**作业 HW1\* 实验报告**

姓名：张嘉麟 学号：2352595 日期：2024年10月03日

# 实验报告格式要求按照模板（使用Markdown等也请保证报告内包含模板中的要素）

# 对字体大小、缩进、颜色等不做强制要求（但尽量代码部分和文字内容有一定区分，可参考vscode配色）

# 实验报告要求在文字简洁的同时将内容表示清楚

# 报告内不要大段贴代码，尽量控制在20页以内

1. **涉及数据结构和相关背景**

# 题目或实验涉及数据结构的相关背景

线性表是一种基础的数据结构，表示一组有序的数据元素。线性表中的元素具有相同的类型，并且可以通过索引进行访问。常见的线性表包括数组和链表，前者是静态结构，后者则是动态结构。

**静态链表**

静态链表使用一组固定大小的数组来实现线性表。每个元素不仅存储数据，还包含指向下一个元素的索引。静态链表的优点是可以通过数组下标快速访问元素，但缺点是无法动态扩展大小，插入和删除操作可能会导致大量的元素移动。

// 静态链表节点定义

struct Node {

int data; // 存储数据

int next; // 下一个节点的索引

};

**动态链表**

动态链表是一种通过动态内存分配实现的线性表。每个节点包含数据和一个指向下一个节点的指针。由于使用动态内存，动态链表可以根据需要动态地增加或减少节点数量，避免了静态链表的大小限制和内存浪费问题。

// 动态链表节点定义

struct Node {

int data; // 存储数据

Node\* next; // 指向下一个节点的指针

};

**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**

实现一个程序，将输入的数组元素向右轮转 k 个位置并输出结果。

**2.1.3 数据结构设计**

// 数组结构体

struct Array {

int \*nums; // 数组元素指针

int size; // 数组大小

};

// 初始化数组结构

Array createArray(int n) {

Array arr;

arr.nums = new int[n]; // 动态分配内存

arr.size = n; // 设置大小

return arr;

}

// 释放数组内存

void destroyArray(Array &arr) {

delete[] arr.nums; // 释放动态数组

}

**2.1.4 功能说明（函数、类）**

**方法一：原地旋转**

void reverse(int arr[], int start, int end) {

// 反转数组中的元素

// 输入：arr - 原数组, start - 起始索引, end - 结束索引

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

int main() {

// 读取输入

// 防止 k 超出 n

// 三步反转法

}

**时间复杂度**：O(n)

**空间复杂度**：O(1)

**方法二：辅助数组**

int main() {

// 读取输入

// 防止 k 超出 n

int rotated[n]; // 辅助数组用于存储旋转后的结果

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

rotated[(i + k) % n] = nums[i]; // 计算旋转后的索引并赋值

}

// 输出结果

}

**时间复杂度**：O(n)

**空间复杂度**：O(n)

**方法三：循环替代**

int main() {

// 读取输入

// 防止 k 超出 n

int count = 0; // 记录已经替换的元素数量

for (int start = 0; count < n; start++) {

int current = start;

int prevValue = nums[start];

do {

// 计算下一个位置并更新

} while (start != current);

}

// 输出结果

}

**时间复杂度**：O(n)

**空间复杂度**：O(1)

**2.1.5 调试分析（遇到的问题和解决方法）**

在调试过程中，遇到的问题主要是处理 k 大于 n 的情况，导致数组访问越界。通过对 k 取模处理，避免了越界的问题。同时，在使用原地旋转法时，需要确保对数组的所有部分都进行了反转，确保正确性。

**2.1.6 总结和体会**

通过这道题目，体会到不同算法的实现方式及其时间复杂度和空间复杂度的影响。原地旋转法虽然实现复杂，但在空间使用上更为高效。处理数据边界情况（如 k 超过 n）是解决问题时的一个关键点。

**2.2 顺序表的基本操作**

**2.2.1 问题描述**

本题要求定义一个顺序表，包含学生信息（学号、姓名），实现以下功能：

1. 输入指定个数的学生信息。
2. 在指定位置插入学生信息。
3. 删除指定位置的学生记录。
4. 根据姓名和学号查找学生。
5. 统计学生个数。

**2.2.2 基本要求**

实现顺序表的基本操作，包括创建、插入、删除、查找和销毁。

**2.2.3 数据结构设计**

// 学生信息结构体

struct Student {

string id; // 学号

string name; // 姓名

};

// 顺序表类

class SequentialList {

private:

Student\* students; // 动态数组

int capacity; // 当前容量

int size; // 当前学生数量

void resize(); // 扩展容量

public:

SequentialList(); // 构造函数

~SequentialList(); // 析构函数

void insert(int i, const Student& s); // 插入学生信息

void remove(int j); // 删除指定位置学生

void checkName(const string& name); // 根据姓名查找学生

void checkNo(const string& id); // 根据学号查找学生

int getSize() const; // 获取学生总数

void inputStudents(int n); // 输入学生信息

};

**2.2.4 功能说明（函数、类）**

void insert(int i, const Student& s) {

// 插入学生信息到第 i 个位置

// 处理不合法位置和扩展容量

}

void remove(int j) {

// 删除第 j 个学生记录

// 处理不合法位置

}

void checkName(const string& name) {

// 根据姓名查找学生并输出结果

}

void checkNo(const string& id) {

// 根据学号查找学生并输出结果

}

**2.2.5 调试分析**

这个作业我做了6h+，最后一直都是40分，怎么修改都是40分。前面我写的时候我想到了会不会是边界问题，我发现前面我的10000的maximum用的是“＜”，所以我改成了“≤”，还是只有40分。

后面我实在不知道改什么了，就试一试地把10000改成了10010就对了。

现在来看就是存贮第一万个学生的时候的内存问题。  


因为元素从1开始，所以0没有算在内。

**2.2.6 总结和体会**

通过此题，增强了对顺序表及内存管理的理解。难点在于高效处理插入和删除操作，以及保证程序的鲁棒性。

**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，输入保证是按照指数项递增有序的。

**2.3.2 基本要求**

实现一元多项式的加法和乘法运算。

输入格式：

* 1. 第一个多项式的长度m。
  2. 第一个多项式的系数和指数。
  3. 第二个多项式的长度n。
  4. 第二个多项式的系数和指数。
  5. 操作类型：0（加法）、1（乘法）、2（同时输出加法和乘法结果）。

输出按指数从小到大的运算结果，且当运算结果为0 0时，不输出。

**2.3.3 数据结构设计**

// 多项式项结构体

struct PolyItem {

int coef; // 系数

int exp; // 指数

};

// 链表节点结构体

struct Node {

PolyItem item; // 多项式项

Node\* next; // 下一个节点指针

};

// 链表类

class LinkedList {

private:

Node\* head; // 头指针

public:

LinkedList(); // 构造函数

~LinkedList(); // 析构函数

void insert(const PolyItem& item); // 插入节点

void print() const; // 打印多项式

LinkedList addPolynomials(const LinkedList& p1, const LinkedList& p2); // 多项式相加

LinkedList multiplyPolynomials(const LinkedList& p1, const LinkedList& p2); // 多项式相乘

};

**2.3.4 功能说明（函数、类）**

**1. 插入节点**

void insert(const PolyItem& item) {

// 在链表末尾插入新节点

// 输入：item - 要插入的多项式项（系数和指数）

}

**2. 打印多项式**

void print() const {

// 打印多项式，格式化输出系数和指数

// 若所有项系数为0，则输出 0 0

}

**3. 多项式相加**

LinkedList addPolynomials(const LinkedList& p1, const LinkedList& p2) {

// 计算两个多项式的和

// 输入：p1 - 第一个多项式，p2 - 第二个多项式

// 输出：返回相加后的多项式链表

}

**4. 多项式相乘**

LinkedList multiplyPolynomials(const LinkedList& p1, const LinkedList& p2) {

// 计算两个多项式的积

// 输入：p1 - 第一个多项式，p2 - 第二个多项式

// 输出：返回相乘后的多项式链表

}

**2.3.5 调试分析（遇到的问题和解决方法）**

在调试过程中，主要遇到以下问题：

**问题一**：链表插入节点时，未正确更新指针，导致链表断裂。

仔细检查指针的更新逻辑，确保每次插入时都能正确链接到新的节点。

**问题二**：多项式相加时，未处理重复指数的情况，导致输出结果不准确。

在相加过程中，检查当前节点的指数是否与其他节点相同，若相同，则合并系数。

**问题三**：多项式相乘时，结果未按指数排序。

乘法结果处理后，按照指数进行排序，并合并同类项。

**2.4 求级数**

**2.4.1 问题描述**

求级数 A + 2A² + 3A³ + ... + N × Aⁿ。

**2.4.2 基本要求**

实现高精度加法和乘法，以支持在给定范围内的整数 N 和 A 进行级数求解。

**2.4.3 数据结构设计**

// 高精度加法函数

string addBigNumbers(string num1, string num2); // 返回 num1 和 num2 的和

// 高精度乘法函数

string multiplyBigNumbers(string num1, int num2); // 返回 num1 乘以 num2 的结果

**2.4.4 功能说明（函数、类）**

string addBigNumbers(string num1, string num2) {

// 返回 num1 和 num2 的高精度和

// 输入：num1 - 第一个大数，num2 - 第二个大数

// 输出：高精度和的字符串表示

}

string multiplyBigNumbers(string num1, int num2) {

// 返回 num1 乘以 num2 的高精度结果

// 输入：num1 - 大数，num2 - 整数

// 输出：高精度乘积的字符串表示

}

**2.4.5 调试分析（遇到的问题和解决方法）**

在实现过程中，遇到的主要问题是如何处理大数的溢出。通过使用字符串来表示大数，并逐位进行加法和乘法，成功解决了溢出的问题。

**时间复杂度和空间复杂度分析**

**时间复杂度**：

高精度加法 addBigNumbers：O(max(len1, len2))，len1 和 len2 是大数的位数。

高精度乘法 multiplyBigNumbers：O(len1)，len1 是大数的位数。

主函数计算级数的时间复杂度为 O(N \* len1)，N 是输入的最大值。

**空间复杂度**：

高精度加法和乘法的空间复杂度分别为 O(len1 + len2) 和 O(len1)，用于存储结果和中间变量。

主函数的空间复杂度为 O(len1)，主要用于存储结果和幂的字符串表示。

**2.5 扑克牌游戏**

**2.5.1 问题描述**

设计一个扑克牌游戏，玩家可以通过一系列命令来操作一个初始为空的牌堆。命令包括添加牌、抽取特定花色的牌并排序、反转牌堆以及弹出牌堆顶部的牌。最终输出牌堆中的所有牌。

**2.5.2 基本要求**

1. **添加（Append）**：将一张牌添加到牌堆的底部。
2. **抽取（Extract）**：抽取某花色的所有牌，按编号排序后放至牌堆顶部。
3. **反转（Revert）**：将牌堆逆序。
4. **弹出（Pop）**：移除并显示牌堆顶部的牌；若牌堆为空，则输出NULL。

**2.5.3 数据结构设计**

// 定义扑克牌结构体

struct Card {

string suit; // 花色

string value; // 面值

int number; // 用于比较的数值

Card\* next; // 指向下一个节点的指针

}

**2.5.4 功能说明（函数、类）**

class CardPile {

// 牌堆类

Card\* head; // 牌堆顶部指针

Card\* tail; // 牌堆底部指针

// 构造函数

CardPile() {

// 初始化牌堆

head = null

tail = null

}

// 添加牌到牌堆底部

void append(suit: string, value: string) {

// 创建新卡牌并添加到牌堆底部

newCard = create new Card with suit and value

if tail is null {

head = tail = newCard

} else {

tail->next = newCard

tail = newCard

}

}

// 抽取某个花色的所有牌并排序后放在顶部

void extract(suit: string) {

// 分离出指定花色的牌，并排序后放置在牌堆的顶部

// 核心逻辑：分离 -> 排序 -> 合并

}

// 反转整个牌堆

void revert() {

// 反转整个牌堆链表

// 核心逻辑：使用三个指针prev、current、nextCard

}

// 弹出牌堆顶部的牌

void pop() {

// 移除并输出牌堆顶部的牌

if head is null {

output NULL

} else {

output head's suit and value

remove head node

}

}

// 打印牌堆中的所有牌

void printPile() {

// 打印牌堆中的所有牌

if head is null {

output NULL

} else {

current = head

while current is not null {

output current's suit and value

current = current->next

}

}

}

// 释放链表内存

void freeMemory() {

// 释放所有节点的内存

while head is not null {

temp = head

head = head->next

delete temp

}

tail = null

}

}

**2.5.5 调试分析（遇到的问题和解决方法）**

在开发过程中遇到的主要问题及其解决方法：

1. 数据存储和判断问题：原先采用char类型的判断，效率较低，将AJQK转换为“1”“11”“12”“13”数字判断效率大大增加。
2. 内存管理问题：确保在append函数中正确处理内存分配失败的情况。
3. 排序问题：在extract函数中，通过正确排序逻辑确保插入时按照正确的顺序进行。
4. 错误处理：在处理pop操作时，增加了对空牌堆的检查，防止程序异常。

**3. 实验总结**

**2.1 轮转数组**

在这个任务中，我们学习了如何有效地旋转数组。通过三种不同的方法——原地旋转、辅助数组以及循环替代，我们探讨了不同的时间和空间复杂度对算法性能的影响。特别是原地旋转法，虽然其实现较为复杂，但由于其空间复杂度为O(1)，在实际应用中具有很高的实用价值。

**2.2 顺序表的基本操作**

顺序表的操作涉及到动态数组的管理和维护。在这个过程中，我们实现了学生的插入、删除、查找等功能。通过手动管理数组的扩展与收缩，我们加深了对内存管理和动态数组操作机制的理解。此外，处理边界条件（如插入位置非法等）也是该任务的重点之一。

**2.3 一元多项式的相加和相乘**

多项式的相加与相乘涉及到链表的应用。通过构建链表来存储多项式的每一项，并实现相应的加法和乘法运算，我们掌握了链表作为一种动态数据结构的使用方法。在实现过程中，确保链表的完整性和正确性是非常重要的，特别是在合并同类项和保持链表有序方面。

**2.4 求级数**

求解高精度数的级数需要克服传统整型数据类型的限制。通过使用字符串来表示大数，并实现高精度的加法和乘法，我们能够有效地解决大数运算的问题。

**2.5 扑克牌游戏**

最后一个任务是一个扑克牌游戏的设计。通过使用链表来模拟牌堆的操作，我们实现了添加、抽取、反转和弹出等基本功能。此任务强化了我们对于链表操作的理解，尤其是在内存管理和链表节点排序等方面。

**总结与体会**

通过此次实验，我不仅巩固了对线性表、链表等数据结构的认识，还学习了如何在实际问题中应用这些知识。每一道题目都让我们有机会深入思考算法设计的原则、数据结构的选择以及编程实践中的常见挑战。通过和本次五道作业的鏖战，我对于边界条件、运行效率、链表的删减等操作有了更深刻的认识，也有了很多教训！