

项目说明文档

数据结构课程设计

——两个有序链表序列的交集

作者姓名：_____张翔_____

学 号：_____2352985_____

指导教师：_____张颖_____

学院、专业：_____计算机科学与技术学院 软件工程_____

同济大学

Tongji University

二〇二四年十二月七日

1 项目分析

1.1 项目背景分析

在计算机科学和实际应用中，链表是一种基础的数据结构，广泛用于存储和操作动态数据。在数据处理中，找到两个有序集合（链表）的交集是一个常见问题。例如：

- (1) 数据库查询中，需要找出满足多个条件的数据（类似集合交集操作）。
- (2) 集合运算中，交集操作可用于分析重叠数据。

在本项目中，两个输入链表和都是按非降序排列的（即从小到大排序），可以利用这一特性设计高效算法，避免冗余计算。

1.2 项目需求分析

本项目需要接受两个非降序排列的链表 S1 和 S2 作为输入，并构造一个包含它们交集的新链表 S3，保持非降序排列。链表节点的数据类型为整数，且输入链表中的元素互不重复。当 S1 和 S2 无交集时，输出为空链表。程序应具有高效性，利用输入链表的有序性减少不必要的计算，处理任意链表为空的边界情况，并保证输出结果的准确性和清晰性。

1.3 项目功能分析

程序分为三个主要功能模块：输入模块用于构造链表 S1 和 S2，支持用户手动输入或从文件加载数据；交集计算模块通过逐一比较 S1 和 S2 的节点值，生成结果链表 S3，确保操作高效且保持非降序排列；输出模块用于显示交集链表的所有元素，若交集为空则提供提示信息。

2 项目设计

2.1 数据结构设计

基于项目分析，本项目选择链表作为数据结构而不是数组，主要原因如下：

- (1) 动态大小需求：链表可以动态地分配内存；
- (2) 内存分配灵活性：链表的内存分配比较灵活，可以根据需求动态分配，而数组需要一次性分配固定大小的内存空间；
- (3) 避免数组扩展的开销：使用数组可能需要额外的扩展和拷贝操作，而链表避免了这种开销；

2.2 MyList 类的设计

2.2.1 概述

MyList 是一个基于模板的链表类，用于实现通用的链表操作。它通过 `MyLinkNode<Type>* head` 维护链表的头节点，支持多种功能，包括动态创建链表（可通过指定终止标志或直接传入头节点进行构造）、遍历打印链表内容，以及获取链表头节点指针等。默认构造函数初始化为空链表，析构函数负责释放内存以防止泄漏。通过模板参数 `Type`，MyList 可存储任意类型的数据，适用于动态数据存储及操作的场景，具有良好的扩展性和适配性。

2.2.2 类定义

```
template <typename Type>
class MyList
{
private:
    MyLinkNode<Type>* head;
public:
    MyList() : head(nullptr) {}
    ~MyList();
    void createList(Type& end);
    void createList(MyLinkNode<Type>* _head);
    void printList();
    MyLinkNode<Type>* getHead();
};
```

2.3 项目主体架构设计



2.3.1 项目主体架构流程图

3 项目功能实现

3.1 输入链表功能的实现

3.1.1 输入链表功能实现思路

这段代码实现了链表的动态创建功能，允许用户输入一组数据以构建链表，并通过指定的结束标志（end）结束输入，具体实现思路如下：

函数首先定义一个临时变量 num 用于接收用户输入，并初始化指针 current 指向链表头节点 head。在循环中，每次从标准输入读取数据，并检查是否等于结束标志 end，如果是则终止输入并清空输入流。对于每个有效数据，动态分配内存创建一个新节点，并根据链表是否为空执行不同的操作：若链表为空，新节点成为链表头节点；否则，将当前节点的 link 指针指向新节点，并移动

current 指针到新节点。若内存分配失败，则输出错误信息并中止操作。该方法确保了链表的动态扩展，同时处理了内存分配失败的情况以提高代码的可靠性。

3.1.2 输入链表功能核心代码

```
template <typename Type>
void MyList<Type>::createList(Type& end)
{
    Type num = end;
    MyLinkNode<Type>* current = head;
    while (std::cin >> num) {
        if (num == end) {
            std::cin.ignore(INT_MAX, '\n');
            break;
        }
        MyLinkNode<Type>* newNode = new(std::nothrow) MyLinkNode<Type>(num);
        if (newNode != nullptr) {
            if (head == nullptr) {
                head = newNode;
                current = head;
            }
            else {
                current->link = newNode;
                current = newNode;
            }
        }
        else {
            std::cerr << "Memory allocation failed for MyLinkNode." << std::endl;
            break;
        }
    }
}
```

3.2 查找链表交集功能的实现

3.2.1 查找链表交集功能实现思路

这段代码实现了两个有序链表的交集计算功能，其核心思想是利用链表的有序性，通过双指针逐步遍历两个链表并比较节点值，具体思路如下：

初始化两个指针 `p1` 和 `p2` 分别指向链表 `l1` 和 `l2` 的头节点，并用 `dummy` 和 `result` 维护交集链表的头节点和尾节点。在遍历过程中，如果 `p1` 和 `p2` 的数据相等，则创建一个新节点加入交集链表，并同时移动 `p1` 和 `p2` 指针；如果 `p1` 的数据小于 `p2`，则移动 `p1`；否则移动 `p2`，从而保证比较的有效性和效率。遍历结束后返回交集链表的头节点。代码通过检查内存分配结果来确保程序的健壮性，处理了节点数据重复及链表长度不同的情况。

3.2.2 查找链表交集功能核心代码

```
template <typename Type>
MyLinkNode<Type>* findIntersection(MyLinkNode<Type>* l1, MyLinkNode<Type>* l2)
{
    MyLinkNode<Type>* p1 = l1;
    MyLinkNode<Type>* p2 = l2;
    MyLinkNode<Type>* dummy = nullptr;
    MyLinkNode<Type>* result = nullptr;
    while (p1 && p2) {
        if (p1->data == p2->data) {
            MyLinkNode<Type>* newNode = new(std::nothrow) MyLinkNode<Type>(p1->data);
            if (newNode != nullptr) {
                if (result == nullptr) {
                    result = newNode;
                    dummy = result;
                }
                else {
                    dummy->link = newNode;
                    dummy = newNode;
                }
            }
            p1 = p1->link;
        }
        else if (p1->data < p2->data) {
            p1 = p1->link;
        }
        else {
            p2 = p2->link;
        }
    }
    return result;
}
```

```

        p2 = p2->link;
    }
    else {
        std::cerr << "Memory allocation failed for MyLinkNode." << std::endl;
    }
}

else if (p1->data < p2->data) {
    p1 = p1->link;
}

else if (p1->data > p2->data) {
    p2 = p2->link;
}

}

return result;
}

```

3.3 异常处理功能的实现

3.3.1 动态内存申请失败的异常处理

在进行动态内存申请时，程序使用 `new(std::nothrow)` 来尝试分配内存。`new(std::nothrow)` 在分配内存失败时不会引发异常，而是返回一个空指针（`nullptr`），代码检查指针是否为空指针，如果为空指针，意味着内存分配失败，这时程序将执行以下操作：

- (1)向标准错误流 `std::cerr` 输出一条错误消息 "Error: Memory allocation failed.";
- (2)调用 `exit` 函数，返回错误码-1，用于指示内存分配错误，并导致程序退出。

3.3.2 输入非法的异常处理

程序通过调用 `inputInteger` 函数输入电网节点个数和电网节点之间的距离。`inputInteger` 函数用于获取用户输入的整数，同时限制输入必须在指定的范围内，函数的代码如下：

```

int inputInteger(int lowerLimit, int upperLimit, const char* prompt)
{
    std::cout << ">>> " << "请输入" << prompt << " 整数范围: [" << lowerLimit << "~"
<< upperLimit << "]: ";
    int input;

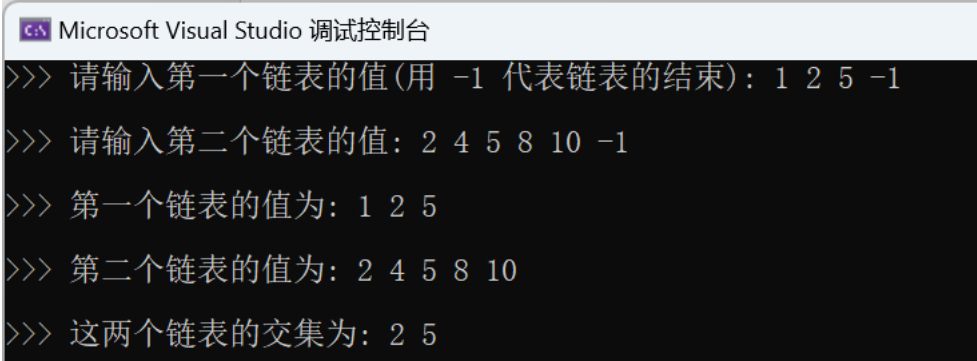
```

```

while (true) {
    std::cin >> input;
    if (std::cin.good() && input >= lowerLimit && input <= upperLimit) {
        std::cin.clear();
        std::cin.ignore(INT_MAX, '\n');
        std::cout << std::endl;
        return input;
    }
    else {
        std::cerr << ">>> " << prompt << "输入不合法，请重新输入!" << std::endl;
        std::cin.clear();
        std::cin.ignore(INT_MAX, '\n');
    }
}
}

```

4 项目测试



```

Microsoft Visual Studio 调试控制台
>>> 请输入第一个链表的值(用 -1 代表链表的结束): 1 2 5 -1
>>> 请输入第二个链表的值: 2 4 5 8 10 -1
>>> 第一个链表的值为: 1 2 5
>>> 第二个链表的值为: 2 4 5 8 10
>>> 这两个链表的交集为: 2 5

```

4.1 一般情况测试示例


```
Microsoft Visual Studio 调试控制台
>>> 请输入第一个链表的值(用 -1 代表链表的结束): 1 3 5 -1
>>> 请输入第二个链表的值: 2 4 6 8 10 -1
>>> 第一个链表的值为: 1 3 5
>>> 第二个链表的值为: 2 4 6 8 10
>>> 这两个链表的交集为: NULL
```

4.2 交集为空的情况测试示例

```
Microsoft Visual Studio 调试控制台
>>> 请输入第一个链表的值(用 -1 代表链表的结束): 1 2 3 4 5 -1
>>> 请输入第二个链表的值: 1 2 3 4 5 -1
>>> 第一个链表的值为: 1 2 3 4 5
>>> 第二个链表的值为: 1 2 3 4 5
>>> 这两个链表的交集为: 1 2 3 4 5
```

4.3 完全相交的情况测试示例

```
Microsoft Visual Studio 调试控制台
>>> 请输入第一个链表的值(用 -1 代表链表的结束): 3 5 7 -1
>>> 请输入第二个链表的值: 2 3 4 5 6 7 8 -1
>>> 第一个链表的值为: 3 5 7
>>> 第二个链表的值为: 2 3 4 5 6 7 8
>>> 这两个链表的交集为: 3 5 7
```

4.4 其中一个序列完全属于交集的情况测试示例

5 集成开发环境与编译运行环境

Windows 系统: Windows 11 x64

Windows 集成开发环境: Microsoft Visual Studio 2022 (Release 模式)

Windows 编译运行环境: 本项目适用于 x86 架构和 x64 架构