第一章 线性表

1.1 线性表

1.1.1 线性表的定义

2.1 顺序表

2.1.1 顺序表的存储方法与特点
2.1.2 顺序表的运算

3.1 单链表

3.1.1 单链表的定义和特点
3.1.2 建立单链表
3.1.3 线性表基本运算在单链表中的实现

4.1 双向链表

4.1.1 双向链表的结构
4.1.2 建立双链表
4.1.3 线性表基本运算在双链表中的实现

5.1 循环链表

6.1 任务

略

1.1 线性表

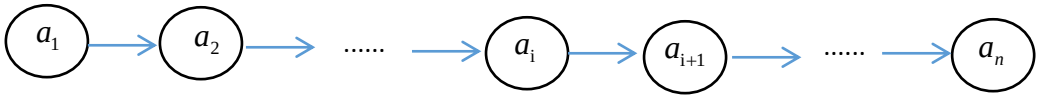
1.1.1 线性表的定义

线性表是具有相同特性的数据元素的一个有限序列。该序列中所含元素的个数叫线性表的长度，用 n 表示，n>=0。当 n=0 时，表示线性表是一个空表，即表中不包含任何元素。在线性表中每个数据元素由逻辑序号唯一确定，设序列中的第 i 个元素为 a_i (1<=i<=n)，则线性表的一般表示为：

$$(a_1, a_2, \ldots a_i, \ldots a_n)$$

其中 a_1 为第一个元素，又称为表头元素，a_2 为第二个元素，a_n 为最后一个元素，也称表尾元素。

线性表中的元素呈现线性关系，即第 i 个元素 a_i 处在第 i-1 个元素 a_{i-1} 的后面，第 i+1 个元素 a_{i+1} 的前面。



线性表结果的图形表示

从线性表的定义可以看出，它有以下特性：

（1）有穷性：一个线性表中的元素个数是有限的。

（2）一致性：一个线性表中的所有元素的性质相同。从实现的角度看，所有的元素具有相同的数据类型。

（3）序列性：一个线性表中的所有元素之间的相对位置是线性的，即存在唯一的开始元素和终端元素，除此之外，每个元素只有唯一的前驱元素和后继元素。各元素在线性表中的位置只取决于它们的序列号，所以在一个线性表中可以存在两个相同的元素。

数据结构的目的是和算法结合，实现各种操作。线性表是一种比较灵活的数据结构，它的长度可根据需要增减，它也可以进行插入、删除等操作。可对线性表进行的基本操作如下：

创建——Create():创建一个新的线性表。

初始化——Init():初始化操作，将新创建的线性表初始化为空。

获取长度——GetLength():获取线性表的长度。

判断表是否为空——IsEmpty():判断线性表是否为空。

获取元素——Get():获取线性表某一个位置插入一个元素。

删除——Delete():删除某一个位置上的元素。

插入——Insert():在线性表的某一个位置插入一个元素。

清空表——Clear():清空线性表，将线性表置为空。

例如，有一个线性表 L=（1,3,1,4,2），求 ListLength(L)、ListEmpty(L)、GetElem(L,3,e)、LocationElem(L,1)、ListInsert(L,4,5)和 ListDelete(L,3)基本运算依次执行后的结果。

答案：ListLength(L)=5,即线性表 L 的长度为 5。

ListEmpty(L)返回 false，即线性表 L 是非空表。

GetElem(L,3,e),e=1，即线性表 L 中的第三个元素是 1。

LocationElem(L,1)=1，即线性表 L 中第一个值为 1 的元素的逻辑序号为 1。

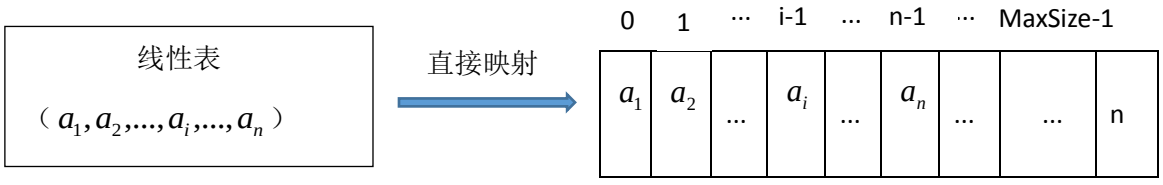
ListInsert(L,4,5)是在线性表 L 中逻辑序号 4 的位置插入元素 5，执行后 L 变成（1,3,1,5,4,2)。

ListDelete(L,3)是在线性表 L 中删除逻辑序号 3 的元素，执行后 L 变成（1,3,5,4,2)。

2.1 顺序表

2.1.1 顺序表的存储方法与特点

线性表的顺序存储结构是把线性表中的所有元素按照其逻辑顺序依次存储到计算机从计算机存储器中指定存储位置开始的一块连续的存储空间中。由于线性表中逻辑上相邻的两个元素在对应的顺序表中它们的存储位置也相邻，所以这种映射称为直接映射。线性表的顺序存储结构简称为顺序表。



这样，线性表 L 中第一个元素的存储位置就是制定的存储位置，第 i+1 个元素（1<=i<=n-1）的存储位置紧接在第 i 个元素的存储位置的后面。假设线性表的元素类型为 ElemType，则每个元素所占用存储空间的大小（即字节数）为 sizeof(ElemType)，整个线性表所占用存储空间的大小为 nXsizeof(ElemType)，其中 n 表示线性表的长度。

线性表中的第一个元素 a_1 存储在对应数组的起始位置，即下标为 0 的位置上，第二个元素 a_2 存储在下标为 1 的位置上，依次类推，第 i 个元素 a_i 存储在下标为 i-1 的位置上。

假设线性表 L 存储在数组 A 中，A 的起始存储位置为 LOC(A),则 L 所对应的顺序表如下图所示。需要注意的是，顺序表采用数组来实现，但不能将任何一个数组都当作是一个顺序表，二者的运算是不同的。

下标位置	线性表存储空间	存储地址
0	a_1	LOC(A)
1	a_2	LOC(A)+sizeof(ElemType)
⋮	⋮	
i-1	a_i	LOC(A)+(i-1)*sizeof(ElemType)
⋮	⋮	
n-1	a_n	LOC(A)+(n-1)*sizeof(ElemType)
⋮	⋮	
MaxSize-1	⋮	LOC(A)+(MaxSize-1)sizeof(ElemType)

顺序表的示意图

数组大小 MaxSize 一般定义为一个整型长量。如果估计一个线性表不会超过 50 个元素，则可以把 MaxSize 定义为 50：

```
#define MaxSize 50
```

在声明线性表的顺序存储结构类型时，定义一个 data 数组来存储线性表中的所有元素，还定义一个整型变量 length 来存储线性表的实际长度，并采用结构体类型 SqList 表示如下：

```
typedef struct {
    ElemType data[MaxSize];          //存放线性表中的元素
    int length;                      //存放线性表的长度
}SqList;                             //顺序表类型
```

例如：使用 C 语言编写代码，创建一个存储上限为 5000 的顺序表，每个数据元素用于表示一个学生的个人信息（包括学生的属性年龄、姓名、性别和身高）。

```
#define MaxSize 5000
Typedef struct Student{
  Char name[20];
  Int age;
  Char sex;
  Double height;
```

```
}DataType;
Typedef struct{
    ElemType data[MaxSize];
    Int lengh;
}SqList;
```

2.1.2 顺序表的运算

一旦采用顺序表存储结构，就可以用 C/C++语言实现线性表的各种基本运算。为了简单，假设 ElemType 为 int 类型，使用以下自定义类型语句：

```
Typedef int ElemType;
```

1. 建立顺序表

这里介绍整体创建顺序表，即由数组元素 a[0..n-1]创建顺序表 L。其方法是将数组 a 中的每个元素依次放入到顺序表中，并将 n 赋给顺序表的长度域。算法如下：

```
void CreateList(SqList *&L,ElemType a[],int n)  //由 a 中的 n 个元素建立顺序表
{
    int i=0,k=0;                                //k 表示 L 中的元素个数，初始值为 0
    L = (SqList *)malloc(sizeof(SqList));  //分配存放线性表的空间
    While(i>n)                                  //i 扫描数组 a
    {
        L->data[i] = a[i];                  //将数组 a[i]存放到 L 中
        k++;i++;
    }
    L->length = k;                              //设置 L 的长度为 k
}
```

当调用上述算法创建好 L 所指的顺序表后，需要回传给对应的实参，也就是说，L 是输出型参数，所以在形参 L 的前面需要加上引用符“&”。

2. 顺序表基本运算算法

（1）初始化线性表 InitList(&L)

该算法发的功能是构造一个空的线性表 L，实际上只需分配线性表的存储空间并将 length 域设置为 0 即可。算法如下：

```
void InitList(SqList *&L)
{
    L=(SqList *)malloc(sizeof(Sqlist));  //分配存放线性表的空间
    L->length = 0;                       //置空线性表的长度为 0
}
```

（2）销毁线性表 DestroyList(&L)

该算法的功能是释放线性表 L 占用的内存空间。算法如下：

```
void DestroyList(SqList *&L)
{
    free(L);                            //释放 L 所指的顺序表空间
}
```

这里顺序表是通过 malloc 函数分配存储空间的，当不再需要顺序表时务必调用 DestroyList 基本运算释放其存储空间；否则，尽管系统会自动释放顺序表指针变量 L，但

不会自动释放 L 所指向的存储空间，如此可能会造成内存泄漏。

（3）判断线性表是否为空表 ListEmpty(L)

该算法返回一个布尔值表示 L 是否为空表。若 L 为空表，则返回 true,否则返回 false。算法如下：

```
Bool ListEmpty(SqList *L)
{
    Return (L->length==0);
}
```

（4）求线性表的长度 ListLength(L)

该运算返回顺序表 L 的长度，实际上只需要返回 length 域的值即可。算法如下：

```
int ListLength(SqList *L)
{
    Return (L->length);
}
```

（5）输出线性表 DispList(L)

该运算依次显示 L 中各元素的值。算法如下：

```
Void DispList(SqList *L)
{
    For(int i=0;i<L->length;i++)    //扫描顺序表输出各元素值
        Printf(“%d”,L->data[i]);
    Printf(“\n”);
}
```

（6）求线性表中的某个数据元素值 GetElem(L,i,&e)

该运算用引用型参数 e 返回 L 中第 i(1<=i<=n)个元素的值。算法如下：

```
Bool GetElem(SqList *L,int i,ElemType &e)
{
    If(i<1||i>L->length)
        Return false;
    E = L->data[i-1];     //取元素值
    Return true;
}
```

（7）按元素值查找 LocateElem(L,e)

该运算顺序查找第一个值域与 e 相等的元素的逻辑序号(找到后返回一个大于 0 的值)，若这样的元素不存在，则返回值为 0。算法如下：

```
Int LocateElem(SqList *L,ElemType e)
{
    Int i=0;
    While(i<L->length && L->data[i]!=e)
        i++;                                    //查找元素
    if(i>=L->length)                            //未找到时返回 0
        Return 0;
    Else
        Return i+1;                             //找到后返回其逻辑序号
}
```

（8）插入数据元素 ListInsert(&L,i,e)

该运算在顺序表 L 的第 i(1<=i<=n+1)个位置上插入新元素 e。如果 i 值不正确，返回 false;否则将顺序表原来的第 i 个元素及以后的元素均后移一个位置，并从最后一个元素 a_n 开始移动起，腾出一个空位置插入新元素，最后顺序表的长度增 1 并返回 true。算法如下：

```
Bool ListInsert(SqList *&L,int i,ElemType e)
{
    Int j;
    If(i<1 || i>L->length+1)
        Return false;
    i--;                                //将顺序表逻辑序号转化为物理序号
    For(j=L->length;j>i;j--)            //将 data[i]及后面的元素后移一个位置
        L->data[j]=L->data[j-1];
    L->data[i]=e;                       //插入元素 e
    L->length++;                        //顺序表长度增 1
    Return true;
}
```

（9）删除数据元素 ListDelete(&L,i,e)

该运算删除顺序表 L 的第 i（1<=i<=n）个元素。如果 i 值不正确，返回 false；否则将线性表第 i 个元素以后的元素均向前移动一个位置，并从元素 a_{i+1} 开始移动起，这样覆盖了原来的第 i 个元素，达到了删除该元素的目的，最后顺序表的长度减 1 并返回 true。算法如下：

```
Bool ListDelete(SqList *&L,int i,ElemType e)
{
    Int i;
    If(i<1 || i>L->length)
        Return false;
    i--;
    e = L->data[i];
    For(j=i;j<L->length-1;j++)            //将 data[i]之后的元素前移一个位置
        L->data[j]=L->data[j+1];
    L->length--;
    Return true;
}
```

3.1 单链表

3.1.1 单链表的定义和特点

为了克服顺序表的缺点，线性表也可以采用链接存储方法。线性表的链接存储结构称为链表。常见的链表有三种：单链表、循环链表和双向链表。

在单链表中，假设每个节点的类型用 LinkNode 表示，它应包括存储元素的数据域，这

里用 data 表示，其类型用通用类型标识符 ElemType 表示，还包括存储后继节结点位置的指针域，这里用 next 表示。LinkNode 类型声明如下：

```
Typedef struct LNode
{
    ElemType data;            //存放元素值
    Struct LNode *next;       //指向后继结点
}LinkNode;                   //单链表结点类型
```

为了简单，假设 ElemType 为 int 类型，使用以下自定义类型语句：

```
Typedef int ElemType;
```

在后面的算法设计中，如果没有特别说明，均采用带头结点的单链表，在单链表中增加一个头结点的优点如下：

（1）单链表中首结点的插入和删除操作与其他节点一致，无须进行特殊处理。

（2）无论单链表是否为空都有一个头结点，因此统一了空表和非空表的处理过程。

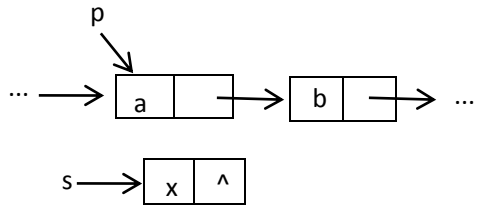
在单链表中，由于每个结点只包含有一个指向后继结点的指针，所以当访问过一个结点后只能接着访问它的后继结点，而无法访问它的前驱结点，因此在进行单链表结点的插入和删除时就不能简单地只对该结点进行操作，还必须考虑其前后的结点。

1. 插入和删除结点的操作

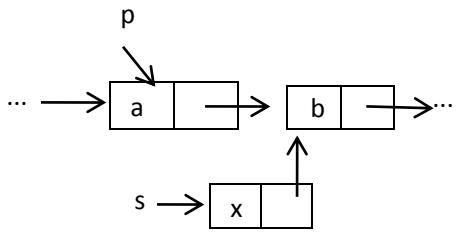
在单链表中，插入和删除结点是最常用的操作，是建立单链表和相关基本运算算法的基础。

（1）插入结点操作

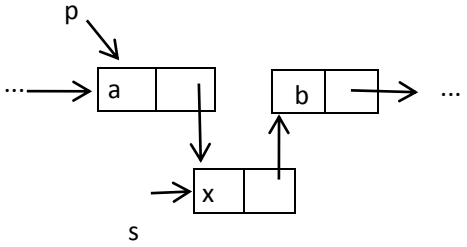
这里插入是指在单链表的两个数据域分别为 a 和 b 的结点（已知 a 结点的指针 p）之间插入一个数据域为 x 的结点（由 s 指向它），如图（a）所示。其操作是首先让 x 结点的指针域（s->next）指向 b 结点（p->next），然后让 a 结点的指针域（p->next）指向 x 结点（s），从而实现 3 个结点之间逻辑关系的变化，插入过程如图（b）和（c）所示，（d）是插入后的结果。



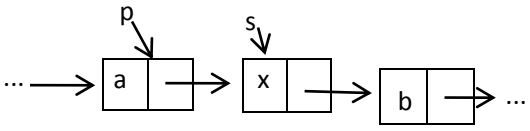
(a)插入前



(b)s->next=p->next



(c)p->next=s



(d 插入后

上述指针修改用 C/C++语句描述如下：

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

在单链表中插入一个新结点需要找到插入后的前驱结点。

（2）删除结点操作

这里删除是指在单链表中删除结点 p 的后继结点，删除 b 结点的操作是让 a 结点的指针域（p->next）指向 c 结点（p->next->next），上述指针修改用 C/C++语句描述如下：

```
P->next = p->next->next;
```

所以在单链表中删除一个结点需要找到其前驱结点。

一般情况下，在删除一个结点后还需要释放其存储空间，实现删除上述 b 结点并释放其存储空间的语句描述如下：

```
q = p->next;              //临时保存被删结点
p->next = q->next;        //从链表中删除结点 q
Free(q);                  //释放结点 q 的空间
```

3.1.2 建立单链表

（1）头插法

该方法从一个空表开始依次读取数组 a 中的元素，生成一个新结点（由 s 指向它），将读取的数组元素存放到该结点的数据域中，然后将其插入到当前链表的表头上（即头结点之后），直到数组 a 的所有元素读完为止。采用头插法建表的算法如下：

```
Void CreateListF(LinkNode *&L,ElemType a[],int n)
{
    LinkNode *s;
    L=(LinkNode *)malloc(sizeof(LinkNode));
    L->next = NULL;
    For(int i=0;i<n;i++)
    {
        S=(LinkNode *)malloc(sizeof(LinkNode));
        S->data=a[i];
        s->next=L->next;
        L->next = s;
    }
}
```

（2）尾插法

该方法从一个空表开始依次读取数组 a 中的元素，生成一个新结点 s，将读取的数组元素存放到该结点的数据域中，然后将其插入到当前链表的表尾上，直到数组 a 的所有元素读完为止。为此需要增加一个尾指针 r，使其始终指向当前链表的尾结点，每插入一个新结点后让 r 指向这个新结点，最后还需要将 r 所指结点（尾结点）的 next 域置为空。采用尾插法建表的算法如下：

```
Void CreateListR(LinkNode *&L,ElemType a[],int n)
{
    LinkNode *s,*r;
    L=(LinkNode *)malloc(sizeof(LinkNode));  //创建头结点
    r=L;                                     //r 始终指向尾结点，初始时指向头结点
```

```
    For(int i=0;i<n;i++)                    //循环建立数据结点
    {
        s=(LinkNode *)malloc(sizeof(LinkNode));
        s->data=a[i];              //创建数据结点 s
        r->next=s;                 //将结点 s 插入到结点 r 之后
        r=s;
    }
    r->next=NULL;                  //尾结点的 next 域置为 null
}
```

3.1.3 线性表基本运算在单链表中的实现

（1）初始化线性表 InitList(&L)

该算法建立一个空的单链表，即创建一个头结点并将其 next 域设置为空。算法如下：

```
  Void InitList(LinkNode *&L)
{
    L=(LinkNode *)malloc(sizeof(LinkNode));
    L->next = NULL;                 //创建头结点，其 next 域置为 NULL
}
```

（2）销毁线性表 DestoryList(&L)

该算法释放单链表 L 占用的内存空间，即逐一释放全部结点的空间。其过程是让 pre、p 指向两个相邻的结点（初始化时 pre 指向头结点，p 指向首结点）。当 p 不为空时循环：释放结点 pre，然后 pre、p 同步后移一个结点。循环结束后，pre 指向尾结点，再将其释放。

```
   Void DestoryList(LinkNode *&L)
  {
        LinkNode *pre=L,*p=L->next;       //pre 指向结点 p 的前驱结点
        While(p!=NULL)                    //扫描单链表 L
        {
            Free(pre);                    //释放 pre 几点
            Pre=p;                        //pre、p 同步后移一个结点
            P=pre->next;
        }
        Free(pre);              //循环结束时 p 为 NULL，pre 指向尾结点，释放它
  }
```

（3）判断线性表是否为空表 ListEmpty(L)

该运算在单链表 L 中没有数据结点时返回真，否则返回假。算法如下：

```
    Bool ListEmpty(Linknode *L)
    {
        Return (L->next==NULL);
    }
```

（4）求线性表的长度 ListLength(L)

该运算返回单链表 L 中数据结点的个数。由于单链表没有存放数据结点个数的信

息，需要通过遍历来统计。其过程是让 p 指向头结点，n 用来累计数据结点个数（初值为 0），当 p 不为空时循环：n 增 1，p 指向下一个结点。循环结束后返回 n。算法如下：

```
Int ListLength(LinkNode *L)
{
    Int n=0;
    LinkNode *p=L;          //p 指向头结点，n 置为 0（即头结点的序号为 0）
    While(p->next!=NULL)
    {
        n++;
        P = p->next;
    }
    Return (n);             //循环结束，p 指向尾结点，其序号 n 为结点个数
}
```

（5）输出线性表 DispList(L)

该算法逐一扫描单链表 L 的每个数据结点，并显示各结点的 data 域值。算法如下：

```
 Void DispList(LinkNode *L)
{
    LinkNode *p=L->next;
    While(p!=NULL)
    {
        printf(“%d”,p->data);
        p=p->next;
    }
    Printf(“\n”);
 }
```

（6）求线性表中的某个数据元素值 GetElem(L,I,&e)

该运算在单链表 L 中从头开始找到第 i 个结点，若存在第 i 个数据结点，则将其 data 域值赋给变量 e。其过程是让 p 指向头结点，j 用来累计遍历过的数据结点个数（初始值为 0），当 j<i 且 p 不为空循环：j 增 1，p 指向下一个结点。循环结束后有两种情况，若 p 为空，表示单链表 L 中没有第 i 个数据结点（参数 i 错误），返回 false；否则找到第 i 个数据结点，提取它的值并返回 true。算法如下：

```
Bool GetElem(LinkNode *L,int i,ElemType &e)
{
    int j=0;
    LinkNode *p=L;              //p 指向头结点，j 置为 0（即头结点的序号为 0）
    if(i<=0) return false;          //i 错误返回假
    while(j<i && p!=NULL)           //找到第 i 个结点 p
    {
        j++;
        p=p->next;
    }
    if(p==NULL)                     //不存在第 i 个数据结点，返回 false
        return false;
    Else                            //存放第 i 个数据结点，返回 true
```

```
        {
            e=p->data;
            return true;
        }
    }
```

（7）按元素值查找 LocateElem(L,e)

该运算在单链表 L 中从头开始找第一个值域与 e 相等的结点，若存在这样的结点，则返回逻辑序号，否则返回 0。算法如下：

```
Int LocateElem(LinkNode *L,ElemType e)
{
    int i=1;
    LinkNode *p=L->next;          //p 指向头结点，i 置为 0（即头结点的序号为 1）
    while(p!=NULL && p->data!=e)    //查找 data 值为 e 的结点，其序号为 i
    {
        p=p->next;
        i++;
    }
    if(p==NULL)                     //不存在值为 e 的结点，返回 0
        return 0;
    Else                            //存在值为 e 的结点，返回其逻辑序号 i
        return(i);
}
```

（8）插入数据元素 ListInsert(&L,i,e)

该运算的实现过程是先在单链表 L 中找到第 i-1 个结点，由 p 指向它。若存在这样的结点，将值为 e 的结点（s 指向它）插入到 p 所指结点的后面。算法如下：

```
 Bool ListInsert(LinkNode *&L,int i,ElemType e)
{
    Int j=0;
    LinkNode *p=L,*s;          //p 指向头结点，j 置为 0（即头结点的序号为 0）
    If(i<=0) return false;              //i 错误返回 false
    While(j<i-1 && p!=NULL)             //查找第 i-1 个结点 p
    {
        J++;
        P=p->next;
    }
    If(p==NULL)                 //未找到第 i-1 个结点，返回 false
        Return false;
    Else                            //找到第 i-1 个结点 p,插入新结点并返回 true
    {
        S=(LinkNode *)malloc(sizeof(LinkNode));
        s->data=e;                  //创建新结点，其 data 域置为 e
        s->next=p->next;            //将结点 s 插入到结点 p 之后
        p->next=s;
        Return true;
```

```
    }
  }
```

（9）删除数据元素 ListDelete(&L,i,&e)

该算法的实现过程是先在单链表 L 中找到第 i-1 个结点，由 p 指向它。若存在这样的结点，且也存在后继结点(由 q 指向它)，则删除 q 所指的结点，返回 true;否则返回 false，表示参数 i 错误。算法如下：

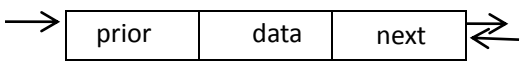
```
Bool ListDelete(LinkNode *&L,int i,ElemType &e)
{
    Int j=0;
    LinkNode *p=L,*q;
    If(i<=0) return false;
    While(j<i-1 && p!=NULL)              //查找第 i-1 个结点
    {
        J++;
        P=p->next;
    }
    If(p==NULL)
        Return false;
    Else                                 //找到第 i-1 个结点 p
    {
        q=p->next;                        //q 指向第 i 个结点
        If(q==NULL)                      //若不存在第 i 个结点，返回 false
            Return false;
        e=q->data;
        p->next=q->next;                 //从单链表中删除 q 结点
        Free(q);                         //释放 q
        Return true;                     //返回 true 表示成功删除第 i 个结点
    }
}
```

4.1 双向链表

由于单链表的每个结点只有一个指示其直接后继指针域，因此，从某个结点出发只能沿着一个方向往后遍历其他结点。对于单链表中的某个已知结点来说：查询其直接后继结点很容易，其时间复杂度为 O(1)，而查询其直接前驱结点则比较麻烦，需要重新从表头指针出发依次查找，其时间复杂度为 O(n)。所以，单链表的前插操作不如后插操作方便，删除某个结点自身不如删除其后继结点方便。为了克服上述缺点，可以采取双向链表的存储结构。

4.1.1 双向链表的结构

在双向链表中，每个结点都由一个数据域和两个指针域组成，两个指针域分别记录其直接前驱和直接后继的地址，因此可以从任何一个结点开始朝两个方向遍历链表。



双向链表的结点结构

在双向链表中的结点类型在 C 语言中可作如下描述：

```
Typedef struct DNode
{
    ElemType data;              //存放元素值
    Struct DNode *prior;        //指向前驱结点
    Struct DNode *next;         //指向后继结点
}DLinkNode;                     //双向表的结点类型
```

在双向链表中，由于每个结点既包含一个指向后继结点的指针，又包含一个指向前驱结点的指针，所以当访问过一个结点后既可以依次向后访问每一个结点，也可以依次向前访问每一个结点。因此与单链表相比，双链表中访问的前、后结点更方便。

4.1.2 建立双链表

整体建立双链表有两种方法，即头插法和尾插法。采用头插法建立双链表的过程和单链表头插法相似，算法如下：

```
Void CreateListF(LinkNode *&L,ElemType a[],int n)
//由含有 n 个元素的数组 a 创建带头结点的双链表 L
{
    DLinkNode *s;
    L=(DLinkNode *)malloc(sizeof(DLinkNode));
    L->prior = L->next = NULL;        //前后指针域置为 NULL
    For(int i=0;i<n;i++)              //循环建立数据结点
    {
        s=(DLinkNode *)malloc(sizeof(DLinkNode));
        s->data=a[i];
        s->next=L->next;          //将 s 结点插入到头结点之后
        If(L->next!=NULL)         //若 L 存在数据结点,修改 L->next 的前驱指针
            L->next->prior = s;
        L->next = s;
        s->prior=L;
    }
}
```

采用尾插法建立双链表的过程和单链表尾插法相似，算法如下：

```
Void CreateListF(LinkNode *&L,ElemType a[],int n)
//由含有 n 个元素的数组 a 创建带头结点的双链表 L
{
    DLinkNode *s，*r;
    L=(DLinkNode *)malloc(sizeof(DLinkNode));
    r=L;                              //r 始终指向尾结点，开始时指向头结点
    For(int i=0;i<n;i++)              //循环建立数据结点
    {
        s=(DLinkNode *)malloc(sizeof(DLinkNode));
        s->data=a[i];
```

```
            r->next=s;                  //将 s 结点插入到 r 结点之后
            s->prior =r;
            r=s;                        //r 指向尾结点
        }
        r->next=NULL;                  //尾结点的 next 域置为 NULL
    }
```

4.1.3 线性表基本运算在双链表的实现

在双链表中，有些运算（如求长度、取元素值和查找元素等）的算法与单链表中的相应算法是相同的，这里不多讨论。但双链表中的插入和删除结点是不同于单链表的，下面分别介绍双链表的插入和删除操作算法。

（1）插入数据元素

在双链表 L 中的第 i 个位置上插入值为 e 的结点时采用类似单链表的插入过程，先查找第 i-1 个结点（由 p 指向它），然后在 p 所指结点之后插入一个新结点。算法如下：

```
Bool ListInsert(DLinkNode *&L,int i,ElemType e)
{
    Int j=0;
    DLinkNode *p=L,*s;
    If(i<=0) return false;
    While(j<i-1 && p!=NULL)
    {
        J++;
        p=p->next;
    }
    If(p==NULL)
        Return false;
    Else                            //找到第 i-1 个结点 p
    {
        s=(DLinkNode *)malloc(sizeof(DLinkNode));
        s->data=e;                   //创建新结点 s
        s->next = p->next;          //若 p 结点之后插入 s 结点
        If(p->next!=NULL)           //若 p 结点存在后继结点，修改其前驱指针
          p->next->prior=s;
        s->prior=p;
        p->next=s;
        Return true;
    }
}
```

（3）删除数据元素

在双链表 L 中删除第 i 个结点时采用类似单链表的删除过程，先查找第 i-1 个结点 p，然后删除结点 p 的后继结点。算法如下：

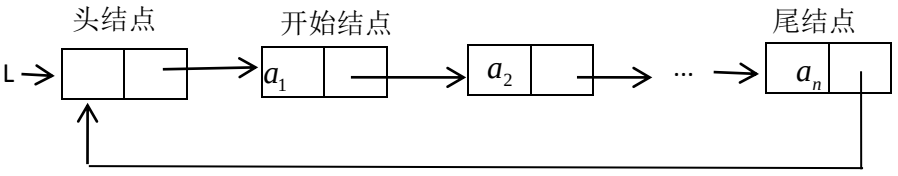
```
Bool ListDelete(LinkNode *&L,int i,ElemType &e)
{
    int j=0;
```

```
    LinkNode *p=L,*q;
    if(i<=0) return false;
    while(j<i-1 && p!=NULL)                //查找第 i-1 个结点
    {
        J++;
        p=p->next;
    }
    if(p==NULL)
        Return false;
    else                                   //找到第 i-1 个结点 p
    {
        q=p->next;                          //q 指向第 i 个结点
        if(q==NULL)                        //若不存在第 i 个结点，返回 false
            return false;
        e=q->data;
        p->next=q->next;                   //从双链表中删除 q 结点
        if(p->next!=NULL)                 //若 p 结点存在后继结点,修改其前驱指针
            p->next->prior=p;
        free(q);                           //释放 q
        return true;                       //返回 true 表示成功删除第 i 个结点
    }
}
```

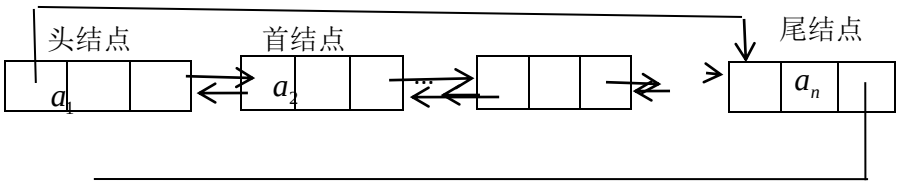
5.1 循环链表

循环链表是另一种形式的链式存储结构。循环链表有循环单链表和循环双链表两种类型，循环单链表的结点类型和非循环单链表的结点类型 LinkNode 相同，循环双链表的结点类型与非循环双链表的结点类型 DLinkNode 相同。

把单链表改为循环单链表的过程是将它的尾结点 next 指针域由原来为空改为指向头结点，整个单链表形成一个环。由此，从表中任一结点出发均可找到链表中的其他结点。以下图所示分别为循环单链表和循环双链表。



(a)循环单链表



(b)循环双链表

把双链表改为循环链表的过程是将它的尾结点 next 指针域由原来为空改为指向头结点，将它的头结点 prior 指针域改为指向尾结点。整个双链表形成两个环。

循环链表的基本运算实现算法与对应非循环链表的算法基本相同，主要差别是对于循环单链表或循环双链表 L，判断表尾结点 p 的条件是 p->next==L；另外在循环双链表 L 中可以通过 L->prior 快速找到结尾点。

6.1 任务

1. 假设有两个集合 A 和 B，分别用两个线性表 LA 和 LB 表示，即线性表中数据元素为集合中的元素。利用线性表的基本运算设计一个算法，求一个新的集合$C = A \cup B$，即将两个集合的并集放在线性表 LC 中。

2. 编写算法 bool InsertList(LinkLNode L,ElemType a,ElemType x)，实现在一个带头结点的单链表 L 中，向数据域为 a 的结点后插入数据域为 x 的结点。

3. 已知一个带头结点的单链表，编写一个算法实现从单链表中删除自第 i 个结点起的 n 的结点。（假设该单链表结点个数大于 i+n）

4. 编写算法 inr CountList(LinkNode L,ElemType x)，实现在给定的带头结点的单链表 L 中，计算数据域为 x 的结点的个数。

5. 已知单链表 L 是一个非空递减有序表，编写算法实现将值为 x 的结点插入 L 中并保持 L 仍然有序。

6. 设计一个算法，删除一个单链表 L 中元素值最大的结点（假设这样的结点唯一）。有一个带头结点的双链表 L（至少有一个数据结点），设计一个算法使其元素递增有序排

7. 有一个带头结点的双链表 L，设计一个算法将其所有元素逆置，即第 1 个元素变为最后一个元素，第 2 个元素变为倒数第 2 个元素。。。。。最后一个元素变为第 1 个元素。

8. 有一个带头结点的循环双链表 L，设计一个 算法删除第一个 data 域值为 x 的结点。

分析：用 p 指针扫描整个循环双链表来查找 data 值为 x 的结点，找到后删除 p 结点，并返回 true，若未找到这样的结点返回 false。算法如下：

9. 设计一个算法，判断带头节点的循环双链表 L（含两个以上的结点）中的数据结点是否对称。