

**课 程 实 验 报 告**

**课程名称： 数据结构实验**

**专业班级： 计算机科学与技术201802**

**学 号： U201814531**

**姓 名： 李响**

**指导教师： 李丹**

**报告日期： 2019年 11月**

**计算机科学与技术学院**

目 录

[1 基于顺序存储结构的线性表实现 3](#_Toc26902650)

[1.1 问题描述 3](#_Toc26902651)

[1.1.1 线性表的基本概念与逻辑结构与基本运算 3](#_Toc26902652)

[1.1.2 线性表的基本运算 3](#_Toc26902653)

[1.2 系统设计 5](#_Toc26902654)

[1.2.1 数据存储结构与形式 5](#_Toc26902655)

[1.2.2 总体系统设计 5](#_Toc26902656)

[1.3 系统实现 6](#_Toc26902657)

[1.3.1 编程环境与运行环境描述 6](#_Toc26902658)

[1.3.2 头文件及预定义常量说明 6](#_Toc26902659)

[1.3.3 算法设计与实现 7](#_Toc26902660)

[1.3.4 算法复杂度分析 10](#_Toc26902661)

[1.4 系统测试 11](#_Toc26902662)

[1.5 实验小结 17](#_Toc26902663)

[2 基于链式存储结构的线性表实现 18](#_Toc26902664)

[2.1 问题描述 18](#_Toc26902665)

[2.1.1 线性表的基本概念与逻辑结构与基本运算 18](#_Toc26902666)

[2.1.2 线性表的基本运算 18](#_Toc26902667)

[2.2 系统设计 20](#_Toc26902668)

[2.2.1 数据存储结构与形式 20](#_Toc26902669)

[2.2.2 总体系统设计 20](#_Toc26902670)

[2.3 系统实现 21](#_Toc26902671)

[2.3.1 编程环境与运行环境描述 21](#_Toc26902672)

[2.3.2 头文件及预定义常量说明 21](#_Toc26902673)

[2.3.3 算法设计与实现 22](#_Toc26902674)

[2.4 系统测试 27](#_Toc26902675)

[2.5 实验小结 32](#_Toc26902676)

[3基于二叉链表的二叉树实现 34](#_Toc26902677)

[3.1 问题描述 34](#_Toc26902678)

[3.1.1 二叉树的基本概念与逻辑结构 34](#_Toc26902679)

[3.1.2 二叉树的基本运算 35](#_Toc26902680)

[3.2 系统设计 36](#_Toc26902681)

[3.2.1 数据存储结构与形式 36](#_Toc26902682)

[3.2.2 总体系统设计 38](#_Toc26902683)

[3.3 系统实现 39](#_Toc26902684)

[3.3.1 编程环境与运行环境描述 39](#_Toc26902685)

[3.3.2 头文件及预定义常量说明 39](#_Toc26902686)

[3.3.3 算法设计与实现 40](#_Toc26902687)

[3.4 系统测试 47](#_Toc26902688)

[3.5 实验小结 54](#_Toc26902689)

[4基于邻接表的图实现 56](#_Toc26902690)

[4.1 问题描述 56](#_Toc26902691)

[4.1.1 图的基本概念与逻辑结构 56](#_Toc26902692)

[4.1.2 图的基本运算 56](#_Toc26902693)

[4.2 系统设计 58](#_Toc26902694)

[4.2.1 数据存储结构与形式 58](#_Toc26902695)

[4.2.2 总体系统设计 60](#_Toc26902696)

[4.3 系统实现 60](#_Toc26902697)

[4.3.1 编程环境与运行环境描述 60](#_Toc26902698)

[4.3.2 头文件及预定义常量说明 61](#_Toc26902699)

[4.3.3 算法设计与实现 61](#_Toc26902700)

[4.4 系统测试 67](#_Toc26902701)

[4.5 实验小结 71](#_Toc26902702)

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

采用顺序表作为物理结构，构造一个具有菜单的功能演示系统。其中，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。演示系统实现多线性表管理，可进行多表操作。

设计线性表文件保存和加载操作合理模式，即将生成的线性表存入到文件中，也可以从文件中获取线性表进行操作；并设计文件数据记录格式，以高效保存线性表数据逻辑结构的完整结构以及信息。

### 1.1.1 线性表的基本概念与逻辑结构与基本运算

线性表是最常用且最简单的一种数据结构，即n个数据元素的有限序列。线性表中元素的个数n定义为线性表的长度，n=0时成为空表。在非空表中的每个数据元素都有一个确定的位置。线性表的存储结构分为线性存储和链式存储。本次实验需要实现的是基于线性存储的线性表。

线性表的数据逻辑结构定义如下:

ADT List｛

数据对象：D=｛ai|ai∈ElemSet，i=1，2，„„，n，n≥0｝

数据关系：R1=｛ <ai-1，ai> | ai-1，ai∈D，i=2，„„，n｝

｝

### 1.1.2 线性表的基本运算

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的12种基本运算，具体运算功能定义如下：

（1）初始化表InitList(L)；

初始条件：线性表L不存在。

操作结果：构造一个空的线性表。

（2）销毁表DestroyList(L)；

初始条件：线性表L已存在。

操作结果：销毁线性表L。

（3）清空表ClearList(L)；

初始条件：线性表L已存在。

操作结果：将L重置为空表。

（4）判定空表ListEmpty(L)；

初始条件：线性表L已存在。

操作结果：若L为空表则返回TRUE，否则返回FALSE。

（5）求表长ListLength(L)；

初始条件：线性表L已存在。

操作结果：返回L中数据元素的个数。

（6）获得元素GetElem(L,i,e)；

初始条件：线性表L已存在，且1≤i≤L.Length。

操作结果：用e返回L中第i个数据元素的值。

（7）查找元素LocateElem(L,e)；

初始条件：线性表L已存在。

操作结果：返回线性表L中第1个与e相同的数据元素的位序；若这样的数

据元素不存在，则返回值为0。

（8）获得前驱PriorElem(L,cur\_e,pre\_e)；

初始条件：线性表L已存在。

操作结果：若cur\_e是L的数据元素，且不是第一个，则用pre\_e返回它的前驱，否则操作失败，pre\_e无定义。

（9）获得后继NextElem(L,cur\_e,next\_e)；

初始条件：线性表L已存在。

操作结果：若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。

（10）插入元素ListInsert(L,i,e)；

初始条件：线性表L已存在，且1≤i≤L.Length+1；

操作结果：在L第i个位置之前插入新的数据元素e。

（11）删除元素ListDelete(L,i,e)；

初始条件：线性表L已存在，线性表非空，且1≤i≤L.Length；

操作结果：删除L第i个数据元素，用e返回其值。

（12）遍历表ListTraverse(L)；

初始条件：线性表L已存在；

操作结果：依次对L的每个数据元素进行输出。

## 1.2 系统设计

### 1.2.1 数据存储结构与形式

线性表的数据物理结构如下：

typedef struct{

ElemType \* elem; //定义元素类型指针，存储空间基址

int length; //线性表的长度

int listsize; //当前分配的存储容量

char name[20]; //线性表的名称

}SqList;

PS.若要实现同时对多个线性表管理，只需要定义一个线性表的结构数组，通过线性表的名称属性对线性表进行操作。

### 1.2.2 总体系统设计

通过WHILE循环与菜单界面，用户通过选择菜单中的选项实现交互，使用OP变量获取用户选择选项值（OP初始化值为1，以便第一次能进入循环）。

进入循环后系统首先显示功能菜单，然后用户输入选择0-14，其中1-12分别代表线性表的一个基本运算，13与14选项分别是线性表的文件保存，以及显示全部线性表信息（**本选项用于方便查看在存在多个线性表的情况下，各个线性表的长度与名称信息**），在主函数中通过SWITCH语句对应到相应的函数功能，执行完该功能后BREAK跳出SWITCH语句，继续执行WHILE循环，直至用户输入0退出当前菜单系统。

在进行线性表的操作时，除了1和14选项（1选项是进行线性表的生成和初始化，14选项用于查看所有线性表信息），系统会要求输入需要处理的线性表的名称（本系统多线性表操作以线性表名称作为查找特征），若线性表不存在则会返回菜单界面重新操作（系统结构如图1-1所示）。

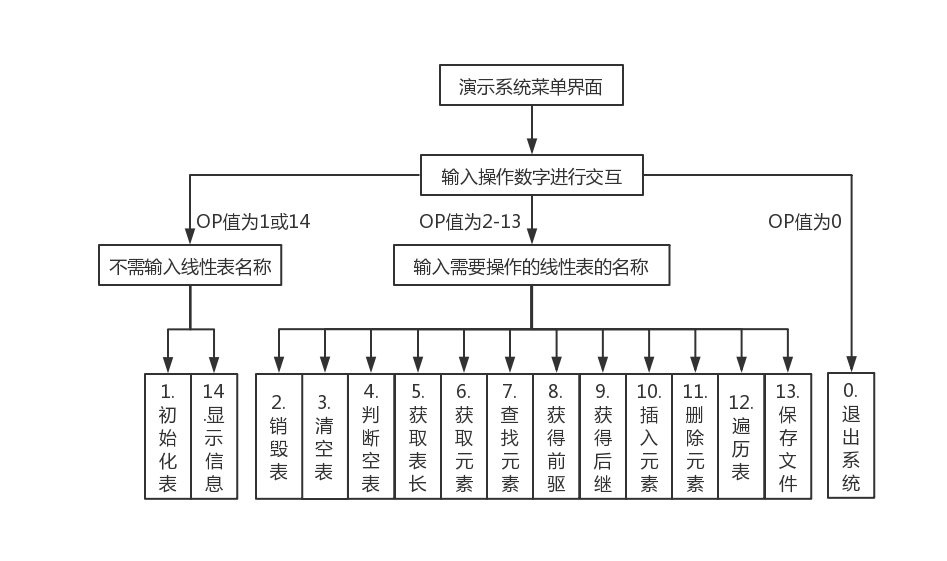


图1-1 系统菜单操作示意图

## 1.3 系统实现

### 1.3.1 编程环境与运行环境描述

编程环境：采用CodeBlocks 16.01编程软件编写。

运行环境：微软Windows 10系统。

### 1.3.2 头文件及预定义常量说明

1.头文件

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

2.预定义常量

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

3.类型表达式

typedef int status;

typedef int ElemType;

### 1.3.3 算法设计与实现

（1）初始化表status InitList(SqList &L)；

算法实现：使用malloc函数分配连续内存空间，将首地址返回赋值给L.elem，将L.length初始化为0，L.listsize初始化为LIST\_INIT\_SIZE。

1. 销毁表status DestroyList(SqList &L)；

算法实现：使用free函数释放掉以L.elem为首地址的连续内存空间，并将L.length，L.listsize重新赋值为0，将L.name清空。

1. 清空表status ClearList(SqList &L)；

算法实现：若线性表不存在返回ERROR，若线性表存在，则将线性表的长度L.length的值置为0。

1. 判定空表status ListEmpty(SqList L)；

算法实现：若线性表不存在，返回-1，若线性表存在则读取线性表L.length，若其值为0则返回FALSE，否则返回TRUE。

1. 求表长int ListLength(SqList L)；

算法实现：若线性表不存在，返回-1，若线性表存在则直接返回L.length即为所求线性表的表长。

1. 获得元素status GetElem(SqList L,int i,ElemType &e)；

算法实现：若1≤i≤L.Length则通过随机访问数组的方式获取元素，将L.elem[i-1]的值赋值给e；否则返回ERROR。

1. 查找元素status LocateElem(SqList L,ElemType e)；

算法实现：通过循环遍历线性表，将每一个元素与给定值比较看是否相等，如果相等就返回该元素的次序，否则返回0。

1. 获得前驱status PriorElem(SqList L,ElemType cur,ElemType &pre\_e)；

算法实现：若cur的值和L.elem[0]相同，则返回-1；若遍历线性表时cur与L.elem[i]相等，则将L.elem[i-1]赋值给pre\_e；若无元素符合，则ERROR（获得前驱函数流程图，如图1-2所示）。

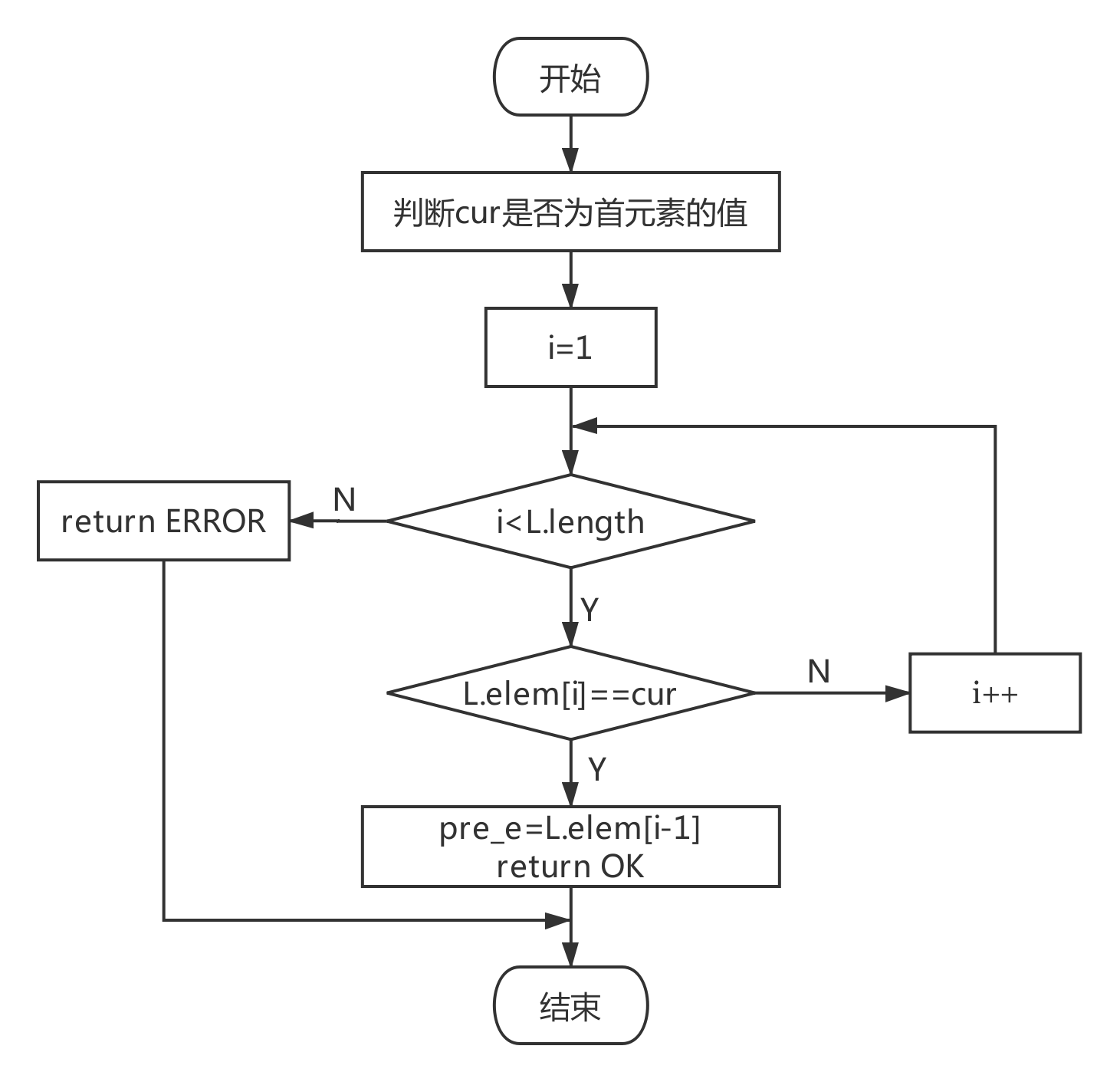


图1-2 获得前驱函数流程图

1. 获得后继status NextElem(SqList L,ElemType cur,ElemType &next\_e)；

算法实现：通过循环遍历线性表，查找前L.length-1个元素是否符合，若符合则将其后继赋值给next\_e，然后判断是否为最后一个元素，若是的话返回-1，否则返回ERROR。

1. 插入元素status ListInsert(SqList &L,int i,ElemType e)；

算法实现：首先判断插入位置的合法性，若1≤i≤L.Length+1则继续，否则返回ERROR；然后判断当前存储空间是否已满，若满了则用realloc函数重新分配空间，插入元素时，从最后一个元素开始向前遍历，每个元素以此往后移一个单元直到插入点，最后插入e，表长L.length数值增加1（插入元素函数流程图，如图1-3所示）。

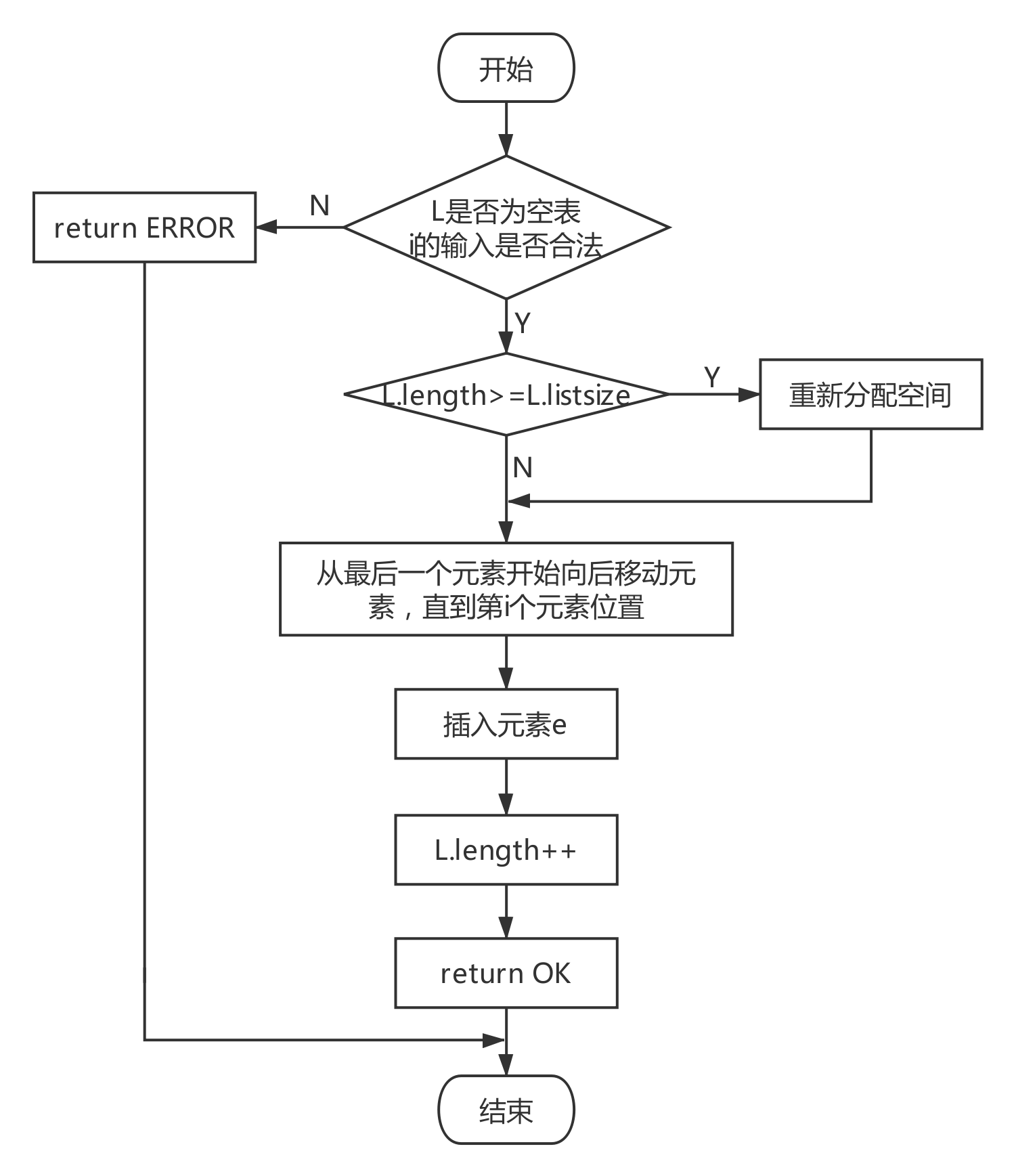


图1-3 插入元素函数流程图

1. 删除元素status ListDelete(SqList &L,int i,ElemType &e)；

算法实现：首先判断位序的合法性，并判断线性表是否为空表，即若1≤i≤L.Length且L.length不为0则继续，否则返回ERROR；在之后直接将删除元素位置后一个元素直到最后一个元素以此从前往后向前移动一个单元。

1. 遍历表status ListTrabverse(SqList L)；

算法实现：采用循环遍历线性表，输出线性表中的每一个元素。

1. 读取文件status OpenFile(SqList &L)；

算法实现：使用fopen函数打开指定文件，然后使用fscanf函数读取文件的数据，并将数据赋值给线性表，同时线性表的长度依次增加；若空间不足则用realloc函数重新分配空间，最后关闭文件。

1. 键盘赋值status SetValue(SqList &L)；

算法实现：在键盘上输入数据，数据集以Ctrl+Z为结束标志，然后使用scanf函数从缓冲区中读取数据，赋值给线性表，同时线性表的长度依次增加；若空间不足则用realloc函数重新分配空间。

1. 文件保存status SaveFile(SqList &L)；

算法实现：首先，选择读写文件的方式，选择1选项不删除原有的线性表，在原有数据之后添加数据，或者选择2选项删除原有线性表数据或生成新文件；然后使用fopen函数创建或打开文件，然后用fputs函数和fprintf函数输出数据，最后关闭文件（文件保存函数流程图，如图1-4所示）。

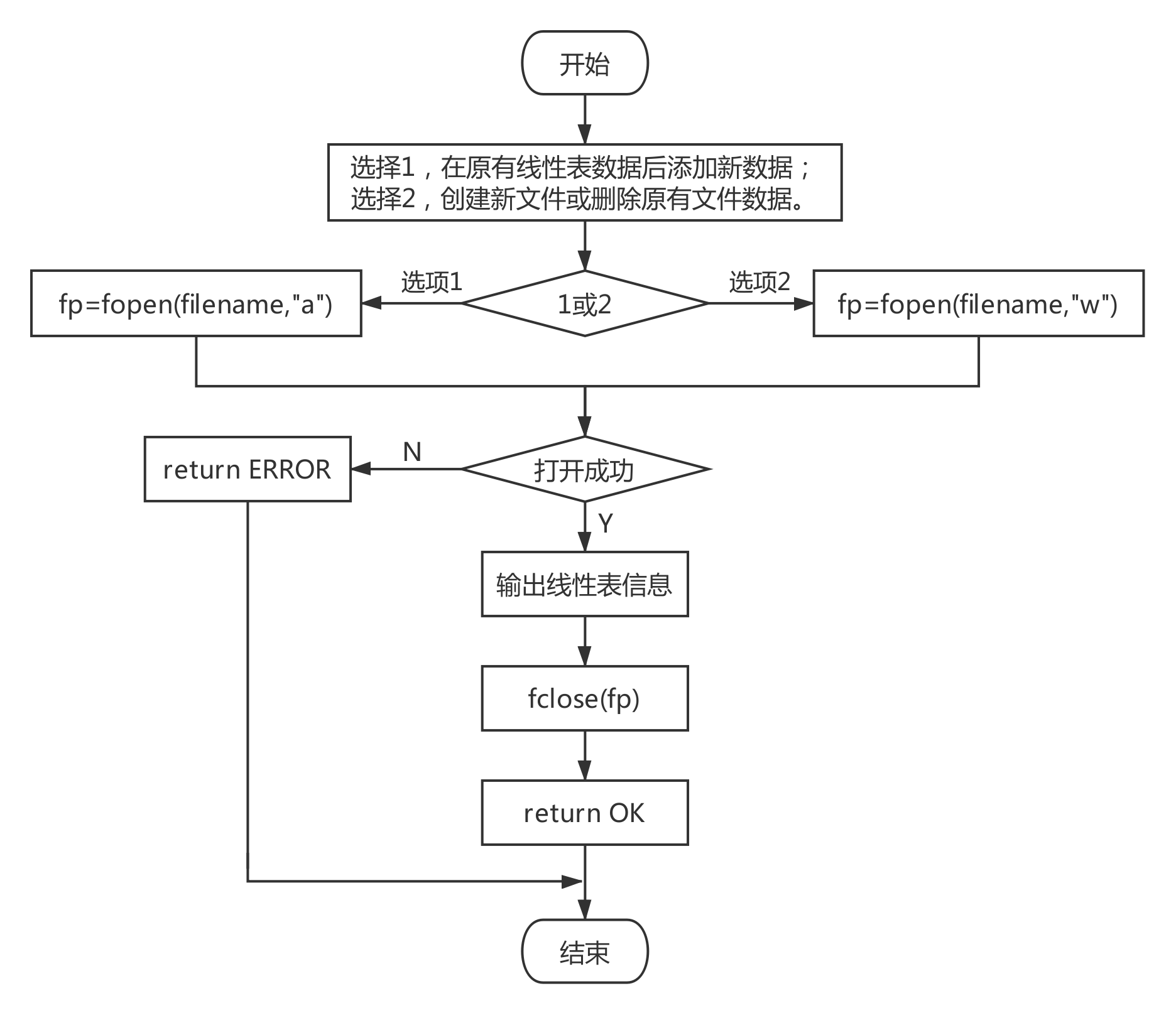


图1-4 文件保存函数流程图

### 1.3.4 算法复杂度分析

通过算法的分析与计算，确定了线性表的12个基本操作函数与自定义的三个文件处理与赋值函数的时间复杂度与空间复杂度（算法时间与空间复杂度表，如表1-1所示）。

表1-1 算法时间与空间复杂度表

|  |  |  |
| --- | --- | --- |
| 函数名称 | 时间复杂度T(n) | 空间复杂度S(n) |
| InitialList | O(1) | O(1) |
| DestroyList | O(1) | O(1) |
| ClearList | O(1) | O(1) |
| ListEmpty | O(1) | O(1) |
| ListLength | O(1) | O(1) |
| GetElem | O(n) | O(1) |
| LocateElem | O(n) | O(1) |
| PriorElem | O(N) | O(1) |
| NextElem | O(N) | O(1) |
| ListInsert | O(n) | O(1) |
| ListDelete | O(n) | O(1) |
| ListTrabverse | O(n) | O(1) |
| OpenFile | O(n) | O(1) |
| SetValue | O(n) | O(1) |
| SaveFile | O(n) | O(1) |

## 1.4 系统测试

1. 当程序开始运行时，会立刻进入菜单演示界面（菜单界面如图1-5所示）。

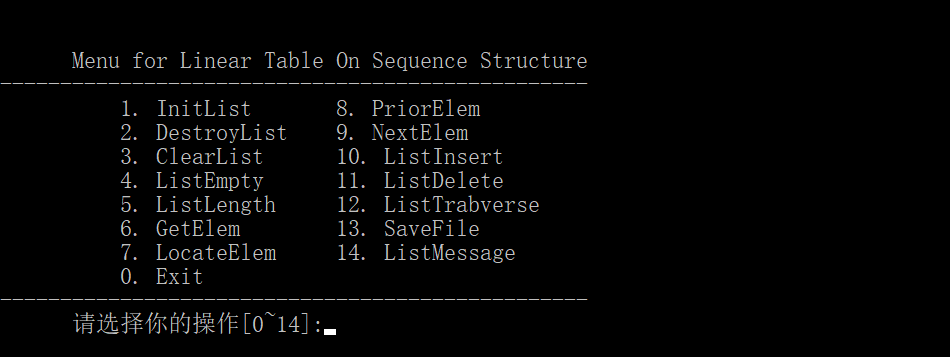


图1-5 演示系统菜单界面图

1. 按照菜单的提示输入需要完成的操作的序号，程序就会进入相对应的操作过程；当系统中没有线性表存在时，进行操作2-13，系统会做出提示显示“当前无线性表存在，请先创建线性表”（操作结果图如图1-6所示）。

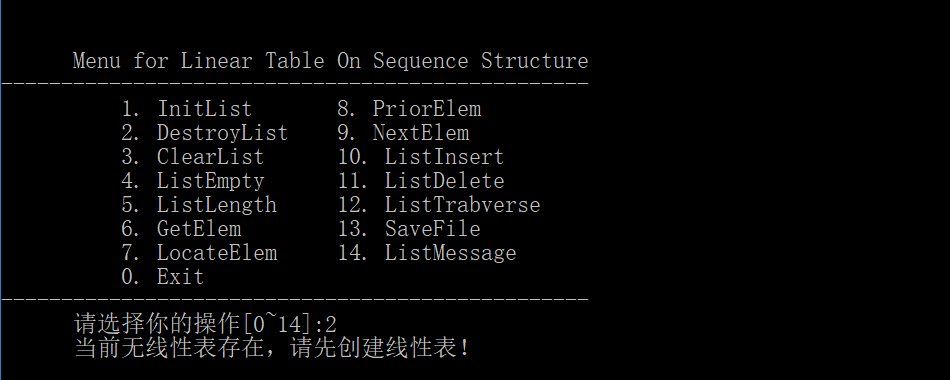


图1-6 系统无线性表存在时进行2-13操作结果图

1. 当系统中存在线性表时，可以进行所有操作，在进行2-13操作时，需要输入线性表的名称确定需要操作的线性表（操作界面图如图1-7所示）。

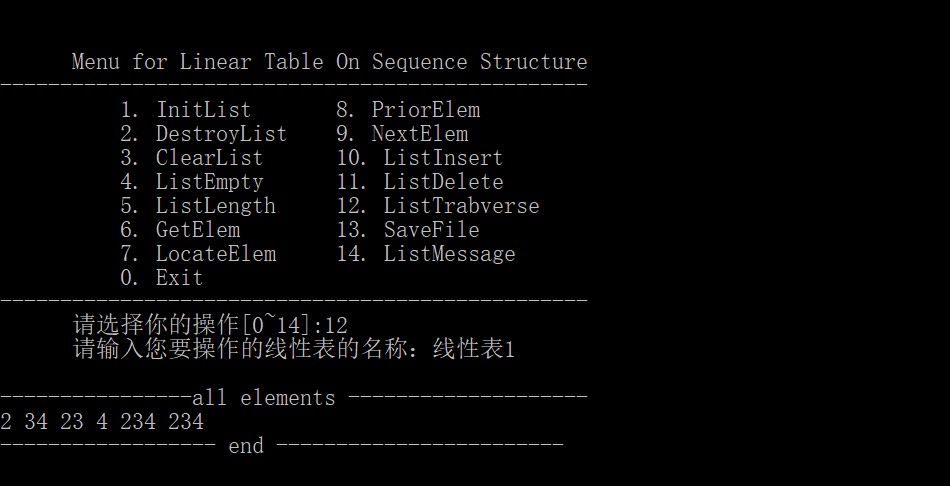


图1-7 输入线性表名称进行操作界面图

1. 当需要进行多表操作时，若需要查看所有线性表的信息，则可使用选项14操作，显示线性表信息（操作结果图如图1-8所示）。

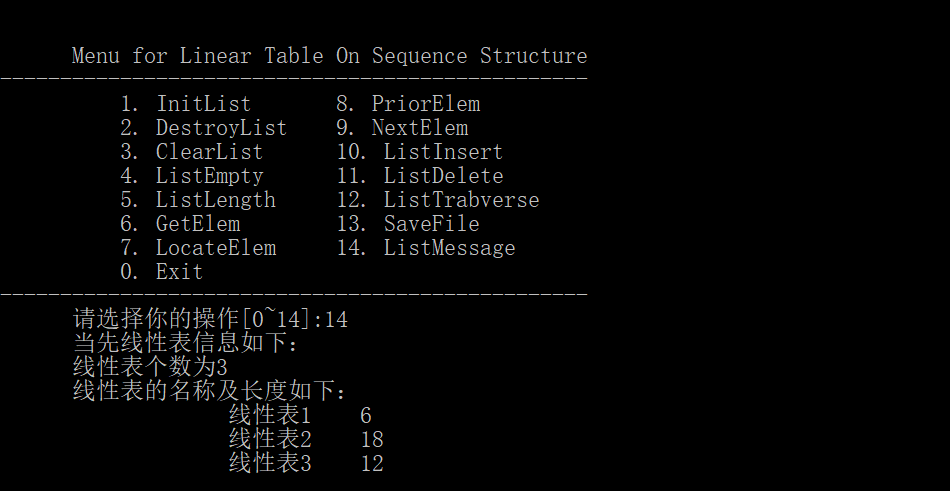


图1-8 显示多线性表信息操作结果图

1. 系统测试计划

使用预先保存的测试样例进行测试，本次测试挑选DestroyList、LocateElem、PriorElem、ListInsert、SaveFile共5个函数进行测试。本次使用的测试样例为多表测试样例，样例中包含4个线性表，其信息如下：

List1（表中包含5,12,0,44,94,137,47,3,9）；

List2（表中包含2,5,7,9,13,15,23）；

List3（空表）；

List4（表中包含4,23,5,33,7,23,5,7,77）；

其中，List1、List2与List3为键盘赋值（键盘赋值操作图如图1-9所示），List4为文件赋值（文件赋值操作图如图1-10所示）。

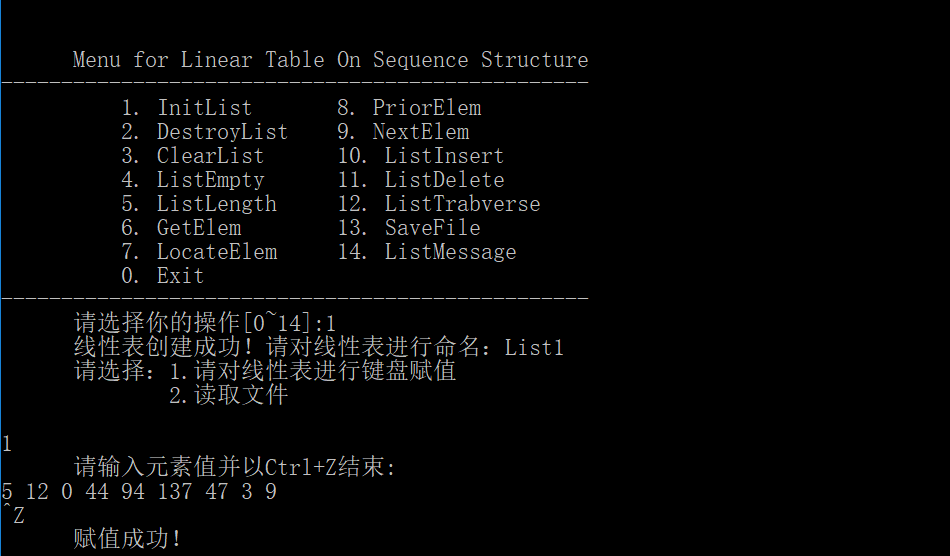


图1-9 线性表通过键盘赋值结果图

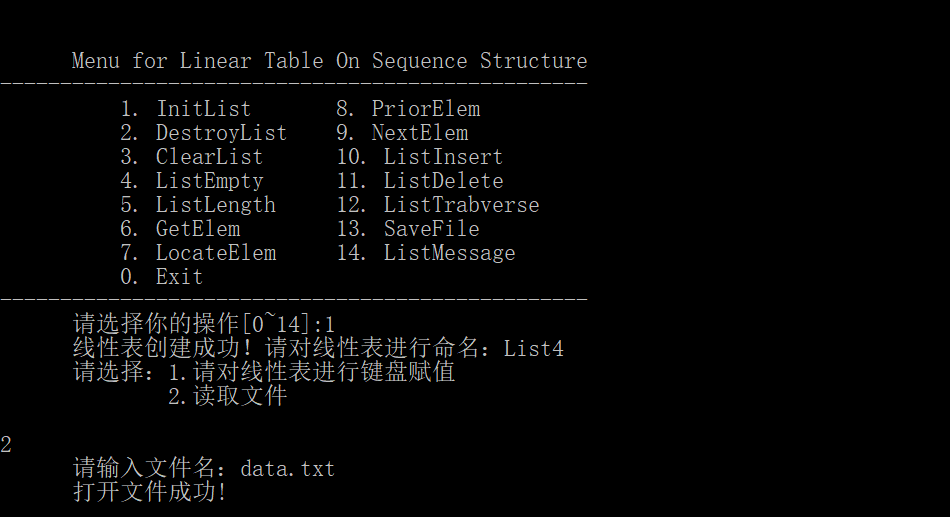
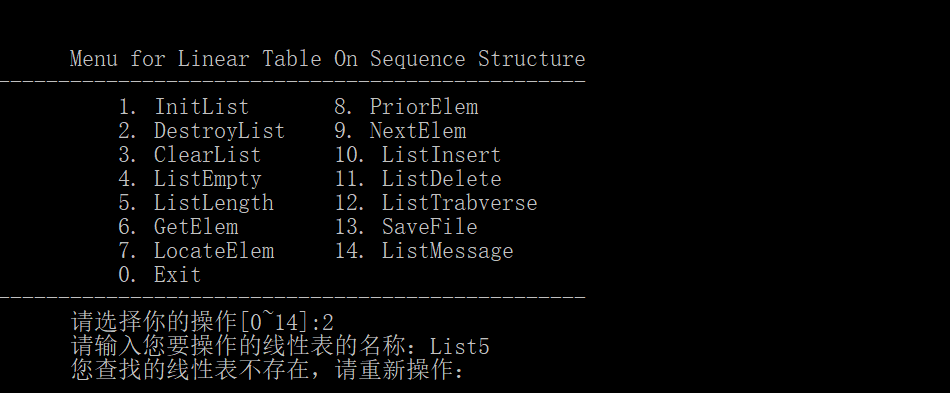


图1-10 线性表通过文件赋值结果图

1. DestroyList函数测试：

表1-2 DestroyList函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| 无 | 1. 主界面输入2进入操作； 2. 按提示输入“List5”作为需要销毁的线性表，此线性表不存在。 | 线性表不存在 | （失败结果图如图1-11所示） |
| List2 | 1. 主界面输入2进入操作； 2. 按提示输入“List2”作为需要销毁的线性表。 | 线性表销毁成功！ | List2线性表删除成功（结果图如图1-12所示） |

图1-11 线性表不存在时显示销毁失败结果图

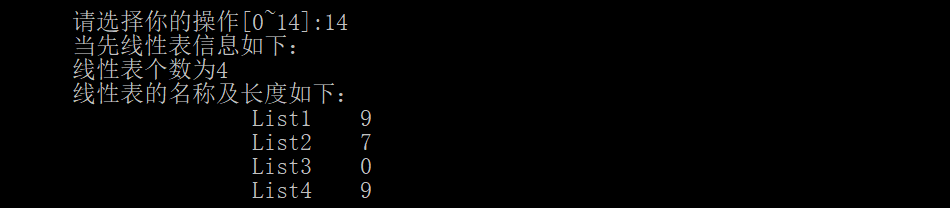
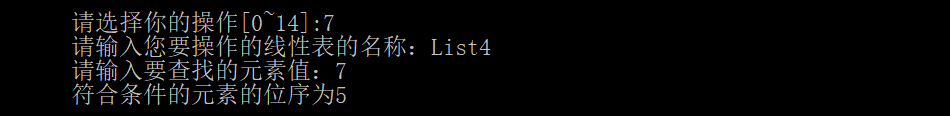
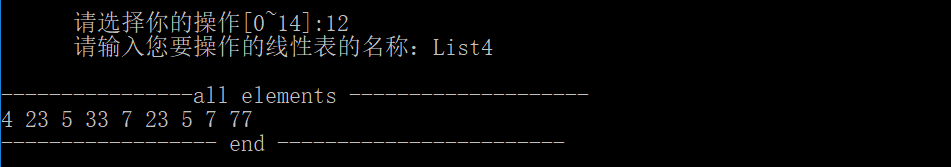


图1-12 线性表销毁前与销毁成功结果对比图

1. LocateElem函数测试：

表1-3 LocateElem函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List4 | 1. 主界面输入7进入操作； 2. 按提示输入“List4”，按照提示输入“7”作为需要查找的元素。 | 符合条件的元素的位序为5 | （输出结果图如图1-13所示） |
| List4 | 1. 主界面输入7进入操作； 2. 按提示输入“List4”，按照提示输入“6”作为需要查找的元素。 | 没有符合条件的元素存在 | （输出结果图如图1-13所示） |



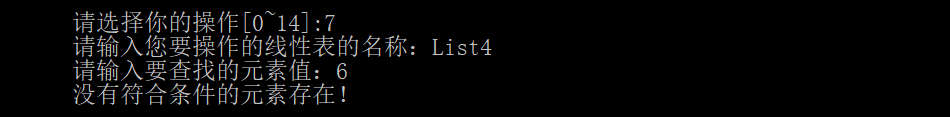
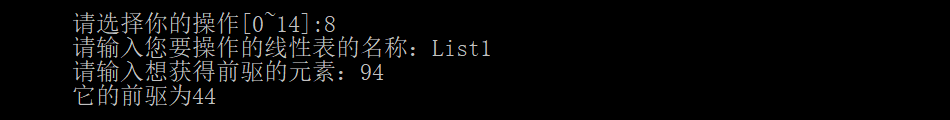
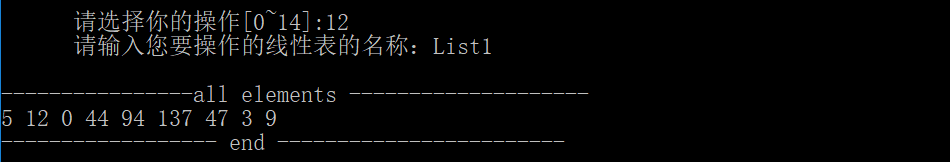


图1-13 查找元素函数测试结果图

1. PriorElem函数测试：

表1-4 PriorElem函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List1 | 1. 主界面输入8进入操作； 2. 按提示输入“List1”，按照提示输入“94”作为需要查找前驱的元素。 | 它的前驱为44 | （输出结果图如图1-14所示） |
| List1 | 1. 主界面输入7进入操作； 2. 按提示输入“List1”，按照提示输入“5”作为需要查找前驱的元素。 | 该元素不存在前驱 | （输出结果图如图1-14所示） |



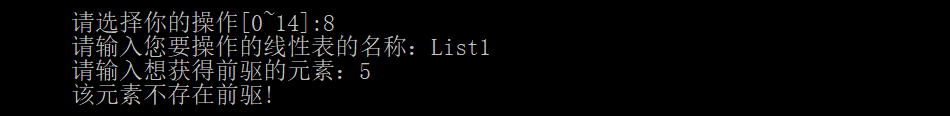


图1-14 查找元素前驱函数测试结果图

1. ListInsert函数测试：

表1-5 ListInsert函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List1 | 1. 主界面输入10进入操作； 2. 按提示输入“List1”，输入“10”作为插入位点，“15”为插入元素。 | 操作成功 | （输出结果图如图1-15所示） |
| List1 | 1. 主界面输入10进入操作； 2. 按提示输入“List1”，输入“5”作为插入位点，“19”为插入元素。 | 操作成功 | （输出结果图如图1-15所示） |

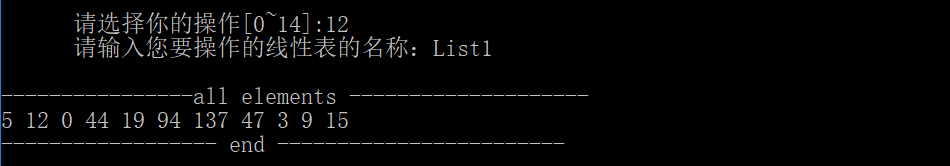
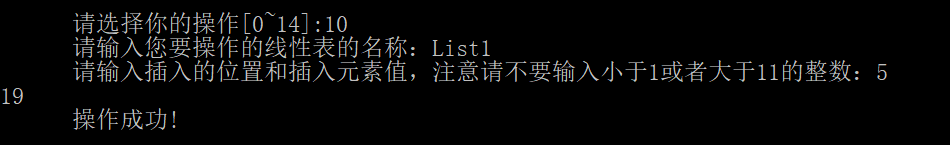
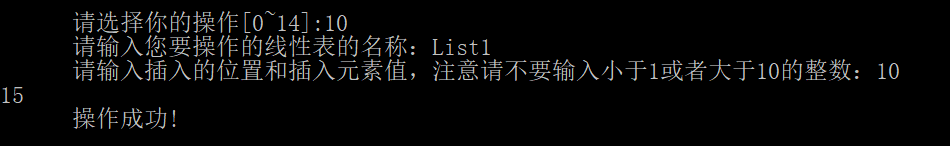
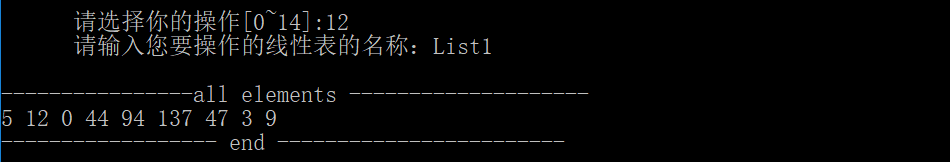
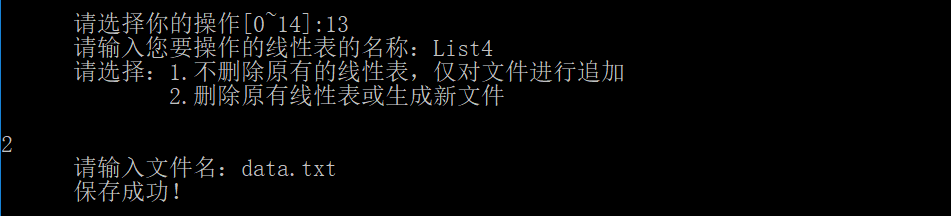


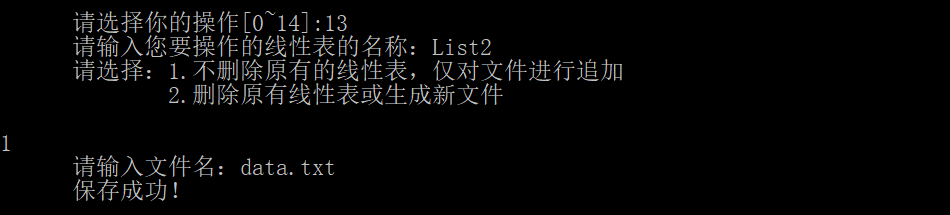
图1-15 插入元素函数结果图

1. SaveFile函数测试：

表1-6 SaveFile函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List4 | 1. 主界面输入13进入操作； 2. 按提示输入“List4”，按照提示输入“2”创建新文件进行文件保存。 3. 输入文件名“data.txt” | 保存成功 | （输出结果图如图1-16所示） |
| List2 | 1. 主界面输入13进入操作； 2. 按提示输入“List4”，按照提示输入“1”在旧文件后添加数据。 3. 输入文件名“data.txt” | 保存成功 | （输出结果图如图1-16所示） |





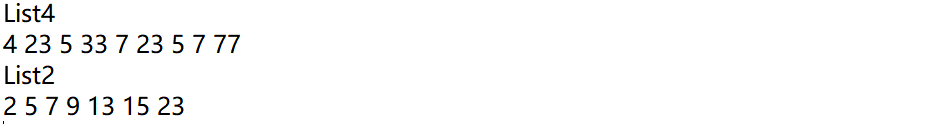


图1-16 文件保存函数测试结果图

## 1.5 实验小结

在本次实验中，我使用的语言为C语言，但是为了传递数据的方便，特意学习和使用了引用传递（C++语言中较为常用）作为传递参数的手段。通过代码的设计和编写完成了线性表的顺序存储。在编写过程中，我充分地了解到了线性表的顺序物理存储结构的编写过程和实现原理，并且熟练掌握了线性表的基本操作。

在设计算法和编写代码的过程，我遇到了一些问题，但是在同学的帮助和自身的努力下，最终我克服了问题，顺利地完成了本次实验的代码实现。其中，最困扰我的问题是关于文件的处理和多表操作，关于文件处理和多表操作，具体实现我在算法设计里有详细描述，这里就不赘述了，不过关于多表操作，为了方便查看多表信息和进行多表管理，我特意设计了14选项用于显示多表信息，以便于操作，这个功能设计是我比较满意的。除此之外，C语言课程结课较早，导致我的代码编写有些生疏，不过在几天的编写过程中，我已经逐渐找回了编代码的感觉，代码编写也变得流畅不少。

通过本次实验，我发现了我自身的一些不足，以及对C语言本身某些操作的生疏，除此之外，单独学习了引用传递的使用，加强了对“&”引用符号的理解。

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

采用单链表作为线性表的物理结构，构造一个具有菜单的功能演示系统。其中，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。演示系统可选择实现多个线性表管理。

设计线性表文件保存和加载操作合理模式，即将生成的线性表存入到文件中，也可以从文件中获取线性表进行操作；并设计文件数据记录格式，以高效保存线性表数据逻辑结构的完整结构以及信息。

### 2.1.1 线性表的基本概念与逻辑结构与基本运算

线性表是最常用且最简单的一种数据结构，即n个数据元素的有限序列。线性表中元素的个数n定义为线性表的长度，n=0时成为空表。在非空表中的每个数据元素都有一个确定的位置。线性表的存储结构分为线性存储和链式存储。本次实验需要实现的是基于链式存储的线性表。

线性表的数据逻辑结构定义如下:

ADT List｛

数据对象：D=｛ai|ai∈ElemSet，i=1，2，„„，n，n≥0｝

数据关系：R1=｛ <ai-1，ai> | ai-1，ai∈D，i=2，„„，n｝

｝

### 2.1.2 线性表的基本运算

依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的12种基本运算，具体运算功能定义如下：

（1）初始化表InitList(L)；

初始条件：线性表L不存在。

操作结果：构造一个空的线性表。

（2）销毁表DestroyList(L)；

初始条件：线性表L已存在。

操作结果：销毁线性表L。

（3）清空表ClearList(L)；

初始条件：线性表L已存在。

操作结果：将L重置为空表。

（4）判定空表ListEmpty(L)；

初始条件：线性表L已存在。

操作结果：若L为空表则返回TRUE，否则返回FALSE。

（5）求表长ListLength(L)；

初始条件：线性表L已存在。

操作结果：返回L中数据元素的个数。

（6）获得元素GetElem(L,i,e)；

初始条件：线性表L已存在，且1≤i≤L.Length。

操作结果：用e返回L中第i个数据元素的值。

（7）查找元素LocateElem(L,e)；

初始条件：线性表L已存在。

操作结果：返回线性表L中第1个与e相同的数据元素的位序；若这样的数

据元素不存在，则返回值为0。

（8）获得前驱PriorElem(L,cur\_e,pre\_e)；

初始条件：线性表L已存在。

操作结果：若cur\_e是L的数据元素，且不是第一个，则用pre\_e返回它的前驱，否则操作失败，pre\_e无定义。

（9）获得后继NextElem(L,cur\_e,next\_e)；

初始条件：线性表L已存在。

操作结果：若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。

（10）插入元素ListInsert(L,i,e)；

初始条件：线性表L已存在，且1≤i≤L.Length+1；

操作结果：在L第i个位置之前插入新的数据元素e。

（11）删除元素ListDelete(L,i,e)；

初始条件：线性表L已存在，线性表非空，且1≤i≤L.Length；

操作结果：删除L第i个数据元素，用e返回其值。

（12）遍历表ListTraverse(L)；

初始条件：线性表L已存在；

操作结果：依次对L的每个数据元素进行输出。

## 2.2 系统设计

### 2.2.1 数据存储结构与形式

基于单链表的线性表的数据物理结构如下：

typedef struct LNode{ //线性表（链式结构结点）的定义

ElemType data;

struct LNode \*next; //存储线性表名称

}LNode,\*LinkList; //结点结构名以及结点指针结构名

若要实现同时对多个线性表管理，需要定义一个头指针和线性表名称的结构以及结构数组，通过线性表的名称属性对线性表进行操作，结构体定义如下：

typedef struct{

LinkList head;

char name[30];

}LkList; //多表操作时，存储线性表的名称信息以及头指针

### 2.2.2 总体系统设计

菜单设计与第一次基本相同，所以将参考第一次的内容。通过WHILE循环与菜单界面，用户通过选择菜单中的选项实现交互，使用OP变量获取用户选择选项值（OP初始化值为1，以便第一次能进入循环）。

进入循环后系统首先显示功能菜单，然后用户输入选择0-14，其中1-12分别代表线性表的一个基本运算，13与14选项分别是线性表的文件保存，以及显示全部线性表信息（**本选项用于方便查看在存在多个线性表的情况下，各个线性表的长度与名称信息**），在主函数中通过SWITCH语句对应到相应的函数功能，执行完该功能后BREAK跳出SWITCH语句，继续执行WHILE循环，直至用户输入0退出当前系统。

在进行线性表的操作时，除了1和14选项（1选项是进行线性表的生成和初始化，14选项用于查看所有线性表信息），系统会要求输入需要处理的线性表的名称（本系统多线性表操作以线性表名称作为查找特征），若线性表不存在表会返回菜单界面重新操作（系统结构如图2-1所示）。

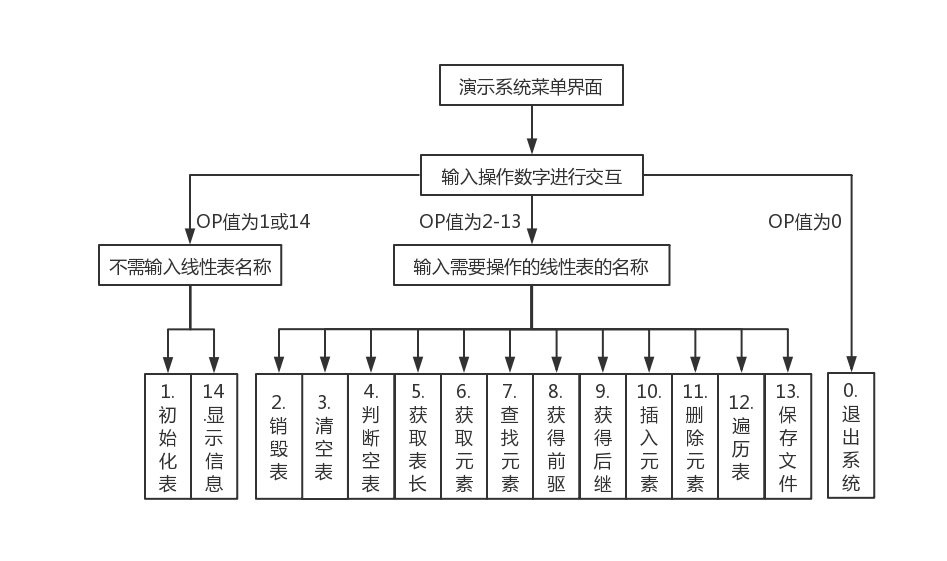


图2-1 系统菜单操作示意图

## 2.3 系统实现

### 2.3.1 编程环境与运行环境描述

编程环境：采用CodeBlocks 16.01编程软件编写。

运行环境：微软Windows 10系统。

### 2.3.2 头文件及预定义常量说明

1.头文件

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

2.预定义常量

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

3.类型表达式

typedef int status;

typedef int ElemType;

### 2.3.3 算法设计与实现

为方便算法的实现和多线性表管理与操作，本次算法设计与实现采用了带头结点的链表，下列所有算法的设计与实现都遵循这个规定。

（1）初始化表status InitList(LinkList &L)；

算法实现：使用malloc函数为头结点分配sizeof(LNode)空间，将首地址返回赋值给L，即将头指针指向头结点，并将头结点的后继指向空，数据域赋为零。

复杂度分析：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（2）销毁表status DestroyList(LinkList &L)；

算法实现：定义两个结构指针p与q，p指向需要释放的结点空间，q指向需要释放的结点的后继，防止结点丢失。以p与q指针遍历链表，使用free函数释放掉包括头结点在内的全部存储空间，L赋值为NULL，即将L指向空地址。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（3）清空表status ClearList(LinkList L)；

算法实现：清空函数与销毁函数的实现类似，主要区别在于是否释放头结点的空间，清空表函数不需要释放头结点的空间。与销毁表函数相同，定义两个结构指针p与q，p指向需要释放的结点空间，q指向需要释放的结点的后继。以p与q指针遍历链表，使用free函数释放掉除去头结点以外的全部结点存储空间，并即将头结点的后继指针指向空地址（清空表函数流程图，如图2-2所示）。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

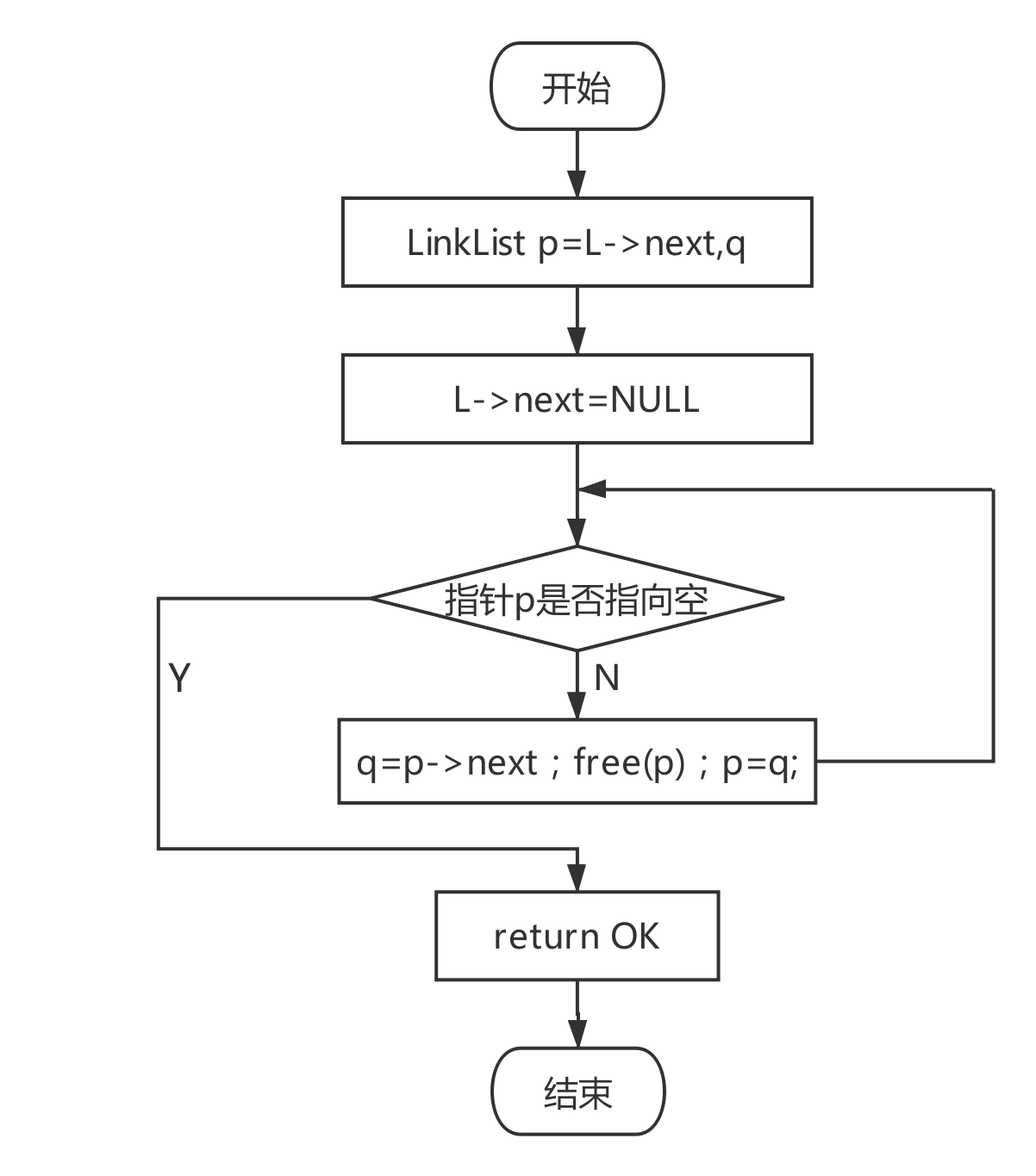


图2-2 清空表函数流程图

（4）判定空表status ListEmpty(LinkList L)；

算法实现：判断L->next的值，若L->next的值为NULL，即头结点指向空地址表为空表，那么返回TRUE，否则返回FALSE。

复杂度分析：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（5）求表长int ListLength(LinkList L)；

算法实现：若当前线性表不存在，则返回ERROR；若当前线性表存在，则定义遍历结构指针p，用于遍历线性表；并定义整型变量length作为计数器记录遍历长度，length自加直到指针p指向空地址时停止。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（6）获得元素status GetElem(LinkList L,int i,ElemType &e)；

算法实现：定义整型变量j作为记录位序的计数器，结构指针p作为遍历指针；由于获取元素函数不用改变前驱与后继结点的指针域指向信息，所以不需要使得指针p滞后于计数器j，因此指针p和计数器j的位序是完全同步的。以p作为遍历指针遍历线性表，j作为计数器记录位置，遍历停止条件为j>=i或p指向空地址；若遍历停止时，p指向空地址或者i非法，那么返回ERROR；否则代表找到正确的元素，将p指向的数据赋值给变量e，并返回OK（获得元素函数流程图，如图2-3所示）。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

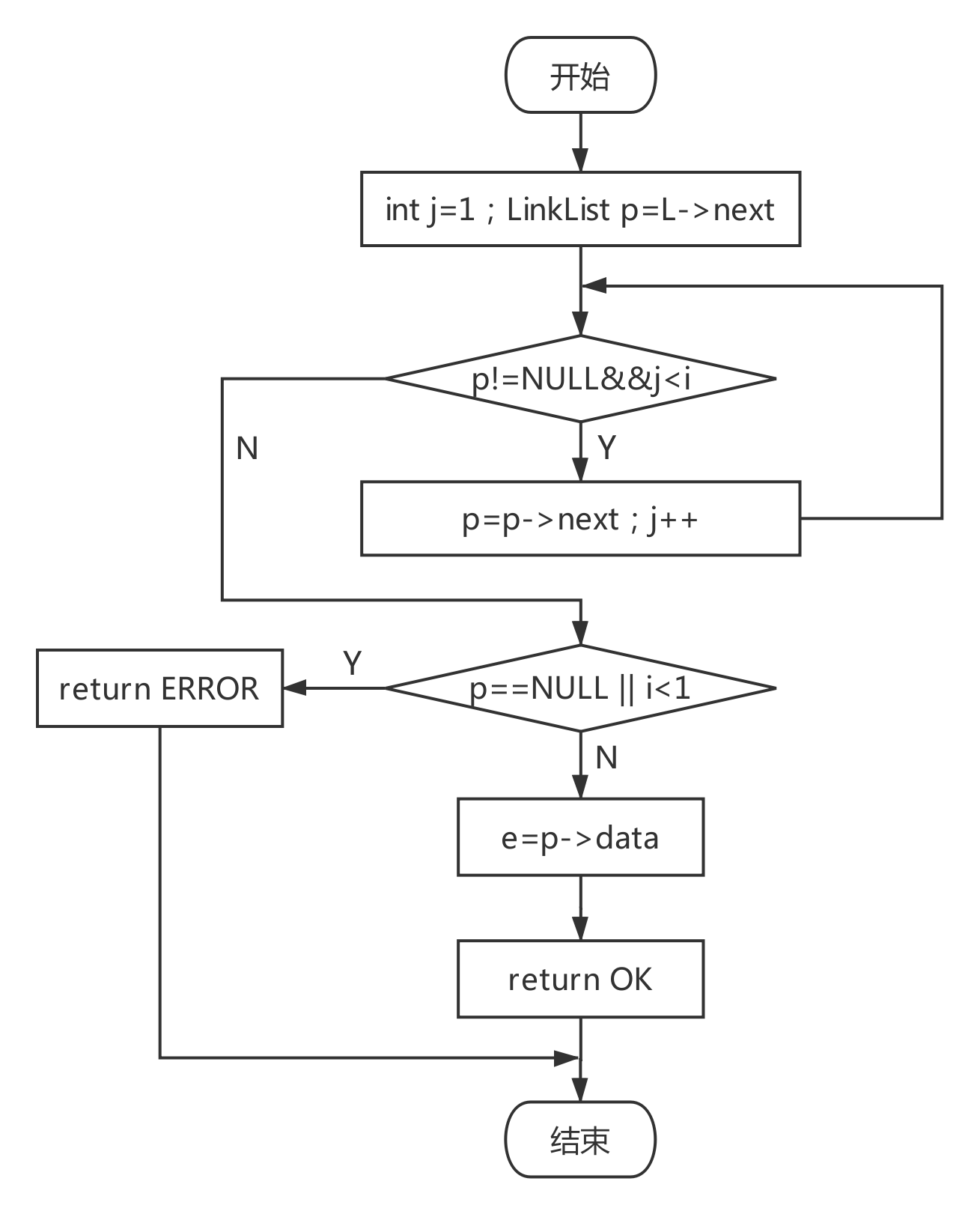


图2-3 获取元素函数流程图

（7）查找元素int LocateElem(LinkList L,ElemType e)；

算法实现：定义整型变量i作为计数器记录元素的位序以及结构指针p作为遍历指针，通过指针p循环遍历线性表，将每一个元素与给定值e比较看是否相等，如果相等就返回该元素的次序，否则返回0。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（8）获得前驱status PriorElem(LinkList L,ElemType cur,ElemType &pre\_e)；

算法实现：定义两个结构指针p与q，其中p作为遍历指针，q指向p指针所指元素的前驱元素。首先，若cur的值和L->next->data，即除去头结点的首个结点数据域数值相同，则返回-1代表元素不存在前驱；然后以p作为遍历指针遍历线性表，q指向p的前驱结点，当cur与p->data相等时，则将q->data赋值给pre\_e，返回OK；若遍历完后无元素符合，则返回ERROR。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（9）获得后继status NextElem(LinkList L,ElemType cur,ElemType &next\_e)；

算法实现：该函数与获得前驱函数类似，但只需要定义一个结构指针p作为遍历指针，以p作为遍历指针遍历线性表，当cur与p->data相等时，则将q->data赋值给next\_e，返回OK，遍历到倒数第二个元素停止；然后判断是否为最后一个元素，若是的话返回-1，否则返回ERROR。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（10）插入元素status ListInsert(LinkList L,int i,ElemType e)；

算法实现：该函数的实现与获取元素函数相似，同样定义整型变量j作为记录位序的计数器，结构指针p作为遍历指针；与获取函数元素的实现不同的是，由于插入元素函数需要改变前驱结点的指针域指向信息，所以要使得指针p滞后于计数器j，以便于进行结点的插入。以p作为遍历指针遍历线性表，j作为计数器记录位置，但是需要使得指针p指向的结点滞后于j所代表的结点一个单位，遍历停止条件为j>=i或p指向空地址；若遍历停止时，p指向空地址或者i非法，那么返回ERROR；否则使用malloc函数创建新结点插入线性表，并返回OK。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（11）删除元素status ListDelete(LinkList L,int i,ElemType &e)；

算法实现：该函数的实现与获取元素函数与插入元素函数都相似，同样定义整型变量j作为记录位序的计数器，结构指针p作为遍历指针；与获取函数元素的实现相同，要使得指针p滞后于计数器j，以便于进行结点的删除。以p作为遍历指针遍历线性表，j作为计数器记录位置，需要使得指针p指向的结点滞后于j所代表的结点一个单位，遍历停止条件为j>=i或p->next指向空地址；若遍历停止时，p->next指向空地址或者i非法，那么返回ERROR；否则使用e返回删除元素值，使用free函数释放需要删除的结点，并返回OK。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（12）遍历表status ListTrabverse(LinkList L)；

算法实现：定义遍历指针p循环遍历线性表，输出线性表中的每一个元素值。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（13）读取文件status OpenFile(LinkList L)；

算法实现：使用fopen函数打开指定文件，然后循环使用fscanf函数读取文件的数据，并用malloc函数创建结点，将数据赋值给结点并将结点插入到线性表的尾部，直到读取到文件尾停止时读入元素，关闭文件。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（14）键盘赋值status SetValue(SqList &L)；

算法实现：在键盘上输入数据，数据集以Ctrl+Z为结束标志，然后使用scanf函数从缓冲区中读取数据，并用malloc函数创建结点，将数据赋值给结点并将结点插入到线性表的尾部，直到读取到Ctrl+Z时停止读入元素。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（15）文件保存status SaveFile(SqList &L)；

算法实现：首先，选择读写文件的方式，选择1选项不删除原有的线性表，在原有数据之后添加数据，或者选择2选项删除原有线性表数据或生成新文件；然后使用fopen函数创建或打开文件，然后用fputs函数和fprintf函数输出数据，最后关闭文件（文件保存函数流程图，如图2-4所示）。

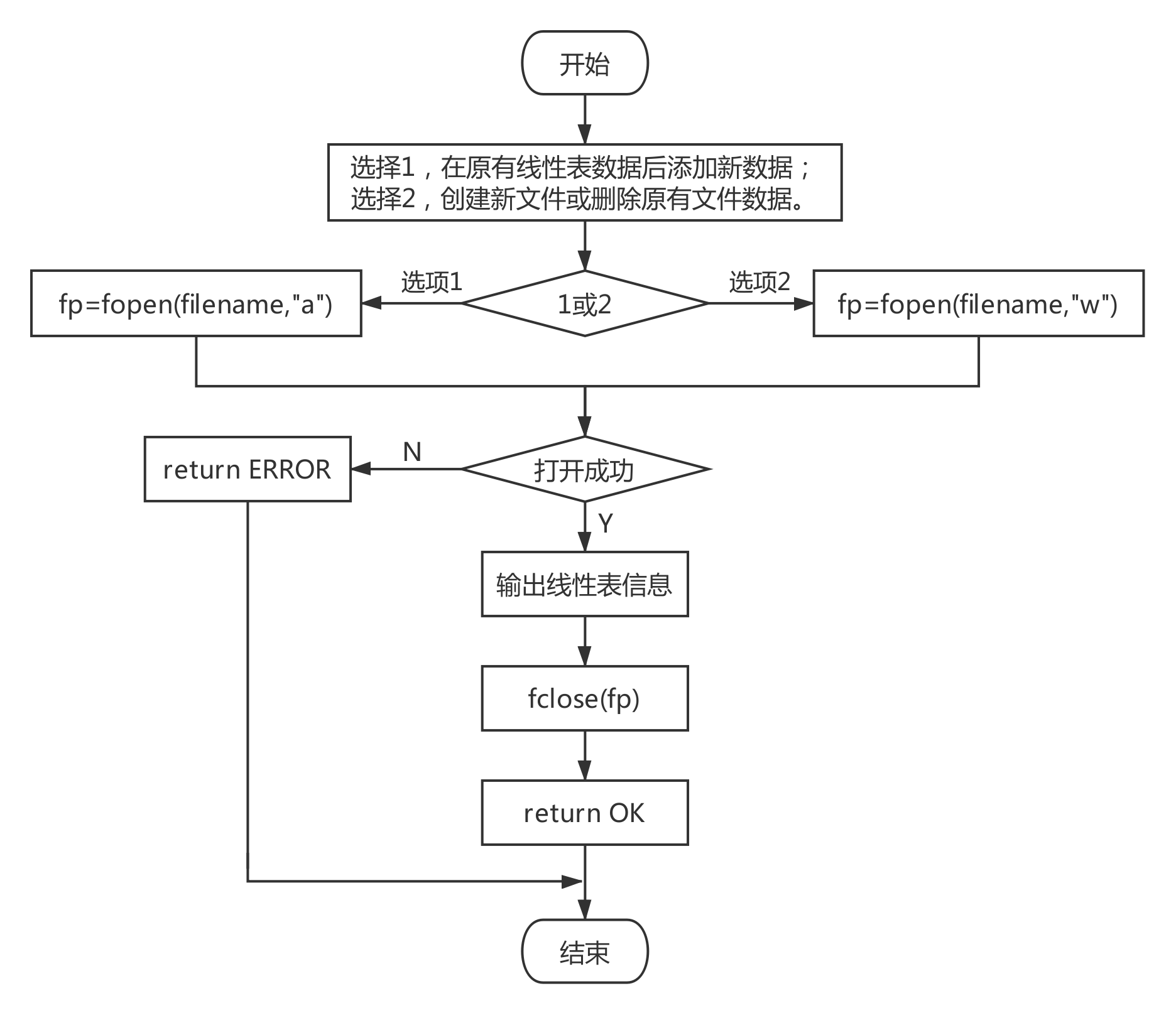


图2-4 文件保存函数流程图

## 2.4 系统测试

1. 当程序开始运行时，会立刻进入菜单演示界面（菜单界面如图2-5所示）。

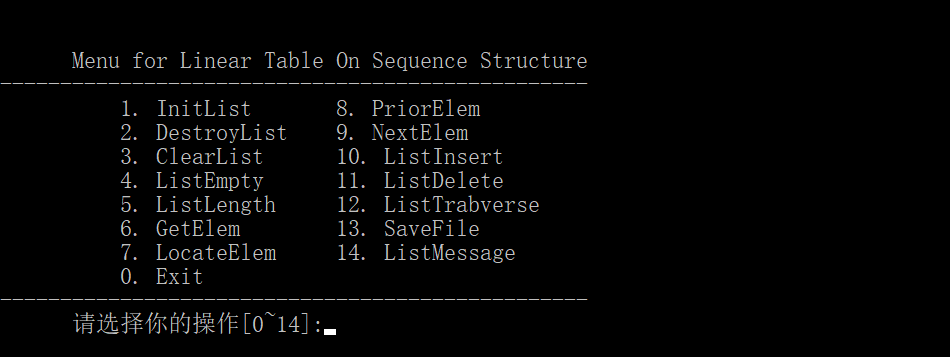


图2-5 演示系统菜单界面图

2. 按照菜单的提示输入需要完成的操作的序号，程序就会进入相对应的操作过程；当系统中没有线性表存在时，进行操作2-13，系统会做出提示显示“当前无线性表存在，请先创建线性表”（操作结果图如图2-6所示）。

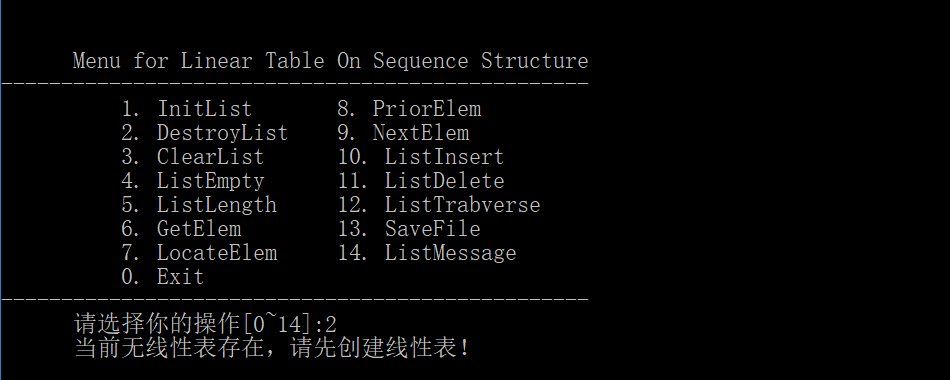


图2-6 系统无线性表存在时进行2-13操作结果图

3. 当系统中存在线性表时，可以进行所有操作，在进行2-13操作时，需要输入线性表的名称确定需要操作的线性表（操作界面图如图2-7所示）。

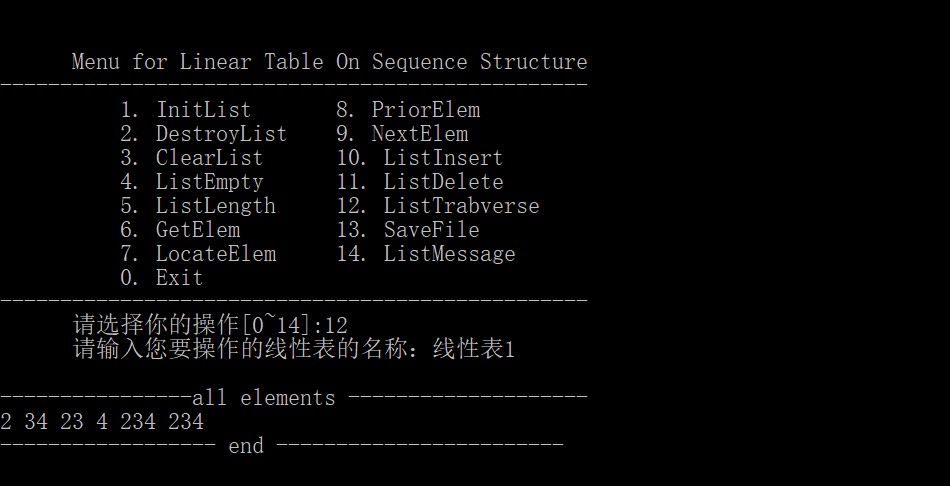


图2-7 输入线性表名称进行操作界面图

4. 当需要进行多表操作时，若需要查看所有线性表的信息，则可使用选项14操作，显示线性表信息（操作结果图如图2-8所示）。

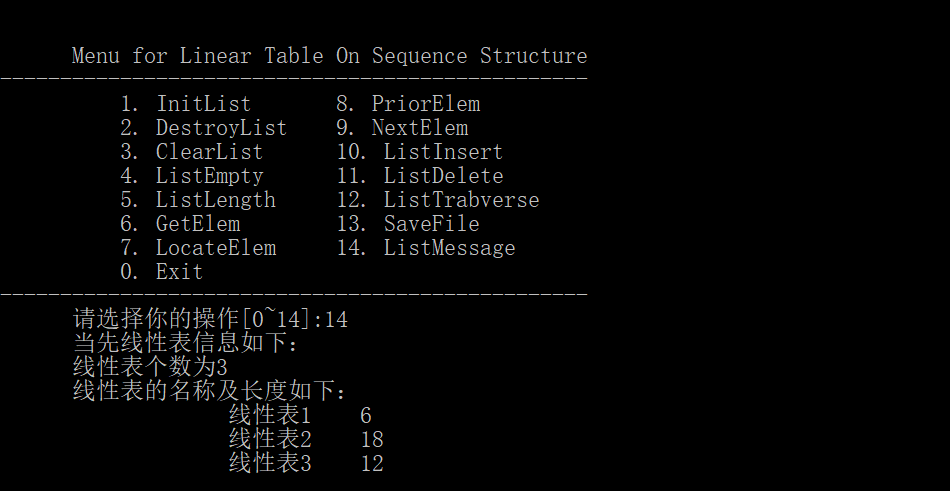


图2-8 显示多线性表信息操作结果图

5. 系统测试计划

使用预先保存的测试样例进行测试，本次测试挑选DestroyList、LocateElem、PriorElem、ListInsert、SaveFile共5个函数进行测试。本次使用的测试样例为多表测试样例，样例中包含4个线性表，其信息如下：

List1（表中包含5,12,0,44,94,137,47,3,9）；

List2（表中包含2,5,7,9,13,15,23）；

List3（空表）；

List4（表中包含4,23,5,33,7,23,5,7,77）；

其中，List1、List2与List3为键盘赋值（键盘赋值操作图如图2-9所示），List4为文件赋值（文件赋值操作图如图2-10所示）。

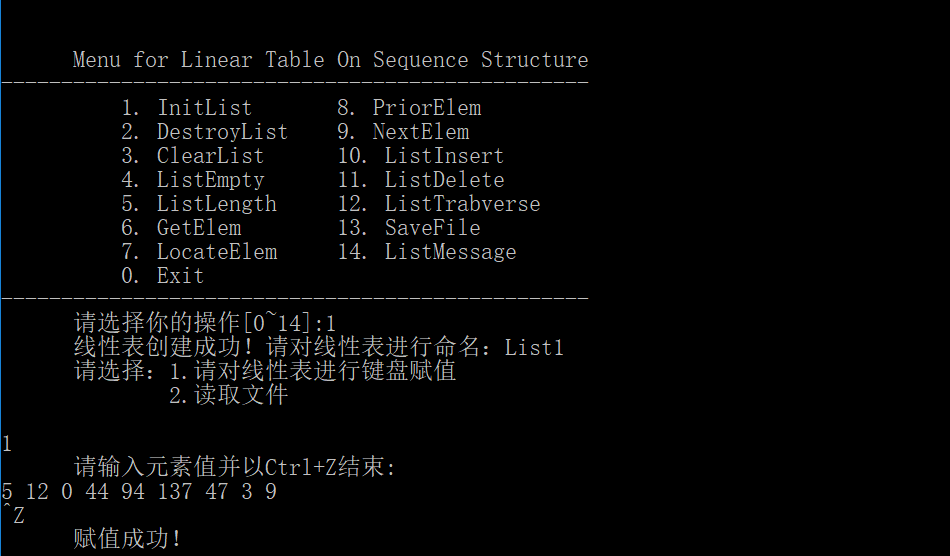


图2-9 线性表通过键盘赋值结果图

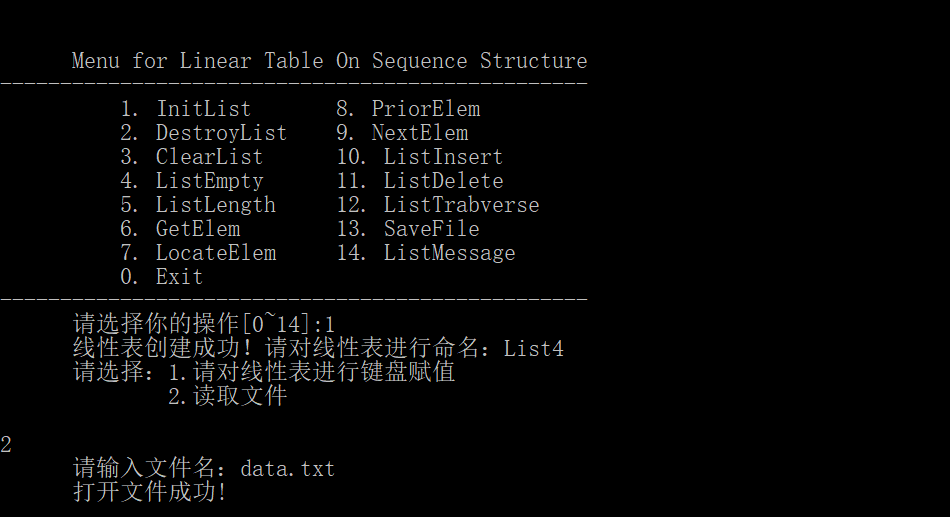
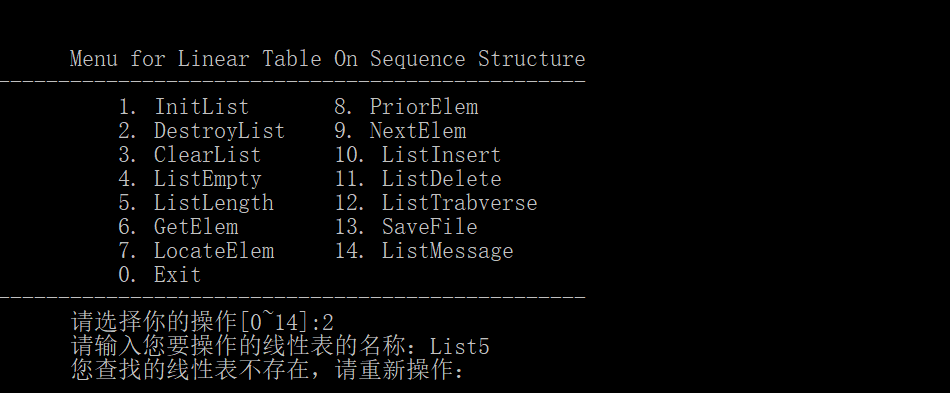


图2-10 线性表通过文件赋值结果图

（1）DestroyList函数测试：

表2-1 DestroyList函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| 无 | 1. 主界面输入2进入操作；  2. 按提示输入“List5”作为需要销毁的线性表，此线性表不存在。 | 线性表不存在 | （失败结果图如图2-11所示） |
| List2 | 1. 主界面输入2进入操作；  2. 按提示输入“List2”作为需要销毁的线性表。 | 线性表销毁成功！ | List2线性表删除成功（结果图如图2-12所示） |

图2-11 线性表不存在时显示销毁失败结果图

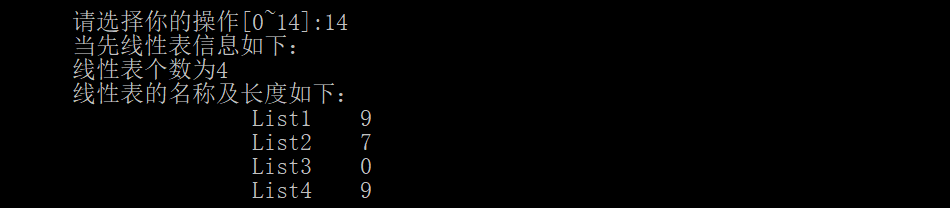
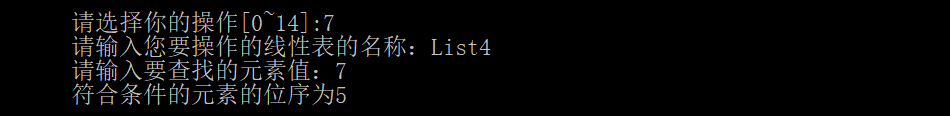
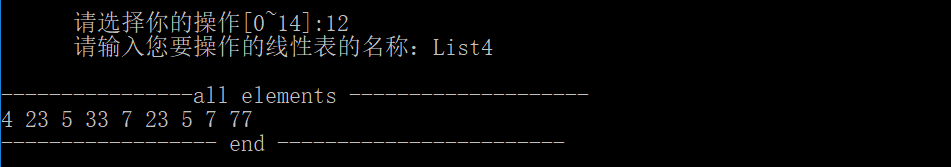


图2-12 线性表销毁前与销毁成功结果对比图

（2）LocateElem函数测试：

表2-2 LocateElem函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List4 | 1. 主界面输入7进入操作；  2. 按提示输入“List4”，按照提示输入“7”作为需要查找的元素。 | 符合条件的元素的位序为5 | （输出结果图如图2-13所示） |
| List4 | 1. 主界面输入7进入操作；  2. 按提示输入“List4”，按照提示输入“6”作为需要查找的元素。 | 没有符合条件的元素存在 | （输出结果图如图2-13所示） |



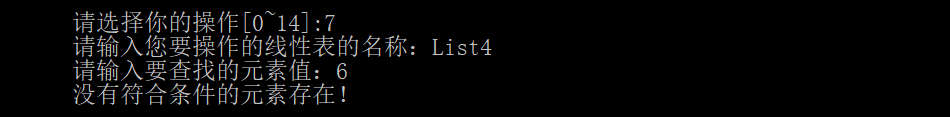
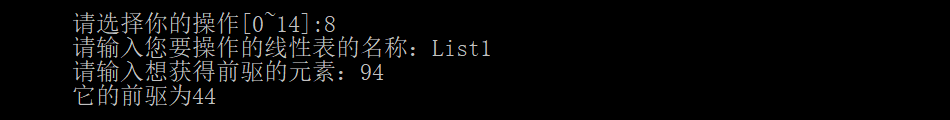
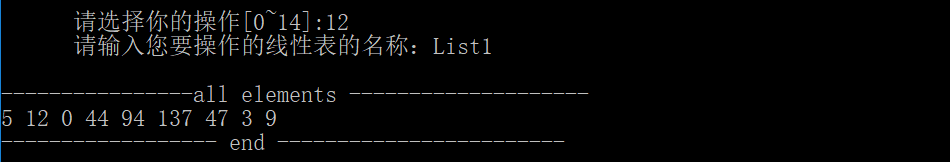


图2-13 查找元素函数测试结果图

（3）PriorElem函数测试：

表2-3 PriorElem函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List1 | 1. 主界面输入8进入操作；  2. 按提示输入“List1”，按照提示输入“94”作为需要查找前驱的元素。 | 它的前驱为44 | （输出结果图如图2-14所示） |
| List1 | 1. 主界面输入7进入操作；  2. 按提示输入“List1”，按照提示输入“5”作为需要查找前驱的元素。 | 该元素不存在前驱 | （输出结果图如图2-14所示） |



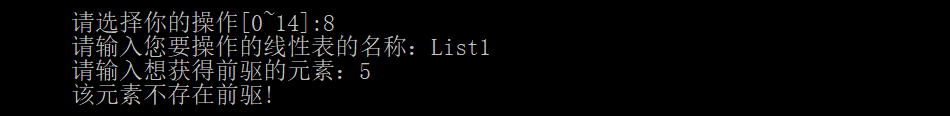


图2-14 查找元素前驱函数测试结果图

（4）ListInsert函数测试：

表2-4 ListInsert函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List1 | 1. 主界面输入10进入操作；  2. 按提示输入“List1”，输入“10”作为插入位点，“15”为插入元素。 | 操作成功 | （输出结果图如图2-15所示） |
| List1 | 1. 主界面输入10进入操作；  2. 按提示输入“List1”，输入“5”作为插入位点，“19”为插入元素。 | 操作成功 | （输出结果图如图2-15所示） |

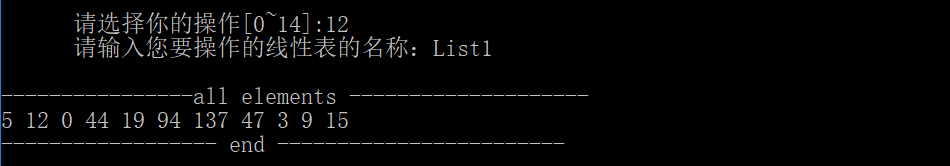
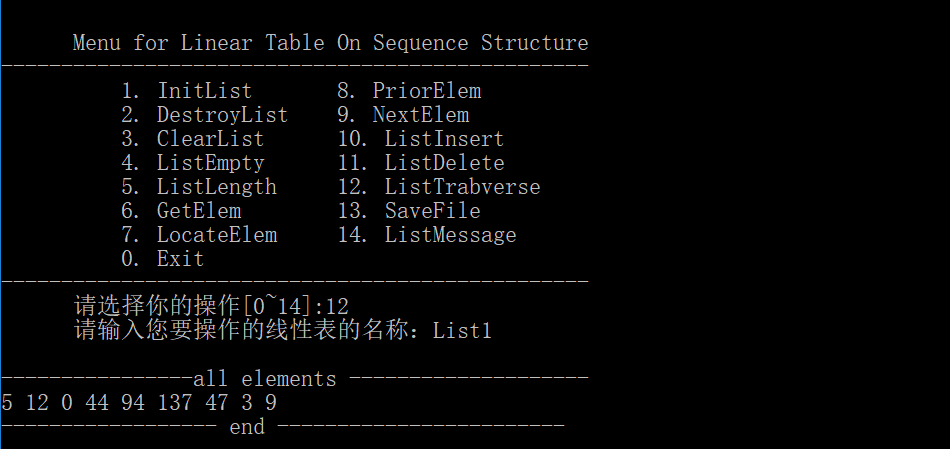
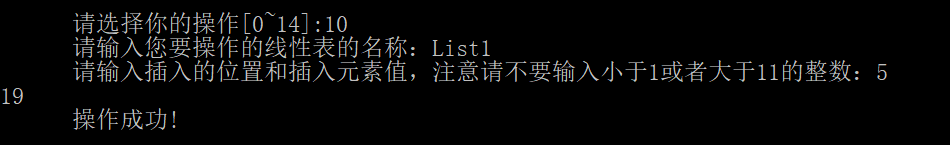
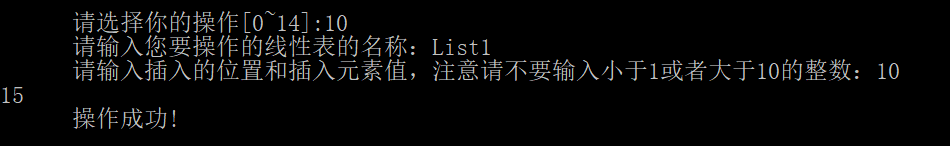
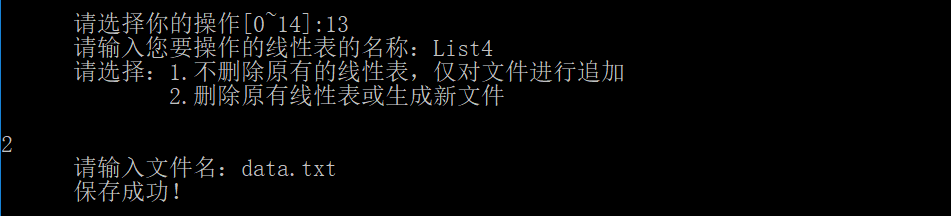


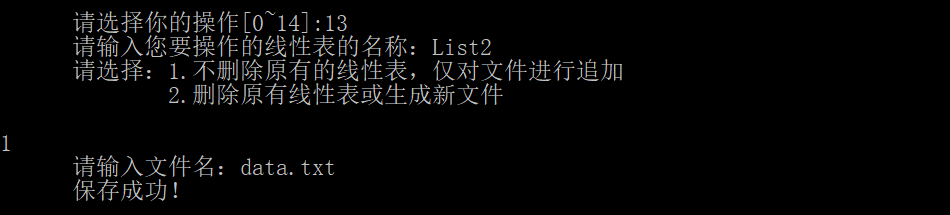
图2-15 插入元素函数结果图

（5）SaveFile函数测试：

表2-5 SaveFile函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试线性表 | 程序输入 | 理论输出 | 实际输出 |
| List4 | 1. 主界面输入13进入操作；  2. 按提示输入“List4”，按照提示输入“2”创建新文件进行文件保存。  3. 输入文件名“data.txt” | 保存成功 | （输出结果图如图2-16所示） |
| List2 | 1. 主界面输入13进入操作；  2. 按提示输入“List4”，按照提示输入“1”在旧文件后添加数据。  3. 输入文件名“data.txt” | 保存成功 | （输出结果图如图2-16所示） |





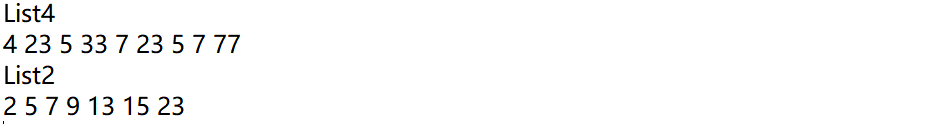


图2-16 文件保存函数测试结果图

## 2.5 实验小结

本次实验与第一次实验类似，都是关于线性表的存储结构的实验，在本次实验中，使用的语言为C语言，同样为了传递数据的方便，使用了引用传递作为传递参数的手段。本次实验完成的任务为线性表的链式存储。

由于本次实验与第一次实验类似，故在主函数的编写调用以及菜单界面和菜单操作的设计上沿用了第一次的实验设计，本次的实验的代码设计与编写主要为函数的设计与编写。在本次设计算法和编写代码的过程中，我遇到的问题不多，可以说是比较顺利地完成了本次实验的代码实现。在本次实验中，比较困扰我的问题主要在尾结点的边界条件判定以及非法情况的判定方面，这两方面的问题我着实思考了许久。

通过本次实验，我对指针和引用传递都有了更加深刻的理解，对链表边界条件的判断也有了一定的体会，可以说收获不少。

# 3基于二叉链表的二叉树实现

## 3.1 问题描述

以二叉链表作为二叉树的物理结构。其中，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。演示系统需实现多个二叉树管理，以及实现二叉树的文件形式保存。其中，需要设计文件数据记录格式，以高效保存二叉树数据逻辑结构的完整信息；以及设计二叉树文件保存和加载操作合理模式。

要求二叉树结点类型为结构型，至少包含二个部分，一个是能唯一标识一个结点的关键字（类似于学号或职工号），另一个是其它部分。采用线性表的方式管理多个二叉树，线性表中的每个数据元素为一个二叉树的基本属性，至少应包含有二叉树的名称。

### 3.1.1 二叉树的基本概念与逻辑结构

二叉树是一种树型结构，即n个结点的有限集，它的每个结点至多只有两棵子树，且二叉树的子树有左右之分，被称作“左子树”和“右子树”，其次序不能任意颠倒。

二叉树的数据逻辑结构定义如下：

ADT BinaryTree {

数据对象D：D是具有相同特性的数据元素的集合。

数据关系R：若D=Φ，则R=Φ，称BinaryTree为空二叉树；

若D≠Φ，则R={H},H是如下二元关系：

1. 在D中存在唯一的成为根的数据元素root，它在关系H中无前驱；
2. 若D-{root}≠Φ,则存在D-{root}={D1,Dr},且D1∩Dr=Φ；
3. 若D1≠Φ，则D1中存在唯一的元素X1，<root,X1>∈H，且存在D1上的关系H1包含于H；同理<root,Xr>∈H,且存在Dr上的关系属于H；
4. (D,{H1})是一棵符合本定义的二叉树，称为根的左子树，(Dr,{Hr})是一棵符合本定义的二叉树，称为根的右子树。

}

### 3.1.2 二叉树的基本运算

依据最小完备性和常用性相结合的原则，以函数形式定义了二叉树的14种基本运算。具体运算功能定义和说明如下：

1. 创建二叉树CreateBiTree(T,definition)；

初始条件：二叉树T不存在，definition 给出二叉树T的定义，如带空子树的二叉树前序遍历序列，或前序+中序，或后序+中序。

操作结果：按definition构造二叉树T。

1. 销毁二叉树DestroyBiTree(T)；

初始条件：二叉树T已存在。

操作结果：销毁二叉树T。

1. 清空二叉树ClearBiTree (T)；

初始条件：二叉树T已存在。

操作结果：将T重置为空二叉树。

1. 判定空二叉树BiTreeEmpty(T)；

初始条件：二叉树T已存在。

操作结果：若T为空二叉树则返回TRUE，否则返回FALSE。

1. 求二叉树深度BiTreeDepth(T)；

初始条件：二叉树T已存在。

操作结果：返回T的深度。

1. 查找结点LocateNode(T,e)；

初始条件：二叉树T已存在，e是和T中结点关键字类型相同的给定值。

操作结果：返回查找到的结点指针，如无关键字为e的结点，返回NULL。

1. 结点赋值Assign(T,e,value)；

初始条件：二叉树T已存在，e是和T中结点关键字类型相同的给定值。

操作结果：关键字为e的结点赋值为value。

1. 获得兄弟结点GetSibling(T,e)；

初始条件：二叉树T已存在，e是和T中结点关键字类型相同的给定值。

操作结果：返回关键字为e的结点的（左或右）兄弟**结点指针**。若关键字为e的结点无兄弟，则返回NULL。

1. 插入结点InsertNode(T,e,LR,c)；

初始条件：二叉树T已存在，且e是和T中结点关键字类型相同的给定值，LR为0或1，c是待插入结点。

操作结果：根据LR为0或者1，插入结点c到T中，作为关键字为e的结点的左或右孩子结点，结点e的原有左子树或右子树则为结点c的右子树。

1. 删除结点DeleteNode(T,e)；

初始条件：二叉树T已存在，e是和T中结点关键字类型相同的给定值。

操作结果：删除T中关键字为e的结点；同时，如果关键字为e的结点度为0，删除即可；如关键字为e的结点度为1，用关键字为e的结点孩子代替被删除的e位置；如关键字为e的结点度为2，用e的左孩子代替被删除的e位置，e的右子树作为e的左子树中最右结点的右子树。

1. 前序遍历PreOrderTraverse(T)；

初始条件：二叉树T已存在。

操作结果：对二叉树进行前序遍历，对每个结点仅输出一次。

1. 中序遍历InOrderTraverse(T)；

初始条件：二叉树T已存在。

操作结果：对二叉树进行中序遍历，对每个结点仅输出一次。

1. 后序遍历PostOrderTraverse(T)；

初始条件：二叉树T已存在。

操作结果：对二叉树进行后序遍历，对每个结点仅输出一次。

1. 按层遍历LevelOrderTraverse(T)；

初始条件：二叉树T已存在。

操作结果：对二叉树进行层序遍历，对每个结点仅输出一次。

## 3.2 系统设计

### 3.2.1 数据存储结构与形式

1. 基于二叉链表的二叉树的数据物理结构如下：

typedef struct BiTNode{

Sign Node\_num; //结点关键字变量

ElemType data; //结点数据项变量

struct BiTNode \*lchild; //左孩子指针变量

struct BiTNode \*rchild; //右孩子指针变量

}BiTNode,\*LinkBiTNode; //LinkBiTNode为结点指针变量类型

1. 链表结点结构体，用于读取带空子树的前序遍历序列并生成相对应的结点，并将对应结点挂在链表上（存储结构如图3-1所示），结构体定义如下：

typedef struct LNode{

LinkBiTNode Node; //树结点指针

struct LNode \*next; //链表后继指针

}LNode,\*LinkList; //作为存储结点的链表

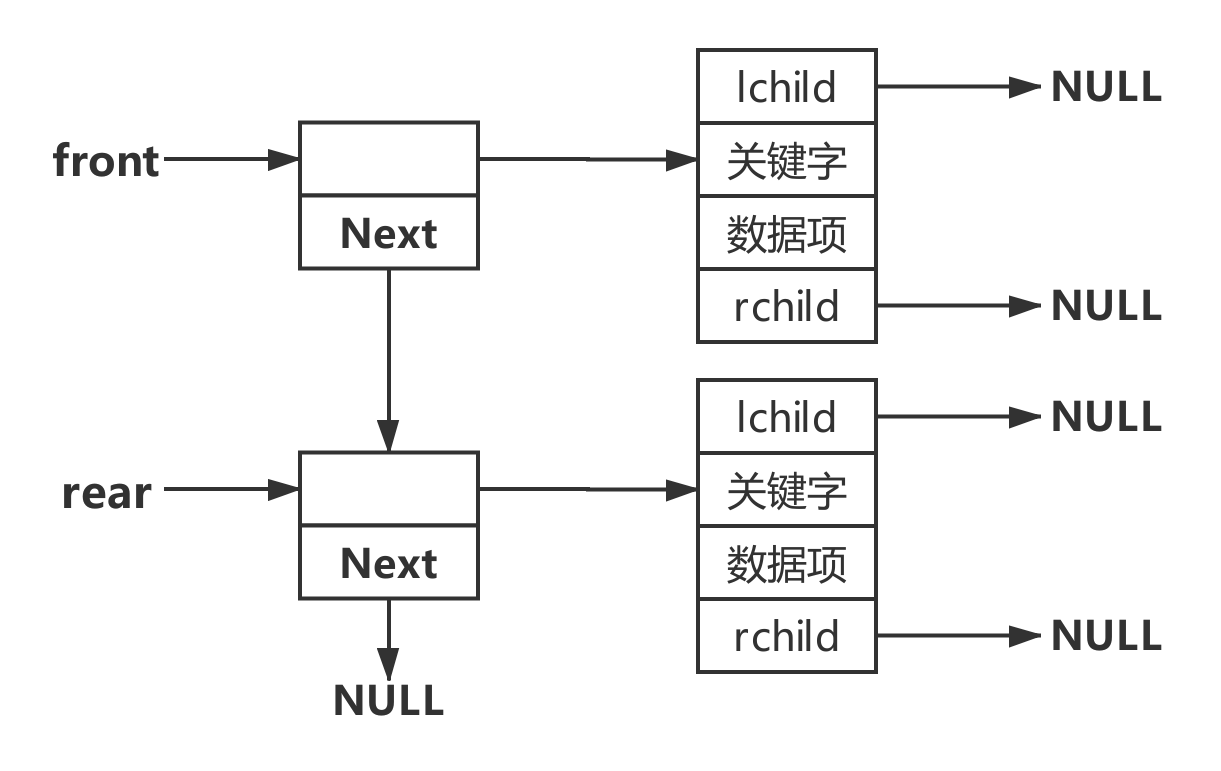


图3-1 读入序列存储结构示意图

1. 多二叉树管理（森林管理），用定义一个指向二叉树根结点的指针变量和存储二叉树名称的字符数组组成的结构体，通过二叉树的名称属性对二叉树进行操作，只要在主函数中定义一个结构数组即可进行多树管理，结构体定义如下：

typedef struct{

LinkBiTNode root; //指向二叉树的根结点

char name[30]; //存储二叉树的名称

}BiTrees; //森林线性表管理

1. 二叉树的图形化显示，通过队列控制二叉树结点的输出，使得结点可以按照层的顺序来输出，其中Node指向结点，num存储结点在满二叉树中的序号（存储结构如图3-2所示），以便于输出树的形状（由于需要多次输出队列中的元素，即队列出队时不需要删除元素，仅仅是front移动），结构体定义如下：

typedef struct{

LinkBiTNode Node;

int num; //记录**结点在满二叉树中的序号**，便于进行树的图形化输出

}Node; //队列线性表

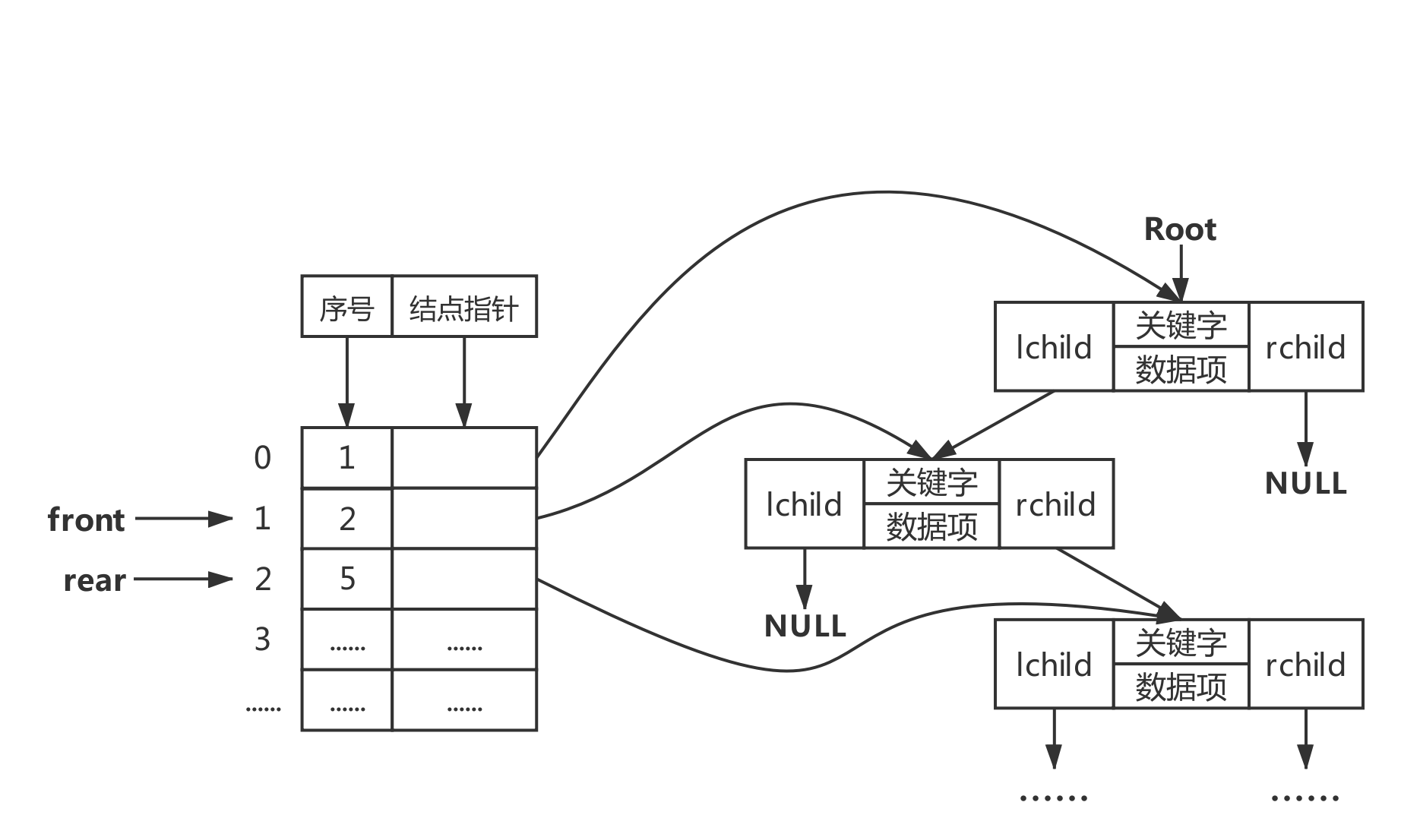


图3-2 图形化显示树队列存储结构示意图

### 3.2.2 总体系统设计

通过WHILE循环与菜单界面，用户通过选择菜单中的选项实现交互，使用OP变量获取用户选择选项值（OP初始化值为1，以便第一次能进入循环）。

进入循环后系统首先显示功能菜单，然后用户输入选择0-17，其中1-14分别代表二叉树的一个基本运算，15选项为二叉树的文件保存，16选项为显示全部二叉树信息（**本选项用于方便查看在存在多个二叉树的情况下，各个二叉树的名称与深度信息**），**17选项为二叉树的图形化显示**。在主函数中通过SWITCH语句对应到相应的函数功能，执行完该功能后BREAK跳出SWITCH语句，继续执行WHILE循环，直至用户输入0退出当前系统。

在进行二叉树的操作时，除了1和16选项（1选项是进行二叉树的生成和初始化，16选项用于查看所有二叉树信息），系统会要求输入需要处理的二叉树的名称（本系统多二叉树操作以二叉树名称作为查找特征），若二叉树不存在会返回菜单界面重新操作（系统结构如图3-3所示）。

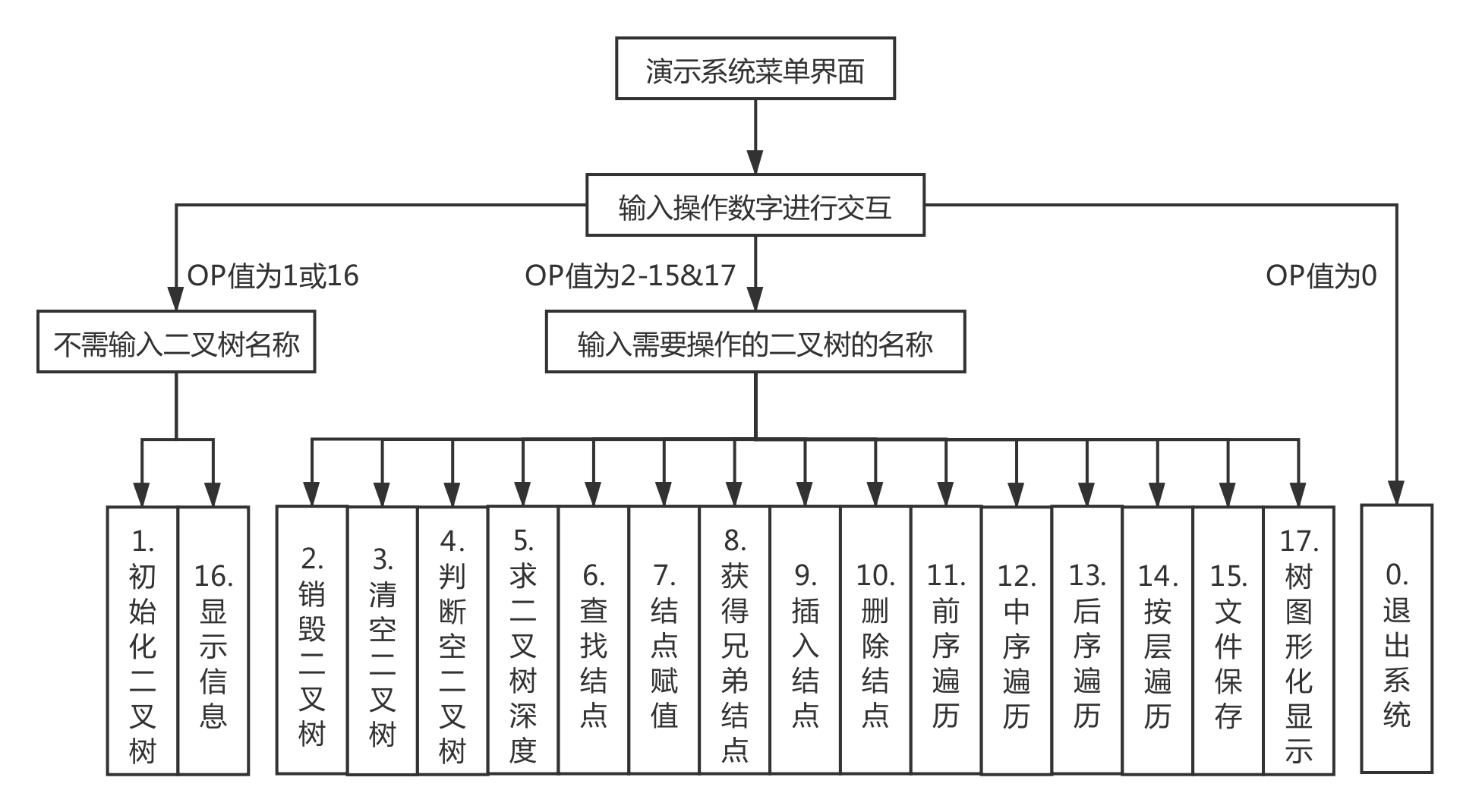


图3-3 系统菜单操作示意图

## 3.3 系统实现

### 3.3.1 编程环境与运行环境描述

编程环境：采用CodeBlocks 16.01编程软件编写。

运行环境：微软Windows 10系统。

### 3.3.2 头文件及预定义常量说明

1.头文件

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

2.预定义常量

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define N 100

#define LIST\_INIT\_SIZE 100

3.类型表达式

typedef int status;

typedef int Sign; //结点关键字元素类型定义

typedef int ElemType; //数据元素类型定义

### 3.3.3 算法设计与实现

为方便算法的实现和多二叉树管理与操作，本次算法设计与实现以二叉链表来实现结点的储存，结构数组来存储多树的根指针与名称，具体算法实现如下：

（1）读取带空的前序序列status ReadBiNode(LinkList &front,LinkList &rear)；

算法实现：本函数采用链式结构作为存储结点以及其顺序信息的物理结构，（示意图详见3.2.1数据存储结构与形式，如上图3-1所示），每读入一个结点的关键字与数据项信息，就直接通过malloc函数生成二叉树结点，并将之挂在链表上，直到读取到Ctrl+Z时停止，完成序列读取。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（2）创建二叉树CreateBiTree1(LinkBiTNode &T,LinkList &front,LinkList &rear)；

算法实现：本函数采用非递归实现，利用根据带空子树的前序遍历序列生成的节点串创建二叉树（另两种方式也已经实现，但在此处不予赘述），利用栈来控制结点是否插入以及插入的位点，为空且还有需要插入的结点时，继续进行循环操作，在每一次循环中，首先，将栈顶元素取出作为需要插入的位点，若flag==0（即上一次插入的结点为空结点）则将结点出栈，否则不出栈；然后将需要插入的结点插入位点，同样以flag判断插入的位置，当插入的结点为空结点，则将指针赋值为NULL，并且需要将flag赋值为0（表示下个循环需要将结点出栈且插入在右结点），若不为空结点，则将结点插入位点，将flag赋值为1（表示下个循环仍需插入在左结点）；一直循环直到栈为空或者没有需要处理的结点，创建完成（创建二叉树函数流程图，如图3-4所示）。

**非法判断：栈为空，但需要处理的结点未处理完，表示序列出错；同理，当栈不为空但需要处理的结点不存在时，表示序列长度不足，也为非法。**

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

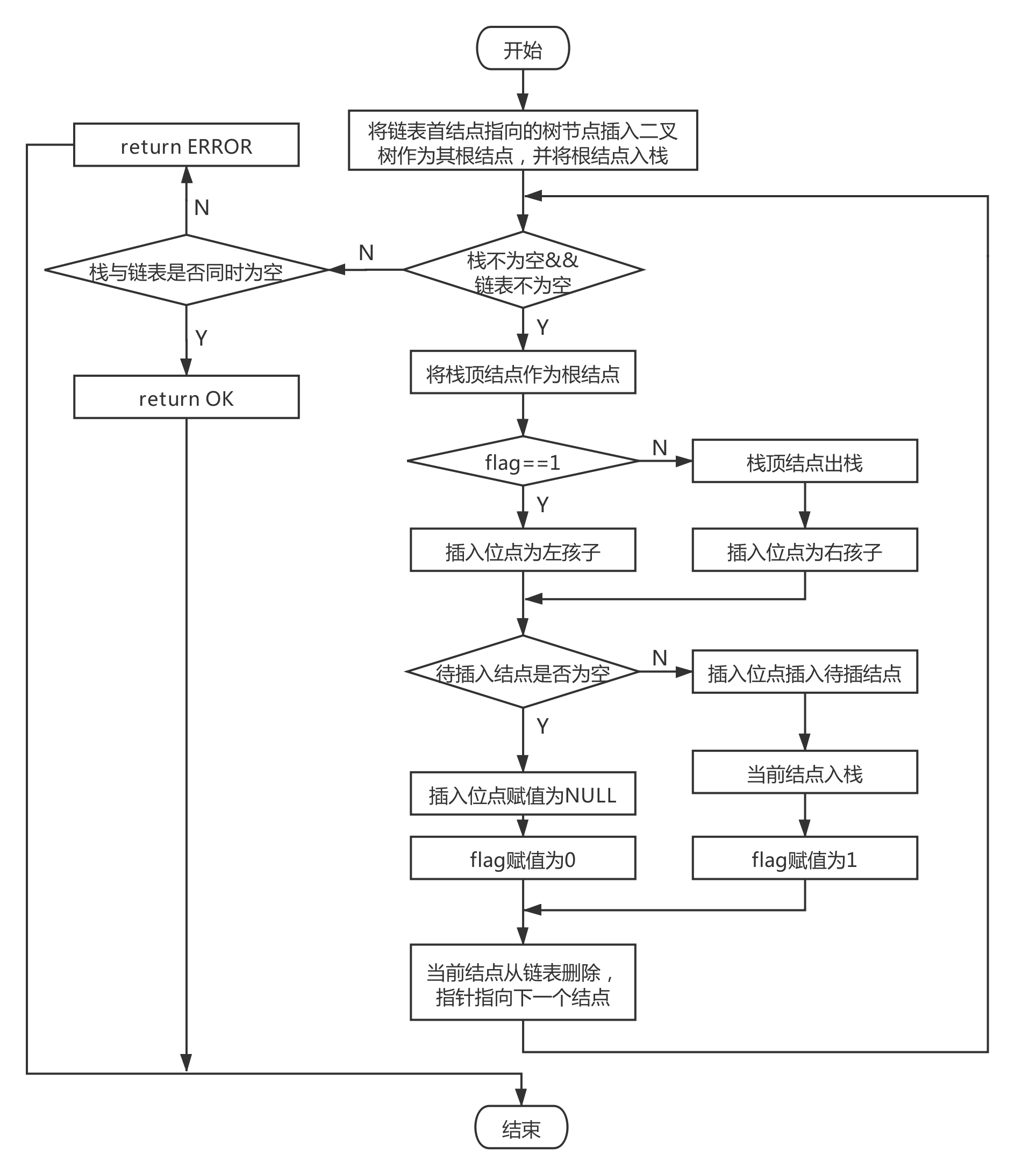


图3-4 创建二叉树函数流程图

（3）销毁二叉树status DestroyBiTree(LinkBiTNode &T)；

算法实现：本函数采用递归实现，递归出口有两个，第一个递归出口为结点为空，第二个递归出口为左右子树都已经完成操作。函数中递归调用本函数，分别处理左孩子与右孩子。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（4）清空二叉树status ClearBiTree(LinkBiTNode &T)；

算法实现：本函数与销毁二叉树实现完全相同，不再赘述。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（5）判定空二叉树status BiTreeEmpty(LinkBiTNode T)；

算法实现：判断T的值，若T的值为NULL，即根指针指向空地址，那么返回TRUE，否则返回FALSE。

复杂度分析：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（6）求二叉树深度int BiTreeDepth(LinkBiTNode T)；

算法实现：本函数为递归实现，递归出口有两个，第一个递归出口为当前结点为空，返回值为0；第二个递归出口为子树的深度加1，若左子树深度大则返回做左子树的深度加1，否则返回右子树的深度加1。函数两次调用自身，分别读取左子树与右子树的深度。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（7）查找结点LinkBiTNode LocateNode(LinkBiTNode T,Sign e)；

算法实现：本函数为递归实现，返回值为指针变量，递归出口有四个，第一

个递归出口为当前结点为空，返回值为空指针；第二个递归出口为查找到需要定位的结点，返回当前结点的指针值；第三个递归出口为左子树或右子树返回值不为NULL，即已查找到结点，返回非空指针值；第四个递归出口为若以上三种情况都不满足，即当前子树无需要查找的结点，返回NULL。函数两次调用自身，分别对左子树与右子树进行递归查找。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（8）结点赋值int LocateElem(LinkList L,ElemType e)；

算法实现：本函数为递归实现，递归出口有三个，第一个递归出口为当前结点为空，返回OK；第二个递归出口为查找到需要赋值的结点，返回OK；第三个递归出口为当前子树无需要查找的结点，返回OK。函数两次调用自身，分别对左子树与右子树进行递归查找赋值。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（9）获得兄弟结点LinkBiTNode GetSibling(LinkBiTNode T,Sign e)；

算法实现：本函数为递归实现，返回值为指针变量，递归出口有四个，为减

少递归的层数，第一个递归出口为当前结点为空或者其左右孩子中有一个为空即表示不可能存在兄弟结点，返回值为空指针；第二个递归出口为查找到需要定位兄弟结点的结点，返回兄弟结点的指针值；第三个递归出口为左子树或右子树返回值不为NULL，即已查找到兄弟结点，返回非空指针值；第四个递归出口为若以上三种情况都不满足，即当前子树无需要查找的兄弟结点，返回NULL。函数两次调用自身，分别对左子树与右子树进行递归查找。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（10）插入结点status InsertNode(LinkBiTNode T,Sign &e,int LR,LinkBiTNode c)；

算法实现：插入算法的实现与查找结点函数的实现类似，仅仅增加插入环节，且将返回值有指针值改为状态值，递归出口与赋值函数相似，不再赘述。除此之外，额外需添加对插入结点的左右孩子的赋值语句。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（11）删除结点status DeleteNode(LinkBiTNode &T,Sign &e)；

算法实现：删除结点函数的实现比较复杂，其递归出口与赋值函数类似，其主要难点在于要对三种不同情况进行处理。首先，需要判断当前结点是否是需要删除的结点，若是，则根据结点删除规则，**当左孩子存在时，无论右孩子存在与否，都应将左孩子结点替换当前需要删除的结点**；然后，**若左孩子存在且右孩子存在，需要将右孩子插入为左孩子最右子树的右孩子结点**；若左孩子存在但右孩子不存在，不需要其他操作。然后判断，若右孩子存在，应将右孩子结点替换当前需要删除的结点，若右孩子不存在，直接删除当前结点，将父节点当前孩子位点赋值为NULL。若当前结点不是需要删除的结点，则递归查找左右孩子结点（删除结点函数流程图，如图3-5所示）。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（12）前序遍历status PreOrderTraverse(LinkBiTNode T)；

算法实现：本函数为非递归实现，采用线性表模拟的栈实现结点的读取和输出。第一层循环的结束条件为当栈为空且当前处理的结点也为空，即表示遍历成功。进入第一层循环后，进行第二层循环，由前序遍历的规则可知，第二层循环遍历当前结点的左孩子，对结点进行输出并将结点入栈，直到遇到空结点；跳出第二层循环，访问栈顶结点的右孩子结点将之作为要处理的结点，并将栈顶结点出栈，第一层循环的一次循环结束，再次循环，直到遇到循环出口条件。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

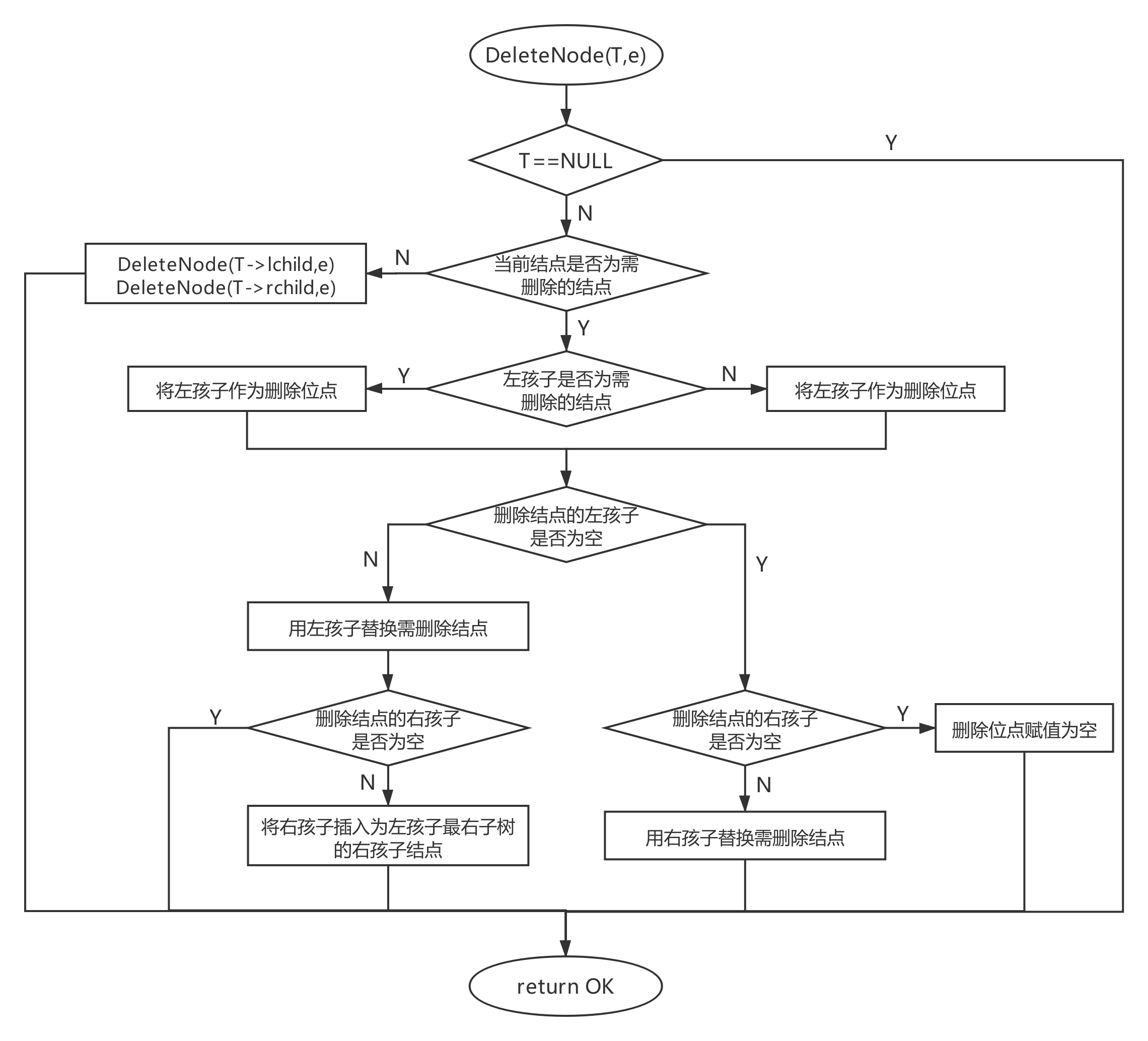


图3-5 删除结点函数流程图

（13）中序遍历status InOrderTraverse(LinkBiTNode T)；

算法实现：本函数实现与前序遍历非常相似，仅仅需要改变结点输出位置，将结点由入栈时输出改为出栈时输出即可，本中序遍历函数也采用非递归实现，采用线性表模拟的栈实现结点的读取和输出。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（14）后序遍历status PostOrderTraverse(LinkBiTNode T)；

算法实现：由于本函数若采取非递归控制十分复杂，故采取递归控制，递归出口有两个，第一个递归出口为当前结点为空，返回OK；第二个递归出口为两次递归调用自身完成并顺利输出结点，返回OK。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（15）按层遍历status LevelOrderTraverse(LinkBiTNode T)；

算法实现：本函数采用非递归实现，以队列作为读取以及存储结点的逻辑结构，队列使用链表进行模拟实现。定义temp指针指向当前需要处理的结点，循环遍历temp的左孩子与右孩子，并将左右孩子依次入队（孩子结点不为空）并在入队前进行输出，当temp指针所指结点的左右孩子都完成输出后，将temp所指的结点出队，temp指向下一个需要处理的结点，当队列为空时，遍历完成。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（16）文件保存status SaveFile(BiTrees T)；

算法实现：首先，选择读写文件的方式，选择1选项不删除原有的二叉树，在原有数据之后添加数据，或者选择2选项删除原有二叉树数据或生成新文件；然后使用fopen函数创建或打开文件，然后用fputs函数和fprintf函数输出数据，最后关闭文件（文件保存函数流程图，如图3-6所示）。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

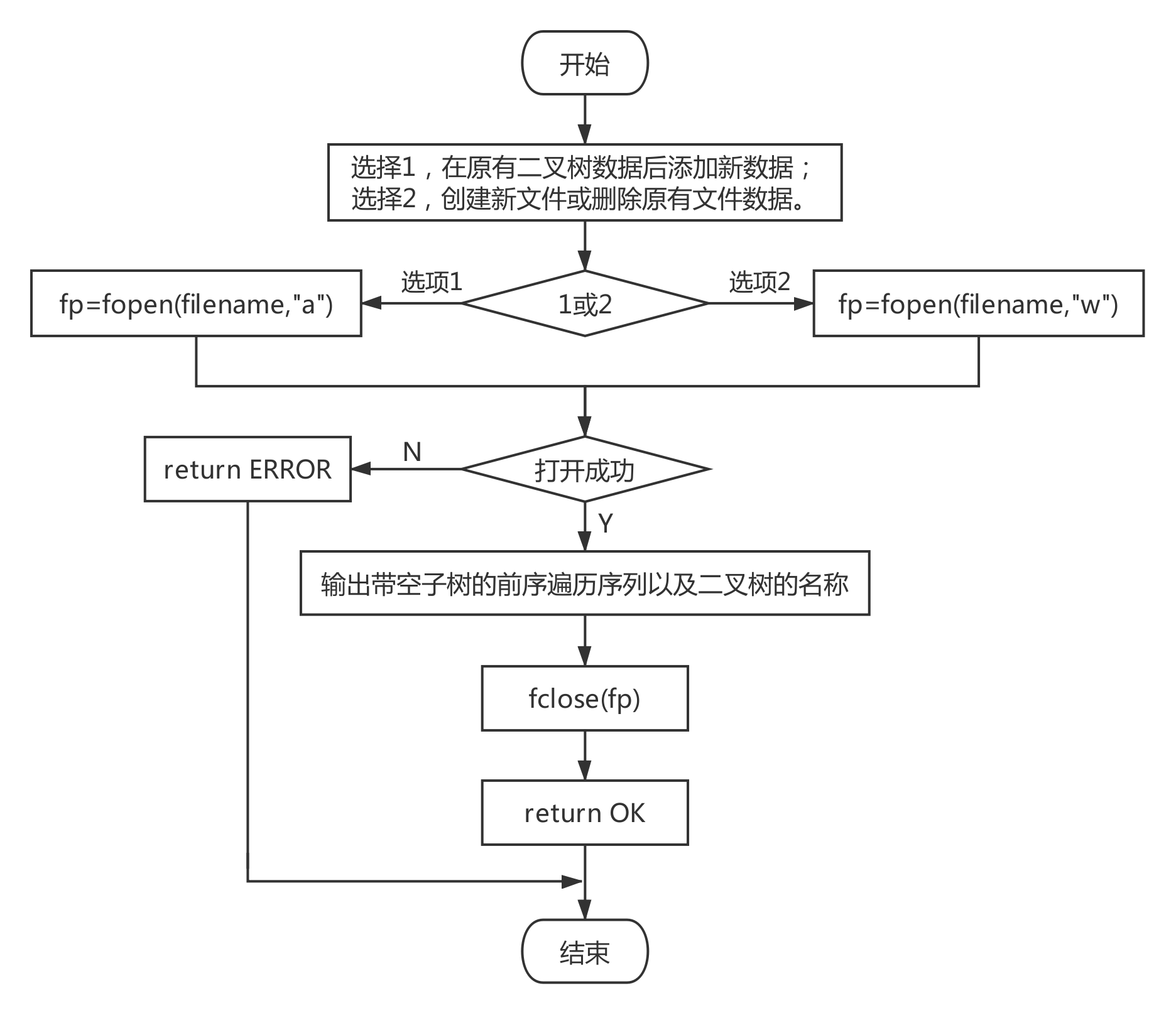


图3-6 文件保存函数流程图

（17）二叉树图形化显示status ShowTree(LinkBiTNode T)；

算法实现：本函数在按层遍历函数的基础上实现，同样采用队列作为存储和输出结点的逻辑结构，为简化操作以及多次输出序列，使用顺序表作为实现队列的物理结构，顺序表结点由两个变量组成，**num变量存储二叉树结点在满二叉树的序号**，以便于输出树结点的空间结构；Node变量为指针变量，指向入队的二叉树结点（物理结构示意图，如图3-7所示）。本函数实现了两个功能，分别为在显示器显示以及输出为文件。由于输出的树以满二叉树作为模板，故当树的深度超过5时，第6层满二叉树结点数为32个，无法在显示器完整输出，故建议文件输出。具体的输出过程为通过计算每个结点的左右空格、下划线以及箭头的数目以输出树的结构，具体输出过程过于琐碎，在此不再赘述。

复杂度分析：时间复杂度最坏情况T(n) = O(2^n)，平均时间复杂度为T(n) = O(n)，空间复杂度S(n) = O(n)。

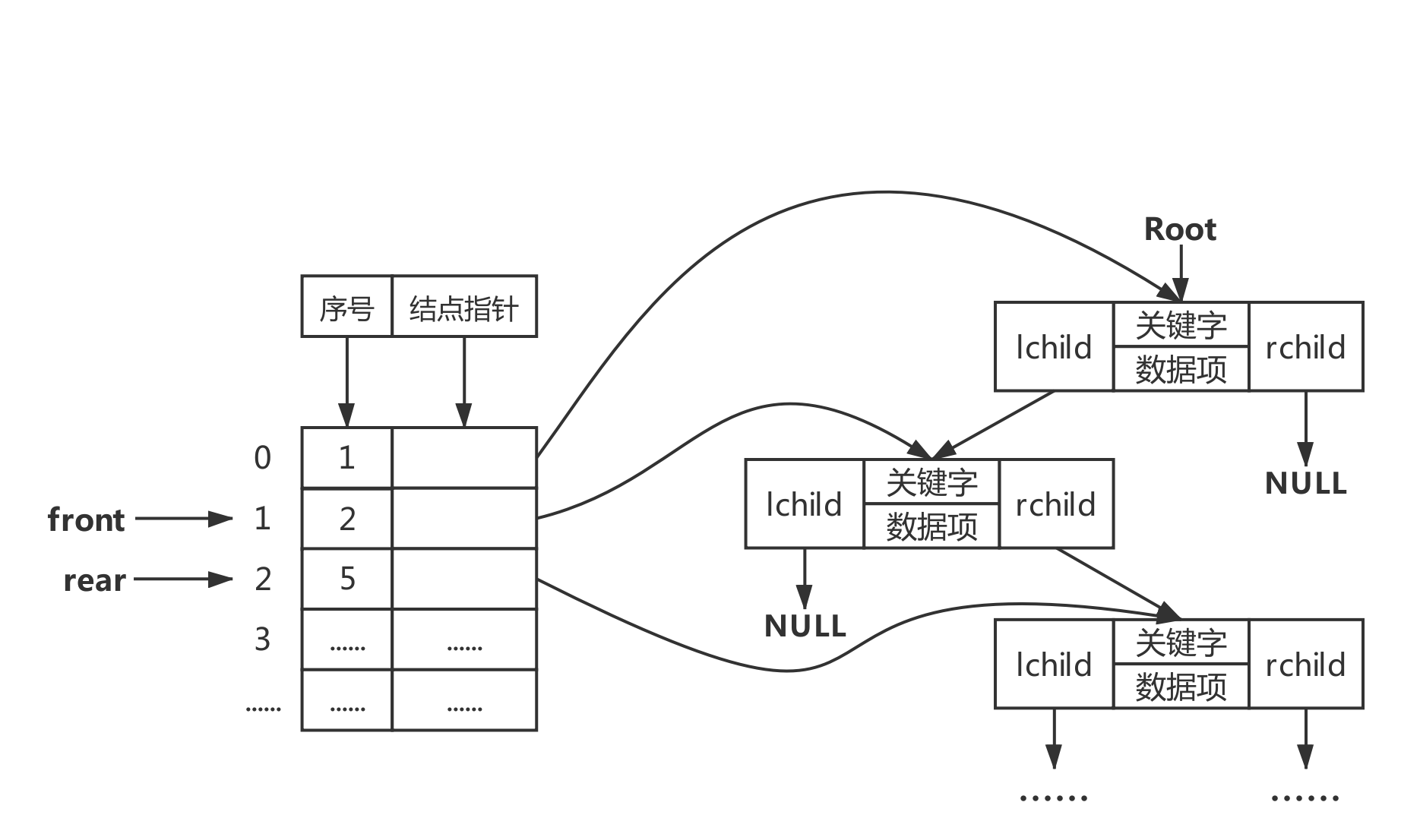


图3-7 二叉树图形化显示物理结构示意图

（17）二叉树命名函数status Assign\_name(BiTrees &T,BiTrees a[],int num)；

算法实现：参数为当前需要命名的二叉树森林结点，以及二叉树森林数组（便于查找是否重复命名），以及数组的长度。先读入字符串作为需要命名的名称，然后遍历数组判断是否重复命名，若重复命名，返回ERROR；若未重复命名则二叉树结点命名，返回OK。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

## 3.4 系统测试

1. 当程序开始运行时，会立刻进入菜单演示界面（菜单界面如图3-8所示）。

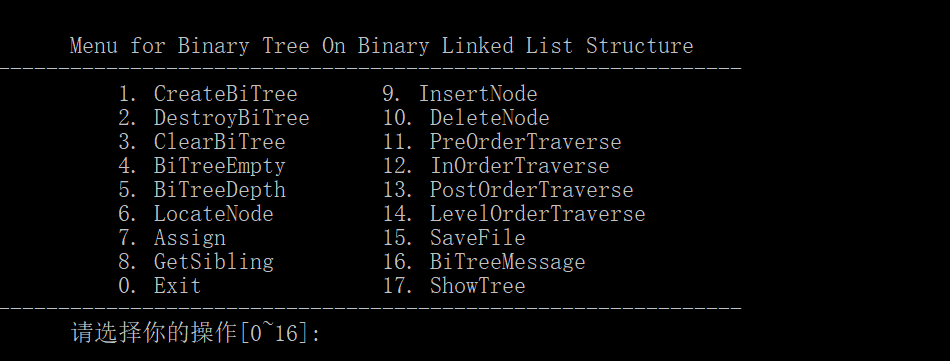


图3-8 演示系统菜单界面图

2. 按照菜单的提示输入需要完成的操作的序号，程序就会进入相对应的操作过程；当系统中没有二叉树存在时，进行操作2-15或17，系统会做出提示显示“当前无二叉树存在，请先创建二叉树”（操作结果图如图3-9所示）。

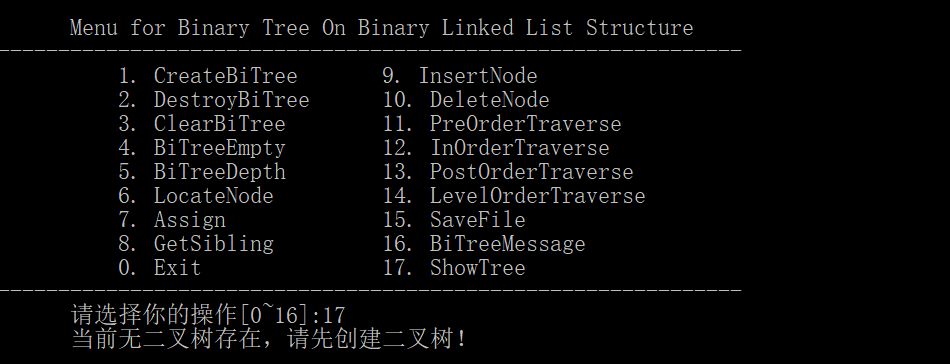


图3-9 系统无二叉树存在时进行2-15或17操作结果图

3. 当系统中存在二叉树时，可以进行所有操作，在进行2-15及17操作时，需要输入二叉树的名称确定需要操作的二叉树。当需要进行多表操作时，若需要查看所有二叉树的信息，则可使用选项16操作，显示二叉树信息。

4. 系统测试计划

使用预先保存的测试样例进行测试，本次测试挑选CreateBiTree、GetSibling、InsertNode、DeleteNode、SaveFile共5个函数进行测试，ShowTree图形化显示二叉树函数将以辅助的形式测试插入、删除等函数结果。本次使用的测试样例为多二叉树（森林）测试样例，样例中包含2个二叉树，其信息如下：

Tree1：

前序遍历序列：21 22 23 24 25 26 27；

中序遍历序列：22 24 23 25 21 26 27；

Tree2：

前序遍历序列：21 22 23 24 25 26 27 28 29 30 31 32 33 34 35；

中序遍历序列：24 23 25 22 27 26 28 21 31 30 32 29 34 33 35；

其中，Tree1为键盘赋值，Tree2为文件赋值。

1. CreateBiTree函数测试：

表3-1 CreateBiTree函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试二叉树 | 程序输入 | 理论输出 | 实际输出 |
| Tree1 | 1. 主界面输入1进入操作；  2. 按提示选择“带空子树的前序遍历序列生成二叉树”；  3. 按提示选择“键盘赋值”；  4. 按提示输入合法序列；  5. 将二叉树命名为“Tree1”； | 二叉树创建成功 | （二叉树创建过程图，如图3-10所示；Tree1二叉树图形化显示如图3-11所示） |
| Tree2 | 1. 主界面输入1进入操作；  2. 按提示选择“带空子树的前序遍历序列生成二叉树”；  3. 按提示选择“文件赋值”；  4. 按提示输入文件名“data.txt”；  5. 将二叉树命名为“Tree2”； | 二叉树创建成功 | （二叉树创建过程图，如图3-12所示；Tree2二叉树图形化显示如图3-13所示） |
| 非法样例1（空子树的数目少于合法数据） | 1. 主界面输入1进入操作；  2. 按提示选择“带空子树的前序遍历序列生成二叉树”；  3. 按提示选择“键盘赋值”；  4. 按提示输入序列； | 输入非法，请重新输入 | 输入序列在4号结点后减少一个空子树（二叉树创建失败过程图，如图3-14所示） |
| 非法样例2（空子树的数目多与合法数据） | 1. 主界面输入1进入操作；  2. 按提示选择“带空子树的前序遍历序列生成二叉树”；  3. 按提示选择“键盘赋值”；  4. 按提示输入序列； | 输入非法，请重新输入 | 输入序列在6号结点后增加一个空子树（二叉树创建失败过程图，如图3-15所示） |

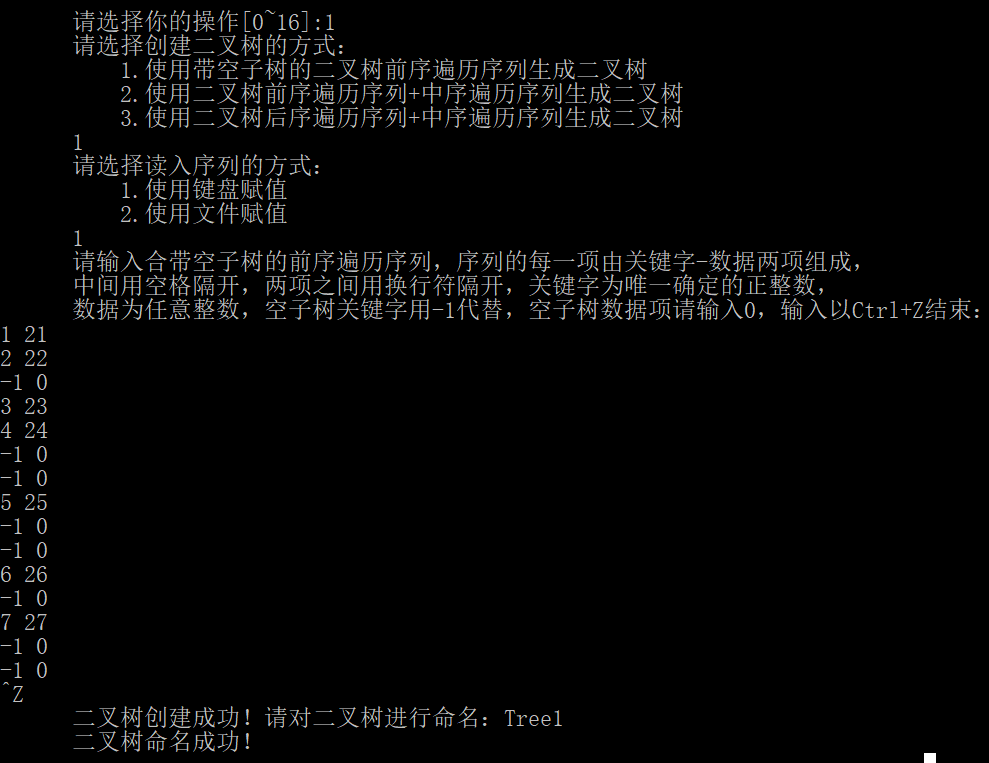


图3-10 二叉树通过键盘赋值创建过程图

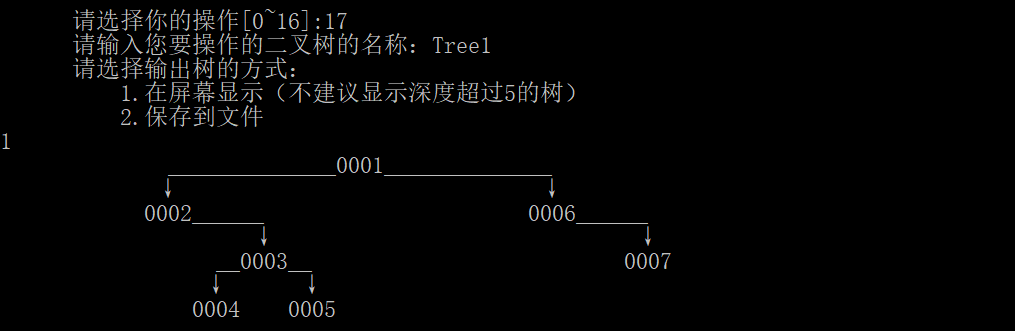


图3-11 二叉树Tree1图形化显示图

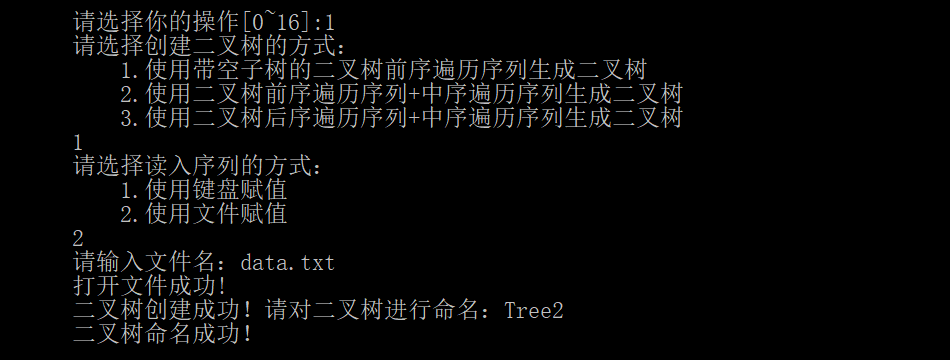


图3-12 二叉树通过文件赋值创建过程图

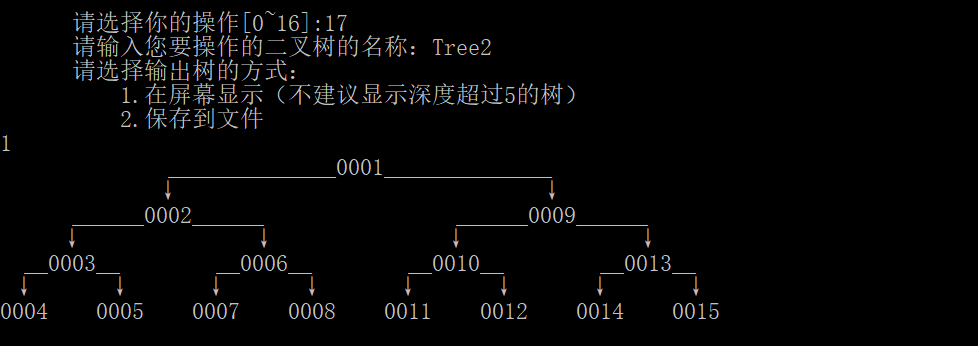


图3-13 二叉树Tree2图形化显示图

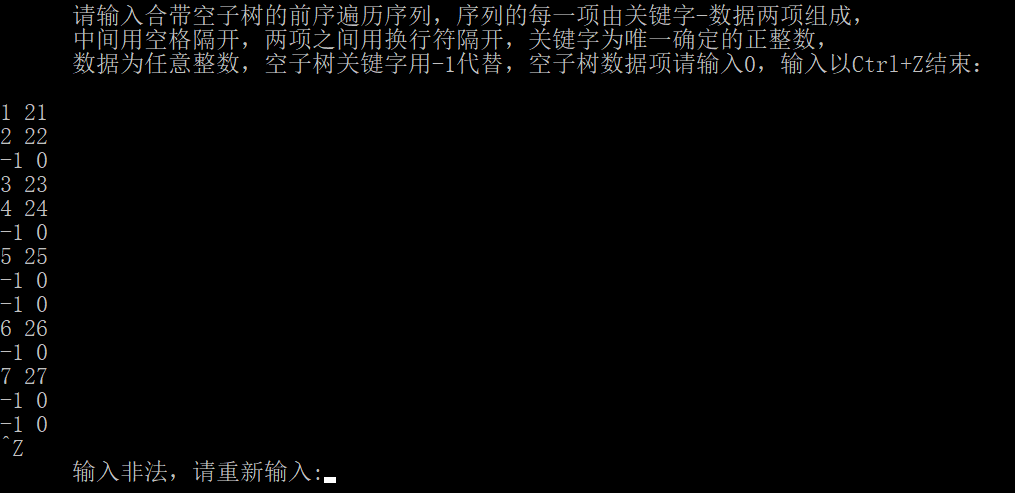


图3-14 非法序列1二叉树创建失败结果图

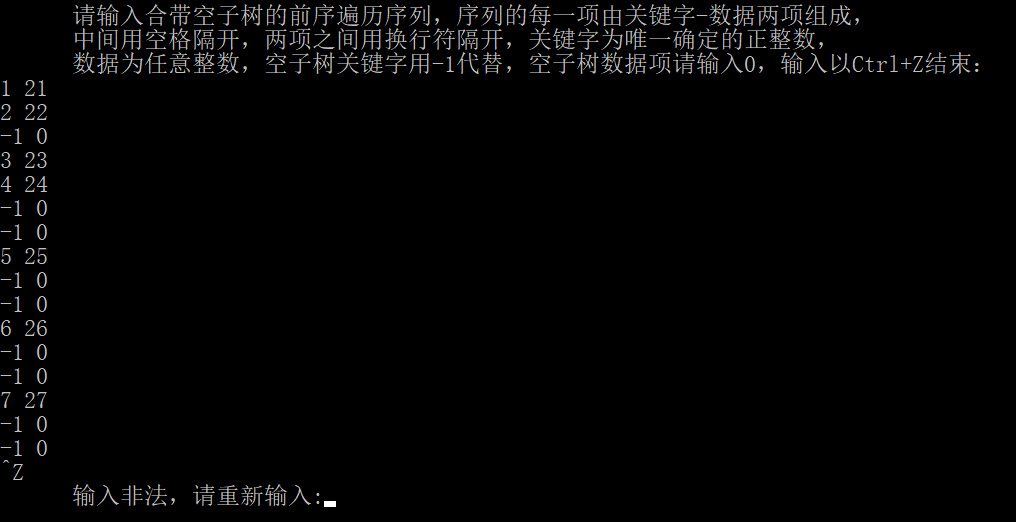
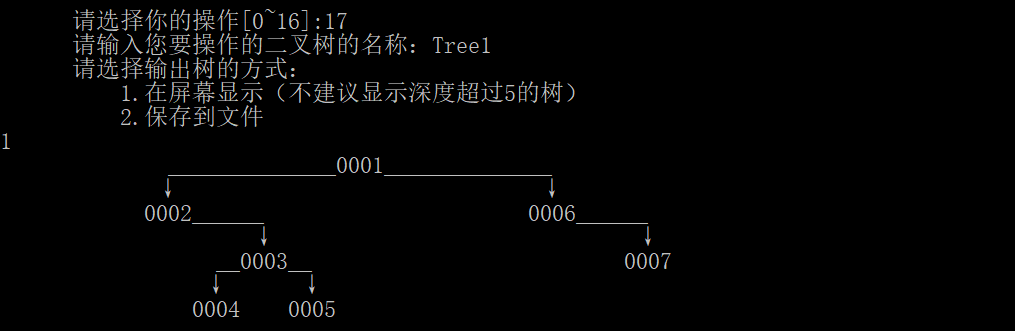


图3-15 非法序列2二叉树创建失败结果图

（2）GetSibling函数测试：

表3-2 GetSibling函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试二叉树 | 程序输入 | 理论输出 | 实际输出 |
| Tree1 | 1. 主界面输入8进入操作；  2. 按提示输入“Tree1”；  3. 按照提示输入“2”作为需要查找兄弟结点的结点的关键字； | 符合条件的结点的关键字为5，数据项为25 | （输出结果图如图3-16-1所示） |
| Tree1 | 1. 主界面输入8进入操作；  2. 按提示输入“Tree1”；  3. 按照提示输入“3”作为需要查找兄弟结点的结点的关键字； | 兄弟结点不存在，操作失败！ | （输出结果图如图3-16-2所示） |
| Tree1 | 1. 主界面输入8进入操作；  2. 按提示输入“Tree1”；  3. 按照提示输入“9”作为需要查找兄弟结点的结点的关键字（该结点不存在）； | 该结点不存在，操作失败！ | （输出结果图如图3-16-3所示） |



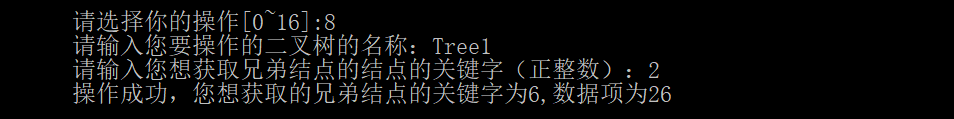
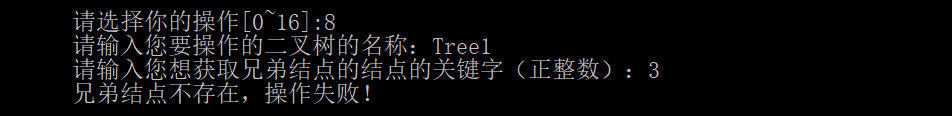


图3-16-1 获得兄弟结点函数测试结果图1

图3-16-2 获得兄弟结点函数测试结果图2

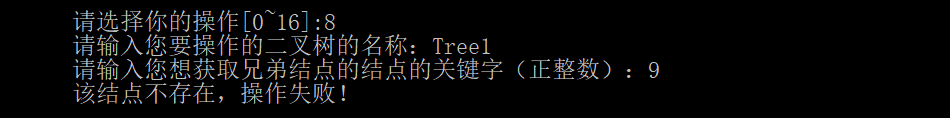


图3-16-3 获得兄弟结点函数测试结果图3

（3）InsertNode函数测试：

表3-3 InsertNode函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试二叉树 | 程序输入 | 理论输出 | 实际输出 |
| Tree1 | 1. 主界面输入9进入操作；  2. 按提示输入“Tree1”；  3. 按提示输入“2”作为插入位点；  4. 按提示输入“0”插入为左结点；  5. 输入“9 29”作为插入结点； | 结点插入成功！ | （输出结果图如图3-17-1所示） |
| Tree1 | 1. 主界面输入9进入操作；  2. 按提示输入“Tree1”；  3. 按提示输入“1”作为插入位点；  4. 按提示输入“1”插入为左结点；  5. 输入“10 30”作为插入结点； | 结点插入成功！ | （输出结果图如图3-17-2所示） |

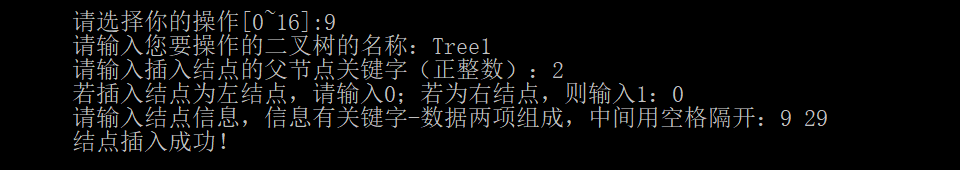


图3-17-1 插入结点函数测试结果图

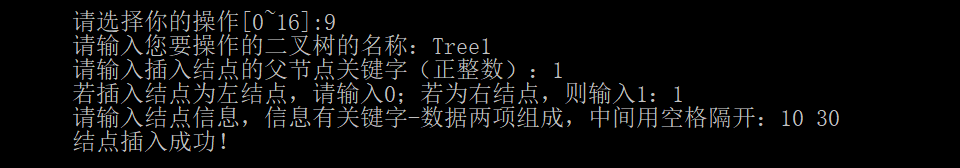
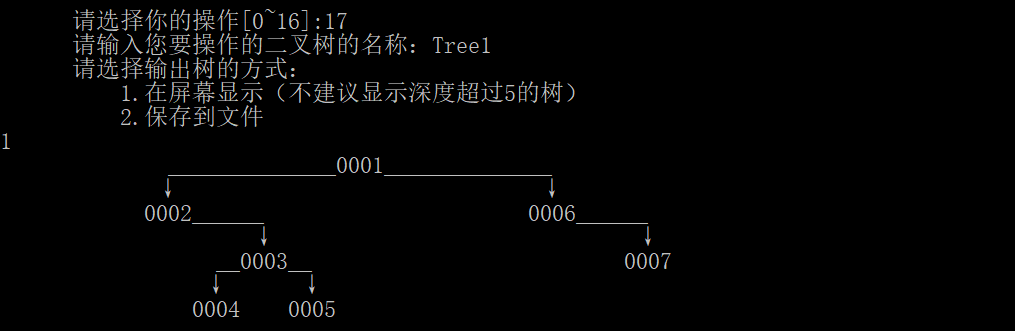


图3-17-2 插入结点函数测试结果图

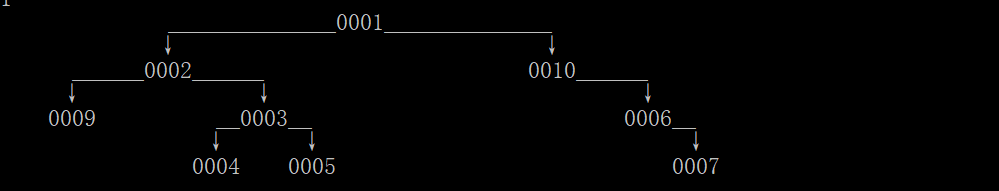
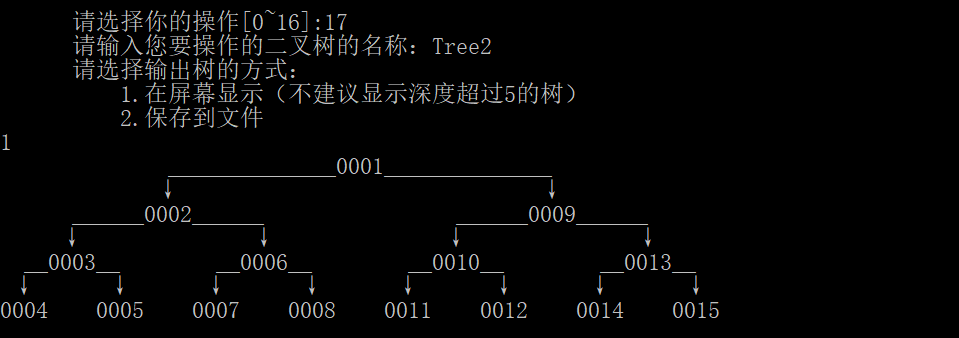


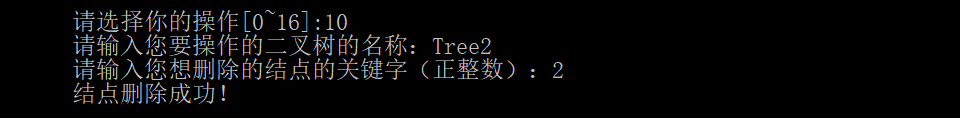
图3-17-3 插入结点函数结果图形化显示图

（4）DeleteNode函数测试：

表3-4 DeleteNode函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试二叉树 | 程序输入 | 理论输出 | 实际输出 |
| Tree2 | 1. 主界面输入10进入操作；  2. 按提示输入“Tree2”；  3. 按提示输入“2”作为删除结点的关键字； | 结点删除成功！ | （输出结果图如图3-18-1所示） |
| Tree1 | 1. 主界面输入10进入操作；  2. 按提示输入“Tree2”；  3. 按提示输入“5”作为删除结点的关键字； | 结点删除成功！ | （输出结果图如图3-18-2所示） |





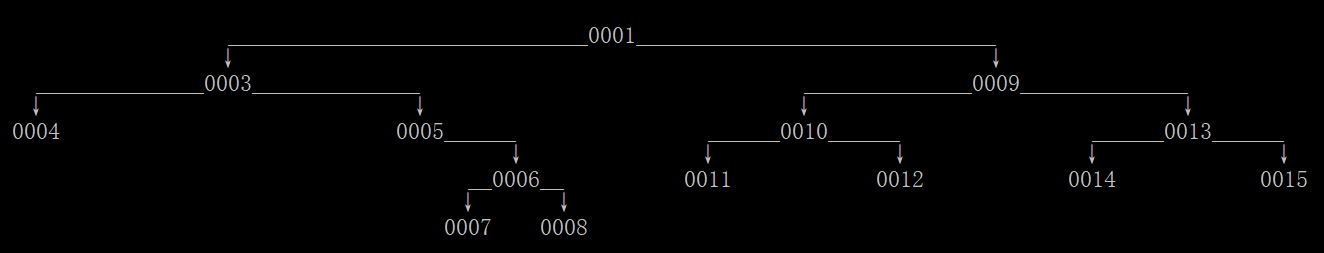
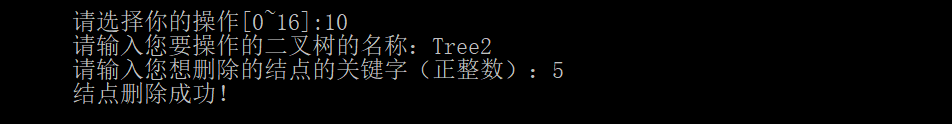


图3-18-1 删除结点左右孩子存在结果图



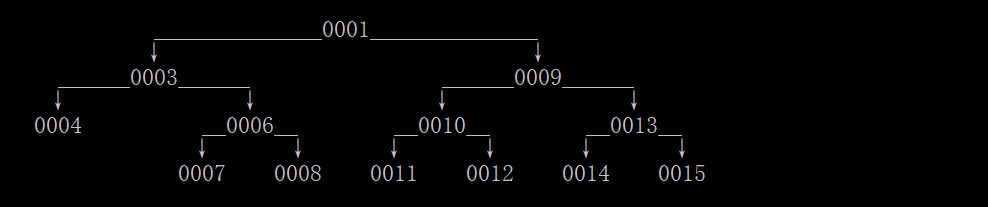


图3-18-2 删除结点仅右孩子存在结果图

（5）SaveFile函数测试：

表3-5 SaveFile函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试二叉树 | 程序输入 | 理论输出 | 实际输出 |
| Tree1 | 1. 主界面输入15进入操作；  2. 按提示输入“Tree1”，按照提示输入“2”创建新文件进行文件保存。  3. 输入文件名“data.txt”； | 保存成功！ | （输出结果图如图3-19所示） |

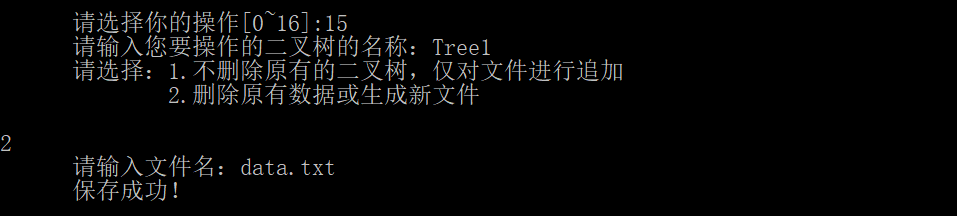


图3-19 文件保存函数测试结果图

## 3.5 实验小结

本次实验的难度较前两次有较大幅度的增加，在本次实验中，使用的语言仍为C语言，同样为了传递数据的方便，使用了引用传递作为传递参数的手段。本次实验完成的任务为，以二叉链表作为物理结构实现二叉树的存储和操作。

本次实验的主函数的编写调用以及菜单界面和菜单操作的设计上沿用了前两次的实验设计的架构。

在本次设计算法和编写代码的过程中，我遇到的问题比前两次都要多一些。在本次实验中，困扰我的问题很多，其中实现最复杂的函数为二叉树的创建函数、删除函数以及二叉树的图形化显示函数，这几个函数的编写都给我造成了较大的麻烦，这三个函数中的创建函数与图形化显示函数为非递归实现，删除结点函数为递归实现，算法实现在上文都有论述，在此就不在赘述。

通过本次实验，我对树的各种操作，其物理结构等方面都有了更加深刻的理解，对非法条件条件的判断也有了一定的体会，可以说收获不少。

# 4基于邻接表的图实现

## 4.1 问题描述

以邻接表作为图的物理结构。其中，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。演示系统需实现多个图管理，以及实现图的文件形式保存。其中，需要设计文件数据记录格式，以高效保存图数据逻辑结构的完整信息；以及设计图文件保存和加载操作合理模式。

其中ElemType为数据元素的类型名，具体含义可自行定义，要求图结点类型为结构型，至少包含二个部分，一个是能唯一标识一个结点的关键字（类似于学号或职工号），另一个是其它部分。采用线性表的方式管理多个图，线性表中的每个数据元素为一个图的基本属性，至少应包含有图的名称。

### 4.1.1 图的基本概念与逻辑结构

图（Graph）是一种较线性表和树更为复杂的数据结构，对一个有n个结点的有限集，它的任意两个结点之间都可能相关。

图的数据逻辑结构定义如下：

ADT Graph {

数据对象V：V是具有相同特性的数据元素的集合，称为顶点集。

数据关系R：

R={VR}

VR={<v,w> | v,w∈V，且P(v,w)，<v,w>表示从v到w的弧，

谓词P<v,w>定义了弧<v,w>的意义或信息 }

}

### 4.1.2 图的基本运算

依据最小完备性和常用性相结合的原则，以函数形式定义了图的12种基本运算。具体运算功能定义和说明如下：

（1）创建图CreateCraph(G,V,VR)；

初始条件：V是图的顶点集，VR是图的关系集。

操作结果：按V和VR的定义构造图G。

（2）销毁图DestroyGraph(G)；

初始条件：图G已存在。

操作结果：销毁图G。

（3）查找顶点LocateVex(G,u)；

初始条件：图G已存在，u是和G中顶点关键字类型相同的给定值。

操作结果：若u在图G中存在，返回关键字为u的顶点位置信息，否则返回其它信息。

（4）顶点赋值PutVex (G,u,value)；

初始条件：图G已存在，u是和G中顶点关键字类型相同的给定值。

操作结果：对关键字为u的顶点赋值value。

（5）获得第一邻接点FirstAdjVex(G, u)；

初始条件：图G已存在，u是G中顶点的位序。

操作结果：返回u对应顶点的第一个邻接顶点位置信息，如果u的顶点没有邻接顶点，否则返回其它表示“空”的信息。

（6）获得下一邻接点NextAdjVex(G, v, w)；

初始条件：图G已存在，v和w是G中两个顶点的位序，v对应G的一个顶点，w对应v的邻接顶点。

操作结果：返回v的（相对于w）下一个邻接顶点的位置信息，如果w是最后一个邻接顶点，返回其它表示“空”的信息。

（7）插入顶点InsertVex(G,v)；

初始条件：图G已存在，v和G中的顶点具有相同特征。

操作结果：在图G中增加新顶点v（保持顶点关键字的唯一性）。

（8）删除顶点DeleteVex(G,v)；

初始条件：图G已存在，v是和G中顶点关键字类型相同的给定值。

操作结果：在图G中删除关键字v对应的顶点以及相关的弧。

（9）插入弧InsertArc(G,v,w)；

初始条件：图G已存在，v、w是和G中顶点关键字类型相同的给定值。

操作结果：在图G中增加弧<v,w>，如果图G是无向图，还需要增加<w,v>。

（10）删除弧DeleteArc(G,v,w)；

初始条件：图G已存在，v、w是和G中顶点关键字类型相同的给定值。

操作结果：在图G中删除弧<v,w>，如果图G是无向图，还需要删除<w,v>。

（11）深度优先搜索遍历DFSTraverse(G)；

初始条件：图G已存在。

操作结果：图G进行深度优先搜索遍历，依次对每个结点仅输出一次。

（12）广度优先搜索遍历BFSTraverse(G)；

初始条件：图G已存在。

操作结果：图G进行广度优先搜索遍历，依次对每个结点仅输出一次。

## 4.2 系统设计

### 4.2.1 数据存储结构与形式

基于邻接表的图的数据物理结构如下：

1. 邻接表的顶点集（表头结点结构体）定义如下：

typedef struct VNode{

VertexType data; //顶点数据，其中包含关键字与数据项

ArcNode \*firstarc; //表头指针

}VNode,AdList[MAX\_VERTEX\_NUM]; //表头结点结构体

2. 邻接表的弧边集（表结点结构体）定义如下：

typedef struct ArcNode{

int adjvex; //所指向结点的位序

struct ArcNode \*nextarc; //后继指针

InfoType \*info; //弧信息项

}ArcNode; //邻接表的表结点结构体

3. 邻接表的顶点数据项以及弧数据项结构体，定义如下（结构体嵌套关系示意图，详见图4-1）：

typedef struct VertexType{

Sign Node\_num; //关键字

ElemType data; //数据项

}VertexType; //结点信息结构体，用于读入和存储结点信息

typedef struct InfoType{

int weight; //权重（本次实验不采用权重）

}InfoType; //弧结点中，弧数据结构体

typedef struct VRType{

Sign Node\_num1; //结点1的关键字

Sign Node\_num2; //结点2的关键字

InfoType data; //弧数据

}VRType; //弧信息结构体，用于读入和存储弧信息

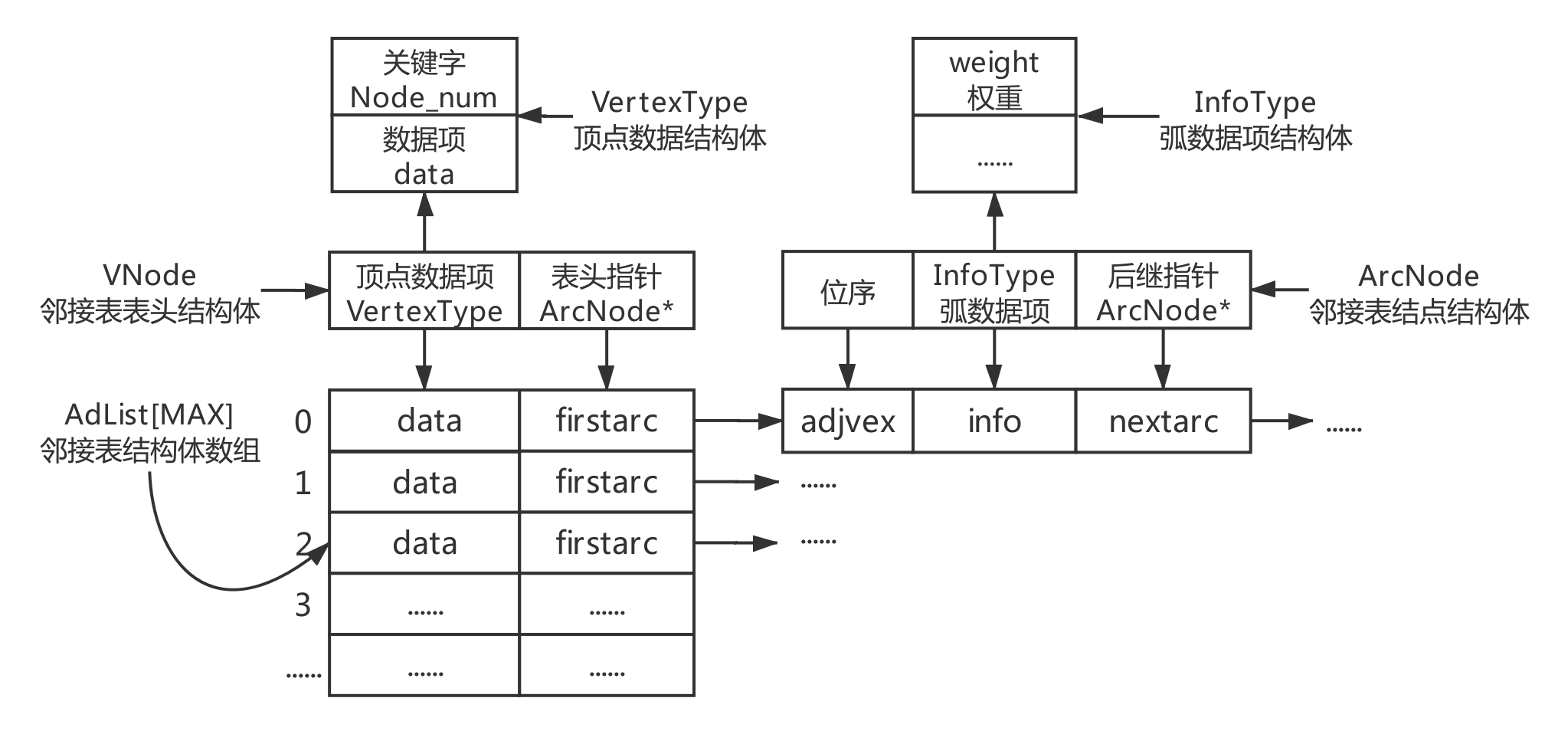


图4-1 结构体嵌套关系示意图

4. 多图管理，通过图的名称属性对图进行操作，只要在主函数中定义一个结构数组即可进行多树管理，结构体定义如下：

typedef struct{

AdList vertices; //直接创建一个邻接表

int vexnum,arcnum; //结点数目以及弧数目

int kind; //记录图的属性，1表示有向图，2表示无向图

char name[30]; //图的名称

}ALGraph; //多图管理结构体

### 4.2.2 总体系统设计

通过WHILE循环与菜单界面，用户通过选择菜单中的选项实现交互，使用OP变量获取用户选择选项值（OP初始化值为1，以便第一次能进入循环）。

进入循环后系统首先显示功能菜单，然后用户输入选择0-14，其中1-12分别代表图的一个基本运算，13选项为图的文件保存，14选项为显示全部图信息（**本选项用于方便查看在存在多个图的情况下，各个图的名称与类型等信息**）。在主函数中通过SWITCH语句对应到相应的函数功能，执行完该功能后BREAK跳出SWITCH语句，继续执行WHILE循环，直至用户输入0退出当前系统。

在进行图的操作时，除了1和14选项（1选项是进行图的生成和初始化，14选项用于查看所有图信息），系统会要求输入需要处理的图的名称（本系统多图操作以图名称作为查找特征），若图不存在会返回菜单界面重新操作（系统结构如图4-2所示）。

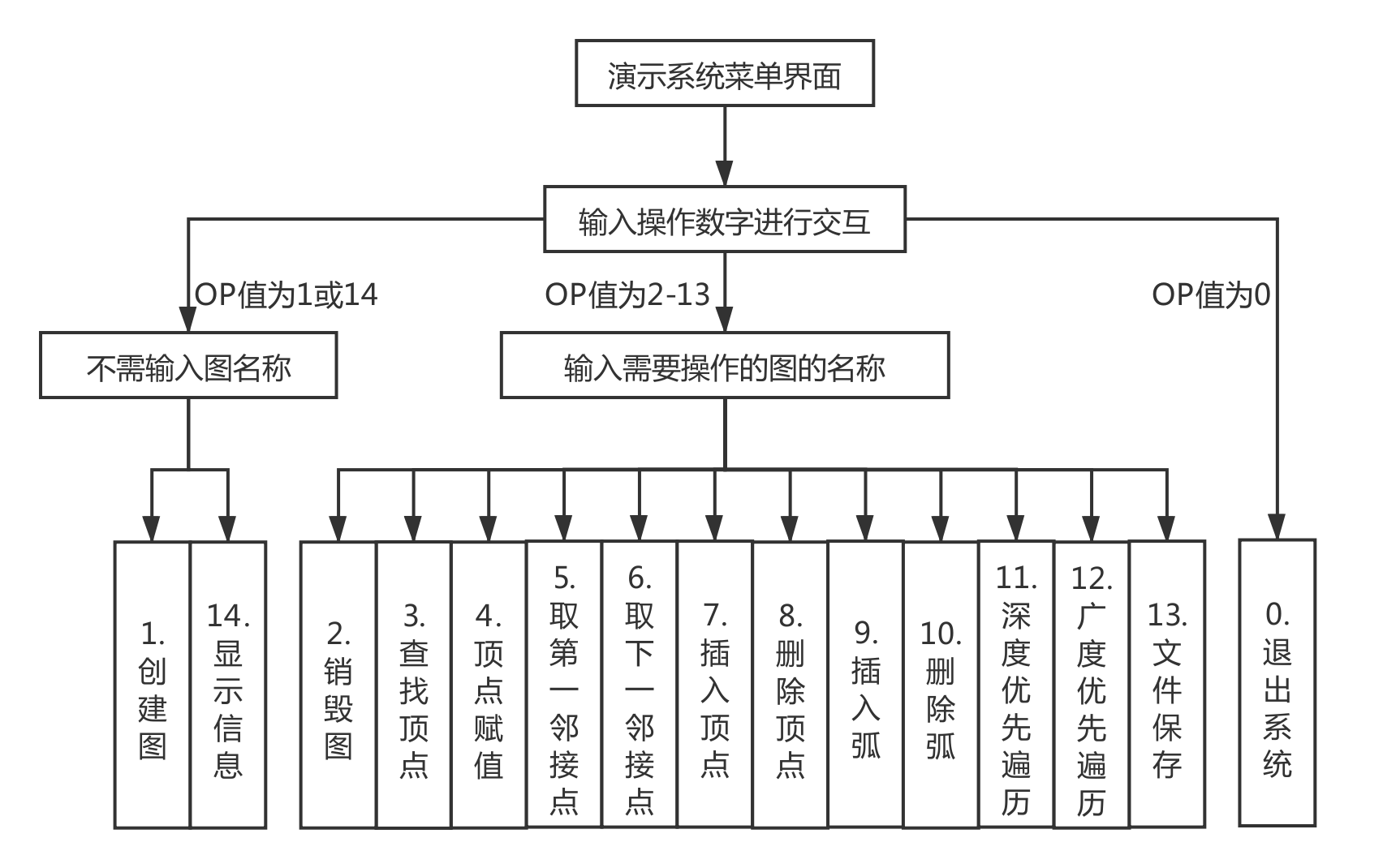


图4-2 系统菜单操作示意图

## 4.3 系统实现

### 4.3.1 编程环境与运行环境描述

编程环境：采用CodeBlocks 16.01编程软件编写。

运行环境：微软Windows 10系统。

### 4.3.2 头文件及预定义常量说明

1.头文件

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

#include <string.h>

2.预定义常量

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define N 100

#define MAX\_VERTEX\_NUM 100 //最大顶点数

#define LIST\_INIT\_SIZE 2000 //最大弧边数

3.类型表达式

typedef int status;

typedef int Sign; //结点关键字元素类型定义

typedef int ElemType; //数据元素类型定义

int visited[MAX\_VERTEX\_NUM]; //访问数组，用于记录结点是否访问

### 4.3.3 算法设计与实现

为方便算法的实现和多图管理与操作，本次算法设计与实现以邻接表来实现结点的储存，具体算法实现如下：

（1）创建图status CreateGraph(ALGraph &G,VertexType V[],VRType VR[])；

算法实现：在创建图函数中，参数V[]传递顶点的信息，其中包含数据项和关键字两项；参数VR[]传递弧的信息，其中包含两个顶点的关键字以及其他弧的信息（本次实验没有其他弧信息）。先通过FOR循环将V[]数组中的结点赋值到邻接表的头结点中，并将每一个头指针firstarc赋值为NULL。然后通过FOR循环遍历VR[]数组中的弧，每次循环生成一个表结点（若为无向图，则生成两个表结点，本程序规定无向图的两条边信息只需要输一条即可），内层再对顶点进行循环，若某个顶点的关键字等于弧信息的第一个关键字，采用首插法将表结点插入到当前顶点的表头结点之后；若某个顶点的关键字等于弧信息的第二个关键字，则将当前表结点的adjvex值赋值为当前顶点的位序。将所有弧遍历完成后，图创建成功（创建图函数流程图，如图4-3所示）。

复杂度分析：时间复杂度T(n) = O(n^2)，空间复杂度S(n) = O(1)。

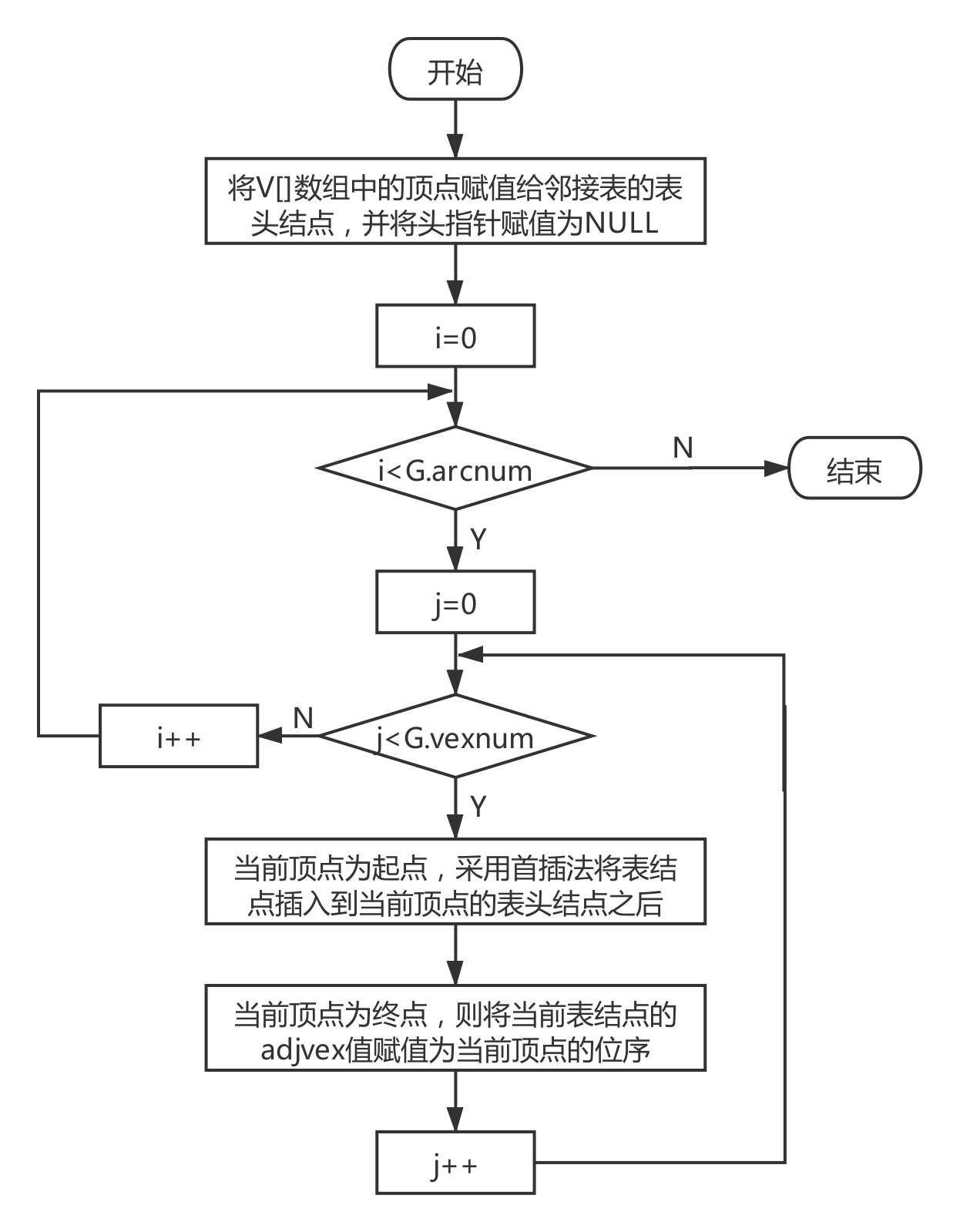


图4-3 创建图函数流程图

（2）销毁图status DestroyGraph(ALGraph &G)；

算法实现：使用FOR循环遍历邻接表头结点，然后用遍历指针p通过WHILE循环释放所有顶点以及其指向的表结点空间，图销毁成功。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（3）查找顶点int LocateVex(ALGraph G,Sign u)；

算法实现：使用FOR循环遍历邻接表头结点，当前结点的关键字与u相同时，返回当前结点的位序，若查找失败，返回-1。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（4）顶点赋值status PutVex(ALGraph &G,Sign u,ElemType value)；

算法实现：算法实现与查找顶点类似，使用FOR循环遍历邻接表头结点，当前结点的关键字与u相同时，对顶点数据项的数据域赋值，将之赋值为value，并返回OK；若赋值失败，返回ERROR。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（5）获取第一邻接点int FirstAdjVex(ALGraph G,int v) ；

算法实现：直接访问位序v所指的顶点的表头结点，返回第一个表结点所指向的顶点的表头结点位序，如果当前位序的表头结点的第一表结点不存在，则返回-1（位序非法判断在进行输入时已经判断，不需要在函数中判断）。

复杂度分析：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（6）获取下一邻接点int NextAdjVex(ALGraph G,int v,int w)；

算法实现：直接访问位序为v的顶点的表头结点，定义遍历指针 p遍历当前顶点的所有表结点，若当前表结点指向的位序为w，则返回下一个表结点所指向的位序，若下一表结点不存在，则返回-1；若遍历所有表结点后仍未找到指向w的表结点，即位序为v的顶点没有指向w的边，则返回-2。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（7）插入顶点status InsertVex(ALGraph &G,VertexType v)；

算法实现：若当前结点数G.vexnum已经达到MAX\_VERTEX\_NUM，即已达到上限，则返回ERROR；否则就直接在邻接表的末尾加上需要插入的顶点，将头结点的指针指向NULL，并修改G.vexnum的数值。

复杂度分析：时间复杂度T(n) = O(1)，空间复杂度S(n) = O(1)。

（8）删除顶点status DeleteVex(ALGraph &G,int v)；

算法实现：删除顶点的操作比较复杂，需要分三个步骤，删除顶点，移动后续所有顶点，并删除所有需要删除的弧，修改其他弧的数值。步骤1，定义遍历指针p遍历，遍历释放位序为v的顶点的表结点空间。步骤2，通过FOR循环将位序为v后的顶点全部向前移动一位。步骤3，遍历所有的表结点，修改表结点的adjvex值或删除表结点；通过2层循环**（虽然是两层循环，但是所有弧边只遍历一遍，所以T(n) = O(n+m)，n为弧边数，m为顶点数）**，外层通过FOR循环遍历所有顶点的头结点，内层使用遍历指针p通过WHILE循环遍历所有表结点，其中，对表结点的操作分两种。**第一，如果当前表结点的adjvex值为v**，即该弧边指向需要删除的顶点，则删除当前表结点；**第二，如果当前表结点的adjvex值大于v**，表示当前弧指向的顶点位序改变，所以需要将adjvex值减小1。当遍历完所有的表结点，删除完成（删除顶点函数流程图，如图4-4所示）。

复杂度分析：时间复杂度T(n) = O(n)，其中n表示顶点数与弧边数中较大的

一个，空间复杂度S(n) = O(1)。

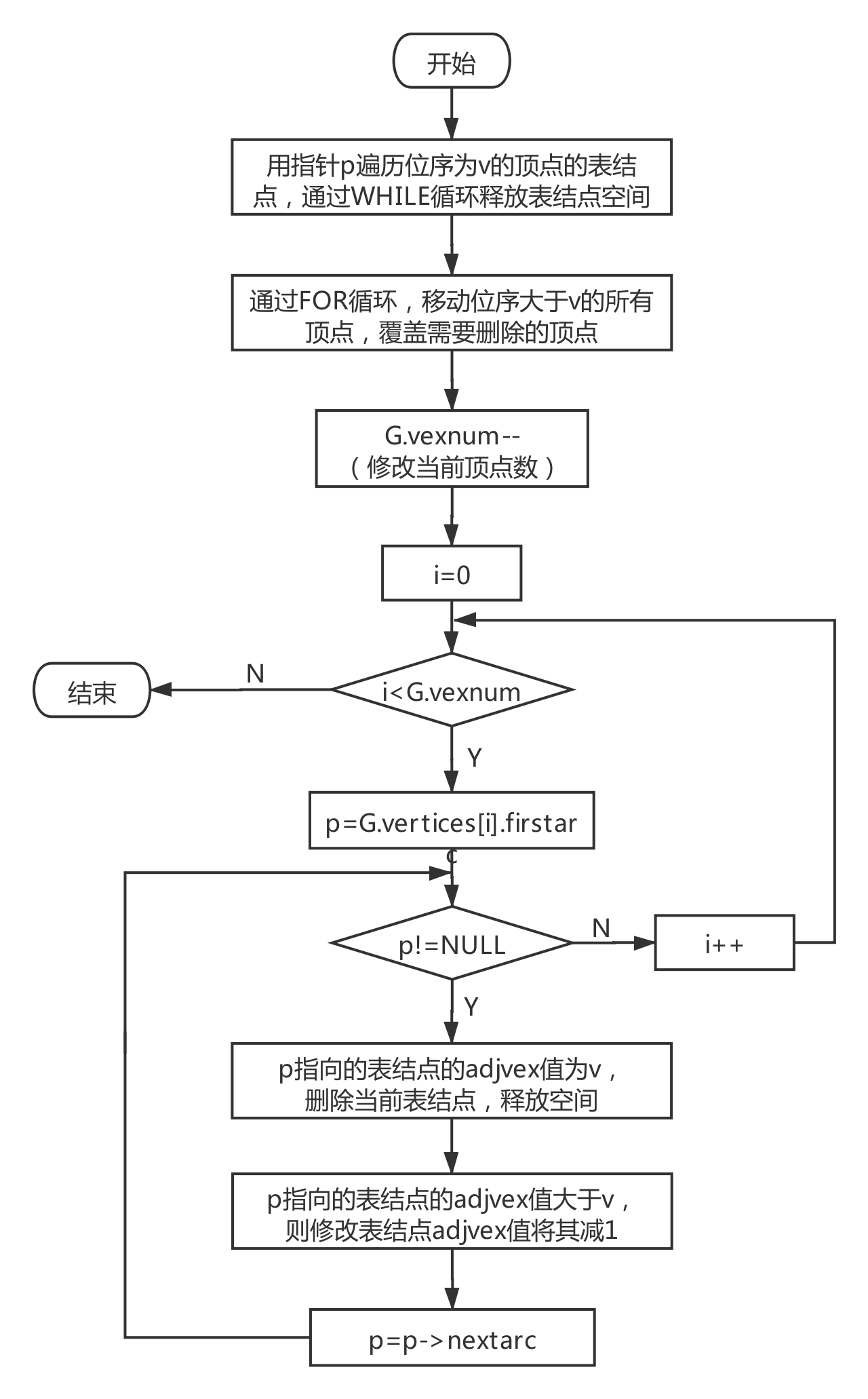


图4-4 删除顶点函数流程图

（9）插入弧status InsertArc(ALGraph &G,int v,int w)；

算法实现：定义一个遍历指针p，遍历位序为v的顶点的表结点，查找需要插入的边是否已经存在，若已经存在，则返回ERROR；否则，用malloc函数生成表结点，并采用首插法插入到头结点之后，返回OK。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（10）删除弧status DeleteArc(ALGraph &G,int v,int w)；

算法实现：删除弧函数的处理和删除结点的操作的一部分相同，不过删除弧比删除结点简单一些。只需定义遍历指针p，遍历位序为v的顶点的表结点，若某个表结点的adjvex值为w，则删除当前表结点，返回OK；若查找所有表结点仍无法找到，则删除失败，返回ERROR。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（11）深度优先遍历status DFSTraverse(ALGraph &G)；

算法实现：深度优先遍历算法分成两个部分，一个为主函数的接口函数，一个为递归访问的处理函数。为防止重复访问顶点，需定义一个visited[]访问数组作为是否访问的标记。在DFSTraverse函数（主函数接口函数）中，首先，先初始化访问数组visited[]，将其元素赋值为FALSE，表示所有顶点未访问；然后遍历所有顶点，若当前结点未访问则调用DFS函数（递归访问处理函数）。在DFS函数中，先将当前结点的访问数组元素值修改为TRUE，并访问输出当前结点；然后用遍历指针p遍历当前结点的所有表结点，若当前表结点所指向的顶点未访问（visited[adjvex]==FALSE），则递归调用DFS函数访问位序为adjvex顶点。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

（12）广度优先遍历status BFSTraverse(ALGraph &G)；

算法实现：定义一个结点数组Q模拟队列，front指向队头，rear指向队尾；同样为防止重复访问顶点，需定义一个visited[]访问数组作为是否访问的标记。算法流程如下：首先，先初始化访问数组visited[]，将其所有元素赋值为FALSE，表示所有顶点未访问；然后，通过FOR循环遍历顶点，若当前结点未访问，则将顶点入队，用遍历指针p遍历当前顶点的所有表结点，将所有表结点指向的顶点入队；然后输出front指向的结点，并将之出队（front++），若队非空（front<rear），则遍历当前队首结点的表结点，重复上诉操作，直到队为空，遍历完成（广度优先遍历BFS函数流程图，如图4-5所示）。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(n)。

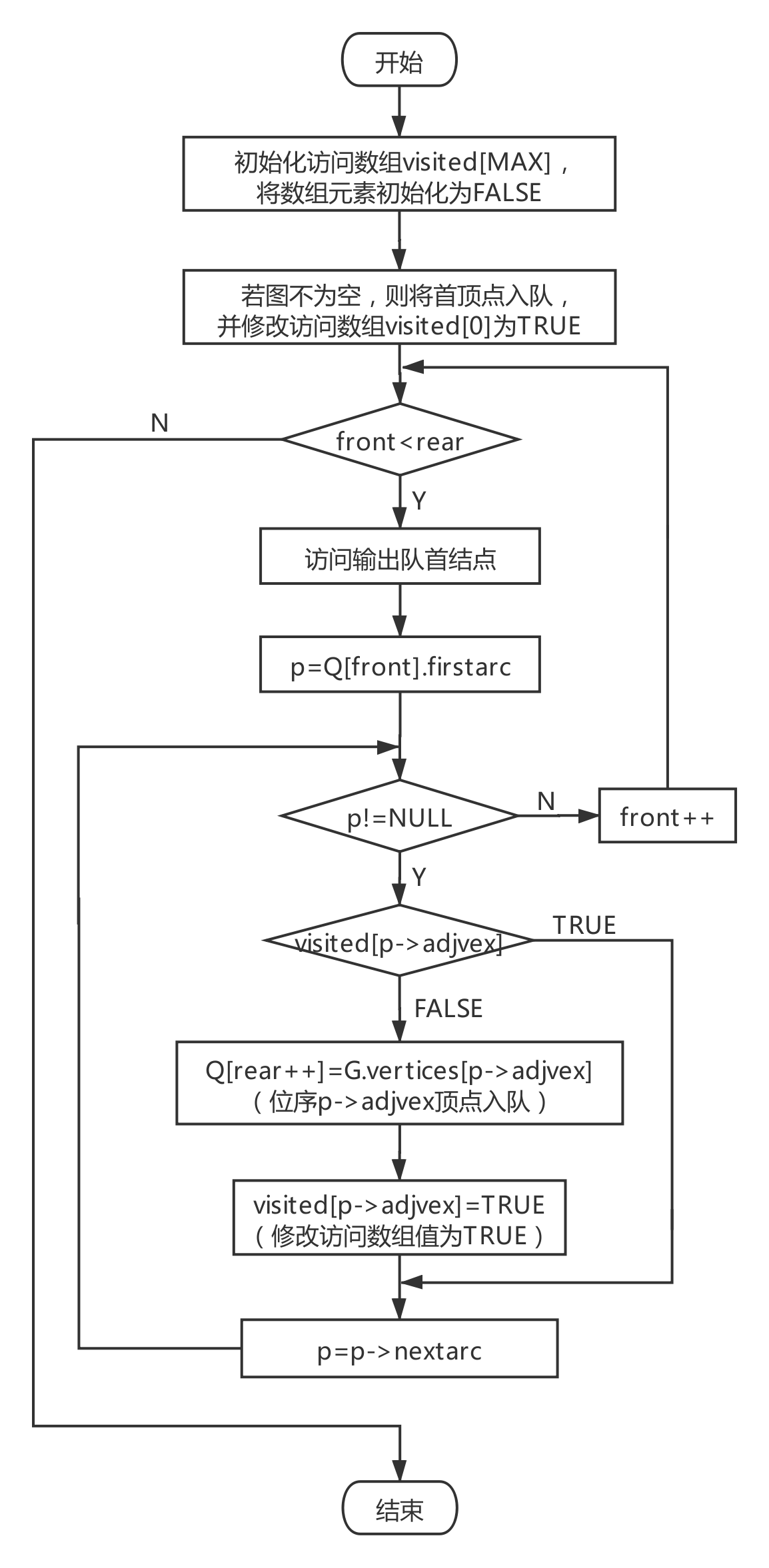


图4-5 广度优先遍历函数流程图

（13）文件保存status SaveFile(ALGraph G)；

算法实现：首先，选择读写文件的方式，选择1选项表示不删除原有的图，只在原有数据之后添加数据，或者选择2选项表示删除原有图数据或生成新文件；然后使用fopen函数创建或打开文件，然后用fputs函数和fprintf函数输出数据，最后关闭文件。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

（14）图命名status Assign\_name(ALGraph &G,ALGraph a[],int num) ；

算法实现：参数为当前需要命名的图，以及图数组（便于查找是否重复命名），以及数组的长度。先读入字符串作为需要命名的名称，然后遍历数组判断是否重复命名，若重复命名，返回ERROR；若未重复命名则图结点命名，返回OK。

复杂度分析：时间复杂度T(n) = O(n)，空间复杂度S(n) = O(1)。

## 4.4 系统测试

1. 当程序开始运行时，会立刻进入菜单演示界面（菜单界面如图4-6所示）。

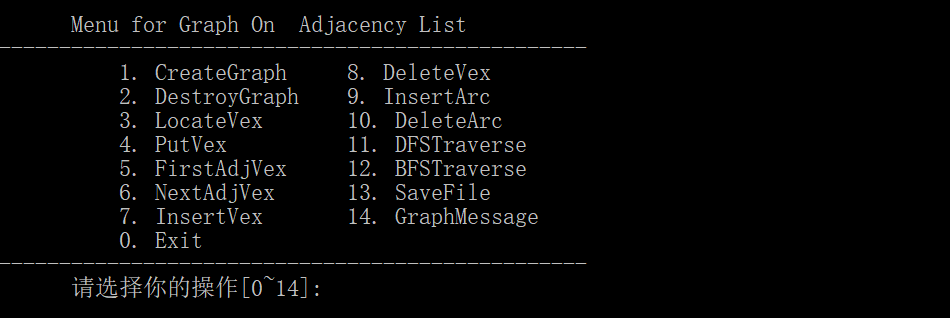


图4-6 演示系统菜单界面图

2. 按照菜单的提示输入需要完成的操作的序号，程序就会进入相对应的操作过程；当系统中没有图存在时，进行操作2-13，系统会做出提示显示“当前无图存在，请先创建图”（操作结果图如图4-7所示）。



图4-7 系统无图存在时进行2-13操作结果图

3. 当系统中存在图时，可以进行所有操作，在进行2-13操作时，需要输入图的名称确定需要操作的图。当需要进行多图操作时，若需要查看所有图的信息，则可使用选项14操作，显示图信息。

4. 系统测试计划

使用预先保存的测试样例进行测试，本次测试挑选NextAdjVex、DeleteVex、InsertArc共3个函数进行测试，文件保存函数，DFS（深度优先遍历）函数以及BFS（广度优先遍历）函数将以辅助的形式测试插入弧、删除顶点等函数结果。本次使用的测试样例为多图测试样例，样例中包含2个图，一个有向图以及一个无向图，其信息如下：

Graph1（有向图）：

顶点数为5，关键字为：1，2，3，4，5；

对应顶点数据项为：21，22，23，24，25；

弧边数为9，弧的起点与终点关键字分别为：

(1,2), (1,4), (2,1), (2,3), (3,4), (3,5), (4,5), (5,1), (5,2)；

Graph 2（无向图）：

顶点数为6，关键字为：1，2，3，4，5，6；

对应顶点数据项为：31，32，33，34，35，36；

弧边数为8，弧的两个端点（无向图）关键字分别为：

(1,2), (1,4), (1,5), (2,3), (2,4), (3,4), (3,6), (5,6)；

其中，Graph 2为键盘赋值，Graph1为文件赋值（图信息显示如图4-8所示）。

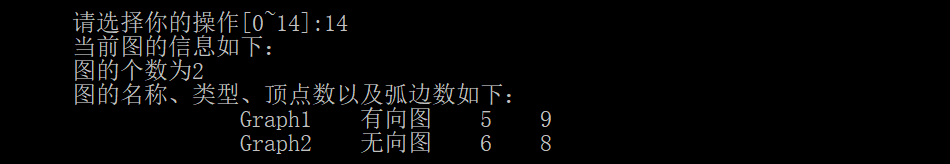


图4-8 图信息显示结果图

（1）NextAdjVex函数测试：

表4-1 NextAdjVex函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试图 | 程序输入 | 理论输出 | 实际输出 |
| Graph1 | 1. 主界面输入6进入操作；  2. 按提示输入“Graph1”；  3. 按提示输入顶点的位序“2”；  4. 按提示输入想要获取下一邻接点的表结点所指的位序“3”； | 操作成功，下一个邻接点的位序为4,数据项为25 | （函数结果图，如图4-9-1所示） |
| Graph1 | 1. 主界面输入6进入操作；  2. 按提示输入“Graph1”；  3. 按提示输入顶点的位序“2”；  4. 按提示输入想要获取下一邻接点的表结点所指的位序“4”； | 下一邻接点不存在，操作失败！ | （函数结果图，如图4-9-2所示） |
| Graph1 | 1. 主界面输入6进入操作；  2. 按提示输入“Graph1”；  3. 按提示输入顶点的位序“2”；  4. 按提示输入想要获取下一邻接点的表结点所指的位序“1”； | 位序为2的顶点不含有位序为1的邻接点，操作失败！ | （函数结果图，如图4-9-3所示） |

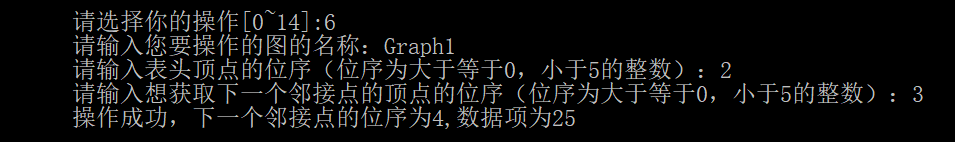


图4-9-1 获取下一邻接点函数样例1测试结果图

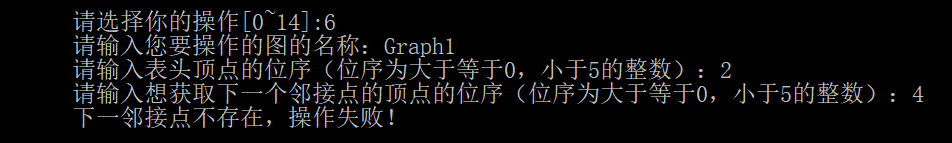


图4-9-2 获取下一邻接点函数样例2测试结果图

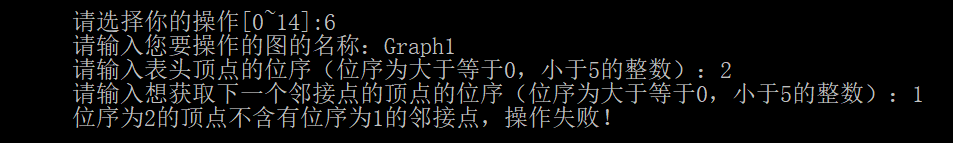
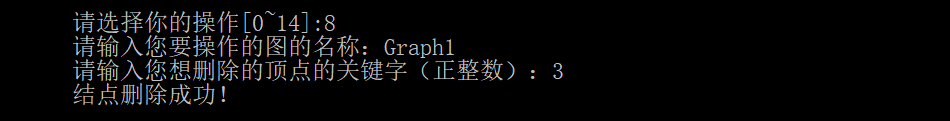


图4-9-3 获取下一邻接点函数样例3测试结果图

（2）DeleteVex函数测试：

表4-2 DeleteVex函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试图 | 程序输入 | 理论输出 | 实际输出 |
| Graph1 | 1. 主界面输入8进入操作；  2. 按提示输入“Graph1”；  3. 按提示输入顶点的关键字“3”； | 结点删除成功！ | （函数结果图，如图4-10-1所示） |
| Graph1 | 1. 主界面输入8进入操作；  2. 按提示输入“Graph1”；  3. 按提示输入顶点的关键字“6”； | 需要删除的顶点不存在，操作失败！ | （函数结果图，如图4-10-2所示） |



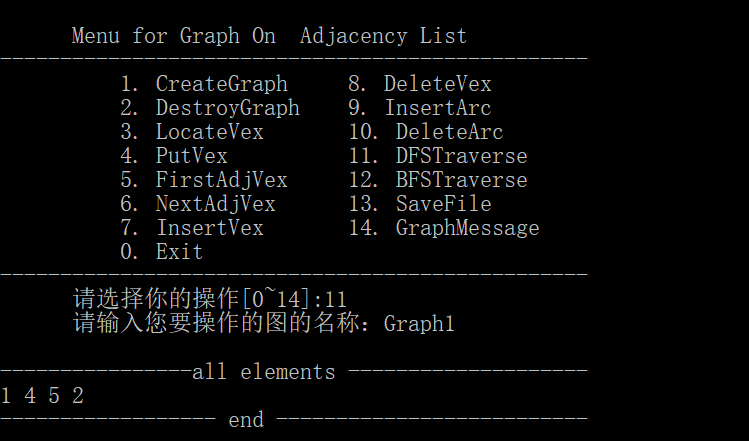
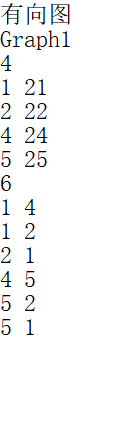
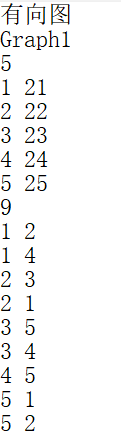


图4-10-1 删除顶点函数样例1测试结果图

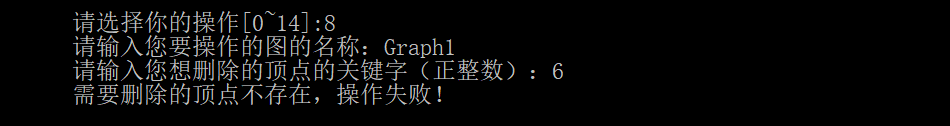


图4-10-2 删除顶点函数样例2测试结果图

（3）InsertArc函数测试：

表4-3 InsertArc函数测试用例表

|  |  |  |  |
| --- | --- | --- | --- |
| 测试图 | 程序输入 | 理论输出 | 实际输出 |
| Graph2 | 1. 主界面输入9进入操作；  2. 按提示输入“Graph2”；  3. 按提示输入两个顶点的关键字“2”和“3”； | 弧边已存在，请勿重复插入！ | （函数结果图，如图4-11-1所示） |
| Graph2 | 1. 主界面输入9进入操作；  2. 按提示输入“Graph2”；  3. 按提示输入两个顶点的关键字“2”和“6”； | 弧插入成功！ | （函数结果图，如图4-11-2所示） |

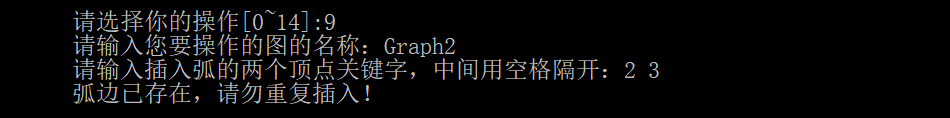
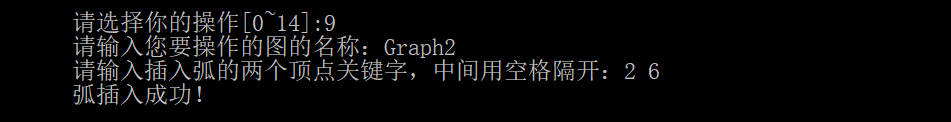


图4-11-1 插入弧函数样例1测试结果图



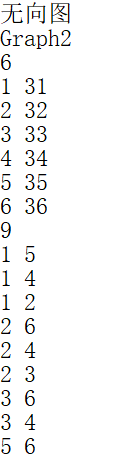
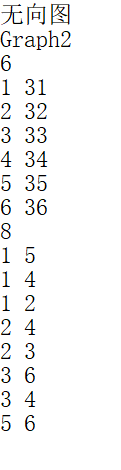


图4-11-2 插入弧函数样例2测试结果图

## 4.5 实验小结

本次实验的难度较上一次实验（二叉树）有所下降，在本次实验中，使用的语言仍为C语言，同样为了传递数据的方便，使用了引用传递作为传递参数的手段。本次实验完成的任务为以邻接表作为物理结构实现图的存储和操作。

本次实验的主函数的编写调用以及菜单界面和菜单操作的设计上沿用了前三次的实验设计的架构。在本次设计算法和编写代码的过程中，我遇到的问题主要出在顶点删除函数以及深度优先遍历算法（DFSTraverse函数+DFS函数）上，这两个函数的编写给我造成了较大的麻烦。顶点删除函数给我最大的困扰在于其处理的繁琐性，由于删除顶点的同时不仅需要移动结点，也需要删除初始点和终点为该顶点的弧，除此由于结点的移动，导致需要修改大量表结点的数值，所以处理起来颇为麻烦；深度优先遍历（DFS）算法的问题在于需要栈进行控制，为减小工作量，采用递归的方式编写，以利用系统栈来控制顶点的输出。

通过本次实验，我对图的各种操作有了更加深刻的理解，收获颇丰。