同济大学

TONGJI UNIVERSITY

数据结构课程设计

项目名称	两个有序链表序列的交集
学　　院	计算机科学与技术学院
专　　业	软件工程
学生姓名	杨瑞晨
学　　号	2351050
指导教师	张颖
日　　期	2024 年 12 月 1 日

目 录

1 项目分析

1.1 项目背景分析

链表是常用的数据结构之一，因其灵活的动态内存分配和操作效率被广泛应用于数据管理。交集查询问题是多个数据集合间分析和比较的重要操作。本项目基于模板类实现链表的定义和操作，通过链表完成两个非降序整数序列的交集计算，具有较强的实用性和扩展性。

1.2 项目功能分析

1.2.1 功能要求

已知两个非降序链表序列 S1 和 S2，设计函数构造出 S1 和 S2 的交集新链表 S3。

1.2.2 输入要求

输入分 2 行，分别在每行给出由若干个正整数构成的非降序序列，用-1 表示序列的结尾（-1 不属于这个序列）。数字用空格间隔。

1.2.3 输出要求

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

1.2.4 项目实例

序号	输入	输出	说明
1	1 2 5 -1 2 4 5 8 10 -1	2 5	一般情况
2	1 3 5 -1 2 4 6 8 10 -1	NULL	交集为空的情况
3	1 2 3 4 5 -1 1 2 3 4 5 -1	1 2 3 4 5	完全相交的情况
4	3 5 7 -1 2 3 4 5 6 7 8 -1	3 5 7	其中一个序列完全属于交集的情况
5	-1 10 100 1000 -1	NULL	其中一个序列为空的情况

2 项目设计

2.1 数据结构设计

采用模板类实现链表（List），每个节点通过 ListNode 类存储数据和指针。链表支持动态增删节点、查找、迭代等基本操作。

链表具有以下一系列优点：

（1）链表是一个动态的数据结构，可以在运行时根据用户的需要来分配内存从而实现链表长度的增加或者缩短，无需初始链表长度；

（2）链表结点的插入、删除、修改等等操作比较方便。在进行这一系列操作的时候，我们只需通过相应函数找到要操作链表节点的地址即可，然后对其进行适当的操作，无需移动其他元素，这样也使操作的时间复杂度减小为 $O(n)$；

（3）链表可以在运行的时候可以根据用户的需要自动地增加和缩短长度，可以减少内存的浪费；

（4）使用模板支持任意数据类型。

2.2 类设计

经典的链表一般包括两个抽象数据类型 (ADT) ——链表结点类 (ListNode) 与链表类 (LinkList)，而两个类之间的耦合关系可以采用嵌套、继承等多种关系。本程序也构造了这两个基本的链表类来进行操作。

2.2.1 ListNode 类

每个链表节点包含数据字段 data 和指针字段 next，支持构造和默认析构。

```
// 定义链表节点类
template <class T>
class ListNode {
    friend class List<T>;
    template <class>
    friend class ListIterator;
private:
    T data;         // 节点数据
    ListNode *next; // 指向下一个节点的指针
public:
    ListNode() : next(nullptr) {}  // 默认构造函数
    ListNode(T val) : data(val), next(nullptr) {} // 带参数的构造函数
};
```

2.2.2 List 类

实现链表操作，主要功能包括：

（1）动态增删节点：push_back、remove

（2）数据查询和修改：search、setData、getData

（3）链表迭代器支持：begin、end

（4）其他功能：链表清空 clear、赋值操作 operator= 等

```
// 定义链表类
template <class T>
class List {
private:
    ListNode<T> *head; // 链表头指针
    ListNode<T> *tail; // 链表尾指针
public:
    List() : head(nullptr), tail(nullptr) {}                    // 默认构造函数
    List(T val) { head = new ListNode<T>(val), tail = nullptr; } // 带参数的构造函数
    ~List() { clear(); }                                        // 析构函数，清空链表

    void push_back(T val);
    bool empty() const { return head == nullptr; } // 判断链表是否为空
    void clear();// 清空链表
    // 返回指向链表头部的迭代器
    ListIterator<T> begin() const { return ListIterator<T>(head); }
    // 返回指向链表尾部的迭代器
    ListIterator<T> end() const { return ListIterator<T>(nullptr); }
    int size() const ;// 计算链表长度
    ListNode<T> *search(T val);// 搜索含数据 val 的元素
    ListNode<T> *locate(int i); // 搜索第 i 个元素的地址
    T *getData(int i);// 取出第 i 个元素的值
    void setData(int i, T &val);// 用 val 修改第 i 个元素的值
    bool insert(int i, T &val) ; // 将 val 插入到链表第 i 个元素
    bool remove(int i) ;// 从链表中删去第 i 个元素
    List<T> &operator=(List<T> &L);// 重载赋值操作符
};
```

2.2.3 ListIterator 类

为链表提供迭代器支持，实现链表的遍历和数据访问。

```cpp
// 定义链表迭代器类
template <typename T>
class ListIterator {
public:
    ListNode<T> *current; // 当前节点指针

    ListIterator(ListNode<T> *node) : current(node) {} // 构造函数

    // 重载 != 操作符，用于比较两个迭代器是否不相等
    bool operator!=(const ListIterator &other) const {
        return current != other.current;
    }

    // 重载 * 操作符，用于获取当前节点的数据
    T &operator*() {
        return current->data;
    }

    // 重载 ++ 操作符，用于将迭代器移动到下一个节点
    ListIterator &operator++() {
        current = current->next;
        return *this;
    }
};
```

3 项目实现

3.1 基本功能的实现

3.1.1 输入链表

通过函数 readList，实现用户输入链表数据：

检查输入合法性，提示用户重新输入非法数据。

支持链表动态扩展，输入以-1 结束。

```
// 读取用户输入，生成链表
List<int> readList() {
    List<int> lst;
    int num;

    bool validInput = false;

    while (!validInput) {
        validInput = true;

        while (true) {
            cin >> num;
            // 检查输入是否合法
            if (cin.fail() || (num < -1 && num != 0)) {
                cerr << " 请输入正整数并以-1 结束! " << endl;
                validInput = false;
                lst.clear();
                cin.clear();
                cin.ignore(numeric_limits<streamsize>::max(), '\n');
                break;
            } else if (num == -1) { break;}
            else { lst.push_back(num); }
        }
    }

    return lst;
}
```

3.1.2 交集计算

函数 findIntersectionLists 通过两个迭代器同步遍历链表，找到相同数据并存入结果链表，特点包括：

利用链表有序性优化比较流程。

只存储交集元素，避免重复。

```
// 寻找两个链表的交集
List<int> findIntersectionLists(const List<int> &lst1, const List<int> &lst2) {
    List<int> result;

    ListIterator<int> it1 = lst1.begin();
    ListIterator<int> it2 = lst2.begin();

    // 遍历两个链表，寻找交集
    while (it1 != lst1.end() && it2 != lst2.end()) {
        if (*it1 == *it2) {
            result.push_back(*it1);
            ++it1;
            ++it2;
        } else if (*it1 > *it2) { ++it2; }
        else { ++it1; }
    }
    return result;
}
```

3.1.3 输出结果

通过 printList 函数，格式化输出交集链表：

若交集为空，输出 NULL；

否则按序输出交集数字。

```
// 打印链表
void printList(const List<int> &lst) {
    if (lst.empty()) {
        cout << "NULL" << endl;
    } else {
        for (ListIterator<int> it = lst.begin(); it != lst.end(); ++it) {
            if (it != lst.begin()) { cout << " "; }
```

```
            cout << *it;
        }
        cout << endl;
    }
}
```

3.2 模板类 List 的实现

```
// 定义链表类
template <class T>
class List {
private:
    ListNode<T> *head; // 链表头指针
    ListNode<T> *tail; // 链表尾指针
public:
    List() : head(nullptr), tail(nullptr) {}                    // 默认构造函数
    List(T val) { head = new ListNode<T>(val), tail = nullptr; } // 带参数的构造函数
    ~List() { clear(); }                                        // 析构函数，清空链表

    void push_back(T val) {
        ListNode<T> *newNode = new ListNode<T>(val);
        if (!head) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }

    // 判断链表是否为空
    bool empty() const {
        return head == nullptr;
    }

    // 清空链表
    void clear() {
        ListNode<T> *current = head;
        while (current) {
```

```
31            ListNode<T> *next = current->next;
32            delete current;
33            current = next;
34        }
35        head = tail = nullptr;
36    }
37
38    // 返回指向链表头部的迭代器
39    ListIterator<T> begin() const {
40        return ListIterator<T>(head);
41    }
42
43    // 返回指向链表尾部的迭代器
44    ListIterator<T> end() const {
45        return ListIterator<T>(nullptr);
46    }
47
48    // 计算链表长度
49    int size() const {
50        int count = 0;
51        ListNode<T> *current = head;
52        while (current) {
53            count++;
54            current = current->next;
55        }
56        return count;
57    }
58
59    // 搜索含数据 val 的元素
60    ListNode<T> *search(T val) {
61        ListNode<T> *current = head;
62        while (current) {
63            if (current->data == val) return current; // 找到
64            current = current->next;
65        }
66        return nullptr; // 没找到返回空指针
67    }
68
69    // 搜索第 i 个元素的地址
```

```
70      ListNode<T> *locate(int i) {
71          if (i < 0) return nullptr;
72          ListNode<T> *current = head;
73          int count = 0;
74          while (current && count < i) {
75              current = current->next;
76              count++;
77          }
78          return current; // 返回第 i 个元素的地址
79      }
80
81      // 取出第 i 个元素的值
82      T *getData(int i) {
83          if (i < 0) return nullptr;
84          ListNode<T> *node = locate(i);
85          if (node) return &(node->data); // 返回寻找到的结点地址
86          return nullptr; // 搜索不成功返回空值
87      }
88
89      // 用 val 修改第 i 个元素的值
90      void setData(int i, T &val) {
91          if (i < 0) return;
92          ListNode<T> *node = locate(i);
93          if (node) node->data = val;
94      }
95
96      // 将 val 插入到链表第 i 个元素
97      bool insert(int i, T &val) {
98          if (i < 0) return false; // 非法值
99          ListNode<T> *newNode = new ListNode<T>(val); // 创建节点
100         if (i == 0) { // 插入到头节点
101             newNode->next = head; // 重新链接
102             head = newNode;
103             if (!tail) tail = newNode;
104             return true; // 成功插入
105         }
106         ListNode<T> *prev = locate(i - 1); // 定位第 i-1 个元素
107         if (prev) {
108             newNode->next = prev->next; // 重新链接
```

```
109             prev->next = newNode;
110             if (!newNode->next) {
111                 tail = newNode; // 插入值后无元素
112             }
113             return true; // 成功插入
114         }
115         delete newNode; // 删除
116         return false; // 未成功插入返回 false
117     }
118
119     // 从链表中删去第 i 个元素
120     bool remove(int i) {
121         if (i < 0 || !head) return false; // 非法值
122         if (i == 0) {
123             ListNode<T> *temp = head;
124             head = head->next;
125             delete temp;
126             if (!head) tail = nullptr;
127             return true; // 成功删除
128         }
129         ListNode<T> *prev = locate(i - 1); // 定位第 i-1 个元素
130         if (prev && prev->next) { // 前后非空
131             ListNode<T> *temp = prev->next; // 将删除元素链接到 temp
132             prev->next = temp->next;
133             if (!prev->next) {
134                 tail = prev; // 更新尾指针
135             }
136             delete temp; // 删除 temp
137             return true; // 成功删除
138         }
139         return false; // 未成功删除
140     }
141
142     // 重载赋值操作符
143     List<T> &operator=(List<T> &L) {
144         if (this != &L) {
145             clear();
146             ListNode<T> *current = L.head;
147             while (current) {
```

```
                push_back(current->data);
                current = current->next;
            }
        }
        return *this;
    }
};
```

3.3 主函数的实现

main 函数逻辑：

（1）调用 readList 读取两个链表；

（2）调用 findIntersectionLists 计算交集；

（3）使用 printList 输出结果。

```
int main() {
    cout << " 请输入第一个链表（若干个正整数构成的非降序序列，用-1 表示结尾，数字用空格间隔）: ";
    List<int> lst1 = readList(); // 读取第一个链表
    cout << " 请输入第二个链表: ";
    List<int> lst2 = readList(); // 读取第二个链表
    List<int> result = findIntersectionLists(lst1, lst2); // 寻找交集
    cout << " 二链表交集为: ";
    printList(result); // 打印交集

    return 0;
}
```

4 项目测试

4.1 输入有效数据

4.1.1 一般情况

测试用例:

1 2 5 -1

2 4 5 8 10 -1

预期结果： 2 5

测试结果:

```
请输入第一个链表（若干个正整数构成的非降序序列，用-1表示结尾，数字用空格间隔）：1 2 5 -1
请输入第二个链表：2 4 5 8 10 -1
二链表交集为：2 5
```

4.1.2 交集为空的情况

测试用例:

1 3 5 -1

2 4 6 8 10 -1

预期结果： NULL

测试结果:

```
请输入第一个链表（若干个正整数构成的非降序序列，用-1表示结尾，数字用空格间隔）：1 3 5 -1
请输入第二个链表：2 4 6 8 10 -1
二链表交集为：NULL
```

4.1.3 完全相交的情况

测试用例:

1 2 3 4 5 -1

1 2 3 4 5 -1

预期结果： 1 2 3 4 5

测试结果:

```
请输入第一个链表（若干个正整数构成的非降序序列，用-1表示结尾，数字用空格间隔）：1 2 3 4 5 -1
请输入第二个链表：1 2 3 4 5 -1
二链表交集为：1 2 3 4 5
```

4.1.4 其中一个序列完全属于交集的情况

测试用例:

3 5 7 -1

2 3 4 5 6 7 8 -1

预期结果： 3 5 7

测试结果:

```
请输入第一个链表（若干个正整数构成的非降序序列，用-1表示结尾，数字用空格间隔）：3 5 7 -1
请输入第二个链表：2 3 4 5 6 7 8 -1
二链表交集为：3 5 7
```

4.1.5 其中一个序列为空的情况

测试用例:

-1

10 100 1000 65472 -1

预期结果： NULL

测试结果:

```
请输入第一个链表（若干个正整数构成的非降序序列，用-1表示结尾，数字用空格间隔）：-1
请输入第二个链表：10 100 1000 65472 -1
二链表交集为：NULL
```

4.2 健壮性测试

输入非法字符或负数（非-1），提示重新输入。

```
请输入第一个链表（若干个正整数构成的非降序序列，用-1表示结尾，数字用空格间隔）：1 3 2 q 9 -1
请输入正整数并以-1结束！
2 -2 3 4
请输入正整数并以-1结束！
```

5 项目心得与体会

通过本项目的深入开发与实践，我全面掌握了以下几项关键技能：

（1）模板类和链表结合的动态数据结构设计：我学会了如何巧妙地将模板类的灵活性与链表的动态扩展性相结合，设计出既能适应不同类型数据存储需求，又能高效进行元素插入、删除和遍历的动态数据结构。这一过程中，我深入理解了模板元编程的奥秘，以及如何在保证类型安全的同时，提升代码的可重用性和可维护性。

（2）C++ 标准库迭代器的实现与应用：通过亲手实现自定义容器的迭代器，我不仅掌握了 C++ 标准库中迭代器模式的精髓，还学会了如何利用迭代器接口来统一访问容器中的元素，无论是顺序容器还是关联容器。这一技能的提升，让我在处理复杂数据结构时，能够更加灵活地遍历、修改和查询数据，极大地提高了编程效率和代码的可读性。

（3）数据验证与用户输入交互的细节优化：在项目实践中，我深刻认识到用户输入的正确性和有效性对于程序稳定运行的重要性。因此，我专注于优化数据验证逻辑，确保所有输入都符合预期格式和范围。同时，通过引入更加友好的用户交互界面和错误提示信息，显著提升了用户体验。这些努力不仅使程序更加健壮，也增强了我在处理用户输入和反馈方面的能力。

此外，项目中涉及的链表交集查询问题，不仅考验了我对链表操作和数据结构算法的深入理解，还为解决更复杂的数据处理问题奠定了坚实的基础。在解决这个问题的过程中，我不断尝试并优化不同的算法策略，如双指针法、哈希表法等，从而深刻体会到了算法选择对效率提升的关键作用。这一系列的实践探索，极大地增强了我对算法效率的理解和实际操作能力，让我在面对复杂数据处理任务时更加自信和有准备。