

# 1 基于顺序存储结构的线性表实现

## 问题描述

为实现基于顺序存储结构的线性表及以此为基础的多表管理和文件读写功能，按如下形式设计数据的逻辑结构及基本运算。

### 线性表抽象数据类型

依据最小完备性和常用性相结合的原则，设计了线性表的数据对象和数据关系，并定义了线性表的初始化表、销毁表、清空表、判定空表、求表长和获得元素等12种基本运算，具体数据和运算功能定义如下。

**ADT** SqList {

**数据对象：**

**数据关系：**

**基本操作：**

SqList\_init(&L)

初始条件：线性表L不存在。

操作结果：构造一个空的线性表。

SqList\_destroy(&L)

初始条件：线性表L已存在。

操作结果：销毁线性表L。

SqList\_clear(&L)

初始条件：线性表L已存在。

操作结果：将L重制为 空表。

SqList\_empty(L)

初始条件：线性表L已存在。

操作结果：若L为空表则返回TRUE，否则返回FALSE。

SqList\_length(L)

初始条件：线性表L已存在。

操作结果：返回L中数据元素的个数。

SqList\_getElem(L, idx, &ele)

初始条件：线性表L已存在且非空， 。

操作结果：用ele返回L中第idx个数据元素的值。

SqList\_locateElem(L, val)

初始条件：线性表L已存在且非空。

操作结果：返回L中第一个值为val的数据元素的位序。若这样的元素不存在则返回0。

SqList\_priorElem(L, cur, &pre)

初始条件：线性表L已存在、非空且cur不是第一个数据元素。

操作结果：若cur是L的数据元素，则将它的前驱存入pre，否则返回FALSE。

SqList\_nextElem(L, cur, &next)

初始条件：线性表L已存在、非空且cur不是最后一个数据元素。

操作结果：若cur是L的数据元素，则将它的后继存入next，否则返回FALSE。

SqList\_insert(&L, key, val)

初始条件：线性表L已存在且非空， 。

操作结果：在L的第key个位置之前插入新的数据元素val。

SqList\_delete(&L, key, &val)

初始条件：线性表L已存在且非空， 。

操作结果：删除L的第key个数据元素，其值由val带回。

SqList\_traverse(L)

初始条件：线性表L已存在。

操作结果：依次访问并输出L的每个数据元素。

} **ADT** SqList

### 1.1.2 多线性表操作

另外了设计了多线性表操作，可以在多个互相独立的线性表中切换工作表。

### 1.1.3 演示系统与数据文件

直接以实验文档中提供的框架作为演示系统。该框架将完成函数调用所需实参值的准备和函数执行结果的现实，并给出适当的操作提示显示。

演示系统可实现线性表的文件形式保存。其中，需要设计文件数据记录格式，以高效保存线性表数据逻辑结构的完整信息；需要设计线性表文件保存和加载操作合理模式。

演示系统还支持选择多个线性表进行管理。

## 1.2 系统设计

### 1.2.1 数据物理结构

1. 线性表的数据存储结构

线性表数据类型在程序中定义为结构体SqList ，具体定义方式如下：

typedef struct \_SqList {

ElemType \*elem; // 指向线性存储区

int length; // 表长

int list\_size; // 表容量

} SqList;

1. 多线性表的数据存储结构

多线性表操作通过定义数据元素类型为线性表SqList的数组，并设置一个整型常量用以保存工作表在线性表数组中的位置实现，具体定义方式如下：

SqList list\_pool[LIST\_POOL\_SIZE]; // 表池

int current\_list = 0; // 保存 L 在 list\_pool 中的位置

在本程序中，数据原子类型ElemType被定义为int整型。

### 1.2.2 演示系统

演示系统由用户操作界面与功能调用部分组成。

用户操作界面输出可选的线性表操作项、操作的结果及错误信息，基于c语言的标准输入输出库（stdio.h）及控制台的clear命令实现。

功能调用部分将用户输入的有关信息传递给线性数据结构的操作函数进行调用，并对函数的返回值进行处理判断输出相应的提示信息。

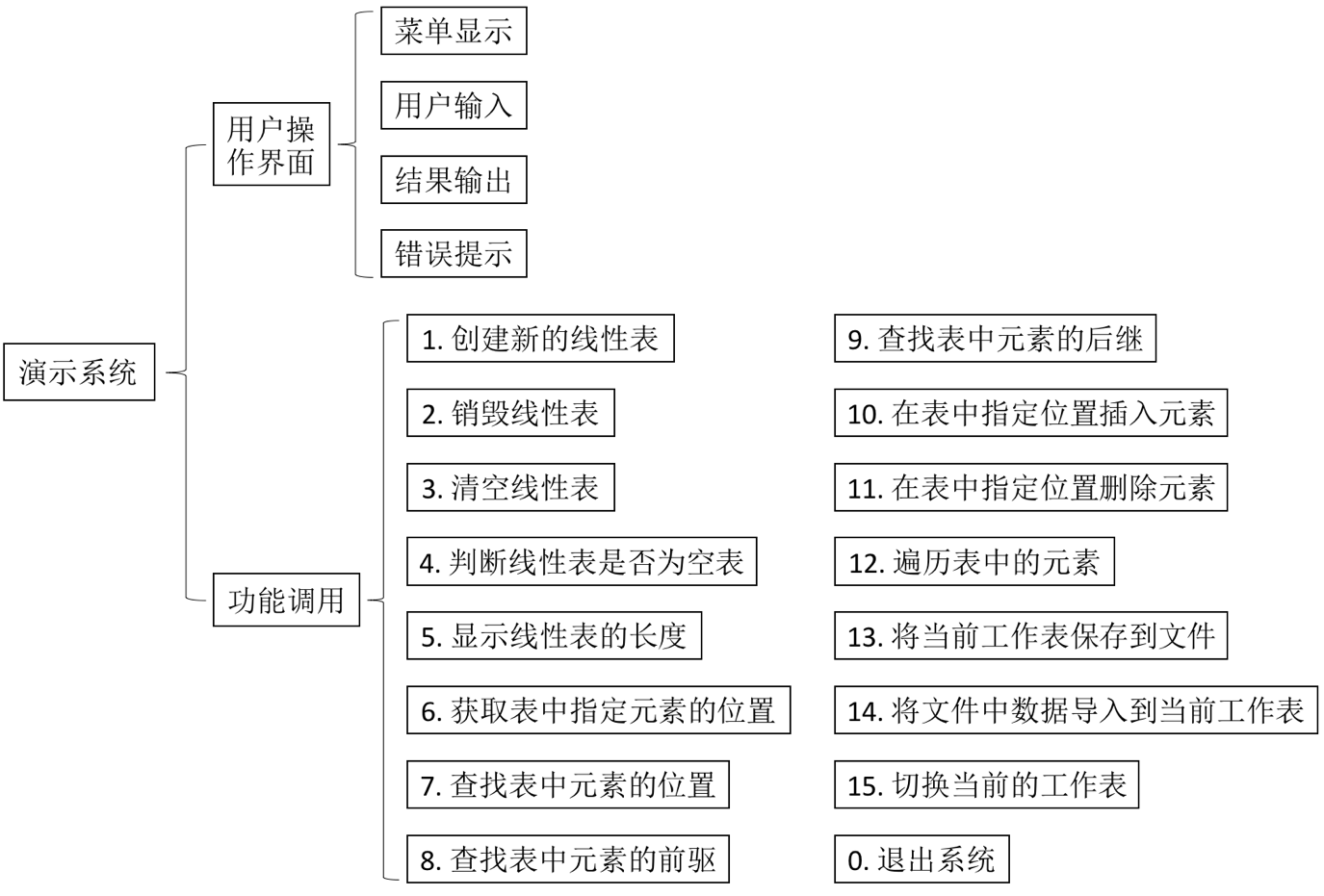


图1.2.2-1 演示系统模块结构图

### 1.2.3 数据文件存储格式

考虑到简洁高效与直观实用的原则，按如下方式设计数据文件存储格式。

文件格式：二进制文件，可用hexdump等工具查看数据值。

存放方式：顺序存放数据值的机器码，即直接将内存区块写入文件中。

### 1.2.4 线性表运算实现算法

1. SqList\_init(&L)

**算法思想：**首先判断L->elem是否为空指针，若不是则返回ERROR；否则使用malloc函数分配LIST\_INIT\_SIZE个ElemType类型的存储空间并使L->elem指向这一段空间，若分配失败则返回ERROR，否则L->length = 0，L->list\_size = LIST\_INIT\_SIZE，返回OK。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

1. SqList\_destroy(&L)

**算法思想：**首先判断L->elem是否为空指针，若是则返回ERROR；否则使用free函数释放L->elem指向的存储空间，并L->elem置空，L->length = 0，L->list\_size = 0。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

1. SqList\_clear(&L)

**算法思想：**首先判断L->elem是否为空指针，若是则返回ERROR；否则直接L->length = 0，再判断L->list\_size是否大于LIST\_INIT\_SIZE，若是则给L->elem重新分配大小为LIST\_INIT\_SIZE的空间，若分配失败则返回ERROR，否则L->list\_size = LIST\_INIT\_SIZE，返回OK。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

1. SqList\_empty(L)

**算法思想：**首先判断L->elem是否为空指针，若是则返回ERROR；然后判断L.length是否为零，若是则返回TRUE，否则返回FALSE。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

1. SqList\_length(L)

**算法思想：**首先判断L->elem是否为空指针，若是则返回ERROR；否则返回L.length。

**时空效率分析：**算法的时空复杂度为常量阶O(1) 。

1. SqList\_getElem(L, key, &val)

**算法思想：**首先判断L.elem是否为空指针，若是则返回ERROR；否则继续判断key是否在区间 [1, L.length] 之间，若不是则返回ERROR，否则将L.elem[key – 1] 赋值给val，返回OK。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

1. SqList\_locateElem(L, val)

**算法思想：**首先判断L.elem是否为空指针，若是则返回ERROR；否则遍历L.elem中的每一个元素，直到找到第一个值为val的元素，返回该元素的位序。若遍历后未找到值为val的元素，则返回0。

**时空效率分析：**算法中有一层循环，由于第一个值为val的元素可能在L.elem中的任意位置出现，也可能不出现，故算法的时间复杂度为线性阶O(n) 。

1. SqList\_priorElem(L, cur\_e, &pre\_e)

**算法思想：**首先判断L.elem是否为空指针，若是则返回ERROR；否则调用SqList\_locate(L, cur\_e) 获取L.elem中第一个值为cur\_e的元素的位序，若返回值为0（即表中没有值为cur\_e的元素）或1（表中第一个元素没有前驱）则返回ERROR，否则将该元素前驱的值赋给pre\_e带回，返回OK。

**时空效率分析：**算法调用了时间复杂度为线性阶O(n) 的SqList\_locateElem(…) ，除此之外再没有循环，故该算法的时间复杂度也为线性阶O(n) 。

1. SqList\_nextElem(L, cur\_e, &next\_e)

**算法思想：**首先判断L.elem是否为空指针，若是则返回ERROR；否则调用SqList\_locate(L, cur\_e) 获取L.elem中第一个值为cur\_e的元素的位序，若返回值为0（即表中没有值为cur\_e的元素）或L.length（表中最后一个元素没有后继）则返回ERROR，否则将该元素后继的值赋给next\_e带回，返回OK。

**时空效率分析：**类似于SqList\_priorElem(…) ，该算法的时间复杂度也为线性阶O(n) 。

1. SqList\_insert(&L, key, val)

**算法流程图：**

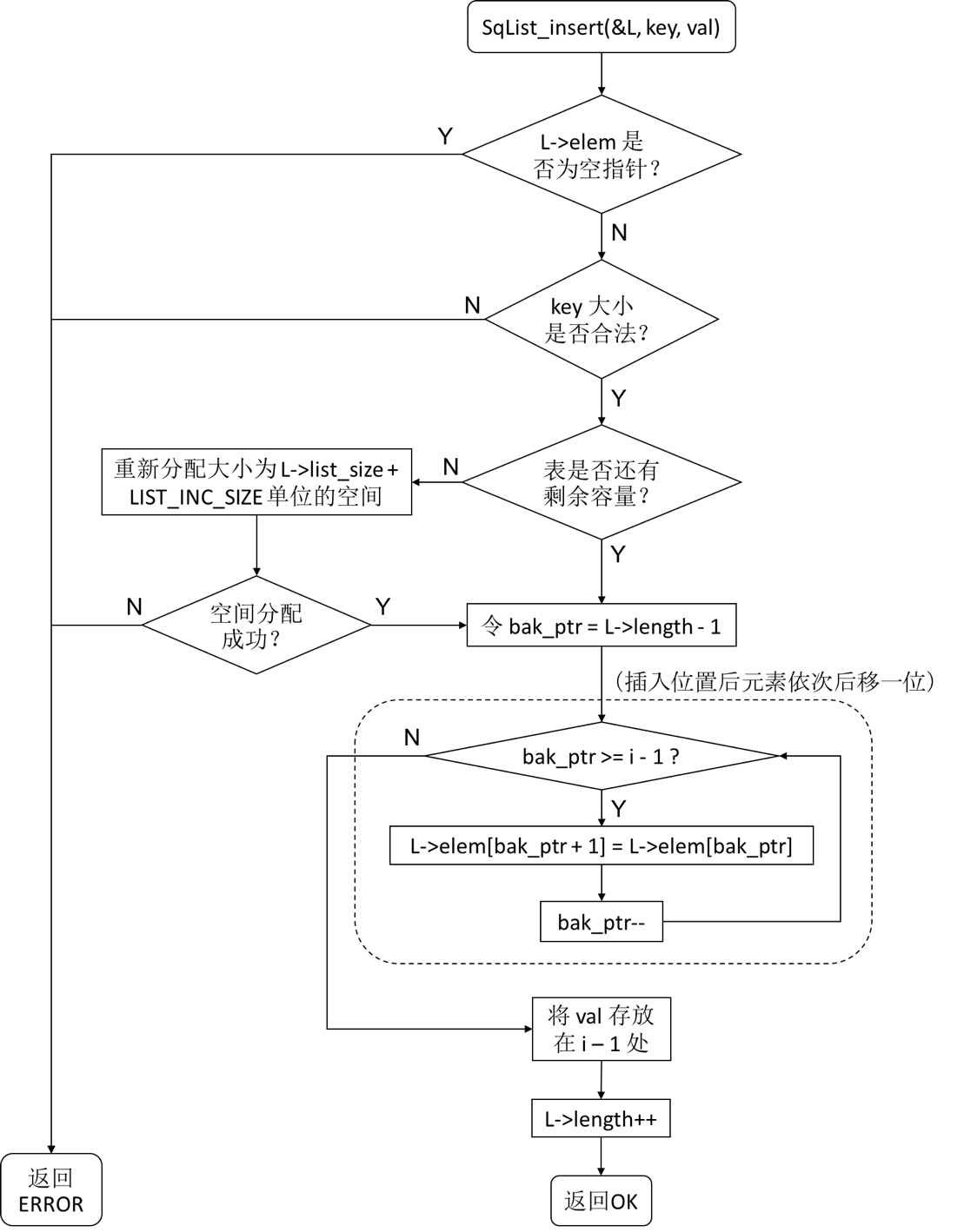


图1.2.4-1 SqList\_insert 算法流程图

**时空效率分析：**算法中要通过循环来移动元素，由于key值的大小不确定，故算法的时间复杂度为线性阶O(n) 。

1. SqList\_delete(&L, key, &val)

**算法思想：**与SqList\_insert(…) 算法类似，将由key指示位置的值由val带出后，将后面的元素依次前移。之后再检查L->list\_size - L->length是否大于LIST\_INC\_SIZE ，若是则重新分配大小为L->list\_size - LIST\_INC\_SIZE的空间，最后L->length-- ，返回OK。

**时空效率分析：**与SqList\_insert(…) 类似，算法的时间复杂度为线性阶O(n) 。

1. SqList\_traverse(L)

**算法思想：**首先判断L.elem是否为空指针，若是则返回ERROR；否则遍历L.elem并依次输出每一个数据元素，返回OK。

**时空效率分析：**算法的时间复杂度为线性阶O(n) 。

1. SqList\_saveList(fp, L)

**算法思想：**首先判断fp和L.elem是否为空指针，若是则返回ERROR；否则调用fwrite(…) 将内存区块写入fp指向的文件，返回OK。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

1. SqList\_loadList(fp, &L)

**算法思想：**首先判断fp是否为空指针，若是则返回ERROR；再判断L->elem是否非空，若非空则返回ERROR（防止覆盖操作）。计算文件大小存入filesize中，为L->elem分配内存空间，调用fread(…) 直接读取filesize个数据单元的数据到L->elem中，返回OK。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

### 1.2.5 多线性表运算实现算法

1. SqList\_selectList(&L, list\_pool, &cur\_l, tgt\_l)

**算法思想：**首先判断tgt\_l是否在合法的范围内，若不是则返回ERROR；否则将L下挂载的数据存到list\_pool[cur\_l] 中（cur\_l存放现操作的L在list\_pool中的位置），再将list\_pool[tgt\_l - 1] （tgt\_l为位序，下标从1开始）中的数据加载到L中，最后改变cur\_l的值为tgt\_l - 1 ，返回OK。

**时空效率分析：**算法的时间复杂度为常量阶O(1) 。

## 系统实现

### 1.3.1 实验环境

实验代码使用文本编辑器编写，clang（Apple LLVM version 9.0.0, clang-990.0.30）编译，源文件名为main.c 。

### 1.3.2 演示系统操作

演示系统采用键盘操作，控制台输出操作提示、可用的操作项与结果输出，输入不存在的操作编号不会执行任何操作。

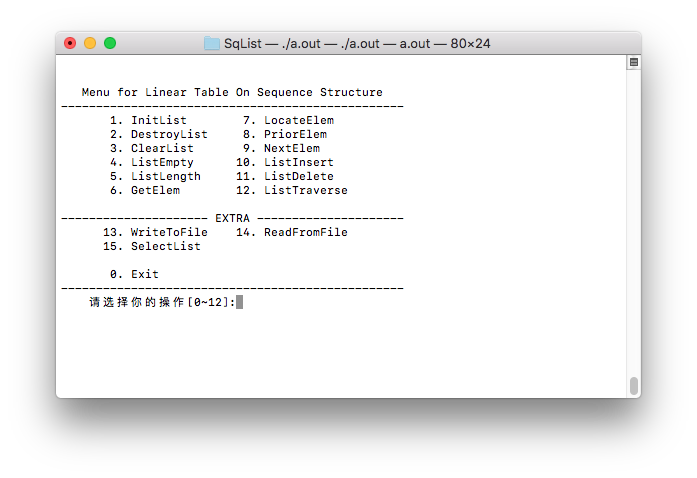


图1.3.2-1 启动程序后的操作菜单

输入操作指令0 ，程序退出。

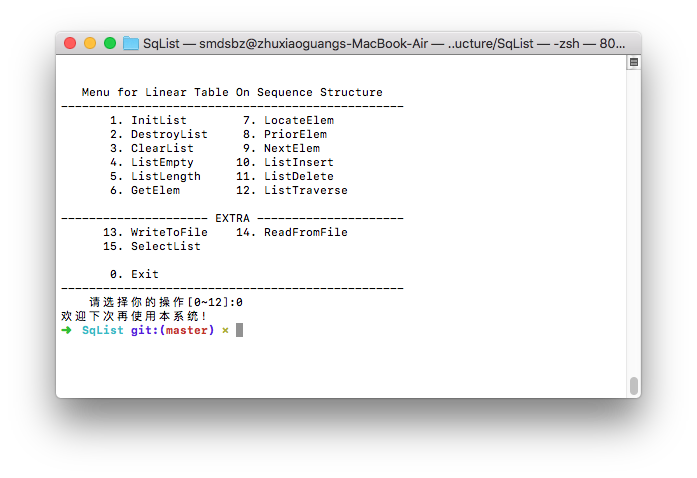


图1.3.2-2 输入操作0程序退出

### 1.3.3 测试计划

对系统的线性表操作功能按以下计划测试：

1. 检查常规状态下系统的工作情况；
2. 重点检查非法边界操作时的工作情况；
3. 检查多线性表操作。

**详见表1.3.3-1**



表1.3.3-1 线性表功能测试计划

### 1.3.4 测试结果

1. 在线性表还没有创建时，大多操作不被允许。

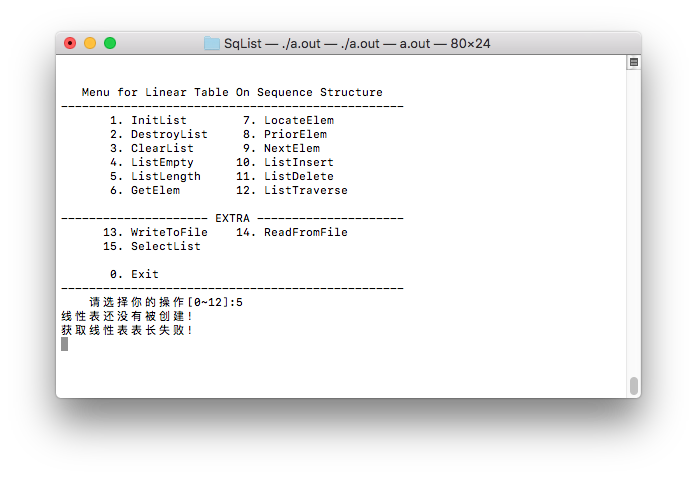


图1.3.4-1 无表状态下无法进行大部分操作

1. 显示 “线性表创建成功！”

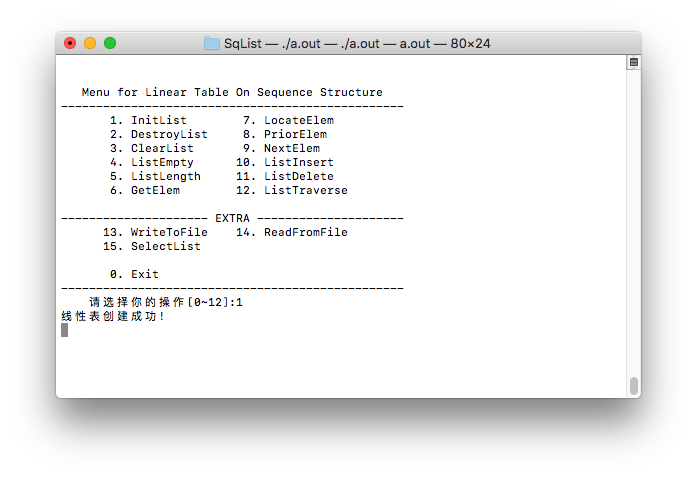


图1.3.4-2成功创建线性表

1. 显示 “线性表为空表！”

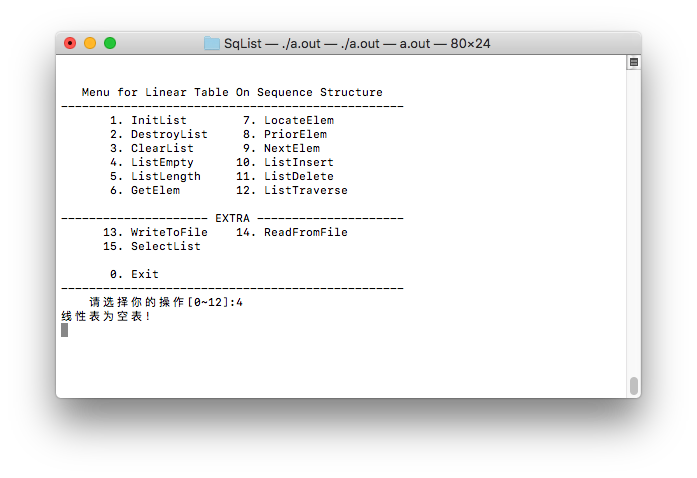


图1.3.4-3成功判断线性表为空

1. 由于FALSE被定义为0 ，因此当线性表表长也为0时，主程序无法区分是哪一种情况。可能出现类似问题的功能还有SqList\_empty(…) ，由于FALSE值也为0 ，故当获取线性表状态失败时，终端输出 “线性表还没有被创建！线性表不为空表！” 。

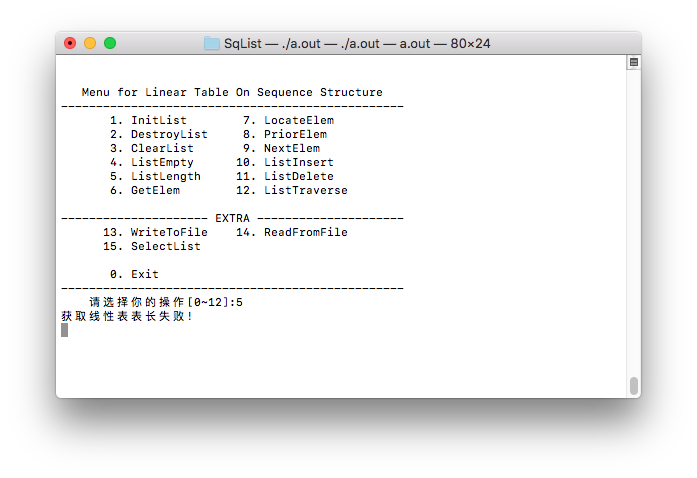


图1.3.4-4由于返回状态码值定义不恰当引起的问题

1. 6. 7. 显示 “成功插入元素！”

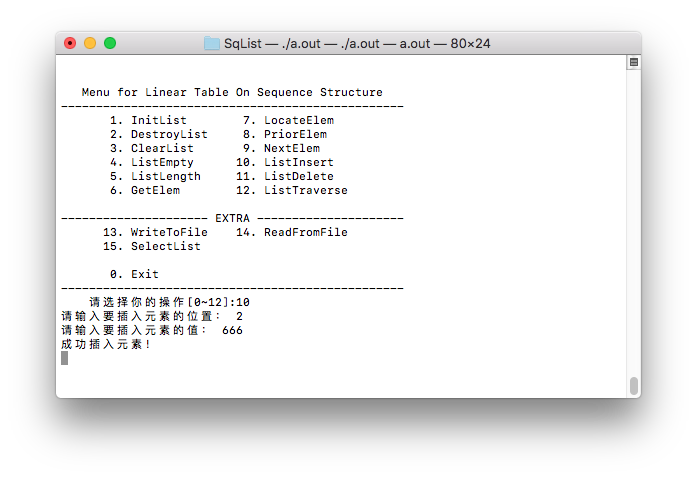


图1.3.4-5成功插入元素

1. 与步骤4不同，由于TRUE值为1 ，故能够正确显示 “线性表不为空表！”

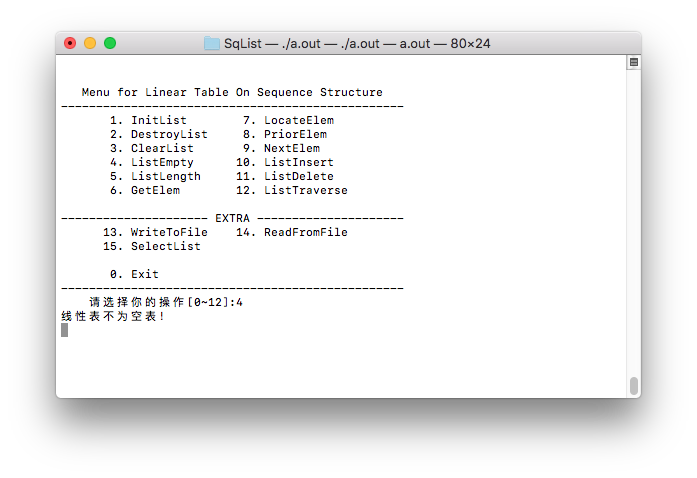


图1.3.4-6成功判断表为非空

1. 类似于步骤8，表长也能正确读取，终端显示 “线性表表长为 3 。”

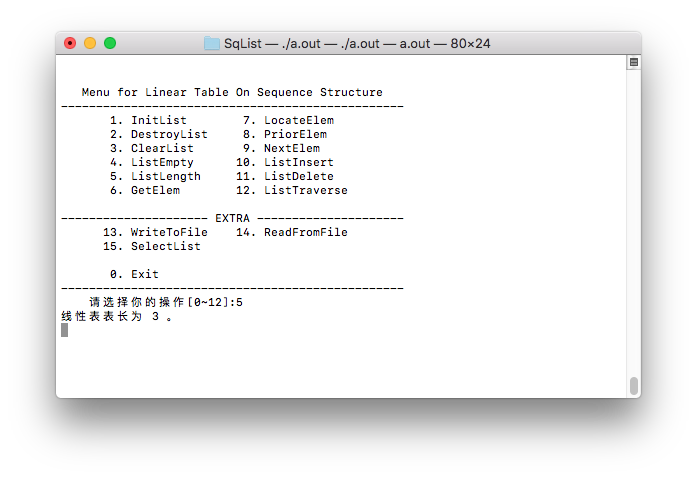


图1.3.4-7成功读取表长

1. 显示 “该元素值为 3 。”

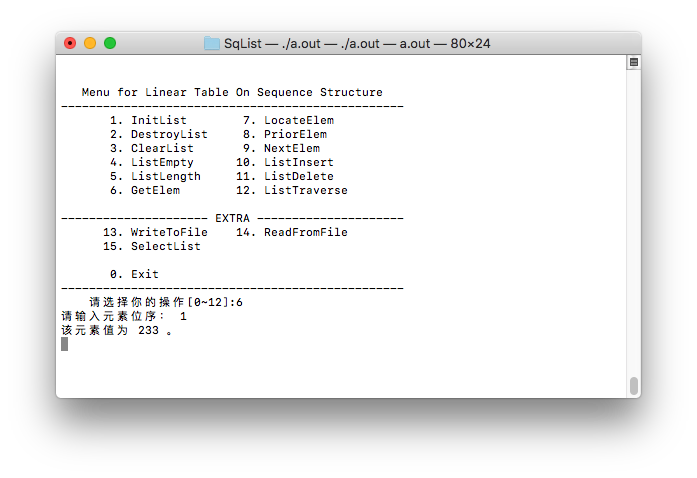


图1.3.4-8成功获取根据位序获取元素值

1. 跨界访问被阻止，显示 “输入地址格式不合法！获取元素值失败！”

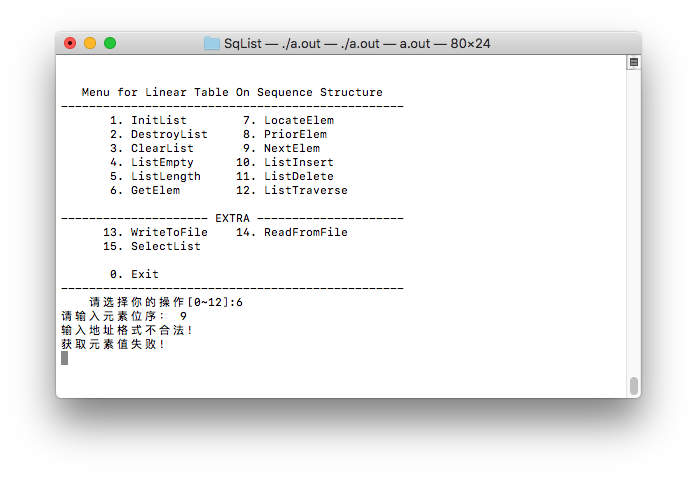


图1.3.4-9输入地址格式不合法时报错

1. 显示 “该元素在线性表中的位序为 2 。”

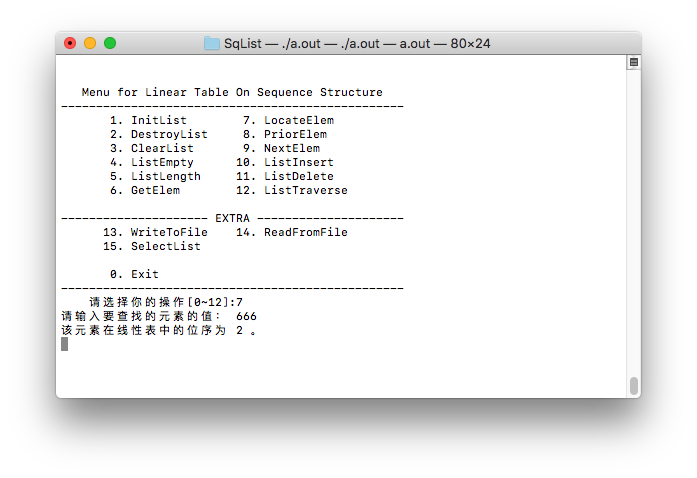


图1.3.4-10成功获取元素位序

1. 正确处理找不到满足条件元素的情况，显示 “该元素不在线性表中！”

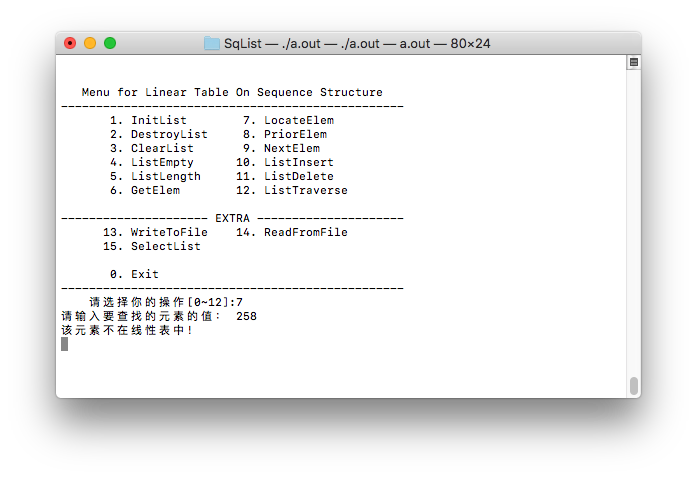


图1.3.4-11表中找不到满足要求元素时报错

1. 正确处理边界情况，显示 “元素值为输入值的元素为表中第一个元素！获取目标元素失败！”

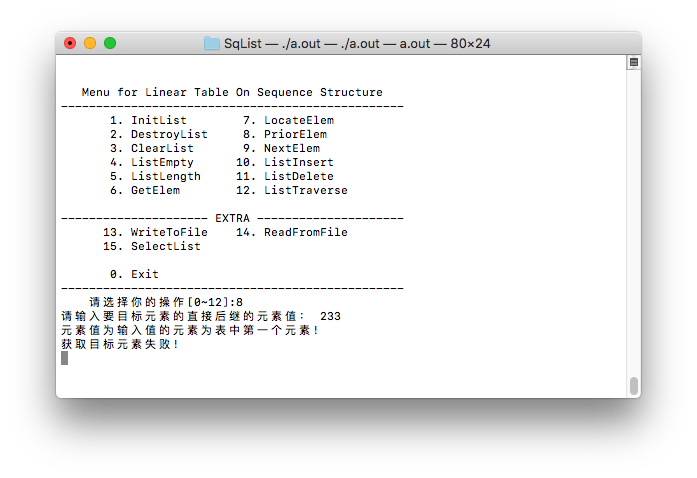


图1.3.4-12输入表中第一个元素值时报错

1. 显示 “目标元素值为 666 。”

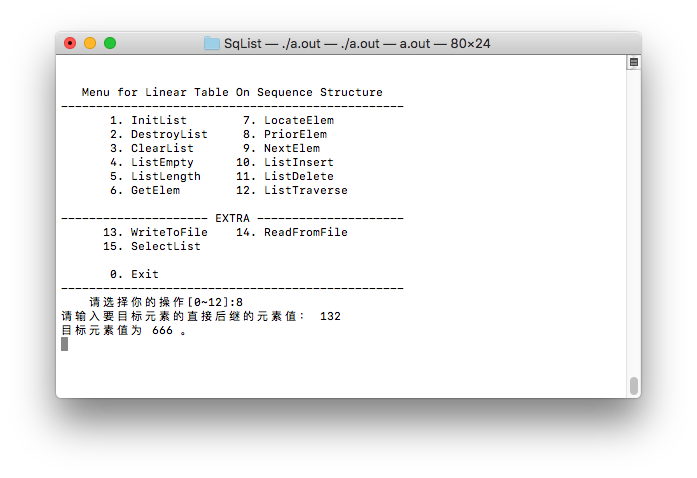


图1.3.4-13正常获取前驱元素的值

1. 显示 “目标元素值为 132 。”

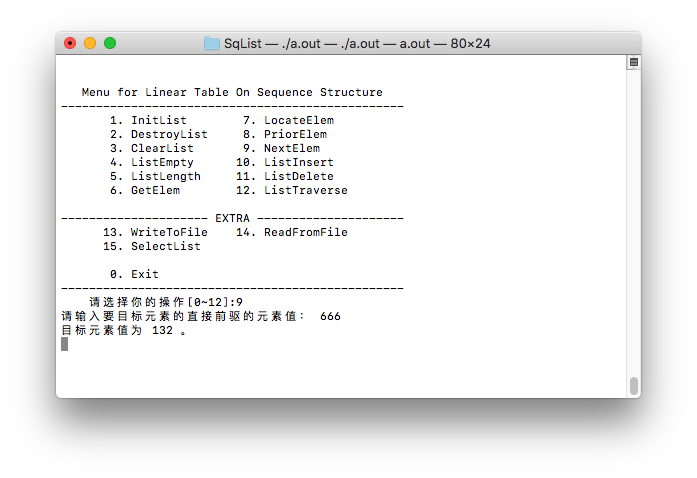


图1.3.4-14正常获取后记元素的值

1. 显示 “成功删除元素，其值为 233 。”

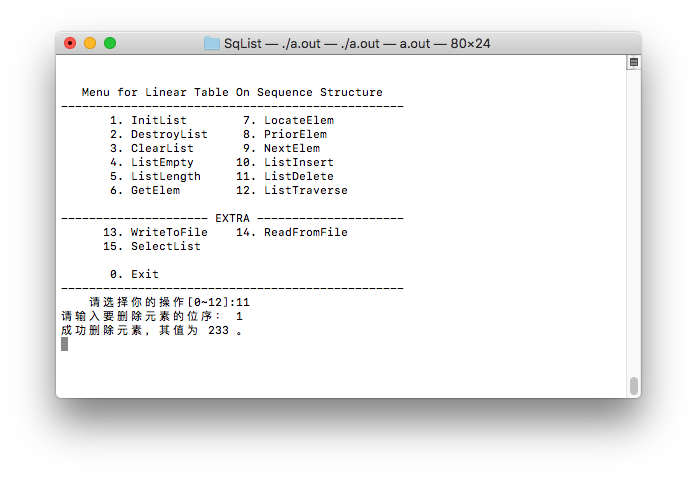


图1.3.4-15成功删除元素并显示其值

1. 显示 “666 132”

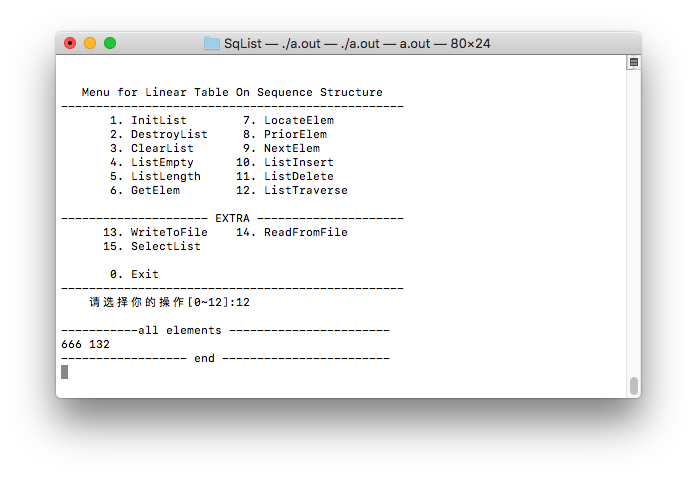


图1.3.4-16成功向终端打印所有元素

1. 显示 “成功写入文件！” 。文件中存放数据的二进制机器码。

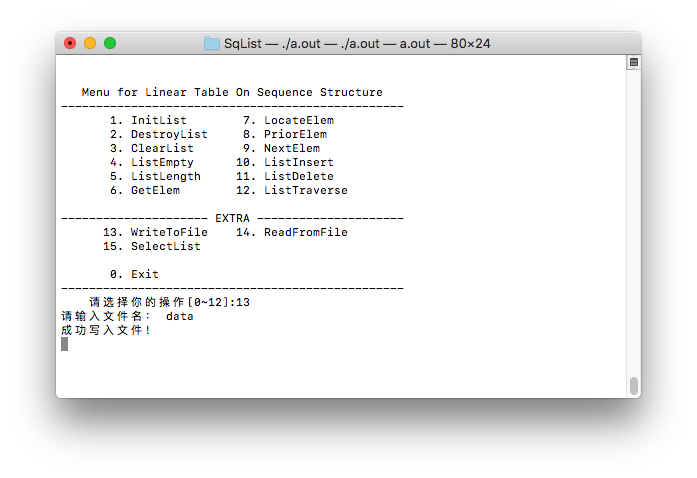


图1.3.4-17成功写入文件

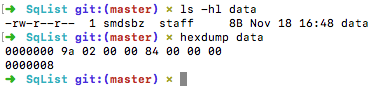


图1.3.4-18文件中为数据的机器码

1. 显示 “线性表置空成功！”

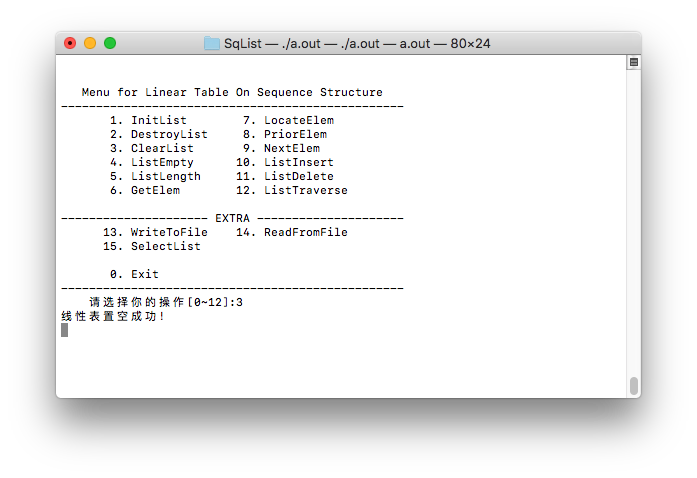


图1.3.4-19成功置空线性表

1. 显示 “切换操作表成功！” ，新表均已初始化为空表。

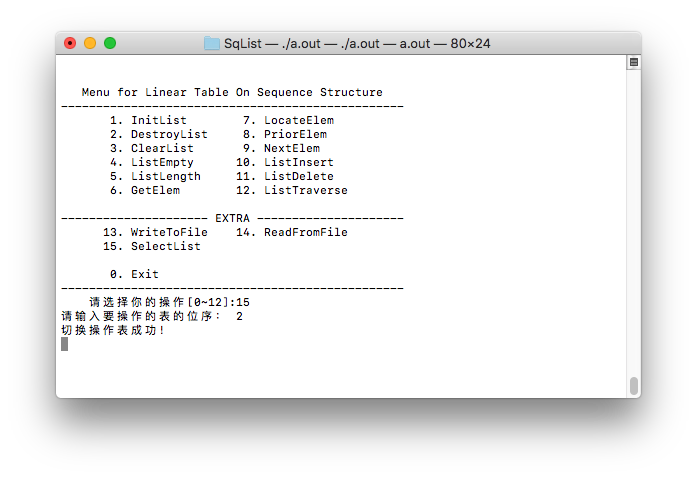


图1.3.4-20成功在多表中切换

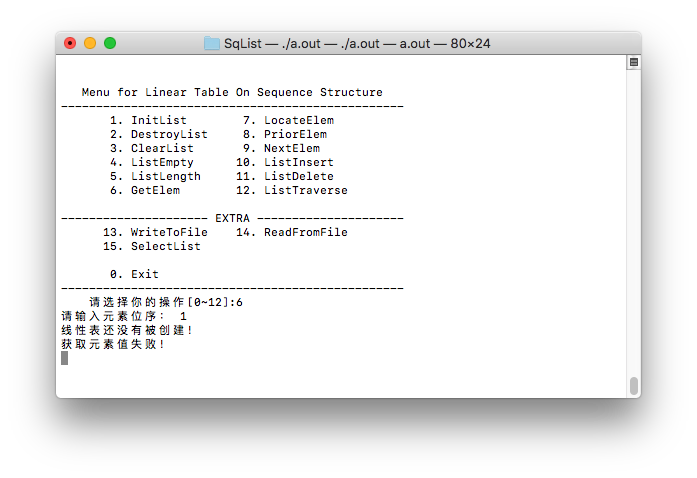


图1.3.4-21新表为空表

1. 显示 “成功从文件读取！”

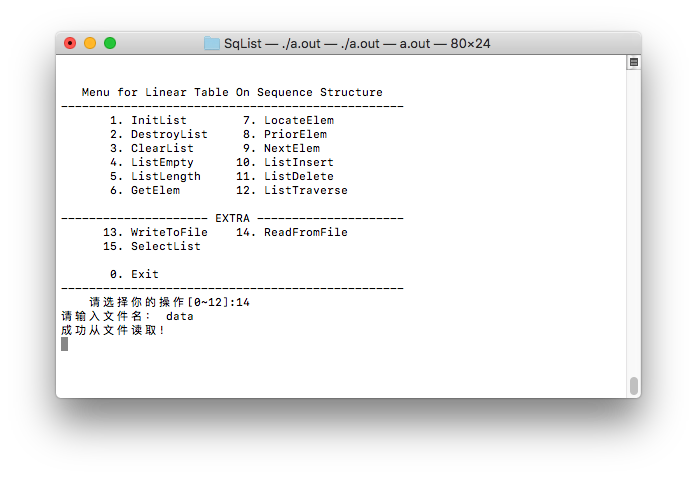


图1.3.4-22成功从文件中读取线性表

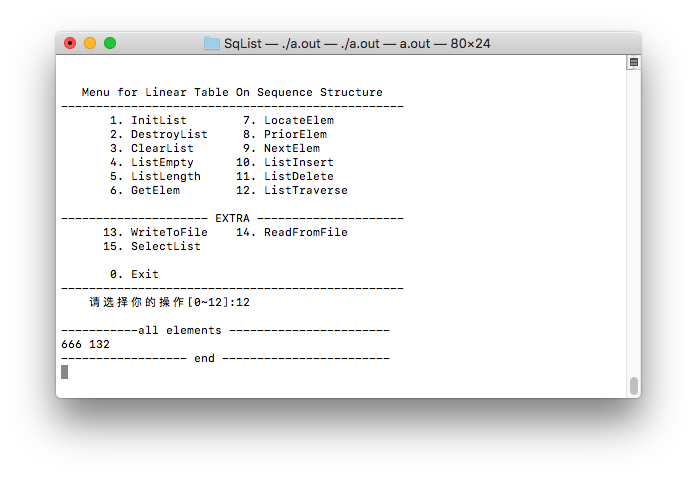


图1.3.4-23读取线性表后工作表不再为空

## 1.4 实验小结

本次上机实验我们亲手编写了基于顺序存储结构的线性表的具体代码实现。

我在实验过程中遇到了以下问题：

1. 如1.3.4步骤4中所述，特定状态下，程序不产生正确的输出。

**原因分析：**由于状态返回值定义不恰当，造成数据值的二义性，这有违算法设计的基本要求。

**解决方案：**重新定义状态返回值；重新设计函数，规定函数返回值一定是状态值，函数所求的数据值统一通过指针指向的单独数据域带回。

1. 无法由已规定的接口SqList\_selectList(&L, List[], tgt) 实现功能。

**原因分析：**由于L为main函数中声明所得变量，在标准C语言中无法通过改变其指向达到改变工作表的目的；而且L有自己的数据域，与List中的数据元素独立，因此无法判断当前工作表L在List中的位置，无法保存工作表中的改动到List中，切换操作将导致所有对工作表的改动丢失。

**解决方案：**增加一个函数参数cur指明L在List中的位置，通过将L的状态复制到List中对应表来保存改动，再执行加载操作，从而完成工作表的切换。

我在实验中感觉到自己编写代码的速度比较慢，大部分时间主要花在了编写函数参数合法性检查上。之后需要多加练习，保证程序健壮性的同时提高速度。

# 附录A 基于顺序存储结构线性表实现的源程序

## main.c

/\*

\* 基于顺序存储结构的线性表实现

\* ---- by U201610136 朱晓光 校际交流1601班

\*/

#include <stdio.h>

#include <stdlib.h>

// #define DEBUG

/\* Page 10 \*/

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASABLE -1

#define OVERFLOW -2

/\* 数据元素类型定义 \*/

typedef int status;

typedef int ElemType;

/\* Page 22 \*/

#define LIST\_INIT\_SIZE 2

#define LIST\_INC\_SIZE 1

#define LIST\_POOL\_SIZE 20

/\* 基于顺序存储结构的线性表数据结构定义 \*/

typedef struct \_SqList {

ElemType \*elem; // 指向线性存储区

int length; // 表长

int list\_size; // 表容量

} SqList;

/\* Page 19 \*/

status SqList\_init(SqList \*l);

status SqList\_destroy(SqList \*l);

status SqList\_clear(SqList \*l);

status SqList\_empty(SqList l);

int SqList\_length(SqList l);

status SqList\_getElem(SqList l, int i, ElemType \*e);

status Sqlist\_locateElem(SqList l, ElemType e);

status SqList\_priorElem(SqList l, ElemType cur, ElemType \*pre\_e);

status SqList\_nextElem(SqList l, ElemType cur, ElemType \*next\_e);

status SqList\_insert(SqList \*l, int i, ElemType e);

status SqList\_delete(SqList \*l, int i, ElemType \*e);

status SqList\_traverse(SqList l);

/\* 文件操作 \*/

status SqList\_writeToFile(SqList l, const char \*filename);

status SqList\_readFromFile(SqList \*l, const char \*path);

/\* 多表操作 \*/

status SqList\_selectList(SqList \*l, SqList pool[], int \*cur, int idx);

/\* Main \*/

int main(int argc, const char \*argv[]) {

// 初始化表池

SqList list\_pool[LIST\_POOL\_SIZE]; // 表池

int current\_list = 0; // 保存 L 在 list\_pool 中的位置

int i = 0; // **HACK:** 重用下面的 `op` 来节省内存空间（不过也就 4 个字节。。

for (; i < LIST\_POOL\_SIZE; i++) {

list\_pool[i].elem = NULL;

list\_pool[i].length = 0;

list\_pool[i].list\_size = 0;

}

SqList L;

int op=1; // 操作表示符

while (op) {

/\*system("cls");\*/ system("clear"); printf("\n\n");

/\* 操作选择提示 \*/

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

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

printf(" 1. InitList 7. LocateElem\n");

printf(" 2. DestroyList 8. PriorElem\n");

printf(" 3. ClearList 9. NextElem\n");

printf(" 4. ListEmpty 10. ListInsert\n");

printf(" 5. ListLength 11. ListDelete\n");

printf(" 6. GetElem 12. ListTraverse\n");

printf("\n");

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

printf(" 13. WriteToFile 14. ReadFromFile\n");

printf(" 15. SelectList\n");

printf("\n");

printf(" 0. Exit\n");

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

printf(" 请选择你的操作[0~12]:");

scanf("%d", &op);

/\* 执行操作 \*/

switch (op) {

case 1: {

if (SqList\_init(&L) == OK) { printf("线性表创建成功！\n"); }

else { printf("线性表创建失败！\n"); }

getchar(); getchar();

break;

}

case 2: {

if (SqList\_destroy(&L) == OK) { printf("线性表销毁成功！\n"); }

else { printf("线性表销毁失败！\n"); }

getchar(); getchar();

break;

}

case 3: {

if (SqList\_clear(&L) == OK) { printf("线性表置空成功！\n"); }

else { printf("线性表置空失败！\n"); }

getchar(); getchar();

break;

}

case 4: {

int result = SqList\_empty(L);

if (result == TRUE) { printf("线性表为空表！\n"); }

// **NOTE:** 由于 FALSE 和 ERROR 值均为 0 ，故当线性表还没有创建时，使用

// 该功能将同时输出“线性表还没有被创建！”和“线性表不为空表！”

else if (result == FALSE) { printf("线性表不为空表！\n"); }

else { printf("操作失败！\n"); }

getchar(); getchar();

break;

}

case 5: {

int ret = SqList\_length(L);

if (ret != ERROR) { printf("线性表表长为 %d 。\n", ret); }

else { printf("获取线性表表长失败！\n"); }

getchar(); getchar();

break;

}

case 6: {

int index; ElemType result; // 函数参数

printf("请输入元素位序： "); scanf("%d", &index);

int ret = SqList\_getElem(L, index, &result); // 存取返回状态码

if (ret == OK) { printf("该元素值为 %d 。\n", result); }

else { printf("获取元素值失败！\n"); }

getchar(); getchar();

break;

}

case 7: {

int target;

printf("请输入要查找的元素的值： "); scanf("%d", &target);

int ret = Sqlist\_locateElem(L, target);

if (ret == 0) { printf("该元素不在线性表中！\n"); }

else { printf("该元素在线性表中的位序为 %d 。\n", ret); }

getchar(); getchar();

break;

}

case 8: {

ElemType current, previous;

printf("请输入要目标元素的直接后继的元素值： "); scanf("%d", &current);

int ret = SqList\_priorElem(L, current, &previous);

if (ret == OK) { printf("目标元素值为 %d 。\n", previous); }

else { printf("获取目标元素失败！\n"); }

getchar(); getchar();

break;

}

case 9: {

ElemType current, next;

printf("请输入要目标元素的直接前驱的元素值： "); scanf("%d", &current);

int ret = SqList\_nextElem(L, current, &next);

if (ret == OK) { printf("目标元素值为 %d 。\n", next); }

else { printf("获取目标元素失败！\n"); }

getchar(); getchar();

break;

}

case 10: {

int key; ElemType value;

printf("请输入要插入元素的位置： "); scanf("%d", &key);

printf("请输入要插入元素的值： "); scanf("%d", &value);

if (SqList\_insert(&L, key, value) == OK) {

printf("成功插入元素！\n");

} else { printf("插入元素失败！\n"); }

getchar(); getchar();

break;

}

case 11: {

int key; ElemType value;

printf("请输入要删除元素的位序： "); scanf("%d", &key);

int ret = SqList\_delete(&L, key, &value);

if (ret == OK) { printf("成功删除元素，其值为 %d 。\n", value); }

else { printf("删除元素失败！\n"); }

getchar(); getchar();

break;

}

case 12: {

if (!SqList\_traverse(L)) { printf("线性表是空表！\n"); }

getchar(); getchar();

break;

}

case 13: {

char filename[100];

printf("请输入文件名： "); scanf("%s", filename);

if (SqList\_writeToFile(L, filename) == OK) {

printf("成功写入文件！\n");

} else { printf("写入文件失败！\n"); }

getchar(); getchar();

break;

}

case 14: {

char filename[100];

printf("请输入文件名： "); scanf("%s", filename);

int ret = SqList\_readFromFile(&L, filename);

if (ret == OK) { printf("成功从文件读取！\n"); }

else { printf("从文件读取失败！\n"); }

getchar(); getchar();

break;

}

case 15: {

int index;

printf("请输入要操作的表的位序： "); scanf("%d", &index);

if (SqList\_selectList(&L, list\_pool,

&current\_list, index) == OK) {

printf("切换操作表成功！\n");

} else { printf("切换操作表失败！\n"); }

getchar(); getchar();

break;

}

case 0: { break; } // 无效操作

} // switch

} // while

printf("欢迎下次再使用本系统！\n");

return 0; // **NOTE:** 这里就不返回 OK 了，不然 shell 会判异常退出的

}

/\*\*\* 函数具体实现 \*\*\*/

status

SqList\_init(SqList \*L) {

if (L->elem) {

printf("该线程下已有挂载的线性表！\n");

return ERROR;

}

L->elem = (ElemType \*)malloc(LIST\_INIT\_SIZE \* sizeof(ElemType));

if (!L->elem) { return ERROR; }

L->length = 0; L->list\_size = LIST\_INIT\_SIZE;

#ifdef DEBUG

L->length = 10; // 直接存入随机值（初始值）用作测试数据

L->elem[2] = 1; L->elem[3] = 2; L->elem[4] = 5;

#endif

return OK;

}

status

SqList\_traverse(SqList L) {

if (!L.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

int i;

printf("\n-----------all elements -----------------------\n");

for (i = 0; i < L.length; i++) { printf("%d ", L.elem[i]); }

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

return L.length;

}

status

SqList\_destroy(SqList \*l) {

// 合法性检测

if (!l->elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// 释放空间

free(l->elem);

l->elem = NULL; l->length = 0; l->list\_size = 0;

return OK;

}

status

SqList\_clear(SqList \*l) {

// 合法性检测

if (!l->elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// 直接将表大小置 0

l->length = 0;

// 减少空间占用

if (l->list\_size > LIST\_INIT\_SIZE) {

free(l->elem);

l->elem = (ElemType \*)malloc(LIST\_INIT\_SIZE \* sizeof(ElemType));

if (!l->elem) { return ERROR; }

}

return OK;

}

status

SqList\_empty(SqList l) {

// 合法性检测

if (!l.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

if (l.length) { return FALSE; }

return TRUE;

}

int

SqList\_length(SqList l) {

// 合法性检测

if (!l.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// 直接返回结构体中定义的表长

return l.length; // **NOTE:** 表长为零时返回 0 ，正好是 ERROR 的值，

// 主函数无法区分！

}

status

SqList\_getElem(SqList l, int idx, ElemType \*e) {

// 合法性检测

if (!l.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

if (idx > l.length || idx < 1) {

printf("输入地址格式不合法！\n");

return ERROR;

}

// 取出并赋值

\*e = l.elem[idx - 1];

return OK;

}

status

Sqlist\_locateElem(SqList l, ElemType e) {

// 合法性检测

if (!l.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// 遍历查找元素

int idx = 1; // 直接从 1 开始加，省得后面序列转位序还要加一

for (; l.elem[idx - 1] != e; idx++) {

if (idx > l.length) { return 0; }

}

// 找到了！返回！

return idx;

}

status

SqList\_priorElem(SqList l, ElemType cur\_e, ElemType \*pre\_e) {

// 合法性检测

if (!l.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// 获取 `cur\_e` 位序

int idx = Sqlist\_locateElem(l, cur\_e); // **NOTE:** 重复了合法性检测

idx -= 2; // 获取 `pre\_e` 对基地址的偏移量

// 第一个 -1 : 前一个元素

// 第二个 -1 : 位序转偏移量

// 检查 `cur\_e` 合法性

if (idx == ERROR - 2) {

printf("未找到元素值值为输入值的元素！\n");

return FALSE;

}

if (idx == -1) {

printf("元素值为输入值的元素为表中第一个元素！\n");

return ERROR;

} else { // 正常

// 给 `pre\_e` 赋值

\*pre\_e = l.elem[idx];

return OK;

}

}

status

SqList\_nextElem(SqList l, ElemType cur\_e, ElemType \*next\_e) {

// 合法性检测

if (!l.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// 获取 `cur\_e` 位序

int idx = Sqlist\_locateElem(l, cur\_e);

// idx = idx + 1 - 1;

// 检查 `cur\_e` 合法性

if (idx == ERROR) {

printf("未找到元素值值为输入值的元素！\n");

return FALSE;

}

if (idx == l.length) {

printf("元素值为输入值的元素为表中最后一个元素！\n");

return ERROR;

} else { // 正常

// 给 `next\_e` 赋值

\*next\_e = l.elem[idx];

return OK;

}

}

status

SqList\_insert(SqList \*l, int i, ElemType e) {

// 合法性检测

if (!l->elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// - `i` 范围限制

if ((l->length == 0 && i != 1) // 空表

|| (l->length > 0

&& (i < 1 || i > l->length))) { // 非空表，`i`应有范围

printf("输入地址格式不合法！\n");

return ERROR;

}

// 空间是否已满？

if (l->length == l->list\_size) {

// 扩充空间

l->list\_size += LIST\_INC\_SIZE;

// 将数据移动到空间更宽裕的地方

l->elem = (ElemType \*)realloc(l->elem, l->list\_size);

if (!l->elem) { return OVERFLOW; } // 检查

}

// 移动插入位置之后的所有元素

i--; // 转化为偏移量

int bak\_ptr = l->length - 1;

for (; bak\_ptr >= i; bak\_ptr--) {

l->elem[bak\_ptr + 1] = l->elem[bak\_ptr];

}

// 给目标位置赋值

l->elem[i] = e;

// 扩充可见表长

l->length += 1;

return OK;

}

status

SqList\_delete(SqList \*l, int i, ElemType \*e) {

// 合法性检测

if (!l->elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

if (i < 1 || i > l->length) {

printf("输入地址格式不合法！\n");

return ERROR;

}

// 删除元素值带出

i--; l->length -= 1;

\*e = l->elem[i];

// 删除位置后元素依次前移

for (; i < l->length; i++) {

l->elem[i] = l->elem[i + 1];

}

// 检查是否可以减小容量

// **HACK:** 与上一步进行合并可以减少时间消耗

if (l->list\_size - l->length > LIST\_INC\_SIZE) {

l->elem = (ElemType \*)realloc(l->elem,

(l->list\_size - LIST\_INC\_SIZE) \* sizeof(ElemType));

if (!l->elem) { return ERROR; }

}

return OK;

}

status

SqList\_saveList(FILE \*fp, SqList l) {

/\*

\* 配合 API 用的 wrapper

\*/

if (!fp) { return ERROR; } // NOTE: \*\*假设\*\* 传进来的 fp 是已打开的文件

if (!l.elem) { return ERROR; }

fwrite(l.elem, sizeof(ElemType), l.length, fp);

return OK;

}

status

SqList\_writeToFile(SqList l, const char \*filename) {

// 合法性检测

if (!l.elem) {

printf("线性表还没有被创建！\n");

return ERROR;

}

// 创建文件

FILE \*fp;

if ((fp = fopen(filename, "w")) == NULL) {

printf("不能够创建文件！\n");

return ERROR;

}

// 写入

if (SqList\_saveList(fp, l) != OK) {

/\* pass \*/

return ERROR;

}

fclose(fp); return OK; // 释放句柄并返回

}

status

SqList\_loadList(FILE \*fp, SqList \*l) {

/\*

\* 配合 API 用的 wrapper

\*/

if (!fp) { return ERROR; }

if (l->elem) { return ERROR; } // 不允许覆盖正挂载的表

// 初始化挂载结构体

// - 获取文件大小

fseek(fp, 0, SEEK\_END);

size\_t filesize = ftell(fp) / sizeof(ElemType);

fseek(fp, 0, SEEK\_SET);

// - 必要时创建空间

if ((!l->elem) || l->length < filesize) { // 没有空间 或 空间太小

if (l->elem) { free(l->elem); } // NOTE: 这个分支不会执行！

// 已禁止覆盖挂载表的操作！

l->elem = (ElemType \*)malloc(filesize \* sizeof(ElemType));

if (!l->elem) { return ERROR; }

}

// 一次性读取文件内容

l->length = filesize; l->list\_size = filesize;

fread(l->elem, sizeof(ElemType), filesize, fp);

return OK;

}

status

SqList\_readFromFile(SqList \*l, const char \*path) {

// 检查是否已有表挂载

if (l->elem) {

printf("已有挂载的线性表！\n");

return ERROR;

}

// 打开文件

FILE \*fp;

if ((fp = fopen(path, "r")) == NULL) {

printf("不能够读取文件！\n");

return ERROR;

}

// 读取

if (SqList\_loadList(fp, l) != OK) {

/\* pass \*/

return ERROR;

}

return OK;

}

status

SqList\_selectList(SqList \*l, SqList pool[], int \*current\_list, int idx) {

// 合法性检测

if (idx < 1 || idx > LIST\_POOL\_SIZE) {

printf("输入地址格式不合法！\n");

return ERROR;

}

idx--; // 位序转偏移量

// 向池中存储表数据

pool[\*current\_list] = \*l;

// 加载目标表数据

// NOTE: 由于 `L` 不是 malloc 出来的，不能直接改变其指向

l->elem = pool[idx].elem;

l->length = pool[idx].length;

l->list\_size = pool[idx].list\_size;

// 改动表指针

\*current\_list = idx;

return OK;

}