数据结构练习题解答（三）

第三章 链表

3-2 试编写一个算法，在带表头结点的单链表中寻找第 *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 0。

【解答】

```
template <class Type>
ListNode <Type> * List <Type> :: GetANode ( int i ) {
//取得单链表中第 i 个结点地址, i 从 0 开始计数, i < 0 时返回指针 0, i = 0 时返回表头结点地址。
    if ( i < 1 ) return NULL;
    ListNode <Type> * p = first;    int k = 0;
    while ( p != NULL && k < i ) { p = p→link;    k++; }
    return p;
}
```

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。

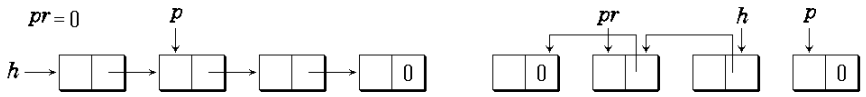
【解答】

```
#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单链表，finished 是停止建表输入的标志，
//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;
        }
    }
```

3-6 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
        }
    }
```

【解答 2】

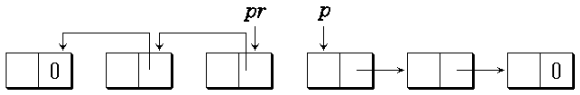
```
    template<class Type> void List<Type> :: Inverse ( ) {
        ListNode<Type> *p, *head = new ListNode<Type> ( );
        while ( first != NULL ) {
            p = first;   first = first→link;          //摘下 first 链头结点
            p→link = head→link;   head→link = p;      //插入 head 链前端
        }
        first = head→link;   delete head;             //重置 first
    }
```

3-7 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
}
```

3-9 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;
}
```

3-12 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
}
```