作业 HW1 实验报告

姓名：刘彦 学号：2352018 日期：2024 年 10 月 2 日

1. 涉及数据结构和相关背景

线性表：用于存储一组有序的元素。线性表的元素具有顺序性和线性关系。通过节点存储线性表的元素，节点包含数据部分和指向下一个节点的指针，支持动态大小的插入和删除。一般分为顺序表和链表。

顺序表：

- 定义：使用连续的内存空间存储元素。
- 优点：支持快速随机访问，存储密度高。
- 缺点：插入和删除操作较慢，需要移动元素；扩容时可能导致性能下降。

链表：

- 定义：由多个节点组成，每个节点包含数据和指向下一个节点的指针。
- 优点：插入和删除操作高效，不需要连续内存。
- 缺点：随机访问较慢，占用额外的指针空间，可能导致空间浪费。

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。

数据范围：

$$1 <= nums.length <= 105$$

$$-231 <= nums[i] <= 231 - 1$$

$$0 <= k <= 105$$

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。

尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。分别讨论其时间复杂度。

2.1.3 数据结构设计

```
struct ListNode {
    int val;            // 节点的值
    ListNode* next;     // 指向下一个节点的指针
    ListNode(int x) : val(x), next(nullptr) {} // 构造函数，初始化节点值和指针
};
```

2.1.4 功能说明（函数、类）

```
//接收链表头指针的引用和旋转次数 k
/**
 * @brief           旋转链表
 * @param   head    链表的头指针
 * @param   k       需要旋转的节点数
 */
void rotate(ListNode*& head, int k) {
    if (head 为空或 k == 0) return;
    通过指针计算链表的长度
    tail->next = head;  // 形成循环链表
    // 找到新的头节点位置
    k = k % length;  //处理 k 大于长度的情况
    通过 steps - 1 次循环，将 newTail 移动到新的尾节点位置
    更新新头节点并断开循环
}
```

main 函数中关键部分如下

```
    ListNode* head = nullptr;  //指向链表的头节点，初始为空
    ListNode* tail = nullptr;  //指向链表的尾节点，初始为空
    for (int i = 0; i < n; i++) {  //每次循环中，读取一个整数并用该值创建一个新的链表节点 newNode
        int value;
        cin >> value;
        ListNode* newNode = new ListNode(value);
        if (!head) {  //如果 head 为空（即链表还未创建），将 newNode 赋值给 head 和 tail，表示链表的开始
            head = newNode;
            tail = head;
        }
        else {  //如果链表已存在，将 newNode 添加到尾部：更新 tail->next 指向新节点,将 tail 更新为 newNode，使其指向新的尾节点
            tail->next = newNode;
            tail = newNode;
        }
    }
```

2.1.5 调试分析（遇到的问题和解决方法）

2.1.5.1 旋转次数 k 大于链表长度

在实验时发现，当 k 大于链表长度时，旋转效果与 k = k % length 相同，如果不处理，将导致不必要的循环和错误。最终使用取模运算进行简化。

2.1.5.2 链表形成循环后未断开

在实验尝试时，曾经出现无限循环的情况。经检查发现，在形成循环链表后，如果没有正确断开，将导致在后续操作中出现无限循环。解决方法：在找到新的头节点后，确保将新尾节点的 `next` 指向 `nullptr`。

2.1.5.3 内存释放

在检查代码并上过其他相关课程后，使用 new 分配节点后，如果没有适时释放，可能会导致内存泄漏。经检查，最终在 main 函数的最后，使用循环遍历并删除所有节点。

```
// 清理内存
    while (head) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
    }
```

2.1.6 总结和体会

2.1.6.1 算法实现

在上述代码中，链表的旋转操作是通过原地算法实现的，时间复杂度 `O(n)`，需要遍历链表计算长度和寻找新头尾。空间复杂度 `O(1)` ，只使用常量空间。将链表的尾节点指向头节点，构成一个循环。这一操作在原地修改了链表结构。并通过断开 `newTail->next` 指针，实现了链表的旋转。这种方法不仅提高了算法的空间效率，还避免额外的内存使用。

2.1.6.2 其他解题方法

还可以通过双指针法和数组法来解决此题，以数组法为例，空间复杂度也为 `O(1)`。

```
void rotate(int nums[], int n, int k) {
    k = k % n;  // 优化 k 值
    reverse(nums, 0, n - 1);  // 反转整个数组
    reverse(nums, 0, k - 1);  // 反转前 k 个元素
    reverse(nums, k, n - 1);  // 反转剩余部分
}
```

2.2 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

输入输出要求：

- 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

```
struct Student {
    char student_id[MAX_ID_LENGTH]; // 学生 ID
    char name[MAX_NAME_LENGTH];       // 学生姓名
    Student* next;                   // 指向下一个学生的指针
};

struct SequentialList {
    Student* head;  // 指向链表第一个学生的指针
    int size;       // 链表中学生的数量

    // 构造函数，初始化空链表
    SequentialList() {
        head = nullptr;  // 初始化时没有学生
        size = 0;        // 大小为 0
    }

    // 析构函数，释放链表内存
    ~SequentialList();

    int insert(int index, const char* student_id, const char* name);
    int remove(int index);
    void checkByName(const char* name);
    void checkById(const char* student_id);
    int count() const;
};
```

2.2.4 功能说明（函数、类）

在指定索引插入新学生的函数

```
/**
 * @brief          在顺序表中指定索引处插入一个新学生
 * @param   index   插入位置的索引
 * @param   student_id  新学生的 ID
 * @param   name  新学生的姓名
```

```
 * @return          成功返回 0，失败返回-1
 */
int SequentialList::insert(int index, const char* student_id, const char* name) {
    if (无效索引)
        return -1;
    创建 newStudent  // 创建新学生
    复制 student_id 到 newStudent->student_id，复制 name 到 newStudent->name
    初始化 next 指针
    if (索引为 1)   // 新学生指向当前头部,然后更新头部为新学生
        newStudent->next 指向 head，新学生 head 更新为 newStudent;
    } else {
        设置 current 为 head;  // 遍历到插入位置前一个学生
        循环所有 current 都更新为 current->next;  // 移动到下一个学生
        }
        // 将新学生连接到后一个学生，然后将当前学生连接到新学生
        newStudent->next 指向 current->next;current->next 指向 newStudent;
    }
    增加链表大小
    return 0;
}
```

在指定索引删除学生的函数

```
/**
 * @brief           从链表中删除指定索引的节点
 * @param   index   要删除节点的索引
 * @return          成功返回 0，失败返回-1
 */
int SequentialList::remove(int index) {
    if (无效索引)
        return -1;
    Student* toDelete 指向要删除的学生

    if (index == 1) {
        获取头部学生，然后将头部移到下一个学生
    } else {
            遍历到删除位置前一个学生
            遍历将每一个 current 更新为 current->next;  // 移动到下一个学生
        }
        获取要删除的学生的地址并跳过
    }
    释放内存并减少链表大小
    return 0;
}
```

按姓名/ID 检查学生的函数(两个函数内部相同，只有一句输出语句不同，已用注释注明)

```
/**
 * @brief           在顺序表中检查是否存在指定姓名的学生/ID
 * @param   name    要检查的学生姓名
 * @param   student_id  要检查的学生的 ID
 */
void SequentialList::checkByName(const char* name) {
void SequentialList::checkById(const char* student_id) {
        从头部开始，当前节点不为空时，循环
        if (当前节点的姓名等于 name) {
            输出索引、ID 和姓名（current 指向 ID/name 中不被检查的项）
            return;  // 找到姓名/ID
        }
        current 移动到下一个学生
    }
    cout << -1 << endl;  // 未找到姓名/ID
}
```

2.2.5 调试分析（遇到的问题和解决方法）

2.2.5.1 时间限制

最初尝试使用顺序表的尝试，由于在一些情况查询时间过长，在学生人数较多的情况下每一步删除和添加的操作都会耗费很长时间，导致测试点超过时间限制，因此最终使用链表进行存储。

2.2.5.2 索引越界

在改用链表后，在插入或删除操作中，未正确处理边界条件（如插入位置为 0 或超过链表长度），导致程序错乱，无法正确查询，个别测试点无法通过。最后添加断点进行测试，在插入和删除操作的开始，添加条件检查，并注意了边界的范围是 1 到 size。并进行边界条件测试，测试链表为空、插入到头部和尾部、删除最后一个元素等边界情况，确保没有越界错误。

2.2.5.3 循环链表未断开

在进行旋转操作时，未正确断开循环链表，导致后续操作出现无限循环，无法正确查询。最后在添加断点后，明确设置新尾节点的 next 指向 nullptr，避免形成循环。

2.2.6 总结和体会

在做题前要仔细分析题目，对题目需要的数据结构进行判断，确定使用哪种线性表。顺序表的缺点：插入和删除操作需要移动大量元素，扩容时可能需要重新分配内存。而链表插入和删除操作效率高，尤其是在头部或尾部操作；不需要预先分配空间。所以若需大量插入和删除操作，要使用链表，这也是本题的要求。

如果题目需要频繁插入和删除操作，尤其是在中间位置，选择链表更为合适。如果需要频繁查找，顺序表可能会表现得更好，特别是在小规模数据时。但使用时需要考虑链表的空状态、头尾节点操作，以及插入或删除时的索引有效性。

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((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。

数据范围：

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

2.3.2 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 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。

2.3.3 数据结构设计

```
// 定义结构体表示多项式项
struct Term {
    int coefficient;
    int exponent;
};
Term polyA[maxTerms], polyB[maxTerms]; // 存储多项式的结构体数组
int sumResult[maxTermsPow2];           // 存储加法结果的数组
int productResult[maxTermsPow2];       // 存储乘法结果的数组
```

2.3.4 功能说明（函数、类）

加法函数

```
/**
 * @brief           将两个多项式相加
 */
void AddPolynomials() {
    memset 确保加法结果数组初始化为 0
```

```
    for (i 从 0 到 maxTerms - 1) 循环
        sumResult[polyA[i].exponent] += polyA[i].coefficient;
        sumResult[polyB[i].exponent] += polyB[i].coefficient;
        //将多项式 A 的项加到结果中
}
```

乘法函数

```
/**
 * @brief           将两个多项式相乘
 */
void MultiplyPolynomials() {
    memset 确保乘法结果数组初始化为 0
    for (i 从 0 到 maxTerms - 1) 循环嵌套 for (j 从 0 到 maxTerms - 1) 循环
            if (非零项)
               //计算当前乘积项的指数，将当前乘积（系数相乘）累加到结果数组中
                 对应的指数位置
                productResult[polyA[i].exponent + polyB[j].exponent] +=
                    polyA[i].coefficient * polyB[j].coefficient;
}
```

输入函数

```
/**
 * @brief           输入多项式的项
 * @param   polynomial  指向多项式项数组的指针
 * @param   numTerms    多项式的项数
 */
void InputPolynomial(Term* polynomial, int numTerms) {
    for (i 从 0 到 maxTerms) 循环
        输入 coefficient 和 exponent
        polynomial[i] = {coefficient, exponent}; // 使用结构体初始化
}
```

输出函数

```
/**
 * @brief           打印多项式
 * @param   polynomial  指向多项式系数数组的指针
 */
void PrintPolynomial(int* polynomial) {
    bool hasOutput = false; // 用于判断是否有输出
    for (i 从 0 到 maxTermsPow2) 循环
        if (系数不为 0)
            输出系数和对应的指数，有输出
     如果没有输出，则输出 0 0
}
```

2.3.5 调试分析（遇到的问题和解决方法）

2.3.5.1 多项式的稀疏性

在调试过程中，一开始初始化结果数组时未考虑到多项式项的稀疏性，导致在输出时可能出现不必要的零项。为了解决这个问题，我确保了结果数组在计算前被清零，确保不会遗留旧数据。

2.3.5.2 复杂度和存储多项式的方式

在前面几次，我的代码的时空复杂度均为O(n×m)，会有很多测试数据无法通过。后来采用基于数组的多项式表示，这段代码时间复杂度 O(maxTerms²)，空间复杂度为O(maxTermsPow2)。减少了用结构体储存时在乘法操作中，可能会有较高的时间复杂度（尤其是多项式项数较多时）和合并过程复杂的问题。

2.3.6 总结和体会

在处理稀疏多项式时，使用固定大小数组可能导致内存浪费，并且对性能的影响需要特别关注。未能有效合并同类项可能会造成结果不正确。

在多项式加法和乘法中，通过使用固定大小的数组来存储结果，避免了动态分配内存的开销。这种方法使得空间使用更加高效，尤其在处理高次多项式时。加法的双循环遍历利用数组索引直接访问多项式的项，可以在 O(maxTerms)时间内完成多项式的加法，避免了在每次计算时进行复杂的条件判断。乘法的嵌套循环，尽管时间复杂度为O(maxTerms²)，但这种方法仍然通过检查非零系数来减少不必要的乘法操作，从而在实际运行中提高效率。

2.4 求级数

2.4.1 问题描述

求出形如

$$\sum_{i=1}^{N} iA^i = A + 2A^2 + \cdots + NA^N$$

的级数的值。

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

2.4.2 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）。

输出：对于每一行，在一行中输出级数的整数值。

2.4.3 数据结构设计

```
// 定义结构体表示大数
struct BigNumber {
    int digits[maxn];  // 存储数字
```

```
    BigNumber() {
        memset(digits, 0, sizeof(digits));  // 初始化为 0
    }
};
```

2.4.4 功能说明（函数、类）

打印大数的函数

```
/**
 * @brief           打印大数
 * @param   num  要打印的大数
 */
void print(const BigNumber& num) {
    bool started = false;  // 是否开始打印
    for (i 从 maxn - 1 到 0) 循环
        if (num.digits[i]) started = true;  // 找到第一个非零数字
        if (started) 打印数字; 如果没有打印任何数字，则输出 0
}
```

加法函数

```
/**
 * @brief           大数加法
 * @param   a  加数，结果将存储在此
 * @param   b  另一个加数
 */
void add(BigNumber& a, const BigNumber& b) {
    for (i 从 0 到 maxn) 循环
        a.digits[i] += b.digits[i];  // 相加对应位
        if (a.digits[i] >= 10)
            a.digits[i] -= 10; a.digits[i + 1] += 1;  // 处理进位
}
```

乘法函数

```
/**
 * @brief           大数乘法
 * @param   a  乘数
 * @param   b  被乘数，结果将存储在此
 */
void mul(const BigNumber& a, BigNumber& b) {
    初始化结果数组为 0
    for (i 从 0 到 maxn / 2) 循环嵌套 for (j 从 0 到 maxn / 2) 循环
            int newn = c.digits[i + j] + a.digits[i] * b.digits[j];
                                                    // 计算当前位的乘积
            c.digits[i + j] = newn % 10;                // 保存当前位
            处理进位（类似加法），并更新 b 为结果
}
```

main 函数中关键部分如下

```
for (int i = 1; i <= n; i++) {
        BigNumber I;
        I.digits[0] = i % 10;       // 当前 i 的数字
        I.digits[1] = i / 10;       // 下一位数字
        mul(A, Apow);         // 计算 A 的 i 次幂
        BigNumber MUL = Apow; // 存储当前幂的值
        mul(I, MUL);          // 将当前幂与 i 相乘
        add(ANS, MUL);        // 将结果加到总和中
    }
```

2.4.5 调试分析（遇到的问题和解决方法）

2.4.5.1 进位问题

在加法和乘法运算过程中，刚开始处理进位时可能会出错，导致最终结果不正确。最终在程序上添加断点，使用调试工具逐步执行代码，查看每个变量的值，尤其是在 mul 和 add 函数调用后。在加法/乘法函数中，确保每次加完一位后都进行进位的处理，并且在处理进位时，要确保不丢失高位的信息。

前导零处理：在进行加法和乘法时，可能会生成多余的前导零。需要在打印结果前进行剔除。当结果为零时，应确保至少输出一个零。

2.4.5.2 时间复杂度

为降低时间复杂度，曾尝试将乘法运算用快速幂的方法实现，结果有时反而效果变差。分析原因可能是高精度乘法通常涉及数组或其他数据结构的操作，每次乘法不仅仅是简单的整数乘法，还包括处理进位、数组复制等。这些额外的操作可能使得快速幂的效率下降，尤其是在基数较小、幂数不大的情况下，普通的逐次乘法可能更简单、直接。在这种情况下，逐次乘法的实现可能更快。

2.4.6 总结和体会

高精度运算的存储方式一般为逆序存储。逆序存储的好处在于可以直接从低位到高位进行计算，特别是在加法和乘法时，进位可以直接向后补位，避免了大量的位移操作。这样可以提高操作的效率，尤其是在处理多位数字时。

在逆序存储中，进位和借位的处理相对简单。比如在加法中，如果当前位和进位的和超过了基数（如 10），则可以直接计算新的当前位和进位，而不需要在数组中移动其他位。

除了本题中的加法和乘法外，高精度的减法和除法同样是采用逆序存储：

- 减法：需要检查被减数是否小于减数，若小，则需要借位。可以采用逐位减法的方式，从低位开始处理，同时处理借位。
- 除法：高精度除法通常比乘法和加法复杂，可能涉及到二分查找等算法来找到商。逆序存储同样可以简化操作，因为你可以逐位计算商并及时处理余数。

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。

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。

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

注意：每种花色和编号的牌数量不限。

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

2.5.2 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

```
typedef struct Card {
    char suit[128]; // 扑克牌花色
    char rank[4];   // 扑克牌点数
    int getValue() const { // 获取牌的数值
        switch (rank[0]) {
            case 'A': return 1;
            case 'J': return 11;
            case 'Q': return 12;
            case 'K': return 13;
            default: return (rank[1] == '0') ? 10 : rank[0] - '0'; // 其他牌
        }
    }
    bool operator<(const Card& other) const { // 重载小于运算符
        return this->getValue() > other.getValue(); // 比较牌的数值按降序排序
    }
} CardType;
CardType extractedCards[256]; // 存放提取的扑克牌
typedef struct Node {
    CardType card;      // 节点存放的扑克牌信息
    Node* previous;
```

```
    Node* next;
} *DoubleLinkedList;
```

2.5.4 功能说明（函数、类）

初始化双向链表的函数

```
/**
 * @brief           初始化双向链表
 * @param   list    双向链表的引用
 * @return          返回操作状态
 */
Status InitializeList(DoubleLinkedList& list) {
    list = (DoubleLinkedList)malloc(sizeof(Node)); // 申请内存
    if (内存分配失败) return LOVERFLOW;  // 检查内存分配
    初始化指针 previous 和 next
    return SUCCESS;
}
```

插入扑克牌的函数

```
/**
 * @brief           插入扑克牌到双向链表
 * @param   list    双向链表的引用
 * @param   card    扑克牌
 * @param   atHead  是否插入到头部
 * @return          返回操作状态
 */
Status InsertCard(DoubleLinkedList& list, CardType card, bool atHead) {
    申请新节点内存
    if (内存分配失败) return LOVERFLOW;  // 检查内存分配
    newNode->card = card; // 设置新节点的扑克牌
    if (插入到头部) {
        list->next->previous = newNode; // 调整指针
        newNode->next = list->next;
        list->next = newNode;
        newNode->previous = list;
    } else  // 如果插入到尾部
        将上面操作中 next 指针全部改为 previous 指针即可
    return SUCCESS; // 返回成功状态
}
```

移除扑克牌的函数

```
/**
 * @brief           从双向链表移除扑克牌
 * @param   list    双向链表的引用
 * @param   card    被移除的扑克牌
 * @param   fromHead 是否从头部移除
```

```
 * @return          返回操作状态
 */
Status RemoveCard(DoubleLinkedList& list, CardType& card, bool fromHead) {
    if (链表为空)
        设置返回值为 NULL; return SUCCESS;
    if (从头部移除) {
        获取下一个节点
        card = nextNode->card; // 保存被移除的扑克牌
        list->next = list->next->next; // 调整指针
        list->next->previous = list;
        释放内存
    } else  // 从尾部移除
            将上面操作中 next 指针全部改为 previous 指针即可
    return SUCCESS; // 返回成功状态
}
```

显示扑克牌的函数

```
/**
 * @brief           显示双向链表中的扑克牌
 * @param   list    双向链表
 * @param   fromHead 是否从头开始显示
 * @return          返回操作状态
 */
Status DisplayList(DoubleLinkedList list, bool fromHead) {
    确定开始显示的节点
    while (遍历链表)
        输出扑克牌信息
        if (插入到头部) 移动到下一个节点   else 移动到前一个节点
    return SUCCESS; // 返回成功状态
}
```

提取特定花色扑克牌的函数

```
/**
 * @brief           提取特定花色的扑克牌
 * @param   list    双向链表的引用
 * @param   suit    要提取的花色
 * @param   fromHead 是否从头部开始提取
 * @return          返回操作状态
 */
Status ExtractCards(DoubleLinkedList& list, char* suit, bool fromHead) {
    确定开始遍历的节点
    临时节点 tempNode 用于释放内存
    int count = 0; // 记录提取的牌的数量
    while (遍历链表) {
            if (花色匹配) {
```

```
            current->previous->next = current->next; // 调整指针
            current->next->previous = current->previous;
            extractedCards[count++] = current->card; // 保存提取的扑克牌
            tempNode = current; // 保存当前节点用于释放内存
            if (插入到头部) 移动到下一个节点   else 移动到前一个节点
            释放内存
        } else    // 花色不匹配
            if (插入到头部) 移动到下一个节点   else 移动到前一个节点
    }
    用 sort 函数对提取的牌进行排序，并 for (i 从 0 到 count)循环，将提取的牌插入链表
    return SUCCESS; // 返回成功状态
}
```

在 main 函数中，输入命令后，判断命令内容并声明当前扑克牌，输入扑克牌信息后，根据命令内容，调用相关函数进行操作。最后显示链表中的扑克牌。

2.5.5 调试分析（遇到的问题和解决方法）

2.5.5.1 重载运算符

在进行牌面排序时，发现排序结果不符合预期。后来利用重载运算符，重载 < 运算符，确保比较逻辑准确。如果需要降序排序，确保比较逻辑是基于牌面值的正确实现。

2.5.5.2 扑克牌数值存储

在存储扑克牌数值时，最初是想用字符串变量进行存储，但是 10 这样的两位字符 char 型变量无法存储，最终改用 char 型数组表示，并将其转化为 int 型数值，方便后续使用。之所以不开始就存为整型，是为了方便 JQK 的表示以及减少 ASCII 码与整型数值换算的麻烦。

2.5.5.3 STL 库

若考虑使用 STL（标准模板库）来实现扑克牌相关的操作，可以简化一些操作。例如使用 std::map 来将点数与相应的面值进行映射。

```
#include <map>
map<string, int> valueMap = {
    {"A", 1}, {"2", 2}, {"3", 3}, {"4", 4},
    {"5", 5}, {"6", 6}, {"7", 7}, {"8", 8},
    {"9", 9}, {"10", 10}, {"J", 11}, {"Q", 12}, {"K", 13}
};
```

2.5.6 总结和体会

双向链表允许我们在头尾两端高效地进行插入和删除操作，可以利用双向链表实现牌堆的模拟。

数据结构设计

- 节点结构：每个节点包含一张牌以及指向前后节点的指针。
- 双向链表结构：维护一个头指针和一个尾指针，用于快速访问链表的两端。

基本操作

- 插入操作：可以在链表的头部或尾部插入新的牌。通过操作头指针或尾指针来实现，
- 删除操作：同样可以从头部或尾部删除牌。
- 遍历操作：可以从头到尾或从尾到头遍历，输出所有牌的信息。

3. 实验总结

本次实验目的是通过使用线性表的两种类型（顺序表和链表），解决一些实际问题，深入理解线性表的应用场景、复杂度以及不同类型线性表的优点。

数据结构选择：

- 顺序表：适合于需要频繁查找的场景。由于其支持快速的随机访问，查找操作的时间复杂度为 O(1)。
- 链表：适合于频繁插入和删除的场景。尤其是双向链表，能更灵活地进行前后操作，但需注意指针的管理。

注意事项：

- 顺序表：使用动态数组实现顺序表，确保能够根据输入数据动态扩展大小。通过索引直接访问元素，优化了查找操作的效率。
- 链表：在构建双向链表时，尤其注意指针的指向，以确保每次插入和删除都不会造成内存泄漏或访问错误。在删除节点时，确保前后节点的指针正确调整，以维持链表的完整性，并确保每次删除都对应释放相应的节点内存。
- 在选择结构体或数组作为存储结构时，综合考虑操作的频率与复杂度，避免因不当选择导致程序时间复杂度较高。

实验收获：

- 操作选择的重要性：在解决实际问题时，要根据操作的性质选择合适的数据结构。如果操作中涉及大量的插入或删除，链表是更优选择；而若以查找为主，顺序表则更为高效。
- 在日常应用中，例如实现一个任务管理系统，可以选择链表存储待办事项，以便随时增删，而顺序表可以存储已完成的任务。
- 内存管理：在动态内存分配的过程中，学习到如何有效管理内存，避免内存泄漏非常重要。