一、实验目的

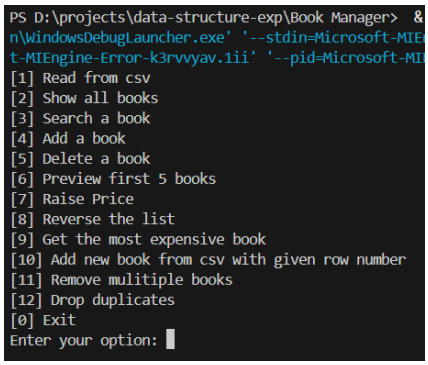
1．掌握线性表的顺序存储表示和链式存储表示。
2．掌握顺序表和链表的基本操作，包括创建、查找、插入和删除等算法。
3. 明确线性表两种不同存储结构的特点及其适用场合，明确它们各自的优缺点。

二、实验设备与环境

实验设备：笔记本电脑
开发环境：Windows 11(64 bit), VS Code + MingGW (g++ 13.1.0)
编译标准：C++17 标准

三、实验内容与结果

由于实验内容很长，将此部分放置在下一页开始至文档结尾。
最终的实现界面：



由于此报告不涉及查找的内容，所以[2]选项暂时没有实现。

四、实验总结及体会

本次实验主要遇到的问题包括以下几点：
1. C++语言本身的一些用法问题；
2. 整个项目类构造的层次问题；
3. 一些细节：例如读入数据读入了空串，输出保留小数，链表修改时一些容易忘记的内容；

主要解决方法：
1. 利用 VSCode 插件的调试功能，在 debug 时观察变量的变化，利用条件断点和普通断点定位出问题的位置；
2. 将报错信息发送给诸如 Claude 的 LLM，向 AI 寻求帮助；
3. 上网搜索一些 C++用法的实现；

实验过程

本部分是实验过程的一个详细说明, 由于原始代码很长, 所以文章中只会截取部分代码, 具体代码在压缩包内, 或者请在 Github仓库 (https://github.com/Jerry050512/Library-Book-Manager) 中查看(可能会有更新).

实现思路

在前半个学期里, 我已经实现过了 `SequentialList` 和 `SinglyLinkedList` 两个类来实现顺序表和线性表, 所以这里只需要基于这两个类构建子类 `BookList` 即可.

并且由于顺序表和链表我们实现的接口名称与参数位置基本都是一致的, 所以两种 `BookList` 类的代码实现基本是相同的, 因此下述说明顺序不会分开来, 而是合并在一起讲, 只有两者不同的实现才会分开.

代码层次说明

两种数据结构的实现放置在不同文件夹下, 分别是 seqlist-ver(顺序表), linklist-ver(链表).

每个文件夹中,

`data_gen.py` 是一个随机生成csv数据的脚本, 用于生成测试数据.

`linklist.cpp` `seqlist.cpp` 分别是顺序表和链表的实现代码.

`book.cpp` 存放书本列表的代码.

`manager.cpp` 是主函数, 用于测试.

具体过程

回顾一下我们两个类共同都实现的一些内容:

```
template <typename T>
class List
{
public:
    // 构造与销毁
    List();
    ~List();

    // 迭代器 - 用于实现 C++ 中的 for-each 语法
    class iterator;
    iterator begin();
    iterator end();

    // 插入, 删除, 获取, 清空, 反转
    bool insert(int index, T element);
    bool remove(int index);
    T get(int index);
```

```
    void clear();
    void reverse();

    // 判断是否包含某个元素
    bool has(T element);
    // 获取长度
    int getLength();

    // 显示
    void display();
}
```

对于 `BookList` 的构建，我们先定义 `Book` 类:

```
#include <string>

typedef struct
{
    string name;
    string author;
    string isbn;
    double price;
} Book;
```

接着再基于线性表构造子类:

```
class BookList : public SinglyLinkedList<Book> {}
```

接下来阐述每一部分任务的实现:

创建与输出

从 csv 读入数据

对应数据在 `book_data.csv` 中

初始化使用默认的构造函数就行, 不过由于基于C-array实现的顺序表需要指定大小, 在测试环境下, 我默认给其设定了100的最大长度.

首先都构造一个 `addBook()` 的方法以应对链表和顺序表尾插接口的差异(实际上, 其实我应该考虑直接把两者都命名为 `append()` 就可以解决问题)

这里读入csv文件使用了C++的流写法, 需要包含头文件 `<fstream>` 和 `<sstream>`.

```cpp
void readFromCSV(const string &filename)
{
    // 打开文件流
    ifstream file(filename);
    if (!file.is_open())
    {
        cout << "Could not open the file: " << filename << endl;
        return;
    }

    // 读取文件
    string line, word;
    // 跳过表头
    getline(file, line);
    while (getline(file, line))
    {
        // 转换成字符串流
        stringstream s(line);
        // TO-DO Check Book Validation
        Book book;
        getline(s, book.isbn, ',');
        getline(s, book.name, ',');
        getline(s, book.author, ',');
        s >> book.price;
        // TO-DO 在这个地方要加上书本信息验证以及去重处理
        addBook(book);
    }
    file.close();
    cout << "Finish reading " << filename << endl;
}
```

输出

因为我们没有定义 `Book` 用 `cout` 的输出, 因此我们需要自定义一个显示方法, 我将其命名为 `BookList::preview(int n)`, 功能是展示前 n 本书本的信息.

这里为了实现小数点的输出控制, 需要使用 `<iomanip>` 中的 `setprecision()`.

```cpp
void BookList::preview(int n)
{
    // 处理过大的n值, 注意这个代码没有对过小的 n 进行处理, 输入 0 或负值没有输出.
    if (n > getLength())
    {
        n = getLength();
        cout << "There are only " << getLength() << " books in the list." << endl;
    }
    // 这里体现了两个类的统一性, 由于都定义了迭代器, 从而可以很方便的进行操作.
    auto cur = begin();
    for (int i = 1; i <= n; i++)
    {
```

```
        Book book = *cur++;
        cout << "Book " << i << ":" << endl;
        cout << "\tBook name: " << book.name << endl;
        cout << "\tAuthor: " << book.author << endl;
        cout << "\tISBN: " << book.isbn << endl;
        // 利用 setprecision 控制小数点后位数
        cout << "\tPrice: " << fixed << setprecision(2) << book.price << endl;
        cout << endl;
    }
}
```

代码验证

针对代码的常规情况和边界条件进行检查:

- 输入一个不存在的文件名
- 输入一个过大的 n 值
- 输入一个过小的 n 值
- 输入一个负数 n 值
- 输入的 n 值为 0

图书信息修改

我认为, 这里比较优雅的方式是定义一个 `BookList::apply(func, args)` 方法, 但是笔者能力有限.

因此此处是唯一一个定义在 `manager.cpp` 中的函数:

```
void raise_price(BookList &booklist, double low, double high)
{
    // 展示基础信息
    double avg_price = booklist.getAveragePrice();
    cout << "Average Price: " << avg_price << endl;
    cout << "\nBefore Price Raise: " << endl;
    booklist.preview(5);
    for (auto &book : booklist)
    {
        if (book.price < avg_price)
        {
            book.price *= low;
        }
        else
        {
            book.price *= high;
        }
    }
    // 展示更改
    cout << "\nAfter Price Raise: " << endl;
    booklist.preview(5);
}
```

这里又体现了之前设置迭代器的优势, 代码中有一个 `BookList::getAveragePrice()` 方法, 用于计算平均价格, 实现逻辑非常简单这里就不贴出来了.

逆序存储

这是定义在链表和顺序表的方法, 两者的实现区别还是蛮大的, 因此这里分开讲.

两者的时间复杂度都是 $O(n)$, 不过实际上, 顺序表只要遍历一半的元素, 而链表需要遍历全部的元素.

顺序表

只需要将前半部分的元素与后半部分互换就行.

```
void reverse()
{
    for (int i = 0; i < length / 2; ++i)
    {
        T temp = list[i];
        list[i] = list[length - i - 1];
        list[length - i - 1] = temp;
    }
}
```

链表

链表的反转相对复杂, 尤其是头节点因为没有存储数据在更改tail指针时需要注意.

```
void reverse()
{
    Node *cur = head->next;
    // 如果cur是null的话, 说明头尾指针都指在头节点, 不能改动尾结点.
    // 注意因为在链表的尾插用的直接是尾指针, 所以如果不修改尾指针后续添加数据就会出现问题.
    if(cur != nullptr) tail = cur;
    Node *prev = nullptr;
    Node *next = nullptr;
    while (cur != nullptr)
    {
        // 遍历链表并更改
        next = cur->next;
        cur->next = prev;
        prev = cur;
        cur = next;
    }

    // 最后一个节点变成首元节点
    head->next = prev;
}
```

代码验证

针对代码的常规情况和边界条件进行检查:

- 顺序表因为算法很简单, 因此不需要验证边界条件
- 一个正常的链表
- 一个空链表
- 一个只有一个节点的链表

查找最贵的图书

这个任务相对简单, 只需要遍历一遍链表或者顺序表, 找到最贵的图书即可.

不过由于会有相同价格的图书, 因此需要遍历两遍, 分别记录最贵的价格和图书.

不过其实也可以在遍历的时候就记录最贵的图书, 每次遇到更贵的就把之前找到的都清空.

```
BookList getMostExpensiveBooks()
{
    // 查找最贵价格
    double highest_price = 0;
    for (auto i=begin(); i!=end(); i++)
    {
        Book book = *i;
        if (book.price > highest_price)
        {
            highest_price = book.price;
        }
    }

    // 将数据添加到 result
    BookList result;
    for (auto i=begin(); i!=end(); i++)
    {
        Book book = *i;
        if (book.price == highest_price)
        {
            result.addBook(book);
        }
    }
    result.preview(result.getLength());
    return result;
}
```

代码验证

针对代码的常规情况和边界条件进行检查:

- 对于一般情况的检查
- 手动修改出多个相同价格的图书
- 把相同价格的图书放在表头, 表中, 表尾

新书入库

数据放置在 `new_book_data.csv` 中

其实是与`BookList::readFromCSV()`非常类似的, 只是要插入到指定位置, 而插入位置合法性的判断其实在线性表中已经实现了.

```
void addFromCSVwithRowNum(const string &filename)
{
    ifstream file(filename);
    if (!file.is_open())
    {
        cout << "Could not open the file: " << filename << endl;
        return;
    }

    string line, word;
    getline(file, line);
    while (getline(file, line))
    {
        stringstream s(line);
        // TO-DO Check Book Validation
        Book book;
        int index;
        // 注意这个文件是插入位置在第一列, 因此需要先读取
        s >> index;
        getline(s, line, ',');
        getline(s, book.isbn, ',');
        getline(s, book.name, ',');
        getline(s, book.author, ',');
        s >> book.price;
        // 原生实现了插入位置合法性判断
        // 依然确实书本信息合法判断和去重处理
        if(insert(index, book) == false)
        {
            cout << "Could not insert book at " << index << endl;
        }

    }
    file.close();
    cout << "Finish reading " << filename << endl;
}
```

代码验证

针对代码的常规情况和边界条件进行检查:

- 具体设置不同的插入位置
- 插入位置为0
- 插入位置为负数
- 插入位置超过线性表长度

删除指定位置的图书

这里不打算如题目要求的从文件读入, 而是直接在命令行中输入多个要删除的图书位置.

这里要注意的一点就是我们删除是有顺序的, 如果你删除前面的图书会移动后面图书的索引, 因此需要索引从大到小进行删除. 考虑到本实验不要求排序部分的内容, 故直接使用 `<algorithm>` 中的 `sort()` 函数进行排序.

为求简便, 所以这个函数实现的并不是很好, 只用了一个100大小的数组来存储多个索引, 即一次最多删除100本书.

```
void removeIndexesFromInput()
{
    int indexes[100];
    int count = 0;
    string line;
    // 此处会先读入一个空串因为之前有一些输出, 所以这里其实应该把getline移到主函数里调用前.
    getline(cin, line);
    getline(cin, line);
    stringstream s(line);
    while(s >> indexes[count])
    {
        count++;
    }
    // 这里第三个参数定义了降序排序的比较规则, 因为默认是升序的.
    sort(indexes, indexes + count, [](int a, int b){return a > b;});
    for (int i = 0; i < count; i++)
    {
        if(indexes[i] >= getLength() || indexes[i] < 0)
        {
            cout << "Invalid index: " << indexes[i] << endl;
            continue;
        }
        remove(indexes[i]);
    }
}
```

代码验证

针对代码的常规情况和边界条件进行检查:

- 输入一个不存在的索引
- 输入一个过大的索引
- 输入一个过小的索引
- 输入一个负数索引

图书去重

这个任务在链表和顺序表的实现是不同的, 这是因为链表我选择了单向链表导致无法从后向前遍历.

顺序表

思路很简单: i从前往后走, j从后往前比较, 删除(这里又是上面那个删除顺序问题, 只能从后往前删)

```
void dropDuplicates()
{
    // TO-DO Compare different books
    for (auto i=begin(); i!=end(); i++)
    {
        Book book = *i;
        for(auto j=end()-1; j!=i; j--)
        {
            Book book2 = *j;
            if (book.isbn == book2.isbn)
            {
                cout << "Duplicate book found: " << book.isbn << ", " << book.name
<< endl;
                remove(j-begin());
            }
        }
    }
}
```

链表

受限于单向的方向, 这里j只能往前走, 那么可以利用链表的性质删除.

```
void dropDuplicates()
{
    // TO-DO Compare different books
    for (auto i = begin(); i != end(); i++)
    {
        Book book = *i;
        auto prev_j = i;
        for (auto j = i + 1; j != end(); j++)
        {
            Book book2 = *j;
            if (book.isbn == book2.isbn)
            {
                cout << "Duplicate book found: " << book.isbn << ", " << book.name
<< endl;
                prev_j->next = j->next;
                delete &j;

                // 这里的回退很重要，因为接下来会调用j++
                j = prev_j;
                lengthDecrease();
            }
            prev_j = j;
        }
    }
}
```

很尴尬的一点就是不得不为了这种比较原生的删除而写一个 `SinglyLinedList::lengthDecrease()` 方法来满足需求.

代码验证

针对代码的常规情况和边界条件进行检查:

- 把重复的图书分别放置在开头, 中间, 结尾.

小结

以上就是一个简易的图书管理系统, 虽然还有很多不足, 但总体比较好的使用上顺序表和链表的不同特点, 也充分利用C++面向对象的特点.