# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

**2.1.1 总述**

采用链式表的物理结构，实现一个带有菜单的演示系统，定义了初始化表，销毁表，清空表，判定空表，求表长，求前驱与后继，数据插入，数据删除，线性表的遍历，多表操作等多个功能；本系统可以使用文件的形式进行存储和数据提取。在主程序中完成函数调用所需实参值的准备和函数执行结果的显示，并给出适当的操作提示显示。本演示系统可选择实现多个线性表管理。

**2.1.2 多表管理与附加功能**

在该系统中，可以对不同序号的链表进行操作，进入系统时可以选择20号来选择要在哪个链式线性表上进行操作，默认情况下操作1号线性表。

附加功能：

1）本系统实现了线性表的文件保存和线性表的文件加载，选择13号功能时进行文件存储，表中首先存储线性表的长度length和所能存储最大元素数量listsize，随后记录各个元素；

2）选择14号功能时首先读取所输入的文件名对应的文件中存储的表长length和listsize属性，然后再将数据元素一个一个加入表中；

3）选择15号功能可以对链式线性表中的数据元素进行排序，选择该功能后会提示输入排序的准则，输入0表示进行升序排序，输入1表示进行降序排序；

4）选择16号功能可以对链式线性表进行反转链表操作，反转后表中数据元素的顺序将为之前的逆序；

5）选择17号功能可以对链式线性表中所有特定的元素进行删除操作，即选择一个数后， 表中所有出现的该数都会被删除；

6）选择18号功能可以对两个升序线性表进行合并操作，即包含两个链表的所有数据元素的新的升序线性表，并存储在两个线性表中的前者的存储空间里；当存在空表时系统会给出提示；

7）选择19号功能可以判断当前正在操作的链表是不是一个回文链表。

各函数功能的条件，输出以及设计思路见下文。

## 2.2 系统设计

**2.2.1 数据的物理结构**

typedef struct LNode {

ElemType data;//结点的数据域

struct LNode\* next;//结点的指针域

}LNode, \* LinkList;

定义LNode为结构体的直接引用，LinkList为结构体的指针引用；

**2.2.2 线性表的逻辑结构与函数运算**

线性表的数据逻辑结构定义：

ADT List{

数据对象：D={};

数据关系：}

};

系统以函数形式定义了初始化表，销毁表，清除表中元素，判断表是否为空，求表长，获取特定位置的元素，元素定位，求元素的前驱，求元素的后继，插入元素，元素删除，线性表的遍历，文件存储线性表，读取文件中的线性表，线性表的排序，链表的反转，移除特定元素，合并升序链表，回文链表的判断等功能，这些功能均可以在多个线性表上实现，使用功能时需输入对应编号。

函数的定义:

1）初始化表（IntiaList）：构造一个空的线性表，初始条件是线性表L不存在或已存在；操作结果是构造一个空的线性表，操作成功返回OK；

2）销毁表（DestroyList）：销毁线性表，初始条件是线性表已存在，操作

结果是销毁线性表，与清空不同，销毁时释放全部空间，操作成功返回OK；

3）清空表（ClearList）：清空线性表，条件是线性表L已存在，操作结果

是将L重置为空表，操作成功返回OK；

4）判定空表（ListEmpty）：判定表是否为空，初始条件是线性表为L存在，若为空表则返回TRUE, 否则返回FALSE；

5）求表长（ListLength）：求线性表的表长，初始条件是线性表L存在，返回值为表中数据元素的个数；

6）获得元素（GetElem）：得到某个位置上元素的值，位置从1开始到表长大小， 操作结果是用int型变量e返回该位置的值；

7）查找元素（LocateElem）：查找与传入参数e满足cmp()关系的元素，程序初始默认为查找与e相同的元素的位置，若元素不存在，返回值为0；

8）获得前驱（PriorElem）：用pre\_e保存传入参数cur\_e的前驱元素的值，若存在返回其值，否则操作失败，pre\_e没有定义；

9）获得后继(NextElem)：用next\_e保存传入参数cur\_e的后继元素的值，若存在返回其值，否则操作失败，next\_e没有定义；

10)插入元素（ListInsert）：L已存在且非空，;结果是向位置i的前方插入元素e，插入成功返回OK，插入元素错误返回ERROR;

11)删除元素(ListDelete)：L已存在且非空，,结果是删除L的第i个元素，用传入参数e返回其值；删除成功返回OK,否则返回ERROR;

12)遍历表（ListTraverse）：按顺序输出线性表中所有元素的值；

13)线性表保存至文件(SaveList)：将线性表保存至文件，文件名为初始化表时所输入的文件名filename，包括写入表长，允许大小，所有元素；

14)加载文件中的线性表(LoadList)：加载filename文件里的线性表，读取表长，允许存储的元素数量，再将所有数据元素依次赋予给当前操作的表；

15）线性表排序（SortList）：选择0或1进行功能选择，输入0是对表中数据元素进行升序排序，选择1是对表中数据元素进行降序排序；

16）线性表反转（ReverseList）：初始条件是L存在，将链式线性表反转，使表尾元素成为新的表头结点，表中元素全部逆序；

17）删除特定元素（RemoveSpecificData）：初始条件是表L存在，输入一个所要删除的元素，函数的操作结果是删除表L中所有数据域与该元素相等的结点；

18）合并链表（MergeList）：初始条件是两个非空的升序线性表（表中也可以只有一个元素），操作的结果是将两个升序链式线性表合并，新的线性表存储在前一个表中，表中包含两个线性表中所有元素并且依然是升序排列；

19）判断是否为回文链表（IsPalindromeList，RecursivelyCheck）：初始条件是线性表L非空，用于判定所操作的链表是否为回文链表，通过递归使用函数RecursivelyCheck来实现，该函数的参数为表头结点的下一个结点，该函数的作用是判断回文并且返回status变量给IsPalindromeList.

20）整数相加（ListAddition）：初始条件是两个记录整数各个位数的链表，高位在前地位在后，模拟真实的整数顺序，同时允许前导零；操作的结果是将两张表所代表的整数进行相加，将结果的各个位数存储在表L1中，依然高位在前，没有前导零。

**2.2.3 输入与输出说明**

进入系统后首先出现功能菜单，首先输入所要操作的线性表的序号，这里允

许在1~10号上操作；菜单会罗列20个选择项，系统会提示首先选择功能20来选择所要操作的表的序号，选择完毕后就可以选择功能对应的序号进行相应操作，倘若不进行所要操作的表的序号选择，则会默认在序号为1的链式表上进行操作。

系统通过switch-case语句进行对应的功能操作，每一个操作完毕后按回车或其他字符跳出当前操作，继续执行while循环选择功能并实现，在每一次操作结束后系统都会显示操作的结果，包括异常情况如“表不存在”或“表为空”的报告。对于表中元素操作，可以通过功能12表的遍历来验证每一个功能实现是否成功。当在系统中输入0后会退出系统。

当要同时使用多张表时，只需在操作所对应的表时先选择功能20，输入所要操作的表的序号，即可转换为对新的表进行操作，原来的表中的数据元素依然会保存。

## 2.3 系统实现

**2.3.1 环境说明**

本系统在Microsoft Visual Studio2019上执行，编译器为MSVC++ 14.2 \_MSC\_VER == 1920.并在其上编译运行。

**2.3.2 预定义常量**

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

const int MAX\_NUM = 10;//所能操作的最多的表数

typedef int status;

typedef int ElemType;

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

**2.3.3 链式线性表的实现**

链式存储结构的线性表使用存储数据元素的链表实现，结构体LNode中，定义数据域ElemType data用于存储数据，定义指针域struct LNode\* next来指向下一个结点或NULL，对于数据元素的插入等操作需要新的存储空间时，使用(LinkList)malloc(sizeof(LNode))来申请新的存储空间来存储新的数据元素，当出现删除结点操作时可通过free（）函数释放内存空间。

本系统中的链表初始化时都会被置为带表头结点的链表，表头结点中的数据元素为0且不会被操作访问，表头结点的指针域指向线性表的首结点或者NULL。

本实验使用结构体指针数组存储多个链表来实现多表管理，定义MAX\_NUM来指定数组的大小，即所能操作的线性表的数量，i\_num为所要进行操作的表的序号，可在功能20中进行选择，初始化为1。初始条件下LinkList指针数组中所有元素均置为NULL;

选择功能后，进入case语句后都会对线性表是否存在进行判断，若存在线性表为空则会显示线性表不存在，因此，下列对函数设计过程的描述除了InitList外均在链式表存在的情况下。

1）status InitList(LinkList\*L)

传入所要初始化的表的指针，当\*L不为空时，即表头结点已经存在，则函数返回ERROR，在主函数中输出“表不为空，无需初始化”；\*L为空时则需要对\*L进行空间申请，若空间不足则返回OVERFLOW，申请空间成功后，对结点中的数据域赋值为0，结点域初始化为NULL,创建成功后返回OK；

2）status DestroyList(LinkList\* L)

定义两个LinkList类型的指针p,q，先让p指向表头结点，q指向p的下一个结点，当p指向不为NULL时，使用free不断释放p指向的空间，然后将p指向q指向的结点，再使q移向下一位，直到p指向的空间为NULL，销毁线性表完成后返回OK；

3）status ClearList(LinkList\* L)

定义两个LinkList类型的指针p,q,让p指向表头结点\*L的下一个结点，当p的指向空间不为NULL时，使q指向p的下一个结点，释放p中的空间后使p指向q；当p指向NULL时退出循环，给表头结点\*L的指针域赋值NULL，返回OK。

4）status ListEmpty(LinkList L)

直接判断表头结点的下一个结点是否为空，若为空则说明没有存储数据元素，返回TRUE,即为空，否则返回FALSE,即不为空；

5）int ListLength(LinkList L)

定义一个用于计数的变量len，初始化为0，定义一个LinkList型的指针p指向表头结点的下一个结点即首结点，进行while循环判定p指向空间是否为空，不为空则p向后移一位,len自增；循环到表尾后返回len即可；

6）status GetElem(LinkList L,int i,ElemType \*e)

i表示所要取数据元素的位置,e用于记录结点值，定义一个整型变量place记录当前位置，定义指针LinkList p指向首结点，当p指向空间不为NULL且place<i时p后移并且place自增,若循环完毕后判定发现p为空（i太大）或者place>i（i太小）则返回ERROR；否则用e记录结点中的值，返回OK；

7）int LocateElem(LinkList L,ElemType e,status(\*compare)(ElemType a,ElemType b))

函数status compare(ElemType a,ElemType b)作为函数参数，用来比较标准的元素和待比较的元素的大小，当比较元素满足compare中的限制条件时，在LocateElem中所定义的是相等条件，即元素与标准元素相等时，会返回true,否则返回false;

定义i代表当前搜索到的位置，每向前移动一个位置，i的值加1，若元素与e满足相等关系，则返回i的值，否则令p指向p的下一个结点，当p指向NULL时退出循环，这意味着不存在该元素，返回0；

8）PriorElem(LinkList L,ElemType cur\_e,ElemType\* pre\_e)

定义一个LinkList型的结点指针p，指向第一个元素的结点，当p中的元素等于cur\_e时，表示表中第一个元素即为cur\_e，那么它没有前驱，返回ERROR;当p->next不为空并且p->next的数据元素不为cur\_e时，p指针不断后移，否则退出while循环，此时若p->next为NULL说明表中不存在这个数据元素，则返回OVERFLOW,否则说明找到了该元素，用pre\_e记录此时p中的数据元素并返回OK;

9)NextElem(LinkList L, ElemType cur\_e,ElemType\* next\_e)

定义一个LinkList型的结点指针指向第一个元素的结点，当p->next不为NULL且p指向的结点的数据元素不为cur\_e时，p不断后移，退出循环时进行判定，若p中的数据元素不为输入的元素且p->next指向NULL，说明表中没有此元素，返回OVERFLOW,若p中的数据元素等于所输入的元素，判断p->next是否为空，若为空，则说明最后一个元素是要找的元素，而它没有后继，若p->next不为空，则用next\_e记录该节点的data值，返回OK；

10）ListInsert(LinkList\* L,int I,ElemType e)

I为插入位置，定义一个整形变量j来表示当前结点指针所在的位置，定义结点指针p指向表头结点，当p不为空且j<i时p不断后移，j也随着每次自增1，当循环退出时，判断p是否为空或者j是否大于i，这两种情况都说明插入位置不正确，否则新建立一个LinkList结点q，并给其分配空间，给q的data域赋值e，让q的next指针指向p的下一个结点，再令p的next指针指向q从而完成结点的插入，插入完成后返回OK；

11）ListDelete(LinkList\* L,int i,ElemType\* e)

要执行删除一个结点的操作，则需要定位到它的上一个结点，仍然定义一个整型变量j来表示当前位置，LinkList指针p指向表头结点，通过while循环找到删除位置，这时判断p->next是否为NULL或者j是否比i大，若是则L是空元素表或者删除位置不对，返回ERROR；否则执行删除操作，定义指向结点的指针q指向p的next结点，修改p的next指针的指向为q->next，用e记录q中的数据元素之后通过free清除q的空间，从而完成删除操作，返回OK；

12）status ListTraverse(LinkList L)

定义一个节点指针p指向表头结点的下一个结点，即首结点，若p指向NULL，则说明L为空表，返回ERROR；否则使用while循环输出p中的数据元素，p不断向后移动直到指向NULL时退出循环，遍历结束后返回OK；

13）SaveList(LinkList L,char\* filename)

filename是初始化时链表的名称，首先定义结点指针p指向首结点，定义文件指针\*fp=fopen(filename,”w”),首先写入表长ListLength(L),再写入表的初始容量，然后使用while循环持续写入p所指向的结点的数据元素，p后移，写入完毕后关闭文件，返回OK；

14）LoadList(LinkList\* L,char\* filename)

定义文件指针fp=fopen(filename,”r”),首先读入表长和表的初始容量，然后读入元素，每次读入元素时使用length控制while循环的进行，在循环语句内，使用ListInsert来插入元素到表中，读取文件完毕后关闭文件并返回OK;

15）SortList(LinkList L,int tag)

Tag表示进行升序排序还是降序排序，定义两个结点指向结点的指针p，q指向首结点，第一次遍历得到表长，若表长为0，则说明为空线性表，返回ERROR,否则根据tag的大小来执行语句，若为0，表示进行升序排序，这里使用冒泡排序交换数据域的方式进行排序，若为1，则使用冒泡排序交换数据域的方式进行降序排序；排序完毕后返回OK；

16）ReverseList(LinkList\* L)

反转链表，首先定义指向结点的指针pre指向首结点，若pre指向NULL则说明表为空，返回ERROR；定义一个LinkList型的指针cur，初始化为NULL，用作反转过程中的后一个结点，当pre不为空时，先用一个指针t记录pre的下一个结点，将pre的next指针域修改为指向cur,实现顺序的逆序，再将cur前移指向pre，pre指向先前的记录值t；当pre指向NULL时即表尾时退出while循环，令表头结点的next指针域修改为最新的cur，即之前的尾结点作为反转后的首结点，返回OK；

17）RemoveSpecificData（LinkList L,int val）

定义一个整形边浪tag，初始化为0；定义一个指向结点的指针指向首结点，当它为NULL时说明表为空，返回INFEASTABLE;使用while循环进行遍历，当遇到p指向的结点域的下一个结点中的元素与传入参数val相同时，先记录这个结点，令p的next域修改为下一个的下一个结点，删除记录的节点的空间，tag置为1，否则p后移；当循环结束时若tag为0，则说明表中没有等于val的元素；

否则返回OK；

18）MergeList(LinkList\* L1,LinkList\* L2)

合并两个升序链表，将合并后的链表赋给L1，首先判断是否两个链表都是空链表若存在则返回ERROR；若L1,为空链表直接将L1指向L2，若L2为空链表直接返回OK；

两表均不为空时，定义一个结点head,分配空间作为首结点，p和q指针分别指向L1和L2的首结点，再定义一个LinkList型的指针move指向head；当p和q均不指向NULL时，给move的next分配空间作为新结点，move移向下一个结点准备进行赋值，对p和q所指向的结点中的数据大小进行判断，当前者小于后者时，将p中的数据赋给move的data，p后移，反之则将q中的数据赋给move中的data，q后移；当p和q中有一个为NULL导致循环退出时，对两个指针进行特判，若p为NULL，则给move的next域指向q，否则指向p，让\*L1指向表头结点head，返回OK；

19）status IsPalindromeList(LinkList L)

status RecursivelyCheck(LinkList currentNode)

使用递归算法判断链表是否是回文链表，定义一个从前向后遍历的结点指针frontPointer指向L，返回函数RecursivelyCheck的返回值，将L作为参数传入，参数名为currentNode，递归传递currentNode的next结点，递归的末尾检查尾结点，此时跳出第一个if语句，currentNode回到上一个结点，开始比较currentNode和frontPointer所指向的结点域的数据元素，若二者不相等，直接返回FALSE;否则frontPointer后移一位，返回TRUE,回到上一层，相当于堆栈中curentNode的上一个结点与后移后的frontPointer中的数据元素继续比较，这样就能一个指针从前往后另一个指针从后往前判断是否是回文链表；若遍历完毕未返回FALSE则说明是回文链表，RecursivelyCheck函数给IsPalindromeList函数返回TRUE；

20)ListAddtion(LinkList\* L1,LinkList\* L2)

首先判断两张表是否存在空表，若存在空表则返回ERROR；定义一个结点head作为结果表示的表头结点，定义两个指向结点的指针p,q分别指向表L1，L2的首结点，用两个整形数组记录两个数字的各位的数字，数组ans用来记录结果，首先将两个链表中各个位上的数字按顺序记录在a,b数组中，前导零通过标记的形式跳过，然后将a，b数组中的数字反转，低位在前进行加法运算，每个位数的相加若超过9则取模10，定义的进位数pro=1，每位的累加都需要加上pro，加上后pro归0，在计算到最高位时特判一下pro是否为0，为1时再进位，ans中最高位为1。

然后将ans中的各个数从下标最大的mx开始，遍历至下标为0，存储入链表中（因为高位在前，所以从后往前记录），然后将表1的表头结点的next域修改为head；返回OK。

**2.4测试用例**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 测试功能 | 输入的参数 | 预计结果 |
| 用例1 | Choose the specific List | 15 | 请选择正确范围！ |
| 用例2 | Choose the specific List | 2 | 选择链表2进行操作 |
| 用例3 | DestroyList | / | 线性表不存在 |
| 用例4 | ListInsert | / | 线性表不存在 |
| 用例5 | ClearList | / | 线性表不存在 |
| 用例6 | ListEmpty | / | 线性表不存在 |
| 用例7 | ListLength | / | 线性表不存在 |
| 用例8 | ListTraverse | / | 线性表不存在 |
| 用例9 | InitList | 表的名称：aa | 创建一个空的线性表，表的名称即filename为“aa” |
| 用例10 | ListTraverse | / | 线性表是空表 |
| 用例11 | ListLength | / | 线性表表长为0 |
| 用例12 | ListInsert | 元素：10 位置：1 | 表：10 |
| 用例13 | ListInsert | 元素：20 位置：2 | 表：10 20 |
| 用例14 | ListInsert | 元素：30 位置：3 | 表：10 20 30 |
| 用例15 | ListInsert | 元素：40 位置：4 | 表：10 20 30 40 |
| 用例16 | ListInsert | 元素：60 位置：6 | 插入位置不正确！  插入数据元素失败！ |
| 用例17 | ListEmpty | / | 线性表不是空表 |
| 用例18 | ListLength | / | 线性表表长为4 |
| 用例19 | GetElem | 2 | 20 |
| 用例20 | GetElem | 5 | 输入位置错误 |
| 用例21 | LocateElem | 40 | 4 |
| 用例22 | LocateElem | 35 | 该元素不存在 |
| 用例23 | PriorElem | 20 | 10 |
| 用例24 | PriorElem | 10 | 它不存在前驱元素 |
| 用例25 | PriorElem | 15 | 表中没有该元素 |
| 用例26 | NextElem | 20 | 30 |
| 用例27 | NextElem | 40 | 它不存在后继元素 |
| 用例28 | NextElem | 60 | 表中没有该元素 |
| 用例29 | ListDelete | 3 | 表：10 20 40 |
| 用例30 | ListDelete | 4 | 删除数据元素失败（此时只有三个元素，位置错误） |
| 用例31 | ListTraverse | / | 10 20 40 |
| 用例32 | SaveList | / | 表中元素为:  10 20 40 存储的表长为3,所能存储数据元素的数量为100.  文件保存成功  文件名为初始化时所输入的“aa” |
| 用例33 | LoadList | aa | 加载得到表长为3,容量为100的线性表.  表中元素为10 20 40 文件加载成功 |
| 用例34 | ClearList | / | 得到长度为0的空表 |
| 用例35 | ListLength | / | 0 |
| 用例36 | ListEmpty | / | 线性表为空表 |
| 用例37 | DestroyList | / | 销毁线性表成功，再操作会显示表不存在（以选择功能3,4,5为例） |
| 用例38 | ListEmpty | / | 线性表不存在 |
| 用例39 | ClearList | / | 线性表不存在 |
| 用例40 | ListLength | / | 线性表不存在 |
| 用例41 | SortList | 1 5 5 14 8 tag:0 | 1 5 5 8 14 |
| 用例42 | SortList | 1 5 5 14 8 tag:1 | 14 8 5 5 1 |
| 用例43 | SortList | 空表 | 表为空表！ |
| 用例44 | ListInsert | 元素：1 2 4 5  位置3插入8 | 1 2 8 4 5 |
| 用例45 | ListInsert | 元素：1 2 4 5  位置2插入-3 | 1 -3 2 4 5 |
| 用例46 | ListInsert | 元素：1 -3 2 4 5  位置-1插入4 | 插入数据元素失败! |
| 用例47 | SortList | 1 -3 2 4 5 tag:0 | -3 1 2 4 5 |
| 用例48 | SortList | 1 -3 2 4 5 tag:1 | 5 4 2 1 -3 |
| 用例49 | ReverseList | 空表 | 表为空表! |
| 用例50 | ReverseList | 元素：-2 | -2 |
| 用例51 | ReverseList | 元素：-2 4 13 -6 9 | 9 -6 13 4 -2 |
| 用例52 | RemoveSpecificData | 空表 0 | 表为空表! |
| 用例53 | RemoveSpecificData | 元素：2 3 2 6 2 -3 删除2 | 3 6 -3 |
| 用例54 | RemoveSpecificData | 元素：-1 -1 -1  删除-1 | 线性表是空表 |
| 用例55 | RemoveSpecificData | 元素：-1 -2 -3 7  删除2 | 表中无此元素！ |
| 用例56 | MergeList | 表1：表不存在  表2：空表 | 线性表不存在！ |
| 用例57 | MergeList | 表1：空表  表2：空表 | 两个表均为空表 |
| 用例58 | MergeList | 表1：-2 -1 0 1 2  表2：空表 | L1：-2 -1 0 1 2 |
| 用例59 | MergeList | 表1：空表  表2：2 -1 0 | 输入不为升序表！ |
| 用例60 | MergeList | 表1：空表  表2：-5 -3 -1 0 1 | L1：-5 -3 -1 0 1 |
| 用例61 | MergeList | 表1:2 1 3  表2:1 2 3 | 输入不为升序表！ |
| 用例62 | MergeList | 表1：-4 -2 0 2 4  表2：-3 -1 1 3 5 | L1：-4 -3 -2 -1 0 1 2 3 4 5 |
| 用例63 | MergeList | 表1：-45 0 0 2 5  表2：-44 0 2 6 | L1：-45 -44 0 0 0 2 2 5 6 |
| 用例64 | IsPalindromeList | 空表 | 线性表为空！ |
| 用例65 | IsPalindromeList | 元素：-1 | 它是回文链表 |
| 用例66 | IsPalindromeList | 元素：-1 -2 0 -1 -2 | 它不是回文链表 |
| 用例67 | IsPalindromeList | 元素：7 32 -1 9 9 -1 32 7 | 它是回文链表 |
| 用例68 | ListAddition | 表1：空表  表2：0 2 3 | 存在空表，无法相加！ |
| 用例69 | ListAddition | 表1:0 0 0  表2:0 2 3 0 | 表1：2 0 3 0 |
| 用例70 | ListAddition | 表1：0 0 9 9 9  表2：0 0 0 9 9 9 | 表1：1 9 9 8 |
| 用例71 | ListAddition | 表1：1 9 9 8  表2：空表 | 存在空表，无法相加！ |
| 用例72 | ListAddition | 表1：1 5 0 0 6 9 4 5 6 3  表2：0 0 4 8 9 9 9 9 9 9 | 表1：1 5 4 9 6 9 4 5 6 2 |

输出界面

|  |  |
| --- | --- |
| 测试用例 | 运行结果 |
| 用例1 |  |
| 用例2 |  |
| 用例3 |  |
| 用例4 |  |
| 用例5 |  |
| 用例6 |  |
| 用例7 |  |
| 用例8 |  |
| 用例9 |  |
| 用例10 |  |
| 用例11 |  |
| 用例12 |  |
| 用例13 |  |
| 用例14 |  |
| 用例15 |  |
| 用例16 |  |
| 用例17 |  |
| 用例18 |  |
| 用例19 |  |
| 用例20 |  |
| 用例21 |  |
| 用例22 |  |
| 用例23 |  |
| 用例24 |  |
| 用例25 |  |
| 用例26 |  |
| 用例27 |  |
| 用例28 |  |
| 用例29 |  |
| 用例30 |  |
| 用例31 |  |
| 用例32 |  |
| 用例33 |  |
| 用例34 |  |
| 用例35 |  |
| 用例36 |  |
| 用例37 |  |
| 用例38 |  |
| 用例39 |  |
| 用例40 |  |
| 用例41 |  |
| 用例42 |  |
| 用例43 |  |
| 用例44 |  |
| 用例45 |  |
| 用例46 |  |
| 用例47 |  |
| 用例48 |  |
| 用例49 |  |
| 用例50 |  |
| 用例51 |  |
| 用例52 |  |
| 用例53 |  |
| 用例54 |  |
| 用例55 |  |
| 用例56 |  |
| 用例57 |  |
| 用例58 |  |
| 用例59 |  |
| 用例60 |  |
| 用例61 |  |
| 用例62 |  |
| 用例63 |  |
| 用例64 |  |
| 用例65 |  |
| 用例66 |  |
| 用例67 |  |
| 用例68 |  |
| 用例69 |  |
| 用例70 |  |
| 用例71 |  |
| 用例72 |  |

**2.4 实验小结**

在链式存储的线性表中，对于元素的插入与删除操作要简便，只需要找到对应位置然后对单个结点进行操作，而不需要像数组存储那样令元素整体移动，但在对线性表整体进行操作的过程则显得比较繁琐，例如求表长需要遍历表计数，清空表和销毁表需要遍历表并逐个释放每个结点的空间；同时实验过程中发现链式存储的线性表可以进行更多样的操作，以及可以用不同的思路实现前一个实验中的相同功能，在判断回文链表时就是用了递归操作结点的方式去判断首尾结点的值是否相等，而不是最容易操作的先将数据元素全部记录入数组再进行判断；受课件中使用链表的结点存储多项式的系数与幂次方的启发，实现了一个用链表存储大整数各个位数并求出两数和的功能。两次实验感觉链式存储的实现需要更多的判定操作，需要考虑的边界情况更多，对我们的细心程度的要求更高，同时由于单向链表实现的局限性，很多情况下需要多指针进行辅助操作。通过这次实验，我对链式存储的线性表的操作有了更深的了解，完成了基本功能的基础上进行了多功能的思考，收获很大。