作业 HW1 实验报告

姓名：贺胡鸣 学号：2353626 日期：2025 年 10 月 28 日

1. 涉及数据结构和相关背景

1.1 背景介绍

线性结构是计算机科学中最基本、最常用的一种数据结构，它用于描述数据元素之间“一个接一个”的线性排列关系。在实际的软件开发和算法设计中，从简单的数据存储（如待办事项列表）、到复杂系统的底层实现（如操作系统的进程就绪队列），线性结构的应用无处不在。

线性表（Linear List）作为线性结构的一种典型代表，是本实验的核心数据结构。通过本实验，旨在深入理解线性表的逻辑结构、物理存储结构及其基本操作的实现原理与效率，为后续更复杂数据结构的学习和应用打下坚实的基础。

1.2 核心数据结构：线性表

1.2.1 线性表的定义

线性表是由 n（n≥0）个具有相同数据类型的元素构成的有限序列。其中，元素的个数 n 定义为线性表的长度。当 n=0 时，线性表为空表。

线性表中的元素具有逻辑上的顺序性，除了第一个元素外，每个元素有且仅有一个直接前驱；除了最后一个元素外，每个元素有且仅有一个直接后继。这种结构可以形式化地表示为：

$$(a_1, a_2, a_3, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$$

其中，a_1是表头元素，a_n是表尾元素。a_{i-1}是 a_i 的直接前驱，a_{i+1}是 a_i 的直接后继。下标 i 表示元素在线性表中的位序。

1.2.2 线性表的基本操作

一个数据结构通常由其逻辑结构、物理存储结构和基本操作共同定义。线性表支持一系列基本操作，主要包括：

-初始化 InitList(&L)：构造一个空的线性表 L。

-销毁 DestroyList(&L)：释放线性表 L 所占用的内存空间。

-插入 ListInsert(&L, i, e)：在线性表 L 的第 i 个位置插入新元素 e。

-删除 ListDelete(&L, i, &e)：删除线性表 L 中第 i 个位置的元素，并用 e 返回其值。

-按值查找 LocateElem(L, e)：在线性表 L 中查找值为 e 的元素，返回其位序。

-按位查找 GetElem(L, i)：获取线性表 L 中第 i 个位置元素的值。

-求表长 Length(L)：返回线性表 L 的长度，即数据元素的个数。

-判空 Empty(L)：判断线性表 L 是否为空，若为空返回 true，否则返回 false。

-输出操作 PrintList(L)：按前后顺序输出线性表 L 的所有元素值。

1.2.3 线性表的存储结构

线性表的逻辑结构在计算机中可以通过不同的物理存储结构来实现，两种最主要的实现方式是：顺序存储 和 链式存储。

(1) 顺序表（Sequential List）

顺序表采用顺序存储结构，即将线性表的元素顺序地存储在一组地址连续的存储单元中，通

常用数组来实现。

特点：

-随机访问：通过首地址和元素位序可以在 O(1)时间内访问到指定元素。

-物理结构与逻辑结构一致：逻辑上相邻的元素，在物理存储上也相邻。

-存储密度高：所有存储空间都用于存放数据元素。

缺点：

-插入/删除操作效率低：平均需要移动半个表的元素，时间复杂度为 O(n)。

-容量固定：静态分配时，表长一经定义难以更改。动态分配虽可扩容，但需要移动大量元素，开销较大。

(2) 链表（Linked List）

链表采用链式存储结构，它通过一组任意的存储单元来存储线性表中的数据元素，这些存储单元可以是连续的，也可以是不连续的。为了表示元素之间的逻辑关系，每个结点除了存储元素本身的信息（数据域）外，还需要存储指向其后继结点的地址（指针域）。

特点：

-非随机访问：必须从头结点开始顺序查找，访问元素的时间复杂度为 O(n)。

-动态分配空间：无需预先分配固定大小，在需要时申请新结点，使得内存利用更充分。

-插入/删除操作效率高：在已知操作位置后，仅需修改指针，时间复杂度为 O(1)。

缺点：

-存储密度较低：因为每个结点都包含指针域，产生了额外的空间开销。

-失去了顺序存储的随机存取特性。

链表根据其指针域的设置方式，又可分为单链表、双链表、循环链表等，提供了不同场景下的灵活性与功能。

2. 实验内容

2.1 问题一 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

题目要求输入：

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

要求输出：

轮转后的顺序表中的元素

2.1.3 数据结构设计

在本问题的解决方案中，我们采用顺序表（Sequential List）作为核心数据结构来实现数组的轮转操作。顺序表是一种通过数组实现的线性表结构，其特点是将数据元素顺序地存储在一组地

址连续的存储单元中。这种连续存储的特性使得顺序表具有优秀的随机访问性能，能够在常数时间复杂度 O(1)内访问任意位置的元素，这对于轮转操作中需要频繁按索引访问元素的需求至关重要。此外，顺序表的连续存储布局具有良好的空间局部性，有利于 CPU 缓存机制的发挥，能够显著提高数据访问的效率。

从实现角度来看，我使用 C++标准库中的 vector 容器来具体实现顺序表。vector 容器动态管理内存空间，既保持了顺序表的随机访问优势，又提供了自动内存管理的便利性。在轮转操作过程中，我们需要对顺序表进行元素的重新排列和位置交换，顺序表的连续存储特性使得这些操作能够高效完成。特别值得注意的是，虽然顺序表在中间位置插入和删除元素的时间复杂度较高，但本问题主要涉及元素的位置交换和重新排列，这些操作在顺序表上能够获得较好的性能表现。

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    int n, k;
    cin >> n >> k;
    vector<int> vec(n), ans(n);
```

2.1.4 功能说明（函数、类）

```
for (int i = 0; i < n; i++) {
    cin >> vec[i];
    if (i + k <= n - 1)
        ans[i + k] = vec[i];
    else
        ans[i + k - n] = vec[i];
}
for (int i = 0; i < n; i++)
    cout << ans[i] << " ";
cout << endl;
return 0;
```

2.1.5 调试分析（遇到的问题和解决方法）

在实现轮转数组算法的过程中，首先是在处理较大的轮转次数 k 时，当 k 超过数组长度 n 时会出现数组越界的问题。这主要是因为最初没有考虑到 k 可能远大于 n 的情况，导致直接使用 k 作为偏移量时访问了非法内存地址。第二个问题是在使用反转法时，边界条件的处理容易出错。特别是在三次反转的过程中，反转区间的起始和结束位置需要精确计算，稍有不慎就会导致元素错位或遗漏。通过详细的例子测试，比如对数组[1,2,3,4,5]分别测试 k=2、k=3、k=0 等边界情况，逐步调整反转的区间范围，最终确保了算法的正确性。

2.1.6 总结和体会

通过本次轮转数组的实现，我深刻认识到算法设计中边界条件处理的重要性。这道题的难点主要体现在几个方面：首先是数学建模能力，需要理解轮转操作的本质是位置的循环移位，进而想到取模运算来简化问题；其次是对空间复杂度的优化要求，从最初使用额外数组的直观解法，到最终采用三次反转的原地算法，体现了对空间效率的不断追求；最后是反转法中区间划分的精确性，需要仔细处理下标计算，确保不遗漏任何元素。在易错点方面，除了 k 大于 n 的边界情况外，还需要注意空数组、单元素数组、k=0 等特殊情况的处理。此外，算法效率的要求促使我们放弃最直观的暴力解法，转而寻求时间复杂度 O(n)和空间复杂度 O(1)的最优解。这次实践让我体会到，优秀的算法设计不仅在于解决问题，更在于如何高效、优雅地解决问题，同时还要具备严密的思维，考虑到所有可能的边界情况，这样才能写出健壮可靠的代码。

2.2 题目二 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

首先定义了学生信息结构体 student，包含两个字符串成员变量 no 和 name，分别用于存储学生的学号和姓名。

```
//定义学生信息结构体
struct student {
    string no;
    string name;
};
```

顺序表的核心数据结构通过 SeqList 类来实现，该类封装了顺序表的所有基本操作。在类的私有成员中，定义了三个关键数据成员：student* data 是一个指向动态数组的指针，用于存储所有学生信息；int capacity 表示顺序表的最大容量，确保不会发生数组越界；int length 记录当前顺序表中实际存储的学生数量。

```
//顺序表类的实现
class SeqList {
private:
    student* data; //存储学生数据的数组
    int capacity;  //顺序表的容量
    int length;    //当前元素的个数
```

构造函数 SeqList(int n)根据初始学生数量动态分配内存空间，析构函数~SeqList()负责在对象生命周期结束时释放动态分配的内存。

```
public:
    //构造函数
    SeqList(int n)
    {
        capacity = n + 10;
        data = new student[capacity];
        length = 0;
    }

    //析构函数
    ~SeqList()
    {
        delete[] data;
    }
```

2.2.4 功能说明

addStudent 函数负责在顺序表末尾添加新的学生信息，首先检查当前长度是否超过容量限

制，在确保有足够空间的情况下将学生信息存储到数组的相应位置，同时更新长度计数器。

```
//在末尾添加学生
void addStudent(const string& no, const string& name)
{
    if (length < capacity) {
        data[length].no = no;
        data[length].name = name;
        length++;
    }
}
```

insert 函数实现了在指定位置插入学生信息的功能，该函数首先进行严格的参数校验，确保插入位置 i 在合法范围内，同时检查顺序表是否已满。在通过校验后，通过 for 循环将插入位置及之后的所有元素向后移动一个位置，为新的学生信息腾出空间，然后将新数据插入到指定位置，最后更新顺序表长度。

```
//在第i个位置插入学生（位置从1开始）
int insert(int i, const string& no, const string& name)
{
    //位置不合法
    if (i<1 || i>length + 1)
        return -1;
    //表已满
    if (length >= capacity)
        return -1;

    //将i-1位置及之后的元素后移
    for (int j = length; j >= i; j--)
        data[j] = data[j - 1];
    //插入新元素
    data[i - 1].no = no;
    data[i - 1].name = name;
    length++;
    return 0;
}
```

remove 函数负责删除指定位置的学生记录，同样先进行位置合法性检查，确保要删除的位置在有效范围内（1 ≤ i ≤ length）。删除操作通过将待删除位置之后的所有元素向前移动一个位置来实现，这样就覆盖了要删除的元素，然后通过减少长度值来逻辑上删除最后一个元素。

```
//删除第i个位置的学生
int remove(int i)
{
    if (i<1 || i>length)
        return -1;

    //将i位置之后的学生前移
    for (int j = i - 1; j < length - 1; j++)
        data[j] = data[j + 1];

    length--;
    return 0;
}
```

查找功能通过 findByName 和 findByNo 两个函数实现，分别支持按姓名和按学号两种查找方式。这两个函数都采用线性查找算法，遍历整个顺序表，逐个比较目标值。当找到匹配项时，将完整的学生信息通过引用参数返回，并返回该学生在顺序表中的位置（从 1 开始

计数）。如果遍历完整个表都没有找到匹配项，则返回-1 表示查找失败。

```
//根据姓名查找
int findByName(const string& name, student& result)
{
    for (int i = 0; i < length; i++) {
        if (data[i].name == name) {
            result = data[i]; //学生信息赋给result
            return i + 1; //返回位置
        }
    }
    return -1; //没找到
}
```

```
//根据学号查找
int findByNo(const string& no, student& result)
{
    for (int i = 0; i < length; i++) {
        if (data[i].no == no) {
            result = data[i]; //学生信息赋给result
            return i + 1; //返回位置
        }
    }
    return -1;
}
```

getLength 函数提供了获取当前顺序表中学生数量的接口，直接返回 length 成员变量的值。

```
//获取学生总数
int getLength()
{
    return length;
}
```

2.2.5 调试分析

最初在实现 findByNo 函数时，错误地将 return -1 语句放在了 for 循环内部，导致该函数在检查第一个元素不匹配后就直接返回查找失败，无法继续检查后续元素。

最初的设计是无论查找成功与否，都输出位置信息和学生详细信息。但当查找失败返回-1 时，result 参数中的学生信息是未定义的，输出这些未定义值会导致不可预测的结果。通过添加条件判断，只在查找成功时输出完整信息，查找失败时仅输出-1，确保了输出格式的正确性。

2.2.6 总结和体会

顺序表的核心优势在于其连续的内存存储方式，这使得随机访问变得非常高效，可以通过下标直接访问任意位置的元素。在实现插入和删除操作时，深刻体会到顺序表在内存管理方面的特点——需要移动大量元素来维持存储的连续性，这在一定程度上影响了这些操作的效率。

在程序设计过程中，体会到了良好的封装性和模块化设计的重要性。将顺序表的实现细节封装在 SeqList 类中，对外提供清晰的操作接口，不仅提高了代码的可读性和可维护性，也使得主函数的逻辑更加清晰。这种设计模式符合面向对象编程的原则，有利于代码的重用和扩展。

2.3 题目三 一元多项式的相加和相乘

2.3.1 题目描述

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

输入要求：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，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 个整数，p1 e1 p2 e2 ...，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。

输出要求：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

本程序采用链表结构来存储一元多项式，每个节点包含系数和指数两个数据域以及一个指向下一个节点的指针域。这种设计能够灵活地处理动态长度的多项式，并且便于进行插入、删除等操作。

```
//多项式节点结构体
struct PolyNode {
    int coef;
    int exp;
    PolyNode* next;
    PolyNode(int c = 0, int e = 0) :coef(c), exp(e), next(NULL) {};
};
```

在链表结构设计中，采用了带头节点的单链表形式，头节点不存储实际数据，仅作为链表的起始标志。这种设计简化了边界条件的处理，使得在链表为空或进行头插操作时代码更加统一。每个 PolyNode 节点包含三个成员：coef 表示系数，exp 表示指数，next 指向下一个节点。构造函数为节点提供了默认参数，方便创建节点时的初始化操作。

```
//多项式类
class Polynomial {
private:
    PolyNode* head; //头指针

public:
    //构造函数
    Polynomial()
    {
        head = new PolyNode(); //头节点
    }

    //析构函数
    ~Polynomial()
    {
        clear();
        delete head;
    }
```

2.3.4 功能实现

清空多项式函数 clear()用于释放链表中的所有节点内存:

```
//清空多项式
void clear()
{
    PolyNode* curr = head->next;
    while (curr) {
        PolyNode* temp = curr;
        curr = curr->next;
        delete temp;
    }
    head->next = NULL;
}
```

构建多项式函数 build()从输入数组创建有序多项式链表:

```
//构建多项式
void build(int* arr, int size)
{
    //重置多项式
    clear();
    PolyNode* tail = head;

    for (int i = 0; i < size; i += 2) {
        int coef = arr[i];
        int exp = arr[i + 1];

        //创建新节点
        PolyNode* newNode = new PolyNode(coef, exp);
        tail->next = newNode; //带表头节点，head指针不动，尾插法
        tail = newNode;
    }
}
```

添加项函数 add()实现了在有序链表中插入新项或合并同类项的功能：

```
//添加项
void add(int coef, int exp)
{
    if (coef == 0)
        return;  //系数为0不添加

    PolyNode* prev = head;
    PolyNode* curr = head->next;

    //找到插入位置
    while (curr && curr->exp < exp) {
        prev = curr;
        curr = curr->next;
    }

    if (curr && curr->exp == exp) {  //指数相同，合并系数
        curr->coef += coef;
        if(curr->coef==0){   //系数为0，删除节点
            prev->next = curr->next;
            delete curr;
        }
    }
    else {   //插入新节点
        PolyNode* newNode = new PolyNode(coef, exp);
        newNode->next = curr;
        prev->next = newNode;
    }
```

多项式加法函数 add()采用双指针法合并两个有序多项式链表

```
Polynomial add(const Polynomial& other) const
{
    Polynomial result;
    PolyNode* p1 = head->next;
    PolyNode* p2 = other.head->next;
    PolyNode* tail = result.head;

    while (p1 && p2) {
        if (p1->exp < p2->exp) {
            tail->next = new PolyNode(p1->coef, p1->exp);
            tail = tail->next;
            p1 = p1->next;
        }
        else if (p1->exp > p2->exp) {
            tail->next = new PolyNode(p2->coef, p2->exp);
            tail = tail->next;
            p2 = p2->next;
        }
        else {  //p1->exp = p2->exp
            int sumcoef = p1->coef + p2->coef;
            if (sumcoef != 0) {
                tail->next = new PolyNode(sumcoef, p2->exp);
                tail = tail->next;
            }
            p1 = p1->next;
            p2 = p2->next;
        }
```

```
    //处理剩余部分
    while (p1) {
        tail->next = new PolyNode(p1->coef, p1->exp);
        tail = tail->next;
        p1 = p1->next;
    }
    while (p2) {
        tail->next = new PolyNode(p2->coef, p2->exp);
        tail = tail->next;
        p2 = p2->next;
    }

    return result;
}
```

多项式乘法函数 multiply()通过双重循环实现多项式相乘：

```
//多项式乘法
Polynomial multiply(const Polynomial& other)const
{
    Polynomial result;
    PolyNode* p1 = head->next;
    while (p1) {
        PolyNode* p2 = other.head->next;
        while (p2) {
            int coef = p1->coef * p2->coef;
            int exp = p1->exp + p2->exp;
            result.add(coef, exp);
            p2 = p2->next;
        }
        p1 = p1->next;
    }
    return result;
}
```

输出函数 print()格式化输出多项式内容:

```
//输出多项式
void print() const
{
    PolyNode* curr = head->next;
    bool first = true;
    while (curr) {
        if (!first)
            cout << " ";
        cout << curr->coef << " " << curr->exp;
        first = false;
        curr = curr->next;
    }
    if (first == false)  //非空多项式
        cout << endl;
}
```

2.3.5 调试分析

首先是 build 函数中参数含义的理解问题，最初在 main 函数中调用 build 时传入的是多项式的项数 m 和 n，但在 build 函数内部却将其作为数组长度来处理，这导致了数据读取错误。通过仔细分析发现，build 函数中的 size 参数应该表示数组的实际长度，即 2*m* 和 2n，而不是简单的项数 m 和 n。

另一个重要问题是内存管理，在多项式相加和相乘操作中需要创建新的节点，必须确保在适当的时候释放内存以避免内存泄漏。程序通过在 Polynomial 类中实现析构函数和 clear 函数来解决这个问题，确保在对象销毁时能够正确释放所有动态分配的内存。

2.3.6 总结与体会

链表结构能够灵活地表示稀疏多项式，避免了使用数组时可能出现的空间浪费问题。带头节点的链表设计虽然在空间上稍有开销，但大大简化了插入、删除等操作的代码复杂度，提高了代码的可读性和可维护性。

在算法设计方面，多项式相加采用的双指针合并算法体现了分治思想，而多项式相乘则展示了如何通过组合基本操作来实现复杂功能。

2.4 题目四 求级数

2.4.1 题目描述

求级数 A + $2A^2$ + $3A^3$ + … + NA^N 的值。输入包含若干行，在每一行中给出整数 N 和 A 的值，其中 1<=N<=150，0<=A<=15。对于不同的数据范围有不同的测试点：对于20%的数据，有 1<=N<=12，0<=A<=5；对于 40%的数据，有 1<=N<=18，0<=A<=9；对于 100%的数据，有 1<=N<=150，0<=A<=15。需要输出对于每一行输入的 N 和 A，计算级数的整数值。

2.4.2 基本要求

程序的输入要求是处理若干行数据，每行包含两个整数N和A，取值范围为1<=N<=150，0<=A<=15。输出要求是对每一行输入，在一行中输出级数 A+$2A^2$+$3A^3$+…+NA^N 的整数值。由于 N 和 A 的最大值会导致计算结果非常大，远远超过普通整数类型的表示范围，因此必须使用高精度计算来处理大整数运算。

2.4.3 数据结构设计

本程序的核心数据结构是自定义的 BigInt 类，用于实现高精度整数运算。BigInt 类使用 vector<int> digits 来存储大整数的各位数字，采用倒序存储方式，即 digits[0]存储个位，digits[1]存储十位，以此类推，这种存储方式便于进行算术运算。

BigInt 类提供了多种构造函数：默认构造函数创建一个空的大整数；基于 long long 的构造函数将普通整数转换为大整数；基于 string 的构造函数将字符串表示的大数转换为内部存储格式。类中还包含 removeLeadingZeros 方法用于去除数字前面多余的零，确保数字表示的规范性。

```
class BigInt {
private:
    vector<int> digits;

public:
    BigInt() {}

    BigInt(long long n) {
        if (n == 0) {
            digits.push_back(0);
        }
        else {
            while (n > 0) {
                digits.push_back(n % 10);
                n /= 10;
            }
        }
    }

    BigInt(const string& s) {
        for (int i = s.length() - 1; i >= 0; i--) {
            digits.push_back(s[i] - '0');
        }
        removeLeadingZeros();
    }
```

2.4.4 功能实现

在运算功能方面，BigInt 类重载了加法运算符和乘法运算符，实现了高精度的加法和乘法运算。加法运算采用逐位相加并处理进位的方式，乘法运算则使用传统的竖式乘法算法，通过两层循环实现各位数字的相乘和进位处理。此外，还实现了 power 方法用于计算大整数的幂运算，采用快速幂算法来提高计算效率。最后，toString 方法将内部存储的大整数转换为字符串形式输出。

```
BigInt operator+(const BigInt& other) const {
    BigInt result;
    int carry = 0;
    int maxSize = max(digits.size(), other.digits.size());

    for (int i = 0; i < maxSize || carry; i++) {
        int sum = carry;
        if (i < digits.size()) sum += digits[i];
        if (i < other.digits.size()) sum += other.digits[i]

        result.digits.push_back(sum % 10);
        carry = sum / 10;
    }

    return result;
}
```

```
BigInt operator*(const BigInt& other) const {
    BigInt result;
    result.digits.resize(digits.size() + other.digits.size(), 0);

    for (int i = 0; i < digits.size(); i++) {
        int carry = 0;
        for (int j = 0; j < other.digits.size() || carry; j++) {
            long long product = result.digits[i + j] +
                (long long)digits[i] * (j < other.digits.size() ? other.digits[j] : 0) +
                carry;
            result.digits[i + j] = product % 10;
            carry = product / 10;
        }
    }

    result.removeLeadingZeros();
    return result;
}
```

```
BigInt power(int exponent) const {
    if (exponent == 0) return BigInt(1);

    BigInt result(1);
    BigInt base = *this;

    while (exponent > 0) {
        if (exponent & 1) {
            result = result * base;
        }
        base = base * base;
        exponent >>= 1;
    }

    return result;
}
```

calculateSeries 函数负责计算级数的和，其实现代码如下：

```
BigInt calculateSeries(int N, int A) {
    if (A == 0) return BigInt(0);  // 所有项都是0

    BigInt sum(0);
    BigInt base(A);

    for (int k = 1; k <= N; k++) {
        BigInt term = BigInt(k) * base.power(k);
        sum = sum + term;
    }

    return sum;
}
```

2.4.5 调试分析

最初尝试使用数学公式直接计算级数和，但在实现过程中发现公式推导和除法运算较为复杂，容易出错。最终选择采用直观的循环累加方法，虽然时间复杂度稍高，但正确性更容易保证。

在 BigInt 类的实现中，乘法运算是一个关键难点。需要正确处理各位数字的相乘和进位，特别是当数字很大时，中间结果可能超出 int 的范围。解决方案是使用 long long 类型存储中间乘积，确保不会溢出。另外，快速幂算法的实现也需要注意边界条件，特别是指数为 0 时的特殊情况。

2.5 题目五 扑克牌游戏

2.5.1 题目描述

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，则输出 NULL。

2.5.2 基本要求

输入描述：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。

输出描述：输出若干行，每次收到 Pop 指令后输出一行（花色和数字或 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

在本程序的设计中，主要采用了两种核心数据结构：Card 结构体和 Node 结构体，以及一个管理这些结构的 PokerDeck 类。

Card 结构体用于表示单张扑克牌，包含两个字符串成员：

```
// 牌的结构
struct Card {
    string suit;
    string rank;
};
```

其中 suit 存储花色（如"Heart"、"Club"等），rank 存储牌面值（如"A"、"2"、"J"、"Q"等）。

Node 结构体构成双向链表的节点，除了包含 Card 数据外，还有前后指针：

```
struct Node {
    Card card;
    Node* prev;
    Node* next;

    Node(string s, string r) {
        card.suit = s;
        card.rank = r;
        prev = nullptr;
        next = nullptr;
    }
};
```

PokerDeck 类作为主要的牌堆管理器，维护着链表的头尾指针和节点计数：

```
class PokerDeck {
private:
    Node* head;
    Node* tail;
    int count;

    // 将牌面转换为数字用于排序
    int rankToValue(const string& rank) {
        if (rank == "A") return 1;
        if (rank == "J") return 11;
        if (rank == "Q") return 12;
        if (rank == "K") return 13;
        return stoi(rank);
    }
```

```
// 从链表中移除节点（不删除内存）
void removeNode(Node* node) {
    if (node->prev) node->prev->next = node->next;
    if (node->next) node->next->prev = node->prev;
    if (node == head) head = node->next;
    if (node == tail) tail = node->prev;
    count--;
}

// 在头部插入节点
void insertAtHead(Node* node) {
    node->prev = nullptr;
    node->next = head;
    if (head) head->prev = node;
    head = node;
    if (!tail) tail = node;
    count++;
}
```

2.5.4 功能实现

构造函数初始化空牌堆，析构函数负责释放所有节点内存，避免内存泄漏。

```
PokerDeck() : head(nullptr), tail(nullptr), count(0) {}
```

```
~PokerDeck() {
    Node* curr = head;
    while (curr) {
        Node* next = curr->next;
        delete curr;
        curr = next;
    }
}
```

append 函数在牌堆底部添加新牌，时间复杂度为 O(1)。如果是第一张牌，同时设置 head 和 tail；否则在尾部插入并更新 tail 指针。

```
// Append操作：在尾部添加牌
void append(string suit, string rank) {
    Node* newNode = new Node(suit, rank);
    if (tail) {
        tail->next = newNode;
        newNode->prev = tail;
        tail = newNode;
    }
    else {
        head = tail = newNode;
    }
    count++;
}
```

extract 函数分为三个步骤：首先遍历链表提取指定花色的所有牌，使用冒泡排序对提取的牌进行排序，最后将排序后的牌插入到原链表头部。

revert 函数通过交换每个节点的前后指针来实现链表反转，最后交换 head 和 tail 指针

```
// Revert操作：反转整个牌堆
void revert() {
    if (count <= 1) return;

    Node* curr = head;
    Node* temp = nullptr;

    while (curr) {
        // 交换prev和next指针
        temp = curr->prev;
        curr->prev = curr->next;
        curr->next = temp;
        // 移动到下一个节点（原来的next，现在的prev）
        curr = curr->prev;
    }

    // 交换head和tail
    temp = head;
    head = tail;
    tail = temp;
}
```

Pop 函数移除并输出顶部牌，处理空牌堆情况，更新 head 指针并释放内存。

```
void pop() {
    if (count == 0) {
        cout << "NULL" << endl;
        return;
    }

    Node* topNode = head;
    cout << topNode->card.suit << " " << topNode->card.rank << endl;

    head = head->next;
    if (head) {
        head->prev = nullptr;
    }
    else {
        tail = nullptr;
    }

    delete topNode;
    count--;
}
```

2.5.5 调试分析

首先是 Extract 操作的实现复杂度较高，需要同时处理节点的移除、排序和重新插入。最初尝试在原始链表中直接排序，但发现指针操作过于复杂，最终采用了先提取到临时链表、排序后再合并的策略。

其次是内存管理问题，在节点移除和插入过程中容易出现内存泄漏或野指针。通过仔细设计 removeNode 函数，确保在移除节点时正确更新前后节点的指针，同时在析构函数中彻底释放所有内存，解决了这些问题。

3.实验总结

线性表从实现方式上主要分为顺序存储结构和链式存储结构两大类。顺序存储结构以数组为代表，其特点是在内存中采用连续的存储空间，通过下标直接访问元素，具有随机存取的高效性，访问时间复杂度为 O(1)。然而这种结构的缺点也十分明显，插入和删除操作需要移动大量元素，时间复杂度为 O(n)，而且在创建时需要预先分配固定大小的空间，容易造成空间浪费或溢出。相比之下，链式存储结构采用非连续的存储方式，通过指针将各个节点连接起来，如我在扑克牌游戏中实现的双向链表。这种结构的最大优势在于插入和删除操作的高效性，只需要修改相关节点的指针即可完成，时间复杂度为 O(1)，同时支持动态内存分配，无需预先确定存储规模。但链表的缺点在于无法随机访问，查找操作需要从头遍历，时间复杂度为 O(n)，而且每个节点都需要额外的指针空间，存储密度相对较低。