

**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： 袁凌**

**报告日期： 2017年 1月 12 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](#_Toc1844647352)

[1.1 问题描述 2](#_Toc596092475)

[1.2 系统设计 2](#_Toc1455301192)

[1.3 系统实现 3](#_Toc1112433839)

[1.4 实验小结 25](#_Toc1723688055)

[2 基于链式存储结构的线性表实现 26](#_Toc1000389263)

[2.1 问题描述 26](#_Toc1589791799)

[2.2 系统设计 26](#_Toc1977263418)

[2.3 系统实现 26](#_Toc609547117)

[2.4 实验小结 26](#_Toc1698740903)

[3 基于二叉链表的二叉树实现 28](#_Toc850952297)

[3.1 问题描述 28](#_Toc1091236247)

[3.2 系统设计 28](#_Toc614267901)

[3.3 系统实现 28](#_Toc1970948769)

[3.4 实验小结 28](#_Toc941968523)

[4 基于邻接表的图实现 29](#_Toc1506424350)

[4.1 问题描述 29](#_Toc2054116356)

[4.2 系统设计 29](#_Toc507010153)

[4.3 系统实现 29](#_Toc1247677615)

[4.4 实验小结 29](#_Toc1082479347)

[参考文献 30](#_Toc1964940435)

[附录A 基于顺序存储结构线性表实现的源程序 33](#_Toc567702926)

[附录B 基于链式存储结构线性表实现的源程序 34](#_Toc1407123527)

[附录C 基于二叉链表二叉树实现的源程序 35](#_Toc825010388)

[附录D 基于邻接表图实现的源程序 37](#_Toc1894454883)

# 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

函数的实现，首先是对表的定义，我是用了类进行定义的：

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

**class** **linearList** {

**public**:

*// 初始化函数*

status InitaList();

*//销毁的函数*

status DestroyList();

*// 清空函数*

status ClearList();

*// 判断是否为空函数*

bool ListEmpty();

*// 求表的长度的函数*

int ListLength();

*// 获取对应位置的元素*

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

*// 定位元素位置*

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

*// 给定元素前一个元素*

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

*// 查找元素下一个元素*

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

*// 插入元素*

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

*// 删除列表元素*

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

*// 遍历列表*

status ListTraverse(Visit<T> visit);

*// 将表存储到文件中*

status ListSave();

~linearList();

**private**:

*// 存储元素的指针数组*

T \*elem = **nullptr**;

*// 表的长度*

int length = 0;

*// 表的容量*

int listSize = 0;

*// 给表扩容*

void largerList();

};

主要就是将成员函数定义为所要求的函数。具体的类外实现代码如下：

*// 初始化函数*

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

status linearList<T>::InitaList() {

*// 先判断是否被初始过*

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

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

**return** ERROR;

}

*// 申请空间*

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

*// 申请失败的情况*

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

std::cerr << "Out of memory**\n**";

**return** ERROR;

}

*// 设置表的现在长度和容量*

listSize = LIST\_INIT\_SIZE;

length = 0;

*// 询问是否从文件进行初始化*

char ch = '\0';

std::cout << "是否从文件初始化(y/n)：";

std::cin >> ch;

*// 处理换行符*

getchar();

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

*// 得到使用文件名称*

std::cout << "请输入要使用的初始化的文件名：";

std::string str;

*// 得到文件名*

std::getline(std::cin, str);

std::fstream fin;

*// 打开文件流*

fin.open(str, std::ios\_base::in);

*// 打开错误说明文件不存在或者损坏*

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

std::cerr << "文件打开错误。**\n**";

**return** ERROR;

} **else** {

*// 先得到文件中元素的个数*

int cnt = 0;

fin >> cnt;

T ele;

*// 获取全部的元素并且插入到表尾*

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

fin >> ele;

ListInsert(length + 1, ele);

}

*// 输出提示信息*

std::cout << "初始化成功。**\n**";

}

*// 注意记得关闭文件*

fin.close();

}

*// 返回状态*

**return** OK;

}

*// 销毁函数*

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

status linearList<T>::DestroyList() {

*// 如果listSize为0表示未进行初始化或者已经被销毁过。*

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

std::cerr << "表未初始化，无法进行销毁。**\n**";

**return** ERROR;

}

*// 只需要对空间进行清理，并把长度等信息置0即可。*

**delete**[] elem;

length = 0;

listSize = 0;

*// 返回销毁成功*

**return** OK;

}

*// 清空一个表*

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

status linearList<T>::ClearList() {

*// 首先是判断表是否存在*

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

std::cerr << "表未被初始化，清空失败**\n**";

**return** ERROR;

}

*// 清空就是把长度设置为0*

length = 0;

**return** OK;

}

*// 判断是否为空表*

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

bool linearList<T>::ListEmpty() {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** false;

}

*// 表为空即长度为0*

**return** length == 0;

}

*// 求表长的函数*

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

int linearList<T>::ListLength() {

*// 未初始化的表其实个人觉得应该认为表长为0*

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** -1;

}

**return** length;

}

*// 获取对应位置的元素*

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

status linearList<T>::GetElem(**const** int &index, T &e) {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** ERROR;

}

*// 判断下表是否超出范围*

**if** (index > length) {

std::cerr << "下标超过表的长度。**\n**";

**return** ERROR;

} **else** **if** (index < 1) {

std::cerr << "下标小于1。**\n**";

**return** ERROR;

}

*// 返回元素*

e = elem[index - 1];

**return** OK;

}

*// 定位元素的函数，参数为需要定位的元素和比较函数，需要存放的位置*

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

status linearList<T>::LocateElem(**const** T &e,

int &index, Compare<T> comp) {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** ERROR;

}

*// 当不满足时设置为0，直接在这里设置即可*

index = 0;

*// 由于元素为无序的，所以直接线性搜索即可，注意搜索的区间即可*

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

*// 如果此位置的元素满足要求，返回位置即可*

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

index = i + 1;

**return** OK;

}

}

*// 返回执行状态*

**return** ERROR;

}

*// 查找前驱元素，参数为需要查找的元素和对应的比较函数*

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

status linearList<T>::PriorElem(**const** T &cur,

T &pre\_e, Compare<T> comp) {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** ERROR;

}

*// 线性查找，结束条件为找到或者未找到*

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

*// 如果i + 1处元素为cur，那么i处即为所找，不考虑元素重复的情况*

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

*// 为参数赋值*

pre\_e = elem[i];

**return** OK;

}

}

*// 返回查找状态*

**return** ERROR;

}

*// 查找后驱元素*

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

status linearList<T>::NextElem(**const** T &cur,

T &next, Compare<T> comp) {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** ERROR;

}

*// 对元素进行一一比较*

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

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

next = elem[i];

**return** OK;

}

}

*// 返回错误状态*

**return** ERROR;

}

*// 插入函数*

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

status linearList<T>::ListInsert(**const** int &index, **const** T &e) {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** ERROR;

}

*// 再判断插入位置是否合理*

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

std::cerr << "下标不合法。**\n**";

**return** ERROR;

}

*// 如果表满了则需要进行扩容*

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

largerList();

}

*// 将对应位置的元素向后移动为插入的元素腾出空间*

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

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

}

*// 将元素放置好*

elem[index - 1] = e;

*// 最后长度加一*

++length;

**return** OK;

}

*// 删除指定位置元素*

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

status linearList<T>::ListDelete(**const** int &index, T &e) {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** ERROR;

}

*// 判断参数是否合法*

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

std::cerr << "下标不合法。**\n**";

**return** ERROR;

}

*// 若下表合法将元素取出*

e = elem[index - 1];

*// 将对应位置元素前移*

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

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

}

*// 长度减一*

--length;

*// 返回运行结果*

**return** OK;

}

*// 对表进行遍历，参数只需要使用的函数*

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

status linearList<T>::ListTraverse(Visit<T> visit) {

*// 首先判断是否是个未被初始化的表*

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

std::cerr << "表未被初始化。**\n**";

**return** ERROR;

}

*// 进行遍历*

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

visit(elem[i]);

}

*// 返回遍历状态*

**return** OK;

}

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

linearList<T>::~linearList() {

*// c++中delete空指针不会产生异常*

**delete** elem;

}

*// 为列表扩容*

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

void linearList<T>::largerList() {

*// 先尝试开辟更大的空间*

T \*temp = **nullptr**;

**try** {

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

}

*// 开辟失败了会抛出异常，正确做法是结束程序*

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

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

exit(EXIT\_FAILURE);

}

*// 如果没有失败就把原来的元素复制过来*

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

temp[i] = elem[i];

}

*// 记得清理空间*

**delete**[] elem;

*// 为elem重新赋值*

elem = temp;

*// 调整容量*

listSize += LISTINCREMENT;

}

*// 保存表的函数*

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

status linearList<T>::ListSave() {

*// 如果长度为0就或者未初始化不进行保存*

**if** (length == 0 || elem == **nullptr**) {

std::cerr << "表为空，无法保存到文件。**\n**";

**return** ERROR;

}

*// 处理换行符*

getchar();

*// 得到文件名*

std::cout << "请输入文件名：";

std::string str;

std::getline(std::cin, str);

*// 打开文件输出流*

std::fstream fout;

fout.open(str, std::ios\_base::out);

*// 检测打开状态*

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

std::cout << "打开文件成功。**\n**";

*// 先将长度信息保存到文件中*

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

*// 依次将表元素保存到文件中，具体保存的方法*

*// 由对模板特例化中对<<运算符的重载决定*

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

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

}

*// 关闭文件流*

fout.close();

**return** OK;

} **else** {

*// 打开写入文件失败，报错并返回状态*

std::cerr << "文件打开失败。**\n**";

**return** ERROR;

}

}

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：

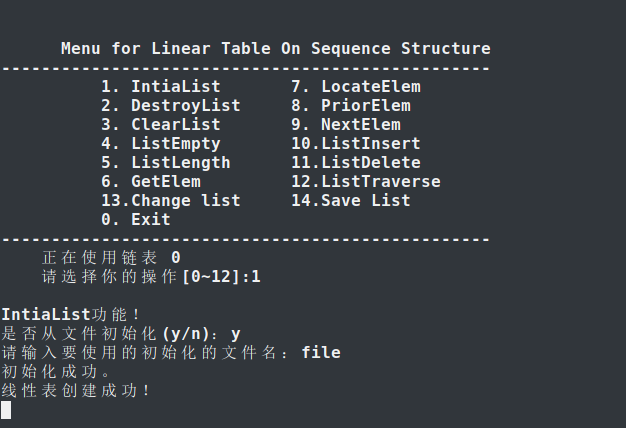


图1-1

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：

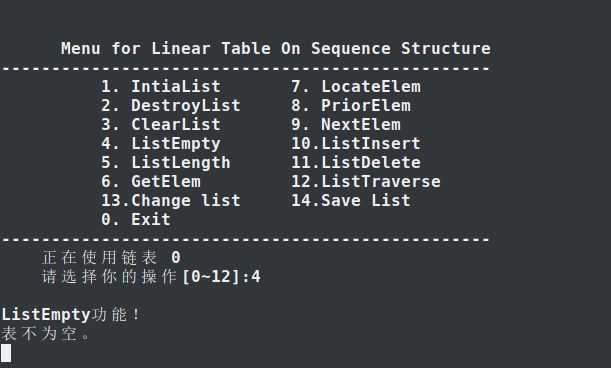


图1-3

得到表的长度，见图1-4：

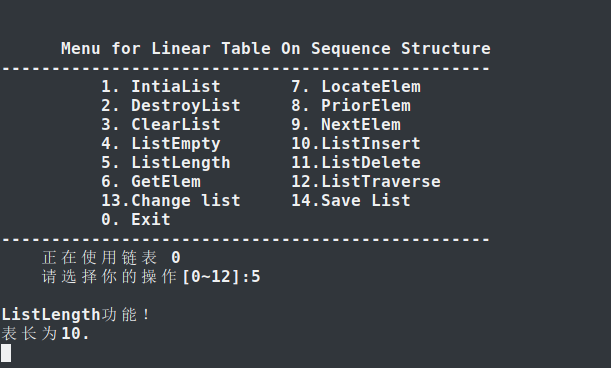


图1-4

使用GetElem函数获取元素，见图1-5,1-6,1-7：

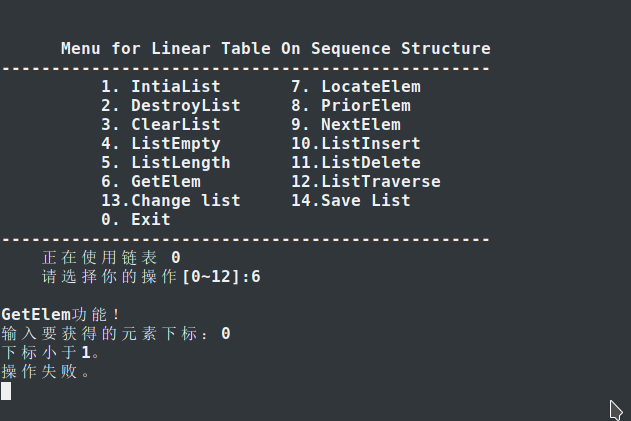


图1-5

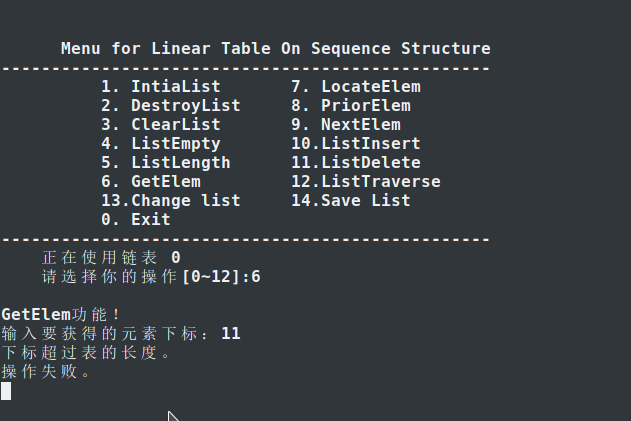


图1-6

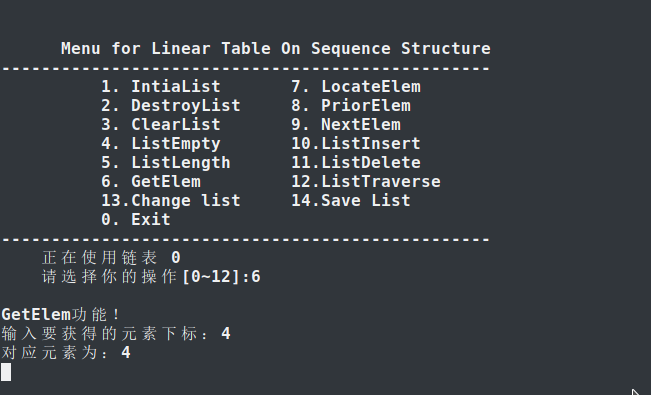


图1-7

定位元素的操作见图1-8,1-9,1-10：

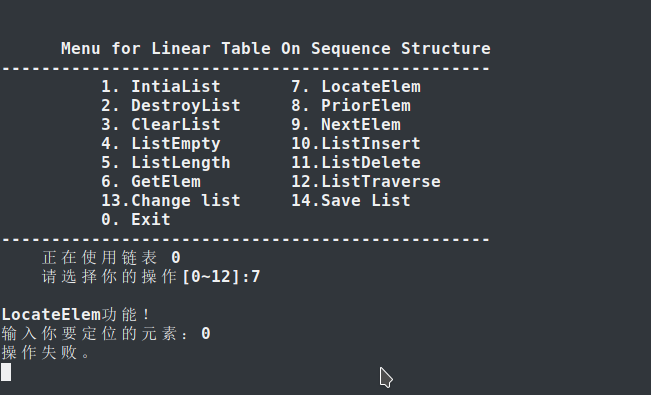


图1-8

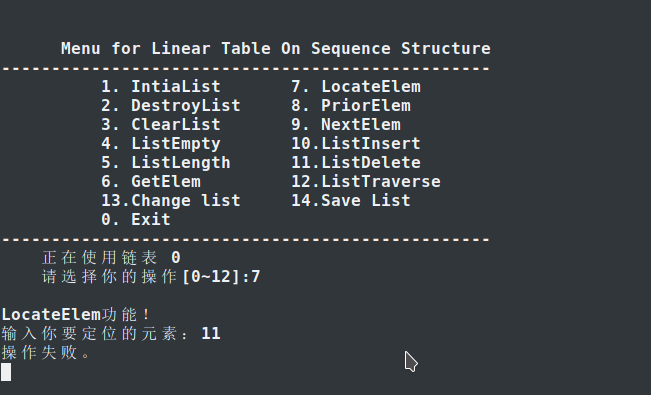


图1-9

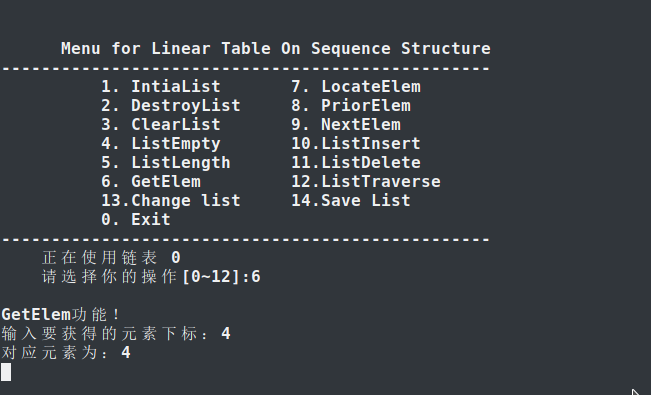


图1-10

获得前驱的操作见图1-11,1-12：

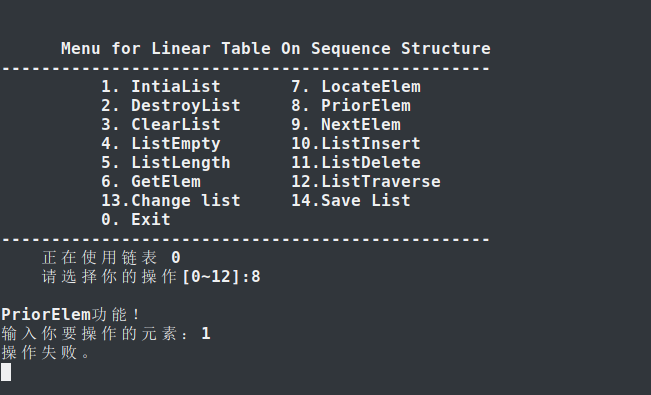


图1-11

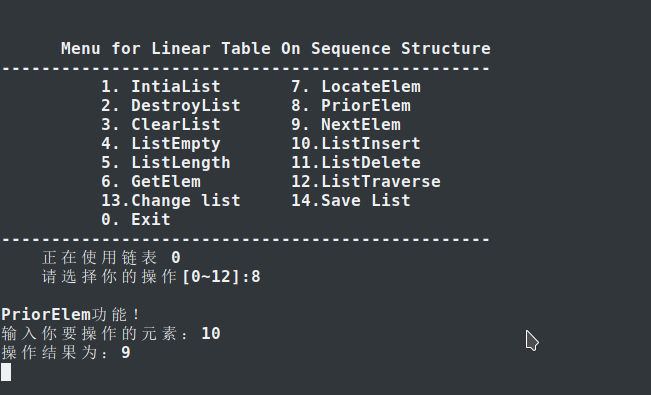


图1-12

获得后驱的函数测试1-13,1-14：

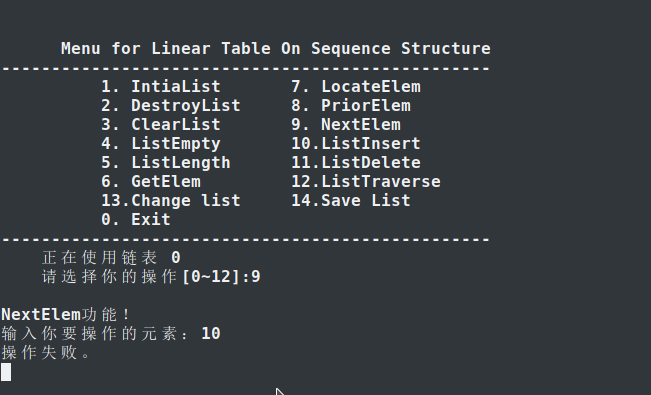


图1-13

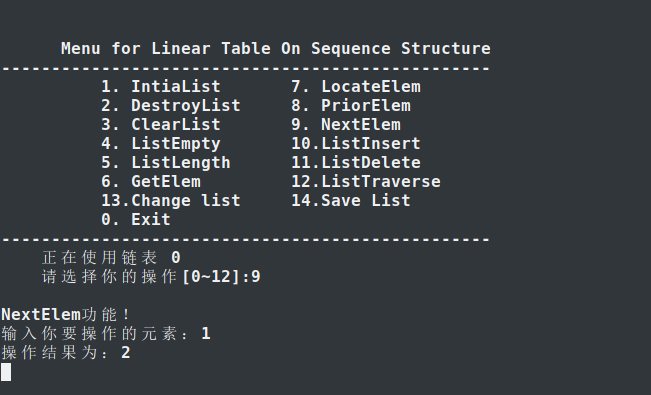


图1-14

元素插入测试见图1-15,1-16，1-17：

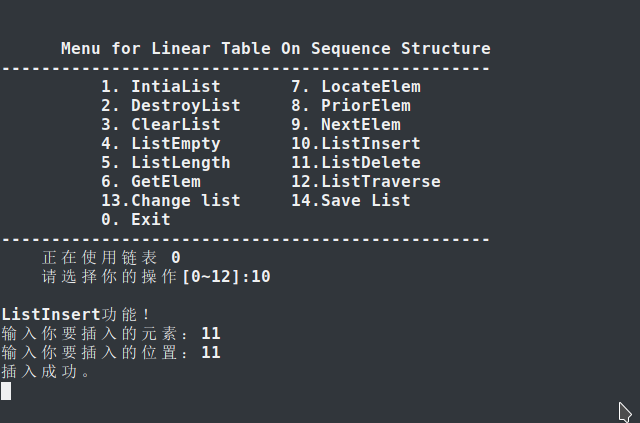


图1-15



图1-16

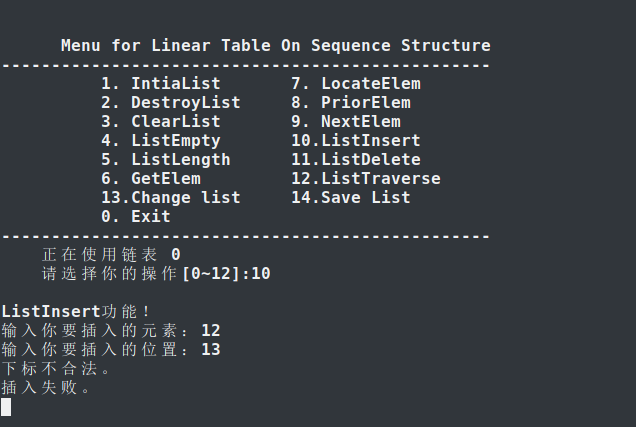


图1-17

元素删除测试如下图1-18,1-19,1-20：

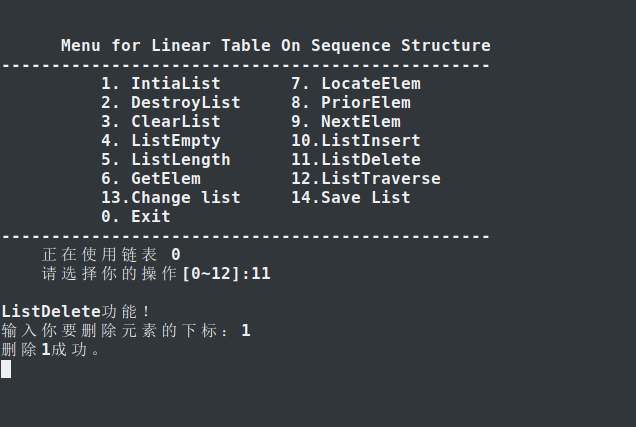


图1-18



图1-19

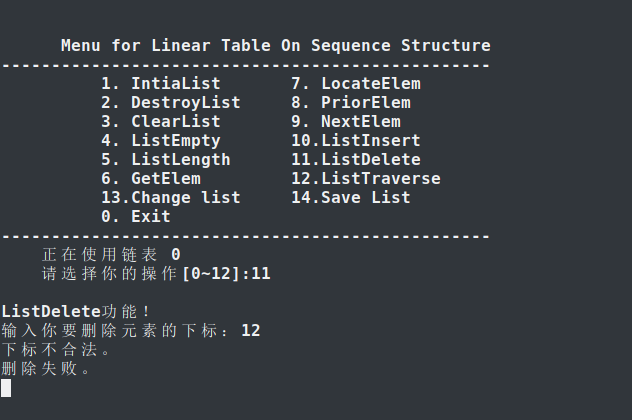


图1-20

保存链表测试见图1-21,1-22：

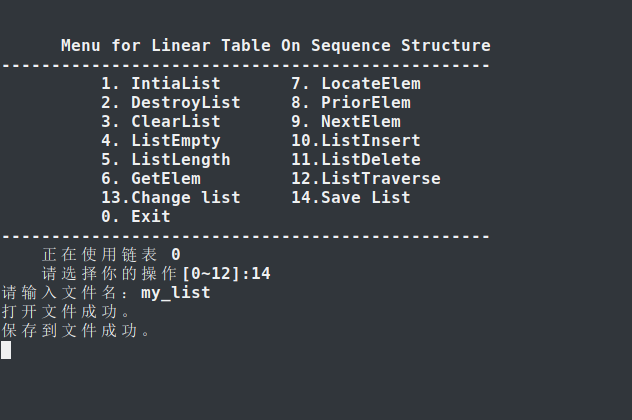


图1-21

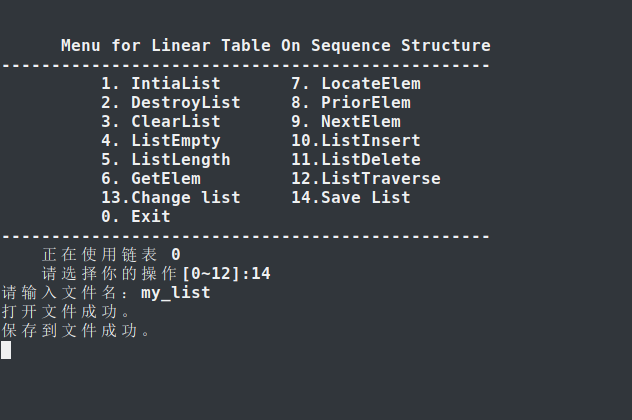


图1-22

切换链表操作见图1-23,1-24,1-25：

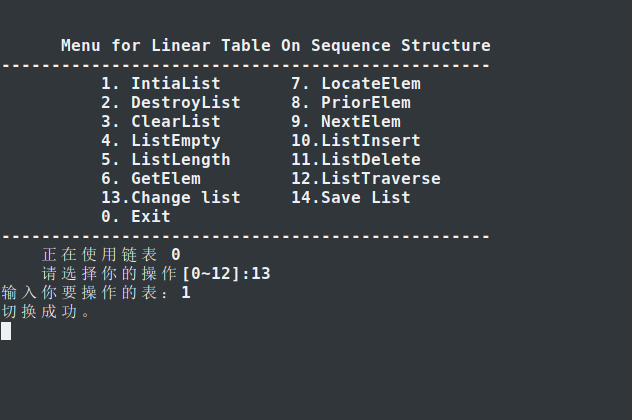


图1-23

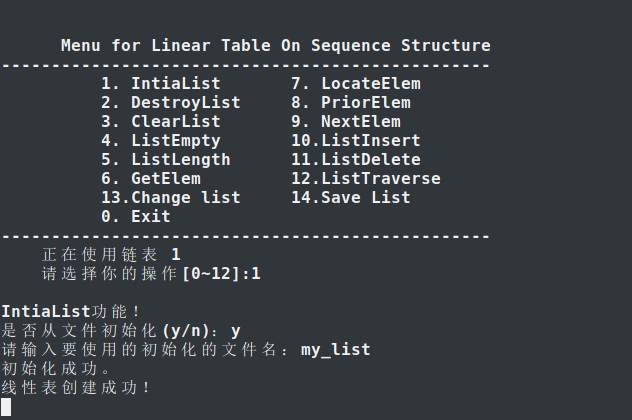


图1-24



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

## 2.2 系统设计

## 2.3 系统实现

## 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

/\* Linear Table On Sequence Structure \*/

#include <stdio.h>

#include <malloc.h>

#include <stdlib.h>

/\*---------page 10 on textbook ---------\*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

# 附录B 基于链式存储结构线性表实现的源程序

# 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