单链表的结点类(ListNode **class**)和链表类(List **class**)的类定义。

```
template <class Type> class List;                    //前视的类定义

template <class Type> class ListNode {               //链表结点类的定义
friend class List<Type>;                             //List 类作为友元类定义
private:
    Type data;                                       //数据域
    ListNode<Type> *link;                            //链指针域
public:
    ListNode ( ) : link (NULL) { }                   //仅初始化指针成员的构造函数
    ListNode ( const Type& item ) : data (item), link (NULL) { }
                                                     //初始化数据与指针成员的构造函数
    ListNode<Type> * getNode ( const Type& item, ListNode<Type> *next = NULL )
                                                     //以 item 和 next 建立一个新结点
    ListNode<Type> * getLink ( ) { return link; }    //取得结点的下一结点地址
    Type getData ( ) { return data; }                //取得结点中的数据
    void setLink ( ListNode<Type> * next ) { link = next; }      //修改结点的 link 指针
    void setData ( Type value ) { data = value; }    //修改结点的 data 值
};

template <class Type> class List {                   //单链表类定义
private:
    ListNode<Type> *first, *current;                 //链表的表头指针和当前元素指针
public:
    List ( const Type& value ) { first = current = new ListNode<Type> ( value ); }
                                                     //构造函数
    ~List ( ) { MakeEmpty ( );   delete first; }     //析构函数
    void MakeEmpty ( );                              //将链表置为空表
    int Length ( ) const;                            //计算链表的长度
    ListNode<Type> * Find ( Type value );            //搜索含数据 value 的元素并成为当前元素
    ListNode<Type> * Locate( int i );                //搜索第 i 个元素的地址并置为当前元素
    Type * GetData ( );                              //取出表中当前元素的值
    int Insert ( Type value );                       //将 value 插在表当前位置之后并成为当前元素
    Type *Remove ( );                                //将链表中的当前元素删去，填补者为当前元素
    ListNode<Type> * Firster ( ) { current = first; return first; }      //当前指针定位于表头结点
    Type *First ( );                                 //当前指针定位于表中第一个元素并返回其值
    Type *Next ( );                                  //将当前指针进到表中下一个元素并返回其值
    int NotNull ( ) { return current != NULL; }      //表中当前元素空否？空返回 1，不空返回 0
    int NextNotNull ( ) { return current != NULL && current->link != NULL; }
                                                     //当前元素下一元素空否？空返回 1，不空返回 0
};
```

3-1 线性表可用顺序表或链表存储。试问：

(1) 两种存储表示各有哪些主要优缺点?

(2) 如果有 n 个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变、在此情况下，应选用哪种存储表示？为什么？

(3) 若表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素，这时，应采用哪种存储表示？为什么？

【解答】

(1) 顺序存储表示是将数据元素存放于一个连续的存储空间中，实现顺序存取或(按下标)直接存取。它的存储效率高，存取速度快。但它的空间大小一经定义，在程序整个运行期间不会发生改变，因此，不易扩充。同时，由于在插入或删除时，为保持原有次序，平均需要移动一半(或近一半)元素，修改效率不高。

链接存储表示的存储空间一般在程序的运行过程中动态分配和释放，且只要存储器中还有空间，就不会产生存储溢出的问题。同时在插入和删除时不需要保持数据元素原来的物理顺序，只需要保持原来的逻辑顺序，因此不必移动数据，只需修改它们的链接指针，修改效率较高。但存取表中的数据元素时，只能循链顺序访问，因此存取效率不高。

(2) 如果有 n 个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变、在此情况下，应选用链接存储表示。

如果采用顺序存储表示，必须在一个连续的可用空间中为这 n 个表分配空间。初始时因不知道哪个表增长得快，必须平均分配空间。在程序运行过程中，有的表占用的空间增长得快，有的表占用的空间增长得慢；有的表很快就用完了分配给它的空间，有的表才用了少量的空间，在进行元素的插入时就必须成片地移动其他的表的空间，以空出位置进行插入；在元素删除时，为填补空白，也可能移动许多元素。这个处理过程极其繁琐和低效。

如果采用链接存储表示，一个表的存储空间可以连续，可以不连续。表的增长通过动态存储分配解决，只要存储器未满，就不会有表溢出的问题；表的收缩可以通过动态存储释放实现，释放的空间还可以在以后动态分配给其他的存储申请要求，非常灵活方便。对于 n 个表(包括表的总数可能变化)共存的情形，处理十分简便和快捷。所以选用链接存储表示较好。

(3) 应采用顺序存储表示。因为顺序存储表示的存取速度快，但修改效率低。若表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素，这时采用顺序存储表示较好。

3-2 针对带表头结点的单链表，试编写下列函数。

(1) 定位函数 Locate：在单链表中寻找第 i 个结点。若找到，则函数返回第 i 个结点的地址；若找不到，则函数返回 NULL。

(2) 求最大值函数 max：通过一趟遍历在单链表中确定值最大的结点。

(3) 统计函数 number：统计单链表中具有给定值 x 的所有元素。

(4) 建立函数 create：根据一维数组 a[n]建立一个单链表，使单链表中各元素的次序与 a[n]中各元素的次序相同，要求该程序的时间复杂性为 O(n)。

(5) 整理函数 tidyup：在非递减有序的单链表中删除值相同的多余结点。

【解答】

(1) 实现定位函数的算法如下：

```
template <class Type> ListNode <Type> * List <Type> :: Locate ( int i ) {
//取得单链表中第 i 个结点地址, i 从 1 开始计数, i <= 0 时返回指针 NULL
    if ( i <= 0 ) return NULL;                          //位置 i 在表中不存在
    ListNode <Type> * p = first;   int k = 0;                  //从表头结点开始检测
    while ( p != NULL && k < i ) { p = p->link;   k++; }       //循环, p == NULL 表示链短, 无第 i 个结点
    return p;                                           //若 p != NULL, 则 k == i, 返回第 i 个结点地址
}
```

(2) 实现求最大值的函数如下：

```
template <class Type> ListNode <Type> * List <Type> :: Max ( ) {
//在单链表中进行一趟检测，找出具有最大值的结点地址，如果表空，返回指针 NULL
    if ( first->link == NULL ) return NULL;                    //空表，返回指针 NULL
    ListNode <Type> * pmax = first->link, p = first->link->link;  //假定第一个结点中数据具有最大值
    while ( p != NULL ) {                                      //循环，下一个结点存在
        if ( p->data > pmax->data ) pmax = p;                  //指针 pmax 记忆当前找到的具最大值结点
        p = p->link;                                           //检测下一个结点
    }
    return pmax;
}
```

(3) 实现统计单链表中具有给定值 x 的所有元素的函数如下：

```
template <class Type> int List <Type> :: Count ( Type& x ) {
//在单链表中进行一趟检测，找出具有最大值的结点地址，如果表空，返回指针 NULL
    int n = 0;
    ListNode <Type> * p = first->link;                         //从第一个结点开始检测
    while ( p != NULL ) {                                      //循环，下一个结点存在
        if ( p->data == x ) n++;                               //找到一个，计数器加 1
        p = p->link;                                           //检测下一个结点
    }
    return n;
}
```

(4) 实现从一维数组 A[n]建立单链表的函数如下：

```
template <class Type> void List <Type> :: Create ( Type A[ ], int n ) {
//根据一维数组 A[n]建立一个单链表，使单链表中各元素的次序与 A[n]中各元素的次序相同
    ListNode<Type> * p;
    first = p = new ListNode<Type>;                            //创建表头结点
    for ( int i = 0; i < n; i++ ) {
        p->link = new ListNode<Type> ( A[i] );                 //链入一个新结点，值为 A[i]
        p = p->link;                                           //指针 p 总指向链中最后一个结点
    }
    p->link = NULL;
}
```

采用递归方法实现时，需要通过引用参数将已建立的单链表各个结点链接起来。为此，在递归地扫描数组 A[n]的过程中，先建立单链表的各个结点，在退出递归时将结点地址 p（被调用层的形参）带回上一层（调用层）的实参 p->link。

```
template<Type> void List<Type> :: create ( Type A[ ], int n, int i, ListNode<Type> *& p ) {
//私有函数：递归调用建立单链表
    if ( i == n ) p = NULL;
    else {   p = new ListNode<Type>( A[i] );                   //建立链表的新结点
             create ( A, n, i+1, p->link );                    //递归返回时 p->link 中放入下层 p 的内容
         }
}
template<Type> void List<Type> :: create ( Type A[ ], int n ) {
```

```
//外部调用递归过程的共用函数
    first = current = new ListNode<Type>;                    //建立表头结点
    create ( A, n, 0, first->link );                         //递归建立单链表
}
```

(5) 实现在非递减有序的单链表中删除值相同的多余结点的函数如下：

```
template <class Type> void List <Type> :: tidyup ( ) {
    ListNode<Type> * p = first->link, temp;                  //检测指针，初始时指向链表第一个结点
    while ( p != NULL && p->link != NULL )                   //循环检测链表
       if ( p->data == p->link->data ) {                     //若相邻结点所包含数据的值相等
           temp = p->first;    p->link = temp->link;         //为删除后一个值相同的结点重新拉链
           delete temp;                                      //删除后一个值相同的结点
       }
       else p = p->link;                                     //指针 p 进到链表下一个结点
}
```

3-3 设 ha 和 hb 分别是两个带表头结点的非递减有序单链表的表头指针，试设计一个算法，将这两个有序链表合并成一个非递增有序的单链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。表中允许有重复的数据。

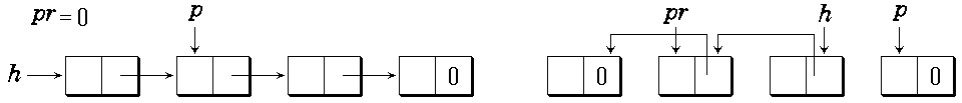
【解答】

```
#include <iostream.h>
template <class Type> class List;
template <class Type> class ListNode {
friend class List<Type>;
public:
      ListNode ( );                                  //构造函数
      ListNode ( const Type& item );                 //构造函数
private:
      Type data;
      ListNode<Type> *link;
};

template <class Type> class List {
public:
      List ( const Type finishied );                 //建立链表
      void Browse ( );                               //打印链表
      void Merge ( List<Type> &hb );                 //连接链表
private:
      ListNode<Type> *first, *last;
};

//各成员函数的实现
template <class Type>
ListNode<Type> :: ListNode ( ) : link ( NULL ) { }
//构造函数，仅初始化指针成员。
```

```
template <class Type> ListNode<Type> :: ListNode ( const Type & item ) : data ( item ), link ( NULL ) { }
//构造函数，初始化数据与指针成员。

template <class Type> List<Type> :: List ( const Type finishied ) {
//创建一个带表头结点的有序单链表, finished 是停止建表输入标志，是所有输入值中不可能出现的数值。
    first = last = new ListNode<Type>( );              //创建表头结点
    Type value;      ListNode<Type> *p, *q, *s;
    cin >> value;
    while ( value != finished ) {                      //循环建立各个结点
        s = new ListNode<Type>( value );
        q = first;   p = first->link;
        while ( p != NULL && p->data <= value )
            { q = p;   p = p->link; }                  //寻找新结点插入位置
        q->link = s;   s->link = p;                    //在 q, p 间插入新结点
        if ( p == NULL ) last = s;
        cin >> value;
    }
}

template <class Type> void List<Type> :: Browse ( ) {
//浏览并输出链表的内容
    cout<<"\nThe List is : \n";
    ListNode<Type> *p = first->link;
    while ( p != NULL ) {
        cout << p->data;
        if ( p != last ) cout << "->";
        else cout << endl;
        p = p->link;
    }
}

template <class Type> void List <Type> :: Merge ( List<Type>& hb) {
//将当前链表 this 与链表 hb 按逆序合并，结果放在当前链表 this 中。
    ListNode<Type> *pa, *pb, *q, *p;
    pa = first->link;   pb = hb.first->link;           //检测指针跳过表头结点
    first->link = NULL;                                //结果链表初始化
    while ( pa != NULL && pb != NULL ) {               //当两链表都未结束时
        if ( pa->data <= pb->data )
            { q = pa;   pa = pa->link; }               //从 pa 链中摘下
        else
            { q = pb;   pb = pb->link; }               //从 pb 链中摘下
        q→link = first->link;   first->link = q;       //链入结果链的链头
    }
```

```
        p = ( pa != NULL ) ? pa : pb;                    //处理未完链的剩余部分
        while ( p != NULL ) {
            q = p;    p = p->link;
            q->link = first->link;    first->link = q;
        }
    }
```

3-4 设有一个表头指针为 h 的单链表。试设计一个算法，通过遍历一趟链表，将链表中所有结点的链接方向逆转，如下图所示。要求逆转结果链表的表头指针 h 指向原链表的最后一个结点。



【解答 1】

```
    template<class Type> void List<Type> :: Inverse ( ) {
        if ( first == NULL ) return;
        ListNode<Type> *p = first->link, *pr = NULL;
        while ( p != NULL ) {
            first->link = pr;                            //逆转 first 指针
            pr = first;    first = p;    p = p->link;      //指针前移
        }
        first->link = pr;
    }
```

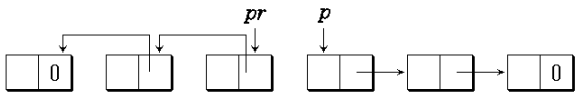
【解答 2】

```
    template<class Type> void List<Type> :: Inverse ( ) {
        ListNode<Type> *p, *head = new ListNode<Type> ( );   //创建表头结点, 其 link 域默认为 NULL
        while ( first != NULL ) {
            p = first;    first = first->link;               //摘下 first 链头结点
            p->link = head->link;    head->link = p;         //插入 head 链前端
        }
        first = head->link;    delete head;                  //重置 first, 删去表头结点
    }
```

3-5 从左到右及从右到左遍历一个单链表是可能的，其方法是在从左向右遍历的过程中将连接方向逆转，如右图所示。在图中的指针 p 指向当前正在访问的结点，指针 pr 指向指针 p 所指结点的左侧的结点。此时，指针 p 所指结点左侧的所有结点的链接方向都已逆转。

(1) 编写一个算法，从任一给定的位置(pr, p)开始，将指针 p 右移 k 个结点。如果 p 移出链表，则将 p 置为 0，并让 pr 停留在链表最右边的结点上。

(2) 编写一个算法，从任一给定的位置(pr, p)开始，将指针 p 左移 k 个结点。如果 p 移出链表，则将 p 置为 0，并让 pr 停留在链表最左边的结点上。



【解答】

(1) 指针 p 右移 k 个结点

```
template<class Type> void List<Type> ::
siftToRight ( ListNode<Type> *& p, ListNode<Type> *& pr, int k ) {
    if ( p == NULL && pr != first ) {                    //已经在链的最右端
        cout << "已经在链的最右端，不能再右移。" << endl;
        return;
    }
    int i;   ListNode<Type> *q;
    if ( p == NULL )                                     //从链头开始
        { i = 1;   pr = NULL;   p = first; }             //重置 p 到链头也算一次右移
    else i = 0;
    while ( p != NULL && i < k ) {                       //右移 k 个结点
        q = p->link;   p->link = pr;                     //链指针 p→link 逆转指向 pr
        pr = p;   p = q;   i++;                          //指针 pr, p 右移
    }
    cout << "右移了" << i << "个结点。" << endl;
}
```

(2) 指针 p 左移 k 个结点

```
template<class Type> void List<Type> ::
siftToLeft ( ListNode<Type> *& p, ListNode<Type> *& pr, int k ) {
    if ( p == NULL && pr == first ) {                    //已经在链的最左端
        cout << "已经在链的最左端，不能再左移。" << endl;
        return;
    }
    int i = 0;   ListNode<Type> *q;
    while ( pr != NULL && i < k ) {                      //左移 k 个结点
        q = pr->link;   pr->link = p;                    //链指针 pr->link 逆转指向 p
        p = pr;   pr = q;   i++;                         //指针 pr, p 左移
    }
    cout << "左移了" << i << "个结点。" << endl;
    if ( i < k ) { pr = p;   p = NULL; }                 //指针 p 移出表外，重置 p, pr
}
```

3-6 试写出用单链表表示的字符串类及字符串结点类的定义，并依次实现它的构造函数、以及计算串长度、串赋值、判断两串相等、求子串、两串连接、求子串在串中位置等 7 个成员函数。要求每个字符串结点中只存放一个字符。

【解答】

```
//用单链表表示的字符串类 string1 的头文件 string1.h
#include <iostream.h>
const int maxLen = 300;          //字符串最大长度为 300（理论上可以无限长）
class string1 {
public:
    string1 ( );                                         //构造空字符串
```

```
    string1 ( char * obstr );                          //从字符数组建立字符串
    ~string1 ( );                                      //析构函数
    int Length ( ) const { return curLen; }            //求字符串长度
    string1& operator = ( string1& ob );               //串赋值
    int operator == ( string1& ob );                   //判两串相等
    char& operator [ ] ( int i );                      //取串中字符
    string1& operator ( ) ( int pos, int len );        //取子串
    string1& operator += ( string1& ob );              //串连接
    int Find ( string1& ob );                          //求子串在串中位置(模式匹配)
    friend ostream& operator << ( ostream& os, string1& ob );
    friend istream& operator >> ( istream& is, string1& ob );
private:
    ListNode<char>*chList;                             //用单链表存储的字符串
    int curLen;                                        //当前字符串长度
}

//单链表表示的字符串类 string1 成员函数的实现，在文件 string1.cpp 中
#include <iostream.h>
#include "string1.h"
string1 :: string1( ) {                                //构造函数
    chList = new ListNode<char> ( '\0' );
    curLen = 0;
}

string1 :: string1( char *obstr ) {                    //复制构造函数
    curLen = 0;
    ListNode<char> *p = chList = new ListNode<char> ( *obstr );
    while ( *obstr != '\0' ) {
        obstr++;
        p = p->link = new ListNode<char> ( *obstr );
        curLen++;
    }
}

string1& string1 :: operator = ( string1& ob ) {       //串赋值
    ListNode<char> *p = ob.chList;
    ListNode<char> *q = chList = new ListNode<char> ( p->data );
    curLen = ob.curLen;
    while ( p->data != '\0' ) {
        p = p->link;
        q = q->link = new ListNode<char> ( p->data );
    }
    return *this;
}
```

```
int string1 :: operator == ( string1& ob ) {                    //判两串相等
    if ( curLen != ob.curLen ) return 0;
    ListNode <char> *p = chList, *q = ob.chList;
    for ( int i = 0; i < curLen; i++ )
        if ( p->data != q->data ) return 0;
        else { p = p->link;    q = q->link; }
    return 1;
}

char& string1 :: operator [ ] ( int i ) {                       //取串中字符
    if ( i >= 0 && i < curLen ) {
       ListNode <char> *p = chList;    int k = 0;
       while ( p != NULL && k < i ) { p = p->link; k++; }
       if ( p != NULL ) return p->data;
    }
    return '\0';
}

string1& string1 :: operator ( ) ( int pos, int len ) {         //取子串
    string1 temp;
    if ( pos >= 0 && len >= 0 && pos < curLen && pos + len - 1 < curLen ) {
        ListNode<char> *q, *p = chList;
        for ( int k = 0; k < pos; k++; ) p = p->link;           //定位于第 pos 结点
        q = temp.chList = new ListNode<char> ( p->data );
        for ( int i = 1; i < len; i++ ) {                       //取长度为 len 的子串
          p = p->link;
          q = q->link = new ListNode<char> ( p->data );
        }
        q->link = new ListNode<char> ( '\0' );                  //建立串结束符
        temp.curLen = len;
    }
    else { temp.curLen = 0;    temp.chList = new ListNode<char> ( '\0' ); }
    return *temp;
}

string1& string1 :: operator += ( string1& ob ) {               //串连接
    if ( curLen + ob.curLen > maxLen ) len = maxLen - curLen;
    else len = ob.curLen;                                       //传送字符数
    ListNode<char> *q = ob.chList, *p = chList;
    for ( int k = 0; k < curLen - 1; k++; ) p = p->link;        //this 串的串尾
    k = 0;
    for ( k = 0; k < len; k++ ) {                               //连接
        p = p->link = new ListNode<char> ( q->data );
```

```
            q = q->link;
        }
        p→link = new ListNode<char> ( '\0' );
    }

    int string1 :: Find ( string1& ob ) {                    //求子串在串中位置(模式匹配)
        int slen = curLen,  oblen = ob.curLen,  i = slen - oblen;
        string1 temp = this;
        while ( i > -1 )
            if ( temp( i, oblen ) == ob ) break;
            else i--  ;
        return i;
    }
```

3-7 如果用循环链表表示一元多项式，试编写一个函数 Polynomial :: Calc(x)，计算多项式在 x 处的值。

【解答】

下面给出表示多项式的循环链表的类定义。作为私有数据成员，在链表的类定义中封装了 3 个链接指针：first、last 和 current，分别指示链表的表头结点、链尾结点和最后处理到的结点。

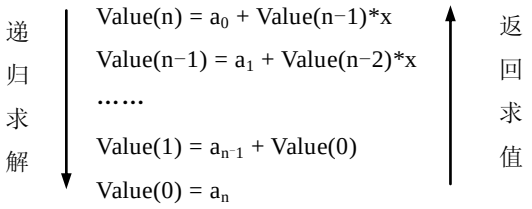
```
    enum Boolean { False, True }
    class Polynomal;                                         //多项式前视类定义

    class Term {                                             //项类定义
    friend class Polynomal;
    private:
        double coef, expn;                                   //系数与指数
        Term *link;                                          //项链接指针
    public:
        Term ( double c = 0, double e = 0, Term * next = NULL ) : coef (c), expn(e), link (next) { }
    }

    class Polynomal {                                        //多项式类定义
    private:
        Term *first, *current;                               //头指针，当前指针
        int n;                                               //多项式阶数
    public:
        Polynomal ( );                                       //构造函数
        ~Polynomal ( );                                      //析构函数
        int Length ( ) const;                                //计算多项式项数
        Boolean IsEmpty ( ) { return first->link == first; } //判是否零多项式
        Boolean Find ( const double& value );                //在多项式中寻找其指数值等于 value 的项
        double getExpn ( ) ( ) const;                        //返回当前项中存放的指数值
        double getCoef ( ) ( ) const;                        //返回当前项中存放的系数值
        void Firster ( ) { current = first; }                //将当前指针置于头结点
        Boolean First ( );                                   //将当前指针指向链表的第一个结点
```

```
    Boolean Next ( );                                   //将当前指针指到当前结点的后继结点
    Boolean Prior ( );                                  //将当前指针指到当前结点的前驱结点
    void Insert ( const double coef, double expn );     //插入新结点
    void Remove ( );                                    //删除当前结点
    double Calc ( double x );                           //求多项式的值
    friend Polynomial operator + ( Polynomial &, Polynomial & );
    friend Polynomial operator * ( Polynomial &, Polynomial & );
};
```

对于多项式 $P_n(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{n-1}x^{n-1} + a_nx^n$，可用 Horner 规则将它改写求值：

$$P_n(x) = a_0 + (a_1x + (a_2 + (a_3 + \cdots + (a_{n-1} + a_n*x)*x \cdots)*x)*x)*x$$

因为不是顺序表，必须采用递归算法实现：




```
double Polynomal :: Value ( Term *p, double x ) {
//私有函数：递归求子多项式的值
    if ( p->link == first ) return p->coef;
    else return p->coef + x * Value ( p->link, x );
}

double Polynomal :: Calc ( double x ) {
//共有函数：递归求多项式的值
    Term * pc = first->link;
    if ( pc == first ) cout << 0 << endl;
    else cout << Value ( pc, x ) << endl;
}
```

但是，当多项式中许多项的系数为 0 时，变成稀疏多项式，如 $P_{50}(x) = a_0 + a_{13}x^{13} + a_{35}x^{35} + a_{50}x^{50}$，为节省存储起见，链表中不可能保存有零系数的结点。此时，求值函数要稍加改变：

```
#include <math.h>
double Polynomal :: Value ( Term *p, double e, double x ) {
//私有函数：递归求子多项式的值。pow(x, y)是求 x 的 y 次幂的函数，它的原型在“math.h”中
    if ( p->link == first ) return p->coef;
    else return p->coef + pow( x, p->expn – e ) * Value ( p->link, p->expn, x );
}

double Polynomal :: Calc ( double x ) {
//共有函数：递归求多项式的值
    Term * pc = first->link;
    if ( pc == first ) cout << 0 << endl;
    else cout << Value ( pc, 0, x ) << endl;
}
```

3-8 设 a 和 b 是两个用带有表头结点的循环链表表示的多项式。试编写一个算法，计算这两个多项式的乘积 c = a*b，要求计算后多项式 a 与 b 保持原状。如果这两个多项式的项数分别为 n 与 m，试说明该算法的执行时间为 $O(nm^2)$或 $O(n^2m)$。但若 a 和 b 是稠密的，即其很少有系数为零的项，那么试说明该乘积算法的时间代价为 O(nm)。

【解答】

假设 $a = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n = \sum_{i=0}^{n} a_i x^i$

$b = b_0 + b_1x + b_2x^2 + \cdots + b_{m-1}x^{m-1} + b_mx^m = \sum_{j=0}^{m} b_j x^j$

则它们的乘积为 $c = a \times b = \left(\sum_{i=0}^{n} a_i x^i\right) \times \left(\sum_{j=0}^{m} b_j x^j\right) = \sum_{i=0}^{n} a_i x^i \sum_{j=0}^{m} b_j x^j = \sum_{i=0}^{n} a_i \sum_{j=0}^{m} b_j x^{i+j}$

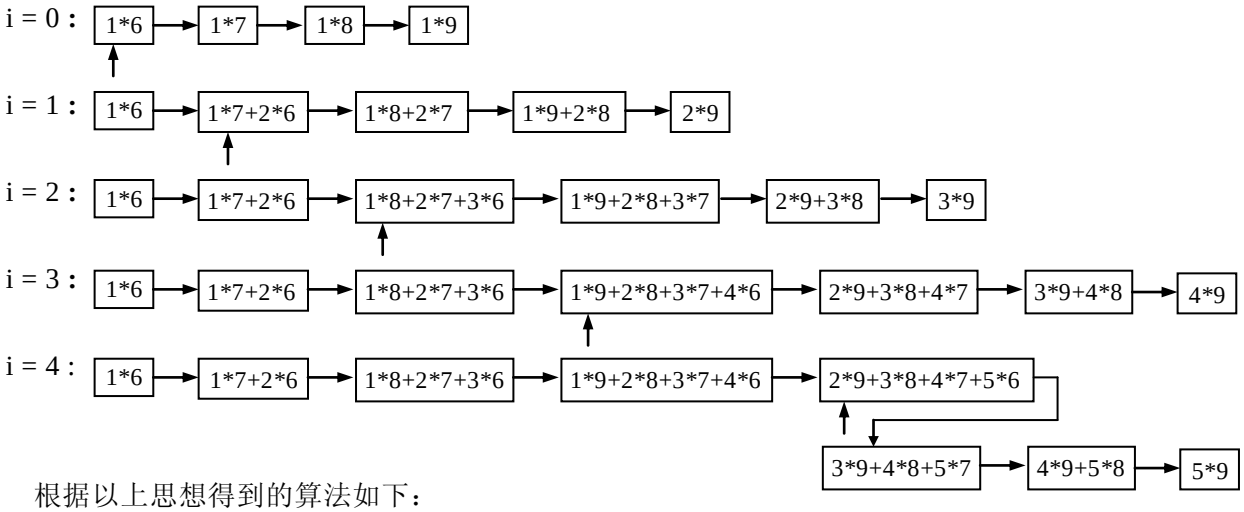
例如，$a = 1 + 2x + 3x^2 + 4x^3 + 5x^4$， $b = 6 + 7x + 8x^2 + 9x^3$，它们的乘积

$c = (1+2x+3x^2+4x^3+5x^4)*(6+7x+8x^2+9x^3) =$

$= 1*6 + (1*7+2*6)x + (1*8+2*7+3*6)x^2 + (1*9+2*8+3*7+4*6)x^3 + (2*9+3*8+4*7+5*6)x^4 +$

$+ (3*9+4*8+5*7)x^5 + (4*9+5*8)x^6 + 5*9x^7$

在求解过程中，固定一个 a_i，用它乘所有 b_j，得到 x^{i+j} 的系数的一部分。这是一个二重循环。



根据以上思想得到的算法如下：

```
Polynomal& Polynomal :: operator * ( Polynomal& a, Polynomal& b ) {
    Term * pa = a.first->link, pb, pc, fc;          //pa 与 pb 是两个多项式链表的检测指针
    first = fc = pc = new Term;                     //fc 是每固定一个 ai 时 ai 结点指针, pc 是存放指针
    while ( pa != NULL ) {                          //每一个 ai 与 b 中所有项分别相乘
       pb = b.first->link;
       while ( pb != NULL ) {                       //扫描多项式 b 所有项
          temp = pa->data * pb->data;               //计算 ai * bj
          if ( pc->link != NULL ) pc->link->data = pc->link->data + temp->      //累加
          else pc->link = new Term (temp);          //增加项, 事实上, 每次 pa 变化, 链结点要随之增加
          pc = pc->link;   pb = pb->link;
       }
       pc = fc = fc->link;   pa = pa->link;         //处理多项式 a 的下一 ai
    }
    pc->link = NULL;
    return *this;
```

```
}
```

这个算法有一个二重循环，内层循环中语句的重复执行次数是 O(n*m)。其中，n 是第一个多项式的阶数，m 是第二个多项式的阶数。这是稠密多项式的情形。

对于稀疏多项式的情形请自行考虑。

3-9 计算多项式 $P_n(x) = a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots\cdots + a_{n-1} x + a_n$ 的值，通常使用的方法是一种嵌套的方法。它可以描述为如下的迭代形式：$b_0 = a_0$， $b_{i+1} = x * b_i + a_{i+1}$， $i = 0, 1, \cdots, n-1$。若设 $b_n = p_n$（x）. 则问题可以写为如下形式：$P_n(x) = x * P_{n-1}(x) + a_n$ ，此处， $P_{n-1}(x) = a_0 x^{n-1} + a_1 x^{n-2} + \cdots\cdots + a_{n-2} x + a_{n-1}$，这是问题的递归形式。试编写一个递归函数，计算这样的多项式的值。

【解答】

如果用循环链表方式存储多项式，求解方法与 3-7 题相同。如果用数组方式存储多项式，当零系数不多时，可用顺序存放各项系数的一维数组存储多项式的信息，指数用数组元素的下标表示

	0	1	2	3		i		n-2	n-1
coef	a_0	a_1	a_2	a_3	…	a_i	…	a_{n-2}	a_{n-1}

多项式的类定义如下：

```
struct Polynomal {
    double * coef;
    int n;
}
```

这样可得多项式的解法：

```
double Polynomal :: Value ( int i, double x ) {
//私有函数：递归求子多项式的值
    if ( i == n-1 ) return coef[n-1];
    else return coef[i] + x * Value ( i+1, x );
}

double Polynomal :: Calc ( double x ) {
//共有函数：递归求多项式的值
    if ( n == 0 ) cout << 0 << endl;
    else cout << Value ( 0, x ) << endl;
}
```

3-10 试设计一个实现下述要求的 Locate 运算的函数。设有一个带表头结点的双向链表 L，每个结点有 4 个数据成员：指向前驱结点的指针 prior、指向后继结点的指针 next、存放数据的成员 data 和访问频度 freq。所有结点的 freq 初始时都为 0。每当在链表上进行一次 Locate (L, x)操作时，令元素值为 x 的结点的访问频度 freq 加 1，并将该结点前移，链接到与它的访问频度相等的结点后面，使得链表中所有结点保持按访问频度递减的顺序排列，以使频繁访问的结点总是靠近表头。

【解答】

```
#include <iostream.h>
//双向循环链表结点的构造函数
DblNode (Type value, DblNode<Type> *left, DblNode<Type> *right ) :
        data ( value ), freq ( 0 ), lLink ( left ), rLink ( right ) { }
DblNode (Type value ) :
        data ( value ), freq ( 0 ), lLink ( NULL ), rLink ( NULL ) { }
```

```
template <class Type>
DblList<Type> :: DblList ( Type uniqueVal ) {
    first = new DblNode<Type>( uniqueVal );
    first->rLink = first->lLink = first;                    //创建表头结点
    current = NULL;
    cout << "开始建立双向循环链表：\n";
    Type value;   cin >> value;
    while ( value != uniqueVal ) {                          //每次新结点插入在表头结点后面
        first->rLink = new DblNode<Type>( value, first, first->rLink );
        cin >> value;
    }
}

template <class Type>
void DblList<Type> :: Locate ( Type & x ) {
//定位
    DblNode<Type> *p = first->rLink;
    while ( p != first && p->data != x ) p = p->rLink;
    if ( p != first ) {                                     //链表中存在 x
        p→freq++;                                           //该结点的访问频度加 1
        current = p;                                        //从链表中摘下这个结点
        current->lLink->rLink = current->rLink;
        current->rLink->lLink = current->lLink;
        p = current->lLink;                                 //寻找从新插入的位置
        while ( p != first && current->freq > p->freq )
            p = p->lLink;
        current->rLink = p->rLink;                          //插入在 p 之后
        current->lLink = p;
        p->rLink->lLink = current;
        p->rLink = current;
    }
    else cout<<"Sorry. Not find!\n";                        //没找到
}
```

3-11 利用双向循环链表的操作改写 2-2 题，解决约瑟夫(Josephus)问题。

【解答】

```
#include <iostream.h>
#include "DblList.h"
Template <class Type> void DblList <Type> :: Josephus ( int n, int m ) {
    DblNode<Type> p = first, temp;
    for ( int i = 0; i < n-1; i++ ) {                       //循环 n-1 趟，让 n-1 个人出列
        for ( int j = 0; j < m-1; j++ ) p = p->rLink;       //让 p 向后移动 m-1 次
        cout << "Delete person " << p->data << endl;
```

```
        p->lLink->rLink = p->rLink;                    //从链中摘下 p
        p->rLink->lLink = p->lLink;
        temp = p->rlink;   delete p;   p = temp;       //删除 p 所指结点后, p 改指下一个出发点
    }
    cout << "The winner is " << p->data << endl;
}

void main ( ) {
    DblList<int> dlist;                                //定义循环链表 dlist 并初始化
    int n, m;                                          //n 是总人数，m 是报数值
    cout << "Enter the Number of Contestants?";
    cin >> n >> m;
    for ( int i = 1; i <= n; i++ ) dlist.insert (i);   //建立数据域为 1, 2, … 的循环链表
    dlist.Josephus (n, m);                             //解决约瑟夫问题，打印胜利者编号
}
```

3-12 试设计一个算法，改造一个带表头结点的双向链表，所有结点的原有次序保持在各个结点的 rLink 域中，并利用 lLink 域把所有结点按照其值从小到大的顺序连接起来。

【解答】

```
template<Type> void DblList<Type> :: sort ( ) {
    DblNode<Type> *
    s = first->link;                                   //指针 s 指向待插入结点, 初始时指向第一个结点
    while ( s != NULL ) {                              //处理所有结点
        pre = first;   p = first->lLink;               //指针 p 指向待比较的结点, pre 是 p 的前驱指针
        while ( p != NULL && s->data < p->data )           //循 lLink 链寻找结点 *s 的插入位置
          { pre = p;   p = p->lLink; }
        pre->lLink = s;   s->lLink = p;                //结点 *s 在 lLink 方向插入到 *pre 与 *p 之间
    }
}
```