同济大学 Tongji University

数据结构课程设计项目说明文档

——两个有序链表序列的交集

作者姓名：陈垲昕

学号：[redacted]

指导教师：张颖

学院/专业：软件学院/软件工程

1. 分析

1.1 背景分析

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。

输入说明：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。

输出说明：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 功能分析

本程序是求两个链表中元素的交集，常见的有两种算法（以下对两个链表简称A和B）：一是对A链表中每个节点元素，通过遍历B链表每个元素找出与该元素相同的元素，其复杂度为:

$$O(len(A) * len(B))$$

第二种算法是采用双指针并行法(需在A\B链表为非降序的情况下进行)：分别将两个指针指向两个待提取链表的开头元素。对当前两个遍历指针中值较小的链表单元，推进遍历指针并删除该单元以节省空间；而对两个链表指针值相等的情况，此时如果交集链表中的尾部没有该元素，才插入元素并推进两个链表指针；反之不插入元素，只推进两个链表指针，该方法的复杂度为：

$$O(len(A) + len(B))$$

可见优于第一种算法，故本程序采用第二种算法生成交集

此外还有定义并查集类的方法生成交集链表等方法。

程序最终生成了交集链表，并清空了原先的两个链表，节省了内存空间

2.设计

2.1 主要数据结构设计

题目已经要求使用链表进行程序设计。本程序使用一个简单的附加头结点的链表模板类SimpleList，含有头、尾指针，功能主要有输入链表，输出链表，尾部添加程序与删除第一个含有元素的结点的功能；另外定义了程序运行主题类，用于生成交集链表。

本程序的主体类为：

1.代表节点的节点模板类Node

基本描述：模板类SimpleList的结点类，实现良好的封装。含有默认构造函数与含值构造函数，数据成员data与下一个节点域指针next

```
template<class T>
struct Node{

    Node() :next(nullptr) {}                    //构造函数
    Node(const T& nodeData) :data(nodeData) {}  //值初始化构造函数

    T data;                                     //数据成员
    Node* next;                                 //链接指针域
};
```

2.链表主体模板类StudentList

基本描述：class SimpleList类为求交集链表所用的链表类，以功能类FindIntersection为友元，含有头、尾指针功能主要有输入链表，输出链表，尾部添加程序与删除第一个含有元素的结点的功能；另外定义了程序运行主题类，用于生成交集链表

```
template<class T>
class SimpleList {
    friend class FindIntersection<T>;
public:
    SimpleList() :_pTop(new Node<T>()), _tail(_pTop) {}

    void inputList();            //输入元素构建连，包含自动排序程序
    void printList()const;       //输出链表

    inline void addNodeToTail(Node<T>* node);
    inline void deleteFirstNodeWithElement();

private:
    Node<T>* _pTop;          //附加头结点
    Node<T>* _tail;          //尾指针，指向尾元素
};
```

2.2 系统类设计

基本描述：本程序功能主要实现，内含两个链表类成员*list1*,list2，定义了输入函数input，并能通过void类型的成员函数generateIntersectionList生成*list1*,list2的交集链表

```
template<class T>
class FindIntersection {
public:

    FindIntersection() :_list1(new SimpleList<T>), _list2(new SimpleList<T>)
{}//初始构造函数


    void input();                                //输入
    SimpleList<T>* generateIntersectionList();  //生成交集链表

private:
    SimpleList<T>* _list1;
    SimpleList<T>* _list2;
};
```

3.核心功能实现

3.1求取交集链表的操作

描述：通过FindIntersection类的 generateIntersectionList() 函数实现，对_list1与__list2求得交集链表并同时清除

双指针遍历算法具体描述：

- 新建链表**list3**

- 使用指针 **pCheck1**, **pCheck2**分别指向_**list1**和_**list2**的附加头结点的下一个结点（即第一个有实值的结点）

```
SimpleList<T>* list3 = new SimpleList<T>();
Node<T>* pCheck1 = _list1->_pTop->next,* pCheck2 = _list2->_pTop->next;
```

- 如果**pCheck2**指向的元素的值大于**pCheck1**指向的元素的值，则**pCheck1**向后移一,并将前一个结点从链表_**list1**中删除；如果**pCheck1**指向的元素的值大于**pCheck2**指向的元素的值，则**pCheck2**向后移一,并将前一个结点从链表_**list2**中删除

```
/*对当前值较小的链表单元，推进遍历指针并删除该单元*/
if (pCheck1->data < pCheck2->data)
{
    pCheck1 = pCheck1->next;
    _list1->deleteFirstNodeWithElement();
}
else if (pCheck1->data > pCheck2->data)
{
    pCheck2 = pCheck2->next;
    _list2->deleteFirstNodeWithElement();
}
```

- 当**pCheck1**和**pCheck2**的值相等时，即取到交集元素，此时如果交集链表**list3**中的尾部没有该元素，此时才插入元素,防止重复元素插入交集链表

```
/*当pCheck1和pCheck2的值相等时，即取到交集元素*/
/*此时如果交集链表中的尾部没有该元素，此时才插入元素*/
if (pCheck1->data != list3->_tail->data)
{
    Node<T>* pLinker = new Node<T>(pCheck1->data);
    if (!pLinker)
    {
        std::cerr << "错误：分配内存失败，将退出程序" << std::endl;
        exit(1);
    }
    list3->addNodeToTail(pLinker);
}

/*推进遍历链表并删除之前的单元*/
pCheck1 = pCheck1->next;
_list1->deleteFirstNodeWithElement();
pCheck2 = pCheck2->next;
_list2->deleteFirstNodeWithElement();
```

- 重复3,4,5步，直到**pCheck1**或**pCheck2**其中一个为空

```
while (pCheck1 && pCheck2)
{
    /*对当前值较小的链表单元，推进遍历指针并删除该单元*/
    ...
    /*当pCheck1和pCheck2的值相等时，即取到交集元素*/
    ...
    /*此时如果交集链表中的尾部没有该元素，此时才插入元素*/
    ...
}
/*遍历结束，生成交集链表，原来的两个链表清空*/
```

- 打印**list3**，得到交集链表

3.2 插入链表&容错输入

描述：在依次插入链表元素时进行排序，确保在乱序输入链表时也能得到非降序链表

```
template<class T>
void SimpleList<T>::inputList()
{
    T nodeData;
    bool hasInserted = false;    /*表示该节点是以插入而非尾后连接的形式添加的，说明输入是乱序*/
    while (std::cin >> nodeData)
    {
        hasInserted = false;
        if (nodeData == -1)
        {
            break;
        }
        else
        {
            Node<T>* newNode = new Node<T>(nodeData), * pLinker = _pTop;

            if (!newNode)
            {
                std::cerr << "错误：分配内存失败，将退出程序" << std::endl;
                exit(1);
            }

            while (pLinker->next)
            {
                /*遍历链表,正序情况下pLinker会遍历到最后一个元素，以执行尾后连接*/
                if (pLinker->next->data < nodeData)
                {
                    pLinker = pLinker->next;
                }
                /*当出现需要插入的情况（乱序）时，进行链表的插入操作*/
                else
                {
                    newNode->next = pLinker->next;
                    pLinker->next = newNode;
                    /*标识插入的情况*/
```

```
                    hasInserted = true;

                    break;
                }
            }
            /*正常情况下，如果节点是依次正序输入，则执行尾后连接程序*/
            if (!hasInserted)
            {
                addNodeToTail(newNode);
            }

        }
    }
}
```

4.测试

4.1 常规结果测试

Task 1 两个链表中存在相同元素

测试样例：

```
1 2 5 -1
2 4 5 8 10 -1
```

预期结果：

```
2 5
```

实际结果：

```
1 2 5 -1
2 4 5 8 10 -1
2 5
```

Task 2 两个链表中不存在相同元素

测试样例：

```
1 3 5 -1
2 4 6 8 10 -1
```

预期结果：

```
NULL
```

实际结果：

```
1 3 5 -1
2 4 6 8 10 -1
NULL
```

Task 3两个链表完全相同

测试样例：

```
1 2 3 4 5 -1
1 2 3 4 5 -1
```

预期结果：

```
1 2 3 4 5
```

实际结果：

Task 4 一个链表被另一个链表所包含

测试样例：

```
3 5 7 -1
2 3 4 5 6 7 8 -1
```

预期结果：

```
3 5 7
```

实际结果：

Task 5 一个链表为空（即交集为空）

测试样例：

```
-1
10 100 1000 -1
```

预期结果：

```
NULL
```

实际结果：

Task 6 两个链表均为空

测试样例：

```
-1
-1
```

预期结果：

```
NULL
```

实际结果：

Task 7 乱序输入链表

测试样例：

```
5 1 2 -1
10 4 5 2 8 -1
```

预期结果：

```
2 5
```

实际结果：