目　　录

第一章　线 性 表

1.1　内 容 要 点

线性表（Linear List）是最简单且最常用的一种数据结构。这种结构具有下列特点：存在一个唯一的没有前驱的（头）数据元素；存在一个唯一的没有后继的（尾）数据元素；此外，每一个数据元素均有一个直接前驱和一个直接后继数据元素。

1.1.1　线性表的定义及其运算

1. 线性表的定义

线性表是 n（n≥0）个数据元素 a_1, a_2, Λ, a_n 组成的有限序列。其中 n 称为数据元素的个数或线性表的长度，当 n=0 时称为空表，n>0 时称为非空表。通常将非空的线性表记为（a_1, a_2, Λ, a_n），其中的数据元素 a_i（1≤i≤n）是一个抽象的符号，其具体含义在不同情况下是不同的，即它的数据类型可以根据具体情况而定。

2. 线性表的特征

从线性表的定义可以看出线性表的逻辑特征：

① 有且仅有一个开始结点(表头结点) a_1，它没有直接前驱，只有一个直接后继；

② 有且仅有一个终端结点(表尾结点) a_n，它没有直接后继，只有一个直接前驱；

③ 其他结点都有一个直接前驱和直接后继；

④ 元素之间为一对一的线性关系。

3. 线性表的运算

常见线性表的运算有：

① 置空表（初始化）setnull（&L） 将线性表 L 置成空表。

② 求长度 length（L） 求给定线性表 L 的长度。

③ 取元素 get（L，i） 若 1≤i≤length(L)，则取第 i 个位置上的元素，否则取得的元素为 NULL。

④ 定位函数 locate（L，X） 在线性表 L 中查找值为 X 的元素位置，若有多个值为 X，则以第一个为准，若没有，位置为 0。

⑤ 插入 insert（&L，X，i） 在线性表 L 中第 i 个数据元素之前插入一个值为 X 的新元素。

⑥ 删除 delete（&L，i） 删除线性表 L 中第 i 个元素。当 1≤i≤len 时，删除成功，返回被删元素的值，否则返回 NULL。

对线性表的操作还有很多，比如取前趋、取后继及排序等。

1.1.2 线性表的顺序存储结构

线性表的顺序存储结构，也称为顺序表。其存储方式为：在内存中开辟一片连续存储空间，但该连续存储空间的大小要大于或等于顺序表的长度，然后让线性表中第一个元素存放在连续存储空间第一个位置，第二个元素紧跟着第一个之后，其余依此类推。

可见，在线性表上，逻辑关系相邻的两个元素在物理位置上也相邻——线性表顺序存储的特点。

很容易确定每个数据元素在存储单元中与起始地址的相对位置：假设线性表中元素为（a_1, a_2, Λ, a_n），设第一个元素 a_1 的内存地址为 $Loc(a_1)$，而每个元素在计算机内占 C 个存储单元，则第 i 个元素 a_i 的首地址为 $Loc(a_i)=Loc(a_1)+(i-1)\times C$ (其中 $1\leq i\leq n$)。

显然，顺序表便于进行随机访问，故线性表的顺序存储结构是一种随机存储的结构。

1. 顺序表的结构

下面用 C 语言，给出顺序表的定义：

```
#define MAXSIZE    maxlen
typedef    int elemtype;
typedef struct
{ elemtype vec[maxlen];
    int len;          //顺序表的长度
 }sequenlist;
```

有了线性表的顺序存储结构后，下面就可以讨论如何在这种结构上实现有关数据元素的运算问题。重点讨论线性表元素的插入和删除。

2. 顺序表的基本运算

（1）插入运算

线性表的插入是指在表的第 i 个位置上插入一值为 x 的新元素，通常在第 i 个元素 ($1\leq i\leq n+1$) 插入一个新元素时，需要有 n-i+1 个元素进行移动。通过以下步骤完成顺序表的插入运算：

① 将 a_n ～ a_i 共 n-i+1 个元素依次向下移动，为新元素让出位置；

② 将 x 置入空出的第 i 个位置；

③ 修改 len 值（即修改表长），使之加 1。

算法描述：

```
   int    insert(sequenlist *L, int    i, elemtype x)
{ int j;
   if((*L).len>=maxlen)
      {printf("the list is overflow\n");
       return NULL;
       }
   else if((i<1)||(i>(*L).len+1))
                  { printf("position is not corrent.\n");
                      return NULL;
```

```
                    }
else{     for(j=(*L).len-1;j>=i-1;j--)
                (*L).vec[j+1]=(*L).vec[j];              //元素后移
          (*L).vec[i-1]=x;                    //插入元素 x
          (*L). len++;                        //表长度增加 1
          return 1;
     }
 }
```

性能分析：

插入算法花费的时间，主要在于循环中元素的后移（其他语句花费的时间可以省去），即从插入位置到最后位置的所有元素都要后移一位，在空出的位置插入元素值 x。但是，插入的位置是不固定的，当插入位置 i=1 时，全部元素都得移动，需 n 次移动，当 i=2 时，移动 n-1 个元素，依次类推，当 i=n 时，仅需移动元素一次。设 p_i 为在第 i 个数据元素之前插入一个元素的概率，假设在顺序表的任何位置上插入元素都是等概率的，即 $p_i=1/(n+1)$，则在长度为 n 的顺序表中插入一个元素时所需的平均移动次数为

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

可见在顺序表上做插入操作需移动表中一半的数据元素，显然时间复杂度为 O(n)。

（2）删除运算

线性表的删除运算是指将表中第 i 个元素从线性表中去掉，i 的取值范围为 $1\le i\le n$。通过以下步骤完成顺序表的删除运算：

① 将 a_{i+1}～a_n 顺序向上移动；

② 修改 len 值（即修改表长），使之减 1。

算法描述：

```
 delete(sequenlist *L, int  i)
{int j;
if((*L).len==0)
   { printf("empty list\n");
     return NULL;
   }
else  if((i<1)||(i>(*L).len))
          { printf("delete fail\n");
            return NULL;
          }//删除的元素越界
else {
          for(int j=i;j<=len-1;j++)
                (*L).vec[j-1]=(*L).vec[j];                    //元素前移
```

```
        (*L).len--;                                    //表长度减 1
        return 1;
      }
  }
```

性能分析：

与插入运算相同，删除第 i **个元素时，后面的元素** a_{i+1} **～** a_n **都要向上移动一个位置，共移动了** n-i **个元素，故平均移动次数：**

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

这说明在顺序表上作删除运算大约需要移动表中一半的元素，显然该算法的时间复杂度为 O(n)**。**

1.1.3 线性表的链式存储结构

线性表的链式存储结构，也称为链表。链式存储是用一些任意的存储单元来存储线性表中的元素，这些单元在物理上可以是连续的，也可以是不连续的。为了能正确地描述元素之间的逻辑关系，除了存储元素本身的信息外，还需存储指示元素之间逻辑关系的信息。这两部分信息组成数据元素 a_i **的存储映像，称为结点。**

在链表中，每个结点所占的存储空间分为两部分：存放数值的域称为数据域，存放结点之间相互关系的域称为指针域。在定义的链表中，若只含有一个指针域来存放下一个元素地址，称这样的链表为单链表或线性链表。

1. 单链表结构

线性链表中的结点结构可描述为

Data	next

其中，data **域用来存放结点本身信息，类型由具体问题而定；**next **域用来存放下一个元素地址，即存储指示其直接后继的信息。可见，用单链表表示线性表时，数据元素之间的逻辑关系是通过链结点中的指针反映出来的，即指针是数据元素之间逻辑关系的映像。图** 1-1**、图** 1-2 **给出单链表结构示意图。**

2. 单链表上的基本运算

为了便于实现各种运算，通常在单链表的第一个结点前增设一个附加结点，称为头结点，它的结构与表结点相同，其数据域可不存储信息，也可存储如线性表长度一类的附加信息。单链表设置头结点的作用归纳如下：

① 头指针是指向头结点的非空指针，无论链表是否为空，头指针始终保持值不变，因此头指针的处理方法对空表和非空表的操作是一致的。这与不带头结点的单链表为空时头指针为空不同。

② 首结点的地址存放在头结点（可看作首结点的前驱结点）的指针域中，故对该结点的操作与对表中其他结点的操作一致，无须进行特殊处理。

头指针		地址	data 域	next 域
head	150	110	a_2	180
		120		
		130	a_4	170
		140	a_6	NULL
		150	a_1	110
		160		
		170	a_5	140
		180	a_3	130

图 1-1　单链表示意图

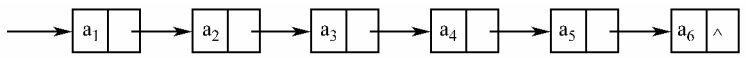


图 1-2　单链表的逻辑表示

用 C 语言描述结点结构如下：

```
typedef int elemtype;    //定义新类型，类型名为 elemtype
typedef struct node
 { elemtype data;      //数据域
    struct node *next; //指针域
   }linklist;
```

（1）单链表的构建

链表与顺序表不同，它是一种动态管理的存储结构，链表中的每个结点占用的存储空间不是预先分配，而是运行时系统根据需求而生成的，因此建立单链表从空表开始，每读入一个数据元素则申请一个结点，然后插在链表的尾部。

算法描述：

```
linklist    *creatlist(int n)
 {
    int x,k;        //设数据元素的类型为 int
   linklist    *head, *r, *p;
   p=(linklist *)malloc(sizeof(linklist));    //构建头结点空间
   head=p;
   p->next=NULL;
   r=p;
   for(k=1;k<=n;k++)            //循环构建 n 个结点
   { printf("input value:\n");
```

```
    scanf("%d",&x);
        p=(linklist *)malloc(sizeof(linklist));
      p->data=x;
      p->next=NULL;
      r->next=p;
      r=r->next;
     }
    return(head);
}
```

（2）单链表上元素的查找

从头指针开始，一般有两种查找方法：按元素的序号和按某个给定值。

按序号查找：

设单链表的长度为 n，要查找表中第 i 个结点，算法思想如下：

从头结点开始顺链扫描，用指针 p 指向当前扫描到的结点，用 j 作统计已扫描结点的计数器，当 p 扫描下一个结点时，j 自动加 1。

P 的初值指向头结点，j 的初值为 0。

当 j=i 时，指针 p 所指的结点就是第 i 个结点。

算法描述:

```
linklist     *find (linklist *head, int i)
{ int j;
   linklist *p;
   p=head->next;
   j=1;
   While((p!=NULL) && (j<i))
     { p=p->next;
       j++;
      }
   return p;
}
```

按值查找：

在链表中，查找是否有结点等于给定值 key 的结点，若有的话，则返回首次找到的值为 key 的结点的存储位置；否则返回 null。

算法思想

从开始结点出发，沿着链表将逐个结点的值和给定值 key 作比较。

算法描述：

```
    linklist *locate(linklist *head, elemtype x)
    { linklist *p;
        p=head->next;
```

```
while(p)      也可改为      while(p!=NULL && p->data!=x)
                                  p=p->next;
 { if(p->data!=x)
        p=p->next;
   else
        break;
 }
return p;
}
```

（3）单链表的插入运算

类似于查找运算，单链表的插入运算也可分两种方式来处理，一种按给定值插入，另一种按给定序号插入，根据给定的问题而定。下面主要讨论按给定值来处理。

① 已知线性链表 head**，在** p **指针所指向的结点后插入一个元素** x**。在一个结点后插入数据元素时，称为后插入法。操作过程如图** 1-3 **所示。**

算法描述：

```
//将值为 x 的新结点插入到值为 k 的结点之后。
void insert( linklist *head, elemtype x, elemtype k)
 { linklist *p,*s;
   s=(linklist *)malloc(sizeof(linklist));   //生成新结点 x
   s->data=x;
  p=head->next;
if( p= =NULL)   //若为空表，则新结点作为表中唯一的结点
{ head->next=s;
   s->next=NULL;
   }
else { while(p && p->data!=k)
            p=p->next;
       if(p->data==k)
       { s->next=p->next;
          p->next=s;
       }
else //没有找到值为 k 的结点，将新结点插入到表尾
      {   p->next=r;
          r->next=NULL;
       }
     }
}
```

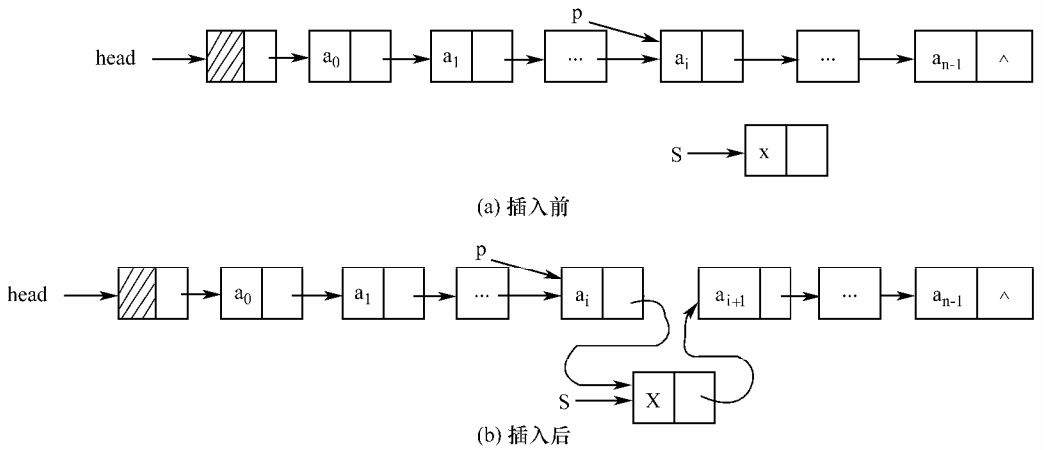


图 1-3　单链表的后插入

② 已知线性链表 head，在 p 指针所指向的结点前插入一个元素 x 称为前插，前插时必须从链表的头结点开始，找到 P 指针所指向的结点的前驱。设一指针 q 从附加头结点开始向后移动进行查找，直到 p 的前趋结点为止。然后在 q 指针所指的结点和 p 指针所指的结点之间插入结点 s。插入过程如图 1-4 所示。

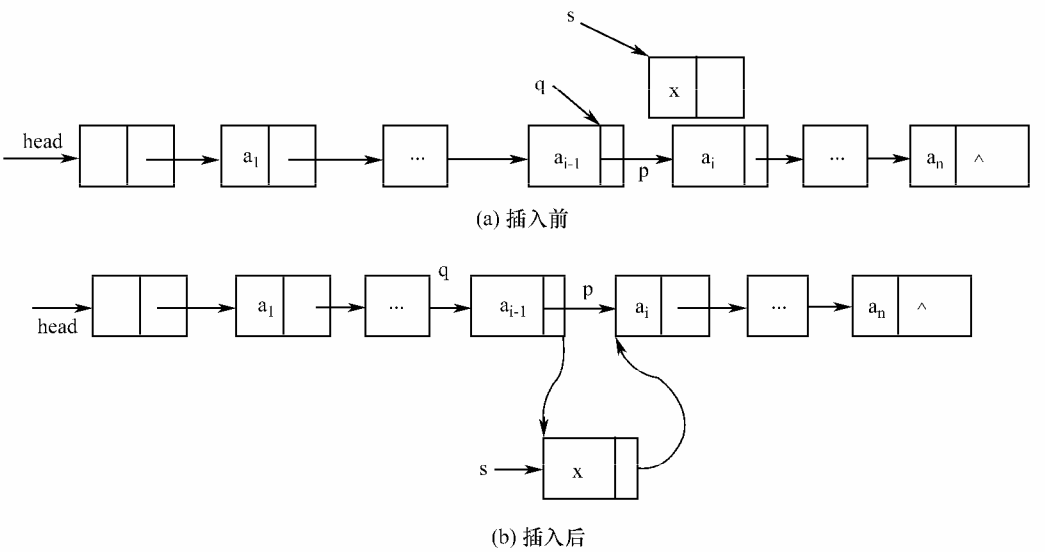


图 1-4　单链表的前插入

算法描述：

```
//将值为 x 的新结点插入到值为 k 的结点之前。
void insert( linklist *head, elemtype x, elemtype k)
{ linklist *q,*p,*s;
  s=(linklist *)malloc(sizeof(linklist));   //生成新结点 x
s->data=x;
if( head->next= =NULL)   //若为空表，则新结点作为链表中唯一的表结点
```

```
    { head->next=s;
      s->next=NULL;
    }
  else {    q=head;
            p=head->next
  while(p!=NULL)
        { if(p->data!=k)
           {q=p;
           p=p->next;
           }
      else
         break;
      }
  if( p!=NULL)
    { q->next=s;
      s->next=p;
    }
  else          //p= =NULL  在表尾插入新结点
      { q->next=s;
        s->next=NULL;
      }
   }   //插入完毕
}
```

不管是后插还是前插，前面描述的都是按给定值找到待插入点，实际上还可以按序号找到待插入点，如在第 i 个结点之前或之后插入新结点，同样也可分为前插与后插，具体处理方法是先利用按序号查找方法找到第 i 个结点，然后再按类似于按给定值的插入方法插入新结点。在此不再赘述。

（4）单链表的删除运算

删除运算也可按两种方法来实现，即按序号删除或按给定值 x 删除，具体根据题意要求来处理。若要删除线性链表 h 中的第 i 个结点，首先要找到第 i 个结点并使指针 p 指向其前驱第 i-1 个结点，然后删除第 i 个结点并释放被删除结点空间。若要删除值为 x 的结点，则删除 x 结点的指针变化如图 1-5 所示。

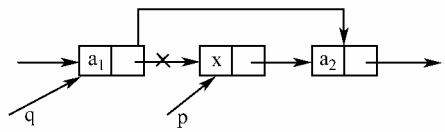


图 1-5　删除结点的指针修改

下面以删除第 i 个结点为例，介绍单链表删除算法的具体描述。

算法描述：

```
Linklist    *delete(linklist *head, int i)
{ linklist *p, int j;
```

```
        p=head->next;
        j=1;
      while(p && j<i-1)
      { p=p->next; j++;}
   if(p!=NULL)
      { q=p->next;
         p->next=p->next->next;
      free(q);
      }
   else
        return NULL;
   return head;
  }
```

1.1.4 循环链表结构

单链表上的访问是一种顺序访问，从其中某一个结点出发，可以找到它的直接后继，但无法找到它的直接前驱。因此，我们可以考虑建立这样的链表，具有单链表的特征，但又不需要增加额外的存储空间，仅对表的链接方式稍作改变，使得对表的处理更加方便灵活。

从单链表可知，最后一个结点的指针域为 NULL 表示单链表已经结束。如果将单链表最后一个结点的指针域改为存放链表中头结点(或第一个结点)的地址，就使得整个链表构成一个环，又没有增加额外的存储空间，称这样的链表为单循环链表，简称为循环表。

1.1.5 双向链表结构

1. 双向链表基本概念

在单链表中，从某个结点出发可以直接找到它的直接后继，但无法直接找到它的直接前驱；在单循环链表中，可以使我们从任何位置开始，沿着链访问链表中的任一结点。但有时，希望能快速找到一个结点的直接前驱，另外，当删除表中一个结点时，需要找出被删除结点的直接前驱结点，只有从表头结点开始搜索或者从被删除结点开始沿循环链表周游一圈，显然十分不方便。为了克服这种单向性的缺点，可以在单链表中的结点中增加一个指针域指向它的直接前驱，这样的链表，就称为双向链表(一个结点中含有两个指针)。如果每条链构成一个循环链表，则会得双向循环链表。

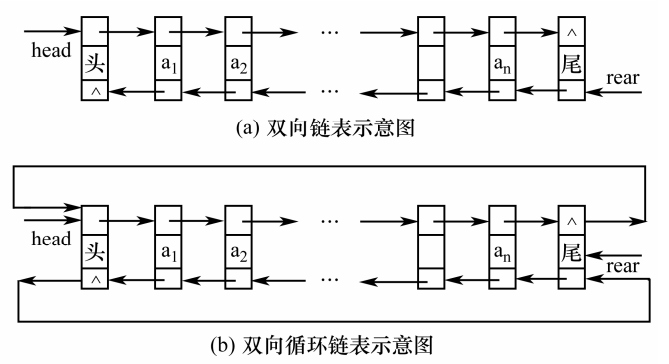
2. 双向链表结点的定义与实现

双向链表结点的定义与单循环链表非常相似。下面给出双向链表的结点定义：

```
typedef int elemtype;
typedef struct dupnode
   { elemtype data;
     struct dupnode *next, *prior;
```

}dulinklist;

和单链表有循环表类似，双向链表也有循环表，结构见图 1-6。



(a) 双向链表示意图

(b) 双向循环链表示意图

图 1-6　双向链表及循环链表示意图

3. 双向链表的基本操作

（1）双向链表插入运算 duinsert (head,x,y)

同单链表一样，双链表的插入也可分为按值插入与按序号插入两种方法。如在 head 为头指针的双向链表中，在值为 y 的结点之后插入值为 x 的结点，插入结点的指针变化见图 1-7(若改为在值为 y 的结点之前插入值为 x 的结点，可以作类似分析)。

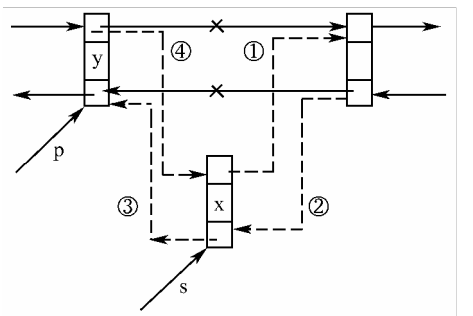


图 1-7　s 插入 p 之后指针变化示意图

```
s->next=p->next;    /*图中步骤①*/
p->next->prior=s;   /*图中步骤②*/
s->prior=p;    /*图中步骤③*/
p->next=s;    /*图中步骤④*/
```

下面给出按序号的插入算法。比如在 head 所指向的双向链表中，将值为 x 的新结点插入到第 i 个结点之前(前插)。

算法思路：

先找到第 i-1 个结点，然后在它后面插入新结点 x，下面所写算法利用了前面所述的后插方法。

算法描述：

```
void insdulist(dulinklist *head, int i, elemtype x)
{ dulinklist *p,*s;
   int j;
   p=head;
   j=0;
 while((p->next!=NULL) &&(j<i-1))    //找到第 i-1 个结点
   {
   p=p->next;
   j++;
```

```
    }
  if(j= =i-1)
   {
    s=(dulinklist *)malloc(sizeof(dulinklist));
    s->data=x;
    s->prior=p;
    s->next=p->next;
    p->next->prior=s;
    p->next=s;
   }
  else
    printf("error\n");
  }
```

本算法也可改为先找到第 i **个结点，然后再前插新结点，区别仅在于插入的四个步骤不同而已。同样也可以在值为** k **的结点前（或后）插入值为** x **的新结点。**

讨论：

我们在双向链表中进行插入操作时，具体写程序时还需注意下面两种特殊情况：

① 当在链表中的第一个结点前插入新结点时，新结点的 prior **应指向头结点，原链表第一个结点的** prior **应指向新结点，新结点的** next **应指向原链表的第一个结点。**

② 当在链表的最后一个结点后插入新结点时，新结点的 next **应为空，原链表的最后一个结点的** next **应指向新结点，新结点的** prior **应指向原链表的最后一个结点。**

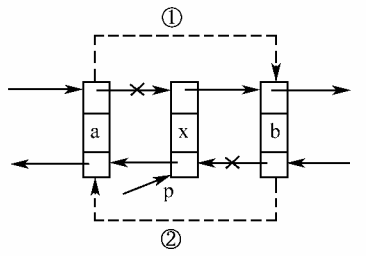


图 1-8　**删除** p **指针所指结点示意图**

（2**）双向链表的删除运算** dudelete(head ,x)

删除运算同插入一样，也可分为按序号或按值删除。如在以 head **为头的双向链表中删除值为** x **的结点，删除算法的指针变化见图** 1-8**。**

```
  p->prior->next=p->next;      /*图中步骤①*/
  p->next->prior=p->prior;     /*图中步骤②*/
```

下面给出按序号删除算法，在 head **所指向的双向链表中，删除第** i **个结点。**

算法描述：

```
void deledulist(dulinklist *head, int i)
{dulinklist *p;
 int j;
p=head;
j=0;
while((p->next!=head) &&(j<i))
   { p=p->next;
     j++;
   }
```

```
if(j= =i)
  {p->prior->next=p->next;
  p->next->prior=p->prior;
  delete p;
  }
else   printf("error\n");
}
```

下面是按给定值删除算法，在 head **所指向的双向链表中，删除值为** x **的结点。**

算法描述：

```
void   dudelete(dulinklist   *head, elemtype x)
{   dulinklist   *p;
  p=head;
    while((p!=NULL)&&(p->data!=x) )
          p=p->next;
    if(p!=NULL)                              //已找到该结点
    {   p->prior->next=p->next;
      p->next->prior=p->prior;
      delete p;
    }                                        //删除后的结点空间回收
  else   printf("没找到，不能删除\n");
}
```

讨论：

我们在双向链表中进行删除操作时，还需注意下面两种特殊情况：

① 当删除链表的第一个结点时，应将链表开始结点的指针指向链表的第二个结点，同时将链表的第二个结点的 prior **置为指向头结点。**

② 当删除链表的最后一个结点时，只需将链表的最后一个结点的前一个结点的 next **置为** NULL **即可。**

1.1.6　线性表顺序存储结构和链式存储结构

对于线性表而言实际采用哪种存储结构，不能一概而论，主要从存储空间、运算时间、程序设计语言等三方面考虑。

1. 存储空间

顺序表的存储空间是静态分配的，要求预先分配空间，也就是说事先对“MAXSIZE**”要有合适的设定。一般程序执行之前难以估计存储空间大小，估计过大造成浪费，估计过小产生空间溢出；而链式结构的存储空间是动态分配，只要有内存空间，就可动态申请，不产生溢出。**

2. 运算时间

顺序表是一种随机存取结构，便于元素的随机访问；而链式结构是一种非随机存储结构，对任一结点的操作都必须从头指针开始顺着链扫描才能取得。当插入、删除运算较多时，对顺序表而言意味着大量数据元素的移动。故对于只进行查找运算而很少做插入和删除运算时，