# 1 需求

## 1.1 项目背景

在生活中时常需要比对数据，并筛选出两份数据中相似的部分。比如对两个生物样本做基因检测，查找它们公有的基因序列；再比如神经网络分类模型中将样本与原型进行比对，判断二者的相似程度，从而精确进行聚类。本项目旨在模拟一个简单的数据比较系统，提取样本的公共属性。为了简化模型，本项目做出了几点假设：1、所比较的对象有且仅有两个。2、所比较的对象均为由若干个正整数构成的非降序序列。3、比较后得出的结果为二者的交集序列。最后的结果按标准格式化输出。

## 1.2 功能分析

用户输入两个非降序的链表序列S1和S2，程序求出S1和S2的交集S3并打印出来。当用户输入不符合要求时，程序给出提示信息并让用户重新输入。

输入要求：

1. 输入共2行，在第i行给出由若干个正整数构成的非降序序列Si（i = 1，2）。
2. ② 用-1表示序列的结尾（-1不属于这个序列）。
3. ② 数字用空格间隔。

输出要求：

1. 在一行中输出两个输入序列的交集序列。
2. 数字间用空格分开，结尾不能有多余空格。
3. 若新链表为空，输出NULL。

# 2 设计

## 2.1 数据结构设计

### 2.1.1 逻辑结构设计

本项目的核心功能是完成对两个由正整数构成的非降序链表求交集。由于元素相对独立，各个元素之间不存在明显的偏序关系或层次关系，也不存在“一对多”与“多对一”的映射，所以可以采用线性结构进行存储。

### 2.1.2 存储结构设计

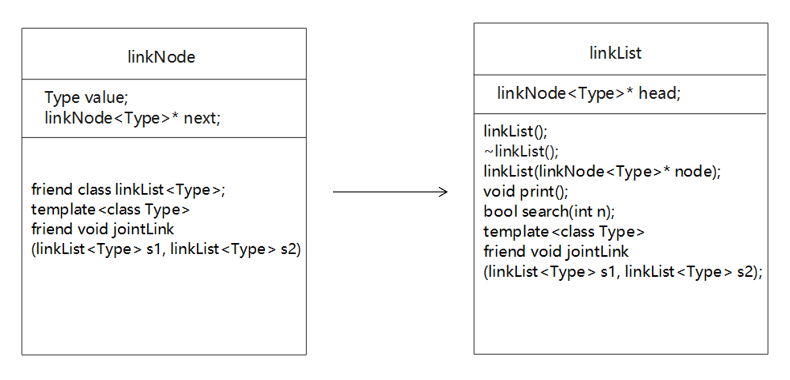
线性表的存储通常通过顺序存储和链式存储方式实现。其中顺序存储在修改操作上的时间复杂度为O(1)，而其他操作均为O(n)，而链式存储在插入、查找、修改等操作上的时间复杂度都是O(1)，综合比较发现在本项目中链式存储的效率明显高于顺序存储，所以本项目采用链式存储方式即链表数据结构存储考生信息。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 顺序存储 | 链式存储 | 比较结果 |
| 插入功能 | O(n) | O(1) | 链式存储更优 |
| 删除功能 | O(n) | O(1) | 链式存储更优 |
| 查找功能 | O(n) | O(n) | 相同 |
| 修改功能 | O(1) | O(1) | 相同 |
| 统计功能 | O(n) | O(n) | 相同 |

为了统一空表与非空表的操作、简化链表操作的实现，本项目在构建链表的时候在链表最前端加入了头节点head，该节点不存储任何考生信息，仅用于访问链表和提升代码的重用性。

## 2.2 类设计

本项目共定义了两个类，分别是链表节点类linkNode与链表类LinkList。为了提高代码的重用性，程序在定义类的时候使用了类模板以适应不同的数据类型。linkNode类与linkList类的示意图如下：



### 2.2.1 链表节点类linkNode

链表节点类的protected属性包含了链表节点的数据域与指针域，其数据域为指向Type类型变量value的指针,其指针域为指向下一个linkNode<Type>元素的指针，从而构建出一条元素为linkNode<Type>类型的链表。链表节点类的public属性主要用于申明友元和存放操作。其具有的成员函数如下：

//链表节点类

**template** <**class** Type> **class** linkNode {

**protected**:

    Type value;    //数据域

    linkNode<Type>\* next;    //指针域

**public**:

**friend** **class** linkList<Type>;

**template**<**class** Type>

**friend** **void** jointLink(linkList<Type>& s1, linkList<Type>& s2);

};

### 2.2.2 链表类linkList

链表类linkList的protected属性包含用于访问链表的头指针，其public属性主要存放必要的操作，包括构造与析构函数、打印链表节点信息等。链表类定义的源代码如下：

//链表类

**template** <**class** Type> **class** linkList {

**protected**:

    linkNode<Type>\* head;    //链表头指针

**public**:

    linkList();    //构造函数

    ~linkList();    //析构函数

    linkList(linkNode<Type>\* node);    //带一个参数的构造函数

**void** print();    //打印链表节点信息

**template**<**class** Type>

**friend** **void** jointLink(linkList<Type> &s1, linkList<Type> &s2);

};

## 2.3 函数设计

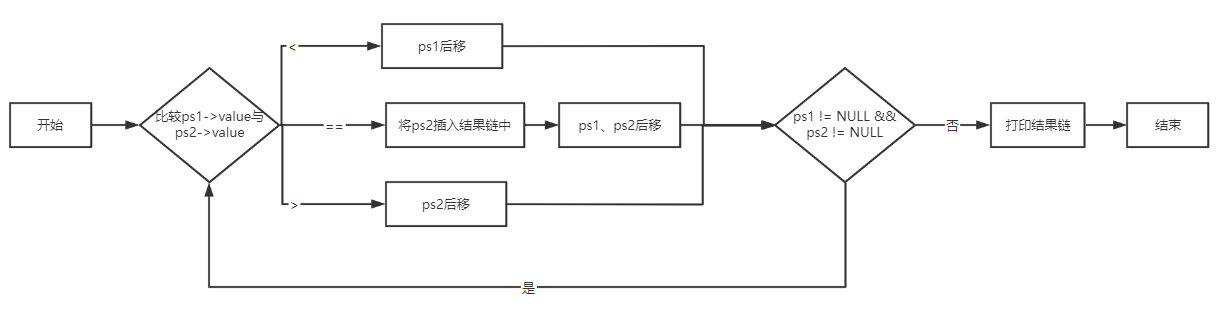
本项目的核心功能是求两个链表的交集，由于其操作对象为两个链表，所以可定义一个类外函数jointLink用于取交集并打印结果。取交集的实现过程需要访问链表节点的

value、next属性以及链表的head属性，而上述数据均为protected类型，故需在linkNode类与linkList类里将jointLink函数声明为友元函数。

**friend** **void** jointLink(linkList<Type> &s1, linkList<Type> &s2);

jointLink函数实现的整体思路与多项式加法相似，先创建两个指针ps1、ps2，分别指向两个链表，然后移动两个指针逐一节点比较。当查找到某个节点同时存在于两个链表中时，将该节点加入结果链，该步骤反复执行直至某一链表为空。最终将结果链的元素依次打印。

jointLink函数的实现流程图如下：



# 3 实现

## 输入功能的实现

### 输入功能实现思路

输入功能封装在链表类的构造函数中，程序循环读取用户输入的数值并赋值给链表节点，直至读取到-1时退出循环，链表创建完毕。由于本项目操作的链表要求为非降序的正整数序列，所以需在输入时进行错误处理。每次循环初始定义一个变量存放前一个节点的数值，并与当前输入比较，若当前输入较大且不为-1，程序进入下一次循环并读取下一节点的数值。若当前输入不为int类型或数值小于前一节点，程序给出错误信息并提示用户重新输入整条序列，当前指针重置为链表头节点。

### 输入功能实现代码

//链表类构造函数

**template**<**class** Type>

linkList<Type>::linkList() {

    head = **new** linkNode<Type>;

**if** (head == NULL) {

        cout << "初始化失败" << endl;

        exit(-1);

    }

    linkNode<Type>\* temp = head;

**int** last = 0;

**while** (1) {

        linkNode<Type>\* node = **new** linkNode<Type>;

**if** (node == NULL) {

            cout << "初始化失败" << endl;

            exit(-1);

        }

**int** m;

        cin >> m;

**if** (m == -1)

**break**;

**if** (cin.fail() || m < 0 || m < last) {

            cout << "输入错误，请重新输入整个序列：";

            cin.clear();

**char** t;

**while** ((t = cin.get()) != '\n');

            temp = head;

            last = 0;

**continue**;

        }

        node->value = m;

        temp->next = node;

        temp = node;

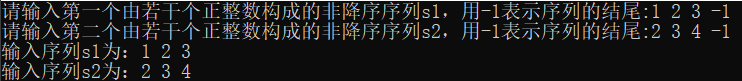
        last = m;

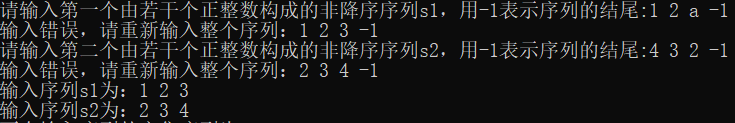
    }

    temp->next = NULL;

}

### 输入功能实现样例





## 求交集功能的实现

### 求交集功能实现思路

求交集函数实现的整体思路与多项式加法相似，先创建两个指针ps1、ps2分别指向两个链表，然后逐一节点比较。若ps1->value == ps2->value，则说明当前节点在两条链表中均存在，其属于两条链表所示集合的交集，故将此节点插入结果链中。若ps1->value > ps2->value，则说明s1的当前节点在s2中不存在，ps1后移继续进行比较。若ps1->value < ps2->value，则说明s2的当前节点在s1中不存在，ps2后移继续进行比较。将如上步骤循环执行直至某一链表为空。

### 求交集功能实现代码

//用于求交集的函数

**template**<**class** Type>

**void** jointLink(linkList<Type>& s1, linkList<Type>& s2) {

    linkNode<Type>\* resultNode = **new** linkNode<Type>;

    linkNode<Type>\* end = resultNode;

**if** (end == NULL) {

        cout << "初始化失败" << endl;

        exit(-1);

    }

    linkNode<Type>\* ps1 = (s1.head)->next;

    linkNode<Type>\* ps2 = (s2.head)->next;

**while** (ps1 != NULL && ps2 != NULL) {

**if** (ps1->value == ps2->value) {

            linkNode<Type>\* node = **new** linkNode<Type>;

            node->value = ps1->value;

            end->next = node;

            end = node;

            ps1 = ps1->next;

            ps2 = ps2->next;

        }

**else** **if** (ps1->value > ps2->value)

            ps2 = ps2->next;

**else**

            ps1 = ps1->next;

    }

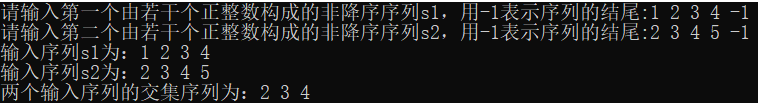
    end->next = NULL;

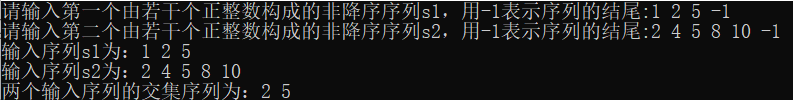
    linkList<**int**> result(resultNode);

    result.print();

}

### 求交集功能实现样例





# 4 测试

## 4.1 一般情况

输入：

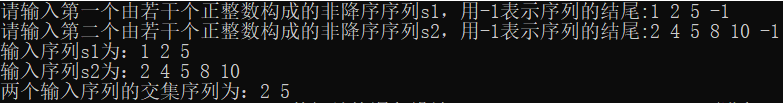
1 2 5 -1

2 4 5 8 10 -1

预期输出：

2 5

实际结果：



## 4.2 交集为空的情况

输入：

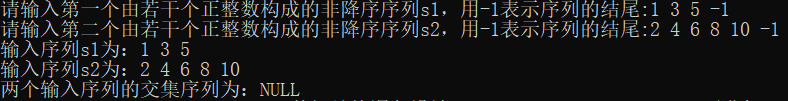
1 3 5 -1

2 4 6 8 10 -1

预期输出：

NULL

实际结果：



## 4.3 完全相交的情况

输入：

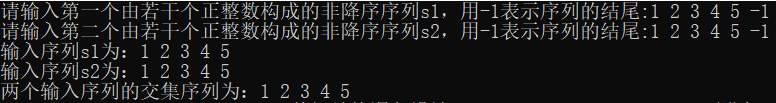
1 2 3 4 5 -1

1 2 3 4 5 -1

预期输出：

1 2 3 4 5

实际结果：



## 4.4 其中一个序列完全属于交集的情况

输入：

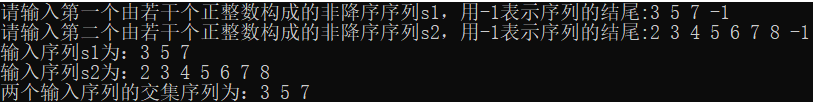
3 5 7 -1

2 3 4 5 6 7 8 -1

预期输出：

3 5 7

实际结果：



## 4.5 其中一个序列为空的情况

输入：

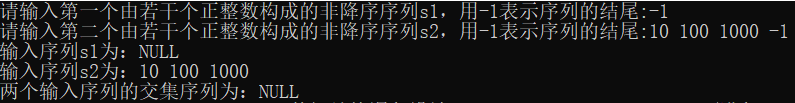
-1

10 100 1000 -1

预期输出：

NULL

实际结果：



## 4.6 错误测试（输入非int型数值）

输入：

1 2 a -1

预期输出：

报错并提示用户重新输入。

实际结果：



## 4.7 错误测试（输入非递增序列）

输入：

3 2 1 -1

预期输出：

报错并提示用户重新输入。

实际结果：



## 4.8 错误测试（输入除-1外的负数）

输入：

1 2 -2 -1

预期输出：

报错并提示用户重新输入。

实际结果：

