華中科技大學

课 程 实 验 报 告

课程名称： 数据结构实验

专业班级 CS2101

学　　号 U202115325

姓　　名 宁毓伟

指导教师 李剑军

报告日期 2022 年 4 月 14 日

计算机科学与技术学院

目　　录

1 基于链式存储结构的线性表实现

所谓基于链式存储结构的线性表其实就是我们常说的链表。当然，再继续细分下去还有链式栈，链式队列这么些分法。下面的实验中主要涉及对于链表的数据类型，数据之间的关系以及基本操作及其实现方式等内容。

1.1 问题描述

线性表在物理内存中可以以链表表的方式实现，即物理上存储位置不一定相邻的两个元素通过指针连结，成为线性表中的相邻元素，且数据元素的前后关系不变。

1.1.1 实验目的

1. 加深对链表的概念、基本运算的理解；
2. 熟练掌握链表的逻辑结构与物理结构的关系；
3. 以链表作为物理结构，熟练掌握链式线性表基本运算的实现。

1.1.2 数据类型

链表元素的数据类型 (ElemType) 应该由链表的使用者进行定义，但为了便于描述链表的功能，笔者将在下文中假定链表的数据元素类型为整型 (int)。

1.1.3 元素关系

链表中的元素之间的关系是前驱和后继的有序关系，在链表的定义过程中将会体现这一点。

1.1.4 基本操作

对于链表而言，最重要的部分是其基本操作，下面是待实现的基本操作:

基本功能（1）初始化表：函数名称是 InitList(L)；初始条件是线性表 L 不存在；操作结果是构造一个空的线性表；

基本功能（2）销毁表：函数名称是 DestroyList(L)；初始条件是线性表 L 已存在；操作结果是销毁线性表 L；

基本功能（3）清空表：函数名称是 ClearList(L)；初始条件是线性表 L 已存在；操作结果是将 L 重置为空表；

基本功能（4）判定空表：函数名称是 ListEmpty(L)；初始条件是线性表 L 已存在；操作结果是若 L 为空表则返回 TRUE, 否则返回 FALSE；

基本功能（5）求表长：函数名称是 ListLength(L)；初始条件是线性表已存在；操作结果是返回 L 中数据元素的个数；

基本功能（6）获得元素：函数名称是 GetElem(L,i,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用 e 返回 L 中第 i 个数据元素的值；

基本功能（7）查找元素：函数名称是 LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回 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；

基本功能（8）获得前驱：函数名称是 PriorElem(L,cur,pre)；初始条件是线性表 L 已存在；操作结果是若 cur 是 L 的数据元素，且不是第一个，则用 pre 返回它的前驱，否则操作失败，pre 无定义；

基本功能（9）获得后继：函数名称是 NextElem(L,cur,next)；初始条件是线性表 L 已存在；操作结果是若 cur 是 L 的数据元素，且不是最后一个，则用 next 返回它的后继，否则操作失败，next 无定义；

基本功能（10）插入元素：函数名称是 ListInsert(L,i,e)；初始条件是线性表 L 已存在，1≤i≤ListLength(L)+1；操作结果是在 L 的第 i 个位置之前插入新的数据元素 e。

基本功能（11）删除元素：函数名称是 ListDelete(L,i,e)；初始条件是线性表 L 已存在且非空，1≤i≤ListLength(L)；操作结果：删除 L 的第 i 个数据元素，用 e 返回其值；

基本功能（12）遍历表：函数名称是 ListTraverse(L)，初始条件是线性表 L 已存在；操作结果是依次对 L 的每个数据元素输出打印到屏幕上。

附加功能（1）最大连续子数组和：函数名称是 MaxSubArray(L); 初始条件是线性表 L 已存在且非空，请找出一个具有最大和的连续子数组（子数组最少包含一个元素），操作结果是其最大和；

附加功能（2）和为 K 的子数组：函数名称是 SubArrayNum(L,k); 初始条件是线性表 L 已存在且非空, 操作结果是该数组中和为 k 的连续子数组的个数；

附加功能（3）顺序表排序：函数名称是 sortList(L)；初始条件是线性表 L 已存在；操作结果是将 L 由小到大排序；

附加功能（4）实现线性表的文件形式保存：其中，需要设计文件数据记录格式，以高效保存线性表数据逻辑结构 (D,R) 的完整信息；需要设计线性表文件保存和加载操作合理模式。

附加功能（5）实现多个线性表管理：设计相应的数据结构管理多个线性表的查找、添加、移除等功能。

1.2 系统设计

本系统中首先定义了一些宏以及数据类型。宏定义用作状态的表示，如“成功状态”：OK ，“错误状态”：ERROR。

系统的线性表采用先进先出的链式存储结构。

系统有一个菜单供用户进行功能选择，用户可以根据提示信息通过键盘输入相应的字符来完成功能的选择。

在多线性表的定义中笔者利用一个数组 (list) 来保存多个线性表的结点，并用结点数组 (listTail) 来保存相应的结点，保存结点的目的是为了在对线性表进行元素追加的时候操作更为简便。在系统中笔者维护一个整型变量 nowIndex 作为当前用户操作的线性表的下标，在用户每次更换操作对象时 nowIndex 的值也随之改变。这样一来就实现了多线性表的操作。

1.3 系统实现

1.3.1 系统环境

```
OS: Manjaro Linux x86_64(操作系统)
Kernel: 5.15.32-1-MANJARO(内核)
CPU: AMD Ryzen 7 5800H with Radeon Graphics (16)
Memory: 13898MiB
gcc version 11.2.0 (GCC)
```

注意，若要在 windows 中运行本程序可以选择在 WSL 中运行或在 window 命令行中调用 g++ 命令将 cpp 文件重新编译生成 windows 下的可执行文件 (.exe)。

1.3.2 宏定义以及数据类型定义

```
#define TRUE 1                  // 状态宏定义
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
typedef int status;
typedef int ElemType;           //数据元素类型定义
#define LIST_INIT_SIZE 100  // 初始大小以及增量的宏定义
#define LISTINCREMENT  10
```

1.3.3 链结点类型

系统的线性表采用先进先出的链式存储结构，为此我们需要对结点类型进行定义：

```
typedef struct LNode{           //单链表（链式结构）结点的定义
    ElemType data;
    struct LNode *next;
}LNode,*LinkList;
```

1.3.4 多线性表的实现方式

多线性表定义如下：

其中的 list 数组用于保存各个链表的结点，listTail 数组用于保存各个链表的结点。保存结点的目的是为了在对线性表进行元素追加的时候操作更为简便。length 表示该结构含有的链表个数，size 表示该结构目前最多可以存放的链表个数。

```
typedef struct {        // 多链表定义!
LinkList* list; LinkList* listTail;
int length; int size;
}Lists;
```

1.3.5 线性表基本操作的实现

下面是线性表基本操作的实现原理，源代码位于附录 B 中。

1. 链表的初始化 InitList(L):

   这个函数的实现比较简单，先对传入的链表 L 进行判断，若 L 不存在，则可以进行初始化，为 L 动态分配一结点的空间作为结点并返回 OK。若 L 存在，初始化会使 L 中元素丢失，故不能将 L 初始化，此时应该返回 INFEASIBLE(不可执行)。

2. 链表的销毁 DestroyList(L):

   首先判断链表是否存在，若不存在则不能够销毁，若链表存在则对结点实施内存释放操作。内存释放过程如下：先用指针 p 指向结点，接着维护一个指针 tmp 使其指向 p 的下一结点，接着将 p 指向的空间释放掉，再让 p 指向 tmp 所指向结点（即原来 p 的下一结点），重复上述操作直到将 L 中的所结点都释放完，最后将 L 设置为 NULL。

3. 清空表 ClearList(L):

   首先判断链表是否存在，如果不存在则不能够清空，返回 INFEASIBLE(不可执行)。若链表存在，则对结点实施内存释放操作。内存释放过程如下：先用指针 p 指向首元结点，接着维护一个指针 tmp 使其指向 p 的下一结点，接着将 p 指向的空间释放掉，再让 p 指向 tmp 所指向结点（即原来 p 的下一结点），重复上述操作直到将 L 中的所结点都释放完，最后将 L 的下一结点 (L->next) 设置为 NULL。

4. 判断空表 ListEmpty(L):

   首先判断链表是否存在，如不存在则返回 INFEASIBLE。若 L 存在，则判断 L 的下一结点 (L->next) 是否为 NULL，是则返回 TRUE，不是则返回 FALSE。

5. 求表长 ListLength(L):

   首先判断链表是否存在，如果不存在则不能够求表长，返回 INFEASIBLE。如果 L 存在，则令指针 p 指向 L 的结点，同时维护一个变量 cnt(count 的缩写)，将其初始化为 0，接着移动指针 p，每移动一次则 cnt 加 1，当到 L 的最后一结点时不再移动，此时得到的 cnt 值就是链表的长度，将其返回即可。

6. 获取元素 GetElem(L,i,e)：

   其中 i 是希望查找的元素的序号（从 1 开始计数），e 是用于返回所找到的

元素值。第一步先判断表是否存在，若不存在，则返回 INFEASIBLE。接着判断 i 的取值是否恰当，如果 i 的取值小于 1 或大于链表的长度，则返回 ERROR 表示下标不存在。接下来就可以用 p 指向结点，让 p 向后移动，每次移动 i 同时减 1，当 i 减到 0 时 p 不再移动，此时 p 指向的便是目结点，将结点的数据域的值赋给 e 则可以返回 OK。

7. 查找元素 LocateElem(L,e)：

   查找元素 e 的下标: 首先判断链表是否存在，如果不存在则不能够查找元素，返回 INFEASIBLE。如果 L 存在，则用指针 p 指向 L 的结点，同时维护一个变量 index，将其初始化为 0，接着移动指针 p，p 每移动一次，index 的值就加 1，当 p 所结点的数据域取值和元素 e 的值相同，则将 index 返回，如果 p 指针移动到了 L 的尾部但仍没有发现元素 e，则返回 ERROR 表示没有找到该元素。

8. 获取前驱 PriorElem(L,e,pre)：

   将 e 元素的前驱元素用 pre 返回: 首先判断链表是否存在，如果不存在则不能够查找前驱元素，返回 INFEASIBLE。如果 L 存在，则用 p 指向 L 的结点的下一结点，向后移动 p，当 p 所结点的后结点的指针域的值与 e 相同，则将 p 所指结点的数据域的值赋给 pre，并且返回 OK。若 p 到达链表的结点还没有找到 e，则返回 ERROR 表示找不到。

9. 获取后继 NextElem(L,e,next)：

   将 e 元素的后继元素用 next 返回：首先判断链表是否存在，如果不存在则不能够查找后继元素，返回 INFEASIBLE。如果 L 存在，则用指针 p 指向 L 的首元结点，移动指针 p，当 p 所结点的后结点的指针域的值与 e 相同时停止移动，此时判断 p 的下一结点是否存在，存在则将其数据域的值赋给 next 并返回 OK，不存在或没有找到元素 e 则返回 ERROR 表示找不到。

10. 插入元素 ListInsert(L,i,e)：

    在位置 i 插入元素 e：首先判断链表是否存在，如果不存在则不能够插入元素，返回 INFEASIBLE。接着判断 i 的取值是否恰当，如果 i 的取值小于 1，则返回 ERROR 表示下标不存在。接着先将 i 减 1，令指针 p 指向结点并且移动 p，每次移动 i 同时减 1，当 i 减到 0 时 p 不再移动，此时先创建一个指针变量 tmp，为该指针变量分配一结点的空间，给 tmp 指向结点的数据域赋值为 e，同时 tm 结点的下一结点设置为 p 的下一结点，而 p 的下一结点则

设置为 tmp，接着让 L 的长度（L.length）加一后返回 OK。若 p 到达链表的结点 i 还没有减到 0，则返回 ERROR 表示下标不存在。

11. 删除元素 ListDelete(L,i,e))：

    在位置 i 删除元素 e：首先判断链表是否存在，如果不存在则不能够删除元素，返回 INFEASIBLE。接着判断 i 的取值是否恰当，如果 i 的取值小于 1，则返回 ERROR 表示下标不存在。接着先将 i 减 1，令指针 p 指向结点并且移动 p，每次移动 i 同时减 1，当 i 减到 0 时 p 不再移动，此时判断 p 的下一结点是否存在，如存在则先用一个临时指针 tmp 指向 p 的下一结点，并且将 p 的下结点设置为 p 的下一结点的下一结点（p->next = p->netx->next），接着将 tm 结点的数据域值赋给 e 并释放 tmp 所指的空间，接着让 L 的长度（L.length）减一后即可返回 OK。若 p 到达链表的结点 i 还没有减到 0，或者 i 为 0 时 p 的下一结点为空，则返回 ERROR 表示下标不存在。

12. 遍历表 ListTraverse(L)：

    按从头到尾的循序输出 L 中的元素：先对传入的链表 L 进行判断，如果 L 不存在则不能够遍历，返回 INFEASIBLE。如果 L 存在，则让指针 p 指向 L 的首元结点，接着移动指针 p，每移动一次则将 p 所结点的数据域的值输出，直到 p 为空则不再移动并返回 OK。

13. 链表反转 reverseList(L)：

    将链表中元素的循序反转：首先判断链表是否存在，如果不存在则不能够反转链表，返回 INFEASIBLE。如果链表的长度为 0 或为 1，则直接返回 OK，因为此时并不需要反转。如果链表存在且长度大于 1，则让指针 p 指向 L 的首元结点，指针 nextOfp 指向 p 的下一结点，接着用一个指针 tmp 记录 nextOfp 的下一结点，将 nextOfp 的下一结点设置为 p(nextOfp->next = p)，接着让将 nextOfp 的值赋给 p，将 tmp 的值赋给 nextOfp，重复上述操作直到将 nextOfp 的值为 NULL 后，将 L 的下一结点设置为 p（L->next = p）后便可以返回 OK。

14. 删除链表中倒数第 n 结点 RemoveNthFromEnd(L, n)：先判断链表是否存在，如果不存在则不能够进行操作，返回 INFEASIBLE。如果存在则创建一个变量 index 并将其初始化为 L.length - n + 1，这就是给定 n 值的从正向数起的元素值，接着调用函数 ListDelete() 将序号为 index 的元素删除即可返回 OK。

15. 链表排序 sortList(L)：

    先判断链表是否存在，如果不存在则不能够进行排序，返回 INFEASIBLE。如果存在则将列表进行升序排序，先创建一个长度和链表长度一样的整型数组，接着将链表元素依次放到该数组中，对该数组进行快速排序，接着将数组中的元素值依次赋给链表中结点，完成后即可返回 OK。

16. 将链表数据保存在文件中 saveList(L, FileName)：

    先判断链表是否存在，如果不存在则不能够进行保存，返回 INFEASIBLE。如果存在则将文件以二进制方打开，让指针 p 指向 L 的首元结点，将 p 所结点的数据域的值保存在文件中后将 p 移动到下一结点，重复上述操作直到 p 的值为 NULL 后即可返回 OK。需要注意的是当打开文件后要判断文件是否打开成功，打开失败则应该返回 ERROR 表示文件未正常打开。

17. 从文件中读取数据创建链表 LoadList(L, FileName)：先判断链表是否存在，如果链表存在且不为空则不能够进行创建，返回 INFEASIBLE。如果链表不存在则为 L 动态分配一结点的空间，若链表存在且为空则不需要进行这一项操作，接着将文件以二进制的形式打开，接着创建一个 tail 指针指向 L，再创建一个临时指针 tmp，为 tmp 动态分配一结点的空间，从文件中读取一个数据到 tmp 中并且将 tail 所指的下一结点设置为 tmp，最后将 tmp 的指针值赋给 tail，重复上述操作直到文件指针移动到 EOF 处。最后将 tail 所指的下一结点设置为 NULL 并返回 OK。需要注意在打开文件时要判断打开是否成功，不成功则返回 ERROR。

1.4 系统测试

1.4.1 菜单功能

下面是菜单的截图：

注意：以下各个函数测试时相互独立，进行各个功能测试初始数据与初始化链表并添加数据时的数据相同。

1.4.2 初始化链表以及添加数据

添加以下链表并初始化，接着填入数据：

list1：(空表)

```
============================MENU================================
0 : Show this menu
1 : Initialize your list                    2 : Destroy your list
3 : Clear your list                         4 : Make sure if your list is empty
5 : Get the length of your lists            6 : Get one particular element
7 : Find the index of a particular element  8 : Get the prior of one element
9 : Get the next of one element            10: Insert element
11: Delete element                         12: Traverse your list
13: Reverse your list                      14: Delete one element from the end of your list
15: Sort your list                         16: Save your list in the file
17: Load your list from a file             18: Add some elements to your list
19: Choose a new list to operate           20: Quit
============================MENU================================
```

图 1-1　菜单

list2：1, 2, 4, 6, 2

list3：-10, 29, 8, 7

填入测试用例的操作以及截图如下：操作：首先在命令行里面输入./LinkList.out 以运行该程序，接着选择功能 19 添加一个名为 listi(i = 1, 2, 3) 的链表，选择功能 1 将其初始化后再选择功能 18 添加元素 (i 为 1 时不添加元素)。

```
============================MENU================================
0 : Show this menu
1 : Initialize your list                    2 : Destroy your list
3 : Clear your list                         4 : Make sure if your list is empty
5 : Get the length of your lists            6 : Get one particular element
7 : Find the index of a particular element  8 : Get the prior of one element
9 : Get the next of one element            10: Insert element
11: Delete element                         12: Traverse your list
13: Reverse your list                      14: Delete one element from the end of your list
15: Sort your list                         16: Save your list in the file
17: Load your list from a file             18: Add some elements to your list
19: Choose a new list to operate           20: Quit
============================MENU================================
```

图 1-2　菜单

1.4.3　销毁表

先销毁 list1(空表)，接着再次销毁 list1。理论结果是第一次销毁成功，第二次销毁失败。实际结果如图所示：

```
============================MENU================================
0 : Show this menu
1 : Initialize your list                    2 : Destroy your list
3 : Clear your list                         4 : Make sure if your list is empty
5 : Get the length of your lists            6 : Get one particular element
7 : Find the index of a particular element  8 : Get the prior of one element
9 : Get the next of one element            10: Insert element
11: Delete element                         12: Traverse your list
13: Reverse your list                      14: Delete one element from the end of your list
15: Sort your list                         16: Save your list in the file
17: Load your list from a file             18: Add some elements to your list
19: Choose a new list to operate           20: Quit
============================MENU================================
```

图 1-3　菜单

jie zhe

1.4.4　清空表

首先对表 list3 进行清空，接着再次选择清空 list3。理论结果是第一次清空成功，第二次清空失败。实际结果如截图所示：

1.4.5　判断空表

选择对 list2 进行判断空表，接着对 list1 进行判断空表，理论结果是得到 list2 不为空，list1 为空的信息。实际结果截图所示：

```
============================MENU================================
0 : Show this menu
1 : Initialize your list                          2 : Destroy your list
3 : Clear your list                               4 : Make sure if your list is empty
5 : Get the length of your lists                  6 : Get one particular element
7 : Find the index of a particular element        8 : Get the prior of one element
9 : Get the next of one element                   10: Insert element
11: Delete element                                12: Traverse your list
13: Reverse your list                             14: Delete one element from the end of your list
15: Sort your list                                16: Save your list in the file
17: Load your list from a file                    18: Add some elements to your list
19: Choose a new list to operate                  20: Quit
============================MENU================================
```

图 1-4　菜单

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-5　获取后继

1.4.6 获取表长

分别对 list1，list2，list3 求表长，理论结果是表长分别为 0, 5, 4。实际结果如图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-6 获取后继

1.4.7 通过下标获取元素

分别获取 list2 中下标为 3 以及下标为 100 的元素，理论结果是得到 4 以及下标不存在的信息。实际结果如图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-7 获取后继

1.4.8 获取元素的下标

获取 list2 中元素 4 的下标以及获取元素 100 的下标，理论结果是得到 4 的下标为 3, 元素 100 不存在。实际结果如图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-8　获取后继

1.4.9　获取前驱

分别获取 list2 中元素 1, 2 的前驱，理论结果是 1 的前驱不存在，2 的前驱为 1。实际结果如图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-9　获取后继

1.4.10　获取后继

分别获取 list3 中元素 8, 7 的后继，理论结果是 8 的后继是 7，7 的后继不存在。实际结果如图所示：

1.4.11　插入元素

向 list3 中下标为 2 的位置插入元素 10 并接着向 list3 中下标为 100 的位置插入元素 100, 接着对表进行遍历，理论结果是第一次插入成功，第二次插入失败，并且遍历将得到 {-10，10，29，8, 7}。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-10　获取后继


```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```


图 1-11　获取后继

1.4.12 删除元素

向 list3 中下标为 2 的位置删除元素 10 并接着向 list3 中下标为 100 的位置删除元素 100, 接着对表进行遍历，理论结果是第一次删除成功，第二次删除失败，并且遍历将得到 {-10，29，8, 7}。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 2 3 4 5 0
OK!
16
input the filename where you want to save the list!
a
OK!
2
OK!
12
this list does not exist!
17
input the filename where you want to load the list!
a
OK!
12
1 2 3 4 5 OK!
```

图 1-12 将链表保存在文件中以及从文件中读取链表

1.4.13 遍历表

分别对 list1, list2, list3 进行遍历，理论结果是分别得到 list1 为空，list2 = {1, 2, 4, 6, 2}，list3 = {-10, 29, 8, 7}。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-13 获取后继

1.4.14 反转链表

对 list2 进行反转后遍历，理论结果得到 {2, 6, 4, 2, 1}。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-14 获取后继

1.4.15 从尾部删除元素

删除 list2 尾部第一个元素后进行遍历，理论结果是得到 {1, 2, 4, 6}。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-15 获取后继

1.4.16 链表排序

对 list2 排序后进行遍历，理论结果是得到 {1, 2, 2, 4, 6}。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-16　获取后继

1.4.17　将链表保存在文件中后从文件中读取链表

将 list2 保存在文件 list 中，接着从 list 中读取链表到 list1 中，对 list1 进行遍历，理论结果是得到 {1, 2, 4, 6, 2}。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-17　获取后继

1.4.18　查找和删除链表

查找名为 list2 的链表并将其删除，接着查找 list2，理论结果是第一次得到 list2 的序号为 2，第二次查找失败。实际结果如下图所示：

```
1
OK!
18
input a list ended with 0!
1 6 4 9 -7 -8 -5 -2 0
OK!
9
input the element whose next element you want to find in the list!
1
you get its next one : 6!
9
input the element whose next element you want to find in the list!
-2
this next does not exist in the list!
```

图 1-18　获取后继

1.5　实验小结

1.5.1　实验中遇到的问题及其解决方案

第一次写链表时没有加入结点，导致自己无法分清楚清空链表和销毁链表的区别，经查阅资料后发现清空链表保留结点，而销毁链表不保留结点。解决方案：为每个链表加入一个结点后问题得到解决。

最开始时求表长功能返回的结果总是比正确结果要多 1，笔者仔细查看代码后发现自己将表结点进行了一次计数，于是我将用于计数的指针 p 先指向首元结点后将问题解决。

由于笔者一开始没有安排插入一串元素的功能而只能一个个元素插入导致在进行测试时十分麻烦，于是我便为这个系统添加了一次性插入一串元素的功能让系统更加人性化。

1.5.2　一些感悟

基于链式存储结构的线性表实现的实验任务让我在学习完链表之后对链表的基本操作以及底层实现原理都更加的熟悉了，实验过程中对于指针的移动操作让我对 c 语言的指针数据类型更加熟悉，在本次实验中我还增强了自己 debug 的能力，对于断点调试等调试方法更加的熟练。同时，我还对 LaTeX 这一排版工具进行了深入的了解，这些知识都将使我受益终身！

2 基于二叉链表的二叉树实现

二叉树（Binary tree）是树形结构的一个重要类型。许多实际问题抽象出来的数据结构往往是二叉树形式，即使是一般的树也能简单地转换为二叉树，而且二叉树的存储结构及其算法都较为简单，因此二叉树显得特别重要。

2.1 问题描述

2.1.1 实验目的

1. 加深对二叉树的概念、基本运算的理解；
2. 熟练掌握二叉树的逻辑结构与物理结构的关系；
3. 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。

2.1.2 数据类型

二叉树元素的数据类型是一个结构体类型，其中包含的三个部分分别是数据域，左孩子指针域以及右孩子指针域。其中数据域的类型 (ElemType) 应该由二叉树的使用者进行定义，但为了便于描述二叉树的功能，笔者将在下文中假定二叉树的数据域数据元素类型为整型 (int)。

2.1.3 元素关系

根据二叉结点类型的定义，每结点（称为结点）都有两个孩结点（孩结点可以为空），且孩结点有左孩结点以及有孩结点之分，故元素之间的关系可以有父子关系，兄弟关系。

2.1.4 基本操作

对于二叉树而言，最重要的部分就是二叉树的基本操作，下面是待实现的基本操作。

基本功能（1）创建二叉树：函数名称是 CreateBiTree(T,definition)；初始条件是 definition 给出二叉树 T 的定义，根据带空子树的二叉树先序遍历序列 definition 构造一个二叉链表 T（要求二叉树 T 中各结点关键字具有唯一性）。根

指针指向根结点，不需要在根结点之上再增加头结点。输入序列为二叉树带空子树的先序遍历结点序列，每个结点对应一个整型的关键字和一个字符串，当关键字为 0 时，表示空子树，为-1 表示输入结束

基本功能（2）销毁二叉树：函数名称是 DestroyBiTree(forest, int index)；将森林 forest 中下标为 index 的二叉树销毁；

基本功能（3）清空二叉树：函数名称是 ClearBiTree (T)；初始条件是二叉树 T 存在；操作结果是将二叉树 T 清空；

基本功能（4）判定空二叉树：函数名称是 BiTreeEmpty(forest,index)；判断森林中下标为 index 的树是否为空，操作结果是若为空二叉树则返回 TRUE，否则返回 FALSE；

基本功能（5）求二叉树深度：函数名称是 BiTreeDepth(T)；初始条件是二叉树 T 存在；操作结果是返回 T 的深度；

基本功能（6）查找结点：函数名称是 LocateNode(T,e)；初始条件是二叉树 T 已存在，e 是和 T 中结点关键字类型相同的给定值；操作结果是返回查找到的结点指针，如无关键字为 e 的结点，返回 NULL；

基本功能（7）结点赋值：函数名称是 Assign(T,e,value)；初始条件是二叉树 T 已存在，e 是和 T 中结点关键字类型相同的给定值；操作结果是关键字为 e 的结点赋值为 value；

基本功能（8）获得兄弟结点：函数名称是 GetSibling(T,e)；初始条件是二叉树 T 存在，e 是和 T 中结点关键字类型相同的给定值；操作结果是返回关键字为 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；

基本功能（9）插入结点：函数名称是 InsertNode(T,e,LR,c)；初始条件是二叉树 T 存在，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；操作结果是根据 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；特殊情况，c 插入作为根结点？可以考虑 LR 为-1 时，作为根结点插入，原根结点作为 c 的右子树。

基本功能（10）删除结点：函数名称是 DeleteNode(T,e)；初始条件是二叉树 T 存在，e 是和 T 中结点关键字类型相同的给定值。操作结果是删除 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；

基本功能（11）前序遍历：函数名称是 PreOrderTraverse(T,Visit())；初始条件是二叉树 T 存在，Visit 是一个函数指针的形参（可使用该函数对结点操作）；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。注：前序、中序和后序三种遍历算法，要求至少一个用非递归算法实现。

基本功能（12）中序遍历：函数名称是 InOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是一个函数指针的形参（可使用该函数对结点操作）；操作结果是中序遍历 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；

基本功能（13）后序遍历：函数名称是 PostOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是一个函数指针的形参（可使用该函数对结点操作）；操作结果是后序遍历 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。

基本功能（14）按层遍历：函数名称是 LevelOrderTraverse(T,Visit))；初始条件是二叉树 T 存在，Visit 是对结点操作的应用函数；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。

基本功能（15）addBinaryTree(forest, BiTreeName, definition); 初始条件是森林 forest 存在，根据 definition 创建名为 BiTreeName 的二叉树加入到森林中。

附加功能（1）最大路径和：函数名称是 MaxPathSum(T)，初始条件是二叉树 T 存在；操作结果是返回结点到叶子结点的最大路径和；

附加功能（2）最近公共祖先：函数名称是 LowestCommonAncestor(T,e1,e2)；初始条件是二叉树 T 存在；操作结果是该二叉树中 e 结点和 e 结点的最近公共祖先；

附加功能（3）翻转二叉树：函数名称是 InvertTree(T)，初始条件是线性表 L 已存在；操作结果是将 T 翻转，使其所结点的左结点互换；

附加功能（4）实现线性表的文件形式保存：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 (D,R) 的完整信息；□ 需要设计二叉树

文件保存和加载操作合理模式。附录 B 提供了文件存取的方法；

附加功能（5）实现多个二叉树管理：可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。

附加功能（6）由于要求输入序列为二叉结点在满二叉树中的序号及结点数据，故需要增加函数 getDefinition(input, definition) 来根据输入序列 input 构造先序遍历序列 definition。

2.2 系统设计

2.2.1 总体设计

本系统采用二叉链表作为二叉树的物理结构，实现二叉树的基本运算。

系统具有一个功能菜单。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现实，并给出适当的操作提示显示。

系统利用一个自定义的结构类型 forest 来管理森林。其中的属性有 elem、length 以及 size。elem 是一个结构数组，其每个元素具有 name 以及 root 两个属性，name 用作保存二叉树的名称，root 用作保存二叉树的结点。length 记录森林中二叉树的个数，size 记录森林可以容纳的二叉树数量。

系统中利用自定义的结构类型 unit 作为存储二叉树时的存储单元，其中的属性有 index 以及 data，。index 记录结点在满二叉树中编号，data 记结点的数据。

2.3 系统实现

```
OS: Manjaro Linux x86_64(操作系统)
Kernel: 5.15.32-1-MANJARO(内核)
CPU: AMD Ryzen 7 5800H with Radeon Graphics (16)
Memory: 13898MiB
gcc version 11.2.0 (GCC)
```

注意，若要在 windows 中运行本程序可以选择在 WSL 中运行或在 window 命令行中调用 g++ 命令将 cpp 文件重新编译生成 windows 下的可执行文件 (.exe)。

2.3.1 有关常量和类型定义

```
#define TRUE 1              // 宏定义
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2
typedef int status;
typedef int KeyType;
```

2.3.2 二叉结点类型

```
typedef struct {    //二叉树结点类型定义
    KeyType  key;
    char others[20];
} TElemType;
typedef struct BiTNode{    //二叉链表结点的定义
    TElemType  data;
struct BiTNode *lchild,*rchild;
}BiTNode, *BiTree;
```

2.3.3 森林结构类型

```
typedef struct{          // 森林定义
    struct{
        char name[20];
        BiTree root;
    }elem[50];
    int length;
    int size;
}forest;
```

2.3.4 存储单元类型

```
typedef struct {            // 存储单元定义
    int index;
    TElemType data;
}unit;
```

2.3.5 队列的简单实现

由于在二叉树的层序遍历中需要用到队列，在此给出队列的简单实现。

```
typedef struct{
    BiTree data[1000];
    int front;  // 队列头
    int back;   // 队列尾
}queue;
BiTree getFront(queue& q){   // 取得队首元素
    return q.data[q.front];
}
BiTree getBack(queue& q){    // 取得队尾元素
    return q.data[q.back-1];
}
bool isEmpty(queue& q){      // 判断空
    return q.front == q.back;
}
void pop(queue& q){         // 弹出队首元素
    q.front ++;
}
void push(queue& q, BiTree elem){   // 从队尾加入元素
    q.data[q.back] = elem;
    q.back ++;
}
int getSize(queue& q){       // 获取队列长度
```

```
    return - q.front + q.back;
}
```

2.3.6 二叉树的基本操作实现

下面是二叉树基本操作的实现原理，源代码位于附录 C 中。

1. 创建二叉树 CreateBiTree(T,definition)：
   由于所给 definition 是先序遍历的序列，自然而然的让人想到通过递归的方法创建二叉树。添加一个辅助函数 create(definition) 以完成递归操作，该函数返回一个创建好的二叉树，同时还需要维护一个全局变量 i 作为读取 definition 数组的下标。每次调用构造函数时先将 i 重置为 0, 接着调用 create() 函数。在 create 函数里先判断 definition[i](即当前读取到结点值) 是否为 0 或者-1, 若是，说明此处应为结点，故直接返回 NULL；若 definition[i] 不是 0 也不是-1, 则利用读取到结点数据创建一个新结点，接着递归创建结点的左子树，再创建结点的右子树，最后将结点返回即可。注意，在进行二叉树构造前应该保结点关键字唯一，这一步的检查在将 definition 作为参数传入之前已经完成，详见函数 getDefinition 的实现，用户仅需输入一系列结点以结点对应的满二叉树中的序号即可。
2. 销毁二叉树 DestroyBiTree(forest, int index)：
   首先调用清空二叉树的函数 ClearBiTree()(详见清空二叉树的实现) 将该二叉树清空，接着将 forest 中的 elem 数组位于 index 之后的元素向前移动一位，最后将 forest 中的 length 值减 1 即可。注意，在进行销毁操作时应保证下标 index 是有效的，对 index 的检查在主函数中已经完成。在主函数中，用户只需要输入二叉树的名称，接着系统便会调用 getBiTree() 函数来查找相应的 index。
3. 清空二叉树 ClearBiTree(T)：
   该函数写成递归的形式比较方便。首先判断结点 T 是否为空，若是则直接返回，若不是，则递归调用该函数清空 T 的左子树，接着递归调用该函数清空 T 的右子树，最后将 T 给指向的内存空间给释放掉即可。
4. 判定空二叉树 BiTreeEmpty(forest,index)：
   此函数的实现十分简单，只需要判断下标为 index 的树的结点是否为空即可，若是则返回 OK, 否者返回 false。注意，在进行判空操作时应保证下标

index 是有效的，对 index 的检查在主函数中已经完成。在主函数中，用户只需要输入二叉树的名称，接着系统便会调用 getBiTree() 函数来查找相应的 index。

5. 求二叉树深度 BiTreeDepth(T)：

   首先判断二叉树结点是否为空，若是，则返回 0，否则创建一个队列以保存二叉树结点，用于层序遍历。队列的实现采用简单的一维数组实现，用 front 变量记录队列头的下标，back 变量记录队列尾部后一个位置的下标，当 front 与 back 相同时队列为空。首先将结点加入到队列中，用变量 depth 记录深度，先将 depth 初始化为 0，接着进入循环，保证每次循环开始前队列中的元素都来自同一层。在每次循环中，用 back-front 获取队列中元素的个数，接着依次将队列头中保存结点的左右孩子 (如果孩子不为空) 加入到队列中后将队结点弹出队列。每进行一次循环后 depth 的值增加 1，当队列内为空时退出循环，此时得到的 depth 就是二叉树的深度。

6. 查找结点 LocateNode(T,e)：

   该函数的实现主要靠遍历二叉树来完成，在具体实现中笔者采用 Morris 先序遍历的方法 (详见二叉树的先序遍历的实现)。首先创建一个指针 ans(初始化为 NULL) 用于记录所查找的结点，以及一个布尔类型的变量 flag(初始化为 false) 记录查找是否成功，在进行先序遍历的过程中比较遍历到的每个结点的关键字与 e 的值是否相同，若是，则将 flag 的值设为 true，并将该结点的地址保存在 ans 中。遍历结束后根据 flag 的值是否为真来选择返回 ans 还是返回 NULL。

7. 结点赋值 Assign(T,e,value)：

   该函数依旧使用 Morris 先序遍历的方法实现，用一个布尔类型的变量 flag(初始化为 false) 记录是否赋值成功，用一个一维数组 hs 作为哈希表来记录每个结点的关键字出现的次数。在进行先序遍历的过程中将遍历到的结点在 hs 表中的值加 1，如果 hs 表中出现元素的值为 2，说明某关键字出现了两次，此时应直接返回 ERROR 报告错误。在进行先序遍历的过程中比较遍历到的每个结点的关键字与 e 的值是否相同，如果相同则将 e 结点赋值为 value，并将 false 赋值为 true。遍历结束后根据 flag 的值是否为真来选择返回 OK 还是返回 ERROR。

8. 获得兄弟结点 GetSibling(T,e)：

该函数依旧使用 Morris 先序遍历的方法实现，在进行先序遍历的过程中比较每一个结点的左右孩子 (如果孩子不为空) 的关键字与 e 是否相同，若相同则将结点的另一个孩子的地址返回。若遍历完成后仍未发现 e 的兄弟结点，则返回 NULL。

9. 插入结点 InsertNode(T,e,LR,c)：

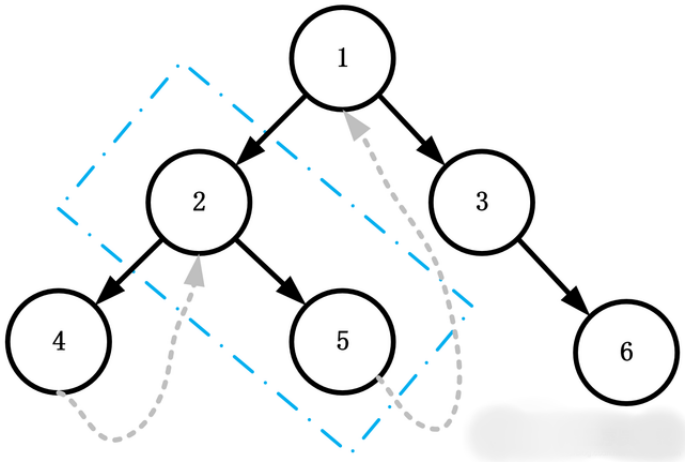
   先利用结点数据 c 创建一个新的结点，接着判断 LR 是否为-1, 若是，则直接将新结点的右孩子赋值为 T 后返回 OK。若 LR 不是-1, 这同样的创建一个布尔类型的变量 flag(初始化为 false)，以及一个一维数组 hs 用作哈希表。在进行先序遍历的过程中将遍历到的结点在 hs 表中的值加 1，如果 hs 表中出现元素的值为 2，则直接返回 ERROR。在进行先序遍历的过程中比较遍历到的每个结点的关键字与 e 的值是否相同，如果相同，则根据 LR 为 0 或者 1，将插入结点 c 作为关键字为 e 的结点的左或右孩子结点，结点 e 的原有左子树或右子树则为结点 c 的右子树，同时将 flag 赋值为 true, 遍历结束后根据 flag 的值是否为真来选择返回 OK 还是返回 ERROR。

10. 删除结点 DeleteNode(T,e)：

    程序一开始先进行特别判断，判断待删除结点 e 是否为结点，若是，进行删除之后需要给结点重新赋值。若待删结点并不是结点，此时则应该进行一次先序遍历来寻找待删除结点 e, 先序遍历依旧采用 Morris 遍历的办法 (故技重施，详见二叉树的先序遍历的实现)。接下来笔者将根据待删结点 e 的度来讲述删除的过程。首先，若待删结点 e 的度为 0, 此时我们仅需要结点 e 用 free 函数释放掉并将其父结点相应的孩子结点设置为空 (NULL) 即可；其次，若待删除节点 e 的度为 1, 我们仅需要将 e 的父结点相应的孩子节点赋值为 e 的孩子结点并将 e 结点释放掉即可；最后，若待删除节点 e 的度为 2, 我们用一个指针 find 指向 e 的左孩子节点，接着让 find 沿着右孩子结点移动找到 e 节点中序遍历下的前驱节点，此时将 e 的右结点作为 find 指向的结点的右子树，接着将 e 的父结点相应的孩子节点设置为 e 的左孩子结点后将 e 结点释放掉即可。注意，本程序不考虑用的 delete 释放结点空间，因为在 educoder 平台上给出的主函数采用了 malloc 函数动态分配内存，相应的应该使用 free 函数释放内存。

11. 前序遍历 PreOrderTraverse(T,Visit())：

    前序遍历的顺序是 (根–> 左–> 右)。前序遍历采用的是 Morris 遍历法，以实



现线性级别的时间复杂度和常数级别的空间复杂度。该方法的核心要点在于维护一个指针 p 指向当前节点 (now) 的中序遍历下的前驱结点 (如果该前驱结点存在)。所谓中序遍历下的前驱结点其实就是当前结点 (now) 左子树最右边的结点。找到该前驱结点后我们将前驱结点 p 的右孩子设置为当前结点 (now)，如此以来我们便能够通过 p 结点回到 now 结点，你可能想问为什么要再次回到 now 结点，这是因为在遍历完 now 的左子树之后我们紧接着需要遍历 now 的右子树，因此我们需要从前驱结点 p 回到 now 之后才能通过 now 进入其右子树 (如图中当前结点 now 为 1 时对应的前驱结点 p 为 5，我们在遍历完 1 的左子树后要从结点 5 返回 1 接着遍历 1 的右子树)。既然我们会重复地访问 now 结点，那么我们如何判断 now 结点是否是第一次访问呢? 没错，答案是根据先序遍历下前驱结点 p 的右孩子是否连接着 now 结点，如果是，则说明当前结点 now 是第二次被访问，并且 now 的左子树已经访问完成 (因为只有在 now 的左子树访问完成后才能通过相应的中序遍历下的前驱结点返回 now)，此时我们应该毫不犹豫的向 now 的右子树前进，因为前序遍历的顺序下遍历完左子树后紧接着就应该遍历右子树；反之，若 now 结点中序遍历下的前驱结点 p 的右孩子没有指向 now, 则说明当前结点 now 是第一次被访问，并且此时 now 的左子树也没有被访问过，因此，我们应该对 now 结点调用 visit() 函数进行访问，接着应该向 now 的左子树前。特别的，若当前结点的左孩子为空，也能够说明当前结点 now 是第一次被访问，此时我们应该对 now 调用 visit 函数进行访问后向 now 的右子树前进。注意，循环开始时 now 结点为根结点，当 now 结点为空时循环结束。

12. 后序遍历 PostOrderTraverse(T,Visit))：

    后序遍历的顺序是 (左–> 右–> 根)。后序遍历采用的是简单的递归实现法，首先调用函数自身递归遍历其左子树，接着递归调用函数自身遍历其右子树，最后对当前结点调用 visit() 函数进行访问。特别注意，递归结束的条件是当前结点为空，因此需要在函数的入口处对当前结点是否为空进行判断，若为空，则应该使用 return 语句结束递归。

13. 按层遍历 LevelOrderTraverse(T,Visit))：

    层序遍历的顺序是一层一层的遍历，即从左到右遍历完当前层后进入下一层。采用队列的方式进行层序遍历。我们将二叉树结点按层依次加入到队列中，并且保证在每一次循环开始时队列中的元素都来自同一层，接着对队列求长度，得到的长度就是当前层的结点个数。接着依次弹出这些结点，对于每一个结点，若该结点有左子树或右子树则将其加入到对立中，作为下一层的结点。开始循环之前我们将根结点加入到队列中作为第一层，当队列内为空时退出循环。

14. 增加二叉树 addBinaryTree(forest, name, definition)：

    首先判断森林是否已经满了，如果是，则直接返回 ERROR。如果森林中还有空间，则声明一个根结点 T，调用 CreateBiTree() 函数进行二叉树的创建，接着在森林中创建一个名字为 name 的二叉树并将其根结点赋值为 T, 最后让森林的大小增加 1 即可。

15. 最大路径和 MaxPathSum(T)：

    该函数的实现需要借助深度优先搜索的方式实现，因此需要定义一个辅助函数 dfs(T, ans, nowSum) 来实现，其中 ans 是用于返回结果的变量，应该初始化为一个很小的数 (-0x3f3f3f3f)，nowSum 是当前得到的路径和。当 dfs 函数搜索到空结点时说明当前路径的搜索已经结束，若 nowSum 的值大于 ans 的值，则说明找到了一条新的路径使路径和更大，此时将 ans 的值更新为 nowSum 即可。在 MaxPathSum() 函数中我们先判断 T 是否为空，若是，则直接返回 INFEASIBLE 表示不可执行，若不为空，则分别对其左右子树调用 dfs 函数来，调用完成后即可将 ans 返回。

16. 最近公共祖先 LowestCommonAncestor(T,e1,e2)：

    该函数的实现采用递归和回溯的方式实现，因此需要用到一个递归辅助函数 isAncestor(T, a, b, ans) 来判断 T 是否是 a 或者 b 的祖先 (是则返回 true，否

则返回 false), 特别的, 如果 T 是 a 或者 b, 也返回 true。在递归函数 isAncestor 中首先判断当前结点 T 是否为空，若是，则直接返回 false；若不是，则分别对 T 的左右子树调用 isAncestor() 函数判断 T 的左右子树是否为 a 或者 b 的祖先，若 T 的左右子树都是 a 或则 b 的祖先，或者其中一个子树为 a 或 b 的祖先且 T 为 a 或 b, 则说明 T 一定就是 a 与 b 的最小公共祖先，此时将 ans 赋值为 T 即可。在 LowestCommonAncestor() 函数中首先判断根结点是否为空，若为空，则直接返回 NULL；若不为空则将 ans 初始化为 NULL，接着对根结点调用一次 isAncestor() 函数即可得到 ans，没有找到相应的结点，则 ans 将保留其初始值 NULL 不变。

17. 翻转二叉树 InvertTree(T)：

    该函数的实现相对简单。需用用到一个递归辅助函数 dfsInvert(T) 来进行递归翻转。那为什么笔者不直接将 InvertTree() 写成递归的形式呢？这是因为在该函数中当根结点 T 为空时要返回 INFEASIBLE(不可执行)，而在递归中结束的标志是当前结点为空，递归结束时不需要返回任何值，故应该增加一个返回值为 void 的辅助函数完成递归。在 dfsInvert() 函数中笔者采用自底向上的翻转方式，首先递归的翻转当前结点 T 的左右子树，接则将 T 的左子树和右子树进行调换即可。在 InvertTree() 函数中我们先判断 T 是否为空，若为空，则返回 INFEASIBLE，若不为空，则调用 dfsInvert() 函数进行翻转。

18. 保存二叉树 SaveBiTree(T, fileName)：

    该函数仍需要一个辅助函数 save(index, T, fp) 来实现，其中 index 是当前结点 T 在满二叉树中的编号，fp 是要写入的文件的指针 (filePointor)。在 save 函数里，首先将当前结点的数据保存在一个 unit 类的对象 p 中，接着将对象 p 写入到文件中，再递归的保存其左子树和右子树，同时根据结点在满二叉树中的编号，index 在进入左子树和右子树时分别变为 index*2 以及 index*2+1。在 SaveBiTree() 函数中，首先判断根结点 T 是否为空，若是则直接返回 OK 表示空树不用存储，接着将文件指针初始化后调用 save() 函数进行保存，保存完成后将文件关闭即可。注意，在打开文件指针时应该判断打开是否成功，不成功则应该直接返回 ERROR。

19. 从文件中加载二叉树 LoadBiTree(T, FileName)：

    该函数首先判断 T 是否为空，不为空则不能够将二叉树加载到其中，接着将

文件指针初始化，将文件中的结点数据依次读出来保存在一个临时指针数组 p 中，以数组 p 的下标代表二叉树结点在满二叉树中的编号，根据父子编号的关系 (父结点编号为 n, 左孩子编号为 n*2, 右孩子编号为 n*2+1)，在相应的结点之间建立父子关系。在读取数据完成后将 T 赋值为 p[1](根结点的编号为 1) 即可返回。注意，在打开文件指针时应该判断打开是否成功，若不成功则应该直接返回 ERROR。

2.4 系统测试

主要说明针对各个函数正常和异常的测试用例及测试结果画图说明网页的整体框架，进行简要的文字描述等。画图说明网页的整体框架，进行简要的文字描述等。画图说明网页的整体框架，进行简要的文字描述等。画图说明网页的整体框架，进行简要的文字描述等。画图说明网页的整体框架，进行简要的文字描述等。画图说明网页的整体框架，进行简要的文字描述等。画图说明网页的整体框架，进行简要的文字描述等。

2.5 实验小结

3 课程的收获和建议

描述通过学习该专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。描述通过学习该专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。描述通过学习该专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。描述通过学习该专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。

3.1 基于顺序存储结构的线性表实现

描述通过学习计算机基础知识专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。描述网页的设计和实现过程中遇到的问题及如何解决。描述网页的设计和实现过程中遇到的问题及如何解决。描述网页的设计和实现过程中遇到的问题及如何解决。描述网页的设计和实现过程中遇到的问题及如何解决。描述网页的设计和实现过程中遇到的问题及如何解决。描述网页的设计和实现过程中遇到的问题及如何解决。描述网页的设计和实现过程中遇到的问题及如何解决。描述网页的设计和实现过程中遇到的问题及如何解决。

3.2 基于链式存储结构的线性表实现

描述通过学习文档撰写工具 LaTeX 专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。描述通过学习文档撰写工具 LaTeX 专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。

3.3 基于二叉链表的二叉树实现

描述通过学习编程工具 Python 专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。描述通过学习编程工具 Python 专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。

3.4 基于二叉链表的二叉树实现

描述通过学习计算机基础知识专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。描述通过学习计算机基础知识专题，有何收获，有何建议，如某专题可适当减少讲授时间、某专题可适当增加讲授内容和时间等。

附录 A 基于顺序存储结构线性表实现的源程序

```
/* Linear Table On Sequence Structure */
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

/*———page 10 on textbook ———*/
#define TRUE 1
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASTABLE -1
#define OVERFLOW -2
```

附录 B 基于链式存储结构线性表实现的源程序

Definition.h

```
# ifndef DEFINITION_H_          // 防止头文件重复引入
# define DEFINITION_H_

#define TRUE 1                  // 状态宏定义
#define FALSE 0
#define OK 1
#define ERROR 0
#define INFEASIBLE -1
#define OVERFLOW -2

#include <stdio.h>

typedef int  status ;
typedef int  ElemType;          //数据元素类型定义

#define LIST_INIT_SIZE 100      // 初始大小以及增量的宏定义
#define LISTINCREMENT 10

typedef struct LNode{           //单链表（链式结构）结点的定义
          ElemType data;
          struct  LNode *next;
        }LNode,*LinkList;              //单链表（链式结构）结点的定义

# endif
```

Main.cpp

```
# include <stdio.h>
# include "Definition.h"
```

```
3  # include "Init.h"
4  # include "Destroy.h"
5  # include "LocateElem.h"
6  # include "NextElem.h"
7  # include "PriorElem.h"
8  # include "ListInsert.h"
9  # include "ListTraverse.h"
10 # include "reverseList.h"
11 # include "RemoveNthFromEnd.h"
12 # include "sortList.h"
13 # include "saveandload.h"
14 # include "Clear.h"
15 # include "ListEmpty.h"
16 # include "GetElem.h"
17 # include "Lists.h"
18 # include "ListDelete.h"
19 # include "ListLength.h"
20
21 void showMenu(){
22  printf ("=========================================MENU
       ==================================================\n"
23 "0 : Show this menu\n"
24 "1 : Initialize your list                2 : Destroy your list\n"
25 "3 : Clear your list                    4 : Make sure if your list
       is empty\n"
26 "5 : Get the length of your lists       6 : Get one particular
       element\n"
27 "7 : Find the index of a particular element 8 : Get the prior of
       one element\n"
28 "9 : Get the next of one element        10: Insert element\n"
29 "11: Delete element                     12: Traverse your list\n"
```

```
30 "13: Reverse your list                    14: Delete one element from
       the end of your list\n"
31 "15: Sort your list                       16: Save your list in the
      file\n"
32 "17: Load your list from a file           18: Add some elements to
      your list\n"
33 "19: Choose a new list to operate         20: Quit\n"
34 "========================================MENU
      ================================================\n"
35 "\n");
36 }
37
38 int main(){
39 int tmp;
40
41 int choice, e, i, index, pre, next;
42 Lists  lists ;
43 // LinkList L = NULL;
44 int nowIndex = 0;
45  initLists ( lists );
46
47  printf ("choose a list from the lists! input an integer ranging from
      1 to 20 : \n");
48 scanf("%d", &nowIndex);
49  printf ("you chose list %d!\n", nowIndex);
50 putchar('\n');
51
52  LinkList  tmpl = NULL, tailOftmpl;
53  status  sta ;
54 char f[100];
55
```

```
56
57 showMenu();
58  printf ("MAKE YOUR CHOICE!\n");
59 while(true){
60 scanf("%d", &choice);
61 switch(choice){
62 case 0:
63          showMenu();
64          break;
65 case 1:
66          sta = InitList ( lists . list [nowIndex]);
67           lists . listTail [nowIndex] = lists . list [nowIndex];
68          if (sta == OK){
69                  printf ("OK!\n");
70          }else{
71                  printf ("this list can not be initialized!\n");
72          }
73          break;
74 case 2:
75 if ( lists . list [nowIndex] == NULL){
76                  printf ("this list hasn't been initialized!\n");
77                 break;
78          }
79          sta = DestroyList( lists . list [nowIndex]);
80          if (sta == OK){
81                  printf ("OK!\n");
82          }else{
83                  printf ("this list can not be Destroyed!\n");
84          }
85          break;
86 case 3:
```

```
87          if ( lists . list [nowIndex] == NULL){
88                  printf ("this list hasn't been initialized!\n");
89                  break;
90          }
91          sta = ClearList ( lists . list [nowIndex]);
92          if (sta == OK){
93                  printf ("OK!\n");
94          }else{
95                  printf ("this list can not be cleared!\n");
96          }
97          break;
98  case 4:
99          sta = ListEmpty( lists . list [nowIndex]);
100         if (sta == INFEASIBLE){
101                 printf ("this list does not exist!\n");
102         }else if (sta == true){
103                 printf ("this list is empty!\n");
104         }else{
105                 printf ("this list is not empty!\n");
106         }
107         break;
108 case 5:
109         sta = ListLength( lists . list [nowIndex]);
110         if (sta == INFEASIBLE){
111                 printf ("this list does not exist!\n");
112         }else{
113                 printf ("this list's length = %d!\n", sta);
114         }
115         break;
116 case 6:
117          printf ("input the index of the element in the list!\n");
```

```
118         scanf("%d", &index);
119         sta = GetElem(lists.list[nowIndex],index,e);
120         if (sta == INFEASIBLE){
121                 printf("this list does not exist!\n");
122         }else if (sta ==ERROR){
123                 printf("this index does not exist in the list!\n");
124         }else{
125                 printf("you get it : %d!\n", e);
126         }
127         break;
128 case 7:
129         printf("input the element you want to find in the list!\n");
130         scanf("%d", &e);
131         sta = LocateElem(lists.list[nowIndex],e);
132         if (sta == INFEASIBLE){
133                 printf("this list does not exist!\n");
134         }else if (sta ==ERROR){
135                 printf("this element does not exist in the list!\n");
136         }else{
137                 printf("you get its index : %d!\n", sta);
138         }
139         break;
140 case 8:
141         printf("input the element whose prior element you want to
              find in the list!\n");
142         scanf("%d", &e);
143         sta = PriorElem(lists.list[nowIndex],e,pre);
144         if (sta == INFEASIBLE){
145                 printf("this list does not exist!\n");
146         }else if (sta ==ERROR){
147                 printf("this prior does not exist in the list!\n");
```

```
148         }else{
149                 printf("you get its prior one : %d!\n", pre);
150         }
151         break;
152 case 9:
153         printf("input the element whose next element you want to find
                in the list!\n");
154         scanf("%d", &e);
155         sta = NextElem(lists.list[nowIndex],e,next);
156         if (sta == INFEASIBLE){
157                 printf("this list does not exist!\n");
158         }else if (sta ==ERROR){
159                 printf("this next does not exist in the list!\n");
160         }else{
161                 printf("you get its next one : %d!\n", next);
162         }
163         break;
164 case 10:
165         if ( lists . list [nowIndex] == NULL){
166                 printf("this list hasn't been initialized!\n");
167                 break;
168         }
169         printf("input the element you want to insert in the list!\n")
                ;
170         scanf("%d", &e);
171         printf("input the index you want to insert in the list!\n");
172         int index;
173         scanf("%d", &index);
174         sta = ListInsert ( lists . list [nowIndex], index, e);
175         if (sta == INFEASIBLE){
176                 printf("this list does not exist!\n");
```

```
177         }else if (sta ==ERROR){
178                 printf("this index does not exist in the list!\n");
179         }else{
180                 printf("OK!\n");
181         }
182         break;
183 case 11:
184         if ( lists . list [nowIndex] == NULL){
185                 printf("this list hasn't been initialized!\n");
186                 break;
187         }
188          printf("input the index you want to delete in the list!\n");
189         scanf("%d", &index);
190         sta = ListDelete( lists . list [nowIndex], index , e);
191         if (sta == INFEASIBLE){
192                 printf("this list does not exist!\n");
193         }else if (sta ==ERROR){
194                 printf("this index does not exist in the list!\n");
195         }else{
196                 printf("you have deleted %d\n", e);
197         }
198         break;
199 case 12:
200         sta = ListTraverse( lists . list [nowIndex]);
201         if (sta == INFEASIBLE){
202                 printf("this list does not exist!\n");
203         }else{
204                 printf("OK!\n");
205         }
206         break;
207 case 13:
```

```
208         sta = reverseList( lists.list[nowIndex]);
209         if (sta == INFEASIBLE){
210                 printf("this list does not exist!\n");
211         }else{
212                 printf("OK!\n");
213         }
214         break;
215 case 14:
216         if ( lists.list[nowIndex] == NULL){
217                 printf("this list hasn't been initialized!\n");
218                 break;
219         }
220         printf("input the index (from the bottom) you want to delete
                in the list!\n");
221         scanf("%d", &index);
222         sta = RemoveNthFromEnd(lists.list[nowIndex], index, e);
223         if (sta == INFEASIBLE){
224                 printf("this list does not exist!\n");
225         }else if (sta ==ERROR){
226                 printf("this index does not exist in the list!\n");
227         }else{
228                 printf("you have deleted %d\n", e);
229         }
230         break;
231 case 15:
232         sta = sortList( lists.list[nowIndex]);
233         if (sta == INFEASIBLE){
234                 printf("this list does not exist!\n");
235         }else{
236                 printf("OK!\n");
237         }
```

```
238         break;
239 case 16:
240         printf("input the filename where you want to save the list!\n
                ");
241         scanf("%s", f);
242         sta = SaveList(lists.list[nowIndex], f);
243         if (sta == INFEASIBLE){
244                 printf("this list does not exist!\n");
245         }else if (sta == ERROR){
246                 printf("open error!\n");
247         }else{
248                 printf("OK!\n");
249         }
250         break;
251 case 17:
252         printf("input the filename where you want to load the list!\n
                ");
253         scanf("%s", f);
254         sta = LoadList(lists.list[nowIndex], f);
255         if (sta == INFEASIBLE){
256                 printf("this list isn't empty so you can't load
                    anything onto it!\n");
257         }else if (sta == ERROR){
258                 printf("open error!\n");
259         }else{
260                 printf("OK!\n");
261         }
262         break;
263 case 18:                                    //添加一串元素;
264 if (lists.list[nowIndex] == NULL){
265         printf("this list hasn't been initialized!\n");
```

```
266             break;
267     }
268             printf("input a list ended with 0!\n");
269
270             tmpl = (LinkList)malloc(sizeof(LNode));
271             tailOftmpl = tmpl;
272             scanf("%d", &tmp);
273             while(tmp != 0){
274                     LinkList p = (LinkList)malloc(sizeof(LNode));
275                     p->data = tmp;
276                     tailOftmpl->next = p;
277                     tailOftmpl = p;
278                     scanf("%d", &tmp);
279             }
280             tailOftmpl->next = NULL;
281             lists.listTail[nowIndex]->next = tmpl->next;
282             lists.listTail[nowIndex] = tailOftmpl;
283             free(tmpl);
284             printf("OK!\n");
285             break;
286     case 19:
287             printf("choose a list from the lists! input an integer
                    ranging from 1 to 20 : \n");
288             scanf("%d", &nowIndex);
289             printf("you chose list %d!\n", nowIndex);
290             break;
291     case 20:
292             goto out;
293                     }
294             }
295             out :
```

```
296         return 0;
297 }
```

Init.h

```
1  # ifndef INIT_H_                    // 防止头文件重复引入
2  # define INIT_H_
3  # include "Definition.h"
4  # include <malloc.h>
5  status  InitList ( LinkList &);    // 函数声明
6  # endif
```

Init.cpp

```
1  # include "Init.h"
2  status  InitList ( LinkList  &L)
3  {
4      if  (L != NULL)                          // 若L已经非空，则不能初始
           化
5          return INFEASIBLE;
6      L = (LinkList )malloc(sizeof(LNode));    // 创建结点
7      L->next = NULL;                          // 结点的下一结点设为空
8      return OK;
9  }
```

Destroy.h

```
1  # ifndef DESTROY_H_                 // 防止头文件重复引入
2  # define DESTROY_H_
3  # include "Definition.h"
4  # include <malloc.h>
5  status  DestroyList ( LinkList  &);    // 函数声明
6  # endif
```

Destroy.cpp

```
# include "Destroy.h"
status  DestroyList ( LinkList  &L)
{

    if  (L == NULL)                                  // 表是空的不能销毁
        return INFEASIBLE;
    LinkList  p = L->next;                           // 从第一个元素开始销毁
    while (p != NULL){
        LinkList  tmp = p->next;
        free (p);
        p = tmp;
    }
    free (L);                                        // 将表结点销毁后设置为空
    L = NULL;
    return OK;
}
```

Clear.h

```
# ifndef CLEAR_H_                 // 防止头文件重复引入
# define CLEAR_H_
# include "Definition.h"
# include <malloc.h>
status  ClearList ( LinkList &);    // 初始化链表的函数声明
# endif
```

Clear.cpp

```
# include "Clear.h"
status  ClearList ( LinkList  &L)
{
    if  (L == NULL)                          // 表不存在不能清空!
```

```
5           return INFEASIBLE;
6       LinkList  p = L->next;                    // 将从表的第一个元素开始清
            空!
7       while (p){
8           LinkList  tmp = p->next;
9           free(p);
10          p = tmp;
11      }
12      L->next = NULL;                           //  清空操作不能删除表头结点!
13      return OK;
14  }
```

ListEmpty.h

```
1   # ifndef LISTEMPTY_H_                  // 防止头文件重复引入
2   # define LISTEMPTY_H_
3   # include "Definition.h"
4   status  ListEmpty( LinkList );     // 函数声明
5   # endif
```

ListEmpty.cpp

```
1   # include "ListEmpty.h"
2   status  ListEmpty( LinkList  L)
3   {
4       if  (L == NULL)                           // 表不存在，不能判断!
5           return INFEASIBLE;
6       return L->next == NULL;                   // 表是空的意味着结点的下一结
            点是空!
7   }
```

ListLength.h

```
1   # ifndef LISTLENGTH_H_                  // 防止头文件重复引入
```

```
2 # define LISTLENGTH_H_
3 # include "Definition.h"
4 # include <malloc.h>
5 int ListLength( LinkList );    // 函数声明
6 # endif
```

ListLength.cpp

```
1  # include "ListLength.h"
2  int ListLength( LinkList L)
3  {
4      if (L == NULL)                          // 表不存在，不能读取长
           度!
5          return INFEASIBLE;
6      int cnt = 0;
7      LinkList p = L->next;                   // 从第一个元素开始算!
8      while (p){
9          cnt ++;
10         p = p->next;
11     }
12     return cnt;                             // 返回结果!
13 }
```

GetElem.h

```
1  # ifndef GETELEM_H_              // 防止头文件重复引入
2  # define GETELEM_H_
3  # include "Definition.h"
4  # include "ListLength.h"
5  status GetElem(LinkList, int, ElemType&);  // 函数声明
6  # endif
```

GetElem.cpp

```
# include "GetElem.h"
status  GetElem(LinkList  L,int  i,ElemType &e)
{
    if  (L == NULL)
        return INFEASIBLE;
    int  length  = ListLength(L);
    if  (i  < 1  ||  i  > length)
        return ERROR;
    LinkList  p = L;
    while (i--){
        p = p->next;
    }
    e = p->data;
    return OK;
}
```

LocateElem.h

```
# ifndef LOCATEELEM_H_              // 防止头文件重复引入
# define LOCATEELEM_H_
# include "Definition.h"
status  LocateElem(LinkList  L,ElemType e);    // 函数声明
# endif
```

LocateElem.cpp

```
# include "LocateElem.h"
status  LocateElem(LinkList  L,ElemType e)
{
    if  (L == NULL)
        return INFEASIBLE;
    LinkList  p = L->next;
    int  index  = 1;
```

```
8       while (p){
9           if (p->data == e)
10              return index;
11          index ++;
12          p = p->next;
13      }
14      return ERROR;
15  }
```

PriorElem.h

```
1   # ifndef PIRORELEM_H_                    // 防止头文件重复引入
2   # define PIRORELEM_H_
3   # include "Definition.h"
4   status PriorElem( LinkList ,ElemType,ElemType& );  // 函数声明
5   # endif
```

PriorElem.cpp

```
1   # include "PriorElem.h"
2   status PriorElem( LinkList L,ElemType e,ElemType &pre)
3   {
4       if (L == NULL)
5           return INFEASIBLE;
6       LinkList p = L->next;
7       while (p && p->next){
8           if (p->next->data == e){
9               pre = p->data;
10              return OK;
11          }
12          p = p->next;
13      }
14      return ERROR;
```

```
}
```

NextElem.h

```
# ifndef NEXTELEM_H_              // 防止头文件重复引入
# define NEXTELEM_H_
# include "Definition.h"
status  NextElem(LinkList  ,ElemType ,ElemType& );   // 函数声明
# endif
```

NextElem.cpp

```
# include "NextElem.h"

status  NextElem(LinkList  L,ElemType e,ElemType &next)
{
    if  (L == NULL)
        return INFEASIBLE;

    LinkList  p = L->next;
    while (p && p->next){
        if  (p->data == e){
            next  = p->next->data;
            return OK;
        }
        p = p->next;
    }

    return ERROR;

}
```

ListInsert.h

```
# ifndef LISTINSERT_H_                    // 防止头文件重复引入
# define LISTINSERT_H_

# include "Definition.h"
# include <malloc.h>

status  ListInsert ( LinkList & ,int  ,ElemType );    // 函数声明

# endif
```

ListInsert.cpp

```
# include "ListInsert.h"

status  ListInsert ( LinkList  &L,int  i ,ElemType e)
{
    if  (L == NULL)
        return INFEASIBLE;

    if  (i  < 1)
        return ERROR;

    int  index  = 1;
    LinkList  p = L;
    while (p && index < i){
        p = p->next;
        index ++;
    }

    if  (p == NULL)
        return ERROR;

```

```
21      if (p->next == NULL){
22          p->next = (LinkList)malloc(sizeof(LNode));
23          p = p->next;
24          p->data = e;
25          p->next = NULL;
26      }else{
27          LinkList tmp = (LinkList)malloc(sizeof(LNode));
28          tmp->data = e;
29          tmp->next = p->next;
30          p->next = tmp;
31      }
32      return OK;
33  }
```

ListDelete.h

```
1  # ifndef LISTDELETE_H_                    // 防止头文件重复引入
2  # define LISTDELETE_H_
3  # include "Definition.h"
4  # include <malloc.h>
5  status  ListDelete ( LinkList &, int , ElemType& );   // 函数声明
6  # endif
```

ListDelete.cpp

```
1  # include "ListDelete.h"
2  status  ListDelete ( LinkList  &L,int  i ,ElemType &e)
3  {
4      if (L == NULL)
5          return INFEASIBLE;
6      if (i < 1)
7          return ERROR;
8      int index = 1;
```

```
    LinkList  p = L;
    while (p && index < i){
        p = p->next;
        index ++;
    }
    if (p == NULL || p->next == NULL)
        return ERROR;
    e = p->next->data;
    LinkList  tmp = p->next;
    p->next = tmp->next;
    free(tmp);
    return OK;
}
```

ListTraverse.h

```
# ifndef LISTTRAVERSE_H_              // 防止头文件重复引入
# define LISTTRAVERSE_H_

# include "Definition.h"
# include <stdio.h>

status  ListTraverse ( LinkList  );    // 函数声明

# endif
```

ListTraverse.cpp

```
# include "ListTraverse.h"

status  ListTraverse ( LinkList  L)
{
    if (L == NULL)
```

```
6           return INFEASIBLE;
7
8       LinkList  p = L->next;
9
10      while (p){
11          printf ("%d", p->data);
12          //  if  (p->next != NULL)
13              putchar(' ');
14          p = p->next;
15      }
16      return OK;
17
18  }
```

reverseList.h

```
1   # ifndef REVERSELIST_H_              // 防止头文件重复引入
2   # define REVERSELIST_H_
3
4   # include "Definition.h"
5   # include "ListLength.h"
6
7   status  reverseList ( LinkList &);    // 函数声明
8
9   # endif
```

reverseList.cpp

```
1   # include "reverseList.h"
2
3   status  reverseList ( LinkList & L){
4       if  (L == NULL)
5           return INFEASIBLE;
```

```
6
7       if (L->next == NULL)
8           return OK;
9
10      LinkList  p = L, tmp = p->next;
11
12      while (p && tmp){
13          LinkList  tmpp = tmp->next;
14          if (p != L)
15              tmp->next = p;
16          else
17              tmp->next = NULL;
18          p = tmp;
19          tmp = tmpp;
20      }
21
22      L->next = p;
23
24      return OK;
25  }
```

RemoveNthFromEnd.h

```
1   # ifndef REMOVENTHFROMEND_H_          // 防止头文件重复引入
2   # define REMOVENTHFROMEND_H_
3
4   # include "Definition.h"
5   # include "ListLength.h"
6   # include "ListDelete.h"
7
8   status  RemoveNthFromEnd(LinkList&, int, int& );    // 函数声明
9
```

```
10 # endif
```

RemoveNthFromEnd.cpp

```
1  # include "RemoveNthFromEnd.h"
2
3  status  RemoveNthFromEnd(LinkList& L, int n, int& e){
4      if  (L == NULL)
5          return ERROR;
6
7      int  len  = ListLength(L);
8      n = len - n + 1;
9
10     return  ListDelete (L, n, e);
11 }
```

sortList.h

```
1  # ifndef SORTLIST_H_                  // 防止头文件重复引入
2  # define SORTLIST_H_
3
4  # include "Definition.h"
5  # include "ListLength.h"
6
7  status   sortList ( LinkList );     // 函数声明
8  void  quickSort (ElemType[], int , int );
9  # endif
```

sortList.cpp

```
1  # include "sortList.h"
2
3  void  quickSort (ElemType arr [],  int  i , int  j){
4      if  (i  >= j)
```

```
5           return;
6
7       int low = i;
8       int high = j;
9       ElemType k = arr[low];
10
11      while (low < high){
12          while (low < high && k <= arr[high]){
13              high --;
14          }
15
16          if (low < high && k > arr[high]){
17              arr[low] = arr[high];
18              low ++;
19          }
20
21          while (low < high && k >= arr[low]){
22              low ++;
23          }
24
25          if (low < high && k < arr[low]){
26              arr[high] = arr[low];
27              high --;
28          }
29      }
30      arr[low] = k;
31
32      quickSort(arr, i, low-1);
33      quickSort(arr, low+1, j);
34  }
35
```

```
36 status  sortList ( LinkList  L){
37     if  (L == NULL)
38         return INFEASIBLE;
39 
40     if  (L ->next == NULL)
41         return OK;
42 
43     int  len  = ListLength(L);
44     ElemType arr[len];
45     LinkList  p = L->next;
46 
47     for  (int  i  = 0; i  < len;  p = p->next, i++){
48         arr[i] = p->data;
49     }
50 
51     quickSort(arr ,  0,  len-1);
52 
53     p = L->next;
54 
55     for  (int  i  = 0; i  < len;  p = p->next, i++){
56         p->data = arr[i];
57     }
58 
59     return OK;
60 }
```

saveandload.h

```
1 # ifndef SAVEANDLOAD_H_            // 防止头文件重复引入
2 # define SAVEANDLOAD_H_
3 
4 # include "Definition.h"
```

```
5  # include <malloc.h>
6  # include <stdio.h>
7
8  status  SaveList( LinkList  ,char  []) ;    // 函数声明
9  status  LoadList( LinkList &, char  []) ;
10
11 # endif
```

saveandload.cpp

```
1  # include "saveandload.h"
2
3  status  SaveList( LinkList  L,char FileName[])
4  {
5      if  (L == NULL)
6          return INFEASIBLE;
7
8      FILE* fp  = NULL;
9      fp  = fopen(FileName, "wb");
10     if  (fp  == NULL)
11         return ERROR;
12
13     LinkList  p = L->next;
14     while (p){
15         fwrite (&p->data, sizeof(ElemType), 1, fp);
16         p = p->next;
17     }
18     fclose (fp);
19
20     return OK;
21
22 }
```

```
23
24 status LoadList( LinkList &L,char FileName[])
25 {
26     if (L != NULL && L->next != NULL)
27         return INFEASIBLE;
28
29     FILE* fp = NULL;
30     fp = fopen(FileName, "rb");
31     if (fp == NULL)
32         return ERROR;
33
34     free(L);
35     L = NULL;
36     L = (LinkList)malloc(sizeof(LNode));
37     LinkList tail = L;
38     while (fgetc(fp) != EOF){
39         fseek(fp, -1, SEEK_CUR);
40
41         LinkList tmp = (LinkList)malloc(sizeof(LNode));
42         fread(&tmp->data, sizeof(ElemType), 1, fp);
43         tail->next = tmp;
44         tail = tmp;
45     }
46     tail->next = NULL;
47     fclose(fp);
48     return OK;
49
50 }
```

lists.h

```
1 # ifndef LISTS_H_                    // 防止头文件重复引入
```

```
2  # define LISTS_H_
3  # include "Definition.h"
4  # include "Destroy.h"
5  # include <malloc.h>
6  #define LISTS_INIT_SIZE 21
7  typedef struct {
8          LinkList * list ;
9          LinkList * listTail ;
10         int length ;
11         int size ;
12 } Lists ;
13 status  initLists ( Lists &);
14 # endif
```

lists.cpp

```
1  # include "Lists.h"
2  status  initLists ( Lists & lists ){
3       lists . length = 0;
4       lists . size = LISTS_INIT_SIZE;
5       lists . list = NULL;
6       lists . list = (LinkList *)malloc(sizeof( LinkList )*(LISTS_INIT_SIZE));
7      if ( lists . list == NULL){
8           return ERROR;
9      }
10      lists . listTail = NULL;
11      lists . listTail = (LinkList *)malloc(sizeof( LinkList )*(LISTS_INIT_SIZE));
12     if ( lists . listTail == NULL){
13          return ERROR;
14     }
15
16     for (int i = 0; i < lists . size ; i++){
```

```
17            lists . list [i] = NULL;
18            lists . listTail [i] = NULL;
19        }
20
21        return OK;
22    }
```

附录 C 基于二叉链表二叉树实现的源程序

附录 D 基于邻接表图实现的源程序