第2章 表

- ADT表
- 用数组实现表
- 用指针实现表
- 用游标实现表
- 用间接寻址方法实现表
- 循环链表
- 双链表
- 线性表的应用

2.1 ADT表

- 表或称线性表是一种非常灵活的结构，可以根据需要改变表的长度，也可以在表中任何位置对元素进行访问、插入或删除等操作。另外，还可以将多个表连接成一个表，或把一个表分拆成多个表。表结构在信息检索、程序设计语言的编译等许多方面有广泛作用。

2.1 ADT表

- 定义：线性表L是n(n>=0)个相同类型数据元素a_1, …, a_{n-1}, a_n构成的有限序列。表示成L=(a_1,… ,a_{n-1},a_n）
- 例：英文字母表（A,B,C,…..Z)是一个线性表
- 例：

学号	姓名	年龄
001	张三	18
002	李四	19
……	……	……

数据元素

2.1 ADT表

- 表长：线性表中元素的个数，n=0时为空表。
- 直接前驱：线性表中a_{i-1}领先于a_i，则a_{i-1}是a_i的直接前驱元素；a_1无直接前驱。
- 直接后继：线性表中a_i领先于a_{i+1}，则a_{i+1}是a_i的直接后继；a_n无直接后继。
- 表元素之间的逻辑关系就是上述的邻接关系。由于这种关系式线性的，所以表是一种线性结构，也称线性表。
- 在上述数学模型上，还要定义一组关于表的运算，才能使这一数学模型成为一个抽象数据类型List。

2.1 ADT表

- 线性表的基本运算
  - **ListEmpty(L)**：测试表**L**是否为空
  - **ListLength(L)**：表**L**的长度
  - **ListLocate(x,L)**：元素**x**在表**L**中的位置。若**x**在表中重复出现多次，则返回最前面的**x**的位置。
  - **ListRetrieve(k,L)**：返回表**L**的位置**k**处的元素。表中没有位置**k**时，该运算无定义。

2.1 ADT表

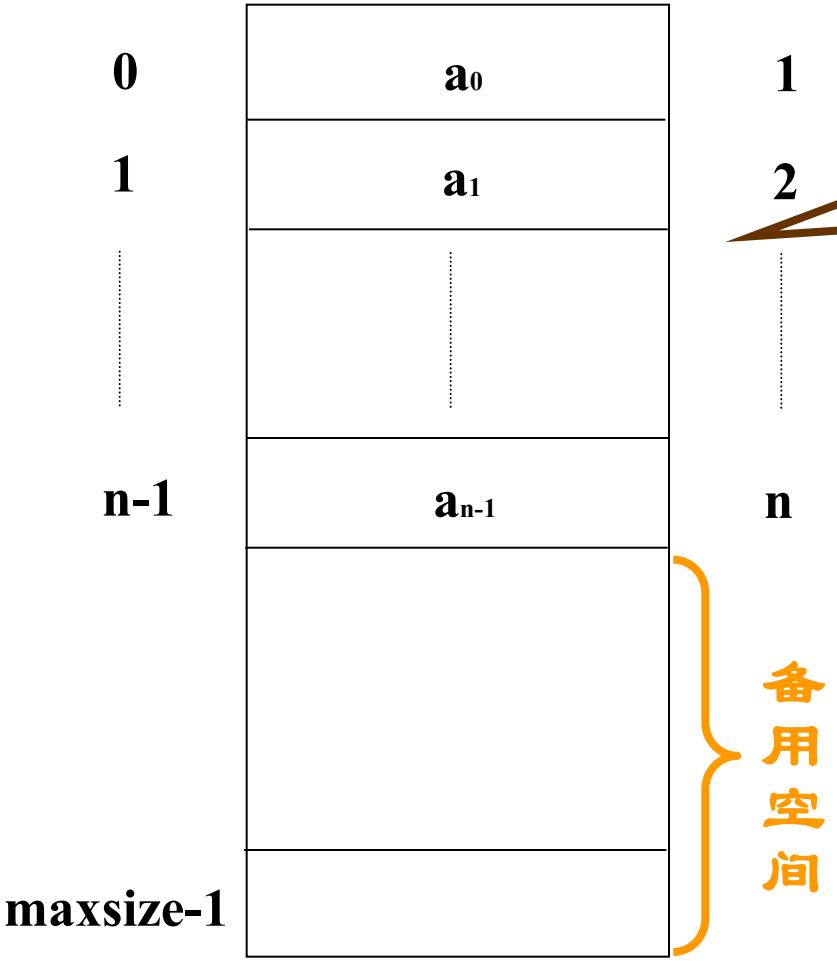
- 线性表的基本运算
  - **ListInsert(k,x,L)**：在表L的位置k之后插入元素x，并将原来占据该位置的元素及其后面的元素依次后移一个位置。若表中没有位置k，则该运算无定义。
  - **ListDelete(k,L)**：从表L中删除位置k处的元素，并返回被删除的元素。当表中没有位置k时，则该运算无定义。
  - **PrintList (L)** ：将表L中所有元素按位置的先后次序打印输出

2.2 用数组实现表

2.2.1 顺序表的结构特点

- 定义：用一组地址连续的存储单元（数组）存放一个线性表叫顺序表。
- 元素地址计算方法
  - $LOC(a_i)=LOC(a_0)+i*L$　　$LOC(a_{i+1})=LOC(a_i)+L$
  - 其中，L是一个元素占用的存储单元个数，$LOC(a_i)$是线性表第i个元素的地址
- 特点：
  - 实现逻辑上相邻—物理地址相邻
  - 实现随机存取
- 实现：可用C语言的一维数组实现

V数组下标　　内存　　元素序号




```
#define maxsize 1000
int   data[maxsize];
```

例
```
typedef  struct  card
{   int num;
    char name[20];
    char author[10];
    char  publisher[30];
    float  price;
}card;
card  library[maxsize];
```

数据元素不是简单类型时,可定义结构体数组

2.2.2 用数组实现的ADT表的类型

- 用数组实现表时，为了适应表元素类型的变化，将表类型List定义为一个结构体。

```
typedef struct alist *List;/* 单链表指针类型 */
typedef struct alist{
    int n, /* 表长 */
        curr;/* 当前位置 */
    int maxsize;/* 数组上界 */
    ListItem *table;/* 存储表元素的数组 */
}Alist;
```

用数组实现表

2.2.3 顺序表LIST的基本运算实现

1、初始化函数

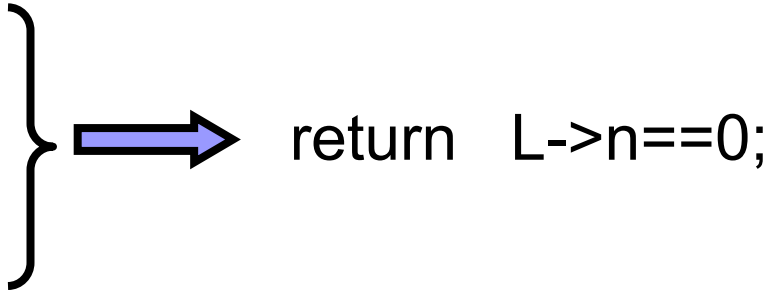
分配大小为size的空间给表数组table，并返回初始化为空的表。

```
List ListInit(int size)
{
   List L=(List)malloc(sizeof *L);
   L->table=(ListItem *)malloc(size*sizeof(ListItem));
   L->maxsize=size;
   L->n=0;
   return L;
}
```

2、判断表是否为空及求表长函数

（1）判断线性表L是否为空

```
int ListEmpty(List L)
{
   if (L->n==0) return TRUE;
   else         return FALSE;
}
```



（2）求表长

```
int GetLength(List L)
{
   return  L->n;
}
```

以上两个程序的时间复杂性均为：O(1)

3、元素x定位函数

返回元素**x**在表中的位置，当元素**x**不在表中时返回**0**。

```
int ListLocate(ListItem x,List L)
{
   for(int i=0;i<L->n;i++)
     if(L->table[i]==x)return ++i;
   return 0;
}
```

此算法所用时间与元素**x**所在位置有关。假设**x**存在于第**i**个位置，则找到**x**需要比较**i**次。故在最坏情况下，时间性能为**O(n)**。

4、获取线性表L中的某个数据元素内容的函数

```
ListItem ListRetrieve(int k,List L)
{
   if(k<1 || k>L->n)exit(1);/* 越界 */
   return L->table[k-1];
}
```

时间复杂性O(1)

5、表元素插入运算

void ListInsert (k, x, L)

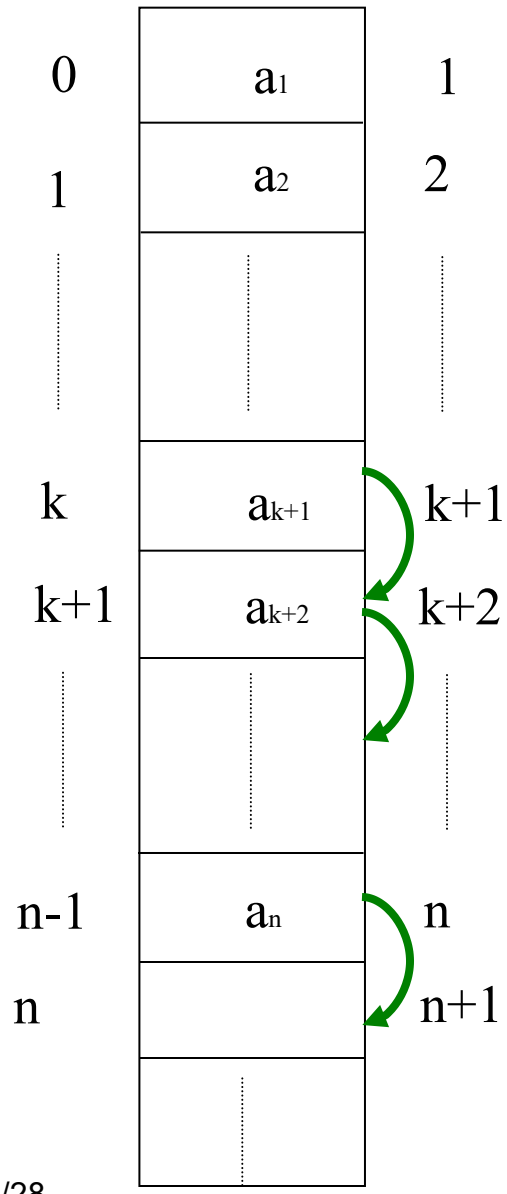
定义：线性表的插入是指在第k（0≤k ≤n）个元素之后插入一个新的数据元素x，使长度为n的线性表

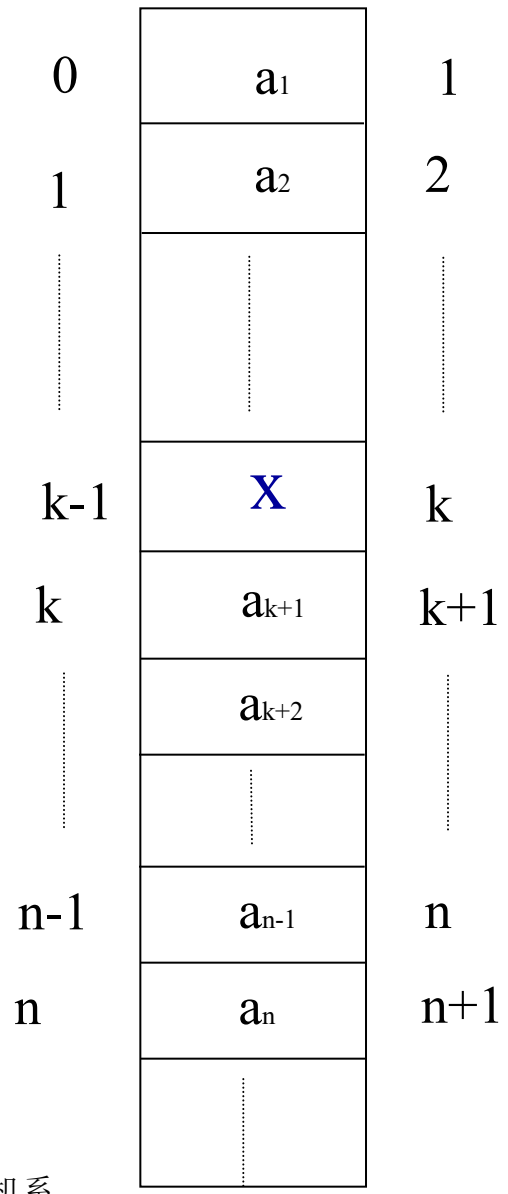
$$(a_1, a_2, \ldots\ldots, a_k, a_{k+1}, \ldots\ldots\ a_n)$$

变成长度为n+1的线性表

$$(a_1, a_2, \ldots\ldots, a_k, x, a_{k+1}, \ldots\ldots\ a_n)$$

需将第k+1至第n共 (n−k)个元素后移


V数组下标
内存
元素序号
0
1
k
k+1
n-1
n
a1
a2
ak+1
ak+2
an
1
2
k+1
k+2
n
n+1


V数组下标
内存
元素序号
0
1
k-1
k
n-1
n
a1
a2
X
ak+1
ak+2
an-1
an
1
2
k
k+1
n
n+1

```
void ListInsert(int k,ListItem x,List L)
{
   if(k<0 || k>L->n)exit(1);/* 越界 */
   for(int i=L->n-1;i>=k;i--)L->table[i+1]=L->table[i];
   L->table[k]=x;
   L->n++;
}
```

- 算法时间复杂度T(n)
  - 问题的规模是表的长度n。
  - 该算法的主要时间花费在for循环的元素后移上，该语句的执行次数为n-k，故所需移动元素位置的次数依赖于表的长度n和插入的位置k。
  - 当k=n时，元素后移语句将不进行，无需移动数组元素；

    这是最好情况，其时间复杂度O(1)。
  - 当k=0时，需移动表中所有元素，即n次；

    这是最坏情况，其时间复杂度为O(n)。

  由于插入可能在表中的任何位置上进行，因此，有必要分析算法的平均性能。

- 算法时间复杂度T(n)
  - 设P_k是在第k个元素之后插入一个元素的概率，则在长度为n的线性表中插入一个元素时，所需移动的元素次数的平均次数为：

$$E_{IN}(n)=\sum_{k=0}^{n}P_k(n-k)$$

若假设在表中任何合法位置插入元素的机会是均等的，则：

$$P_0=P_1=\cdots=P_n=\frac{1}{n+1}$$

$$\text{则}E_{\mathrm{IN}}(\mathrm{n})=\sum_{\mathrm{k}=0}^{n}P_{\mathrm{k}}(n-\mathrm{k})$$

$$=\frac{1}{n+1}\sum_{\mathrm{k}=0}^{n}(n-\mathrm{k})=\frac{n}{2}$$

$$\therefore T(n)=O(n)$$

6、表元素删除运算

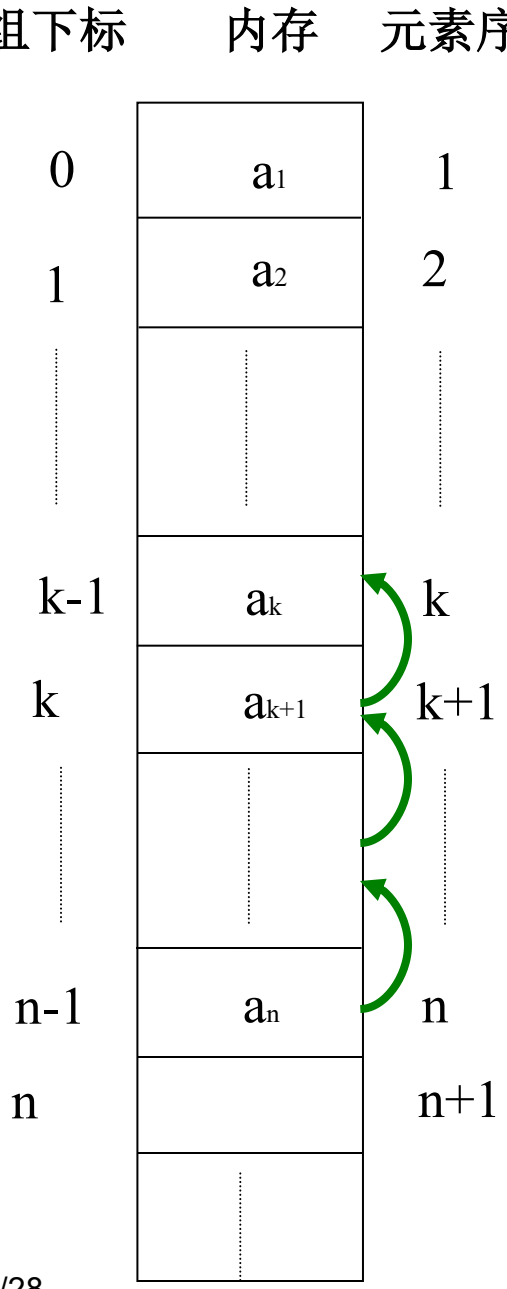
定义：线性表的删除是指将第k（1≤k ≤n）个元素删除，使长度为n的线性表

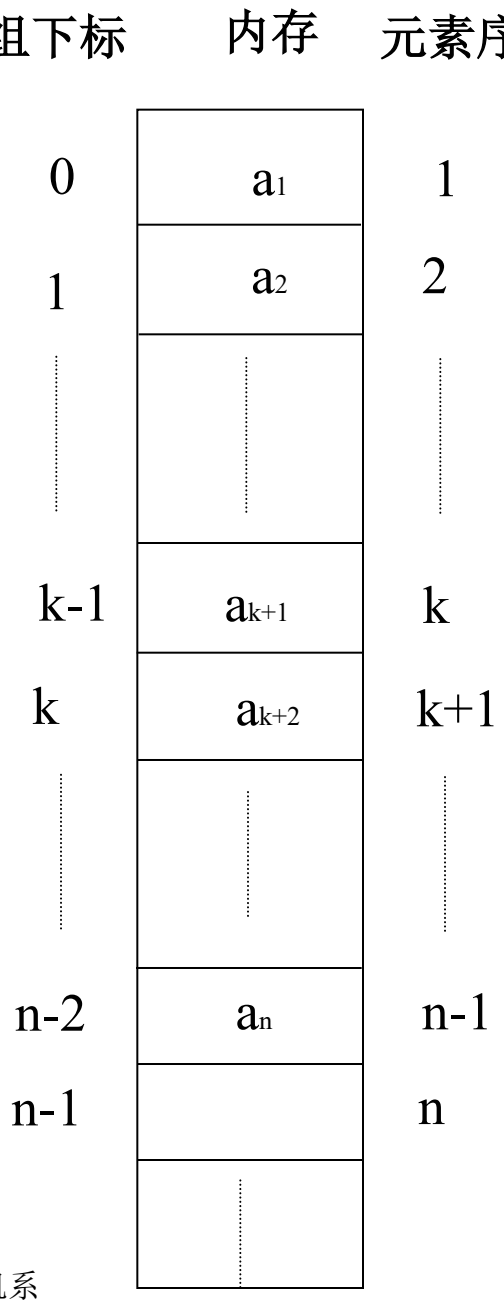
$$(a_1, a_2, \ldots\ldots, a_{k-1}, a_k, \ldots\ldots a_n)$$

变成长度为n-1的线性表

$$(a_1, a_2, \ldots\ldots, a_{k-1}, a_{k+1}, \ldots\ldots a_n)$$

需将第k+1至第n共 (n-k)个元素前移


V数组下标
内存
元素序号
0
1
k-1
k
n-1
n
a1
a2
ak
ak+1
an
1
2
k
k+1
n
n+1


V数组下标
内存
元素序号
0
1
k-1
k
n-2
n-1
a1
a2
ak+1
ak+2
an
1
2
k
k+1
n-1
n

```
ListItem ListDelete(int k,List L)
{
   if(k<1 || k>L->n)exit(1);/* 越界 */
   ListItem x=L->table[k-1];
   for(int i=k;i<L->n;i++)L->table[i-1]=L->table[i];
   L->n--;
   return x;
}
```

- 算法时间复杂度T(n)
  - 该算法的时间分析与插入算法类似，元素的移动次数也是由表长n和删除的位置k决定的。
  - 当k=n时，元素前移语句将不进行，无需移动数组元素；

    这是最好情况，其时间复杂度O(1)。
  - 当k=1时，前移语句将循环执行n-1次，需移动表中除删除元素外的所有元素；

    这是最坏情况，其时间复杂度为O(n)。

  删除运算的平均性能分析与插入运算类似。

- 算法时间复杂度T(n)
  - 设Q_i是删除第i个元素的概率，则在长度为n的线性表中删除一个元素所需移动的元素次数的平均次数为：

$$E_{DE}(n)=\sum_{k=1}^{n}Q_k(n-k)$$

$$若认为Q_k=\frac{1}{n}$$

$$则E_{DE}=\frac{1}{n}\sum_{k=1}^{n}(n-k)=\frac{n-1}{2}$$

$$\therefore T(n)=O(n)$$

故在顺序表中插入或删除一个元素时，平均移动表的一半元素，当n很大时，效率很低。

7、输出顺序表中所有元素的运算

```
void PrintList(List L)
{
   for(int i=0;i<L->n;i++)ItemShow(L->table[i]);
}
```

时间复杂性：O(n)

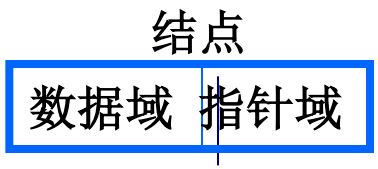
2.2.4 顺序存储结构的优缺点

- 优点
  - 逻辑相邻，物理相邻
  - 无须为表示表中元素之间的顺序关系增加额外的存储空间
  - 可随机存取任一元素
  - 存储空间使用紧凑
- 缺点
  - 插入、删除操作需要移动大量的元素（除操作在表尾的位置进行外）
  - 预先分配空间需按最大空间分配，利用不充分
  - 表容量难以扩充

2.3 用指针实现表

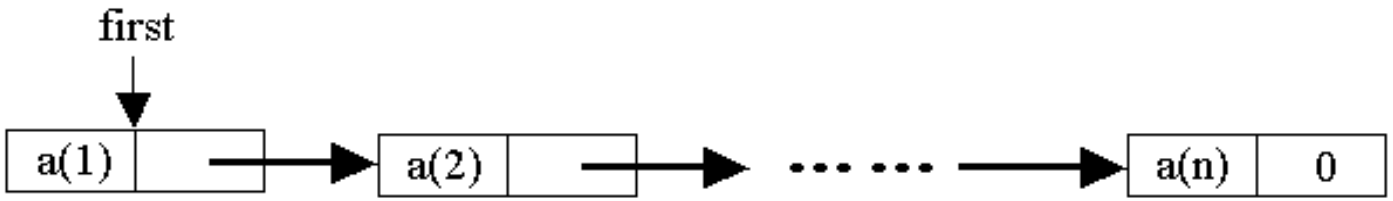
2.3.1 用指针实现表的特点

- 用一组任意的存储单元存储线性表的数据元素，利用指针将其串联在一起，实现了用不相邻的存储单元存放逻辑上相邻的元素。
- 以这种链式存储结构存储的表称为链表。
- 为能正确表示表中结点（即数据元素）间的逻辑关系，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信息，这两部分组成了链表中的结点结构。

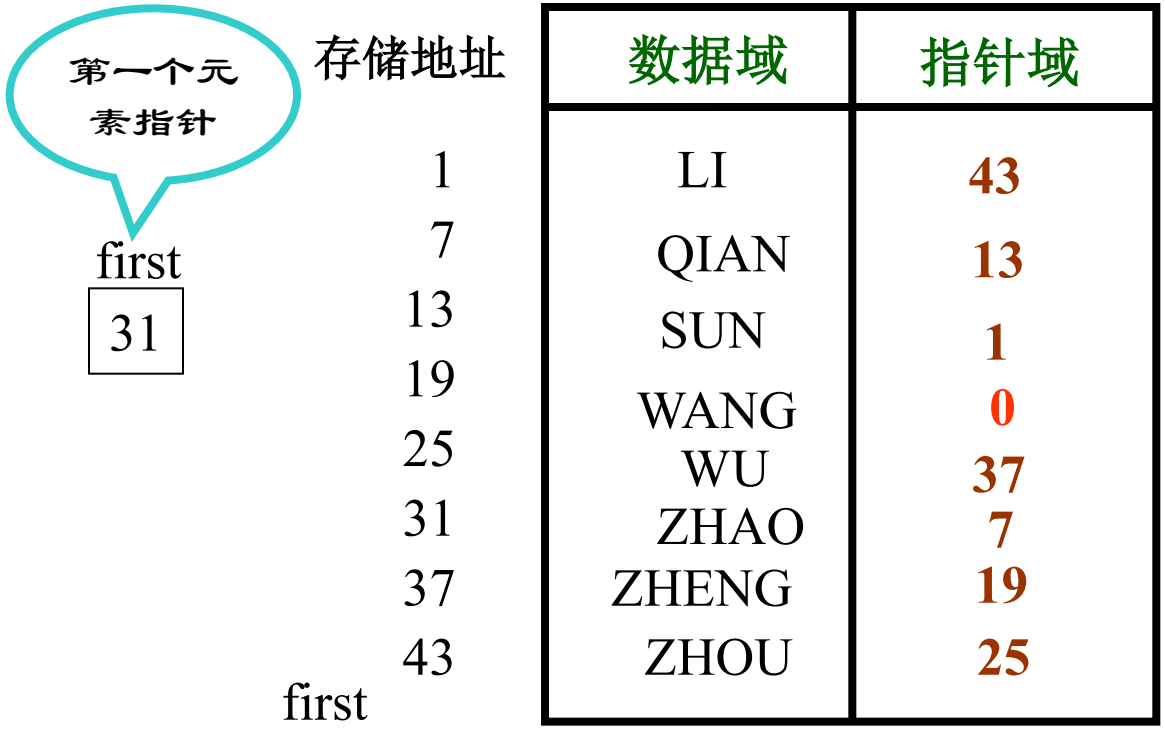


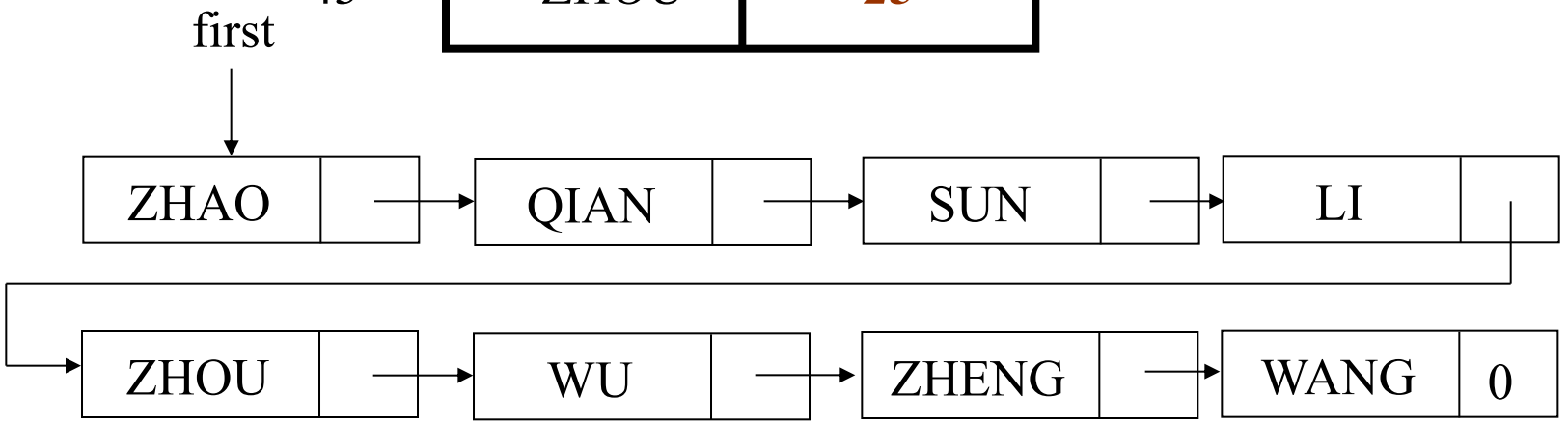
- 例如，如果表a(1),a(2),…a(n)，那么结点a(k)中的指针应指向结点a(k+1)的存储单元(k=1,2,…,n-1)。

其中，开始结点无前趋，应设头指针first指向开始结点；终端结点无后继，终端结点的指针域为空，即NULL。



例 线性表 (ZHAO,QIAN,SUN,LI,ZHOU,WU,ZHENG,WANG)





2.3.2 单链表的数据类型

- 定义：结点中只含一个指针域的链表，也叫线性链表。
- 单链表的结点结构说明为：

```
typedef struct node *link;/* 表结点指针类型 */
typedef struct node{/* 表结点类型 */
    ListItem element;/* 表元素 */
    link next;/* 指向下一结点的指针 */
} Node;
```

- 产生一个新结点的函数NewNode ( )：

```
link NewNode( )
{
    link p;
    if ((p=(link)malloc(sizeof(Node)))==0)
       Error("Exharsted memory.");
    else return p;
}
```

- 据此可定义用指针实现表的结构**List**如下：

```
typedef struct llist *List;/* 单链表指针类型 */
typedef struct llist{/* 单链表类型 */
    link first,/* 链表表首指针 */
        curr,/* 链表当前结点指针 */
        last;/* 链表表尾指针 */
}Llist;
```

表List的数据成员first为指向表中第一个元素的指针，当表为空表时first指针是空指针。

2.3.3 单链表的基本运算

1、创建一个空表

```
List ListInit()
{
   List L=(List)malloc(sizeof *L);  ？？？
   L->first=0;
   return L;
}
```

2、判断当前表L是否为空表

```
int ListEmpty(List L)
{
   return L->first==0;
}
```

3、求表长函数

```
int ListLength(List L)
{
   int len=0;
   link p=L->first;
   while(p){
     len++;
     p=p->next;
   }
   return len;
}
```

时间复杂性为：O(n)

4、获取链表L中的某个数据元素的内容

```
ListItem ListRetrieve(int k,List L)
{
   if(k<1)exit(1);
   link p=L->first;
   int i=1;
   while(i<k && p){
      p=p->next;
      i++;
    }
   return p->element;
}
```

时间复杂性为：O(n)

5、定位元素x (查找运算)

查找单链表中是否存在结点X，若有则返回X结点的位置；否则返回0；

```
int ListLocate(ListItem x,List L)
{
   int i=1;
   link p=L->first;
   while(p && p->element!=x){
      p=p->next;
      i++;
   }
   return p?i:0;
}
```

//如果p为空，说明x不存在返回0；否则返回其位置i

算法评价

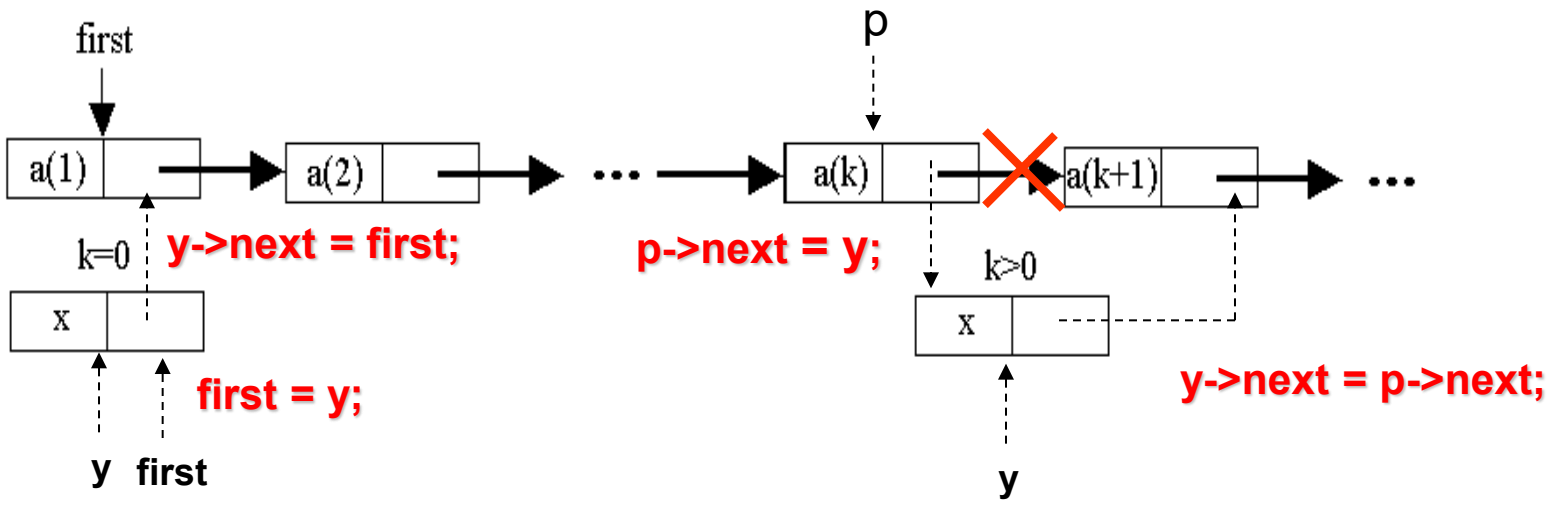
While循环中语句频度为 $\begin{cases} \text{若找到结点X，为结点X在表中的序号} \\ \text{否则，为n} \end{cases}$

$$\Rightarrow T(n)=O(n)$$

6、在位置k后插入元素x

函数的运算步骤是：先扫描链表找到插入位置p，然后建立一个存储待插入元素x的新结点y，再将结点y插入到结点p之后。

插入的示意图如下：



```
void ListInsert(int k,ListItem x,List L)
{
   if(k<0)exit(1);
   link p=L->first;
   for(int i=1;i<k && p;i++)p=p->next;
   link y=NewNode();
   y->element=x;
   if(k){y->next=p->next;p->next=y;}
   else{y->next=L->first;L->first=y;}
}
```

算法的主要计算时间用于寻找正确的插入位置，故其时间复杂性为O(n)。

7、删除位置k处的元素x

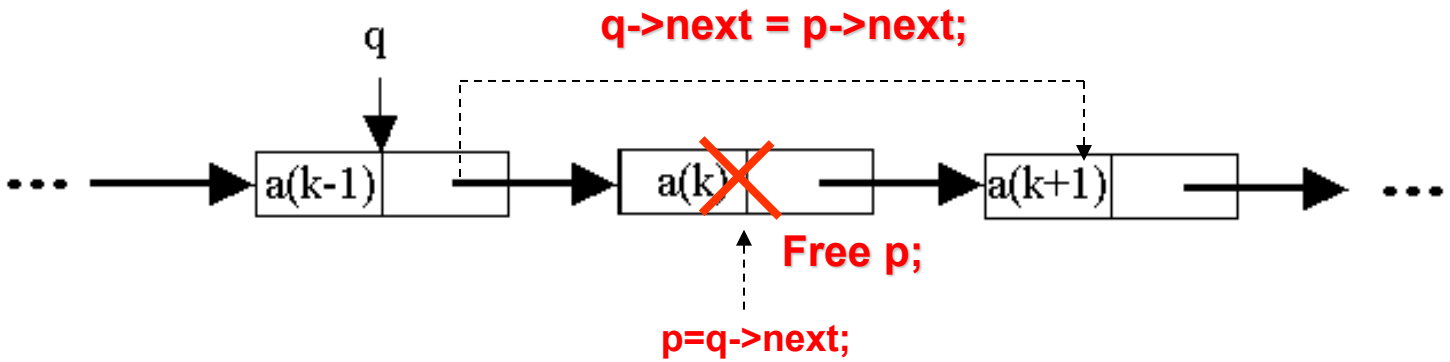
函数的运算步骤是：依次处理以下3种情况。

(1) k<1或链表为空 → 给出越界信息；

(2) 删除的是表首元素，即k=1 →

直接修改表首指针**first**，删除表首元素

(3) 删除的是非表首元素，即k>1，则先找到表中第k-1个元素所在结点q，然后再修改结点q的指针域，删除第k个元素所在的结点p。其删除的示意图如下：



```
ListItem ListDelete(int k,List L)
{
  if(k<1 || !L->first)exit(1);
  link p=L->first;
  if(k==1)L->first=p->next;
  else{
    link q=L->first;
    for(int i=1;i<k-1 && q;i++)q=q->next;
    p=q->next;
    q->next=p->next;
  }
  ListItem x=p->element;
  free(p);
  return x;
}
```

算法主要时间用于寻找待删除元素所在结点，故其所需时间为O(n)

8、输出链表中所有元素:

```
void  PrintList (List  L)
{
   link  p;
   for (p = L->first; p; p = p->next)
       ItemShow(p->element);
  }
```

时间复杂性 O(n)

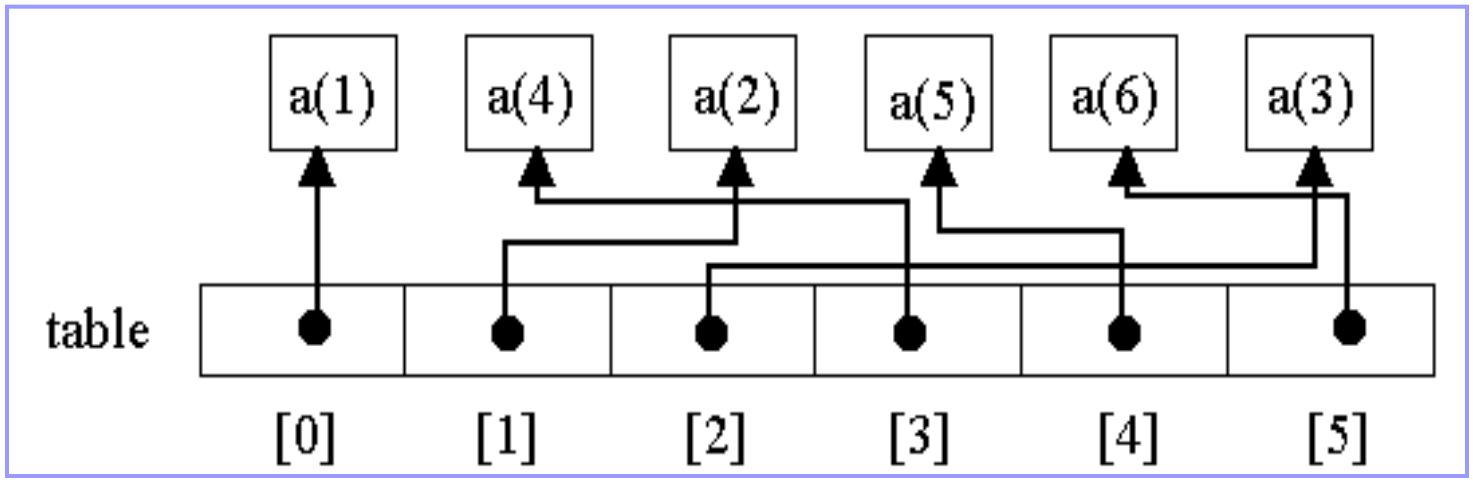
2.3.4 单链表的特点

- 优点
  - 它是一种动态结构，整个存储空间为多个链表共用
  - 不需预先分配空间（动态生成链表）
  - 避免了数组要求连续的存储单元的缺点，因而在执行插入或删除运算时，不再需要移动元素来腾出空间或填补空缺。
- 缺点
  - 指针占用额外存储空间
  - 不能随机存取，查找速度慢

2.4 用间接寻址方法实现表

2.4.1 用间接寻址方法实现表的原因及实现

- 原因：综合数组和指针实现两者的优点。
- 实现：将数组和指针两种实现方式结合起来，让数组中原来存储元素的地方改为存储指向元素的指针。
- 其结构图如下：



- 用间接寻址方法实现表的结构说明如下：

```
typedef struct indlist *List;/* 表指针类型 */
typedef struct indlist{/* 表结构类型 */
    int n,/* 表长 */
        curr;/* 当前位置 */
    int maxsize;/* 数组上界 */
    addr *table;/* 存储表元素指针的数组 */
}Indlist;
```

- 创建一个最大长度为**size**的空表

```
List ListInit(int size)
{
   List L=(List)malloc(sizeof *L);
   L->n=0;
   L->maxsize=size;
   L->table=(addr *)malloc(size*sizeof(addr));
   return L;
}
```

- 判断当前表L是否为空表

```
int ListEmpty(List L)
{
   return L->n==0;
}
```

- 求表长

```
int ListLength(List L)
{
   return L->n;
}
```

时间复杂性*O(1)*

- 返回表**L**的位置**k**处的元素内容

```
ListItem ListRetrieve(int k,List L)
{
   if(k<1 || k>L->n)exit(1);/* 越界 */
   return *L->table[k-1];
}
```

时间复杂性*O(1)*

- 返回元素**x**在表**L**中的位置

```
int ListLocate(ListItem x,List L)
{
   for(int i=0;i<L->n;i++)
     if(*L->table[i]==x)return ++i;
   return 0;
}
```

最坏情况下，时间复杂性*O(n)*

- 在位置**k**后插入元素**x**

```
void ListInsert(int k,ListItem x,List L)
{
   if(k<0 || k>L->n)exit(1);/* 越界 */
   for(int i=L->n-1;i>=k;i--)L->table[i+1]=L->table[i];
   L->table[k]=NewNode();
   *L->table[k]=x;
   L->n++;
}
```

- 算法评价
  - 与数组实现表的情形类似，算法**ListInsert(k,x,L)**将位于**k+1，…，n**处的元素指针分别移到位置**k+2，…，n+1**处，然后将指向新元素**x**的指针插入位置**k+1**处。
  - 与数组实现表的不同点是这里不实际移动元素而只移动指向元素的指针。虽然该算法所需的计算时间仍为***O(n-k)***，但在每个元素占用的空间较大时，该算法比数组实现的表的插入算法快得多。

- 删除位置**k**处的元素给**x**

```
ListItem ListDelete(int k,List L)
{
   if(k<1 || k>L->n)exit(1);/* 越界 */
   addr p=L->table[k-1];
   ListItem x=*p;
   for(int i=k;i<L->n;i++) L->table[i-1]=L->table[i];
   L->n--;
   free(p);
   return x;
}
```

时间复杂性*O(n)*

- 输出表中所有的元素

```
void PrintList(List L)
{
   for (int i=0;i<L->n;i++)ItemShow(*L->table[i]);
}
```

➔ 2.4.2 用间接寻址实现表的优缺点

- 优点
  - 与用数组实现一样，可以方便地随机存取表中任一位置的元素。
  - 与用指针实现一样，在执行插入或删除运算时，不需要移动元素来腾出空间或填补空缺。
- 缺点
  - 与用数组实现一样，需要预先确定**table**的大小。当表长变化很大时，这比较难。
  - 与用指针实现一样，需要额外的存储空间，即额外的指针数组**table**。

2.5 用游标实现表

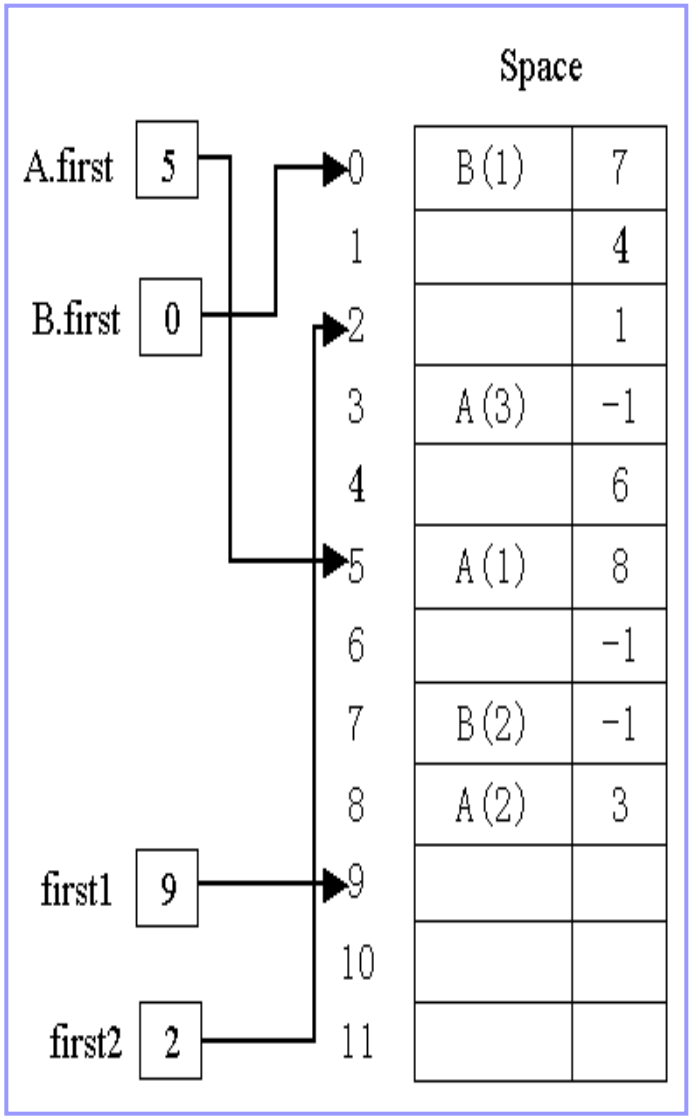
2.5.1 用游标实现表原因及实现

原因：对于有多个同类表的应用，希望通过自主调济内存资源，达到资源的更有效合理的利用。

实现：从操作系统申请一个较大的数组S，然后自主地支配S中的单元，在S中用游标模拟指针实现表，并让多个同类的表共享这个数组，如右图。数组S[12]存放着相同类型的两个表A和B,其中表A含有3个元素；表B含有2个元素。

如图：first1指示S中尚未被使用的子段的起始单元；

first2指示S中被使用过、但目前处于闲置的单元组成的一条可管理的链。让其中的单元优先于first1指示的子段中的单元满足应用的需要。



2.5.2 用游标实现表的优缺点

优点:可实现多个同类的表共享同一片连续存储空间，给用户予资源调济的自主权。

缺点:应该向操作系统申请多大的连续存储空间依赖于具体的应用，不容易把握。

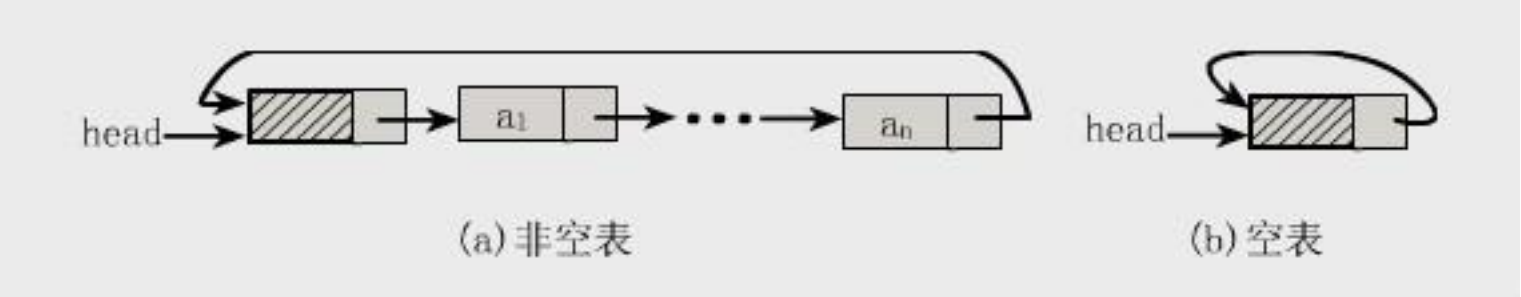
2.6 循环链表

2.6.1 单循环链表

- 循环链表：是一种首尾相接的链表。
- 特点：无需增加存储量，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。从表中任一结点出发均可找到表中其他结点，提高查找效率。
- 单循环链表：在单链表中，将终端结点的指针域NULL改为指向第一个结点，就得到了单链形式的循环链表，并简称为单循环链表。

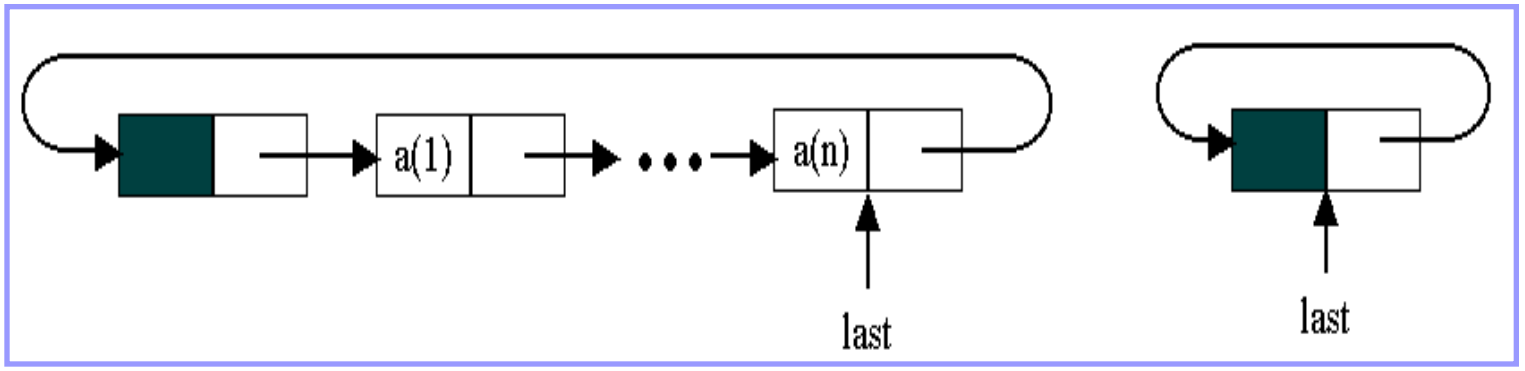
2.6.1 单循环链表

- 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：



- 单循环链表上实现表的各种运算的算法与单链表基本一致，只是循环条件不同
  - 单链表p或p->next=NULL
  - 循环链表p->next=head

- 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)*。
- 如果改用尾指针last来表示单循环链表，则查找开始结点a_1和终端结点a_n都很方便，它们的存储位置分别是(last–>next)—>next和last，显然，查找时间都是*O(1)*。
- 因表的操作多在表的首尾位置上进行，故实用中多采用尾指针表示单循环链表。

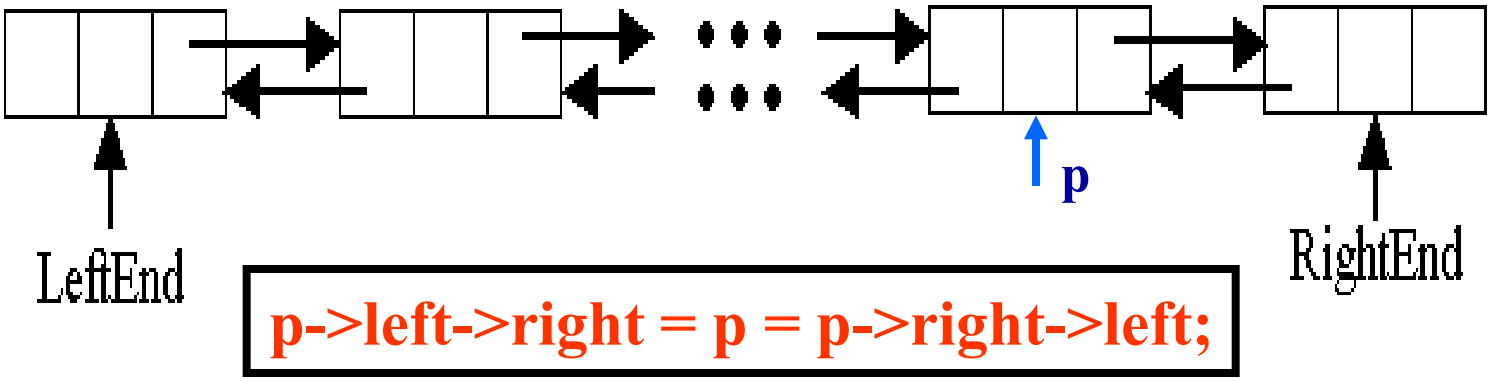


(a)非空表　　(b)空表

2.7 用双链实现表

2.7.1 双向链表

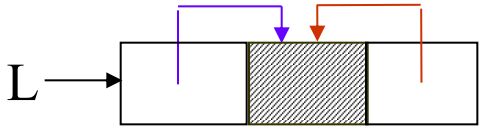
- 原因：用单循环链表实现表时，虽然从表的任一结点出发都可以找到其前驱结点，但需要*O(n)*时间。为了能在*O(1)*时间里既能找到前驱又能找到后继，提出了双链表。
- 在单链表的每一个结点中增加一个指向其直接前驱的指针。这样形成的链表中有两个方向不同的链，故称为双向链表。结构如下图：



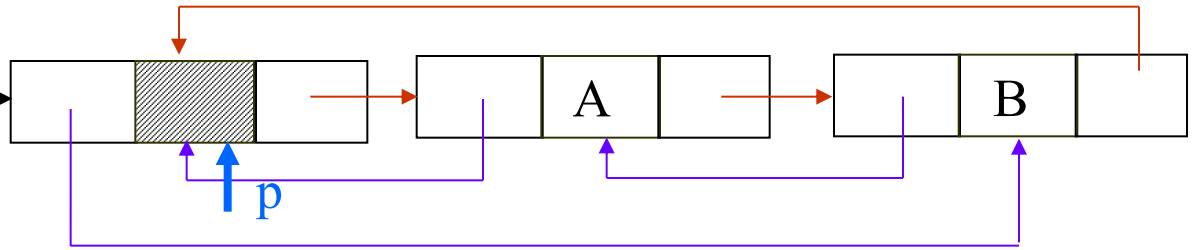
2.7.2 双向循环链表

- 和单循环链表类似，双链表也可以有循环表。用一个表首哨兵结点header将双链表首尾相接，即将表首哨兵结点中的left指针指向表尾，并将表尾结点的right指针指向表首哨兵结点。

空双向循环链表：

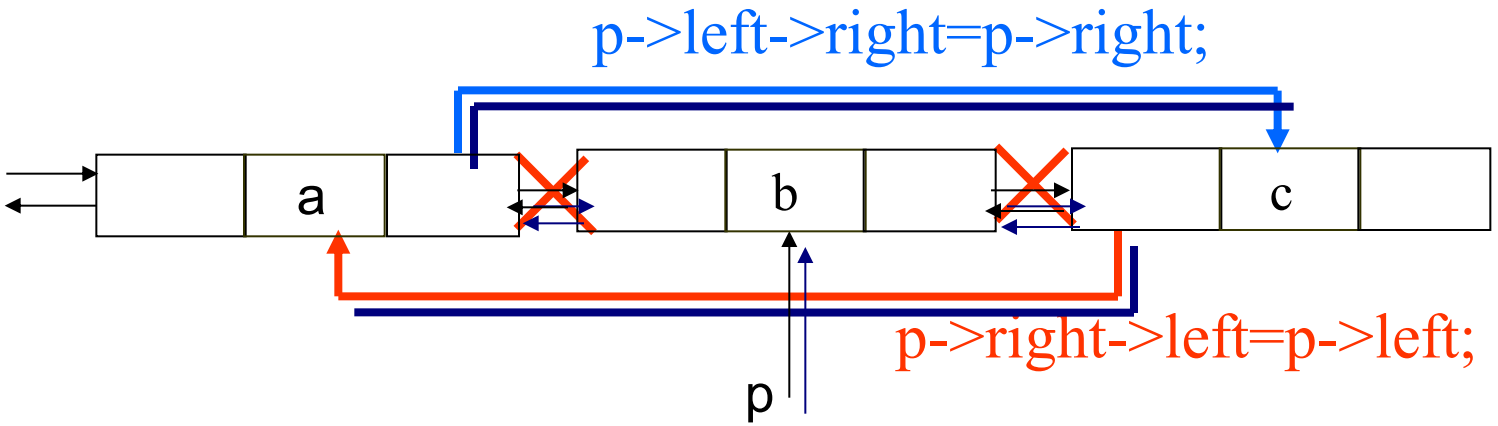


非空双向循环链表：




```
p->left->right = p = p->right->left;
```

- 双向链表中删除结点p，应怎样修改指针？

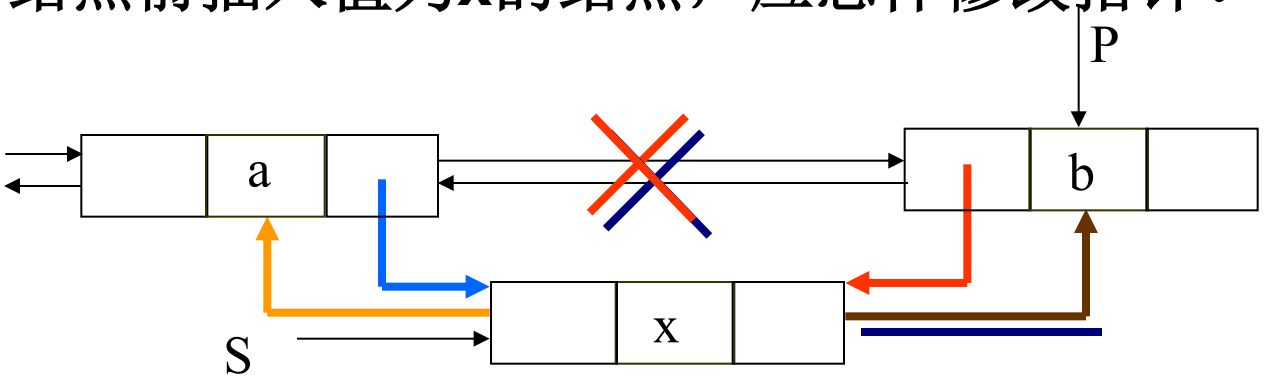


- 算法描述

```
p->left->right=p->right;
p->right->left=p->left;
free(p);
```

- 算法评价：T(n)=*O(1)*

- 在p所指结点前插入值为x的结点，应怎样修改指针？



- 算法描述

```
s->left=p->left;
p->left->right=s;
s->right=p;
p->left=s;
```

- 算法评价：T(n)=*O(1)*

2.8 线性表的应用举例：一元多项式的表示及相加

一元多项式的表示

$$P_n(x) = P_0 + P_1x + P_2x^2 + \cdots\cdots + P_nx^n$$

可用系数线性表P表示 $P = (P_0, P_1, P_2, \cdots\cdots, P_n)$

但对S(x)这样的多项式浪费空间 $S(x) = 1 + 3x^{1000} + 2x^{20000}$

一般 $P_n(x) = P_1x^{e1} + P_2x^{e2} + \cdots\cdots + P_mx^{em}$

其中 $0 \le e1 \le e2 \cdots\cdots \le em$（$P_i$为非零系数）

用数据域含两个数据项的线性表表示

$$((P_1, e1), (P_2, e2), \cdots\cdots (P_m, em))$$

其存储结构可以用顺序存储结构，也可以用单链表

- 单链表的结点定义

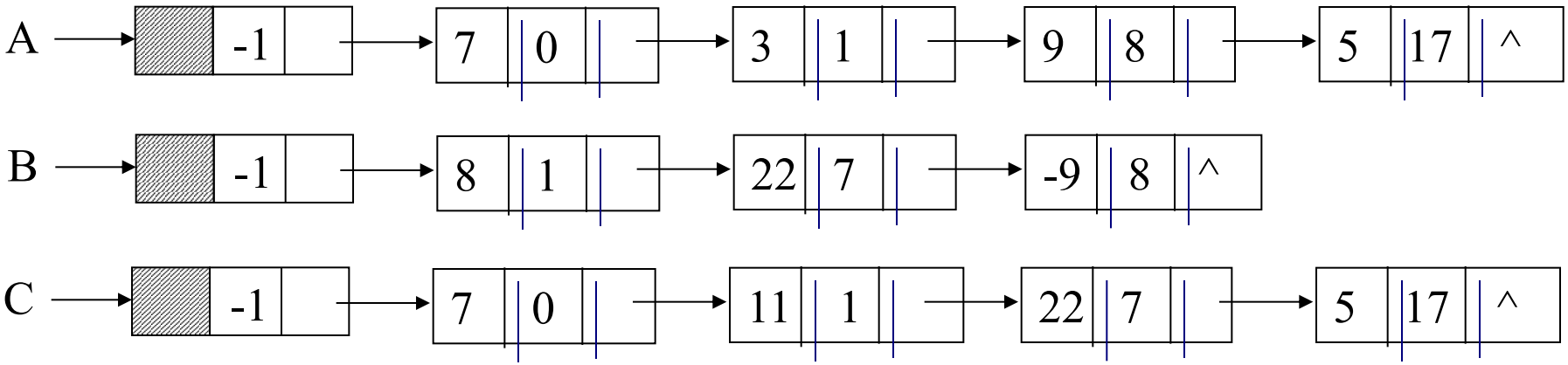
```
typedef   struct node
{   int coef,exp;
    struct node  *next;
}JD;
```

- 一元多项式相加

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

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

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



- 运算规则

设p,q分别指向A,B中某一结点，p,q初值是第一结点

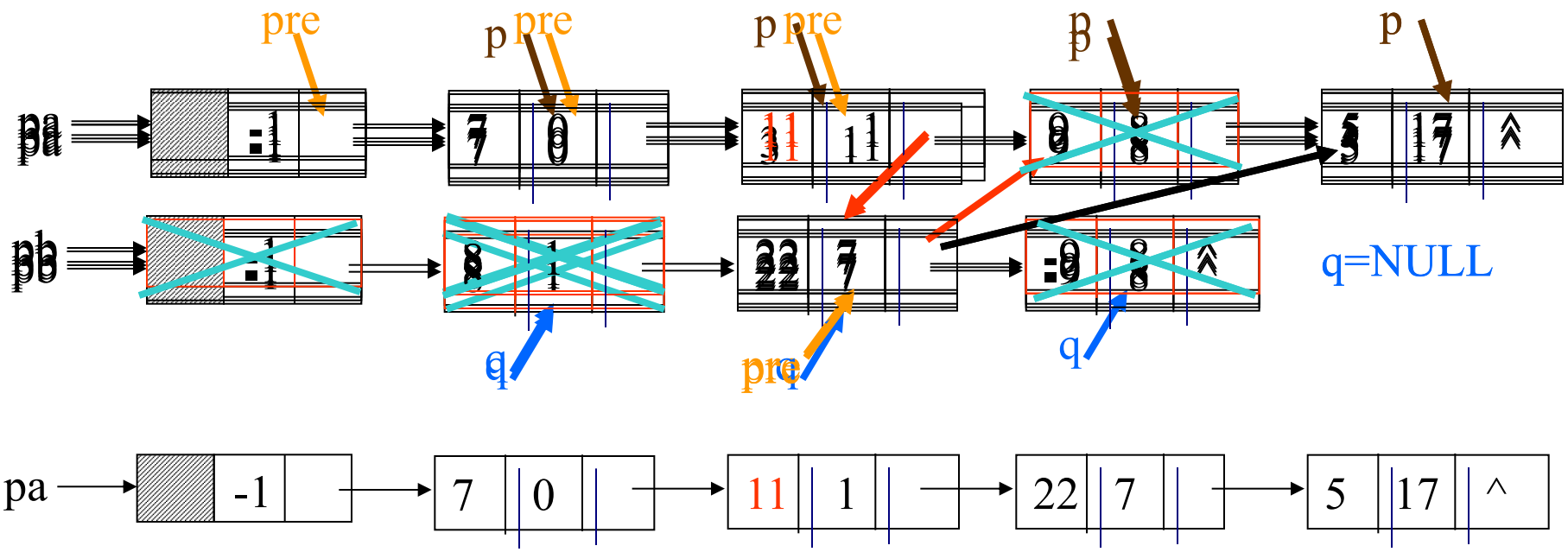
比较 p->exp与q->exp：

- p->exp < q->exp：p结点是和多项式中的一项，p后移,q不动
- p->exp > q->exp：q结点是和多项式中的一项，将q插在p之前,q后移,p不动
- p->exp = q->exp：系数相加
  - 0：从A表中删去p, p,q后移
  - ≠0：修改p系数域, p,q后移

直到p或q为NULL：

- 若q==NULL，结束
- 若p==NULL，将B中剩余部分连到A上即可

- 算法描述


pre
p
pre
p
pre
p
p
pa
pb
-1
7
0
11
1
22
7
5
17
^
q=NULL
q
pre
q
pa
-1
7
0
11
1
22
7
5
17
^

本章小结

- 理解表是由同一类型的元素组成的有限序列的概念。
- 熟悉定义在抽象数据类型表上的基本运算。
- 掌握实现抽象数据类型的一般步骤。
- 掌握用数组实现表的步骤和方法。
- 掌握用指针实现表的步骤和方法。
- 掌握用间接寻址技术实现表的步骤和方法。
- 掌握用游标实现表的步骤和方法。
- 掌握单循环链表的实现方法和步骤。
- 掌握双链表的实现方法和步骤。

表的搜索游标

- P43