第二章　线性表

一、线性表概述

1．线性表：具有相同类型的 n 个数据元素的有限序列，可表示为$(a_1, a_2, \cdots, a_n)$

2．线性表长度：数据元素个数 n。

3．空线性表：长度为零的线性表。

4．关键字/键：能唯一标识元素的字段。

5．相邻元素关系：a_i 为 a_{i+1} 前驱元素，a_{i+1} 为 a_i 后继元素，存在序偶关系。

a_1 无前驱元素，a_n 无后继元素。

例：字母表：(A, B, C, ⋯, Z)

学生成绩表：(790631, 790632, ⋯, 790645)每个元素为一条记录，由若干数据项组成，线性表中可只写关键字。

二、线性表的顺序存储

1．顺序存储：线性表的各个元素依次存储在一组地址连续的存储单元里。

2．元素位置确定：$LOC(a_i)=LOC(a_1)+(i-1)*L$　其中 a_1 为线性表第一个元素的存储位置，L 为每个元素所占字节数。

例：已知 a_1 的地址为 1000，每个元素占 2 字节，求 a_5 的地址

$LOC(a_5)=1000+(5-1)*2=1008$

3．顺序表的特点

（1）是一种随机存取的存储结构

（2）逻辑上相邻的元素物理位置必定紧邻

（3）适用于事先能确定线性表的大小，存取速度快

（4）插入和删除操作时需移动大量的元素,

4．插入或删除元素时移动次数

（1）向含有 n 个元素的线性表第 i 个位置插入元素，需移动 n-i+1 次元素，平均移动 $\frac{n}{2}$ 次

（2）删除含有 n 个元素的线性表第 i 个位置的元素，需移动 n-i 次元素，平均移动 $\frac{n-1}{2}$ 次

5．存储结构的描述

```
#define list_max    100
typedef    struct{int data[list_max];
                  int    len;}sqlist;
```

6．顺序结构线性表的运算

（1）线性表的初始化

```
    void InitList(sqlist &L)
    {L.len=0;}
```

（2）求线性表的长度

```
    int ListLen(sqlist L)
    {return    L.len;}
```

（3）在线性表中查询某个值为 x 的结点，返回结点在线性表中的位置

```
    int    search( int x; sqlist L; int n )
    { int    i;
       for( i=0; i<n; i++)
         if(x==L.data[i]) break;
      if( i==n)return 0;
      return i+1; }
```

（4）在顺序线性表中第 i 个位置插入新元素

```
bool insert_sq(sqlist &L; int i; int e)
{ int p;
  if( i<1||i>L.len+1) return false;
  if( L.len>=list_max) return false;
  for( p=L.len; p>=i; p-- )
     L.data[p]=L.data[p-1];
  L.data[i-1]=e;
  L.len++;
  return true;}
```

（5）在顺序线性表中删除第 i 个元素

```
  bool delete_sq(sqlist &L,int i)
```

```
{int p;
  if(i<1||i>L.len) return false;
  for( p=i; p<L.len; p++)
     L.data[p-1]=L.data[p];
   L.len--;
   return true;}
```

（6）合并两个递增有序的顺序线性表，合并后仍为递增有序

```
  bool mergelist(sqlist LA,sqlist LB,sqlist &LC)
  {   if(LA.len+LB.len>LC.list_max)
        return false;
      int i=0,j=0,k=0;
     while(i<LA.len&&j<LB.len)
       {if(LA.data[i]<=LB.data[j])
            LC.data[k++]=LA.data[i++];
          else
            LC.data[k++]=LB.data[j++];}
       while(i<LA.len)
            LC.data[k++]=LA.data[i++];
       while(j<LB.len)
            LC.data[k++]=LB.data[j++];
       LC.len= k;
       return true;}
```

三、线性表的链式存储

（一）线性链表

1．特点

1）逻辑上相邻的元素，物理上不一定紧邻

2）插入和删除操作时，不需移动大量元素，只需修改有限的指针变量

3）动态分配和释放存储单元，避免了空间浪费

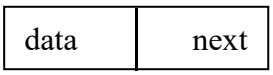
4）不具备随机存取的优点，查找结点需从表头开始，结点的存储利用率较低

5）适用于经常进行插入、删除操作或事先不能确定结点的最大数量

2．线性链表：具有链式存储结构的线性表

3．单链表：每个结点只包含一个指针域

4．结点结构：



5．结点类型

```
typedef struct node
{ int data;
   struct node *next;}NODE;
```

6．头指针：指向链表的头结点或第一个结点的位置

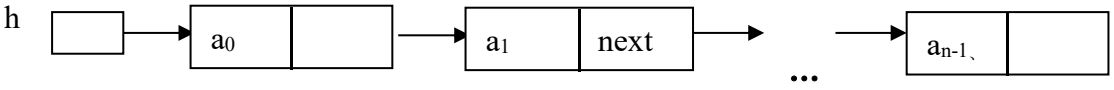
7．头结点：在链表第一个结点之前附设的结点，其数据域可不存任何信息或存表长，指针域保存第一个结点的地址

8．设置头结点的作用：插入或删除首元素时不必对头指针进行修改

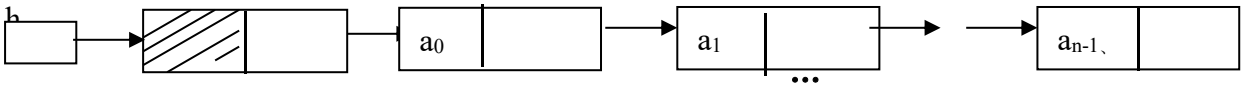
9．第一个结点：第一个实际存储数据的结点

10．单链表的结构

1）不带头结点



2)带头结点



11．建立带头结点的单链表,在表尾插入，以-1 结束

```
NODE *creat( )
{NODE   *h,*p,*r;
  int x;
  h=r=(NODE *)malloc(sizeof(NODE));
  scanf("%d",&x);
  while(x!=-1)
    {p=(NODE *)malloc(sizeof(NODE));
    p->data=x;
```

```
    r->next=p;
    r=p;
    scanf("%d",&x);}
r->next=NULL;
return h;}
```

12．建立带头结点的单链表,在表头插入，以-1 结束

```
NODE *creat( )
{NODE    *h,*p;
  int x;
  h= (NODE *)malloc(sizeof(NODE));
  h->next=NULL;
  scanf("%d",&x);
  while(x!=-1)
        {p=(NODE *)malloc(sizeof(NODE));
        p->data=x;
        p->next=h->next;
        h->next=p;
        scanf("%d",&x);}
 return h;}
```

13．在单链表第 I 个结点插入一个元素

```
  bool insert_L(NODE *h,int I,int x)
  {if(I<1) return false;
   NODE *p=h, *s;
  int    j=0;
  s=(NODE *)malloc(sizeof(NODE));
  s->data=x;
  while(p!=NULL&&j<I-1)
     {p=p->next; j++;}
  if(!p||j>I-1)return false;
  s->next=p->next;
```

```
  p->next=s;
  return true;}
```

14．删除单链表中第 I 个结点

```
  int delete_L(NODE *h, int I, int *e)
 {if(I<1) return false;
   NODE   *p=h, *q;
   int j=0;
   while(p !=null &&j<I-1)
        {p=p->next; j++;}
   if(p= =null）return false;
   if(p->next= =null）return false;
   q=p->next;
   *e=q->data;
   p->next=q->next;
   free(q);
   return true;}
```

15．查找与给定值相同的第一个结点

```
NODE   *locatenode(NODE   *h, int   key)
{NODE   *p=h->next;
   while(p&&p->data!=key)
        p=p->next;
   return   p;}
```

16．合并两个递减有序的链表，合并后仍为递减有序

```
NODE *merge_L(NODE *ha, NODE *hb)
{NODE   *p=ha->next, *q=hb->next, *r, *s;
s=(NODE *)malloc(sizeof(NODE));
s->next=NULL;
r=s;
while(p!=NULL&&q!=NULL)
      if(p->data>q->data)
```

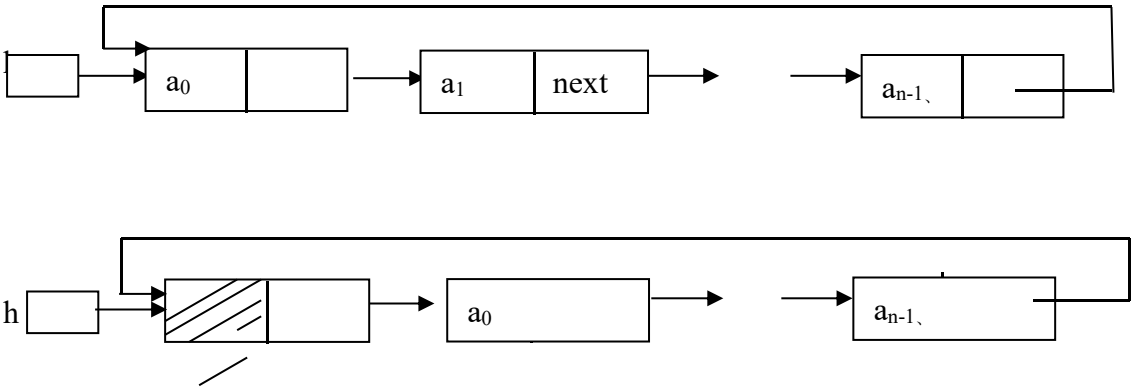
```
        {r->next=p;    r=p;    p=p->next;}
      else
        {r->next=q;    r=q;    q=q->next;}
  if(p==NULL) r->next=q;
  else    r->next=p;
  return s;}
```

（二）循环链表

1．定义：最后一个结点的指针域指向链表的头结点或第一个结点。

2．优点：解决了无法从指定结点到达该结点的前驱结点的问题。

3．结构：



4．判别是否为空

1）带头结点：当 h->next==h 时为空

2）不带头结点：当 h==NULL 时为空

5．删除第一个结点

1）带头结点：h->next=h->next->next

2）不带头结点：h=h->next

6．建立循环链表

将尾结点 r->next=NULL;改为 r->next=h;

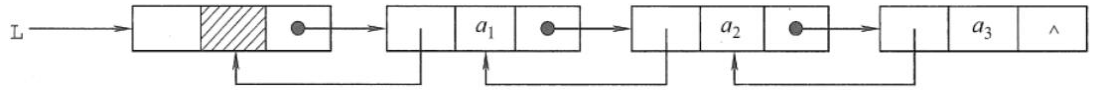
（三）双向链表

1．特点：两个指针域，可用于表示树型结构，一个指向前驱，一个指向后继

2．结点类型：

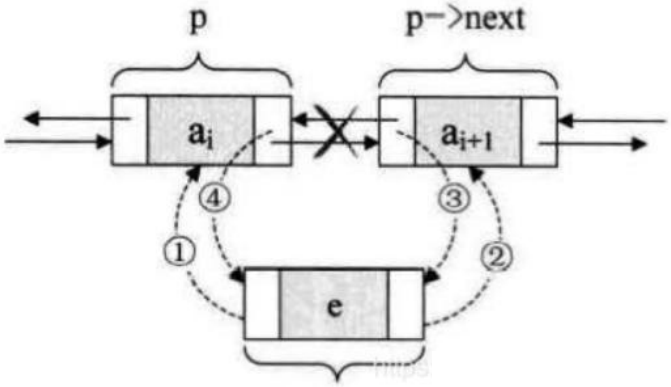
```
/*双向链表存储结构*/
    typedef struct DulNodse{
        ElemType data;
        struct DulNode *prior;     //直接前驱指针
        struct DulNode *next;      //直接后继指针
    } DulNode, *DuLinkList;
```

双链表示意图如下所示：



（1）双向链表的插入操作

在双链表中 p 所指的结点之后插入结点*s，其指针的变化过程如下图所示：

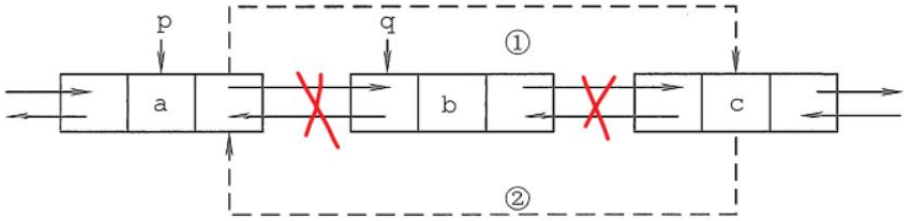


插入操作的代码片段如下：

```
//第一步：把 p 赋值给 s 的前驱
s->prior = p;
//第二步：把 p->next 赋值给 s 的后继
s->next = p->next;
//第三步：把 s 赋值给 p->next 的前驱
p->next->prior = s;
//第四步：把 s 赋值给 p 的后继
p->next = s;
```

（2）双向链表的删除操作

如果要删除 q 结点，只需下面两步：



代码片段如下：

```
//第一步
p->next = q->next;
//第二步
q->next->prior = p;
free(q);
```

四、顺序表和链表的比较

1、存取(读写)方式

顺序表可以顺序存取,也可以随机存取,链表只能从表头顺序存取元素。例如在第 i 个位置上执行存或取的操作,顺序表仅需一次访问,而链表则需从表头开始依次访问 i 次。

2、逻辑结构与物理结构

采用顺序存储时,逻辑上相邻的元素,对应的物理存储位置也相邻。而采用链式存储时,逻辑上相邻的元素,物理存储位置则不一定相邻,对应的逻辑关系是通过指针链接来表示的。

3、查找、插入和删除操作

对于按值查找,顺序表无序时,两者的时间复杂度均为 O(n);顺序表有序时,可采用折半查找,此时的时间复杂度为 O(log2n)。

对于按序号查找,顺序表支持随机访问,时间复杂度仅为 O(1),而链表的平均时间复杂度为 O(n)。顺序表的插入、删除操作,平均需要移动半个表长的元素。链表的插入、删除操作,只需修改相关结点的指针域即可。由于链表的每个结点都带有指针域,故而存储密度不够大。

4、空间分配

顺序存储在静态存储分配情形下,一旦存储空间装满就不能扩充,若再加入新元素,则会出现内存溢出,因此需要预先分配足够大的存储空间。预先分配过大,可能会导致顺序表后部大量闲置;预先分配过小,又会造成溢出。动态存储分配虽然存储空间可以扩充,但需要移动大量元素,导致操作效率降低,而且若内存中没有更大块的连续存储空间,则会导致分配失败。链式存储的结点空间只在需要时申请分配,只要内存有空间就可以分配,操作灵活、高效。