**作业 HW1\* 实验报告**

日期：2024年10月4日

1. **涉及数据结构和相关背景**

**1.1线性表**

* 理解线性表的逻辑结构、抽象数据类型的定义，以及不同存储结构的描述方式
* 在顺序存储和链式存储这两种线性表存储结构的基础上，完成基本的增加、删除、修改、查找等操作

**1.2 线性表的分析**

* 顺序表
* 优点 ：
* 结构简单
* 高
* 在尾部存储简便（仅尾部）
* 缺点 ：
* 插入删除需移动大量元素，效率低
* 所需存储空间不确定，可能溢出或者浪费大量空间
* 存储必需要大量连续空间 ，空间利用率低
* 链表
* 类型：单链表、双链表、循环链表
* 优点：
* 存储空间动态分配（按需分配），一般不会溢出或浪费空间
* 插入和删除不需要去运动大量元素
* 存储不要求连续空间，空间利用率高
* 缺点：
* 读取、修改的效率低，耗时相对较长，劣于线性表
* 结构较为复杂
* 结论
* 如查找较多：线性表
* 如插入删除多：链表
* 综合运用两种数据结构能获得更好的时间空间性能

**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**

输入要求：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

输出要求：输出轮转后的顺序表中的元素

**2.1.3 数据结构设计**

定义一个结果数组（顺序表），和一个表示当前所在位置的int型变量，首先读入结果数组长度和旋转位置数，然后再读入输入的元素，通过计算得到输出的起始位置，最后进行输出。

**2.1.4功能说明（函数、类）**

int input[100001] = { 0 }, len, k;

cin >> len >> k; //读入结果数组长度和旋转位置数

for (int i = 1; i <= len; i++)

cin >> input[i]; //读入输入元素

int start = len-k%len; //通过旋转位置数计算输出的起始位置

for (int i = 1; i <= len; i++)

{ //从计算得到的起始位置依次输出

start = (start == len) ? 1 : start +1; //如果输出到了数组尾部则返回头部

cout << input[start] << " ";

}

**2.1.5 调试分析（遇到的问题和解决方法）**

调试过程中，输入都没有问题，但是在刚开始，由于没有考虑k（旋转量）可能比数组大，导致输出起始位置不正确，导致了数组的越界

**2.1.6 总结和体会**

本题n与k均为10^5级别，所以把n个元素全部向右k次（o(nk)级别的时间复杂）度是不可行的，而直接把向右旋转k次得到的结果存储在一个结果数组中，时间复杂度是o(n)，可行（未给出对应源程序）

而在上述方法中，仅改变标志起始位置的int型元素，复杂度仅为o(1),是理论上最优解，效率高、数据边界处理也方便

**2.2 学生信息管理**

**2.2.1 问题描述**

本题要求定义一个顺序表来管理学生信息（学号、姓名），并实现以下功能：

1. 根据指定学生个数，逐个输入学生信息；

2. 插入学生信息到指定位置；

3. 删除指定位置的学生记录；

4. 根据姓名或学号查找学生信息；

5. 统计并输出表中学生个数。

**2.2.2 基本要求**

输入输出要求：

1. 初始化时输入`n`名学生的学号和姓名。

2. 支持以下操作：

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

remove j: 删除第j个位置的学生。若j位置不合法，输出-1，否则输出0。

check name 姓名y: 查找姓名为y的学生，输出其位置、学号和姓名。若找不到，输出-1。

check no 学号x: 查找学号为x的学生，输出其位置、学号和姓名。若找不到，输出-1。

end: 输出表中学生个数，结束操作。

**2.2.3 数据结构设计**

使用链表存储学生信息。每个链表节点包含学生的姓名、学号，以及指向下一个节点的指针。

struct Student\_no {

string name; // 姓名

string id; // 学号

Student\_no\* next; // 指向下一个节点的指针

};

**2.2.4 功能说明**

**插入函数**

功能：在指定位置插入学生信息。

/\*\*

\* @brief 插入学生信息

\* @param head 链表头指针

\* @param len 当前链表长度

\*/

void insert(Student\_no\*& head, int& len) {

int place;

string no, name;

// 输入：读入需要插入的位置、学号、姓名，并判断位置是否合法

if (place < 1 || place > len + 1) {

cout << -1 << endl; // 插入位置不合法

return;

}

// 创建新的学生节点

Student\_no\* new\_stu = new Student\_no(no, name, nullptr);

if (place == 1) { // 插入到链表头部

new\_stu->next = head;

head = new\_stu;

}

else { // 插入到链表其他位置

Student\_no\* prev = head;

for (int i = 1; i < place - 1; ++i) {

prev = prev->next;

}

new\_stu->next = prev->next;

prev->next = new\_stu;

}

len++;

cout << 0 << endl; // 插入成功

}

**删除函数**

功能：删除指定位置的学生信息。

/\*\*

\* @brief 删除指定位置的学生信息

\* @param head 链表头指针

\* @param len 当前链表长度

\*/

void remove(Student\_no\*& head, int& len) {

int place;

// 输入：读入需要删除的位置，并判断是否合法

if (place < 1 || place > len) {

cout << -1 << endl; // 删除位置不合法

return;

}

Student\_no\* toDelete;

if (place == 1) { // 删除头节点

toDelete = head;

head = head->next;

}

else { // 删除其他位置

Student\_no\* prev = head;

for (int i = 1; i < place - 1; ++i) {

prev = prev->next;

}

toDelete = prev->next;

prev->next = toDelete->next;

}

delete toDelete; // 释放内存

len--;

cout << 0 << endl; // 删除成功

}

**按姓名查找函数**

功能：根据学生姓名查找学生信息。

/\*\*

\* @brief 按姓名查找学生信息

\* @param head 链表头指针

\* @param name 要查找的学生姓名

\*/

void checkName(Student\_no\* head, string name) {

Student\_no\* current = head;

int place = 1;

while (current) {

if (current->name == name) {

cout << place << " " << current->id << " " << current->name << endl;

return;

}

current = current->next;

place++;

}

cout << -1 << endl; // 未找到

}

**按学号查找函数**

功能：根据学生学号查找学生信息。

本程序与上一个程序几乎完全相同，不重复了

**2.2.5 调试分析**

问题：插入位置不合法

解决方法：在插入时应考虑位置是否为1到`len+1`，并将`place`进行边界检查。

问题：删除时链表尾部指针指向非法地址

解决方法：删除节点后未正确更新指针，导致指针指向未定义位置。通过设置尾节点指针为`NULL`解决。

**2.2.6 总结和体会**

在进行大量插入和删除操作时。如果一开始使用的是静态数组或顺序表，在插入和删除操作时，需要移动大量元素，导致时间复杂度达到 O(n)，性能下降非常明显。而选择链表后，插入和删除操作的时间复杂度可以降低到 O(1)，从而使整个程序的执行效率大幅提升。

同时，边界检查和指针处理是此类问题的关键，细节上容易出错。

**2.3 一元多项式的相加和相乘**

**2.3.1 问题描述**

给定两个按指数递增排序的一元多项式，要求实现它们的相加和相乘操作。输入的多项式项数和指数保证有序，输出的结果也要按照指数递增排序，且系数为0的项不输出。

**2.3.2 基本要求**

 输入两个多项式的长度和各自的系数与指数；

 输出多项式的加法或乘法结果，且结果中系数为0的项不输出；

 按指数递增顺序输出结果；

**2.3.3 数据结构设计**

每个多项式项由coeff（系数）和exp（指数）组成的链表节点来表示，每个节点指向下一个节点。

struct Element {

int coeff; // 系数

int exp; // 指数

Element\* next; // 指向下一个元素的指针

};

**2.3.4功能说明（函数、类）**

**多项式加法函数**

函数功能：实现两个多项式的相加操作，输入为两个按指数递增排序的多项式链表，输出为合并后的多项式链表

/\*\*

\* @brief 多项式加法函数

\* @param poly1 第一个多项式链表

\* @param poly2 第二个多项式链表

\* @return 加法结果链表头指针

\*/

Element\* add (Element\* poly1, Element\* poly2) {

初始化一个空的结果链表;

使用两个指针分别遍历poly1和poly2;

while (poly1和poly2都没有遍历完) {

if (poly1的指数 < poly2的指数) {

将poly1的当前项加入结果链表;

移动poly1指针;

}

else if (poly1的指数 > poly2的指数) {

将poly2的当前项加入结果链表;

移动poly2指针;

}

else {

// 指数相同，系数相加

如果系数和不为0，加入结果链表;

同时移动两个指针;

}

}

// 处理剩余的多项式

将poly1或poly2中未遍历完的部分直接加入结果链表;

返回结果链表的头指针;

}

### 多项式乘法函数

函数功能：实现两个多项式的相乘操作。每一项分别相乘，结果按指数递增顺序返回。

/\*\*

\* @brief 多项式乘法函数

\* @param poly1 第一个多项式链表

\* @param poly2 第二个多项式链表

\* @return 乘法结果链表头指针

\*/

Element\* mul(Element\* poly1, Element\* poly2) {

初始化一个空的结果链表;

for (poly1的每一项) {

初始化一个临时链表;

for (poly2的每一项) {

计算poly1当前项与poly2当前项的乘积;

将结果项按指数顺序加入临时链表;

}

将临时链表与结果链表相加;

}

返回结果链表的头指针;

}

**2.3.5 调试分析（遇到的问题和解决方法）**

**问题：多项式加法结果未按顺序输出**

解决方法：在合并多项式时未处理好指数相等时的情况，导致重复输出。通过仔细检查双指针遍历过程，正确处理指数相等的情况。

**问题：系数为0的项输出**

解决方法：在输出结果时添加条件判断，系数为0的项不输出。

**问题：乘法运算耗时长**

解决方法：优化了乘法函数的遍历方式，避免了多余的链表遍历操作。同时，使用双指针的方式一次性遍历两个链表，减少了遍历次数，确保了算法能够处理极限数据。

**2.3.6 总结和体会**

在加法操作中，双指针的使用是最为关键的，尤其是在两个有序链表中进行加法运算时，使用双指针能够有效减少遍历次数，能够将时间复杂度从 O(n^2) 降低到 O(n)，特别适合处理多项式这种有序的数据结构，显著提升了程序的性能。

在删除、插入链表项时，指针的一定要正确移动，避免链表断裂或循环。

在加法、乘法的链表合并中，尤其要注意链表项的指针更新，确保指针正确指向下一项。

**2.4 求级数**

**2.4.1 问题描述**

给定一个整数 N 和 A，求形如 A+2A2+3A3+⋯+NAN的级数值。

**2.4.2 基本要求**

输入要求： 输入N，A，其中 1≤N≤150， 0≤A≤15。

输出要求：输出A+2A2+3A3+⋯+NAN 的级数值

**2.4.3 数据结构设计**

由于 N 的最大值为 150，且 A 的最大值为 15，数值可能超出 64 位整数范围。因此需要使用高精度运算来存储和计算这些大数。

这里用数组逆序存储每个大整数，模拟高精度数的加法和乘法运算，再进行输出。

const int MAX\_DIGITS = 1000; // 假设大整数最多有1000位

int result[MAX\_DIGITS]; // 存储大整数的每一位，逆序存储

**2.4.4功能说明（函数、类）**

### 高精度加法

/\*\*

\* @brief 高精度加法函数

\* @param a 第一个大整数数组

\* @param b 第二个大整数数组

\* @param sizeA 第一个数组的长度

\* @param sizeB 第二个数组的长度

\* @return 高精度加法结果数组

\*/

int\* add(int\* a, int\* b, int sizeA, int sizeB) {

初始化一个数组 result[] 用于存放加法结果;

carry = 0; // 初始化进位为0

// 遍历两个数组的每一位，进行逐位相加

for (i 从 0 到 max(sizeA, sizeB) - 1) {

sum = (a[i] 如果 i 小于 sizeA 否则 0) + (b[i] 如果 i 小于 sizeB 否则 0) + carry;

result[i] = sum 的个位数; // 只保留个位数

carry = sum 的十位数; // 更新进位

}

如果 carry 不为 0:

将 carry 加到结果数组的下一位;

返回结果数组;

}

### 高精度乘法

/\*\*

\* @brief 高精度乘法函数

\* @param a 高精度整数数组

\* @param x 整数乘数

\* @param sizeA 高精度整数数组的长度

\* @return 高精度乘法结果数组

\*/

int\* mul(int\* a, int x, int sizeA) {

初始化一个数组 result[] 用于存放乘法结果;

carry = 0; // 初始化进位为0

// 遍历 a[] 数组的每一位，逐位进行乘法

for (i 从 0 到 sizeA - 1) {

product = a[i] \* x + carry;

result[i] = product 的个位数; // 只保留个位数

carry = product 的十位数; // 更新进位

}

当 carry > 0 时:

将 carry 的每一位存入 result 数组的下一位;

返回结果数组;

}

### 高精度快速幂

/\*\*

\* @brief 高精度幂函数

\* @param A 底数

\* @param k 指数

\* @return 高精度幂的结果数组

\*/

int\* power\_(int A, int k) {

初始化一个数组 result[] 代表 1, 用于存放最终结果;

初始化一个数组 base[] 代表 A, 用于存放底数;

// 使用二进制快速幂计算 A^k

while (k > 0) {

如果 k 是奇数 :

result[] = mul (result, base, size of result); // result 乘以 base

base[] = mul(base, base, size of base); // base 自乘

k = k / 2;

}

返回 result;

}

### 主函数

/\*\*

\* @brief 计算级数和的函数

\* @param N 级数的项数

\* @param A 底数

\* @return 级数和的高精度结果

\*/

Int main() {

初始化一个数组 result[] 用于存放级数和;

输入N与A

A为0，直接输出0，结束函数

// 逐项计算级数和

for (i 从 1 到 N) {

term[] = powerBigInt(A, i); // 计算 A^i

term[] = multiplyBigInt(term, i, size of term); // 计算 i \* A^i

result[] = addBigInt(result, term, size of result, size of term); // 累加到结果数组中

}

输出结果（逆序输出）

}

**2.4.5 调试分析（遇到的问题和解决方法）**

**问题：高精度加法的进位处理**

解决方法：每次相加后正确进位进位，并处理最终的进位。

**问题：高精度乘法的进位处理**

解决方法：每次相乘后保存个位，进位放到下一位继续相加。

**问题：对于A=0没有非零位，无法正常输出**

解决方法：特殊判断，若A为0，直接输出0即可。

**2.4.6 总结和体会**

高精度运算是本题的核心，特别是大数的加法和乘法实现。快速幂减少了多次相乘的计算量，显著提升效率。

实现高精度运算需要特别注意进位处理和前导零的处理。

**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，编号依次为A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K。实现一个牌堆，可以对其进行Append、Extract、Revert、Pop操作。  
执行完所有指令后，最后输出剩余牌堆的牌，从牌堆顶到牌堆底，若牌堆为空则输出NULL。

**2.5.2 基本要求**

 **Append**：添加一张扑克牌到牌堆的底部。

 **Extract**：从牌堆中抽取某种花色的所有牌，排序后放到牌堆顶部。

 **Revert**：使牌堆逆序。

 **Pop**：弹出牌堆顶部的牌并输出，如果牌堆为空则输出NULL。

**2.5.3 数据结构设计**

花色：使用字符串存储（如"Spade", "Heart"等）。

点数：用数字代表字母表示编号（如"A", "10", "J", "Q", "K"）。

使用双向链表模拟牌堆，可以方便地实现从两端插入和删除操作。

反转通过参数来操作（见下）。

int rev=0; //初始为0，只有0和1两种状态（这个便于进行revert操作）

struct Node {

string color; // 牌的花色

int num; // 点数，1: A，2-10: 对应数字，11: J，12: Q，13: K

Node\* prev\_next[2]; //始终用第rev个元素表示前一个元素，第1-rev个元素表示后一个元素（这样仅需改变rev即可将牌堆反转）

};

**2.5.4功能说明（函数、类）**

#### Append 函数

/\*\*

\* @brief 将扑克牌插入到牌堆的底部。

\* @param color 牌的花色，int类型（0: 黑桃，1: 红心，2: 梅花，3: 方块）

\* @param num 牌的编号，int类型（1: A, 2-10: 数字, 11: J, 12: Q, 13: K）

\*/

void append(int color, int num) {

创建一张扑克牌;

将扑克牌添加到链表尾部;

}

#### Pop 函数

/\*\*

\* @brief 弹出牌堆顶部的牌。

\* @return 返回扑克牌的花色和编号，如果牌堆为空则返回NULL。

\*/

void pop() {

if (牌堆为空) {

输出 "NULL";

}

else {

弹出链表头部的扑克牌;

根据花色和点数输出扑克牌;

}

#### }Revert 函数

/\*\*

\* @brief 将牌堆逆序。

\*/

void revert() {

调整链表的头和尾节点指针;

rev如果是0则变为1，为1变为0

#### }Extract 函数

/\*\*

\* @brief 抽取某种花色的牌，排序后插入到牌堆顶部。

\* @param color 需要抽取的花色

\*/

void extract(string color) {

创建一个临时链表保存该花色的牌;

删除牌堆中该花色的牌;

按照点数对牌进行排序;

将排序后的牌插入链表头部;

}

#### 输出牌堆

/\*\*

\* @brief 输出牌堆中所有扑克牌，从顶部到底部。

\*/

void print\_() {

if (牌堆为空) {

输出 "NULL";

}

else {

遍历链表，从头到尾输出扑克牌;

输出花色，并使用映射函数将int 点数转换为对应的字符串表示;

}

}

/\*\*

\* @brief 将int类型的点数转换为字符串

\* @param num 点数int值

\* @return 返回对应的点数字符串

\*/

string num\_string(int num) {

if (num == 1) return "A";

if (num >= 2 && num <= 10) return to\_string(num);

if (num == 11) return "J";

if (num == 12) return "Q";

if (num == 13) return "K";

return "";

}

**2.5.5 调试分析（遇到的问题和解决方法）**

**问题：链表的反转耗时过长，容易超时**

解决方法：使用一个rev标记表示反转，把时间复杂度从o（n）降到了o（1）。

**问题：使用string类直接保存点数，不变比较大小**

解决方法：将点数转换为对应的int型变量，之后再转换成string类输出。

**边界处理，尤其是Pop和print函数中，要尤其注意越界问题**

**2.5.6 总结和体会**

本题的关键在于需要实现对牌堆的各种操作，包括在底部插入牌、从顶部弹出牌、反转牌堆顺序以及根据花色提取和排序牌。由于需要频繁地对牌堆进行插入、删除和逆序等操作，选择双向链表是非常合适的，因为它能在头部和尾部都支持快速的插入和删除操作，同时指针的灵活性使得反转操作能够在 O(1) 时间内完成。

而相对来说较为复杂的反转，也通过使用一个int型变量进行标记的方法，成功以o(1)复杂度完成。

**3. 实验总结**

这些实验重点探讨了线性表的一些应用，通过使用顺序表和链表两种不同类型的线性表来解决问题。在实验过程中，我对这两种数据结构的特性、使用场景以及操作细节有了更深入的理解。

**线性表类型的选择**

在解决实际问题时，我们需要根据操作的频率和特性来选择合适的线性表存储方式。线性表的两种常见实现方式：顺序表和链表，各有其优势和适用场景。

1. 顺序表：适用于需要频繁查找且较少进行插入和删除操作的场景。顺序表的元素存储在连续的内存空间中，因此可以通过索引快速访问，查找效率较高。然而，如果涉及到频繁的插入和删除操作，顺序表需要移动大量元素，会导致效率低下。因此，对于一次性导入大量数据后主要进行查找的应用场景，顺序表是更优的选择。
2. 链表：链表在插入和删除操作上具有显著优势，尤其是在需要频繁修改数据的情况下。链表通过指针连接各个节点，插入和删除操作只需要调整指针即可，不需要移动其他元素，效率高。尤其是\*\*双向链表\*\*，它支持在两端高效进行插入和删除操作，适合那些需要频繁修改表中数据的应用场景。相比之下，链表的随机访问性能较差，查找操作需要从头遍历，时间复杂度为 O(n)。

**链表构造中的注意事项**

在构造链表的过程中，特别是在双向链表中，指针的指向问题至关重要。链表的每个节点不仅要维护当前节点的值，还需要正确地指向前后节点。在进行插入、删除、反转等操作时，需要仔细调整指针，以确保链表的结构不会破坏。例如，在双向链表中，插入一个节点时需要同时调整前后节点的next和prev指针，反转链表时则需要调整next和 prev 指针，并重新设定头尾节点。

**实验中的收获与思考**

通过本次实验，我不仅加深了对线性表的理解，也提高了实际应用数据结构解决问题的能力。尤其是在处理链表的操作时，我更加意识到指针操作的复杂性和精确性。例如，反转链表或删除节点时稍有不慎就可能导致链表断裂或出现内存泄漏问题。这要求我们在编写代码时不仅要关注算法逻辑，还要充分考虑内存管理和指针的正确性。

总之，本次实验让我在理论与实践相结合的过程中，深入理解了线性表的操作和应用，并为今后更复杂的数据结构学习奠定了坚实的基础。在今后的编程实践中，我将继续总结和优化，不断提高数据结构的应用能力。