《数据结构》上机报告

2020 年 10 月 25 日

姓名： 王上游　　学号：1850767　班级：　19 计科 2 班　　得分：

<table>
<tr><td>实验题目</td><td colspan="2">链表综合实验报告
（带头结点单链表、带头结点双向链表、链表实现一元多项式）</td></tr>
<tr><td>实验目的</td><td colspan="2">1 、掌握线性表的链式表示（单链表、循环链表、双向循环链表）；
2 、掌握链表实现线性表的基本操作，如建立、查找、插入、删除、去重、逆置、头部插入、尾部插入、销毁等；
3、 掌握有序线性表的插入、删除、合并操作，及一元多项式的表示、相加和乘法等；</td></tr>
<tr><td>问题描述</td><td colspan="2">(1) 链表的基本操作；
(2) 链表的逆置；
(3) 链表的去重；
(4) 一元多项式的表示、相加和乘法；</td></tr>
<tr><td>基本要求</td><td colspan="2">(1) 程序要添加适当的注释，程序的书写要采用 缩进格式 。
(2) 程序要具在一定的 健壮性，即当输入数据非法时， 程序也能适当地做出反应，如插入删除时指定的位置不对 等等。
(3) 程序要做到界面友好，在程序运行时用户可以根据相应的提示信息进行操作。
(4) 参考实验报告模板详细书写实验报告,在实验报告中给出主要算法的复杂度分析。
(5) 给出逆置算法和去重算法的流程图和复杂度分析。
(6) 给出带头指针和尾指针及长度属性的双向循环链表的存储结构描述。
(7) 总结链表的优缺点。</td></tr>
<tr><td>选做要求</td><td colspan="2">无</td></tr>
<tr><td></td><td>已完成选做内容（序号）</td><td>无</td></tr>
<tr><td>数据结构设计</td><td colspan="2">(1) 带头结点的单链表


```
class LNode {
protected:
    ElemType data;    //数据域
    LNode* next;  //指针域
public:
    friend class LinkList;
    //不定义任何函数，相当于struct LNode
};

class LinkList {
protected:
    LNode* head; //头指针
```
</td></tr>
</table>

```
public:
    /* P.19-20的抽象数据类型定义转换为实际的C++语言 */
    LinkList(); //构造函数，替代InitList
    ~LinkList();    //析构函数，替代DestroyList
    Status ClearList();
    Status ListEmpty();
    int ListLength();
    Status GetElem(int i, ElemType& e);
    int LocateElem(ElemType e, Status(*compare)(ElemType e1, ElemType e2));
    Status PriorElem(ElemType cur_e, ElemType& pre_e);
    Status NextElem(ElemType cur_e, ElemType& next_e);
    Status ListInsert(int i, ElemType e);
    Status ListDelete(int i, ElemType& e);
    Status ListTraverse(Status(*visit)(ElemType e));
    Status ListReverse(const int start, const int end);
    int ListUnique();
};
```

(2) 带头结点的双向链表

```
typedef struct DuLNode {
    ElemType            data;   //存放数据
    struct DuLNode* prior;      //存放直接前驱的指针
    struct DuLNode* next;       //存放直接后继的指针
} DuLNode, * DuLinkList;

Status InitList(DuLinkList* L);
Status DestroyList(DuLinkList* L);
Status ClearList(DuLinkList* L);
Status ListEmpty(DuLinkList L);
int ListLength(DuLinkList L);
Status GetElem(DuLinkList L, int i, ElemType* e);
int LocateElem(DuLinkList L, ElemType e, Status(*compare)(ElemType e1,
ElemType e2));
Status PriorElem(DuLinkList L, ElemType cur_e, ElemType* pre_e);
Status NextElem(DuLinkList L, ElemType cur_e, ElemType* next_e);
Status ListInsert(DuLinkList* L, int i, ElemType e);
Status ListDelete(DuLinkList* L, int i, ElemType* e);
Status ListTraverse(DuLinkList L, Status(*visit)(ElemType e));
```

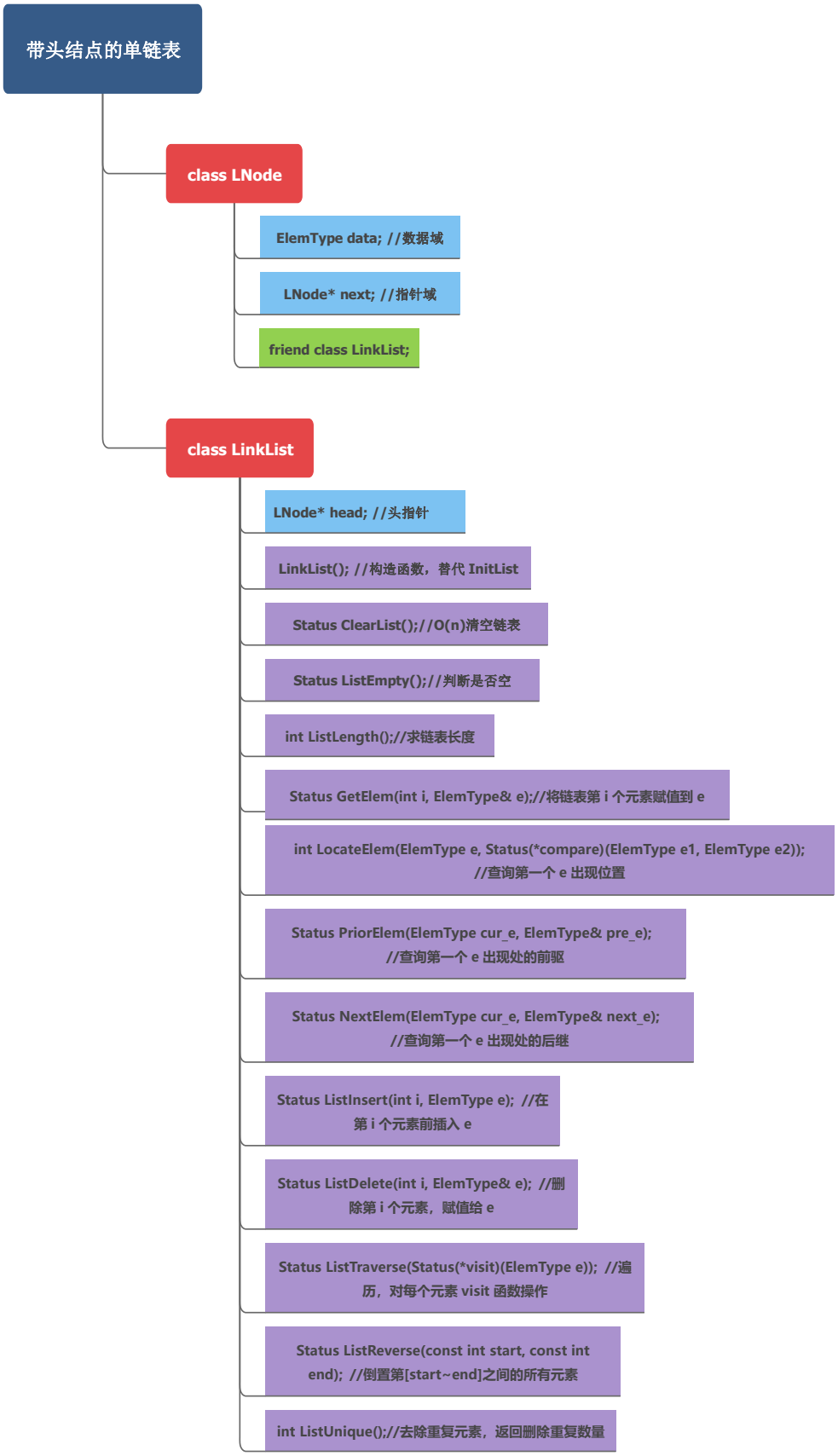
(3) 一元多项式

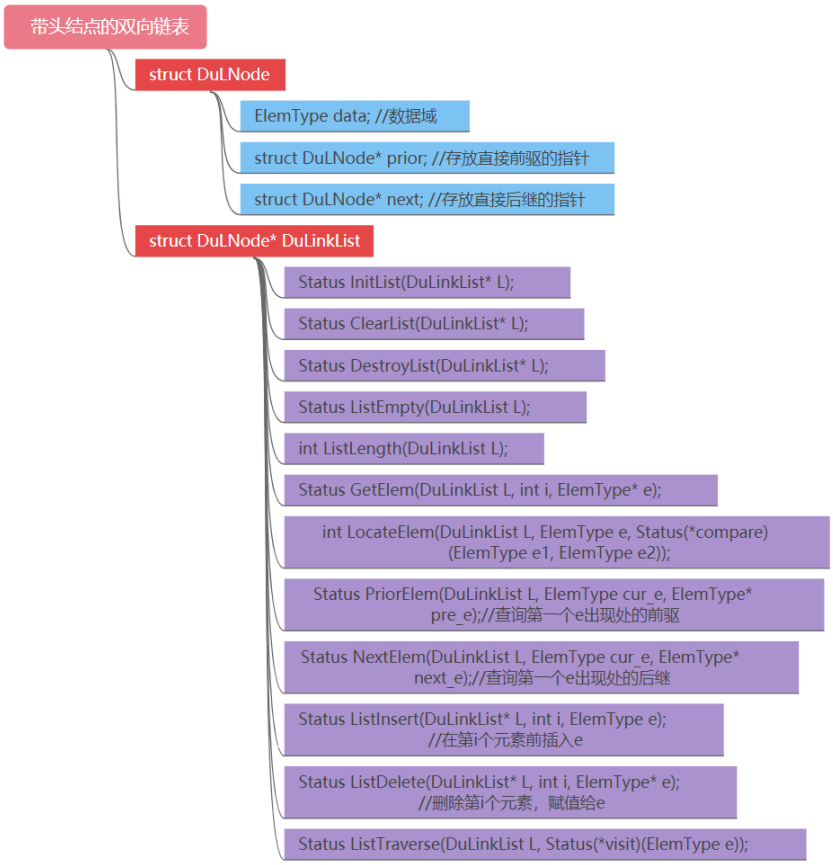
```
class PLNode {
public:
    int coef;//系数
```

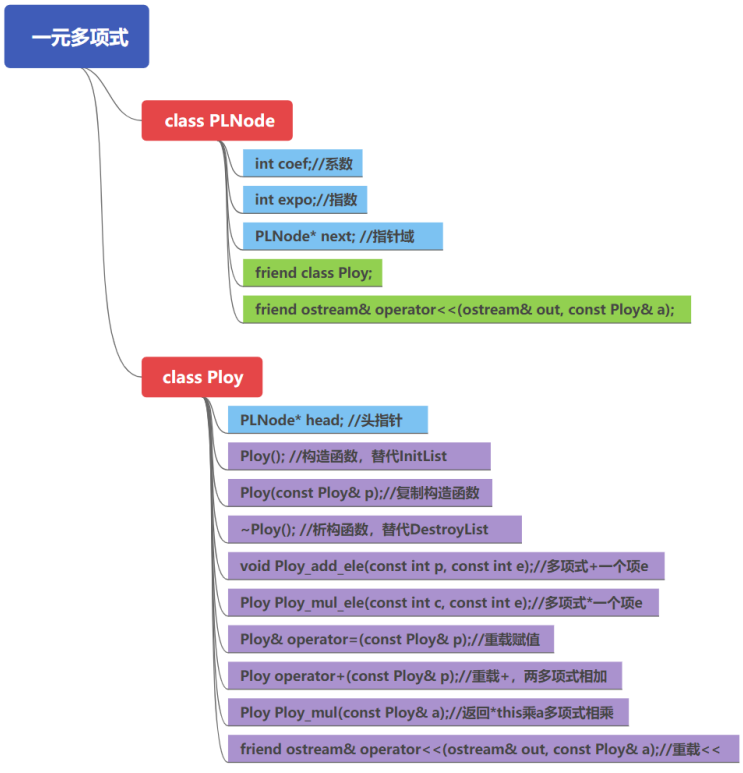
```
    int expo;//指数
    PLNode* next; //指针域
    friend class Ploy;
    friend ostream& operator<<(ostream& out, const Ploy& a);
    //不定义任何函数，相当于struct LNode
};

class Ploy {
protected:
    PLNode* head;     //头指针
public:
    /* P.19-20的抽象数据类型定义转换为实际的C++语言  */
    Ploy(); //构造函数，替代InitList
    Ploy(const Ploy& p);
    ~Ploy();     //析构函数，替代DestroyList
    void Ploy_add_ele(const int p, const int e);
    Ploy Ploy_mul_ele(const int c, const int e);
    Ploy& operator=(const Ploy& p);
    Ploy operator+(const Ploy& p);
    Ploy Ploy_mul(const Ploy& a);
    friend ostream& operator<<(ostream& out, const Ploy& a);
 };
```

功能（函数）说明	对于只有头结点的单链表，头插法插入节点复杂度 O(1)，删除第一个节点复杂度 O(1)。其他例如清空或删除整个链表、指定第几个节点位置（除第一个节点）的插入删除修改、遍历全链表、存在性查找元素、求链表长度、指定区域的逆置等均为 O(n)。头结点的存在使得空链表与非空链表处理一致，也方便对链表的开始结点的插入或删除操作。 对于只有尾节点的单循环链表，由于尾指针 next 指向链表头第一个元素，因此头插和尾插时间均为 O(1)，其他基本操作复杂度同上。 对于双向链表，视其为仅有头节点/仅有尾节点，基本操作复杂度与单链表一致。由于可以在两个方向上循环，因此一些操作比单向链表的算法更简便高效。 对于去重算法，若链表定义为无序链表，即相等元素不必连续，则去重的复杂度为 O(n^2);若链表定义为有序链表，即相等元素必连续，则去重的复杂度为 O(n)。 对于逆置算法，其时间复杂度与 start、end 有关，最坏情况为 O(n)。

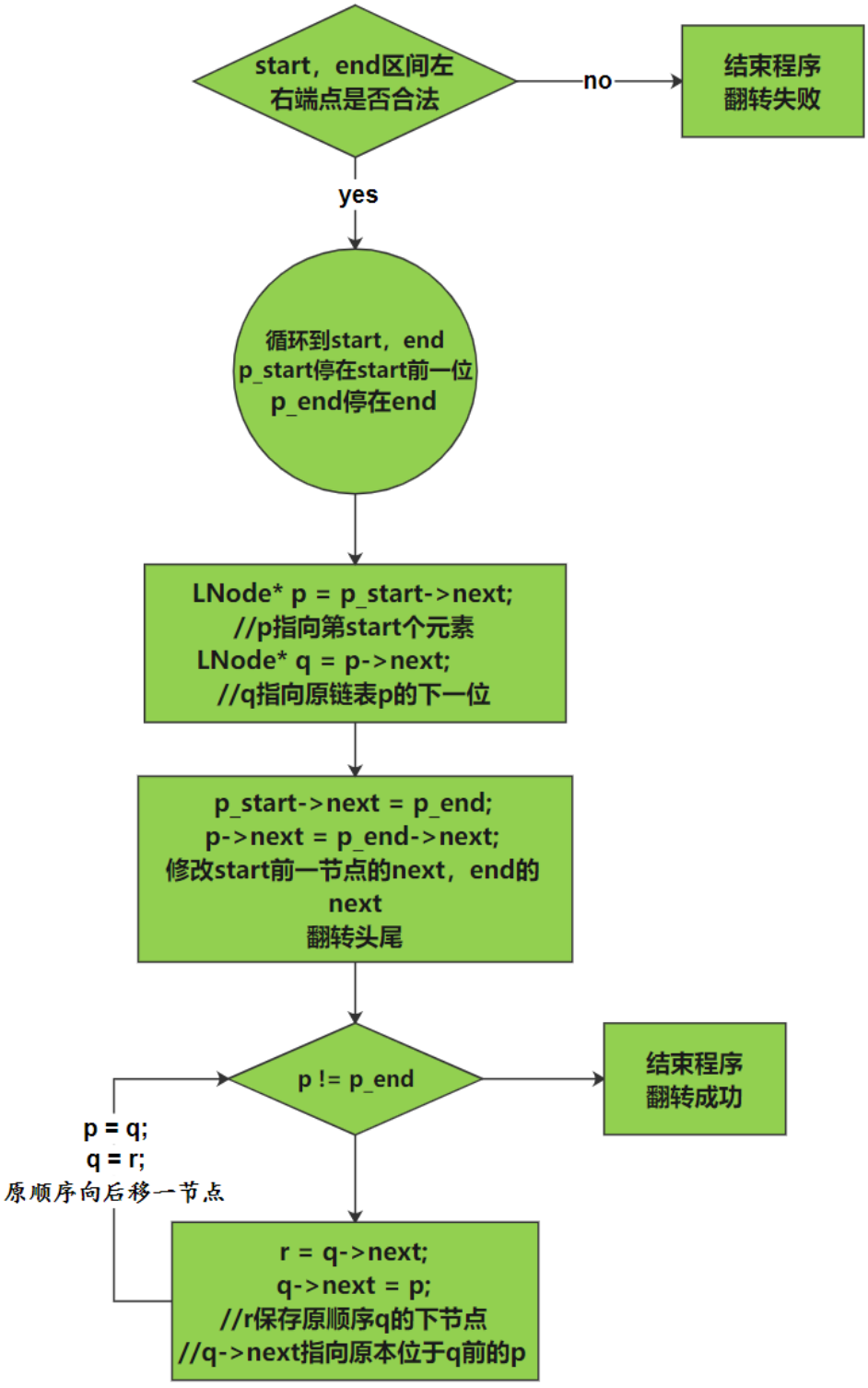
以下是功能函数图谱

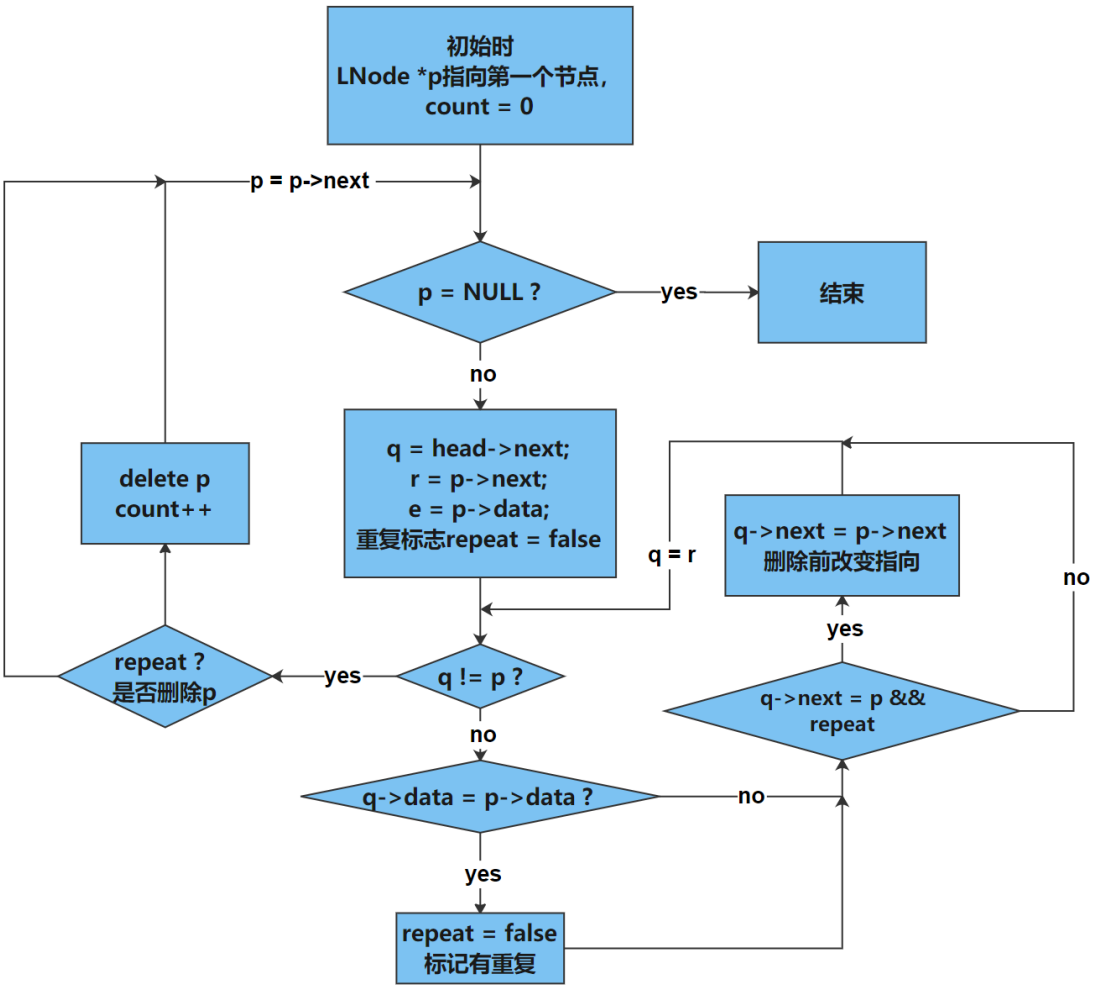
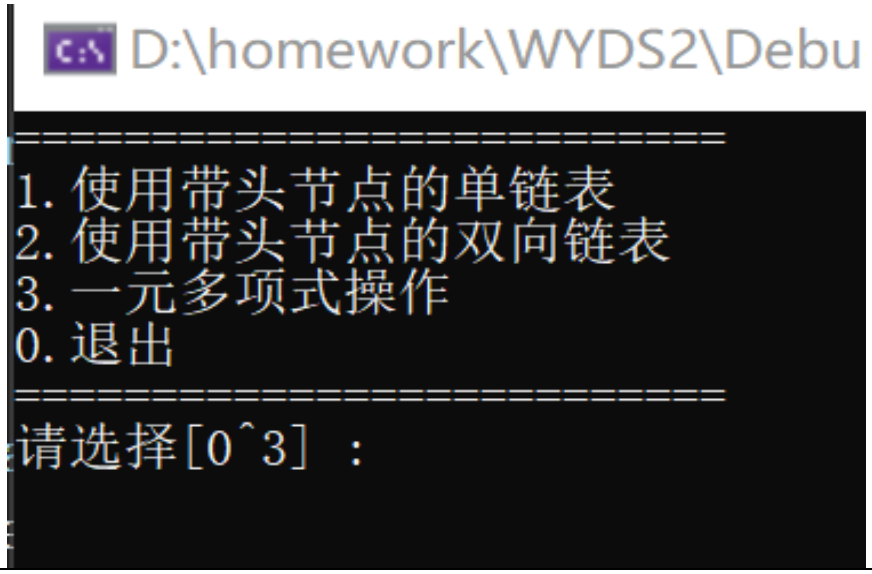

带头结点的单链表
class LNode
ElemType data; //数据域
LNode* next; //指针域
friend class LinkList;
class LinkList
LNode* head; //头指针
LinkList(); //构造函数，替代 InitList
Status ClearList();//O(n)清空链表
Status ListEmpty();//判断是否空
int ListLength();//求链表长度
Status GetElem(int i, ElemType& e);//将链表第 i 个元素赋值到 e
int LocateElem(ElemType e, Status(*compare)(ElemType e1, ElemType e2));
//查询第一个 e 出现位置
Status PriorElem(ElemType cur_e, ElemType& pre_e);
//查询第一个 e 出现处的前驱
Status NextElem(ElemType cur_e, ElemType& next_e);
//查询第一个 e 出现处的后继
Status ListInsert(int i, ElemType e); //在第 i 个元素前插入 e
Status ListDelete(int i, ElemType& e); //删除第 i 个元素，赋值给 e
Status ListTraverse(Status(*visit)(ElemType e)); //遍历，对每个元素 visit 函数操作
Status ListReverse(const int start, const int end); //倒置第[start~end]之间的所有元素
int ListUnique();//去除重复元素，返回删除重复数量

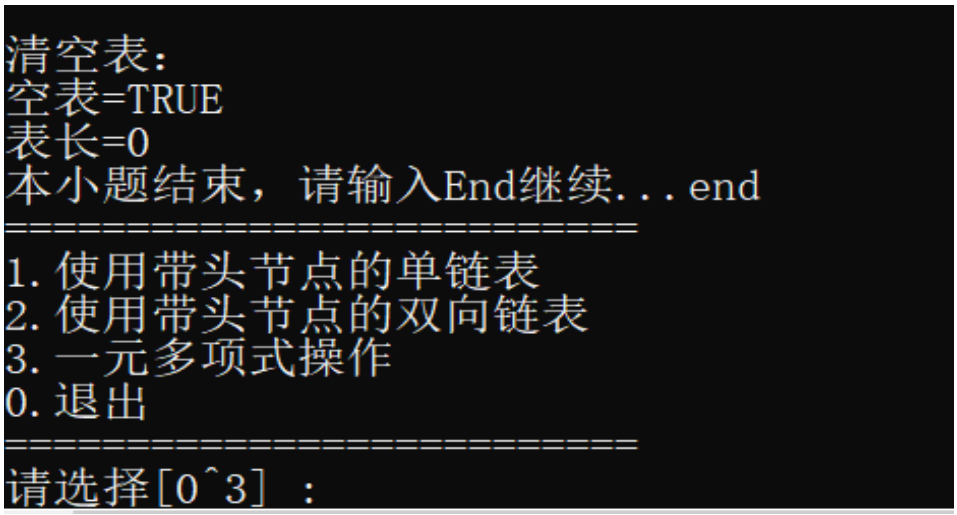
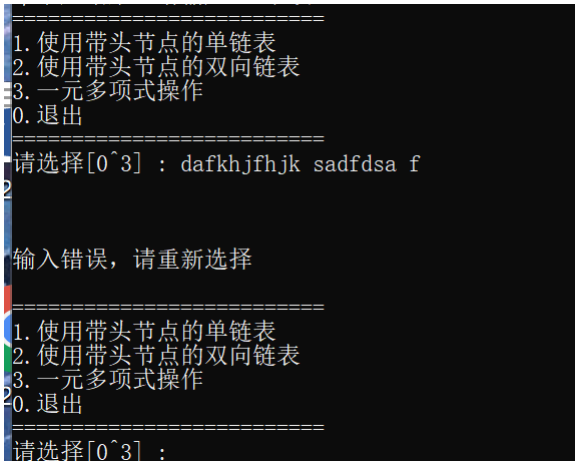
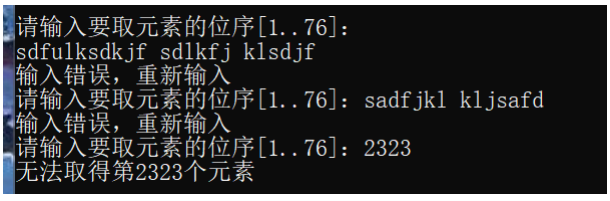
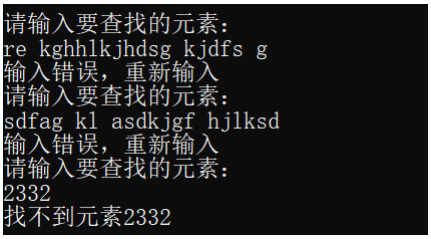
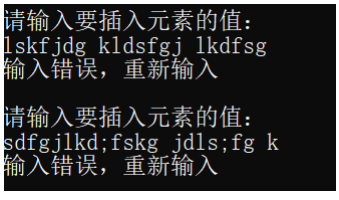
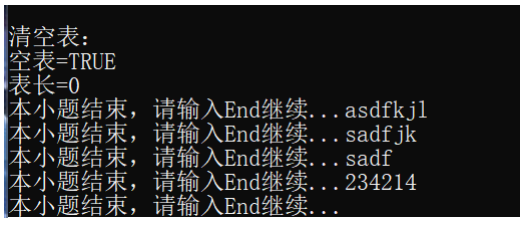

带头结点的双向链表
struct DuLNode
ElemType data; //数据域
struct DuLNode* prior; //存放直接前驱的指针
struct DuLNode* next; //存放直接后继的指针
struct DuLNode* DuLinkList
Status InitList(DuLinkList* L);
Status ClearList(DuLinkList* L);
Status DestroyList(DuLinkList* L);
Status ListEmpty(DuLinkList L);
int ListLength(DuLinkList L);
Status GetElem(DuLinkList L, int i, ElemType* e);
int LocateElem(DuLinkList L, ElemType e, Status(*compare)
(ElemType e1, ElemType e2));
Status PriorElem(DuLinkList L, ElemType cur_e, ElemType*
pre_e);//查询第一个e出现处的前驱
Status NextElem(DuLinkList L, ElemType cur_e, ElemType*
next_e);//查询第一个e出现处的后继
Status ListInsert(DuLinkList* L, int i, ElemType e);
//在第i个元素前插入e
Status ListDelete(DuLinkList* L, int i, ElemType* e);
//删除第i个元素，赋值给e
Status ListTraverse(DuLinkList L, Status(*visit)(ElemType e));


一元多项式
class PLNode
int coef;//系数
int expo;//指数
PLNode* next; //指针域
friend class Ploy;
friend ostream& operator<<(ostream& out, const Ploy& a);
class Ploy
PLNode* head; //头指针
Ploy(); //构造函数，替代InitList
Ploy(const Ploy& p);//复制构造函数
~Ploy(); //析构函数，替代DestroyList
void Ploy_add_ele(const int p, const int e);//多项式+一个项e
Ploy Ploy_mul_ele(const int c, const int e);//多项式*一个项e
Ploy& operator=(const Ploy& p);//重载赋值
Ploy operator+(const Ploy& p);//重载+，两多项式相加
Ploy Ploy_mul(const Ploy& a);//返回*this乘a多项式相乘
friend ostream& operator<<(ostream& out, const Ploy& a);//重载<<

以下是逆置区间[start,end]的流程图，首先判断区间合法性。对于合法的待翻转区间，首先将原第 start-1 节点的 next 指向原第 end 节点，再将原 start 节点的 next 指针指向原第 end+1 个节点，然后从第 start 遍历到第 end，将原(p->next = q)的节点前后关系翻转为(q->next = p)。复杂度 O(n)。



	以下是去重算法的流程图，对于每个节点，遍历其之前所有节点，若有重复，删除该节点。复杂度 $O(n^2)$。 
界面设计和使用说明	进入界面后，首先选择 0，1，2，3，分别对应退出/单链表/双链表/一元多项式演示。 

	按下回车观察链表情况。在需要输入的地方，按照提示输入即可。非输入的链表节点数据，均为随机数生成。一种模式结束，输入 end 可以返回菜单。可以切换其他模式继续测试。 
调试分析	（1） 健壮性测试 测试验证了程序的健壮性，所有输入位置可以处理错误。     

（2） 一般测试

以下是一次对带头结点的单链表的测试

```
1.使用带头节点的单链表
2.使用带头节点的双向链表
3.一元多项式操作
0.退出
==========================
请选择[0^3] : 1空表=TRUE
表长=0
插入115个元素:
 46-> 33-> 79-> 46-> 20-> 41-> 82-> 19->106-> 16->
111-> 45->106-> 17-> 55-> 63->102->113-> 54-> 59->
 39-> 48-> 41-> 51->108->111-> 89-> 70->104-> 52->
 11-> 33-> 96-> 85-> 77->  4-> 36-> 25-> 98-> 42->
 44-> 43-> 73-> 15->  1->107-> 35-> 97-> 88->100->
  4-> 83-> 55-> 24-> 38->113->111->112-> 18->  3->
 44-> 29->104-> 22-> 13-> 18-> 83-> 10->100-> 55->
 51-> 64-> 40-> 58->105-> 41->105-> 14-> 72-> 47->
 33->  1-> 51->101-> 13-> 99-> 21-> 98->  6->103->
 58-> 61->  3-> 88->  6-> 57-> 26->105-> 26-> 43->
 50-> 35->109-> 56-> 18->114->  5-> 98-> 73-> 36->
 50-> 53-> 93-> 97-> 23->
空表=FALSE
表长=115
按下任意键，进行链表去重
去重成功，已删除37个重复元素
 46-> 33-> 79-> 20-> 41-> 82-> 19->106-> 16->111->
 45-> 17-> 55-> 63->102->113-> 54-> 59-> 39-> 48->
 51->108-> 89-> 70->104-> 52-> 11-> 96-> 85-> 77->
  4-> 36-> 25-> 98-> 42-> 44-> 43-> 73-> 15->  1->
107-> 35-> 97-> 88->100-> 83-> 24-> 38->112-> 18->
  3-> 29-> 22-> 13-> 10-> 64-> 40-> 58->105-> 14->
 72-> 47->101-> 99-> 21->  6->103-> 61-> 57-> 26->
 50->109-> 56->114->  5-> 53-> 93-> 23->
按下任意键，演示倒置链表第[34-73]元素
 46-> 33-> 79-> 20-> 41-> 82-> 19->106-> 16->111->
 45-> 17-> 55-> 63->102->113-> 54-> 59-> 39-> 48->
 51->108-> 89-> 70->104-> 52-> 11-> 96-> 85-> 77->
  4-> 36-> 25-> 56->109-> 50-> 26-> 57-> 61->103->
  6-> 21-> 99->101-> 47-> 72-> 14->105-> 58-> 40->
 64-> 10-> 13-> 22-> 29->  3-> 18->112-> 38-> 24->
 83->100-> 88-> 97-> 35->107->  1-> 15-> 73-> 43->
 44-> 42-> 98->114->  5-> 53-> 93-> 23->
第1个元素=46
无法取得第1个元素(46)的前驱
第1个元素(46)的后继=33

第78个元素=23
第78个元素(23)的前驱=93
无法取得第78个元素(23)的后继

无法取得第-1个元素

无法取得第79个元素
```

```
请输入要取元素的位序[1..78]:
45
第45个元素=47
第45个元素(47)的前驱=101
第45个元素(47)的后继=72

请输入要查找的元素:
21
元素21的位序=42

请输入要插入元素的值:
34
请输入要插入元素的位序:
67

请输入要插入元素的值:
在67前插入元素34成功
新表为:
 46-> 33-> 79-> 20-> 41-> 82-> 19->106-> 16->111->
 45-> 17-> 55-> 63->102->113-> 54-> 59-> 39-> 48->
 51->108-> 89-> 70->104-> 52-> 11-> 96-> 85-> 77->
  4-> 36-> 25-> 56->109-> 50-> 26-> 57-> 61->103->
  6-> 21-> 99->101-> 47-> 72-> 14->105-> 58-> 40->
 64-> 10-> 13-> 22-> 29->  3-> 18->112-> 38-> 24->
 83->100-> 88-> 97-> 35->107-> 34->  1-> 15-> 73->
 43-> 44-> 42-> 98->114->  5-> 53-> 93-> 23->

请输入要删除元素的位序:
43
删除第43元素=99成功
新表为:
 46-> 33-> 79-> 20-> 41-> 82-> 19->106-> 16->111->
 45-> 17-> 55-> 63->102->113-> 54-> 59-> 39-> 48->
 51->108-> 89-> 70->104-> 52-> 11-> 96-> 85-> 77->
  4-> 36-> 25-> 56->109-> 50-> 26-> 57-> 61->103->
  6-> 21->101-> 47-> 72-> 14->105-> 58-> 40-> 64->
 10-> 13-> 22-> 29->  3-> 18->112-> 38-> 24-> 83->
100-> 88-> 97-> 35->107-> 34->  1-> 15-> 73-> 43->
 44-> 42-> 98->114->  5-> 53-> 93-> 23->

清空表:
空表=TRUE
表长=0
本小题结束，请输入End继续...
```

以下是一元多项式相加相乘的演示

```
==========================
1.使用带头节点的单链表
2.使用带头节点的双向链表
3.一元多项式操作
0.退出
==========================
请选择[0^3] : 3随机生成的一元多项式 PA = (13^2)+(13^3)+(16^5)+(16^8)+(17^9)+(9^10)+(3^13)+(16^15)
随机生成的一元多项式 PB = (13^8)+(12^9)+(6^12)+(4^13)+(29^14)+(16^17)
演示多项式相加PA+PB的结果 = (13^2)+(13^3)+(16^5)+(29^8)+(29^9)+(9^10)+(6^12)+(7^13)+(29^14)+(16^15)+(16^17)
演示多项式相乘PA*PB的结果 = (169^10)+(325^11)+(156^12)+(208^13)+(270^14)+(130^15)+(637^16)+(886^17)+(385^18)+(780^19)+(304^20)+(20
64^29)+(48^30)+(256^32)
本小题结束，请输入End继续...
```

以下是一次对带头结点的双链表的测试

```
请选择[0~3] : 2空表=TRUE
表长=0

插入115个元素：
 20->  8-> 14-> 47-> 50-> 49-> 66-> 99-> 83-> 94->
 44-> 25->110-> 31-> 82-> 11-> 46->103-> 92-> 80->
 29-> 68-> 84-> 26-> 13-> 14-> 25-> 17-> 30-> 65->
 21->100-> 49-> 11-> 88->109-> 63->108->114-> 29->
 26-> 89->  6-> 24-> 68-> 29->106-> 91-> 78->  7->
  5-> 93->106-> 41->102-> 22-> 64-> 68->110-> 48->
 60->112-> 91->106-> 95-> 90-> 19-> 91-> 35-> 61->
 84-> 88->108->100->101->  5->106-> 13-> 35-> 40->
 89->102->101-> 29-> 35-> 12->107->  9-> 57-> 40->
 89-> 96-> 83-> 95-> 63-> 98-> 25-> 48-> 98-> 42->
 85-> 26-> 30-> 32-> 54-> 42-> 48-> 50-> 24-> 83->
102-> 85->112-> 65-> 15->
空表=FALSE
表长=115

第1个元素=20
无法取得第1个元素(20)的前驱
第1个元素(20)的后继=8

第115个元素=15
第115个元素(15)的前驱=65
无法取得第115个元素(15)的后继

无法取得第-1个元素

无法取得第116个元素

请输入要取元素的位序[1..115]：45
第45个元素=68
第45个元素(68)的前驱=29
第45个元素(68)的后继=84

请输入要查找的元素：
34
找不到元素34

请输入要插入元素的值：
34
请输入要插入元素的位序：
23
在23前插入元素34成功
新表为：
 20->  8-> 14-> 47-> 50-> 49-> 66-> 99-> 83-> 94->
 44-> 25->110-> 31-> 82-> 11-> 46->103-> 92-> 80->
 29-> 68-> 34-> 84-> 26-> 13-> 14-> 25-> 17-> 30->
 65-> 21->100-> 49-> 11-> 88->109-> 63->108->114->
 29-> 26-> 89->  6-> 24-> 68-> 29->106-> 91-> 78->
  7->  5-> 93->106-> 41->102-> 22-> 64-> 68->110->
 48-> 60->112-> 91->106-> 95-> 90-> 19-> 91-> 35->
 61-> 84-> 88->108->100->101->  5->106-> 13-> 35->
 40-> 89->102->101-> 29-> 35-> 12->107->  9-> 57->
 40-> 89-> 96-> 83-> 95-> 63-> 98-> 25-> 48-> 98->
 42-> 85-> 26-> 30-> 32-> 54-> 42-> 48-> 50-> 24->
 83->102-> 85->112-> 65-> 15->

请输入要删除元素的位序：
56
删除第56个元素=102成功
新表为：
 20->  8-> 14-> 47-> 50-> 49-> 66-> 99-> 83-> 94->
 44-> 25->110-> 31-> 82-> 11-> 46->103-> 92-> 80->
 29-> 68-> 34-> 84-> 26-> 13-> 14-> 25-> 17-> 30->
 65-> 21->100-> 49-> 11-> 88->109-> 63->108->114->
 29-> 26-> 89->  6-> 24-> 68-> 29->106-> 91-> 78->
  7->  5-> 93->106-> 41-> 22-> 64-> 68->110-> 48->
 60->112-> 91->106-> 95-> 90-> 19-> 91-> 35-> 61->
 84-> 88->108->100->101->  5->106-> 13-> 35-> 40->
 89->102->101-> 29-> 35-> 12->107->  9-> 57-> 40->
 89-> 96-> 83-> 95-> 63-> 98-> 25-> 48-> 98-> 42->
 85-> 26-> 30-> 32-> 54-> 42-> 48-> 50-> 24-> 83->
102-> 85->112-> 65-> 15->

清空表：
空表=TRUE
表长=0
本小题结束，请输入End继续...
```

心得体会	（1） 复杂度分析 对于只有头结点的单链表，头插法插入节点复杂度 O(1)，删除第一个节点复杂度 O(1)。其他例如清空或删除整个链表、指定第几个节点位置（除第一个节点）的插入删除修改、遍历全链表、存在性查找元素、求链表长度、指定区域的逆置等均为 O(n)。头结点的存在使得空链表与非空链表处理一致，也方便对链表的开始结点的插入或删除操作。 对于只有尾节点的单循环链表，由于尾指针 next 指向链表头第一个元素，因此头插和尾插时间均为 O(1)，其他基本操作复杂度同上。 对于双向链表，视其为仅有头节点/仅有尾节点，基本操作复杂度与单链表一致。由于可以在两个方向上循环，因此一些操作比单向链表的算法更简便高效。 对于去重算法，若链表定义为无序链表，即相等元素不必连续，则去重的复杂度为 $O(n^2)$;若链表定义为有序链表，即相等元素必连续，则去重的复杂度为 O(n)。 （2） 链表的优缺点 由于链表内存的动态分配，物理上不连续性，链表的动态操作（插入删除等）相比线性表需要整体移动的数组时间降为线性的，而随机访问查找指定位置（诸如第几个）元素的时间复杂度，线性表为 O(1)，链表为 O(n)。因此链表适合频繁插入删除、但较少查找的需求。线性表适合需要频繁查询、但较少插入删除的需求。 由于指针域的存在，存储相同数据链表需要消耗比线性表更多的空间。但链表对空间的利用更为合理，可以通过删除多余节点，避免空间的闲置。 头结点的优点是可以统一链表的各项操作在链表元素为空或非空的情况。
代码实现	``` #define _CRT_SECURE_NO_WARNINGS #include <iostream> #include <cstdio> #include <conio.h> #include <iomanip> #include <cstring> #include <cstdlib> using namespace std; #ifdef _MSC_VER #pragma warning(disable:6031) #endif #define updown 1 #define leftright 0 /* P.10 的预定义常量和类型 */ #define TRUE 1 #define FALSE 0 #define OK 1 #define ERROR 0 ```

```
#define INFEASIBLE   -1
#define MYOVERFLOW    -2

#define INSERT_NUM       115      //初始插入表中的元素数量
#define MAX_NUM_PER_LINE     10       //每行最多输出的元素个数

typedef int Status;
typedef int ElemType;

int line_count = 0;     //打印链表时的计数器

/*************************************************************************
  函数名称：
  功      能：
  输入参数：
  返 回 值：
  说      明：
*************************************************************************/
void wait_for_end(void)
{
    const char* prompt = "本小题结束，请输入End继续...";
    char input_str[1024];

    while (1) {
        cout << prompt;
        cin >> input_str;

        /* 清空缓冲区 */
        cin.clear();
        cin.ignore(1024, '\n');
        //          fgets(input_str, sizeof(input_str), stdin);

                /* 输入在合理范围内则退出输入循环 */
        if (_strcmpi(input_str, "end") == 0)
            break;
    }

    return;
}

/*************************************************************************
  函数名称：
  功      能：
  输入参数：dis = 1 不区分大小写，且若输入小写，返回值为大写
```

```
  返 回 值：
  说     明：目前此函数仅适用于game1，需要改成同时适应game1/game2
            返回类型不允许修改
            参数个数、类型可以修改
*****************************************************************************/
const char simple_menu(const char* menu[], char legal_menu_input[], const char*
menu_input_reminder, int dis = 1, const char* menu_title = NULL, int order =
updown, int col = 1, int colwidth = 28, char frameicon = '=', int framelength = 26)
{
    int i;
    char sel;
    for (i = 0; menu[i]; i++)
        ;
    int menu_count = i;
    int row = menu_count;
    while (1)
    {
        if (col >= 2)
        {
            row = menu_count / col;
            if (menu_count % col)
                row++;
        }
        if (menu_title != NULL)
            cout << menu_title << endl;
        for (int i = 1; i <= framelength; i++)
            cout << frameicon;
        cout << endl;
        if (col >= 2)
        {
            for (int j = 0; j < row; j++)
            {
                for (int k = 0; k < col; k++)
                {
                    if (order == leftright && ((j * col + k) < menu_count))
                    {
                        cout << setw(colwidth) << setiosflags(ios::left) << menu[j * col
+ k];
                    }
                    else if (order == updown && ((k * row + j) < menu_count))
                    {
                        cout << setw(colwidth) << setiosflags(ios::left) << menu[k *
row + j];
                    }
```

```
                }
                cout << endl;
            }
        }
        else if (col == 1)
        {
            for (i = 0; menu[i]; i++)
                cout << menu[i] << endl;
        }
        for (int i = 1; i <= framelength; i++)
            cout << frameicon;
        cout << endl;
        cout << menu_input_reminder;

        sel = _getch();
        cout << sel;
        if (_getch() == '\r' && legal_menu_input[sel])
            break;

        cin.ignore(0x7fffffff, '\n');
        cout << endl;

        if (dis && sel >= 'a' && sel <= 'z' && legal_menu_input[sel - 32])
        {
            sel -= 32;
            break;
        }
        /* 选择错误则给出提示并再次打印菜单（此处不用优化） */
        cout << endl << endl << "输入错误，请重新选择" << endl << endl;
    }

    return sel;
}

class LinkList;//提前声明，因为定义友元要用到

class LNode {
protected:
    ElemType data;    //数据域
    LNode* next;  //指针域
public:
    friend class LinkList;
    //不定义任何函数，相当于struct LNode
```

```
};

class LinkList {
protected:
    LNode* head;  //头指针
public:
    /* P.19-20的抽象数据类型定义转换为实际的C++语言 */
    LinkList(); //构造函数，替代InitList
    ~LinkList();    //析构函数，替代DestroyList
    Status  ClearList();
    Status  ListEmpty();
    int  ListLength();
    Status  GetElem(int i, ElemType& e);
    int  LocateElem(ElemType e, Status(*compare)(ElemType e1, ElemType e2));
    Status  PriorElem(ElemType cur_e, ElemType& pre_e);
    Status  NextElem(ElemType cur_e, ElemType& next_e);
    Status  ListInsert(int i, ElemType e);
    Status  ListDelete(int i, ElemType& e);
    Status  ListTraverse(Status(*visit)(ElemType e));
    Status ListReverse(const int start, const int end);
    int ListUnique();
};

/* 构造函数（初始化线性表） */
LinkList::LinkList()
{
    /* 申请头结点空间，赋值给头指针 */
    head = new LNode;
    if (head == NULL)
        exit(OVERFLOW);

    head->next = NULL;
}

/* 析构函数（删除线性表） */
LinkList::~LinkList()
{
    LNode* q, * p = head;

    /* 从头结点开始依次释放（含头结点） */
    while (p) {     //若链表为空，则循环不执行
        q = p->next; //抓住链表的下一个结点
        delete p;
        p = q;
```

```
    }

    head = NULL;  //头指针置NULL
}

/* 清除线性表（保留头结点） */
Status LinkList::ClearList()
{
    LNode* q, * p = head->next;

    /* 从首元结点开始依次释放 */
    while (p) {
        q = p->next;    //抓住链表的下一个结点
        delete p;
        p = q;
    }

    head->next = NULL; //头结点的next域置NULL
    return OK;
}

/* 判断是否为空表 */
Status LinkList::ListEmpty()
{
    /* 判断头结点的next域即可 */
    if (head->next == NULL)
        return TRUE;
    else
        return FALSE;
}

/* 求表的长度 */
int LinkList::ListLength()
{
    LNode* p = head->next; //指向首元结点
    int      len = 0;

    /* 循环整个链表，进行计数 */
    while (p) {
        p = p->next;
        len++;
    }

    return len;
```

```
}

/* 取表中元素 */
Status LinkList::GetElem(int i, ElemType& e)
{
    LNode* p = head->next;   //指向首元结点
    int   pos = 1;        //初始位置为1

    /* 链表不为NULL 且 未到第i个元素 */
    while (p != NULL && pos < i) {
        p = p->next;
        pos++;
    }

    if (!p || pos > i)
        return ERROR;

    e = p->data;
    return OK;
}

/* 查找符合指定条件的元素 */
int LinkList::LocateElem(ElemType e, Status(*compare)(ElemType e1, ElemType
e2))
{
    LNode* p = head->next;   //指向首元结点
    int   pos = 1;        //初始位置为1

    /* 循环整个链表  */
    while (p && (*compare)(e, p->data) == FALSE) {
        p = p->next;
        pos++;
    }

    return p ? pos : 0;
}

/* 查找符合指定条件的元素的前驱元素 */
Status LinkList::PriorElem(ElemType cur_e, ElemType& pre_e)
{
#if 1
    LNode* p = head->next;   //指向首元结点

    if (p == NULL) //空表直接返回
```

```
        return ERROR;

    /* 从第2个结点开始循环整个链表(如果比较相等，保证有前驱) */
    while (p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next == NULL) //未找到
        return ERROR;

    pre_e = p->data;
    return OK;
#else
    LNode* p = head; //指向头结点

    /* 循环整个链表并比较值是否相等 */
    while (p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next == NULL || p == head) //未找到或首元结点或空表
        return ERROR;

    pre_e = p->data;
    return OK;
#endif
}

/* 查找符合指定条件的元素的后继元素 */
Status LinkList::NextElem(ElemType cur_e, ElemType& next_e)
{
    LNode* p = head->next;   //首元结点

    if (p == NULL) //空表直接返回
        return ERROR;

    /* 有后继结点且当前结点值不等时继续 */
    while (p->next && p->data != cur_e)
        p = p->next;

    if (p->next == NULL)
        return ERROR;

    next_e = p->next->data;
    return OK;
}
```

```
/* 在指定位置前插入一个新元素 */
Status LinkList::ListInsert(int i, ElemType e)
{
    LNode* s, * p = head;  //p指向头结点
    int    pos = 0;

    /* 寻找第i-1个结点 */
    while (p && pos < i - 1) {
        p = p->next;
        pos++;
    }

    if (p == NULL || pos > i - 1)   //i值非法则返回
        return ERROR;

    //执行到此表示找到指定位置，p指向第i-1个结点

    s = new LNode;
    if (s == NULL)
        return OVERFLOW;

    s->data = e;    //新结点数据域赋值
    s->next = p->next;//新结点的next是第i个
    p->next = s;    //第i-1个的next是新结点

    return OK;
}

/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status LinkList::ListDelete(int i, ElemType& e)
{
    LNode* q, * p = head;  //p指向头结点
    int    pos = 0;

    /* 寻找第i个结点（p->next是第i个结点） */
    while (p->next && pos < i - 1) {
        p = p->next;
        pos++;
    }

    if (p->next == NULL || pos > i - 1) //i值非法则返回
        return ERROR;
```

```
    //执行到此表示找到了第i个结点，此时p指向第i-1个结点

    q = p->next;          //q指向第i个结点
    p->next = q->next; //第i-1个结点的next域指向第i+1个

    e = q->data;         //取第i个结点的值
    delete q;              //释放第i个结点

    return OK;
}

/* 遍历线性表 */
Status LinkList::ListTraverse(Status(*visit)(ElemType e))
{
    line_count = 0;
    LNode* p = head->next;

    line_count = 0;          //计数器恢复初始值（与算法无关）
    while (p && (*visit)(p->data) == TRUE)
        p = p->next;

    if (p)
        return ERROR;

    cout << endl;  //最后打印一个换行，只是为了好看，与算法无关
    return OK;
}

/* 逆置链表中[strat,end]的元素*/
Status LinkList::ListReverse(const int start, const int end)
{
    const int len = ListLength();
    if (start <= 0 || end <= 0 || start > len || end > len || start >= end)
        return ERROR;

    LNode* p_start = head;
    LNode* p_end = head;
    int pos_start = 0;
    int pos_end = 0;

    while (pos_start < start - 1)//循环结束时，p_start停在start前一位
    {
        p_start = p_start->next;
        pos_start++;
```

```
    }

    while (pos_end < end)//循环结束时，p_end停在end
    {
        p_end = p_end->next;
        pos_end++;
    }

    LNode* p = p_start->next;//p指向第start个元素
    LNode* q = p->next;//q指向原链表p的下一位
    LNode* r;
    p_start->next = p_end;
    p->next = p_end->next;
    while (p != p_end)
    {
        r = q->next;
        q->next = p;
        p = q;
        q = r;
    }
    return OK;
}

/* 链表元素去重复 */
int LinkList::ListUnique()
{
    int count = 0;
    LNode* p = head->next;//p指向第一个节点
    while (p)//外层循环，遍历所有节点
    {
        LNode* q = head->next;//q从第一个节点开始
        LNode* r = p->next;//若有重复，p被删，提前留好后继方便循环
        int e = p->data;//p的数值
        bool repeat = false;//标记p之前是否有和p一样的，无罪推定
        while (q != p)//内层循环，q从第一个节点遍历到p之前，找p有没有和p一样的
        {
            if (q->data == e)//发现重复
            {
                repeat = true;
            }
            if (q->next == p && repeat)//q移动到p之前了，检查完了，如果p前有重复项，
删除p之前移动这里指针
            {
                q->next = p->next;
```

```
                    break;//不写的话下一步就跳不出去了
                }
                q = q->next;//q向后移动
            }
            if (repeat)//有重复，删p
            {
                delete p;
                count++;
            }
            p = r;//p移到下一节点
        }
        return count;
}

/* 用于比较两个值是否相等的具体函数，与LocateElem中的函数指针定义相同，调用时代入
    int LocateElem(sqlist L, ElemType e, Status (*compare)(ElemType e1, ElemType e2)) */
Status MyCompare(ElemType e1, ElemType e2)
{
    if (e1 == e2)
        return TRUE;
    else
        return FALSE;
}

/* 用于访问某个元素的值的具体函数，与ListTraverse中的函数指针定义相同，调用时代入
    Status ListTraverse(sqlist L, Status (*visit)(ElemType e)) */
Status MyVisit(ElemType e)
{
    cout << setw(3) << e << "->";

    /* 每输出MAX_NUM_PER_LINE个，打印一个换行 */
    if ((++line_count) % MAX_NUM_PER_LINE == 0)
        cout << endl;

    return OK;
}

void LinkList_display()
{
    LinkList L;
    ElemType e1, e2;
```

```
    int         i, pos;

    cout << "空表=" << ((L.ListEmpty() == TRUE) ? "TRUE" : "FALSE") << endl;
    cout << "表长=" << L.ListLength() << endl;

    cout << "插入" << INSERT_NUM << "个元素：" << endl;
    for (i = INSERT_NUM; i > 0; i--)
        L.ListInsert(1, rand() % INSERT_NUM);
    L.ListTraverse(MyVisit);//此处传入MyVisit函数名

    cout << "空表=" << ((L.ListEmpty() == TRUE) ? "TRUE" : "FALSE") << endl;
    cout << "表长=" << L.ListLength() << endl;

    cout << "按下任意键，进行链表去重" << endl;
    while (!_kbhit())
        ;

    cout << "去重成功，已删除" << L.ListUnique() << "个重复元素" << endl;
    L.ListTraverse(MyVisit);//此处传入MyVisit函数名
    int start = rand() % L.ListLength() + 1;
    int end = rand() % L.ListLength() + 1;
    if (end < start)
        start ^= end ^= start ^= end;

    cout << "按下任意键，演示倒置链表第[" << start << "-" << end << "]元素" << endl;
    while (!_kbhit())
        ;
    L.ListReverse(start, end);
    L.ListTraverse(MyVisit);//此处传入MyVisit函数名
    /*  分别取第1、最后、以及小于第1、大于最后的4种情况下的的元素值、前驱值、后继值
        最后再加一个任意输入值  */
    for (i = 0; i < 5; i++) {
        int pos;
        switch (i) {
            case 0:
                pos = 1;
                break;
            case 1:
                pos = L.ListLength();
                break;
            case 2:
                pos = -1;
                break;
```

```
            case 3:
                pos = L.ListLength() + 1;
                break;
            case 4:
                while (1)
                {
                    printf("请输入要取元素的位序[1..%d]： ", L.ListLength());
                    if (!(cin >> pos))
                    {
                        cin.clear();
                        cin.ignore(0x7fffffff, '\n');
                        cout << "输入错误，重新输入" << endl;
                    }
                    else
                        break;
                }
                break;
        }

        if (L.GetElem(pos, e1) == OK) {
            cout << "第" << pos << "个元素=" << e1 << endl;

            /* 只有取得了某个元素，才能取前驱和后继 */
            if (L.PriorElem(e1, e2) == OK)
                cout << "第" << pos << "个元素(" << e1 << ")的前驱=" << e2 << endl;
            else
                cout << "无法取得第" << pos << "个元素(" << e1 << ")的前驱" << endl;

            if (L.NextElem(e1, e2) == OK)
                cout << "第" << pos << "个元素(" << e1 << ")的后继=" << e2 << endl <<
endl;
            else
                cout << "无法取得第" << pos << "个元素(" << e1 << ")的后继" << endl
<< endl;
        }
        else
            cout << "无法取得第" << pos << "个元素" << endl << endl;
    } // end of for

    while (1)
    {
        cout << "请输入要查找的元素： " << endl;
        if (!(cin >> e1))
        {
```

```
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
        else
            break;
    }
    if ((pos = L.LocateElem(e1, MyCompare)) > 0)
        cout << "元素" << e1 << "的位序=" << pos << endl;
    else
        cout << "找不到元素" << e1 << endl;

    while (1)
    {
        cout << endl << "请输入要插入元素的值：" << endl;
        if (!(cin >> e1))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
        else
            break;
    }

    while (1)
    {
        cout << "请输入要插入元素的位序：" << endl;
        if (!(cin >> pos))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
        else
            break;
    }

    cout << endl << "请输入要插入元素的值：" << endl;
    if (L.ListInsert(pos, e1) == OK) {
        cout << "在" << pos << "前插入元素" << e1 << "成功" << endl;
        cout << "新表为：" << endl;
        L.ListTraverse(MyVisit);
    }
```

```
    else
        cout << "在" << pos << "前插入元素" << e1 << "失败" << endl;

    while (1)
    {
        cout << endl << "请输入要删除元素的位序：" << endl;
        if (!(cin >> pos))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
        else
            break;
    }

    if (L.ListDelete(pos, e1) == OK) {
        cout << "删除第" << pos << "元素=" << e1 << "成功" << endl;
        cout << "新表为：" << endl;
        L.ListTraverse(MyVisit);
    }
    else
        cout << "删除第" << pos << "元素=" << e1 << "失败" << endl;

    cout << endl << "清空表：" << endl;
    L.ClearList();
    cout << "空表=" << ((L.ListEmpty() == TRUE) ? "TRUE" : "FALSE") << endl;
    cout << "表长=" << L.ListLength() << endl;

    return;
}

typedef struct DuLNode {
    ElemType          data;    //存放数据
    struct DuLNode* prior;     //存放直接前驱的指针
    struct DuLNode* next;      //存放直接后继的指针
} DuLNode, * DuLinkList;

/* P.19-20的抽象数据类型定义转换为实际的C语言 */
Status  InitList(DuLinkList* L);
Status  DestroyList(DuLinkList* L);
Status  ClearList(DuLinkList* L);
Status  ListEmpty(DuLinkList L);
int  ListLength(DuLinkList L);
```

```
Status  GetElem(DuLinkList L, int i, ElemType* e);
int  LocateElem(DuLinkList  L,  ElemType  e,  Status(*compare)(ElemType  e1,
ElemType e2));
Status  PriorElem(DuLinkList L, ElemType cur_e, ElemType* pre_e);
Status  NextElem(DuLinkList L, ElemType cur_e, ElemType* next_e);
Status  ListInsert(DuLinkList* L, int i, ElemType e);
Status  ListDelete(DuLinkList* L, int i, ElemType* e);
Status  ListTraverse(DuLinkList L, Status(*visit)(ElemType e));

/* 初始化线性表 */
Status InitList(DuLinkList* L)
{
    /* 申请头结点空间，赋值给头指针 */
    *L = (DuLNode*)malloc(sizeof(DuLNode));
    if (*L == NULL)
        exit(MYOVERFLOW);

    (*L)->prior = NULL;
    (*L)->next = NULL;
    return OK;
}

/* 删除线性表 */
Status DestroyList(DuLinkList* L)
{
    DuLinkList q, p = *L; //指向首元

    /* 整个链表(含头结点)依次释放(同单链表，不考虑prior指针，只用next) */
    while (p) {      //若链表为空，则循环不执行
        q = p->next; //抓住链表的下一个结点
        free(p);
        p = q;
    }

    *L = NULL;//头指针置NULL
    return OK;
}

/* 清除线性表（保留头结点） */
Status ClearList(DuLinkList* L)
{
    DuLinkList q, p = (*L)->next;

    /* 从首元结点开始依次释放(同单链表，不考虑prior指针，只用next) */
```

```
    while (p) {
        q = p->next;     //抓住链表的下一个结点
        free(p);
        p = q;
    }

    (*L)->next = NULL; //头结点的prior域置NULL
    (*L)->next = NULL; //头结点的next域置NULL
    return OK;
}

/* 判断是否为空表 */
Status ListEmpty(DuLinkList L)
{
    /* 判断头结点的next域即可(同单链表) */
    if (L->next == NULL)
        return TRUE;
    else
        return FALSE;
}

/* 求表的长度 */
int ListLength(DuLinkList L)
{
    DuLinkList p = L->next; //指向首元结点
    int len = 0;

    /* 循环整个链表，进行计数(同单链表) */
    while (p) {
        p = p->next;
        len++;
    }

    return len;
}

/* 取表中元素 */
Status GetElem(DuLinkList L, int i, ElemType* e)
{
    DuLinkList p = L->next;    //指向首元结点
    int       pos = 1;          //初始位置为1

    /* 链表不为NULL 且 未到第i个元素(同单链表) */
    while (p != NULL && pos < i) {
```

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

    if (!p || pos > i)
        return ERROR;

    *e = p->data;
    return OK;
}

/* 查找符合指定条件的元素 */
int LocateElem(DuLinkList L, ElemType e, Status(*compare)(ElemType e1,
ElemType e2))
{
    DuLinkList p = L->next;   //首元结点
    int      pos = 1;         //初始位置

    /* 循环整个链表(同单链表)  */
    while (p && (*compare)(e, p->data) == FALSE) {
        p = p->next;
        pos++;
    }

    return p ? pos : 0;
}

/* 查找符合指定条件的元素的前驱元素 */
Status PriorElem(DuLinkList L, ElemType cur_e, ElemType* pre_e)
{
    DuLinkList p = L->next;   //指向首元结点

    if (p == NULL) //空表直接返回
        return ERROR;

    /* 从第2个结点开始循环整个链表(如果比较相等，保证有前驱)(同单链表) */
    while (p->next && p->next->data != cur_e)
        p = p->next;

    if (p->next == NULL) //未找到
        return ERROR;

    *pre_e = p->data;
    return OK;
```

```
}

/* 查找符合指定条件的元素的后继元素 */
Status NextElem(DuLinkList L, ElemType cur_e, ElemType* next_e)
{
    DuLinkList p = L->next;   //首元结点

    if (p == NULL) //空表直接返回
        return ERROR;

    /* 有后继结点且当前结点值不等时继续(同单链表) */
    while (p->next && p->data != cur_e)
        p = p->next;

    if (p->next == NULL)
        return ERROR;

    *next_e = p->next->data;
    return OK;
}

/* 在指定位置前插入一个新元素 */
Status ListInsert(DuLinkList* L, int i, ElemType e)
{
#if 0
    DuLinkList s, p = *L;   //p指向头结点
    int         pos = 1;

    /* 寻找第i个结点(i可能是表长+1) */
    while (p->next && pos < i - 1) {
        p = p->next;
        pos++;
    }

    if (p->next == NULL || pos > i - 1)   //i值非法则返回
        return ERROR;

    //执行到此表示找到指定位置，p指向第i-1个结点

    s = (DuLinkList)malloc(sizeof(DuLNode));
    if (s == NULL)
        return MYOVERFLOW;

    if (pos == i - 1) {
```

```
    }
    else {
        /* 注意，p可能是NULL */
        s->data = e;    //新结点数据域赋值
        s->prior = p->prior;
        p->prior->next = s;
        s->next = p;    //新结点的next是第i个
        p->prior = s;   //第i-1个的next是新结点
    }
#else
    DuLinkList s, p = *L;  //p指向头结点
    int        pos = 0;

    /* 寻找第i-1个结点 */
    while (p && pos < i - 1) {
        p = p->next;
        pos++;
    }

    if (p == NULL || pos > i)   //i值非法则返回
        return ERROR;

    //执行到此表示找到指定位置，p指向第i-1个结点
    s = (DuLinkList)malloc(sizeof(DuLNode));
    if (s == NULL)
        return MYOVERFLOW;

    s->data = e;    //新结点数据域赋值
    s->next = p->next;   //新结点的next是第i个结点(即使p->next是NULL也没问题)
    if (p->next) //如果有第i个结点
        p->next->prior = s; //第i个结点的prior是s
    s->prior = p;           //s的前驱是p
    p->next = s;            //p的后继是s
#endif

    return OK;
}

/* 删除指定位置的元素，并将被删除元素的值放入e中返回 */
Status ListDelete(DuLinkList* L, int i, ElemType* e)
{
#if 1
    DuLinkList p = *L; //p指向头结点
    int        pos = 0;
```

```
    /* 寻找第i个结点（p是第i个结点） */
    while (p && pos < i) {
        p = p->next;
        pos++;
    }

    if (p == NULL || pos > i)    //i值非法则返回
        return ERROR;

    //执行到此表示找到了第i个结点，此时p指向第i个结点

    *e = p->data;          //取第i个结点的值
    p->prior->next = p->next;
    if (p->next) //要加判断条件
        p->next->prior = p->prior;
    free(p);               //释放第i个结点
#else
    DuLinkList q, p = *L;  //p指向头结点
    int        pos = 0;

    /* 寻找第i个结点（p->next是第i个结点） */
    while (p->next && pos < i - 1) {
        p = p->next;
        pos++;
    }

    if (p->next == NULL || pos > i - 1) //i值非法则返回
        return ERROR;

    //执行到此表示找到了第i个结点，此时p指向第i-1个结点

    q = p->next;           //q指向第i个结点
    p->next = q->next; //第i-1个结点的next域指向第i+1个(即使NULL也没错)
    if (q->next)
        q->next->prior = p;

    *e = q->data;          //取第i个结点的值
    free(q);               //释放第i个结点
#endif

    return OK;
}
```

```c
/* 遍历线性表 */
Status ListTraverse(DuLinkList L, Status(*visit)(ElemType e))
{
    line_count = 0;
    DuLinkList p = L->next;
    line_count = 0;          //计数器恢复初始值（与算法无关）

#if 1
    while (p && (*visit)(p->data) == TRUE) //同单链表
        p = p->next;

    if (p)
        return ERROR;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
#else
    /* 逆序输出 */
    while (p->next) //同单链表
        p = p->next;

    /* 执行到此，p指向最后一个结点 */
    while (p && p->prior && (*visit)(p->data) == TRUE) //同单链表
        p = p->prior;

    printf("\n");//最后打印一个换行，只是为了好看，与算法无关
#endif

    return OK;
}


void DL_display()//双向链表展示main函数
{
    DuLinkList L;
    ElemType    e1, e2;
    int         i, pos;

    InitList(&L);

    printf("空表=%s\n", (ListEmpty(L) == TRUE) ? "TRUE" : "FALSE");
    printf("表长=%d\n\n", ListLength(L));

    printf("插入%d个元素：\n", INSERT_NUM);
    for (i = INSERT_NUM; i > 0; i--)
```

```
        ListInsert(&L, 1, rand() % INSERT_NUM);
    ListTraverse(L, MyVisit);//此处传入MyVisit函数名

    printf("空表=%s\n", (ListEmpty(L) == TRUE) ? "TRUE" : "FALSE");
    printf("表长=%d\n\n", ListLength(L));

    /* 分别取第1、最后、以及小于第1、大于最后的4种情况下的的元素值、前驱值、后继值
       最后再加一个任意输入值 */
    for (i = 0; i < 5; i++) {
        int pos;
        switch (i) {
            case 0:
                pos = 1;
                break;
            case 1:
                pos = ListLength(L);
                break;
            case 2:
                pos = -1;
                break;
            case 3:
                pos = ListLength(L) + 1;
                break;
            case 4:
                while (1)
                {
                    printf("请输入要取元素的位序[1..%d]： ", ListLength(L));
                    if (!(cin >> pos))
                    {
                        cin.clear();
                        cin.ignore(0x7fffffff, '\n');
                        cout << "输入错误，重新输入" << endl;
                    }
                    else
                        break;
                }
                break;
        }

        /* 取第1个元素以及前驱、后继 */
        if (GetElem(L, pos, &e1) == OK) {
            printf("第%d个元素=%d\n", pos, e1);
```

```
            /* 只有取得了某个元素，才能取前驱和后继 */
            if (PriorElem(L, e1, &e2) == OK)
                printf("第%d个元素(%d)的前驱=%d\n", pos, e1, e2);
            else
                printf("无法取得第%d个元素(%d)的前驱\n", pos, e1);

            if (NextElem(L, e1, &e2) == OK)
                printf("第%d个元素(%d)的后继=%d\n\n", pos, e1, e2);
            else
                printf("无法取得第%d个元素(%d)的后继\n\n", pos, e1);
        }
        else
            printf("无法取得第%d个元素\n\n", pos);
    } // end of for

    while (1)
    {
        printf("请输入要查找的元素：\n");
        if (!(cin >> e1))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
        else
            break;
    }

    if ((pos = LocateElem(L, e1, MyCompare)) > 0)          //此处传入MyCompare函数
名
        printf("元素%d的位序=%d\n", e1, pos);
    else
        printf("找不到元素%d\n", e1);


    while (1)
    {
        printf("\n请输入要插入元素的值：\n");
        if (!(cin >> e1))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
```

```
        else
            break;
    }

    while (1)
    {
        printf("请输入要插入元素的位序：\n");
        if (!(cin >> pos))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
        else
            break;
    }

    if (ListInsert(&L, pos, e1) == OK) {
        printf("在%d前插入元素%d成功\n", pos, e1);
        printf("新表为：\n");
        ListTraverse(L, MyVisit);
    }
    else
        printf("在%d前插入元素%d失败\n", pos, e1);

    while (1)
    {
        printf("\n请输入要删除元素的位序：\n");
        if (!(cin >> pos))
        {
            cin.clear();
            cin.ignore(0x7fffffff, '\n');
            cout << "输入错误，重新输入" << endl;
        }
        else
            break;
    }


    if (ListDelete(&L, pos, &e1) == OK) {
        printf("删除第%d个元素=%d成功\n", pos, e1);
        printf("新表为：\n");
        ListTraverse(L, MyVisit);
    }
```

```
    else
        printf("删除第%d个元素失败\n", pos);

    printf("\n清空表：\n");
    ClearList(&L);
    printf("空表=%s\n", (ListEmpty(L) == TRUE) ? "TRUE" : "FALSE");
    printf("表长=%d\n", ListLength(L));

    DestroyList(&L);

    return;
}

class Ploy;//提前声明，因为定义友元要用到

class PLNode {
public:
    int coef;//系数
    int expo;//指数
    PLNode* next; //指针域
    friend class Ploy;
    friend ostream& operator<<(ostream& out, const Ploy& a);
    //不定义任何函数，相当于struct LNode
};

class Ploy {
protected:
    PLNode* head;     //头指针
public:
    /* P.19-20的抽象数据类型定义转换为实际的C++语言  */
    Ploy(); //构造函数，替代InitList
    Ploy(const Ploy& p);
    ~Ploy();     //析构函数，替代DestroyList
    void Ploy_add_ele(const int p, const int e);
    Ploy Ploy_mul_ele(const int c, const int e);
    Ploy& operator=(const Ploy& p);
    Ploy operator+(const Ploy& p);
    Ploy Ploy_mul(const Ploy& a);
    friend ostream& operator<<(ostream& out, const Ploy& a);
};



/*  构造函数（初始化线性表）  */
```

```
Ploy::Ploy()
{
    /* 申请头结点空间，赋值给头指针 */
    head = new PLNode;
    if (head == NULL)
        exit(MYOVERFLOW);

    head->next = NULL;
}

/* 构造函数（初始化线性表） */
Ploy::Ploy(const Ploy& p)
{
    /* 申请头结点空间，赋值给头指针 */
    head = new PLNode;
    if (head == NULL)
        exit(MYOVERFLOW);

    PLNode* q = head;
    PLNode* r = p.head;
    while (r->next)
    {
        q->next = new(nothrow) PLNode;
        if (!q->next)exit(MYOVERFLOW);
        q->next->coef = r->next->coef;
        q->next->expo = r->next->expo;
        q->next->next = r->next->next;
        q = q->next;
        r = r->next;
    }
}

/* 析构函数（删除线性表） */
Ploy::~Ploy()
{
    PLNode* q, * p = head;

    /* 从头结点开始依次释放（含头结点） */
    while (p) {      //若链表为空，则循环不执行
        q = p->next; //抓住链表的下一个结点
        delete p;
        p = q;
    }
```

```
    head = NULL;  //头指针置NULL
}

Ploy& Ploy::operator=(const Ploy& p)
{
    PLNode* t = head->next;
    PLNode* s;
    while (t)
    {
        s = t->next;
        delete t;
        t = s;
    }

    PLNode* q = head;
    PLNode* r = p.head;
    while (r->next)
    {
        q->next = new(nothrow) PLNode;
        if (!q->next)exit(MYOVERFLOW);
        q->next->coef = r->next->coef;
        q->next->expo = r->next->expo;
        q->next->next = r->next->next;
        q = q->next;
        r = r->next;
    }
    return *this;
}

Ploy Ploy::operator+(const Ploy& p)
{
    Ploy PC;
    PLNode* q = head->next;
    while (q)
    {
        PC.Ploy_add_ele(q->coef, q->expo);
        q = q->next;
    }
    q = p.head->next;
    while (q)
    {
        PC.Ploy_add_ele(q->coef, q->expo);
        q = q->next;
    }
```

```
    return PC;
}

void Ploy::Ploy_add_ele(const int c, const int e)
{
    if (!c)return;
    PLNode* l = head;
    PLNode* r = l->next;
    while (r)
    {
        if (r->expo == e)//该指数=r节点指数
        {
            r->coef += c;
            if (r->coef == 0)//加后系数0
            {
                l->next = r->next;
                delete r;
            }
            return;
        }
        else if (r->expo > e)//该指数未出现过，插在l和r之间
        {
            PLNode* s = new(nothrow) PLNode;
            if (!s)exit(MYOVERFLOW);
            s->expo = e;
            s->coef = c;
            l->next = s;
            s->next = r;
            return;
        }

        l = l->next;
        r = r->next;
    }

    //r移动到了NULL,还没有插进去，此时尾节点是l
    l->next = new(nothrow) PLNode;
    if (!l->next)exit(MYOVERFLOW);
    l->next->expo = e;
    l->next->coef = c;
    l->next->next = NULL;
    return;
}
```

```
Ploy Ploy::Ploy_mul_ele(const int c, const int e)
{
    Ploy temp;
    if (!c)return temp;
    PLNode* q = head->next;
    while (q)
    {
        temp.Ploy_add_ele(q->coef * c, q->expo + e);
        q = q->next;
    }

    return temp;
}

Ploy Ploy::Ploy_mul(const Ploy& a)
{
    Ploy res;
    PLNode* q = a.head->next;
    while (q)
    {
        Ploy temp = *this;
        temp = temp.Ploy_mul_ele(q->coef, q->expo);
        res = res + temp;
        q = q->next;
    }
    return res;
}

ostream& operator<<(ostream& out, const Ploy& a)
{
    PLNode* p = a.head->next;
    if (!p)
        return out;

    while (p)
    {
        out << "(" << p->coef << "^" << p->expo << ")" << (p->next == NULL ? '\n' :
'+');
        p = p->next;
    }

    return out;
}
```

```
void Ploy_display()
{
    Ploy PA, PB;
    const int n = rand() % 10 + 5;
    const int m = rand() % 10 + 5;
    for (int i = 0; i < n; i++)
    {
        int e = rand() % 20;
        int c = rand() % 20;
        PA.Ploy_add_ele(c, e);
    }
    for (int i = 0; i < m; i++)
    {
        int e = rand() % 20;
        int c = rand() % 20;
        PB.Ploy_add_ele(c, e);
    }

    cout << "随机生成的一元多项式 PA = " << PA;
    cout << "随机生成的一元多项式 PB = " << PB;
    cout << "演示多项式相加PA+PB的结果 = " << PA + PB;
    cout << "演示多项式相乘PA*PB的结果 = " << PA.Ploy_mul(PB);

    return;
}

int main()
{
    srand((unsigned int)(time(0)));
    const char* menu[] = {
        "1.使用带头节点的单链表",
        "2.使用带头节点的双向链表",
        "3.一元多项式操作",
        "0.退出",
        NULL        //约定这种简易菜单的最后一项一定是NULL
    };
    const char* menu_input_reminder = "请选择[0^3] : ";
    char legal_input[UCHAR_MAX];
    memset(legal_input, 0, sizeof(char) * CHAR_MAX);
    legal_input['0'] = legal_input['1'] = legal_input['2'] = legal_input['3'] = 1;
    char sel;
    bool loop = true;

    while (loop) {
```

```
        /* 允许增加simple_menu的参数，个数及类型不限，不要改变返回值 */
        sel = simple_menu(menu, legal_input, menu_input_reminder);

        switch (sel) {
            case '1':
                LinkList_display();
                break;
            case '2':
                DL_display();
                break;
            case '3':
                Ploy_display();
                break;
            case '0':
                loop = false;
                break;
        }

        if (loop)
            wait_for_end();
    }

    return 0;
}
```