

**课 程 实 验 报 告**

**课程名称： 数据结构实验**

**专业班级： 计卓1501**

**学 号： U201514893**

**姓 名： 陈志浩**

**指导教师： 许贵平**

**报告日期： 2016年 12月 8 日**

**计算机科学与技术学院**

目录

1.基于顺序存储结构的线性表实现1

1.1问题描述1

1.2模块设计1

1.3模块实现4

1.4模块测试8

1.5实验总结13

2.基于链式存储结构的线性表实现14

2.1问题描述14

2.2模块设计14

2.3模块实现17

2.4模块测试23

2.5实验总结29

3.基于二叉链表的二叉树实现30

3.1问题描述30

3.2模块设计30

3.3模块实现33

3.4模块测试41

3.5实验总结47

4. 基于邻接表的图实现48

4.1问题描述48

4.2模块设计48

4.3模块实现52

4.4模块测试57

4.5实验总结62

5. 参考文献63

6. 指导教师评定意见64

附录1 基于顺序存储结构的线性表实现65

附录2 基于链式存储结构的线性表实现80

附录3 基于二叉链表的二叉树实现99

附录4 基于邻接表的图实现138

**基于顺序存储结构的线性表实现**

1. **问题描述**

线性表是n个数据元素的有限序列。线性表的顺序表示指的是用一组地址连续的存储单元依次存储线性表的数据元素。本实验要求封装一个基于顺序存储结构的线性表ADT模块，提供线性表ADT基本的、常见的12种运算，并且为该线性表ADT提供文件的I/O方法，且该实现方案要具有良好的鲁棒性以及可移植性，并配套简单的演示系统。

1. **模块设计**

## 2.1总体架构

根据实验的要求以及C语言的相关规范标准，给出整个模块的总体架构：

模块包括一个头文件（.h）和一个实现文件（.c）

模块的宏定义#define、类型别名typedef、结构定义和声明、函数原型被包括在头文件中

模块的ADT操作实现以及文件操作实现（函数）被包括在实现文件中

## 2.2数据结构设计

数据结构的定义

根据实验的要求以及线性表ADT的定义，给出基于C语言实现的线性表ADT的结构定义：

**struct** SequenceList {

Element \* elem;

**int** length;

**int** listsize;

};

其中，Element的定义由具体的需求决定，在本次实验的中，Element的定义如下：

**struct** Element {

**int** value;

};

## 2.3 ADT操作设计

根据实验的要求以及线性表ADT的定义，该线性表ADT应包括如下的操作：

InitList（初始化），DestroyList（销毁），ClearList（清空），ListEmpty（判断表是否为空），ListLength（求表长），GetElem（按下标取得元素），LocateElem（按满足关系取得元素），PriorElem（返回满足关系的元素的前驱），NextElem（返回满足关系元素的后继），ListInsert（插入元素），ListDelete（删除元素），ListTraverse（遍历表）

根据相关ADT操作的特性，给出如下基于C语言的相关定义。

为了方便InitList和ListInsert相关操作的管理，在C语言中给出如下的宏定义：

#define LIST\_INIT\_SIZE 100

#define LIST\_INCREMENT 10

为了ADT操作统一性以及错误处理，在C语言中给出如下的通用函数原型：

Status MethodName (Typename1 arg1, Typename2 arg2, ...);

其中，Status表示为预定义的枚举类型，用来表明函数执行的结果以及错误的处理，在C语言中的定义如下：

**typedef** **enum** Status {OK, NULL\_POINTER = 100, LIST\_NOT\_EXIST, MEMORY\_OVERFLOW, OUT\_OF\_RANGE, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

OK表明函数执行成功，状态正常；NULL\_POINTER表明函数的参数中带有空指针；LIST\_NOT\_EXIST表明操作的表还未初始化，不存在；MEMORY\_OVERFLOW表明内存已满，分配空间行为失败；OUT\_OF\_RANGE表明下标越界；EXTERNAL\_FUNCTION\_ERROR表明函数调用的外部函数错误；FILE\_ERROR表明文件操作失败。

为了存储判断的结果，还需要预定义一个枚举类型Bool，在C语言中的定义如下：

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

FALSE为假，TRUE为真

为了模块的扩展性，Locate、PriorElem、NextElem所用到的判断两个元素是否满足关系的函数Compare以及ListTraverse所用到的访问函数Visit由外部提供，使用函数指针作为参数，在C语言中的定义如下：

**typedef** Bool (\*Compare)(Element elem1, Element elem2);

**typedef** Status (\*Visit)(Element elem);

基于以上的定义，给出所有ADT操作在C语言中对应得函数原型：

Status InitList (SequenceList\* pList);

Status DestroyList (SequenceList\* pList);

Status ClearList (SequenceList\* pList);

Status ListEmpty (SequenceList list, Bool\* pBool);

Status ListLength (SequenceList list, **int**\* pCount);

Status GetElem (SequenceList list, **int** index, Element\* pElem);

Status LocateElem (SequenceList list, Element elem1, Compare fp, **int**\* pIndex);

Status PriorElem (SequenceList list, Element curElem, Element\* pPreElem, Compare fp);

Status NextElem (SequenceList list, Element curElem, Element\* pNextElem, Compare fp);

Status ListInsert (SequenceList\* pList, **int** index, Element elem);

Status ListDelete (SequenceList\* pList, **int** index, Element\* pElem);

Status ListTraverse (SequenceList list, Visit fp);

## 2.4文件存储设计

根据实验的要求以及线性表ADT的数据结构，该线性表ADT应包括如下的文件操作：

FileWrite（将线性表写入文件）、FileRead（从文件中读取数据表）

根据ADT操作设计中的定义，给出文件存储操作在C语言中对应的函数原型：

Status FileWrite(SequenceList list, **char**\* filename);

Status FileRead(SequenceList\* pList, **char**\* filename);

## 2.5配套演示系统设计

配套演示系统采用文本菜单界面，该系统每次从输入流接受一个数字，并且根据数字的不同执行不同的操作。此外，该系统还应该实现模块中Locate、PriorElem、NextElem和ListTraverse函数所要求实现的compare和visit操作。

1. **模块实现**

## 3.1开发环境

开发本系统时的操作系统环境为macOS Sierra(10.12)。

编程环境

编程时没有使用IDE（集成开发环境）。

编程时使用了文本编辑器Atom进行了代码的编写。

编程时采用Apple LLVM编译器进行编译。版本为8.0.0(clang-800.0.38)。

运行环境

在调试时使用的操作系统环境为macOS Sierra(10.12)。

## 3.2 ADT操作实现

InitList：

该操作接收一个SequenceList指针，以传入的指针初始化一个线性表。

该操作首先检查指针是否为空，非空的话则通过动态分配空间的方式分配一个Element数组，并且将首地址存储于该线性表结构的成员变量elem中，最后配置该线性表的长度和初始大小。

该线性表的初始长度为0，初始大小由宏LIST\_INIT\_SIZE给定。

该操作的时间复杂度O(1)，空间复杂度O(1)。

DestroyList：

该操作接收一个SequenceList指针，将指针所对应的线性表销毁。

该操作首先检查指针以及线性表是否存在，都非空的话则释放动态分配空间的数组，并且将成员变量elem置空。

该操作的时间复杂度为O(1)，空间复杂度为O(1)。

ClearList：

该操作接收一个SequenceList指针，将指针所对应的线性表销毁。

该操作首先检查指针以及线性表是否存在，都非空的话则将线性表的长度置0。这样的删除方法是懒删除。可以减少操作的开销。

该操作的时间复杂服为O(1)，空间复杂度为O(1)。

ListEmpty：

该操作接收一个SequenceList结构变量和一个Bool指针，判断该线性表是否为空，并且将判断结果存入Bool指针所指向的变量中。

该操作首先检查线性表是否存在，存在的话则判断线性表的长度是否为0，是的话则将Bool指针所指向的变量修改为TRUE，否的话则将Bool指针所指向的变量修改为FALSE。

该操作的时间复杂度是O(1)，空间复杂度为O(1)。

ListLength：

该操作接收一个SequenceList结构变量和一个int指针，计算该线性表的长度，并且将长度值存入int指针所指向的变量中。

该操作首先判断线性表是否存在，存在的话则直接将线性表的长度存入int指针所指向的变量中。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

GetElem：

该操作接收一个SequenceList结构变量、一个int变量和一个Element指针，获取线性表中特定下标的元素，并且将该元素存入Element指针所指向的变量中。

该操作首先判断表是否存在以及传入作为下标的int值是否合法，如果线性表存在且下表合法，则直接将线性表中对应下标的元素存入Element指针所指向的变量中。

该操作的时间复杂度是O(1)，空间复杂度O(1)。

LocateElem：

该操作接收一个SequenceList结构变量、一个Element变量、一个Compare函数指针和一个int指针，获取线性表中第一个和Element变量满足关系Compare的元素的下标，并且将该下标存入int指针所指的变量中，如果没有满足条件的元素，则将0存入int指针所指的变量中。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的Element变量满足关系Compare，如果满足则直接结束遍历，并且将该元素下标存入int指针所指的变量中。如果遍历至表尾，则讲0存入int指针所指的变量中。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

PriorElem：

该操作接收一个SequenceList结构变量、一个Element变量、一个Element指针和一个Compare函数指针，找到和给定的Element变量满足关系Compare的元素的前驱（如果存在），并且将其存入Element指针所指的变量中，否则操作失败，Element指针所指向的变量无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的Element变量满足关系Compare，如果满足则直接结束遍历，并且将该元素的前驱存入Element指针所指的变量中。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

NextElem：

该操作接收一个SequenceList结构变量、一个Element变量、一个Element指针和一个Compare函数指针，找到和给定的Element变量满足关系Compare的元素的后继（如果存在），并且将其存入Element指针所指的变量中，否则操作失败，Element指针所指向的变量无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的Element变量满足关系Compare，如果满足则直接结束遍历，并且将该元素的后继存入Element指针所指的变量中。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

ListInsert：

该操作接收一个SequenceList指针、一个int变量、一个Element结构变量，将Element变量插入线性表中下标为int的元素之前。

该操作首先判断是否为空指针、表是否存在以及表示下标的int是否越界，如果不为空指针、表存在且未越界，则进一步判断线性表是否已满，如果已满，则重新分配空间，确认分配成功后修正线性表的大小listsize。然后使用一个指针指向线性表中对应下标的元素，从线性表的最后一个元素开始到给定下标的元素为止，每次将元素向后移动一位。最后将Element变量存入已经空出来的线性表特定下标所对应的位置并更新线性表长度。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

ListDelete：

该操作接收一个SequenceList指针、一个int变量和一个Element指针，删除线性表中下标int的元素，并将其存入Element指针所指的变量中。

该操作首先判断是否为空指针、表是否存在以及表示下标的int是否越界，如果不为空指针、表存在且未越界，则将线性表中对应的下标的元素存入Element指针所指的变量中， 然后从线性表中对应下标的下一个元素开始到最后一个元素为止，每次将元素向前移动一位。最后更新线性表长度。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

ListTraverse：

该操作接收一个SequenceList结构变量、一个Visit函数指针，依次对线性表中的每个元素调用Visit函数。

该操作首先检查表是否为空，非空的话，则循环遍历整个线性表，并且对每个元素调用Visit函数，如果Visit函数返回错误，则终止遍历，并抛出错误。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

## 3.3文件存储实现

FileWrite：

该操作接收一个SequenceList结构变量以及一个指向文件名的字符指针，实现将线性表存储进文件的功能。

该操作首先检查表是否为空表以及文件是否能够被打开／创建，如果非空且文件可以正常打开，则依次将线性表结构变量中的长度，大小和表写入文件，最后再关闭文件。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

FileRead：

该操作接收一个SequenceList指针以及一个指向文件名的字符指针，实现从文件中读取线性表的功能。

该操作首先检查指针是否为空以及文件操作是否正常，如果指针非空且文件操作正常则依次读入线性表的长度以及大小，然后为防止内存泄漏先执行销毁表命令，根据线性表的大小以动态分配内存的方式为表分配空间，根据读入的数据初始化表的长度和大小，再从文件中依次读取原线性表的元素，直到文件尾为止，最后再关闭文件。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

## 3.4配套演示系统实现

外部函数实现

compare：

该操作接收两个Element元素，判断他们是否相等，如果相等返回TRUE，不相等则返回FALSE。

visit：

该操作接收一个Element元素，并将它输出。

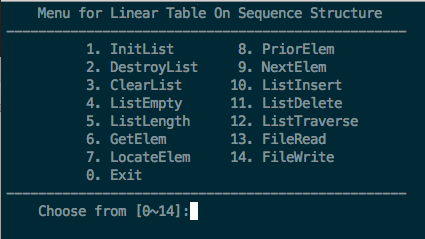
演示功能实现

该系统采用简单文本界面，首先输出系统菜单，然后从输入流中读入一个数字，在根据数字所对应的功能执行相应的代码。

1. **模块测试**

## 4.1配套演示系统操作介绍

演示系统的界面如下图所示：



该演示系统操作简单，输入对应的数字并按下回车即可，输入1-14表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入0退出系统。

## 4.2测试方案

测试数据应同时包括有效数据无效数据，有效数据旨在验证系统功能的正确性，而无效甚至是错误的数据旨在验证系统的鲁棒性。具体的测试方案如下。

方案一：

1. 初始化一个线性表。
2. 判断线性表是否为空以及测试线性表的长度。
3. 依次将1-10插入线性表下标为1的位置。
4. 再判断线性表是否为空以及测试线性表的长度。
5. 遍历线性表。
6. 得到下标为10的元素。
7. 获取第一个等于5的元素的下标。
8. 获取第一个等于8的元素的前驱。
9. 获取第一个等于2的元素的后继。
10. 删除下标为1和5的元素。
11. 遍历线性表。
12. 将线性表存入文件。
13. 重启系统，从文件中加载线性表。
14. 遍历线性表。

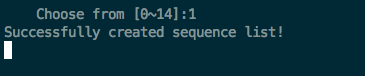
方案二：

1. 在不存在数据文件的情况下依次执行13、14。
2. 在未初始化线性表的情况下执行操作3、5、7、11、12、14。
3. 初始化线性表，在下标为2处插入数据1。

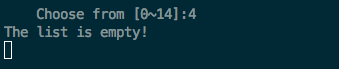
## 4.3测试结果

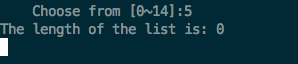
方案一：

1. 线性表初始成功

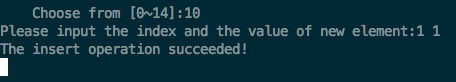


1. 线性表为空并且线性表长度为0

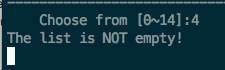


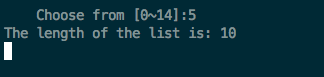


1. 插入成功



1. 线性表不为空且线性表长度为10

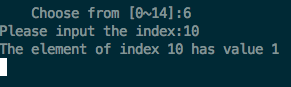




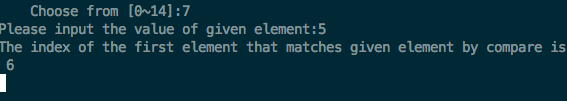
1. 遍历成功



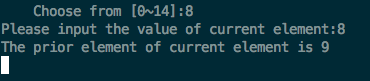
1. 下标10的元素值为1



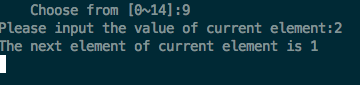
1. 第一个等于5的元素的下标为6



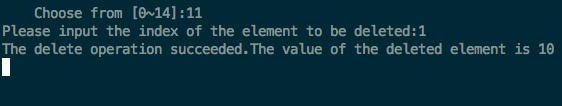
1. 第一个等于8的元素的前驱为9



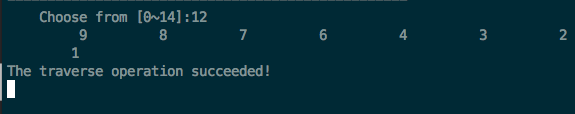
1. 第一个等于2的元素的后继是1



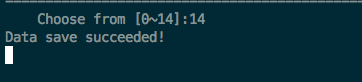
1. 删除成功



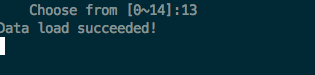
1. 遍历成功



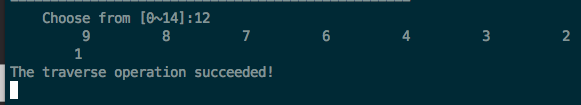
1. 数据保存成功



1. 数据读取成功

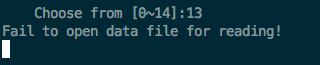


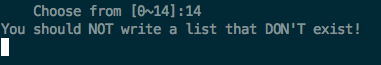
1. 遍历成功



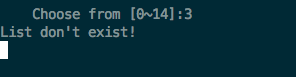
方案二：

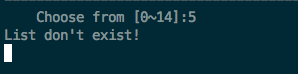
1. 加载失败：文件无法打开。操作失败：无法将空的线性表写入文件。

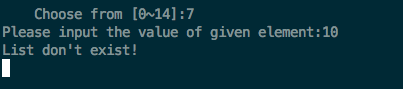


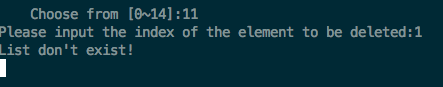


1. 操作失败：线性表不存在



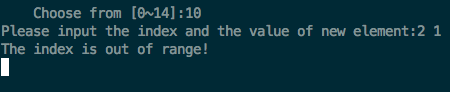






../../../../屏幕快照%202016-10-12%20下午10.30.27.png

1. 操作失败：下标越界



1. **实验总结**

通过本次实验，我加深了对于基于顺序结构的线性表的认识和了解。

在这次实验中，我己封装了一个可以使用的线性表模块。在这之中，我复习了学习过的C语言特性，了解了如何将函数作为参数传递等知识，为自己的模块建立了良好的规范，也尽可能的保证了该线性表ADT模块的独立性，方便其移植使用。

但是这次实验也有让我感到不满意的地方，由于C语言不支持范型，所以该线性表ADT模块在移植上仍然存在不优雅的地方，针对这种情况，我打算在下一次实验中利用C++的范型进行数据结构的封装。

**基于链式存储结构的线性表实现**

1. **问题描述**

线性表是n个数据元素的有限序列。线性表的链式表示指的是相邻两个数据元素在物理位置上不相邻的线性表，而是通过附加指针域的方式提供逻辑上的相邻关系。本实验要求封装一个基于链式存储结构的线性表ADT模块，提供线性表ADT基本的、常见的12种操作，并且为该线性表ADT提供文件的I/O方法，且该实现方案要具有良好的鲁棒性以及可移植性，并配套简单的演示系统。

1. **模块设计**

## 2.1总体架构

根据实验的要求以及C语言的相关规范标准，给出整个模块的总体架构：

模块包括一个头文件（.h）和一个实现文件（.c）

模块的枚举定义enum、结构定义、函数指针以及相应的别名typedef、函数原型被包括在头文件中

模块的结构声明、ADT操作实现以及文件操作实现（函数）被包括在实现文件中

## 2.2数据结构设计

数据结构的定义

根据实验的要求以及线性表ADT的定义，给出基于C语言实现的线性表ADT的结构定义和别名：

**struct** List {

**int** length;

**size\_t** elem\_size;

**struct** ListNode \* head;

};

**typedef** **struct** List \* List;

其中，length表示该线性表的长度，elem\_size表示该线性表中数据元素的字节数，head指向该线性表的表头。同时为了防止指针写法的混乱，将List \*定义为List线性表ADT的结点ListNode的定义和别名如下：

**struct** ListNode {

**void** \* elem;

**struct** ListNode \* next;

};

**typedef** **struct** ListNode \* Node;

其中，elem是基于C语言的泛型实现，通过void \*接收任何类型的数据；struct ListNode \*是指向下一个结点的指针，同时为了防止指针写法的混乱，将ListNode \*定义成Node。

在本次的实验中，采用了如下的结构作为数据元素：

**typedef** **struct** Contact {

**char** name[20];

**char** age;

**char** tel[12];

}Contact;

该结构表示一个用户通讯录，其中，name表示的是联系人的姓名，age表示的是联系人的年龄，tel表示的是联系人的电话。

## 2.3 ADT操作设计

根据实验的要求以及线性表ADT的定义，该线性表ADT应包括如下的操作：

InitList（初始化），DestroyList（销毁），ClearList（清空），ListEmpty（判断表是否为空），ListLength（求表长），GetElem（按下标取得元素），LocateElem（按满足关系取得元素），PriorElem（返回满足关系的元素的前驱），NextElem（返回满足关系元素的后继），ListInsert（插入元素），ListDelete（删除元素），ListTraverse（遍历表）

根据相关ADT操作的特性，给出如下基于C语言的相关定义。

为了ADT操作统一性以及错误处理，在C语言中给出如下的通用函数原型：

Status MethodName (Typename1 arg1, Typename2 arg2, ...);

其中，Status表示为预定义的枚举类型，用来表明函数执行的结果以及错误的处理，在C语言中的定义如下：

**typedef** **enum** Status {OK, NULL\_POINTER = 100, LIST\_NOT\_EXIST, MEMORY\_OVERFLOW, OUT\_OF\_RANGE, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

OK表明函数执行成功，状态正常；NULL\_POINTER表明函数的参数中带有空指针；LIST\_NOT\_EXIST表明操作的表还未初始化，不存在；MEMORY\_OVERFLOW表明内存已满，分配空间失败；OUT\_OF\_RANGE表明下标越界；EXTERNAL\_FUNCTION\_ERROR表明函数调用的外部函数错误；FILE\_ERROR表明文件操作失败。

为了存储判断的结果，还需要预定义一个枚举类型Bool，在C语言中的定义如下：

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

FALSE为假，TRUE为真

为了模块的扩展性，LocateElem、PriorElem、NextElem所用到的判断两个元素是否满足相等的函数Compare以及ListTraverse所用到的访问函数Visit由外部提供，使用函数指针作为参数，在C语言中的定义如下：

**typedef** Bool (\*Compare)(Element elem1, Element elem2);

**typedef** Status (\*Visit)(Element elem);

考虑到用户定义的数据元素结构中可能存在指针对象以及malloc，因此给出可选的两个函数指针Assign和Dealloc用于深拷贝和释放，由用户自行决定是否使用，如果不使用，模块将调用memcpy函数执行默认的浅拷贝。Assign和Dealloc函数指针在C语言中的定义如下：

**typedef** Status (\*Assign)(**void**\*, **const** **void**\*); //Include malloc and memcpy

**typedef** Status (\*Dealloc)(**void**\*); //Include free

基于以上的定义，给出所有ADT操作在C语言中对应得函数原型：

Status InitList (List\* pList, **size\_t** elem\_size);

Status DestroyList (List\* pList, Dealloc fp);

Status ClearList (List list, Dealloc fp);

Status ListEmpty (List list, Bool\* pBool);

Status ListLength (List list, **int**\* pCount);

Status GetElem (List list, **int** index, **void**\* pElem);

Status LocateElem (List list, **void**\* elem, Compare fp, **int**\* pIndex);

Status PriorElem (List list, **void**\* curElem, **void**\* pPreElem, Compare fp);

Status NextElem (List list, **void**\* curElem, **void**\* pNextElem, Compare fp);

Status ListInsert (List list, **int** index, **void**\* elem, Assign fp);

Status ListDelete (List list, **int** index, **void**\* elem, Dealloc fp, Assign fp2);

Status ListTraverse (List list, Visit fp);

## 2.4文件存储设计

根据实验的要求以及线性表ADT的数据结构，该线性表ADT应包括如下的文件操作：

FileWrite（将线性表写入文件）、FileRead（从文件中读取数据表）

根据ADT操作设计中的定义，给出文件存储操作在C语言中对应的函数原型：

Status FileWrite (List list, **char**\* filename);

Status FileRead (List\* pList, **char**\* filename, Dealloc fp);

## 2.5配套演示系统设计

配套演示系统采用文本菜单界面，该系统每次从输入流接受一个数字，并且根据数字的不同执行不同的操作。该系统还应该实现模块中LocateElem、PriorElem、NextElem和ListTraverse函数所要求实现的compare和visit操作。此外，数据元素的类型不再由模块提供，而是由演示系统提供，这表明，该模块已经有了良好的可移植性，可以应用于不同的情况。

1. **模块实现**

## 3.1开发环境

开发本系统时的操作系统环境为macOS Sierra(10.12)。

编程环境

编程时没有使用IDE（集成开发环境）。

编程时使用了文本编辑器Atom进行了代码的编写。

编程时采用Apple LLVM编译器进行编译。版本为8.0.0(clang-800.0.38)。

运行环境

在调试时使用的操作系统环境为macOS Sierra(10.12)。

## 3.2 ADT操作实现

由于**typedef** **struct** ListNode \* Node，以下皆用List指针表示ListNode \*，用List \*指针表示ListNode \*\*。

使用了void \*实现泛型，但是void \*的指针级别可能并不相同，需要注意。

InitList：

该操作接收一个List \*指针和一个size\_t变量，以传入的List \*指针和数据元素的大小初始化一个线性表。

该操作首先检查指针是否为空，非空的话则通过动态分配空间的方式分配一个List结构变量，将其指针存入\*List中，设置该List的长度和数据元素的大小。

然后，为该线性表初始化表头head，通过动态分配空间的方式分配一个ListNode结构变量并将其指针存入head中，并且将其elem成员变量设置为NULL，以标记表头，将其next成员变量初始化为NULL。最终，将List结构变量的成员指针head设置为表头head。

该线性表的初始长度为0，并且带有表头。

该操作的时间复杂度O(1)，空间复杂度O(1)。

DestroyList：

该操作接收一个List \*指针和一个Dealloc函数指针，将指针所指向的线性表销毁。

该操作首先检查指针以及线性表是否存在，非空且存在的话则从线性表的表头开始依次调用free函数释放链表的每一个结点。释放每一个结点前，都先检查用户是否传入了非空的Dealloc指针，如果Dealloc指针非空，则调用Dealloc方法释放结点所指向的数据元素，否则则调用free函数释放数据元素。最终，释放整个线性表的结构变量List并将其置空。

该操作的时间复杂度为O(n)，空间复杂度为O(1)。

ClearList：

该操作接收一个List指针和一个Dealloc函数指针，将指针所对应的线性表清空。

该操作与DestroyList操作相似，不同在于该操作不释放表头head，也不释放List。因此该操作不需要传入List \*指针。

该操作的时间复杂服为O(n)，空间复杂度为O(1)。

ListEmpty：

该操作接收一个List指针和一个Bool指针，判断该线性表是否为空，并且将判断结果存入Bool指针所指向的变量中。

该操作首先检查线性表是否存在，存在的话则判断线性表的表头head是否有后继，有的话则将Bool指针所指向的变量修改为FALSE，没有的话则将Bool指针所指向的变量修改为TRUE。

该操作也可以通过检查List指针所指向的length成员变量是否为0来实现。

该操作的时间复杂度是O(1)，空间复杂度为O(1)。

ListLength：

该操作接收一个List结构变量和一个int指针，计算该线性表的长度，并且将长度值存入int指针所指向的变量中。

该操作首先判断线性表是否存在，存在的话则直接将线性表的长度存入int指针所指向的变量中。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

GetElem：

该操作接收一个List指针、一个int变量和一个void二级指针，获取线性表中特定下标的元素，并且使void指针指向的指针指向该元素。

该操作首先判断表是否存在以及传入作为下标的int值是否合法，如果线性表存在且下表合法，则通过表头访问到线性表中对应下标的结点，并且将对应结点中指向数据元素指针的地址通过memcpy存入void指针所指的指针中。

该操作直接进行指针的赋值，被操作的两个指针指向同一块内存区域。

该操作的时间复杂度是O(n)，空间复杂度O(1)。

LocateElem：

该操作接收一个List指针、一个void一级指针、一个Compare函数指针和一个int指针，其中void一级指针指向一个特定的数据元素，获取线性表中第一个和该元素满足关系Compare的元素的下标，并且将该下标存入int指针所指的变量中，如果没有满足条件的元素，则将0存入int指针所指的变量中。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与void指针所指向的数据元素满足关系Compare，如果满足则直接结束遍历，并且将该元素下标存入int指针所指的变量中。如果遍历至表尾，则将0存入int指针所指的变量中。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

PriorElem：

该操作接收一个List指针、一个void一级指针、一个void二级指针和一个Compare函数指针，其中void一级指针指向一个特定的数据元素，void二级指针指向一个数据元素的指针，找到和特定数据元素满足关系Compare的元素的前驱（如果存在），并且将该前驱的地址通过memcpy赋值给void二级指针指向的数据元素的指针，否则操作失败，该二级指针指向的数据元素的指针无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的元素满足关系Compare，如果满足则直接结束遍历，并且将该元素的前驱的地址赋值给二级指针所指的指针。

该操作使用两个指针，两个指针相距一个元素的距离，以方便地获取对应元素的前驱。此外，需要注意空指针的处理。

该操作进行的是指针的赋值，被操作的两个指针指向同一快内存区域。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

NextElem：

该操作接收一个List指针、一个void一级指针、一个void二级指针和一个Compare函数指针，其中void一级指针指向一个特定的数据元素，void二级指针指向一个数据元素的指针，找到和特定数据元素满足关系Compare的元素的后继（如果存在），并且将该后继的地址通过memcpy赋值给void二级指针指向的数据元素的指针，否则操作失败，该二级指针指向的数据元素的指针无定义。

该操作首先判断表是否存在，如果存在则依次遍历整个线性表，判断线性表中的每个元素是否与传入的元素满足关系Compare，如果满足则直接结束遍历，并且将该元素的后继（如果不为NULL）的地址赋值给二级指针所指的指针。

该操作进行的是指针的赋值，被操作的两个指针指向同一快内存区域。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

ListInsert：

该操作接收一个List指针、一个int变量、一个void一级指针和一个Assign函数指针，将void一级指针所指向的数据元素插入线性表中下标为int的元素之前。

该操作首先判断表是否存在以及表示下标的int是否越界，如果表存在且未越界，则以动态分配的方式创建一个新的结点，并且以List中的elem\_size以动态分配的方式创建数据元素，将其设置为新建结点的数据域，然后判断Assign函数指针是否为空，如果非空，则调用用户自定义的函数Assign执行复制，如果为空，则调用memcpy将void一级指针指向的数据元素复制到动态分配的数据元素中。然后创建一个新的结点指针，根据int变量的值从头结点开始移动到所要插入位置的前一个结点处，执行插入操作并且将表长加1。

如果没有用户定义的复制方式，该操作执行默认的复制过程，在这种情况下，数据元素结构内通过指针存储的数据（调用malloc函数分配空间的数据等）将不会被正确的复制，而是会共用同一块内存。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

ListDelete：

该操作接收一个List指针、一个int变量、一个void一级指针、一个Dealloc函数指针和一个Assign函数指针，删除线性表中下标为int的元素，并将其存入void一级指针所指向的数据元素中。

该操作首先判断表是否存在以及表示下标的int是否越界，如果表存在且未越界，则先新建一个结点指针，根据int变量的值从表头开始移动到待删除结点的位置，修改相关结点的指针域以达到删除结点的目的，并且用temp指针暂存待释放的结点。然后判断Assign函数指针是否为空，如果非空，则调用用户定义的Assign函数执行复制，如果为空，则调用memcpy将该结点数据域所指向的数据元素复制到void一级指针所指向的数据元素中。然后判断Dealloc函数指针是否为空，如果非空，则执行用户定义的Dealloc函数进行数据元素释放，如果为空，则调用free函数进行数据元素的释放。最后释放结点并修改表长。

如果没有用户定义的复制和释放方式，该操作执行默认的复制和释放方式，即调用memcpy函数进行数据元素的复制，调用free函数进行数据元素的释放，与ListInsert函数相对应。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

ListTraverse：

该操作接收一个List指针、一个Visit函数指针，依次对线性表中的每个元素调用Visit函数。

该操作首先检查表是否为空，非空的话，则从首元结点开始循环遍历整个线性表，并且对每个元素调用Visit函数，如果Visit函数返回错误，则终止遍历，并抛出错误。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

## 3.3文件存储实现

FileWrite：

该操作接收一个List指针以及一个指向文件名的字符指针，实现将线性表存储进文件的功能。

该操作首先检查表是否为空表以及文件是否能够被打开／创建，如果非空且文件可以正常打开，则首先将线性表的长度和数据元素的大小（字节数）存入文件中，再创建一个新的结点指针，从首元结点开始，依次将数据元素存入文件中。最后关闭文件。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

FileRead：

该操作接收一个List \*指针、一个指向文件名的字符指针和一个Dealloc函数指针，实现从文件中读取线性表的功能。

该操作首先检查指针是否为空以及文件操作是否正常，如果指针非空且文件操作正常则依次读入线性表的长度以及数据元素的大小。为了防止内存泄漏先以List \*指针和Dealloc函数指针执行一次摧毁表DestroyList操作，然后以List \*指针和数据元素的大小elem\_size执行一次InitList操作，并且将线性表的长度和数据元素的大小存入新初始化的线性表中。然后根据读入的线性表的长度依次以动态存储的方式分配结点以及数据元素的空间，同时读入数据元素。最终为尾元结点设置next为NULL并且关闭文件。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

## 3.4配套演示系统实现

外部函数实现

compare：

该操作接收两个void一级指针，均指向特定类型的数据元素，判断他们是否相等，如果相等返回TRUE，不相等则返回FALSE。

在该实验的演示系统中，数据元素的类型已经在总体设计给出，在此不做赘述，该操作通过调用strcmp函数比较两个Contact数据结构的name（char \*）是否相等，如果相等则返回TRUE，不相等则返回FALSE。

visit：

该操作接收一个void一级指针，指向特定类型的数据元素，将其输出。

在该实验的演示系统中，该操作将Contact数据结构的三个成员变量name，age，tel的值依次打印输出。

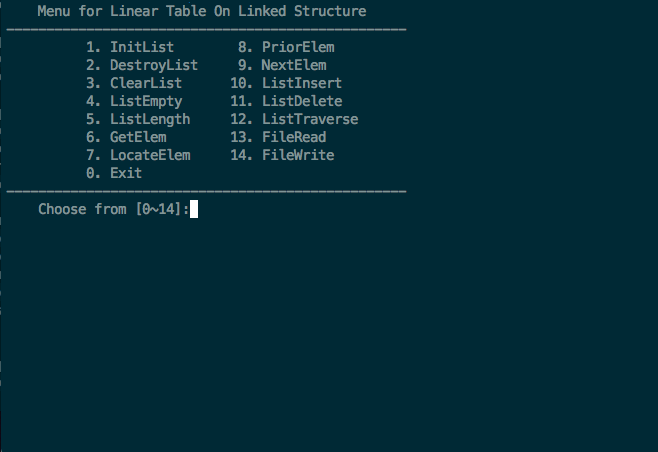
演示功能实现

该系统采用简单文本界面，实现的过程相对而言简单——首先输出系统菜单，然后从输入流中读入一个数字，在根据数字所对应的功能执行相应的代码。

1. **模块测试**

## 4.1配套演示系统操作介绍

演示系统的界面如下图所示：



该演示系统操作简单，输入对应的数字并按下回车即可，输入1-14表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入0退出系统。

## 4.2测试方案

测试数据应兼顾包括有效数据无效数据两个方面，有效数据旨在验证系统功能的正确性，而无效甚至是错误的数据旨在验证系统的鲁棒性。根据以上的要求，拟定了以下两个测试方案。

方案一：

1. 初始化一个线性表。
2. 判断线性表是否为空以及测试线性表的长度。
3. 依次将a，b，c，d四个联系人插入线性表下标为1-4的位置。
4. 再判断线性表是否为空以及测试线性表的长度。
5. 遍历线性表。
6. 得到下标为4的元素。
7. 获取第一个姓名成员为a的元素的下标。
8. 获取第一个姓名成员为c的元素的前驱。
9. 获取第一个姓名成员为b的元素的后继。
10. 删除下标为1和3的元素。
11. 遍历线性表。
12. 将线性表存入文件。
13. 重启系统，从文件中加载线性表。
14. 遍历线性表。

方案二：

1. 在不存在数据文件的情况下依次执行13、14。
2. 在未初始化线性表的情况下执行操作3、4、6、9、11、12。
3. 在初始化线性表但表为空的情况下执行操作8、9。
4. 在3的条件下使用6查询表中下标为2的元素。
5. 在只插入一个元素的情况下执行8、9。

## 4.3测试结果

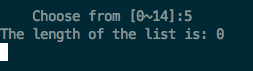
方案一：

1. 线性表初始化成功

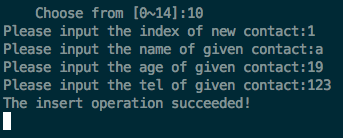
../../../../屏幕快照%202016-10-20%20下午9.52.30.png

1. 线性表为空并且线性表长度为0

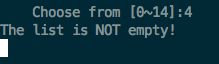
../../../../屏幕快照%202016-10-20%20下午9.58.27.png

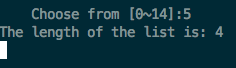


1. 插入成功

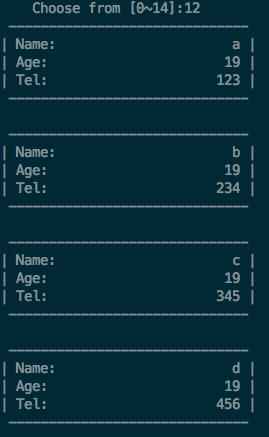


1. 线性表不为空且线性表长度为4

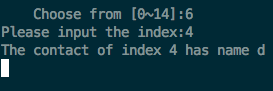




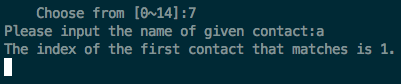
1. 遍历成功



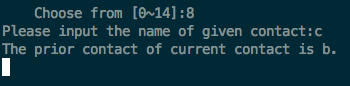
1. 下标为4的元素name成员为1



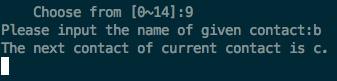
1. 第一个name成员为a的元素的下标为1



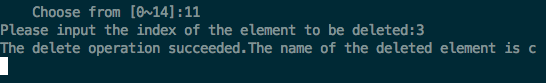
1. 第一个name成员为c的元素的前驱为b



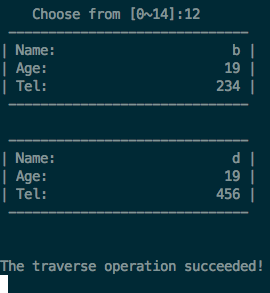
1. 第一个name成员为b的元素的后继为c



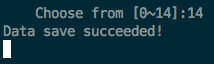
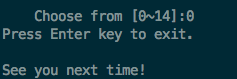
1. 删除成功



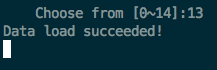
1. 遍历成功



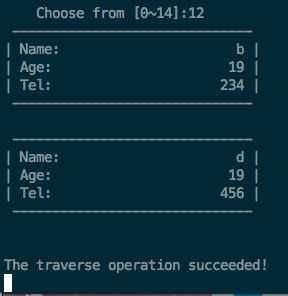
1. 数据保存成功并且退出系统

1. 数据读取成功



1. 遍历成功



方案二：

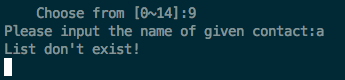
1. 加载失败：文件无法打开。操作失败：无法将空的线性表写入文件。

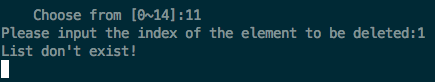
../../../../屏幕快照%202016-10-20%20下午10.35.53.png

../../../../屏幕快照%202016-10-20%20下午10.36.01.png

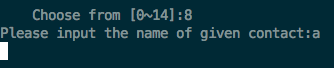
1. 操作失败：线性表不存在（其他操作结果相似，不赘述）

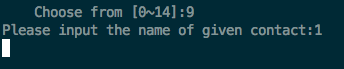
../../../../屏幕快照%202016-10-20%20下午10.36.52.png



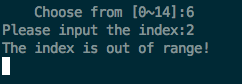


1. 操作失败：elem未定义，但程序没有崩溃





1. 操作失败：下标越界



1. 操作失败：原因和结果同3，但是程序没有崩溃
2. **实验总结**

通过本次实验，我加深了对于基于链式结构的线性表的理解。

在本次实验中，我根据C的void \*可以指向任意数据类型的特点，实现了一个简单的泛型编程，在保留上次实验的代码风格以及规范之外，我成功地将数据元素的申明与定义从模块中抽离了出来，保证了模块的低耦合性、扩展性与可移植性。

但是同样我们也要注意到，在C语言中实现泛型是一件不安全的事情，C语言仅能提供有限的检查，这就要求程序员必须保证代码的严谨与正确，否则，可能会导致无法预料的错误。

除此以外，通过这次实验，我还加深了对于C语言中指针的理解，在刚开始编写函数的过程中，由于我对于指针以及泛型的理解不足，再加上对于模块功能的理解不够全面，导致了一些设计上的失误，如：对GetElem操作执行数据元素的复制而忽略了GetElem操作的功能是返回指针以供进一步修改，对于ListInsert和ListDelete操作生搬硬套顺序表的函数原型传递指针的指针而忽略了链表本身是不需要修改指针的特点。好在我都能再调试的过程中发现这些不足并及时修改。

这次实验还反映出了我对于边界条件和错误的处理不够细致，比如在编写PriorElem和NextElem函数时，我虽然考虑到了要在这两个函数检查并处理指针为空的情况，却忘记了在演示系统的代码中对于空指针的处理，进而导致了程序的崩溃，这些都是我在以后的编程中需要注意的地方。

在下一次实验中，我会考虑采用C++强大的面向对象以及泛型特性来实现相应的数据结构模块，以实现一个安全性更高，封装的更好的数据结构模块。

**基于二叉链表结构的二叉树实现**

1. **问题描述**

二叉树是一种树型结构，二叉树的每个结点至多只有两棵子树。二叉树的二叉链表表示指的是数据元素在物理位置上不相邻的二叉树，而是通过附加指针域的方式提供逻辑上的关系。本实验要求封装一个基于二叉链表结构的二叉树ADT模块，提供二叉树ADT基本的、常见的20种操作，并且为该二叉树ADT提供文件的I/O方法，且该实现方案要具有良好的鲁棒性以及可移植性，并配套简单的演示系统。

1. **模块设计**

## 2.1总体架构

根据实验的要求以及C语言的相关规范标准，给出整个模块的总体架构：

模块包括一个二叉树头文件（.h）和一个二叉树实现文件（.c）

模块还包括了一个栈ADT模块和一个队列ADT模块，以实现非递归遍历二叉树以及层序遍历二叉树

模块的枚举定义enum、结构定义、结构声明、函数指针以及相应的别名typedef、函数原型被包括在头文件中

模块的ADT操作实现以及文件操作实现（函数）以及相关的内部静态例程被包括在实现文件中

## 2.2数据结构设计

数据结构的定义

根据实验的要求以及二叉树ADT的定义，给出基于C语言实现的二叉树ADT的结构定义和别名：

**struct** BiTree {

**size\_t** elem\_size;

**struct** BiTreeNode \* root;

};

**typedef** **struct** BiTree \* Tree;

其中，elem\_size表示该二叉树中数据元素的字节数，root指向该二叉树的根结点。同时为了防止指针写法的混乱，将BiTree \*定义为Tree

二叉树ADT的结点BiTreeNode的定义和别名如下：

**struct** BiTreeNode {

**void** \* elem;

**struct** BiTreeNode \* lchild;

**struct** BiTreeNode \* rchild;

};

**typedef** **struct** BiTreeNode \* Node;

**typedef** **struct** BiTreeNode \* Root;

其中，elem是基于C语言的泛型实现，通过void \*接收任何类型的数据；struct BiTreeNode \* lchild是指向左孩子的指针，rchild是指向右孩子的指针，同时为了防止指针写法的混乱以及区分根节点，将BiTreeNode \*定义成Root和Node。

在本次的实验中，采用了如下的结构作为数据元素：

**typedef** **struct** Contact {

**char** name[20];

**int** age;

**char** tel[12];

}Contact;

该结构表示一个用户通讯录，其中，name表示的是联系人的姓名，age表示的是联系人的年龄，tel表示的是联系人的电话。

## 2.3 ADT操作设计

根据实验的要求以及线性表ADT的定义，该线性表ADT应包括如下的操作：

InitTree（初始化），DestroyTree（销毁），CreateTree（构造），ClearTree（清空），TreeEmpty（判断树是否为空），TreeDepth（求树的深度），TreeRoot（取得根节点），Value（取得结点的值），Assign（为结点赋值），Parent（取得结点的父结点），LeftChild（取得结点的左孩子），RighitChild（取得结点的右孩子），LeftSibling（取得结点的左兄弟），RightSibling （取得结点的右兄弟），InsertChild（插入子树），DeleteChild（删除子树），PreOrderTraverse（前序遍历），InOrderTraverse（中序遍历），PostOrderTraverse（后序遍历）。

根据相关ADT操作的特性，给出如下基于C语言的相关定义。

为了ADT操作统一性以及错误处理，在C语言中给出如下的通用函数原型：

Status MethodName (Typename1 arg1, Typename2 arg2, ...);

其中，Status表示为预定义的枚举类型，用来表明函数执行的结果以及错误的处理，在C语言中的定义如下：

**typedef** **enum** Status {OK, NULL\_POINTER = 100, TREE\_NOT\_EXIST, RIGHT\_CHILD\_NOT\_EMPTY, MEMORY\_OVERFLOW, OUT\_OF\_RANGE, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

OK表明函数执行成功，状态正常；NULL\_POINTER表明函数的参数中带有非法的空指针；TREE\_NOT\_EXIST表明操作的表还未初始化，不存在；RIGHT\_CHILD\_NOT\_EMPTY表明右孩子非空，MEMORY\_OVERFLOW表明内存已满，分配空间失败；OUT\_OF\_RANGE表明下标越界；EXTERNAL\_FUNCTION\_ERROR表明函数调用的外部函数错误；FILE\_ERROR表明文件操作失败。

为了存储判断的结果，还需要预定义一个枚举类型Bool，在C语言中的定义如下：

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

FALSE为假，TRUE为真

为了模块的扩展性，可能会用到的判断两个元素是否满足相等的函数Compare、Traverse所用到的访问函数Visit以及构造树所用的数据元素构造函数Construct由外部提供，使用函数指针作为参数，在C语言中的定义如下：

**typedef** Bool (\*Compare)(Element elem1, Element elem2);

**typedef** Status (\*Visit)(Element elem);

**typedef** Status (\*Construct)(**void**\* pElem);

基于以上的定义，给出所有ADT操作在C语言中对应得函数原型：

Status InitTree(Tree\* pTree, **size\_t** elem\_size);

Status DestroyTree(Tree\* pTree);

Status PreOrderCreateTree(Tree tree, Construct construct);

Status ClearTree(Tree tree);

Status TreeEmpty(Tree tree, Bool\* pBool);

Status TreeDepth(Tree tree, **int**\* pCount);

Status TreeRoot(Tree tree, Root\* root);

Status NodeValue(Tree tree, Node node, **void**\* pElem);

Status NodeAssign(Tree tree, Node node, **void**\* elem);

Status NodeParent(Tree tree, Node node, Node\* pNode);

Status NodeLeftChild(Tree tree, Node node, Node\* pNode);

Status NodeRightChild(Tree tree, Node node, Node\* pNode);

Status NodeLeftSibling(Tree tree, Node node, Node\* pNode);

Status NodeRightSibling(Tree tree, Node node, Node\* pNode);

Status InsertChild(Tree tree, Node node, Child orient, Node child);

Status DeleteChild(Tree tree, Node node, Child orient);

Status PreOrderTraverse(Tree tree, Visit visit);

Status InOrderTraverse(Tree tree, Visit visit);

Status PostOrderTraverse(Tree tree, Visit visit);

Status LevelOrderTraverse(Tree tree, Visit visit);

## 2.4文件存储设计

根据实验的要求以及二叉树ADT的数据结构，该二叉树ADT应包括如下的文件操作：

FileWrite（将二叉树写入文件）、FileRead（从文件中读取二叉树）

根据ADT操作设计中的定义，给出文件存储操作在C语言中对应的函数原型：

Status FileWrite(Tree tree, **char**\* filename);

Status FileRead(Tree\* pTree, **char**\* filename);

## 2.5配套演示系统设计

配套演示系统采用文本菜单界面，该系统每次从输入流接受一个数字，并且根据数字的不同执行不同的操作。该系统还应该实现模块中所要求实现的construct和visit操作。此外，数据元素的类型不再由模块提供，而是由演示系统提供，这表明，该模块已经有了良好的可移植性，可以应用于不同的情况。

1. **模块实现**

## 3.1开发环境

开发本系统时的操作系统环境为macOS Sierra(10.12)。

编程环境

编程时没有使用IDE（集成开发环境）。

编程时使用了文本编辑器Atom进行了代码的编写。

编程时采用Apple LLVM编译器进行编译。版本为8.0.0(clang-800.0.38)。

运行环境

在调试时使用的操作系统环境为macOS Sierra(10.12)。

## 3.2 ADT操作实现

由于**typedef** **struct** BiTree \* Tree，以下皆用Tree指针表示BiTree \*用

Tree \*指针表示BiTree \*\*，BiTreeNode同理。

使用了void \*实现泛型，但是void \*的指针级别可能并不相同，需要注意。

InitTree：

该操作接收一个Tree \*指针和一个size\_t变量，以传入的Tree \*指针和数据元素的大小初始化一个二叉树。

该操作首先检查指针是否为空，非空的话则通过动态分配空间的方式分配一个Tree结构变量，将其指针存入\*Tree中，设置该二叉树的根节点为空。

该二叉树初始为空树。

该操作的时间复杂度O(1)，空间复杂度O(1)。

DestroyTree：

该操作接收一个Tree \*指针，将指针所指向的二叉树销毁。

该操作首先检查指针以及二叉树是否存在，非空且存在的话则会检查该树是否非空，若非空的话则调用递归例程后序释放树的所有结点，最后释放Tree指针所指向的结构变量并置空。

递归例程接收一个Node指针，按后序依次调用释放左孩子和释放右孩子的例程，最后释放数据域以及自身。

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

PreOrderCreateTree：

该操作接收一个Tree指针，和一个Construct函数指针，先序拓展构建二叉树。

该操作首先检查树是否存在以及construct函数是否非空，存在且非空的话则调用递归例程先序拓展构建二叉树。

递归例程接收一个Node \*指针，一个Construct函数指针和一个size\_t变量，首先调用构造函数construct，然后检查构造函数是否生成了数据元素，生成的话则以size\_t变量按动态分配的方式对\*Node指针生成结点，并且配置结点的数据域，并且依次对左孩子和右孩子调用子例程，如果构造函数未生成数据元素的话，则直接将\*Node置NULL。

该操作的时间复杂度为O(n)，空间复杂度为O(logn)。

ClearTree：

该操作接收一个Tree指针，将指针所对应的二叉树清空。

该操作与DestroyTree操作相似，不同在于该操作不释放Tree，而仅仅释放各结点并且将root指针置NULL。因此该操作不需要传入Tree \*指针。

该操作的时间复杂服为O(n)，空间复杂度为O(logn)。

TreeEmpty：

该操作接收一个Tree指针和一个Bool指针，判断该二叉树是否为空，并且将判断结果存入Bool指针所指向的变量中。

该操作首先检查二叉树是否存在以及Bool指针是否非空，存在且非空的话则判断二叉树的根节点root是否为空，否的话则将Bool指针所指向的变量修改为FALSE，是的话则将Bool指针所指向的变量修改为TRUE。

该操作的时间复杂度是O(1)，空间复杂度为O(1)。

TreeDepth：

该操作接收一个Tree指针和一个int指针，计算该二叉树的深度，并且将深度存入int指针所指向的变量中。

该操作首先判断二叉树是否存在以及int指针是否非空，存在且非空的话则调用递归例程计算二叉树的深度，并且将深度存入int指针所指向的变量中。

递归例程接收一个Node指针表示当前树的根节点，该例程会判断自己的左孩子和右孩子是否非空，对于非空的孩子调用子例程计算深度，并且返回更大的返回值+1。

该操作的时间复杂度是O(n)，空间复杂度是O(logn)。

TreeRoot：

该操作接收一个Tree指针、一个Root \*指针，获取二叉树的根节点的指针。

该操作首先判断二叉树是否存在以及Root \*指针是否非空，如果二叉树存在且指针非空，则通过Tree指针的root成员变量访问到二叉树的根节点，并且存入Root \*指针所指向的区域。

该操作的时间复杂度是O(1)，空间复杂度O(1)。

NodeValue：

该操作接收一个Tree指针、一个Node指针、一个void二级指针，返回Node指针所指向的结点的数据域指针。

该操作首先二叉树是否存在以及指针是否非空，如果存在且非空则调用memcpy将Node指针的elem成员变量拷贝到void二级指针所指向的内存区域。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

NodeAssign：

该操作接收一个Tree指针，一个Node指针和一个void一级指针，以void指针所指向的数据元素为Node指针所指向的结点数据域赋值。

该操作首先判断树是否存在以及指针是否非空，如果存在且非空则检查结点的数据域是否为空，为空则为其动态分配存储空间，然后调用memcpy将elem所指向的数据元素拷贝到node元素的数据域所指向的内存区域中。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

NodeParent：

该操作接收一个Tree指针、一个Node指针和一个Node \*指针，取得Node指针所指向的结点的双亲结点，并且存入Node \*指针指向的指针中。

该操作首先判断二叉树是否存在以及指针是否为空，如果存在且非空则判断二叉树是否为空树，非空的话则检查Node指针所指向的结点是否为根结点，是的话则将\*(Node \*)置空，否则调用递归例程寻找Node指针的双亲结点，并且存入Node \*指针所指向的指针中。

递归例程接收一个Node指针表示当前树的根节点以及另一个Node指针表示目标结点，该例程检查当前树的根节点是否存在，存在的话该根节点的孩子是否是目标结点，是的话直接直接返回当前树的根节点，否则调用子例程检查当前树的孩子，如果不存在则直接返回NULL。

该操作的时间复杂度是O(logn)，空间复杂度是O(logn)。

NodeLeftChild、NodeRightChild：

该操作接收一个Tree指针、一个Node指针和一个Node \*指针，返回Node指针所指向结点的左孩子或右孩子（如果存在），否则返回NULL，并且存入Node \*指针所指向的指针中。

该操作首先检查二叉树是否为存在以及参数是否非空，如果二叉树存在且参数非空的话则将Node指针所指向的结点的左孩子或者是右孩子保存进Node \*指针所指向的指针。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

NodeLeftSibling、NodeRightSibling：

该操作接收一个Tree指针、一个Node指针和一个Node \*指针，返回Node指针所指向的结点的左兄弟或右兄弟（如果存在），否则返回NULL，并且存入Node \*指针所指向的指针中。

该操作首先判断二叉树是否为存在以及参数是否非空，如果存在且非空则继续检查二叉树的根节点是否为空，如果为空则直接将NULL存入Node \*指针所指向的指针中；否则调用findParent递归例程找到Node指针所指向结点的父结点，并且返回相应的左兄弟或右兄弟。

该操作的时间复杂度是O(logn)，空间复杂度是O(logn)。

InsertChild：

该操作接收一个Tree指针、一个Node指针、一个Child变量和一个Node指针表示子树的根节点（右子树为空），根据需求将子树插入Node指针所指向的结点的左子树或者是右子树，原结点的左子树或右子树为原子树的右子树。

该操作首先检查二叉树是否存在以及参数是否非空以及子树的右子树是否非空，如果为空则返回错误，如果非空则继续判断Child变量是左子树还是右子树，然后执行对应的插入操作。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

DeleteChild：

该操作接收一个Tree指针，一个Node指针和一个Child变量，删除Node指针所指向的结点的左子树或右子树。

该操作首先检查二叉树是否存在以及参数是否非空，如果存在并且非空的话，则判断Child变量的值，以对于左孩子或者是右孩子执行freeTree递归例程。

该操作的时间复杂度为O(m)，空间复杂度为O(logm)。（m为子树的规模）

PreOrderTraverse：

该操作接收一个Tree指针和一个Visit函数指针，按照先序遍历给定的二叉树。

根据宏RECURSIVE\_TRAVERSE的不同值，编译器将根据预编译指令执行不同的编译过程，因此，该函数具有递归和非递归两个版本。

递归版本首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则执行递归例程先序遍历整棵二叉树。

递归例程接收一个Node指针和一个Visit函数指针，对当前结点执行Visit函数，然后依次对于当前结点的左孩子和右孩子调用子例程。

非递归版本首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则新建一个栈。并且以二叉树的根结点初始化一个临时的Node指针。之后以如下的方式遍历：

1. 访问结点node，并将其入栈
2. 判断node的左孩子是否为NULL，如果为NULL，则访问栈顶结点并且出栈，并且将其右结点设为node，执行1。如果不为NULL，则将node的左孩子设为node。
3. 若node为NULL且栈为空，遍历结束。

该操作递归版本的时间复杂度是O(n)，空间复杂度是O(logn)。非递归版本的时间复杂度是O(n)，空间复杂度是O(n)。

InOrderTraverse：

该操作接收一个Tree指针和一个Visit函数指针，按照中序遍历给定的二叉树。

根据宏RECURSIVE\_TRAVERSE的不同值，编译器将根据预编译指令执行不同的编译过程，因此，该函数具有递归和非递归两个版本。

递归版本首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则执行递归例程中序遍历整棵二叉树。

递归例程接收一个Node指针和一个Visit函数指针，对当前结点的左孩子调用子例程，执行Visit函数，然后对于当前结点的右孩子调用子例程。

非递归版本首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则新建一个栈。并且以二叉树的根结点初始化一个临时的Node指针。之后以如下的方式遍历：

1. 若当前结点node的左孩子不为空，将其入栈并且将左孩子设为node，然后重复执行1。
2. 如果当前结点的左孩子node为空，则取栈顶元素并执行出栈操作，访问该结点，并且将node设为该结点的右孩子。
3. 若node为NULL且栈为空，遍历结束。

该操作递归版本的时间复杂度是O(n)，空间复杂度是O(logn)。非递归版本的时间复杂度是O(n)，空间复杂度是O(n)。

PostOrderTraverse：

该操作接收一个Tree指针和一个Visit函数指针，按照后序遍历给定的二叉树。

根据宏RECURSIVE\_TRAVERSE的不同值，编译器将根据预编译指令执行不同的编译过程，因此，该函数具有递归和非递归两个版本。

递归版本首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则执行递归例程后序遍历整棵二叉树。

递归例程接收一个Node指针和一个Visit函数指针，依次对当前结点的左孩子和右孩子调用子例程，最后对当前结点执行Visit函数。

非递归版本首先检查二叉树是否存在以及函数指针是否为空，如果存在且非空，则继续判断根节点是否存在，如果存在，则新建两个栈。之后以如下的方式遍历：

1. 将根结点压入栈1
2. 取栈1顶元素并且执行出栈操作，将该结点压入栈2，之后再将该结点非空的孩子压入栈1。
3. 重复执行2直到栈1为空。
4. 将栈2的元素依次取出并且出栈，并执行访问操作，直到栈2为空。

该操作递归版本的时间复杂度是O(n)，空间复杂度是O(logn)。非递归版本的时间复杂度是O(n)，空间复杂度是O(n)。

## 3.3文件存储实现

FileWrite：

该操作接收一个Tree指针以及一个指向文件名的字符指针，实现将二叉树存储进文件的功能。

该操作首先检查二叉树是否存在以及文件是否能够被打开／创建，如果非空且文件可以正常打开，则调用递归例程按照先序将二叉树存入文件中，最终关闭文件。

递归例程接收一个Node指针，一个FILE指针，以及一个size\_t变量，该例程首先检查Node指针所指向的结点是否为空，是的话则将’n’写入文件中，否的话则将’y’以及该Node指针所指向的结点写入文件中，然后对该结点的所有孩子调用子例程。

该操作的时间复杂度是O(n)，空间复杂度是O(logn)。

FileRead：

该操作接收一个Tree \*指针、一个指向文件名的字符指针，实现从文件中先序读取并构建二叉树的功能。

该操作首先检查指针是否为空、二叉树是否存在以及文件操作是否正常，如果指针非空，二叉树存在且文件操作正常则调用递归例程依次从文件中读取数据元素并构造二叉树。

递归例程接收一个Node \*指针，一个FILE指针和一个size\_t变量，该递归例程首先从文件中读取一个char变量，如果为’n’，则直接将Node \*指针所指向的指针置NULL；如果为’y’，则再从文件中读取数据元素，并创建结点，然后对于该结点的所有孩子的地址调用子例程。

该操作的时间复杂度是O(n)，空间复杂度是O(logn)。

## 3.4配套演示系统实现

外部函数实现

visit：

该操作接收一个void一级指针，指向特定类型的数据元素，将其输出。

在该实验的演示系统中，该操作将Contact数据结构的三个成员变量name，age，tel的值依次打印输出。

construct：

该操作接收一个void二级指针，视情况选择构造一个数据元素，并且存入该二级指针所指向的内存中。

该操作首先检查用户是否输入’y’，如果是的话则从标准输入流依次读入name，age和tel，并且以动态分配的方式创建数据元素，并且将其存入void二级指针所指向的内存中。

演示功能实现

该系统采用简单文本界面，实现的过程相对而言简单——首先输出系统菜单，然后从输入流中读入一个数字，在根据数字所对应的功能执行相应的代码。

1. **模块测试**

## 4.1配套演示系统操作介绍

演示系统的界面如下图所示：



该演示系统操作简单，输入对应的数字并按下回车即可，输入1-23表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入0退出系统。

## 4.2测试方案

测试数据应兼顾包括有效数据无效数据两个方面，有效数据旨在验证系统功能的正确性，而无效甚至是错误的数据旨在验证系统的鲁棒性。根据以上的要求，拟定了以下一个测试方案：

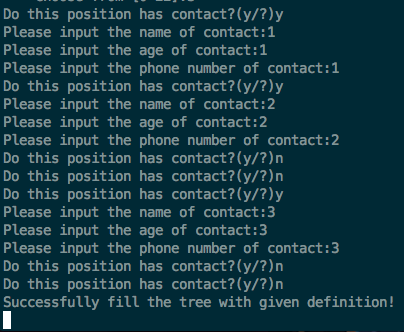
1. 初始化一个二叉树。
2. 构建一棵有3个元素的满二叉树。
3. 测试该树是否为空以及该树的深度。
4. 先序以及后序遍历该二叉树。
5. 将该二叉树写入文件。
6. 创建一棵新的二叉树，该二叉树只有两个元素，且根结点的右孩子为空。
7. 切换回原二叉树，将当前结点移动至根结点。
8. 打印根结点的值。
9. 将第二课树插入进第一棵树的右子树。
10. 测试该树的深度。
11. 层序遍历该二叉树。
12. 销毁第一棵二叉树。
13. 从文件中加载第一棵二叉树。
14. 中序遍历该二叉树。

## 4.3测试结果

1. 二叉树初始化成功

../../../../屏幕快照%202016-11-10%20下午11.04.09.png

1. 二叉树构建成功

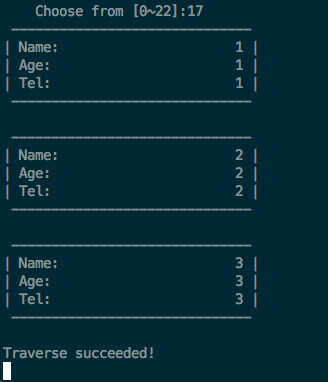


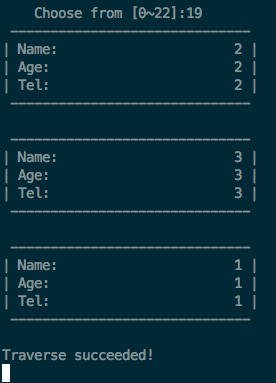
1. 二叉树非空且深度为2

../../../../屏幕快照%202016-11-10%20下午11.06.32.png

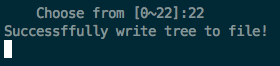
../../../../屏幕快照%202016-11-10%20下午11.06.41.png

1. 先序遍历以及后序遍历成功

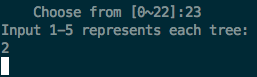




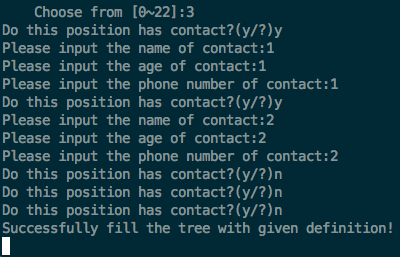
1. 文件写入成功



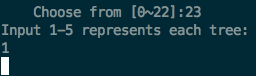
1. 成功切换至第二棵树，初始化并且构建



../../../../屏幕快照%202016-11-10%20下午11.08.02.png



1. 成功切换回第一棵树并且设置当前结点为根节点

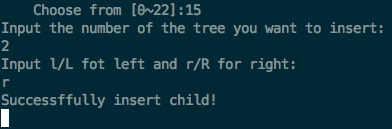


../../../../屏幕快照%202016-11-10%20下午11.08.41.png

1. 成功打印根结点的值

../../../../屏幕快照%202016-11-10%20下午11.08.51.png

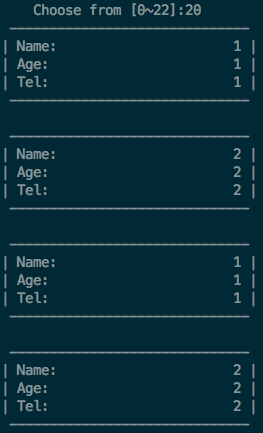
1. 插入操作成功

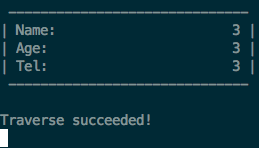


1. 该树的深度为3

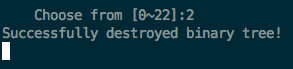
../../../../屏幕快照%202016-11-10%20下午11.10.09.png

1. 层序遍历成功





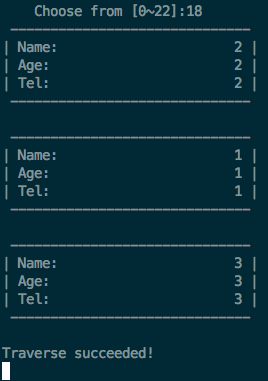
1. 第一棵树销毁成功



1. 从文件中加载成功

../../../../屏幕快照%202016-11-10%20下午11.10.56.png

1. 中序遍历成功



1. **实验总结**

通过本次实验，我加深了对于基于二叉链表的二叉树的理解。

在本次实验中，我根据C的void \*可以指向任意数据类型的特点，实现了一个简单的泛型编程，在保留上次实验的代码风格、编程规范以及低耦合度高模块化之外，我修改了上次编程中一些没有注意到的一些编程缺陷，比如，在这次的实验中，我加入了对于空指针的检查，使得代码变得更加安全。

此外，由于这次的二叉树的遍历操作需要使用递归，我将递归分区驱动例程以及相应的递归例程，由驱动例程负责进行错误检查以及第一次递归的调用，递归例程执行相应的递归操作，增加了代码的可读性。

此外，我的递归例程使用了static保留字，保证了该递归例程只能够在本文件内调用，保证了该模块的安全性。

此外，在这次的实验中，我使用宏定义以及预编译指令成功地实现了前中后序遍历的递归方法以及非递归方法，加深了我对于二叉树操作的理解，也使得我的二叉树更加的全面和完整。

**基于邻接表结构的图实现**

1. **问题描述**

一个图由顶点集V和边集E组成，每条边是一个点对（v，w），其中v，w属于V，如果点对是有序的，则图称为有向图，反之称为无向图。网，是边带有权值的图。邻接表是对于每一个顶点，都有一个表保存所有邻接的顶点，而将边的权保存在表示邻接顶点的元素中的一种图的表示方法。本实验要求封装一个基于邻接表结构的图ADT模块，提供图ADT基本的、常见的13种操作，并且为该二叉树ADT提供文件的I/O方法，且该实现方案要具有良好的鲁棒性以及可移植性，并配套简单的演示系统。

1. **模块设计**

## 2.1总体架构

根据实验的要求以及C语言的相关规范标准，给出整个模块的总体架构：

模块包括一个图头文件（.h）和一个图实现文件（.c）

模块还包括了一个队列ADT模块，以实现广度优先搜索（BFS）

模块的宏定义define、枚举定义enum、结构定义、结构声明、函数指针以及相应的别名typedef、函数原型被包括在头文件中

模块的ADT操作实现以及文件操作实现（函数）以及相关的内部静态例程被包括在实现文件中

## 2.2数据结构设计

数据结构的定义

根据实验的要求以及图ADT的定义，给出基于C语言实现的图ADT的结构定义和别名：

**struct** ALGraph {

VexNode vertices[MAX\_VERTEX\_NUMBER];

**int** vexNum, arcNum;

Kind kind;

};

**typedef** **struct** ALGraph \* ALGraph;

其中vertices是一个元素为VexNode的线性表，维护了该图的所有顶点的信息，由于VexNode是结构变量的指针，所以执行的是引用拷贝，不会浪费空间。vexNum和arcNum是该图的顶点数和弧数。kind为该图的类型，包括有向图，无向图，有向网和无向网。

Kind的枚举定义如下：

**typedef** **enum** Kind {DirectedGraph, DirectedNet, UndirectedGraph, UndirectedNet}Kind;

然后给出顶点VexNode的结构定义和别名：

**struct** VexNode {

VertexType data;

ArcNode firstArc;

};

**typedef** **struct** VexNode \* VexNode;

其中VertexType是基于typedef的C范型实现，ArcNode是弧结点，为一链表。

最后给出弧结点的结构定义和别名：

**struct** ArcNode {

**int** adjVex;

**struct** ArcNode \* nextArc;

**int** weight;

};

**typedef** **struct** ArcNode \* ArcNode;

其中，adjVex为该弧的邻接顶点的位置信息，nextArc为指向下一个弧结点的指针，weight表明该弧的权值。

此外，为了操作的统一性以及模块的封装性，创建图时需要由外部提供图的种类，图的顶点数和边数以及图的顶点集和边集，因此，给出顶点集的别名如下：

**typedef** VertexType \* VexSet;

然后给出边集的结构定义和别名：

**struct** ArcSet {

**int** headPos;

**int** tailPos;

**int** weight;

};

**typedef** **struct** ArcSet \* ArcSet;

其中，headPos，tailPos和weight分别表示弧头和弧尾的位置信息以及弧的权值。

在本次的实验中，采用了如下的结构作为数据元素：

**typedef** **struct** Contact {

**char** name[20];

**int** age;

**char** tel[12];

}Contact;

该结构表示一个用户通讯录，其中，name表示的是联系人的姓名，age表示的是联系人的年龄，tel表示的是联系人的电话。

为了实现范型，还需要如下的结构别名：

**typedef** Contact VertexType;

## 2.3 ADT操作设计

根据实验的要求以及图ADT的定义，该图ADT应包括如下的操作：

CreateGraph（创建），DestroyGraph（销毁），LocateVex（返回顶点位置信息），GetVex（去顶点值），PutVex（设顶点值），FirstAdjVex（求顶点的第一邻接点），NextAdjVex（求顶点相对某顶点的下一邻接点），InsertVex（插入顶点），DeleteVex（删除顶点），InsertArc（插入弧），DeleteArc（删除弧），DFSTraverse（深度优先遍历），BFSTraverse（广度优先遍历）。

根据相关ADT操作的特性，给出如下基于C语言的相关定义。

为了ADT操作统一性以及错误处理，在C语言中给出如下的通用函数原型：

Status MethodName (Typename1 arg1, Typename2 arg2, ...);

其中，Status表示为预定义的枚举类型，用来表明函数执行的结果以及错误的处理，在C语言中的定义如下：

**typedef** **enum** Status {OK, NULL\_POINTER = 100, GRAPH\_NOT\_EXIST, MEMORY\_OVERFLOW, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

OK表明函数执行成功，状态正常；NULL\_POINTER表明函数的参数中带有非法的空指针；GRAPH\_NOT\_EXIST表明操作的图还未初始化，不存在；MEMORY\_OVERFLOW表明内存已满，分配空间失败； EXTERNAL\_FUNCTION\_ERROR表明函数调用的外部函数错误；FILE\_ERROR表明文件操作失败。

为了存储判断的结果，还需要预定义一个枚举类型Bool，在C语言中的定义如下：

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

FALSE为假，TRUE为真

为了模块的扩展性，Traverse所用到的访问函数Visit由外部提供，使用函数指针作为参数，在C语言中的定义如下：

**typedef** Status (\* Visit)(VertexType elem);

基于以上的定义，给出所有ADT操作在C语言中对应得函数原型：

Status CreateGraph(ALGraph \* pGraph, Kind kind, VexSet vexSet, **int** vexNum, ArcSet arcSet, **int** arcNum);

Status DestroyGraph(ALGraph \* pGraph);

Status LocateVex(ALGraph graph, VexNode node, **int** \* pPos);

Status GetVex(ALGraph graph, VexNode node, VertexType \* pElem);

Status PutVex(ALGraph graph, VexNode node, VertexType elem);

Status FirstAdjVex(ALGraph graph, VexNode node, VexNode \* pNode);

Status NextAdjVex(ALGraph graph, VexNode curNode, VexNode adjNode, VexNode \* pNextAdjNode);

Status InsertVex(ALGraph graph, VexNode node);

Status DeleteVex(ALGraph graph, VexNode node);

Status InsertArc(ALGraph graph, VexNode curNode, VexNode adjNode, **int** weight);

Status DeleteArc(ALGraph graph, VexNode curNode, VexNode adjNode);

Status DFSTraverse(ALGraph graph, Visit visit);

Status BFSTraverse(ALGraph graph, Visit visit);

## 2.4文件存储设计

根据实验的要求以及图ADT的数据结构，该图ADT应包括如下的文件操作：

FileWrite（将图写入文件）、FileRead（从文件中读取图）

根据ADT操作设计中的定义，给出文件存储操作在C语言中对应的函数原型：

Status FileWrite(ALGraph graph, **char** \* filename);

Status FileRead(ALGraph \* pGraph, **char** \* filename);

## 2.5配套演示系统设计

配套演示系统采用文本菜单界面，该系统每次从输入流接受一个数字，并且根据数字的不同执行不同的操作。该系统还应该实现模块中所要求实现的visit操作。此外，在本次实验中，使用宏定义的方式为数据元素创建了别名，实现了范型，同第二次和第三次实验不同。

1. **模块实现**

## 3.1开发环境

开发本系统时的操作系统环境为macOS Sierra(10.12)。

编程环境

编程时没有使用IDE（集成开发环境）。

编程时使用了文本编辑器Atom进行了代码的编写。

编程时采用Apple LLVM编译器进行编译。版本为8.0.0(clang-800.0.38)。

运行环境

在调试时使用的操作系统环境为macOS Sierra(10.12)。

## 3.2 ADT操作实现

CreateGraph：

该操作接收一个ALGraph \*指针、一个Kind变量、一个VexSet指针，一个表明顶点数目的int变量、一个ArcSet指针和一个表明弧数目的int变量，以传入的所有参数创建一张图。

该操作首先检查指针是否为空，非空的话则通过动态分配空间的方式分配一个Graph结构变量，根据顶点集VexSet和顶点数目依次为每一个顶点动态分配存储空间并设置顶点的数据域和置NULL弧指针。

然后该操作按循环依次创建每一条弧，具体来说，首先动态为弧结点分配存储空间，然后按照边集为弧设置邻接结点的位置adjVex的信息，然后检查图的类型，如果为网，则为该弧初始化权值，最后获取相应的顶点，依顶点的弧结点域是否为空将弧结点插入。然后再检查图的类型，如果为无向，则类似地再创建一条反向的弧。

该操作的时间复杂度O(n+e)，空间复杂度O(1)。

DestroyGraph：

该操作接收一个ALGraph \*指针，将指针所指向的图销毁。

该操作首先检查指针以及图是否存在，非空且存在的话则会先按照循环依次释放所有的弧结点所动态分配的空间，再循环依次释放所有顶点所动态分配的空间。最后释放整个ALGraph结构，并将其置NULL。

该操作的时间复杂度为O(n+e)，空间复杂度为O(1)。

LocateVex：

该操作接收一个ALGraph指针，一个VexNode指针和一个int指针，获取VexNode所对应结点再图中的位置信息。

该操作首先检查图是否存在以及指针是否非空，存在且非空的话则循环遍历图的所有顶点，如果顶点与传入的顶点相同，则返回将位置信息存入int指针所指向的变量中。如果不存在，则存入-1。

该操作的时间复杂度为O(n)，空间复杂度为O(1)。

GetVex：

该操作接收一个ALGraph指针，一个VexNode指针和一个VertexType指针，将VexNode所对应顶点的数据域存入VertexType指针所指向的变量中。

该操作首先检查图是否存在以及指针是否为空，如果存在且非空则对VexNode调用LocateVex函数，再确定VexNode所对应的顶点在图中后，直接将VexNode的数据域存入VertexType指针所指的变量中。

该操作的时间复杂服为O(n)，空间复杂度为O(1)。

PutVex：

该操作接收一个ALGraph指针、一个VexNode指针和一个VertexType变量，将VertexType变量的值设置为VexNode的数据域。

该操作首先检查图是否存在以及指针是否非空，如果存在且非空则对VexNode调用LocateVex函数，再确定VexNode所对应的顶点在图中后，直接将VexNode的数据域设置为VertexType变量。

该操作的时间复杂度是O(n)，空间复杂度为O(1)。

FirstAdjVex：

该操作接收一个ALGraph指针、一个VexNode指针和一个VexNode \*指针，将VexNode所对应顶点的第一个邻接顶点存入VexNode \*指针所指向的VexNode指针中。

该操作首先判断图是否存在以及指针是否为空，如果存在且非空的话，则首先对VexNode指针调用LocateVex函数，再确定VexNode所对应的顶点在图中后，直接将vexNode的指针域的第一条弧所对应的VexNode指针存入VexNode \*指针所指向的指针中，否则存入NULL。

该操作的时间复杂度是O(n)，空间复杂度是O(1)。

NextAdjVex：

该操作接收一个ALGraph指针、两个VexNode指针，其中一个表示顶点，另一个表示该顶点的一个邻接顶点和一个VexNode \*指针，将顶点对应于该邻接顶点的下一个邻接顶点存入VexNode \*指针所指向的指针中。

该操作首先判断图是否存在以及指针是否非空，如果存在且非空的话则首先对表示顶点的VexNode指针调用LocateVex函数，再确定VexNode所对应的顶点在图中后，依次遍历当前顶点的指针域中的所有弧结点，找到当前邻接顶点后将其下一弧结点所对应的邻接顶点存入VexNode \*指针所对应的指针中，否则存入NULL。

该操作的时间复杂度是O(n+e)，空间复杂度O(1)。

InsertVex：

该操作接收一个ALGraph指针和一个VexNode指针，将该顶点加入图中。

该操作首先检查图是否存在以及指针是否为空，存在且非空的话则为顶点动态分配空间，在直接将VexNode赋值并且将图的顶点数自增。

该操作的时间复杂度是O(1)，空间复杂度是O(1)。

DeleteVex：

该操作接收一个ALGraph指针和一个指针，将VexNode指针所对应的顶点从图中删除。

该操作首先检查图是否存在以及指针是否非空，如果存在且非空则对VexNode调用LocateVex函数，再确定VexNode所对应的顶点在图中后，依次遍历所有的顶点并释放顶点的的数据域中的所有的和待删除顶点有关的弧结点，最后再释放VexNode本身并且将其从顺序表中删除。

该操作的时间复杂度是O(n+e)，空间复杂度是O(1)。

InsertArc：

该操作接收一个ALGraph指针、两个VexNode指针——一个表示当前的顶点另一个表示和当前顶点邻接的顶点和一个int变量，将连接两个顶点的弧插入图中，如果该图为网，则权值由int变量提供。

该操作首先判断图是否存在以及指针是否为空，如果存在且非空则对表示邻接顶点的vexNode调用LocateVex函数，然后遍历当前顶点的指针域所指向的弧链表，如果找到相同的弧则直接返回，否则动态分配一个弧结点，配置其邻接顶点和指针域，在根据图是否为网来配置其权值，最后将其插入顶点的指针域所指向的弧链表中，然后判断图是否为无向，是的话则再动态分配一个弧结点插入。

该操作的时间复杂度是O(e)，空间复杂度是O(1)。

DeleteArc：

该操作接收一个ALGraph指针和两个VexNode指针——一个表示当前的顶点另一个表示和当前顶点邻接的顶点，删除链接这两个顶点的弧。

该操作首先检查图是否为存在以及指针是否非空，如果图存在且指针非空的话则对表示邻接顶点的vexNode调用LocateVex函数，然后再依次遍历表示当前顶点的指针域中的所有弧结点，如果找到了则将其释放，然后再判断图的类型，如果是无向图或网，则再找到对应的弧并释放。

该操作的时间复杂度是O(e)，空间复杂度是O(1)。

DFSRTraverse：

该操作接收一个ALGraph指针和一个Visit函数指针，深度优先遍历全图。

该操作首先判断图是否为存在以及指针是否非空，如果存在且非空则首先创建标志数组visited并置0，然后循环遍历该图的全部顶点，对没有访问过的顶点调用递归例程依次遍历全图。

该递归例程接收一个ALGraph指针、一个VexNode指针、一个Visit函数指针和标志数组visited，该例程检查VexNode所对应顶点的所有邻接顶点，如果没有被访问过，则访问并对其调用递归例程。

该操作的时间复杂度是O(n+e)，空间复杂度是O(n)。

BFSTraverse：

该操作接收一个ALGraph指针和一个Visit函数指针，广度优先遍历全图。

该操作首先检查图是否存在以及指针是否非空，如果存在且非空则首先建立标志数组visited并置0，然后初始化一个队列并且循环遍历图的所有顶点，在每一次循环中先判断该顶点是否被访问过，若没有被访问过则访问并将其入队，然后以队列不为空为条件循环，首先将元素出队，遍历其所有邻接顶点，若未被访问则访问并且将其入队，依此循环。

该操作的时间复杂度是O(n+e)，空间复杂度是O(n)。

## 3.3文件存储实现

FileWrite：

该操作接收一个ALGraph指针以及一个指向文件名的字符指针，实现将图存储进文件的功能。

该操作首先检查图是否存在以及文件是否能够被打开／创建，如果存在且文件可以正常打开，则依次将图的类型，图的顶点数，图的弧数写入文件中，然后重新根据图的内容生成顶点集VexSet和边集ArcSet，并且依次将它们写入文件，最后关闭文件。

该操作的时间复杂度是O(n+e)，空间复杂度是O(n+e)。

FileRead：

该操作接收一个ALGraph \*指针、一个指向文件名的字符指针，实现从文件中读取图的功能。

该操作首先检查指针是否为空以及文件操作是否正常，如果指针非空且文件操作正常则从文件中一次读取图的种类、图的顶点数、图的边数、图的顶点集和图的边集，首先执行销毁图的操作，然后以上面的参数执行生成图操作即可，最后关闭文件。

该操作的时间复杂度是O(n+e)，空间复杂度是O(n+e)。

## 3.4配套演示系统实现

外部函数实现

visit：

该操作接收一个Contact结构变量，将其输出。

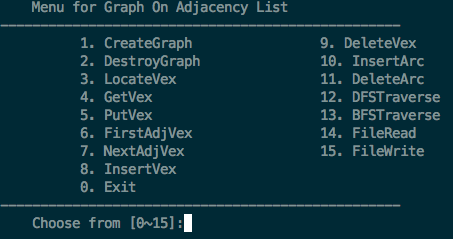
在该实验的演示系统中，该操作将Contact数据结构的三个成员变量name，age，tel的值依次打印输出。

演示功能实现

该系统采用简单文本界面，实现的过程相对而言简单——首先输出系统菜单，然后从输入流中读入一个数字，在根据数字所对应的功能执行相应的代码即可。

1. **模块测试**

## 4.1配套演示系统操作介绍

演示系统的界面如下图所示：

该演示系统操作简单，输入对应的数字并按下回车即可，输入1-15表明运行对应的功能，且相应操作的结果会被输出到屏幕上，输入0退出系统。

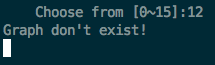
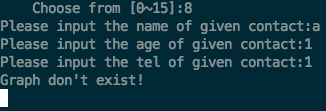
## 4.2测试方案

测试数据应兼顾包括有效数据无效数据两个方面，有效数据旨在验证系统功能的正确性，而无效甚至是错误的数据旨在验证系统的鲁棒性。根据以上的要求，拟定了以下一个测试方案：

1. 在未创建图的情况下执行销毁图，插入结点、深度优先遍历和文件读取
2. 创建一个有3个顶点的无向图，有3条边（0，1），（0，2）和（1，2）
3. 确定下标为1的顶点的位置信息（比较是否正确）
4. 确定其第一和第二邻接顶点
5. 取得下标为1的顶点的值
6. 修改下标为1的顶点的值
7. 插入一个新的顶点和一条新的弧（2，3）
8. 广度优先遍历该图
9. 保存该图
10. 关闭并重新打开后读取该图
11. 深度优先遍历该图
12. 删除下标为0的顶点
13. 广度优先遍历该图

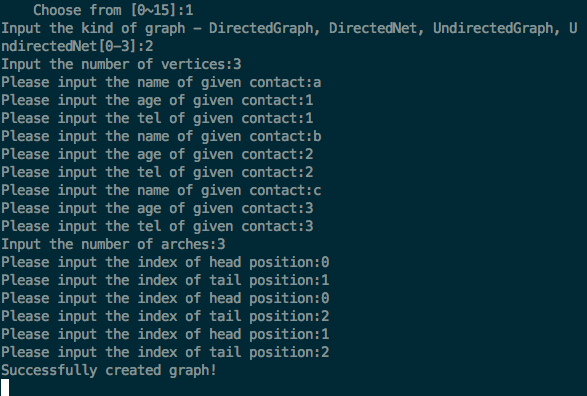
## 4.3测试结果

1. 图不存在，提示错误

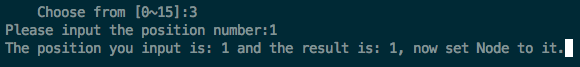
../../../../屏幕快照%202016-12-08%20下午7.37.49.png

../../../../屏幕快照%202016-12-08%20下午7.38.14.png

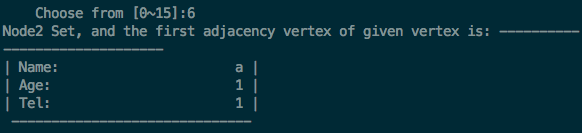
1. 创建成功

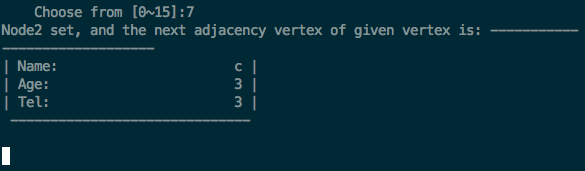


1. 正确

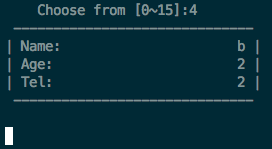


1. 第一和第二邻接点正确

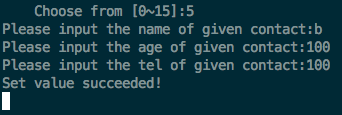




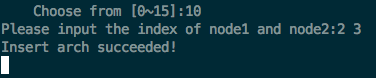
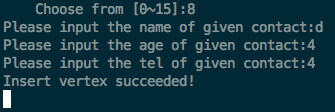
1. 取得了下标为1的顶点的值



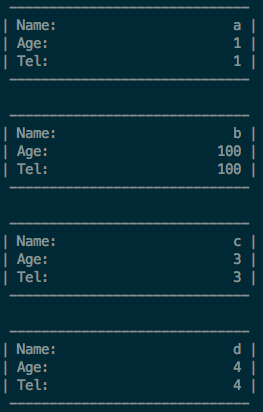
1. 设置了下标为1的顶点的值



1. 插入成功



1. 遍历结果



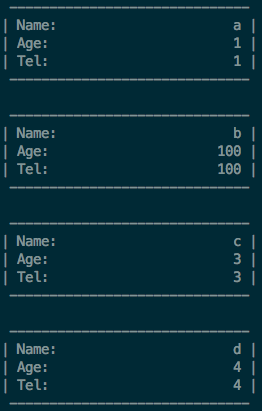
1. 保存成功

../../../../屏幕快照%202016-12-08%20下午7.40.43.png

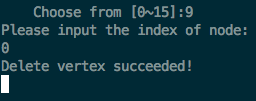
1. 读取成功

../../../../屏幕快照%202016-12-08%20下午7.41.20.png

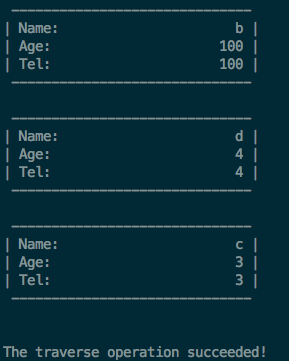
1. 遍历结果



1. 删除成功



1. 遍历结果



1. **实验总结**

通过本次实验，我加深了对于基于邻接表的图的理解。

在本次实验中，我采用了和之前两次实验不一样的方法实现范型——即基于宏定义和别名实现范型，这种方式较为简单，不需要考虑各种指针间的转换，但同时也使得程序不是那么优雅——在本次实验中，我不得不讲结构的声明从.c文件中移动到了.h文件中，而这么做显然是不好的，这失去了封装的作用，而直接将结构体的所有成员变量对外公开，这是不安全的。

此外，通过该实验，我熟悉了如何用邻接表表示图，这种表示方法相比较于邻接矩阵而言节省了空间（尤其是对于稀疏矩阵），同时也能表示有向无向等各种类型的图，具有通用性。

此外，本次实验，我将文件保存于创建图相连接，通过在保存图的时候按照创建图的格式保存数据，从而统一了函数接口，达到了封装和抽象的目的，这在我之前的实验中是没有实现的。

总之，通过了本次实验，我收获颇丰，但是也仍然存在一些不足，因为时间的原因，程序可能不尽如人意，在对于有向无向，图和网的不同处理，以及删除结点、弧以及销毁图这几个函数上，我觉得我的抽象仍然是不够的，导致了大量的代码冗余，造成了维护的困难，需要在数据结构课设以及今后的学习实践中予以避免。

**参考文献**

[1] 严蔚敏等.数据结构(C语言版). 清华大学出版社

[2] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

[3] Mark Allen Weiss. 数据结构与算法分析：C语言描述. 机械工业出版社

[4] GCC Manual

**指导教师评定意见**

一、对实验报告的评语

|  |
| --- |
|  |

二、对实验报告评分

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 评分项目  (分值) | 程序内容  (36.8分) | 程序规范  (9.2分) | 报告内容  (36.8分) | 报告规范  (9.2分) | 考勤  （8分） | 逾期扣分 | 合 计  (100分) |
| 得分 |  |  |  |  |  |  |  |

附录一 基于顺序存储结构线性表实现的源程序

Linear\_Table\_On\_Sequence\_Structure.h

/\* Linear Table on Sequence Structure \*/

#ifndef LINEAR\_TABLE\_ON\_SEQUENCE\_STRUCTURE\_H

#define LINEAR\_TABLE\_ON\_SEQUENCE\_STRUCTURE\_H

/\* Include Necessary Standard Library \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Macro Definition of Structure \*/

#define LIST\_INIT\_SIZE 100

#define LIST\_INCREMENT 10

/\* Definition and Typedef of Data and Structure and typedef of Function Pointers \*/

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

**typedef** **enum** Status {OK, NULL\_POINTER = 100, LIST\_NOT\_EXIST, MEMORY\_OVERFLOW, OUT\_OF\_RANGE, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

**struct** Element;

**typedef** **struct** Element Element;

**struct** SequenceList;

**typedef** **struct** SequenceList SequenceList;

**typedef** Bool (\*Compare)(Element elem1, Element elem2);

**typedef** Status (\*Visit)(Element elem);

/\* Declaration of Data and Structure \*/

**struct** Element {

**int** value;

};

**struct** SequenceList {

Element \* elem;

**int** length;

**int** listsize;

};

/\* Definition of Data Operations \*/

Status InitList (SequenceList\* pList);

Status DestroyList (SequenceList\* pList);

Status ClearList (SequenceList\* pList);

Status ListEmpty (SequenceList list, Bool\* pBool);

Status ListLength (SequenceList list, **int**\* pCount);

Status GetElem (SequenceList list, **int** index, Element\* pElem);

Status LocateElem (SequenceList list, Element elem1, Compare fp, **int**\* pIndex);

Status PriorElem (SequenceList list, Element curElem, Element\* pPreElem, Compare fp);

Status NextElem (SequenceList list, Element curElem, Element\* pNextElem, Compare fp);

Status ListInsert (SequenceList\* pList, **int** index, Element elem);

Status ListDelete (SequenceList\* pList, **int** index, Element\* pElem);

Status ListTraverse (SequenceList list, Visit fp);

/\* Definition of File Operations\*/

Status FileWrite(SequenceList list, **char**\* filename);

Status FileRead(SequenceList\* pList, **char**\* filename);

#endif

Linear\_Table\_On\_Sequence\_Structure.c

#include "Linear-Table-On-Sequence-Structure.h"

/\* Implementation of Operations \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InitList

Description: Init a list.

Input: pList that represents the pointer of the list

structure variable.

Return: Status variable that represents the state of

this operaiton.

Others: Won't throw exception if using one variable

to create two or more list without freeing.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InitList (SequenceList\* pList) {

**if** (!pList)

**return** NULL\_POINTER;

pList->elem = **malloc**(LIST\_INIT\_SIZE \* **sizeof**(Element));

**if** (!pList->elem)

**return** MEMORY\_OVERFLOW;

pList->length = 0;

pList->listsize = LIST\_INIT\_SIZE;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DestroyList

Description: Destroy a list as well as freeing the

memory.

Input: pList that represents the pointer of the list

structure variable.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DestroyList (SequenceList\* pList) {

**if** (!pList)

**return** NULL\_POINTER;

**if** (!pList->elem)

**return** LIST\_NOT\_EXIST;

**free**(pList->elem);

pList->elem = **NULL**;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ClearList

Description: Clear a list.

Input: pList that represents the pointer of the list

structure variable.

Return: Status variable that represents the state of

this operaiton.

Others: Lazy clear.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ClearList (SequenceList\* pList) {

**if** (!pList)

**return** NULL\_POINTER;

**if** (!pList->elem)

**return** LIST\_NOT\_EXIST;

pList->length = 0;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListEmpty

Description: Decide if a given list is empty

Input: list that represents the list structure variable.

Return: Bool variable that represents if the list is

empty.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListEmpty (SequenceList list, Bool\* pBool) {

**if** (!list.elem) {

**return** LIST\_NOT\_EXIST;

}

**if** (!list.length)

\*pBool = TRUE;

**else**

\*pBool = FALSE;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListLength

Description: Count the length of the given list.

Input:

list that represents the list structure variable.

pCount that represents the pointer to the int

variable.//return value

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListLength (SequenceList list, **int**\* pCount) {

**if** (!list.elem)

**return** LIST\_NOT\_EXIST;

\*pCount = list.length;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: GetElem

Description: Get the element of given index.

Input:

list that represents the list structure variable.

index that represents the index of the list.

pElem that represents the pointer to the element.//return value

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status GetElem (SequenceList list, **int** index, Element\* pElem) {

**if** (!list.elem)

**return** LIST\_NOT\_EXIST;

**if** (index < 1 || index > list.length)

**return** OUT\_OF\_RANGE;

\*pElem = list.elem[index - 1];

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: LocateElem

Description: Locate the element of given element and

given compare function.

Input:

list that represents the list structure variable.

elem1 that represents the given element.

fp that represents the pointer to the compare function.

pIndex that represents the pointer to the index variable.//return value.

Return: Status variable that represents the state of

this operaiton.

Others: index will be set to 0 if don't find specific

element.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status LocateElem (SequenceList list, Element elem1, Compare fp, **int**\* pIndex) {

**if** (!list.elem)

**return** LIST\_NOT\_EXIST;

**int** count;

**for** (count = 0 ; count < list.length ; count++) {

**if** (fp(elem1, list.elem[count]))

**break**;

}

\*pIndex = (count >= list.length)?0:count+1;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PriorElem

Description: Find the prior element of given element

and given compare function.

Input:

list that represents the list structure variable.

curElem that represents the given element.

pPreElem that represents the target element.//return value

fp that represents the pointer to the compare function.

Return: Status variable that represents the state of

this operaiton.

Others: \*pPreElem won't be defined if don't find

specific element.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PriorElem (SequenceList list, Element curElem, Element\* pPreElem, Compare fp) {

**if** (!list.elem)

**return** LIST\_NOT\_EXIST;

**int** count;

**for** (count = 0 ; count < list.length ; count++) {

**if** (fp(curElem, list.elem[count]))

**break**;

}

**if** (count > 0)

\*pPreElem = list.elem[count - 1];

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NextElem

Description: Find the next element of given element

and given compare function.

Input:

list that represents the list structure variable.

curElem that represents the given element.

pNextElem that represents the target element.//return value

fp that represents the pointer to the compare function.

Return: Status variable that represents the state of

this operaiton.

Others: \*pNextElem won't be defined if don't find

specific element.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NextElem (SequenceList list, Element curElem, Element\* pNextElem, Compare fp) {

**if** (!list.elem)

**return** LIST\_NOT\_EXIST;

**int** count;

**for** (count = 0 ; count < list.length ; count++) {

**if** (fp(curElem, list.elem[count]))

**break**;

}

**if** (count < list.length)

\*pNextElem = list.elem[count + 1];

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListInsert

Description: Insert one element into given list.

Input:

pList that represents the pointer of the list

structure variable.

index that represents the index of element to

be inserted.

elem that represents the element to be inserted.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListInsert (SequenceList\* pList, **int** index, Element elem) {

**if** (!pList)

**return** NULL\_POINTER;

**if** (!pList->elem)

**return** LIST\_NOT\_EXIST;

**if** (index < 1 || index > pList->length + 1)

**return** OUT\_OF\_RANGE;

**if** (pList->length >= pList->listsize) {

pList->elem = **realloc**(pList->elem, (pList->listsize + LIST\_INCREMENT) \* **sizeof**(Element));

**if** (!pList->elem)

**return** MEMORY\_OVERFLOW;

pList->listsize += LIST\_INCREMENT;

}

Element\* pos = &(pList->elem[index - 1]), \* end;

**for** (end = &(pList->elem[pList->length - 1]) ; end >= pos ; --end)

\*(end + 1) = \*(end);

\*pos = elem;

++pList->length;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListDelete

Description: Delete one element of given list.

Input:

pList that represents the pointer of the list

structure variable.

index that represents the index of element to

be deleted.

pElem that represents the pointer to the element

to be deleted.//Return value.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListDelete (SequenceList\* pList, **int** index, Element\* pElem) {

**if** (!pList)

**return** NULL\_POINTER;

**if** (!pList->elem)

**return** LIST\_NOT\_EXIST;

**if** (index < 1 || index > pList->length)

**return** OUT\_OF\_RANGE;

Element \* pos = &(pList->elem[index - 1]), \*end = &(pList->elem[pList->length - 1]);

\*pElem = \*pos;

**for** (++pos ; pos <= end ; ++pos)

\*(pos - 1) = \*(pos);

--pList->length;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListTraverse

Description: Traverse the given list with given

visit().

Input:

list that represents the list structure variable.

fp that represents the pointer to the visit function.

Return: Status variable that represents the state of

this operaiton.

Others: ListTraverse fails if visit fails.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListTraverse (SequenceList list, Visit fp) {

**if** (!list.elem)

**return** LIST\_NOT\_EXIST;

**int** count;

**for** (count = 0 ; count < list.length ; count++) {

**if** (fp(list.elem[count]))

**return** EXTERNAL\_FUNCTION\_ERROR;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileWrite

Description: Write the list to the given file.

Input:

list that represents the list structure variable.

filename that represents the name of the file.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileWrite(SequenceList list, **char**\* filename) {

**if** (!list.elem)

**return** LIST\_NOT\_EXIST;

**FILE** \* fp;

fp = **fopen**(filename, "wb");

**if** (fp == **NULL**)

**return** FILE\_ERROR;

**fwrite**(&list.length, **sizeof**(**int**), 1, fp);

**fwrite**(&list.listsize, **sizeof**(**int**), 1, fp);

**fwrite**(list.elem, **sizeof**(Element), list.length, fp);

**fclose**(fp);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileRead

Description: Read the list from the given file.

Input:

list that represents the list structure variable.

filename that represents the name of the file.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileRead(SequenceList\* pList, **char**\* filename) {

**if** (!pList)

**return** NULL\_POINTER;

**FILE** \* fp;

fp = **fopen**(filename, "rb");

**if** (fp == **NULL**)

**return** FILE\_ERROR;

**int** length, listsize;

**if** (!**feof**(fp))

**fread**(&length, **sizeof**(**int**), 1, fp);

**if** (!**feof**(fp))

**fread**(&listsize, **sizeof**(**int**), 1, fp);

DestroyList(pList);

pList->elem = **malloc**(listsize \* **sizeof**(Element));

**if** (!pList->elem)

**return** MEMORY\_OVERFLOW;

pList->length = length;

pList->listsize = listsize;

**if** (!**feof**(fp))

**fread**(pList->elem, **sizeof**(Element), length, fp);

**fclose**(fp);

**return** OK;

}

Display.c

#include <stdlib.h>

#include "Linear-Table-On-Sequence-Structure.h"

**char** \* data\_file\_name = "sequence\_list\_display.dat";

Bool compare(Element elem1, Element elem2) {

**if** (elem1.value == elem2.value)

**return** TRUE;

**else**

**return** FALSE;

}

Status visit(Element elem) {

**printf**("%10d", elem.value);

**return** OK;

}

**int** **main**(**void**) {

**int** op=1;

SequenceList list;

**char** ch;

**int** count, index;

Status result;

Bool empty;

Element elem, elem2;

**while**(op){

**system**("clear");

**printf**(" Menu for Linear Table On Sequence Structure \n");

**printf**("--------------------------------------------------\n");

**printf**(" 1. InitList 8. PriorElem\n");

**printf**(" 2. DestroyList 9. NextElem\n");

**printf**(" 3. ClearList 10. ListInsert\n");

**printf**(" 4. ListEmpty 11. ListDelete\n");

**printf**(" 5. ListLength 12. ListTraverse\n");

**printf**(" 6. GetElem 13. FileRead\n");

**printf**(" 7. LocateElem 14. FileWrite\n");

**printf**(" 0. Exit\n");

**printf**("--------------------------------------------------\n");

**printf**(" Choose from [0~14]:");

**scanf**("%d",&op);

**getchar**();

**switch**(op){

**case 1:**

result = InitList(&list);

**if**(result == OK)

**printf**("Successfully created sequence list!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Failed to allocate memory for list!\n");

**break**;

**case 2:**

result = DestroyList(&list);

**if** (result == OK)

**printf**("Successfully destroyed sequence list!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("List don't exist!\n");

**break**;

**case 3:**

result = ClearList(&list);

**if** (result == OK)

**printf**("Successfully cleared sequence list!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("List don't exist!\n"); **break**;

**case 4:**

**if** (ListEmpty(list, &empty) == OK) {

**if** (empty == TRUE)

**printf**("The list is empty!\n");

**else**

**printf**("The list is NOT empty!\n");

} **else**

**printf**("List don't exist!\n");

**break**;

**case 5:**

**if** (ListLength(list, &count) == OK)

**printf**("The length of the list is: %d\n", count);

**else**

**printf**("List don't exist!\n");

**break**;

**case 6:**

**printf**("Please input the index:");

**scanf**("%d", &index);

**getchar**();

result = GetElem(list, index, &elem2);

**if** (result == OK)

**printf**("The element of index %d has value %d\n", index, elem2.value);

**else** **if** (result == LIST\_NOT\_EXIST)

**printf**("List don't exist!\n");

**else**

**printf**("The index is out of range!\n");

**break**;

**case 7:**

**printf**("Please input the value of given element:");

**scanf**("%d", &elem2.value);

**getchar**();

**if** (LocateElem(list, elem2, &compare, &index) == OK) {

**if** (index == 0) {

**printf**("There is no element that matches given element by compare.\n");

} **else** {

**printf**("The index of the first element that matches given element by compare is %d\n", index);

}

} **else** {

**printf**("List don't exist!\n");

}

**break**;

**case 8:**

**printf**("Please input the value of current element:");

**scanf**("%d", &elem2.value);

**getchar**();

**if** (PriorElem(list, elem2, &elem, &compare) == OK) {

**printf**("The prior element of current element is %d\n", elem.value);

} **else** {

**printf**("List don't exist!\n");

}

**break**;

**case 9:**

**printf**("Please input the value of current element:");

**scanf**("%d", &elem2.value);

**getchar**();

**if** (NextElem(list, elem2, &elem, &compare) == OK) {

**printf**("The next element of current element is %d\n", elem.value);

} **else** {

**printf**("List don't exist!\n");

}

**break**;

**case 10:**

**printf**("Please input the index and the value of new element:");

**scanf**("%d%d", &index, &elem2.value);

**getchar**();

result = ListInsert(&list, index, elem2);

**if** (result == OK)

**printf**("The insert operation succeeded!\n");

**else** **if** (result == OUT\_OF\_RANGE)

**printf**("The index is out of range!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else** **if** (result == LIST\_NOT\_EXIST)

**printf**("List don't exist!\n");

**else**

**printf**("Failed to allocate memory for new element!\n");

**break**;

**case 11:**

**printf**("Please input the index of the element to be deleted:");

**scanf**("%d", &index);

**getchar**();

result = ListDelete(&list, index, &elem2);

**if** (result == OK)

**printf**("The delete operation succeeded.The value of the deleted element is %d\n", elem2.value);

**else** **if** (result == OUT\_OF\_RANGE)

**printf**("The index is out of range!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("List don't exist!\n");

**break**;

**case 12:**

**if** (ListTraverse(list, &visit) == OK) {

**printf**("\n");

**printf**("The traverse operation succeeded!\n");

}

**else**

**printf**("The list don't exist!\n");

**break**;

**case 13:**

result = FileRead(&list, data\_file\_name);

**if** (result == OK)

**printf**("Data load succeeded!\n");

**else** **if** (result == FILE\_ERROR)

**printf**("Fail to open data file for reading!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Fail to allocate memory for list!\n");

**break**;

**case 14:**

result = FileWrite(list, data\_file\_name);

**if** (result == OK)

**printf**("Data save succeeded!\n");

**else** **if** (result == FILE\_ERROR)

**printf**("Fail to open data file for writing!\n");

**else**

**printf**("You should NOT write a list that DON'T exist!\n");

**break**;

**case 0:**

op = 0;

**break**;

}

**getchar**();

}

**printf**("See you next time!\n");

**return** 0;

}

附录二 基于链式存储结构线性表实现的源程序

Linear\_Table\_On\_Linked\_Structure.h

/\* Linear Table on Link Structure \*/

#ifndef LINEAR\_TABLE\_ON\_LINKED\_STRUCTURE\_H

#define LINEAR\_TABLE\_ON\_LINKED\_STRUCTURE\_H

/\* Include Necessary Library \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Typedef of Enumeration \*/

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

**typedef** **enum** Status {OK, NULL\_POINTER = 100, LIST\_NOT\_EXIST, MEMORY\_OVERFLOW, OUT\_OF\_RANGE, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

/\* Definition and Typedef of Data and Structure and typedef of Function Pointers \*/

**struct** ListNode ;

**typedef** **struct** ListNode \* Node;

**struct** List;

**typedef** **struct** List \* List;

/\* Required Function Pointers \*/

**typedef** Bool (\*Compare)(**void**\* elem1, **void**\* elem2);

**typedef** Status (\*Visit)(**void**\* elem);

/\* Optional Function Pointers\*/

**typedef** Status (\*Assign)(**void**\*, **const** **void**\*); //Include malloc and memcpy

**typedef** Status (\*Dealloc)(**void**\*); //Include free

/\* Definition of Data Operations \*/

Status InitList (List\* pList, **size\_t** elem\_size);

Status DestroyList (List\* pList, Dealloc fp);

Status ClearList (List list, Dealloc fp);

Status ListEmpty (List list, Bool\* pBool);

Status ListLength (List list, **int**\* pCount);

Status GetElem (List list, **int** index, **void**\* pElem);

Status LocateElem (List list, **void**\* elem, Compare fp, **int**\* pIndex);

Status PriorElem (List list, **void**\* curElem, **void**\* pPreElem, Compare fp);

Status NextElem (List list, **void**\* curElem, **void**\* pNextElem, Compare fp);

Status ListInsert (List list, **int** index, **void**\* elem, Assign fp);

Status ListDelete (List list, **int** index, **void**\* elem, Dealloc fp, Assign fp2);

Status ListTraverse (List list, Visit fp);

/\* Definition of File Operations\*/

Status FileWrite (List list, **char**\* filename);

Status FileRead (List\* pList, **char**\* filename, Dealloc fp);

#endif

Linear\_Table\_On\_Linked\_Structure.c

#include "Linear-Table-On-Linked-Structure.h"

/\* Declaration of Data and Structure \*/

struct List {

int length;

size\_t elem\_size;

struct ListNode \* head;

};

struct ListNode {

void \* elem;

struct ListNode \* next;

};

/\* Implementation of Operations \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InitList

Description: Init a list.

Input:

pList that represents the pointer of the list

structure pointer.

elem\_size that represents the size of the specific

structure.

Return: Status variable that represents the state of

this operaiton.

Others: Won't throw exception if using one variable

to create two or more list without freeing.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InitList (List\* pList, size\_t elem\_size) {

if (!pList)

return NULL\_POINTER;

\*pList = malloc(sizeof(struct List));

if (!\*pList)

return MEMORY\_OVERFLOW;

(\*pList)->length = 0;

(\*pList)->elem\_size = elem\_size;

Node node = malloc(sizeof(struct ListNode));

if (!node)

return MEMORY\_OVERFLOW;

node->elem = NULL;

node->next = NULL;

(\*pList)->head = node;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DestroyList

Description: Destroy a list as well as freeing the

memory.

Input:

pList that represents the pointer of the list

structure pointer.

fp that represents the user defined dealloc

function that used to free the specific structure.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DestroyList (List\* pList, Dealloc fp) {

if (!pList)

return NULL\_POINTER;

if (!\*pList)

return LIST\_NOT\_EXIST;

Node head, temp;

head = (\*pList)->head;

temp = head->next;

free(head);

head = temp;

while (head) {

temp = head->next;

if (fp)

fp(head->elem);

else

free(head->elem);

free(temp);

head = temp;

}

free(\*pList);

\*pList = NULL;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ClearList

Description: Clear a list.

Input:

list that represents the pointer of the list

structure variable.

fp that represents the user defined dealloc

function that used to free the specific structure.

Return: Status variable that represents the state of

this operaiton.

Others: NO Lazy clear.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ClearList (List list, Dealloc fp) {

if (!list)

return LIST\_NOT\_EXIST;

Node frst, temp;

frst = list->head->next;

while (frst) {

temp = frst->next;

if (fp)

fp(frst->elem);

else

free(frst->elem);

free(frst);

frst = temp;

}

list->head->next = NULL;

list->length = 0;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListEmpty

Description: Decide if a given list is empty

Input:

list that represents the list structure variable.

pBool that represents the pointer of the Bool

variable.

Return: Status variable that represents the state of

this operation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListEmpty (List list, Bool\* pBool) {

if (!list) {

return LIST\_NOT\_EXIST;

}

if (list->head->next)

\*pBool = FALSE;

else

\*pBool = TRUE;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListLength

Description: Count the length of the given list.

Input:

list that represents the list structure variable.

pCount that represents the pointer to the int

variable.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListLength (List list, int\* pCount) {

if (!list)

return LIST\_NOT\_EXIST;

\*pCount = list->length;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: GetElem

Description: Get the element of given index.

Input:

list that represents the list structure variable.

index that represents the index of the list.

pElem that represents the pointer to the element.

Return: Status variable that represents the state of

this operaiton.

Others: pElem is a second-level pointer.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status GetElem (List list, int index, void\* pElem) {

if (!list)

return LIST\_NOT\_EXIST;

if (index < 1 || index > list->length)

return OUT\_OF\_RANGE;

int count = 1;

Node node = list->head->next;

while (count < index) {

count++;

node = node->next;

}

memcpy(pElem, &(node->elem), sizeof(void \*));

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: LocateElem

Description: Locate the element of given element and

given compare function.

Input:

list that represents the list structure variable.

elem that represents the given element.

fp that represents the pointer to the compare function.

pIndex that represents the pointer to the index variable.

Return: Status variable that represents the state of

this operaiton.

Others: index will be set to 0 if don't find specific

element.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status LocateElem (List list, void\* elem, Compare fp, int\* pIndex) {

if (!list)

return LIST\_NOT\_EXIST;

int index = 1;

Node node = list->head->next;;

while (node) {

if (fp(elem, node->elem))

break;

index++;

node = node->next;

}

\*pIndex = node?index:0;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PriorElem

Description: Find the prior element of given element

and given compare function.

Input:

list that represents the list structure variable.

curElem that represents the given element.

pPreElem that represents the target element.

fp that represents the pointer to the compare function.

Return: Status variable that represents the state of

this operaiton.

Others:

\*pPreElem WON'T be defined if don't find

specific element.

pPreElem is a second-level pointer.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PriorElem (List list, void\* curElem, void\* pPreElem, Compare fp) {

if (!list)

return LIST\_NOT\_EXIST;

Node preNode = list->head->next;

Node node = NULL;

if (preNode)

node = preNode->next;

while (node) {

if (fp(node->elem, curElem))

break;

preNode = preNode->next;

node = node->next;

}

if (node)

memcpy(pPreElem, &(preNode->elem), sizeof(void \*));

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NextElem

Description: Find the next element of given element

and given compare function.

Input:

list that represents the list structure variable.

curElem that represents the given element.

pNextElem that represents the target element.

fp that represents the pointer to the compare function.

Return: Status variable that represents the state of

this operaiton.

Others:

\*pNextElem WON'T be defined if don't find

specific element.

pNextElem is a second-level pointer.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NextElem (List list, void\* curElem, void\* pNextElem, Compare fp) {

if (!list)

return LIST\_NOT\_EXIST;

Node node = list->head->next;

while (node) {

if (fp(node->elem, curElem))

break;

node = node->next;

}

if (node && node->next)

memcpy(pNextElem, &(node->next->elem), sizeof(void \*));

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListInsert

Description: Insert one element into given list.

Input:

pList that represents the pointer of the list

structure variable.

index that represents the index of element to

be inserted.

elem that represents the element to be inserted.

fp that represents the user defined assign

function to deep copy the given element.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListInsert (List list, int index, void\* elem, Assign fp) {

if (!list)

return LIST\_NOT\_EXIST;

if (index < 1 || index > list->length + 1)

return OUT\_OF\_RANGE;

Node newNode = malloc(sizeof(struct ListNode));

if (!newNode)

return MEMORY\_OVERFLOW;

newNode->elem = malloc(list->elem\_size);

if (fp)

fp(newNode->elem, elem);

else

memcpy(newNode->elem, elem, list->elem\_size);

Node node = list->head;

int count = 1;

while (count < index) {

count++;

node = node->next;

}

newNode->next = node->next;

node->next = newNode;

list->length++;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListDelete

Description: Delete one element of given list.

Input:

pList that represents the pointer of the list

structure variable.

index that represents the index of element to

be deleted.

elem that represents the pointer to the element

to be deleted.

fp that represents the user defined dealloc

function that used to free the specific structure.

fp2 that represents the user defined assign

function to deep copy the given element.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListDelete (List list, int index, void\* elem, Dealloc fp, Assign fp2) {

if (!list)

return LIST\_NOT\_EXIST;

if (index < 1 || index > list->length)

return OUT\_OF\_RANGE;

int count = 1;

Node node = list->head, temp = NULL;

while (count < index) {

count++;

node = node->next;

}

temp = node->next;

node->next = temp->next;

if (fp2)

fp2(elem, temp->elem);

else

memcpy(elem, temp->elem, list->elem\_size);

if (fp)

fp(temp->elem);

else

free(temp->elem);

free(temp);

list->length--;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ListTraverse

Description: Traverse the given list with given

visit().

Input:

list that represents the list structure variable.

fp that represents the pointer to the visit function.

Return: Status variable that represents the state of

this operaiton.

Others: ListTraverse fails if visit fails.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ListTraverse (List list, Visit fp) {

if (!list)

return LIST\_NOT\_EXIST;

Node node = list->head->next;

while (node) {

if (fp(node->elem))

return EXTERNAL\_FUNCTION\_ERROR;

node = node->next;

}

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileWrite

Description: Write the list to the given file.

Input:

list that represents the list structure variable.

filename that represents the name of the file.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileWrite (List list, char\* filename) {

if (!list)

return LIST\_NOT\_EXIST;

FILE \* fpw;

fpw = fopen(filename, "wb");

if (!fpw)

return FILE\_ERROR;

size\_t elem\_size = list->elem\_size;

fwrite(&(list->length), sizeof(int), 1, fpw);

fwrite(&elem\_size, sizeof(size\_t), 1, fpw);

Node node = list->head->next;

while (node) {

fwrite(node->elem, elem\_size, 1, fpw);

node = node->next;

}

fclose(fpw);

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileRead

Description: Read the list from the given file.

Input:

list that represents the list structure variable.

filename that represents the name of the file.

fp that represents the user defined dealloc

function that used to free the specific structure.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileRead (List\* pList, char\* filename, Dealloc fp) {

if(!pList)

return NULL\_POINTER;

FILE \* fpr;

fpr = fopen(filename, "rb");

if (!fpr)

return FILE\_ERROR;

int length;

size\_t elem\_size;

fread(&length, sizeof(int), 1, fpr);

fread(&elem\_size, sizeof(size\_t), 1, fpr);

DestroyList(pList, fp);

InitList(pList, elem\_size);

(\*pList)->length = length;

(\*pList)->elem\_size = elem\_size;

int count = 0;

Node node = (\*pList)->head;

while (count < length) {

node->next = malloc(sizeof(struct ListNode));

node = node->next;

node->elem = malloc(elem\_size);

fread(node->elem, elem\_size, 1, fpr);

count++;

}

node->next = NULL;

fclose(fpr);

return OK;

}

Display.c

#include <stdlib.h>

#include <string.h>

#include "Linear-Table-On-Linked-Structure.h"

/\* Global variables \*/

**char** \* data\_file\_name = "linked\_list\_display.dat";

/\* Declaration of Structure \*/

**typedef** **struct** Contact {

**char** name[20];

**char** age;

**char** tel[12];

}Contact;

/\* Implementation of Function Pointers \*/

Bool compare(**void**\* elem1, **void**\* elem2) {

**if** (**strcmp**(((Contact \*)elem1)->name, ((Contact \*)elem2)->name) == 0)

**return** TRUE;

**else**

**return** FALSE;

}

Status visit(**void**\* elem) {

**printf**(" ------------------------------ \n");

**printf**("| Name: %22s |\n", ((Contact \*)elem)->name);

**printf**("| Age: %23hhd |\n", ((Contact \*)elem)->age);

**printf**("| Tel: %23s |\n", ((Contact \*)elem)->tel);

**printf**(" ------------------------------ \n");

**printf**("\n");

**return** OK;

}

/\* Program Entry \*/

**int** **main**(**void**) {

**int** op=1;

List list = **NULL**;

**char** ch;

**int** count, index;

Status result;

Bool empty;

Contact elem1, elem2;

Contact \* elem = **NULL**;

**while**(op){

**system**("clear");

**printf**(" Menu for Linear Table On Linked Structure \n");

**printf**("--------------------------------------------------\n");

**printf**(" 1. InitList 8. PriorElem\n");

**printf**(" 2. DestroyList 9. NextElem\n");

**printf**(" 3. ClearList 10. ListInsert\n");

**printf**(" 4. ListEmpty 11. ListDelete\n");

**printf**(" 5. ListLength 12. ListTraverse\n");

**printf**(" 6. GetElem 13. FileRead\n");

**printf**(" 7. LocateElem 14. FileWrite\n");

**printf**(" 0. Exit\n");

**printf**("--------------------------------------------------\n");

**printf**(" Choose from [0~14]:");

**scanf**("%d",&op);

**getchar**();

**switch**(op){

**case 1:**

result = InitList(&list, **sizeof**(Contact));

**if**(result == OK)

**printf**("Successfully created linked list!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Failed to allocate memory for list!\n");

**break**;

**case 2:**

result = DestroyList(&list, **NULL**);

**if** (result == OK)

**printf**("Successfully destroyed linked list!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("List don't exist!\n");

**break**;

**case 3:**

result = ClearList(list, **NULL**);

**if** (result == OK)

**printf**("Successfully cleared linked list!\n");

**else**

**printf**("List don't exist!\n");

**break**;

**case 4:**

**if** (ListEmpty(list, &empty) == OK) {

**if** (empty == TRUE)

**printf**("The list is empty!\n");

**else**

**printf**("The list is NOT empty!\n");

} **else**

**printf**("List don't exist!\n");

**break**;

**case 5:**

**if** (ListLength(list, &count) == OK)

**printf**("The length of the list is: %d\n", count);

**else**

**printf**("List don't exist!\n");

**break**;

**case 6:**

**printf**("Please input the index:");

**scanf**("%d", &index);

**getchar**();

result = GetElem(list, index, &elem);

**if** (result == OK)

**printf**("The contact of index %d has name %s\n", index, elem->name);

**else** **if** (result == LIST\_NOT\_EXIST)

**printf**("List don't exist!\n");

**else**

**printf**("The index is out of range!\n");

**break**;

**case 7:**

**printf**("Please input the name of given contact:");

**scanf**("%s", elem2.name);

**getchar**();

**if** (LocateElem(list, &elem2, &compare, &index) == OK) {

**if** (index == 0) {

**printf**("There is no contact that matches.\n");

} **else** {

**printf**("The index of the first contact that matches is %d.\n", index);

}

} **else** {

**printf**("List don't exist!\n");

}

**break**;

**case 8:**

**printf**("Please input the name of given contact:");

**scanf**("%s", elem2.name);

**getchar**();

**if** (PriorElem(list, &elem2, &elem, &compare) == OK) {

**if**(elem)

**printf**("The prior contact of current contact is %s.\n", elem->name);

} **else** {

**printf**("List don't exist!\n");

}

**break**;

**case 9:**

**printf**("Please input the name of given contact:");

**scanf**("%s", elem2.name);

**getchar**();

**if** (NextElem(list, &elem2, &elem, &compare) == OK) {

**if** (elem)

**printf**("The next contact of current contact is %s.\n", elem->name);

} **else** {

**printf**("List don't exist!\n");

}

**break**;

**case 10:**

**printf**("Please input the index of new contact:");

**scanf**("%d", &index);

**getchar**();

**printf**("Please input the name of given contact:");

**scanf**("%s", elem2.name);

**getchar**();

**printf**("Please input the age of given contact:");

**scanf**("%hhd", &elem2.age);

**getchar**();

**printf**("Please input the tel of given contact:");

**scanf**("%s", elem2.tel);

**getchar**();

result = ListInsert(list, index, &elem2, **NULL**);

**if** (result == OK)

**printf**("The insert operation succeeded!\n");

**else** **if** (result == OUT\_OF\_RANGE)

**printf**("The index is out of range!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else** **if** (result == LIST\_NOT\_EXIST)

**printf**("List don't exist!\n");

**else**

**printf**("Failed to allocate memory for new element!\n");

**break**;

**case 11:**

**printf**("Please input the index of the element to be deleted:");

**scanf**("%d", &index);

**getchar**();

result = ListDelete(list, index, &elem1, **NULL**, **NULL**);

**if** (result == OK)

**printf**("The delete operation succeeded.The name of the deleted element is %s\n", elem1.name);

**else** **if** (result == OUT\_OF\_RANGE)

**printf**("The index is out of range!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("List don't exist!\n");

**break**;

**case 12:**

**if** (ListTraverse(list, &visit) == OK) {

**printf**("\n");

**printf**("The traverse operation succeeded!\n");

}

**else**

**printf**("The list don't exist!\n");

**break**;

**case 13:**

result = FileRead(&list, data\_file\_name, **NULL**);

**if** (result == OK)

**printf**("Data load succeeded!\n");

**else** **if** (result == FILE\_ERROR)

**printf**("Fail to open data file for reading!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Fail to allocate memory for list!\n");

**break**;

**case 14:**

result = FileWrite(list, data\_file\_name);

**if** (result == OK)

**printf**("Data save succeeded!\n");

**else** **if** (result == FILE\_ERROR)

**printf**("Fail to open data file for writing!\n");

**else**

**printf**("You should NOT write a list that DON'T exist!\n");

**break**;

**case 0:**

op = 0;

**printf**("Press Enter key to exit.\n");

**break**;

}

**getchar**();

}

**printf**("See you next time!\n");

**return** 0;

}

Makefile（Linux和Mac下）

SRC\_DIR = src

INC\_DIR = include

OBJ\_DIR = objs

BIN\_DIR = bin

DIRS := $(OBJ\_DIR) $(BIN\_DIR)

CC = gcc

CFLAGS = -I $(INC\_DIR)

RM = rm -f

MKDIR = mkdir

SRCS = $(**wildcard** ${SRC\_DIR}/**\***.c)

OBJS = $(**patsubst** %.c, ${OBJ\_DIR}/%.o, $(**notdir** ${SRCS}))

HEADERS = $(**wildcard** ${INC\_DIR}/**\***.h)

TARGET = $(BIN\_DIR)/display

all **:** $(DIRS) $(TARGET)

$(DIRS) **:**

$(MKDIR) $@

$(TARGET) **:** $(OBJS)

$(CC) -o $@ $(OBJS)

$(OBJS) **:** $(OBJ\_DIR)/%.o : $(SRC\_DIR)/%.c $(HEADERS)

$(CC) $(CFLAGS) -c $< -o $@

.PHONY **:** clean

clean **:**

$(RM) $(OBJS) $(TARGET)

附录三 基于二叉链表结构二叉树实现的源程序

Binary\_Tree\_On\_Linked\_Structure.h

/\* Binary Tree on Link Structure \*/

#ifndef BINARY\_TREE\_ON\_LINKED\_STRUCTURE\_H

#define BINARY\_TREE\_ON\_LINKED\_STRUCTURE\_H

/\* Include Necessary Standard Library \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Include Stack and Queue Stucture \*/

#include "Queue-On-Sequence-Structure.h"

#include "Stack-On-Sequence-Structure.h"

/\* Typedef of Enumeration \*/

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

**typedef** **enum** Child {LEFT = 0, RIGHT}Child;

**typedef** **enum** Status {OK, NULL\_POINTER = 100, TREE\_NOT\_EXIST, RIGHT\_CHILD\_NOT\_EMPTY, MEMORY\_OVERFLOW, OUT\_OF\_RANGE, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

/\* Macro Define \*/

#define MAX\_NODE\_NUMBER (31) //Defines the max capacity of stack and queue

#define RECURSIVE\_TRAVERSE 0 //Defines whether to use recursive traverse

/\* Definition and Typedef of Data and Structure and typedef of Function Pointers \*/

**struct** BiTree;

**typedef** **struct** BiTree \* Tree;

**struct** BiTreeNode;

**typedef** **struct** BiTreeNode \* Node;

**typedef** **struct** BiTreeNode \* Root;

/\* Declaration of Data and Structure \*/ //NOT RECOMMENDED!!!

**struct** BiTree {

**size\_t** elem\_size;

**struct** BiTreeNode \* root;

};

**struct** BiTreeNode {

**void** \* elem;

**struct** BiTreeNode \* lchild;

**struct** BiTreeNode \* rchild;

};

/\* Function Pointers \*/

**typedef** Bool (\*Compare)(**void**\* elem1, **void**\* elem2);

**typedef** Status (\*Visit)(**void**\* elem);

**typedef** Status (\*Construct)(**void**\* pElem);

/\* Definition of Data Operations \*/

Status InitTree(Tree\* pTree, **size\_t** elem\_size);

Status DestroyTree(Tree\* pTree);

Status PreOrderCreateTree(Tree tree, Construct construct);

Status ClearTree(Tree tree);

Status TreeEmpty(Tree tree, Bool\* pBool);

Status TreeDepth(Tree tree, **int**\* pCount);

Status TreeRoot(Tree tree, Root\* root);

Status NodeValue(Tree tree, Node node, **void**\* pElem);

Status NodeAssign(Tree tree, Node node, **void**\* elem);

Status NodeParent(Tree tree, Node node, Node\* pNode);

Status NodeLeftChild(Tree tree, Node node, Node\* pNode);

Status NodeRightChild(Tree tree, Node node, Node\* pNode);

Status NodeLeftSibling(Tree tree, Node node, Node\* pNode);

Status NodeRightSibling(Tree tree, Node node, Node\* pNode);

Status InsertChild(Tree tree, Node node, Child orient, Node child);

Status DeleteChild(Tree tree, Node node, Child orient);

Status PreOrderTraverse(Tree tree, Visit visit);

Status InOrderTraverse(Tree tree, Visit visit);

Status PostOrderTraverse(Tree tree, Visit visit);

Status LevelOrderTraverse(Tree tree, Visit visit);

/\* Definition of File Operations\*/

Status FileWrite(Tree tree, **char**\* filename);

Status FileRead(Tree\* pTree, **char**\* filename);

#endif

Binary\_Tree\_On\_Linked\_Structure.c

#include "Binary-Tree-On-Linked-Structure.h"

/\* Define the flag for file I/O \*/

**char** yes = 'y';

**char** no = 'n';

/\* Static Functions Supporting Operations \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: freeTree

Description: free the given tree recursively.

Input:

node represents the root for current tree.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** freeTree(Node node) {

**if** (node) {

freeTree(node->lchild);

freeTree(node->rchild);

**free**(node->elem);

**free**(node);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: treeDepth

Description: return the depth of tree recursively.

Input:

node represents the root for current tree.

Return:

Int variable represents the tree depth.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **int** treeDepth(Node node) {

**if** (!node) **return** 0;

**int** left = 0, right = 0;

**if** (node->lchild) left = treeDepth(node->lchild);

**if** (node->rchild) right = treeDepth(node->rchild);

**return** left > right ? (left + 1) : (right + 1);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: findParent

Description: find the parent node of given node.

Input:

node represents the current node.

targetNode represents the given node.

Return:

Node variable represents the parent node of

given node.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** Node findParent(Node node, Node targetNode) {

**if** (node) {

**if** (node->lchild == targetNode || node->rchild == targetNode)

**return** node;

**else** {

Node parent = findParent(node->lchild, targetNode);

**if** (parent)

**return** parent;

**else**

**return** findParent(node->rchild, targetNode);

}

} **else** {

**return** **NULL**;

}

}

//precompile if RECURSIVE\_TRAVERSE switch is ON.

#if (RECURSIVE\_TRAVERSE == 1)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: recursivePreOrderTraverse

Description: traverse the tree in preorder recursively.

Input:

node represents the root for current tree.

visit represents the external visit function.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** recursivePreOrderTraverse(Node node, Visit visit) {

**if** (node) {

visit(node->elem);

recursivePreOrderTraverse(node->lchild, visit);

recursivePreOrderTraverse(node->rchild, visit);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: recursiveInOrderTraverse

Description: traverse the tree in inorder recursively.

Input:

node represents the root for current tree.

visit represents the external visit function.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** recursiveInOrderTraverse(Node node, Visit visit) {

**if** (node) {

recursiveInOrderTraverse(node->lchild, visit);

visit(node->elem);

recursiveInOrderTraverse(node->rchild, visit);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: recursivePostOrderTraverse

Description: traverse the tree in postorder recursively.

Input:

node represents the root for current tree.

visit represents the external visit function.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** recursivePostOrderTraverse(Node node, Visit visit) {

**if** (node) {

recursivePostOrderTraverse(node->lchild, visit);

recursivePostOrderTraverse(node->rchild, visit);

visit(node->elem);

}

}

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: recursivePreOrderCreateTree

Description: create the tree in preorder recursively.

Input:

pNode represents the pointer to the node.

construct represents the external construct function.

elem\_size represents the size of elem struct.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** recursivePreOrderCreateTree(Node\* pNode, Construct construct, **size\_t** elem\_size) {

**void** \* elem = **NULL**;

construct(&elem);

**if** (elem) {

\*pNode = **malloc**(**sizeof**(**struct** BiTreeNode));

(\*pNode)->elem = **malloc**(elem\_size);

**memcpy**((\*pNode)->elem, elem, elem\_size);

**free**(elem);

recursivePreOrderCreateTree(&((\*pNode)->lchild), construct, elem\_size);

recursivePreOrderCreateTree(&((\*pNode)->rchild), construct, elem\_size);

} **else** {

\*pNode = **NULL**;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: recursivePreOrderSaveTree

Description: save the tree in preorder recursively.

Input:

node represents root for current tree.

fp represents the file pointer.

elem\_size represents the size of elem struct.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** recursivePreOrderSaveTree(Node node, **FILE**\* fp, **size\_t** elem\_size) {

**if** (node) {

**fwrite**(&yes, **sizeof**(**char**), 1, fp);

**fwrite**(node->elem, elem\_size, 1, fp);

recursivePreOrderSaveTree(node->lchild, fp, elem\_size);

recursivePreOrderSaveTree(node->rchild, fp, elem\_size);

} **else** {

**fwrite**(&no, **sizeof**(**char**), 1, fp);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: recursivePreOrderLoadTree

Description: load the tree in preorder recursively.

Input:

pNode represents the pointer to the node.

fp represents the file pointer.

elem\_size represents the size of elem struct.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** recursivePreOrderLoadTree(Node\* pNode, **FILE**\* fp, **size\_t** elem\_size) {

**char** flag = '\0';

**fread**(&flag, **sizeof**(**char**), 1, fp);

**if** (flag == yes) {

\*pNode = **malloc**(**sizeof**(**struct** BiTreeNode));

(\*pNode)->elem = **malloc**(elem\_size);

(\*pNode)->lchild = **NULL**;

(\*pNode)->rchild = **NULL**;

**fread**((\*pNode)->elem, elem\_size, 1, fp);

recursivePreOrderLoadTree(&((\*pNode)->lchild), fp, elem\_size);

recursivePreOrderLoadTree(&((\*pNode)->rchild), fp, elem\_size);

} **else** **if** (flag == no) {

\*pNode = **NULL**;

}

}

/\* Implementation of Operations \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InitTree

Description: Init a Tree.

Input:

pTree that represents the pointer to the tree

structure pointer.

elem\_size that represents the size of the specific

structure.

Return: Status variable that represents the state of

this operaiton.

Others: Won't throw exception if using one variable

to create two or more tree without freeing.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InitTree(Tree\* pTree, **size\_t** elem\_size) {

**if** (!pTree)

**return** NULL\_POINTER;

\*pTree = **malloc**(**sizeof**(**struct** BiTree));

**if** (!\*pTree)

**return** MEMORY\_OVERFLOW;

(\*pTree)->elem\_size = elem\_size;

(\*pTree)->root = **NULL**;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DestroyTree

Description: Destroy a Tree.

Input:

pTree that represents the pointer to the tree

structure pointer.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DestroyTree(Tree\* pTree) {

**if** (!pTree)

**return** NULL\_POINTER;

**if** (!\*pTree)

**return** TREE\_NOT\_EXIST;

**if** ((\*pTree)->root) {

freeTree((\*pTree)->root);

(\*pTree)->root = **NULL**;

}

**free**(\*pTree);

\*pTree = **NULL**;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PreOrderCreateTree

Description: create a tree with preorder.

Input:

tree represents the tree variable.

construct represents the external construct

function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PreOrderCreateTree(Tree tree, Construct construct) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!construct)

**return** NULL\_POINTER;

freeTree(tree->root);

tree->root = **NULL**;

recursivePreOrderCreateTree(&(tree->root), construct, tree->elem\_size);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: ClearTree

Description: clear a tree in postorder ecursively.

Input:

tree represents the tree variable.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status ClearTree(Tree tree) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

freeTree(tree->root);

tree->root = **NULL**;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: TreeEmpty

Description: judge if a tree is empry.

Input:

tree represents the tree variable.

pBool represents the pointer to the bool variable.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status TreeEmpty(Tree tree, Bool\* pBool) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!pBool)

**return** NULL\_POINTER;

**if** (tree->root)

\*pBool = FALSE;

**else**

\*pBool = TRUE;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: TreeDepth

Description: give the depth of a tree.

Input:

tree represents the tree variable.

PCount represents the pointer to the int variable.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status TreeDepth(Tree tree, **int**\* pCount) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!pCount)

**return** NULL\_POINTER;

\*pCount = treeDepth(tree->root);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: TreeRoot

Description: give the root of a given tree.

Input:

tree represents the tree variable.

root represents the pointer to the root variable.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status TreeRoot(Tree tree, Root\* root) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!root)

**return** NULL\_POINTER;

\*root = tree->root;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NodeValue

Description: give the value of the node.

Input:

tree represents the tree variable.

node represents the node variable.

pElem represents the pointer to the elem variable(pointer).

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NodeValue(Tree tree, Node node, **void**\* pElem) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!node || !pElem)

**return** NULL\_POINTER;

**memcpy**(pElem, &(node->elem), **sizeof**(**void** \*));

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NodeAssign

Description: assign the node with given elem value.

Input:

tree represents the tree variable.

node represents the node variable.

elem represents the elem variable(pointer).

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NodeAssign(Tree tree, Node node, **void**\* elem) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!node || !elem)

**return** NULL\_POINTER;

**if** (!node->elem)

node->elem = **malloc**(tree->elem\_size);

**memcpy**(node->elem, elem, tree->elem\_size);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NodeParent

Description: give the parent of the node.

Input:

tree represents the tree variable.

node represents the node to be searched.

pNode represents the pointer to the external node

variable.(return value)

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NodeParent(Tree tree, Node node, Node\* pNode) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!pNode || !node)

**return** NULL\_POINTER;

**if** (tree->root) {

**if** (tree->root == node) {

\*pNode = **NULL**;

} **else** {

\*pNode = findParent(tree->root, node);

}

} //\*pNode undefined if root is NULL.

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NodeLeftChild

Description: give the left child of the node.

Input:

tree represents the tree variable.

node represents the node to be searched.

pNode represents the pointer to the external node

variable.(return value)

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NodeLeftChild(Tree tree, Node node, Node\* pNode) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!pNode || !node)

**return** NULL\_POINTER;

\*pNode = node->lchild;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NodeRightChild

Description: give the right child of the node.

Input:

tree represents the tree variable.

node represents the node to be searched.

pNode represents the pointer to the external node

variable.(return value)

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NodeRightChild(Tree tree, Node node, Node\* pNode) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!pNode || !node)

**return** NULL\_POINTER;

\*pNode = node->rchild;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NodeLeftSibling

Description: give the left sibling of the node.

Input:

tree represents the tree variable.

node represents the node to be searched.

pNode represents the pointer to the external node

variable.(return value)

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NodeLeftSibling(Tree tree, Node node, Node\* pNode) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!pNode || !node)

**return** NULL\_POINTER;

**if** (tree->root) {

Node parent = findParent(tree->root, node);

**if** (parent) {

**if** (parent->lchild == node)

\*pNode = **NULL**;

**else**

\*pNode = parent->lchild;

} **else**

\*pNode = **NULL**;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NodeRightSibling

Description: give the right sibling of the node.

Input:

tree represents the tree variable.

node represents the node to be searched.

pNode represents the pointer to the external node

variable.(return value)

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NodeRightSibling(Tree tree, Node node, Node\* pNode) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!pNode || !node)

**return** NULL\_POINTER;

**if** (tree->root) {

Node parent = findParent(tree->root, node);

**if** (parent) {

**if** (parent->rchild == node)

\*pNode = **NULL**;

**else**

\*pNode = parent->rchild;

} **else**

\*pNode = **NULL**;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InsertChild

Description: insert node(subtree) into another tree.

Input:

tree represents the tree variable.

node represents the node to be searched.

orient represents the orient of the tree(left or right).

child represents the node or subtree.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InsertChild(Tree tree, Node node, Child orient, Node child) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!node || !child)

**return** NULL\_POINTER;

**if** (child->rchild != **NULL**)

**return** RIGHT\_CHILD\_NOT\_EMPTY;

Node temp;

**if** (orient == LEFT) {

temp = node->lchild;

node->lchild = child;

child->rchild = temp;

} **else** {

temp = node->rchild;

node->rchild = child;

child->rchild = temp;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DeleteChild

Description: delete the left or right subtree of

given node.

Input:

tree represents the tree variable.

node represents the node to be searched.

orient represents the orient of the tree(left or right).

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DeleteChild(Tree tree, Node node, Child orient) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!node)

**return** NULL\_POINTER;

**if** (orient == LEFT) {

freeTree(node->lchild);

node->lchild = **NULL**;

} **else** {

freeTree(node->rchild);

node->rchild = **NULL**;

}

**return** OK;

}

//Precompile according to RECURSIVE\_TRAVERSE flag

#if (RECURSIVE\_TRAVERSE == 1)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PreOrderTraverse

Description: traverse the tree in preorder recursively.

Input:

tree represents the tree variable.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PreOrderTraverse(Tree tree, Visit visit) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!visit)

**return** NULL\_POINTER;

recursivePreOrderTraverse(tree->root, visit);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InOrderTraverse

Description: traverse the tree in inorder recursively.

Input:

tree represents the tree variable.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InOrderTraverse(Tree tree, Visit visit) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!visit)

**return** NULL\_POINTER;

recursiveInOrderTraverse(tree->root, visit);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PostOrderTraverse

Description: traverse the tree in postorder recursively.

Input:

tree represents the tree variable.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PostOrderTraverse(Tree tree, Visit visit) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!visit)

**return** NULL\_POINTER;

recursivePostOrderTraverse(tree->root, visit);

**return** OK;

}

#elif (RECURSIVE\_TRAVERSE == 0)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PreOrderTraverse

Description: traverse the tree in preorder non-recursively.

Input:

tree represents the tree variable.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PreOrderTraverse(Tree tree, Visit visit) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!visit)

**return** NULL\_POINTER;

Stack S = CreateStack(MAX\_NODE\_NUMBER);

Node node = tree->root;

**while** (node || !IsStackEmpty(S)) {

**while** (node) {

visit(node->elem);

Push(node, S);

node = node->lchild;

}

**if** (!IsStackEmpty(S)) {

node = TopAndPop(S);

node = node->rchild;

}

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InOrderTraverse

Description: traverse the tree in inorder non-recursively.

Input:

tree represents the tree variable.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InOrderTraverse(Tree tree, Visit visit) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!visit)

**return** NULL\_POINTER;

Stack S = CreateStack(MAX\_NODE\_NUMBER);

Node node = tree->root;

**while** (node || !IsStackEmpty(S)) {

**while** (node) {

Push(node, S);

node = node->lchild;

}

**if** (!IsStackEmpty(S)) {

node = TopAndPop(S);

visit(node->elem);

node = node->rchild;

}

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PostOrderTraverse

Description: traverse the tree in postorder non-recursively.

Input:

tree represents the tree variable.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PostOrderTraverse(Tree tree, Visit visit) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!visit)

**return** NULL\_POINTER;

**if** (tree->root) {

Node node = **NULL**;

Stack S1 = CreateStack(MAX\_NODE\_NUMBER);

Stack S2 = CreateStack(MAX\_NODE\_NUMBER);

Push(tree->root, S1);

**while** (!IsStackEmpty(S1)) {

node = TopAndPop(S1);

Push(node, S2);

**if** (node->lchild) Push(node->lchild, S1);

**if** (node->rchild) Push(node->rchild, S1);

}

**while**(!IsStackEmpty(S2)) {

node = TopAndPop(S2);

visit(node->elem);

}

}

**return** OK;

}

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: LevelOrderTraverse

Description: traverse the tree in levelorder

using queue.

Input:

tree represents the tree variable.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status LevelOrderTraverse(Tree tree, Visit visit) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**if** (!visit)

**return** NULL\_POINTER;

**if** (tree->root) {

Queue Q = CreateQueue(MAX\_NODE\_NUMBER);

Node temp = **NULL**;

Enqueue(tree->root, Q);

**while** (!IsQueueEmpty(Q)) {

temp = FrontAndDequeue(Q);

visit(temp->elem);

**if** (temp->lchild) Enqueue(temp->lchild, Q);

**if** (temp->rchild) Enqueue(temp->rchild, Q);

}

DisposeQueue(Q);

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileWrite

Description: write the tree to file.

Input:

tree represents the tree variable.

filename represents the pointer to the filename string.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileWrite(Tree tree, **char**\* filename) {

**if** (!tree)

**return** TREE\_NOT\_EXIST;

**FILE** \* fpw = **fopen**(filename, "wb");

**if** (!fpw)

**return** FILE\_ERROR;

recursivePreOrderSaveTree(tree->root, fpw, tree->elem\_size);

**fclose**(fpw);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileRead

Description: read the tree from file.

Input:

tree represents the tree variable.

filename represents the pointer to the filename string.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileRead(Tree\* pTree, **char**\* filename) {

**if** (!pTree)

**return** NULL\_POINTER;

**if** (!\*pTree)

**return** TREE\_NOT\_EXIST;

**FILE** \* fpr = **fopen**(filename, "rb");

**if** (!fpr)

**return** FILE\_ERROR;

recursivePreOrderLoadTree(&((\*pTree)->root), fpr, (\*pTree)->elem\_size);

**fclose**(fpr);

**return** OK;

}

Queue-On-Sequence-Structure.h

/\* Queue on Sequence Structure \*/

#ifndef QUEUE\_ON\_SEQUENCE\_STRUCTURE\_H

#define QUEUE\_ON\_SEQUENCE\_STRUCTURE\_H

/\* Include Necessary Standard Library \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Include Binary Tree Structure \*/

#include "Binary-Tree-On-Linked-Structure.h"

/\* Definition and Typedef of Data and Structure and typedef of Function Pointers \*/

**typedef** **struct** BiTreeNode \* Element;

**struct** Queue;

**typedef** **struct** Queue \* Queue;

/\* Definition of Data Operations \*/

**int** IsQueueEmpty(Queue Q);

**int** IsQueueFull(Queue Q);

Queue CreateQueue(**int** capacity);

**void** DisposeQueue(Queue Q);

**void** MakeQueueEmpty(Queue Q);

**void** Enqueue(Element X, Queue Q);

**void** Dequeue(Queue Q);

Element Front(Queue Q);

Element FrontAndDequeue(Queue Q);

#endif

Queue-On-Sequence-Structure.c

#include "Queue-On-Sequence-Structure.h"

/\* Implementation of Operations \*/

**struct** Queue {

**int** capacity;

**int** rear;

**int** front;

**int** size;

Element \* elemArray;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Succ

Description: return the succ of the front/rear of queue.

Input:

value represents the index of certain element of queue.

Q represents the queue variable.

Return:

int represents the succ.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **int** Succ(**int** value, Queue Q) {

**if** (++value == Q->capacity)

value = 0;

**return** value;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: IsQueueEmpty

Description: judge if the queue is empty

Input:

Q represents the queue variable.

Return:

int variable represents the judge result.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**int** IsQueueEmpty(Queue Q) {

**return** (Q->size == 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: IsQueueFull

Description: judge if the queue is full.

Input:

Q represents the queue variable.

Return:

int variable represents the judge result.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**int** IsQueueFull(Queue Q) {

**return** (Q->size == Q->capacity);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: CreateQueue

Description: create an empty queue.

Input:

capacity represents the capacity of queue.

Return:

Queue variable represents the created queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Queue CreateQueue(**int** capacity) {

**if** (capacity > 0) {

Queue Q = **malloc**(**sizeof**(**struct** Queue));

**if** (!Q) **exit**(-1);

Q->capacity = capacity;

Q->elemArray = **malloc**(**sizeof**(Element)\*capacity);

**if** (!Q->elemArray) **exit**(-1);

MakeQueueEmpty(Q);

**return** Q;

} **else** {

**return** **NULL**;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DisposeQueue

Description: dispose the given tree.

Input:

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** DisposeQueue(Queue Q) {

**if** (Q) {

**if** (Q->elemArray) **free**(Q->elemArray);

**free**(Q);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: MakeQueueEmpty

Description: clear the given queue.

Input:

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** MakeQueueEmpty(Queue Q) {

Q->size = 0;

Q->front = 1;

Q->rear = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Enqueue

Description: enqueue the certain element to given queue.

Input:

X represents the element variable.

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** Enqueue(Element X, Queue Q) {

**if** (IsQueueFull(Q))

**return**;

**else** {

Q->size++;

Q->rear = Succ(Q->rear, Q);

Q->elemArray[Q->rear] = X;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Dequeue

Description: dequeue the certain queue.

Input:

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** Dequeue(Queue Q) {

**if** (IsQueueEmpty(Q))

**return**;

**else** {

Q->size--;

Q->front = Succ(Q->front, Q);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Front

Description: return the front of given queue.

Input:

Q represents the queue variable.

Return:

Element represents the front element of queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Element Front(Queue Q) {

**if** (IsQueueEmpty(Q))

**return** **NULL**;

**else** {

**return** Q->elemArray[Q->front];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FrontAndDequeue

Description: front and then dequeue the given queue.

Input:

Q represents the queue variable.

Return:

Element represents the front element of queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Element FrontAndDequeue(Queue Q) {

**if** (IsQueueEmpty(Q))

**return** **NULL**;

**else** {

Element temp = Q->elemArray[Q->front];

Dequeue(Q);

**return** temp;

}

}

Stack-On-Sequence-Structure.h

/\* Queue on Sequence Structure \*/

#ifndef STACK\_ON\_SEQUENCE\_STRUCTURE\_H

#define STACK\_ON\_SEQUENCE\_STRUCTURE\_H

/\* Include Necessary Standard Library \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Include Binary Tree Structure \*/

#include "Binary-Tree-On-Linked-Structure.h"

/\* Definition and Typedef of Data and Structure and typedef of Function Pointers \*/

**typedef** **struct** BiTreeNode \* Element;

**struct** Stack;

**typedef** **struct** Stack \* Stack;

/\* Definition of Data Operations \*/

**int** IsStackEmpty(Stack S);

**int** IsStackFull(Stack S);

Stack CreateStack(**int** capacity);

**void** DisposeStack(Stack S);

**void** MakeStackEmpty(Stack S);

**void** Push(Element X, Stack S);

**void** Pop(Stack S);

Element Top(Stack S);

Element TopAndPop(Stack S);

#endif

Stack-On-Sequence-Structure.c

#include "Stack-On-Sequence-Structure.h"

/\* Implementation of Operations \*/

#define EmptyPos (-1)

**struct** Stack {

**int** capacity;

**int** top;

Element \* elemArray;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: IsStackEmpty

Description: judge if the stack is empty.

Input:

S represents the stack variable.

Return:

int variable represents the result of judge.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**int** IsStackEmpty(Stack S) {

**return** (S->top == EmptyPos);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: IsStackFull

Description: judge if the stack is full.

Input:

S represents the stack variable.

Return:

int variable represents the result of judge.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**int** IsStackFull(Stack S) {

**return** (S->top == S->capacity-1);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: CreateStack

Description: create an empty stack.

Input:

capacity represents the capacity of the stack.

Return:

Stack variable represents the created stack.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Stack CreateStack(**int** capacity) {

**if** (capacity > 0) {

Stack S = **malloc**(**sizeof**(**struct** Stack));

**if** (!S) **exit**(-1);

S->capacity = capacity;

S->elemArray = **malloc**(**sizeof**(Element)\*capacity);

**if** (!S->elemArray) **exit**(-1);

MakeStackEmpty(S);

**return** S;

} **else** {

**return** **NULL**;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DisposeStack

Description: dispose the given stack.

Input:

S represents the stack variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** DisposeStack(Stack S) {

**if** (S) {

**if** (S->elemArray) **free**(S->elemArray);

**free**(S);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: MakeStackEmpty

Description: clear the given stack.

Input:

S represents the stack variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** MakeStackEmpty(Stack S) {

S->top = EmptyPos;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Push

Description: push the given element into stack.

Input:

X represents the element variable.

S represents the stack variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** Push(Element X, Stack S) {

**if** (IsStackFull(S))

**return**;

**else** {

S->elemArray[++S->top] = X;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Pop

Description: pop the top of stack.

Input:

S represents the stack variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** Pop(Stack S) {

**if** (!IsStackEmpty(S)) {

S->top--;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Top

Description: return the top of stack.

Input:

S represents the stack variable.

Return:

Element represents the top of stack.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Element Top(Stack S) {

**if** (!IsStackEmpty(S)) {

**return** S->elemArray[S->top];

} **else**

**return** **NULL**;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: TopAndPop

Description: top and then pop the top of stack.

Input:

S represents the stack variable.

Return:

Element represents the top of stack.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Element TopAndPop(Stack S) {

**if** (!IsStackEmpty(S)) {

**return** S->elemArray[S->top--];

} **else**

**return** **NULL**;

}

Display.c

#include <stdlib.h>

#include <string.h>

#include "Binary-Tree-On-Linked-Structure.h"

/\* Global variables \*/

**char** \* data\_file\_name = "binary\_tree\_display.dat";

/\* Declaration of Structure \*/

**typedef** **struct** Contact {

**char** name[20];

**int** age;

**char** tel[12];

}Contact;

/\* Implementation of Function Pointers \*/

Bool compare(**void**\* elem1, **void**\* elem2) {

**if** (**strcmp**(((Contact \*)elem1)->name, ((Contact \*)elem2)->name) == 0)

**return** TRUE;

**else**

**return** FALSE;

}

Status visit(**void**\* elem) {

**printf**(" ------------------------------ \n");

**printf**("| Name: %22s |\n", ((Contact \*)elem)->name);

**printf**("| Age: %23d |\n", ((Contact \*)elem)->age);

**printf**("| Tel: %23s |\n", ((Contact \*)elem)->tel);

**printf**(" ------------------------------ \n");

**printf**("\n");

**return** OK;

}

Status construct(**void** \* pElem) {

**char** flag = '\0';

**printf**("Do this position has contact?(y/?)");

**scanf**("%c", &flag);

**getchar**();

**if** (flag == 'y' || flag == 'Y') {

Contact temp;

**printf**("Please input the name of contact:");

**scanf**("%s", temp.name);

**getchar**();

**printf**("Please input the age of contact:");

**scanf**("%d", &temp.age);

**getchar**();

**printf**("Please input the phone number of contact:");

**scanf**("%s", temp.tel);

**getchar**();

Contact \*\* contact = (Contact \*\*)pElem;

\*contact = **malloc**(**sizeof**(Contact));

\*\*contact = temp;

}

**return** OK;

}

/\* Program Entry \*/

**int** **main**(**void**) {

**int** op=1;

Tree treeArray[5] = {**NULL**};

Tree\* tree = &treeArray[0];

**char** ch;

**int** count, index;

Status result;

Bool empty;

Child orient;

Contact elem1, elem2;

Contact \* elem = **NULL**;

Node node = **NULL**;

**int** tree\_num;

**while**(op){

**system**("clear");

**printf**(" Menu for Binary Tree On Linked Structure \n");

**printf**("--------------------------------------------------\n");

**printf**(" 1. InitTree 12. RightChild\n");

**printf**(" 2. DestroyTree 13. LeftSibling\n");

**printf**(" 3. CreateTree 14. RightSibling\n");

**printf**(" 4. ClearTree 15. InsertChild\n");

**printf**(" 5. TreeEmpty 16. DeleteChild\n");

**printf**(" 6. TreeDepth 17. PreOrderTraverse\n");

**printf**(" 7. TreeRoot 18. InOrderTraverse\n");

**printf**(" 8. Value 19. PostOrderTraverse\n");

**printf**(" 9. Assign 20. LevelOrderTraverse\n");

**printf**(" 10. Parent 21. FileRead\n");

**printf**(" 11. LeftChild 22. FileWrite\n");

**printf**(" 0. Exit 23. SelectTree\n");

**printf**("--------------------------------------------------\n");

**printf**(" Choose from [0~22]:");

**scanf**("%d",&op);

**getchar**();

**switch**(op){

**case 1:**

result = InitTree(tree, **sizeof**(Contact));

**if**(result == OK)

**printf**("Successfully created binary tree!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Failed to allocate memory for tree!\n");

**break**;

**case 2:**

result = DestroyTree(tree);

**if** (result == OK)

**printf**("Successfully destroyed binary tree!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Tree don't exist!\n");

**break**;

**case 3:**

result = PreOrderCreateTree(\*tree, construct);

**if** (result == OK)

**printf**("Successfully fill the tree with given definition!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("Required construct function!\n");

**else**

**printf**("Tree don't exist!\n");

**break**;

**case 4:**

result = ClearTree(\*tree);

**if** (result == OK)

**printf**("Successfully cleared binary tree!\n");

**else**

**printf**("Tree don't exist!\n");

**break**;

**case 5:**

result = TreeEmpty(\*tree, &empty);

**if** (result == OK) {

**if** (empty == TRUE)

**printf**("The tree is empty!\n");

**else**

**printf**("The tree is NOT empty!\n");

} **else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Bool variable is required!\n");

**break**;

**case 6:**

result = TreeDepth(\*tree, &count);

**if** (result == OK)

**printf**("The depth of the tree is: %d\n", count);

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Int variable is required!\n");

**break**;

**case 7:**

result = TreeRoot(\*tree, &node);

**if** (result == OK)

**printf**("Successffully move node to root");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Root \* variable is required!\n");

**break**;

**case 8:**

result = NodeValue(\*tree, node, &elem);

**if** (result == OK)

**printf**("Current node has name:%s\n", elem->name);

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("NULL Pointer(s) passed in!\n");

**break**;

**case 9:**

**printf**("Please input the name of contact:");

**scanf**("%s", elem1.name);

**getchar**();

**printf**("Please input the age of contact:");

**scanf**("%d", &elem1.age);

**getchar**();

**printf**("Please input the phone number of contact:");

**scanf**("%s", elem1.tel);

**getchar**();

result = NodeAssign(\*tree, node, &elem1);

**if** (result == OK)

**printf**("Assign new value to node!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("NULL Pointer(s) passed in!\n");

**break**;

**case 10:**

result = NodeParent(\*tree, node, &node);

**if** (result == OK) {

**if** (node)

**printf**("Successffully move node to its parent!\n");

**else**

**printf**("node is NULL.\n");

} **else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("NULL Pointer(s) passed in!\n");

**break**;

**case 11:**

result = NodeLeftChild(\*tree, node, &node);

**if** (result == OK) {

**if** (node)

**printf**("Successffully move node to its left child!\n");

**else**

**printf**("node is NULL.\n");

} **else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("NULL Pointer(s) passed in!\n");

**break**;

**case 12:**

result = NodeRightChild(\*tree, node, &node);

**if** (result == OK) {

**if** (node)

**printf**("Successffully move node to its right child!\n");

**else**

**printf**("node is NULL.\n");

} **else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("NULL Pointer(s) passed in!\n");

**break**;

**case 13:**

result = NodeLeftSibling(\*tree, node, &node);

**if** (result == OK) {

**if** (node)

**printf**("Successffully move node to its left sibling!\n");

**else**

**printf**("node is NULL.\n");

} **else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("NULL Pointer(s) passed in!\n");

**break**;

**case 14:**

result = NodeRightSibling(\*tree, node, &node);

**if** (result == OK) {

**if** (node)

**printf**("Successffully move node to its right sibling!\n");

**else**

**printf**("node is NULL.\n");

} **else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("NULL Pointer(s) passed in!\n");

**break**;

**case 15:**

{

**printf**("Please input the name of contact:");

**scanf**("%s", elem1.name);

**getchar**();

**printf**("Please input the age of contact:");

**scanf**("%d", &elem1.age);

**getchar**();

**printf**("Please input the phone number of contact:");

**scanf**("%s", elem1.tel);

**getchar**();

Node newNode = **malloc**(**sizeof**(**struct** BiTreeNode));

newNode->elem = **malloc**(**sizeof**(Contact));

**memcpy**(newNode->elem, &elem1, **sizeof**(Contact));

newNode->lchild = **NULL**;

newNode->rchild = **NULL**;

**printf**("Input l/L fot left and r/R for right:\n");

**scanf**("%c", &ch);

**getchar**();

**if** (ch == 'l' || ch == 'L')

orient = LEFT;

**else** **if** (ch == 'r' || ch == 'R')

orient = RIGHT;

result = InsertChild(\*tree, node, orient, newNode);

**if** (result == OK)

**printf**("Successffully insert child!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL Pointer(s) passed in!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Right child of node is NOT empty!\n");

}

**break**;

**case 16:**

**printf**("Input l/L fot left and r/R for right:\n");

**scanf**("%c", &ch);

**getchar**();

**if** (ch == 'l' || ch == 'L')

orient = LEFT;

**else** **if** (ch == 'r' || ch == 'R')

orient = RIGHT;

result = DeleteChild(\*tree, node, orient);

**if** (result == OK)

**printf**("Successffully delete the subtree of node!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL Pointer(s) passed in!\n");

**else**

**printf**("Tree don't exist!\n");

**break**;

**case 17:**

result = PreOrderTraverse(\*tree, visit);

**if** (result == OK)

**printf**("Traverse succeeded!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Visit is required!\n");

**break**;

**case 18:**

result = InOrderTraverse(\*tree, visit);

**if** (result == OK)

**printf**("Traverse succeeded!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Visit is required!\n");

**break**;

**case 19:**

result = PostOrderTraverse(\*tree, visit);

**if** (result == OK)

**printf**("Traverse succeeded!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Visit is required!\n");

**break**;

**case 20:**

result = LevelOrderTraverse(\*tree, visit);

**if** (result == OK)

**printf**("Traverse succeeded!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Visit is required!\n");

**break**;

**case 21:**

result = FileRead(tree, data\_file\_name);

**if** (result == OK)

**printf**("Successffully load tree from file!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL Pointer passed in!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("You must init tree first!\n");

**else**

**printf**("Can't open file for reading!\n");

**break**;

**case 22:**

result = FileWrite(\*tree, data\_file\_name);

**if** (result == OK)

**printf**("Successffully write tree to file!\n");

**else** **if** (result == TREE\_NOT\_EXIST)

**printf**("Tree don't exist!\n");

**else**

**printf**("Can't open file for writing!\n");

**break**;

**case 23:**

**printf**("Input 1-5 represents each tree:\n");

**scanf**("%d", &tree\_num);

**getchar**();

**if** (tree\_num < 1 || tree\_num > 5)

**printf**("Invalid number!\n");

**else**

tree = &treeArray[tree\_num-1];

**break**;

**case 0:**

op = 0;

**printf**("Press Enter key to exit.\n");

**break**;

}

**getchar**();

}

**printf**("See you next time!\n");

**return** 0;

}

Makefile（Linux和Mac下）

SRC\_DIR = src

INC\_DIR = include

OBJ\_DIR = objs

BIN\_DIR = bin

DIRS := $(OBJ\_DIR) $(BIN\_DIR)

CC = gcc

CFLAGS = -I $(INC\_DIR)

RM = rm -f

MKDIR = mkdir

SRCS = $(**wildcard** ${SRC\_DIR}/**\***.c)

OBJS = $(**patsubst** %.c, ${OBJ\_DIR}/%.o, $(**notdir** ${SRCS}))

HEADERS = $(**wildcard** ${INC\_DIR}/**\***.h)

TARGET = $(BIN\_DIR)/display

all **:** $(DIRS) $(TARGET)

$(DIRS) **:**

$(MKDIR) $@

$(TARGET) **:** $(OBJS)

$(CC) -o $@ $(OBJS)

$(OBJS) **:** $(OBJ\_DIR)/%.o : $(SRC\_DIR)/%.c $(HEADERS)

$(CC) $(CFLAGS) -c $< -o $@

.PHONY **:** clean

clean **:**

$(RM) $(OBJS) $(TARGET)

附录四 基于邻接表的图实现的源程序

Graph-On-Adjacency-List.h

/\* Graph On Adjacency List \*/

#ifndef GRAPH\_ON\_ADJACENCY\_LIST

#define GRAPH\_ON\_ADJACENCY\_LIST

/\* Include Necessary Standard Libraries \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Include Queue Header File\*/

#include "Queue-On-Sequence-Structure.h"

/\* Typedef of Enumeration \*/

**typedef** **enum** Status {OK, NULL\_POINTER = 100, GRAPH\_NOT\_EXIST, MEMORY\_OVERFLOW, EXTERNAL\_FUNCTION\_ERROR, FILE\_ERROR}Status;

**typedef** **enum** Bool {FALSE = 0, TRUE}Bool;

**typedef** **enum** Kind {DirectedGraph, DirectedNet, UndirectedGraph, UndirectedNet}Kind;

/\* Macro Define \*/

#define MAX\_VERTEX\_NUMBER (20) //Defines the max capacity of arc list

/\* Definition Declaration and Typedef of Data Structures and typedef of Function Pointers \*/

**typedef** **struct** Contact {

**char** name[20];

**char** age;

**char** tel[12];

}Contact;

**typedef** Contact VertexType;

**struct** ArcNode {

**int** adjVex;

**struct** ArcNode \* nextArc;

**int** weight;

};

**typedef** **struct** ArcNode \* ArcNode;

**struct** VexNode {

VertexType data;

ArcNode firstArc;

};

**typedef** **struct** VexNode \* VexNode;

**struct** ALGraph {

VexNode vertices[MAX\_VERTEX\_NUMBER];

**int** vexNum, arcNum;

Kind kind;

};

**typedef** **struct** ALGraph \* ALGraph;

**typedef** VertexType \* VexSet;

**struct** ArcSet {

**int** headPos;

**int** tailPos;

**int** weight;

};

**typedef** **struct** ArcSet \* ArcSet;

/\* Function Pointers \*/

**typedef** Status (\* Visit)(VertexType elem);

/\* Definition of Data Operations \*/

Status CreateGraph(ALGraph \* pGraph, Kind kind, VexSet vexSet, **int** vexNum, ArcSet arcSet, **int** arcNum);

Status DestroyGraph(ALGraph \* pGraph);

Status LocateVex(ALGraph graph, VexNode node, **int** \* pPos);

Status GetVex(ALGraph graph, VexNode node, VertexType \* pElem);

Status PutVex(ALGraph graph, VexNode node, VertexType elem);

Status FirstAdjVex(ALGraph graph, VexNode node, VexNode \* pNode);

Status NextAdjVex(ALGraph graph, VexNode curNode, VexNode adjNode, VexNode \* pNextAdjNode);

Status InsertVex(ALGraph graph, VexNode node);

Status DeleteVex(ALGraph graph, VexNode node);

Status InsertArc(ALGraph graph, VexNode curNode, VexNode adjNode, **int** weight);

Status DeleteArc(ALGraph graph, VexNode curNode, VexNode adjNode);

Status DFSTraverse(ALGraph graph, Visit visit);

Status BFSTraverse(ALGraph graph, Visit visit);

/\* Definition of File Operations\*/

Status FileWrite(ALGraph graph, **char** \* filename);

Status FileRead(ALGraph \* pGraph, **char** \* filename);

#endif

Graph-On-Adjacency-List.c

#include "Graph-On-Adjacency-List.h"

/\* Implementation of Operations \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: recursiveDFSTraverse

Description: the recursive function of DFS.

Input:

graph that represents the pointer to the graph.

node that represents the pointer to the vertex

node.

visit that represents the pointer to the visit

function.

visited that represents the markup array.

Return: None

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **void** recursiveDFSTraverse(ALGraph graph, VexNode node, Visit visit, **int** \* visited) {

visit(node->data);

ArcNode arc = node->firstArc;

**while** (arc != **NULL**) {

**if** (visited[arc->adjVex] == 0) {

visited[arc->adjVex] = 1;

recursiveDFSTraverse(graph, (graph->vertices)[arc->adjVex], visit, visited);

}

arc = arc->nextArc;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: CreateGraph

Description: Create a new graph with given arch set,

vertex set and graph kind.

Input:

pGraph represents the pointer to the graph pointer.

kind represents the kind of graph.

vexSet represents the vertex set.

vexNum represents the number of vertices.

arcSet represents the arch set.

arcNum represents the number of arches.

Return: Status variable that represents the state of

this operaiton.

Others: Won't throw exception if using one variable

to create two or more graphs without freeing.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status CreateGraph(ALGraph \* pGraph, Kind kind, VexSet vexSet, **int** vexNum, ArcSet arcSet, **int** arcNum) {

**if** (!pGraph || !vexSet || !arcSet) {

**return** NULL\_POINTER;

}

\*pGraph = **malloc**(**sizeof**(**struct** ALGraph));

**if** (!(\*pGraph)) {

**return** MEMORY\_OVERFLOW;

}

(\*pGraph)->kind = kind;

(\*pGraph)->vexNum = vexNum;

**if** (kind == UndirectedGraph || kind == UndirectedNet) {

(\*pGraph)->arcNum = 2 \* arcNum;

} **else** {

(\*pGraph)->arcNum = arcNum;

}

**int** index;

**for** (index = 0 ; index < vexNum ; ++index) {

((\*pGraph)->vertices)[index] = **malloc**(**sizeof**(**struct** VexNode));

((\*pGraph)->vertices)[index]->data = vexSet[index];

((\*pGraph)->vertices)[index]->firstArc = **NULL**;

}

**for** (index = 0 ; index < arcNum ; ++index) {

VexNode vex = ((\*pGraph)->vertices)[arcSet[index].headPos];

ArcNode arc = **malloc**(**sizeof**(**struct** ArcNode));

**if** (!arc) {

**return** MEMORY\_OVERFLOW;

}

arc->adjVex = arcSet[index].tailPos;

**if** (kind == UndirectedNet || kind == DirectedNet) {

arc->weight = arcSet[index].weight;

}

**if** (vex->firstArc == **NULL**) {

arc->nextArc = **NULL**;

vex->firstArc = arc;

} **else** {

arc->nextArc = vex->firstArc->nextArc;

vex->firstArc->nextArc = arc;

}

**if** (kind == UndirectedGraph || kind == UndirectedNet) {

VexNode peerVex = ((\*pGraph)->vertices)[arcSet[index].tailPos];

ArcNode peerArc = **malloc**(**sizeof**(**struct** ArcNode));

**if** (!peerArc) {

**return** MEMORY\_OVERFLOW;

}

peerArc->adjVex = arcSet[index].headPos;

**if** (kind == UndirectedNet) {

peerArc->weight = arcSet[index].weight;

}

**if** (peerVex->firstArc == **NULL**) {

peerArc->nextArc = **NULL**;

peerVex->firstArc = peerArc;

} **else** {

peerArc->nextArc = peerVex->firstArc->nextArc;

peerVex->firstArc->nextArc = peerArc;

}

}

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DestroyGraph

Description: Destroy the given graph.

Input:

pGraph represents the pointer to the graph pointer.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DestroyGraph(ALGraph \* pGraph) {

**if** (!pGraph) {

**return** NULL\_POINTER;

}

**if** (!\*pGraph) {

**return** GRAPH\_NOT\_EXIST;

}

**int** index;

**for** (index = 0 ; index < (\*pGraph)->vexNum ; ++index) {

VexNode vex = ((\*pGraph)->vertices)[index];

ArcNode arc = vex->firstArc, temp;

**while** (arc) {

temp = arc->nextArc;

**free**(arc);

arc = temp;

}

}

**for** (index = 0 ; index < (\*pGraph)->vexNum ; ++index) {

VexNode vex = ((\*pGraph)->vertices)[index];

**free**(vex);

}

**free** (\*pGraph);

\*pGraph = **NULL**;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: LocateVex

Description: get the position of given vertex.

Input:

graph represents the pointer to the graph.

node represents the pointer to the vex node.

pPos represents the pointer to the position

integer.(Return Value)

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status LocateVex(ALGraph graph, VexNode node, **int** \* pPos) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!node || !pPos) {

**return** NULL\_POINTER;

}

**int** index;

**for** (index = 0 ; index < graph->vexNum ; ++index) {

**if** (node == (graph->vertices)[index]) {

\*pPos = index;

**return** OK;

}

}

\*pPos = -1;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: GetVex

Description: Get the value of given vertex.

Input:

graph represents the pointer to the graph.

node represents the pointer to the vex node.

pElem represents the pointer to the element.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status GetVex(ALGraph graph, VexNode node, VertexType \* pElem) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!node || !pElem) {

**return** NULL\_POINTER;

}

**int** index = -1;

LocateVex(graph, node, &index);

**if** (index >= 0) {

\*pElem = (graph->vertices)[index]->data;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: PutVex

Description: set the value of given vertex.

Input:

graph represents the pointer to the graph.

node represents the pointer to the vex node.

elem represents the element.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status PutVex(ALGraph graph, VexNode node, VertexType elem) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!node) {

**return** NULL\_POINTER;

}

**int** index = -1;

LocateVex(graph, node, &index);

**if** (index >= 0) {

node->data = elem;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FirstAdjVex

Description: Get the first adjacency vertex of given

vertex.

Input:

graph represents the pointer to the graph.

node represents the pointer to the vex node.

pNode represents the pointer to the adjacency

vertex.(Return Value)

Return: Status variable that represents the state of

this operaiton.

Others: pNode could be NULL.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FirstAdjVex(ALGraph graph, VexNode node, VexNode \* pNode) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!node || !pNode) {

**return** NULL\_POINTER;

}

**int** index = -1;

LocateVex(graph, node, &index);

**if** (index >= 0) {

**if** (node->firstArc) {

\*pNode = (graph->vertices)[node->firstArc->adjVex];

} **else** {

\*pNode = **NULL**;

}

} **else** {

\*pNode = **NULL**;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: NextAdjVex

Description: Get next adjacency vertex of current

vertex and one of its adjacency vertex.

Input:

graph represents the pointer to the graph.

node represents the pointer to the vex node.

adjNode represents current adjacency vertex.

pNextAdjNode represents the pointer to the next

adjacency vertex.

Return: Status variable that represents the state of

this operaiton.

Others: pNextAdjNode could be NULL.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status NextAdjVex(ALGraph graph, VexNode curNode, VexNode adjNode, VexNode \* pNextAdjNode) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!curNode || !adjNode || !pNextAdjNode) {

**return** NULL\_POINTER;

}

**int** pos;

LocateVex(graph, adjNode, &pos);

**if** (pos < 0) {

**return** OK;

}

ArcNode arc;

**for** (arc = curNode->firstArc ; arc != **NULL** ; arc = arc->nextArc) {

**if** (arc->adjVex == pos) {

**if** (arc->nextArc) {

\*pNextAdjNode = (graph->vertices)[arc->nextArc->adjVex];

**return** OK;

} **else** {

\*pNextAdjNode = **NULL**;

**return** OK;

}

**break**;

}

}

\*pNextAdjNode = **NULL**;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InsertVex

Description: Create a new vertex and insert it into

the graph.

Input:

graph represents the pointer to the graph.

node represents the pointer to the new vex node.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InsertVex(ALGraph graph, VexNode node) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!node) {

**return** NULL\_POINTER;

}

graph->vertices[graph->vexNum] = **malloc**(**sizeof**(**struct** VexNode));

graph->vertices[graph->vexNum] = node;

graph->vexNum++;

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DeleteVex

Description: Delete the given vertex from the

graph.

Input:

graph represents the pointer to the graph.

node represents the pointer to the vex node

to be deleted.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DeleteVex(ALGraph graph, VexNode node) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!node) {

**return** NULL\_POINTER;

}

**int** pos;

LocateVex(graph, node, &pos);

**if** (pos >= 0) {

**int** index;

VexNode vex;

ArcNode arc, temp;

**for** (index = 0 ; index < graph->vexNum ; ++index) {

vex = (graph->vertices)[index];

arc = vex->firstArc;

**if** (vex == node) {

**while** (arc != **NULL**) {

vex->firstArc = arc->nextArc;

**free**(arc);

arc = vex->firstArc;

}

} **else** {

**if** (vex->firstArc == arc && arc && arc->adjVex == pos) {

vex->firstArc = arc->nextArc;

**free**(arc);

arc = vex->firstArc;

}

**while** (arc && arc->nextArc != **NULL**) {

**if** (arc->nextArc->adjVex == pos) {

temp = arc->nextArc;

arc->nextArc = temp->nextArc;

**free**(temp);

**break**;

}

arc = arc->nextArc;

}

}

}

**free**((graph->vertices)[pos]);

**for** (index = pos ; index < graph->vexNum - 1 ; ++index) {

(graph->vertices)[index] = (graph->vertices)[index + 1];

}

graph->vexNum -= 1;

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: InsertArc

Description: Create a new arch and insert it into

graph.

Input:

graph represents the pointer to the graph.

curNode represents the pointer to the vex node.

adjNode represents the adjacency vertex.

weight represents the weight of arch.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status InsertArc(ALGraph graph, VexNode curNode, VexNode adjNode, **int** weight) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!curNode || !adjNode) {

**return** NULL\_POINTER;

}

**int** existFlag = 0;

ArcNode node;

ArcNode peerNode;

**int** targetPos;

LocateVex(graph, adjNode, &targetPos);

**for** (node = curNode->firstArc ; node && node->nextArc ; node = node->nextArc) {

**if** (node->adjVex == targetPos) {

existFlag = 1;

**break**;

}

}

**if** (!existFlag) {

ArcNode newArc = **malloc**(**sizeof**(**struct** ArcNode));

**if** (!newArc) {

**return** MEMORY\_OVERFLOW;

}

newArc->adjVex = targetPos;

newArc->nextArc = **NULL**;

**if** (graph->kind == DirectedNet || graph->kind == UndirectedNet) {

newArc->weight = weight;

}

**if** (!node) {

curNode->firstArc = newArc;

} **else** {

node->nextArc = newArc;

}

graph->arcNum += 1;

}

**if** (graph->kind == UndirectedGraph || graph->kind == UndirectedNet) {

peerNode = adjNode->firstArc;

**int** pos;

LocateVex(graph, curNode, &pos);

**if** (!existFlag) {

ArcNode peerArc = **malloc**(**sizeof**(**struct** ArcNode));

**if** (!peerArc) {

**return** MEMORY\_OVERFLOW;

}

peerArc->adjVex = pos;

**if** (graph->kind == UndirectedNet) {

peerArc->weight = weight;

}

**if** (!peerNode) {

peerArc->nextArc = **NULL**;

adjNode->firstArc = peerArc;

} **else** {

peerArc->nextArc = peerNode->nextArc;

peerNode->nextArc = peerArc;

}

graph->arcNum += 1;

}

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DeleteArc

Description: Delete the arch from graph.

Input:

graph represents the pointer to the graph.

curNode represents the pointer to the vex node.

(arch head)

adjNode represents the adjacency vertex.

(arch tail)

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DeleteArc(ALGraph graph, VexNode curNode, VexNode adjNode) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!curNode || !adjNode) {

**return** NULL\_POINTER;

}

**int** targetPos;

LocateVex(graph, adjNode, &targetPos);

**int** existFlag = 0;

ArcNode node = curNode->firstArc;

ArcNode peerNode;

**if** (!node) {

**return** OK;

}

ArcNode temp;

**if** (node->adjVex == targetPos) {

existFlag = 1;

curNode->firstArc = node->nextArc;

**free**(node);

graph->arcNum -= 1;

} **else** {

**for** (; node->nextArc != **NULL** ; node = node->nextArc) {

**if** (node->nextArc->adjVex == targetPos) {

existFlag = 1;

temp = node->nextArc;

node->nextArc = temp->nextArc;

**free**(temp);

graph->arcNum -= 1;

**break**;

}

}

}

**if** (graph->kind == UndirectedGraph || graph->kind == UndirectedNet) {

**if** (existFlag) {

peerNode = adjNode->firstArc;

**int** pos;

LocateVex(graph, curNode, &pos);

**if** (peerNode->adjVex == pos) {

adjNode->firstArc = peerNode->nextArc;

**free**(peerNode);

graph->arcNum -= 1;

} **else** {

**for** (; peerNode->nextArc != **NULL** ; peerNode = peerNode->nextArc) {

**if** (peerNode->nextArc->adjVex == pos) {

temp = peerNode->nextArc;

peerNode->nextArc = temp->nextArc;

**free**(temp);

graph->arcNum -= 1;

**break**;

}

}

}

}

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DFSTraverse

Description: Traverse the graph according to

depth first search.

Input:

graph represents the pointer to the graph.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status DFSTraverse(ALGraph graph, Visit visit) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!visit) {

**return** NULL\_POINTER;

}

**int** index, visited[graph->vexNum];

**for** (index = 0 ; index < graph->vexNum ; ++index) {

visited[index] = 0;

}

**for** (index = 0 ; index < graph->vexNum ; ++index) {

**if** (visited[index] == 0) {

visited[index] = 1;

recursiveDFSTraverse(graph, (graph->vertices)[index], visit, visited);

}

}

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: BFSTraverse

Description: Traverse the graph according to

breadth first search.

Input:

graph represents the pointer to the graph.

visit represents the external visit function.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status BFSTraverse(ALGraph graph, Visit visit) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!visit) {

**return** NULL\_POINTER;

}

**int** index, visited[graph->vexNum];

**for** (index = 0 ; index < graph->vexNum ; ++index) {

visited[index] = 0;

}

Queue queue = CreateQueue(MAX\_VERTEX\_NUMBER);

**for** (index = 0 ; index < graph->vexNum ; ++index) {

**if** (!visited[index]) {

visited[index] = 1;

visit((graph->vertices)[index]->data);

Enqueue((graph->vertices)[index], queue);

**while** (!IsQueueEmpty(queue)) {

VexNode node = FrontAndDequeue(queue);

VexNode node2;

**int** index;

**for** (FirstAdjVex(graph, node, &node2) ; node2 != **NULL** ; NextAdjVex(graph, node, node2, &node2)) {

LocateVex(graph, node2, &index);

**if** (index >= 0 && !visited[index]) {

visited[index] = 1;

visit(node2->data);

Enqueue(node2, queue);

}

}

}

}

}

DisposeQueue(queue);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileWrite

Description: Write the graph into the file.

Input:

graph represents the pointer to the graph.

filename represents the name of file.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileWrite(ALGraph graph, **char** \* filename) {

**if** (!graph) {

**return** GRAPH\_NOT\_EXIST;

}

**if** (!filename) {

**return** NULL\_POINTER;

}

**FILE** \* fpw = **fopen**(filename, "wb");

**if** (!fpw) {

**return** FILE\_ERROR;

}

**int** index, count = 0;

VexNode vex;

ArcNode arc;

**fwrite**(&(graph->kind), **sizeof**(**int**), 1, fpw);

**fwrite**(&(graph->vexNum), **sizeof**(**int**), 1, fpw);

**fwrite**(&(graph->arcNum), **sizeof**(**int**), 1, fpw);

VertexType vexSet[graph->vexNum];

**struct** ArcSet arcSet[graph->arcNum];

**for** (index = 0 ; index < graph->vexNum ; ++index) {

vex = (graph->vertices)[index];

vexSet[index] =vex->data;

**for** (arc = vex->firstArc; arc != **NULL** ; arc = arc->nextArc) {

arcSet[count].headPos = index;

arcSet[count].tailPos = arc->adjVex;

arcSet[count].weight = arc->weight;

count++;

}

}

**fwrite**(vexSet, **sizeof**(vexSet), 1, fpw);

**fwrite**(arcSet, **sizeof**(arcSet), 1, fpw);

**fclose**(fpw);

**return** OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FileRead

Description: Read the graph from the file.

Input:

pGraph represents the pointer to the graph

pointer.

filename represents the name of file.

Return: Status variable that represents the state of

this operaiton.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Status FileRead(ALGraph \* pGraph, **char** \* filename) {

**if** (!pGraph || !filename) {

**return** NULL\_POINTER;

}

**FILE** \* fpr = **fopen**(filename, "rb");

**if** (!fpr) {

**return** FILE\_ERROR;

}

Kind kind;

**int** vexNum, arcNum;

**fread**(&kind, **sizeof**(**int**), 1, fpr);

**fread**(&vexNum, **sizeof**(**int**), 1, fpr);

**fread**(&arcNum, **sizeof**(**int**), 1, fpr);

VertexType vexSet[vexNum];

**struct** ArcSet arcSet[arcNum];

**fread**(vexSet, **sizeof**(vexSet), 1, fpr);

**fread**(arcSet, **sizeof**(arcSet), 1, fpr);

DestroyGraph(pGraph);

**int** result = OK;

result = CreateGraph(pGraph, kind, vexSet, vexNum, arcSet, arcNum);

**fclose**(fpr);

**return** result;

}

Queue-On-Sequence-Structure.h

/\* Queue on Sequence Structure \*/

#ifndef QUEUE\_ON\_SEQUENCE\_STRUCTURE\_H

#define QUEUE\_ON\_SEQUENCE\_STRUCTURE\_H

/\* Include Necessary Standard Library \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

/\* Include Binary Tree Structure \*/

#include "Graph-On-Adjacency-List.h"

/\* Definition and Typedef of Data and Structure and typedef of Function Pointers \*/

**typedef** **struct** VexNode \* Element;

**struct** Queue;

**typedef** **struct** Queue \* Queue;

/\* Definition of Data Operations \*/

**int** IsQueueEmpty(Queue Q);

**int** IsQueueFull(Queue Q);

Queue CreateQueue(**int** capacity);

**void** DisposeQueue(Queue Q);

**void** MakeQueueEmpty(Queue Q);

**void** Enqueue(Element X, Queue Q);

**void** Dequeue(Queue Q);

Element Front(Queue Q);

Element FrontAndDequeue(Queue Q);

#endif

Queue-On-Sequence-Structure.c

#include "Queue-On-Sequence-Structure.h"

/\* Implementation of Operations \*/

**struct** Queue {

**int** capacity;

**int** rear;

**int** front;

**int** size;

Element \* elemArray;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Succ

Description: return the succ of the front/rear of queue.

Input:

value represents the index of certain element of queue.

Q represents the queue variable.

Return:

int represents the succ.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**static** **int** Succ(**int** value, Queue Q) {

**if** (++value == Q->capacity)

value = 0;

**return** value;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: IsQueueEmpty

Description: judge if the queue is empty

Input:

Q represents the queue variable.

Return:

int variable represents the judge result.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**int** IsQueueEmpty(Queue Q) {

**return** (Q->size == 0);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: IsQueueFull

Description: judge if the queue is full.

Input:

Q represents the queue variable.

Return:

int variable represents the judge result.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**int** IsQueueFull(Queue Q) {

**return** (Q->size == Q->capacity);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: CreateQueue

Description: create an empty queue.

Input:

capacity represents the capacity of queue.

Return:

Queue variable represents the created queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Queue CreateQueue(**int** capacity) {

**if** (capacity > 0) {

Queue Q = **malloc**(**sizeof**(**struct** Queue));

**if** (!Q) **exit**(-1);

Q->capacity = capacity;

Q->elemArray = **malloc**(**sizeof**(Element)\*capacity);

**if** (!Q->elemArray) **exit**(-1);

MakeQueueEmpty(Q);

**return** Q;

} **else** {

**return** **NULL**;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: DisposeQueue

Description: dispose the given tree.

Input:

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** DisposeQueue(Queue Q) {

**if** (Q) {

**if** (Q->elemArray) **free**(Q->elemArray);

**free**(Q);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: MakeQueueEmpty

Description: clear the given queue.

Input:

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** MakeQueueEmpty(Queue Q) {

Q->size = 0;

Q->front = 1;

Q->rear = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Enqueue

Description: enqueue the certain element to given queue.

Input:

X represents the element variable.

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** Enqueue(Element X, Queue Q) {

**if** (IsQueueFull(Q))

**return**;

**else** {

Q->size++;

Q->rear = Succ(Q->rear, Q);

Q->elemArray[Q->rear] = X;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Dequeue

Description: dequeue the certain queue.

Input:

Q represents the queue variable.

Return:

None.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**void** Dequeue(Queue Q) {

**if** (IsQueueEmpty(Q))

**return**;

**else** {

Q->size--;

Q->front = Succ(Q->front, Q);

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: Front

Description: return the front of given queue.

Input:

Q represents the queue variable.

Return:

Element represents the front element of queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Element Front(Queue Q) {

**if** (IsQueueEmpty(Q))

**return** **NULL**;

**else** {

**return** Q->elemArray[Q->front];

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Function: FrontAndDequeue

Description: front and then dequeue the given queue.

Input:

Q represents the queue variable.

Return:

Element represents the front element of queue.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Element FrontAndDequeue(Queue Q) {

**if** (IsQueueEmpty(Q))

**return** **NULL**;

**else** {

Element temp = Q->elemArray[Q->front];

Dequeue(Q);

**return** temp;

}

}

Display.c

/\* Include Necessary Standard Libraries and Header Files \*/

#include <stdlib.h>

#include <string.h>

#include "Graph-On-Adjacency-List.h"

/\* Global variables \*/

**char** \* data\_file\_name = "adjacency\_list\_display.dat";

/\* Implementation of Function Pointers \*/

Status visit(Contact elem) {

**printf**(" ------------------------------ \n");

**printf**("| Name: %22s |\n", elem.name);

**printf**("| Age: %23hhd |\n", elem.age);

**printf**("| Tel: %23s |\n", elem.tel);

**printf**(" ------------------------------ \n");

**printf**("\n");

**return** OK;

}

/\* Program Entry \*/

**int** **main**(**void**) {

**int** op=1;

ALGraph graph = **NULL**;

**int** count, vexNum, arcNum;

Status result;

**int** index, index2;

**int** weight;

VexNode node = **NULL**, node2 = **NULL**;

Contact elem;

**while**(op){

**system**("clear");

**printf**(" Menu for Graph On Adjacency List \n");

**printf**("--------------------------------------------------\n");

**printf**(" 1. CreateGraph 9. DeleteVex\n");

**printf**(" 2. DestroyGraph 10. InsertArc\n");

**printf**(" 3. LocateVex 11. DeleteArc\n");

**printf**(" 4. GetVex 12. DFSTraverse\n");

**printf**(" 5. PutVex 13. BFSTraverse\n");

**printf**(" 6. FirstAdjVex 14. FileRead\n");

**printf**(" 7. NextAdjVex 15. FileWrite\n");

**printf**(" 8. InsertVex\n");

**printf**(" 0. Exit\n");

**printf**("--------------------------------------------------\n");

**printf**(" Choose from [0~15]:");

**scanf**("%d",&op);

**getchar**();

**switch**(op){

**case 1:**

**printf**("Input the kind of graph - DirectedGraph, DirectedNet, UndirectedGraph, UndirectedNet[0-3]:");

**scanf**("%d", &count);

**getchar**();

**if** (count < 0 || count > 3) {

**printf**("Out of range!\n");

} **else** {

**printf**("Input the number of vertices:");

**scanf**("%d", &vexNum);

**getchar**();

Contact contact[vexNum];

**for** (index = 0 ; index < vexNum ; ++index) {

**printf**("Please input the name of given contact:");

**scanf**("%s", elem.name);

**getchar**();

**printf**("Please input the age of given contact:");

**scanf**("%hhd", &elem.age);

**getchar**();

**printf**("Please input the tel of given contact:");

**scanf**("%s", elem.tel);

**getchar**();

contact[index] = elem;

}

**printf**("Input the number of arches:");

**scanf**("%d", &arcNum);

**getchar**();

**struct** ArcSet arcSet[arcNum];

**for** (index = 0 ; index < arcNum ; ++index) {

**printf**("Please input the index of head position:");

**scanf**("%d", &arcSet[index].headPos);

**getchar**();

**printf**("Please input the index of tail position:");

**scanf**("%d", &arcSet[index].tailPos);

**getchar**();

**if** (count == DirectedNet || count == UndirectedNet) {

**printf**("Please input the weight of arch:");

**scanf**("%d", &arcSet[index].weight);

**getchar**();

}

}

result = CreateGraph(&graph, count, contact, vexNum, arcSet, arcNum);

**if**(result == OK)

**printf**("Successfully created graph!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Failed to allocate memory for graph!\n");

}

**break**;

**case 2:**

result = DestroyGraph(&graph);

**if** (result == OK)

**printf**("Successfully destroyed graph!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Graph don't exist!\n");

**break**;

**case 3:**

**printf**("Please input the position number:");

**scanf**("%d", &index);

**getchar**();

**if** (graph) {

**if** (index >= 0 && index < graph->vexNum) {

result = LocateVex(graph, graph->vertices[index], &index2);

**if** (result == OK) {

**printf**("The position you input is: %d and the result is: %d, now set Node to it.", index, index2);

node = graph->vertices[index2];

}

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

} **else** {

**printf**("Index out of range!\n");

}

}

**break**;

**case 4:**

result = GetVex(graph, node, &elem);

**if** (result == OK)

visit(elem);

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**break**;

**case 5:**

**printf**("Please input the name of given contact:");

**scanf**("%s", elem.name);

**getchar**();

**printf**("Please input the age of given contact:");

**scanf**("%hhd", &elem.age);

**getchar**();

**printf**("Please input the tel of given contact:");

**scanf**("%s", elem.tel);

**getchar**();

result = PutVex(graph, node, elem);

**if** (result == OK)

**printf**("Set value succeeded!\n");

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

**break**;

**case 6:**

result = FirstAdjVex(graph, node, &node2);

**if** (result == OK) {

**printf**("Node2 Set, and the first adjacency vertex of given vertex is:");

**if** (node2)

visit(node2->data);

**else**

**printf**("NULL\n");

}

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

**break**;

**case 7:**

result = NextAdjVex(graph, node, node2, &node2);

**if** (result == OK) {

**printf**("Node2 set, and the next adjacency vertex of given vertex is:");

**if** (node2)

visit(node2->data);

**else**

**printf**("NULL\n");

} **else** **if** (result == GRAPH\_NOT\_EXIST){

**printf**("Graph don't exist!\n");

} **else**

**printf**("NULL pointer passed in!\n");

**break**;

**case 8:**

**printf**("Please input the name of given contact:");

**scanf**("%s", elem.name);

**getchar**();

**printf**("Please input the age of given contact:");

**scanf**("%hhd", &elem.age);

**getchar**();

**printf**("Please input the tel of given contact:");

**scanf**("%s", elem.tel);

**getchar**();

VexNode newNode;

newNode->data = elem;

newNode->firstArc = **NULL**;

result = InsertVex(graph, newNode);

**if** (result == OK)

**printf**("Insert vertex succeeded!\n");

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

**break**;

**case 9:**

**printf**("Please input the index of node:\n");

**scanf**("%d", &index);

**getchar**();

**if** (graph) {

**if** (index >= 0 && index < graph->vexNum) {

result = DeleteVex(graph, (graph->vertices)[index]);

**if** (result == OK)

**printf**("Delete vertex succeeded!\n");

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

} **else** {

**printf**("Index out of range!\n");

}

}

**break**;

**case 10:**

**printf**("Please input the index of node1 and node2:");

**scanf**("%d%d", &index, &index2);

**getchar**();

weight = -1;

**if** (graph) {

**if** (graph->kind == UndirectedNet || graph->kind == DirectedNet) {

**printf**("Pleas input the weight of the arc:");

**scanf**("%d", &weight);

**getchar**();

}

**if** (index >= 0 && index < graph->vexNum && index2 >= 0 && index2 < graph->vexNum && index != index2) {

result = InsertArc(graph, (graph->vertices)[index], (graph->vertices)[index2], weight);

**if** (result == OK)

**printf**("Insert arch succeeded!\n");

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Failed to allocate memory for new arch!\n");

}

}

**break**;

**case 11:**

**printf**("Please input the index of node1 and node2:");

**scanf**("%d%d", &index, &index2);

**getchar**();

**if** (graph) {

result = DeleteArc(graph, (graph->vertices)[index], (graph->vertices)[index2]);

**if** (result == OK)

**printf**("Delete arch succeeded\n");

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

}

**break**;

**case 12:**

result = DFSTraverse(graph, &visit);

**if** (result == OK) {

**printf**("\n");

**printf**("The traverse operation succeeded!\n");

} **else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

**break**;

**case 13:**

result = BFSTraverse(graph, &visit);

**if** (result == OK) {

**printf**("\n");

**printf**("The traverse operation succeeded!\n");

}

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("Graph don't exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

**break**;

**case 14:**

result = FileRead(&graph, data\_file\_name);

**if** (result == OK)

**printf**("Data load succeeded!\n");

**else** **if** (result == FILE\_ERROR)

**printf**("Fail to open data file for reading!\n");

**else** **if** (result == NULL\_POINTER)

**printf**("NULL pointer passed in!\n");

**else**

**printf**("Failed to allocate memory for graph!\n");

**break**;

**case 15:**

result = FileWrite(graph, data\_file\_name);

**if** (result == OK)

**printf**("Data save succeeded!\n");

**else** **if** (result == FILE\_ERROR)

**printf**("Fail to open data file for writing!\n");

**else** **if** (result == GRAPH\_NOT\_EXIST)

**printf**("You should NOT write a graph that DON'T exist!\n");

**else**

**printf**("NULL pointer passed in!\n");

**break**;

**case 0:**

op = 0;

**printf**("Press Enter key to exit.\n");

**break**;

}

**getchar**();

}

**printf**("See you next time!\n");

**return** 0;

}

Makefile（Linux和Mac下）

SRC\_DIR = src

INC\_DIR = include

OBJ\_DIR = objs

BIN\_DIR = bin

DIRS := $(OBJ\_DIR) $(BIN\_DIR)

CC = gcc

CFLAGS = -I $(INC\_DIR)

RM = rm -f

MKDIR = mkdir

SRCS = $(**wildcard** ${SRC\_DIR}/**\***.c)

OBJS = $(**patsubst** %.c, ${OBJ\_DIR}/%.o, $(**notdir** ${SRCS}))

HEADERS = $(**wildcard** ${INC\_DIR}/**\***.h)

TARGET = $(BIN\_DIR)/display

all **:** $(DIRS) $(TARGET)

$(DIRS) **:**

$(MKDIR) $@

$(TARGET) **:** $(OBJS)

$(CC) -o $@ $(OBJS)

$(OBJS) **:** $(OBJ\_DIR)/%.o : $(SRC\_DIR)/%.c $(HEADERS)

$(CC) $(CFLAGS) -c $< -o $@

.PHONY **:** clean

clean **:**

$(RM) $(OBJS) $(TARGET)