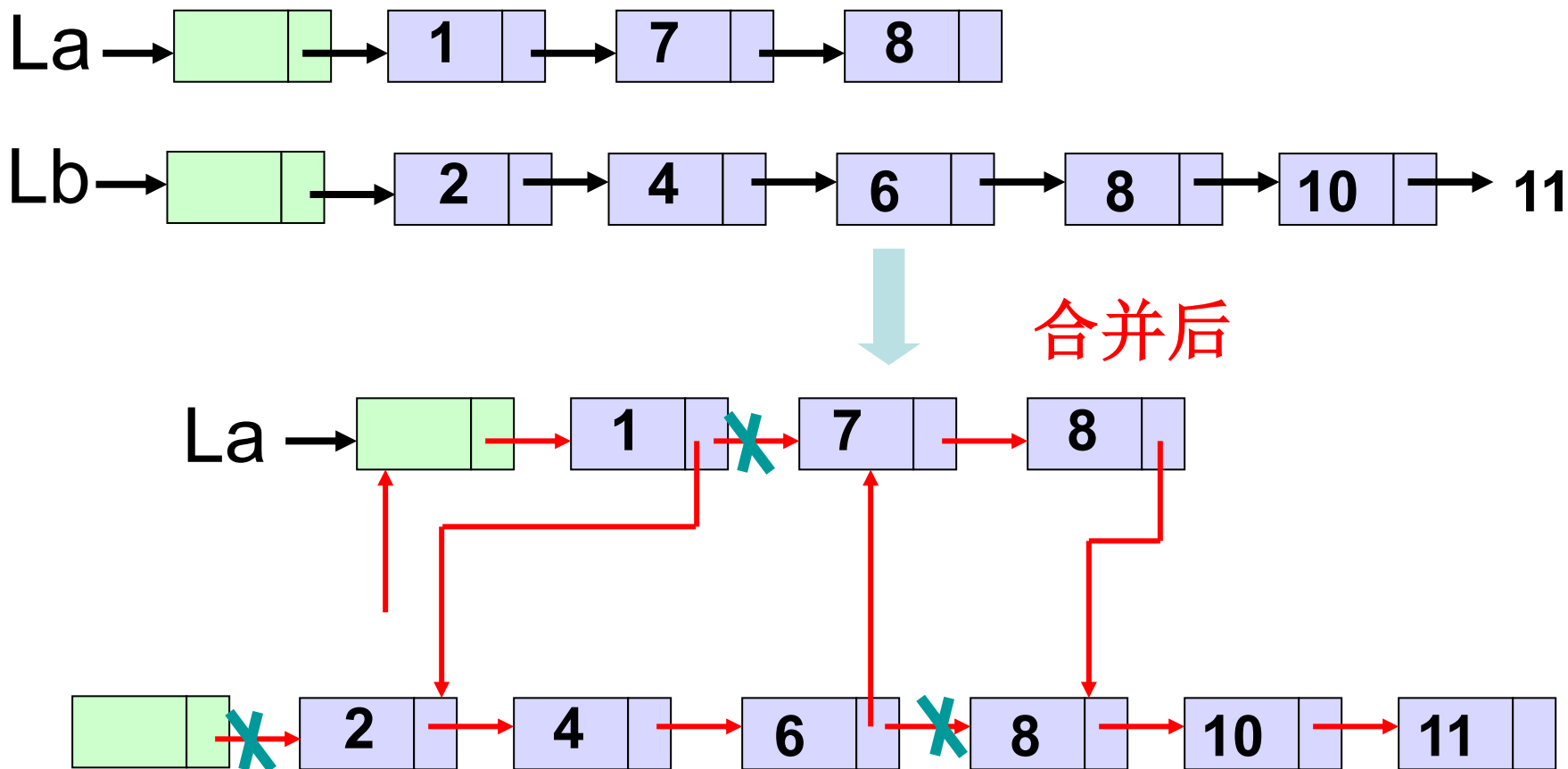
$$T(n) = O(ListLength(LA) + ListLength(LB))$$

$$S(n) = O(n)$$

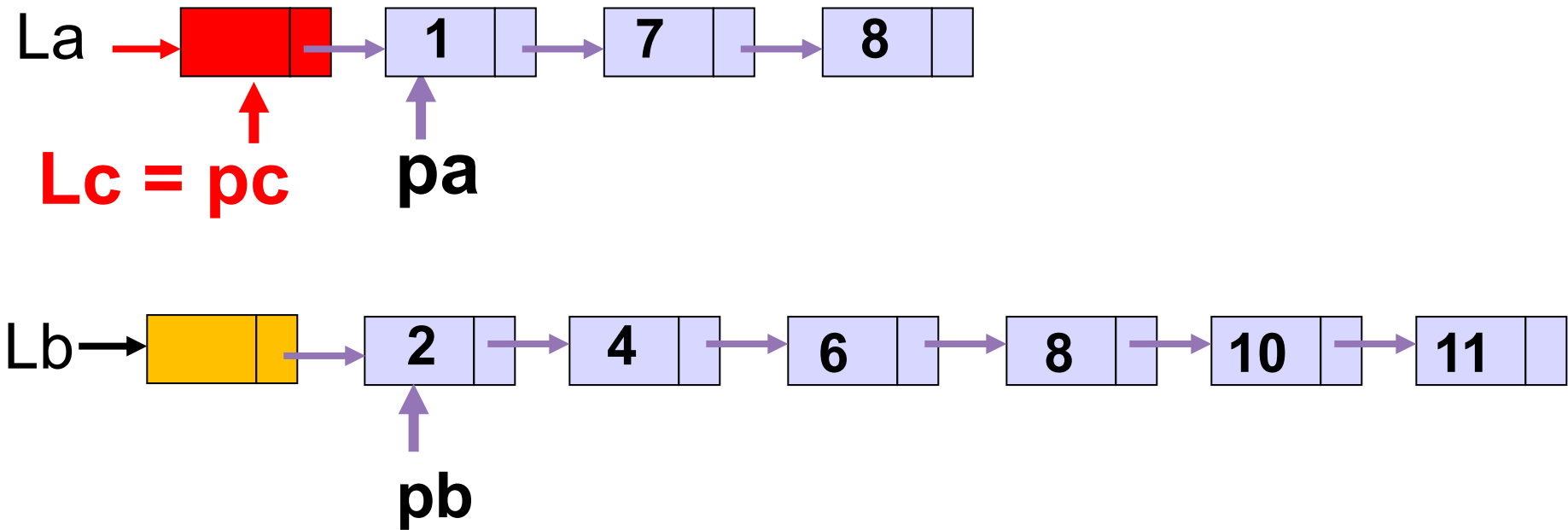
(2) 有序链表的合并

两种方式: $Lc = La \cup Lb$ $La = La \cup Lb$

采用第二种, 使用原来两个链表的存储空间, 不另外占用其它的存储空间



初始状态



```
if(pa->data<=pb->data) {  
    pc->next=pa; pc=pa; pa=pa->next; }  
else {  
    pc->next=pb; pc=pb; pb=pb->next;}
```



3. 多项式的表示

1. 一元多项式表示

$$P(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

用线性表表示多项式的一种方法：

所有系数用线性表表示： $(a_0, a_1, a_2, \dots, a_n)$

可用顺序或链接存储结构实现，顺序结构方便。

如： $P(x) = 2 + 4x - 8x^3 - x^5$

可用线性表 $(2, 4, 0, -8, 0, -1)$

如多项式: $P1(x) = 10 + 5x - 4x^2 + 3x^3 + 2x^4$

$P2(x) = 3 + 4x - 8x^2 + 13x^3 + 6x^4 - 7x^5$

P1

下标	0	1	2	3	4
系数	10	5	-4	3	2

求值:

```
for( i=0; i<n; i++)  
    y=y+p1[i]*pow(x,i)
```

P2

下标	0	1	2	3	4	5
系数	3	4	-8	13	6	-7

多项式相加:

对应下标的系数相加

2. 稀疏多项式表示

但如: $P(x) = 1 + 7x^{10000} + 4x^{15000}$

线性表 (1, 0, ..., 0, 7, 0, ..., 0, 4)

用上述方法太浪费空间

一般采用存储非零系数与相应指数的方法:

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

其中: p_i 为非零系数, $0 \leq e_1 \leq e_2 \leq \dots \leq e_m = n$

线性表表示: ($(p_1, e_1), (p_2, e_2), \dots, (p_m, e_m)$)

如: $P(x) = 1 + 7x^{10000} + 4x^{15000}$

线性表 ((1, 0), (7, 10000), (4, 15000))

实现方法一：用顺序存储结构

```
typedef struct {  
    double coef; //系数  
    int exp;      //指数  
} Term;
```

```
typedef Term ElemType; //顺序表元素类型
```

```
typedef struct  
{  
    ElemType *elem; //动态存储空间的首地址  
    int length;     //当前元素的个数  
} SqList;
```

如多项式: $A(x) = 7 + 3x + 9x^8 + 5x^{17}$
 $B(x) = 8x + 22x^7 - 9x^8$

A

下标	0		1		2		3	
系数/ 指数	7	0	3	1	9	8	5	17

求值:

```
for( i=0; i<A.length; i++)  
    y=y+A.elem[i].coef * pow(x,A.elem[i].exp);
```

多项式相加:

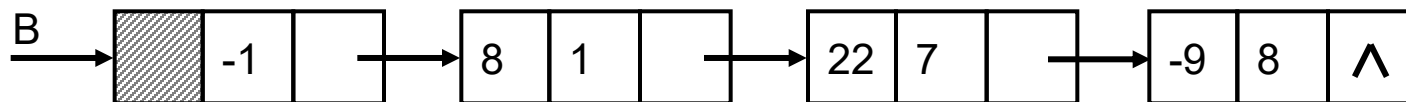
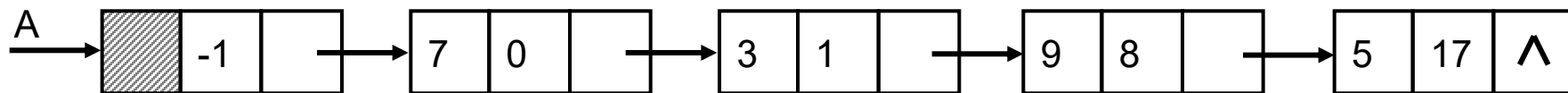
和顺序结构相比, 链式结构更加灵活

实现方法二：用链式存储结构

```
typedef struct PNode {  
    double coef;  
    int exp;  
    struct PNode *next;  
} PNode *Polynomial;
```

如多项式： $A(x) = 7 + 3x + 9x^8 + 5x^{17}$
 $B(x) = 8x + 22x^7 - 9x^8$

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



$$B_8(x) = 8x + 22x^7 - 9x^8$$

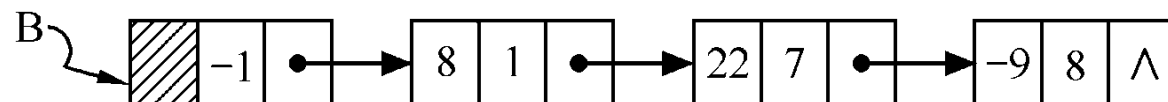
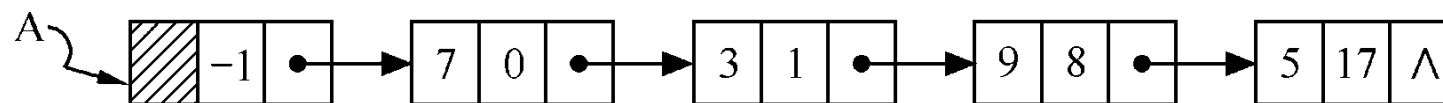
多项式相加（用链接存储结构）

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

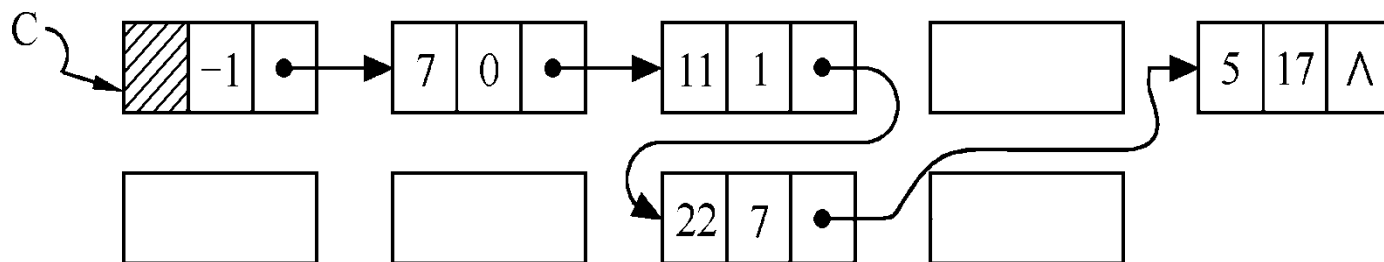
$$B_8(x) = 8x + 22x^7 - 9x^8$$

指数相同的项：相加，若和不为0，构成一项。

指数不同的项：复制到多项式中。



A=A+B



思考:

设用线性表 $((a_1, e_1), (a_2, e_2), \dots, (a_m, e_m))$ 表示多项式 $P(x) = a_1 * x^{e_1} + a_2 * x^{e_2} + \dots + a_m * x^{e_m}$,
请编写用链式存储结构存储该多项式时, 插入一项 $a * x^e$ 的算法 $\text{InsertPoly}(P, a, e)$ 。

$$A_{17}(x) = 7 + 3x + 9x^8 + 5x^{17} \quad \text{插入 } -10x^{12}$$

