

作业 HW1 线性表 实验报告

姓名：段威呈 学号：2252109 日期：2023 年 10 月 28 日

1. 涉及数据结构和相关背景

顺序表 链表 （借助线性表实现对排列组数据的增删改等操作）

2. 实验内容

2.1 轮转数组

2.1.1 问题描述

给定一个整数顺序表 nums 将顺序表中的元素向右轮转 k 个位置，其中 k 是非负数。

2.1.2 基本要求

输入：

第一行两个整数 n 和 k，分别表示 nums 的元素个数 n，和向右轮转 k 个位置；

第二行包括 n 个整数，为顺序表 nums 中的元素数据。

输出：轮转后的元素

数据范围要求：

$1 \leq \text{nums.length} \leq 105$

$-231 \leq \text{nums}[i] \leq 231 - 1$

$0 \leq k \leq 105$

用至少三种方法

2.1.3 数据结构设计 顺序表的构建

①定义线性表结构：存储数据 data，记录顺序表长度 length

```
//定义线性表的结构
typedef struct
{
    ElemType* data;//顺序表元素
    int length;//顺序表当前长度
}SqList;
```

②初始化线性表

```
int SqList_Init(SqList& L)
{
    L.data = new ElemType[Maxsize];//为顺序表分配一个大小为Maxsize的数组空间
    L.length = 0;//空表长度为0
    return 0;
}
```

③建立顺序表：如果 n 取值不合理，返回 false，无法建立；如果合理，则依次键入 int 型数据作为 data 值

```
bool SqList_Create(SqList& L, int n)
{
    if (n < 0 || n > Maxsize)
    {
        return false;
    }
    else
    {
        for (int i = 0; i < n; i++)
        {
            cin >> L.data[i];
            L.length++;
        }
        return true;
    }
}
```

④打印顺序表：将顺序表的内容打印在屏幕上

```
void SqList_Print(SqList& L) {
    for (int t = 0; t < L.length; t++) {
        cout << L.data[t] << " ";
    }
    cout << endl;
}
```

2.1.4 功能说明（函数、类）

方法一：额外数组法（时间复杂度 $O(n)$ ，空间复杂度 $O(n)$ ）

轮转过程可通过如下步骤实现：

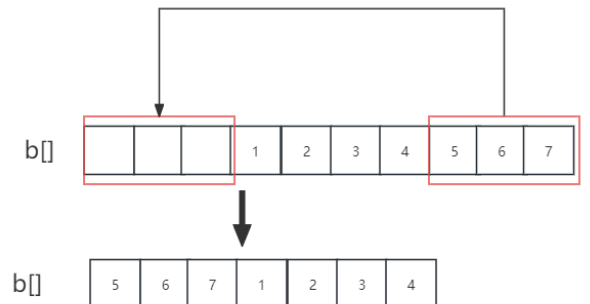
①函数 `SqList_Copy1()` 把原顺序表错位 k 项 5 存到额外数组 `b[]` 中

```
void SqList_Copy1(SqList& L, int k, int *b) {
    // 将原序列赋值到额外数组中
    for (int j = 0; j < L.length; j++) {
        b[j + k] = L.data[j];
    }
}
```



②将 `b[]` 大于 k 的部分移回到头部

```
for (int s = n; s < n + k; s++) {
    b[s - n] = b[s]; // 将超出长度的范围轮回到数组首端
}
```

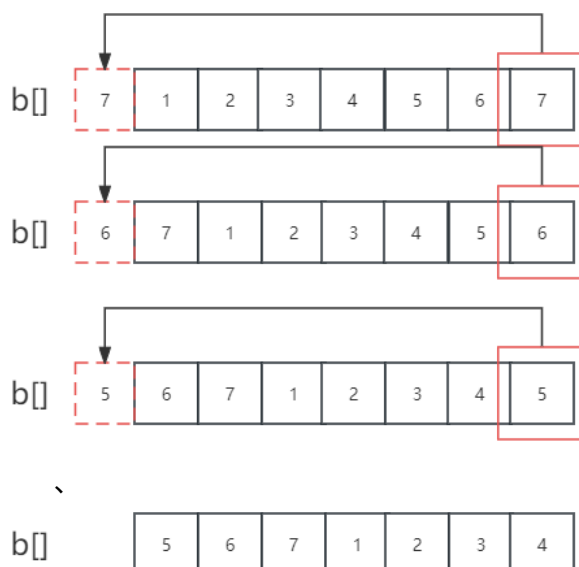


③函数 `SqList_Copy2()` 把 `b[]` 的元素复制到 `L` 中（覆盖原先元素）

```
void SqList_Copy2(SqList& L, int k, int* b) {
    // 将额外数组赋值到原序列中
    for (int j = 0; j < L.length; j++) {
        L.data[j] = b[j];
    }
}
```

方法二：冒泡法（时间复杂度 $O(n^2)$ ，空间复杂度 $O(1)$ 的原地算法）

每一步轮转可以视为把最后一个元素通过冒泡交换移动到顺序表首位，过程如下：



对应执行该功能的为 `Bubble()` 函数：

```
void Bubble(SqList& L, int k, int n) {
    while (k-->0) {
        for (int i = n - 1; i > 0; i--) {
            swap(&(L.data[i - 1]), &(L.data[i]), L);
        }
    }
}
```

其中 `swap(a,b)` 是将顺序表中两个元

素交换：

```
void swap(int* a, int* b, SqList& L) {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

方法三：翻转法（时间复杂度为 $O(n)$,空间复杂度为 $O(1)$ 的“原地”算法）

数组轮换可以经过如下过程实现：

名称	结果
原数组	1 2 3 4 5 6 7
整体翻转	7 6 5 4 3 2 1
翻转 $[0, k\%n-1]$ 区间的元素	5 6 7 4 3 2 1
翻转 $[k\%n, n-1]$ 区间的元素	5 6 7 1 2 3 4

翻转操作可以借助翻转函数 `reverse()`实现：

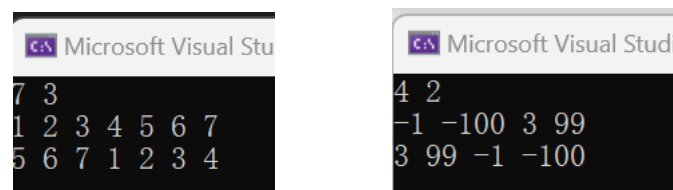
```
void reverse(int start, int end, SqList& L)
{
    while (start < end) {
        swap(&(L.data[start]), &(L.data[end]), L);
        start++;
        end--;
    }
}
```

即两端对称部分依次交换, 借助交换函数 `swap()`:

```
void swap(int* a, int* b, SqList& L) {
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

2.1.5 调试分析（遇到的问题解决方法）

①测试样例：



②以上三种方法在 oj 平台提交后均通过所有测试点。

但运行的时间效率：方法三=方法一>方法二

运行的空间效率：方法三=方法二>方法一

2.1.6 总结和体会

本题难点重在选取合适的算法。

其中 方法一：额外数组法 较为直接，但是用到了额外的数组，空间复杂度达到 $O(n)$ ，较高。

而 方法二：冒泡法 从结果倒推，模拟出分步轮换的过程，是一种原地算法，空间复杂度降低到 $O(1)$ ，但是时间复杂度达到 $O(n^2)$ 。

最后的 方法三：翻转法 是发现了轮换数组的规律，只借助翻转的方法实现了耗时少、占空间小的轮换方法，时间复杂度和空间复杂度均最优。

2.2 学生信息管理

2.2.1 问题描述

本题 定义一个包含学生信息（学号，姓名）的的顺序表，使其具有如下功能：(1) 根据指定学生个数，逐个输入学生信息；(2) 给定一个学生信息，插入到表中指定的位置；(3) 删除指定位置的学生记录；(4) 分别根据姓名和学号进行查找，返回此学生的信息；(5) 统计表中学生个数。

2.2.2 基本要求

本题须使用顺序表完成。应当包含如下指令操作：

insert i 学号 姓名：表示在第 i 个位置插入学生信息，若 i 位置不合法，输出 -1，否则输出 0

remove j:表示删除第 j 个元素，若元素位置不合适，输出 -1，否则输出 0

check name 姓名 y: 查找姓名 y 在顺序表中是否存在，若存在，输出其位置序号及学号、姓名，若不存在，输出 -1。

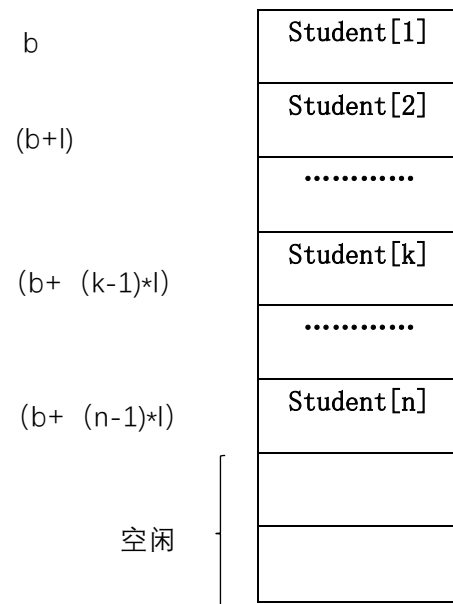
check no 学号 x: 查找学号 x 在顺序表中是否存在，若存在，输出其位置序号及学号、姓名，若不存在，输出 -1。

end: 操作结束，输出学生总人数，退出程序。

注：全部数值 ≤ 10000 ，元素位置从 1 开始。学生信息有重复数据（输入时未做检查），查找时只需返回找到的第一个。

2.2.3 数据结构设计 顺序表的构建、元素插入、删除、访问等

内存中顺序表结构如下：



①定义顺序表元素的结构：Student 结构体，包含次序号，学生学号以及学生姓名

```
//学生信息结构
struct Student {
    int num;
    string ID;
    string name;
};
```

②定义顺序表结构：首指针，元素，顺序表长度，顺序表占用空间

```
//定义顺序表结构
typedef struct {
    ElemType* elem; //elem数组指针指向当前线性表的基址
    int length; //线性表当前长度
    int listsize; //指示顺序表分配的空间的大小
    Student student[100001];
} SqList;
```

③初始化顺序表：为顺序表申请一块动态内存，设置各参量初始值

```
//初始化顺序表
bool SqListInit(SqList& L)
{
    L.elem = (int*)malloc(LIST_INIT_SIZE * sizeof(int)); //为表申请一块动态内存空间
    L.length=0; //初始长度定义为0
    L.listsize = LIST_INIT_SIZE; //设置初始化长度
    return true;
}
```

2.2.4 功能说明（函数、类）

①向顺序表中输入内容：SqListInput()

```
//向顺序表中输入内容
void SqListInput(SqList& L, int N) { //N由外部输入，代表顺序表的长度
    for (int i = 1; i <= N; ++i) {
        L.length++;
        L.student[i].num = i; //次序更新
        cin >> L.student[i].ID >> L.student[i].name ; //输入学号、姓名
    }
}
```

②向顺序表中插入内容:SqListInsert()

当插入的位置不合法时：

```
if (i<1 || i>L.length+1 ) { //插入位置不合法
    cout << "-1" << endl;
}
```

当插入位置合法时：

```
L.length++;
Student* p = (&L.student[i]); //位置指针
Student* m = (&L.student[L.length]); //尾指针
for (; p <= m; m--) {
    *(m + 1) = *m; //借助尾指针依次插入位置后的元素往后移出一个位置，为插入元素腾出空间
}
```

插入元素，并更新序号：

```
//插入元素并输出0
cin >>L.student[i].ID >>L.student[i].name ; //插入元素
for (int i = 1; i <= L.length; ++i) { //更新位置序号
    L.student[i].num = i;
}
cout << "0" << endl;
```

③向顺序表中删除内容: SqListDelete()

删除位置不合法时, 同插入操作;

当删除位置合法时:

```
Student* p = (&L.student[i]);           //位置指针
Student* m = (&L.student[L.length]);     //尾指针
for (; p <= m; p++) {
    *p = *(p + 1);                       //从删除位置开始, 元素从前往后, 依次将后一个覆盖前一个
}
```

④通过学号查询: GetElemID()

```
//查询学号
void GetElemID(SqList& L, string q)      //输入待查询的学号
{
    int count = 1; //count作用是记录查询的总次数
    for (int i = 1; i <= L.length; i++) {
        if (q.compare(L.student[i].ID) == 0) { //依次遍历查询
            cout << L.student[i].num << " " << L.student[i].ID << " " << L.student[i].name << endl;
            break;
        }
        else count++;
    }
    if (count > L.length) cout << "-1" << endl; //在循环结束后如果count超过表长, 说明没查询出来, 意味着该元素不存在
}
```

⑤通过姓名查询: GetElemName()

构造同学号查询, 只是遍历比对的对象变为 L.student[i].Name

⑥在主函数中根据不同输入调用不同的功能即可

```
while(true) {
    cin >> m;
    if (m == "insert") {
        cin >> i;
        SqListInsert(L, i);
    }
    if (m == "remove") {
        cin >> i;
        SqListDelete(L, i);
    }
    if (m == "check") {
        cin >> n;
        if (n == "no") {
            cin >> t;
            GetElemID(L, t);
        }
        if (n == "name") {
            cin >> t;
            GetElemName(L, t);
        }
    }
    if (m == "end") {
        cout << L.length << endl;
        break;
    }
}
```

2.2.5 调试分析（遇到的问题解决方法）

遇到的问题：

①输入顺序表初始信息：

```
4
2252109 a
2252108 b
2252107 c
2252106 d
```

插入一条，插入位置合理，输出 0：

```
insert 1 2252110 e
0
```

这时再查询后续元素，其输出的次序号应当 + 1：

这一点在一开始做题时并没有注意到，导致元素序号没有改变，注意在每次插入和删除操作时要更新位置序号。加入更新位置序号的操作后，显示正确、：

```
insert 1 2252110 e
0
check name a
2 2252109 a
check no 2252107
4 2252107 c
```

②在运行时部分样例未通过：

原因：注意到题干有对于输入元素 N 的要求：

全部数值 ≤ 10000 ，元素位置从1开始。

如果 Student[]开小了在 OJ 平台上会显示运行时间错误，这是因为提供的测试样本超出了数组的大小范围，这时只需要把数组范围开大即可，同时在调整数组范围的同时还要调整顺序表的最大长度限制。

2.2.6 总结和体会

本题只需要按照题目要求完成对应功能即可，但是在对于位置不合法时输出、插入删除后序号位置的变动、线性表长度设置、起首位置为 1 等方便都有很多细节需要注意；要格外关注逻辑执行的顺序、边界情况取等的问题，否则很容易出现错误的输出值或者栈溢出的情况。

2.3 一元多项式的相加和相乘

2.3.1 问题描述

一元多项式是有序线性表的典型应用，用一个长度为 m 且每个元素有两个数据项（系数项和指数项）的线性表 $((p_1, e_1), (p_2, e_2), \dots, (p_m, e_m))$ 可以唯一地表示一个多项式。本题实现多项式的相加和相乘运算。本题输入保证是按照指数项递增有序的。

2.3.2 基本要求

输入：

第 1 行一个整数 m ，表示第一个一元多项式的长度

第 2 行有 $2m$ 个整数， $p_1 e_1 p_2 e_2 \dots$ ，中间以空格分割，表示第 1 个多项式系数和指数

第 3 行一个整数 n ，表示第二个一元多项式的项数

第 4 行有 $2n$ 个整数， $p_1 e_1 p_2 e_2 \dots$ ，中间以空格分割，表示第 2 个多项式系数和指数

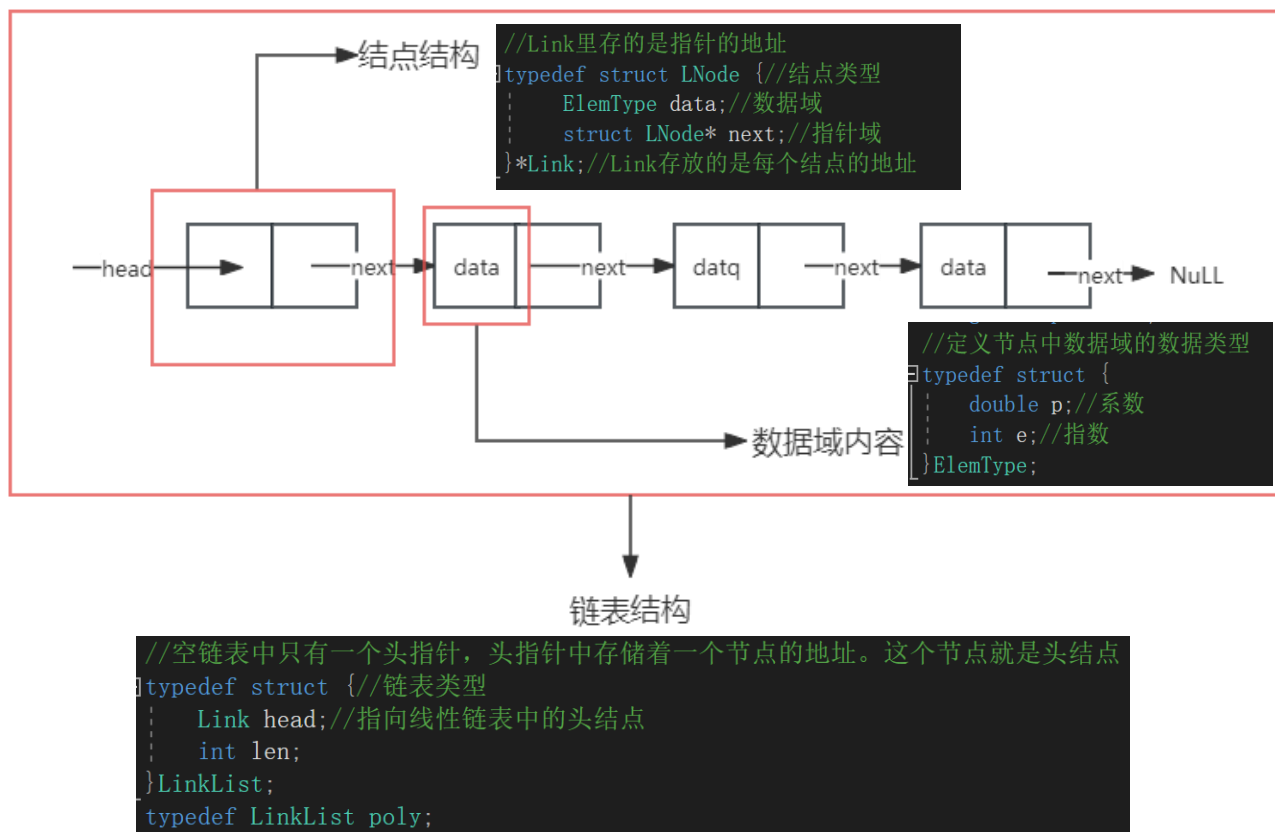
第 5 行一个整数，若为 0，执行加法运算并输出结果，若为 1，执行乘法运算并输出结果；若为 2，输出一行加法结果和一行乘法的结果。

输出：

运算后的多项式链表，要求按指数从小到大排列

当运算结果为 0 0 时，不输出。

2.3.3 数据结构设计：链表的构造、插入删除等操作



初始化链表：得到一个仅含头结点的链表



```
//初始化空的线性链表
int InitList(LinkList* L){
    if (L == NULL)
        return -1;
    Link p = (Link)malloc(sizeof(struct LNode)); //p指向链表所分配的存储空间的基址
    p->next = NULL; //头结点的指针域指向空
    L->head = p; //头指针指向头结点
    return 0;
}
```

2.3.4 功能说明（函数、类）

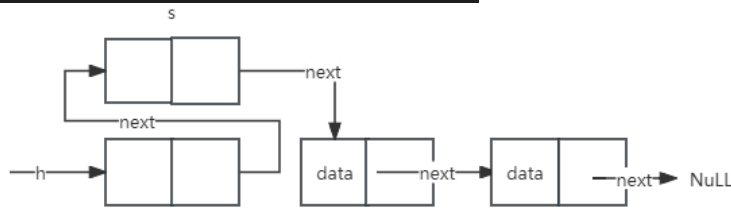
①链表的基本功能

链表基本操作

CreatPolyn(poly * P, int m)	创建多项式链表P，长度为m
GetCurElem(Link p)	获得结点p的data值
SetCurElem(Link p, ElemType e)	设置结点p的data为e
GetHead(LinkList * L)	获取链表L的头结点
MakeNode(Link* p, ElemType e)	新建结点p，data为e
LocateElem(LinkList * L, ElemType e, Link * q, int (*fun)(ElemType, ElemType))	在链表L中检索是否有数据域为e的结点，检索到后返回
DelFirst(Link * h, Link * q)	h指向头节点，将结点第一个结点删除
InsFirst(Link * h, Link * s)	h指向头节点，将s结点插入到第一个结点位置
NextPos(LinkList * L, Link p)	返回p结点的后继结点
CountLen(poly P)	返回多项式链表的长度
PrintPolyn(poly P)	打印链表data的内容
Append(LinkList * L, Link * s)	将s添加到链表L的尾部
FreeNode(Link p)	释放结点p及其后面一系列结点的存储空间

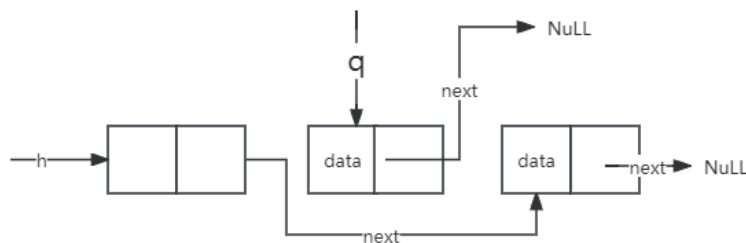
1) **InsFirst(Link *h, Link *s)** h 指向头结点，在第一个结点位置处插入结点 s

```
(*s)->next = (*h)->next;
(*h)->next = (*s);
```



2) **DelFirst(Link *h, Link *q)** h 指向头结点，删除第一个结点

```
(*q) = (*h)->next;
(*h)->next = (*q)->next;
(*q)->next = NULL;
```



3) **Append(LinkList *L, Links)** 将结点 s 添加到链表 L 尾部

```
Link p = L->head->next;
Link q = L->head;
while (p) { //两个相邻的指针，向后移动，直到后指针指向空，说明到了结尾，将s加到尾部
    q = p;
    p = p->next;
}
q->next = (*s);
```

4) **MakeNode(Link *p, ElemType e)** 生成结点

```
(*p)->next = NULL;
(*p)->data = e;
```



5) **CreatPolyn(poly P*, int m)** 创建链表

```
SetCurElem(h1, e1); //设置头结点的数据元素
for (i = 1; i <= m; i++) { //依次输入m个非零项
    cin >> e1.p >> e1.e;
    if (LocateElem(P, e1, &q1, cmp) < 0) { //如果当前链表中不存在该指数项
        if (MakeNode(&s1, e1) == 0) { //生成结点并插入链表
            InsFirst(&q1, &s1);
        }
    }
}
```

5) **FreeNode(Link p)**释放结点 p 及其后续一系列结点的存储空间

```
Link q;
while (p) { //两个相邻指针q, p, 顺次遍历整个链表, 并释放q指向的每一个结点
    q = p;
    p = p->next;
    free(q);
}
```

②多项式计算函数

1) **TidyPolyn(poly* P)** 整理多项式: 合并指数相同的项, 并按指数升序排列项

合并指数相同的项 关键代码:

```
while (q) { //将链表中结点数据域中的指数两两比较 (时间复杂度 $O(n^2)$ )
    if (q != NULL) { //q指向一个结点, p指向q后的某一个结点, h为辅助指针, 指向p前紧邻的节点, 作为虚拟头结点作为删除后一
        a = GetCurElem(q);
        p = NextPos(P, q);
        h = q;
        while (p) {
            b = GetCurElem(p);
            if (a.e == b.e) { //一列中遇到两个指数相等的
                a.p = a.p + b.p;
                SetCurElem(q, a);
                //p此时指向待删除的结点, h为p紧邻的前一个节点
                DelFirst(&h, &s);
                len--;
            }
            h = p;
            p = NextPos(P, p);
        }
        q = NextPos(P, q); //q步进到下一个结点
    }
}
```

冒泡排序, 实现升序 关键代码:

```
for(int i = 1; i <= len-1; i++){
    h = GetHead(P);
    q = NextPos(P, h); //q指针指向前结点
    p = NextPos(P, q); //p指针指向后结点
    for (int j = 1; j <= len - i; j++){
        a = GetCurElem(q);
        b = GetCurElem(p);

        if (a.e > b.e) { //若前结点中的指数比后结点的指数大, 则需要相互交换
            temp = GetCurElem(q);
            SetCurElem(q, p->data);
            SetCurElem(p, temp);
        }

        q = p;
        p = NextPos(P, p); //p, q指针同时步进到下一个结点, 从而进行下一次比较
    }
}
```

2) **AddPolyn(poly* Pa, poly* Pb)** 将 Pa 和 Pb 多项式链表进行相加操作，相加结果更新到 Pa 链表中 关键代码：

遍历 Pa 和 Pb，若遇到指数相等的情况：

```
case 0://a==b, 两者的指数值相等
    e.e = a.e;
    e.p = a.p + b.p;
    //修改多项式Pa中当前结点的系数值
    if (e.p != 0.0) {    //相加不为零更新Pa节点对应的数据
        SetCurElem(qa, e);
        ha = qa;
    }
    else {                //相加为零则直接删除该结点
        DelFirst(&ha, &qa);
        FreeNode(qa);
    }
    DelFirst(&hb, &qb);
    FreeNode(qb);
    qb = NextPos(Pb, hb);
    qa = NextPos(Pa, ha);
```

3) **MultiplyPolyn(poly* Pa, poly* Pb)** 将 Pa 和 Pb 多项式链表进行相乘操作，相乘结果更新到 Pa 链表中 关键代码：

```
qb = NextPos(&Pb, hb); //初始化qb, qb在计算中指向Pb中的某一项
while (qb) //模拟数学计算的分配律, Pb逐项与Pa整个多项式相乘
{
    ha = GetHead(&Pa);
    qa = NextPos(&Pa, ha);
    InitList(&Pc); //Pc是辅助链表, 保存Pb的其中一项与Pa相乘的结果
    while (qa) //qb指向的项与qa中的每一项依次相乘
    {
        e.p = qa->data.p * qb->data.p;
        e.e = qa->data.e + qb->data.e;
        MakeNode(&s, e); //Pc一开始是一个空链表, 相乘结果存到一个新结点中, 再添加到Pc里
        Append(&Pc, &s);
        ha = qa;
        qa = NextPos(&Pa, ha);
    }

    AddPolyn(&Pd, &Pc); //Pd也是一个新链表
    hb = qb;
    qb = NextPos(&Pb, hb); //qb步进, 指向Pb中的下一项
    free(hb);
}
```

2.3.5 调试分析（遇到的问题和解决方法）

在一开始写程序时，只写了 `AddPolyn()` 和 `MultiplyPolyn()`，这时候计算结果是乱序的，还有的项没有被合并，如下：



```
选择 Microsoft Visual Studio 调试控制台
3
2 4 5 8 1 2
2
4 7 2 4
2
2 4 4 7 2 4 5 8 1 2
4 8 8 11 10 12 2 6 20 15 4 9
```

于是又写了一个 `TidyPolyn()` 函数，对链表中的相同项进行合并，再升序排列，此时输出结果的样式符合题目要求：



```
选择 Microsoft Visual Studio 调试控制台
3
2 4 5 8 1 2
2
4 7 2 4
2
1 2 4 4 4 7 5 8
2 6 4 8 4 9 8 11 10 12 20 15
```

提交到 OJ 平台上运行，通过了前 10 个测试样例。

2.3.6 总结和体会

本题用链表实现，难点在于如何设计多项式相加和相乘的算法。同时注意到题干对于输出结果有指数从小到大排列的要求，因此还需要在计算后进行单独的排序调整。

由于本题需要多次对于链表某个结点的数据进行访问，但是链表并不支持随机访问，应当减少后续访问修改的次数以提高运行效率。

另外注意设置好边界量，否则会在运行的时候提醒访问出错。

2.4 求级数

2.4.1 问题描述

在一行中输出级数 $A+2A^2+3A^3+...+NA^N=\sum_{i=1}^N iA^i$ 的整数值.

2.4.2 基本要求

若干行，在每一行中给出整数 N 和 A 的值， $(1 \leq N \leq 150, 0 \leq A \leq 15)$
对于 20%的数据，有 $1 \leq N \leq 12, 0 \leq A \leq 5$
对于 40%的数据，有 $1 \leq N \leq 18, 0 \leq A \leq 9$
对于 100%的数据，有 $1 \leq N \leq 150, 0 \leq A \leq 15$

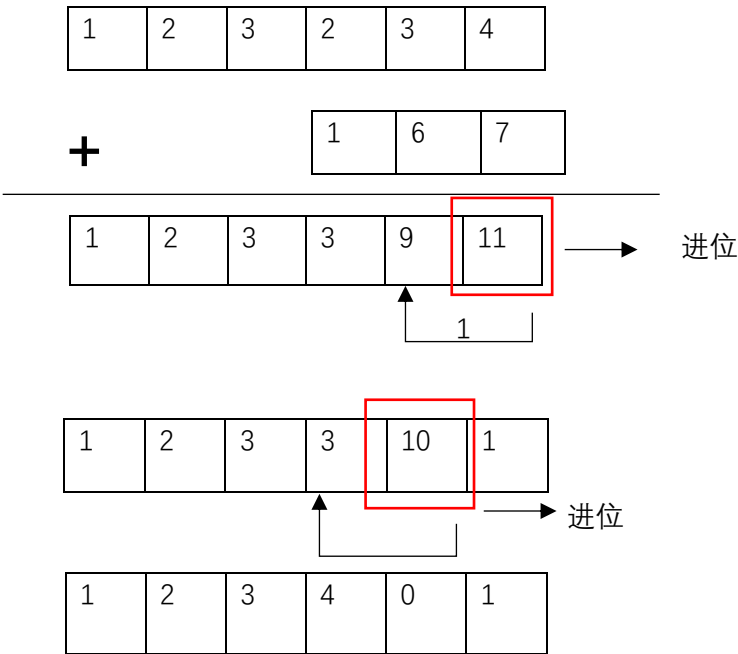
2.4.3 算法设计：高精度算法

高精度可以计算超过 23 位的大数（即可以表示 double 无法表示的数），其是借助数组模拟了进位的过程。首先高精度中所有字符的存储都是倒序的：

原数	倒序存储在 char a[]
5	5
15	5 1
250	0 5 2
123456	6 5 4 3 2 1

这样的实质是位数的小端和数组的小端对齐，方便后续进行相加/相乘以及进位操作。

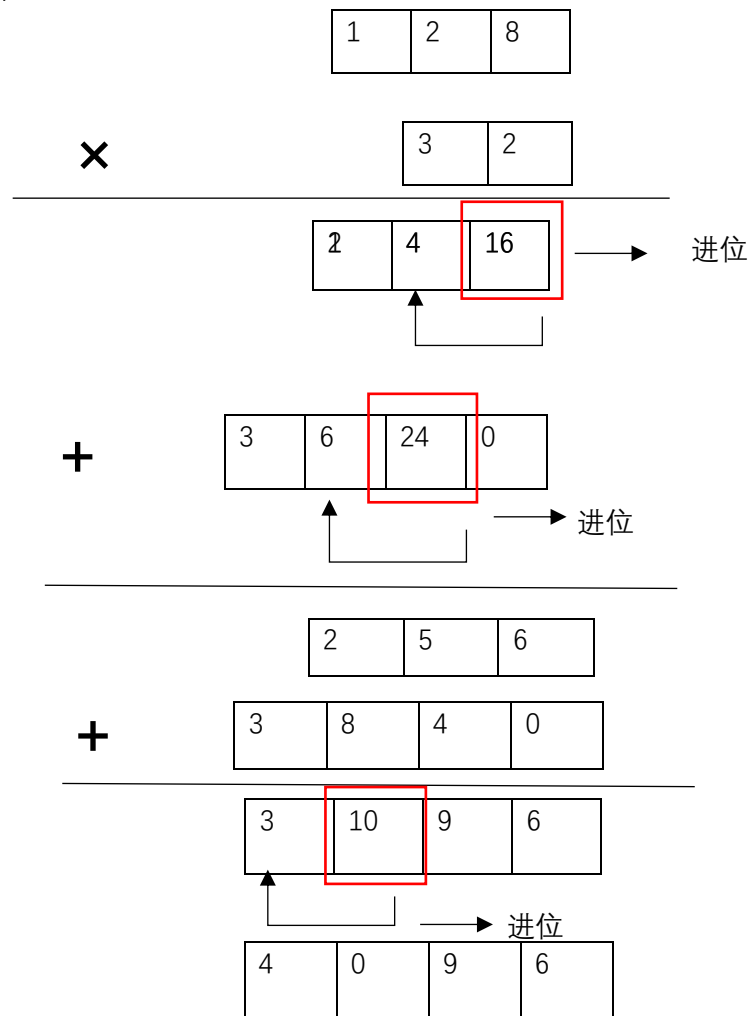
加法操作：竖式计算



由此可知：加法实际是对齐的每一位分别相加，所得结果对每一位再进位处理。

进位过程：除 10，结果为进位位数，加到高位，余数为该位保留的数。

乘法操作：竖式计算



由此可知：乘法操作时，进行的是每位对应两两相乘，第 i 位和第 j 位相乘，结果放到 $(i+j)$ 位，进位后的得到的多个数按照加法法则再相加。

2.4.4 实现过程说明

根据以上原理，可编写出加法计算和乘法计算的公式

① 计算 A^i , k 保存上一次 $A^{(m-1)}$ 的计算结果，用于这次 $A \cdot A^{(M-1)} = A^m$ 的计算

```
for (int m = 0; m < 10002; m++) {    //借助middle保存中间运算的量，初始化middle为0
    middle[m] = 0;
}
for (int e = 0; e < len_k; e++) {    //len_k为k的数位
    for (int j = 0; j < len_A; j++)  //len_A为A的数位
    {
        middle[e + j] += k[e] * A[j];    //每位对应两两相乘
    }
}
len_all = len_A + len_k;    //len_all为乘后结果的数位
for (int jinwei = 0; jinwei < len_all; jinwei++) {    //进位
    if (middle[jinwei] >= 10) {
        middle[jinwei + 1] += middle[jinwei] / 10;
        middle[jinwei] %= 10;
    }
}
//计算 i*A^i
//将middle的有效部分给k
for (int r = 0; r < len_all; r++) {
    k[r] = middle[r];
}
```

② 方法与 A^i 相同，还是两数相乘

③ 将 $i \times A^i$ 加到 sum 中

```
for (int r = 0; r < 10000; r++) //最终结果不超过10000位的整数的加法
{
    sum[r] = sum[r] + k[r] + x; //每一位由三部分相加而成，原来该位的值，k的在该位的值，来自低位的进位值
    x = sum[r] / 10;           //计算进位的位数
    sum[r] = sum[r] % 10;      //只保留余数
}
```

2.4.5 调试分析（遇到的问题和解决方法）

对于所有存放大数的变量，初始化都是将每位赋为 0，从而再开始由低到高读入字符。

在输出数组元素的时候，需要知道位数的长度，故要对数组有效位数进行单独统计（即不是一开始初始化赋予的 0 的部分），统计时终止标准不能是从低到高开始遇 0 终止，因为原数中可能也包含 0，会导致统计结果比实际短（如 100，存储在数组中为 0 0 1，如果用遇零终止统计长度，结果则为 0，比实际长度 3 短），因此不能用该方法。

于是选用从高到低开始统计，第一次遇到不为零的数终止（即遇到最高位，最高位必然不为 0），这样可以避免问题。

2.4.6 总结和体会

本题用到了高精度算法，但是难点并不在于如何构造高精度，而在于如何正确逆序存放好输入的数据以及中间运算量，并将其倒序正确输出。

2.5 扑克牌游戏

2.5.1 问题描述

扑克牌有 4 种花色：黑桃（Spade）、红心（Heart）、梅花（Club）、方块（Diamond）。每种花色有 13 张牌，编号从小到大为：A,2,3,4,5,6,7,8,9,10,J,Q,K。

对于一个扑克牌堆，定义以下 4 种操作命令：

- 1) 添加（Append）：添加一张扑克牌到牌堆的底部。如命令“Append Club Q”表示添加一张梅花 Q 到牌堆的底部。
- 2) 抽取（Extract）：从牌堆中抽取某种花色的所有牌，按照编号从小到大进行排序，并放到牌堆的顶部。如命令“Extract Heart”表示抽取所有红心牌，排序之后放到牌堆的顶部。
- 3) 反转（Revert）：使整个牌堆逆序。
- 4) 弹出（Pop）：如果牌堆非空，则除去牌堆顶部的第一张牌，并打印该牌的花色和数字；如果牌堆为空，则打印 NULL。

初始时牌堆为空。输入 n 个操作命令（ $1 \leq n \leq 200$ ），执行对应指令。所有指令执行完毕后打印牌堆中所有牌花色和数字（从牌堆顶到牌堆底），如果牌堆为空，则打印 NULL。

2.5.2 基本要求

注意：每种花色和编号的牌数量不限。

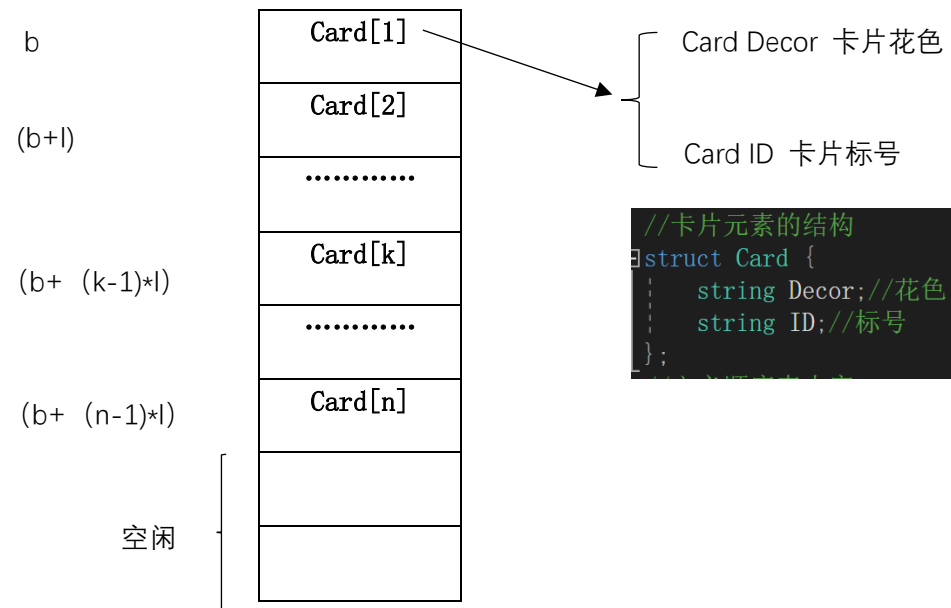
对于 20% 的数据， $n \leq 20$ ，有 Append、Pop 指令

对于 40% 的数据， $n \leq 50$ ，有 Append、Pop、Revert 指令

对于 100% 的数据， $n \leq 200$ ，有 Append、Pop、Revert、Extract 指令

2.5.3 数据结构设计：顺序表的建立、元素查询、插入、删除操作

内存中顺序表结构如下：



①定义顺序表内容：

```
//定义顺序表内容
typedef struct {
    ElemType* elem; //elem数组指针指向当前线性表的基址
    int length; //线性表当前长度
    int listsize; //指示顺序表分配的存储空间的大小
    Card card[301];
} SqList;
```

2.3.4 功能说明（函数、类）

①顺序表的功能

1) 初始化顺序表 SqListInit(SqList &L)

```
bool SqListInit(SqList& L)
{
    L.elem = (int*)malloc(LIST_INIT_SIZE * sizeof(int)); //顺序表的基址指向分配的动态存储空间
    L.length = 0;
    L.listsize = LIST_INIT_SIZE;
    return true;
}
① 初始化线性表。
```

2) 顺序表元素插入 SqListInsert(SqList& L, int i)（从键盘输入插入项）与 SqListInsertInput(SqList& L, int i, string L_Decor, string L_ID)（形参包括插入项）原理类似 题 2 中学生信息管理关于顺序表插入的操作

```
L.length++;
Card* p = (&L.card[i]); //首指针
Card* m = (&L.card[L.length]); //尾指针
for (; p <= m; m--) {
    *(m + 1) = *m;
}
cin >> L.card[i].Decor >> L.card[i].ID;
```

```
L.length++;
Card* p = (&L.card[i]); //首指针
Card* m = (&L.card[L.length]); //尾指针
for (; p <= m; m--) {
    *(m + 1) = *m;
}
L.card[i].Decor = L_Decor;
L.card[i].ID = L_ID;
```

3) 顺序表第 i 个元素删除出 `SqListDelete(SqList&L, int i)`

类似 题 2 中学生信息管理关于顺序表元素删除的操作

```
L.length--;  
Card* p = (&L.card[i]);  
Card* m = (&L.card[L.length]);  
for (; p <= m; p++) {  
    *p = *(p + 1);  
}
```

4) 顺序表元素逆序 `ReverseList(SqList &L)`

```
void RevertList(SqList& L) {  
    int i = 1;  
    int j = L.length;  
    Card temp;  
    for (; i <= L.length / 2; i++, j--) {  
        temp = L.card[i];  
        L.card[i] = L.card[j];  
        L.card[j] = temp;  
    }  
}
```

5) 打印顺序表元素 `PrintSqList(SqList &L)`

```
void PrintSqList(SqList& L) {  
    if (L.length > 0) {  
        for (int i = 1; i <= L.length; i++) {  
            cout << L.card[i].Decor << " " << L.card[i].ID << endl;  
        }  
    }  
    if (L.length == 0) {  
        cout << "NULL" << endl;  
    }  
}
```

② 实现输入指令功能

Pop, Delete, Reverse 指令均可直接调用链表功能直接实现, 这里仅展示较为复杂的 Extract 的实现:

从牌堆中依次取出对应花色的牌, 并将牌的序号升序存在一个数组 `ID_Store[]` 中:

```
if (str == "Extract") {  
    cin >> L_Decor;  
    i = 0;  
    L_ID = "0";  
    for (int q = 1; q < 301; q++) {  
        ID_Store[q] = "0"; //存储对应花色的序号 (初始化均赋予无意义的"0")  
    }  
    while (L_ID != "NuLL") { //在序号取出为NuLL时说明已经全都取完了, 直接终止进程  
        L_ID = GetElemID(L, L_Decor); //根据花色取卡片, 取到对应的从牌堆删除  
        if (L_ID == "NuLL") {  
            break;  
        }  
        ++i;  
        ID_Store[i] = L_ID;  
    }  
}
```

将取出的牌依次放到牌堆顶:

```
ShuzuTidy(L, ID_Store, i); //按序号升序排序  
for (int j = 1; j <= i; j++) {  
    SqListInsertInput(L, j, L_Decor, ID_Store[j]); //将ID_Store[]中的内容依次放到牌堆  
}
```

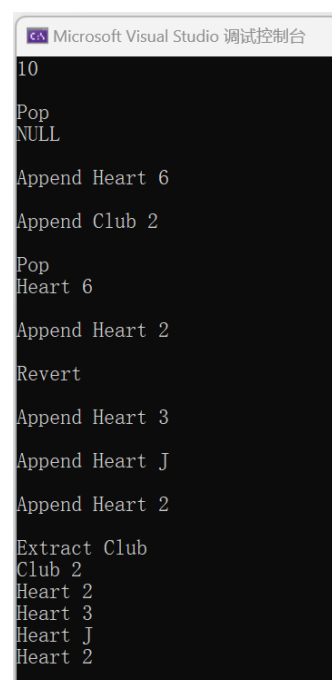
其中 ShuzuTidy()函数核心是一个冒泡排序:

现将对应卡牌的序号映射为 1~13 的数字, 并保存到数组 Temp[]中, 对 Temp[]冒泡排序:

```
//冒泡排序
for (int i = 1; i <= k - 1; i++) {
    for (int j = 1; j <= k - i; j++) {
        if (Temp[j] >= Temp[j + 1]) {
            t = Temp[j];
            Temp[j] = Temp[j + 1];
            Temp[j + 1] = t;
        }
    }
}
```

将排序后的 Temp[]结果反向映射回来保存到原卡牌序号的数组中即可。

2.5.5 调试分析（遇到的问题解决方法）



```
Microsoft Visual Studio 调试控制台
10
Pop
NULL
Append Heart 6
Append Club 2
Pop
Heart 6
Append Heart 2
Revert
Append Heart 3
Append Heart J
Append Heart 2
Extract Club
Club 2
Heart 2
Heart 3
Heart J
Heart 2
```

测试样例通过。在 OJ 平台上测试全部测试点均通过。

2.5. 总结和体会

本题除了使用顺序表编写, 也可用链表实现。本题借助顺序表实现, 在执行 reverse 反向指令上效率更高, 因为顺序表可以随机访问表中的元素。另外本题在编写顺序表功能的时候可以考虑功能可拓展性。如题目中要求的是删除牌堆顶元素以及添加到牌堆顶端, 但是可以直接编写插入到任意位置的功能, 方便维护和灵活修改。

3.实验总结

本次实验的核心内容是线性表，总共包括顺序表和链表两类。两者都很经典的线性存储的数据结构，但在存储特点、修改方面的效率都有不同，适用于不同场合。

其中顺序表的内存空间是连续的，且可以直接随机访问表中的任意一个序号的元素，但是在表中间删除、插入等方面需要借助循环覆盖，运行效率低，在表尾增删效率高；此外顺序表中经常会出现空余空间，空间利用效率低。总的来看可以当做可以存储任意数据类型元素的数组使用。适用于高频次访问表中的元素的任务中，如本次实验中**题目 1 轮转数组**、**题目 2 学生信息管理**。

而链表的内存空间不连续的，有可能会出现问题，而空间利用效率高，在表中就占内存，不在表中的结点可以直接释放内存空间，且在表中插入、删除数据只需要简单调整 next 指针指向。但是不支持直接随机访问任意位置的数据，需要从头结点开始依次向后遍历。链表适用于需要经常扩充缩减表内容的任务，如**题目 3 一元多项式的相加和相乘**。

而对于**题目 5 扑克牌游戏**，借助链表与顺序表均可以实现，两者各有优势。因为题目中所有的添加和删除都是在牌堆底部，两者效率都很高。但是在 reverse 反向操作中，由于顺序表可以直接随机访问元素，因此顺序表实现效率更高；在 extract 中，涉及到多个中间元素的删除，因此链表效率更高。由此可见具体选择哪种数据结构要充分考量任务的特点。

此外本次任务还涉及到了简单的算法设计。**题目 4 求级数**涉及到了高精度算法，本质是模拟自然运算的进位思想，将 C++ 可存储、运算的数据范围进一步扩大。**题目 1 轮换数组**采用了三种方法，在空间复杂度、时间复杂度上各有不同。对于大部分题目往往都有多种解决的算法，这时就要注意分析时间复杂度和空间复杂度并综合考量代码可读性选择更好的算法完成任务。

另外通过此次实验，也认识到将各个功能通过函数、类单独封装成可调用的接口可以大大提高代码的可读性并且方便后期维护，在定义大量的函数和类的时候，要特别关注全局变量、局部变量以及动态表动态变量的使用，以免因为变量的范围不同导致程序出错。