

**课 程 实 验 报 告**

**课程名称： 数据结构实验**

**专业班级： 计算机科学与技术1607**

**学 号： U201614700**

**姓 名： 王亚宁**

**指导教师： 李剑军**

**报告日期： 2017年 12月 1 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 2](#_Toc501057144)

[1.1 问题描述 2](#_Toc501057145)

[1.2 系统设计 2](#_Toc501057146)

[1.3 系统实现 3](#_Toc501057147)

[1.3.1代码实现 3](#_Toc501057148)

[1.3.2程序测试 3](#_Toc501057149)

[1.4 实验小结 16](#_Toc501057150)

[2 基于链式存储结构的线性表实现 17](#_Toc501057151)

[2.1 问题描述 17](#_Toc501057152)

[2.2 系统设计 17](#_Toc501057153)

[2.3 系统实现 18](#_Toc501057154)

[2.3.1代码实现 18](#_Toc501057155)

[2.3.2实例测试 18](#_Toc501057156)

[2.4 实验小结 27](#_Toc501057157)

[3 基于二叉链表的二叉树实现 29](#_Toc501057158)

[3.1 问题描述 29](#_Toc501057159)

[3.2 系统设计 29](#_Toc501057160)

[3.3 系统实现 29](#_Toc501057161)

[3.4 实验小结 29](#_Toc501057162)

[4 基于邻接表的图实现 30](#_Toc501057163)

[4.1 问题描述 30](#_Toc501057164)

[4.2 系统设计 30](#_Toc501057165)

[4.3 系统实现 30](#_Toc501057166)

[4.4 实验小结 30](#_Toc501057167)

[参考文献 31](#_Toc501057168)

[附录A 基于顺序存储结构线性表实现的源程序 34](#_Toc501057169)

[附录B 基于链式存储结构线性表实现的源程序 46](#_Toc501057170)

[附录C 基于二叉链表二叉树实现的源程序 60](#_Toc501057171)

[附录D 基于邻接表图实现的源程序 62](#_Toc501057172)

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

基于内存中的连续空间实现线性表，依据最小完备性和常用性相结合的原则，以函数形式定义了线性表的初始化表、销毁表、清空表、判定空表、求表长和获得元素等12种基本运算。

## 1.2 系统设计

实现所需要的几种功能：初始化表、销毁表、清空表、判断空表、求表长、根据位置获取元素、查找元素、获得前驱、获得后驱、插入元素、删除元素和遍历表，根据实际需求我还自己添加了从文件初始化和将表保存到文件的功能。下面是具体的每个函数声明。

由于我使用的是C++，所以函数在使用时不需要传入表参数，直接使用表名进行调用即可，在构造函数中我将表的参数初始化，并没有为其分配空间，实际的函数如下。

1. 初始化表：status InitaList(),使用对象名调用，不需参数。当表已经存在时不进行初始化，输出错误信息并返回，其中我添加了从文件初始化的选项。返回值为初始化结果。
2. 销毁表：status DestroyList()，使用对象名调用，不需参数，当表未初始化或者已经销毁过的时候返回错误。返回值为是否销毁成功。
3. 清空表：函数名称是ClearList()；初始条件是线性表已存在；操作结果是将表重置为空表。
4. 判定空表：函数名称是ListEmpty()；初始条件是线性表已存在；操作结果是若表为空表则返回true,否则返回false。
5. 求表长：函数名称是ListLength()；初始条件是线性表已存在；操作结果是返回表中数据元素的个数。
6. 获得元素：函数名称是GetElem(i,e)；初始条件是线性表已存在，1≤i≤ListLength()；操作结果是用e返回表中第i个数据元素的值。
7. 查找元素：函数名称是LocateElem(e, comp)；初始条件是线性表已存在；操作结果是返回L中第1个与e满足关系comp（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。
8. 获得前驱：函数名称是PriorElem(cur\_e,pre\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是第一个，则用pre\_e返回它的前驱，否则操作失败，pre\_e无定义。
9. 获得后继：函数名称是NextElem(L,cur\_e,next\_e)；初始条件是线性表L已存在；操作结果是若cur\_e是L的数据元素，且不是最后一个，则用next\_e返回它的后继，否则操作失败，next\_e无定义。
10. 插入元素：函数名称是ListInsert(L,i,e)；初始条件是线性表L已存在且非空，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的数据元素e。
11. 删除元素：函数名称是ListDelete(L,i,e)；初始条件是线性表L已存在且非空，1≤i≤ListLength(L)；操作结果：删除L的第i个数据元素，用e返回其值。
12. 遍历表：函数名称是ListTraverse(L,visit)，初始条件是线性表L已存在；操作结果是依次对L的每个数据元素调用函数visit()。

## 1.3 系统实现

1.3.1代码实现

代码实现请见附带文件。

1.3.2程序测试

对程序的测试如下：

首先是表的初始化，使用文件初始化，首先读入元素的个数，然后将元素一次读入表中，见图1-1：

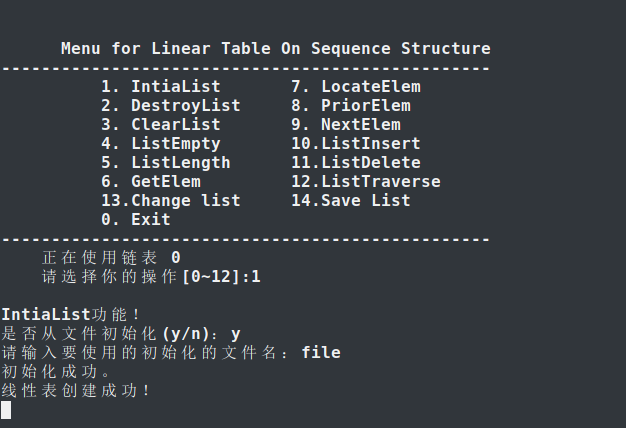


图1-1

初始化完成后元素如图1-2：



图1-2

检测一下链表现在是否为空，见图1-3：

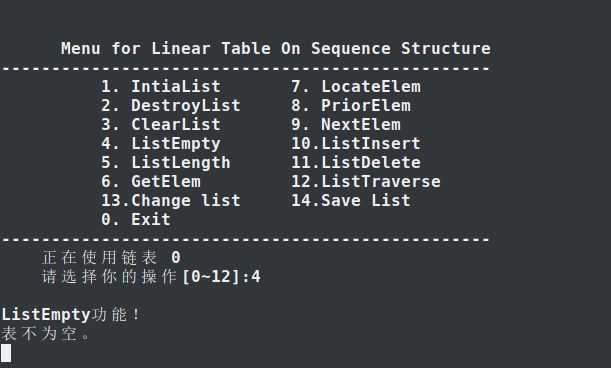


图1-3

得到表的长度，见图1-4：

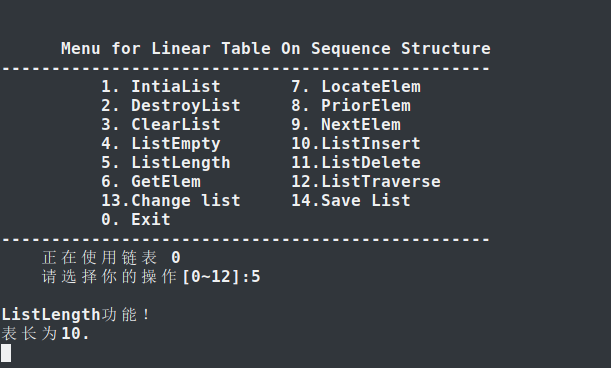


图1-4

使用GetElem函数获取元素，见图1-5,1-6,1-7：

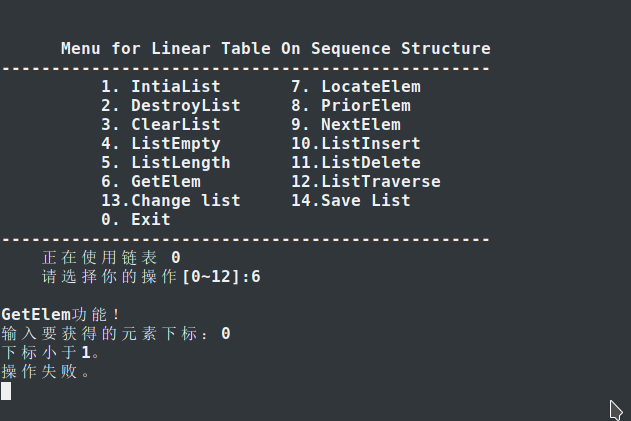


图1-5

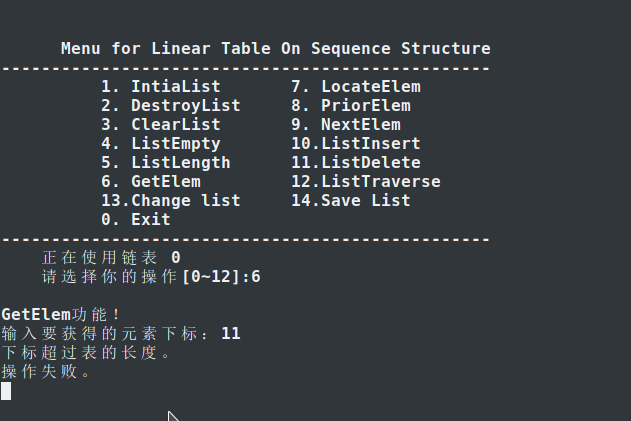


图1-6

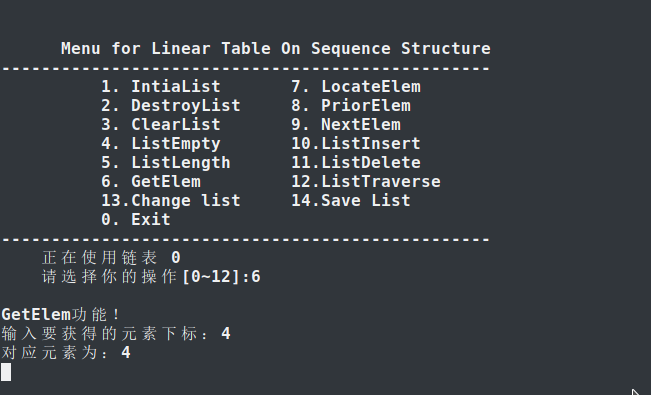


图1-7

定位元素的操作见图1-8,1-9,1-10：

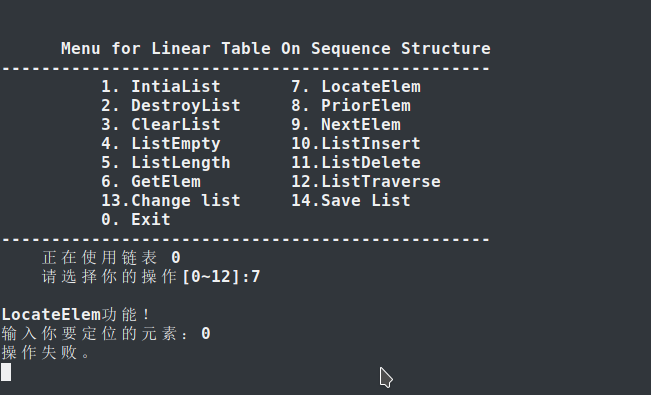


图1-8

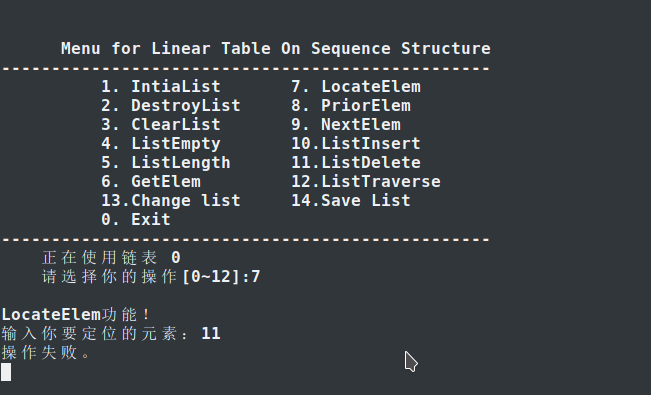


图1-9

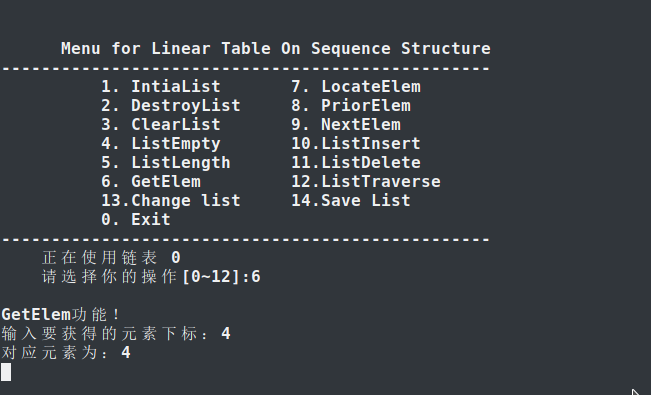


图1-10

获得前驱的操作见图1-11,1-12：

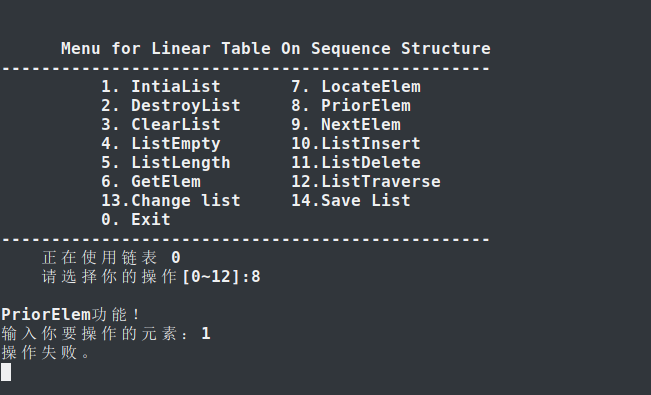


图1-11

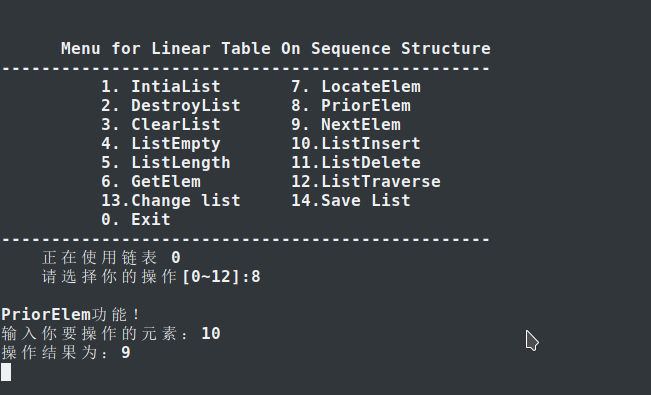


图1-12

获得后驱的函数测试1-13,1-14：

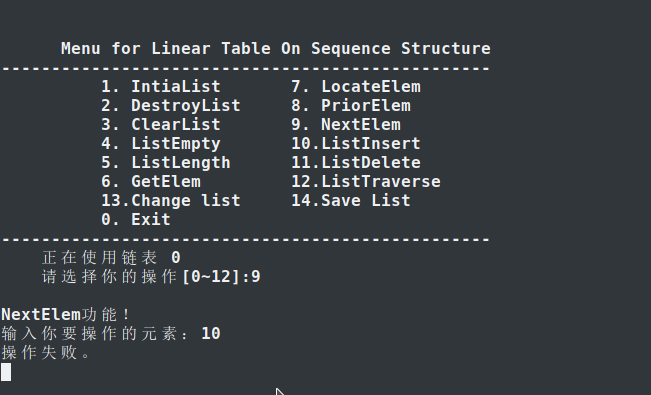


图1-13

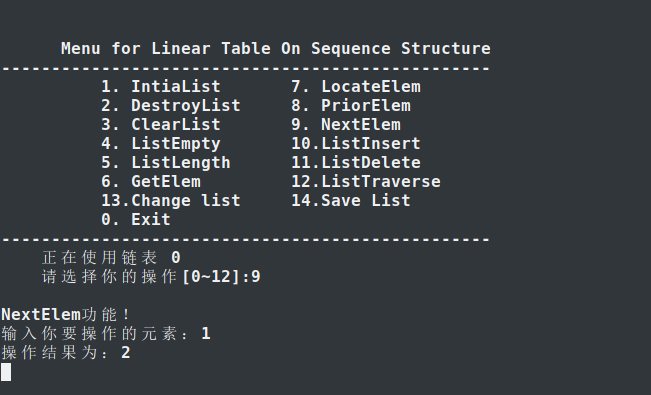


图1-14

元素插入测试见图1-15,1-16，1-17：

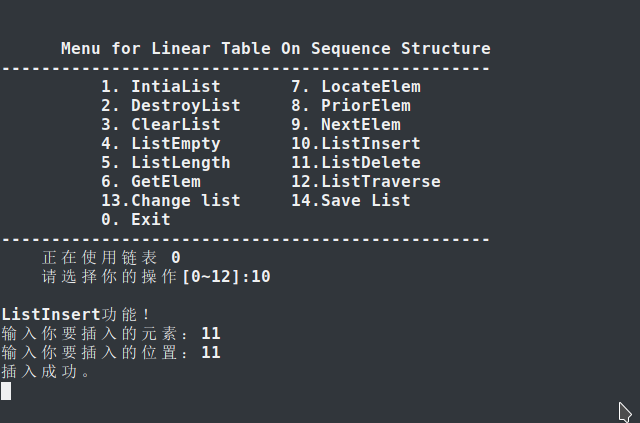


图1-15



图1-16

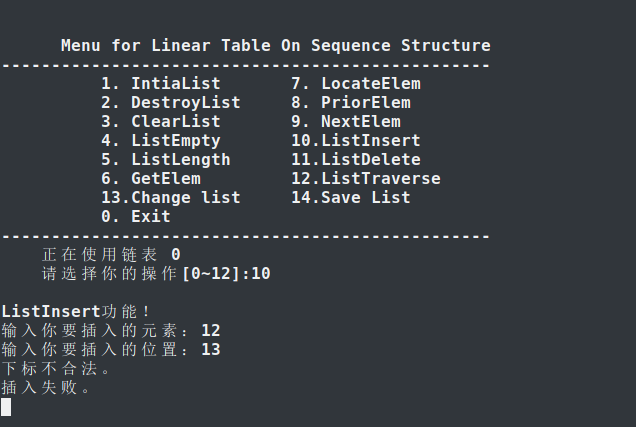


图1-17

元素删除测试如下图1-18,1-19,1-20：

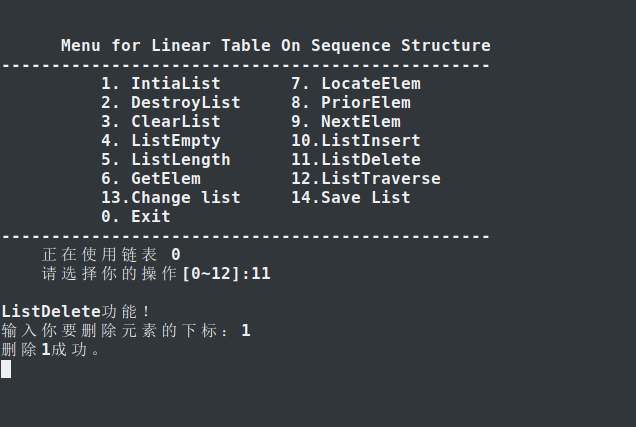


图1-18



图1-19

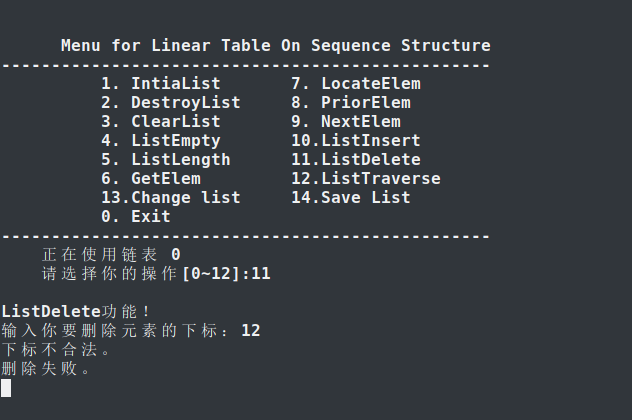


图1-20

保存链表测试见图1-21,1-22：

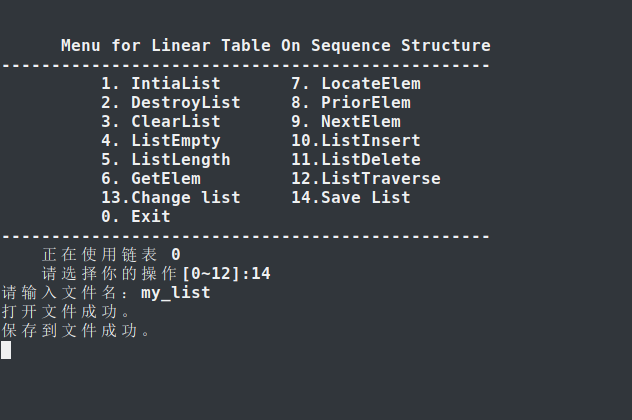


图1-21

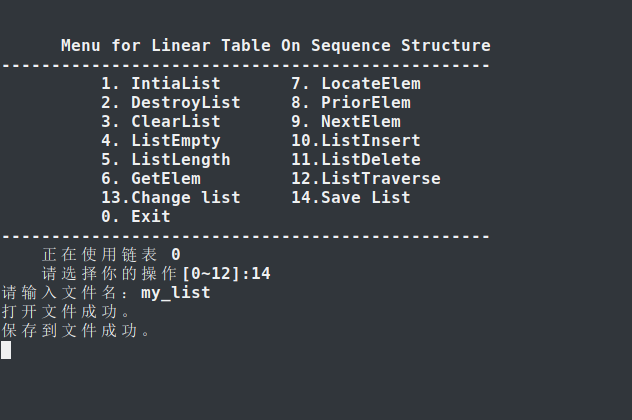


图1-22

切换链表操作见图1-23,1-24,1-25：

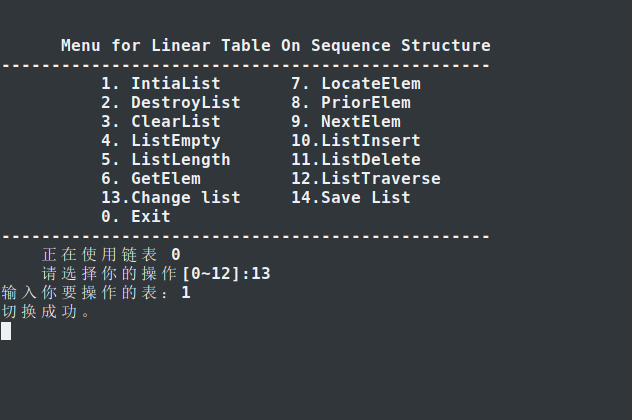


图1-23

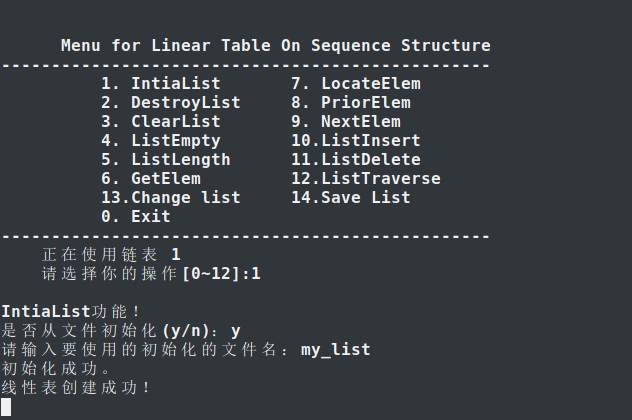


图1-24



图1-24

## 1.4 实验小结

通过自己实现线性表，我发现了一些在程序设计中比较重要的问题，在设计循环的时候要特别注意边缘检测，良好的测试数据和测试习惯也很重要。对自己的程序设计能力有很大的提升。

# 2 基于链式存储结构的线性表实现

## 2.1 问题描述

本次实验主要是使用链表来实现一个线性表，主要涉及初始化、插入、删除等操作，操作和上次的基于顺序表的线性表的操作是一样的。

## 2.2 系统设计

使用链表实现线性表，主要就是使用链表和相应的函数完成对和线性表一样的操作，首先需要设计一个结构体作为结点类型：

首先是初始化操作，初始化操作首先判断此表是否已经进行了初始化，如果已经进行了就返回错误，否则进行空间开辟和变量初始化等操作。

摧毁表，首先判断表是否未初始化（由于判断初始化操作太普遍，以下操作说明时省略此步判断，代码中会书写），未初始化返回错误，不能进行摧毁，否则进行摧毁操作释放空间并设置变量值，初始化状态更改为false。

清空表，清空表就是摧毁表后将初始化状态设置为true，对于链表不需要为后续元素预留空间。

判定空表，空表就是长度为0的表，返回长度是否为0即可，表未初始化也返回true。

求表长操作，未初始化的表长度默认为0，已经初始化的表长度即为表长变量的值，返回变量即可。

获得元素操作，需要根据给定的下标来返回对应的元素，首先应该判断给定的下表是否合法，不合法表示元素不可能存在，返回错误标志，若下表合理就对从表头指针向后移动适当距离即可得到需要的元素所在的结构指针，返回元素即可。

获得前驱元素，判断空表，然后将表头指针向后移动，并判断指针的后继所指的结构体的元素是否为和给定的元素相等，相等就返回当前指针的元素，即为所求前驱。

获得后继，操作大部分与获得前驱相同，区别在于判断的是当前元素是否与给定相同，相同返回next指针指向的元素。

插入操作，首先判断下标合法性，需要注意插入到头指针的时head指针需要改变，普通下标插入后移适当位置到达其前驱结点处即可，再进行插入，完成后长度加一即可。

删除操所，和插入操作大部分类似，删除后长度减一。

遍历表，使用while或者for循环即可。

## 2.3 系统实现

2.3.1代码实现

系统的代码实现可见附带文件。

2.3.2实例测试

进行测试：首先测试初始化功能，使用自带文件初始化，如图2-1：

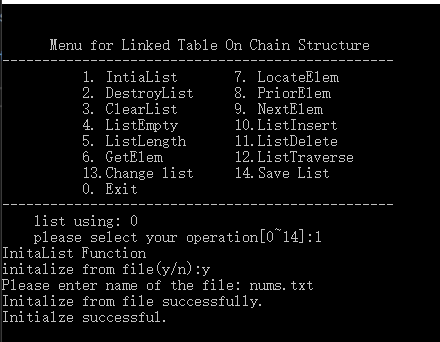


图2-1

初始化成功，接下来检测表是否为空，如图2-1：

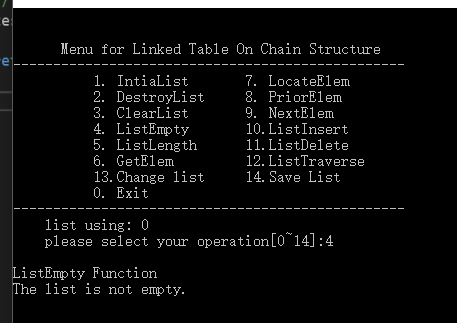


图2-1

获取表长操作见图2-3：

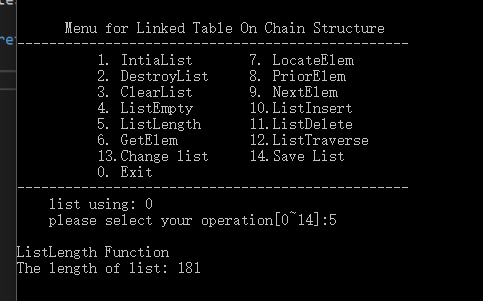


图2-3

获取指定下标的元素，如图2-4，2-5：

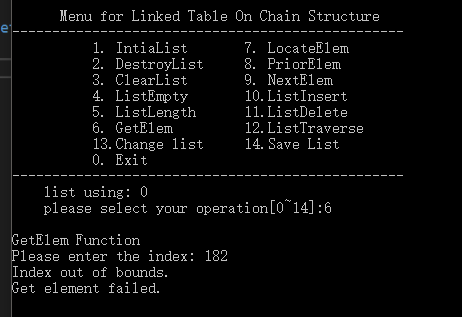


图2-4

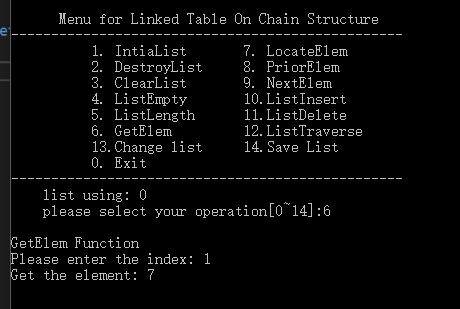


图2-5

定位元素，见图2-6、2-7：

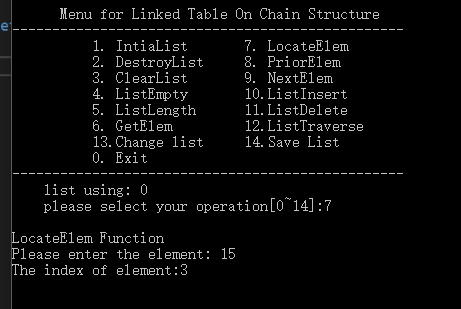


图2-6

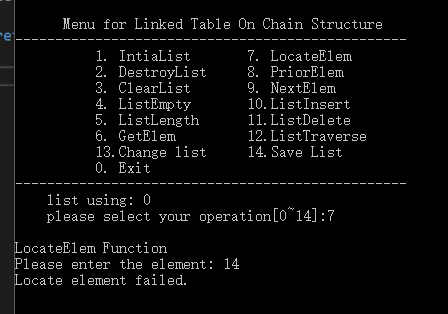


图2-7

测试获取前驱，如图2-8、2-9：

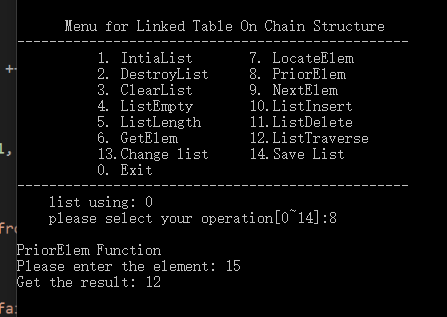


图2-8

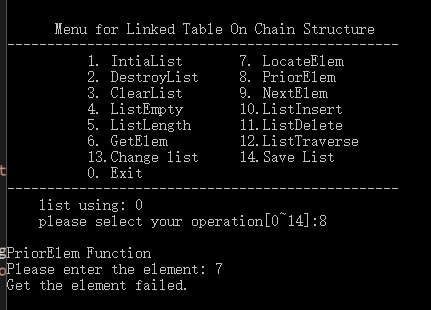


图2-9

获得后继函数测试，见图2-10、2-11：

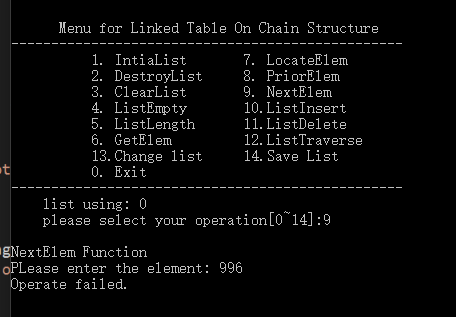


图2-10

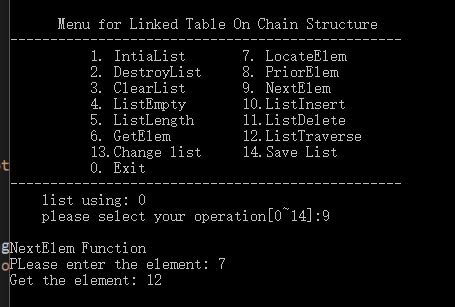


图2-11

插入函数测试，见图2-12，图2-13，图2-14：

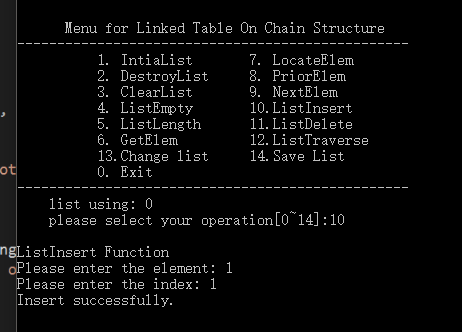


图2-12

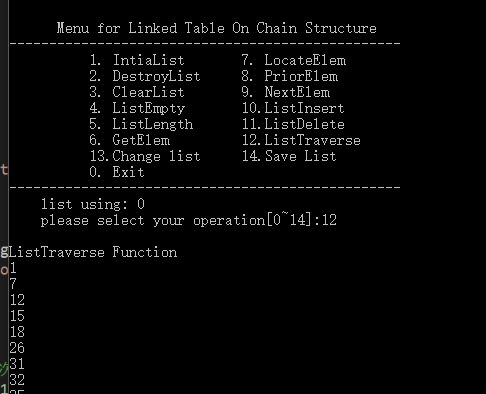


图2-13

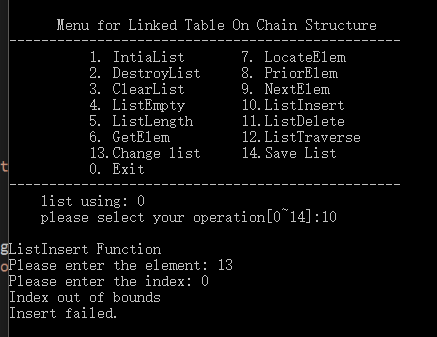


图2-14

对链表进行删除元素如图2-15、图2-16：

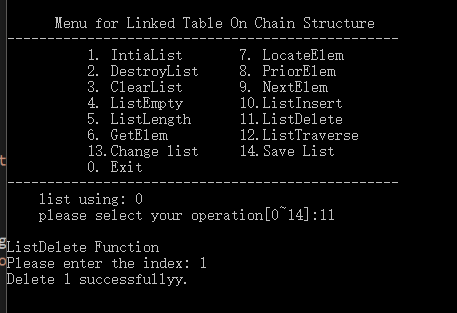


图2-15

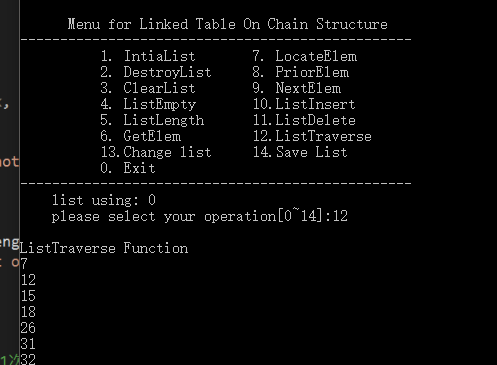


图2-16

链表遍历如图2-17、2-18：

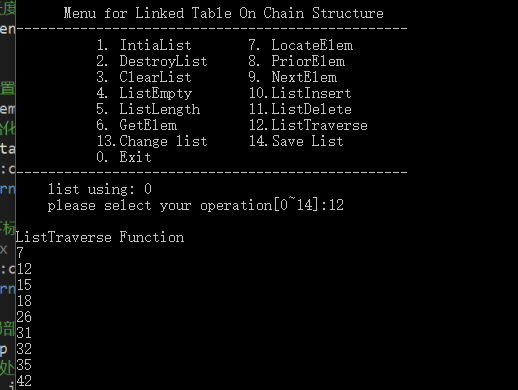


图2-17

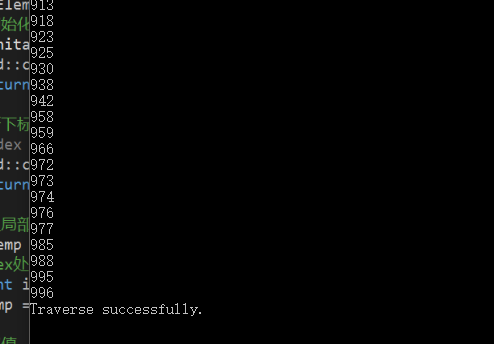


图2-18

将表保存到文件，如图2-19：

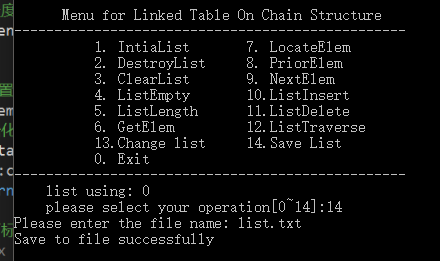


图2-19

通过ChangeList可以切换链表。

## 2.4 实验小结

# 3 基于二叉链表的二叉树实现

## 3.1 问题描述

## 3.2 系统设计

## 3.3 系统实现

## 3.4 实验小结

# 4 基于邻接表的图实现

## 4.1 问题描述

## 4.2 系统设计

## 4.3 系统实现

## 4.4 实验小结

# 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] [Larry Nyhoff](http://www.calvin.edu/~nyhl/index.html). [ADTs, Data Structures, and Problem Solving with C++.](http://vig.prenhall.com/catalog/academic/product/0,1144,0131409093,00.html)Second Edition, [Calvin College](http://cs.calvin.edu/), 2005

[3] 殷立峰. Qt C++跨平台图形界面程序设计基础. 清华大学出版社,2014:192～197

[4] 严蔚敏等.数据结构题集(C语言版). 清华大学出版社

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A 基于顺序存储结构线性表实现的源程序

文件linearList.h内容

#ifndef \_LINEARLIST\_H

#define \_LINEARLIST\_H

#include *"status.h"*

*// 初始长度*

#define LIST\_INIT\_SIZE 100

*// 增长速率*

#define LISTINCREMENT 10

*// 定义模板参数*

**template**<**typename** T>

*// 类定义*

**class** **LinearList** {

*// 私有变量及函数部分*

**private**:

*// 数组的指针，初始化为nullptr*

T \* elem = **nullptr**;

*// 表的长度，初始化为0*

int length = 0;

*// 表的容量，初始化为0*

int listSize = 0;

*// 扩大链表的函数，只允许内部调用，设置访问权限为private*

void largerList() {

T \*temp = **nullptr**;

**try** {

*// 尝试开辟空间*

temp = **new** T[listSize + LISTINCREMENT];

}*// 失败抛出异常，异常处理*

**catch** (**const** std::bad\_alloc &e) {

*// 输出错误信息*

std::cerr << "ERROR: " << e.what() << std::endl;

*// 程序系统面临崩溃，及时退出*

exit(EXIT\_FAILURE);

}

*// 否则表示空间分配完成，将原来的数据复制到新的地方*

**for** (int i = 0; i < length; ++i) {

temp[i] = std::move(elem[i]);

}

*// 释放原来的空间*

**delete**[] elem;

*// 移交指针*

elem = temp;

*// 容量增长*

listSize += LISTINCREMENT;

}

**public**:

*// 公开函数部分*

*// 初始化函数*

status InitaList() {

*// 已经进行了初始化*

**if** (listSize != 0) {

std::cerr << "线性表已存在，初始化失败，请先进行销毁操作。**\n**" << std::endl;

**return** ERROR;

}

*// 初始化需要分配空间*

**try** {

*// new空间*

elem = **new** T[LIST\_INIT\_SIZE];

}

**catch** (**const** std::bad\_alloc& e) {

*// 分配失败*

std::cerr << "Error: " << e.what() << std::endl;

*// 返回错误*

**return** ERROR;

}

*// 表的容量即为LIST\_INIT\_SIZE*

listSize = LIST\_INIT\_SIZE;

*// 长度设为0*

length = 0;

char ch;

*// 询问是否从文件初始化*

std::cout << "if initalize from file(y/n): ";

std::cin >> ch;

getchar();

**if** (ch == 'y') {

*// 从文件初始化需要输入文件名*

std::cout << "Please enter file name: ";

std::string file\_name;

*// 得到文件名*

std::getline(std::cin, file\_name);

*// 打开输入流*

std::fstream fin(file\_name, std::ios\_base::in);

*// 判断是否打开完成*

**if** (!fin.is\_open()) {

std::cerr << "Open file failed.**\n**";

**return** ERROR;

}

**else** {

int cnt = 0;

*// 输入文件中的元素数量*

fin >> cnt;

*// 局部元素*

T ele;

*// 进行插入操作*

**for** (int i = 0; i < cnt; ++i) {

fin >> ele;

ListInsert(length + 1, ele);

}

std::cout << "Initalize from file successfully.**\n**";

}

*// 关闭文件流*

fin.close();

}

**return** OK;

}

*// 销毁线性表*

status DestroyList() {

*// 如果未进行初始化*

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 清理空间*

**delete**[] elem;

*// 置为空指针*

elem = **nullptr**;

*// 长度和容量置为0*

length = 0;

listSize = 0;

**return** OK;

}

*// 清空线性表*

status ClearList() {

*// 未初始化*

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 清空只需重置表长即可*

length = 0;

**return** OK;

}

*// 判断表是否为空*

bool ListEmpty() {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** true;

}

*// 判断表长是否为0即可*

**return** length == 0;

}

*// 获取表长*

int ListLength() {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** -1;

}

*// 返回表长*

**return** length;

}

*// 获取下表为index处的元素*

status GetElem(**const** int &index, T &e) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 判断下表合法性*

**if** (index > length || index < 1) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 直接获取*

e = elem[index - 1];

**return** OK;

}

*// 定位元素*

status LocateElem(**const** T &e, int &index, Compare<T> comp) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 下标初始化*

index = 0;

*// 遍历*

**for** (int i = 0; i < length; ++i) {

**if** (comp(e, elem[i])) {

index = i + 1;

**return** OK;

}

}

**return** ERROR;

}

*// 获取元素前驱*

status PriorElem(**const** T &cur, T &pre\_e, Compare<T> comp) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 遍历即可，注意范围*

**for** (int i = 0; i < length - 1; ++i) {

**if** (comp(cur, elem[i + 1])) {

pre\_e = elem[i];

**return** OK;

}

}

**return** ERROR;

}

*// 获取元素后驱*

status NextElem(**const** T &cur, T &next, Compare<T> comp) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 和前驱部分相同，遍历即可*

**for** (int i = 1; i < length; ++i) {

**if** (comp(cur, elem[i - 1])) {

next = elem[i];

**return** OK;

}

}

**return** ERROR;

}

*// 插入元素*

status ListInsert(**const** int &index, **const** T &e) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 判断插入位置合法性*

**if** (index < 1 || index > length + 1) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 如果表已经满了，则需要扩容*

**if** (length >= listSize) {

*// 进行扩容*

largerList();

}

*// 将对应元素依次向后移动*

**for** (int i = length; i >= index; --i) {

elem[i] = elem[i - 1];

}

*// 插入元素*

elem[index - 1] = e;

*// 长度要加1*

++length;

**return** OK;

}

*// 删除元素*

status ListDelete(**const** int &index, T &e) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 判断下标合法性*

**if** (index < 1 || index > length) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 直接返回即可*

e = elem[index - 1];

*// 将后面的函数向前移*

**for** (int i = index - 1; i < length - 1; ++i) {

elem[i] = elem[i + 1];

}

*// 长度减1*

--length;

**return** OK;

}

*// 遍历线性表*

status ListTraverse(Visit<T> visit) {

**if** (listSize == 0) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

*// 依次遍历即可*

**for** (int i = 0; i < length; ++i) {

visit(elem[i]);

}

**return** OK;

}

*// 保存到文件*

status ListSave() {

**if** (elem == **nullptr**) {

std::cerr << "List has not been initalized.**\n**";

**return** ERROR;

}

getchar();

*// 保存文件需要输入文件名*

std::cout << "Please enter file name: ";

std::string file\_name;

*// 输入文件名*

std::getline(std::cin, file\_name);

*// 打开文件流*

std::fstream fout(file\_name, std::ios\_base::out);

*// 判断打开是否成功*

**if** (fout.is\_open()) {

std::cout << "Open file successfully.**\n**";

*// 先将长度写入到文件中*

fout << length << "**\n**";

*// 将元素写入*

**for** (int i = 0; i < length; ++i) {

fout << elem[i] << "**\n**";

}

*// 关闭文件流*

fout.close();

std::cout << "Save successfully.**\n**";

**return** OK;

}

**else** {

std::cerr << "Open file failed.**\n**";

**return** ERROR;

}

}

*// 析构函数，释放空间即可*

~LinearList() {

**delete** elem;

}

};

#endif *//\_LINEARLIST\_H*

文件testLinearList.h内容：

#ifndef DATASTRUCTURE\_TESTLINEARLIST\_H

#define DATASTRUCTURE\_TESTLINEARLIST\_H

#include *"linearList.h"*

#include *"status.h"*

#define LEN 100

void testLinearList() {

int temp = 0;

long op = 1;

int cur = 0, next = 0, pre = 0, elem = 0;

int list\_index = 0;

int temp\_index = 0;

int index;

LinearList<int> lists[LEN];

Compare<int> comp = [](**const** int &a, **const** int &b) -> bool { **return** a == b; };

Visit<int> visit = [](**const** int &a) { std::cout << a << std::endl; };

**while** (op != 0) {

cls();

std::cout << "**\n\n**";

std::cout << " Menu for Linked Table On Chain Structure **\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " 1. IntiaList 7. LocateElem**\n**";

std::cout << " 2. DestroyList 8. PriorElem**\n**";

std::cout << " 3. ClearList 9. NextElem **\n**";

std::cout << " 4. ListEmpty 10.ListInsert**\n**";

std::cout << " 5. ListLength 11.ListDelete**\n**";

std::cout << " 6. GetElem 12.ListTraverse**\n**";

std::cout << " 13.Change list 14.Save List**\n**";

std::cout << " 0. Exit**\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " list using: " << list\_index << "**\n**";

std::cout << " please select your operation[0~14]:";

std::cin >> op;

**auto** &L = lists[list\_index];

**switch** (op) {

**case** 1:

std::cout << "InitaList Function**\n**";

**if** (L.InitaList() == OK) std::cout << "Initialze successful.**\n**";

**else** std::cout << "Initailze failed.**\n**";

getchar();

getchar();

**break**;

**case** 2:

std::cout << "**\n**DestroyList Function**\n**";

**if** (L.DestroyList() == OK) std::cout << "Destroy successfully.**\n**";

**else** std::cout << "Destroy failed.**\n**";

getchar();

getchar();

**break**;

**case** 3:

std::cout << "**\n**ClearList Function**\n**";

**if** (L.ClearList() == OK) std::cout << "Clear successfully**\n**";

**else** std::cout << "CLear failed.**\n**";

getchar();

getchar();

**break**;

**case** 4:

std::cout << "**\n**ListEmpty Function**\n**";

**if** (L.ListEmpty()) {

std::cout << "The list is empty.**\n**";

}

**else** {

std::cout << "The list is not empty.**\n**";

}

getchar();

getchar();

**break**;

**case** 5:

std::cout << "**\n**ListLength Function**\n**";

std::cout << "The length of list: " << L.ListLength() << std::endl;

getchar();

getchar();

**break**;

**case** 6:

std::cout << "**\n**GetElem Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.GetElem(index, elem) == OK) {

std::cout << "Get the element: " << elem << "**\n**";

}

**else** std::cout << "Get element failed.**\n**";

getchar();

getchar();

**break**;

**case** 7:

std::cout << "**\n**LocateElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

**if** (L.LocateElem(elem, index, comp) == OK) {

std::cout << "The index of element:" << index << std::endl;

}

**else** {

std::cout << "Locate element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 8:

std::cout << "**\n**PriorElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> cur;

**if** (L.PriorElem(cur, pre, comp) == OK) {

std::cout << "Get the result: " << pre << "**\n**";

}

**else** {

std::cout << "Get the element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 9:

std::cout << "**\n**NextElem Function**\n**";

std::cout << "PLease enter the element: ";

std::cin >> cur;

**if** (L.NextElem(cur, next, comp) == OK) {

std::cout << "Get the element: " << next << "**\n**";

}

**else** {

std::cout << "Operate failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 10:

std::cout << "**\n**ListInsert Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

std::cout << "Please enter the index: ";

scanf\_s("%d", &index);

**if** (L.ListInsert(index, elem) == OK) {

std::cout << "Insert successfully.**\n**";

}

**else** {

std::cout << "Insert failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 11:

std::cout << "**\n**ListDelete Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.ListDelete(index, elem) == OK) {

std::cout << "Delete " << elem << " successfullyy.**\n**";

}

**else** {

std::cout << "Delete failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 12:

std::cout << "**\n**ListTraverse Function**\n**";

**if** (L.ListTraverse(visit)) {

std::cout << "Traverse successfully.**\n**";

}

**else** {

std::cout << "Traverse failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 13:

std::cout << "Enter the index of list:[0-99] ";

std::cin >> temp\_index;

**if** (temp\_index >= LEN || temp\_index < 0) {

std::cout << "Change failed**\n**";

}

**else** {

list\_index = temp\_index;

std::cout << "Change successfully**\n**";

}

getchar();

getchar();

**break**;

**case** 14:

**if** (L.ListSave() == OK) {

std::cout << "Save to file successfully**\n**";

}

**else** {

std::cout << "Save to file failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 0:

**break**;

**default**:

**break**;

}*//end of switch*

}*//end of while*

std::cout << "Good bye.**\n**";

}

#endif *//DATASTRUCTURE\_TESTLINEARLIST\_H*

# 附录B 基于链式存储结构线性表实现的源程序

文件linkedList.h内容：

#ifndef DATASTRUCTURE\_LINKEDLIST\_H

#define DATASTRUCTURE\_LINKEDLIST\_H

*// 包含所需头文件*

#include *"status.h"*

*// 使用模板*

**template**<**typename** T = int>

*// 定义类*

**class** **LinkedList** {

**private**:

*// 结点类型*

**struct** Node {

*// 值*

T value;

*// 指向下一个文件的指针*

Node \*next;

*// 删除默认构造函数*

Node() = **delete**;

*// 定义构造函数*

Node(**const** T &value) : value(value), next(**nullptr**) {}

};

*// 定义头指针*

Node \*head = **nullptr**;

*// 链表长度*

int length = 0;

*// 是否进行了初始化*

bool initaled = false;

**public**:

*// 初始化函数*

status InitaList() {

*// 已经初始化过的返回ERROR*

**if** (initaled) {

*// 输出错误信息*

std::cerr << "The list had been initalized.**\n**";

**return** ERROR;

}

*// 长度初始化为0*

length = 0;

*// 询问是否从文件初始化*

std::cout << "initalize from file(y/n):";

char ch;

*// 输入选择*

std::cin >> ch;

**if** (ch == 'y') {

getchar();

*// 得到文件名*

std::cout << "Please enter name of the file: ";

std::string file\_name;

std::getline(std::cin, file\_name);

*// 打开文件流*

std::fstream fin(file\_name, std::ios\_base::in);

*// 查看打开是否成功*

**if** (fin.is\_open()) {

*// 进行输入数据*

initaled = true;

int cnt = 0;

*// 得到数据个数*

fin >> cnt;

*// 定义局部变量*

T temp;

**for** (int i = 0; i < cnt; ++i) {

*// 输入变量*

fin >> temp;

*// 插入到链表后面*

ListInsert(length + 1, temp);

}

*// 关闭文件流*

fin.close();

std::cout << "Initalize from file successfully.**\n**";

}

**else** {

*// 否则输出错误*

std::cerr << "Open file failed.**\n**";

**return** ERROR;

}

}

initaled = true;

**return** OK;

}

*// 销毁链表*

status DestroyList() {

*// 如果未初始化返回错误*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

Node \*temp = head;

Node \*pre;

*// 清理所有链表空间*

**while** (temp != **nullptr**) {

pre = temp->next;

**delete** temp;

temp = pre;

}

*// 头指针置为空*

head = **nullptr**;

*// 长度置零*

length = 0;

*// 初始化置为false*

initaled = false;

**return** OK;

}

*// 清空链表*

status ClearList() {

*// 未初始化不能清空*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 唯一区别就是摧毁后的初始化状态为true*

DestroyList();

initaled = true;

**return** OK;

}

*// 检测是否为空表*

bool ListEmpty() **const** {

*// 未初始化的默认是空表*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** true;

}

*// 判断长度是否为0*

**return** length == 0;

}

*// 得到链表长度*

int ListLength() **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** -1;

}

*// 返回长度*

**return** length;

}

*// 获取特定位置的元素*

status GetElem(**const** int &index, T &e) **const** {

*// 未初始化报错*

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 判断下标是否合法*

**if** (index < 0 || index > length) {

std::cerr << "Index out of bounds.**\n**";

**return** ERROR;

}

*// 定义局部变量*

**auto** temp = head;

*// index处元素向后移动index-1次即可*

**for** (int i = 0; i < index - 1; ++i) {

temp = temp->next;

}

*// 获取值*

e = temp->value;

**return** OK;

}

*// 定位元素*

status LocateElem(**const** T &e, int &index, Compare<T> comp) **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 局部头指针*

**auto** temp\_ptr = head;

*// 遍历元素，看是否有comp为真的*

**for** (int i = 0; i < length; i++) {

**if** (comp(e, temp\_ptr->value)) {

*// 存在就返回下标*

index = i + 1;

**return** OK;

}

*// 指针后移*

temp\_ptr = temp\_ptr->next;

}

*// 否则就是没有*

index = 0;

**return** ERROR;

}

*// 获取前驱元素*

status PriorElem(**const** T &cur, T &pre\_e, Compare<T> comp) **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 局部头指针*

**auto** temp\_ptr = head;

*// 遍历元素*

**while** (temp\_ptr != **nullptr** && temp\_ptr->next != **nullptr**) {

*// 如果此处元素的后驱是给出的元素，返回此元素*

**if** (comp(temp\_ptr->next->value, cur)) {

pre\_e = temp\_ptr->value;

**return** OK;

}

temp\_ptr = temp\_ptr->next;

}

**return** ERROR;

}

*// 获得后驱*

status NextElem(**const** T &cur, T &next, Compare<T> comp) **const** {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

**auto** temp\_ptr = head;

*// 如果此元素满足comp为真，下一个即为所求元素*

**while** (temp\_ptr != **nullptr** && temp\_ptr->next != **nullptr**) {

**if** (comp(temp\_ptr->value, cur)) {

next = temp\_ptr->next->value;

**return** OK;

}

temp\_ptr = temp\_ptr->next;

}

**return** ERROR;

}

*// 插入元素*

status ListInsert(**const** int &index, **const** T &e) {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 判断下标是否合法*

**if** (index < 1 || index > length + 1) {

std::cerr << "Index out of bounds**\n**";

**return** ERROR;

}

*// 使用初始值新建结点*

**auto** t = **new** Node(e);

*// 插入到首元素位置特殊处理*

**if** (index == 1) {

t->next = head;

head = t;

}

**else** {

*// 向后移动到适当位置*

**auto** pre = head;

**for** (int i = 0; i < index - 2; ++i) {

pre = pre->next;

}

*// 将t插入*

t->next = pre->next;

pre->next = t;

}

*// 长度加1*

++length;

**return** OK;

}

*// 删除元素*

status ListDelete(**const** int &index, T &e) {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 判断下标合法性*

**if** (index < 1 || index > length) {

std::cerr << "Index out of bounds**\n**";

**return** ERROR;

}

*// 删除首元素特殊处理*

**if** (index == 1) {

Node \*temp = head;

head = head->next;

e = temp->value;

**delete** temp;

}

**else** {

**auto** pre = head;

*// 后移适当位置，找到要删除元素的前驱*

**for** (int i = 0; i < index - 2; ++i) {

pre = pre->next;

}

*// 删除元素*

**auto** to\_del = pre->next;

pre->next = to\_del->next;

e = to\_del->value;

**delete** to\_del;

}

*// 长度减 1*

--length;

**return** OK;

}

*// 遍历元素*

status ListTraverse(Visit<T> visit) {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

*// 直接遍历即可*

**auto** temp\_ptr = head;

**for** (int i = 0; i < length; ++i) {

*// 使用visit指针*

visit(temp\_ptr->value);

*// 指针后移*

temp\_ptr = temp\_ptr->next;

}

**return** OK;

}

*// 将链表保存到文件中*

status ListSave() {

**if** (!initaled) {

std::cerr << "List is not initialized.**\n**";

**return** ERROR;

}

getchar();

*// 得到文件名*

std::cout << "Please enter the file name: ";

std::string file\_name;

std::cin >> file\_name;

*// 打开输出文件流*

std::fstream fout(file\_name, std::ios\_base::out);

*// 先写入元素个数*

fout << length << "**\n**";

*// 遍历写入*

**auto** temp\_ptr = head;

**for** (int i = 0; i < length; ++i) {

fout << temp\_ptr->value << "**\n**";

temp\_ptr = temp\_ptr->next;

}

*// 关闭文件流保存文件*

fout.close();

**return** OK;

}

*// 析构函数*

~LinkedList() {

*// 主要就是释放空间了*

**auto** temp = head;

**while** (temp != **nullptr**) {

**auto** next = temp->next;

**delete** temp;

temp = next;

}

head = **nullptr**;

}

};

#endif *//DATASTRUCTURE\_LINKEDLIST\_H*

文件testLinkedList.h内容：

#ifndef DATASTRUCTURE\_TESTLINKEDLIST\_H

#define DATASTRUCTURE\_TESTLINKEDLIST\_H

#define LEN 100

#include *"linkedList.h"*

void testLinkedList() {

int list\_index = 0;

long op = 1;

int cur = 0, next = 0, pre = 0, elem = 0;

int l\_index = 0;

int index;

LinkedList<int> lists[LEN];

Compare<int> comp = [](**const** int &a, **const** int &b) -> bool { **return** a == b; };

Visit<int> visit = [](**const** int &a) { std::cout << a << std::endl; };

**while** (op != 0) {

cls();

std::cout << "**\n\n**";

std::cout << " Menu for Linked Table On Chain Structure **\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " 1. IntiaList 7. LocateElem**\n**";

std::cout << " 2. DestroyList 8. PriorElem**\n**";

std::cout << " 3. ClearList 9. NextElem **\n**";

std::cout << " 4. ListEmpty 10.ListInsert**\n**";

std::cout << " 5. ListLength 11.ListDelete**\n**";

std::cout << " 6. GetElem 12.ListTraverse**\n**";

std::cout << " 13.Change list 14.Save List**\n**";

std::cout << " 0. Exit**\n**";

std::cout << "-------------------------------------------------**\n**";

std::cout << " list using: " << l\_index << "**\n**";

std::cout << " please select your operation[0~14]:";

std::cin >> op;

**auto** &L = lists[l\_index];

**switch** (op) {

**case** 1:

std::cout << "InitaList Function**\n**";

**if** (L.InitaList() == OK) std::cout << "Initialze successful.**\n**";

**else** std::cout << "Initailze failed.**\n**";

getchar();

getchar();

**break**;

**case** 2:

std::cout << "**\n**DestroyList Function**\n**";

**if** (L.DestroyList() == OK) std::cout << "Destroy successfully.**\n**";

**else** std::cout << "Destroy failed.**\n**";

getchar();

getchar();

**break**;

**case** 3:

std::cout << "**\n**ClearList Function**\n**";

**if** (L.ClearList() == OK) std::cout << "Clear successfully**\n**";

**else** std::cout << "CLear failed.**\n**";

getchar();

getchar();

**break**;

**case** 4:

std::cout << "**\n**ListEmpty Function**\n**";

**if** (L.ListEmpty()) {

std::cout << "The list is empty.**\n**";

}

**else** {

std::cout << "The list is not empty.**\n**";

}

getchar();

getchar();

**break**;

**case** 5:

std::cout << "**\n**ListLength Function**\n**";

std::cout << "The length of list: " << L.ListLength() << std::endl;

getchar();

getchar();

**break**;

**case** 6:

std::cout << "**\n**GetElem Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.GetElem(index, elem) == OK) {

std::cout << "Get the element: " << elem << "**\n**";

}

**else** std::cout << "Get element failed.**\n**";

getchar();

getchar();

**break**;

**case** 7:

std::cout << "**\n**LocateElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

**if** (L.LocateElem(elem, index, comp) == OK) {

std::cout << "The index of element:" << index << std::endl;

}

**else** {

std::cout << "Locate element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 8:

std::cout << "**\n**PriorElem Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> cur;

**if** (L.PriorElem(cur, pre, comp) == OK) {

std::cout << "Get the result: " << pre << "**\n**";

}

**else** {

std::cout << "Get the element failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 9:

std::cout << "**\n**NextElem Function**\n**";

std::cout << "PLease enter the element: ";

std::cin >> cur;

**if** (L.NextElem(cur, next, comp) == OK) {

std::cout << "Get the element: " << next << "**\n**";

}

**else** {

std::cout << "Operate failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 10:

std::cout << "**\n**ListInsert Function**\n**";

std::cout << "Please enter the element: ";

std::cin >> elem;

std::cout << "Please enter the index: ";

scanf\_s("%d", &index);

**if** (L.ListInsert(index, elem) == OK) {

std::cout << "Insert successfully.**\n**";

}

**else** {

std::cout << "Insert failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 11:

std::cout << "**\n**ListDelete Function**\n**";

std::cout << "Please enter the index: ";

std::cin >> index;

**if** (L.ListDelete(index, elem) == OK) {

std::cout << "Delete " << elem << " successfullyy.**\n**";

}

**else** {

std::cout << "Delete failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 12:

std::cout << "**\n**ListTraverse Function**\n**";

**if** (L.ListTraverse(visit)) {

std::cout << "Traverse successfully.**\n**";

}

**else** {

std::cout << "Traverse failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 13:

std::cout << "Enter the index of list:[0-99] ";

std::cin >> list\_index;

**if** (list\_index >= LEN || list\_index < 0) {

std::cout << "Change failed**\n**";

}

**else** {

l\_index = list\_index;

std::cout << "Change successfully**\n**";

}

getchar();

getchar();

**break**;

**case** 14:

**if** (L.ListSave() == OK) {

std::cout << "Save to file successfully**\n**";

}

**else** {

std::cout << "Save to file failed.**\n**";

}

getchar();

getchar();

**break**;

**case** 0:

**break**;

**default**:

**break**;

}*//end of switch*

}*//end of while*

std::cout << "Good bye.**\n**";

}

#endif *//DATASTRUCTURE\_TESTLINKEDLIST\_H*

# 附录C 基于二叉链表二叉树实现的源程序

# 附录D 基于邻接表图实现的源程序