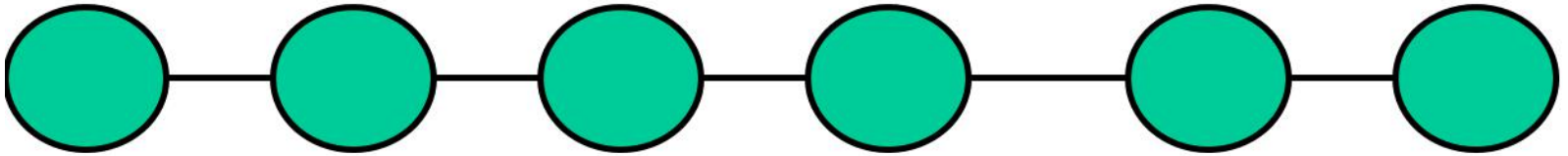
线性结构的特点：



① 只有一个首结点和尾结点；

② 除首尾结点外，其他结点只有一个直接前驱和一个直接后继。

简言之，线性结构反映结点间的逻辑关系是一对一的

线性结构包括线性表、堆栈、队列、字符串、数组等等，其中，最典型、最常用的是------

线性表 → 见第2章

第二章 线性表

- 线性表
- 顺序表（线性表的顺序表示和实现）
- 链表（线性表的链式表示和实现）
- 顺序表与链表的比较

线性表（一种最简单的数据结构）

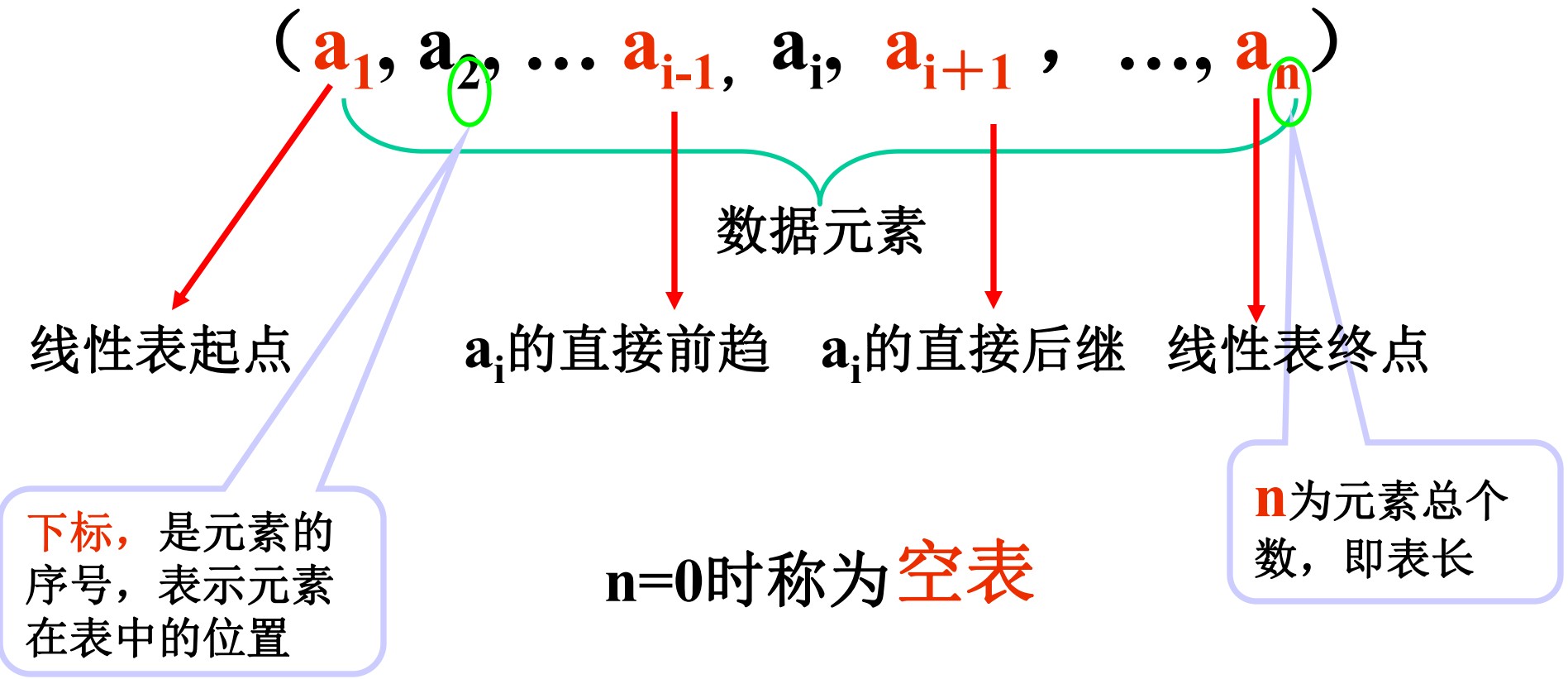
线性表是 一个数据元素的有序（次序）集

定义： n（≥0）个数据元素的有限序列，记作（$a_1, \ldots a_{i-1}, a_i, a_{i+1}, \ldots, a_n$）

其中，a_i 是表中数据元素，n 是表长度。

基本特征：

- 同一线性表中元素具有相同特性。
- 相邻数据元素之间存在序偶关系。
- 除第一个元素外，其他每一个元素有一个且仅有一个直接前驱。
- 除最后一个元素外，其他每一个元素有一个且仅有一个直接后继。


$(a_1, a_2, \cdots a_{i-1}, a_i, a_{i+1}, \cdots, a_n)$
数据元素
线性表起点
a_i的直接前趋
a_i的直接后继
线性表终点
下标，是元素的序号，表示元素在表中的位置
n=0时称为空表
n为元素总个数，即表长

抽象数据类型**线性表**的定义如下：

ADT List {

数据对象：

$D=\{ a_i \mid a_i \in ElemSet, i=1,2,...,n, n\geq 0 \}$

{ 称 **n** 为线性表的**表长**;

称 **n=0** 时的线性表为**空表**。 }

数据关系：

$R1=\{ <a_{i-1}, a_i> \mid a_{i-1}, a_i \in D, i=2,...,n \}$

{ 设线性表为 $(a_1, a_2, \ldots, a_i, \ldots, a_n)$,

称 **i** 为 a_i 在线性表中的**位序**。 }

基本操作：

结构初始化操作

结构销毁操作

引用型操作

加工型操作

} **ADT** List

初始化操作

InitList(List &L)

操作结果：

构造一个空的线性表 L。

结构销毁操作

DestroyList(List &L)

初始条件：线性表 L 已存在。

操作结果：销毁线性表 L。

访问型操作：

ListEmpty(List L)

ListLength(List L)

PriorElem(List L, ElemType cur_e, ElemType &pre_e)

NextElem(List L, ElemType cur_e, ElemType &next_e)

GetElem(List L, int i, ElemType &e)

LocateElem(List L, ElemType e, compare( ))

ListTraverse(List L, visit( ))

ListEmpty(List L)
（线性表判空）

初始条件：线性表 L 已存在。

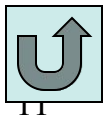
操作结果：若 L 为空表，则返回TRUE，否则FALSE。

ListLength(List L)

(求线性表的长度)

初始条件：线性表 **L** 已存在。

操作结果：返回 **L** 中元素个数。



PriorElem(List L, ElemType cur_e, ElemType &pre_e)

（求数据元素的前驱）

初始条件：线性表 L 已存在。

操作结果：若 cur_e 是 L 的元素，则用 pre_e 返回它的前驱，否则操作失败，pre_e无定义。

NextElem(List L, ElemType cur_e, ElemType &next_e)

（求数据元素的后继）

初始条件：线性表 L 已存在。

操作结果：若 cur_e 是 L 的元素，则用 next_e 返回它的后继，否则操作失败，next_e无定义。

GetElem(List L, int i, ElemType &e)

（求线性表中某个数据元素）

初始条件：线性表 L 已存在，

且 1≤i≤LengthList(L)

操作结果：用 e 返回L中第 i 个元素的值。

LocateElem(List L, ElemType e, compare( ))

(定位函数)

初始条件：线性表 L 已存在，e 为给定值，compare( ) 是元素判定函数。

操作结果：

返回 L 中第 1 个与 e 满足关系 compare( ) 的元素的位序。

若这样的元素不存在，则返回值为 0。

ListTraverse(List L, visit( ))

(遍历线性表)

初始条件： 线性表 L 已存在。

Visit() 为某个访问函数。

操作结果： 依次对 L 中每个元素调用函数visit( )。一旦 visit( )失败，则操作失败。

加工型操作

ClearList(List &L)

PutElem(List &L, int i, ElemType &e)

ListInsert(List &L, int i, ElemType e)

ListDelete(List &L, int i, ElemType &e)

ClearList(List &L)

（线性表置空）

初始条件： 线性表 L 已存在。

操作结果： 将 L 重置为空表。

PutElem(List &L, int i, ElemType &e)

（改变数据元素的值）

初始条件：线性表 L 已存在，

且 1≤i≤LengthList(L)

操作结果：L 中第 i 个元素赋值 e 。

ListInsert(List &L, int i, ElemType e)

（插入数据元素）

初始条件：线性表 L 已存在，

且 1≤i≤LengthList(L)+1

操作结果：在 L 的第 i 个元素之前插入新的元素 e，L 的长度增1。

ListDelete(List &L, int i, ElemType &e)

(删除数据元素)

初始条件：线性表 L 已存在且非空，

1≤i≤LengthList(L)

操作结果：删除 L 的第 i 个元素，并用 e 返回其值，L 的长度减1。

利用上述定义的**线性表类型**

可以实现其它更复杂的操作

如**合并、拆分、复制**等等

算法2.1

假设:有两个集合 **A** 和 **B** 分别用两个线性表 **LA** 和 **LB** 表示，即：线性表中的数据元素即为集合中的成员。

上述问题可演绎为：

要求对线性表作如下操作：

扩大线性表 LA，将存在于线性表 LB 中而不存在于线性表 LA 中的数据元素插入到线性表 LA 中去。

操作步骤：

1. 从线性表 LB 中依次取出每个数据元素;

   GetElem(LB, i)→e

2. 依值在线性表 LA 中进行查访;

   LocateElem(LA, e, equal( ))

3. 若不存在，则插入之。

   ListInsert(LA, n+1, e)

   (n 表示线性表 LA 当前长度)

```
void union(List &La, List Lb) {
    La_len = ListLength(La);     // 求线性表的长度
    Lb_len = ListLength(Lb);
     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, equal( )) )
           ListInsert(La, ++La_len, e);
            // La中不存在和 e 相同的数据元素，则插入之
     }//for
} // union
```

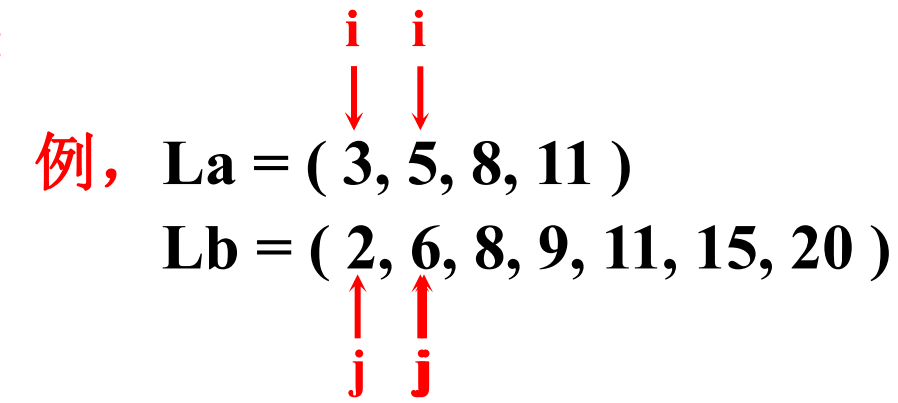
算法时间复杂度

$O($ ListLength(La)×ListLength(Lb) $)$

算法2.2，两个有序线性表La，Lb的合并

要求: 线性表La、Lb中的数据元素按值非递减有序排列，合并La、Lb构造Lc，使Lc中的数据元素仍按值非递减有序排列。

思想:



构造 Lc = (2, 3, 5, 6, 8, 8, 9, 11, 11, 15, 20)

基本操作：

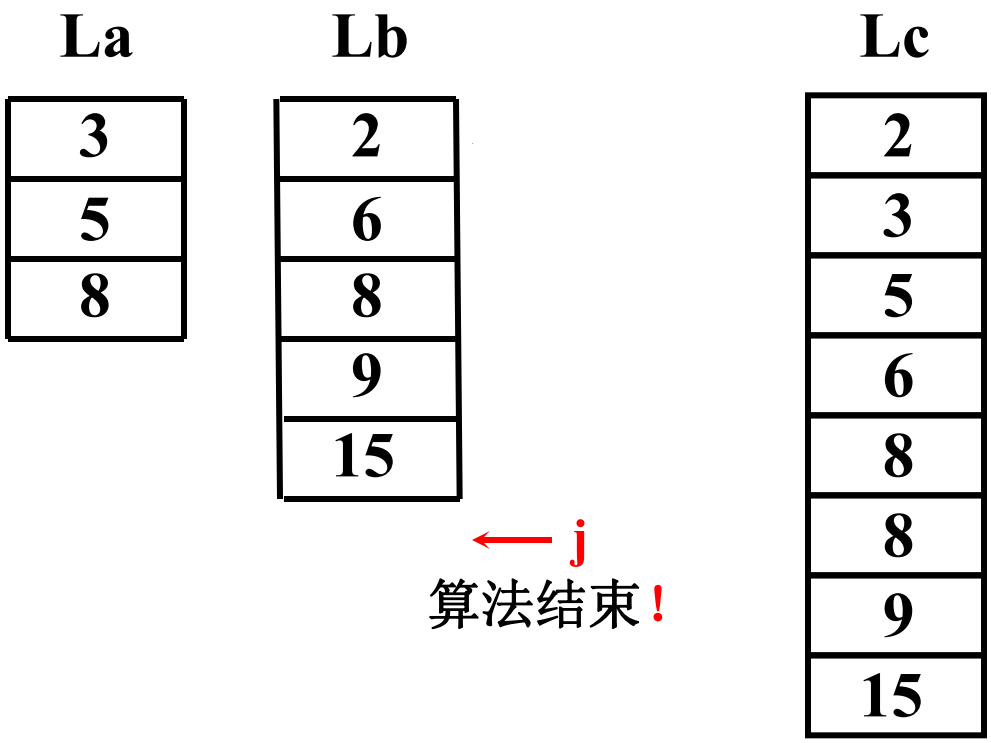
1. 初始化 LC 为空表；
2. 分别从 LA和LB中取得当前元素 a_i 和 b_j；
3. 若 $a_i \leqslant b_j$，则将 a_i 插入到 LC 中，否则将 b_j 插入到 LC 中；
4. 重复 2 和 3 两步，直至 LA 或 LB 中元素被取完为止；
5. 将 LA 表或 LB 表中剩余元素复制插入到 LC 表中。

例，La = (3, 5, 8)

Lb = (2, 6, 8, 9, 15)

构造 Lc = (2, 3, 5, 6, 8, 8, 9, 15)

首先，La_len = 3；Lb_len = 5；



```
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，a )；GetElem ( Lb，j，b );
        if ( a <= b ) { ListInsert( Lc，++k，a )；++i；}
        else { ListInsert( Lc，++k，b )；++j；}
    }
    // 处理超长部分
}
```

```
// 处理超长部分
while ( i <= La_len )    {
    GetElem ( La，i++，a );   ListInsert( Lc，++k，a ) ;
}  // 如果剩余 La

while ( j <= Lb_len )    {
    GetElem ( Lb，j++，b );   ListInsert( Lc，++k，b ) ;
}  // 如果剩余 Lb
```

算法时间复杂度

O(ListLength(La) + ListLength(Lb))

算法时间复杂度

前提：GetElem() 和 ListInsert() 的执行时间与表长无关 LocateElem() 的执行时间与表长成正比

算法2.1

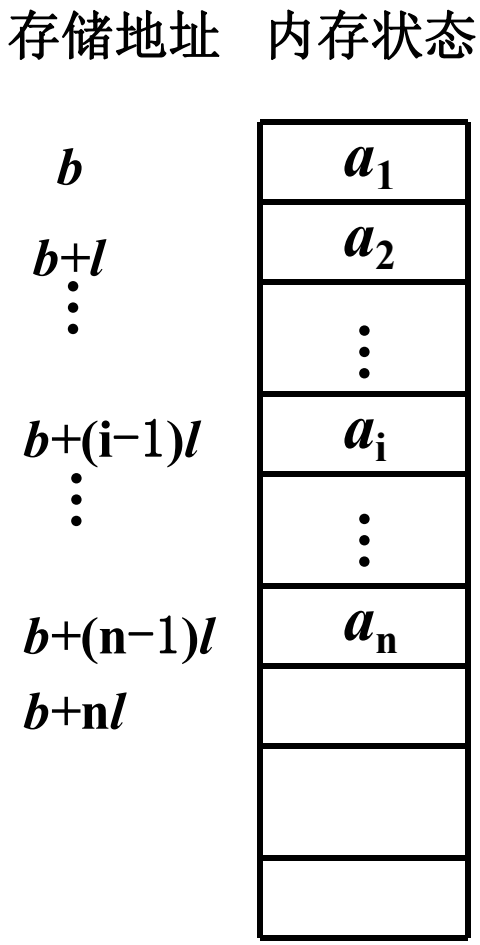
$$O(\text{ListLength(La)} \times \text{ListLength(Lb)})$$

算法2.2

$$O(\text{ListLength(La)} + \text{ListLength(Lb)})$$

2.2 线性表的顺序表示和实现

线性表的顺序表示指的是用一组地址连续的存储单元依次存储线性表的数据元素。



设每个元素需占用 l 个存储单元(字节)

$LOC(a_i)$表示元素a_i的存储地址

则$LOC(a_1)$是第一个数据元素a_1的存储地址，也是整个线性表的起始地址

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

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

特点：

以元素在计算机内的“物理位置相邻”来表示线性表中数据元素之间的“逻辑关系相邻”，线性表的这种计算机表示（实现）称做线性表的顺序存储结构或顺序映像。

只要确定线性表的起始地址，线性表中的任意数据元素都可随意存取，故线性表的顺序存储结构又称为随机存取存储结构。

例，数组数据类型

线性表顺序存储结构表示

```
# define    LIST_INIT_SIZE      100 //初始分配量

# define    LISTINCREMENT    10   //分配增量

typedef    struct {

        Elemtype     *elem；// 存储空间起始地址

        int              length ；  // 表长，初始为 0

        int              listsize ；  // 表存储容量

} SqList ;
```

```
Status   InitList_Sq ( SqList  &L ) {

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

    if  ( ! L.elem )   exit(OVERFLOW) ;
    L.length = 0 ;

    L.listsize = LIST_INIT_SIZE ;

    return  OK ;

}
```

Status == typedef int Status　　参考【P10】

ElemType==顺序表里数据元素的类型。例：typedef struct student ElemType

线性表的顺序存储结构的优缺点

优点：

- 可随机存取表中任意数据元素

  例，**L.elem[i−1]**表示第 **i** 个数据元素

- 直接可获取线性表的长度

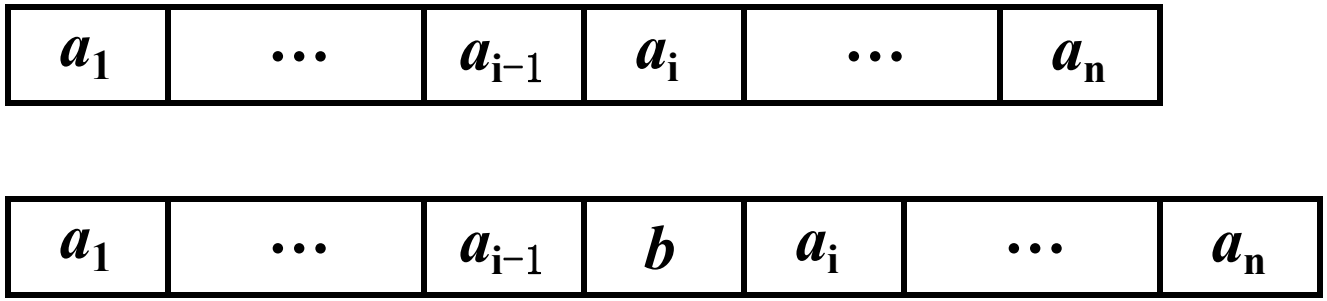
  例，**L.length**表示线性表长度

缺点：

- 数据元素的插入、删除相对麻烦

算法2.4　在第 *i* 个数据元素之前插入一个新的元素

例，在第 i 个元素前插入 b

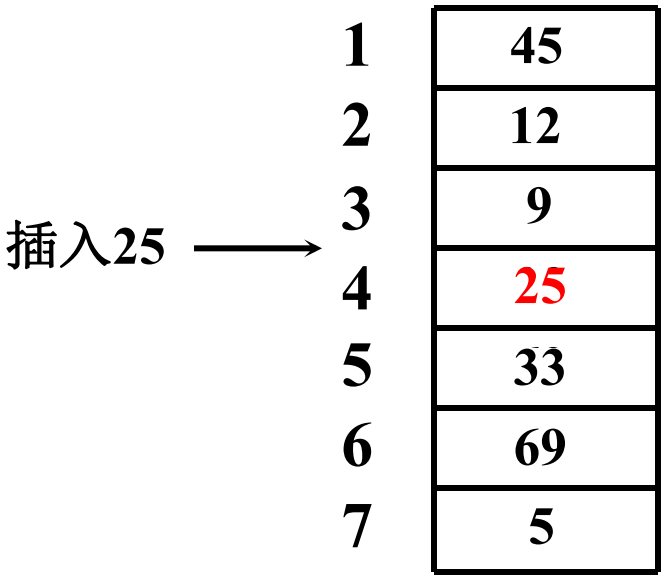


思想：

1. 将第 *n* 到 *i* 个元素均向后移动一个位置。

2. 将新元素放置在第 *i* 个位置。

例，在第4个元素之前插入元素25。



```
Status ListInsert_Sq ( Sqlist &L , int i , ElemType e ) {
    if ( i < 1 || i > L.length +1 ) return ERROR ;
    if ( L.length >= L.listsize )      // 越界
    {
    ....                               // 越界处理;
    }
  q = & ( L.elem[i-1]) ;
  for ( p = & L.elem[L.length-1] ; p >= q ; -- p )
        * (p+1) = * p ;                // 后移元素
  * q = e ;                            // 插入新元素
  ++L. length ;
  return OK ;
}
```

```
// 越界处理
if  ( L.length >= L.listsize )  {
    newbase = ( ElemType * ) realloc ( L.elem ,
                ( L.listsize + LISTINCREMENT ) * sizeof(ElemType) );
   if  ( ! newbase )  exit(OVERFLOW) ;
   L.elem = newbase ;
   L.listsize += LISTINCREMENT ;
}
```

算法时间复杂度:

时间主要花在移动元素上，而移动元素的个数取决于插入元素位置。

position=1，需移动 n 个元素；

position=n+1，需移动 0 个元素；

position=i，需移动 n – i +1 个元素

(i>=1&&i<=n+1);

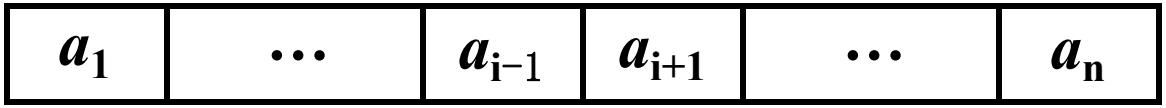
假设p_i是在第 i 个元素之前插入一个新元素的概率

则长度为 n 的线性表中插入一个元素所需移动元素次数的期望值为：$E_{is} = \sum_{i=1}^{n+1} p_i (n - i + 1)$

设在任何位置插入元素等概率，$p_i = \frac{1}{n+1}$

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

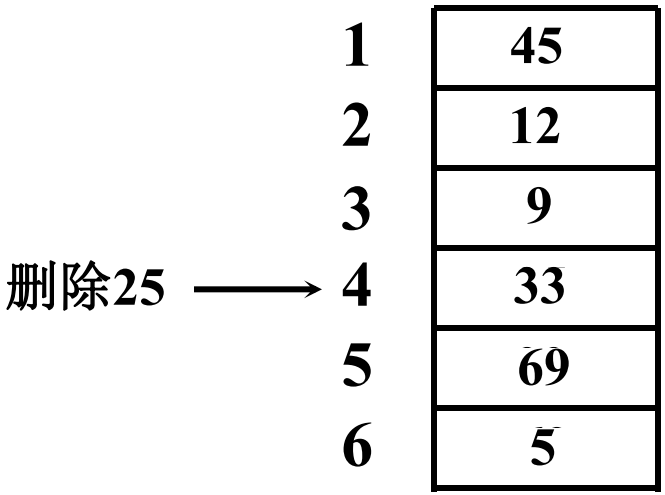
算法2.5　删除第 *i* 个数据元素



思想：

1. 删除第 *i* 个数据元素。
2. 将第 *i*+1 到 *n* 个元素均向前移动一个位置。

例，删除第4个元素25。



```
Status ListDelete_Sq ( Sqlist &L , int i , ElemType &e ) {
    if ( i < 1 || i > L.length ) return ERROR ;
    p = & ( L.elem[i-1]) ;
    e = * p ;                              // 取第 i 个元素的值
    q = &L.elem[ L. length-1] ;
    for ( ++p; p <= q; ++p )
        * (p - 1) = * p ;                  // 前移
    -- L. length;
    return OK ;
}
```

算法时间复杂度:

时间主要花在移动元素上，而移动元素的个数取决于删除元素位置。

position=1，需移动n－1个元素；

position=n，需移动0个元素；

position=i，需移动n – i 个元素；

假设q_i是删除第 i 个元素的概率

则长度为 n 的线性表中删除一个元素所需移动元素次数的期望值为：$E_{dl} = \sum_{i=1}^{n} q_i (n - i)$

设删除任何位置的元素等概率，$q_i = \frac{1}{n}$

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

2.3 线性表的链式表示和实现

线性表的链式存储结构的特点是用一组任意的存储单元存储线性表的数据元素。

存储单元可以是连续的，也可以是不连续的。

为了表示数据元素与其直接后继数据元素之间的逻辑关系，数据元素除了存储本身信息外，还需存储一个指示其直接后继的信息。

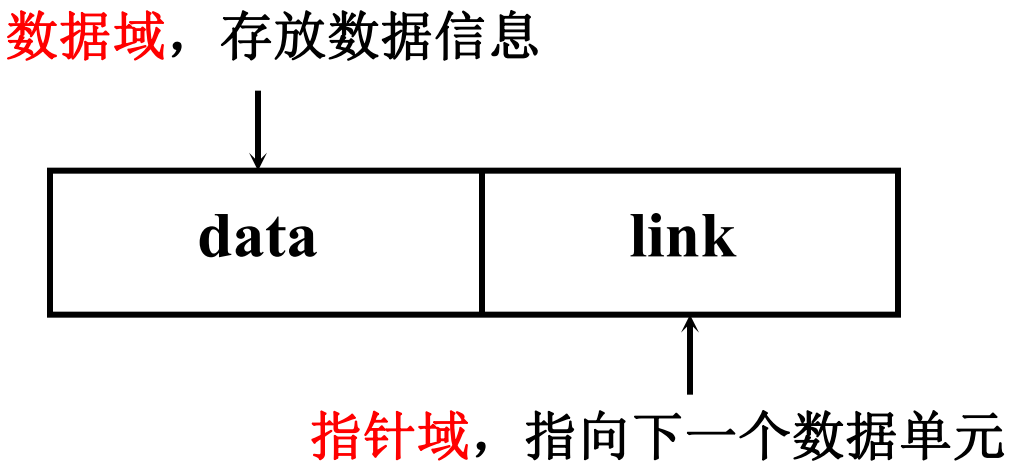
以元素(数据元素的映象)

+ 指针(指示后继元素存储位置)

= 结点

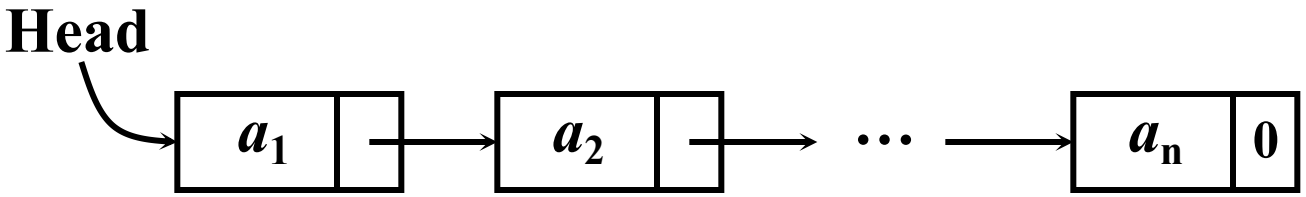
结点：两部分信息组成，存储数据元素信息的数据域，

存储直接后继存储位置信息的指针域。

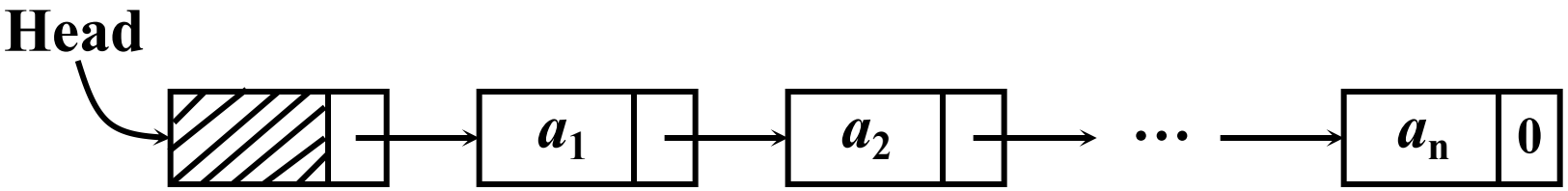


以“结点的序列”表示线性表

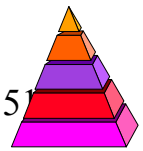
——称作链表



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



线性表的链式存储结构的特点

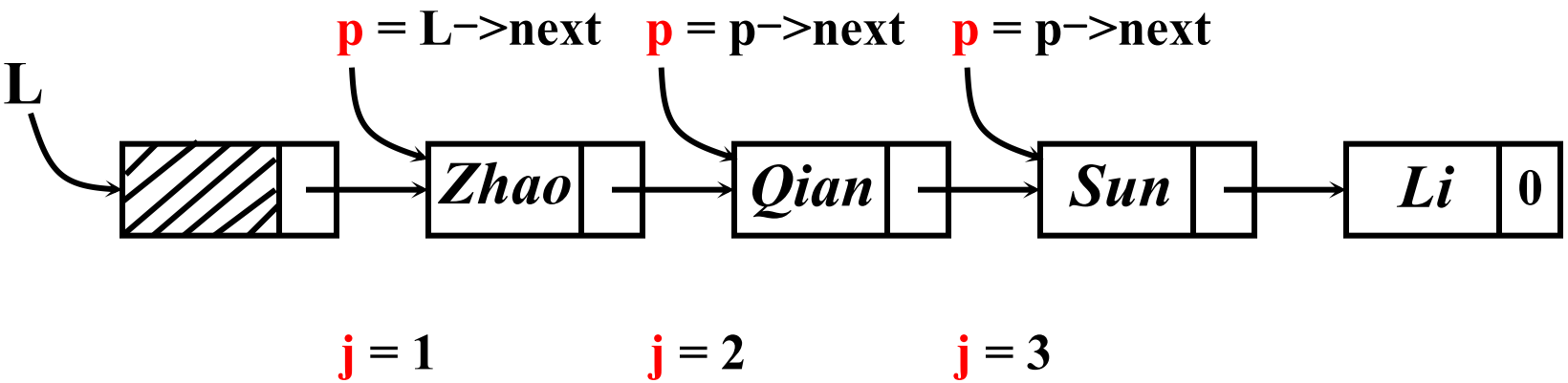
缺点：

- 不可随机存取表中任意数据元素
- 不可直接获取线性表的长度

线性表的单链表存储结构

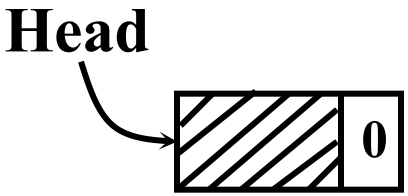
```
typedef   struct   LNode  {
    ElemType          data ;
    struct  LNode     * next ;
} LNode ,  * LinkList ;
```

例，取第i=3个元素。



e = p->data = *Sun*

空表：



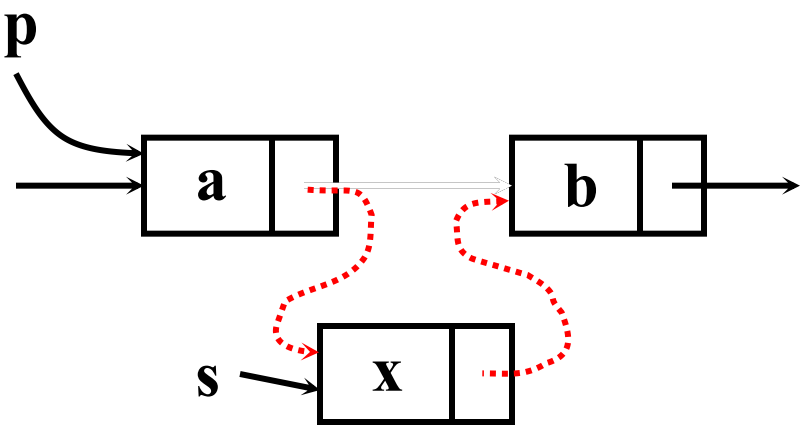
算法2.8：取线性单链表第 i 个元素。

```
Status  GetElem_L ( LinkList  L，  int  i，  ElemType  &e ) {
        p = L->next ；  j = 1 ；
        while  ( p &&  j < i ) {
              p = p->next ；  ++j ；
         }
        if  ( !p || j >i )  return  ERROR ；
        e = p->data ；
        return  OK ；
}
```

时间复杂度： $O(n)$

优点： 数据元素的插入、删除相对方便

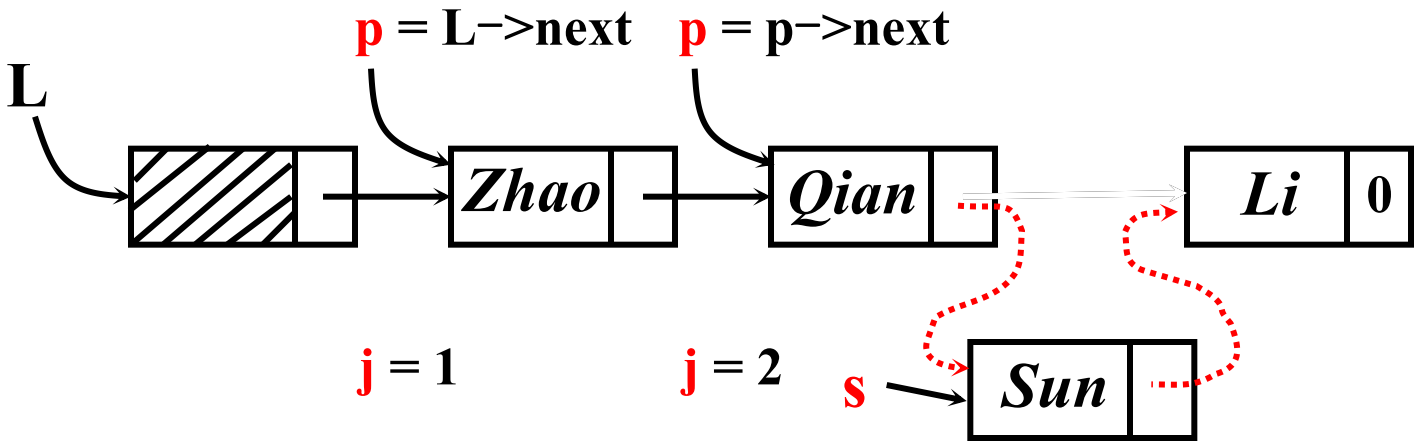
在a，b之间插入元素x：



s->next = p->next

p->next = s

例，在第3个元素之前插入一个新元素。

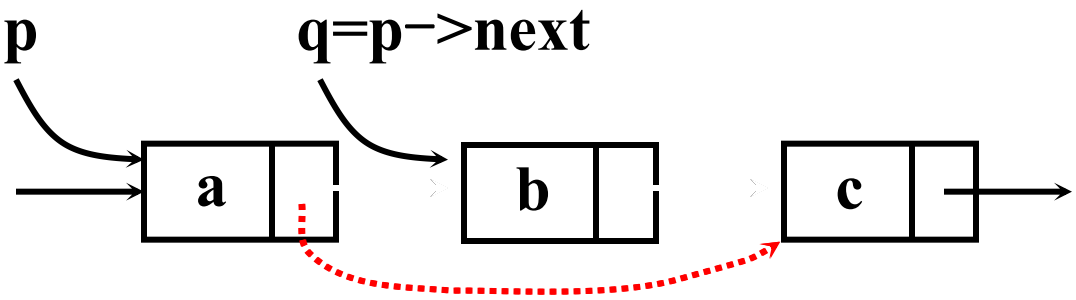


算法2.9　在第 *i* 个数据元素之前插入一个新的元素

```
Status ListInsert_L ( LinkList &L , int i , ElemType e ) {
   p = L ;  j = 0 ;
   while  ( p && j < i − 1 ) { p = p−>next ;  ++j ;  }
                                   //找到第 i 个结点的前驱结点
   if ( ! p || j > i − 1 ) return ERROR ;
   s = ( LinkList ) malloc ( sizeof (LNode) ) ;
   s−>data = e ;                   //建立新结点
   s−>next = p−>next ;
   p−>next = s ;                   //插入新结点
   return OK ;
}
```

时间复杂度:　*O*(n)

删除元素b：

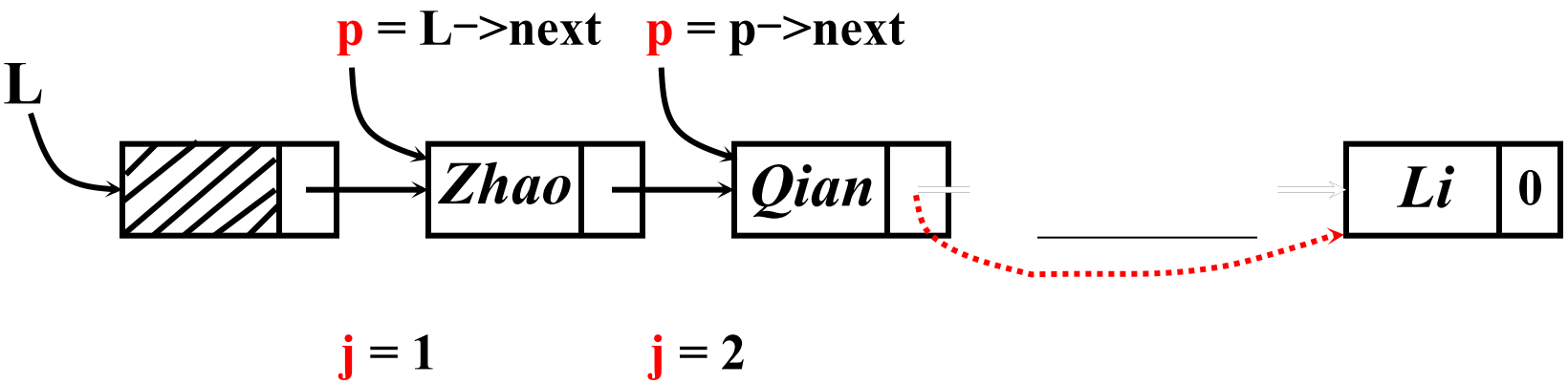


1) p ->next = p ->next ->next ?

2) q = p->next

p->next = q->next

例，删除第3个元素。




```
p->next = q->next ;  //删除第 i 个结点

e = q->data = Sun

free(q) ;
```

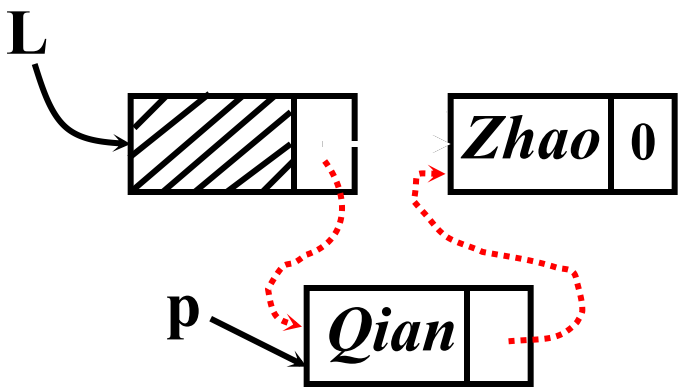
算法2.10　删除第 *i* 个数据元素

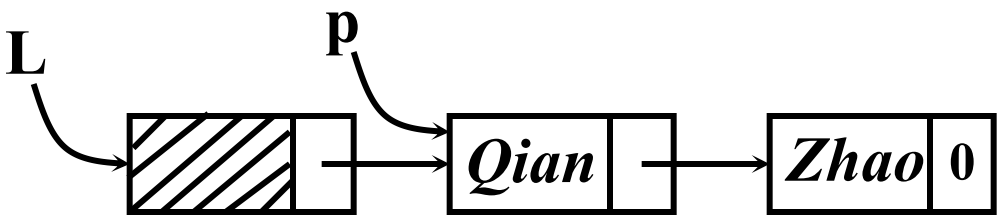
```
Status ListDelete_L ( LinkList &L，int i，ElemType &e ) {
   p = L；j = 0 ；
   while ( p->next && j < i - 1 ) { p = p->next ； ++j ； }
                                    //找到第 i 个结点的前驱结点
   if ( ! p->next || j > i - 1 ) return ERROR ；
   q = p->next ；
   p->next = q->next ； //删除第 i 个结点
   e = q->data ；
   free(q) ；
   return OK ；
}
```

时间复杂度：　*O*(n)

算法2.11　利用插入操作构造一条完整的单链表。

例，



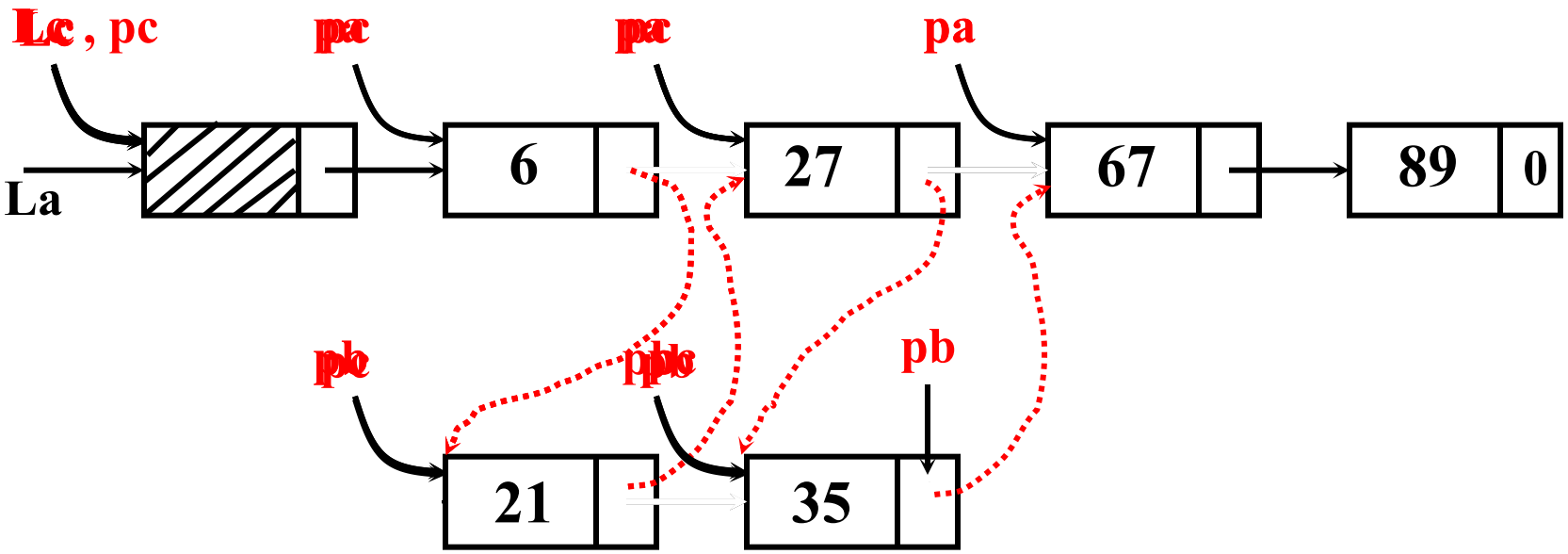


算法2.11　利用插入操作构造一条完整的单链表。

```
void    CreateList_L ( LinkList  &L，int  n ) {
        L = ( LinkList ) malloc ( sizeof (LNode) ) ;
        L->next = NULL ;  //建立头结点
        for  ( i = n ;   i > 0 ;  --i ) {
            p = ( LinkList ) malloc ( sizeof (LNode) ) ;
            Scanf ( &p->data ) ;
            p->next = L->next ;
            L->next = p ;  //在表头插入新结点
        }
}
```

时间复杂度:　　$O(n)$

算法2.12　将两个有序单链表合并为一个有序单链表



算法2.12 将两个有序单链表合并为一个有序单链表

思想： 通过比较不断后移指针合并链表。

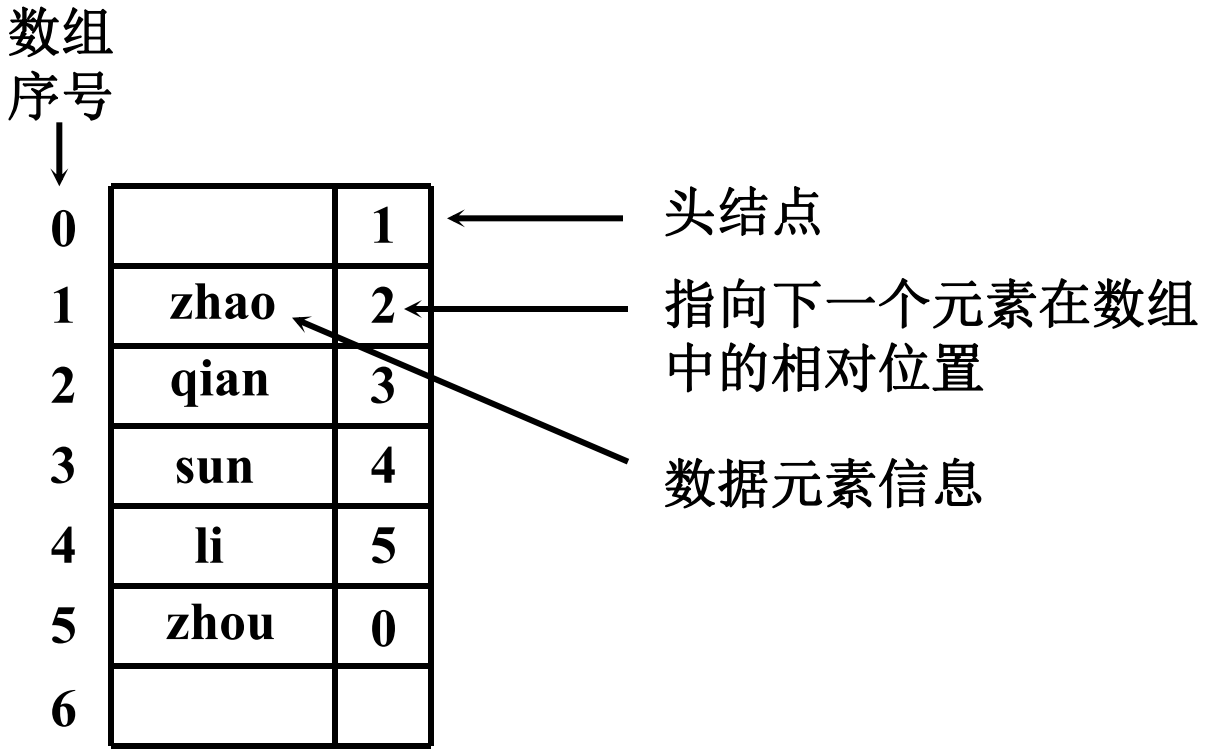
```
void MergeList_L (LinkList &La，LinkList &Lb，LinkList &Lc) {
    pa = La->next ；  pb = Lb->next ；  //分别指向第一个结点
    Lc = pc = La ；
    while ( pa && pb ) {
        if ( pa->data <= pb->data ) {
            pc->next = pa ；  pc = pa ；  pa = pa->next ；  }
        else { pc->next = pb ；  pc = pb ；  pb = pb->next ；  }
    }
    pc->next = pa ? pa : pb ；  //处理剩余部分
    free (Lb) ；
}
```

2.3.2 其他线性链表

- 静态链表
- 循环链表
- 双向链表

1. 静态链表

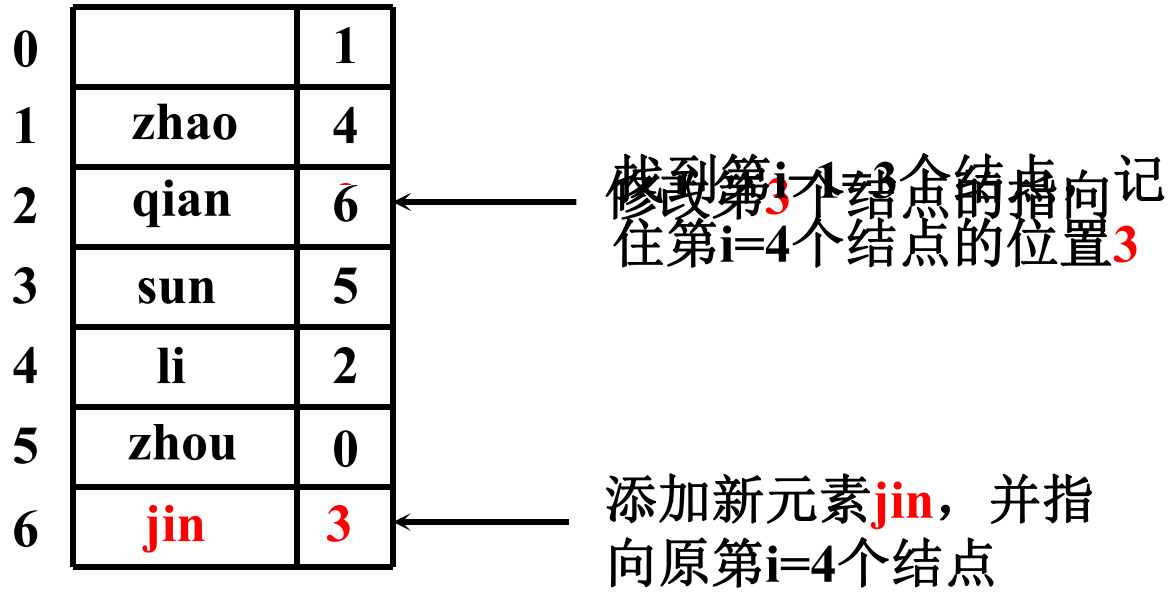
某些语言不支持指针类型，通常使用一维数组描述单链表。



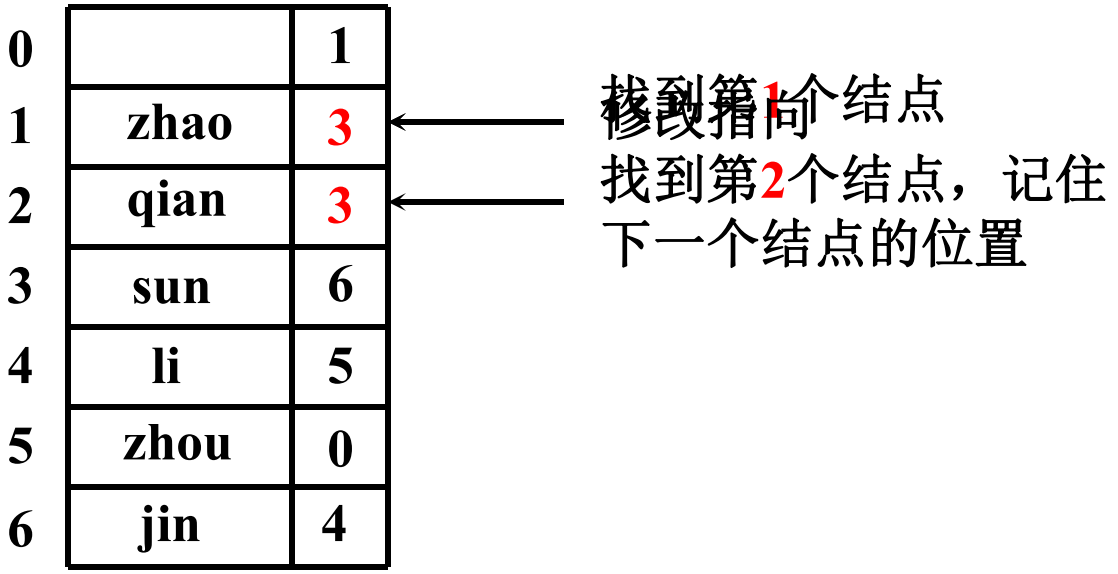
空表：

插入和删除操作

1) 在第i=4个结点前插入新元素 jin

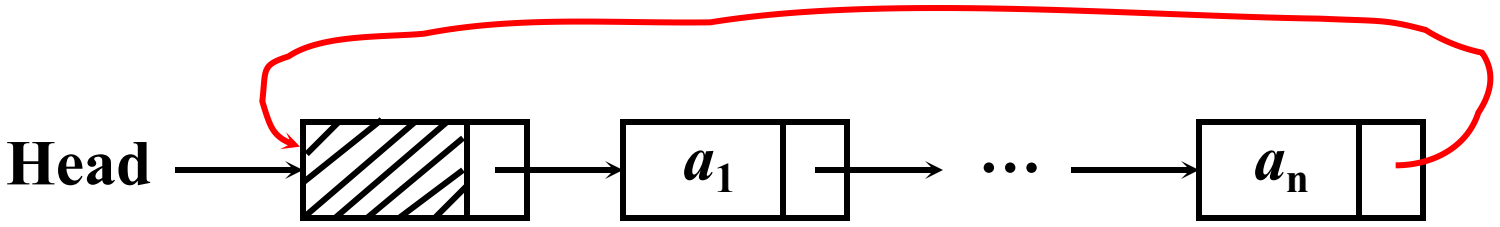


2) 删除第2个结点

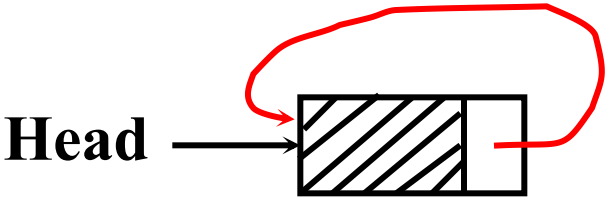

0
1
2
3
4
5
6
1
zhao
3
qian
3
sun
6
li
5
zhou
0
jin
4
找到第1个结点
修改指向
找到第2个结点，记住
下一个结点的位置

2. 循环链表

表中最后一个节点的指针域指向头结点，形成一个环。



空表：



优点：从表的任意结点出发均可以找到表中的其他结点。

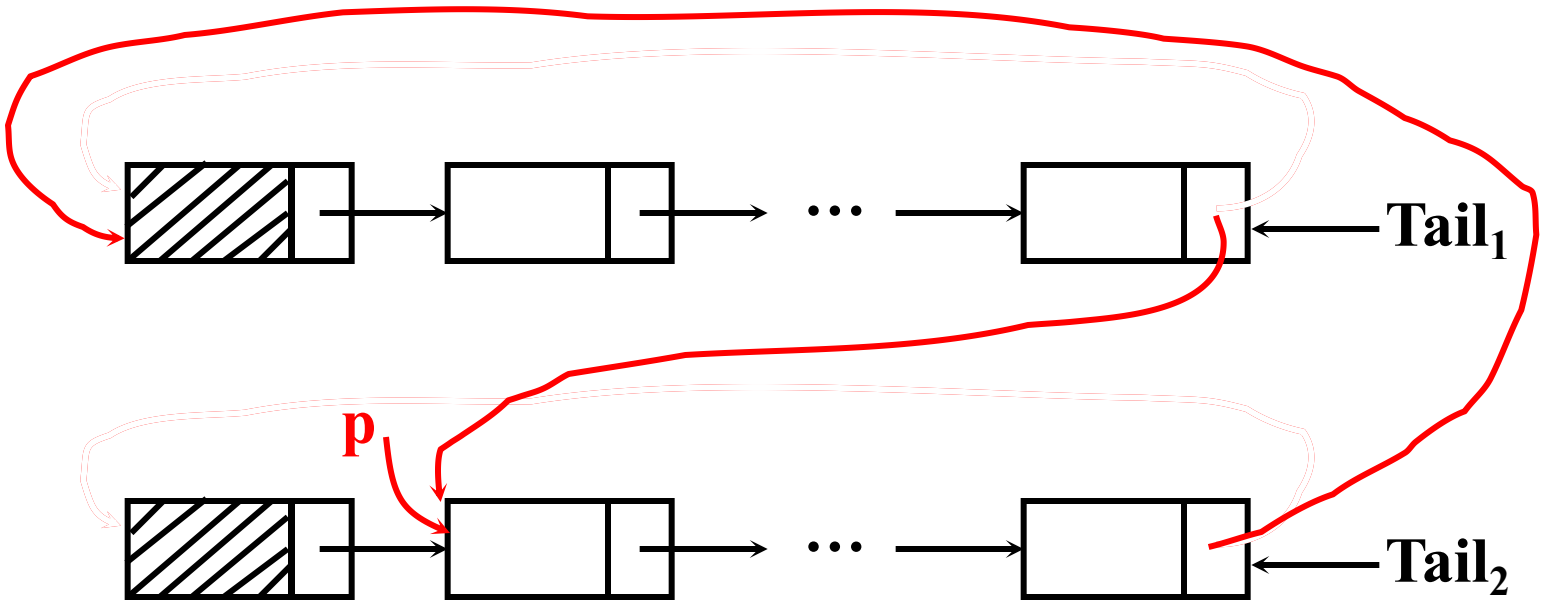
操作与线性单链表基本一致，差别只是在于算法中的循环结束条件不是p是否为空，而是p是否等于头指针。

例，取循环链表第 i 个元素。

```
Status GetElem_L (LinkList L，int i，ElemType &e) {
    p = L->next ; j = 1 ;
    while (p != L && j < i) {
        p = p->next ; ++j ;
    }
    if (p = =L || j > i) return ERROR ;
    e = p->data ;
    return OK ;
}
```

有时为了方便某些操作，通常在循环链表中设立尾指针。

例如：顺次合并两个线性表。



$p = Tail_2->next->next$

$Tail_2->next = Tail_1->next$

$Tail_1->next = p$

3. 双向链表

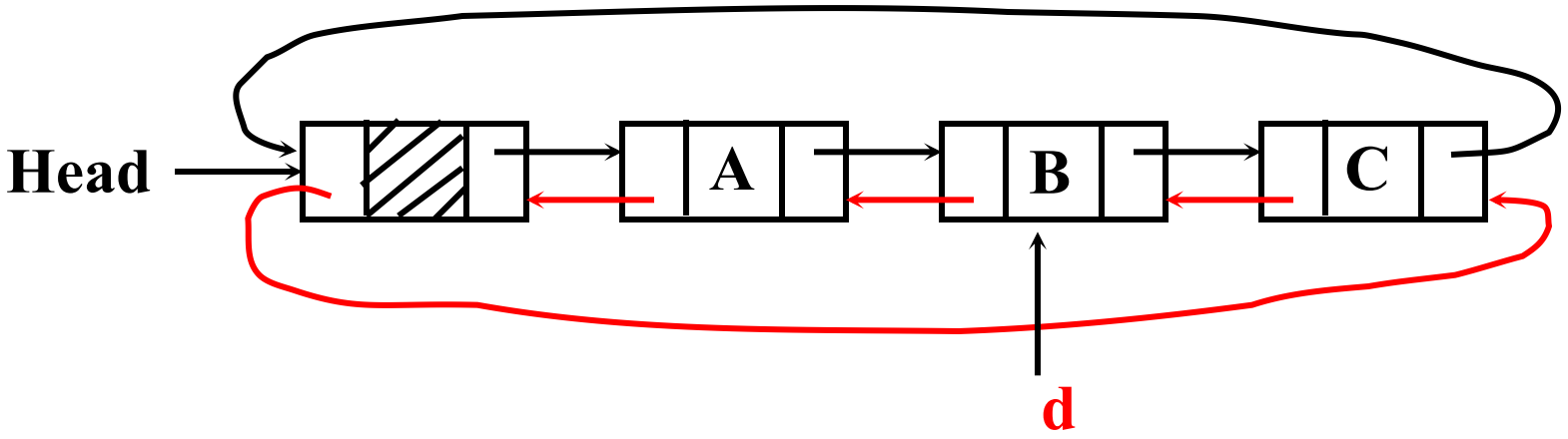
在循环链表中寻找结点的直接后继很简单，只需要 $O(1)$；

但要寻找结点的直接前趋就要从表头指针找起，需要$O(n)$。

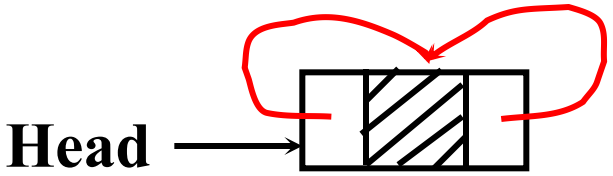
双向链表的结点有两个指针域: 一个指向直接后继，一个指向直接前趋。

类型表示:

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



空表：

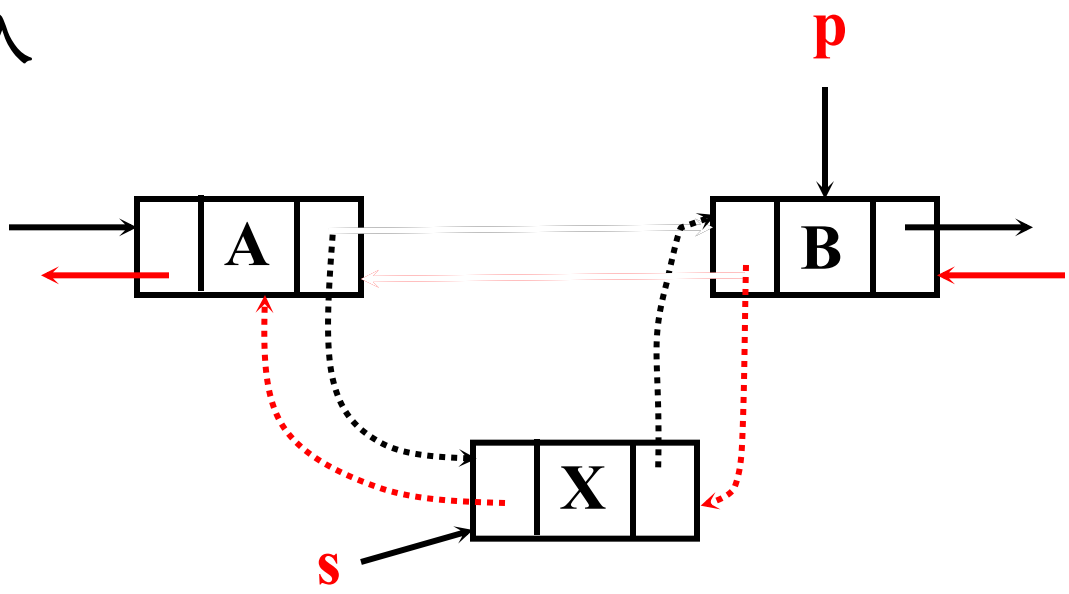


性质：设d 是指向某个结点的指针，则有

d->next->prior = =d->prior->next == d

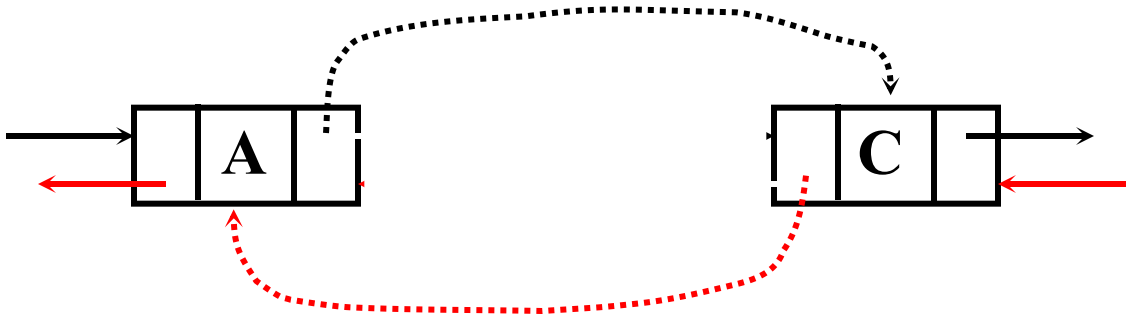
操作：只涉及单向的操作基本相同，但插入、删除操作变化很大。

1) 插入



1. 找到要在之前插入的结点，p记录。
2. s->prior = p->prior ;
3. p->prior->next = s ;
4. s->next = p ;
5. p->prior = s ;

2) 删除



1. 找到要删除的结点，p记录。
2. p->prior->next = p->next ;
3. p->next->prior = p->prior ;
4. free(p) ;

1、在需要经常查找结点的前驱与后继的场合中，使用(　　)比较合适。

A．单链表　　B．双链表　　C．顺序表　　D．循环链表

2、在一个单链表HL中，若要删除由指针q所指向结点的后继结点，则执行（　）。

A．p = q->next ; p->next = q->next;

B．p = q->next ; q->next = p;

C．p = q->next ; q->next = p->next;

D．q->next = q->next->next; q->next = q;

3、已知单链表A长度为m，单链表B长度为n，若将B联接在A的末尾，其时间复杂度应为(　)。

A．O(1)　　B．O(m)

C．O(n)　　D．O(m+n)

4、链表不具有的特点是（　）。

A．可随机访问任一元素

B．插入删除不需要移动元素

C．不必事先估计存储空间

D．所需空间与线性表长度成正比

5、若要在单链表中的结点p之后插入一个结点s，则应执行的语句是(　　)。

A．s->next=p->next; p->next=s;

B．p->next=s; s->next=p->next;

C．p->next=s->next; s->next=p;

D．s->next=p; p->next=s->next;

6、带头结点head的循环链表为空的判断条件是(　　)。

A．head==NULL　　B．head->next==NULL

C．head->next=head　　D．head!=NULL

7、在(　　)运算中，使用顺序表比链表好。

A．插入　　B．删除

C．根据序号查找　D．根据元素值查找

8、带头结点head的单链表为空的判断条件是(　　)。

A．head= =NULL

B．head->next= =NULL

C．head->next=head

D．head!=NULL

11、在链表中进行插入和_____操作的效率比在顺序存储结构中进行相同操作的效率高。

- 下列函数的功能是，对以带头结点的单链表作为存储结构的两个递增有序表（表中不存在值相同的数据元素）进行如下操作：将所有在Lb表中存在而La表中不存在的结点插入到La中，其中La和Lb分别为两个链表的头指针。请在空缺处填入合适内容，使其成为一个完整的算法。

- void union (LinkList La, LinkList Lb)

  {

  //本算法的功能是将所有Lb表中存在而La表中不存在的结点插入到La表中

  LinkList pre = La, q;

  LinkList pa = La -> next;

  LinkList pb = Lb -> next;

  free (Lb);

- while (pa && pb)
{
  if (pa -> data <pb -> data)
    { pre = pa; pa = pa -> next;}
  else
  if (pa -> data > pb ->data)
  {
        (1) ;
        pre = pb;
        pb = pb -> next;
        (2) ;
  }
  else
  {
        q = pb; pb = pb -> next; free (q);
  }
}
if (pb)
    (3)   ;
}

顺序表与链表的比较

基于空间的比较

- 存储分配的方式
  - 顺序表的存储空间是静态分配的
  - 链表的存储空间是动态分配的
- 存储密度 = 结点数据本身所占的存储量/结点结构所占的存储总量
  - 顺序表的存储密度 = 1
  - 链表的存储密度 < 1

顺序表与链表的比较

基于时间的比较

- 存取方式
  - 顺序表可以随机存取，也可以顺序存取
  - 链表是顺序存取的
- 插入/删除时移动元素个数
  - 顺序表平均需要移动近一半元素
  - 链表不需要移动元素，只需要修改指针

2.4 应用举例

一、一元多项式的表示

$$p_n(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_n x^n$$

在计算机中，可以用一个线性表来表示：

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

但是对于形如

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

的多项式，上述表示方法是否合适？

一般情况下的一元稀疏多项式可写成

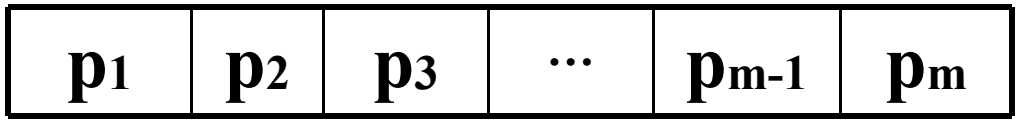
$$P_n(x) = p_1x^{e1} + p_2x^{e2} + \text{---} + p_mx^{em}$$

其中：p_i 是指数为ei 的项的非零系数，

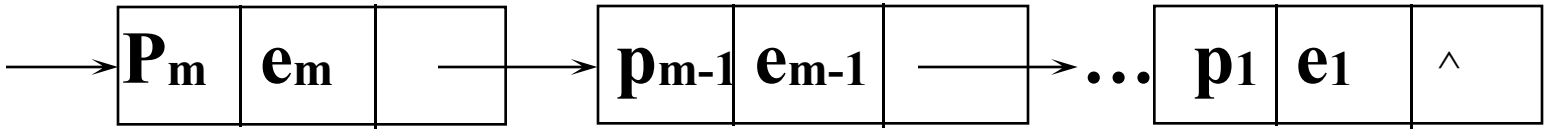
$$0 \le e_1 < e_2 < \text{---} < e_m = n$$

分析：一元多项式在计算机内存储时，既可用顺序表存储，又可用链表存储。但当多项式的次数很高且零系数项很多时，则更适于用链表存储（通常设计两个数据域和一个指针域）。

顺序表



链表



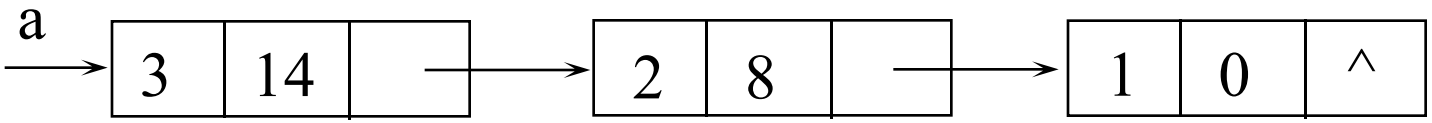
3. 用C语言如何具体描述它的定义？

```
typedef struct poly_node {
    float coef;
    int    expon;
    stuct poly_node *link;
};
typedef struct poly_node *poly_pointer;
poly_pointer a, b, c;
```

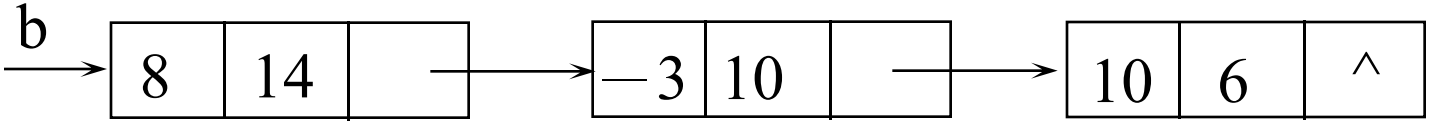
coef	expon	link

如何编程实现两个一元多项式相加？

例：$a = 3x^{14} + 2x^{8} + 1$



$b = 8x^{14} - 3x^{10} + 10x^{6}$



运算规则：两多项式中指数相同的项对应系数相加，若和不为0，则构成多项式c(=a+b)中的一项；a和b中所有指数不相同的项均应复抄到c中。

实现思路：

依次比较Pa和Pb所指结点中的指数项，依

`Pa->expon` 、=、<、`>Pb->expon`等情况，再决定是将两系数域的数值相加（并判其和是否为0），还是将较高指数项的结点插入到新表c中。