作业 HW1* 实验报告

姓名：　　学号　　日期：2024 年 10 月 4 日

1. 涉及数据结构和相关背景

本次实验基于线性表的相关知识，运用基本的线性表操作（如线性表的初始化、赋值、求长度等）进行对应的程序设计，实现题目要求的功能。具体为：1.一维静态数组及其操作；2.链表及其操作。

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。
输出：轮转后的顺序表中的元素。

2.1.3 数据结构设计

方法一&二：定义并初始化一个数组：

```
int nums[100010] = { 0 };  //题中说明数组长度不超过10^5，故初始化一个值为0，长度为100010的数组

int main() {

    int length = 0, move = 0;   // length为数组长度，move为移动次数
    cin >> length >> move;

    move = move % length;       // 防止移动次数超过数组长度，取其余数
```

方法三：建立一个单循环链表

```
/* 创建节点对象 */
class Node {
public:

    int data;       // 存储每个节点的数据
    Node* next;     // 指向下一个节点的指针

};

/* 创建链表对象 */
class Linklist {
private:
    Node* head;  // 创建头指针
    Node* last;  // 创建尾指针

public:
    Linklist() {
        /* 初始化头指针和尾指针(单循环链表) */
        head = new Node();       // 指向虚拟头节点
        last = nullptr;
    }

    /* 用于初始化数据的函数 */
    void initialize(int value);

    /* 用于打印移动结果的函数 */
    void print(int length, int move);

};
```

2.1.4 功能说明（函数、类）

方法一：建立一维静态数组，按照原始数据的顺序读入数据。在输出时，先从倒数第 k 个元素输出到最后一个元素，再从第一个元素输出到第 n-k 个元素。
这样做的时间复杂度为 O(n)，空间复杂度为 O(1)

```
for (int i = 0; i <= length - 1; i++) {
    cin >> nums[i];
}                          // 循环length次，读入原始顺序的元素

for (int i = (length - move + 1) - 1; i <= length - 1; i++) {
    cout << nums[i] << " ";
}                          //先输出：第length - move + 1个元素至最后一个元素

for (int i = 0; i <= (length - move) - 1; i++) {
    cout << nums[i] << " ";
}                          //再输出：按照原始顺序，输出原始数组的第一个元素至第length - move个元素
```

方法二:建立一维静态数组，存储时，先将第一个数据到第 n-k 个数据存储在数组的第 k+1 个位置到第 n 个位置；再将第 n-k+1 个数据到第 n 个元素存储在数组的第 1 个位置到第 k 个位置
这样做的时间复杂度为 O(n)，空间复杂度为 O(1)

```
for (int i = (move + 1) - 1; i <= length - 1; i++) {
    cin >> nums[i];
}       // 先输入：第move + 1个位置的元素至最后位置的一个元素
for (int i = 0; i <= move - 1; i++) {
    cin >> nums[i];
}       // 再输入：第一个位置的元素至第move个位置的元素

for (int i = 9; i <= length - 1; i++) {
    cout << nums[i] << " ";
}       // 按照存储顺序输出
```

方法三：建立一个单循环链表，按照原始数据的顺序读入数据，输出时找到第 n-k+1 个节点，从其正向输出一圈(即输出到第 n 个节点后，从第一个节点输出到第 n-k 个节点)
这样做的时间复杂度位 O(n)，空间复杂度位 O(1)

```
Linklist list;    //创建单循环链表

int value = 0;   // value用于存储当前输入的值

/* 循环length次，读入原始顺序的元素 */
for (int i = 1; i <= length; i++) {
    cin >> value;
    list.initialize(value);
}

/* 输出结果 */
list.print(length, move);
```

2.1.5 调试分析（遇到的问题和解决方法）

调试过程中首先输入样例中给出的测试数据，查看程序的输出是否一致，确保一定的输出准确性；

再使用一些特殊的边界数据进行测试。如本题中可输入 k>n 的数据，来检验是否能够正确轮转输出。

2.1.6 总结和体会

本题虽然思路比较简单，但是要求运用至少三种方法进行程序设计，这就对思维要求更高，同时还要联系课堂所学知识，对思考题进行解答。总之，做完此次题目，我觉得我的思维更开阔，

同时对课堂所学的理解也更深刻了。

2.2 求级数

2.2.1 问题描述

求级数：$A+2A^2+3A^3+\cdots\cdots+NA^N=\sum_{i=1}^{N} iA^i$ 的整数值

2.2.2 基本要求

输入：若干行，在每一行中给出整数 N 和 A 的值，（1<=N<=150，0<=A<=15）
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

输出：对于每一行，在一行中输出级数的值

2.2.3 数据结构设计

```
vector <int> result(300);   // 用于存储总和的数组
vector <int> Power(1, 1);     // 用于记录幂的数组，初始化为 1
vector <int> item(300);      // 用于记录当前项的数组

int Size = 0;   // 用于记录最终结果的长度
```

2.2.4 功能说明（函数、类）

此处设计了三个函数（1.计算幂的函数；2.计算当前项的函数；3.求和函数）以便后续调用

```
/* 用于计算幂的函数 */
void Multiplication_power(int A) {    // 传入参数为底数

    int carry = 0;                                  // 表示上一位的进位
    for (int i = 0; i <= Power.size() - 1; i++) {
        int Bit = Power[i] * A + carry;             // 计算当前位的值
        Power[i] = Bit % 10;                        // 保留当前位
        carry = Bit / 10;                           // 进位
    }
    /* 检查是否需要新的位 */
    while (carry) {
        Power.push_back(carry % 10);            // 添加进位
        carry /= 10;
    }
```

某些项求幂后会超过 long long 型的数据范围，所以改用数组存储每一位的值，防止数据溢出产生的错误。

```
/* 用于计算当前项的函数 */
int Multiplication_coefficient(int coefficient) {    // 传入参数为系数

    int carry = 0, length = Power.size();
    for (int i = 0; i <= length - 1; i++) {
        int Bit = Power[i] * coefficient + carry;    // 计算当前位的值
        item[i] = Bit % 10;                          // 保留当前位
        carry = Bit / 10;                            // 进位
    }
    /* 检查是否需要新的位 */
    while (carry) {
        item[length] = carry % 10; // 添加进位
        length++;
        carry /= 10;
    }

    /* 返回当前总位数 */
    return length;
```

```c
/* 用于整体求和的函数 */
int addition(int length) {                          // 传入参数为当前总位数

    int carry = 0;
    for (int i = 0; i <= length - 1; i++) {
        int Bit = item[i] + result[i] + carry;      // 计算当前位的值
        result[i] = Bit % 10;                       // 保留当前位
        carry = Bit / 10;                           // 进位
    }
    /* 检查是否需要新的位 */
    while (carry) {
        result[length] += carry % 10;               // 添加进位
        length++;
        carry /= 10;
    }

    /* 返回当前总位数 */
    return length;
}
```

2.2.5 调试分析（遇到的问题和解决方法）

数据测试的过程中发现，某些项求幂后会超过 long long 型的数据范围，所以我修改了第一次用 long long 型存储当前项的方法，改用数组存储每一位的值，防止数据溢出产生的错误。

2.2.6 总结和体会

基于调试过程中发现的问题，对程序进行多次优化：1.计算过程模拟竖式乘法，先计算每一位的值，再进位；2.由于该题的前一项与后一项的次数仅差一位，所以每次更新幂的计算结果以便下一项计算节约时间

优化过程中提高了自己发现问题、解决问题的能力。

2.3 扑克牌游戏

2.3.1 问题描述

对于一个扑克牌堆，定义以下 4 种操作命令：1) 添加 2) 抽取 3)反转 4)弹出

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。

对于 20%的数据，n<=20，有 Append、Pop 指令；

对于 40%的数据，n<=50，有 Append、Pop、Revert 指令；

对于 100%的数据，n<=200，有 Append、Pop、Revert、Extract 指令。

2.3.2 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，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

2.3.3 数据结构设计

定义两个类，4 个函数（对应四种操作）

```
/* 创建每张牌的对象 */
class card {
public:
    string nums;              /* 扑克牌的数字 */
    string type;              /* 扑克牌的花色 */
    card* left = NULL;        /* 指向上一张扑克牌的指针*/
    card* right = NULL;       /* 指向下一张扑克牌的指针 */

};
/* 创建牌堆的对象 */
class Decks {
private:
    card* head;     /* 创建头指针 */
    card* last;     /* 创建尾指针 */

public:

    /* 初始化头指针和尾指针 */
    Decks() {
        head = NULL;
        last = NULL;
    }

    /* 添加一张扑克牌至牌堆的底部 */
    void Append(string type, int nums);

    /* 抽取指定花色的所有扑克牌 */
    void Extract(string type);

    /* 使扑克牌顺序颠倒 */
    void Revert();

    /* 去除牌堆顶部的扑克牌 */
    void Pop();

    /* 打印牌堆的所有扑克牌 */
    void Print();

};
```

2.3.4 功能说明（函数、类）

```
if (head == NULL) {

    head = newcard;             // 更新头指针
    last = newcard;             // 更新尾指针
    newcard->left = NULL;       // 左指针指向空
    newcard->right = NULL;      // 左指针指向空
    return;

}

last->right = newcard;          // 使原始牌堆的最后一张扑克牌的左指针指向新节点
newcard->left = last;           // 新节点的右指针指向原始牌堆的最后一张扑克牌
newcard->right = NULL;          // 新节点的左指针指向空
last = newcard;                 // 更新尾指针
```

关于 Append 操作的解决：
设置尾指针，便于判断牌底的扑克牌，插入新的扑克牌后，更新尾指针

```
else {

    copy = temp;

    temp = temp->right;      // 更新临时指针

    /* 解决左右相邻节点的关系*/
    if (copy->right == NULL) {       // 判断是否是最后一个节点
        (copy->left)->right = NULL;
        last = copy->left;           //  更新尾指针
    }
    else {
        (copy->left)->right = copy->right;
        (copy->right)->left = copy->left;
    }

    /* 解决移至牌顶的问题 */
    copy->left = NULL;
    copy->right = head;
    head->left = copy;
    head = copy;                     // 更新头指针

    continue;
}

/* 交换左指针和右指针 */
while (current != NULL) {

    nextTemp = current->right;               // 保存下一个节点

    current->right = current->left;
    current->left  = nextTemp;

    current = nextTemp;                      // 移动到下一个节点
}

/* 交换头指针和尾指针 */
current = head;
head = last;
last = current;

/* 打印牌堆顶部扑克牌的花色和数字 */
if (head->nums == "B") {
    cout << head->type << " " << "10" << endl;
}
else if (head->nums == "1") {
    cout << head->type << " " << "A" << endl;
}
else if (head->nums == "Z") {
    cout << head->type << " " << "K" << endl;
}
else {
    cout << head->type << " " << head->nums << endl;
}
```

关于 Extract 操作的解决：

先遍历链表，将所有符合花色的扑克牌都移动至牌顶(先不排序)

对符合花色的扑克牌的数据进行排序(不对链表的节点排序，而是对数据排序)，将排序后的数据赋值给每个链表节点

关于 Revert 操作的解决：

选择双链表结构，不移动链表的节点，而是依次交换每个节点左右指针，最后交换头尾指针

关于 pop 操作的解决：

设置头指针

1.判断链表是否为空

2.判断牌顶的扑克牌

每次弹出扑克牌，更新头指针

2.3.5 调试分析（遇到的问题和解决方法）

排序时的简化操作：关于扑克牌数值的比较——将 A 改为 1，将 10 改为 B，将 K 改为 Z，这样可以直接用 ASCII 码进行排序。不过输出时记得改回。

2.3.6 总结和体会

使用改良方法时一定不要脱离原有题目！！比如我在简化操作的时候忘记把 1 改回 A，导致程序一直执行错误。

2.4 一元多项式的相加和相乘

2.4.1 问题描述

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）
对于%15 的数据，有 1<=n,m<=15；对于%33 的数据，有 1<=n,m<=50
对于%66 的数据，有 1<=n,m<=100；对于 100%的数据，有 1<=n,m<=2050

2.4.2 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，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：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。

2.4.3 数据结构设计

```
/* 系数在前，指数在后*/
int polynomial_1[2100][2] = { 0 };     // 创建数组用于存储第一个多项式
int polynomial_2[2100][2] = { 0 };     // 创建数组用于存储第二个多项式
int result_add[2100] = { 0 };       // 创建数组用于存储加法运算的结果
int result_mul[4210][2] = { 0 };       // 创建数组用于存储乘法法运算的结果
```

2.4.4 功能说明（函数、类）

2 个函数

```
void addition(int n) {          // 传入的参数为第二个多项式的长度

    int index = 0;          // 用于临时存储指数

    /* 在输入第一个多项式时，也存储在了加法结果数组 */
    for (int i = 1; i <= n; i++) {

        index = polynomial_2[i][1];                 // 获取此时的指数
        result_add[index] += polynomial_2[i][0];    // 与第二个多项式的相应指数项相加

    }

    /* 遍历存储加法结果的数组，如果不为0就输出 */
    for (int i = 0; i <= 2050; i++) {
        if (result_add[i] != 0) {
            cout << result_add[i] << " " << i << " ";
        }
    }
    cout << endl;
```

```
void multiplication(int m, int n) {      // 传入的参数分别为两个多项式的长度

    int index = 0;          // 用于临时存储指数

    /* 嵌套循环，第一个多项式的每一项与第二个的多项式的每一项相乘 */
    for (int i = 1; i <= m; i++) {
        for (int k = 1; k <= n; k++) {

            index = polynomial_1[i][1] + polynomial_2[k][1];
            result_mul[index][0] += (polynomial_1[i][0] * polynomial_2[k][0]);

        }
    }
        /* 遍历存储乘法结果的数组，如果不为0就输出 */
    for (int i = 0; i <= 4100; i++) {
        if (result_mul[i][0] != 0) {
            cout << result_mul[i][0] << " " << i << " ";
        }
    }
    cout << endl;
}
```

2.4.5 调试分析（遇到的问题和解决方法）

调试过程中发现时间复杂度有些高，因此采用以下方法——存储计算结果的数组用下标存储指数，避免了因为判断两个多项式是否存在某个指数而浪费时间。

2.4.6 总结和体会

该题目主要使用线性表结构(主要为一维数组)。通过此题了解到多项式数组存储指数和系数，可以避免因为遍历指数而浪费时间。是一个值得积累的算法思想。

2.5 学生信息管理

2.5.1 问题描述

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。

2.5.2 基本要求

输入&输出：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)

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。

2.5.3 数据结构设计

```
class Node {
public:

    string nums;            // 学号
    string name;            // 姓名
    Node* left = NULL;      // 指向上一个节点
    Node* right = NULL;     // 指向下一个节点
```

```
class Linklist {
private:
    Node* head;   // 创建头指针
    Node* last;   // 创建尾指针

public:

    /* 初始化头指针和尾指针 */
    Linklist() {
        head = NULL;
        last = NULL;
    }

    /* 用于初始化的函数 */
    void initialize(string name, string nums);            // 传入参数值分别为学生姓名和学号

    /* 用于插入的函数 */
    int insert(int position, string name, string nums);   // 传入参数值分别为位置，学生姓名和学号

    /* 根据姓名查找 */
    void check_name(string name);                          // 传入值为姓名

    /* 根据学号查找*/
    void check_nums(string nums);                          // 传入值为学号

    /* 用于删除的函数 */
    int remove(int position);                             // 传入值为位置

    /* 输出学生总数 */
    void end();
```

2.5.4 功能说明（函数、类）

5 个函数对应五种操作

```
void Linklist::initialize(string name, string nums) {

    Node* newnode = new Node();     // 为新节点开辟空间

    newnode->name = name;
    newnode->nums = nums;

    /* 判断链表是否为空 */
    if (head == NULL) {

        newnode->left = NULL;
        newnode->right = NULL;
        head = newnode;             // 更新头指针
        last = newnode;             // 更新尾指针
        return;

    }

    last->right = newnode;
    newnode->left = last;
    newnode->right = NULL;
    last = newnode;                 // 更新尾指针
```

Insert

```
int Linklist::insert(int position, string name, string nums) {

    Node* newnode = new Node();     // 为新节点开辟空间

    newnode->name = name;
    newnode->nums = nums;

    /* 判断位置是否非法 */
    if ((position > total + 1) || (position < 1)) {
        return -1;
    }

    total++;                // 学生人数增加


    /* 判断链表是否为空 */
    if (head == NULL) {

        head = newnode;          // 更新头指针
        last = newnode;          // 更新尾指针
        newnode->left = NULL;
        newnode->right = NULL;
        return 0;

    }

    Node* temp = head;       // 创建临时指针

    for (int i = 2; i <= position; i++) {

        temp = temp->right;    // 更新临时指针
```

关于 insert 函数：
遍历链表，到达目标位置

```
/* 判断是否在头部 */
if (temp == head) {

    head->left = newnode;
    newnode->left = NULL;
    newnode->right = head;
    head = newnode;              // 更新头指针
}

/* 判断是否在尾部*/
else if (temp == NULL) {

    last->right = newnode;
    newnode->left = last;
    newnode->right = NULL;
    last = newnode;             // 更新尾指针

}

else {

    (temp->left)->right = newnode;     // 目标位置的上一个节点的右指针指向新节点
    newnode->left = temp->left;        // 新节点的左指针指向目标位置的上一个节点
    temp->left = newnode;              // 目标位置的左指针指向新节点
    newnode->right = temp;             // 新节点的右指针指向目标位置
```

Check_name

```
void Linklist::check_name(string name) {

    Node* temp = head;        // 创建临时指针
    int position = 1;

    while (temp != NULL) {

        /* 判断是否符合 */
        if (temp->name == name) {

            cout << position << " " << temp->nums << " " << temp->name << endl
            break;

        }

        position++;             // 更新位置
        temp = temp->right;     // 更新临时指针

    }

    /* 判断是否找到目标信息 */
    if (temp == NULL) {

        cout << -1 << endl;
```

关于 check 函数
遍历链表，找到匹配的第一个节点

Check_nums

```
void Linklist::check_nums(string nums) {

    Node* temp = head;        // 创建临时指针
    int position = 1;

    while (temp != NULL) {

        /* 判断是否符合 */
        if (temp->nums == nums) {

            cout << position << " " << temp->nums << " " << temp->name << endl;
            break;

        }

        position++;             // 更新位置
        temp = temp->right;     // 更新临时指针

    }

    /* 判断是否找到目标信息 */
    if (temp == NULL) {

        cout << -1 << endl;
```

Remove

```
int Linklist::remove(int position) {

    /* 判断位置是否合法 */
    if ((position > total) || (position < 1)) {
        return -1;
    }

    Node* temp = head;          // 创建临时指针

    for (int i = 2; i <= position; i++) {

        temp = temp->right;    // 更新临时指针

    }

    /* 判断此时学生人数 */
    if (total == 1) {
        head = NULL;
        last = NULL;
        total--;
        return 0;
    }

    /* 判断是否在头部*/
    if (position == 1) {

        head = head->right;    // 更新头指针
        head->left = NULL;

    }

    /* 判断是否在尾部 */
    else if (position == total) {

        last = last->left;    // 更新尾指针
        last->right = NULL;

    }

   else {

      (temp->left)->right = temp->right;      // 目标位置的上一个节点的右指针指向目标位置的下一个节点
      (temp->right)->left = temp->left;       // 目标位置的下一个节点的右指针指向目标位置的上一个节点

   }

   total--;                  // 学生人数减少

   return 0;
```

关于 remove 函数:
遍历链表，到达目标位置

2.5.5 调试分析（遇到的问题和解决方法）

调试时发现 linux 系统和 windows 系统的 ASCII 表好像有差异。oj 为为 linux 系统，用 getline 读数据，换行符 windows 里是读\n，linux 系统是读\r，所以在 oj 写程序时，尽量使用 cin，这样不容易出错。

2.5.6 总结和体会

由于此题的操作主要涉及插入，删除的操作，所以采用链表结构。

3. 实验总结

经过此次实验作业，我熟悉了线性表的基本操作，并能运用到解决实际问题的实践中。同时，在独立调试解决问题的过程中，深感个人发现问题和与别人讨论问题的能力显著提高。