

第 3 章 线性表

线性表是一种最基本的数据结构，它不仅自己有着广泛的应用，而且也是其它一些数据结构的基础。后面几章即将讨论的栈、队、串等就是线性表的特例。本章先讨论线性表的逻辑结构与抽象操作，接着给出线性表的两种存贮映射方法，最后以一个经典的问题——一元 n 次多项式相加说明线性表的应用。本章虽讨论的是线性表，但涉及的许多问题（如存贮映射的实现、抽象操作的表示与应用方式等）都带有一定的普遍性。

§ 3.1 线性表的逻辑结构

§ 3.1.1 基本概念

线性表（Linear list）是数据元素的一个有限序列，在这个序列中，每个元素有一个唯一的（直接）前趋和一个唯一的（直接）后继，第一个元素可以无前趋，而最后一个元素也可以无后继。线性表可记为

$$L = (a_1, a_2, \dots, a_n);$$

这里， a_i 为数据元素， $n \geq 0$ 为整数， a_{i-1} 称为 a_i 的前趋（ $i \geq 2$ ）， a_{i+1} 称为 a_i 的后继（ $i < n$ ）， $i = 1, 2, \dots, n$ 。

线性表中元素的个数称为线性表的**长度**。无元素的表（ $n=0$ ）称为**空表**，**空表长度为 0**。

按形式化方法，线性表定义为

$$LL = (D, S)$$

$$D = \{a_1, a_2, \dots, a_n\}$$

$$S = \{r\}$$


$$r = \{ \langle a_{i-1}, a_i \rangle \mid a_i \in D, i = 2, \dots, n \}。$$


从上面定义看出，在一个结构中，如果只关心数据元素所具有的线性相对位置，则可将视为线性表处理。另外，对集合结构（数据元素除了同属于一个集合外，别无其它关系）也可按线性表方法处理

§ 3.1.2 线性表抽象模型

这里，我们将线性表视为一个抽象对象/类（亦称接口），即不考虑它的具体数据结构存储，不考虑基本操作的实现，只考虑它的基本操作的接口（输入/输出）。该抽象对象/类从面向对象观点定义了线性表的属性、方法。由于是抽象的，所以，该类无具体对象，只用做派生各种对象/类，例如，下两节介绍的线性表的顺序存储和链式存储所对应类均是这里的抽象类的派生类。

对抽象类的形式化描述，方法很多，如 C++（抽象类、纯虚类）、OMG IDL 等。

 OMG IDL (Interface Definition Language) 主要用在 OMG (Object Management Group) 的通用对象请求代理结构 CORBA (Common Object Request Broker Architecture) 中描述分布对象接口。在 OMG IDL 中，接口 (Interface) 被定义为某对象的一组可被客户调用的操作和属性，可认为是对象中的面向客户的部分，它忽略了对象中与具体实现有关的部分。所以，接口可认为是对象的使用级别上的抽象。OMG IDL 语法与 C++ 的类十分相似，其许多保留字也与 C++ 相同。

 一般情况下，对象的实现（主要是成员函数）与调用是位于同一机器的同一进程中。如果它们是分开的（分别位于不同的机器或不同的进程中），则这种对象称为分布对象 (Distributed Objects)。在网络环境下，逻辑上属于一个应用系统的软件，往往分布在多部机器上，合作完成任务，此时，分布对象技术可以大大简化这种软件的开发。目前比较流行的分布对象技术/模型/架构有 CORBA、J2EE(EJB)、.NET(DCOM) 等。

这里，我们采用大家比较熟悉 C++。C++ 一般主要用于对象/接口的具体实现，对抽象描述，当然更好的工具是 IDL。C++ 中，纯虚类（类中存在以 virtual 开始，以 "=0" 结束的函数声明）可用来描述抽象对象。

做为准备，先定义专用于线性表类的异常处理类 TExceptionLinearlist。

```
class TExcepLinearList
{
public:
    int errNo;
    char errMsg[CNST_SizeErrMsg]; //以“CNST”打头的，均为自定义常量，
    程序运行时，需在头文件中给出具体的定义。下同

    TExcepLinearList(int mErrNo)
    {
        errNo=mErrNo;
        strcpy(errMsg, errMsgList.GetMessage(errNo));
    }
};
```

在线性表类的成员函数中可能产生异常的地方检测到异常时，使用 throw 抛掷一个 TExceptionLinearlist 对象，产生一个该类型的异常，形式为：


```
throw TExceptionLinearlist (no);
```


这里，no 为具体的异常代码。

该类只设构造函数。当在程序中使用

```
throw TExceptionLinearlist (no);
```

时，所返回的 TExceptionLinearlist 对象的 errNo 被设置为 no，且 errMsg 也被设置为 no 所对应的异常说明信息。

 在 C++ 中，如果一个类含有纯虚函数，则该类称为纯虚类（抽象类）。纯虚函数是型为 “Func(...)=0” 的成员函数，即函数原型后紧跟 “=0”，它不需要（也不容许）给出实现，相当于纯粹的函数原型声明。抽象类不能实例化（即不能用来定义对象），只供继承使用。

 在 C++ 中，可用关键字 `template` 定义/声明模板，基本形式为 `template <class T>`，它表示，T 是个模板（模型）类型（不是具体的类型），在它随后的类或函数（称为类模板或函数模板）定义中，可以将 T 看作已知类型使用。而具体 T 代表什么类型，在使用/调用相应的类模板或函数模板时动态确定。所以，C++ 的模板实现的是可变类型！在传统的语言中，在程序中可以改变变量的值，但不能改变定义变量的类型。可变类型就是解决该问题的。

下面是线性表抽象类：

```
template <class TElem>
```

//上面是模板声明，表明 TElem 是一个可变（调）类型，在使用 TLinearList0 时动态决定

//有了这个声明，TLinearList0 中可直接将 TElem 做为已知类型使用。

```
class TLinearList0
{
protected:

public:
    long len;
    virtual TBool IsEmpty() {if (len <=0) return True; else return False;}
    virtual TElem &Get(long idx) = 0; //声明虚函数，不需要给出实现
    virtual TElem *Set(long idx, TElem &elem) = 0;
    virtual TElem *Prior(long idx) = 0;
    virtual TElem *Next(long idx) = 0;
    virtual TElem *GetAddress(long idx)=0;

    virtual long CountElem(TElem &elem)=0;

    virtual long Locate(TElem &elem, long sn=1) = 0;
    virtual long Locate(TElem &elem, long *foundElem) = 0;

    virtual long LocateFirst(TElem &elem) = 0;
```

```

virtual long LocateNext(TElem &elem) = 0;

virtual TElem *Insert(TElem &elem, long sn=1) = 0;

virtual TElem *Delete(long sn=1)=0;
virtual long Delete(TIndexSelector &sel, TElem **elemDeleted=NULL)=0;
virtual long DeleteByIndex(long *idxTobeDel,long numIdx,
                           TElem *elemDeleted=NULL)=0;

};

```

对程序中变量、类型（类）、函数等成分的命名，采用一套良好的命名体系是一种良好的编程风格，它不仅充分提高程序的易读性，而且有助于避免由名字混乱引起的错误！对多人合作的大型程序，特别如此。我们在本教材中，采用这样的命名体系：**类名**以大写字母 T 打头，其后第一个字母亦大写；**函数名**以大写字母打头；**普通变量名**以小写字母打头，对一些明显的计数或临时值变量，采用简单的字母，如 i,j,k,x,y,z 等，其他采用较明确简单的英文单词；**各种名称**，都尽量采用明确简单的英文单词（或较固定缩写），在同一个名称中需要多个单词时，各单词用大写打头，以示分隔（变量名的第一个单词小写开始）。

各主要部分说明如下。

len：属性，表示线性表的长度。

IsEmpty()：检查表是否为空，空时返回逻辑 True，否则返回 False

Get(long idx)：将序号为 idx 的元素的地址作为函数值返回。idx 非法时，触发异常 TExceptionLinearlist(1)。

Set(long idx, TElem &elem)：将序号为 idx 的元素的值，设置为 elem 所指的元素的值。idx 非法时，触发异常 TExceptionLinearlist(1)。

Prior(long idx)：返回序号为 idx 的元素的上一个元素的地址。idx 非法时，触发异常 TExceptionLinearlist(1)。

Next(long idx)：返回序号为 idx 的元素的下一个元素的地址。idx 非法时，触发异常 TExceptionLinearlist(1)。

GetAddress(long idx)：返回序号为 idx 的元素的地址，不存在该元素时触发异常 TExceptionLinearlist(1)

CountElem(TElem &elem)：返回值为 elem 的元素的个数。

Locate(TElem &elem, long sn=1)：在线性表中查找值为 elem 的第 sn 个元素。sn 为负数时，表示“倒数第 sn 个”。sn 为 1（省缺值）表示第一个，sn 为 -1 表示倒数第一个，余类推。查找到时返回一个整数，表示所找到的第 sn 个值为 elem 的元素的序号，否则，触发异常 TExceptionLinearlist(4)，表示未找到。

Locate(TElem &elem, long *foundElem)：在线性表中查找值为 elem 的各元素，将所找到的各元素的序号，按升序存入 foundElemIndex 所指向的整数数组中。查找到时，返回所找到的元素的个数；找不到时，返回 0。

LocateFirst(TElem &elem)：查找值为 elem 的第一（从序号小的方向起）出现的元素，返回其对应的序号。未发现时返回-1。


LocateNext(TElem &elem)：查找值为 elem 的下一个出现的元素，返回其对应的序号。未发现时返回-1。这里的“下一个”，是指最近一次使用 LocateFirst(对第一次使用 LocateNext)或 LocateNext 所获得的元素的下一个。

利用 LocateFirst 和 LocateNext 可逐个处理值等于给定值的各元素。对链式结构，比使用直接序号定位速度更快。

Insert(TElem &elem, long sn=1)：在第 sn 个元素之前插入一个新元素，使其值为 elem 指向的元素的值。这里，sn 的含义与上面类似。sn 指出的范围超出最大范围时，认为是在最后插入，太小时认为是在最前面插入。成功时返回被插入的对象的地址，否则（即空间不足时）触发异常 TExceptionLinearlist(4)。

Delete(long sn=1)：删除第 sn 个元素，并将其地址返回。sn 非法时触发异常 TExceptionLinearlist(2)。这里，sn 的含义与上面类似。

Delete(TIndexSelector &sel, TElem **elemDeleted=NULL)：删除由 sel 指出的下标选择器（下标选择器的说明见下面的节）所指出的各个元素，并将它们的地址存入 elemDeleted 所指出的数组，将所删除的元素的个数作为函数值返回。sel 非法时触发异常 TExceptionLinearlist(2)，由于内部缓冲空间不足，触发异常 TExceptionLinearlist(4)。

 **关于函数返回的设计的考虑**：在大多数情况下，让函数直接返回一个欲得到的对象/量是很自然的。如，比较读元素函数的两种设计：

```
int Get(long idx, TElem *elem);
```

```
TElem *Get(long idx);
```

前者将所读到的元素的指针返回到 elem，而函数返回值表示操作状态（成功与否）。后者直接将所读到的元素的指针作为函数值返回，此时，函数返回值就不能用来表示状态了，但从使用方面看，后者更为自然，而且也容易作为左值被使用，特别是在面向对象中，返回对象的函数可构成一个调用链，十分方便、自然。此时，函数的操作状态通过异常机制解决。

线性表抽象类 TLinearList0 只描述了线性表的逻辑结构，主要是操作（使用）接口，以接口定义了线性表的逻辑特性。接下来的任务是给出它的具体的计算机程序实现----接口的实现。这首先要涉及具体的存储结构。主要有两种存储结构与实现----顺序和链式。从 C++ 角度讲，这主要是从 TLinearList0 派生具体的类并给出具体的实现。

§ 3.2 线性表的顺序存贮结构

§ 3.2.1 基本存储方法

具体地讲，线性表的顺序存贮（也称连续存贮）方法是，将表中各元素依它们的逻辑次序存贮在一片连续的存贮区域中，使任意两相邻元素之间的存贮单元个数相等。通常，元素之间不留空闲存贮区。

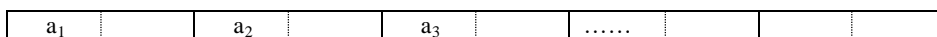


图 3-1 线性表的顺序存储

设线性表为 $L=(a_1, a_2, \dots, a_n)$ ，每个元素占 2 个存贮单元，则 L 的连续存贮方式如图 3-1 所示。

在这种存贮结构中，有下列关系成立

$$\text{Loc}(a_i) = (i-1) * c$$

这里， $\text{Loc}(a_i)$ 表示第 i 个元素 a_i 的相对地址。 c 是每个元素所占的单元个数。

讨论地址问题时，为了不失一般性，我们选定一个基准（参考）存贮单元，问题中所涉及的地址值均以此参考单元为基准（为起点），称这种地址为**偏移地址（相对地址）**，它将基准单元地址值视为 0，任一单元之前到基准单元之间的单元数目，为该单元的相对于该参考单元的偏移地址。显然，任一单元的偏移地址，加上它的参考单元的绝对地址即为该单元的绝对地址。

§ 3.2.2 面向对象描述

(一) 对象结构

线性表的这种存贮结构的存储区域，可用高级语言中的数组模拟，即用一维数组表示存贮区。相应的基本操作，也都是针对该一维数组进行的，因此，从面向对象观点看，顺序存储的线性表，是一个对象，其主要数据成员是一维数组。

为了增强灵活性和实用性，一维数组采用动态空间表示（不是使用高级语言的数组类型），在线性表初始化时，申请指定数目的空间，当在使用中溢出时，重新申请一块较大空间。但元素数目少到一定程度（比例）时，自动还回多余的空间。

设置了两个重要的量：**len** 和 **size**。其中 **len** 用于保存线性表当前实有的元素个数（表长度），是公共量，向外开放，可被作为属性使用，它已在抽象类中定义。**size** 是内部变量，用于存放当前存储区的空间大小。

由于线性表的元素类型可以是任意的（在使用时才决定），所以采用 C++ 的模板机制，将线性表元素的类型定义为模板类型。

下面是顺序存储结构对应的类 TLinearListSqu:

```
template <class TElem>
```

//上面是模板声明, 表明 TElem 是一个可变(调)类型。具体的类型在使用 TLinearListSqu 时

//动态决定。有了这个声明, TLinearListSqu 中可直接将 TElem 做为已知类型使用。

```
class TLinearListSqu : public TLinearList0<TElem> //表示从 TLinearList0<TElem>派生
{
```

```
protected:
```

```
    TElem *room; //room 相当于一维数组, 其元素类型为可变类型 TElem
```

```
    long size;
```

```
    long lastVisited;
```

```
    TElem buffElem;
```

```
    long ResizeRoom(long newSize);
```

```
    long CopyRoom(TElem *objRoom, long n1, TElem *srcRoom, long n2);
```

```
public:
```

```
    TLinearListSqu(void);
```

```
    TLinearListSqu(long mSize);
```

```
    ~TLinearListSqu(void);
```

```
    virtual TElem &Get(long idx);
```

```
    virtual TElem *GetAddress(long idx);
```

```
    virtual TElem *Set(long idx, TElem &elem);
```

```
    virtual TElem *Prior(long idx);
```

```
    virtual TElem *Next(long idx);
```

```
    virtual long CountElem(TElem &elem);
```

```
    virtual long Locate(TElem &elem, long sn=1);
```

```
    virtual long Locate(TElem &elem, long *foundElemIndex);
```

```
    virtual long LocateFirst(TElem &elem);
```

```
    virtual long LocateNext(TElem &elem);
```

```
    virtual TElem *Insert(TElem &elem, long sn=1);
```

```
    virtual TElem *Delete(long sn=1);
```

```

virtual long Delete(TIndexSelector &sel, TElem *elemDeleted=NULL);
virtual long DeleteByIndex(long *idxTobeDel, long numIdx,
                           TElem *elemDeleted=NULL);

void Print(); //Only for test

};

```

该类为上节给出的抽象类 [TLinearList0](#) 的派生类，其函数接口，都在上节中阐述过，这里介绍它的几个成员变量。

room: 指针，指向存储线性表的存储区。我们这里使用动态存储区存储线性表，具体的存储区的申请，在初始化（构造）函数中进行。该存储区按一维数组使用，每个数组元素对应一个线性表元素，类型为类型模板（可变类型）TElem。

size: 指示 room 所指存储区的大小。

len: 线性表长度。已在父类中定义。


其他量将在相应的函数实现中解释。下面介绍其主要函数的实现。

(二) 初始化

初始化的主要工作是设置存储区、给类变量赋初值，使线性表处于空的可使用状态（可插入元素）。这里的初始化通过对象的构造函数实现。

有两个构造函数。第一个不分配存储空间，只进行相应的变量初始化工作。第二个构造函数负责分配指定数量的空间。

与构造函数对应的是析构函数，它的任务是当对象生命期结束后，释放所分配的存储空间。

 在 C++ 中，new 是 算符（注：是运算符吗？？？ --答：正确），其功能是申请动态存储空间，与 Pascal 中的 new 和 C 中的 malloc() 类似。但不同的是，如果使用 new 失败（未能申请到空间），则在一般情况下触发一个异常（bad_alloc）。如果不想让它触发异常，而是象 C 和 Pascal 那样返回空指针，则在 new 后加(nothrow)，即 new(nothrow)。注：有的 C++ 编译器不支持 nothrow 参数。

```

template <class TElem>
TLinearListSqu< TElem>::TLinearListSqu()
{
    size=0;
    len=0;
    room=NULL;
};

template <class TElem>

```



```
TLinearListSqu< TElem>::TLinearListSqu(long mSize)
{
    size=0;
    room=NULL;
    len=0;

    if (mSize<1) throw TExcepLinearList(2);

    room = new(nothrow) TElem[mSize]; //分配存储空间. 注: 有的 C++编译器不支持 nothrow
    if (room==NULL) throw TExcepLinearList(3); //分配失败时抛掷异常

    size = mSize;
};

template <class TElem>
TLinearListSqu< TElem>::~~TLinearListSqu(void)
{
    if (room!=NULL) delete[] room; //释放所分配的空间
};
```

(三) 元素直接访问

Get 和 Set 类函数用于实现按序号（逻辑关系）访问元素。对顺序存储，它们的实现是直接的。

```
template <class TElem>
TElem &TLinearListSqu< TElem>::Get(long idx)
{
    if (idx<0 || idx>=len)
        throw TExcepLinearList(1);
    return room[idx];
};

template <class TElem>
TElem *TLinearListSqu< TElem>::GetAddress(long idx)
{
    if (idx<1 || idx>=len)
        throw TExcepLinearList(1);
    return &room[idx];
};
```

```
};

template <class TElem>
TElem *TLinearListSqu< TElem>::Set(long idx, TElem &elem)
{
    if (idx<1 || idx>=len)
        throw TExcepLinearList(1);
    room[idx] = elem;
    return &room[idx];
};
```

(四) 前驱/后继操作

对顺序存储结构，根据元素的序号求该元素的前驱/后继是很简单的。下面是对应的程序。

```
template <class TElem>
TElem *TLinearListSqu< TElem>::Prior(long idx)
{
    if (idx<1 || idx>=len)
        throw TExcepLinearList(1);
    return &room[idx-1];
};

template <class TElem>
TElem *TLinearListSqu< TElem>::Next(long idx)
{
    if (idx<0 || idx>=len-1)
        throw TExcepLinearList(1);
    return &room[idx+1];
};
```

(五) 查找定位操作

这里给出的一组操作，用于根据元素内容，求出该元素的位置（序号）。由于表中可能有重复值的元素，所以满足条件的元素可能不只一个。

1. Locate(TElem &elem, long sn): 该函数在表中查找值等于 elem 的第 sn 个元素，

存在时返回其在表中的序号（设表第一个元素的序号为 0）。由于 sn 可能是负值（表示倒数第 sn 个，设最后一个的 sn 为-1），所以，分两种情况处理。

```
template <class TElem>
long TLinearListSqu< TElem>::Locate(TElem &elem, long sn)
{
    long i, k;

    k=0;
    if (sn>0) //sn 大于 0 时，从前（小）往后（大）计数
    { for (i=0; i<len; i++)
        if (room[i]==elem)
        {
            k++; //k 记录值等于 elem 的元素的个数
            if (k==sn) return i;
        }
    }
    else //sn 不大于 0 时，从后往前计数
        for (i=len-1; i>=0; i--)
            if (room[i]==elem)
            {
                k++;
                if (k== -sn) return i;
            }

    if (k>0) return i; // "sn" 大于匹配的元素个数，返回最后匹配元素的下标
    else return -1; //未发现
};
```

2. Locate(TElem &elem, long *foundElemIndex): 用于求出值为 elem 的所有元素的序号，并存入 foundElemIndex 所指出的一维数组中。该数组必须由调用者提供，且要保证足够的空间。

```
template <class TElem>
long TLinearListSqu< TElem>::Locate(TElem &elem, long *foundElemIndex)
{
    long i, k;

    k=0;
```

```
for (i=0; i<len; i++)
    if (room[i]==elem) //假定 TElem 型的量支持算符"=="
    {
        foundElemIndex[k]= i;
        k++;
    }
return k;
};
```

3. LocateFirst(TElem &elem)和:LocateNext(TElem &elem): 这是两个配合使用的函数，用于逐个访问值为 elem 的各元素。LocateFirst 查找值为 elem 的第一个出现的元素的序号，LocateNext(TElem &elem)则查找值为 elem 的下一个出现的元素，这里的下一个是指最近一次调用 LocateNext 所找到的元素的下一个。若调用 LocateFirst 后尚未调用 LocateNext，则下一个是指 LocateFirst 所找到的元素的下一个。当在明确调用 LocateFirst 前调用 LocateNext，则结果不可预料。

这两个函数的实现，主要使用了一个内部变量 lastVisited，它记下最近一次调用 LocateFirst/LocateNext 所找到的元素的序号。

```
template <class TElem>
long TLinearListSqu< TElem>::LocateFirst(TElem &elem)
{
    long i;

    for (i=0; i<len; i++)
        if (room[i]==elem)
        {
            lastVisited=i;
            return i;
        }
    return -1; //未发现
};

template <class TElem>
long TLinearListSqu< TElem>::LocateNext(TElem &elem)
{
    long i;

    for (i=lastVisited+1; i<len; i++)
        if (room[i]==elem)
```

```

{
    lastVisited=i;
    return i;
}
return -1; //未发现
};

```

(六) 插入操作

Insert(TElem &elem, long sn): 该函数用于在表中第 sn 个元素的前面插入指定的元素 elem，这里的 sn 的含义同 Locate 函数。

由于 sn 可能是负值（表示倒数第 sn 个，设最后一个的 sn 为-1），所以，为了能统一处理，程序在开始时，先将负的 sn 通过表长度 len 转化为正值序号（注意，是序号，即下标，而不是第几）：

```

if (sn<0) k=len+sn;
else k=sn-1;

```

此后，程序将序号为 k~(len-1)的元素，从最后一个开始，依次后移一位。移动后，将待插入的元素 elem 存入位置 k.:

```

for (i=len-1; i>=k; i--) room[i+1] = room[i];
room[k]=elem;

```

由于插入新元素时，可能导致存储溢出，所以在移动元素前，先检查当前存储空间（由 size 反映）是否够用，若不够用，则 ResizeRoom 重新申请一块较大空间，并保留原数据。这里新空间大小定为 size+10，增量 10 是个示意性的值，它的大小影响效率。太小了，会导致频繁地调用 ResizeRoom，时间消耗大。但太大了，会造成空间浪费。该值的确定，最好是根据具体的访问历史的统计数据动态确定。

```

template <class TElem>
TElem *TLinearListSqu< TElem>::Insert(TElem &elem, long sn)
{
    long i, k;
    int ret=0;

    if (sn<0) k=len+sn;
    else k=sn-1;

    if (k>len) k=len; //sn 太大时，在最后追加
    else if (k<0) k=0; //sn 太小时，在最前面插入

```

```

if (len>=size) //剩余空间不足时，调用成员函数重新分配空间
{
    ret=ResizeRoom(size+10); //将空间扩大为 size+10，并保留原内容
    if (ret<0) throw TExcepLinearList(3); //分配不到空间时，抛掷异常
}

for (i=len-1; i>=k; i--)
    room[i+1] = room[i];
room[k]=elem;
len++;

return &room[k];
}

```

我们分析一下该插入操作的时间复杂度。如果不考虑由于重新分配空间所消耗的时间，该算法中耗时最大的是执行元素移动的语句：

for (i=len-1; i>=k; i--) room[i+1] = room[i];

可以以它为整个算法的时间复杂度。它的循环次数（即移动元素的次数）是(len - k)。但在这里，若相对程序员，插入点 k 是个随机量。对这类包含随机量的式子，要用概率论方法处理，即移动元素的次数应为概率平均（数学期望）：

$$f(n) = \sum_{i=0}^n p_i c_i$$

p_i ----在 i 号位置上插入的概率；

c_i ----算法在 i 号位置上插入时所需移动元素的次数；

n ----元素的个数。考虑可以在最后插入（Append），故 i 可到 n.

显然， p_i 是个与算法无关的量，只与算法的调用情况有关。 c_i 与算法有关，在这里有

$$c_i = (n - i) \quad // \text{注意，} i \text{ 为下标，从 } 0 \text{ 起}$$

现假定插入位置是等概率的，则 $p_i = 1/(n+1)$ ，则

$$\begin{aligned}
 f(n) &= \sum_{i=0}^n p_i c_i = \sum_{i=0}^n \frac{c_i}{n+1} = \\
 &= \frac{1}{n+1} \sum_{i=0}^n (n-i) = \frac{1}{n+1} \frac{n(n+1)}{2} \\
 &= \frac{n}{2} = O(n)
 \end{aligned}$$

因此，此算法的时间复杂度是线性的。至于空间复杂度，由于此算法未用到与元素个数相关的集合数据（如数组），所以为常量。

(七) 删除操作

1. Delete(long sn): 该函数删除表中第 sn 个元素(sn 的含义同 Locate)，并将其值的地址返回。sn 的处理方法同 Insert。

删除与插入相反，需从删除点(k)开始，依次将其后面的各元素分别前移一位：

```
for (i=k; i<len; i++) room[i] = room[i+1];
```

由于被删除的元素要返回（以继续使用），所以，在移动元素前先将其保存起来：

```
buffElem=room[k];
```

这里，buffElem 是类中的一个私有变量，它的生命期与所在对象相同。但它是共用的，在执行下个 Delete 后，上次的结果就被置换了，因此，调用者应及时将其值复制。

元素被删除后，就让出了存储空间。为了节省存储空间，这里，每删除一次元素，就及时检查是否已有足够多的空闲空间，若已空出足够多空间（这里假定空余空间已达 50% 以上时），就调用 ResizeRoom 重新分配空间（为表长的 3/2）。与 Insert 中相似，这几个数目的确定，也最好根据统计数据来进行。

```
template <class TElem>
TElem *TLinearListSqu< TElem>::Delete(long sn)
{
    long i, k;

    if (sn<0) k=len+sn;
    else k=sn-1;

    if (k<0 || k>=len)
        throw TExcepLinearList(2);

    buffElem=room[k];
    for (i=k; i<len; i++)
        room[i] = room[i+1];
    len--;

    if (2*len < size) //如果空余空间已达到一定程度，就缩小之
        ResizeRoom(len+long(len/2.0));

    return &buffElem;
}
```

2. Delete(TIndexSelector &sel, TElem *elemDeleted): 该函数用于删除下标选择器 sel 所指出的各元素，并将所删除的元素的值存入一维数组 elemDeleted.

为了方便按 sel 访问元素，首先通过 sel 的成员函数 GetEachIndex(indexDeleted) 获得 sel 的展开式 indexDeleted，indexDeleted 是一维数组，存放了 sel 所指出的各个下标的列举（已将区间展开）。indexDeleted 的空间的申请在 sel 对象外进行，这里先调用 sel 的 GetIndexNum() 获取 sel 中下标列举的总数，然后根据此数值为 indexDeleted 分配空间。关于类 TIndexSelector，我们将在后面介绍。

实际的删除工作是通过 DeleteByIndex(indexDeleted, i, elemDeleted) 完成的。

```
template <class TElem>
long TLinearListSqu< TElem>::Delete(TIndexSelector &sel, TElem *elemDeleted)
{
    long i;
    long *indexDeleted;

    i = sel.GetIndexNum(); //获得 sel 中下标的总数
    indexDeleted = new(nothrow) long[i];
    if (indexDeleted==NULL) throw TExcepLinearList(4);

    i=sel.GetEachIndex(indexDeleted); //获得 sel 中各个下标值，存放在数组 indexDeleted
    i=DeleteByIndex(indexDeleted, i, elemDeleted); //调用 DeleteByIndex 完成实际的删除工作

    delete indexDeleted; //释放临时空间
    return i;
};
```

3. DeleteByIndex(long *idxTobeDel, long numIdx, TElem *elemDeleted): 该函数执行具体的删除操作。由于是根据下标列举 idxTobeDel 进行删除，所以处理方法有所不同。

程序中，依次检查表中各元素，当发现当前检查到的元素是待删元素时，就将其传输到 elemDeleted，并计数（到 k 中）；当发现当前检查到的元素不是待删元素时，就前移其，位移量为计数值 k。该原理如图 3-1 所示，图中假定 x 为元素，星号*表示删除标记。

```
template <class TElem>
long TLinearListSqu< TElem>::
DeleteByIndex(long *idxTobeDel, long numIdx, TElem *elemDeleted)
{
    long i, k;
```



```

k=0;
for (i=0; i<len; i++)
{
    if (i==idxTobeDel[k])
    {
        if (elemDeleted!=NULL) // elemDeleted 为空时表示不保留所删除的元素
            elemDeleted[k] = room[i];
        k++;
    }
    else room[i-k] =room[i];
}
len = len - k;
if (2*len < size) // 剩余空间足够大时, 调用 ResizeRoom 释放多余的空间
    ResizeRoom(len+long(len/2.0));

return k;
}

```

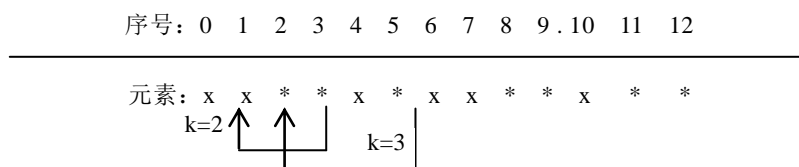


图 3-2 删除带标记的元素

这几个删除算法的时间复杂度也都是 $O(n)$, 具体分析方法类似于插入算法, 留做练习。但对空间复杂度, `Delete(TIndexSelector &sel, TElem *elemDeleted)` 中用到了一个辅助数组 `indexDeleted` 存放下标列举, 其最大空间量与元素个数相同, 即为 $O(n)$ 。

对其他成员函数, 这里就不给出源代码了, 完整的代码在附录中给出。

§ 3.3 异常处理与下标选择器*

这里特别介绍函数库中异常处理方法, 另外还介绍线性表中常用的下标选择的处理。这两种处理的实现, 本身也是个线性表问题。

§ 3.3.1 异常处理

(一) 异常处理的概念

在程序设计语言中，**异常(Exception)**是指程序运行中，由于运行环境或数据输入或操作不当，所出现的使程序不能物理地运行（而不论运行结果是否正确）的错误。这里，物理运行是指程序在实际环境下运行。这种错误与纯粹的算法逻辑错误不同（带有逻辑错误的程序可正常运行，但结果可能不正确），它的出现导致程序运行中途停止，不能继续往下执行。异常的另一重要特征是，不能通过静态程序发现异常（只能估计它的可能性），而只能通过程序的运行发现，也就是说，从程序员角度讲，异常可以预料，但不能避免。

例如，用户调用函数时，参数设置不正确（越界）；申请/使用内/外存时，存储空间不能满足要求；当使用了不存在的预定义/已定义对象，如读软盘时，软盘未插入或盘片损坏或驱动器故障，使用了故障/未连接的打印机/MODEM，网络通信故障，远程服务器故障等。

一个良好的程序（库），必须提供处理异常的机制。事实上，异常处理也应该是“正确”程序的一个必要条件。否则，只在“理想”状态下运行的程序，是实用价值不大的程序。

在数据结构中，由于我们所构造的是很底层的模块，所以异常处理更为重要。

最基本的异常处理方法是，在可能产生异常的地方，通过 if 语句判断，若发生异常，则子程序返回，并置相应的异常发生标志（一般是通过函数返回值）。在调用函数时，当函数返回时先检查异常标志，然后进行相应的动作。这种处理方式，首先是比较复杂，会使程序结构复杂化，另一重要问题是，对将函数直接作为左值使用的情况（在面向对象中，若函数返回值是对象，则这种情况很常见），显得更不自然。

在 C++ 中，为程序员提供了良好的处理异常的机制，其中心点是提供下列语句：

try----检测/捕获异常；

catch----处理 try 所捕获的异常；

throw----生成一个异常，交由 try 捕获；

我们这里直接采用 C++ 的异常机制。

(二) C++ 异常处理模式

C++ 的异常处理分两个方面，**第一是抛郑异常，第二是捕获异常**。**抛郑异常**是在可能产生异常的子程序中，用 if 检测异常，如果异常产生的条件成立了，则用 throw 语句抛郑该异常。注意，throw 的作用与 return 类似，也是子程序返回，但 throw 的“返回值”只能用 catch 捕获。下面的子程序片断在 p 为 NULL 的情况下抛郑一个异常：

```
void FuncExcep(...)  
{ ...
```

```

    if (p==NULL) throw TExcepComm(2);
    //此条件成立时, FuncExcep ()结束, 并将对象 TExcepComm(2)返回
    ...
}

```

捕获异常是在调用可能产生异常的子程序的地方, 使用 **try** 和 **catch** 来检测和处理异常。例如, 设有另外一个程序调用 **FuncExcep()**, 则可在它中用 **try** 和 **catch** 捕获并处理异常 **FuncExcep()**:

```

...
try //将有可能抛郑异常的语句放到 try 中, 以监控其异常是否被抛郑
{
    ...
    FuncExcep();
    ...
}
catch (TExcepComm e)
    //这里的 catch 相当于 if, 当它前面的 try 捕获到的 (即 FuncExcep 抛郑的) 是 TExcepComm 型
    //异常时, 程序的执行转到该 catch 中, 并将 FuncExcep 所抛郑的值以参数的形式带入。
{
    cout<<"\n"<<e.errMessage; //显示 e 中的信息。e 是由 FuncExcep 所抛郑的对象
};

```

(三) 出错信息管理

从上面介绍的异常处理机制看出, 为出错信息编码并设立专门的代码管理机构是有好处的。

首先, 设立一个表, 存储异常代码和用于说明异常的文字信息。为方便, 我们也将其定义为类, 如下所示。

```

struct TErrMessageRec //异常记录
{
    int no; //异常代码
    char msg[CNST_SizeErrMsg]; //异常信息
};

class TErrMessageList //异常表类
{
    TErrMessageRec msgList[CNST_MaxNumErrMsg]; //用一维数组做异常表
public:

```

```

int len; //指示异常表的长度（元素个数）

TErrMessageList() //构造函数，这里主要是装入异常表
{
    int i=0;
//下面示例性地为异常表装入数据
/* 0*/   msgList[i].no=i;   strcpy(msgList[i].msg, "Unknown error"); i++;
/* 1*/   msgList[i].no=i;   strcpy(msgList[i].msg, "Parameters Out of range"); i++;
/* 2*/   msgList[i].no=i;   strcpy(msgList[i].msg, "Parameters illegal"); i++;
/* 3*/   msgList[i].no=i;   strcpy(msgList[i].msg, "Mmemory space low"); i++;
/* 4*/   msgList[i].no=i;   strcpy(msgList[i].msg, "Not found"); i++;

    len = i;
};

char *GetMessage(int no) //成员函数，根据编号读出异常信息
{
    if (no<0 || no>=CNST_MaxNumErrMsg)
        return NULL;
    return msgList[no].msg;
};
};

```

我们这里只是示意性地设置了一些出错代码。在实际中，可设置更详细更合理的信息。当该类的实例产生时，这些信息也被载入。

为了后面的各异常处理类的使用，需定义一个全局的出错代码表 `errMessageList`：

```
extern TErrMessageList errMessageList;
```

在该类的基础上，可以定义各种针对具体数据结构的异常处理类了，如下一节要定义的 `TExcepLinearList`。这里，我们先定义一个通用的异常处理类 `TExcepComm`。

```

class TExcepComm
{
public:
    int errNo;
    char errMsg[CNST_SizeErrMsg];

    TExcepComm(int mErrNo)
    {
        errNo=mErrNo;
    }
};

```

```

        strcpy(errMessage, errMessageList.GetMessage(errNo));
    }
};

```

该类的实例主要用于存放一条出错信息（代码及其文字说明）。当用一个出错代码做为参数定义（生成）一个对象时，代码对应的出错信息就存入对象的 `errMessage` 中。该给类一般用在 `throw` 语句，通过它传递异常的编码与说明信息。

§ 3.3.2 下标选择器

对可按下标（序号）访问元素的结构，常常需要指定若干下标范围，为此，我们引入下标选择器。

`TIndexSelector` 是一个字符串形式的下标选择器。其内容是用逗号分隔的若干项（称为**选择项**），每个选择项可以是一个整数，用以指出一个下标，或是用下划线“`_`”分隔的两个整数，用以指出一个下标范围，“`_`”前的数缺省时表示起始序号，“`_`”后的数缺省时表示结束序号。多个项则指出多个范围。正整数表示序号是从前（左）往后（右）数（递增），负整数表示序号是从后（右）往前（左）数（递减）。下标选择器的语法格式严格定义如下：

```

IndexSelector ::= IndexScope | (IndexScope  "," IndexSelector)
IndexScope  ::= INTEGER | ( INTEGER? "_" INTEGER?)
INTEGER     ::= ("-" | "+")? DIGITS
DIGITS      ::= DIGIT | ( DIGIT DIGITS)
DIGIT       ::= 0|1|2|3|4|5|6|7|8|9

```

例如，下面的串都是形式合法的下标选择器

```

5
1_
_8
-
1,2
1,2_3, -1_-3
3_4,2,7_8, 10_

```

下面给出 `TIndexSelector` 类的定义，其中一些成员函数未给出具体实现，其实现留做作业。

```

struct TSelectorItem //选择项类型
{
    long lower; //项的下界
    long upper; //项的上界

```

```
};

class TIndexSelector
{
    TSelectorItem room[CNST_SizeIndexSelector];
    //存储选择项的一维数组，每个项用一个数组元素存储
    int lastVisitedItem;

    int LoadFromArray(TSelectorItem *se, int n, long maxIdx);
    long Standardize(void);

public:
    int len;
    long maxIndex;

    TIndexSelector() {len=0;};
    TIndexSelector(TSelectorItem *se, int n, long maxIdx);

    int GetFirst(long &lw, long &up);
    int GetNext(long &lw, long &up);
    long GetEachIndex(long *idxs);
    int Get(int idx, long &lw, long &up);
    int Set(int idx, long lw, long up);
    long GetIndexNum(void);

    int ResetByArray(TSelectorItem *se, int n, long maxIdx);

    long GetLower(int idx);
    long GetUpper(int idx);

    int SetLower(int idx, int lw);
    int SetUpper(int idx, int up);

    int Insert(int idx, long lw, long up);
    int InsertLower(int idx, long lw);
    int InsertUpper(int idx, long up);

    int Delete(int idx);
```

```
};
```

结构 `TSelectorItem` 用于存储选择项的上下界。类 `TIndexSelector` 的实例代表一个下标选择器，它中主要部分说明如下：

room: 一维数组，每个元素是 `TSelectItem` 类型，存储下标选择器中一个选择项。

len: 整数，为属性，表示下标选择器中项目的数目。

TIndexSelector(char *selStr): 构造函数，字符串参数 `selStr` 用来指出对象初始值，如，

```
TIndexSelector sel("-1_-5, 2, 3, 7_10")
```

定义了下标：倒数第 1 到倒数第 5、第 2、第 3、第 7 到第 10；

TIndexSelector(TSelectorItem *se, int n, long maxIdx): 构造函数，参数 `se` 为一维数组，每个元素是一个完整的选择项；参数 `n` 为 `se` 中元素个数；参数 `maxIdx` 用来指出对应的下标体系中最大下标值，其主要用于将负值下标转换为正值。该函数的功能是用存储在 `TSelector` 型数组中的值，初始化下标选择器。使用方法举例说明如下：

```
TSelectorItem si[] = { {1, 3}, {-1, -5}, {6, 6} };
```

```
TIndexSelector sel(si, 3, 20);
```

LoadFromArray(TSelectorItem *se, int n, long maxIdx): 具体负责将选择项数组装入，并将项中负值转化为正值。上面的构造函数通过调用它和 `Standardize` 实现初始化。参数同上面的构造函数。

Standardize(void): 用于将选择器表 `room` 规格化。所谓规格化，是将各负值项转化为正值（为提高效率，该工作已放到了 `LoadFromArray` 中执行），将重叠的和相连的项合并，然后按各项的下界值排序，并剔除已被归并了的项。

ResetByArray(TSelectorItem *se, int n, long maxIdx): 为选择器重新置值，类似于构造函数（但构造函数不能在对象存在后被调用）。

Get(int idx, long &lw, long &up): 读取序号为 `idx` 的选择项，将其下上界分别存放到 `lw` 和 `up` 中。成功时返回 `idx` 值，不成功时返回 -1。

Set(int idx, long lw, long up): 将序号为 `idx` 的项置为 `(lw, up)`。成功时返回 `idx`，否则返回 -1（表示越界）。

GetFirst(long &lw, long &up): 读取第一个选择项，将其下上界分别存放到 `lw` 和 `up` 中。成功时返回 0，不成功时，返回 -1。

GetNext(long &lw, long &up): 读取下一个选择项，将其下上界分别存放到 `lw` 和 `up` 中。成功时返回 0，不成功时返回 -1。这里的“下一个”，是针对上次使用 `GetNext` 或 `GetFirst` 所访问的项。其一般使用模式为：

```
TSelectorItem si[] = { {1, 3}, {-1, -5}, {6, 6} };
```

```
TIndexSelector sel(si, 3, 20);
```

```
... ...
```

```
long lw, up;
```

```
int k;  
k=sel.GetFirst(lw, up);  
while (k>0)  
{  
    //使用 lw 和 up 访问数据;  
    k=sel.GetNext(lw, up);  
}
```

GetEachIndex(long *idxs): 读取各个下标值, 将它们都展开存放在一维数组 idxs 中。返回展开后下标总数目。所谓展开, 是指将各区间内值都列举出来。例如,

[2,4], 7, [9,10]

展开后变为:

(2, 3, 4, 7, 9, 10)

GetIndexNum(void): 返回 (展开后) 下标总数。

GetLower(int idx): 以整数形式返回序号为 idx 的项的起点部分。

GetUpper(int idx): 以整数形式返回序号为 idx 的项的终点部分。

SetLower(int idx, int lw): 将序号为 idx 的项的起点置为 lw。成功时返回 0, 否则返回非 0 (表示越界)。

SetUpper(int idx, int up): 将序号为 idx 的项的终点置为 up。成功时返回 0, 否则返回非 0 (表示越界)。

Insert(int idx, long lw, long up) : 将项(lw, up)插入到序号为 idx 的项的前面。成功时返回下标个数, 否则触发异常。

InsertLower(int idx, long lw) : 将项(lw) (表示单项) 插入到序号为 idx 的项的前面。成功时返回下标个数, 否则触发异常。

InsertUpper(int idx, long up): 将项(1, up)插入到序号为 idx 的项的前面。成功时返回下标个数, 否则触发异常。

Delete(int idx): 删除序号为 idx 的项。成功时返回下标个数, 否则触发异常。

事实上, 这里的 TIndexSelector 本身也是一种具体的线性表。等到学完本章后, 再温习这个类的设计, 会加深理解。

§ 3.4 线性表的链式存储----线性链表

上节介绍过的连续存贮结构, 数据元素间的逻辑关系是用它们的存贮次序表达的, 逻辑上相邻的两个元素, 它们的存贮位置也相邻, 所以, 任一元素的地址, 可通过公式获得, 其优点是存贮利用率高, 访问元素的速度快。但也有几方面的缺点, 第一, 当进

行插入与删除时，需移动元素。如果线性表很大，移动量就非常大。第二，由于它要求一片连续的存贮区域，所以存贮要求较高，不能利用小块存贮区。如果采用纯粹的静态存储（如表空间为数组），则其还不适合表容量的动态扩充。如果对线性表经常要进行插入或删除，并需动态扩充容量，则需采用链式存贮结构。

§ 3.4.1 链式存贮方法

线性表采用链式存贮结构时称为**线性链表**，它的具体存储方法也可能有多种，我们这里先介绍以后继或前驱地址为链的存储方法，这样的链表也称**单链表**。具体的存贮映射方法是：

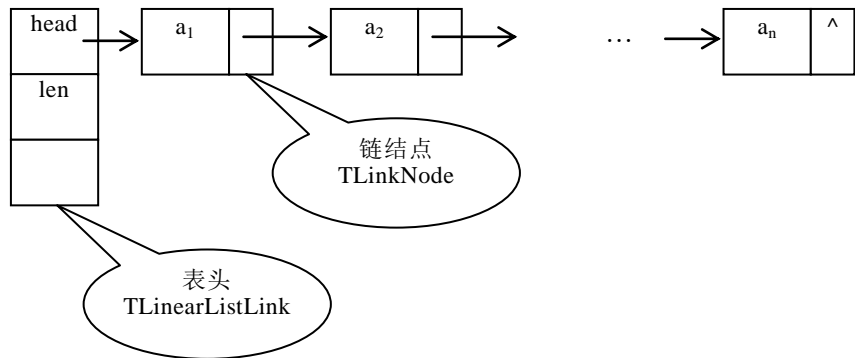


图 3-3 单链表基本形式

● 对线性表中每个元素 a_i ，为它分配一块存贮区。存贮区的分配问题，本不属存贮方法的讨论范围，但具体实现时，此问题不可回避。有两种分配方式，一是设计者对一片足够大的存贮区自行管理（分配与释放），这种方式称为**静态**方法。相应的链式结构称为**静态链表**。另一种方式是利用高级语言的动态存贮管理机制（如 PASCAL 中的 new、dispose、C/C++ 中的 malloc、free、new、delete 等）。在这种方式中，对存贮空间的使用不需涉及存贮管理的实现问题。在本教程中，以这种方式为主。

● 每个元素的存贮区分为两大部分：

内容	前驱/后继地址
----	---------

其中，内容部分用于存放元素本身的信息；“前驱/后继地址”部分存放该元素的前驱或后继的存贮地址。这里我们一般使用后继地址。

对表中最后一个结点，令其“后继地址”为空，作为链表的结束标志。

● 为了能方便地访问链表，设置链表头结点，记下链表中首结点的地址（有时也要记尾结点地址）和链表中当前结点个数等有关链表的信息。这种头结点作为链表的描述结点，是对链表的整体描述，是整个链表的代表，故它的类型可做为线性表的类型。

一个典型的单链表的形式如图 3-1 所示。

§ 3.4.2 线性链表的面向对象描述

我们这里给出上面介绍的存储方法的面向对象描述。

(一) 元素与关系描述

下面是链表中结点的 C++ 描述，它代表着线性表的元素和关系。可以在这个类中设置关于 info 和 next 的 Get 和 Set 操作，以隐蔽该结构。但为了简便，这里没有设置这两类操作。结点结构的隐蔽，交由链表对象完成。注意，由于线性表元素类型可以是任一类型（可变的），所以所用 C++ 的模板机制实现这种可变类型。

```
template <class TElem>
//上面是模板声明，表明 TElem 是一个可变（调）类型，在使用 TLinkNode 时动态决定
struct TLinkNode
{
    TElem info; //info 的类型是可变的类模板 TElem
    TLinkNode *next; //注：这里的 TLinkNode 后可以省略<TElem>
};
```

(二) 链表对象描述

链表对象应该是前面介绍的线性表抽象类 TLinearList0 的派生类，代表着整个线性链表。它需要记录首结点的地址和链表中当前结点个数等有关链表的信息，并针对其设置有关操作。

```
template <class TElem>
class TLinearListLink : public TLinearList0<TElem>
{
protected:
    TLinkNode<TElem> *head;

    TLinkNode<TElem> *lastVisited;
    long lastVisitedIndex;
    TElem buffElem;

    void ReleaseAll();

public:

    TLinearListLink(void);
    ~TLinearListLink(void);
```

```

virtual TElem &Get(long idx);
virtual TElem *GetAddress(long idx);
virtual TElem *Set(long idx, TElem &elem);

virtual long CountElem(TElem &elem);

virtual TElem *Prior(long idx);
virtual TElem *Next(long idx);

virtual long Locate(TElem &elem, long sn=1);
virtual long Locate(TElem &elem, long *foundElemIndex=NULL);
virtual long LocateFirst(TElem &elem);
virtual long LocateNext(TElem &elem);

virtual TElem *Insert(TElem &elem, long sn=1);

virtual TElem *Delete(long sn=1);
virtual long Delete(TIndexSelector &sel, TElem *elemDeleted=NULL);
virtual long DeleteByIndex(long *idxTobeDel, long numIdx,
TElem *elemDeleted=NULL);

void Print(); //Only for test

//Newly added functions
virtual TElem &Get(TLinkNode<TElem> *pNode);
virtual TLinkNode<TElem> *GetNode(long idx);
virtual long GetNodeIndex(TLinkNode<TElem> *pNode);

virtual TLinkNode<TElem> *SetNode(TLinkNode<TElem> *pNode, TElem &elem,
                                TLinkNode<TElem> *pNext);
virtual TLinkNode<TElem> *SetNodeElem(TLinkNode<TElem> *pNode,
                                TElem &elem);
virtual TLinkNode<TElem> *SetNodeNext(TLinkNode<TElem> *pNode,
                                TLinkNode<TElem> *pNext);

virtual TLinkNode<TElem> *PriorNode(TLinkNode<TElem> *pNode);
virtual TLinkNode<TElem> *NextNode(TLinkNode<TElem> *pNode);

```

```

virtual TLinkNode<TElem>    *PriorNode(long idx);
virtual TLinkNode<TElem>    *NextNode(long idx);

virtual TLinkNode<TElem>    *LocateNode(TElem &elem, long sn=1);
virtual long                LocateNode(TElem &elem,
                                       TLinkNode<TElem> *foundElemPointer[]);

virtual TLinkNode<TElem> *LocateNodeFirst(TElem &elem);
virtual TLinkNode<TElem> *LocateNodeNext(TElem &elem);

virtual TLinkNode<TElem> *InsertAfter(TLinkNode<TElem> *pNodeTobeInserted,
                                       TLinkNode<TElem> *pNodeBase);
virtual TLinkNode<TElem> *Insert(TLinkNode<TElem> *pNodeTobeInserted,
                                  TLinkNode<TElem> *pNodeBase);
virtual TLinkNode<TElem> *DeleteAfter(TLinkNode<TElem> *pNode);
virtual TLinkNode<TElem> *Delete(TLinkNode<TElem> *pNode);
virtual TLinkNode<TElem> *Insert(TLinkNode<TElem> *pNode, long sn=1);

};

```

该类仍然是从线性表抽象类 `TLinearList0` 继承而来。除了包含了 `TLinearList0` 的全部成员外，还增加了一套针对链指针的操作。这些操作实质上是很直接的、很基本的，是实现抽象类 `TLinearList0` 所提出的操作的重要基础，同时，由于链表本身的特殊性，也直接需要将这些有关链的操作作为基本操作。

它的几个重要的数据成员为：

head: 头指针，指向链表中第一个链结点，通过它，可顺藤摸瓜，访问到表中所有结点。当然，如果为了提高访问尾结点的速度，可以设立尾指针 `rear`。

len: 表长度，即链表中结点的数目。

至于成员函数，在下面介绍它们的实现时一并介绍。

§ 3.4.3 线性链表的面向对象实现

下面给出 `TLinearListLink` 中成员函数的实现。

(一) 初始化

初始化工作由构造函数实现。构造函数的任务是生成一个可加入结点的空链表，所以，它的主要工作只是将 `head` 和 `len` 置为空（或 0）。与顺序存储不同，这里的初始化程序不负责分配存储空间。结点的存储空间的分配是在插入元素时进行。

析构函数负责回收结点空间，它通过调用 **ReleaseAll** 实现结点回收。下面是具体的源代码。

```
template <class TElem>
TLinearListLink< TElem>::TLinearListLink()
{
    head=NULL;
    len=0;

    lastVisitedIndex=0;
    lastVisited=NULL;
};

template <class TElem>
TLinearListLink< TElem>::~~TLinearListLink()
{
    if (head!=NULL) ReleaseAll();
};

template <class TElem>
void TLinearListLink< TElem>::ReleaseAll()
{//释放链表中所有元素
    TLinkNode<TElem> *p, *q;

    p=head; //p 指向当前要处理的结点
    while (p!=NULL) //从头到尾依次释放各结点
    {
        q=p; //移动 p 之前，令 q 指向 p 所指结点，以防 p 后移后丢失 p 原指的结点
        p=p->next; //p 后移一步，使得下次循环时，p 指向下一个要处理的结点
        delete q; //释放 q 所指结点
    }
};
```

(二) 指针类 **Get** 类操作

设置这组 **Get** 和下小节的 **Set**，一起起到隐蔽底层数据结构（链结点结构）的作用。

1. Get(TLinkNode<TElem> *pNode): 该函数返回指定结点中的元素内容。通过该函数，可屏蔽结点的 **info** 域的读取，再结合 **SetNode** 可完全屏蔽 **info**。

```
template <class TElem>
TElem &TLinearListLink< TElem>::Get(TLinkNode<TElem> *pNode)
{
    if ( pNode==NULL) TExcepLinearList(4);
    return pNode->info;
}
```

2. GetNodeIndex(TLinkNode<TElem> *pNode): 返回指定结点在表中的序号。有了该函数，就可将链表特有的结点访问，转化为一般的序号。再结合 GetNode(long idx)，可实现序号和结点指针互换。

该函数的实现，是在链表中从头查找给定结点 pNode，同时对“摸”过的结点进行计数。该计数值即为对应结点的序号。显然，其最小查找时间是 1，即表中第一个结点即为所求，最大时间是 n，即所求结点在最后，或不存在。其平均查找时间是 $O(n)$

```
template <class TElem>
long TLinearListLink< TElem>::GetNodeIndex(TLinkNode<TElem> *pNode)
{
    if ( pNode==NULL) TExcepLinearList(4);

    TLinkNode<TElem> *p;
    long i;

    i=0;
    p=head;
    while (p!=NULL) //从头到尾扫描各结点，一旦发现地址为 pNode 的结点，即跳出循环
    {
        if (p==pNode) break;
        i++;
        p=p->next;
    }
    if ( p==NULL) TExcepLinearList(4); //未找到
    return i; //已找到，返回其序号
}
```

3. GetNode(long idx): 该函数通过序号求指针，它返回指定序号 idx 所对应的结点的指针。

在一般情况下，该函数的实现与 GetNodeIndex 类似，也是从头起“摸”结点，并计数。不同的是，“摸”到的标志是计数到 idx。

为了提高速度，我们在类中设置了两个私有变量：

lastVisited----记录最近被访问到的结点的指针；

lastVisitedIndex----记录最近被访问到的结点的序号;

程序开始“摸”前,先检查给定的 idx 是否大于或等于 lastVisitedIndex,若是,则不必从头“摸”,只需从 lastVisited 起向后“摸”;否则才从头摸。

为了保证正确性,必须使 lastVisited 和 lastVisitedIndex 保持一致。同时,为了使 lastVisited 真实表示最近访问,应及时调整 lastVisited 的值,使它指向最近访问到的结点。

```
template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::GetNode(long idx)
{
    TLinkNode<TElem> *p;
    long i;

    if (idx<0 || idx>=len)
        throw TExcepLinearList(2);

    if (lastVisited!=NULL && idx >= lastVisitedIndex)
    { p=lastVisited;
      i=lastVisitedIndex;
    }
    else
    {
        p=head;
        i=0;
    }
    while (p!=NULL)
    {
        if (i==idx) break;
        i++;
        p=p->next;
    }
    if ( p==NULL) throw TExcepLinearList(0);
    lastVisitedIndex=i; //使 lastVisitedIndex 记下本次最后访问到的结点的序号
    lastVisited=p; //使 lastVisited 记下本次最后访问到的结点的地址
    return p;
}
```

显然,若用连续的 idx 值调用 GetNode(idx),则程序中 while 循环只执行一次,显著地提高了速度。例如,对下面的程序片段:

```

TLinearListLink<long> a; //定义一个单链表 a,其元素类型为 long
long i;
i=0;
while (i<a.len)//输出各结点的值
{
    cout<<*a.GetNode(i)<<" "; //将地址为 a.GetNode(i)返回值的结点的内容输出
    i++;
}

```

实际的访问结点次数是 len ，但若没有 `lastVisited` 配合，平均访问次数是 $len(len+1)/2$ ，为平方数量级。

(三) 指针类 Set 类操作

1. SetNode(TLinkNode<TElem> *pNode, TElem &elem, TLinkNode<TElem> *pNodeNext): 该函数用于对给定结点(pNode)的 info 和 next 字段同时置值，并返回被置值的结点指针。

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>
    ::SetNode(TLinkNode<TElem> *pNode, TElem &elem,
              TLinkNode<TElem> *pNodeNext)
{
    if (pNode==NULL) throw TExcepLinearList(4);
    pNode->info = elem;
    pNode->next = pNodeNext;
    return pNode;
}

```

2. SetNodeElem(TLinkNode<TElem> *pNode, TElem &elem): 该函数用于只给指定的结点(pNode)置内容(elem)

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>
    ::SetNodeElem(TLinkNode<TElem> *pNode, TElem &elem)
{
    if (pNode==NULL) throw TExcepLinearList(4);
    pNode->info = elem;
}

```



```

    return pNode;
}

```

3. SetNodeNext(TLinkNode<TElem> *pNode, TLinkNode<TElem> *pNodeNext):

该函数用于给指定的结点(pNode)置后继(pNodeNext)

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>
    ::SetNodeNext(TLinkNode<TElem> *pNode, TLinkNode<TElem> *pNodeNext)
{
    if (pNode==NULL) throw TExcepLinearList(4);

    pNode->next = pNodeNext;
    return pNode;
}

```

(四) 链指针类前驱/后继操作

1. NextNode(TLinkNode<TElem> *pNode): 返回给定结点(pNode)的后继结点的指针。

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::
NextNode( TLinkNode<TElem> *pNode)
{
    if ( pNode->next==NULL) TExcepLinearList(4);

    lastVisited=pNode;

    return pNode->next;
};

```

2. PriorNode(TLinkNode<TElem> *pNode): 返回给定结点(pNode)的前驱结点的指针。由于这里的链表是后继链，所以求已知结点的前驱并不象求后继那样直接。一般情况下是从头“摸”。与前面介绍的 GetNode(long idx)类似，这里也通过 lastVisited 优化访问。但这里并不象 GetNode(long idx)的效果那么好，因为若下次再次使用 PriorNode 时，lastVisited 就没效果了。

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::
PriorNode(TLinkNode<TElem> *pNode)
{

```

```

TLinkNode<TElem> *p;
long i;

if (lastVisited!=NULL)
    if (lastVisited->next==pNode)
        return lastVisited;

i=0;
p=head;
if (pNode==head)
    throw TExcepLinearList(4);

while (p!=NULL)
{
    if (p->next==pNode) break;
    i++;
    p=p->next;
}

if (p==NULL) throw TExcepLinearList(4);
lastVisited=p;
lastVisitedIndex=i;
return p;
};

```

3. PriorNode(long idx): 根据序号求对应结点的前驱结点的指针。直接调用 GetNode(idx-1)即可。

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::PriorNode(long idx)
{
    return GetNode(idx-1); //Throw the same exception as GetNode if idx is illegal;
}

```

4. NextNode(long idx): 根据序号求对应结点的后继结点的指针。直接调用 GetNode(idx+1)即可。

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::NextNode(long idx)
{
    return GetNode(idx+1); //如果 idx 非法，则抛郑与 GetNode 相同的异常;
}

```

(五) 指针类定位操作


下面一组函数，通过元素值查找结点，返回所找到的结点的指针。由于知道了指针，就可访问结点的一切信息，所以这组函数是很基本的，是其他类定位函数的实现的基础。

1. LocateNode(TElem &elem, long sn): 返回元素值为 elem 的第 sn 次出现的结点的指针。这里，sn 含义与前相同。若 sn 值太大，则视其为查找最后一个匹配。

```
template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::LocateNode(TElem &elem, long sn)
{
    TLinkNode<TElem> *p, *p0;
    long k, i;

    k=sn;
    if (k<0) k=k+CountElem(elem)+1; //sn 为负时，将其换算为正
    // 成员函数 CountElem(elem)返回值为 elem 的元素的个数，用此值换算 sn
    if (k<0) throw TExcepLinearList(2);

    p0=NULL;
    p=head;
    i=0;
    while (p!=NULL)
    {
        if (p->info==elem)
        {
            i++;
            p0=p;
            if (k==i) return p;
        }
        p=p->next;
    }
    if (p0==NULL) throw TExcepLinearList(3); //Not found
    else return p0; //找到，但 sn 大于匹配的元素个数
}
```

 由于这里的 sn 相对于值等于 elem 的元素的元素，所以，将负的 sn 转化为正值时（正向数），不能简单地用表中元素总数。

2. LocateNode(TElem &elem, TLinkNode<TElem> *foundElemPointer[]): 查找值

为 elem 的所有元素，将它们的指针依次存储在 foundElemPoint[] 中，并返回所找到的元素的个数。

```
template <class TElem>
long TLinearListLink< TElem>
::LocateNode(TElem &elem, TLinkNode<TElem> *foundElemPointer[])
{
    TLinkNode<TElem> *p;
    long i;

    p=head;
    i=0;
    while (p!=NULL)
    {
        if (p->info==elem)
        {
            foundElemPointer[i]=p;
            i++;
        }
        p=p->next;
    }

    if (i<=0) throw TExcepLinearList(3); //Not found
    return i;
}
```

3. LocateNodeFirst(TElem &elem): 查找值为 elem 的第一个出现的结点，并返回其指针。该函数与下面 LocateNodeNext 配合，以能快速地依次访问各个匹配结点。

为了配合 LocateNodeNext，使用 lastVisitedIndex 和 lastVisited 记录下最近查找到的结点。

```
template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::LocateNodeFirst(TElem &elem)
{
    TLinkNode<TElem> *p;
    int k;

    p=head;
    k=0;
    while (p!=NULL)
    {
```

```
if (p->info==elem)
{
    lastVisited=p;
    lastVisitedIndex = k;
    break;
}
k++;
p=p->next;
}

if (p==NULL) throw TExcepLinearList(3); //Not found
else return p; //找到
}
```

4. LocateNodeNext(TElem &elem): 查找值为 elem 的下一个（即最近一次调用相同查找目标的 LocateNodeNext 所查到的结点，对第一次使用 LocateNodeNext，则是相对于最近使用的 LocateFirst）出现的结点，并返回其指针。

```
template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::LocateNodeNext(TElem &elem)
{
    TLinkNode<TElem> *p;
    long k;

    if (lastVisited==NULL) p=head;
    else p=lastVisited->next; //令 p 指向上次访问过的结点的下个结点

    k=1;
    while (p!=NULL) //从 p 其向后扫描
    {
        if (p->info==elem)
        {
            lastVisited=p;
            lastVisitedIndex +=k;
            break;
        }
        p=p->next;
        k++;
    }
}
```

```

if (p==NULL) throw TExcepLinearList(3); //Not found
else return p; //找到
}

```

(六) 指针类插入操作

下面给出的一组插入函数，插入点是通过结点指针描述的。这与抽象类中插入点是通过序号给出的不同。由于知道了序号，可通过 `GetNode(long idx)` 求得对应的指针，故可通过这组函数实现抽象类中的插入操作。

1 . InsertAfter(TLinkNode<TElem> *pNodeTobeInserted, TLinkNode<TElem> *pNodeBase): 将指针 `pNodeTobeInserted` 所指结点，插入到指针 `pNodeBase` 所指结点的后面。由于是后继链表，所以该操作的实现很直接。图 3-1 是插入示意。

```
template <class TElem>
```

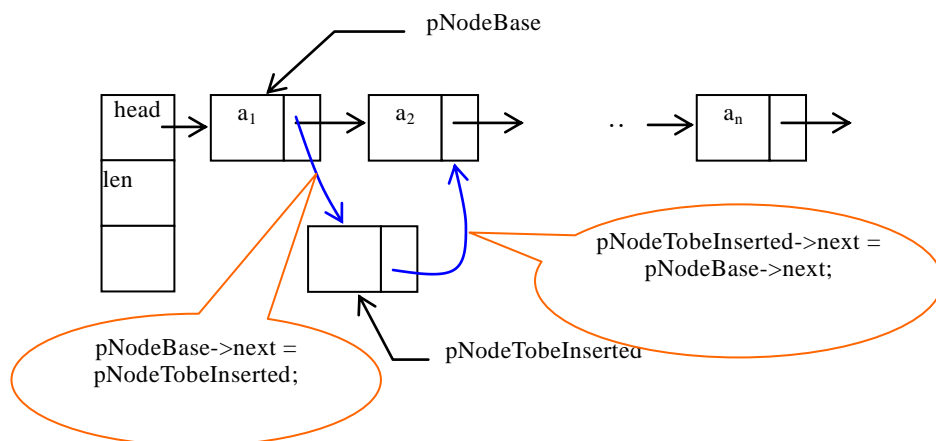


图 3-4 单链表插入结点

```

TLinkNode<TElem> *TLinearListLink< TElem>::
    InsertAfter(TLinkNode<TElem> *pNodeTobeInserted,
TLinkNode<TElem> *pNodeBase)
{
    if (pNodeTobeInserted==NULL || pNodeBase==NULL)
        throw TExcepLinearList(2);
    pNodeTobeInserted->next = pNodeBase->next;
    pNodeBase->next = pNodeTobeInserted;
    len++;
    return pNodeTobeInserted;
}

```

2 . Insert(TLinkNode<TElem> *pNodeTobeInserted, TLinkNode<TElem>

***pNodeBase):** 在 pNodeBase 所指结点之前插入 pNodeTobeInserted 所指结点。

该函数先调用 PriorNode(pNodeBase) 获取 pNodeBase 的前驱的指针 p，然后调用上面的 InsertAfter 将 pNodeTobeInserted 插入到 p 之后。

由于我们这里的链表不带头结点（若链中第一个结点不作为元素结点，则称为带头结点的链表），故在第一个结点之前插入，与在其他位置插入不同。所以程序中应区分这两种情况。

```
template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::
Insert(TLinkNode<TElem> *pNodeTobeInserted, TLinkNode<TElem> *pNodeBase)
{
    if (pNodeTobeInserted==NULL || pNodeBase==NULL)
        throw TExcepLinearList(2);

    TLinkNode<TElem> *p;

    if (pNodeBase==head) //在第一个结点的前面插入
    {
        pNodeTobeInserted->next = head;
        head = pNodeTobeInserted;
    }
    else //在其他位置插入
    {
        p=PriorNode(pNodeBase); //求出基准点 pNodeBase 的前驱结点 p
        pNodeTobeInserted->next = pNodeBase; //将 pNodeTobeInserted 插入到 p 的后面
        p->next = pNodeTobeInserted;
    }
    len++;
    return pNodeTobeInserted;
}
```

3. Insert(TLinkNode<TElem> *pNode, long sn): 此函数将 pNode 所指结点插入到第 sn 个元素之前。原则上讲，它不属于指针类操作，仍为按序号指称（指称的说法对吗??? ---**答：不太通俗，但比较准确，故可采用**）。但对链表，引入它是必要的，因为在链表的实际使用中，常用到直接插入结点（不是元素）的操作。

```
template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::
    Insert(TLinkNode<TElem> *pNode, long sn)
{
    long i, k;
```

```

if (sn<0) k=len+sn; //将 sn 换算为下标值 k
else k=sn-1;
if (k>len)k=len; //sn 太大时追加到最后，故插入点为 len

if (k<0 ) throw TExcepLinearList(2);

if (k==0) //在最前面插入
{
    pNode->next = head;
    head = pNode;
    len++;
    return pNode;
}
else
{
    TLinkNode <TElem> *p;
    p=GetNode(k-1); //求出序号为 k-1 的结点的指针 p
    return InsertAfter(pNode, p); //将 p 插入到 pNode 后面
}
}

```

(七) 指针类删除操作

与插入函数类似，下面给出的一组删除函数，删除点是通过结点指针描述的。这与抽象类中删除点是通过序号给出的不同。由于知道了序号，可通过 `GetNode(long idx)` 求得对应的指针，故可通过这组函数实现抽象类中的删除操作。

1. DeleteAfter(TLinkNode<TElem> *pNode): 从表中摘除 pNode 所指结点的后继结点，并将其做为函数值返回。其操作如图 3-1 所示。

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::
DeleteAfter(TLinkNode<TElem> *pNode)
{
    if (pNode==NULL)    throw TExcepLinearList(2);

    TLinkNode<TElem> *p;
    p = pNode->next;

```



```

pNode->next = p->next;
len--;
return p;
}

```

2. Delete(TLinkNode<TElem> *pNode): 在表中摘除 pNode 所指出的结点，并返回其。要完成此操作，先调用 PriorNode(pNode)获得 pNode 的前驱的指针 p，然后调用上面的 DeleteAfter 删除 p 的后继。同样由于无头结点，所以要先检查所删的结点 pNode 是否是链中第一个，如是，则需特殊处理。

```

template <class TElem>
TLinkNode<TElem> *TLinearListLink< TElem>::Delete(TLinkNode<TElem> *pNode)
{
    if (pNode==NULL    throw TExcepLinearList(2);

    TLinkNode<TElem> *p;

    if (pNode==head)
        head = pNode->next; //摘除第一个结点
    else
        {p=PriorNode(pNode); //求出 pNode 的前驱 p
        pNode->next = p->next; //在链表中排除 pNode
        }
    len--;
    return pNode;
}

```

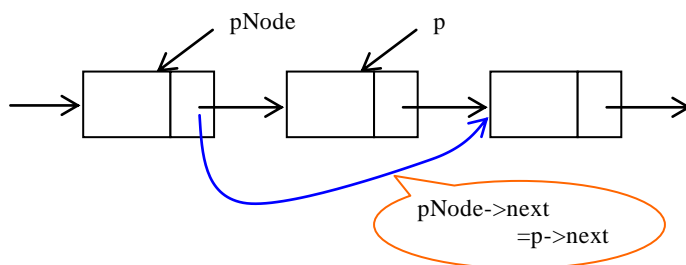


图 3-5 单链表中删除结点

(八) 一般的 Get/Set 操作

这类 Get/Set 操作以元素序号为目标操作元素，它们的含义前面都有介绍。有了前面给出的指针类操作，它们的实现也变的很简单。下面是具体的程序。

1. Get(long idx)

```
template <class TElem>
TElem &TLinearListLink< TElem>::Get(long idx)
{
    return Get(GetNode(idx)); // 如果 idx 非法，则此 Get 抛郑与 GetNode 相同的异常;
}
```

2. GetAddress(long idx)

```
template <class TElem>
TElem *TLinearListLink< TElem>::GetAddress(long idx)
{
    TLinkNode<TElem> *p;

    p= GetNode(idx); ///如果 idx 非法，则此 GetAddress 抛郑与 GetNode 相同的异常;

    return &(p->info);
}
```

3. Set(long idx, TElem &elem)

```
template <class TElem>
TElem *TLinearListLink< TElem>::Set(long idx, TElem &elem)
{
    TLinkNode<TElem> *p;

    p= GetNode(idx); //如果 idx 非法，则此 Set 抛郑与 GetNode 相同的异常;
    p->info = elem; //假定 TElem 型支持"="
    return &(p->info);
}
```

(九) 一般的前驱/后继操作

这里的前驱/后继操作也以元素序号标识元素，它们的实现，同样由于有了前面的指针类操作而变得简单。

1. Prior(long idx)

```
template <class TElem>
TElem *TLinearListLink< TElem>::Prior(long idx)
```

```

{
    TLinkNode<TElem> *p;

    p= GetNode(idx-1); //如果 idx 非法，则此处抛郑与 GetNode 相同的异常
    return &(p->info);
}

```

2. Next(long idx)

```

template <class TElem>
TElem *TLinearListLink< TElem>::Next(long idx)
{
    TLinkNode<TElem> *p;

    p= GetNode(idx+1); //如果 idx 非法，则此处抛郑与 GetNode 相同的异常
    return &(p->info);
}

```

(十) 一般的定位操作

这组操作按元素值查找元素的序号。它们的实现与前面给出的指针类定位很类似，也是以“摸”为主。

1. Locate(TElem &elem, long sn)

```

template <class TElem>
long TLinearListLink< TElem>::Locate(TElem &elem, long sn)
{
    TLinkNode<TElem> *p;
    long k, i, j, j0;

    k=sn;
    if (k<0) k = CountElem(elem)+sn+1;
    //使用 CountElem(elem)求出的值为 elem 的元素的个数，将负的 sn 转化为正
    if (k<0 )    throw TExcepLinearList(2);

    p=head;
    i=0; j=0;j0=0;
    while (p!=NULL) //从头到尾扫描各结点
    {
        if (p->info==elem) //遇到值为 elem 的结点时计数

```

```
{
    i++;
    j0=j;
    if (k==i) break;
}
p=p->next;
j++;
}

if (j0<=0) throw TExcepLinearList(3); //Not found
else return j; //发现，但 sn 大于匹配的元素个数
}
```

2. Locate(TElem &elem, long *foundElemIndex)

```
template <class TElem>
long TLinearListLink< TElem>::Locate(TElem &elem, long *foundElemIndex)
{
    TLinkNode<TElem> *p;
    long i, j;

    p=head;
    i=0; j=0;
    while (p!=NULL)
    {
        if (p->info==elem)
        {
            foundElemIndex[i]=j;
            i++;
        }
        j++;
        p=p->next;
    }

    if (i<=0) throw TExcepLinearList(3); //未发现
    return i;
}
```

3. LocateFirst(TElem &elem)

```
template <class TElem>
long TLinearListLink< TElem>::LocateFirst(TElem &elem)
{
    TLinkNode<TElem> *p;
    int k;

    k=0;
    p=head;
    while (p!=NULL)
    {
        if (p->info==elem)
        {
            lastVisited = p;
            lastVisitedIndex =k;
            break;
        }
        k++;
        p=p->next;
    }

    if (p==NULL) return -1; //Not found
    else return k; //Found
}
```

4. LocateNext(TElem &elem)

```
template <class TElem>
long TLinearListLink< TElem>::LocateNext(TElem &elem)
{
    TLinkNode<TElem> *p;
    int k;

    k=1;
    p=lastVisited->next;
    while (p!=NULL)
    {
        if (p->info==elem)
        {
```

```

        lastVisited = p;
        lastVisitedIndex+=k;
        break;
    }
    k++;
    p=p->next;
}

if (p==NULL) return -1; //Not found
else return k; //Found
}

```

5. CountElem(TElem &elem)

```

template <class TElem>
long TLinearListLink< TElem>::CountElem(TElem &elem)
{
    TLinkNode<TElem> *p;
    long cnt;

    cnt=0;
    p=head;
    while (p!=NULL)
    {
        if (p->info == elem) {cnt++;p=p->next;}
        else p=p->next;
    }
    return cnt;
}

```

(十一) 一般的插入/删除操作

这类操作在进行插入/删除时，以序号指称插入元素的位置或被删除的元素位置。由于通过 GetNode(idy)可获得序号 idy 对应的结点的指针，故这些操作可通过上面介绍的指针类插入/删除操作实现。

1. Insert(TElem &elem, long sn): 在第 sn 个元素之前插入元素 elem.

```

template <class TElem>
TElem *TLinearListLink< TElem>::Insert(TElem &elem, long sn)
{

```

```

TLinkNode<TElem> *p;
p = new(nothrow) TLinkNode<TElem>;
if (p==NULL) throw TExcepLinearList(4);

SetNodeElem(p, elem); //为新结点 p 置值 elem
Insert(p, sn); //调用成员函数将 p 插入到第 sn 个元素之前
return &elem;
}

```

2. Delete(long sn): 删除第 sn 个元素.

```

template <class TElem>
TElem *TLinearListLink< TElem>::Delete(long sn)
{
    TLinkNode<TElem> *p;
    long k;

    if (sn<0) k=len+sn; //换算 sn 为序号
    else k=sn-1;

    if (k<1) //删除第一个结点
    {
        p = head;
        head = head->next;
        len--;
    }
    else
    {
        p=GetNode(k-1); //调用 GetNode() 求出下标为 k-1 的元素的指针
        p=DeleteAfter(p); //删除 p 的后继
    }

    buffElem = p->info;
    delete p;
    return &buffElem;
}

```

3. Delete(TIndexSelector &sel, TElem *elemDeleted)

该操作的实现留作练习。

4. DeleteByIndex(long *idxTobeDel, long numIdx, TElem *elemDeleted)

该操作的实现留作练习。

§ 3.5 几种特殊线性链表

§ 3.5.1 带头结点的链表.

有时候为了处理上的方便，在线性链表的第一个结点的前面增设一个特殊的结点，称为头结点。头结点逻辑上不属于相应的线性链表，它的作用主要有两点，一是存贮一些有关线性表的信息（如表中结点总数……），另一是为了算法处理上的方便。前面已看到，在实现插入与删除操作时，要考虑插入或删除操作是否针对的是表中第一个结点。但若增设了头结点，就可用统一的方法处理了。

增设头结点后，链表的头指针指向头结点，而头结点才指向第一个元素结点。

与普通链表相比，带头结点的链表只是规定第一个结点不存放表元素，其存储结构仍然同普通链表。对这类链表的处理，与不带头结点的链表的处理类似（有些操作更简单了！）。

图 3-1 给出了一个带头结点的线性表链表的例子，它所代表的线性表为(10,30,20)，

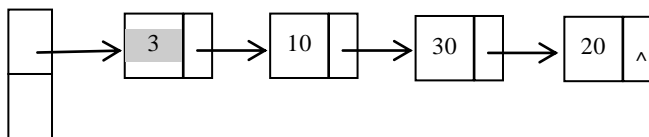


图 3-6 带头结点的链表

链中第一个结点（带底纹的结点）的数据域存放线性表长度。

🔊 要特别注意的是，此类表空表时仍含有头结点。

§ 3.5.2 循环链表

在线性表中，如果我们将第 1 个结点视为最后一个结点的后继，将最后一个结点视为第 1 个结点的前趋，则这种线性表称为循环线性表，相应的链表称循环线性链表（简称循环链表）。见**错误!未找到引用源。**。

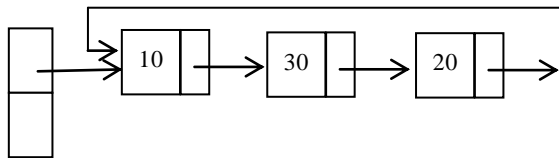
使用循环链表的主要目的是为了在搜索到最后一个结点时能转到第 1 个结点上。若不带头结点，则链表中各结点处于“平等”地位，无先后（即序号大小）之分，有些应用正需要这种逻辑结构，如循环队列、菜单等。

为了处理方便，循环链表多带头结点，构成带头结点的循环链表。

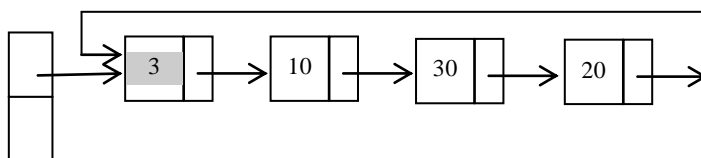
与普通链表相比，循环链表也只是在结点使用方面的改变，其存储结构仍然同普通链表，但对需扫描表的算法，原来通过判断当前结点指针是否为空，来判断是否已扫描

完（因为最后一个结点的 `next` 域空）。现在，最后一个结点的 `next` 域不空，因此不能再使用该条件判别了。

在循环链表的情况下，最后一个结点的特征是，它的 `next` 等于第一个结点的指针。利用此条件可控制扫描是否结束（是否扫到了尾）。对**不带头结点的循环链表**，扫描一



(a) 不带头结点



(b) 带头结点

图 3-7 循环单链表示意

般的模式为：

```
p = head;
if (p!=NULL) //查找链表是否为空
{
while (p->next!=head) //当 p 不是最后一个结点 p 时循环
{
... .. //访问结点 p
p = p->next;
}
... .. //这里要访问最后一个结点 p;
}
```

对**带头结点的循环链表**，扫描一般的模式为：

```
p = head->next;
while (p!=head)
//空链表时，仍然有一个结点 head，其不空，
//但它的 next 直接指向自己，故此时该循环条件不满足，即空链表时不进入循环。
{
... .. //访问结点 p
```

```
p = p->next;  
}
```

§ 3.5.3 双向链表

前面介绍的链表，每个结点有一个指针，指向相应结点的后继，因此，在某一结点上只能直接访问该结点的后继，而不能直接访问到它的前趋，即只能向后继方向搜索。为区别，有时称这种链表为单向链表。但有些应用要求能向两个方向搜索，对此，前面的存贮方法就不适应了。双向链表就是为了解决该问题提出的。

为单向链表的每个结点增设一个指向相应结点的前趋的指针，使其同时包含前驱和后继链，这样的链表称为**双向链表**（简称双链表）。双向链表的结点的结构如下所示：

prior	info	next
-------	------	------

这里各项含义为：

info----存放元素内容；

next----存放该元素的后继的指针（地址）；

prior----存放该元素的前驱的指针（地址）；

在双链表中扫描和搜索与单向链表类似，不同的是其更加方便，可随时向前或向后移动。但对插入与删除，由于增加了一个链，所以存在维护所增加的链的问题。下面单独介绍。

(一) 双链表的插入

现设在结点 p 之前插入结点 q ，其程序片段如下。

- (1) $q \rightarrow \text{next} = p;$
- (2) $q \rightarrow \text{prior} = p \rightarrow \text{prior};$
- (3) $p \rightarrow \text{prior} \rightarrow \text{next} = q;$
- (4) $p \rightarrow \text{prior} = q;$

图 3-8 说明了上面的程序。注意，上列语句是有次序关系的：语句（4）必须位于语句（3）之后，否则，先打断了 $p \rightarrow \text{prior}$ 这条链，后面就不能用 p 获得 p 的前驱了。一般情况下，应先改变那些不会影响其他结点的链，如我们这里，先改变 q 所指结点的链，就不影响整个链表。

至于双向链表的其他插入方式，如按序号插入，都以此算法为基础，此算法与下面的删除算法一起，完全屏蔽了单双链表的区别，使得双链表的其他插入算法，与单链表相同。故这里不再介绍这些算法。

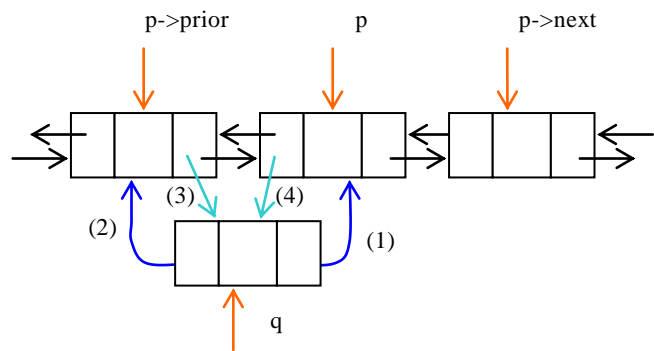


图 3-8 双链表插入

(二) 双链表删除

现设删除结点 p 所指结点，程序片段如下。

- (1) $p->prior->next=p->next;$
- (2) $p->next->prior=p->prior;$
- (3) $\text{return } p;$

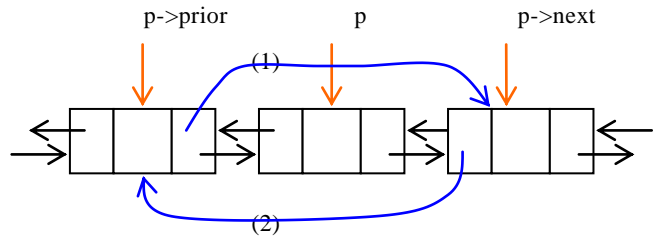


图 3-9 双链表的删除

图 3-1 说明了上面的程序。显然，这里与插入不同，上列语句的执行次序无关。

至于双向链表的其他删除方式，如按序号删除，也都以此算法为基础，此算法与上面的插入算法一起，完全屏蔽了单双链表的区别，使得双链表的其他删除算法，与单链表相同。故这里不再介绍这些算法。

§ 3.5.4 对称链表*

这里给出双向链表的一种压缩实现——**对称链表**。这是一种利用结点的“伪指针”与其它信息获得它的前趋与后继地址的方法。

在双向链表中，为了能直接访问结点的前趋与后继，为结点设置了两个指针，分别指向它的前趋与后继。这样虽带来了灵活性，但增加了存贮开销。那么，能否只通过一个指针域与其它一些辅助信息，就可起到双向链表的作用呢？对称链表就是为了解决这一问题而引入的。

(一) 异或运算

让我们回忆一下异或（按位加）运算。用“ \oplus ”代表异或运算符。设 a 与 b 为逻辑值（二进制 0 或 1），定义

$$a \oplus b = 0$$

当且仅当 a 与 b 相等，否则 $a \oplus b$ 定义为 1。

异或运算有许多奇妙的性质，我们这里只关心下列两个性质：

$$(a \oplus b) \oplus b = a$$

$$(a \oplus b) \oplus a = b$$

这两个公式也称吸收律，即用一个量通过与另一个包含它的异或式异或，可吸收掉它。

设 x 与 y 是两个二进制位串， $x \oplus y$ 表示按位异或，则容易验证上列两个式子在这种情况下亦成立，即

$$(x \oplus y) \oplus y = x$$

$$(x \oplus y) \oplus x = y$$

该等式表明， x 、 y 与 $x \oplus y$ 这三者中，从任意两个，经异或运算，可求得另一个。

(二) 对称链表结构

上面给出的异或的性质提示我们，对线性表 $L=(a_1, a_2, \dots, a_n)$ ，如果令结点 a_i 中保存 a_{i-1} 与 a_{i+1} 的地址的异或值，则当已知 a_{i-1} 与 a_i 的地址时，就可求出 a_{i+1} 的地址（因 a_i 中存有 a_{i-1} 与 a_{i+1} 地址的异或）。反过来，当已知 a_i 与 a_{i+1} 地址时，可求得 a_{i-1} 的地址。据此，可以按下法组织线性表。设链表结点结构为：

A_i	a_i 前趋的地址 \oplus a_i 后继的地址
-------	----------------------------------

这里，用 A_i 表示结点 a_i 的内容。图 3-1 给出一个非循环的不带头结点的对称链表的示意。

由于 0 与任何数 X 异或的结果仍为 X ，所以，在非循环链表中，第一个结点 a_1 的指针域的值可视为 $0 \oplus a_2$ ，最后一个结点 a_n 的指针域的值可视为 $a_{n-1} \oplus 0$ 。

如果将空指针值定义为 0，则当某结点的前驱或后继为空时，在该结点的地址域中，用 0 代替相应的地址。这样，对所有结点都可统一处理了。对循环链表，显然不存在此问题。

由于地址间进行异或运算的结果仍为地址，所以这种表的结点的 C 语言描述同单链

表。

下面考虑一些基本操作的实现。

对称链表中的涉及链指针的操作要比普通链表复杂一些。这里就几个基本问题进行讨论。

下面的讨论中，设当前结点为 a_i （当前正处理到 a_i ，并持有它的地址），它的地址为

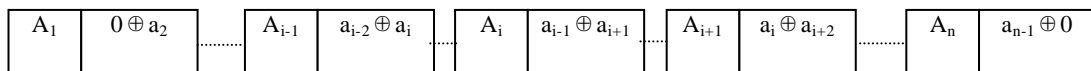


图 3-10 对称链表示意

p ，它的前趋与后继的地址分别为 pre 与 suc （无前趋或后继时，相应的地址用 0 值代替），它的地址域仍用 $next$ 表示。

(三) 指针移动

1. 向后继方向移动

应一直持有当前结点的前趋的地址（当然也应持有当前结点的地址）。当前结点 p 的后继的地址为（C 中用 \wedge 作按位异或运算符）

$p \rightarrow next \wedge pre$

则将 p 移到下一个结点（后继）的操作为：

$suc = p \rightarrow next \wedge pre$; // 使 suc 保存 p 的后继 a_{i+1} 的地址 $(a_{i-1} \oplus a_{i+1}) \oplus a_{i-1}$

$pre = p$; // 当前 p 即为下个结点的前驱 pre

$p = suc$; // 使 p 为它的后继的地址

$suc = suc \rightarrow next \wedge pre$; // 使 suc 为新的 p 的后继。若不考虑向前移动,可不要此句

2. 向前趋方向移动

应一直持有当前结点后继的地址（当然也要持有当前结点的地址）。当前结点 p 的前趋的地址为：

$p \rightarrow next \wedge suc$

则将 p 移动到它的前趋的操作为

$pre = p \rightarrow next \wedge suc$; // 使 pre 保存 p 的前驱 a_{i-1} 的地址 $(a_{i-1} \oplus a_{i+1}) \oplus a_{i+1}$

$suc = p$; // 当前 p 即为上个结点的后继 suc

$p = pre$; // 使 p 为它的前驱的地址，即令 p 移到它的前驱

$pre = pre \rightarrow next \wedge suc$; // 使 pre 为新的 p 的前驱。若不考虑向前移动，则可不要此句

(四) 插入与删除

1. 插入

我们考虑在当前结点 p 之前插入结点 q 。

此问题与普通双向链表的插入类似，要改变三个结点的链域： p 、 p 的前趋与 q 。因要访问 p 的前趋，所以必须持有 p 的后继的地址 suc 。

因 q 要插入到 p 之前，故 q 的链域应置为

$$pre \wedge p$$

而 p 的前趋 pre 的链域应改为（设 $pre0$ 为 pre 的前趋）：

$$pre0 \wedge q$$

p 的链域应改为

$$q \wedge suc$$

具体操作为：

//设已知 p 与 suc

$pre = p \rightarrow next \wedge suc$; //为简明，引入 pre

$q \rightarrow next = pre \wedge p$;

$pre \rightarrow next = (pre \rightarrow next \wedge p) \wedge q$;

$p \rightarrow next = q \wedge suc$;

2. 删除

考虑删除结点 p 。此时， p 的前趋与后继的链域均需改变。这里仍假定已知被删结点的地址 p 及其后继的地址 suc 。

//设已知 p 与 suc

$pre = p \rightarrow next \wedge suc$; //为易读引入 pre

$pre \rightarrow next = (pre \rightarrow next \wedge p) \wedge suc$;

$suc \rightarrow next = pre \wedge (suc \rightarrow next \wedge p)$;

§ 3.5.5 基于取模运算的对称表*

前面介绍的对称表是通过异或运算的对称表，如果所用工具未提供按位异或功能，可以改用加减法与取模运算实现类似的功能，这就是我们要介绍的基于取模运算的对称表。

为了书写方便，记（即重定义运算符 \oplus 与 \odot ）

$$x \oplus y = (x + y) \bmod M$$

$$x \odot y = (x - y) \bmod M$$

这里， x 与 y 为整数， M 是大于所有 x 与 y 的一个整数。可以验证：

$$(x \oplus y) \odot y = x$$

$$(x \oplus y) \odot x = y$$

此两式与上节中介绍的式类似，不同的是多了一种运算 \odot ，因此它与有类似的功用。

在结点 a_i 的链域中仍存放 $a_{i-1} \oplus a_{i+1}$ ，这与前面介绍的方法相同。但在求 a_i 的前趋或后继地址时，使用运算 \odot ，其它方面与基于异或的对称表类似，此处不再赘述。

§ 3.6 线性表应用示例

本节用两个较为典型的例子说明线性表的应用。

§ 3.6.1 集合运算*

对集合结构，一般可用线性表表示。所以，对集合的操作，可以在线性表上进行。

事实上，完全可以模仿线性表设置一个集合类。这里只从算法角度介绍一种集合运算—— $(A-B) \cup (B-A)$ 的实现，完整的集合类设计，留作练习。

由集合运算的定义知， $(A-B) \cup (B-A)$ 的结果是 A 与 B 的非共部分所构成的集合。在我们的实现中，假定最后将 $(A-B) \cup (B-A)$ 的结果存储在 A 中，B 保持不变。

实现策略是，对 B 中每个元素，检查其在 A 中是否出现，若是，从 A 中删除之，否则将之加到 A 中。这里还假定 A 与 B 是整数集合，它们中已无重复元素。

为了说明前面给出的线性表类的使用，我们的实现在类 TLinearListSqu 的基础上进行。对应的子程序如下。

```
long DiDiff(TLinearListSqu<long> &a, TLinearListSqu<long> &b)
{ //将 a 中的出现在 b 中的元素删除，将 b 中的未出现在 a 中的元素插入到 a 中。
  //返回 a 中元素数目的变化量；
  //a 和 b 都是前面定义的线性表类，元素类型实例化为 long。
  long i, j, k;
  j = 0;
  for (i = 0; i < b.len; i++) //扫描 b
  {
    k = a.Locate(b.Get(i), 1); //依次检查 b 中每个元素是否在 a 中
    if (k > 0) //如在 a 中，则从 a 中将其删除
      { a.Delete(k+1); j--; }
    else { a.Insert(b.Get(i)); j++; }
  }
  return j;
}
```

☞ 我们这里假定集合用顺序存储结构的线性表表示。显然，如果用链式结构，即函数原型定义为

```
long DiDiff(TLinearListLink<long> &a, TLinearListLink<long> &b)
```

程序体也不需要改变，这也说明所用抽象结构带来的好处。

§ 3.6.2 一元多项式相加

本小节中介绍一元多项式的符号运算（以符号加法为例）的计算机实现问题。这也是表处理中的一个经典例子。

(一) 一元多项式的表示——数据结构

在数学上，一个一元 n 次多项式可表示为

$$P_n(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$$

它由 $(n+1)$ 个系数序列 p_0 、 p_1 、 \dots 、 p_n 唯一地确定。因此，在计算机中，它可用一个线性表 P 来表示：

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

其中， p_i 代表 $P_n(x)$ 中的 i 次项系数。

在这种表示法中，系数所对应的指数隐含在系数的序号中，所以值为 0 的系数也要列出。

现考虑两多项式进行符号相加的问题。设 $Q_m(x)$ 是另一个一元 m 次多项式，它的线性表表示为

$$Q = (q_0, q_1, \dots, q_m)$$

不失一般性，设 $m < n$ ，则两个多项式相加结果 $R_n(x) = P_n(x) + Q_m(x)$ 仍为一个一元 n 次多项式，它的线性表表示为

$$R = (p_0 + q_0, p_1 + q_1, \dots, p_m + q_m, p_{m+1}, \dots, p_n)$$

但是，一般多项式的系数很不完整，许多系数的值为 0，若按上面的表示方法，为了能够隐含表示指数，值为零的系数也要写出，这样就会出现很多 0 值元素，造成存贮浪费。例如，

$$P_{100}(x) = 1 + 2x^4 + 5x^{100}$$

只有 3 个非零项，101 个系数中有 98 个为 0。

为解决上述问题，可以不存贮 0 值元素。但这样就不能利用位置关系隐含指出系数对应的指数了，而必须显式地给出指数。

对任一个一元 n 次多项式，若不写出系数为 0 的项，则可表示为

$$p_n(x) = p_1x^{e_1} + p_2x^{e_2} + \dots + p_nx^{e_n}$$

这里， p_1, p_2, \dots, p_n 均非 0， $e_1 < e_2 < \dots < e_n = n$ 。

显然，对此形式多项式，可确定地用下列形式的线性表表示

$$((p_1, e_1), (p_2, e_2), \dots, (p_n, e_n))$$

该线性表每个元素是个二元组 (p_i, e_i) ，分别指出第 i 个非 0 项的系数和指数，二元组按指数递增的次序排列。在这种结构中，元素之间的次序关系仅代表元素对应的指数的大小关系。

对这种线性表，既可用顺序存贮结构，也可用链式存贮结构。但考虑到在进行符号加法时，要经常进行插入与删除操作，所以采用链式存贮结构较为合适。这里，我们采用动态链式存贮结构，线性表每个元素的结构为

系数	指数

它的 C/C++ 语言描述为

```
struct TPolynomialItem
{
    float coef;
    int exp;
}
```

该类型在我们前面给出的线性表中，相当于元素类型 TElem，在具体使用线性表类时，应使用 TPolynomialItem 实例化 TElem 对应的类模板。

为处理方便，在具体存储多项式时，我们规定：

所存储的多项式已约简，即已合并同类项，不保留 0 系数项；

各项按指数的升序排列；

在实际中，为了使原始的多项式线性表满足上述两点，可能需要设置专门的处理程序，这里不作讨论，留作练习。

下面我们给出多项式加法算法。为了比较，我们给出两种算法：一是直接操作链表，另一是借前面给出的 TLinearListLink 的基本操作。前者效率高，后者简便且可移植性强。

(二) 多项式加法实现——直接操作链表

为操作方便，我采用带头结点的非循环链表，下面给出一个例子说明多项式的这种表示法。

设有一个一元 5 次多项式：

$$P_5(x) = 7 + 3x - 9x^3 + x^5$$

它的带头结点的非循环链表表示如图 3-1。

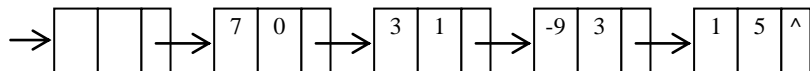


图 3-11 一个一元 5 次多项式的链表表示

一元 n 次多项式的（符号）相加，实质上是合并同类项的过程，即指数相同的项，其系数相加，指数不同的项复抄。

下面考虑 $A(x)+B(x) \rightarrow A(x)$ 的实现方法。即将多项式 $B(x)$ 加到多项 $A(x)$ 上面，这里假定各多项式均已约简，且各项已按升序排列。

用高级语言实现时，设两个指针 p 和 q ，初始时它们分别指向 $A(x)$ 与 $B(x)$ 中当前未被处理的结点中指数最小者。进行相加的过程，实质上是重复进行下列几个步骤：

- 若 $p \rightarrow \text{exp} < q \rightarrow \text{exp}$ ：则结点 p 应为和的一个结点，故 p 后移一步， q 不动。
- 若 $p \rightarrow \text{exp} > q \rightarrow \text{exp}$ ：则结点 q 应为和的一个结点，故应将 q 从 B 摘除后插入到 $A(x)$ 中 p 之前，然后 q 向后移一步， p 不动。
- 若 $p \rightarrow \text{exp} == q \rightarrow \text{exp}$ ：则表明 p 与 q 所指项为同类项，应合并，故要将 q 的系数加到 p 的系数上，若相加结果为 0，则表明和式中无此项，故应从 $A(x)$ 中删除 p ，从 $B(x)$ 中删除 q ，并令 p 与 q 分别指向下一结点。若相加和不为 0，则表明相加结果应为和式中的一个结点， p 后移一步，然后将 q 从 $B(x)$ 中摘除，令 q 指向下一结点。

在算法运行中， $B(x)$ 的链表中的结点，或被转移到 $A(x)$ 链表上，或被删除，运行结束后， $B(x)$ 链表就不存在了，而 $A(x)$ 链表就是所求的和。当然，也可以设计一个将 B 加到 A 上，但保留 B ，或将 $A+B$ 之和放当另一链表中的算法，具体留作练习。

下面先给出算法的伪码。

```

p=A 的第一个元素；
q=B 的第一个元素；
while (p 存在 && q 存在)
{
    if (p 的指数 < q 的指数)
        { p0 = p; p = p->next; }
    else
        if (p 的指数 > q 的指数)
        {
            将 q 插入到 p 之前；
            令 p0 指向插入后的 q，即 p 的当前前驱；
            令 q 指向它原所指的下一个结点；
        }
    else
    {
        x = p 的系数 + q 的系数之和；
        if (x==0)
        {
            删除 p；
            使 p 指向它原指结点的下一个；
        }
    }
}
else

```

```

{
    令 p 的系数为 x;
    使 p 指向它原指结点的下一个;
}
删除 q;
使 q 指向它原指结点的下一个;
} // if (p 的指数 > q 的指数) ... else

} //while
if (q 不为空) 将 q 挂接在 p 之后;

```

该程序不断比较 A 链和 B 链中的一对结点的指数值（我们称其为当前结点）。开始时 A 链和 B 链中参加比较的（当前结点）都是它们的第一个元素。

主循环 while 结束后，有三种情况之一：

- A 链和 B 链同时被处理完；
- 只有 B 链处理完；
- 只有 A 链处理完。

对第一种和第二种情况，不需要“善后”处理。对第三种情况，B 链中尚有未被处理完的结点，需将其挂接在结果链的尾部。循环外的“if (q 不为空) 将 q 挂接在 p 之后”就是处理该情况的。

有了上面的伪码，结合具体的数据结构，就不难写出完整的程序了。

一元 n 次多项式加法程序 PolynomialAdd 如下。为了能访问到 TLinearListLink 的私有成员，下面的 PolynomialAdd 函数应指定为 TLinearListLink 的友元。

```

int PolynomialAdd(TLinearListLink<TPolynomialItem> &a,
                  TLinearListLink<TPolynomialItem> &b)
{
    //线性表 a 和 b 的元素类型被实例化为 TPolynomialItem
    TLinkNode<TPolynomialItem> *p, *p0, *q, *q0;
    float x;
    long k;

    k=0;
    p = a.head->next;
    p0 = a.head;
    q = b.head->next;

    while (p!=NULL && q!=NULL)
    {

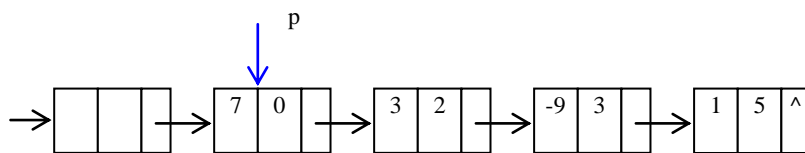
```

```

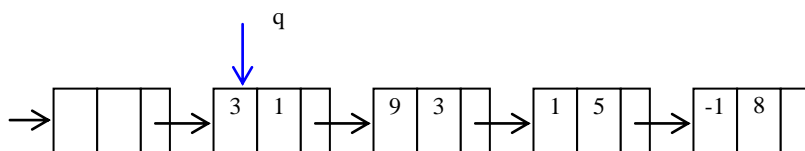
if (p->exp < q->exp)
{
    p0 = p; //永远使 p0 保持为 p 的前驱，以方便对在 p 前插入，或删除 p
    p = p->next;
    k++;
}
else
if (p->exp > q->exp )
{
    q0 = q; //在下面用 q0 操作原 q
    q = q->next; //q 先行一步
    q0->next = p;
    p0->next = q0; // 以上两句，将 q0 插入到 p0 之后（即 p 之前）
    k++;
}
else
{
    x = p->coef + q->coef;
    if (x==0)
    {
        p0->next = p->next;
        delete p; //以上两句，将 p 从表中删除并将其所指对象释放
        p = p0->next; //令 p 指向相对于它原指的下一个
    } // if (x==0)
    else
    {
        p->coef = x;
        p0 = p;
        p = p->next;
    } // if (x==0) ... else ...
    q0 = q;
    q = q->next;
    delete q0; //将 q 所指结点从表中删除并释放，令 q 新指向原所指的下一个
    } // if (p->exp > q->exp ) ... else ...
} //while
if (q!=NULL) p0->next = q;
b.head->next = NULL; //此时，b 中已只剩第一个结点（头），为其置空表标志
return k; //返回结果链表中的元素个数
}

```

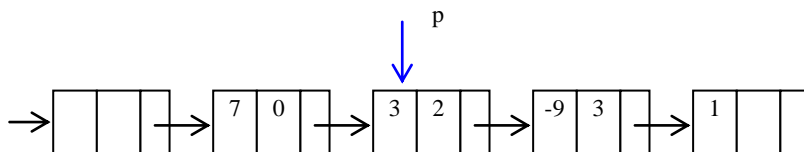
为了进一步说明上列程序，我们给出一个程序运行例子，其各循环的运行结果如图 3-1 所示。



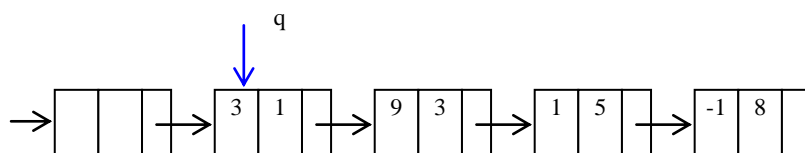
(a) $A(x) = P_5(x) = 7 + 3x^2 - 9x^3 + x^5$
进入循环前



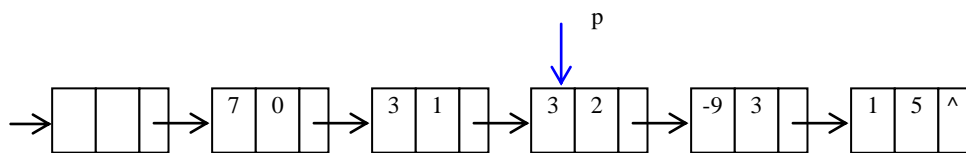
(b) $B(x) = 3x + 9x^3 + x^5 - x^8$
进入循环前



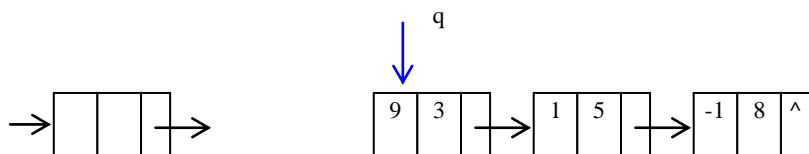
(c) $A(x)$: 第 1 次循环后
p 后移



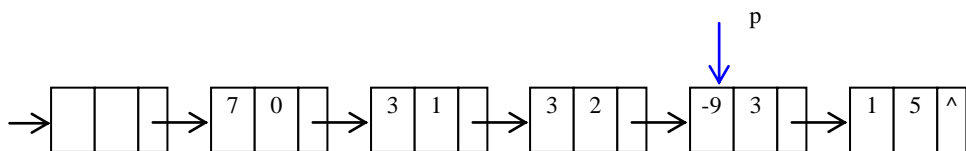
(d) $B(x)$: 第 1 次循环后
q 保持不变



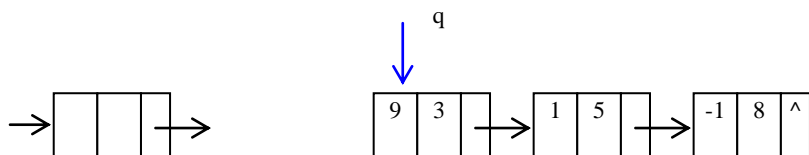
(e) A(x):第2次循环后
q 被插入到 p 前



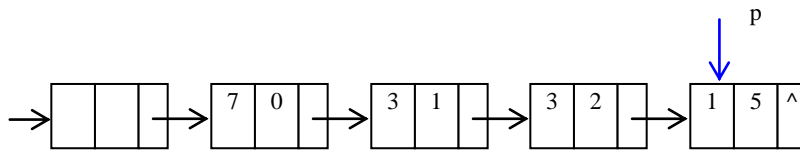
(f) B(x): 第2次循环后
q 被插入到 p 之前, q 新指向下一个



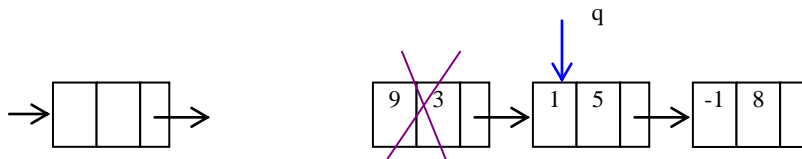
(g) A(x):第3次循环后
p 后移



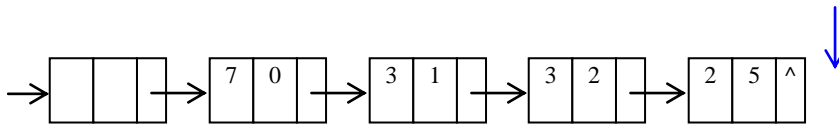
(h) B(x): 第3次循环后
q 保持



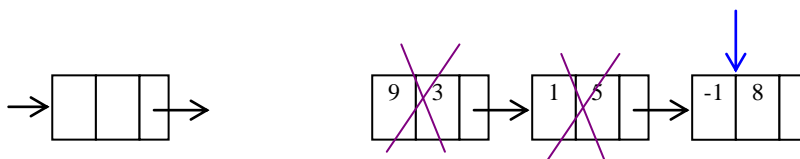
(i) A(x): 第 4 次循环后
p 被删除, 新指向下一个



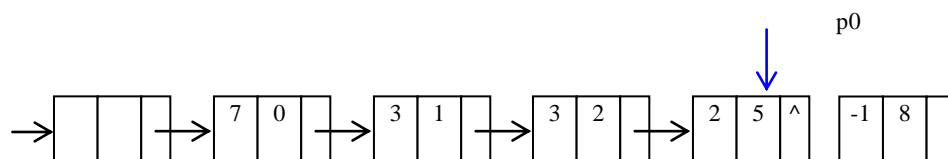
(j) B(x): 第 4 次循环后
q 被删除, 新指向下一个



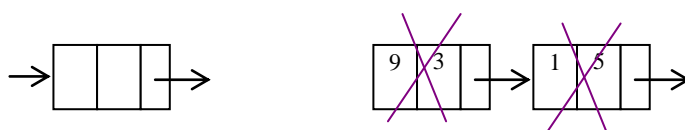
(k) A(x): 第 5 次循环后
p 与 p 合并, p 新指向下一个 (空)



(l) B(x): 第 5 次循环后
q 合并到 p, 然后被删除, 新指向下一个



(m) A(x):退出循环后
q 后面的挂接到了 p 的前驱的尾



(n) B(x): 退出循环后, q 挂到了 p 的前驱后面

图 3-12 多项式链表相加

(三) 多项式加法实现—借助抽象操作

下面在前面给出的 TLinearListSqu 类(对 TLunearListLink 也相同)的基础上实现一元多项式相加。

首先, 定义表示多项式的元素的类型。多项式的元素类型已不是象 long、float 等那样的简单类型, 所以必须考虑如何兼容基本操作中使用的赋值(=)运算、恒等运算(==)和输出运算(<<)等, 即定义相应的运算符重载函数。

```
struct TPolynomialItem
{
    float coef;
    int exp;

    char operator ==(TPolynomialItem &i1) //重载恒等算符
    {
        return (coef == i1.coef && exp==i1.exp); //指数和系数均分别相等才为恒等
    };

    TPolynomialItem& operator =(TPolynomialItem &i1) //重载赋值算符
    {
```



```

        this->coef = i1.coef; //定义多项式项赋值为指数和系数分别赋值
        this->exp = i1.exp;
        return i1;
    };
};

ostream& operator << (ostream& oo, TPolynomialItem &i1)
{ //重载标准输出算符，以支持 Print()
    oo<<i1.coef<<","<<i1.exp<<" ";
    return oo;
};

```

有了上面的定义，我们可以写出多项式加法程序了：

```

long PolynomialAdd(TLinearListSqu<TPolynomialItem> &a,
                  TLinearListSqu<TPolynomialItem> &b)
{
    long currE1, currE2, k;
    TPolynomialItem e1,e2;

    k=0;
    currE1=0;
    currE2=0;
    while (currE1 <a.len  && currE2<b.len)
    {
        e1=a.Get(currE1);
        e2=b.Get(currE2);

        if (e1.exp < e2.exp)
        { currE1++; k++; }
        else
        if (e1.exp > e2.exp)
        { a.Insert(e2,currE1+1);
          currE1++;
          b.Delete(currE2+1);
          k++;
        }
        else
        {
            e1.coef = e1.coef + e2.coef;

```

```

        if (e1.coef==0)
            a.Delete(currE1+1);
        else
            {a.Set(currE1,e1);
              currE1++;
            }
        b.Delete(currE2+1);
    }
}
if (currE2 < b.len)
    for (int i = currE2+1; i<b.len+1; i++)
        a.Insert(*(b.Delete(i)), a.len+1);
return k;
}

```

在该程序中，我们只用到了抽象类 `TLinearList0` 中定义的操作，因此，只要将程序中的 `TLinearListSqu` 改为 `TLinearListLink`，也完全适合于链式存储。

但对链式存储，由于上列程序有较高的抽象性，没有充分利用链表的特点，效率不高。我们也可以利用 `TLinearListLink` 新引入的基本操作实现该算法，这样会有较高的效率。具体实现，留做练习。

§ 3.6.3 一元多项式的乘法*

设 $A_m(x)$ 与 $B_n(x)$ 为两个一元多项式，设

$$A_m(x) = \sum_{i=1}^m a_i x^{e_i}$$

$$B_n(x) = \sum_{i=1}^n a_i x^{p_i}$$

它们的积为

$$A_m(x) \cdot B_n(x) = B_n(x) \sum_{i=1}^m a_i x^{e_i} = \sum_{i=1}^m (a_i B_n(x)) x^{e_i}$$

它中的每一项 $a_i B_n(x)$ ，都是一个关于 x 的一元多项式。由此可知，两个一元多项式相乘，可以化为多个一元多项式相加，这可利用前面给出的算法实现。

本章小结

本章介绍了最基本的一种数据结构----线性表，包括它的面向对象抽象模型和两种存储实现（顺序和链式）。

线性表的主要特征是表中每个元素有唯一的前驱和唯一的后继（第一个元素可以无前驱，最后一个可以无后继）。

数据结构的面向对象抽象模型用面向对象的观点定义了数据结构逻辑特性与使用接口，它是与存储结构无关的。本书采用 C++ 的纯虚类描述抽象模型。

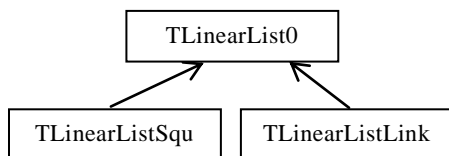
本章定义的线性表的面向对象抽象模型是 TLinearList0，一方面它代表着线性表的元素集合（数据对象），另一方面代表着线性表的基本性质（即能提供什么样的“服务”----基本操作），这些都是通过“接口”描述的。在这里，结构的形式是成员函数。

TLinearList0 提供的接口主要有以下几类：

- 按序号访问元素：Get(idx)、Set(idx,elem)、GetAddress(idx);
- 前驱/后继访问：Prior(idx)、Next(idx);
- 定位操作：Locate();
- 插入与输出操作：Insert()、Delete()。

这里，需要重点掌握的是接口的设置。设置什么样的接口才能完备地、清晰地描述对应数据结构是问题的焦点。另外，接口的参数的设置要特别注意，应该主要从使用角度考虑，使参数设置自然，同时符合模块化的准则。这方面请认真学习 Locate、Insert、Delete 几个接口的设置技巧。在这几个接口中，通过引入序号 sn 和下标选择器，使接口在不破坏简单性的前提下，功能大大扩展。其次，许多情况下要设置一些内部的接口，使接口的实现合理地层次化、模块化。

数据结构在计算机上的实现就要考虑具体的存储结构了。我们这里给出了两种存储结构：顺序和链式。从面向对象和接口的观点出发，在具体存储结构下的数据结构，是对应结构的抽象对象模型的派生类，所以，针对顺序存储结构的类 TLinearListSqu 和针对链式存储结构的类 TLinearListLink 都是从 TLinearList0 派生而来。与 TLinearList0 不同，在 TLinearListSqu 和 TLinearListLink 中都必须考虑存储空间的表示。这三个类的关系图示如下（带箭头的线段表示继承，箭头一端的对象是父对象）：



在顺序存储结构中，我们没有采用静态的存储空间，而是采用动态分配方式，即每次都利用高级语言的动态存储管理机构申请一块连续的区域，做为元素集合的存储空间。当在使用中发现存储空间不足时，自动重新分配一块更大的空间，并将原数据也复制过去。当空闲空间太多时也自动回收。这实质上是一种动静结合的方法。至于接口（基本操作）的实现，一般是很直接的。

对链式存储方式，由于元素的存储空间是动态分配的，所以不象顺序存储那样需要为整个元素空间预分配空间，具体的存储空间的分配是在插入元素的时候进行的。对基本操作的实现方面，为了针对链表的特殊性，增设了一组新的成员函数，用于按结点指针指称对象（即函数参数中表示操作对象是指针类型）。同时，这组成员函数是实现抽象类 `TLinearList0` 中规定的接口基础。

类 `TLinearListLink` 是针对单链表的（即每个元素只带有一个后继（也可以是前驱，我们这里选择后继）指针）。也可以根据需要设置其他类型的链表，如：

■ 带头结点的链表：链表中第一个结点做为头结点，不存放元素，只存放一些附加信息，元素从第二个结点起存放。

■ 循环链表：链表中的最后一个结点的后继链（若有的话）指向第一个结点，而第一个结点的前驱链（若有的话）指向最后一个结点。

■ 双链表：每个元素均同时带前驱指针和后继指针。针对双链表的类 `TLinearListDLink` 可以是单链表类 `TLinearListLink` 的派生类。派生类所做的“扩展”主要有：A)增加一个指向前驱的指针；B)修改（覆盖）一些涉及指针操作的成员函数，如结点插入、结点删除、求前驱等。

此外，我们在最后介绍了线性表的两个应用例子：集合运算和多项式相加。重点说明我们前面构造的类 `TLinearListSqu/TLinearListSqu` 的应用。

习 题 3

1 编写程序，实现在有序表（表中元素按元素关键字大小排序）中插入一个元素，使原有序表保持有序。分别假定有序表是连续存贮结构和链式存贮结构。

2 编定实现合并两个有序表，使得合并结果仍为有序表的程序。分别假定有序表按顺序方式存贮和链接方式存贮。

3 [完善§3.3.2](#) 给出的 `TindexSelector` 类。

4 假定用线性表表示集合，请定义相应的集合类，实现集合的各种基本运算（交、并、差等）及判别成员、求子集、查找元素等操作。

5 编写一个通过终端交互建立链表（动态或静态）的程序，使得较先输入的结点放在链表较前的位置。

6 写一个单链表倒链程序，即将单链表中每个结点的前趋与后继关系颠倒。

7 写一个建立双向链表的程序（分别针对动态与静态两种链表）。

8 分别针对链式与顺序存贮结构，编写程序实现 Josephus 问题：设有 n 个人围坐在一个圆桌周围，从第 s 个人开始数，数到第 m 的人就出列，然后从出列者的下一个人开始重新按上述方式数，数到 m 就出列，如此重复，直到所有的人都出列为止。要求的结果是，给出每个人的出列次序（给定 n ，对任意合法的 s 与 m ），即求一个按出列次序排列的 n 个人的顺序表。

9 编写一个一元 n 次多项式的化简程序，即将多项式内的同类项合并，0 系数项去除，并使各项按指数升序排列。

-
- 10 假定一元 n 次多项式用连续存贮结构的线性表表示（不含 0 系数项），编写两多项式相加的程序。
- 11 写一个实现两一元 n 次多项式相乘的程序。
- 12 实现线性表类中的 `Delete(TindexSelector &sel)`（注：此 s 是否英大？？？---答：小写）
- 13 在类 `TLinearListSqu` 或 `TlinearListLink` 基础上，实现 `Copy`、`Merge` 等操作。
- 14 在类 `TLinearListSqu` 或 `TlinearListLink` 基础上，实现一个学生基本信息登记表的操作，包括，输入、修改、打印、查询、统计等功能。