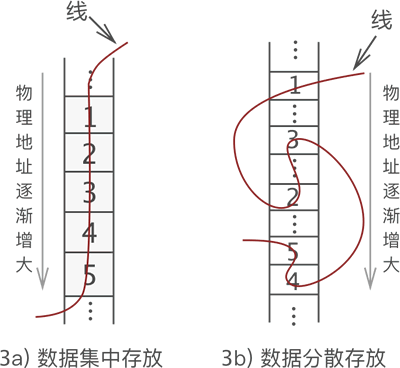
**作业 HW1线性表 实验报告**

姓名：刘禹锡 学号：2256666 日期：2023年10月7日

1. **涉及数据结构和相关背景**

HW1线性表的作业涉及了线性表的使用，线性表分为顺序存储结构和链式存储结构，如图 3a)所示，将数据依次存储在连续的整块物理空间中，这种存储结构称为顺序存储结构（简称顺序表）；如图 3b) 所示，数据分散的存储在物理空间中，通过一根线保存着它们之间的逻辑关系，这种存储结构称为链式存储结构（简称链表）

针对不同的问题，选择不同的数据结构，以及算法来高效地完成一些问题。



**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
* 输出： 轮转后的顺序表中的元素

**2.1.3 数据结构设计**

class SequenceList {

private:

    int\* data;    *//用于存储数据的数组*

    int size;     *//顺序表的大小*

    int capacity; *//数组的容量*

public:

    SequenceList(int initialCapacity = 10); *//初始化数组*

    ~SequenceList();                        *//析构*

    void push\_back(int value);              *//向顺序表中加元素*

    void rotate(int k);                     *//轮转数组*

    void print();                           *//打印*

};

使用顺序表来储存数组数据，并使用动态内存来扩充数组。

**2.1.4功能说明（函数、类）**

1. 插入元素

*/\*\**

\* @brief           在顺序表末尾添加一个元素

\* @param value     要添加的元素的值

\*/

void push\_back(int value) {

    如果数组已满，将容量扩展为当前容量的两倍，以减少扩展次数

        int\* newData = new int[capacity]; *// 创建一个新的、更大的数组*

        for (int i = 0; i < size; ++i)

            newData[i] = data[i]; *// 将旧数组中的元素复制到新数组中*

        释放旧数组的内存，并将指针指向新数组

    data[size++] = value; *// 将新元素添加到数组末尾，并递增 size*

}

1. 轮转数组

轮转数组函数rotate将数组中的元素向右循环移动 k 个位置，其中 k 是传递给函数的参数。这个算法使用了额外的临时数组来存储需要移动的元素，然后将元素移动到新的位置。

* 由于开辟了额外的空间来储存需要移动的元素，空间复杂度为O(n)
* 时间复杂度为O(k) + O(n) + O(k)，可以简化为O(n)

*/\*\**

\* @brief 旋转顺序表中的元素

\* @param k 旋转的步数

\*/

void rotate(int k) {

k = k % size; *// 确保旋转步数在有效范围内*

if (k == 0) { 如果 k 为 0，不需要旋转 }

int\* temp = new int[size]; *// 创建一个临时数组来存储旋转的元素*

for (int i = 0; i < k; ++i)

temp[i] = data[size - k + i]; *// 复制旋转的元素到临时数组*

for (int i = size - 1; i >= k; --i)

data[i] = data[i - k]; *// 移动数组中的元素来完成旋转*

for (int i = 0; i < k; ++i)

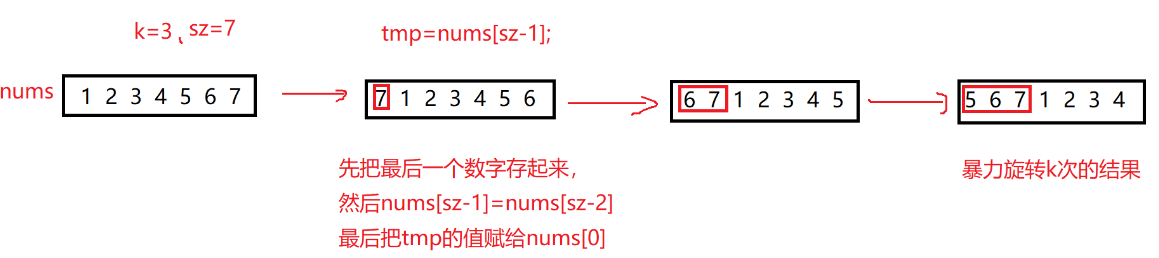
data[i] = temp[i]; *// 将旋转后的元素复制回原数组*

delete[] temp; *// 释放临时数组的内存*

}

**2.1.5 调试分析（遇到的问题和解决方法）**

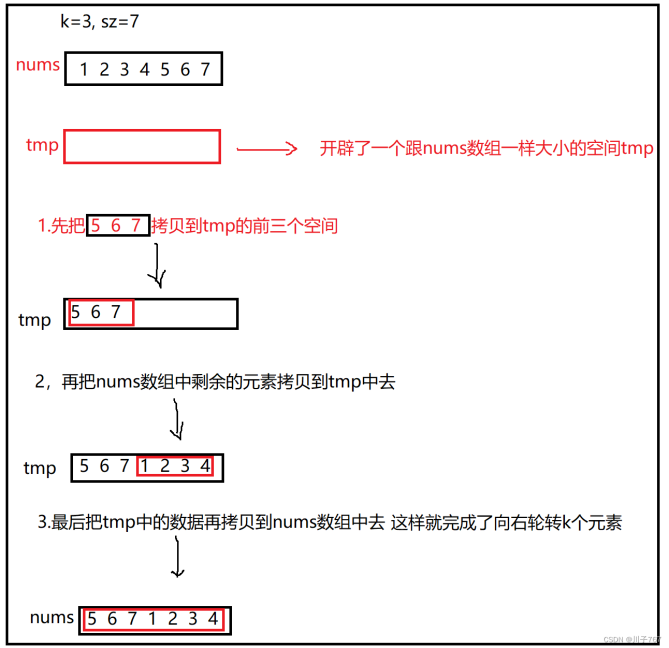
1. 暴力求解超时



**时间复杂度分析:**算法中有两个嵌套的循环：外部的循环是 k 次轮转的循环，内部的循环在每次轮转中将数组中的元素向右移动一位。对于外部循环，它的时间复杂度是 O(k)。内部循环时间复杂度O(len)

总时间复杂度为O(k \* size)

**解决方法:**



开拓额外内存来储存需要移动的元素，其余元素依次向右移动k个单位，如图(图源网络)。

开辟一个与原数组大小相等的空间tmp，将需要前移的元素先放到tmp前面，再将数组中剩余的元素拷贝到tmp中，这样就完成了向右轮转。

1. 容器扩容的思考

由于数组长度，以及要轮转位数未知，所以开辟的数组扩充也未知，关于扩容方法，有两种，一种递增式扩容，一种倍增式扩容，两者分摊时间成本如下：

递增扩容：递增扩容每次增加固定长度的可用空间，所需要的时间复杂度为0+I+2I+⋯+(m−1)I=O(n ^2)，因此每次扩容的分摊成本为 O(n)

倍增扩容：倍增扩容每次增加现有可用空间的的1倍，因此所需的时间复杂度为：1+2+4+8+⋯+2m=O(n)

因此每次扩容的分摊成本为O(1)，显然，时间成本较递增扩容减少。

但是，从另一个角度，空间复杂度考虑，用装填因子这一指标来衡量。装填因子是向量的实际规模与其内部数组容量的比值。显而易见，递增式的装填因子当空间较大时，趋近于100%，而倍增式我们可以保证的是>50%，换句话说，倍增扩容其实是用空间换取了时间。

**2.1.6 总结和体会**

体会到了不同算法的效率不同，减少时间复杂度可以提高算法效率，也学习了不同的容器扩容方法。

**2.2 学生信息管理**

**2.2.1 问题描述**

顺序表是指采用顺序存储结构的线性表，它利用内存中的一片连续存储区域存放表中的所有元素。可以根据需要对表中的所有数据进行访问，元素的插入和删除可以在表中的任何位置进行。

顺序表的基本操作，包括顺序表的创建，第i个位置插入一个新的元素、删除第i个元素、查找某元素、顺序表的销毁。

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**

* 第1行是学生总数n
* 接下来n行是对学生信息的描述，每行是一名学生的学号、姓名，用空格分割；(学号、姓名均用字符串表示,字符串长度<100)
* 接下来是若干行对顺序表的操作：(每行内容之间用空格分隔)
* 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。每个操作都在上一个操作的基础上完成。

**2.2.3 数据结构设计**

*// 定义一个名为 Student 的结构体，用于存储学生信息*

struct Student {

string no; *// 学号*

string name; *// 姓名*

};

选择使用动态数组（顺序表）来存储学生信息。通过动态内存的申请构造出顺序表。

**2.2.4功能说明（函数、类）**

1. 顺序表的创建与数据读入

cin >> n; *// 读入学生数量*

*// 创建一个动态数组 studentList 来存储学生信息，初始大小为 n*

Student\* studentList = new Student[n];

*// 逐个读取学生的学号和姓名，并存储在数组中*

for (int i = 0; i < n; ++i) {

cin >> studentList[i].no >> studentList[i].name;

currentSize++; *// 递增学生数量*

}

1. 插入操作

* 时间复杂度：O(n)，因为在最坏情况下需要复制 n 个学生的数据。
* 空间复杂度：O(n)，因为需要额外的内存来存储新的动态数组。

cin >> i >> stuNo >> stuName; *//读入插入学生信息*

if (i < 1 || i > currentSize + 1) { *// 插入操作，检查位置是否合法*

cout << -1 << endl; *// 位置不合法，输出错误消息*

}

else {

*//创建新数组 newStudentList，比原数组+1*

Student\* newStudentList = new Student[currentSize + 1];

for (int j = 0; j < i - 1; j++) { *// 复制插入位置前的元素*

newStudentList[j] = studentList[j];

}

newStudentList[i - 1] = { stuNo, stuName }; *// 插入新学生信息*

for (int j = i - 1; j < currentSize; j++) { *// 复制插入位置后的元素*

newStudentList[j + 1] = studentList[j];

}

currentSize++; *// 更新当前学生数量，释放原数组，指向新数组*

delete[] studentList; *// 删除原数组*

studentList = newStudentList;

cout << 0 << endl; *// 插入成功*

}

1. 删除操作

* 时间复杂度：O(n)，同样需要复制 n 个学生的数据。
* 空间复杂度：O(n)，因为需要额外的内存来存储新的动态数组。

*// \*\*检查位置是否合法\*\**

*// \*\*合法执行下列\*\**

*// 创建新数组 newStudentList，比原数组-1*

Student\* newStudentList = new Student[currentSize - 1];

for (int k = 0; k < j - 1; k++) { *// 复制删除位置前的元素*

newStudentList[k] = studentList[k];

}

for (int k = j; k < currentSize; k++) { *// 复制删除位置后的元素*

newStudentList[k - 1] = studentList[k];

}

currentSize--; *// 更新当前学生数量，释放原数组，指向新数组*

delete[] studentList;

studentList = newStudentList;

1. 查找操作

* 时间复杂度：O(n)，因为在最坏情况下需要遍历整个学生列表来查找。
* 空间复杂度：O(1)，只需要有限的局部变量。

string searchType, searchValue; *//读入搜索目标*

cin >> searchType >> searchValue;

*// 遍历学生数组，根据搜索类型和值查找学生信息*

for (int i = 0; i < currentSize; ++i) {

if ((searchType == "name" && studentList[i].name == searchValue) ||

(searchType == "no" && studentList[i].no == searchValue)) {

foundIndex = i + 1;

cout << foundIndex << " " << studentList[i].no << " " << studentList[i].name << endl;

break; *// 找到匹配的学生信息后，结束遍历*

}

}

*// 输出结果或错误信息*

if (foundIndex == -1 && searchType == "no") {

cout << -1 << " " << " " << endl; *// 未找到学号匹配的学生信息，输出 -1*

}

*// \*\*未找到姓名匹配的学生信息，输出 -1\*\**

**2.2.5 调试分析（遇到的问题和解决方法）**

1. 顺序表写成链表，顺便比较下两者优缺点

* 储存方式:顺序表使用连续一段内存，如要插入元素，特别是在中间插入元素必须要新开辟一块内存，内存的复制很耗时；而链表可以直接添加节点，节省了时间开支
* 访问元素：顺序表由于是一块连续的内存空间，所以可以直接通过其下标定位其元素，时间复杂度为O(1)，而链表必须要遍历查找，时间复杂度为O(n)

1. 缺少一些异常处理

例如删除操作，需要检查位置是否合法，同时要及时修改一些标记，例如学生数量等。最后也要释放掉删除的节点，避免内存遗留。

**2.2.6 总结和体会**

通过本题，我加深了对顺序表与链表的理解，在犯错误中体会到了两者的优点与缺点，同时，也提醒了我，编写一个健壮的程序需要考虑一些异常情况。

**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((p1,e1),(p2,e2),...,(pm,em))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项，p1 e1 p2 e2 ...，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项，p1 e1p2e2...，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**

*//定义多项式节点*

struct Node {

int coef; *//系数项*

int exp; *//指数项*

Node\* next; *//存储下一节点*

Node(int c, int e) : coef(c), exp(e), next(NULL) {} *//节点赋值*

};

typedef Node\* link; *//节点别名*

**2.3.4功能说明（函数、类）**

1. 插入节点

* **insertNode** 的时间复杂度是 O(n)。
* 空间复杂度主要是由新节点 **newnode** 占用的空间，因此是 O(1)。

*/\**

\* @brief 插入节点到链表中

\* @param head 链表的头指针，通过引用方式传递链表头

\* @param coefficient 节点的系数

\* @param exponent 节点的指数

\*/

void insertNode(link& head, int coefficient, int exponent)

link newnode = new Node(coefficient, exponent);*//创建新节点*

if (!head) \*\*\* *// 如果链表为空，将新节点设为头节点*

else

link curr/prev; *// 初始化当前位置指针，和上一位置指针*

*// 遍历链表，查找插入位置，按照指数递增顺序*

while (curr 不为空 且 curr的指数 < exponent)

prev = curr; *// 保存当前位置*

curr = curr->next; *// 移动到下一个节点*

if (curr 不为空 且 curr的指数 == exponent)

curr->coef += coefficient; *// 合并同一指数项的系数*

删除新节点(newnode); *// 删除新节点，因为已合并*

else *// 新节点的指数未出现过*

if (!prev)

newnode->next = head; *// 将新节点插入为头节点*

head = newnode;

else

prev->next = newnode; *// 插入新节点到prev和curr之间*

newnode->next = curr;

1. 多项式相加

* 该函数通过迭代遍历两个多项式链表，将它们合并成一个新的链表。
* 在最坏情况下，需要遍历两个输入链表的所有项，所以时间复杂度是 O(n1 + n2)
* 删除多项式项时，需要遍历结果链表，O(n1 + n2) 。
* 空间复杂度取决于结果链表 **result** 的长度，即 O(n1 + n2)。

*/\*\**

\* @brief 将两个多项式相加

\* @param poly1 第一个多项式的头节点指针

\* @param poly2 第二个多项式的头节点指针

\* @return 返回结果多项式的头节点指针

\*/

link addPolynomials(link poly1, link poly2)

*//初始化curr1，curr2，分别代表两个多项式*

while (curr1和curr2都不为空)

if (curr1和curr2指数相同)

*// 计算系数和，如果不为零则插入*

insertNode(result, sum\_coeff, curr1->exp);

移动 curr1 和 curr2;

else if (curr1->exp < curr2->exp)

*// 处理第一个多项式的当前项，目的是把支书较小项先插入*

insertNode(result, curr1->coef, curr1->exp);

移动 curr1;

else

*// 处理第二个多项式的当前项*

insertNode(result, curr2->coef, curr2->exp);

移动 curr2;

*// 处理剩余项，将两个多项式剩余的项添加到结果多项式后面*

while (不为空)

insertNode(result, curr？->coef, curr？->exp);

移动 curr;

*清除系数为零的项*

返回 result;

1. 多项式相乘

* 该函数执行多项式的乘法，通过嵌套循环遍历两个多项式链表的所有项。
* 在最坏情况下，需要遍历两个输入链表的所有项，所以时间复杂度是 O(n1 \* n2)
* 删除多项式项时，需要遍历结果链表，这也需要 O(n1 \* n2) 的时间。
* 空间复杂度取决于结果链表 **result** 的长度，即 O(n1 \* n2)

*/\*\**

\* @brief 将两个多项式相乘

\* @param poly1 第一个多项式的头节点指针 @param poly2 第二个多项式的头节点指针

\* @return 返回结果多项式的头节点指针

\*/

link multiPolynomials(link poly1, link poly2)

link result，curr1*//初始化结果多项式指针和当前位置指针*

while (curr1不为空)

link curr2 = poly2; *// 当前位置指针，用于遍历第二个多项式*

while (curr2也不为空) *//依次将多项式1和多项式2的每一项相乘，包括系数相乘，指数相加*

int result\_exp = curr1->exp + curr2->exp; *// 计算结果项的指数*

int result\_coeff = curr1->coef \* curr2->coef; *//计算结果项的系数*

while (结果项指向不为空)

if (existing->exp == result\_exp) *// 如果结果项已存在*

existing->coef += result\_coeff; *// 合并同一指数项的系数*

found = true;

break;

if (!found) *// 如果结果项不存在，插入新项到结果多项式*

insertNode(result, result\_coeff, result\_exp);

curr2 = curr2->next; *//移向下一项*

curr1 = curr1->next; *//移向下一项*

link prev = NULL;

link curr = result;

\*清除结果多项式中系数为零的项\*

返回 result;

1. 反转链表

* 该函数通过迭代遍历链表并反转节点的指针，需要遍历整个链表一次。时间复杂度是 O(n)。
* 空间复杂度是 O(1)，因为只使用了常数额外空间

*/\*\**

\* @brief 反转链表

\* @param head 链表的头节点指针

\* @return 返回反转后链表的头节点指针

\*/

link reverseList(link head)

{

link prev and next = NULL; *// 前一个节点指针，下一个节点指针，初始化为空*

link curr = head; *// 当前节点指针，指向链表的头节点*

while (curr != NULL) { *// 当前节点不为空时*

next = curr->next; *// 保存下一个节点的指针*

curr->next = prev; *// 将当前节点指向前一个节点，反转连接*

prev = curr; *// 更新前一个节点指针为当前节点*

curr = next; *// 更新当前节点为下一个节点*

}

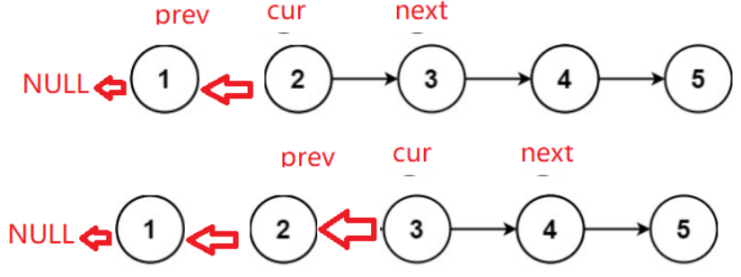
返回 prev; *// 返回反转后链表的头节点指针*

}

**2.3.5 调试分析（遇到的问题和解决方法）**

1. 多项式乘法后得到的目标多项式未按照指数排序

使用迭代法逆转链表，通过迭代遍历链表，将每个节点的 next 指针指向它的前一个节点，从而反转链表的方向。这样，在遍历完整个链表后，原来的链表变成了倒置的链表，最后返回新链表的头节点（原链表的尾节点）作为反转后的链表的新头部。时间复杂度上是线性的（O(n)，n 是链表的长度），效率较高。



1. 细节优化

提高代码重用度，将复用的代码整合为函数调用。处理多项式加法及乘法出现系数为0的情况。在编写时没有考虑内存回收，对于申请的动态内存，必须将其回收。对于链表，必须遍历链表，将每一个节点都释放掉。

**2.3.6 总结和体会**

本题学习了单链表的基本操作，例如遍历，插入，删除等，并认识到单链表中指针的重要性，边界的判断与处理。

对于算法效率，应该尽量降低时间复杂度和空间复杂度，来提高效率。

**2.4 求级数**

**2.4.1 问题描述**

输出级数A+2A^2+3A^3+...+NA^N =

**2.4.2 基本要求**

高精度计算以及时间复杂度要低，不能超时。

**2.4.3 数据结构设计**

分析:由于使用int，longlong等数据类型会溢出，并且频繁运算时间复杂度高导致超时，转而想到string型，string型包含许多内置操作，并且类似动态数组可以自动扩充，不会浪费空间。将级数以及结果转换至string储存，可以保证不溢出。在计算时将string在转化为整数型计算。

**result**字符串：这个字符串用于存储最终的计算结果，它的初始值是 "0"。

**term**字符串：这个字符串用于存储级数中的每一项。

string result = "0";

string term = "1";

**2.4.4功能说明（函数、类）**

1. 级数相加

* 该函数实现两个字符串的相加操作，采用类似手工相加的方式，从字符串的末尾开始逐位相加，考虑进位。
* 时间复杂度：O(max(N, M))，其中 N 和 M 分别是 **num1** 和 **num2** 的长度。
* 空间复杂度：O(max(N, M))，用于存储结果字符串。

*/\*\**

\* @brief 将两个字符串表示的整数相加

\* @param num1 第一个整数的字符串表示

\* @param num2 第二个整数的字符串表示

\* @return 相加后的结果的字符串表示

\*/

string addStrings(string num1, string num2) {

int carry = 0; (初始化进位为 0 ) string result = ""; (初始化结果字符串为空)

int i 或 j = num.length() - 1; *// 获取第一个和第二个整数字符串的最后一个字符的索引*

*// 从右到左遍历两个整数的字符串，同时考虑进位*

while (i >= 0 || j >= 0 || carry > 0) {

int x 或 y = (i或j >= 0) ? num[i或j] - '0' : 0; *// 将字符转换为整数，或者为 0*

int sum = x + y + carry; *// 计算当前位的和，包括进位*

result = to\_string(sum % 10) + result; *// 将和的个位数添加到结果字符串的最前面*

carry = sum / 10; *// 更新进位，以便下一次迭代使用*

*//更新 i j*

}

return result; *// 返回相加后的结果的字符串表示*

}

1. 级数相乘

* 时间复杂度分析：
* 在这个函数中，有两个嵌套的 **for** 循环。循环次数取决于两个字符串长度
* 因此，整个函数的时间复杂度是 O(len1 \* len2)
* 空间复杂度分析：
* 函数中创建了一个名为 **result** 的字符串，其大小为 **len1 + len2**，用于存储相乘的结果。
* 因此，整个函数的空间复杂度主要由 **result** 字符串决定，为 O(len1 + len2)。
* 虽然该函数的时间复杂度是 O(len1 \* len2)，但它的空间复杂度相对较低，因为它没有创建大型的中间数组来存储临时结果。这种实现方式对于大数相乘来说是比较高效的。

*/\*\**

\* @brief 将两个字符串表示的整数相乘

\* @param num1 第一个整数的字符串表示

\* @param num2 第二个整数的字符串表示

\* @return 相乘后的结果的字符串表示

\*/

string multiplyStrings(string num1, string num2)

int len？ = num？.length(); *// 获取两个整数字符串的长度*

string result(len1 + len2, '0'); *// 初始化结果字符串为零串，长度为 len1 + len2*

*// 从右到左遍历第一个整数的字符串*

for (int i = len1 - 1; i >= 0; i--) {

int carry = 0; *// 初始化进位为 0*

*// 从右到左遍历第二个整数的字符串*

for (int j = len2 - 1; j >= 0; j--) {

int res = num1与num2当前位相乘 + (result[i + j + 1] - '0') + carry;

*// 计算当前位的乘积，包括进位*

carry = res / 10; *// 更新进位，以便下一次迭代使用*

result[i + j + 1] = res % 10 + '0'; *// 更新结果字符串的当前位置*

}

result[i] += carry; *// 将最终进位添加到结果的当前位置*

}

*// 查找第一个不为 '0' 的位置*

int startPos = 0;

while (startPos < result.length() && result[startPos] == '0') startPos++;

if (startPos < result.length())

return result.substr(startPos); *// 返回结果字符串的子串，从 startPos 开始*

else 如果结果全为零，则返回 "0"

**2.4.5 调试分析（遇到的问题和解决方法）**

1. 超时

在级数N较大时，超出常见数据类型的最大范围，例如int只能20分，扩大到longlong40分，同时为了满足精度，不可以使用double型，因此考虑使用string字符串来储存。

1. 运算问题

由于题目只要求输出最后结果，而不考虑过程，所以可以使用叠加的方法，将每一项存在同一个字符串中，将结果存在另一个字符串中，每次让结果+新一项，同时，更新这一项的数据，从而实现叠加。

**2.4.6 总结和体会**

这道题关键在于数据结构的选择，按理本题应该运用链表的知识点，但是个人认为string更便捷一些，并且也好操作，但是要注意字符型和整型的区别，两者之间转化需要 -‘0’。

通过这道题，我体会到了数据结构选择的重要性。

**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 ≤ n ≤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**

输入:第一行输入一个整数n，表示命令的数量。接下来的n行，每一行输入一个命令。

输出：输出若干行，每次收到Pop指令后输出一行（花色和数子或NULL），最后将牌堆中的牌从牌堆顶到牌堆底逐一输出（花色和数字），若牌堆为空则输出NULL

**2.5.3 数据结构设计**

**struct Card {**

**string flower; *// 扑克牌的花色***

**string number; *// 扑克牌的点数或数字***

**Card(string f, string n) : flower(f), number(n) {} *//构造函数，用于初始化 Card 结构***

**};**

**2.5.4功能说明（函数、类）**

1. Append添加操作

* 时间复杂度：O(1)，只需要将一张卡片添加到卡片数组中。
* 空间复杂度：O(1)，只有少量局部变量。

***/\*\****

**\* @brief 向扑克牌集合中添加一张牌**

**\* @param cards 扑克牌集合（vector）**

**\*/**

**void Append(vector<Card>& cards) {**

***//读入花色和数字，检查花色和数字是否有效***

**if (花色和数字都有效) {**

***// 创建一张新的扑克牌并添加到集合中***

**cards.push\_back(Card(flower, number));**

**}**

***// 花色或数字无效，输出 "NULL"***

**}**

1. Extract提取操作

* 时间复杂度：O(n)，其中n是卡片数组的长度
* 空间复杂度：O(m)，其中m是满足条件的卡片数量
* 在函数内创建了一个临时的 **extractedCards** 数组来存储这些卡片。

***/\*\****

**\* @brief 从扑克牌集合中提取指定花色的牌并重新排序**

**\* @param cards 扑克牌集合（vector）**

**\*/**

**void Extract(vector<Card>& cards) {**

**读入花色**

**vector<Card> extractedCards; *// 存储提取出的牌的临时向量***

***// 遍历扑克牌集合并提取指定花色的牌***

**for (auto it = cards.begin(); it != cards.end(); )**

**if (it->flower == flower) {**

**extractedCards.push\_back(\*it); *// 添加匹配的牌到提取牌的向量***

**it = cards.erase(it); *// 从原集合中移除匹配的牌***

**}**

**else 移动it**

***// 函数对提取出的牌进行排序，先按花色再按数字***

**sort(extractedCards.begin(),extractedCards.end(),比较a，b) {**

**如果（花色相同） {**

**numA = 转换数字(a.number);**

**numB = 转换数字(b.number);**

**返回 numA < numB;**

**}**

**return a.flower < b.flower;**

**});**

***// 将提取出的牌插入回原集合***

**cards.insert(cards.begin(),extractedCards.begin(),extractedCards.end());**

**}**

1. Revert反转

* 时间复杂度：O(n)，其中n是卡片数组的长度。函数中使用了**reverse**函数来颠倒整个卡片数组
* 空间复杂度：O(1)，没有额外的内存分配

***/\*\****

**\* @brief 反转扑克牌集合的顺序**

**\* @param cards 扑克牌集合（vector）**

**\*/**

**void Revert(vector<Card>& cards)**

**reverse(cards.begin(), cards.end()); *// 使用 STL 函数反转牌的顺序***

1. Pop弹出

* 时间复杂度：O(1)，因为只是删除数组的第一个元素，并且不需要遍历整个数组
* 空间复杂度：O(1)，因为只有少量局部变量

*/\*\**

\* @brief 弹出并打印扑克牌集合中的第一张牌

\* @param cards 扑克牌集合（vector）

\*/

void Pop(vector<Card>& cards) {

if (牌堆不空)

*// 打印第一张牌的花色和数字*

cards.erase(cards.begin()); *// 从集合中移除第一张牌*

else *// 如果集合为空，则打印 "NULL"*

}

**2.5.5 调试分析（遇到的问题和解决方法）**

1. 未处理牌号为字母的牌

测试数据前四个没有出现字母牌(A,J,Q,K)，因此可以通过，但是后几个数据中，含有字母牌，而我的程序调用比较函数比较牌号时，只考虑了整数的比较，如果出现字母，则字母一定会比数字要大，存在逻辑错误。

解决方法：

if (a.number == "A") numA = 1;

else if (a.number == "J") numA = 11;

else if (a.number == "Q") numA = 12;

else if (a.number == "K") numA = 13;

1. 细节处理

处理非法命令，非法花色，非法牌号。

1. 使用vector容器

vector相当于动态数组，并且内置许多函数，可以实现很多较复杂操作，例如排序等，而且，内置的sort函数可以自定义，使用起来很便捷。

vector有以下优点：动态数组容器，随机访问，可迭代，空间开销少。

**2.5.6 总结和体会**

这道题给了我很多启发，最主要是vector容器的使用，十分便捷，同时了解到C++内置的STL库中还有许多有趣有用的函数。

**3. 实验总结**

1. 数据结构的应用：通过实验中涉及的不同题目，我应用了各种数据结构，如顺序表、链表和vector容器等，更深地理解了各种数据结构优缺点以及适用情形。

2. 算法效率:通过实验，我更深刻地理解了时间复杂度的概念，学会了如何分析算法时间复杂度，空间复杂度，并通过比较，选择出效率更高的算法来解决问题，优化我的代码。

3. 同时，我认识到自己还有许多不足，我的代码健壮性不够，应该更关注异常情况的处理。同时应该提高代码复用度，更多使用函数和模块化编程。对于类和对象的使用，也应当在日后更多地使用。