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： 杨滕超

学 号： 2151298

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目 录

[1 项目概述 3](#_Toc495668153)

[1.1 项目背景 3](#_Toc495668154)

[1.2 项目功能 3](#_Toc495668155)

[1.3 项目分析 4](#_Toc495668154)

[2 项目设计 4](#_Toc495668156)

[2.1 总体设计 4](#_Toc495668157)

[2.2 类设计 5](#_Toc495668158)

[3 项目实现 6](#_Toc495668161)

[3.1 总体实现 6](#_Toc495668162)

[3.1.1 总体实现流程图 6](#_Toc495668163)

[3.2 输入序列实现 7](#_Toc495668166)

[3.2.1 输入序列实现流程图 7](#_Toc495668167)

[3.2.2 输入序列实现思路](#_Toc495668167) 7

[3.2.3 输入序列实现主要代码](#_Toc495668168) 7

[3.3 排序实现 8](#_Toc495668170)

[3.3.1 排序实现思路 8](#_Toc495668171)

[3.3.2 递归归并排序实现思路](#_Toc495668171) 8

[3.3.3 递归归并排序实现流程图 9](#_Toc495668171)

[3.3.4 递归归并排序实现主要代码 9](#_Toc495668171)

[3.3.5 非递归归并排序实现思路 1](#_Toc495668172)0

[3.3.6 非递归归并排序实现流程图 1](#_Toc495668172)1

[3.3.7 非递归归并排序实现主要代码 1](#_Toc495668172)1

[3.4 求交集实现 13](#_Toc495668174)

[3.4.1 求交集实现流程图 13](#_Toc495668175)

[3.4.2 求交集实现思路 13](#_Toc495668175)

[3.4.3 求交集实现主要代码 13](#_Toc495668176)

[4 项目测试 14](#_Toc495668186)

[4.1 输入测试 14](#_Toc495668187)

[4.1.1 正常输入测试 14](#_Toc495668188)

[4.1.2 非法字符 14](#_Toc495668189)

[4.2 排序及求交集测试](#_Toc495668193) 14

[4.2.1 两条空序列](#_Toc495668194) 14

[4.2.2 一条空序列 1](#_Toc495668195)5

[4.2.3 两条序列部分相交 1](#_Toc495668195)5

[4.2.4 两条序列完全相交 1](#_Toc495668195)5

[4.2.5 非升序输入 1](#_Toc495668195)5

1 项目概述

* 1. 项目背景

链表作为一种十分常用的数据结构，在增删频繁的情况下具有良好的表现。这是一种逻辑上连续，但是物理存储上不连续的数据结构，通过链表结点中的指针进行连接，从而使得其在逻辑上连续。但同时，这样的操作也使得相较于顺序表，链表的存储效率变低；还需要注意的一点是：对与访问修改特定编号的结点一般来说需要O(n)的时间复杂度，因此我们也应该对具体问题情况进行分析，不要盲目选择数据结构。

对于本项目来说，获得两个有序链表序列的交集，最好是遍历完成的同时也获得相交的集合。这时候考验了我们对链表结点指针的处理，何时移动指针称为本项目的关键，对于此，将在后文详细分析。

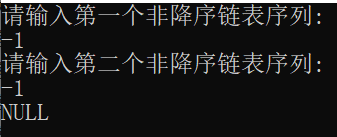
求有序链表的交集看似简单，其中的思想需要好好体会，对指针的移动和处理、合适为nullptr等等，需要我们深思背后的原理。

* 1. 项目功能

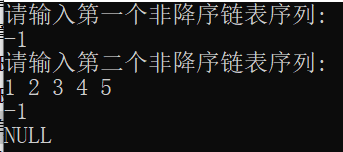
输入两个升序序列，以-1结尾，存入两条链表中，构造第三条链表，存储两个链表重复的元素，最后输出。

示例如下：

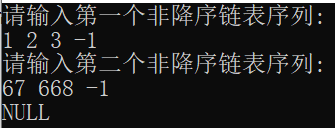
·两条空序列



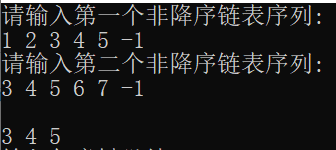
·其中一条空序列



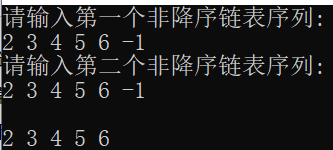
·完全不相交



·部分相交

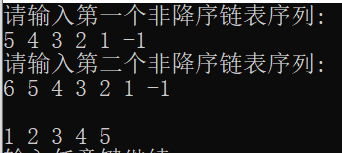


·完全相交



考虑到用户因为种种原因，未能按照升序输入序列，会造成输出结果的不正确，甚至程序的崩溃，因此我们对获得的链表序列进行归并排序，以保证程序的正确运行。

·非升序输入序列示例：



* 1. 项目分析

1. 对于程序效率的考虑，采用双指针，利用排序的特点，实现遍历完两条链表即可以得出目标链表，而非采用两层嵌套循环，否则对于数据量大的序列，表现会变得十分糟糕。
2. 对于用户输入的考虑，链表序列依次输入，其中若出现非法字符，则提示重新输入，出现-1则表示输入完成。具备健壮性。
3. 对于用户输入非升序，为了保证程序顺利且正确运行，我们对链表序列调用List类内部的排序函数，进行排序。
4. 考虑各种输入情况。其中包括：两条序列不等长、为空等情况。

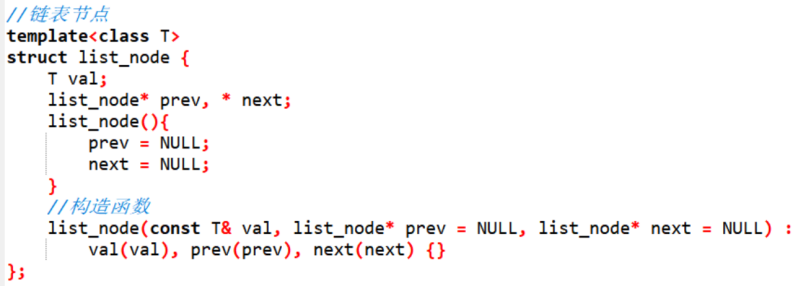
2 项目设计

2.1 总体设计

采用链表储存输入的序列，但是作为计算机中常用的数据结构，链表的种类繁多，例如：单链表、双链表、循环链表等等，考虑到链表List类的完整性和复用性，我们决定使用双链表作为链表的种类。调用链表的排序函数，时间复杂度为O(nlogn)，其中n为链表结点的个数；利用函数对链表的遍历，从而得出目标链表的序列，其中的时间复杂度为O(max(m + N))，其中m，n分别为两条链表的结点个数。

2.2 类设计

·链表结点类设计：其中包括需要储存的数据，前驱节点和后继节点。如下图所示：



·链表类设计：数据成员包括，头结点，尾结点以及记录节点个数数据。

主要功能(与本题有关)包括：

·//返回链表大小inline int size() const;

·//判断链表是否为空inline bool empty() const;

·//清空链表void clear();

·//最后添加元素void emplace\_back(const T& x)；

·//最后删除元素void pop\_back();

·//开头添加元素void emplace\_front(const T& x);

·//开头删除元素void pop\_front();

·//插入void insert(const List<T>::list\_iterator& it, const T& x);

·//删除void remove(const List<T>::list\_iterator& it);

·//返回迭代器的查找typename List<T>::list\_iterator find(const T& x);

·//迭代器begin inline List<T>::list\_iterator begin();

·//迭代器end inline List<T>::list\_iterator end();

**·//递归排序void mergeSort1()**

**·//归并递归函数list\_node<T>\* \_mergeSort(list\_node<T>\* head);**

**·//迭代排序void mergeSort2();**

**·//从from开始切断step结点，并返回最后一个list\_node<T>\* cut(list\_node<T>\* from, int step);**

**·//归并排序的合并list\_node<T>\* merge(list\_node<T>\* l1, list\_node<T>\* l2);**

·游标类设计：数据成员包括指向链表结点的指针

主要功能包括：

·//重载\* T& operator\*();

·//重载-> T\* operator->();

·//重载自加inline List<T>::list\_iterator& operator++();

·//重载自减inline List<T>::list\_iterator& operator--();

·//重载==inline bool operator==(const list\_iterator& it) const;

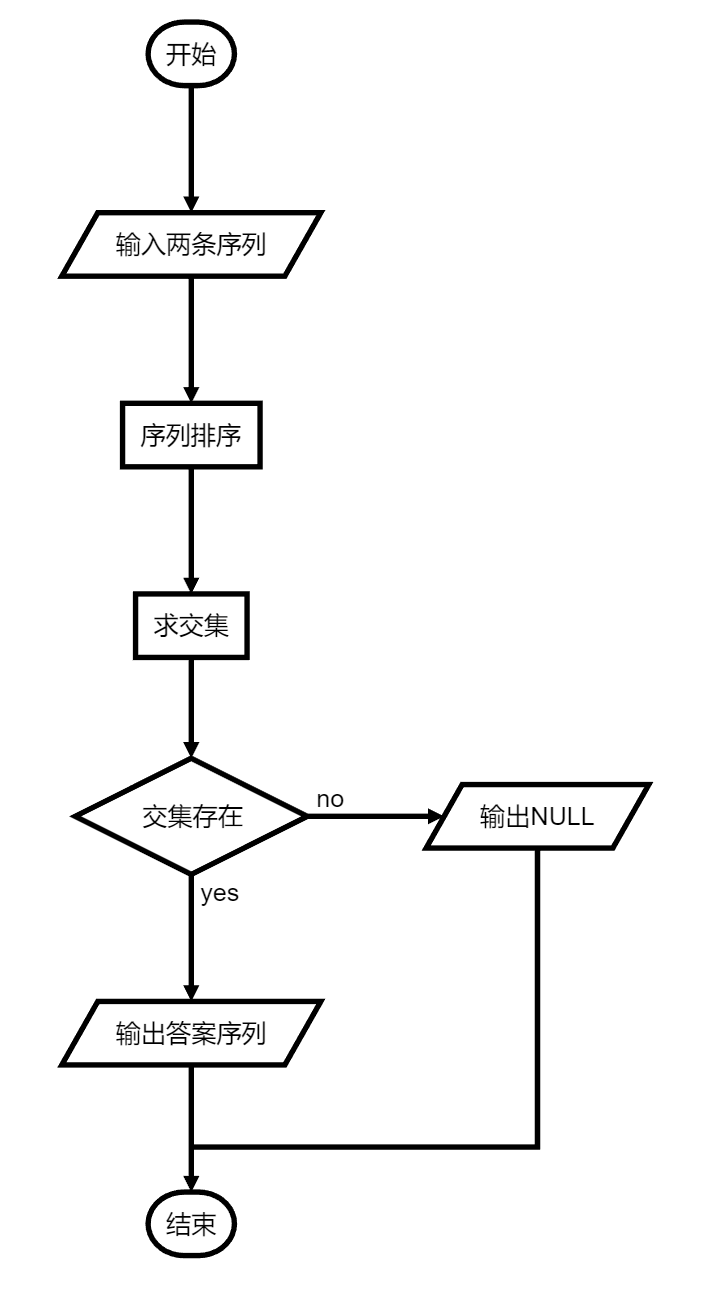
·//重载!=inline bool operator!=(const list\_iterator& it) const;

此处不完全列出所有功能，但基本按照C++迭代器标准实现其功能。

3 项目实现

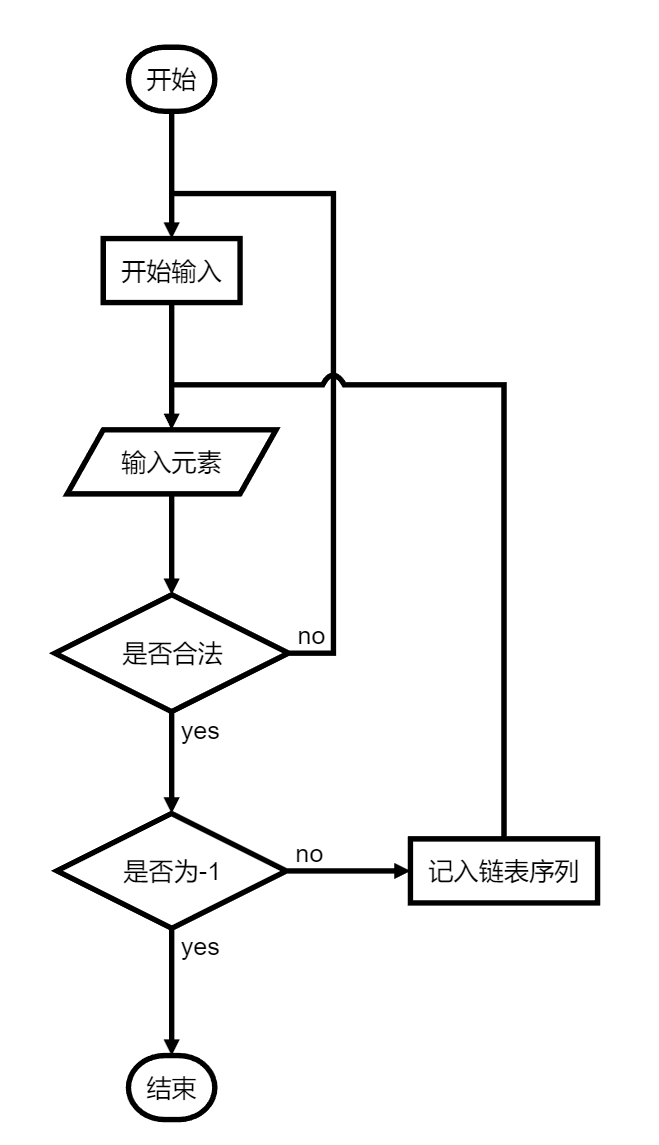
3.1 总体实现

3.1.1 总体实现流程图



3.2 输入序列实现

3.2.1 输入序列实现流程图



3.2.2 输入序列实现思路

输入的数据类型，我们采用int。最外层循环while(1)输入有限个列表元素，遇到负数退出。此时需要判断有效输入，内层循环，对于非法输入，用cin.fail()函数进行检查，输入无误则跳出循环，若为非法输入则输出提示并且重新输入。

其中利用布尔型变量记录是否存在非法输入，原因在于得知存在非法输入需要清空已经存在的链表序列，重新输入

3.2.3 输入序列实现主要代码

List<int> l1, l2, l3;

int num = 0;

bool cinFail = false;

cout << "请输入第一个非降序链表序列: " << endl;

while (1){

l1.clear();

while (1){

cin >> num;

if (cinFail = cin.fail()){

cout << "输入不是整数，请重新输入" << endl;

cin.clear();

cin.ignore(65536, '\n');

break;

}//end of if

if (num == -1) break;

else

l1.emplace\_back(num);

}//end of while

if (!cinFail)

break;

}//end of while

3.3 排序实现

3.3.1 排序实现思路

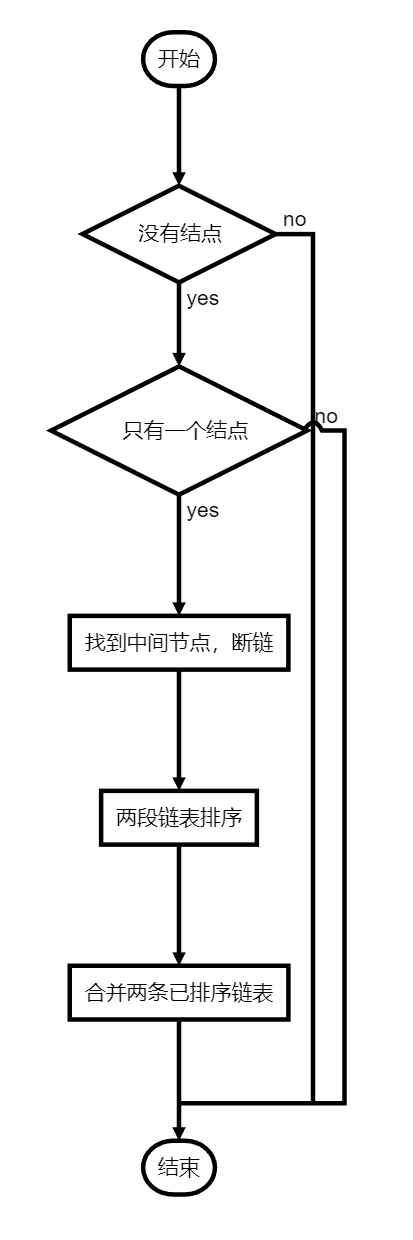
对于链表的排序，采用归并排序思想，最大程度地提高了算法的效率，时间复杂度为O(nlogn)，相较于一般插入排序性能得到了极大提高。其中的思考在于：要求性能较好的排序算法，可以考虑快速排序、堆排序、归并排序，但快速排序的时间复杂度不稳定，堆排序并不适合链表，因此我们采用归并排序。

对于归并排序，其思想有两个重要操作：其一是将链表一分为二，从中间分开，对分别对两条链进行排序；其二再进行合并。分为递归和迭代两种方法实现。虽然两种方法时间复杂度相同，但考虑到递归需要使用递归工作栈，使得程序的性能大大降低，因此我们最后选择使用迭代方法。

3.3.2 递归归并排序实现思路

1. 设置递归结束条件为：没有或只有一个结点
2. 通过快慢指针找到链表物理意义上最中间的结点，并断开连接，得到两条链表
3. 将两条链表作为参数进入递归函数，得到两条排好序的链表
4. 利用两条链表的有序性，设置两个指针，分别指向链表结点，取较小的结点为合成后链表的连接对象，直到其中一条链表为空，退出循环，最后连接没有连接完成的另一条。
5. 上述过程需要注意，利用一个哑结点辅助插入
6. 并且最后链表完成后更新头结点和尾结点

3.3.3 递归归并排序实现流程图



3.3.4 递归归并排序实现主要代码

template<class T>

list\_node<T>\* List<T>::\_mergeSort(list\_node<T>\* head)

{

//没有结点或只有空结点，直接返回

if (head == NULL || head->next == NULL)

return head;

list\_node<T>\* slow = head, \* fast = head, \*prev = NULL;

//快慢指针找中间

while (fast && fast->next)

{

prev = slow;

slow = slow->next;

fast = fast->next->next;

}

//断开中间，分成两端

prev->next = NULL;

list\_node<T>\* p1, \* p2, \* D = new list\_node<T>(0), \* p3 = D;

//交给递归函数排序两条链

p1 = \_mergeSort(head);

p2 = \_mergeSort(slow);

//合并

while (p1 && p2)

{

if (p1->val < p2->val)

{

p3->next = p1;

p1->prev = p3;

p1 = p1->next;

p3 = p3->next;

}

else

{

p3->next = p2;

p2->prev = p3;

p2 = p2->next;

p3 = p3->next;

}

}

//剩下的一条链

while (p1)

{

p3->next = p1;

p1->prev = p3;

p1 = p1->next;

p3 = p3->next;

}

while (p2)

{

p3->next = p2;

p2->prev = p3;

p2 = p2->next;

p3 = p3->next;

}

p3->next = NULL;//收尾工作

list\_node<T>\* ans = D->next;

//释放空间

delete D;

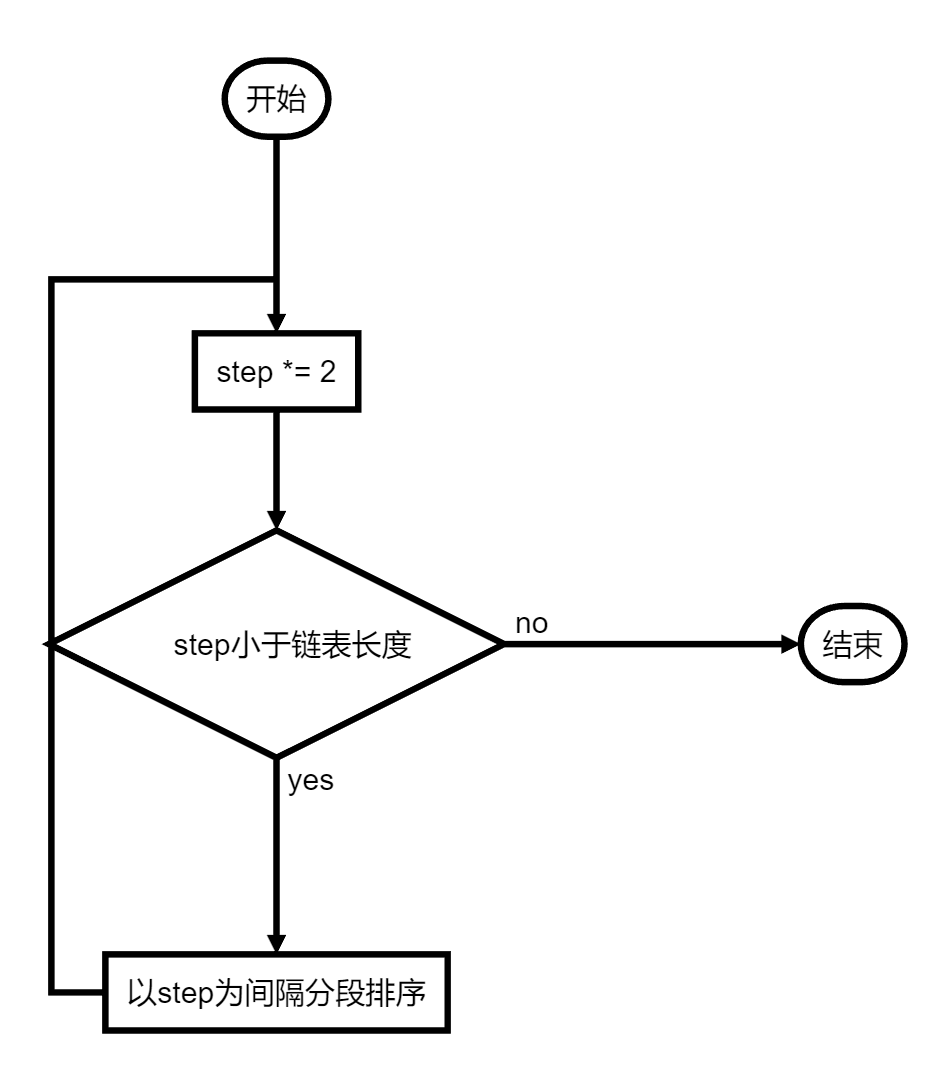
return ans;

}

3.3.5 非递归归并排序实现思路

1. 不使用递归，只能采用step记录每次操作处理的间隔，模拟递归的操作。
2. Step初始为1，在循环中不断 step \*= 2，保证间隔的增加。

3.3.6 非递归归并排序实现流程图



3.3.7 非递归归并排序实现主要代码

template<class T>

void List<T>::mergeSort2()

{

//只有一个或没有结点

if (head->next == NULL || head->next->next == NULL)

return;

list\_node<T>\* cur;

for (int step = 1 ;step < Size; step \*= 2)//间隔不断乘2

{

list\_node<T>\* tempTail = head;

cur = head->next;

//每个间隔进行一分为二和排序合并

while (cur)

{

list\_node<T>\* left = cur;

list\_node<T>\* right = cut(left, step);

cur = cut(right, step);

tempTail->next = merge(left, right);

while (tempTail->next)

tempTail = tempTail->next;

}

tail = tempTail;//更新尾结点

}

}

//从from开始切掉前step个元素，返回最后一个元素

template<class T>

list\_node<T>\* List<T>::cut(list\_node<T>\* from, int step)

{

--step;

while (from && step)

{

from = from->next;

--step;

}

//不够step个元素

if (from == NULL)

return NULL;

list\_node<T>\* next = from->next;

//切断

from->next = NULL;

return next;

}

//合并函数，两条有序链合并为一条有序链

template<class T>

list\_node<T>\* List<T>::merge(list\_node<T>\* l1, list\_node<T>\* l2)

{

list\_node<T>\* D = new list\_node<T>(0);

list\_node<T>\* cur = D;

while (l1 && l2) {

if (l1->val < l2->val)

{

cur->next = l1;

l1->prev = cur;

l1 = l1->next;

}

else

{

cur->next = l2;

l2->prev = cur;

l2 = l2->next;

}

cur = cur->next;

}

while (l1)//连接第一条链剩下的

{

cur->next = l1;

l1->prev = cur;

l1 = l1->next;

cur = cur->next;

}

while (l2)//连接第二条链剩下的

{

cur->next = l2;

l2->prev = cur;

l2 = l2->next;

cur = cur->next;

}

cur->next = NULL;//收尾工作

list\_node<T>\* ans = D->next;

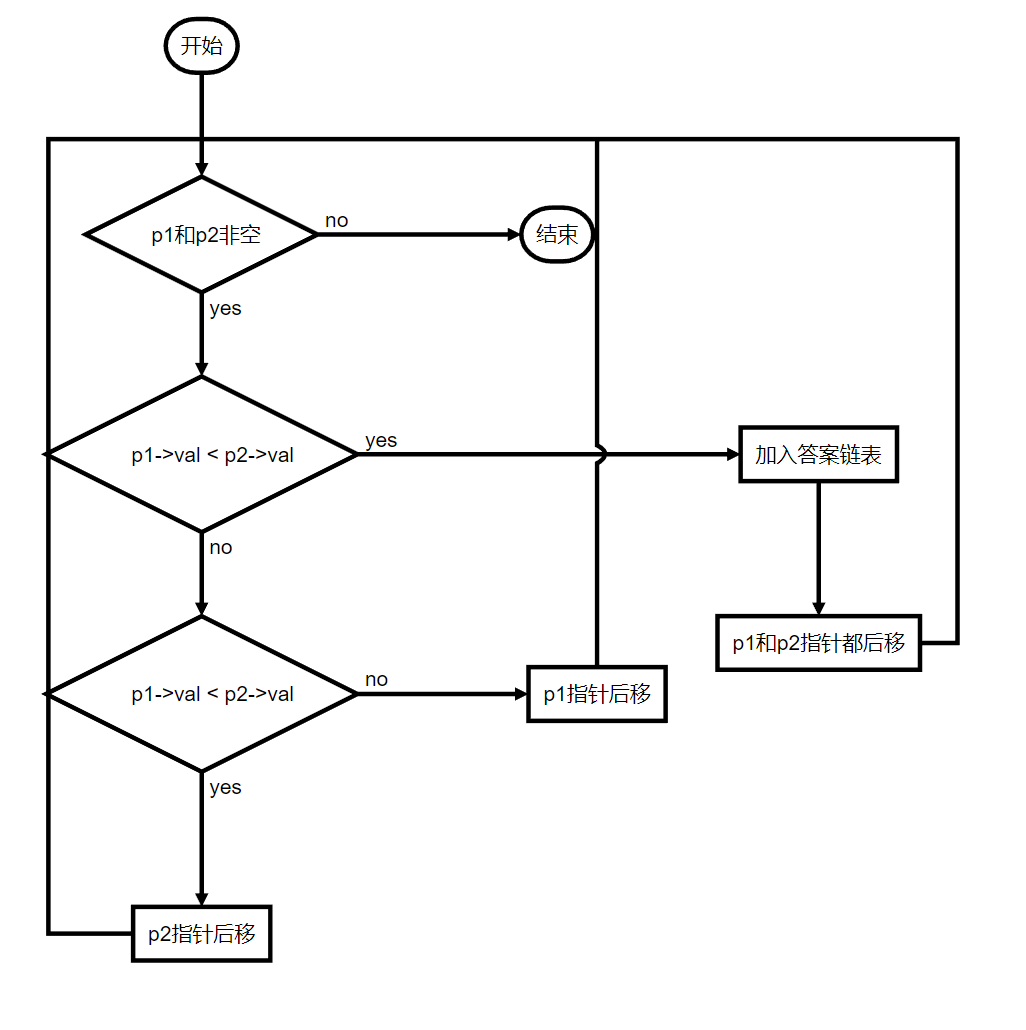
delete D;

return ans;

}

3.4 求交集实现

3.4.1 求交集实现流程图



3.4.2 求交集实现思路

利用两个迭代器指向链表结点，并且利用链表的有序性，从前往后，比较两个迭代器所指向的结点的值，若相等则放入目标数组，若不相等则移动值较小的迭代器，直到其中一条链表结束，则说明求交集结束，之后再也不会存在相交元素。

3.4.3 求交集实现主要代码

template<class T>

List<T> intersection(List<T>& l1, List<T>& l2)

{

List<int> ans;

auto it1 = l1.begin(), it2 = l2.begin();

//两条链指向都有东西

while (it1 != l1.end() && it2 != l2.end())

{

//有交集就加入目标链

if (\*it1 == \*it2)

{

ans.emplace\_back(\*it1);

++it1;

++it2;

}

//移动小的

else if(\*it1 < \*it2)

++it1;

else

++it2;

}

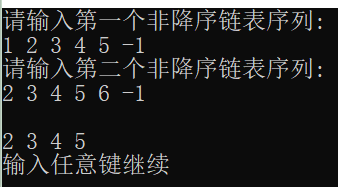
return ans;

}

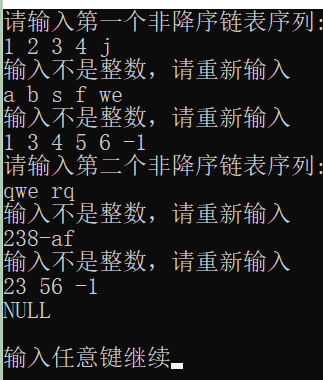
4 项目测试

4.1 输入测试

4.1.1 正常输入测试

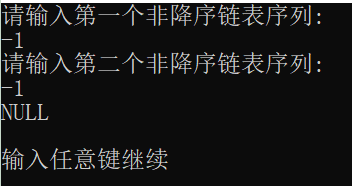


4.1.2 非法字符

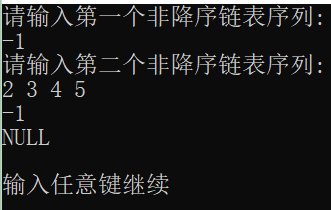


4.2 排序及求交集测试

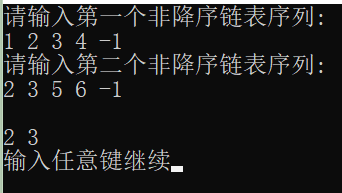
4.2.1 两条空序列



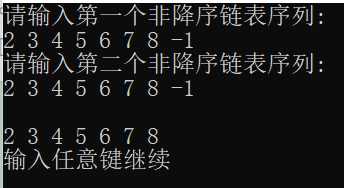
4.2.2 一条空序列



4.2.3 两条序列部分相交



4.2.4 两条序列完全相交



4.2.5 非升序输入

