作业 HW1实验报告

郑智毅 2453374

2025 年 10 月 11 日

1 涉及数据结构和相关背景

线性表是数据结构的一种，一个线性表是 n 个具有相同特性的数据元素的有限序列。

线性表可以根据两个主要维度进行分类：一是基于其逻辑结构上的操作限制，二是基于数据在计算机中的存储结构。如下表格：

分类维度	主要类型	核心特点
逻辑结构	一般线性表	可在任意合法位置进行插入、删除操作。
	受限线性表（如栈、队列）	对数据的插入和删除位置有特定限制。
存储结构	顺序表（顺序存储）	数据元素存储在连续的内存单元中。
	链表（链式存储）	数据元素存储在任意的内存单元中，通过指针连接。

表 1: 线性表的分类

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。我们可以将最后一个元素放到第一个元素，这样就是轮转了一步，这个操作栈可以完成，所以我们需要利用栈。

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：

轮转后的顺序表中的元素。

2.1.3 数据结构设计

```
1 class Stack {
```

```
private:
    std::vector<int> data; //使用 STL 中的 vector（数组实现的一般线性表）
};
```

2.1.4 功能说明

```
void push(const int& value) {
    data.push_back(value); //vector 类似栈的 push 操作，尾插元素
}
void pop() {
    if (!data.empty())
        data.pop_back();  //vector 类似栈的 pop 操作，弹出最后一个元素
}
int& top() {
    return data.back();   //获得最后一个元素的引用（可修改）
}
bool empty() const {
    return data.empty();  //判断是否空栈
}
size_t size() const {
    return data.size();   //获得栈的大小
}
```

2.1.5 调试分析

在错误地弄错了 pop 与 top 的关系，用调试功能发现 top 并不会改变底层的 vector，问题得以解决。

2.1.6 总结和体会

总结：

使用栈进行数组的轮转操作，是一种 in-place 的操作方法（即对数组本身进行操作，不需要进行复制），可以节省空间，而且思维量并不大。

体会：

使用合适的数据结构解决问题，可以简化思维方式，提高程序运行效率，节省时间与空间。

2.2 学生信息管理

2.2.1 问题描述

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

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 {
    string no;
    string name;
};
class SeqList {
private:
    vector<Student> data;
}
```

2.2.4 功能说明

```
//创建学生顺序表
void create(int n) {
    data.clear(); //清除数据，以免有数据残留
    //传入 n，即创建 n 个学生元素
    for (int i = 0; i < n; ++i) {
        Student s;
        cin >> s.no >> s.name;
        data.push_back(s);
```

```
    }
}
//在 pos 处插入，传入 const Student& 以防修改
int insert(int pos, const Student& s) {
    if (pos < 1 || pos > data.size() + 1) return -1;
    data.insert(data.begin() + (pos - 1), s);
    return 0;
}
//移除 pos 处的学生数据
int remove(int pos) {
    if (pos < 1 || pos > data.size()) return -1;
    data.erase(data.begin() + (pos - 1));
    return 0;
}
//check 方法的 name 版本
int check_name(const string& name) {
    for (size_t i = 0; i < data.size(); ++i) {
        if (data[i].name == name) {
            cout << (i + 1) << " " << data[i].no << " " << data[i].name << endl;
            return 0;
        }
    }
    cout << -1 << endl;
    return -1;
}
//check 方法的 no 版本
int check_no(const string& no) {
    for (size_t i = 0; i < data.size(); ++i) {
        if (data[i].no == no) {
            cout << (i + 1) << " " << data[i].no << " " << data[i].name << endl;
            return 0;
        }
    }
    cout << -1 << endl;
    return -1;
}
//返回学生数量
int count() const {
```

```
47      return data.size();
48  }
```

2.2.5 调试分析

2.2.6 总结和体会

总结：

该数据结构成功实现了一个功能完整、结构清晰的学生信息管理顺序表。它引出了对数据结构选择的深入思考：在需要频繁随机访问而插入删除操作较少的场景下，顺序表是高效且实现简便的选择；反之，如果需要频繁在任意位置插入或删除，链式结构（如链表）可能是更合适的方案。

体会：

理解不同数据结构的特性及其代价，是有效运用它们解决实际问题的关键。

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。

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

本题总分 150 分，得到 100 分或以上可视为满分

尚未测试极限数据。

2.3.2 基本要求

输入输出要求：

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。

2.3.3 数据结构设计

```
1  typedef pair<int, int> Term; // 项：(系数, 指数)
2  typedef vector<Term> Polynomial;
```

2.3.4 功能说明

```
// 多项式加法
Polynomial addPolynomials(const Polynomial& poly1, const Polynomial& poly2) {
    Polynomial result;
    int i = 0, j = 0;
    int size1 = poly1.size(), size2 = poly2.size();

    while (i < size1 && j < size2) {
        int exp1 = poly1[i].second;
        int exp2 = poly2[j].second;

        if (exp1 == exp2) {
            // 指数相同，系数相加
            int coef_sum = poly1[i].first + poly2[j].first;
            if (coef_sum != 0) {
                result.push_back(make_pair(coef_sum, exp1));
            }
            i++;
            j++;
        }
        else if (exp1 < exp2) {
            // poly1 的指数较小
            result.push_back(poly1[i]);
            i++;
        }
        else {
            // poly2 的指数较小
            result.push_back(poly2[j]);
            j++;
        }
    }

    // 处理剩余项
    while (i < size1) {
        result.push_back(poly1[i]);
        i++;
    }
```

```
    while (j < size2) {
        result.push_back(poly2[j]);
        j++;
    }

    return result;
}

// 多项式乘法
Polynomial multiplyPolynomials(const Polynomial& poly1, const Polynomial& poly2) {
    map<int, int> exp_map; // 指数-> 系数映射，自动按指数排序

    // 遍历所有项对
    for (const Term& term1 : poly1) {
        for (const Term& term2 : poly2) {
            int coef_product = term1.first * term2.first;
            int exp_sum = term1.second + term2.second;
            exp_map[exp_sum] += coef_product;
        }
    }

    // 构建结果多项式，过滤零系数项
    Polynomial result;
    for (const auto& entry : exp_map) {
        if (entry.second != 0) {
            result.push_back(make_pair(entry.second, entry.first));
        }
    }

    return result;
}

// 输出多项式
void printPolynomial(const Polynomial& poly) {
    if (poly.empty()) return; // 空多项式不输出

    for (size_t i = 0; i < poly.size(); i++) {
        if (i > 0) cout << " ";
```

```
75          cout << poly[i].first << " " << poly[i].second;
76      }
77      cout << endl;
78  }
```

2.3.5 调试分析

2.3.6 总结和体会

总结:

该数据结构是顺序表应用于多项式运算的一个典型范例。它清晰地展示了顺序表在实现诸如多项式加法（归并算法）这类需要顺序遍历和高效访问的场景下的优势。同时，在乘法实现中引入 map 也提示我们，在实际问题中可以灵活地将顺序表与其他数据结构结合，以平衡效率与实现的复杂性。

2.4 求级数

2.4.1 问题描述

求级数

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)

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 $A + 2A^2 + 3A^3 + ... + NA^N$ 的整数值

2.4.3 数据结构设计

```
1  typedef vector<int> BigNum;
```

2.4.4 功能说明

```
1  // 打印大数函数
2  void printBigNum(const BigNum& num) {
3      // 检查数字是否为空，为空则输出 0
4      if (num.empty()) {
5          cout << 0;
6          return;
7      }
```

```
    // 从最高位到最低位依次输出数字 [7](@ref)
    for (int i = num.size() - 1; i >= 0; i--) {
        cout << num[i];
    }
}

// 大数加法函数
BigNum addBigNum(const BigNum& a, const BigNum& b) {
    BigNum result; // 创建结果对象
    int carry = 0; // 进位标志，初始为 0
    int i = 0;
    // 循环直到处理完所有位数且无进位 [3,7](@ref)
    while (i < a.size() || i < b.size() || carry) {
        int sum = carry; // 当前位总和初始化为进位值
        // 加上 a 的当前位（如果存在）
        if (i < a.size()) sum += a[i];
        // 加上 b 的当前位（如果存在）
        if (i < b.size()) sum += b[i];
        carry = sum / 10; // 计算新的进位值
        result.push_back(sum % 10); // 将当前位的结果加入结果中
        i++;
    }
    return result;
}

// 大数乘以整数函数
BigNum multiplyBigNum(const BigNum& a, int b) {
    // 如果乘数为 0，直接返回结果为 0[8](@ref)
    if (b == 0) {
        return BigNum(1, 0);
    }
    BigNum result; // 创建结果对象
    int carry = 0; // 进位标志，初始为 0
    // 遍历大数的每一位或直到进位处理完毕 [3](@ref)
    for (int i = 0; i < a.size() || carry; i++) {
        int product = carry; // 当前乘积初始化为进位值
        // 加上当前位与整数的乘积（如果该位存在）
        if (i < a.size()) product += a[i] * b;
```

```
46          carry = product / 10; // 计算新的进位值
47          result.push_back(product % 10); // 将当前位的结果加入结果中
48      }
49      // 去除结果中高位的多余 0（保持数字的规范表示）[3](@ref)
50      while (result.size() > 1 && result.back() == 0) {
51          result.pop_back();
52      }
53      return result;
54  }
```

2.4.5 调试分析

2.4.6 总结和体会

总结:

该数据结构实现了一个简洁而高效的大数运算模块，核心思路是使用数组按位存储大数（低位在数组前端，高位在后端），并模拟手工计算过程处理进位与借位。整体结构清晰，涵盖了输出、加法和乘法（大数与整数相乘）三种基本操作，对于理解大数运算的基本原理非常有帮助。

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。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

从右上方下载 p98.py 并运行以生成随机测试数据

2.5.2 基本要求

输入输出要求：

2.5.3 数据结构设计

```
#include <list>
#include <string>
struct Card {
    string suit;
    string rank;
};
list<Card> deck;  // 牌堆：链表前端为顶部，后端为底部
```

2.5.4 功能说明

```
// 将牌面值转换为数字以便排序
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);
}
for (int i = 0; i < n; i++) {
        string command;
        cin >> command;

        if (command == "Pop") {
            if (deck.empty()) {
                cout << "NULL" << endl;
            }
            else {
                Card card = deck.front();
                deck.pop_front();
                cout << card.suit << " " << card.rank << endl;
            }
        }
        else if (command == "Append") {
```

```
            string suit, rank;
            cin >> suit >> rank;
            deck.push_back({ suit, rank });
        }
        else if (command == "Revert") {
            deck.reverse();
        }
        else if (command == "Extract") {
            string suit;
            cin >> suit;
            list<Card> extracted;

            // 遍历牌堆，提取指定花色的牌
            auto it = deck.begin();
            while (it != deck.end()) {
                if (it->suit == suit) {
                    extracted.push_back(*it);
                    it = deck.erase(it);  // 从原牌堆移除
                }
                else {
                    it++;
                }
            }

            // 按数字值升序排序提取的牌
            extracted.sort([](const Card& a, const Card& b) {
                return rankToValue(a.rank) < rankToValue(b.rank);
                });

            // 将排序后的牌按降序插入牌堆顶部（确保最小牌在最上面）
            for (auto rit = extracted.rbegin(); rit != extracted.rend(); rit++) {
                deck.push_front(*rit);
            }
        }
    }

    // 输出最终牌堆状态
    if (deck.empty()) {
```

```
62          cout << "NULL" << endl;
63      }
64      else {
65          for (const auto& card : deck) {
66              cout << card.suit << " " << card.rank << endl;
67          }
68      }
```

2.5.5 调试分析

2.5.6 总结和体会

总结：

程序的核心是使用 C++ 标准库中 list 容器来管理一副扑克牌。它将牌堆抽象为一个链表，其中链表前端代表牌堆顶部，后端代表牌堆底部。这种设计使顶部（front）和底部（back）的操作非常直观。

扑克牌本身通过 Card 结构体进行建模，包含 suit（花色）和 rank（点数）两个字符串成员。这种面向对象的数据封装方式，使代码逻辑清晰，易于理解和管理。

3 实验总结

本次实验通过五个编程实践，系统探究了线性表这一基础数据结构在不同场景下的应用与实现。实验内容覆盖了顺序表、链表、栈等线性结构的典型操作，并涉及算法效率、数据封装和实际问题的建模方法。以下从实验内容回顾、数据结构应用分析、问题与解决方案、总体收获四个方面进行总结。

3.1 实验内容回顾

实验包含五个核心部分：轮转数组问题利用栈的 LIFO 特性实现数组元素的高效轮转；学生信息管理系统基于顺序表实现数据的增删查改，体现了线性表的随机访问优势；一元多项式运算通过有序顺序表结合归并算法和映射结构，实现了加法和乘法操作；级数求和问题引入大数运算模型，通过数组模拟手工计算过程处理大整数问题；扑克牌游戏则采用链表结构动态管理牌堆，支持频繁的插入、删除和排序操作。这些案例共同展示了线性结构在逻辑建模和存储设计中的灵活性。

3.2 数据结构应用分析

实验中最显著的特点是针对不同操作需求选择适配的存储结构。例如，学生信息管理系统中，由于需要频繁按位置查询和批量处理，顺序表通过连续内存存储实现了 O(1) 时间的随机访问，但其插入删除操作需移动大量元素，时间复杂度为 O(n) 。相反，扑克牌游戏需要动态调整牌序，链表的离散存储特性使其在插入、删除和反转操作中仅需 O(1) 时间（需配合迭代器定位），但牺牲了随机访问效率。此外，多项式乘法中引入的 map 结构，以及级数问题中的大数数组，均体现了在核心线性结构基础上混合其他数据结构以优化复杂度的设计思想。

3.3 问题与解决方案

实验过程中的主要挑战在于对数据结构底层行为理解的准确性。例如轮转数组问题中，初版代码误将栈的 top 操作视为修改操作，通过调试发现其仅返回引用而非执行弹出，从而修正了元素处理逻辑。在扑克牌游戏的“Extract”指令实现中，需注意链表遍历时迭代器的失效问题：当使用 erase 删除元素后，迭代器会失效，需通过返回值更新其指向。这些问题的解决强化了对数据封装和内存管理的认识。同时，大数运算中进位处理的边界条件（如最高位进位）需通过循环检测保障正确性，体现了算法鲁棒性的重要性。

3.4 总体收获

本实验验证了数据结构选择对程序性能的关键影响。顺序表适合读多写少的场景，而链表在频繁修改的动态数据中表现更优。通过实现多项式运算和扑克牌游戏等综合案例，进一步掌握了如何将抽象问题转化为数据模型（如用 (系数, 指数) 对表示多项式项），并通过模块化设计提高代码可维护性。这些实践不仅深化了对线性表理论的理解，更培养了根据实际需求权衡时空效率、设计高效算法的能力，为后续学习树、图等复杂数据结构奠定了基础。