# 1基于顺序存储结构的线性表实现

## **问题描述**

用顺序表的物理结构储存类型为ElemType的元素，并实现多表管理。

要求要求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线性表的文件形式保存。其中，①需要设计文件数据记录格式，以高效保存线性表数据逻辑结构(D,{R})的完整信息；②需要设计线性表文件保存和加载操作合理模式。

**1.1.1需要完成的功能**

实验需要对每一个顺序表实现创建，销毁，清空，插入等相关操作及文件的读写等。

**1.1.2实验要求**

通过实验达到⑴加深对线性表的概念、基本运算的理解；⑵熟练掌握线性表的逻辑结构与物理结构的关系；⑶物理结构采用顺序表,熟练掌握线性表的基本运算的实现。

**1.2系统设计**

**1.2.1 系统的总体设计**

系统实现顺序表的基本操作功能，系统总体模块设计如图1-1。

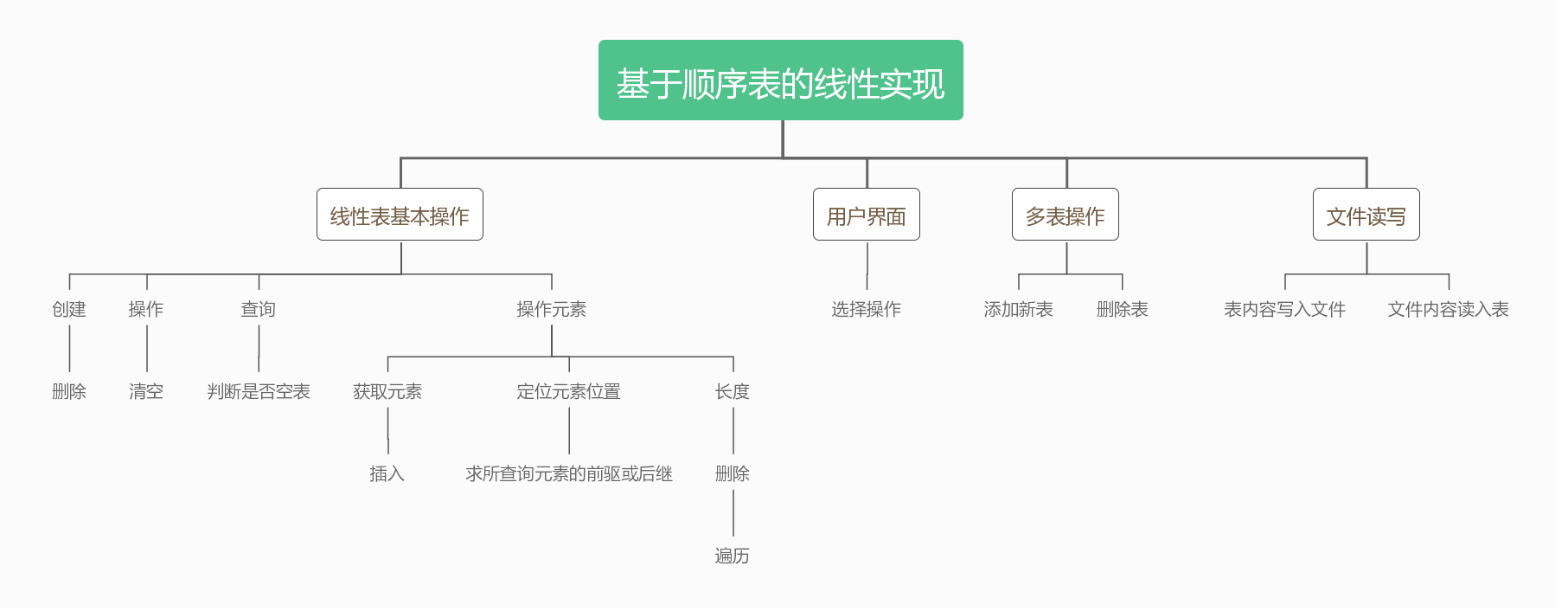


图1-1 系统模块设计

一共保存了三个文件。common.h文件用来定义公用常量（如代码1-1所示）和函数。sqlist.h文件用来定义顺序表结构体，如代码1-2所示，此外，还有对顺序表进行相关操作的函数声明和表的相关定义。main.c文件用来实现线性表相关函数，文件读写函数，系统界面显示函数和用来调用其他函数的main函数。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <malloc.h>  /\*书本第10页，预定义常量和类型\*/  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  typedef int status;  typedef int ElemType; //数据元素类型定义  status equal(ElemType x,ElemType y){  return (x-y);  }  #define LIST\_INIT\_SIZE 100 //初始分配量  #define LISTINCREMENT 10 //分配增量 |

代码1-1

|  |
| --- |
| typedef struct SqList{ //顺序表（顺序结构）的定义  ElemType\* elem; //存储空间基址  int length; //当前长度  int listsize;//当前分配的存储容量  }SqList; |

代码1-2

1.2.2多表设计

多表管理的实现是通过定义一个SqList的结构类型数组L[10]，详情如代码段1-3。利用一个循环，在每一次利用菜单界面初始化一个顺序表并进行完相应操作之后，由用户判断是否继续创建下一个表。

|  |
| --- |
| SqList L[10];  int count=0,op=1;//count用于计数，即多表个数  char c;  //初始化线性表  while(op)  {  PrintMune(&L[count],count);  count++;  printf("继续创建下一个线性表? [Y/N]\n");  getchar();op=((c=getchar())=='Y')?1:0; //由用户确定需要创建线性表的数量  }  printf("欢迎下次再使用本系统！\n"); |

代码段1-3

op可以理解为一个操作数，每次打印菜单界面对一个顺序表完成操作退出后，op都会变为0，此时就用c获取用户意愿，c==Y时，op=1，循环判断条件为真，继续创建下一个表，否则为0，不再创建下一个。

1.2.3多表管理菜单操作流程

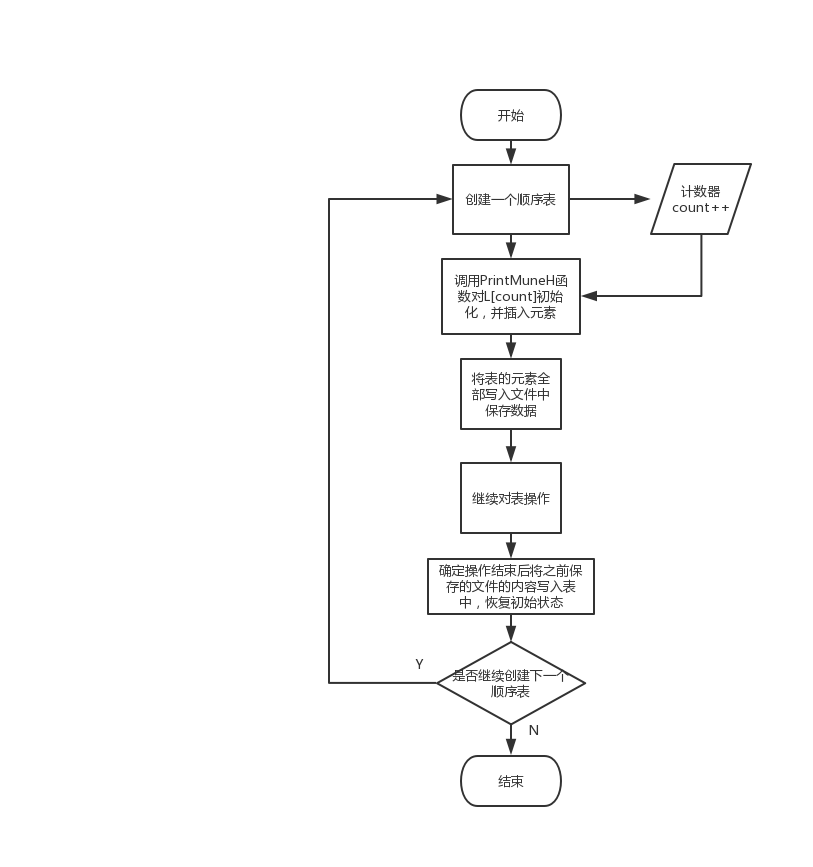


图1-2多表管理系统菜单操作流程

**1.2.4函数设计**

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初始化表、销毁表、清空表、判定空表、求表长和获得元素等12种基本运算，具体运算功能定义如下。

（1）初始化表：函数名称是InitList(SqList\* L)；

初始条件：线性表L不存在；

算法设计：动态分配空间，分配成功，则长度为0，初始容量为初始分配量。若分配不到，则返回失败。

操作结果：构造一个空的线性表。

（2）销毁表：函数名称是DestroyList(SqList\* L)；

初始条件：线性表L已存在；

算法设计：表的容量和长度均为零，释放掉L->elem，并定义为NULL；

操作结果：销毁线性表L。

（3）清空表：函数名称是ClearList(SqList\* L)；

初始条件：线性表L已存在；

算法设计：长度为0，返回OK

操作结果：将L重置为空表。

（4）判定空表：函数名称是ListEmpty（SqList\* L)；

初始条件：线性表L已存在；

算法设计：长度为0，则为空表，返回TRUE；否则，返回FALSE

操作结果：若L为空表则返回TRUE,否则返回FALSE。

（5）求表长：函数名称是ListLength(SqList\* L)；

初始条件：线性表已存在；

算法设计：返回L->length；

操作结果：返回L中数据元素的个数。

（6）获得元素：函数名称是GetElem(SqList\* L,int i,ElemType\* e)；

初始条件：线性表已存在，1≤i≤ListLength(L)；

算法设计：首先判断输入的i值是否合法，若i<1或者i>Length则返回ERROR；否则用e返回L.elem[i-1]元素；

操作结果：用e返回L中第i个数据元素的值。

（7）查找元素：函数名称是LocateElem(SqList\* L,ElemType e,int (\*compare)(ElemType x,ElemType y))；

初始条件：线性表已存在；

算法设计：首先判断输入的i值是否合法，然后用equal函数循环遍历比较元素，当比较结果为0时返回该元素，否则返回失败；

操作结果：返回L中第1个与e满足关系compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。

（8）获得前驱：函数名称是PriorElemSqList\* L,ElemType cur,ElemType \*pre\_e)；

初始条件：线性表L已存在；

算法设计：调用locatelist函数循环遍历找到查找元素的位置，在用pre\_e返回该位置的前一个位置的元素

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：函数名称：NextElem(SqList\* L,ElemType cur,ElemType\* next\_e)；

初始条件：线性表L已存在；

算法设计：利用locatelist函数找到查找元素的位置，在用next\_e这个位置元素的后继元素；

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：函数名称是ListInsert(SqList\* L,int i,ElemType e)；

初始条件：线性表L已存在，1≤i≤ListLength(L)+1；

算法设计：首先判断表的长度是否大于等于表的容量，如是，则重新分配增量，表的新地址等于重新分配返回的地址，容量自加增量；否则将插入位置后的元素从表尾元素开始全部后移，长度加一；

操作结果：在L的第i个位置之前插入新的数据元素e。

（11）删除元素：函数名称是ListDelete(SqList\* L,int i,ElemType\* e)；

初始条件：线性表L已存在且非空，1≤i≤ListLength(L)；

算法设计：首先判断输入要删除的位置是否合法，合法则先用e返回，在将删除位置的后继元素全部前移。否则，返回失败；

操作结果：删除L的第i个数据元素，用e返回其值。

（12）遍历表：函数名称是ListTraverse(SqList\* L)

初始条件：线性表L已存在；

算法设计：表不存在，返回失败，否则循环遍历每一个元素并输出；

操作结果：输出所有元素；

（13）写入文件：函数名称是SaveList(SqList\* L)

初始条件是：线性表L已经存在；

算法设计：输入保存文件的名称，判断打开文件是否成功，是，则用fwrite

函数将顺序表写入文件中，否则返回失败；

操作结果：将顺序表中的元素内容写入文件中；

（14）读取文件：函数名称是LoadList(SqList\* L)

初始条件：线性表已经存在；

算法设计：输入需要载入的文件的名称，判断是否可以打开，若可以，则将文件的内容用fread写入顺序表，返回成功，否则返回失败；

操作结果：顺序表恢复到初始未修改状态。

1.3顺序演示实现与测试

1.3.1 功能实现

本次实验使用的环境配置如下：

操作系统版本：Windows 10 专业版

编译器及其版本：GNN GCC Compiler

编程环境：CodeBlocks ver17.12

实验的完整代码参见附录A。

测试时通过用户界面对线性表进行操作。用户界面你参见图1-3。

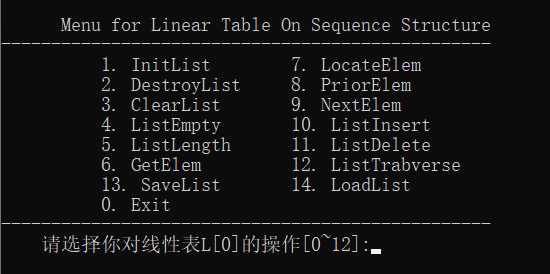


图1-3演示程序用户界面菜单

1.3.2功能测试

测试中使用的测试数据

表1-1 测试用例

|  |  |
| --- | --- |
| 表名称 | 表中元素（按顺序排列） |
| L[0] | 0，1，2，3，4 |
| L[1] | 1，2，4，4 |
| L[2] | 2，3，5 |

1) 测试函数：InitList

测试步骤及结果如表 1-2 所示

表1-2 InitList 函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 创建表 | 表L[0]不存在，输入1 | 建立表L[0] | 输出“线性表创建成功！” |

2) 测试函数：DestroyList

测试步骤及结果如表 1-3 所示

表1-3 DestroyList函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 销毁表 | 表L[0]存在，输入2 | 销毁表L[0] | 输出“线性表销毁成功” |
| 表L[0]不存在，输入2 | 操作失败 | 输出“表不存在，无法销毁！” |

3) 测试函数：ClearList

测试步骤及结果如表 1-4 所示

表1-4 ClearList函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 清空表内元素 | 表L[0]存在时，输入3 | 表L[0]仍然存在，但是表内元素为空 | 输出“线性表清空成功！” |
| 表L[0]不存在时，输入3 | 操作失败 | 输出“表不存在，清空失败！” |

4) 测试函数：ListEmpty

测试步骤及结果如表 1-5 所示

表1-5 ListEmpty函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 判断表是否为空 | 表L[0]存在且有元素时，输入4 | 表L[0]不为空 | 输出“线性表是空表！” |
| 表L[0]存在但没有元素时，输入4 | 表L[0]为空 | 输出“线性表不是空表！” |
| 表L[0]不存在时，输入4 | 操作失败 | 输出“线性表不存在！” |

5) 测试函数：ListLength

测试步骤及结果如表 1-6所示

表1-6 ListLength函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 输出表长 | 表L[0]存在时，输入5 | 返回L[0]表长 | 输出“线性表的长度为：5” |
| 表L[0]不存在时，输入5 | 操作失败 | 输出“线性表不存在！” |

6) 测试函数：GetElem

测试步骤及结果如表 1-7 所示

表1-7 GetElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 获得元素 | 表L[0]存在且有元素时，输入6 1 | 得到表L[0]的第一个元素 | 输出“表中第1个元素的值为0：” |
| 表L[0]存在其有元素时，输入6 2 | 得到第二个元素 | 输出“表中第2个元素的值为1” |
| 表L[0]存在其有元素时，输入6 5 | 得到尾元素 | 输出“表中第5个元素的值为4” |
| 表L[0]存在且有元素，输入6 6 | 操作失败 | 输出“输入的数据不合法!” |
| 表L[0]不存在输入6 1 | 操作失败 | 输出“输入的数据不合法!” |

7) 测试函数：LocateElem

测试步骤及结果如表 1-8 所示

表1-8 LocateElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 查找相应的元素的位置 | 表L[0]存在且不为空，输入7 0 | 查找成功 | 输出“你所要查找的数的位置为：1” |
| 表L[0]存在且不为空，输入7 5 | 查找失败（表中无5） | 输出“查找失败！” |
| 表L[0]存在但为空，输入7 0 | 查找失败 | 输出“查找失败！” |
| 表L[0]不存在，输入7 | 操作失败 | 输出“查找失败！” |

8) 测试函数：PriorElem

测试步骤及结果如表 1-9 所示

表1-9 PriorElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 返回查找元素的前驱元素 | 表L[0]存在且不为空，输入8 0 | 查找首元素前驱元素失败 | 输出“查找失败！” |
| 表L[0]存在且不为空，输入8 1 | 查找成功 | 输出“你所要查找的数的前驱为0” |
| 表L[0]存在且不为空，输入8 5 | 查找失败（5不在表中） | 输出“查找失败！” |
| 表L[0]不存在 | 操作失败 | 输出“查找失败！” |

9) 测试函数：NextElem

测试步骤及结果如表 -10 所示

表1-10 PriorElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 获得查找元素的后继 | 表L[0]存在且不为空。输入9 0 | 查找成功 | 输出“你所要查找的数的后继为：1” |
| 表L[0]存在且不为空。输入9 4 | 查找失败（尾元素无后继） | 输出“在线性表中找不到你所要查找的数的后继！” |
| 表L[0]存在且不为空，输入9 5 | 查找失败（表中元素没有5） | 输出“在线性表中找不到你所要查找的数的后继！” |
| 表L[0]不存在，输入9 0 | 操作失败 | 输出“在线性表中找不到你所要查找的数的后继！” |

10) 测试函数：ListInsert

测试步骤及结果如表 1-11 所示

表1-11 ListInsert函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 表L[0]存在，表头插入 | 输入10 1 1 | L[0]顺序表变为1-0-1-2-3-4 | 输出“"插入成功！” |
| 表L[0]存在，表尾插入 | 输入10 2 1 | L[0]顺序表变为1-1-0-1-2-3-4 | 输出“"插入成功！” |
| 表L[0]存在，表中插入 | 输入10 7 5 | L[0]顺序表变为1-1-0-1-2-3-4-5 | 输出“"插入成功！” |
| 表L[0]存在，插入位置不合法 | 输入10 8 6 | 操作失败 | 输出“输入位置不合法，插入失败！” |
| 表L[0]不存在，插入元素 | 输入10 1 1 | 操作失败 | 输出“输入位置不合法，插入失败！” |

11) 测试函数：ListDelete

测试步骤及结果如表 1-12 所示

表1-12 ListDelete函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 表L[0]存在删除表头元素 | 输入11 1 | L[0]顺序表变为1-2-3-4 | 输出“删除成功！你所删除的数据为：0” |
| 表L[0]存在删除表尾元素 | 输入11 2 | L[0]顺序表变为1-3-4 | 输出“删除成功！你所删除的数据为：2” |
| 表L[0]存在删除表中元素 | 输入 | L[0]顺序表变为1-3 | 输出“删除成功！你所删除的数据为：4” |
| 表L[0]存在删除元素的位置不合法 | 输入 | 删除失败 | 输出“输入位置不合法，删除失败！” |
| 表L[0]不存在删除元素 | 输入 | 操作失败 | 输出“输入位置不合法，删除失败！” |

12) 测试函数：ListTraverse

测试步骤及结果如表 1-13 所示

**表1-13 ListTraverse函数测试**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 遍历元素 | 表存在，输入12 | 遍历元素 | 输出0-1-2-3-4 |
| 表存在表为空，输入12 | 遍历元素 | 输出“线性表是空表！” |
| 表不存在，输入12 | 操作失败 | 输出“线性表不存在！” |

13) 测试函数：SaveList & LoadList

测试步骤及结果如表 1-14 所示

**表1-14 保存/读取函数测试**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 读取顺序表元素并保存到文件中 | 表L[0]存在时，输入13 storagelist0 | 保存成功 | 无显示 |
| 读取文件内容并保存到顺序表中 | 输入14 storagelist0 | 读取成功 | 无显示 |

**1.3.3结果分析**

实验测试结果说明了此次程序符合要求，实现了顺序表的相关操作、多表管理、文件读写的功能。

**1.4实验小结**

这次是数据结构的第一次实验，也是大二以来的第一次实验。通过这次实验，我对c语言及数据结构有了进一步的了解，在设计程序的过程中遇到的问题和涉及到的知识及解决如下：

1.问题：变量引用

对于结构体，当顺序表L作为结构变量时，引用结构成员需要用“.”，即L.elem；但如果将L定义为一个结构指针，则需要用->引用。

2.问题：参数

首先是对于函数的参数，当一个函数作为一个参数被另一个函数调用时，其指针表示应该带上返回值，表达形式和函数声明类似，但是函数名称要用指针代替，同时在调用的时候，去掉返回值。如LocateList函数中调用equal函数，参数内表达为 int (\*compare) (ElemType x,ElemType y), 调用时则用(\*compare)(x,y)，同时在PrintMune函数中调用LocateList函数时传入的参数中compare函数直接可写为equal。其次是当参数是变量且需要对用传入的参数返回里另外一个值（即进行修改时），一定要传入地址（即取址），否则不会改变。需要注意的一点是，在初始化顺序表L时也需要传入L所属类型的指针，因为会对L进行修改。

3.问题：文件读写

文件读写在大一上实际上就掌握的不够好，很多地方都是一知半解。此次出现的问题主要在fread函数。因为附录B给出了文件读写的实例，但是是将读和写放在一个函数里面，当我在设计程序需要分别编写时就出现了问题--fread函数不知道如何使用。后来问过助教才知道，因为fwrite函数和fread函数是分开写的，所以在每次读取文件时需要输入之前写入的文件的名称。在文件方面还是要多加学习，多多尝试。

4.问题：实验设计

由于是第一次实验，即使给了任务书也参考了上一届学长的报告也还是比较迷，对于实验到底要干什么，需要实现什么功能，用户操作的流程都不清楚。导致第一节课只是实现了相关函数的编写，对于程序总体把握还是一脸懵逼。后来也是通过询问助教才逐渐把握实验设计流程的脉络。这给了我很大的启发，以后工作若是碰到啥都没交代的甲方，那可咋办？所以，还是要多看看程序，多询问，和别人探讨一下想法在动手。

5.问题：细节处理

在设计相关函数的时候碰到过许多细节上的error和warning以及虽然调试成功但结果不是预料的结果等各种细节错误，大都在源文件中有注释，希望能吸取教训，避免再犯，毕竟处理这种小问题太花时间了！

# 

# 2基于链式存储结构的线性表实现

2.1 问题描述

2.1.1 需要完成的功能

用链式表的物理结构储存类型为ElemType的元素，并实现多表管理。

要求要求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线性表的文件形式保存。其中，①需要设计文件数据记录格式，以高效保存线性表数据逻辑结构(D,{R})的完整信息；②需要设计线性表文件保存和加载操作合理模式。

2.1.2 实验要求

通过实验达到：⑴加深对线性表的概念、基本运算的理解；⑵熟练掌握线性表的逻辑结构与物理结构的关系；⑶物理结构采用单链表,熟练掌握线性表的基本运算的实现。

2.2 系统设计

2.2.1 系统总体设计

系统实现顺序表的基本操作功能，系统总体模块设计如图2-1。

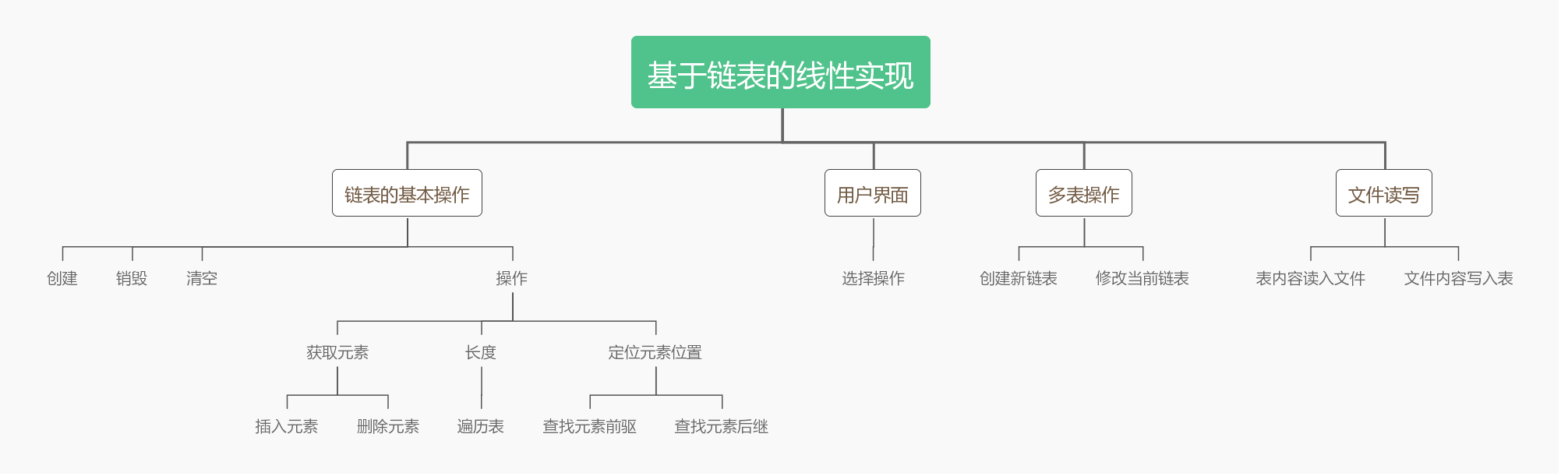


图2-1 系统模块设计

一共保存了三个文件。common.h文件用来定义公用常量（如代码1-1所示）和函数。sqlist.h文件用来定义链表结点结构体，如代码1-2所示，此外，还有对链表进行相关操作的函数声明和表的相关定义。main.c文件用来实现链表相关函数，文件读写函数，系统界面显示函数和用来调用其他函数的main函数。

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <malloc.h>  /\*书本第10页，预定义常量和类型\*/  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  typedef int status;  typedef int ElemType; //数据元素类型定义  status equal(ElemType x,ElemType y){  return (x-y);  } |

代码2-1 公用常量和函数设计

|  |
| --- |
| #define LIST\_INIT\_SIZE 100 //初始分配量  #define LISTINCREMENT 10 //分配增量  //结点类型  typedef struct LNode{ //顺序表（顺序结构）的定义  ElemType elem; //结点的数据  struct LNode \*next; //指向下一个结点的指针  }LNode,\*LinkList; |

代码2-2 链表节点和链表相关常数设计

2.2.2 多表设计

多表管理的实现是通过定义一个LinkList的结构类型数组L[10]数组的每一个元素都是一个指向链表节点类型的指针。详情如代码段2-3。利用一个循环，在每一次利用菜单界面初始化一个链表并进行完相应操作之后，由用户判断是否继续创建下一个表。

|  |
| --- |
| LinkList L[10];  int op=1; int ans=0;int count=0;char c;  while(op)  {  while(op)  {  ... ...(此处省略详细代码，仅显示多表操作代码)  }  count++;  printf("继续创建下一个线性表? [Y/N]\n");  getchar();op=((c=getchar())=='Y')?1:0;  } |

代码2-3 实现多表操作设计

2.2.3 多表管理菜单操作流程

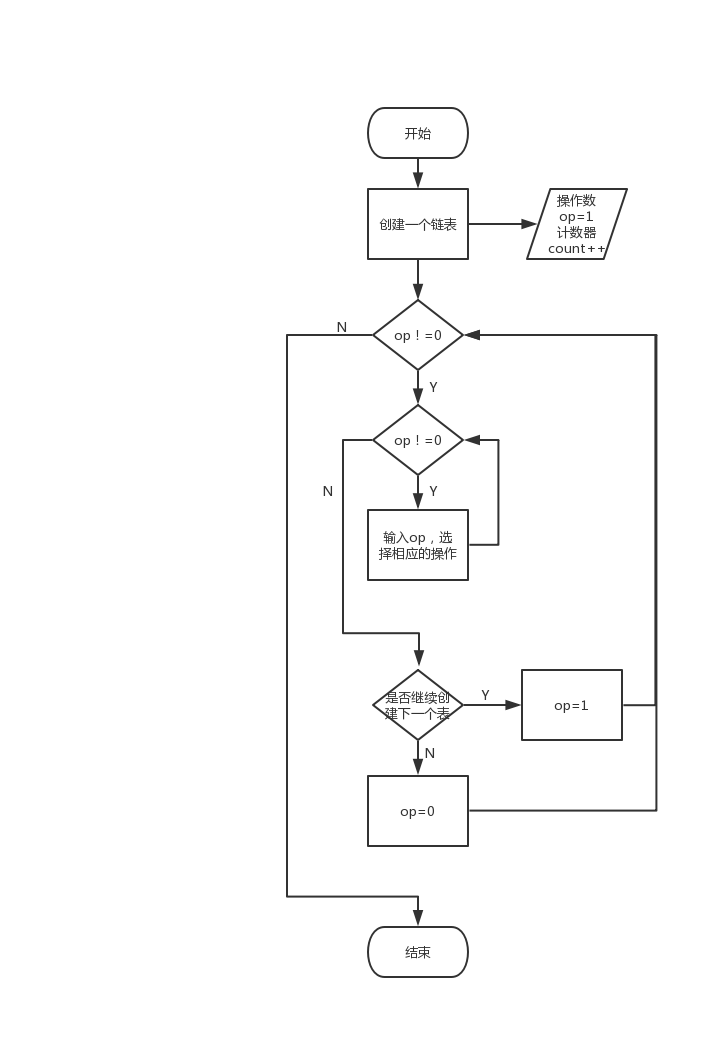


图1-2 多表管理系统菜单操作流程

2.2.4 函数设计

依据最小完备性和常用性相结合的原则，以函数形式定义了链表的初始化表、销毁表、清空表、判定空表、求表长和获得元素等15种基本运算，具体运算功能定义如下。

（1）初始化表：函数名称是InitList(LinkList\* headp，int\* flag)；

初始条件：线性表L未被初始化；

算法设计：malloc动态分配一个链表节点空间，分配成功，则将分配的该节点作为首节点，将flag置为0，表示已经初始化其指针域指向空，返回OK，否则，返回OVERFLOW；

操作结果：构造一个空的链表。

（2）销毁表：函数名称是DestroyList(LinkList\* headp,int flag)；

初始条件：无；

算法设计：如果线性表不存在，返回INFEASTABLE；否则，初始化一个指针指向传入的链表的首节点，当p不为空时，头指针指向p的next，释放掉p，p指向头指针。p为空时，则说明全部节点销毁完毕，返回OK；

操作结果：销毁线性表L。

（3）清空表：函数名称是ClearList(LinkList\* headp,int flag)；

初始条件：无；

算法设计：如果链表不存在，则返回-1；否则，p指向第一个节点，当p不为空时，指针q指向p的下一个节点，释放掉p节点，p指向q。当p为空时，说明链表已经全部清空，将首节点next指向空，返回OK;

操作结果：将L重置为空表。

（4）判定空表：函数名称是ListEmpty（LinkList headp,int flag)；

初始条件：无；

算法设计：如果链表不存在，则返回-1，如果线性表的头指针的next为空，则证明为空，返回1，否则，不为空，返回0；

操作结果：若L为空表则返回TRUE,否则返回FALSE。

（5）求表长：函数名称是ListLength(LinkList headp,int flag)；

初始条件：无；

算法设计：如果链表不存在，则返回-1，否则p指向头结点的next，即第一个节点，计数器count=0，当p不为空时，count加一，p指向下一个节点；循环结束后，返回count；

操作结果：返回L中数据元素的个数。

（6）获得元素：函数名称是GetElem(LinkList headp,int i,ElemType\* e,int flag)；

初始条件：无

算法设计：如果链表不存在，则返回-1，否则，p指向第一个节点，计数器j=1，当p不为空且j小于查找的位置i时，p=p->next，j++；循环退出时，如果p不为空或者j>i，则返回0，否则，将p指向的节点的数据域的值赋给e，返回1；

操作结果：用e返回L中第i个数据元素的值。

（7）查找元素：函数名称是LocateElem(LinkList headp,ElemType e,int (\*compare)(ElemType x,ElemType y),int flag)；

初始条件：无；

算法设计：如果链表不存在，则返回-1，否则，p指向第一个节点，计数器count=0，当p不为空时，用compare函数比较p指向的每一个节点的数据和e，如果比较函数返回1，则表明查找成功，返回（计数器+1），否则，p指向下一给节点，计数器自增1；推出循环后表明查找失败，返回0；

操作结果：返回L中第1个与e满足关系compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。

（8）获得前驱：函数名称是PriorElem（LinkList headp,ElemType cur,ElemType \*pre\_e,int flag)；

初始条件：无；

算法设计：如果链表不存在，则返回-1，否则，p指向第一个节点，如果p为空，则证明链表时空表，否则，如果p指向的节点的数据等于查找的数据，则说明是查找的数是第一个元素，返回0；否则，当p->next不为空时，如果p的下一个节点的值等于查找的cur的值，则返回1，否则，p指向下一个节点。循环结束但是未找到，则返回0.

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：函数名称：NextElem(LinkList headp,ElemType cur,ElemType\* next\_e,int flag)；

初始条件：无；

算法设计：如果链表不存在，则返回-1，否则，p指向第一个节点，如果p为空，则证明链表时空表，返回0，否则令q指向p的下一个节点，当q不为空时，如果p指向的节点的数据等于cur，则证明q就是要查找的后继节点，将其数据赋给next\_e，并返回1，否则，p、q均后移，退出循环后如果找不到值则返回0；

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：函数名称是ListInsert(LinkList headp,int i,ElemType e,int flag)；

初始条件：无；

算法设计：如果链表不存在，则返回-1，否则，p指向首节点，计数器J等于1，当p不为看空且j小于i-1时，说明没有到达要插入元素的位置的前一个位置，p后移，j自增1，不满足循环条件退出时如果p为空或者输入的j大于i-1，则说明输入的i不合法，返回0；否则，molloc分配一个节点的空间s，输入节点的数据域的值和并将新节点的next指向p的next，而p的next重新又指向s，返回1；

操作结果：在L的第i个位置之前插入新的数据元素e。

（11）删除元素：函数名称是ListDelete(SqList L,int i,ElemType\* e,int flag)；

初始条件：线性表L已初始化；

算法设计：如果链表不存在，则返回-1，否则，p指向首节点，计数器J等于1，当p不为看空且j小于i-1时，说明没有到达要插入元素的位置的前一个位置，p后移，j自增1，不满足循环条件退出时如果p为空或者输入的j大于i-1，则说明输入的i不合法，返回0；否则，说明指针p正好指向删除位置的节点，则令指针q指向p的next，p的next指向q的next，节点删除成功，并用e返回p节点的数据域的值，释放掉p指向的节点，返回1；

操作结果：删除L的第i个数据元素，用e返回其值。

（12）遍历表：函数名称是ListTraverse(LinkList headp,int flag)

初始条件：线性表L已初始化；

算法设计：如果链表不存在，则返回-1，否则，p指向第一个节点，当p不为空时输出p的节点的数据域的值，p指向下一个节点，返回i；

操作结果：输出所有元素；

（13）写入文件：函数名称是SaveList(LinkList headp)

初始条件是：线性表L已经存在；

算法设计：输入保存文件的名称，判断打开文件是否成功，是，p指向第一个节点，当p不为空时，用fwrite函数每次写入一个节点数据到文件中，然后p指向下一个节点。循环结束后，关闭指针指向的文件，返回0；

操作结果：将链表中的元素内容写入文件中；

（14）读取文件：函数名称是LoadList(LinkList headp)

初始条件：线性表已经存在；

算法设计：输入要载入的文件的名称，如果该文件打开失败，则退出程序，否则，令p指向首节点，指针add指向molloc分配的一个节点空间，当指向文件的指针fp不为空时，将fp的内容一次读入一个元素到add节点中，并让p的next指向add节点，而将add的next置为空。循环结束时表明文件内的内容重新写入链表中，则释放掉多增加的add节点，关闭指向文件的指针fp，返回0；

操作结果：顺序表恢复到初始未修改状态。

2.3 顺序演示实现与测试

2.3.1 功能实现

本次实验使用的环境配置如下：

操作系统版本：Windows 10 专业版

编译器及其版本：GNN GCC Compiler

编程环境：CodeBlocks ver17.12

实验的完整代码参见附录A。

测试时通过用户界面对线性表进行操作。用户界面参见图2-3。

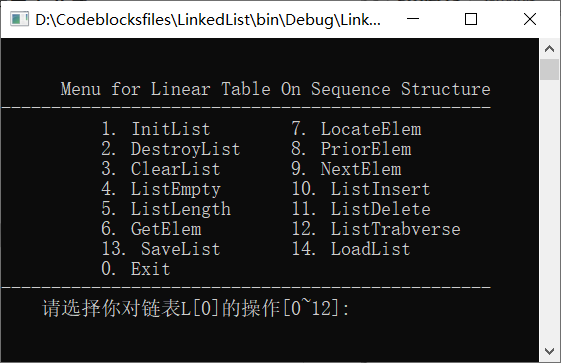


图2-3 演示程序用户界面菜单

2.3.2 功能测试

测试中使用的测试数据

表2-1 测试用例

|  |  |
| --- | --- |
| 表名称 | 表中元素（按顺序排列） |
| L[0] | 0，1，2，3，4 |
| L[1] | 1，2，4，4 |
| L[2] | 2，3，5 |

1) 测试函数：InitList

测试步骤及结果如表 2-2 所示

表2-2 InitList 函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 创建表 | 表L[0]未被初始化，输入1 | 建立表L[0] | 输出“线性表创建成功！” |

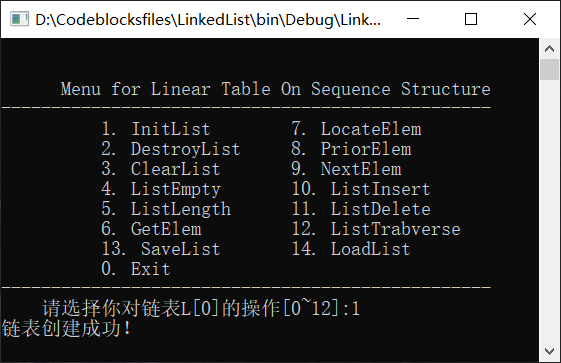


图2-4

2) 测试函数：DestroyList

测试步骤及结果如表 2-3 所示

表2-3 DestroyList函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 销毁表 | 表L[0]存在，输入2 | 销毁表L[0] | 输出“线性表销毁成功” |
| 表L[0]未被初始化，输入2 | 操作失败 | 输出“表不存在，无法销毁！” |

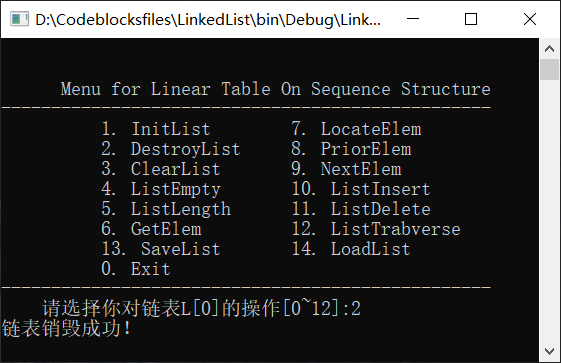


图2-5

3) 测试函数：ClearList

测试步骤及结果如表 2-4 所示

表2-4 ClearList函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 清空表内元素 | 表L[0]初始化后，输入3 | 表L[0]仍然存在，但是表内元素为空 | 输出“链表清空成功！” |
| 表L[0]未被初始化，输入3 | 操作失败 | 输出“表不存在，清空失败！” |

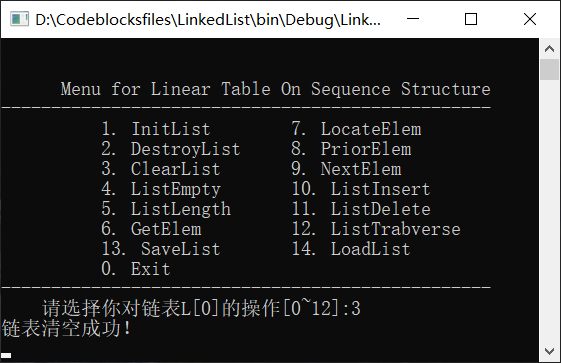


图2-6

4) 测试函数：ListEmpty

测试步骤及结果如表 2-5 所示

表2-5 ListEmpty函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 判断表是否为空 | 表L[0]初始化且有元素时，输入4 | 表L[0]不为空 | 输出“链表是空表！” |
| 表L[0]初始化但没有元素时，输入4 | 表L[0]为空 | 输出“链表不是空表！” |
| 表未被初始化，输入4 | 操作失败 | 输出“链表不存在！” |

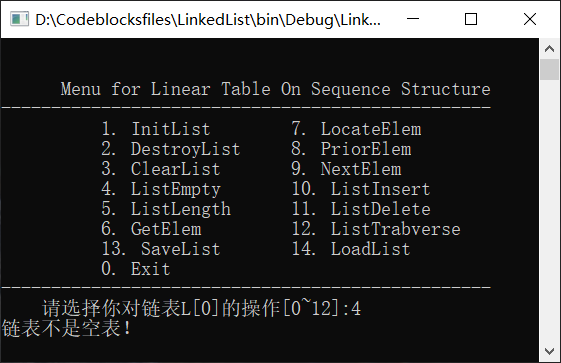


图2-7

5) 测试函数：ListLength

测试步骤及结果如表 2-6所示

表2-6 ListLength函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 输出表长 | 表L[0]初始化后，输入5 | 返回L[0]表长 | 输出“线性表的长度为：5” |
| 表未被初始化，输入5 | 操作失败 | 输出“链表不存在！” |

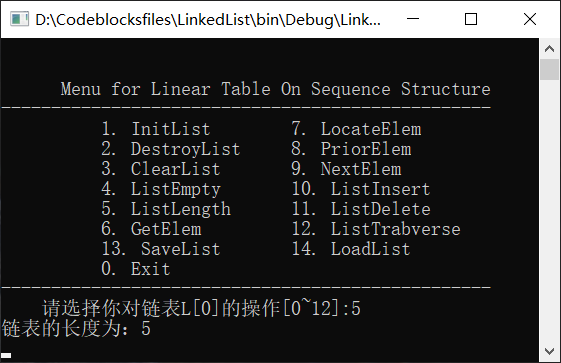


图2-8

6) 测试函数：GetElem

测试步骤及结果如表 2-7 所示

表2-7 GetElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 获得元素 | 表L[0]存在且有元素时，输入6 1 | 得到表L[0]的第一个元素 | 输出“表中第1个元素的值为0：” |
| 表L[0]存在其有元素时，输入6 2 | 得到第二个元素 | 输出“表中第2个元素的值为1” |
| 表L[0]存在其有元素时，输入6 5 | 得到尾元素 | 输出“表中第5个元素的值为4” |
| 表L[0]存在且有元素，输入6 6 | 操作失败 | 输出“输入的数据不合法!” |
| 表未被初始化，输入6 1 | 操作失败 | 输出“链表不存在！” |



图2-9



图2-10

7) 测试函数：LocateElem

测试步骤及结果如表 2-8 所示

表2-8 LocateElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 查找相应的元素的位置 | 表L[0]存在且不为空，输入7 0 | 查找成功 | 输出“你所要查找的数的位置为：1” |
| 表L[0]存在且不为空，输入7 5 | 查找失败（表中无5） | 输出“查找失败！” |
| 表L[0]存在但为空，输入7 0 | 查找失败 | 输出“查找失败！” |
|  | 表未被初始化，输入7 1 | 操作失败 | 输出“链表不存在！” |



图2-11



图2-12

8) 测试函数：PriorElem

测试步骤及结果如表 2-9 所示

表2-9 PriorElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 返回查找元素的前驱元素 | 表L[0]存在且不为空，输入8 0 | 查找首元素前驱元素失败 | 输出“查找失败！” |
| 表L[0]存在且不为空，输入8 1 | 查找成功 | 输出“你所要查找的数的前驱为0” |
| 表L[0]存在且不为空，输入8 6 | 查找失败（5不在表中） | 输出“查找失败！” |
| 表为被初始化，输入8 1 | 操作失败 | 输出“链表不存在！” |



图2-13



图2-14



图2-14

9) 测试函数：NextElem

测试步骤及结果如表 -10 所示

表1-10 PriorElem函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 获得查找元素的后继 | 表L[0]存在且不为空。输入9 0 | 查找成功 | 输出“你所要查找的数的后继为：1” |
| 表L[0]存在且不为空。输入9 4 | 查找失败（尾元素无后继） | 输出“查找失败！” |
| 表L[0]存在且不为空，输入9 5 | 查找失败（表中元素没有5） | 输出“查找失败！” |
| 表未被初始化输入9 1 | 操作失败 | 输出“链表不存在！” |



图2-15



图2-16



图2-17

10) 测试函数：ListInsert

测试步骤及结果如表 2-11 所示

表2-11 ListInsert函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 表L[0]存在，表头插入 | 输入10 1 1 | L[0]顺序表变为1-0-1-2-3-4 | 输出“"插入成功！” |
| 表L[0]存在，表尾插入 | 输入10 2 1 | L[0]顺序表变为1-1-0-1-2-3-4 | 输出“"插入成功！” |
| 表L[0]存在，表中插入 | 输入10 7 5 | L[0]顺序表变为1-1-0-1-2-3-4-5 | 输出“"插入成功！” |
| 表L[0]存在，插入位置不合法 | 输入10 9 6 | 操作失败 | 输出“输入位置不合法，插入失败！” |
| 表未被初始化 | 输入10 1 1 | 操作失败 | 输出“链表不存在！” |



图2-18



图2-19



图2-20

11) 测试函数：ListDelete

测试步骤及结果如表 2-12 所示

表2-12 ListDelete函数测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 表L[0]存在删除表头元素 | 输入11 1 | L[0]顺序表变为1-2-3-4 | 输出“删除成功！你所删除的数据为：0” |
| 表L[0]存在删除表中元素 | 输入11 2 | L[0]顺序表变为1-3-4 | 输出“删除成功！你所删除的数据为：2” |
| 表L[0]存在删除表尾元素 | 输入11 3 | L[0]顺序表变为1-3 | 输出“删除成功！你所删除的数据为：4” |
| 表L[0]存在删除元素的位置不合法 | 输入11 5 | 删除失败 | 输出“输入位置不合法，删除失败！” |
| 表未被初始化 | 输入 11 1 | 操作失败 | 输出“链表不存在！” |



图2-21



图2-22

12) 测试函数：ListTraverse

测试步骤及结果如表 1-13 所示

**表1-13 ListTraverse函数测试**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 遍历元素 | 表存在，输入12 | 遍历元素 | 输出0-1-2-3-4 |
| 表存在表为空，输入12 | 遍历元素 | 输出“链表是空表！” |
|  | 表未被初始化，输入12 | 操作失败 | 输出“链表不存在！” |

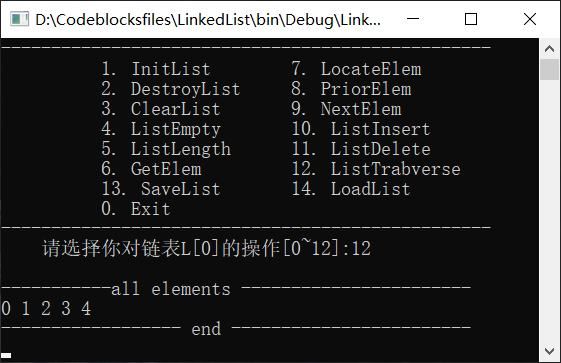


图2-23

13) 测试函数：SaveList & LoadList

测试步骤及结果如表 1-14 所示

**表2-14 保存/读取函数测试**

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件和输入 | 理论结果 | 运行结果 |
| 读取链表元素并保存到文件中 | 表L[0]存在时，输入13 storagelist0 | 保存成功 | 无显示 |
| 读取文件内容并保存到链表中 | 输入14 storagelist0 | 读取成功 | 无显示 |

2.3.3 结果分析

实验测试结果说明了此次程序符合要求，实现了链表的相关操作、多表管理、文件读写的功能。

2.4实验小结

这次实验室关于链式存储结构的线性表实现。链表和顺序表显然还是有很大的差别的，尤其体现在表的长度和确定元素位置方面，在很多函数比如求长度、找位置、找前驱和后继方面都需要循环遍历，不如顺序表位置的下标简捷。但是这次实验有了第一次的顺序表的基础，在设计方面还是要简单一些。但依旧存在一些问题。大体如下：

1.问题：结构变量声明

即便链表结构的定义在书上有，但在main函数中声明一个链表及多个链表时还是遇到了问题--不知道链表L到底定义为什么类型的数据。同时遇到的还有相关函数的参数问题，也是链表类型无法确定。后来经过借鉴同学和自己多次尝试后才敲定代码。多个链表L可以定义为LinkList指针类型的数组，数组的每个元素都是一个链表结点类型的指针，在涉及到需要对链表进行修改的函数，就需要传入指针，即LinkList\*,更应该注意到，当传入的是双重指针的时候，在引用结点的数据或者next指针就需要写成（\*headp）->elem。当定义的函数不需要更改链表时，直接传入LinkList类型的一重指针即可。要注意的是，在函数内部定义一个结构变量的指针是直接LinkList p就可以了，不需要带\*号，因为LinkList本来就是一个指针类型（写的时候忘了结果报出好多error）。

2.问题：文件的读写

果然在文件的读写方面还是存在很大的问题。在这里，链表和顺序表在操作上就体现出了不同（虽然读写的思想是一样的）。首先是fwrite函数，多个顺序表的写入可以直接一句话调用fwrite函数即可，但是链表的话需要遍历，并且每次只能想文件中写入一个结点的数据元素（因为在链表定义中没有定义链表的长度，无法直接利用）。其次就是fread函数。这里出的问题最大。fread的主要思想是要将原本的内容清除，然后才能写入新的数据，在顺序表中的体现就是表L长度置为0（即清空，但是表仍然存在），然后每次在写入一个元素后，表的长度都要加一，为后面读进去的元素留出位置。所以应用在链表上，就需要在第一次malloc一个新的结点，将文件中的数据写入新的结点中的之后，让新的结点链接在头指针的后面，之后在指向文件内容的指针不为空时循环malloc一个结点，并接在新的表后面。具体代码见源文件。

3.问题：细节

程序在实现上还是有一定的限制的，很多相关函数都建立在表存在的前提。程序的健壮性仍然有待提高。

1. 关于程序声明：因为在编写出L[10]代码后，计算机自动分配了存储空间，并默认初始化，但是我们想要的结果是什么也没有的链表并由自己初始化，所以这个时候就需要定义一个标志flag,当用户没有对链表进行初始化时，flag值为1，此时调用destroy等其他函数，都会得出链表不存在的结果，当且仅当用户自己初始化之后，在初始化函数中将flag值重置为0，此时在调用其他函数，就显示链表存在可以进行操作。
2. 问题：参数

所有定义的函数中有初始化和销毁函数需要传入指针的指针的参数，其余都可以只要是指针。

6.此次实验写的程序比上一次写的功能更加完善，感谢助教的指教。

# 3 基于二叉链表的二叉树实现

3.1问题描述

通过实验达到⑴加深对二叉树的概念、基本运算的理解；⑵熟练掌握二叉树的逻辑结构与物理结构的关系；⑶以二叉链表作为物理结构，熟练掌握二叉树基本运算的实现。

要求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。

此外，演示系统需要实现多个二叉树管理与实现二叉树的文件形式保存。其中，二叉树的保存需要 ①设计文件数据记录格式，以高效保存二叉树数据逻辑结构(D,{R})的完整信息；②设计二叉树文件保存和加载操作合理模式。

3.1.1需要完成的功能

依据最小完备性和常用性相结合的原则，以函数形式定义了销毁二叉树、创建二叉树、清空二叉树、判定空二叉树和求二叉树深度等17种基本运算。

3.1.2实验要求

通过实验达到 ⑴加深对二叉树的概念、基本运算的理解；⑵熟练掌握二叉树的逻辑结构与物理结构的关系；⑶物理结构采用二叉树,熟练掌握二叉树的基本运算的实现。

3.2系统设计

3.2.1系统总体设计

本系统实现二叉树的基本操作功能，系统总体模块设计参见图3-1。

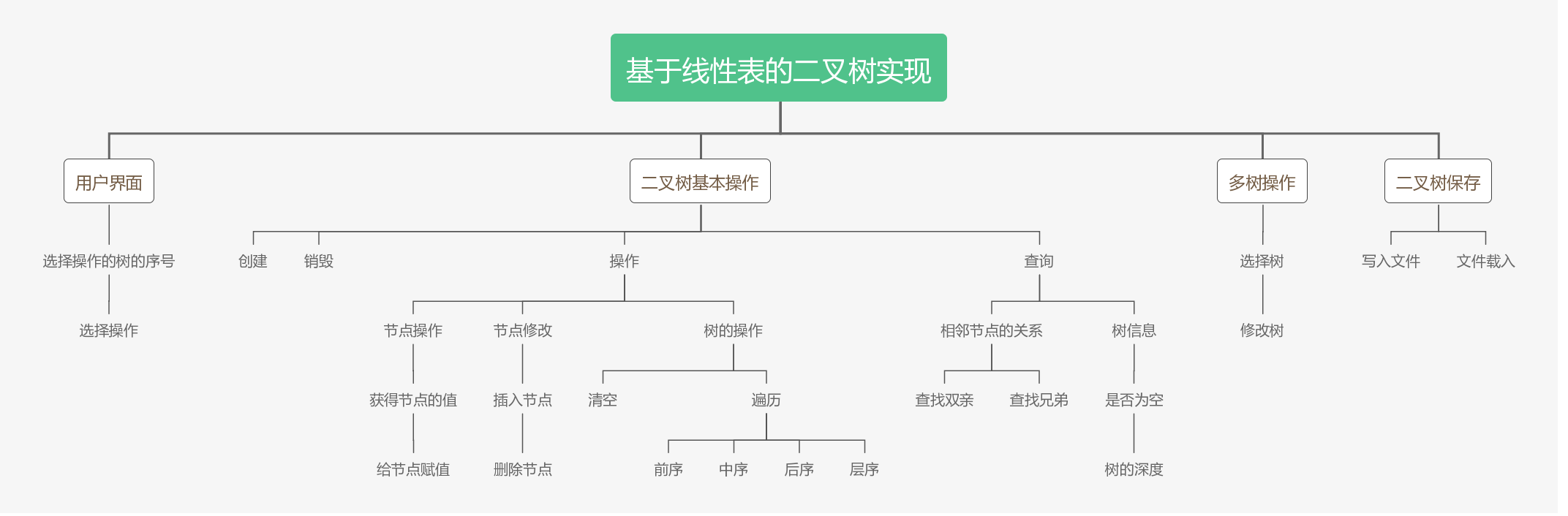


图3-1 系统总体功能设计

本系统实现了基于线性表的二叉树的基本操作（函数实现详见3.2.4）。二叉树实现的功能比线性表和链表更加复杂，为了方便操作，为每一个二叉树的节点的数据域增加一个唯一的关键字，即二叉树节点存储的其实是一个键值对key-value。

|  |
| --- |
| typedef struct Elem{  int key;  int value;  }ElemType |

代码段3-1

为了方便，将关键字和数值的数据类型都定义为int型，并且规定在输入序列时，采用先序带空格的输入，并让输入0表示输入空格，输入-1表示结束输入。

二叉树的节点除了键值对以外，还有左右孩子指针。节点的结构体定义如下：

|  |
| --- |
| typedef struct BiTNode{  ElemType elem;  struct BiTNode\* lchild;  struct BiTNode\* rchild;  }BiTNode,\*BiTree; |

代码段 3-2

3.2.2多树设计

为实现多棵树的管理，采用线性表的形式储存多棵二叉树。用户可以用户界面选择任意一棵树进行操作。二叉树的线性管理结构如下：

|  |
| --- |
| BiTree T[10]; |

代码段3-3

必须要指出的是，在创建二叉树时，并未为二叉树创建头结点，所以本系统对二叉树的销毁和清空有很大的区别。销毁是将所有节点都销毁，而清空则是保留根节点的内存空间和其存储的数据信息而释放其余所有节点。

一颗多二叉树的物理结构如图3-2所示：



图3-2 多二叉树的存储关系

3.2.3 多树管理菜单操作流程

用户进入界面后首先选择要操作的二叉树的序号（0-9），通过order计数，

确定操作的二叉树后即可进入具体操作树的界面。设置两层循环进行操作，循环条件为op的值。同时用变量op记录选择的操作序号。当输入op=0后，系统询问是否继续创建下一个二叉树，输入Y，op=1，输入N，op=0，若op==0,则询问是否返回之前的二叉树，同理Y对于op=1，N对应op=0。



图3-3 用户操作实例

3.2.5 相关常量和类型定义

|  |
| --- |
| #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  #define MAXLENG 100  #define ILLEAGAL -3  #define EMPTY 2  typedef int status; |

代码段3-4

3.2.6队列设计

由于需要对二叉树进行层序遍历，定义可一个关于队列操作的头函数，其中定义了初始化队列，在队列中插入一个节点，删除队列中的一个节点并返回删除的节点3个函数，并定义了队列的一些结构体。详情见附录C。

3.2.7部分函数调用声明

（1）由于在main函数声明BiTree T[10]变量时，计算机就自动为多棵二叉树分配了内存空间，并为其节点的数据域赋了默认值，在本系统中凡是没有调用CreateTree函数我们就称二叉树没有初始化，也就是没有创建二叉树，所有的函数均无法调用（用flag来实现）。

（2）关于销毁函数和清空函数，在本系统中由于在创建时并没有创建头结点，所以清空会保留根节点及其数据域的值而释放其他所有节点，而销毁则是将所有节点都释放。在其他函数中的体现就是，如果二叉树是被destory，则输出二叉树是空二叉树，如果二叉树只是clear，则仅有插入、删除、载入文件，创建二叉树四个函数可以继续对二叉树进行操作，其余都会输出“二叉树被清空了！”（通过isempty全局变量实现）

3.2.8函数设计

依据最小完备性和常用性相结合的原则，以函数形式定义了树的创建，清空，销毁，求深度，遍历（前序、中序、后序、层序），查找，删除，赋值，查找兄弟节点，文件读入和读写等15种基本运算，具体运算功能定义如下。

1. 初始化树：函数名称是：status CreateBiTree(BiTree\* t,ElemType\* defination,int count,int\* flag)

初始化条件：二叉树尚未被创建

算法设计：

1. 如果传入的序列数组的关键字为0，将该节点置为空，计数器num加一
2. 当num的值小于count时
3. 为t malloc一个内存空间
4. 如果分配失败，返回-1
5. 将序列数组defination[num]的数据赋给树节点t。并将节点t的左右孩子指针置为空
6. 计数器num加一
7. 递归创建t的左孩子，重复上述步骤
8. 递归创建t的右孩子，重复上述步骤
9. 将判断是否创建的flag赋值为0，表示已经创建
10. 返回成功

操作结果：利用输入的先序序列创建一棵二叉树

（2）销毁树：函数名称是 status DestroyBiTree(BiTree\* t,int flag);

初始条件：无

算法设计：

1. 如果二叉树尚未创建，返回-1
2. 否则，如果传入的树t不为空
3. 递归销毁t的左孩子
4. 递归销毁t的右孩子
5. 释放节点t的空间
6. 将节点t置为空；
7. 返回OK

操作结果：销毁二叉树的所有节点

（3）清空二叉树：函数名称是 status ClearBiTree(BiTree t,int flag);

初始条件：无

算法设计：

1. 如果二叉树尚未创建，返回-1
2. 如果t为空，表示是空二叉树，返回0
3. 销毁t的左右孩子
4. 将t的左右孩子置为空，表示保留根节点，但清楚其中的
5. 返回OK

操作结果：保留根节点和根节点的数据域，清空二叉树其他节点

（4）函数是否为空： 函数名称是 status BiTreeEmpty(BiTree t,int flag)

初始条件：无

算法设计：

1. 如果二叉树尚未创建，返回-1

2. 否则，返回 t==NULL，即，若二叉树被销毁，则为空，返回1，否则，返回0

操作结果：如果二叉树被clear，不调用此函数，只在main函数里面输出被清空，若没有被clear, 则用来判断是否为空二叉树

（5）求树的深度：函数名称是 status BiTreeDepth(BiTree t,int flag)

初始条件：二叉树没有别clear

算法设计：

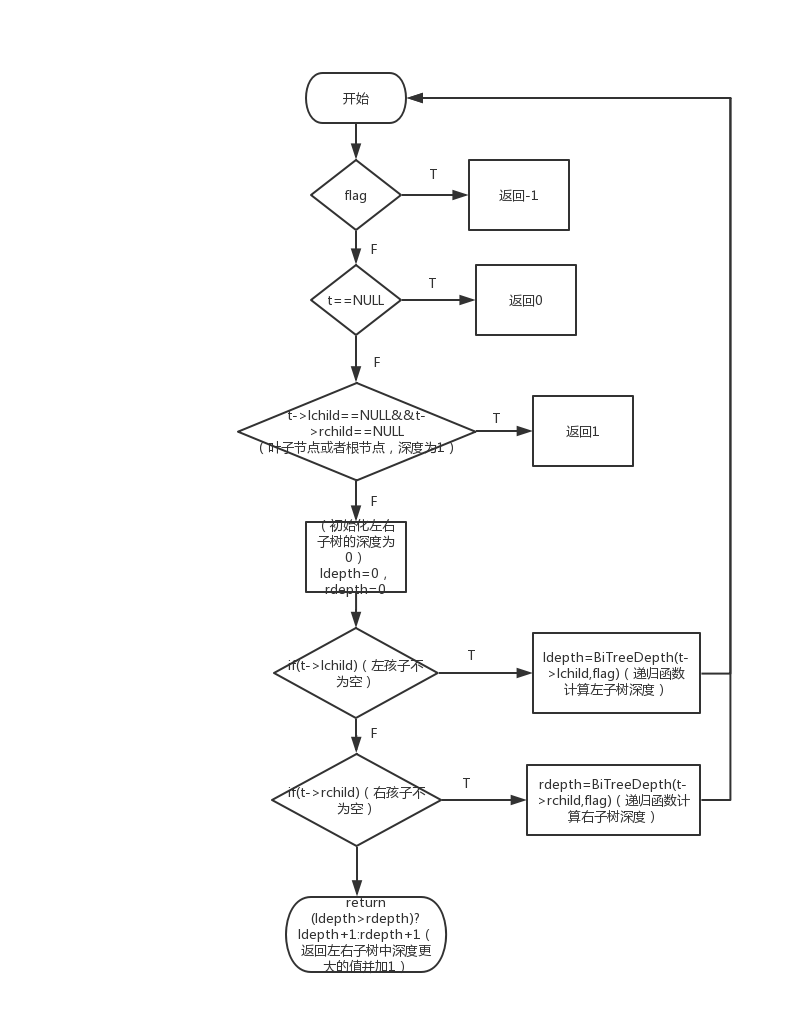


图3-4 求深度算法

操作结果：求出树的深度，并返回

（6）查找节点：函数名称是 BiTree LocateNode(BiTree t,int e)

初始条件：二叉树没有被clear

算法设计：

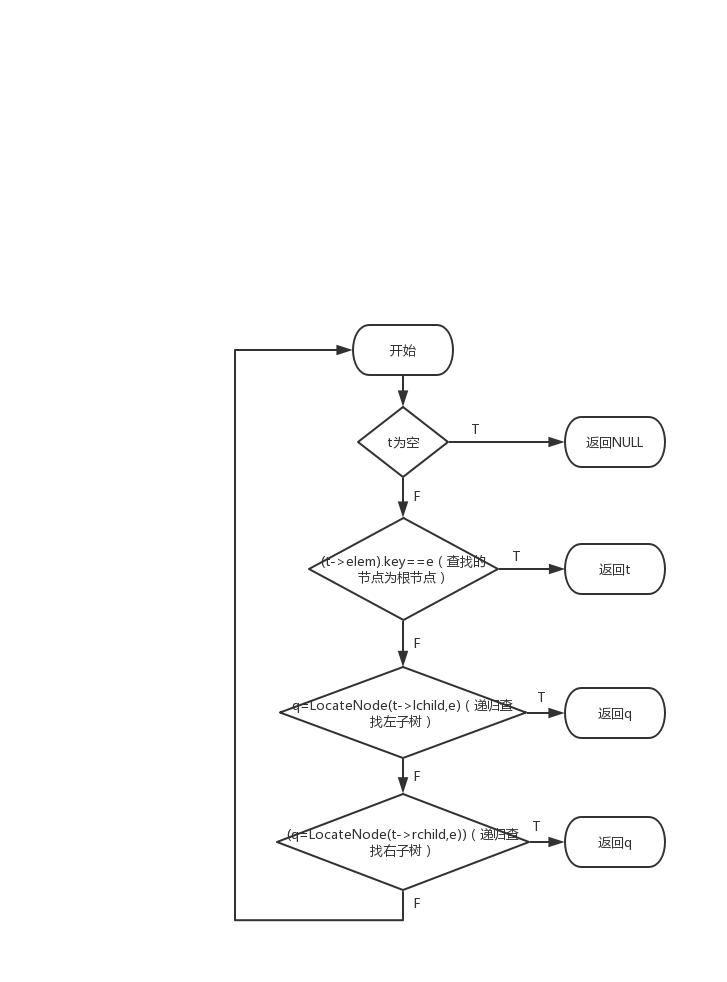


图3-5 查找节点流程图

操作结果：返回与查找的关键字相同的节点

（7）获得双亲节点：函数名称是BiTree GetParent(BiTree t,int e)

初始条件：树未被clear

算法设计：

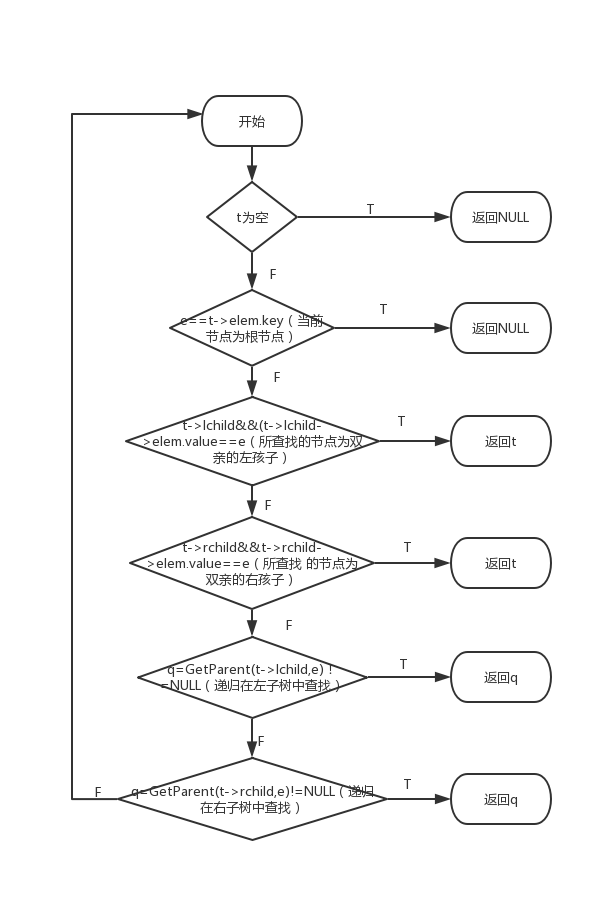


图3-6 查找双亲流程图

操作结果：返回与查找的关键字相同的节点的双亲节点

（8）为二叉树节点赋值：函数名称是status Assign(BiTree t,int e,int val,int flag)

初始条件：二叉树未被clear，输入的关键字是合法的

算法设计

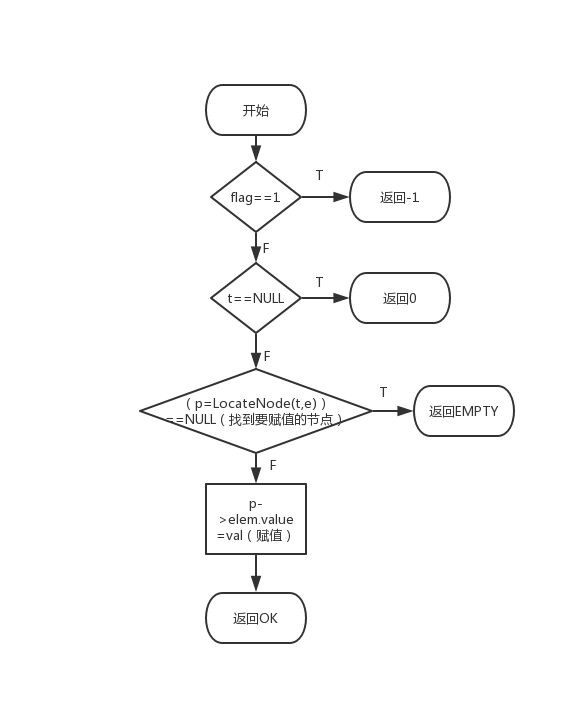


图3-7赋值函数流程图

操作结果：为与查找的关键字相同的节点赋值

（9）获得兄弟节点：函数名称是BiTree GetSibling(BiTree t,int e)

初始条件：二叉树未被clear，输入的关键字合法

算法设计：

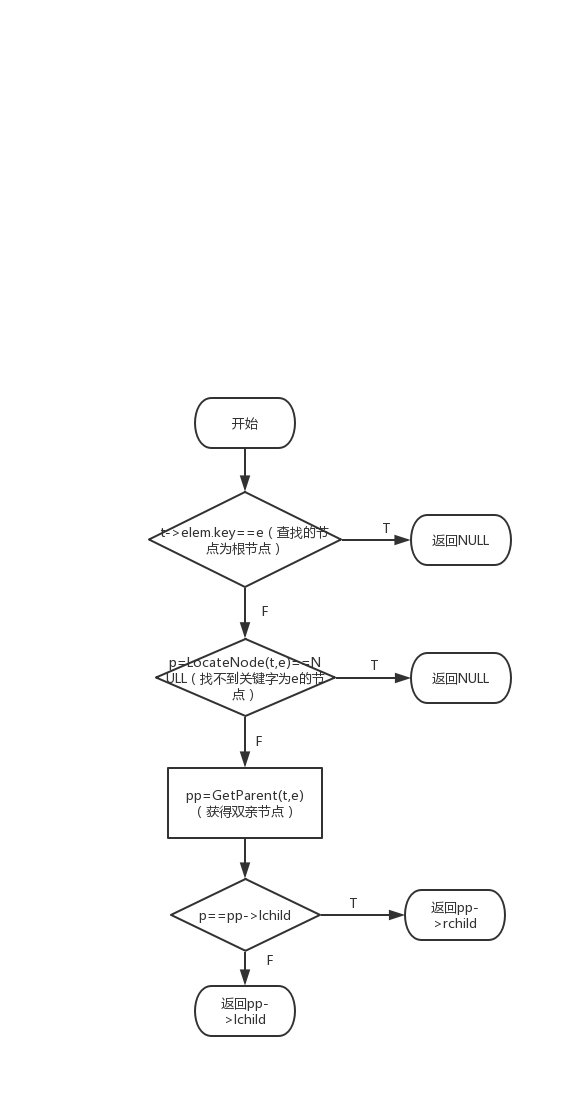


图3-8 查找兄弟节点函数流程图

操作结果：返回含有查找关键字的节点的兄弟

（10）插入节点：函数名称是status InsertNode(BiTree t,int e,int LR,BiTree c,int flag,int isempty)

初始条件：无

算法设计：

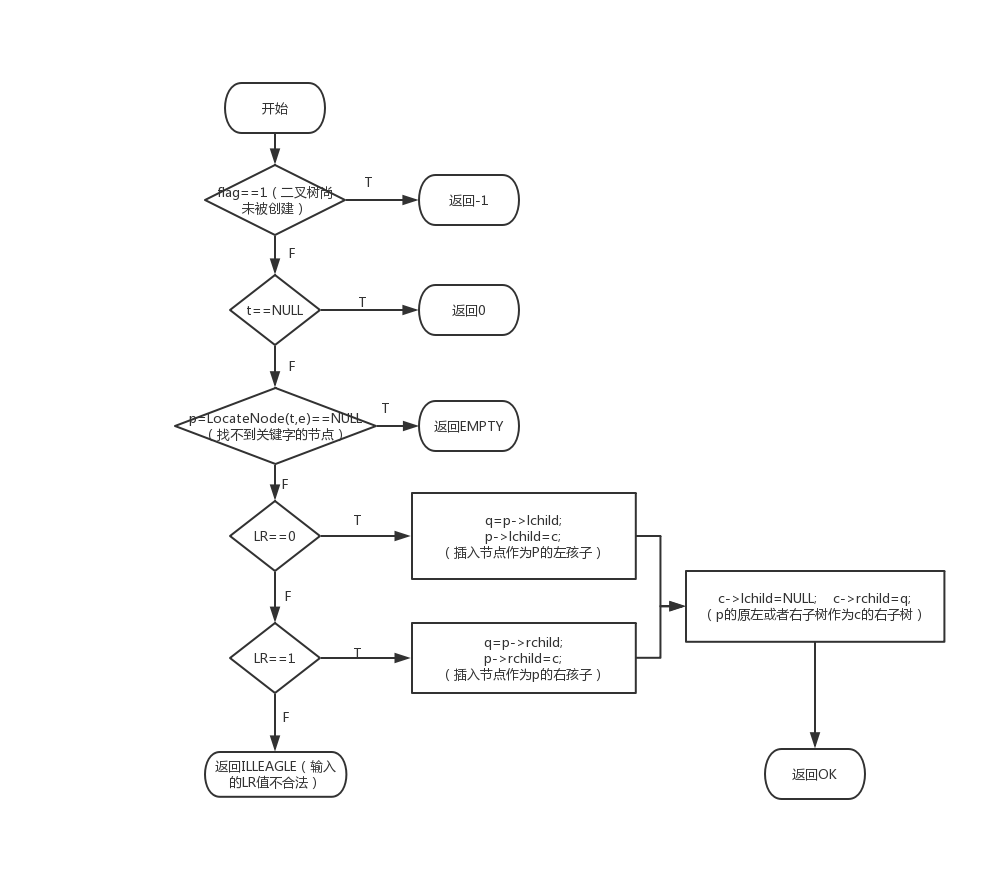


图3-9 插入节点函数流程图

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。

（11）删除节点：函数名称是status DeleteNode(BiTree\* t,int e,int flag)

初始条件：二叉树未被clear

算法设计：

1. 如果flag==1，表示二叉树尚未被创建，返回-1
2. 如果\*t==NULL，表示二叉树为空树，返回0
3. 如果\*t的关键字等于e，说明删除的节点恰好为根节点
4. 如果\*t的左孩子为空，则以根节点的右孩子为新的根节点
5. 否则根节点的右孩子为空，以其左孩子作为新的根节点
6. 否则根的度为2，pp=\*t=(\*t)->lchild，根的左孩子作为新的根节点
7. While(pp->rchild), pp=pp->rchild，遍历到待删的根的左孩子的最右节点
8. pp->rchild=p->rchild; 待删的根的右子树作为其左子树的最右节点的右子树
9. 否则待删的节点不为根节点
10. p=LocateNode(\*t,e);找到待删节点的位置
11. 若p==NULL,说明找不到该节点
12. 找到待删节点的双亲节点pp
13. 若p是叶子节点
14. 若p是pp的左孩子，将pp的左孩子置为空
15. 否则pp右孩子置为空
16. 若p的度为1
17. 用关键字为e的结点孩子代替被删除的e位置
18. 若p的度为2
19. 用e的左孩子代替被删除的e位置
20. e的右子树作为e的左子树中最右节点的右子树
21. 释放删除的节点的空间

（12）中序遍历：函数名称是status InOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty)

初始条件：无

算法设计：

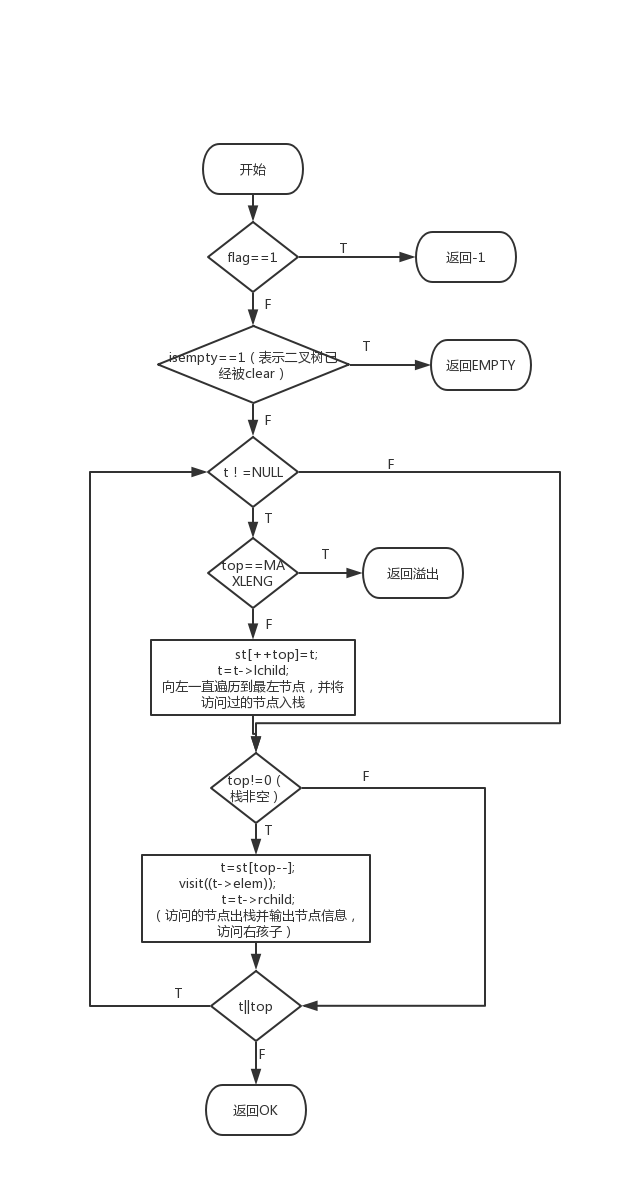


图3-10 中序遍历函数流程图

操作结果：中序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败

（13）前序遍历：函数名称是status PreOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty)

初始条件：无

算法设计：

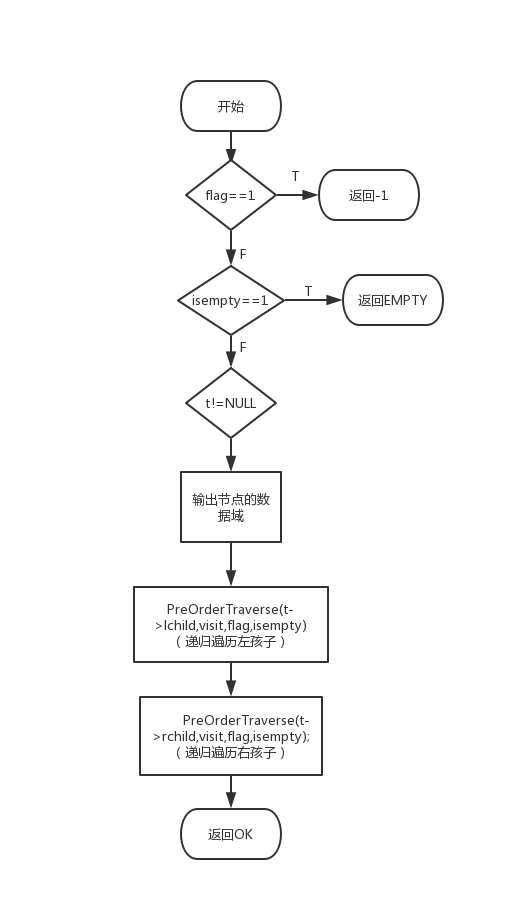


图3-11 先序遍历函数流程图

操作结果：先序遍历，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

（14）后序遍历：函数名称是status PostOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty)

初始条件：无

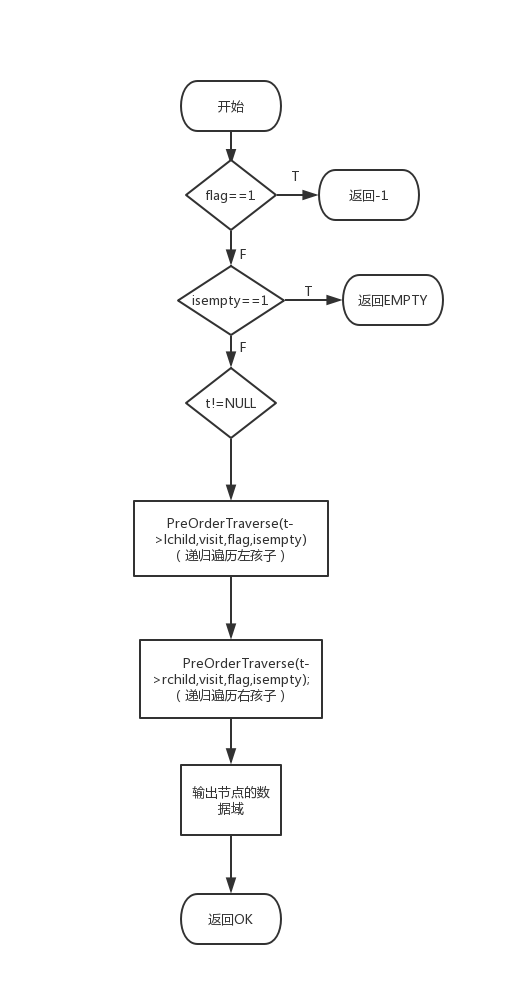
算法设计：

图3-12 后序遍历函数流程图

操作结果：后序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

（15）层序遍历：函数名称是status LevelOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag)

初始条件：无

算法设计：

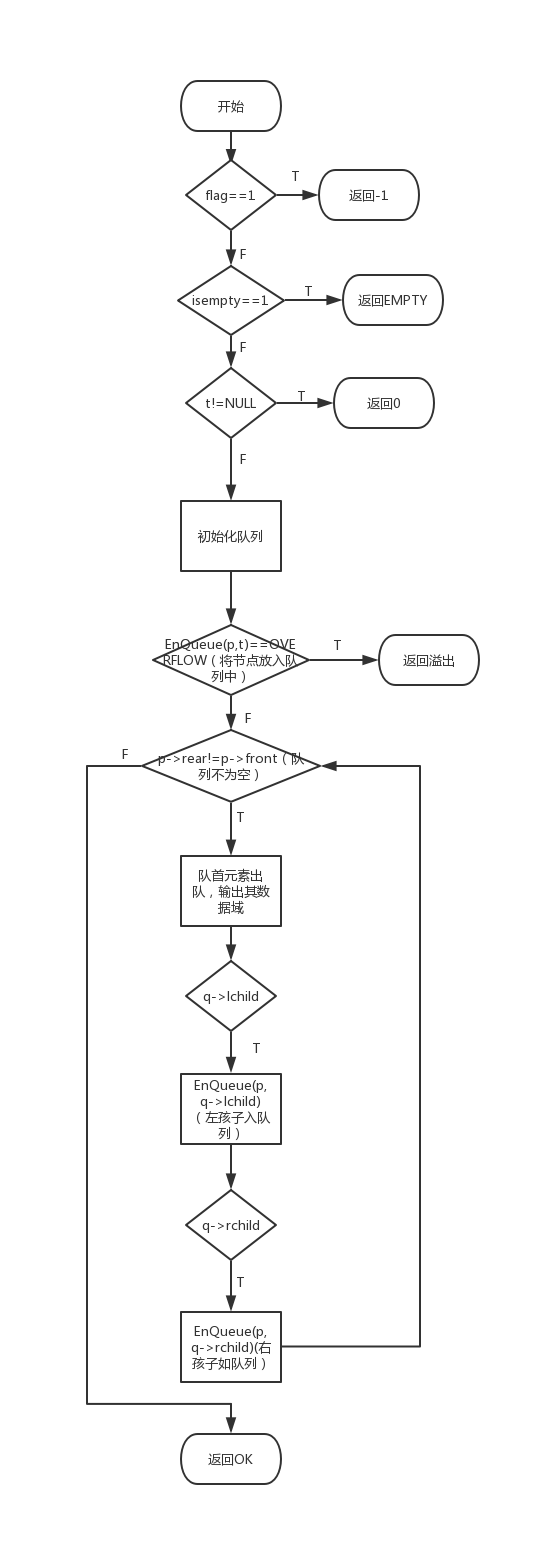


图3-13 层序遍历函数流程图

操作结果：操作结果是层序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败

（16）读入文件：函数名称是：status SaveTree(BiTree t,ElemType def,FILE\* fp)

初始条件：二叉树不为空且二叉树未被clear

算法设计：

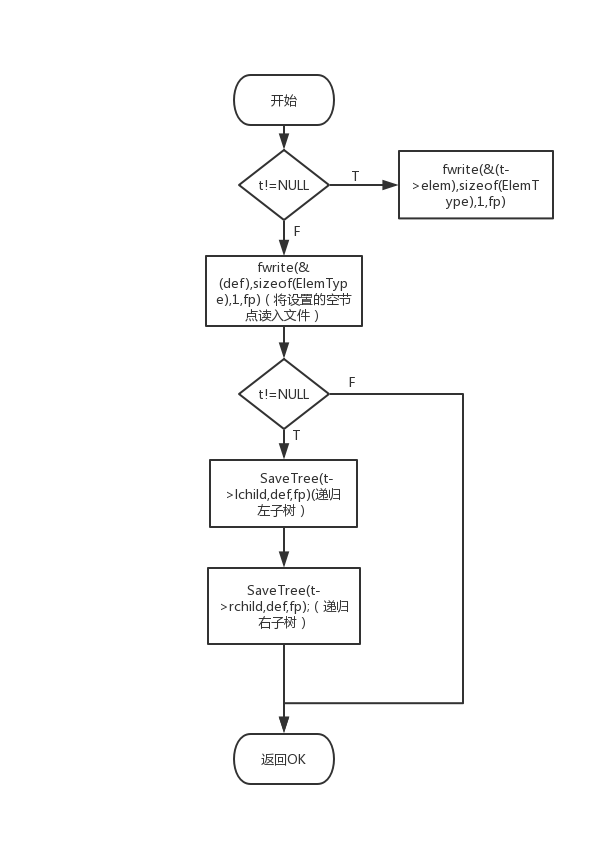


图3-14 文件读入函数流程图

操作结果：将当前树的节点信息和关系读入文件中

（17）载入二叉树：函数名称是status LoadTree(BiTree\* t,ElemType\* def,int count,int\* flag)

初始条件：无

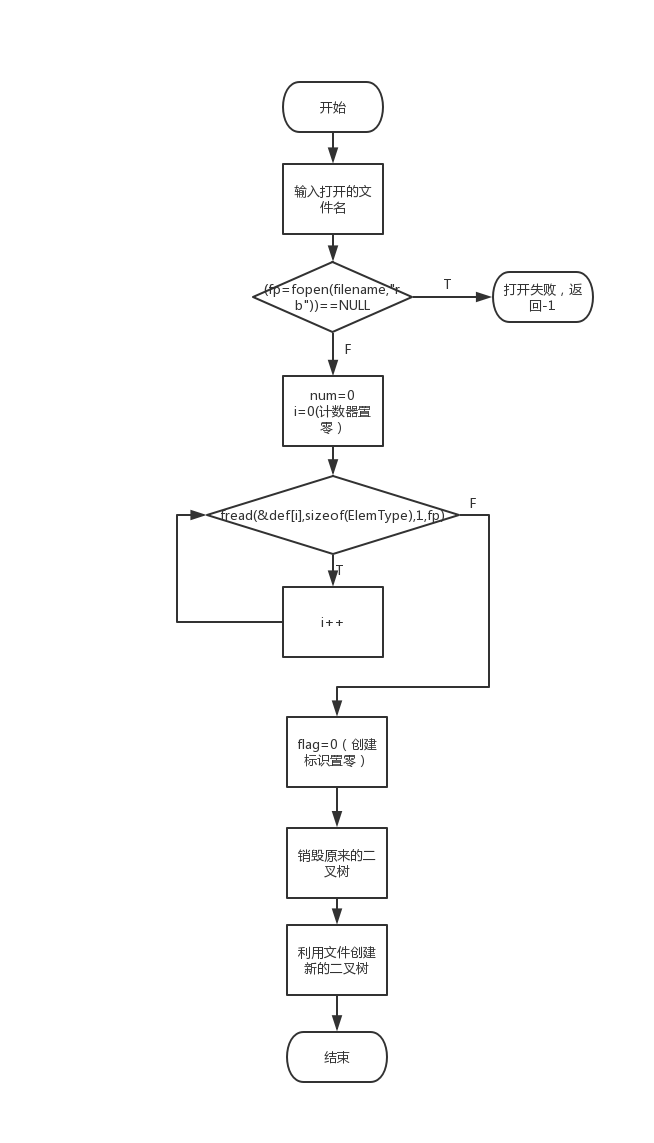
算法设计：

图3-15 载入二叉树函数流程图

操作结果：将之前读入文件的二叉树内容重新载入二叉树

3.3 系统实现

3.3.1功能实现

本次实验中使用的环境配置如下：

操作系统版本：Windows 10 专业版

编译器及其版本：GNN GCC Compiler

编程环境：CodeBlocks ver17.12

实验的完整代码参见附录A。

测试时通过用户界面对二叉树进行操作。用户界面参见图3-16，3-17。

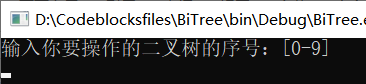


图3-16 选择操作的树的序号

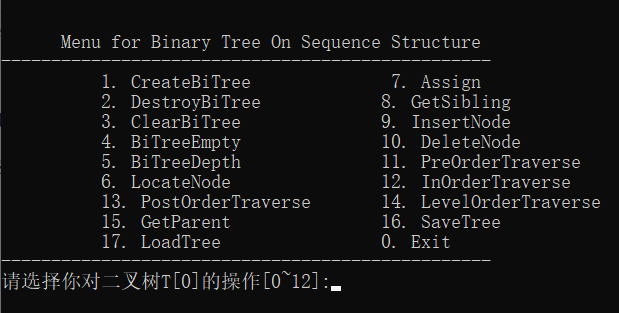


图3-17 选择对树的操作

3.3.2功能测试

测试时使用的数据：

|  |  |
| --- | --- |
| 树 | 树定义序列 |
| T[0] | 1 10，2 20，0，3 30，0，0 |
| T[1] | 1 1，2 2，3 3，0，4 4，0，0，5 5，6 6，0，0，0，7 7，8 8，0，0，9 9，0，0 |
| T[2] | 10 10，0，0 |

1. 测试函数：CreateTree

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 有空二叉树和正确定义时创建树 | 在已有空树  T[0]，输入命令 1,输入序列10 2 20 0 0 3 30 0 -1 | 根据定义创建T[0]树 | 输出“二叉树创建成功！” |



图3-18 创建二叉树

1. 测试函数：ClearTree

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 二叉树非空时清空树 | 在已有树时，输入命令3 | 清空A树，根节点保留。 | 输出“二叉树清空成功” |
| 二叉树为空时清空树 | 二叉树为空树时，输入命令3 | 操作失败 | 输出“二次函数是空树”。 |
| 无二叉树时清空树 | 在无树A时，输入命令3 | 操作失败 | 输出“二叉树尚未建立” |
| 二叉树未初始化 | 输入3 | 操作失败 | 输出“二叉树尚未创建！” |

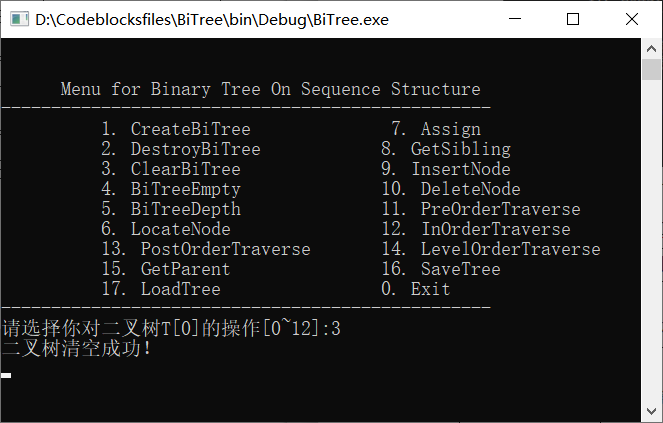


图3-19 二叉树存在清空

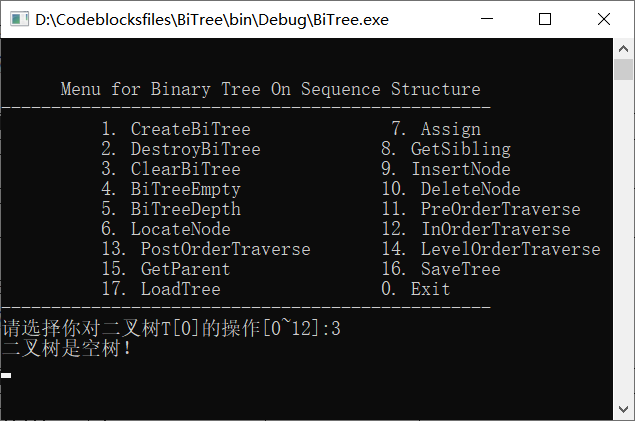


图3-20 二叉树不存在清空

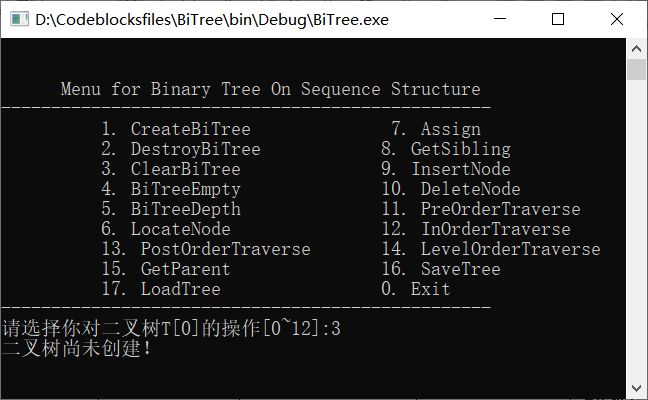


图3-21 二叉树未被初始化清空

1. 测试函数：DestroyTree

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 有二叉树时销毁树 | 在已有树存在，输入指令2 | 销毁树树。 | 输出“二叉树销毁成功！” |
| 无二叉树时销毁树 | 二次函数不存在时，输入指令2 | 操作失败 | 输出“二叉树是空树” |
| 二叉树未初始化 | 输入4 | 操作失败 | 输出“二叉树尚未创建！” |

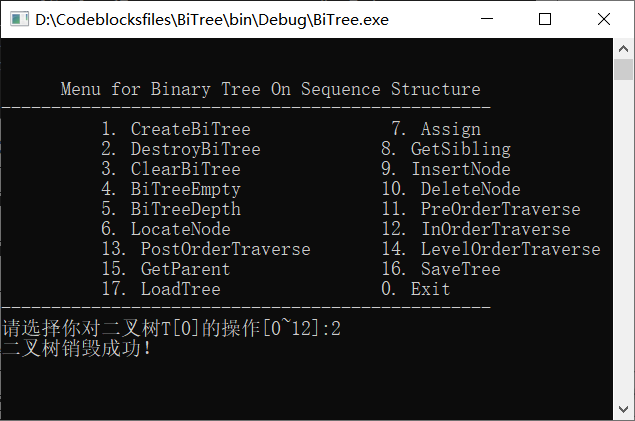


图3-22 二叉树存在销毁

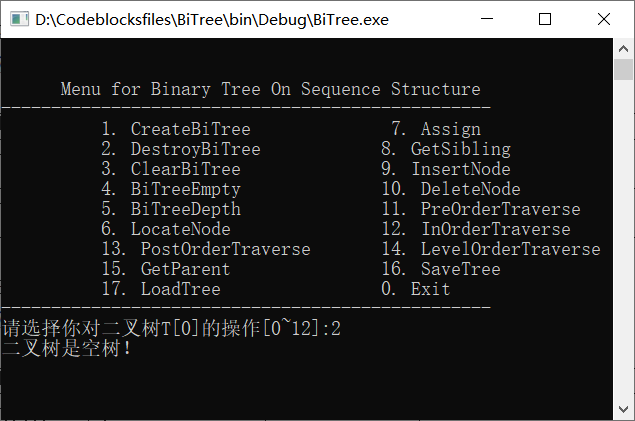


图3-23 二叉树不存在销毁

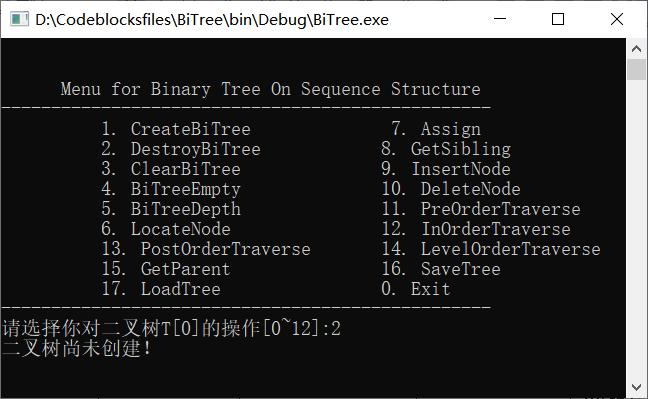


图3-24 二叉树未被初始化销毁

1. 测试函数：BiTreeEmpty

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 二叉树非空时判断树是否为空 | 在已有树时，输入命令4 | 二叉树非空。 | 输出“二叉树不是空的！” |
| 二叉树为空时判断树是否为空 | 在已有空树时，输入命令4 | 二叉树为空树。 | 输出“二次函数为空二叉树！” |
| 无二叉树时判断树是否为空 | 在无树时，输入命令 | 操作失败 | 输出“二叉树尚未创建” |
| 二叉树被clear后判断是否为空 | 输入4 | 操作失败 | 输出“二叉树被清空了！” |
| 二叉树未初始化 | 输入4 | 操作失败 | 输出“二叉树尚未创建！” |

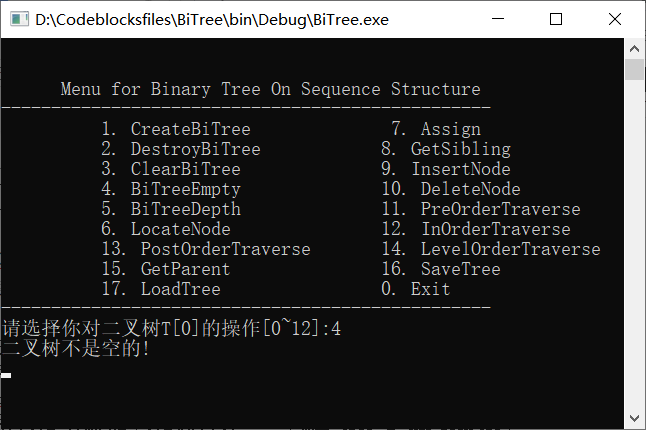


图3-25 树存在判空

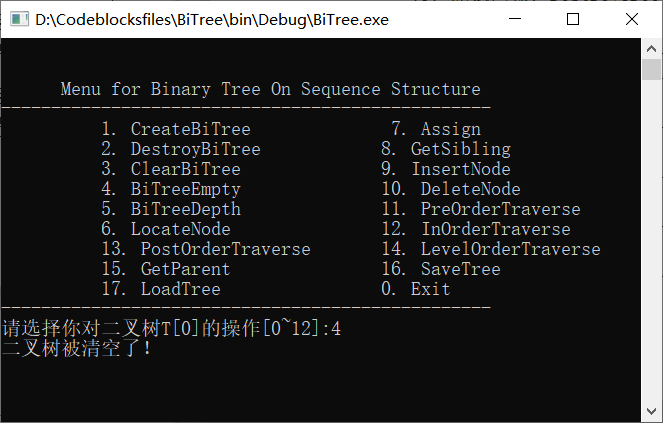


图3-26 树被clear判空

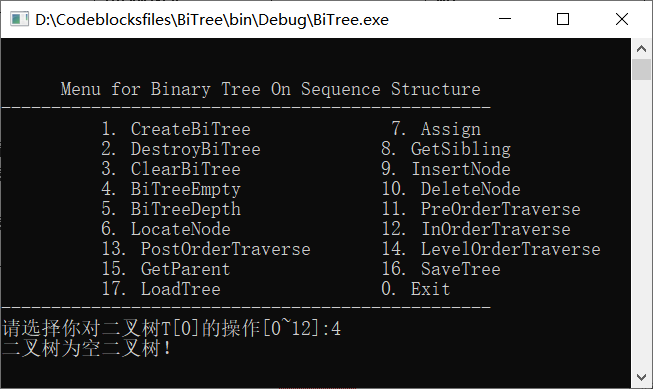


图3-27 树不存在判空

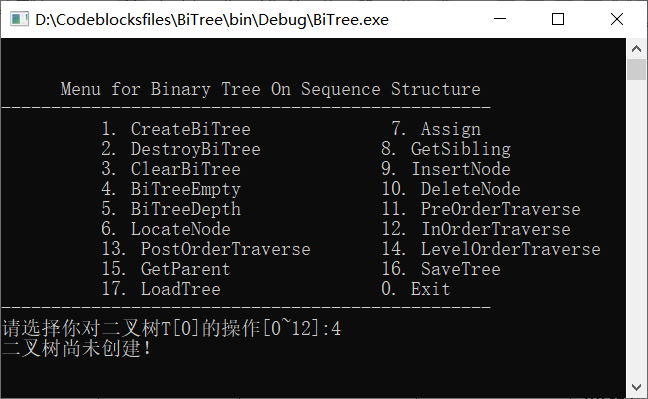


图3-28 树未被初始化判空

1. 测试函数：BiTreeDepth

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 二叉树非空时获取树深度 | 在已有树时，输入命令5 | 获取B树的深度，应为2 | 输出“二叉树的深度为：2.” |
| 二叉树为空时获取树深度 | 在已有空树时，输入命令5 | 空树深度为0 | 输出“二叉树的深度为：0.” |
| 无二叉树时获取树深度 | 在无树5时，输入命令5 | 操作失败 | 输出“Can't find tree "A"” |
| 二叉树被clear后判断是否为空 | 输入5 | 操作失败 | 输出“二叉树被清空了！” |
| 二叉树未初始化 | 输入6 | 操作失败 | 输出“二叉树尚未创建！” |

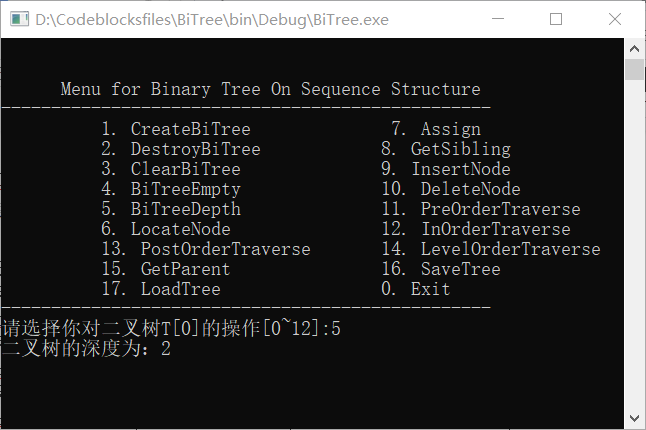


图3-29 树存在求深度

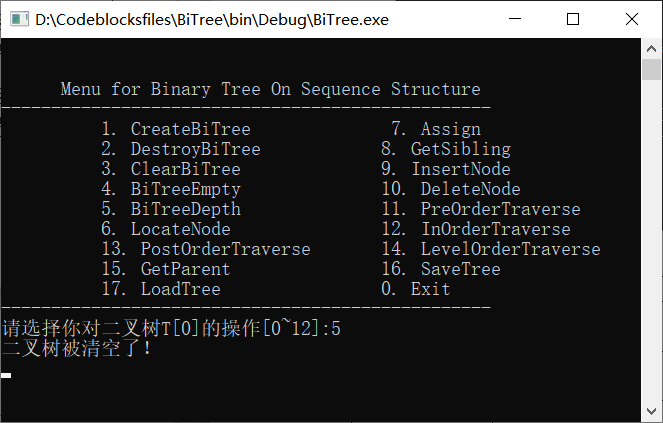


图3-30树被clear求深度

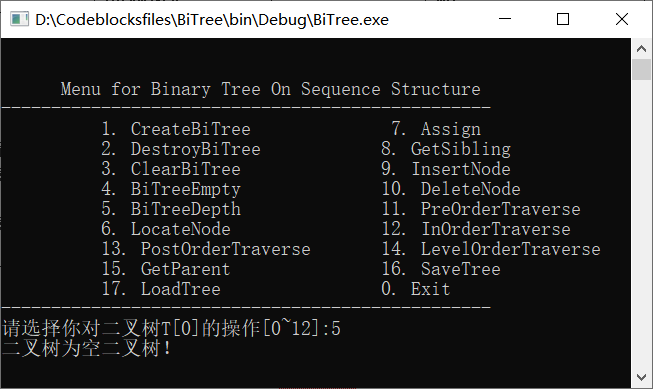
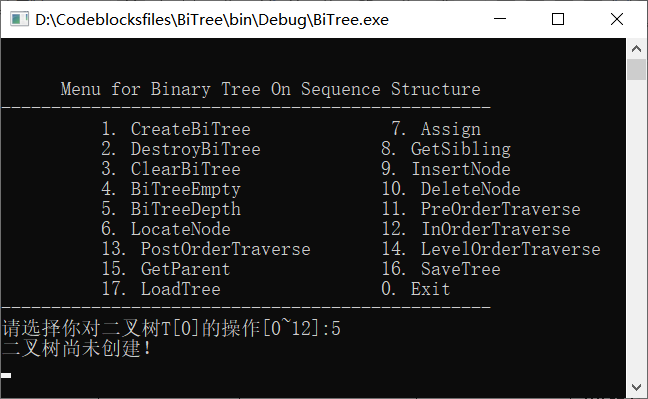


图3-31 树不存在求深度

图3-32 树未被初始化求深度

1. 测试函数：LocateNode

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 二叉树中某节点存在时获得其值 | 在已有树时，输入命令6，输入关键字1 | 获得树节点1的值，应为10 | 输出“关键字：1 内容：10.” |
| 二叉树中某节点不存在时获得其值 | 在已有树时，输入命令6，输入关键字 4 | 树中不存在节点4，获取失败 | 输出“找不到关键字为4的节点！” |
| 无二叉树时获得其值 | 在无树时，输入命令6 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树被clear后判断是否为空 | 输入6 | 操作失败 | 输出“二叉树被清空了！” |
| 二叉树未初始化 | 输入6 | 操作失败 | 输出“二叉树尚未创建！” |

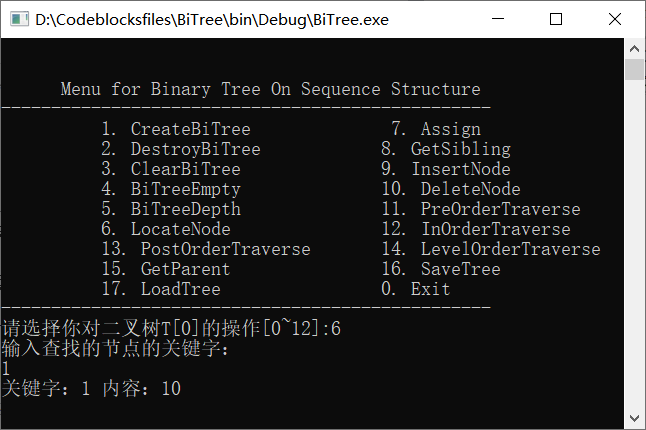


图3-33 树存在查找

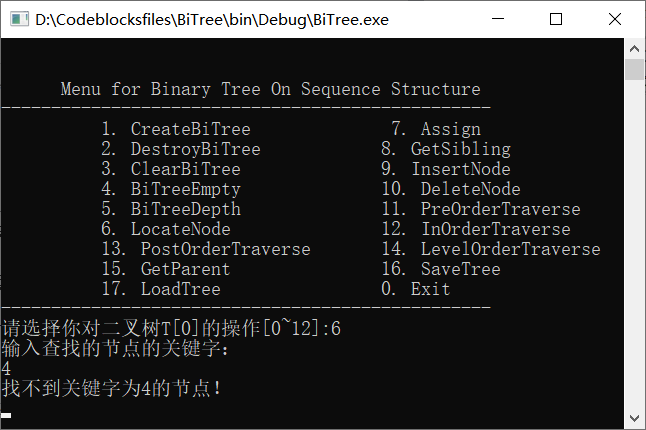


图3-34 书存在关键字不合法

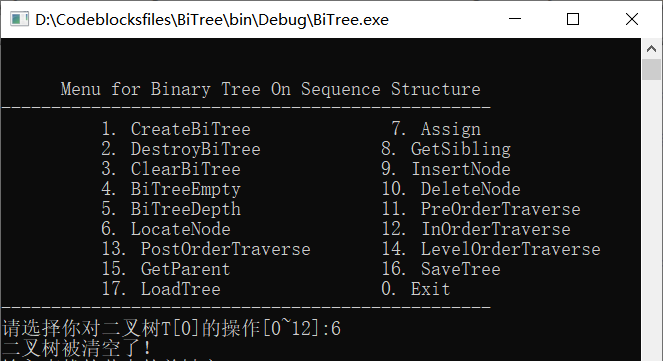


图3-35 树被clear查找

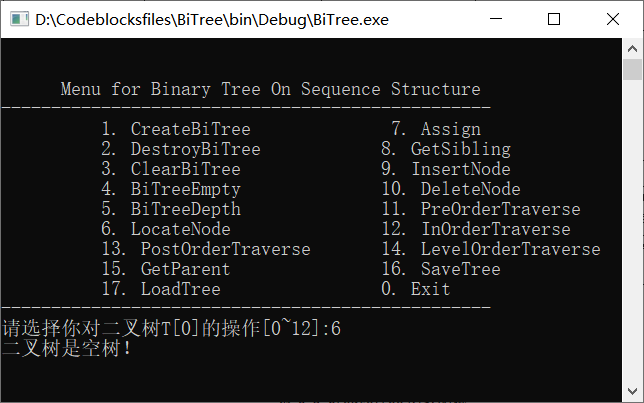


图3-36 树不存在查找

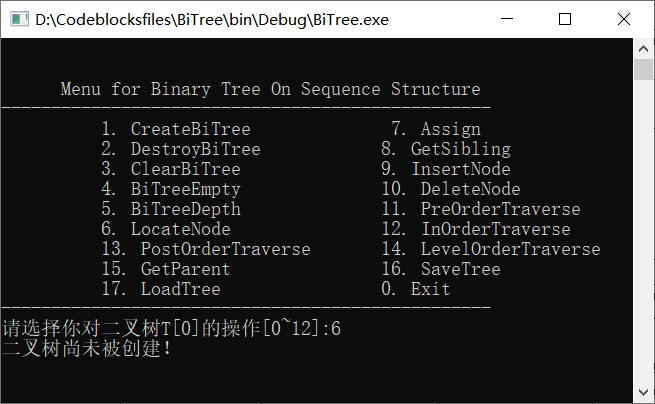


图3-37 树未被初始化查找

1. 测试函数：Assign

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 二叉树中某节点存在时赋值给该节点 | 在已有树时，输入命令7 输入1 50 | 给A树节点1赋值50 | 输出“赋值成功！” |
| 二叉树中某节点不存在时赋值给该节点 | 在已有树时，输入命令7，输入关键字和内容 4 50 | A树中不存在节点4，赋值失败 | 输出“输入的关键字不合法！” |
| 无二叉树时获得根节点 | 在无树时，输入命令7，输入关键字和内容1 20 | 操作失败 | 输出“二叉树是空二叉树!” |
| 二叉树被clear后判断是否为空 | 输入7，输入赋值的关键字1 20 | 操作失败 | 输出“二叉树被清空了！” |
| 二叉树未初始化 | 输入指令7输入关键字和内容1 2 | 操作失败 | 输出“二叉树尚未创建！” |

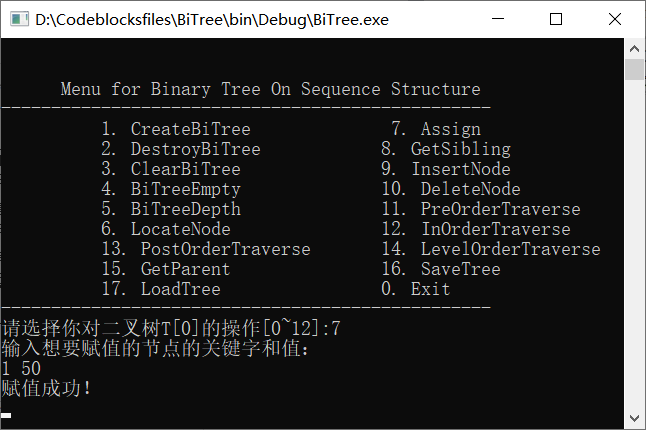


图3-38 树存在赋值

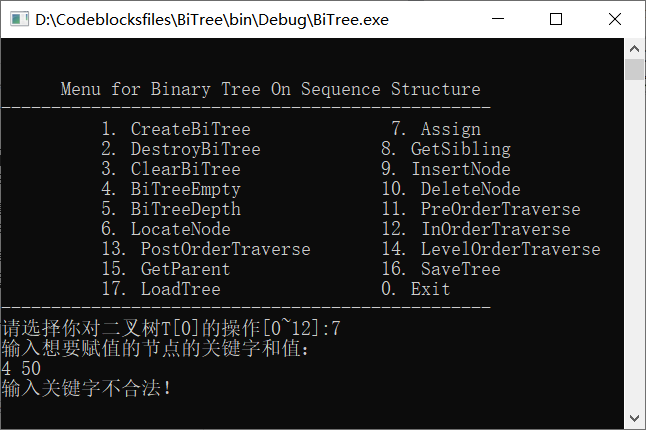


图3-39 树存在输入值不合法

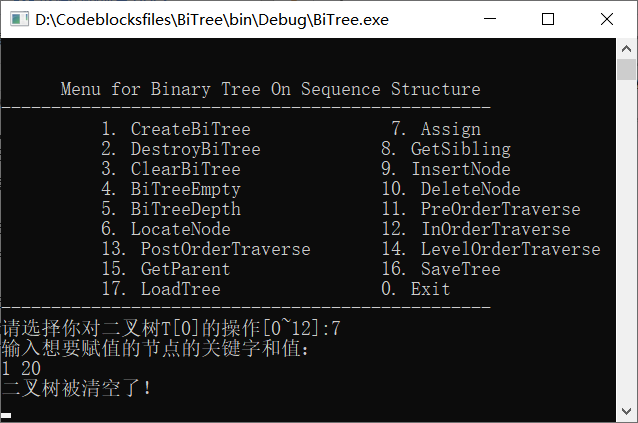


图3-40树被clear赋值

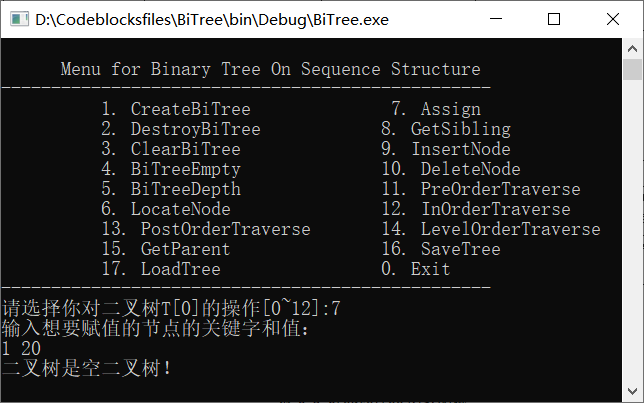


图3-41 树不存在赋值

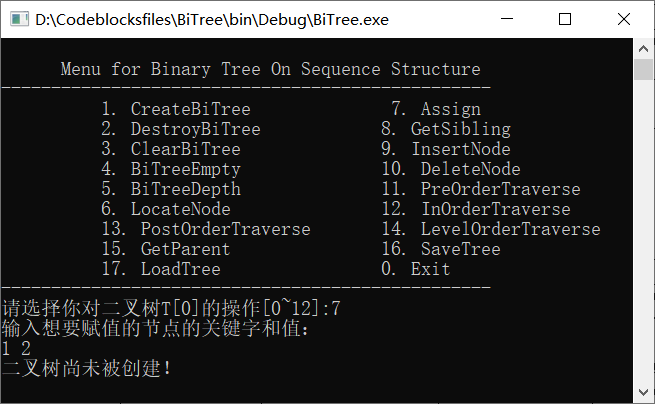


图3-42 树未被初始化赋值

1. 测试函数：GetSibling

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 二叉树中某节点存在兄弟时获得其兄弟 | 在已有树时，输入命令8,输入关键字3 | 获得A树节点3的左兄弟，应为{2,20} | 输出“该节点的兄弟节点的关键字：2 内容：20” |
| 二叉树中某节点不存在兄弟时获得其兄弟 | 在已有树时，输入命令L8，输入关键字1 | A树中节点1没有左兄弟，获取失败 | 输出“该节点没有兄弟节点！” |
| 二叉树中某节点不存在时获得其左兄弟 | 在已有树时，输入命令 8，输入关键字4 | 操作失败 | 输出“该节点没有兄弟节点！” |
| 无二叉树时获得兄弟 | 在无树时，输入命令8 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树被clear后判断是否为空 | 输入8，输入赋值的关键字1 | 操作失败 | 输出“二叉树被清空了！” |
| 二叉树未初始化 | 输入8 | 操作失败 | 输出“二叉树尚未创建！” |

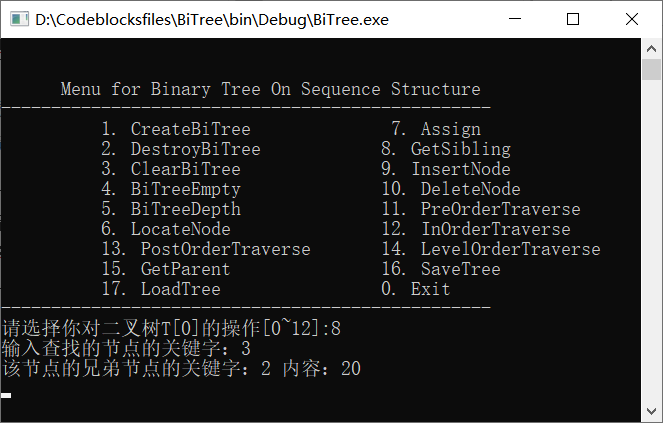


图3-43 树存在查兄弟

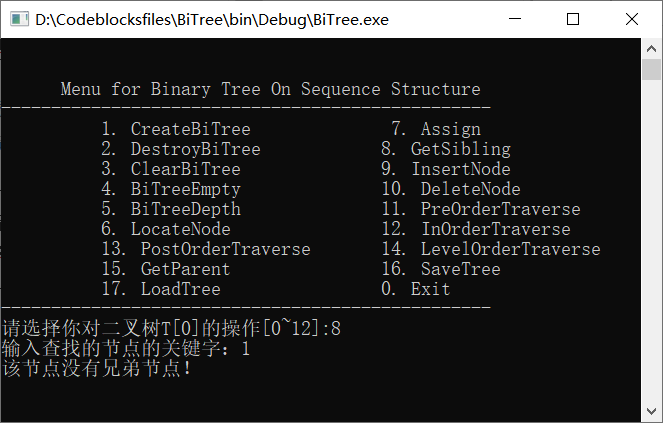


图3-44 树存在无兄弟

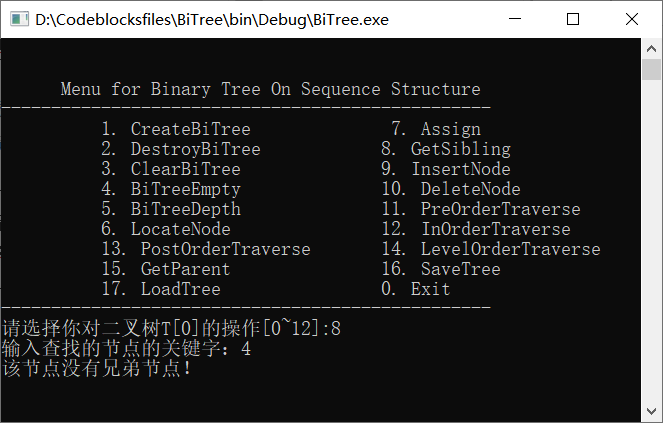


图3-45 树存在输入不合法

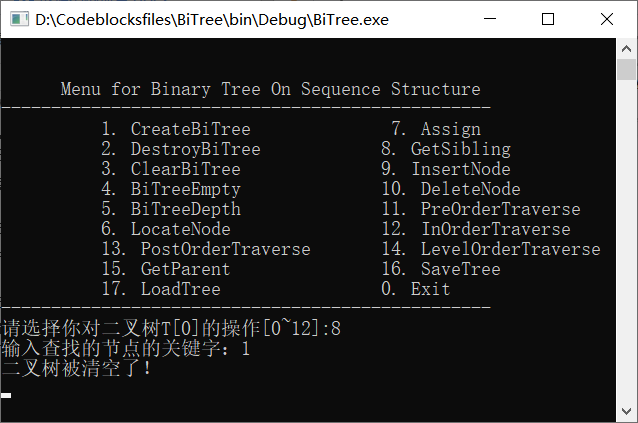


图3-46 树被clear查兄弟

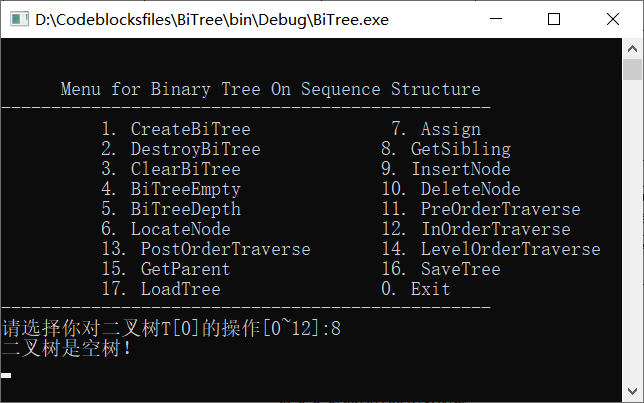


图3-47 树不存在查兄弟

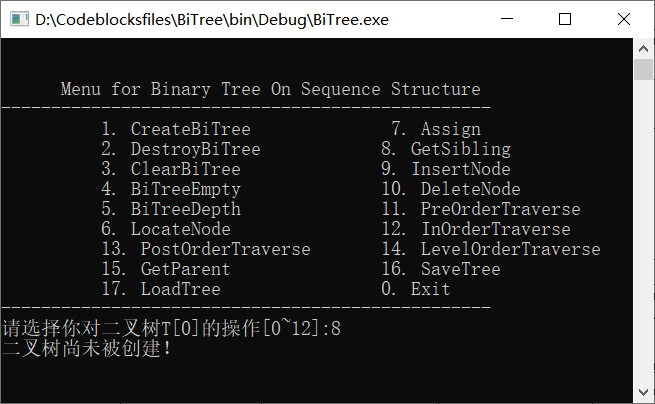


图3-48 树未被初始化查兄弟

1. 测试函数：InsertNode

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 向二叉树中某节点插入节点 | 在已有树A，输入9，输入关键字2，输入插入节点的关键字和内筒4 40，输入位置 0 | 在A树的2节点的左子树插入子树C | 输出“插入成功！” |
| 二叉树中某节点不存在时插入节点 | 在已有树，输入9，输入关键字5，输入插入的节点的关键字和内容5 50，输入位置0 | 操作失败 | 输出“输入关键字不合法！” |
| 无二叉树时插入节点 | 在无树时，输入命令9，输入关键字1，输入关键字和内容 2 3，输入位置 0 | 操作失败 | 输出“二叉树是空二叉树！” |
| 二叉树未初始化 | 输入指令9，输入关键字1，输入关键字和内容5 5 ，输入位置0 | 操作失败 | 输出“二叉树尚未创建！” |

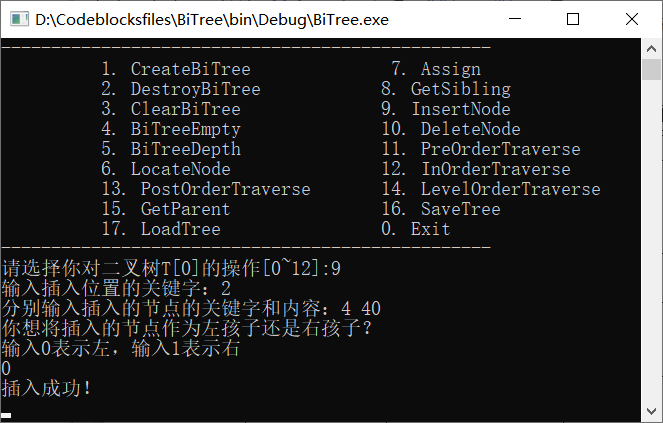


图3-49 树存在插入

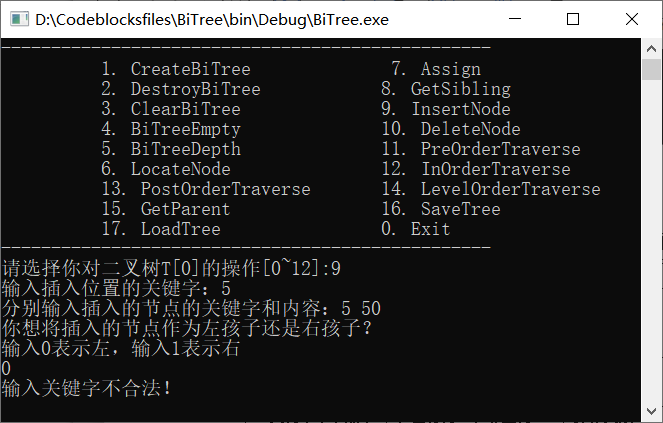


图3-50 树存在输入不合法

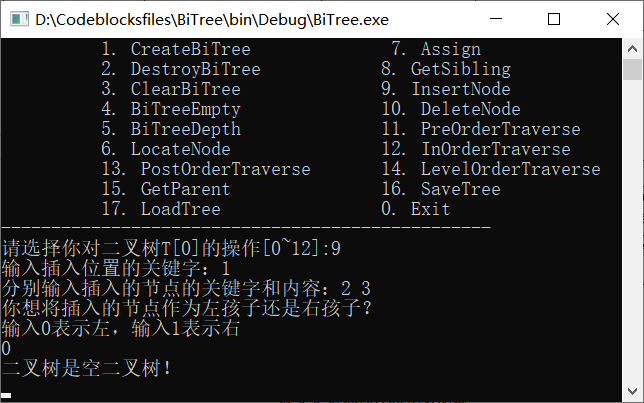


图3-51 树不存在插入

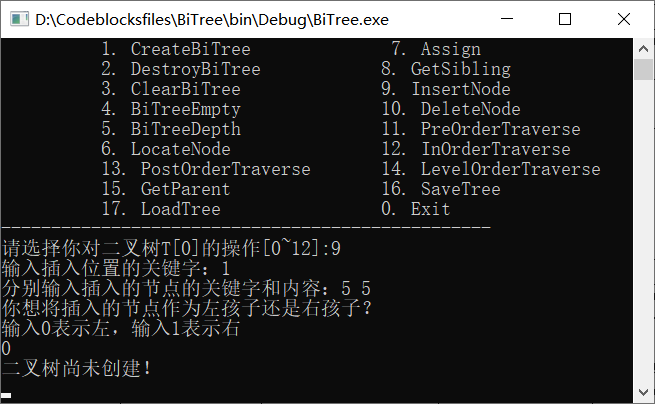


图3-52 树未被初始化插入

1. 测试函数：DeleteNode

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 输入的节点的关键字存在于二叉树中 | 在已有树时，输入命令10 输入删除的节点的关键字 1 | 删除二叉树节点关键字为1的节点 | 输出“删除成功！.” |
| 输入的关键字不存在于二叉树的任何节点中 | 在已有树时，输入命令10 输入关键字 5 | 二叉树中没有关键字为5的节点，删除失败 | 输出“输入关键字不合法！.” |
| 无二叉树时删除该节点的子树 | 在无树时，输入命令10，输入关键字1 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树未初始化 | 输入命令10，输入关键字1 | 操作失败 | 输出“二叉树尚未创建！” |

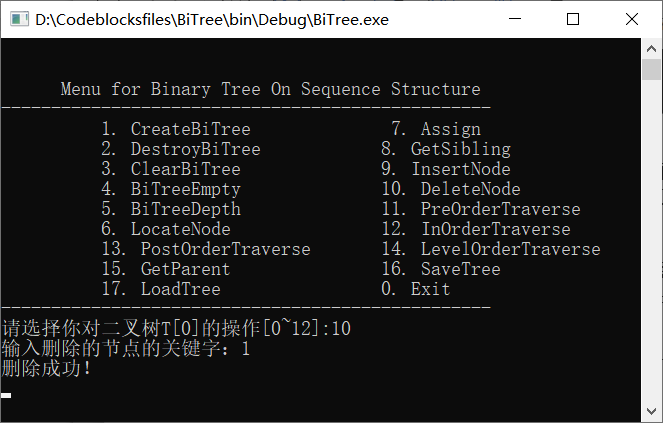


图3-53 树存在删除

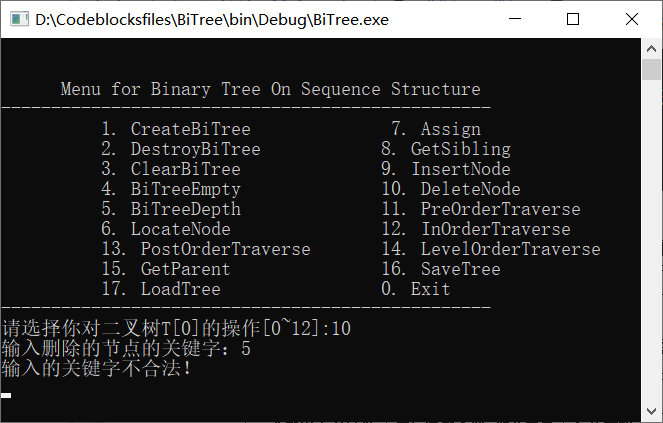


图3-54 树存在输入不合法

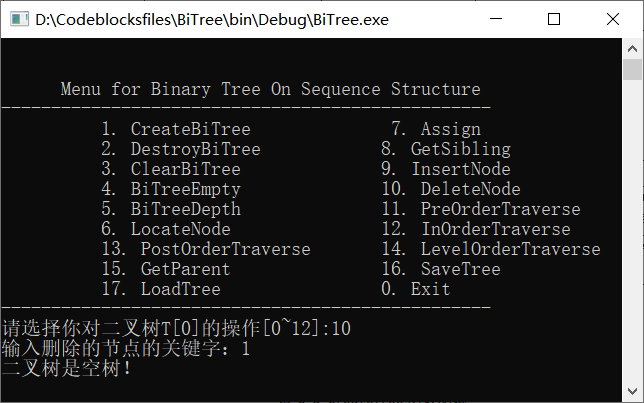


图3-55 树不存在

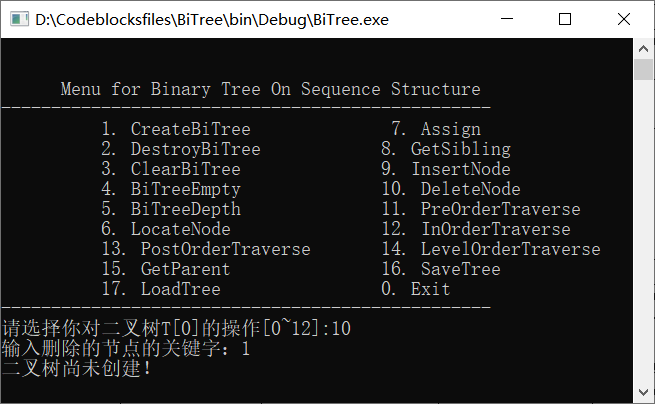


图3-56 树未被初始化

1. 测试函数：PreOrderTraverse

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 测试前序遍历 | 树存在时，输入命令11 | 前序遍历树A，序列应为“1 10，2 20， 3 30” | 输出“  前序遍历二叉树  关键字： 1，内容： 10  关键字：2 内容： 20  关键字：3 内容： 30” |
|  | 树被clear，输入11 | 操作失败 | 输出“前序遍历二叉树  二叉树被清空了！” |
|  | 树为空树，输入11 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树未初始化 | 输入11 | 操作失败 | 输出“前序遍历二叉树  二叉树尚未创建！” |

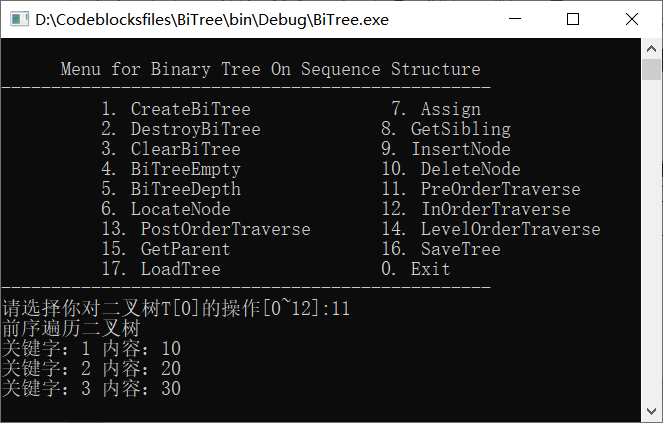


图3-57 树存在前序遍历

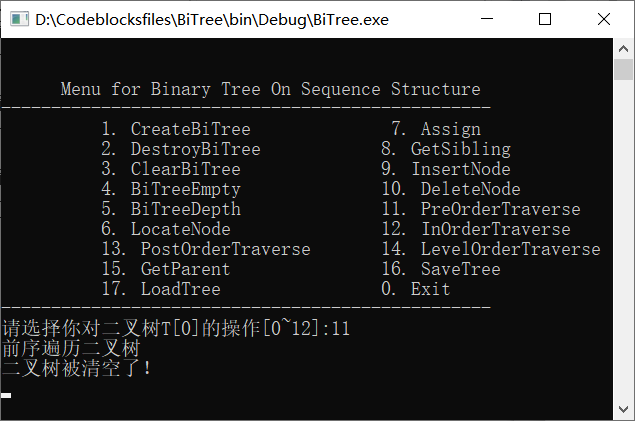


图3-58 树被clear遍历

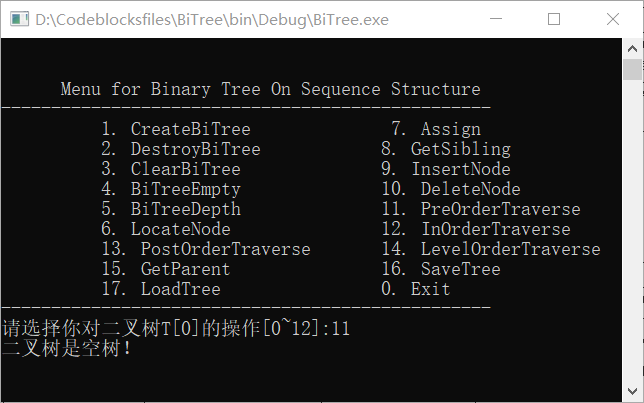


图3-59 树不存在遍历

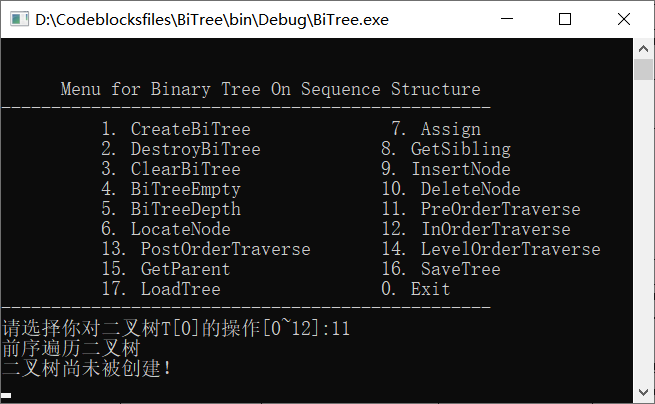


图3-60 树未被初始化遍历

1. 测试函数：InOrderTraverse

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 测试中序遍历 | 树存在时，输入命令12 | 中序遍历树，序列应为“2 20，1 10，3 30” | 输出“中序遍历二叉树  关键字：2，内容： 20  关键字：1，内容： 10  关键字：3，内容： 30” |
| 二叉树被clear后判断是否为空 | 输入12 | 操作失败 | 输出“中序遍历二叉树  二叉树被清空了！” |
| 二叉树为空树 | 输入12 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树未初始化 | 输入12 | 操作失败 | 输出“中序遍历二叉树  二叉树尚未创建！” |

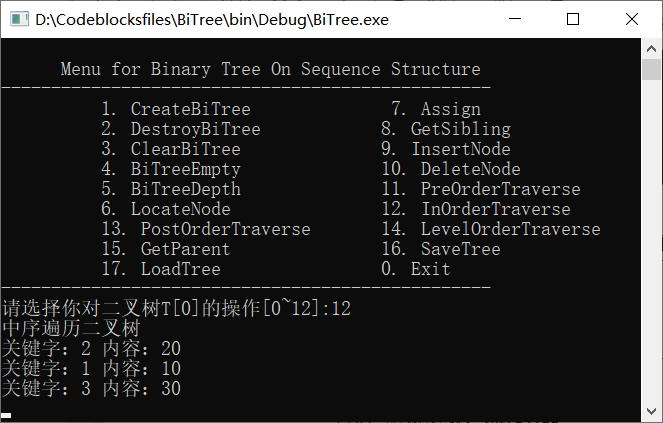


图3-61 树存在中序遍历

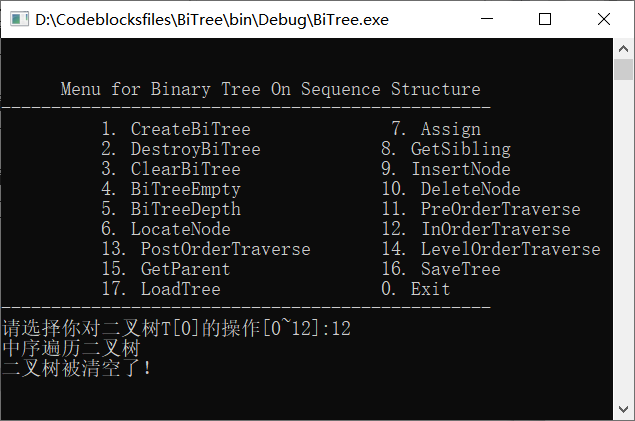


图3-62 树被clear遍历

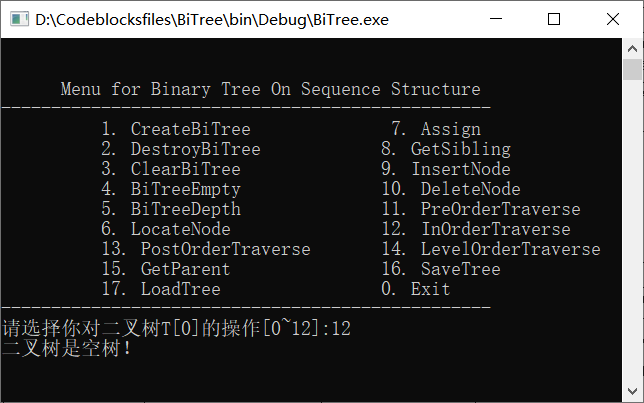


图3-63 树不存在遍历

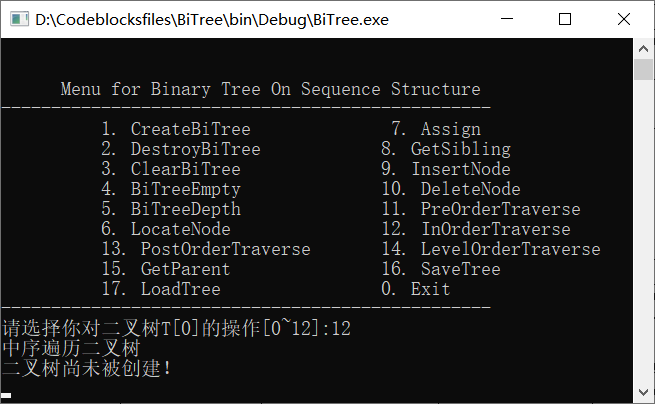


图3-64 树未被初始化遍历

1. 测试函数：PostOrderTraverse

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 测试后序遍历 | 二叉树存在时，输入命令13 | 后序遍历二叉树，序列应为“2 20，3 30，1 10” | 输出“后序遍历二叉树  关键字：2，内容： 20  关键字：3，内容： 30  关键字：1，内容： 10 |
| 二叉树被clear后判断是否为空 | 输入13 | 操作失败 | 输出“后序遍历二叉树  二叉树被清空了！” |
| 二叉树不存在 | 输入13 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树未初始化 | 输入13 | 操作失败 | 输出“后续遍历二叉树  二叉树尚未创建！” |

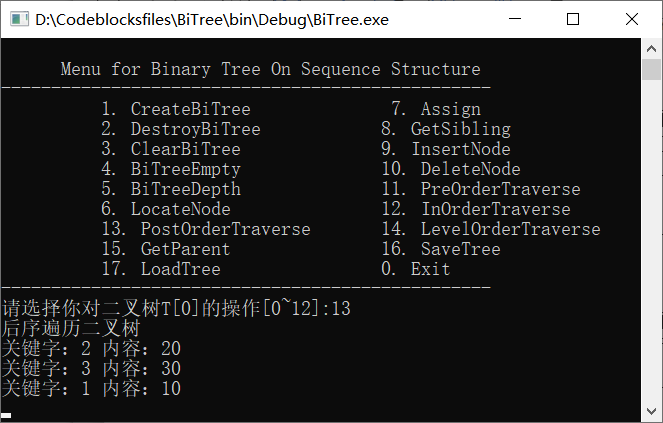


图3-65 树存在后序遍历

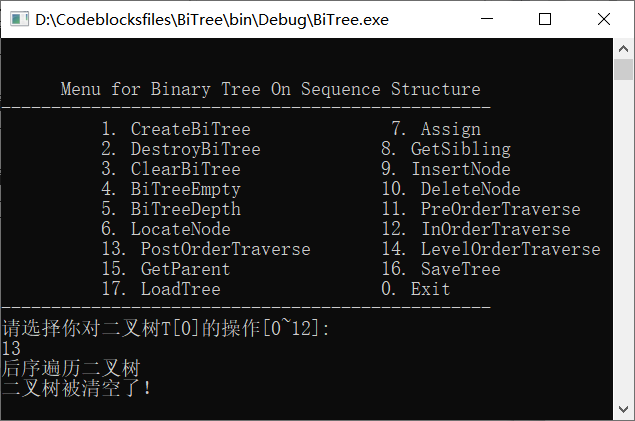


图3-66 树被clear遍历

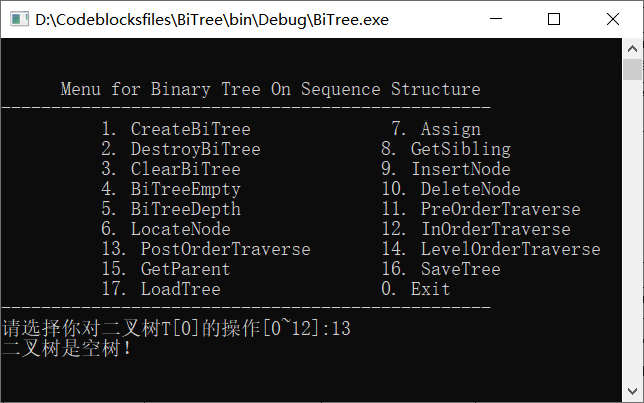


图3-67 树不存在遍历

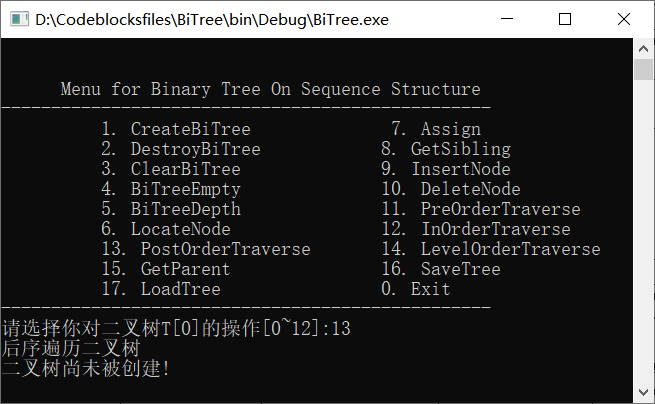


图3-68 树未被初始化遍历

1. 测试函数：LevelOrderTraverse

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 测试层序遍历 | 二叉树存在时，输入14 | 层序遍历二叉树，序列应为“1 10，2 20， 3 30” | 输出“层序遍历二叉树  关键字：1，内容： 10  关键字：2，内容： 20  关键字：3，内容： 30” |
| 二叉树被clear后判断是否为空 | 输入14 | 操作失败 | 输出“后序遍历二叉树  二叉树被清空了！” |
| 二叉树不存在 | 输入14 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树未初始化 | 输入14 | 操作失败 | 输出“二叉树尚未创建！” |

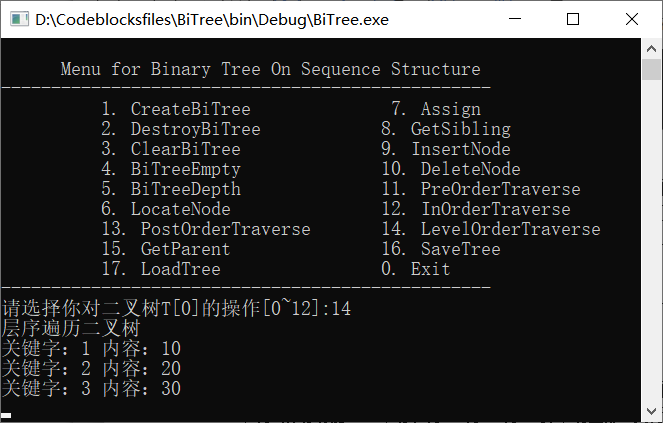


图3-69树存在层序遍历

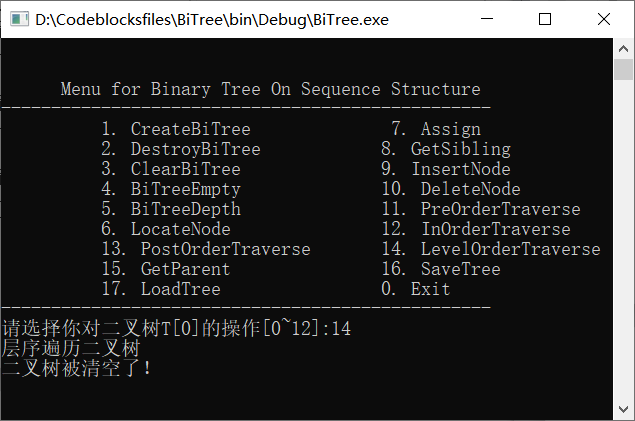


图3-70 树被clear遍历

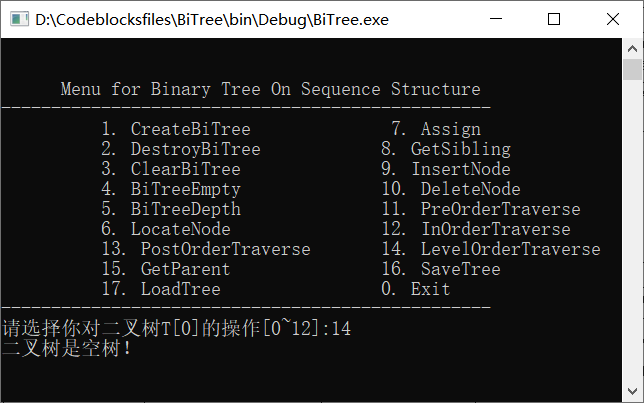


图3-71 树不存在遍历

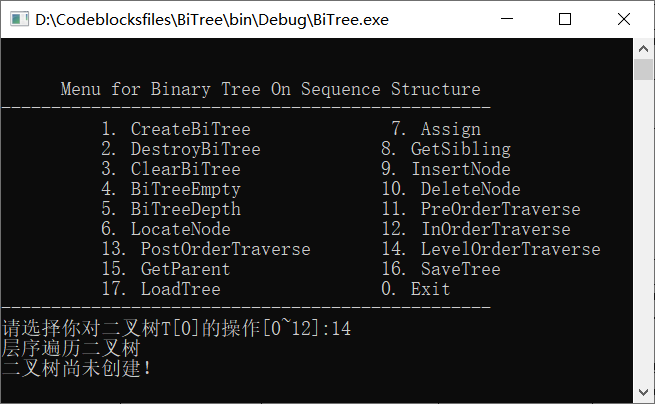


图3-72 树未被初始化遍历

1. 测试函数：GetParent

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 二叉树中某节点存在父节点时获得父节点 | 在已有树存在，输入指令15，输入关键字 2 | 获得节点2的父节点，应为{1,10} | 输出“该节点的双亲节点的关键字：1 内容：10” |
| 二叉树中某节点不存在父节点时获得父节点 | 在已有树存在，输入指令15，输入关键字1 | 树中节点1没有父节点，获取失败 | 输出“找不到双亲节点！.” |
| 二叉树中某节点不存在时获得父节点 | 在已有树存在，输入指令15，输入关键字 4 | 操作失败 | 输出“找不到双亲节点！” |
| 无二叉树时获得父节点 | 在无树时，输入命令15 | 操作失败 | 输出“二叉树是空树” |
| 二叉树被clear后判断是否为空 | 输入指令15，输入关键字 1 | 操作失败 | 输出“后序遍历二叉树  二叉树被清空了！” |
| 二叉树未初始化 | 输入14 | 操作失败 | 输出“二叉树尚未创建！” |

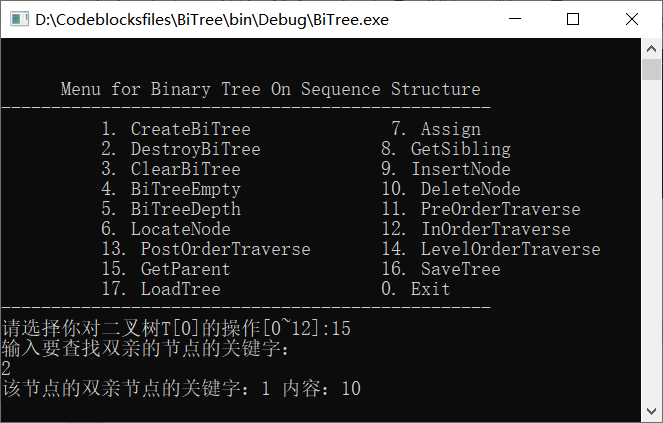


图3-73 树存在找双亲

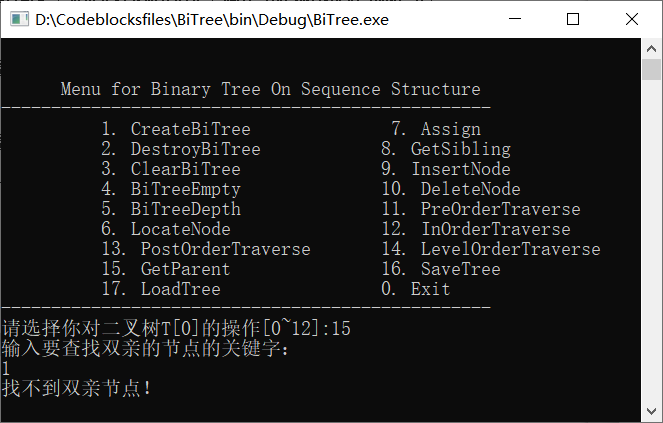


图3-74 树存在无双亲

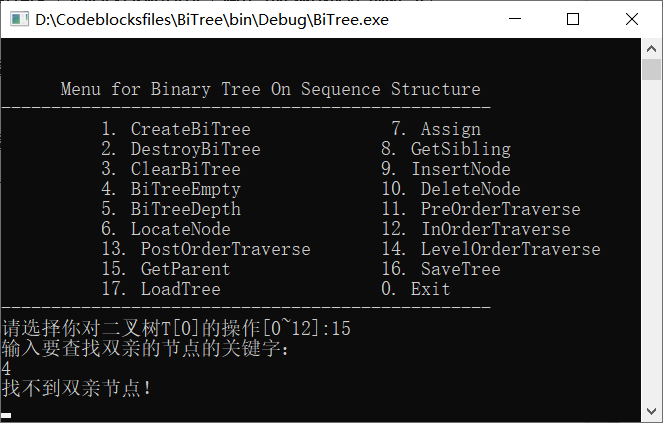


图3-75 树存在输入不合法

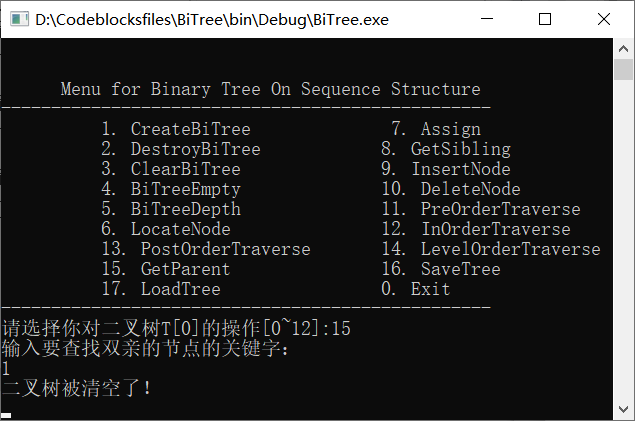


图3-76 树被clear



图3-77 树不存在

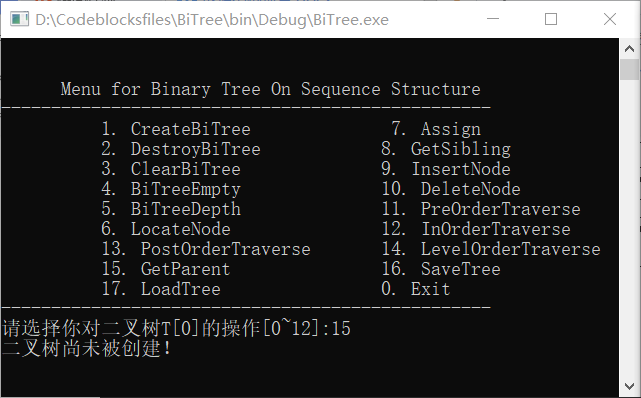


图3-78 树未被初始化

（16）测试函数：SaveTree

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 测试保存功能 | 存在树A、树B。树C时，输入 16 输入 data1 | 将全部二叉树保存到文件 |  |
| 二叉树被clear后判断是否为空 | 输入指令16，输入关键字 1 | 操作失败 | 输出“二叉树被清空了！” |
| 二叉树不存在 | 输入指令16 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树未初始化 | 输入14 | 操作失败 | 输出“二叉树尚未创建！” |

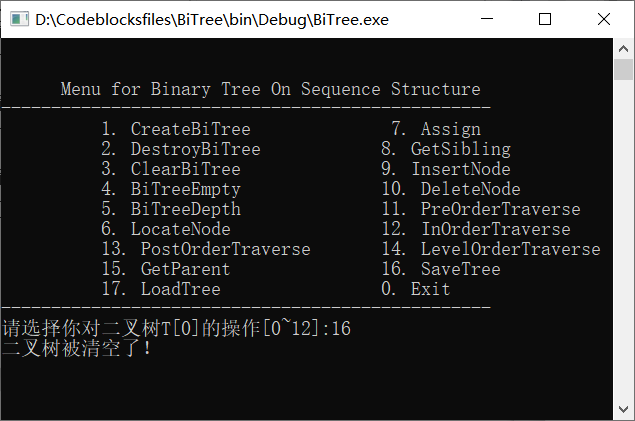


图3-79 树被clear

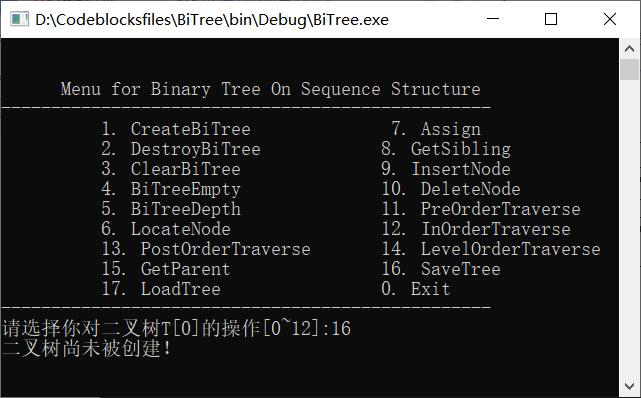


图3-80 树未被初始化

（17）测试函数：LoadTree

|  |  |  |  |
| --- | --- | --- | --- |
| 测试目的 | 测试条件&输入 | 理论结果 | 运行结果 |
| 测试读取功能 | 存在二叉树，存在文件 | 将文件的内容写入二次函数中 |  |
| 不存在二叉树时 | 操作失败 | 输出“二叉树是空树！” |
| 二叉树尚未被初始化时 | 操作失败 | 输出“二叉树尚未被创建！” |

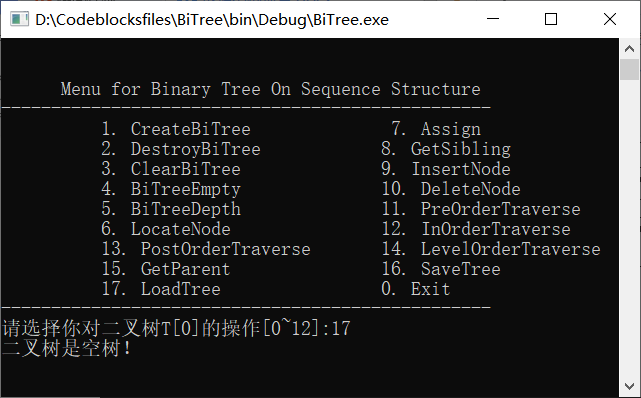


图3-81 树不存在

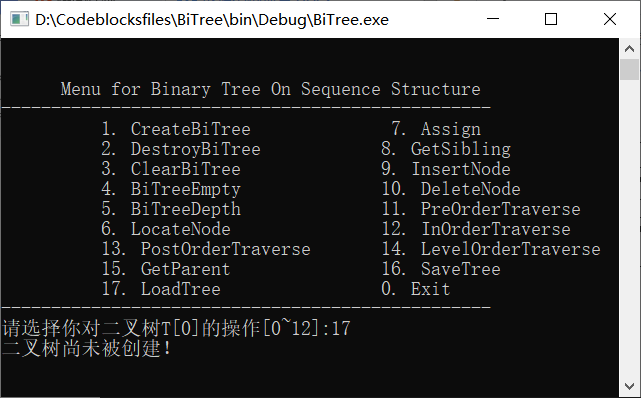


图3-82 树未被初始化

ps:

没有对文件的读写输出运行结果，需要调用遍历函数进行检验

插入节点时，若二叉树只是被clear,则仍然保留根节点，可以进行插入但仅能作为根节点的子节点，删除也是一样，但仅仅只能删除唯一的根节点

3.3.3 结果分析

实验测试结果说明了本次程序符合之前的要求，即完成实现了一个可以多二叉树操作、带有文件读取保存功能的基本二叉树操作程序。

3.4 实验小结

这次实验是关于二叉树的基本相关操作。二叉树是一种非线性结构，比之前的两个线性实验明显要复杂很多。随着实验的进行，我对菜单演示的基本流程有了更深一步的了解，也发现了几个前面实验可以改进的地方。此次实验遇到的问题和帮助如下：

1. 对defination的定义不够了解
2. 存在各种小问题
3. 这次实验代码长度明显比前两次要长，因此第一次调试的时候出现了更多小错误，比如不加分号等。
4. 无法在多棵树之间切换（之后可以了）
5. 此次实验几乎完全不同于前两次实验，所有函数几乎都要重新编写，在设计函数过程中，发现对一些基本的操作还是不够熟练，还有就是递归方面出现了结束递归语句的多次问题，这些均在代码中有做备注
6. 这次实验给我最大的感触出了递归的代码简洁性和思想复杂性，就是关于if和else if了，我在编写递归代码时开始都是直接全部用的if，没有用到else if,甚至也没有用else,但其实这样是很不好的，很有可能会导致递归调用后返回一个空指针而无法调用其数据域使程序异常结束，而且会多增加计算机的计算，下次应当注意
7. 还有一点发现就是关于首先判断二叉树是否为空或者被clear，发现可直接在main函数里判断，这样还不需要传入标志变量的参数，使代码更简洁
8. 这次实验还加深了我对全局变量的理解，全局变量定义在main函数之外，任何地方均可修改其值，并且直到整个程序运行完毕才会释放内存，这在递归计数时很有帮助。但如果在另外一个地方也要用到这个全局变量一定要记得修改值（代码备注中给出）

# 附录A 基于顺序存储结构线性表实现的源程序

代码段common.h

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <malloc.h>  /\*书本第10页，预定义常量和类型\*/  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  typedef int status;  typedef int ElemType; //数据元素类型定义  status equal(ElemType x,ElemType y){  return (x-y);  } |

代码段sqlist.h

|  |
| --- |
| #include"common.h"  /\*page22 \*/  /\*线性表的动态分配顺序存储结构\*/  #define LIST\_INIT\_SIZE 100 //初始分配量  #define LISTINCREMENT 10 //分配增量  typedef struct SqList{ //顺序表（顺序结构）的定义  ElemType\* elem; //存储空间基址  int length; //当前长度  int listsize;//当前分配的存储容量  }SqList;  status InitList(SqList\* L);  status DestroyList(SqList\* L);  status ClearList(SqList\* L);  status ListEmpty(SqList\* L);  int ListLength(SqList\* L);  status GetElem(SqList\* L,int i,ElemType\* e);  status LocateElem(SqList\* L,ElemType e,int (\*compare)(ElemType x,ElemType y)); //简化过  status PriorElem(SqList\* L,ElemType cur,ElemType\* pre\_e);  status NextElem(SqList\* L,ElemType cur,ElemType\* next\_e);  status ListInsert(SqList\* L,int i,ElemType e);  status ListDelete(SqList\* L,int i,ElemType\* e);  status ListTrabverse(SqList\* L); //简化过  void PrintMune(SqList\* L,int i);  status SaveList(SqList\* L);  status LoadList(SqList\* L); |

代码段main.c

|  |
| --- |
| #include"sqlist.h"  int main(int argc,char\* argv[])  {  SqList L[10];  int count=0,op=1;  char c;  //初始化线性表  while(op)  {  PrintMune(&L[count],count);  count++;  printf("继续创建下一个线性表? [Y/N]\n");  getchar();op=((c=getchar())=='Y')?1:0; //由用户确定需要创建线性表的数量  }  printf("欢迎下次再使用本系统！\n");  return 0;  }  void PrintMune(SqList\* L,int order)  {  int op=1; int ans=0;  while(op){  system("cls"); printf("\n\n");  printf(" Menu for Linear Table On Sequence Structure \n");  printf("-------------------------------------------------\n");  printf(" 1. InitList 7. LocateElem\n");  printf(" 2. DestroyList 8. PriorElem\n");  printf(" 3. ClearList 9. NextElem \n");  printf(" 4. ListEmpty 10. ListInsert\n");  printf(" 5. ListLength 11. ListDelete\n");  printf(" 6. GetElem 12. ListTrabverse\n");  printf(" 13. SaveList 14. LoadList\n");  printf(" 0. Exit\n");  printf("-------------------------------------------------\n");  printf(" 请选择你对线性表L[%d]的操作[0~12]:",order);  scanf("%d",&op);  switch(op){  case 1:  //printf("\n----IntiList功能待实现！\n");  if((ans=InitList(L))==OK) printf("线性表创建成功！\n");  else if(ans==OVERFLOW) printf("创建失败！\n");  else printf("线性表已经存在！\n");  getchar();getchar();  break;  case 2:  //printf("\n----DestroyList功能待实现！\n");  if(DestroyList(L)==OK) printf("线性表销毁成功！\n");  else printf("表不存在，无法销毁！\n");  getchar();getchar();  break;  case 3:  //printf("\n----ClearList功能待实现！\n");  if(ClearList(L)==OK) printf("线性表清空成功！\n");  else printf("表不存在，清空失败！\n");  getchar();getchar();  break;  case 4:  //printf("\n----ListEmpty功能待实现！\n");  if((ans=ListEmpty(L))) printf("线性表是空表！\n");  else if(ans==INFEASTABLE) printf("线性表不存在！\n");  else printf("线性表不是空表！\n");  getchar();getchar();  break;  case 5:  //printf("\n----ListLength功能待实现！\n");  if((ans=ListLength(L))>=0)  printf("线性表的长度为：%d\n",ans);  else  printf("线性表不存在！\n");  getchar();getchar();  break;  case 6:  //printf("\n----GetElem功能待实现！\n");  printf("请输入你想要获取的数据的位置：\n");  int i,e=0;  scanf("%d",&i);  if((ans=GetElem(L,i,&e)))//这里的ans只是一个布尔值  printf("表中第%d个元素的值为：%d\n",i,e);//不是ans应该是e  else  printf("输入的数据不合法!\n");  getchar();getchar();  break;  case 7:  {  //printf("\n----LocateElem功能待实现！\n");  int e;  printf("输入你想要查找的数：\n");  scanf("%d",&e);  if((ans=LocateElem(L,e,equal)))  printf("你所要查找的数的位置为：%d\n",ans);  else  printf("查找失败！\n");  getchar();getchar();  break;  }  case 8:  //printf("\n----PriorElem功能待实现！\n");  {  int cur\_e,pre\_e=0;  printf("输入你想要查找的数：\n");  scanf("%d",&cur\_e);  if(PriorElem(L,cur\_e,&pre\_e))  printf("你所要查找的数的前驱为：%d\n",pre\_e);  else  printf("查找失败！\n");  getchar();getchar();  break;  }  case 9:  {  //printf("\n----NextElem功能待实现！\n");  int cur\_e,next\_e=0;  printf("输入你想要查找的数：\n");  scanf("%d",&cur\_e);  if((NextElem(L,cur\_e,&next\_e)))  printf("你所要查找的数的后继为：%d\n",next\_e);  else  printf("在线性表中找不到你所要查找的数的后继！\n");  getchar();getchar();  break;  }  case 10:  {  //printf("\n----ListInsert功能待实现！\n");  printf("输入你想插入的位置和输入的元素：\n");  int i,e;  scanf("%d %d",&i,&e);  if(ListInsert(L,i,e))  printf("插入成功！\n");  else  printf("输入位置不合法，插入失败！\n");  getchar();getchar();  break;  }  case 11:  {  //printf("\n----ListDelete功能待实现！\n");  printf("输入你想删除的位置：\n");  int i,e=0;  scanf("%d",&i);  if(ListDelete(L,i,&e))  printf("删除成功！你所删除的数据为：%d\n",e);  else  printf("输入位置不合法，删除失败！\n");  getchar();getchar();  break;  }  case 12:  {  //printf("\n----ListTrabverse功能待实现！\n");  if(!ListTrabverse(L)) printf("线性表是空表！\n");  else if(ListTrabverse(L)==INFEASTABLE) printf("线性表不存在！\n");  getchar();getchar();  break;  }  case 0:  break;  case 13:  SaveList(L);break;  case 14:  LoadList(L);  break;  }//end of switch  }//end of while  }//end of main()  status SaveList(SqList\* L){ //调用时传递参数&L[i]  FILE \*fp;char filename[30];  printf("input file name: ");  scanf("%s",filename);  if ((fp=fopen(filename,"wb"))==NULL)  {  printf("File open error\n ");  exit(-1);  }  fwrite(L->elem,sizeof(ElemType),L->length,fp);//将线性表的数据写入file中做备份  fclose(fp);  return 0;  }  status LoadList(SqList\* L){  L->length=0;  FILE \*fp;char filename[30];  printf("输入需要载入的文件的名称：");  scanf("%s",filename);  if ((fp=fopen(filename,"rb"))==NULL)  {  printf("File open error\n ");  exit(-1);  }  while(fread(&L->elem[L->length],sizeof(ElemType),1,fp))  L->length++;  //当线性表种的数据被修改后，可以重新将备份文件中的数据读入线性表表中，恢复其原来的状态  fclose(fp);  return 0;  }  status InitList(SqList\* L){  /\*构造一个空的线性表\*/  L->elem=(ElemType\*)malloc(LIST\_INIT\_SIZE\*sizeof(ElemType));  if(!L->elem) exit(OVERFLOW);  L->length=0;  L->listsize=LIST\_INIT\_SIZE;  return OK;  }  status DestroyList(SqList\* L){  /\*销毁一个已存在的线性表\*/  if(!L->elem)  return INFEASTABLE;  L->listsize=L->length=0;  free(L->elem);  L->elem=NULL;  return OK;  }  status ClearList(SqList\* L){  /\*清空一个已存在的线性表\*/  if(!L->elem)  return INFEASTABLE;  L->length=0;  return OK;  }  status ListEmpty(SqList\* L){  /\*判断线性表是否为空\*/  if(!L->elem)  return INFEASTABLE;  if(L->length) return FALSE;  return TRUE;  }  status ListLength(SqList\* L){  /\*返回线性表的元素长度\*/  if(!L->elem)  return INFEASTABLE;  return L->length;  }  status GetElem(SqList\* L,int i,ElemType\* e){  /\*用e返回L中的第i个元素的值\*/  if(!L->elem)  return INFEASTABLE;  if(i<1||i>L->length)  return ERROR; //输入的i值不合法  \*e=\*(L->elem+i-1);  return OK;//第一种写法return \*e,这样会导致如果第i个元素的值为0出现判断错误  }  status LocateElem(SqList\* L,ElemType e,int (\*compare)(ElemType x,ElemType y)){  /\*compare函数为函数的指针调用\*/  if(!L->elem)  return INFEASTABLE;  for(int i=0;i<L->length;i++)  if((\*compare)(L->elem[i],e)==0)  return i+1;  return FALSE;  }  status PriorElem(SqList\* L,ElemType cur,ElemType \*pre\_e){  /\*若cur是L的数据元素且不是第一个，则用pre\_e返回前驱\*/  /\*否则操作失败\*/  if(!L->elem)  return INFEASTABLE;  int j;  for(int i=0;i<L->length;i++)  if((j=LocateElem(L,cur,equal))>1)  {  \*pre\_e=\*(L->elem+j-2); //本来Locate返回的是位序而不是数组中的下标，位序比下标要大1，所以应该减2  return OK;  }  return ERROR;  }  status NextElem(SqList\* L,ElemType cur,ElemType\* next\_e){  /\*若cur是L的数据元素且不是最后一个，用next返回后继\*/  /\*否则操作失败\*/  if(!L->elem)  return INFEASTABLE;  int j;  for(int i=0;i<L->length;i++)  if((j=LocateElem(L,cur,equal))>0&&j<L->length)  {  \*next\_e=\*(L->elem+j); // 同理上面的prior函数，这里只需要j,不需要加1  return OK;  }  return ERROR;  }  status ListInsert(SqList\* L,int i,ElemType e){  /\*在L的第i个位置前插入新的数据e，长度加1\*/  if(!L->elem)  return INFEASTABLE;  if(i<0||i>L->length+1) return ERROR;  if(L->length>=L->listsize){  //当前分配空间已满，增加分配  ElemType\* newbase=(ElemType\*)realloc(L->elem,(L->listsize+LISTINCREMENT)\*sizeof(ElemType));  if(!newbase) exit(OVERFLOW); //存储内存分配失败  L->elem=newbase;  L->listsize +=LISTINCREMENT;  }  for(int j=L->length-1;j>=i-1;j--)  \*(L->elem+j+1)=\*(L->elem+j); //插入位置及之后的元素后移  \*(L->elem+i-1)=e;  ++L->length;  return OK;  }  status ListDelete(SqList\* L,int i,ElemType\* e){  /\*删除L的第i个数据，并用e返回其值，L的长度减1\*/  if(i<0||i>L->length) return ERROR;  \*e=\*(L->elem+i-1); //第二次修改时并没有在e前面加取值符号，导致e的值在调用函数后没有改变  for(int j=i-1;j<L->length;j++)  L->elem[j]=L->elem[j+1];  --L->length;  return OK; //第一次修改时返回的是\*e,但实际上，\*e的值变成了前移后的地i个数，即想返回的数的后一个数  }  status ListTrabverse(SqList\* L){  int i;  if(!L->elem)  return INFEASTABLE;  printf("\n-----------all elements -----------------------\n");  for(i=0;i<L->length;i++)  printf("%d ",L->elem[i]);  printf("\n------------------ end ------------------------\n");  return L->length;  } |

附录B 基于链式存储结构线性表实现的源程序

代码段common.h

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <malloc.h>  /\*书本第10页，预定义常量和类型\*/  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  typedef int status;  typedef int ElemType; //数据元素类型定义  status equal(ElemType x,ElemType y){  return (x-y);  } |

代码段sqlist.h

|  |
| --- |
| #include"common.h"  /\*page22 \*/  /\*线性表的动态分配顺序存储结构\*/  #define LIST\_INIT\_SIZE 100 //初始分配量  #define LISTINCREMENT 10 //分配增量  //结点类型  typedef struct LNode{ //顺序表（顺序结构）的定义  ElemType elem; //结点的数据  struct LNode \*next; //指向下一个结点的指针  }LNode,\*LinkList;  //链表类型  status InitList(LinkList\* headp,int\* flag);  status DestroyList(LinkList\* headp,int flag);  status ClearList(LinkList headp,int flag);  status ListEmpty(LinkList headp,int flag);  int ListLength(LinkList headp,int flag);  status GetElem(LinkList headp,int i,ElemType\* e,int flag);  status LocateElem(LinkList headp,ElemType e,int (\*compare)(ElemType x,ElemType y),int flag); //简化过  status PriorElem(LinkList headp,ElemType cur,ElemType\* pre\_e,int flag);  status NextElem(LinkList headp,ElemType cur,ElemType\* next\_e,int flag);  status ListInsert(LinkList headp,int i,ElemType e,int flag);  status ListDelete(LinkList headp,int i,ElemType\* e,int flag);  status ListTrabverse(LinkList headp,int flag); //简化过  status SaveList(LinkList headp);  status LoadList(LinkList headp); |

代码段main.c

|  |
| --- |
| #include "sqlist.h"  void main(int argc,char\* argv[])  {  LinkList L[10];  int op=1; int ans=0;int count=0;char c;int flag=1;  while(op)  {  while(op){  system("cls"); printf("\n\n");  printf(" Menu for Linear Table On Sequence Structure \n");  printf("-------------------------------------------------\n");  printf(" 1. InitList 7. LocateElem\n");  printf(" 2. DestroyList 8. PriorElem\n");  printf(" 3. ClearList 9. NextElem \n");  printf(" 4. ListEmpty 10. ListInsert\n");  printf(" 5. ListLength 11. ListDelete\n");  printf(" 6. GetElem 12. ListTrabverse\n");  printf(" 13. SaveList 14. LoadList\n");  printf(" 0. Exit\n");  printf("-------------------------------------------------\n");  printf(" 请选择你对链表L[%d]的操作[0~12]:",count);  scanf("%d",&op);  switch(op){  case 1:  //printf("\n----IntiList功能待实现！\n");  if((ans=InitList(&L[count],&flag))==OK) printf("链表创建成功！\n");  else if(ans==OVERFLOW) printf("创建失败！\n");  else printf("链表已经存在！\n");  getchar();getchar();  break;  case 2:  //printf("\n----DestroyList功能待实现！\n");  if(DestroyList(&L[count],flag)==OK) printf("链表销毁成功！\n");  else printf("表不存在，无法销毁！\n");  getchar();getchar();  break;  case 3:  //printf("\n----ClearList功能待实现！\n");  if(ClearList(L[count],flag)==OK) printf("链表清空成功！\n");  else printf("表不存在，清空失败！\n");  getchar();getchar();  break;  case 4:  //printf("\n----ListEmpty功能待实现！\n");  if((ans=ListEmpty(L[count],flag))==1) printf("链表是空表！\n");  else if(ans==INFEASTABLE) printf("链表不存在！\n");  else printf("链表不是空表！\n");  getchar();getchar();  break;  case 5:  //printf("\n----ListLength功能待实现！\n");  if((ans=ListLength(L[count],flag))>=0)  printf("链表的长度为：%d\n",ans);  else  printf("链表不存在！\n");  getchar();getchar();  break;  case 6:  //printf("\n----GetElem功能待实现！\n");  printf("请输入你想要获取的数据的位置：\n");  int i,e=0;  scanf("%d",&i);  if((ans=GetElem(L[count],i,&e,flag))==1)//这里的ans只是一个布尔值  printf("表中第%d个元素的值为：%d\n",i,e);//不是ans应该是e  else if(ans==0)  printf("输入的数据不合法!\n");  else printf("链表不存在！\n");  getchar();getchar();  break;  case 7:  {  //printf("\n----LocateElem功能待实现！\n");  int e;  printf("输入你想要查找的数：\n");  scanf("%d",&e);  if((ans=LocateElem(L[count],e,equal,flag))>0)  printf("你所要查找的数的位置为：%d\n",ans);  else if(ans==0)  printf("查找失败！\n");  else  printf("链表不存在！\n");  getchar();getchar();  break;  }  case 8:  //printf("\n----PriorElem功能待实现！\n");  {  int cur\_e,pre\_e=0;  printf("输入你想要查找的数：\n");  scanf("%d",&cur\_e);  if((ans=PriorElem(L[count],cur\_e,&pre\_e,flag))==1)  printf("你所要查找的数的前驱为：%d\n",pre\_e);  else if(ans==0)  printf("查找失败！\n");  else  printf("链表不存在！\n");  getchar();getchar();  break;  }  case 9:  {  //printf("\n----NextElem功能待实现！\n");  int cur\_e,next\_e=0;  printf("输入你想要查找的数：\n");  scanf("%d",&cur\_e);  if((ans=NextElem(L[count],cur\_e,&next\_e,flag))==1)  printf("你所要查找的数的后继为：%d\n",next\_e);  else if(ans==0)  printf("查找失败！\n");  else  printf("链表不存在！\n");  getchar();getchar();  break;  }  case 10:  {  //printf("\n----ListInsert功能待实现！\n");  printf("输入你想插入的位置和输入的元素：\n");  int i,e;  scanf("%d %d",&i,&e);  if((ans=ListInsert(L[count],i,e,flag))==1)  printf("插入成功！\n");  else if(ans==0)  printf("输入位置不合法，插入失败！\n");  else  printf("链表不存在！\n");  getchar();getchar();  break;  }  case 11:  {  //printf("\n----ListDelete功能待实现！\n");  printf("输入你想删除的位置：\n");  int i,e=0;  scanf("%d",&i);  if((ans=ListDelete(L[count],i,&e,flag))==1)  printf("删除成功！你所删除的数据为：%d\n",e);  else if(ans==0)  printf("输入位置不合法，删除失败！\n");  else  printf("链表不存在！\n");  getchar();getchar();  break;  }  case 12:  {  //printf("\n----ListTrabverse功能待实现！\n");  if((ans=ListTrabverse(L[count],flag))==0) printf("链表表是空表！\n");  else if(ans==INFEASTABLE) printf("链表不存在！\n");  getchar();getchar();  break;  }  case 0:  printf("欢迎下次再使用本系统！\n");break;  case 13:  SaveList(L[count]);break;  case 14:  LoadList(L[count]);  break;  }//end of switch  }//end of while  count++;  printf("继续创建下一个线性表? [Y/N]\n");  getchar();op=((c=getchar())=='Y')?1:0; //由用户确定需要创建线性表的数量  }  }//end of main()  status SaveList(LinkList headp){  FILE \*fp;char filename[30];  printf("input file name: ");  scanf("%s",filename);  if ((fp=fopen(filename,"wb+"))==NULL)  {  printf("File open error\n ");  exit(-1);  }  LinkList p=(headp)->next;  while(p)  {  fwrite(&(p->elem),sizeof(ElemType),1,fp);//将线性表的数据写入file中做备份  p=p->next;  }  fclose(fp);  return 0;  }  status LoadList(LinkList headp){  FILE \*fp;char filename[30];  printf("输入需要载入的文件的名称：");  scanf("%s",filename);  if ((fp=fopen(filename,"rb"))==NULL)  {  printf("File open error\n ");  exit(-1);  }  LinkList p=headp;  LinkList add=(LinkList)malloc(sizeof(LNode));  while(fread(&(add->elem),sizeof(ElemType),1,fp))  {//将文件中的数据元素写入新的结点中，再新的结点插入链表中  add->next=NULL;  p->next=add;  p=add;  add=(LinkList)malloc(sizeof(LNode));  }  //当线性表种的数据被修改后，可以重新将备份文件中的数据读入线性表表中，恢复其原来的状态  free(add);  fclose(fp);  return 0;  }  status InitList(LinkList\* headp,int \*flag){  /\*构造一个空的线性表\*/  \*headp=(LinkList)malloc(sizeof(LNode));  if(!headp) exit(OVERFLOW);  (\*headp)->next=NULL;  \*flag=0;  return OK;  }  status DestroyList(LinkList\* headp,int flag){  /\*销毁一个已存在的带头节点的线性表\*/  if(!\*headp||flag)  return INFEASTABLE;  LinkList p=\*headp;  while(p){  \*headp=p->next;  free(p);  p=\*headp;  }  return OK;  }  status ClearList(LinkList headp,int flag){  /\*清空一个已存在的线性表\*/  /\*清空是链表没节点，但是链表还在，可以继续插入节点。\*/  if(!headp||flag)  return INFEASTABLE;  LinkList p=headp->next,q;  while(p)  {  q=p->next;  free(p);  p=q;  }  headp->next=NULL;  return OK;  }  status ListEmpty(LinkList headp,int flag){  /\*判断线性表是否为空\*/  if(!headp||flag)  return INFEASTABLE;  if(headp->next==NULL)  return TRUE;  else return FALSE;  }  status ListLength(LinkList headp,int flag){  /\*返回线性表的元素长度\*/  if(!headp||flag) return INFEASTABLE;  LinkList p=headp->next;  int count=0;  while(p)  {  count++;  p=p->next;  }  return count;  }  status GetElem(LinkList headp,int i,ElemType\* e,int flag){  /\*用e返回L中的第i个元素的值\*/  if(!headp||flag) return INFEASTABLE;  LinkList p=headp->next;  int j=1; //计数器  while(p&&j<i){ //顺序查找直到p指向第i个元素或者为空  p=p->next;  ++j;  }  if(!p||j>i) return ERROR; // 第i个元素不存在  \*e=p->elem; // 取出元素  return OK;  }  status LocateElem(LinkList headp,ElemType e,int (\*compare)(ElemType x,ElemType y),int flag){  /\*compare函数为函数的指针调用\*/  if(!headp||flag) return INFEASTABLE;  LinkList p=headp->next;  int count=0;  while(p)  {  if((\*compare)(p->elem,e)==0) return count+1;  else  {  p=p->next;  count++;  }  }  return FALSE;  }  status PriorElem(LinkList headp,ElemType cur,ElemType \*pre\_e,int flag){  /\*若cur是L的数据元素且不是第一个，则用pre\_e返回前驱\*/  /\*否则操作失败\*/  if(!headp||flag) return INFEASTABLE;  LinkList p=headp->next;  if(!p) return ERROR; //判定是否为空表  if(p->elem==cur) return ERROR; // cur是第一个元素  while(p->next)  {  if(p->next->elem==cur){  \*pre\_e=p->elem;return OK;  }  p=p->next;  }  return ERROR;  }  status NextElem(LinkList headp,ElemType cur,ElemType\* next\_e,int flag){  /\*若cur是L的数据元素且不是最后一个，用next返回后继\*/  /\*否则操作失败\*/  if(!headp||flag) return INFEASTABLE;  LinkList p=headp->next;  if(!p) return ERROR;  LinkList q=p->next;  while(q){  if(p->elem==cur){  \*next\_e=q->elem;  return OK;  }  else{  p=q;q=q->next;  }  }  return ERROR;  }  status ListInsert(LinkList headp,int i,ElemType e,int flag){  /\*在L的第i个位置前插入新的数据e，长度加1\*/  if(!headp||flag) return INFEASTABLE;  LinkList p=headp;  int j=0;  while(p&&j<i-1) {p=p->next;++j;}  if(!p||j>i-1) return ERROR;  LinkList s=(LinkList)malloc(sizeof(LNode));  s->elem=e;s->next=p->next;  p->next=s;  return OK;  }  status ListDelete(LinkList headp,int i,ElemType\* e,int flag){  /\*删除L的第i个数据，并用e返回其值，L的长度减1\*/  if(!headp||flag) return INFEASTABLE;  LinkList p=headp;  int j=0;  while(p->next&&j<i-1){ //寻找第i个结点，并令p指向其前驱  p=p->next;++j;  }  if(!(p->next)||j>i-1) return ERROR; //删除位置不合理  LinkList q=p->next;  p->next=q->next;  \*e=q->elem;free(q); //释放删除结点  return OK;  }  status ListTrabverse(LinkList headp,int flag){  if(!headp||flag)  return INFEASTABLE;  int i=0;  LinkList p=headp->next;  printf("\n-----------all elements -----------------------\n");  for(;p;p=p->next,i++)  printf("%d ",p->elem);  printf("\n------------------ end ------------------------\n");  return i;  } |

附录C 基于二叉顺序表的二叉树实现的源程序

代码段main.c

|  |
| --- |
| #include"queue.h"  int num=0;  int isempty=0; //isempty用来判断是否被clear，即根节点存在但是内容为空  int main(void){  int count=0; //记录需要创建的二叉树节点的个数，包括空节点  int flag=1;  ElemType defination[100]; //存放二叉树的节点序列  BiTree T[10]; //线性表多棵二叉树管理  BiTree pans=NULL; //注意防止指针悬挂  int op=1,ans=0;  int order=0; //当前操作的二叉树的二叉树数组下标  char c;  ElemType def; //用于读入文件时做空节点用  def.key=0;def.value=0;  while(op)  {  printf("输入你要操作的二叉树的序号：[0-9]\n");  scanf("%d",&order);  while(op)  {  system("cls"); printf("\n\n");  printf(" Menu for Binary Tree On Sequence Structure \n");  printf("-------------------------------------------------\n");  printf(" 1. CreateBiTree 7. Assign\n");  printf(" 2. DestroyBiTree 8. GetSibling\n");  printf(" 3. ClearBiTree 9. InsertNode\n");  printf(" 4. BiTreeEmpty 10. DeleteNode\n");  printf(" 5. BiTreeDepth 11. PreOrderTraverse\n");  printf(" 6. LocateNode 12. InOrderTraverse\n");  printf(" 13. PostOrderTraverse 14. LevelOrderTraverse\n");  printf(" 15. GetParent 16. SaveTree\n");  printf(" 17. LoadTree 0. Exit\n");  printf("-------------------------------------------------\n");  printf("请选择你对二叉树T[%d]的操作[0~12]:",order);  scanf("%d",&op);  switch(op)  {  case 1:  printf("请按照先序序列分别输入二叉树T[%d]节点关键字和需要存储的数据，并以空格或回车键分隔：\n",order);  printf("注意输入的关键字和数据均为整数！\n");  printf("输入关键字为0表示节点为空，且不需要输入数据！\n");  printf("输入关键字为-1结束！\n");  scanf("%d",&((defination[count]).key));  while((defination[count]).key!=-1)//输入节点编号  {  if(defination[count].key>0) //节点的数据域存在则为数据域赋值，否则为系统自动分配的值  {  scanf("%d",&((defination[count]).value));  }  count++;  scanf("%d",&((defination[count]).key));  }  if((ans=CreateBiTree(&T[order],defination,count,&flag,isempty))==1)  printf("二叉树创建成功!\n");  else if(ans==OVERFLOW)  printf("二叉树创建失败！\n");  getchar();getchar();  break;  case 2:  if(!T[order]) printf("二叉树是空树！\n");  else  {  if((ans=DestroyBiTree(&T[order],flag))==INFEASTABLE)  printf("二叉树尚未创建！\n");  else  printf("二叉树销毁成功！\n");  }  getchar();getchar();  break;  case 3:  if(!T[order]) printf("二叉树是空树！\n");  else  {  if((ans=ClearBiTree(T[order],flag))==INFEASTABLE)  printf("二叉树尚未创建！\n");  else  {  printf("二叉树清空成功！\n");  isempty=1;  }  }  getchar();getchar();  break;  case 4:  if(isempty)  printf("二叉树被清空了！\n");  else if((ans=BiTreeEmpty(T[order],flag))==INFEASTABLE)  printf("二叉树尚未创建！\n");  else if(ans==1)  printf("二叉树为空二叉树！\n");  else  printf("二叉树不是空的!\n");  getchar();getchar();  break;  case 5:  if(isempty)  printf("二叉树被清空了！\n");  else if((ans=BiTreeDepth(T[order],flag))>=0)  printf("二叉树的深度为：%d\n",ans);  else  printf("二叉树尚未创建！\n");  getchar();getchar();  break;  case 6:  if(flag) printf("二叉树尚未被创建！\n");  else if(isempty) printf("二叉树被清空了！\n");  else if(!T[order]) printf("二叉树是空树！\n");  else  {  printf("输入查找的节点的关键字：\n");  int e;  scanf("%d",&e);  if((pans=LocateNode(T[order],e)))  visit(pans->elem);  else if(pans==NULL)  printf("找不到关键字为%d的节点！\n",e);  }  getchar();getchar();  break;  case 7:  {  int e;  int val;  printf("输入想要赋值的节点的关键字和值：\n");  scanf("%d",&e);  scanf("%d",&val);  if(flag) printf("二叉树尚未被创建！\n");  else if(isempty)  printf("二叉树被清空了！\n");  else if((ans=Assign(T[order],e,val,flag))==INFEASTABLE)  printf("二叉树尚未创建！\n");  else if(ans==0)  printf("二叉树是空二叉树！\n");  else if(ans==EMPTY)  printf("输入关键字不合法！\n");  else  printf("赋值成功！\n");  getchar();getchar();  break;  }  case 8:  {  if(!T[order]) printf("二叉树是空树！\n");  else if(flag) printf("二叉树尚未被创建！\n");  else  {  int e;  printf("输入查找的节点的关键字：");  scanf("%d",&e);  //无法区分空二叉树和无兄弟节点  if(isempty)  printf("二叉树被清空了！\n");  else if((pans=GetSibling(T[order],e))==NULL)  printf("该节点没有兄弟节点！\n");  else  {  printf("该节点的兄弟节点的");  visit(pans->elem);  }  }  getchar();getchar();  break;  }  case 9:  {  printf("输入插入位置的关键字：");  int e;  scanf("%d",&e);  BiTree c=(BiTree)malloc(sizeof(BiTNode));  printf("分别输入插入的节点的关键字和内容：");  scanf("%d",&c->elem.key);  scanf("%d",&c->elem.value);  c->lchild=NULL;  c->rchild=NULL;  printf("你想将插入的节点作为左孩子还是右孩子？\n");  printf("输入0表示左，输入1表示右\n");  int LR;  scanf("%d",&LR);  if((ans=InsertNode(T[order],e,LR,c,flag,isempty))==INFEASTABLE)  printf("二叉树尚未创建！\n");  else if(ans==ERROR)  printf("二叉树是空二叉树！\n");  else if(ans==OK)  printf("插入成功！\n");  else if(ans==ILLEAGAL)  printf("输入的LR不合法！\n");  else if(ans==EMPTY)  printf("输入关键字不合法！\n");  getchar();getchar();  break;  }  case 10:  {  int e;  printf("输入删除的节点的关键字：");  scanf("%d",&e);  if((ans=DeleteNode(&T[order],e,flag))==INFEASTABLE) //不要把ans的右括号括在infeastable后面，会导致ans赋值为表达式的布尔值  printf("二叉树尚未创建！\n");  else if(ans==ERROR)  printf("二叉树是空树！\n");  else if(ans==OK)  printf("删除成功！\n");  else if(ans==EMPTY)  printf("输入的关键字不合法！\n");  getchar();getchar();  break;  }  case 11:  if(!T[order]) printf("二叉树是空树！\n");  else  {  printf("前序遍历二叉树\n");  if((ans=PreOrderTraverse(T[order],visit,flag,isempty))==OK) ;  else if(ans==EMPTY)  printf("二叉树被清空了！\n");  else if(ans==INFEASTABLE)  printf("二叉树尚未被创建！\n");  }  getchar();getchar();  break;  case 12:  if(!T[order]) printf("二叉树是空树！\n");  else  {  printf("中序遍历二叉树\n");  if((ans=InOrderTraverse(T[order],visit,flag,isempty))==OK) ;  else if(ans==EMPTY)  printf("二叉树被清空了！\n");  else if(ans==INFEASTABLE)  printf("二叉树尚未被创建！\n");  }  getchar();getchar();  break;  case 13:  if(!T[order]) printf("二叉树是空树！\n");  else  {  printf("后序遍历二叉树\n");  if((ans=PostOrderTraverse(T[order],visit,flag,isempty))==OK);  else if(ans==INFEASTABLE)  printf("二叉树尚未被创建!\n");  else if(ans==EMPTY)  printf("二叉树被清空了！\n");  }  getchar();getchar();  break;  case 14:  if(!T[order]) printf("二叉树是空树！\n");  else  {  printf("层序遍历二叉树\n");  if((ans=LevelOrderTraverse(T[order],visit,flag))==OK) ;  else if(ans==INFEASTABLE)  printf("二叉树尚未创建！");  else if(ans==EMPTY)  printf("二叉树被清空了！\n");  }  getchar();getchar();  break;  case 15:  {  if(!T[order]) printf("二叉树是空树！\n");  else if(flag) printf("二叉树尚未被创建！\n");  else  {  printf("输入要查找双亲的节点的关键字：\n");  int e;  scanf("%d",&e);  if(isempty)  printf("二叉树被清空了！\n");  else if((pans=GetParent(T[order],e))==NULL)  printf("找不到双亲节点！\n");  else  {  printf("该节点的双亲节点的");  visit(pans->elem);  }  }  getchar();getchar();  break;  }  case 16:  {  if(isempty)  printf("二叉树被清空了！\n");  else if(flag) printf("二叉树尚未被创建！\n");  else if(!T[order]) printf("二叉树是空树！\n");  else  {  FILE \*fp;char filename[30];  printf("input file name: ");  scanf("%s",filename);  if ((fp=fopen(filename,"wb+"))==NULL)  {  printf("File open error\n ");  exit(-1);  }  else  SaveTree(T[order],def,fp);  fclose(fp);  }  getchar();getchar();  break;  }  case 17:  {  if(flag) printf("二叉树尚未被创建！\n");  else if(!T[order]) printf("二叉树是空树！\n");  else  {  LoadTree(&T[order],defination,count,&flag);  isempty=0;  }  getchar();getchar();  break;  }  case 0:  break;  }//end of switch  }//end of the inner while  printf("继续创建下一个二叉树? [Y/N]（输入大写）\n");  getchar();op=((c=getchar())=='Y')?1:0;  if(op==0)  {  printf("是否返回到之前的二叉树？ [Y/N]（输入大写）\n");  getchar();op=((c=getchar())=='Y')?1:0;  }  isempty=0;  }//end of the outer while  return 0;  }//end of main  status SaveTree(BiTree t,ElemType def,FILE\* fp)  {  //按先序序列的顺序将初始二叉树数据写入文件中  if(t)  fwrite(&(t->elem),sizeof(ElemType),1,fp);  else  fwrite(&(def),sizeof(ElemType),1,fp);  if(t)  {  SaveTree(t->lchild,def,fp);  SaveTree(t->rchild,def,fp);  }  return OK;  }  status LoadTree(BiTree\* t,ElemType\* def,int count,int\* flag)  {  FILE \*fp;char filename[30];  printf("输入需要载入的文件的名称：");  scanf("%s",filename);  if ((fp=fopen(filename,"rb"))==NULL)  {  printf("File open error\n ");  exit(-1);  }  num=0; // 一定要记得对num进行修改，不然无法创建二叉树  int i=0;  while(fread(&def[i],sizeof(ElemType),1,fp)) i++;  \*flag=0;  DestroyBiTree(t,flag);  CreateBiTree(t,def,i,&flag,isempty);  return 0;  }  status CreateBiTree(BiTree\* t,ElemType\* defination,int count,int\* flag,int isempty)  {  if(defination[num].key==0) //关键字为0，意味着是空节点  {  \*t=NULL;  num++;  }  else if(num<count)  {  if(!(\*t=(BiTree)malloc(sizeof(BiTNode))))  exit(OVERFLOW);  (\*t)->elem.key=defination[num].key;  (\*t)->elem.value=defination[num].value;  (\*t)->lchild=(\*t)->rchild=NULL;  num++;  CreateBiTree(&((\*t)->lchild),defination,count,flag,isempty);  CreateBiTree(&((\*t)->rchild),defination,count,flag,isempty);  }  \*flag = 0 ;  isempty=0;  return OK;  }  status DestroyBiTree(BiTree\* t,int flag)  {  //后序递归销毁二叉树  //必须传入双重指针才能销毁成功  if(flag) return INFEASTABLE; //没有创建二叉树就不能销毁  if(\*t)  {  DestroyBiTree(&((\*t)->lchild),flag);  DestroyBiTree(&((\*t)->rchild),flag);  free(\*t);  }\*t=NULL;  isempty=0;  return OK;  }  status ClearBiTree(BiTree t,int flag)  {  //不带头结点，t直接指向根节点，则销毁其他节点只保留根节点并保留其内容  if(flag) return INFEASTABLE;  DestroyBiTree(&(t->lchild),flag);  DestroyBiTree(&(t->rchild),flag);  t->lchild=t->rchild=NULL; //要把根节点的左右孩子指针置为空  return OK;  }  status BiTreeEmpty(BiTree t,int flag)  {  if(flag) return INFEASTABLE;  return t==NULL;  }  status BiTreeDepth(BiTree t,int flag)  {  if(flag) return INFEASTABLE;  if(t==NULL) return ERROR;  if(t->lchild==NULL&&t->rchild==NULL)  return 1;  int ldepth=0;  int rdepth=0;  if(t->lchild)  ldepth=BiTreeDepth(t->lchild,flag);  if(t->rchild)  rdepth=BiTreeDepth(t->rchild,flag);  return (ldepth>rdepth)?ldepth+1:rdepth+1;  }  BiTree LocateNode(BiTree t,int e)  {  //找到关键字与e相同的节点并返回改节点  if(t==NULL) return NULL; //空  if((t->elem).key==e) return t;  BiTree q;  if(q=LocateNode(t->lchild,e)) return q; //传入的节点参数不是要查找的节点就返回空  //当且仅当找到查找的节点才返回非空  //此时就不用继续查找右子树，直接返回，注意不能不带返回  if(q=LocateNode(t->rchild,e)) return q;  }  BiTree GetParent(BiTree t,int e)  {  if(t==NULL) return NULL; //一定要注意这条语句和下面的语句顺序不能调换  //因为传入的t可能为空指针，调用空指针的数据域会出错，所以要先判断是否为空  if(e==t->elem.key) return NULL; //查询的节点为根节点，无双亲  if(t->lchild&&(t->lchild->elem.key==e))  return t;  if(t->rchild&&t->rchild->elem.key==e)  return t;  else  {  BiTree q;  if(q=GetParent(t->lchild,e)) return q;  if(q=GetParent(t->rchild,e)) return q;  }  }  status Assign(BiTree t,int e,int val,int flag)  {  //给关键字为e的节点赋值val  if(flag) return INFEASTABLE;  if(t==NULL) return ERROR;  BiTree p=LocateNode(t,e); //找到节点  if(p==NULL) return EMPTY;  p->elem.value=val; //赋值  return OK;  }  BiTree GetSibling(BiTree t,int e)  {  if(t->elem.key==e) return NULL; //查找的节点为根节点，没有兄弟  BiTree p=NULL,pp=NULL; //p为关键字为e的节点，pp为p的双亲节点  p=LocateNode(t,e);  if(p==NULL) return NULL;  pp=GetParent(t,e);  if(p==pp->lchild) return pp->rchild;  else return pp->lchild;  }  status InsertNode(BiTree t,int e,int LR,BiTree c,int flag,int isempty)  {  if(flag) return INFEASTABLE; //二叉树没有被创建  if(t==NULL) return ERROR; //空二叉树  isempty=0;  BiTree p,q;  p=LocateNode(t,e);  if(p==NULL) return EMPTY; //表示找不到关键字  if(LR==0) //插入节点c作为p的左孩子  {  q=p->lchild;  p->lchild=c;  }  else if(LR==1) //插入节点c作为p的右孩子  {  q=p->rchild;  p->rchild=c;  }  else return ILLEAGAL; //LR输入不合法  c->lchild=NULL;  c->rchild=q;  return OK;  }  status DeleteNode(BiTree\* t,int e,int flag)  {  //要传入双重指针否则对根节点删除之后程序会异常结束运行  if(flag) return INFEASTABLE;  if(\*t==NULL) return ERROR;  BiTree p=NULL,pp=NULL;  if((\*t)->elem.key==e) //删除的是根节点  {  p=\*t;  if((\*t)->lchild==NULL) //根的度为一，且左孩子为空，则右孩子作为根节点  \*t=(\*t)->rchild;  else if((\*t)->rchild==NULL) //根的度为一，且右孩子为空，则左孩子作为根节点  \*t=(\*t)->lchild;  else  { //根的度为二  pp=\*t=(\*t)->lchild; //根的左孩子为新的根节点  while(pp->rchild) pp=pp->rchild; //遍历到待删的根的左孩子的最右节点  pp->rchild=p->rchild; //待删的根的右子树作为其左子树的最右节点的右子树  }  }  else  {  p=LocateNode(\*t,e);  if(p==NULL); return EMPTY;  pp=GetParent(\*t,e);  if(p->lchild==NULL&&p->rchild==NULL) //待删节点p为叶子节点  {  if(pp->lchild==p) pp->lchild=NULL; //p为pp的左孩子，删除后pp的左孩子为空  else  pp->rchild=NULL; //p为pp右孩子  }  else if(pp->lchild==p) // 待删节点不为叶子节点，且为pp的左孩子  {  if(p->lchild==NULL&&p->rchild) //待删节点p度为一，且左孩子为空  pp->lchild=p->rchild; //p的右孩子代替p  else if(p->rchild==NULL&&p->lchild)  pp->lchild=p->lchild;  else if(p->lchild&&p->rchild) //待删节点p的度为二  {  pp->lchild=p->lchild; //p节点的左孩子代替p  BiTree q=pp->lchild;  while(q->rchild) q=q->rchild; //找到p的左子树的最右节点  q->rchild=p->rchild; //p的右子树做其左子树的最右节点的右子树  }  }  else  {  //p为pp的右孩子，同理上述步骤  if(p->lchild==NULL&&p->rchild)  pp->rchild=p->rchild;  else if(p->rchild==NULL&&p->lchild)  pp->rchild=p->lchild;  else if(p->lchild&&p->rchild)  {  pp->rchild=p->lchild;  BiTree q=pp->rchild;  while(q->rchild) q=q->rchild;  q->rchild=p->rchild;  }  }  }  free(p); // 记得释放内存空间  return OK;  }  status InOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty)  {  //中序非递归算法  if(flag) return INFEASTABLE; //二叉树不存在  if(isempty) return EMPTY;  else  {  BiTree st[MAXLENG+1];  int top=0;  do  {  while(t)  {  if(top==MAXLENG) exit(OVERFLOW);  st[++top]=t;  t=t->lchild;  }//先左一直到最左节点  if(top) // 非空栈  {  t=st[top--];  visit((t->elem));  t=t->rchild;  }//end of if  }while(t||top);  return OK;  }  }  status PreOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty)  {  //前序递归算法  if(flag) return INFEASTABLE;  if(isempty) return EMPTY;  else if(t)  {  visit((t->elem));  PreOrderTraverse(t->lchild,visit,flag,isempty);  PreOrderTraverse(t->rchild,visit,flag,isempty);  }return OK;  }  status PostOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty)  {  //后序递归算法  if(flag) return INFEASTABLE;  if(isempty) return EMPTY;  else if(t)  {  PostOrderTraverse(t->lchild,visit,flag,isempty);  PostOrderTraverse(t->rchild,visit,flag,isempty);  visit((t->elem));  }  return OK;  }  status LevelOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag)  {  if(flag) return INFEASTABLE;  if(t==NULL) return ERROR;  if(isempty) return EMPTY;  pLinkQueue p=(pLinkQueue)malloc(sizeof(LinkQueue));  p=InitQueue(p);  if(EnQueue(p,t)==OVERFLOW) exit(OVERFLOW);  else  {  while(p->rear!=p->front) //队列不为空  {  BiTree q=DeQueue(p);  visit(q->elem);  if(q->lchild)  EnQueue(p,q->lchild);  if(q->rchild)  EnQueue(p,q->rchild);  }  }  return OK;  } |

代码段bitree.h

|  |
| --- |
| #include"common.h"  typedef struct BiTNode{  ElemType elem;  struct BiTNode\* lchild;  struct BiTNode\* rchild;  }BiTNode,\*BiTree;  status CreateBiTree(BiTree\* t,ElemType\* defination,int count,int\* flag,int isempty);  status DestroyBiTree(BiTree\* t,int flag);  status ClearBiTree(BiTree t,int flag);  status BiTreeEmpty(BiTree t,int flag);  status BiTreeDepth(BiTree t,int flag);  BiTree LocateNode(BiTree t,int e);  BiTree GetParent(BiTree t,int e);  status Assign(BiTree t,int e,int val,int flag);  BiTree GetSibling(BiTree t,int e);  status InsertNode(BiTree t,int e,int LR,BiTree c,int flag,int isempty);  status DeleteNode(BiTree\* t,int e,int flag);  status PreOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty);  status InOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty);  status PostOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag,int isempty);  status LevelOrderTraverse(BiTree t,void (\*visit)(ElemType node),int flag);  status SaveTree(BiTree t,ElemType def,FILE\* fp);  status LoadTree(BiTree\* t,ElemType\* def,int count,int\* flag); |

代码段common.h

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <malloc.h>  #include <math.h>  /\*书本第10页，预定义常量和类型\*/  #define TRUE 1  #define FALSE 0  #define OK 1  #define ERROR 0  #define INFEASTABLE -1  #define OVERFLOW -2  #define MAXLENG 100  #define ILLEAGAL -3  #define EMPTY 2  typedef int status;  typedef struct Elem{  int key;  int value;  }ElemType;//定义键值，key为节点编号，value为二叉树节点存储的数据  void visit(ElemType node)  {  printf("关键字：%d 内容：%d\n",node.key,node.value);  } |

代码段queue.h

|  |
| --- |
| #include "bitree.h"  //有关链式队列的定义和操作  //只用在遍历上不涉及删除，所以不用设计循环队列  typedef struct QNode{  BiTree T;  struct QNode \*next;  }QNode,\*QueuePtr;  typedef struct{  QueuePtr front; //头指针  QueuePtr rear; //尾指针  }LinkQueue,\*pLinkQueue;  pLinkQueue InitQueue(pLinkQueue q)  {  q->front=q->rear=(QueuePtr)malloc(sizeof(QNode));  if(!q->front) exit(OVERFLOW);  q->front->next=NULL;  return q;  }  status EnQueue(pLinkQueue q,BiTree t)  {  //插入元素t作为q的新的队尾元素  QueuePtr p;  p=(QueuePtr)malloc(sizeof(QNode));  if(!p) exit(OVERFLOW);  p->T=t;p->next=NULL;  q->rear->next=p;  q->rear=p;  return OK;  }  BiTree DeQueue(pLinkQueue q)  {  QueuePtr p=(QueuePtr)malloc(sizeof(QNode));  p=q->front->next;  if(p->next==NULL) //删除后队列中只有一个元素  q->rear=q->front;  else  q->front->next=p->next;  BiTree t=p->T;  free(p);  return t;  } |