第 2 章　线性表

2.1 引言

日常生活中，在银行办理业务时需要排队，做游戏时也常会围坐成一圈击鼓传花，新生入学时需要在长长的表格中核对信息…… 这些表象的背后都是什么呢？它们都与逻辑表示中的线性结构相关。

线性结构的特点是：在数据元素的非空有限集合中，

（1）存在唯一一个被称为“第一个”的数据元素；

（2）存在唯一一个被称为“最后一个”的数据元素；

（3）除第一个元素外，集合中每个元素均只有一个前驱；

（4）除最后一个元素外，集合中每个元素均只有一个后继。

简单的说，数据元素之间存在一对一的关系，即前驱或后继的关系，也就是线性关系，所以称为线性结构。线性表是线性结构的基础，并由此衍生出栈、队列等。

2.2 线性表的抽象数据类型

线性表（Linear List）是一种简单的数据结构，是由 n 个**相同特性**的数据元素组成的有限序列。

线性表的数据元素在不同的情况下有不同的含义，可以是一个整数、一个符号、一条记录等。例如，某学生一学期 6 门课成绩组成的线性表为（76，88，90，67，87，85），表中数据元素为整数。再如某班级学生信息组成的线性表中（见表 2.1），每个数据元素就是一个学生的信息，包括学生学号、姓名、性别、出身年月、家庭地址等**数据项**（item）构成，称为一条**记录**（record），整个线性表就形成了一个**数据文件**（file）。

表 2.1　某班级学生信息表

学号	姓名	性别	出生年月	家庭住址
161910101	程　强	男	2000. 9	江苏省南京市
161910102	王晓红	女	2001. 4	四川省成都市
161910103	徐　平	男	2000.10	浙江省绍兴市
161910104	刘　磊	男	2001. 5	湖南省长沙市
…	…	…	…	…

从上面的例子可以看出，线性表中的数据元素可以各种各样的，但同一线性表中的元素必定具有相同特性，相邻数据元素之间存在着一种有序关系。一般的，将线性表记为

$$L=(a_1,a_2,...,a_{i-1},a_i,a_{i+1},...,a_n) \tag{2.1}$$

则表 L 中 a_1 称为第一个元素，无前驱；a_n 称为最后一个元素，无后继；a_{i-1} 是 a_i 的前驱，a_{i+1} 是 a_i 的后继。数据元素个数 n 为线性表 L 的**长度**，当 $n=0$ 时称为**空表**。a_i 是线性表 L 的第 i 个元素，i 称为数据元素 a_i 在线性表 L 中的位置。

线性表是一种灵活、方便的数据结构，对线性表的数据元素不仅可以访问，还可以进行插入和删除。线性表的长度可随数据元素的增删而改变。

线性表的抽象数据类型定义如下。

```
ADT List
```

{

数据对象：D = $\{a_i \mid a_i \in \text{ElemSet}, i = 1,\ 2,\ ...,\ n,\ n \geq 0\}$

数据关系：R = $\{\ < a_{i-1}, a_i > \mid a_{i-1}, a_i \in D, i = 2, ..., n\}$

基本操作：

//1. 初始化、销毁、清空操作

InitList(&L)

操作结果：构造一个空的线性表 L 。

DestroyList(&L)

初始条件：线性表 L 已存在。

操作结果：销毁线性表 L 。

ClearList(&L)

初始条件：线性表 L 已存在。

操作结果：将线性表 L 重置为空表。

//2. 访问类操作

ListEmpty(L)

初始条件：线性表 L 已存在。

操作结果：若线性表 L 为空表，则返回 TRUE，否则返回 FALSE。

ListLength(L)

初始条件：线性表 L 已存在。

操作结果：返回线性表 L 中元素个数。

GetElem(L, i, &e)

初始条件：线性表 L 已存在，且 $1 \leq i \leq ListLength(L)$ 。

操作结果：用参数 e 返回线性表 L 中第 i 个元素的值。

LocateElem(L, e)

初始条件：线性表 L 已存在。

操作结果：返回线性表 L 中第 1 个与参数 e 相同的数据元素的位置。若这样的元素不存在，则返回值为 0。

PriorElem(L, cur_e, &pre_e)

初始条件：线性表 L 已存在。

操作结果：若 cur_e 是线性表 L 中的数据元素，且不是第一个元素，则用 pre_e 返回其前驱元素，否则操作失败， pre_e 无意义。

NextElem(L, cur_e, &next_e)

初始条件：线性表 L 已存在。

操作结果：若 cur_e 是线性表 L 中的数据元素，且不是最后一个元素，则用 $next\_e$ 返回其后继元素，否则操作失败， pre_e 无意义。

ListTraverse(L)

初始条件：线性表 L 已存在。

操作结果：从线性表 L 第一个元素开始，依次逐个访问并输出线性表的数据元素。

//3. 加工类操作

SetElem(&L, i, &e)

初始条件：线性表 L 已存在，且 $1 \leq i \leq ListLength(L)$ 。

操作结果：将线性表 L 中第 i 个元素的值用参数 e 替换，并将旧值用参数 e 返回。

InsertElem(&L, i, e)

初始条件：线性表 L 已存在，且 $1 \leq i \leq ListLength(L)+1$ 。

操作结果：在线性表 L 中第 i 个位置上插入新的数据元素 e ，原来第 i 个到第 n 个元素依次向后移动一个位置，线性表 L 的长度加 1。

DeleteElem(&L, i, &e)

初始条件：线性表 L 已存在，且 $1 \leq i \leq ListLength(L)$ 。

操作结果：删除线性表 L 中第 i 个位置上的数据元素，并用参数 e 返回其元素值，原来第 $i+1$ 个到第 n 个元素依次向前移动一个位置，线性表 L 的长度减 1。

} // ADT List

上面线性表抽象数据类型中定义了三类基本操作。需要注意的是 DestroyList 和 ClearList 的区别，ClearList 只是把线性表中数据元素清空，线性表所占用的存储空间并不释放，而 DestroyList 则是彻底销毁线性表，并将从系统中申请的存储空间交还给操作系统。这两个操作可类比于学生离校时清空宿舍空间和宿舍楼不再需要被拆除这两件事。另外，还要注意的是线性表中元素的位置 i 和元素值 a_i 这两者的区别，类比一下，元素位置相当于房间号码，元素值相当于房间中住户。

根据上述定义的线性表抽象数据类型，就可以应对一些实际问题了。下面举个例子，也可以说明如何将现实中的问题转换到信息世界中的逻辑表示并解决。

例 2-1 旅游达人比拼。两位旅游达人小文和小明在南京偶遇，自然聊起去过的城市并相互比拼。小文对小明去过的但自己没去过的城市比较向往。你有什么办法帮小文尽快找出这些城市呢？

自然的，我们想到的方法就是把小文和小明分别去过的城市各列一个清单，然后在小明的清单中划掉小文也去过的城市，小明清单上剩下的城市就是小明去过但小文没去过的城市了。显然，这个里面最关键的操作就是在小明的清单里划掉小文去过的城市。

相应的，在信息世界里，可以用线性表来表示和解决这个问题。 小文去过的城市清单可用线性表 La 表示，小明去过的城市清单可用线性表 Lb 表示。则找出小明去过但小文没去过的城市列表 Lc 就可以用集合操作 $Lc = Lb - (La \bigcap Lb)$ 表示。至此，我们就用数学的语言将现实世界中的问题在信息世界中表述清楚了。

那具体如何实现这个操作呢？我们就要结合前面给出的 List 的基本操作来组合实现。

首先，我们用自然语言来描述这个操作的基本步骤。

第 1 步：从 Lb 中取出一个元素（城市） bi；

第 2 步：将 bi 与 La 中元素进行比较，看看是否有相同，如过没有相同，则转第 3 步，否则放弃该元素 bi 并转第 1 步；

第 3 步：将 bi 添加到列表 Lc 中。

显然，从 Lb 中依次取出每个元素，执行上述操作，最终得到的就是所需要的结果 Lc。

其次，将上述的自然语言描述步骤转换成 List 的基本操作实现。

第 1 步： GetElem(Lb, i, &bi)

第 2 步： pos = LocateElem(La, bi)

if(pos == 0) 转第 3 步； else 转第 1 步

第 3 步：ListInsert(Lc, j, bi)

至此，我们已经写出了该操作中重要的部分，后面就是进行补全这个算法的其他部分了。最终的结果如下。

```
1   void FindNewCity( List La, List Lb, List &Lc )
2   {     // Lc = Lb – (La ⋂ Lb)
3         InitList( Lc );
4         i = 1;  j = 1;
5         Lb_Len = ListLength( Lb );
6         while ( i <= Lb_Len )
7         {
8               // 第 1 步：从 Lb 中取出一个元素（城市） bi ；
9               GetElem( Lb, i, bi );
10              // 第 2 步：将 bi 与 La 中元素进行比较，看看是否有相同
11              pos = LocateElem( La, bi );
12              // 第 3 步：将 bi 添加到列表 Lc 中
```

```
13                  if ( pos == 0 )
14                  {
15                      ListInsert( Lc, j, bi );
16                      j++;
17                  } //end if ( pos == 0 )
18                  i++;    // 依次向后取一个元素
19          } //end while
20  } //end  FindNewCity
```

算法 2.1 查找小明去过但小文没去过的城市

可见，将一个实际问题在计算机中进行表示和解决，有两个要点：一是选择合适的数据结构进行表示；二是基于该数据结构的基本操作，设计相关算法步骤解决问题。上例中，选择线性表结构进行表示，通过一系列基本操作的组合解决了该问题。需要注意的是，算法的设计也是从关键步骤的推导开始，逐步完善的。上例可以看出，先抓住问题的本质是$Lc = Lb - (La \bigcap Lb)$的操作，首先用 3 个步骤实现最核心的解决方法，然后再逐步完善整个算法描述。这提供了算法设计的通常思路，并不是从头至尾一行一行顺序写出来的，而是从核心步骤开始，逐步向外扩展出来的。

我们不能止步于此，还需要分析一下该算法的时间复杂度，并考虑在后续具体实现时的优化策略。该算法主要有两重循环构成，外层是第 6 行确定的，执行次数是 Lb_len，内层是由第 11 行确定的，执行次数最多是 La_len，那这个算法总的时间复杂度将为$O(Lb_len \times La_len)$，如果两者的规模都为n的话，则复杂度是$O(n^2)$量级。那么如何优化呢？显然问题的重点在第 11 行语句 LocateElem 操作上，如何让查找定位的时间变少。我们可以规定每个城市有一个整数型的城市代码 CityNo，每个线性表中城市列表按照城市代码的升序排列。这样，就不需要在整个线性表La中查找，只需从上次查找结束的位置继续向后查找，直至找到相同城市或者后续的城市代码比待查找城市代码大，即可停止。这样，算法总的时间复杂度可降为$O(Lb_len + La_len)$，成为$O(n)$量级的。这个优化的例子可以看出数据结构的魅力所在。

2.3 线性表的顺序表示与实现

用顺序存储方式实现的线性表称为**顺序表**（Sequential List），它是用一维数组作为其存储结构的。

2.3.1 顺序表的定义和特点

（1）顺序表的定义

顺序表是用一组地址连续的存储空间依次存储线性表的数据元素。顺序表数据元素的地址决定了它们之间的关系，也就是说，以数据元素在计算机内物理位置的相邻来表示线性表中数据元素之间的逻辑关系。

（2）顺序表的特点

顺序表的特点如下：

1）各个数据元素的逻辑顺序与其存放的物理顺序一致。

2）对顺序表中所有元素，既可以进行顺序依次访问，也可以进行随机直接访问。

3）顺序表用一维数组实现时，存储空间可以是静态分配的，也可以是动态分配的。

4）顺序表所能存放的数据元素个数受数组的空间大小约束。

以 C/C++代码实现为例，顺序表的存储示意图如下图所示。值得注意的是，在逻辑描述中，我们从 1 开始，但在 C/C++的数组实现时，下标是从 0 开始的。

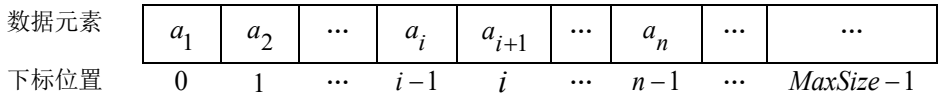


图 2.1 顺序表的存储示意图

假设顺序表的起始地址是 $Loc(a_1)$，第 i 个数据元素的存储位置为 $Loc(a_i)$，则有

$$Loc(a_i) = Loc(a_1) + (i-1) \times sizeof(ElemType) \qquad (2.2)$$

2.3.2 顺序表的存储结构

描述顺序表的存储表示有两种方式，静态方式和动态方式。

（1）静态存储表示

静态存储描述方式如下：

```
#define MAXSIZE  256
typedef struct
{
   ElemType data[MAXSIZE];       //静态连续存储空间
   int   length;                 //存储数据元素的个数
}SeqFixedList;
```

静态存储描述方式下，顺序表的大小在声明时已经确定，一旦数据空间占满，再加入新元素时就会溢出。

（2）动态存储表示

动态存储描述方式如下：

```
#define LISTINITSIZE       256    //初次分配空间大小
#define LISTINCREMENT   128    //空间分配增量大小
typedef struct SeqList
{
   ElemType *pData;              //动态存储空间的基地址
   int   length;                 //存储数据元素的个数
   int   size;                   //当前已分配的存储空间大小
}SeqList;
```

动态存储描述方式下，顺序表的存储空间是在程序执行过程中通过动态存储分配语句 malloc 或 new 进行申请，一旦数据空间占满，可以申请更大的存储空间以替换原来的存储空间，从而达到扩充存储空间的目的。由于存储空间的大小不固定，所以在结构体中增加了 size 变量来记录当前已分配的存储空间的大小。

从这两种方式可见，动态存储表示的顺序表灵活性较大，下节的顺序表主要操作实现部分将以动态存储表示为基础进行讨论。

2.3.3 顺序表基本操作的实现与性能分析

本节根据线性表抽象数据类型中给出的基本操作，采用顺序表的动态存储方式，给出具体实现的伪代码。

（1）初始化、销毁和清空操作

```
Status InitList( SeqList &L )
{  //初始化顺序表
   L.pData = (ElemType *)malloc(LISTINITSIZE*sizeof(ElemType);    //申请存储空间
   if( L.pData == NULL )  exit(OVERFLOW);                          // 存储空间申请失败
   L.size    = LISTINITSIZE;                                       //当前已分配的存储空间大小
```

```
    L.length = 0;                                   //存储数据元素个数为零
    return OK;
} // InitList
```

算法 2.2 顺序表初始化

```
Status DestroyList( SeqList &L )
{   //销毁顺序表
    if( L.pData ！ = NULL )
    {
        free(L.pData);
        L.pData = NULL;
    }
    L.size   = 0;
    L.length = 0;
    return OK;
} //DestroyList
```

算法 2.3 顺序表销毁

```
Status ClearList( SeqList &L )
{   //清空顺序表
    L.length = 0;
    return OK;
} //ClearList
```

算法 2.4 顺序表清空

从算法 2.3 和算法 2.4 的实现代码可以看出顺序表销毁和清空操作两者之间的区别。销毁操作时彻底释放顺序表的存储空间；而清空操作时只是将 L.length 置为 0 表示清空数据元素，但存储空间并未释放。

（2）访问类操作

顺序表访问类操作的实现较为简单，下面给出两个实例，其他操作读者可以自行实现。

```
Status GetElem( SeqList L, int i, ElemType &e )
{   //获取顺序表第 i 个数据元素
    if( i<1 || i>L.length )     //参数检查
        return PARA_ERROR;
    e = L.pData[i-1];           //获得数据元素
    return OK;
} //GetElem
```

算法 2.5 顺序表元素获取

```
Status LocateElem( SeqList L, ElemType e )
{   //查找元素 e 所在的位置
    for( i=0; i<L.length; i++)
    {
        if ( L.pData[i] == e )
            return i+1;    //查找成功
    }
    return 0;              //查找失败
} //LocateElem
```

算法 2.6 顺序表元素查找

在算法 2.5 和算法 2.6 中，数据元素所在的位置均从 1 开始计数，与逻辑描述保持一致；数组中的实际物理位置从 0 开始，与 C/C++语言实现保持一致，所以存在减一和加一的处理。

（3）加工类操作

顺序表加工类操作中，增加和删除数据元素时需要移动顺序表内相应元素，移动时需注意移动的顺序和是否有足够的空间。具体示例可见算法 2.7 和算法 2.8。

```
Status InsertElem( SeqList &L, int i, ElemType e )
{   //在顺序表第 i 个位置上（逻辑位置）插入数据元素 e
    if( i<1 || i>L.length+1 )     // 参数检查
        return PARA_ERROR;
    if(  L.legth >= L.size )      // 当前存储空间已满，需增加存储空间
    {
        newbase = ( ElemType* ) realloc( L.elem, (L.size+LISTINCREMENT)*sizeof(ElemType) );
        if ( newbase == NULL )  exit(OVERFLOW); // 内存申请失败
        L.pData = newbase;
        L.size   +=  LISTINCREMENT;
    }
    //从最后一个元素开始，直到第 i-1 个（物理位置）的位置，依次向后挪一个位置
    for( j = L.length-1; j>=i-1; j-- )
```

```
        L.pData[j+1] = L.pData[j];
    L.pData[i-1] =  e;          //在数组第 i-1 的位置（物理位置）上插入元素 e
    L.length     += 1 ;         //顺序表的长度加一
    return OK;
} //InsertElem
```

算法 2.7 顺序表元素插入

算法 2.7 中，主要分为三个步骤。

第 1 步是参数合法性检查和存储空间是否足够的检查。空间的申请扩充使用了 realloc 这个函数，它将申请一块新的存储空间，并将原有数据迁移到新空间中并释放原有空间，所以需要将新空间的首地址赋值给 pData。同时要注意顺序表的大小 size 也随之改变。

第 2 步是腾出位置并在数组第 i-1 的位置（物理位置）上插入新元素。数据元素移动是从最后一个元素开始依次向后移动一个位置，不能反方向操作，否则会覆盖其他数据元素。图 2.2 给出了在顺序表第 4 个位置插入数据元素 20 的示意图。

第 3 步是修改顺序表长度，值增加 1。

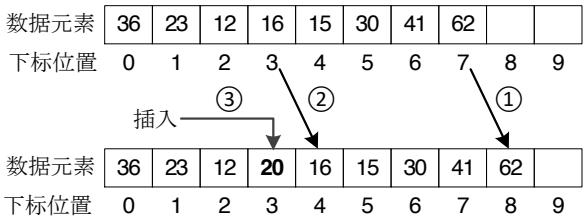


图 2.2 顺序表数据元素插入操作示意图（在顺序表第 4 个位置（逻辑位置）插入 20）

```
Status DeleteElem( SeqList &L, int i，ElemType &e )
{   //将顺序表第 i 个（逻辑位置）元素删除，并用 e 返回
    if( i<1 || i>L.length )   // 参数检查
        return PARA_ERROR;
    e = L.pData[i-1];          // 第 i 个元素存储在数组 i-1 位置上
    //从第 i 个位置（物理位置）开始到最后一个元素，依次向前挪一个位置
    for( j=i; j<=L.length-1; j++)
    {
        L.pData[j-1] = L.pData[j];
    }
    L.length  -= 1;            //顺序表长度减一
    return 0;                  //查找失败
} //DeleteElem
```

算法 2.8 顺序表元素删除

算法 2.8 与算法 2.7 类似，也是三个步骤。读者可以根据这两段代码的对比分析得出该算法的注意点。

不难发现，插入和删除算法的时间复杂度主要受数据移动次数影响。 对于插入算法来说，最好的情况是在顺序表的表尾第 $n+1$ 位置处插入，不需要移动数据元素，也即移动 0 次；最坏的情况是在顺序表的表头第 1 位置处插入，需要将表中 n 个元素全部向后移动一次，也即移动 n 次。最好情况和最坏情况两者的差距是较大的。再讨论一般情况，假设在顺序表第 i 位置处插入，需要移动数据元素的次数则为 $n-i+1$ 次，再假设顺序表中各位置插入数据元素的概率是相等的，则数据元素平均移动次数 AMN（Average Moving Number）为

$$AMN=\frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1)=\frac{1}{n+1}(n+...+1+0)=\frac{1}{n+1}\cdot\frac{n(n+1)}{2}=\frac{n}{2} \qquad (2.3)$$

同理，对于删除算法来说，最好的情况也是在顺序表的表尾删除，不需要移动数据元素；最坏的情况是在顺序表的表头删除，需要将表中剩下的$n-1$个元素向前移动一次，也即移动$n-1$次。对于一般情况，在第 i 位置处删除，需要移动数据元素的次数则为$n-i$次，同样假设各位置删除概率相等，则数据元素平均移动次数 AMN 为

$$AMN=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{1}{n}[(n-1)+...+1+0)=\frac{1}{n}\cdot\frac{n(n-1)}{2}=\frac{n-1}{2} \qquad (2.4)$$

可见，对于顺序表来说，不论是插入还是删除操作，最好的情况都是在表尾进行，最坏的情况都是在表头进行。这是顺序表操作的显著特点，在下节中再与链式表示的方式进行对比。

2.4 线性表的链式表示与实现

用顺序方式来表示线性表，其优点是存储结构简单，空间利用率高，存取速度快。但对于数据元素的动态变化，如插入或删除元素时，平均需要移动一半元素，效率较低。为解决这个问题，另外一种存储方式链式表示出现了。用链式表示的线性表也称为**链表**（Linked List）。 链表不要求逻辑上相邻的数据元素在物理位置上也相邻，是通过指针来表示前驱、后继的关系，因此它使用于插入或删除频繁、存储空间需求不定的情形。当然，链表没有顺序表所具有的弱点，但同时也失去了顺序表可随机存取的优点。

2.4.1 单链表

2.4.1.1 单链表的定义和特点

线性表链式存储结构的特点是用被称为结点（node）的存储单元存储元素，结点的物理位置可以任意。为了表示数据元素之间的前驱、后继的关系，结点中包含指针项，用来指出它的前驱、后继结点的物理位置。也就是说，结点的指针项是存放它的前驱、后继结点的存储地址，所谓的链表中的链就是由指针串起来的。

单链表（single linked list）是一种最简单的链表表示，也叫做线性链表。单链表每个结点有两个域组成：一个数据域（data）存放数据元素，一个指针域（next）存放指向该链表中下一个结点的指针（后继结点的存储地址）。 单链表的结构示意如下图所示。

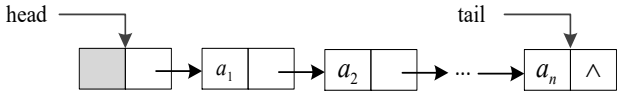


图 2.3 单链表结构示意图

由上图可见，单链表的特点如下：

（1）单链表中数据元素的逻辑顺序与其物理存储顺序可能不一致，一般通过单链表的指针将各个数据元素按照线性表的逻辑顺序链接起来。

（2）单链表的长度扩充较方便。 只要可用存储空间足够，就可以为新的数据元素分配一个结点，并通过指针修改操作链接到现有单链表中。

（3）对单链表的访问操作只能从头指针开始，逐个结点进行访问，不能像顺序表一样直接访问某个指定结点。

（4）当进行插入和删除操作时，只需修改相关结点的指针域即可，不需要移动其他元素的存储位置。

（5）由于单链表的每个结点带有指针域，因此存储空间消耗要比顺序表大。

2.4.1.2 单链表的存储结构

单链表结点的存储结构定义如下：

```
typedef struct LNode
{
   ElemType      data;          //数据域
   struct LNode  *next;         //指针域
} LNode, *LinkList;
```

为了操作的方便，经常会同时记录单链表的头结点指针、当前指针位置、尾指针位置和单链表的长度信息等，可采用如下结构体记录：

```
typedef struct SListInfo
{
   LinkList      head;          //表头结点指针
   LinkList      tail;          //表尾结点指针
   LNode         *pCurNode;     //当前结点指针位置
   int           length;        //单链表的长度（元素个数）
} SListInfo;
```

通过记录尾指针和当前指针位置，便于在链表的表尾插入新元素结点，便于在当前结点后插入和删除新的元素结点。当然，由于尾指针和当前指针的引入，在各种操作时需要及时更新信息，避免成为无效的“野”指针。记录并及时更新单链表长度 length 信息，可在跟单链表长度相关操作时无需再遍历整个单链表得到其长度。除了头结点指针，所有这些额外信息的引入，都是采用了“以空间换时间”的思路，降低跟查找定位相关的操作时间复杂度。例如在表尾插入新元素结点，就无需再从表头结点逐个遍历到表尾再插入新结点，而是直接在表尾结点后插入新结点并同时更新表尾结点指针。需要注意的是，其他教材中大多没有给出尾指针等附加信息，阅读时注意区分。

2.4.1.3 单链表的基本操作实现与性能分析

（1）初始化、销毁和清空操作

```
Status InitList( SListInfo &L )
{  //初始化单链表
   L.head = (LNode *)malloc(sizeof(LNode);                  //申请头结点存储空间
   if( L.head == NULL )  exit(OVERFLOW);                    // 存储空间申请失败
   L.head.next   = NULL;                                    //头结点后无其他结点
   L.tail        = L.head;                                  //尾指针也指向头结点
   L.pCurNode = L.head;                                     //当前指针也指点头结点
   L.length      = 0;                                       //单链表长度为零
   return OK;
} // InitList
```

算法 2.9 单链表初始化

```
Status DestroyList( SListInfo &L )
{  //销毁单链表
   while ( L.head.next != NULL )       //从头结点处，逐个释放链表中的结点
   {
       p = head.next;
       head.next = p.next;
       free(p);
   }
   free(L.head);
   L.head        = NULL;
   L.tail        = NULL;
   L.pCurNode = NULL;
```

```
    L.length      = 0;
    return OK;
} // DestroyList
```

算法 2.10 单链表销毁

上述算法 2.10 单链表销毁操作中，关键是从头结点处开始逐个释放链表中的结点，这样的好处是只要 n 次释放操作即可，如果是从表尾开始释放，每次要从表头走到表尾前一个结点，然后释放表尾结点，需要额外 $n(n-1)/2$ 次遍历操作。

单链表清空操作与销毁操作相似，只是最后不需要释放头结点。单链表的清空操作无法与顺序表一样只是将 length 赋值为 0 而保留结点复用。

（2）访问类操作

单链表访问类操作中访问第 i 个结点，与顺序表的直接读取不同，需要从表头开始遍历。

```
Status GetElem( SListInfo &L, int i, ElemType &e  )
{    //获取单链表第 i 个数据元素
     if( i<1 ‖ i>L.length )      //参数检查
          return PARA_ERROR;
     p = L.head->next;
     j = 1;
     while( j<i )                 // 还未到达第 i 个元素，指针和计数器同步更新
     {
          p = p->next;
          j++;
     }
     e = p->data;                 //获得数据元素
     L.pCurNode = p;              //单链表的当前指针指向该结点
     return OK;
} //GetElem
```

算法 2.11 单链表数据元素获取

上述算法可见，在单链表的访问中，由于不存在物理存储上的位序概念，链表中结点的序号可以与逻辑描述保持一致。

访问类其他操作实现较为简单，读者可以自行实现。

（3）加工类操作

单链表加工类操作中，增加和删除数据元素时虽然不需要移动其他数据元素，但将新的结点接入单链表或从单链表中取下也需注意，特别是防止链表出现断裂，剩余部分由于指针的丢失而无法访问。具体示例可见算法 2.12 和算法 2.13。

```
Status InsertElemAfterCurNode( SListInfo &L, ElemType e )
{    //在单链表当前结点之后插入新结点存入数据元素 e
     //1. 申请新的结点 s
     s = (LNode *) malloc( sizeof(LNode) );
     if ( s == NULL ) exit(OVERFLOW); // 内存申请失败
     s.data = e;
     //2. 将结点 s 链接到 pCurNode 结点之后
     s->next              = L.pCurNode->next;
     L.pCurNode->next  = s;
     //3. 根据当前结点是否为表尾结点，进行表尾结点指针更新
     if ( L.tail == L.pCurNode)
     {    //更新表尾指针
          L.tail = s;
     }
     //4. 单链表的长度加 1
     L.length  += 1;
     return OK;
```

```
} //InsertElemAfterCurNode
```

算法 2.12 单链表当前结点后插入新结点

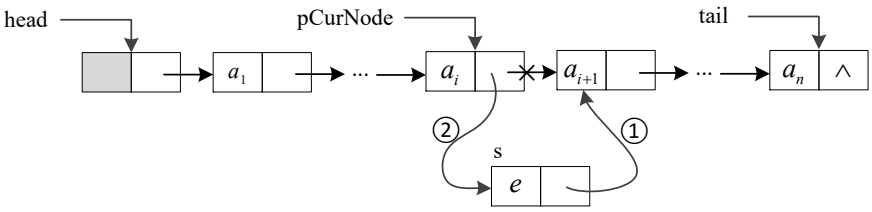
实现新结点插入的算法 2.12 中，关键操作在于算法中第 2 部分，通过两步操作将新结点 s 链接到单链表中，这两部操作顺序的示意图如图 2.4(a)所示，这两者的顺序不能颠倒，否则就会失去当前结点的后续链接的控制。

相应的，当前结点后续结点的删除算法如下。

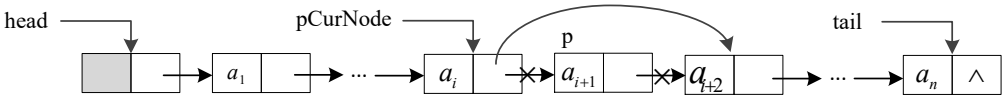
```
Status DeleteElemAfterCurNode( SListInfo &L, ElemType &e )
{    //将单链表当前结点之后的结点删除，并用 e 返回
     if ( L.pCurNode->next == NULL ) return DELE_FAIL;  // 当前结点为最后结点，后续无结点可删除
     //1. 将待删除结点的数据元素赋值给 e
      e = L.pCurNode->next.data;
     //2. 删除当前结点的下一个结点
     p = L.pCurNode->next;
     L.pCurNode->next = p->next;
     free(p);
     if (L.pCurNode->next == NULL)  L.pTail = L.pCurNode;  // 若删除结点是尾结点则修改尾指针
     //3. 单链表长度减 1
     L.length  -= 1;
     return OK;
} //DeleteElemAfterCurNode
```

算法 2.13 单链表当前结点后删除下一结点

上述算法 2.13 中，同样第 2 部分的操作要注意，在结点删除前首先要用一个指针变量指向待删结点，然后将链表重新链接，最后在内存里释放待删结点，可见图 2.4(b)。



（a）在 pCurNode 后插入新结点 s



（b）在 pCurNode 后删除结点 p

图 2.4 单链表插入删除数据元素示意图

另外，如果没有当前指针和表尾指针等辅助信息，则结点插入和删除操作的时间复杂度会增大，每次需要从表头开始，走到待插入或删除的位置才行。最坏的情况就是每次在表尾插入或删除，都要将链表遍历一遍；最好的情况则是每次在表头插入或删除，无需遍历链表。这与顺序表的情况刚好相反。

还有一个值得注意的问题，即使记录了当前结点指针，由于单链表是单向链接，在当前结点之前进行结点的插入和删除依然麻烦，需要再增加一个指针从表头开始遍历并停在当前结点之前的结点位置才行，这将显著增加算法的时间复杂度。为此，其他形式的链表也被提出，下一小节将具体讨论。

2.4.2 其他形式的链表

在基本的单链表基础上，为了满足各种应用的需要，双向链表、循环链表和双向循环链表等其他形式的链表将在本小节具体阐述。

2.4.2.1 双向链表

从上节单链表的算法分析可见，单链表的单向性导致了无法逆向搜索的缺点。 自然的，人们就想到增加一个从后向前的链接，这样就形成了**双向链表**（double linked list），如图 2.5 所示，对应的结构体修改如下。

```
typedef struct DuLNode
{
    ElemType          data;          //数据域
    struct DuLNode   *prev;          //指向前一个结点
    struct DuLNode   *next;          //指向下一个结点
} DuLNode, *DuLinkList;
typedef struct DuListInfo
{
    DuLinkList        head;          //表头结点指针
    DuLinkList        tail;          //表尾结点指针
    DuLNode          *pCurNode;      //当前结点指针位置
    int               length;        //单链表的长度（元素个数）
} DuListInfo;
```

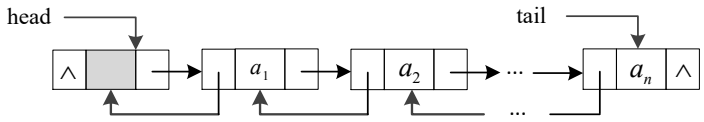


图 2.5 双向链表结构示意图

显然，双向链表的访问类操作要比单链表更为便利些，既可以从前向后搜索，也可以从后向前搜索。对于当前结点之后和之前的插入或删除，时间复杂度也没有区别。只是由于是双链，在插入和删除时需要修改双向的指针，相比于对链表的遍历操作来说额外增加的指针修改还是值得的。 算法 2.14 和算法 2.15 分别给出了在当前结点之前进行结点插入和删除的操作。

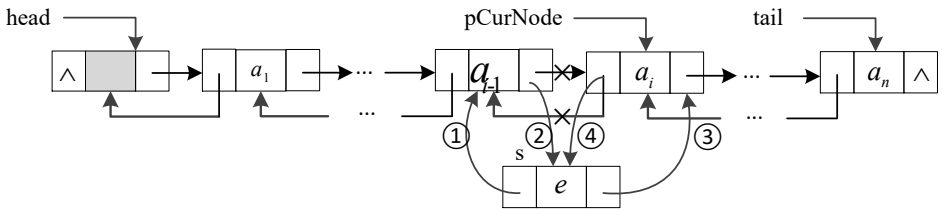
```
Status InsertElemBeforeCurNode( DuListInfo &L, ElemType e )
{    //在双向链表当前结点之前插入新结点存入数据元素 e
     //1. 申请新的结点 s
     s = (DuLNode *) malloc( sizeof(DuLNode) );
     if ( s == NULL ) exit(OVERFLOW); // 内存申请失败
     s.data = e;
     //2. 将结点 s 链接到 pCurNode 结点之前
     s->prev                  = L.pCurNode->prev;
     L.pCurNode->prev->next   = s;
     s->next                  = L.pCurNode;
     L.pCurNode->prev         = s;
     //3. 单链表的长度加 1
     L.length      += 1 ;
     return OK;
} //InsertElemBeforeCurNode
```

算法 2.14 双向链表当前结点之前插入新结点

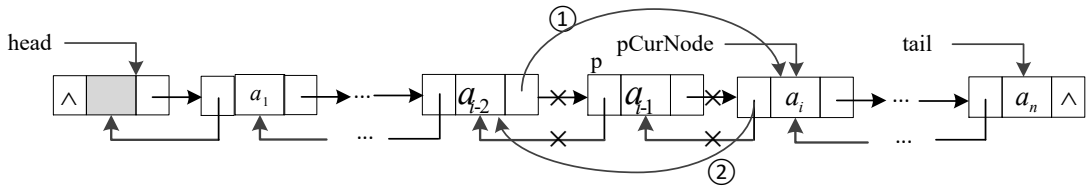
```
Status DeleteElemBeforeCurNode( DuListInfo &L, ElemType &e )
{    //将双向链表当前结点之前的结点删除，并用 e 返回
     if ( L.pCurNode->prev == L.head ) return DELE_FAIL;  // 当前结点为第一个结点，前面无结点可删除
     //1. 将待删除结点的数据元素赋值给 e
      e = L.pCurNode->prev.data;
     //2. 删除当前结点的前一个结点
     p = L.pCurNode->prev;
```

```
    p->prev->next = L.pCurNode;
    L.pCurNode->prev = p->prev;
    free(p);
    //3. 单链表长度减 1
    L.length  -= 1;
    return OK;
} //DeleteElemAfterCurNode
```

算法 2.15 双向链表当前结点之前删除结点



（a）在 pCurNode 前插入新结点 s



（b）在 pCurNode 前删除结点 p

图 2.6 双向链表插入删除数据元素示意图

对在当前结点之后插入和删除结点的算法也类似实现，读者可自行尝试。

在解决实际问题过程中，采用单链表还是双向链表形式，可根据问题需求来决定。如果更多的插入和删除操作是在当前结点之后，采用单链表就可以了；如果要经常进行前后结点位序的变换，例如在输入法中根据词频来调整词语的顺序，则采用双向链表合适。

2.4.2.2 循环链表和双向循环链表

还有一类问题经常也会遇到，正如本章开头提及的围坐一圈进行击鼓传花游戏，会出现到达表尾后折回表头的情形。**循环链表**（circular linked list）就出现了，它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。从表中任一结点出发均可找到表中其他结点，如图 2.7 所示。

```
typedef struct CircListInfo        // 循环链表定义
{
    LinkList        head;          //循环链表表头结点指针
    LNode           *pCurNode;     //当前结点指针位置
    int             length;        //循环链表的长度（元素个数）
} CircListInfo;
```

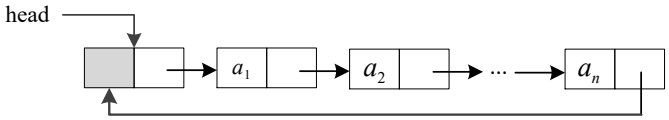


图 2.7 循环链表结构示意图

循环链表的操作与单链表基本一致，差别在于表尾结点的 next 域指针不为空，而是指向表头结点。

相应的，也有双向循环链表，如图 2.8 所示，链表中存在两个环。

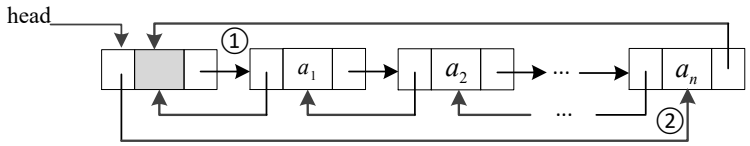


图 2. 8 双向循环链表结构示意图

2. 4. 2. 3 静态链表

在某些环境下无法动态申请结点空间时，可用一维数组作为链表的存储结构，这就是**静态链表**。静态链表中，每个数组元素包括两个数据项，一个是数据元素本身，另外一个是静态链接指针，也即指向下一个数据元素的数组下标，它给出逻辑上下一个结点在数组中的位置。对静态链表进行操作时，可不改变各元素的物理位置，只要修改链接数组元素的链接下标值就可以改变这些元素的逻辑顺序。

静态链表的存储示意图见图 2.9。 图 2.9 中，对应的数据链是从 0 号位置头结点开始，后续的数据元素依次为 23,12,16,20,41,18。另外，数组中未被使用的数组元素也构成了一个空闲空间链接，从 5 号位置开始，还包括 6 号、8 号共 3 个空闲单元。

数组下标	0	1	2	3	4	5	6	7	8	9
data		23	12	20	16			41		18
next	1	2	4	7	3	*6*	*8*	9	*0*	0

图 2. 9 静态链表结构示意图

对应的静态链表结构体定义如下：

```
#define MAXSIZE  256
typedef struct node                    // 静态链表的结点定义
{
    ElemType  data;                    // 结点数据元素
    int           next;                // 指向下一结点的数组下标
} SLinkNode;
typedef struct
{
   SLinkeNode node[MAXSIZE];    // 静态链表的连续存储空间
   int   length;                        // 存储数据元素的个数
}StaticList;
```

2. 5 线性表应用举例

众多《数据结构》教材中都给出了采用单链表进行一元多项式的表示与运算的例子。该例子是单链表应用的经典案例，涉及了有序单链表的生成，两个单链表之间的结点插入、合并、删除等操作。读者可参阅相关教材学习，这里不再赘述。

本节将再通过 3 个实例来说明线性表的常用情形。

例 2-2 旅游达人比拼（单链表实现）。

本章例 2-1 中给出该应用的逻辑表示层面的实现算法 2-1，这里采用单链表的形式具体实现。针对该应用，首先对 ElemType 给出一个具体的定义。

```
typedef struct ElemType          // 城市数据元素定义
```

```
{
    int    cityNo;              // 城市编号
    char  cityName[48];         // 城市名称
} ElemType;
```

假设已经建立的两个单链表 La 和 Lb 已经按城市编号升序排序，创建一个新的单链表 Lc，记录小明去过但小文没去过的城市。

```
void FindNewCity( SListInfo La, SListInfo Lb, SListInfo &Lc )
{    //  Lc = Lb – (La ⋂ Lb)
     InitList( Lc );       // 调用算法 2.9，创建新的单链表 Lc
     Lb_Len = Lb.length;
     if ( Lb_Len==0 )  return LIST_EMPTY;      // Lb 为空表，直接返回
     Lb.pCurNode = Lb.head->next;
     La.pCurNode = La.head->next;
     Lc.pCurNode = Lc.head;
     while ( Lb.pCurNode  != NULL )
     {
          // 第 1 步：从 Lb 中取出一个元素（城市） bi ；
          bi = Lb.pCurNode.data;
          // 第 2 步：将 bi 与 La 中元素进行比较，看看是否有相同
          while ( La.pCurNode != NULL && La.pCurNode->data.cityNo < bi.cityNo )
          {
               La.pCurNode = La.pCurNode->next;  // La 中当前结点的城市编号小于 bi，则继续向前查找
          }
          // 第 3 步：将 bi 添加到列表 Lc 中
          if ( La.pCurNode != NULL && La.pCurNode->data.cityNo == bi.cityNo )
          {    // La 中当前结点城市与 bi 相同，直接跳过
               La.pCurNode = La.pCurNode->next;
          }
          else if ( La.pCurNode == NULL || La.pCurNode->data.cityNo != bi.cityNo )
          {
              InsertElemAfterCurNode( Lc, bi );        // 调用算法 2.12，将 bi 插入到中当前结点之后
              Lc.pCurNode = Lc.pCurNode->next;     // 当前结点指针后移一次
          }
          Lb.pCurNode = Lb.pCurNode->next;    // 依次向后取一个元素
     } //end while
} //end  FindNewCity
```

算法 2.16 采用单链表查找小明去过但小文没去过的城市

对比算法 2.16 和算法 2.1，两者的大体框架是一致的，但算法 2.16 已经结合单链表进行了具体实现。一般在解决实际问题时，首先进行逻辑表示层面的算法设计，如算法 2.1；然后再结合数据结构具体的物理实现方式对算法进行具体化和完善，如算法 2.16。直接看算法 2.16 会容易让人直接陷入细节而困惑，甚至产生畏惧心理。这也是算法设计时的思路，抓住核心问题，逐步完善。

再讨论一下本例中的时间复杂度，其实在 2.2 节中已经给出了该算法的时间复杂度为 $O(Lb\_len + La\_len)$。虽然在算法中有 2 个 while 的嵌套，但实际执行时只遍历单链表 La 和 Lb 各一遍，这得益于单链表中城市结点已经按照城市代码升序排列。

例 2-3 输入法词频调整。使用中文拼音输入法时，同一拼音下会有很多同音词，将使用频率高的词在候选框中向前调整从而更有利于提升用户的使用体验。

针对该问题，直观可以想到的是将候选词集合组成一个线性表，同时记录每个候选词被使用的频度，根据候选词的使用频度进行结点间的顺序调整。具体的，每当选中一个词时，将其使用频度加 1，并跟它前面的结点频度进行比较，如果前面结点频度比它低，则前移，直至前面结点频

度跟它相等或者比它高，后面结点频度比它低。显然，结点向前移动，需要用到双向链表。该问题涉及到 3 个操作步骤：

（1）在双向链表中查找待访问词的结点，找到后将其频度加 1；

（2）将该结点从当前位置摘下来；

（3）找到满足条件的位置后，将其再插入到双向链表中。

对应的，ElemType 做如下定义，其中 freq 的初始值均为 0。

```
typedef struct ElemType          //候选词数据元素定义
{
    char  word[48];              //  词语
    int    freq;                 // 词语频度
} ElemType;
```

整个算法用下面的 Locate 函数表示。

```
Status Locate( DuListInfo &L, char word[ ] )
{     // 输入法词频调整
      //1. 在双向链表中查找待访问词的结点，找到后将其频度加 1
      p = L.head->next;
      while ( p != NULL && strcmp(p->data.word, word)!=0 ) p = p->next;
      if ( p == NULL) return FAILURE;          //  该词语不在双向链表中
      p->data.freq += 1;                       //   找到后将其频度增加 1
      q = p->prev;                             //  q 指向待摘结点的前一个结点
      if (q->data.freq >= p->data.freq )   return OK;              // 位置无需调整，直接返回
      //2. 将该结点从当前位置摘下来
      q->next  = p->next;
      if ( p != L.tail )    p->next->prev = q;        // 待摘结点不是尾结点
      else  L.tail = q;                               // 待摘结点是尾结点
      //3. 找到满足条件的位置后，将其再插入到双向链表中
      while ( q != L.head && q->data.freq < p->data.freq )  q = q->prev;  // q 停在 freq 大于等于 p 的结点处
      //将摘下来的结点 p 插在结点 q 之后
      p->next = q->next;       q->next = p;      p->prev = q;
      if ( q != L.tail )     p->next->prev = p;
      else   L.tail = p;                              // 如果 q 就是尾结点，则更新尾结点指针
      return OK;
} //end  Locate
```

算法 2. 17　词频调整操作算法

从算法 2.17 来看，整体流程与逻辑描述的 3 个步骤一致，但算法实现细节中增加两处处理：一是直接跟前一结点词语的频度比较一次，如果不需要调整位置则直接返回；二是判断待摘结点是否为尾结点和待插入位置是否为尾结点，做相应的处理保证算法的正确性。 从逻辑描述的 3 个步骤到算法 2.17 的实现细节再到用具体编程语言实现，再一次体现了算法设计与实现的思路，抓住核心问题，逐步完善，而不是直接从编程实现开始入手，那将会一筹莫展。

例 2-4 约瑟夫环问题。已知 n 个人（以编号 1，2，…，n 分别表示）围坐在一张圆桌周围，从编号为 1 的人开始报数，数到 m 的那个人出列；他的下一个人又从 1 开始报数，数到 m 的那个人又出列；依此规律重复下去，直到圆桌周围的人全部出列。要求给出这 n 个人的出列结果展示。

该应用是典型的循环链表问题。每个结点依次存放值为 1，2，…，n 的整型数据，整个出列过程可描述为下面 2 个步骤：

（1）从当前结点开始，数到第 m 个结点时，该结点从循环链表中删除并输出对应值；

（2）重复第 1 个步骤直至所有结点都从链表中删除。

对应的算法如算法 2.18 所示。

```
Status Josephus( CircListInfo &L,  int m )
```

```
{    // 约瑟夫环问题
     pre = L.head;                          //用 pre 指针指向当前结点的前一结点，从而便于删除当前结点
     p   = L.head->next;
     if ( p ==  L.head )   return EMPTY_LIST;      //空表直接返回
    //1. 从当前结点开始，继续向前走 m 个结点，并输出删除
    while ( L.head->next != L.head )       // 2.  L.head->next == L.head 为空表，则终止
    {
        // 1. 1 向前走 m 次。
        i = 1;
        while ( i<m )
        {
            pre = p;   p = p->next;   i++;       //向前走一步
            if ( p == L.head )                   // 已折回表头，则再跳过表头结点
            {   pre = p;
                p = p->next;
            }
        }
        //1.2 输出并删除当前结点
        cout<<p->data<<endl;
        pre->next = p->next;
        free(p);
        p = pre->next;
    } //end while
    return OK;
} //end   Josephus
```

算法 2.18 约瑟夫环问题算法

上述算法只给出了约瑟夫环的结点出列操作过程。作为完整的应用，约瑟夫环建立过程也是必须的，读者可自行完成。另外，简单分析一下该算法的时间复杂度，主要有两个 while 循环来确定，外层循环控制的是 n 个结点的输出总次数，内层循环控制的是每次走 m 步，则该算法总的时间复杂度为$O(n\times m)$。

这三个例子分别对应了单链表、双向链表和循环链表的应用。 在各类线性表的应用中，常涉及的操作包括结点的查找、删除和插入。还需要注意的是，在线性表的表头、表尾边界处的操作与普通结点操作的异同。

2.6 小结

本章讨论了最基本的数据结构—线性表，包括其抽象数据类型的描述，顺序表示与实现和链式表示与实现，以及相关的应用举例。

通过本章学习，首先理解如何定义一个数据结构的逻辑表示以及基于该结构的基本操作；然后重点掌握顺序存储和链式存储两种不同方式的具体实现，包括相同操作在两种实现方法中时间复杂度的差异分析，以及为了更好适应应用需要，如何对基本数据结构进行修改和优化；最后，通过实例分析，理解并掌握如何从应用出发，选择恰当的数据结构，构造合适的解决方案，并从逻辑层面的重点步骤描述再到算法具体实现的问题解决全过程。

要更好理解和掌握本章内容，大量的练习是不可或缺的。 同时，本章也是后续各章学习的基础，需要多实践训练，打下扎实基础。 在编程练习时，注意数据本身的排列特性和恰当数据结构的选择与改造，设计高效的算法，追求空间复杂度和时间复杂度的优化。

2.7 练习

（1）通过本章的学习，画出本章内容的思维导图，总结分析顺序存储和链式存储的优缺点。

（2）假设整数集合中有重复元素，给出无序和有序情况下，顺序表和单链表实现时如何删除重复元素进行集合提纯。

（3）采用顺序表和单链表分别实现两个整数纯集合的交、并、补、差运算，并考虑集合数据元素本身是无序和有序两种情况下操作步骤和算法时间复杂度的差异。

（4）给出顺序表和单链表两种情况下线性表的逆置操作算法。所谓逆置，就是将原来的序列$(a_1,a_2,...,a_n)$变成$(a_n,a_{n-1},...,a_1)$形式。

（5）给出单链表形式下，两个一元多项式相加的算法，以及对一元多项式求导的算法。

（6）将本章 2.5 节三个例子编程实现。