

**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**

**学 号： U201614xxx**

**姓 名： xxx**

**指导教师： 周时阳**

**报告日期： 2017年 12月 14 日**

**计算机科学与技术学院**

# 1 基于顺序存储结构的线性表实现

## 1.1 问题描述

通过实验达到⑴加深对线性表的概念、基本运算的理解；⑵熟练掌握线性表的逻辑结构与物理结构的关系；⑶物理结构采用顺序表,熟练掌握线性表的基本运算的实现。

## 1.2 系统设计

本系统提供一个顺序存储的线性表。该演示系统提供的操作有：表的初始化、销毁、清空、判空，求表长、获取数据元素、查找数据元素、获得前驱、获得后继、创建线性表、插入数据元素、删除数据元素、表的遍历。

### 1.2.1有关常量和类型定义

数据元素类型的定义：

typedef int status;

typedef int ElemType;

有关常量的定义：

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

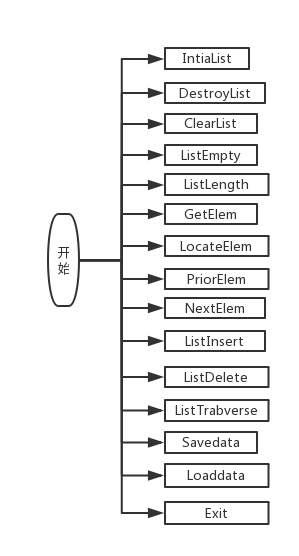
#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100

#define LISTINCREMENT 10

### 1.2.2 算法设计



**图一·算法设计流程图**

⑴初始化表：函数名称是InitaList(L)；初始条件是线性表L不存在已存在；操作结果是构造一个空的线性表。

⑵销毁表：函数名称是DestroyList(L)；初始条件是线性表L已存在；操作结果是销毁线性表L。

⑶清空表：函数名称是ClearList(L)；初始条件是线性表L已存在；操作结果是将L重置为空表。

⑷判定空表：函数名称是ListEmpty(L)；初始条件是线性表L已存在；操作结果是若L为空表则返回TRUE,否则返回FALSE。

⑸求表长：函数名称是ListLength(L)；初始条件是线性表已存在；操作结果是返回L中数据元素的个数。

⑹获得元素：函数名称是GetElem(L,i,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用e返回L中第i个数据元素的值。

⑺查找元素：函数名称是LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回L中第1个与e满足关系compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。

⑻获得前驱：函数名称是PriorElem(L,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。

⑼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。

⑽插入元素：函数名称是ListInsert(L,i,e)；初始条件是线性表L已存在，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的数据元素e。

⑾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。

⑿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 程序代码

/\*Linear Table On Sequence Structure\*/

#include <stdio.h>

#include <stdlib.h>

#include <malloc.h>

#define TRUE 1

#define FALSE 0

#define OK 1

#define ERROR 0

#define INFEASTABLE -1

#define OVERFLOW -2

#define LIST\_INIT\_SIZE 100//最大储存空间

#define LISTINCREMENT 10//增量

typedef int status;

typedef int ElemType;

typedef struct

{

ElemType \*elem;//数据存放空间

int length;

int listsize;

}sqList;

status IntialList(sqList \*L);

status DestoryList(sqList \*L);

status ClearList(sqList \*L);

status ListEmpty(sqList L);

status ListLength(sqList L);

status GetElem(sqList L, int i, ElemType \*e);

status LocateElem(sqList L,ElemType e);

status PriorElem(sqList L,ElemType cur,ElemType \*pre\_e);

status NextElem(sqList L,ElemType cur,ElemType \*next\_e);

status ListInsert(sqList \*L,int i,ElemType e);

status ListDelete(sqList \*L, int i, ElemType \*e);

status ListTrabverse(sqList L);

void main(void)

{

sqList L;

int choice = 1;

int position;//在数组中的位置

ElemType e = 0;

ElemType cur, pre\_e, next\_e;

FILE \*fin, \*fout;

IntialList(&L);

while (choice) {

system("cls");

printf("\n\n");

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

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

printf(" 1.IntialList 8.PriorElem \n");

printf(" 2.DestoryList 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.SaveList \n");

printf(" 7.LocateElem 14.OpenList \n");

printf(" 0.Exit\n");

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

printf("Please input your choice[0~14]:\n");

scanf("%d", &choice);

switch (choice) {

case 1:

{

if (IntialList(&L) == OK) printf("OK！\n");

else printf("FAIL！\n");

getchar();

getchar();

break;

}

case 2:

{

//printf("该功能未实现\n");

if (DestoryList(&L) == 1) {

printf("OK！\n");

}

else printf("FAILED！\n");

getchar();

getchar();

break;

}

case 4:

{

//printf("该功能未实现\n");

if (ListEmpty(L) == 1) {

printf("It's empty\n");

}

else printf("It's not empty\n");

getchar();

getchar();

break;

}

case 3:

{

//printf("该功能未实现\n");

if (ClearList(&L) == OK) {

printf("Clear\n");

}

else printf("Failed\n");

getchar();

getchar();

break;

}

case 5:

{

//printf("该功能未实现\n");

getchar();

printf("The length is%d\n", ListLength(L));

getchar();

getchar();

break;

}

case 6:

{

//printf("该功能未实现\n");

printf("input the position\n");

int order = 0;

scanf("%d", &order);

if (GetElem(L, order, &e)) {

printf("the %d th elemrnt is%d\n", position, e);

}

else {printf("not exist\n");}

getchar();

getchar();

break;

}

case 7:

{

printf("please input the element\n");

scanf("%d", &e);

if ((position = LocateElem(L, e))) printf("%d's position is%d\n", e, position);

else

printf("\n");

getchar();

getchar();

break;

}

case 8:

{

printf("please input the \n");

scanf("%d", &cur);

getchar();

if (PriorElem(L, cur, &pre\_e)) {

printf("%d\n", pre\_e);

}

else

{

printf("do not exist\n");

}

getchar();

getchar();

break;

}

case 9:

{

printf("please input th element\n");

scanf("%d", &cur);

if (NextElem(L, cur, &next\_e)){

printf("%d\n",next\_e);

}

else

{

printf("do not exist\n");

}

getchar();

getchar();

break;

}

case 10:

{

printf("please input the element\n");

scanf("%d", &e);

printf("please input the positon\n");

scanf("%d", &position);

if (ListInsert(&L, position, e) == OK)

{

printf("OK\n");

}

else

{

printf("FAIL!\n");

}

getchar();

getchar();

break;

}

case 11:

{

if (ListEmpty(L) == FALSE)

{

printf("input the position of the element\n");

scanf("%d\n", &position);

if (ListDelete(&L, position, &e) == OK) {

printf("OK\n");

}

else printf("FAIL!\n");

getchar();

getchar();

break;

}

}

case 12:

{

if (!ListTrabverse(L)) {

printf("OVER\n");

}

getchar();

getchar();

break;

}

case 13:

{

if ((fout = fopen("Input.dat", "w")) == NULL)

{

printf("File open erroer\n ");

return 1;

}

fwrite(L.elem, sizeof(ElemType), L.length, fout);

printf("OK\n");

fclose(fout);

getchar(); getchar();

break;

}

case 14:

{

if ((fin = fopen("Input.dat", "r")) == NULL)

{

printf("File open erroe\n ");

return 1;

}

while (fread(&L.elem[L.length], sizeof(ElemType), 1, fin))

L.length++;

printf("OK\n");

fclose(fin);

getchar();

getchar();

break;

}

case 0:

{

return 0;

}

}//end of switch

}//end of while

}

status IntialList(sqList \*L)

{

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

if (!L->elem)exit(OVERFLOW);

L->length = 0;

L->listsize = LIST\_INIT\_SIZE;

return OK;

}//1

DestoryList(sqList \*L)

{

if (L->length) {

free(L->elem);

L->elem = NULL;

L->length = 0;

return 0;

}

}//2

status ClearList(sqList \*L)

{

L->length = 0;

return OK;

}//3

status ListEmpty(sqList L)

{

if (L.length == 0) {

return TRUE;

}

else return FALSE;

}//4

int ListLength(sqList L)

{

return L.length;

}//5

status GetElem(sqList L, int i, ElemType \*e)

{

if (L.length == 0 || i < 1 || i > L.length) {

return ERROR;

}

\*e = L.elem[i - 1];

return \*e;

}//6

status LocateElem(sqList L, ElemType e)

{

int i = 0;

while (i < L.length) {

if (L.elem[i] == e)

return (i + 1);

else

++i;

}

return ERROR;

}//7

status PriorElem(sqList L, ElemType cur, ElemType \*pre\_e)

{

int tag;

if ((tag = LocateElem(L, cur)) && tag != 1) {

\*pre\_e = L.elem[tag - 2];

return OK;

}

else

{

return ERROR;

}

}//8

status NextElem(sqList L, ElemType cur\_e, ElemType \*next\_e)

{

int tag;

if ((tag = LocateElem(L, cur\_e))&&tag!=L.length) {

\*next\_e = L.elem[tag];

return OK;

}

else

{

return ERROR;

}

}//9

status ListInsert(sqList \*L, int i, ElemType e)

{

ElemType \*newbase;

ElemType \*p, \*q;

if (i<1 || i>L->length + 1) return ERROR;

if (L->length >= L->listsize)

{

newbase = (ElemType \*)realloc(L->elem,

(L->listsize + LISTINCREMENT) \* sizeof(ElemType));

if (!newbase) exit(OVERFLOW);

L->elem = newbase;

L->listsize += LISTINCREMENT;

}

q = &(L->elem[i - 1]);

for (p = &(L->elem[L->length - 1]); p >= q; --p) \*(p + 1) = \*p;

\*q = e;

++L->length;

return OK;

}//10

status ListDelete(sqList \*L, int i, ElemType \*e)

{

ElemType \*p, \*q;

if (i<1 || i>L->length + 1) return ERROR;

p = &(L->elem[i - 1]);

e = \*p;

q = L->elem + L->length - 1;

for (++p; p <= q; ++p) \*(p - 1) = \*p;

--L->length;

return OK;

}

status ListTrabverse(sqList L)

{

int a = 0;

if (L.length) {

for (a; a <L.length; a++) {

printf("%d\n", L.elem[a]);

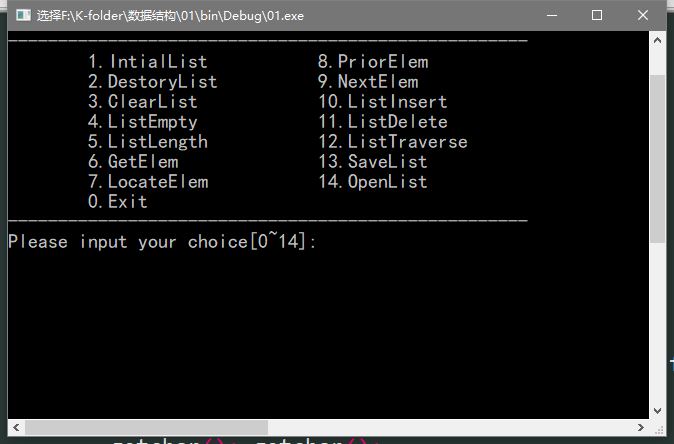
}

}

return L.length;

}//12

### 1.3.2 系统测试



**图二·程序主界面**

表1-1 程序测试用例

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
|  | 1234 | 1234 |  |
| 用例1 | 1 | 线性表创建成功 |  |
| 用例2 | 2 | 线性表销毁成功 |  |
| 用例3 | 3 | 线性表清空成功 |  |
| 用例4 | 4 | 线性表不为空 |  |
| 用例5 | 5 | 线性表长度为4 |  |
| 用例6 | 6 | 表中第1个数据为1 |  |
| 用例7 | 7 | 值为3的元素在表中的第三个位置 |  |
| 用例8 | 8 | 前驱为3 |  |
| 用例9 | 9 | 后继为3 |  |
| 用例10 | 10 | 略 |  |
| 用例11 | 11 | 元素2删除成功 | ↓ |
| 用例12 | 12 | 1234 |  |

## 1.4 实验小结

隔了一段时间没有写代码，所以第一次做实验的时候，C语言的一些东西都忘了，在做实验的过程中一边复习忘掉东西，一边理解新的知识。经过了这次实验，自己去实现课本上的算法，对顺序表有了一定的了解。

**2 基于链式存储结构的线性表实现**

**2.1 问题描述**

通过实验达到⑴加深对线性表的概念、基本运算的理解；⑵熟练掌握线性表的逻辑结构与物理结构的关系；⑶物理结构采用单链表,熟练掌握线性表的基本运算的实现。

**2.2 系统设计**

利用链表结构完成对线性表元素的基本操作：包括元素的读取，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。通过实验，熟悉链表的基本操作，加深对链表的了解。

⑴初始化表：函数名称是CreateList(L)；初始条件是线性表L不存在已存在；操作结果是构造一个空的线性表。

⑵销毁表：函数名称是DestroyList(L)；初始条件是线性表L已存在；操作结果是销毁线性表L。

⑶清空表：函数名称是ClearList(L)；初始条件是线性表L已存在；操作结果是将L重置为空表。

⑷判定空表：函数名称是ListEmpty(L)；初始条件是线性表L已存在；操作结果是若L为空表则返回TRUE,否则返回FALSE。

⑸求表长：函数名称是ListLength(L)；初始条件是线性表已存在；操作结果是返回L中数据元素的个数。

⑹获得元素：函数名称是GetElem(L,i,e)；初始条件是线性表已存在，1≤i≤ListLength(L)；操作结果是用e返回L中第i个数据元素的值。

⑺查找元素：函数名称是LocateElem(L,e,compare())；初始条件是线性表已存在；操作结果是返回L中第1个与e满足关系compare（）关系的数据元素的位序，若这样的数据元素不存在，则返回值为0。

⑻获得前驱：函数名称是PriorElem(L,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。

⑼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。

⑽插入元素：函数名称是ListInsert(L,i,e)；初始条件是线性表L已存在，1≤i≤ListLength(L)+1；操作结果是在L的第i个位置之前插入新的数据元素e。

⑾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。

⑿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()。

**2.3 系统实现**

### 2.3.1 程序代码

#include<stdio.h>

#include<malloc.h>

#include<Windows.h>

#define TRUE 1//是

#define FALSE 0//否

#define OK 1//正确

#define ERROR 0//错误

#define INFEASTABLE -1//不可执行

#define OVERFLOW -2//溢出

#define LIST\_INIT\_SIZE 100//表的大小

#define LISTINCREMENT 10//增量

typedef int Status;

typedef int ElemType;

typedef struct LNode

{

ElemType data;

struct LNode \*next;

}LNode, \*LinkList;//LinkList为指向该链表的指针

Status CreateList(LinkList \*L, int n);

Status DestoryList(LinkList \*L);

Status ClearList(LinkList L);

Status ListEmpty(LinkList L);

Status ListLength(LinkList L);

Status GetElem(LinkList L, int i, ElemType \*e);

Status LocateElem(LinkList L, ElemType e);

Status PriorElem(LinkList L, ElemType cur, ElemType \*pre\_e);

Status NextElem(LinkList L, ElemType cur, ElemType \*next\_e);

Status ListInsert(LinkList L, int i, ElemType \*e);

Status ListDelete(LinkList L, int i, ElemType a);

Status ListTrabverse(LinkList L);

LinkList p;

int length = 0;

void main(void)

{

p = (LinkList)malloc(sizeof(LNode));//动态指针

LinkList L;//创建新链表

int position = 0;//某元素在链表中的次序

int choice = 1;//选项

ElemType e = 0;//用于返回该元素

ElemType cur\_e, pre\_e, next\_e;//该元素，上一元素，下一元素

FILE \*fin, \*fout;

while (choice)

{

system("cls");

printf("\n\n");

printf(" 线性表的链式表示和实现 \n");

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

printf(" 1.CreateList 8.PriorElem \n");//初始化 查找前一元素

printf(" 2.DestoryList 9.NextElem \n");//删除链表 查找下一元素

printf(" 3.ClearList 10.ListInsert \n");//清空链表 插入元素

printf(" 4.ListEmpty 11.ListDelete \n");//链表是否为空 删除元素

printf(" 5.ListLength 12.ListTraverse \n");//链表长度 遍历

printf(" 6.GetElem 13.SaveList \n");//得到元素 存储链表

printf(" 7.LocateElem 14.OpenList \n");//返回元素位置 打开链表

printf(" 0.exit\n");//退出

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

printf("请输入操作选项:【0~14】\n");

scanf("%d", &choice);

switch (choice)

{

case 1:

{

int n = 1;

printf("请输入您想创建的节点数目。\n");

scanf("%d", &n);//创建长度为n的链表

CreateList(&L, n);

p = L;//头指针指向头节点

length = 0;

printf("初始化成功\n");

getchar();

getchar();

break;

}

case 2:

{

if (DestoryList(&L) == OK)

{

printf("破坏成功\n");

}

else printf("破坏失败\n");

getchar();

getchar();

break;

}

case 3:

{

if (ClearList(L) == OK) {

printf("清除成功\n");

}

else printf("清除失败\n");

getchar();

getchar();

break;

}

case 4:

{

if (ListEmpty(L) == 1) {

printf("It's empty\n");

}

else printf("It's not empty\n");

getchar();

getchar();

break;

}

case 5:

{

printf("链表的长度是%d\n", ListLength(L));

getchar();

getchar();

break;

}

case 6:

{

printf("输入您要查找的元素位置\n");

int order = 0;

scanf("%d", &order);

if (GetElem(L, order, &e)) {

printf("第%d个元素是%d\n", order, e);

}

else { printf("不存在\n"); }

getchar();

getchar();

break;

}

case 7:

{

printf("输入您想查找的元素\n");

scanf("%d", &e);

if ((position = LocateElem(L, e))) printf("%d'的位置是%d\n", e, position);

else

printf("\n");

getchar();

getchar();

break;

}

case 8:

{

printf("请输入指定元素\n");

scanf("%d", &cur\_e);

getchar();

if (PriorElem(L, cur\_e, &pre\_e)) {

printf("该元素的前驱元素是%d\n", pre\_e);

}

else

{

printf("前驱元素不存在\n");

}

getchar();

break;

}

case 9:

{

printf("请输入指定元素\n");

scanf("%d", &cur\_e);

if (NextElem(L, cur\_e, &next\_e))

{

printf("该元素的后继元素为%d\n", next\_e);

}

else

{

printf("后继元素不存在\n");

}

getchar();

getchar();

break;

}

case 10:

{

printf("输入想要插入的元素\n");

scanf("%d", &e);

printf("输入想要插入该元素的位置\n");

scanf("%d", &position);

length += 1;

if (ListInsert(L, position, e) == OK)

{

printf("插入成功\n");

}

else

{

printf("插入失败\n");

}

getchar();

getchar();

break;

}

case 11:

{

if (ListEmpty(L) == FALSE)

{

printf("输入元素的位置\n");

scanf("%d", &position);

if (ListDelete(L, position, e) == OK)

{

printf("删除成功");

}

else printf("删除失败\n");

getchar();

getchar();

break;

}

else

{

printf("此链表为空\n");

}

}

case 12:

{

if (ListTrabverse(L)) {

printf("遍历结束\n");

}

getchar();

getchar();

break;

}

}

}

}

Status CreateList(LinkList \*L, int n)

{

\*L = (LinkList)malloc(sizeof(LNode));

(\*L)->next = NULL;

for (int i = n; i > 0; --i)

{

printf("请输入元素\n");

p = (LinkList)malloc(sizeof(LNode));

scanf("%d", &(p->data));

p->next = (\*L)->next;

(\*L)->next = p;

}

return OK;

}

Status DestoryList(LinkList \*L)

{

if (\*L)

{

free(\*L);

return OK;

}

else return ERROR;

}

Status ClearList(LinkList L)

{

if (L)

{

free(L);

L->next = NULL;

return OK;

}

else return ERROR;

}

Status ListEmpty(LinkList L)

{

p = L;

int n = 1;

while (p != NULL)

{

p = p->next;

n++;

}

length = n;

if (n)

{

return FALSE;

}

else return TRUE;

}

Status ListLength(LinkList L)

{

p = L->next;

int n = 0;

while (p)

{

p = p->next;

n++;

}

return n;

}

Status GetElem(LinkList L, int i, ElemType \*e)

{

p = L->next;

int j = 1;

while (p&&j<i)

{

p = p->next;

++j;

}

if (!p||j >i)

{

return ERROR;

}

\*e = p->data;

return OK;

}

Status LocateElem(LinkList L, ElemType e)

{

int n = 1;

p = L->next;

while (e != p->data && p)

{

p = p->next;

if (p == NULL)

{

printf("该元素不存在\n");

return ERROR;

}

n++;

}

return n;

}

Status PriorElem(LinkList L, ElemType cur\_e, ElemType \*pre\_e)

{

int n = 0;

p = L->next;

while (p->data != cur\_e)

{

p = p->next;

if (p == NULL)

{

printf("该元素不存在于此链表中\n");

return ERROR;

}

n++;

}

if (n == 0)

{

return ERROR;

}

p = L->next;

int i = 0;

while (i != n - 1)

{

p = p->next;

++i;

}

\*pre\_e = p->data;

return OK;

}

Status NextElem(LinkList L, ElemType cur\_e, ElemType \*next\_e)

{

int n = 0;

p = L;

while (cur\_e != p->data)

{

p = p->next;

if (p == NULL)

{

printf("该元素不存在于此链表中\n");

return ERROR;

}

n++;

}

p = p->next;

if (p == NULL)

{

return ERROR;

}

else

{

\*next\_e = p->data;

return OK;

}

}

Status ListInsert(LinkList L, int i, ElemType \*e)//i是位置，e是数据

{

p = L;

int j = 0;

while (p && j < i - 1)

{

p = p->next;

++j;//寻找第i - 1个结点

}

if (!p || j > i - 1)

{

return ERROR;

}

LinkList s = (LinkList)malloc(sizeof(LNode));

s->data = e;

s->next = p->next;

p->next = s;

return OK;

}

Status ListDelete(LinkList L, int i, ElemType \*e)

{

p = L;

int j = 0;

while (p->next&&j < i - 1)

{

p = p->next;

++j;

}

if (!(p->next) || j > i - 1)return ERROR;

LinkList q;

q = p->next;

p->next = q->next;

e = q->data;

printf("%d\n", e);

free(q);

return OK;

}

Status ListTrabverse(LinkList L)

{

p = L->next;

printf("遍历结果如下\n");

if (p)

{

while (p)

{

printf("%d\n", p->data);

p = p->next;

}

return length;

}

else return ERROR;

}

2.3.2 系统测试

|  |  |  |  |
| --- | --- | --- | --- |
| 测试用例 | 程 序 输 入 | 理 论 结 果 | 运 行 结 果 |
| 1，12 | 4，1234 | 4321 |  |
| 用例1 | 1 | 线性表创建成功 |  |
| 用例2 | 2 | 线性表销毁成功 |  |
| 用例3 | 3 | 线性表清空成功 |  |
| 用例4 | 4 | 线性表不为空 |  |
| 用例5 | 5 | 线性表长度为4 |  |
| 用例6 | 6 | 表中第1个数据为4 |  |
| 用例7 | 7 | 值为3的元素在表中的第三个位置 |  |
| 用例8 | 8 | 前驱为3 |  |
| 用例9 | 9 | 后继为3 |  |
| 用例10 | 10 | 0，5 |  |
| 用例11 | 11 | 元素2删除成功 |  |
| 用例12 | 12 | 1234 | 略 |

**2.4 实验小结**

这次实验我做得无比艰难，先是因为不熟悉指针操作而不得不从头开始重写，又因为马虎而在内存分配上出了问题。虽然最后是赶在deadline前完成了，但这也给了我一个教训。下次我一定会认真学习数据结构相关知识，保质保量地完成实验。

3 基于二叉链表的二叉树实现

3.1实验目的

通过构造一个二叉链表，加深对二叉树概念和基本运算的理解。与此同时，熟练掌握二叉树逻辑结构和物理结构之间的关系。

## 3.2 二叉树演示系统设计

### 3.2.1 系统总体设计

本实验旨在设计一个基于二叉链表的二叉树存储系统。本演示系统不仅提供对二叉树的整体操作运算，包括构造空二叉树、销毁二叉树、创建二叉树、清空二叉树、判定空二叉树、求二叉树的深度以及前中后序遍历和按层遍历，而且支持对二叉树的结点进行运算，包括获得根结点、获得结点、结点赋值、获得双亲结点、获得左孩子结点、获得右孩子结点、获得左兄弟结点、获得右兄弟结点、插入子树和删除子树。此外还可以将文本文档中的字符读入程序从而构建一棵二叉树，以及将已构建的二叉树输出到文本文档中保存。

程序需要实现的20个基本功能分别是：InitBiTree(构造空二叉树)，DestroyBiTree(销毁二叉树)，CreateBiTree(先序创建二叉树)，ClearBiTree(清空二叉树)，BiTreeEmpty(判断二叉树是否为空)，BiTreeDepth (求二叉树的深度)，Root(返回二叉树的根)，Value(返回结点的值)，Assign(给结点赋值)，Parent(求双亲结点)，LeftChild(求结点左孩子)，RightChild(求结点右孩子)，LeftSibling(求结点左兄弟)，RightSiling(求结点右兄弟)，InsertChild(插入子树)，DeleteChild(删除子树)，PreOrderTraverse(先序遍历)，InOrderTrabverse(中序遍历)，PostOrderTrabverse(后序遍历)，LevelOrderTrabverse(层序遍历)。

函数名，对应的参数与操作结果如下所示。

（1）InitBiTree(&T)

操作结果：构造空二叉树T。

（2）DestroyBiTree(&T)

初始条件：二叉树T已存在。

操作结果：销毁二叉树T。

（3）CreateBiTree(&T,definition)

初始条件：definition 给出二叉树T的定义。

操作结果：按definition构造二叉树T。

（4）ClearBiTree (&T)

初始条件：二叉树T存在。

操作结果：将二叉树T清空。

（5）BiTreeEmpty(T)

初始条件：二叉树T存在。

操作结果：若T为空二叉树，则返回TRUE,否则返回FALSE.

（6）BiTreeDepth(T)

初始条件：二叉树T存在。

操作结果：返回T的深度。

（7）Root(T)

初始条件：二叉树T已存在。

操作结果：返回T的根。

（8）Value(T，e)

初始条件：二叉树T已存在，e是T中的某个结点。

操作结果：返回e的值。

（9）Assign(T，&e，value)

初始条件：二叉树T已存在，e是T中的某个结点。

操作结果：结点e赋值为value。

（10）Parent(T，e)

初始条件：二叉树T已存在，e是T中的某个结点。

操作结果：若e是T的非根结点，则返回它的双亲结点指针，否则返回NULL。

（11）LeftChild(T，e)

初始条件：二叉树T存在，e是T中某个节点。

操作结果：返回e的左孩子结点指针。若e无左孩子，则返回NULL。

（12）RightChild(T，e)

初始条件：二叉树T已存在，e是T中某个结点。

操作结果：返回e的右孩子结点指针。若e无右孩子，则返回NULL。

（13）LeftSibling(T，e)

初始条件：二叉树T存在，e是T中某个结点。

操作结果：返回e的左兄弟结点指针。若e是T的左孩子或者无左兄弟，则返回NULL。

（14）RightSibling(T，e)

初始条件：二叉树T已存在，e是T中某个结点。

操作结果：返回e的右兄弟结点指针。若e是T的右孩子或者无有兄弟，则返回NULL。

（15）InsertChild(T,p,LR,c)

初始条件：二叉树T存在，p指向T中的某个结点，LR为0或1，,非空二叉树c与T不相交且右子树为空。

操作结果：根据LR为0或者1，插入c为T中p所指结点的左或右子树，p所指结点的原有左子树或右子树则为c的右子树。

（16）DeleteChild(T，p，LR)

初始条件：二叉树T存在，p指向T中的某个结点，LR为0或1。

操作结果：根据LR为0或者1，删除c为T中p所指结点的左或右子树。

（17）PreOrderTraverse（T,Visit（））

初始条件：二叉树T存在，Visit是对结点操作的应用函数。

操作结果：先序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

（18）InOrderTraverse（T,Visit（））

初始条件：二叉树T存在，Visit是对结点操作的应用函数。

操作结果：中序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

（19）PostOrderTraverse（T,Visit（））

初始条件：二叉树T存在，Visit是对结点操作的应用函数。

操作结果：后序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

（20）LevelOrderTraverse（T,Visit（））

初始条件：二叉树T存在，Visit是对结点操作的应用函数。

操作结果：层序遍历t，对每个结点调用函数Visit一次且一次，一旦调用失败，则操作失败。

（21）SaveBiTree（T）

初始条件：二叉树T存在。

操作结果：先序遍历二叉树，将结点和空（“#”）以先序存入文件。

（22）ReadBiTree（filename）

操作结果：读取文件中储存的二叉树信息，并构建新二叉树。

### 3.2.2变量的定义及数据结构的设计

#define OVERFLOW -2

#define OK 1

#define TRUE 1

#define FALSE 0

#define ERROR 0

typedef int Status;

typedef char ElemType;

3.2.1中定义的22个函数均属于对二叉树的操作。二叉树由一个根结点和若干个树枝结点构成。该二叉树的逻辑结构如图3.1所示。

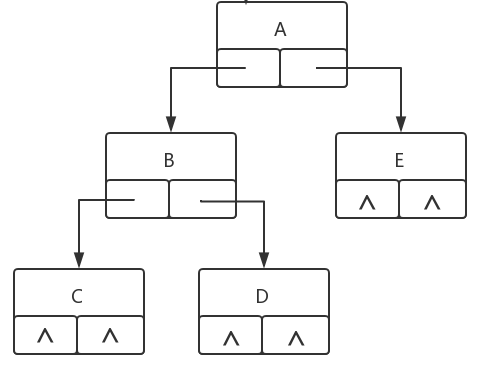


图3.1 二叉树的存储结构

### 3.2.3 算法与函数设计

函数的设计可以分为四大类：①构造类函数：包括创建、销毁、清空、插入子树和删除子树。②遍历函数：进行前序、中序、后序、层序等遍历操作。②求值函数：判空、求深度、求根、给特定结点赋值、求特定结点的值、求特定结点的父母结点、求特定结点的左右子女结点、求特定结点的左右兄弟结点。④系统功能函数：包括保存、读取二叉树以及更换当前操作的二叉树。

除构造和销毁函数外，其余的函数在调用前均会判断二叉树是否已被构造。

各函数的功能及复杂度如下所示。

（1）InitBiTree(&T)

创建（或称为初始化）一棵二叉树，令其根节点为空，令标记flag=1。此时并不为其分配空间，即没有树的头结点，在创建二叉树时在为期分配空间。

时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（2）DestroyBiTree(&T)

使用后序遍历的方法，将二叉树的每个结点的空间释放，最后将根节点的空间释放，令标记flag=0。

时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（3）CreateBiTree(&T,definition)

传入根节点指针，使用递归算法，由用户自行输入每个根节点的信息，并且在创建时按照层序遍历的方式对二叉树的每个结点编号。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（4）ClearBiTree (&T)

使用后序遍历的方法，将二叉树的每个结点的空间释放，最后并不释放根节点，而是令根节点为空，并且使标记flag=2。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（5）BiTreeEmpty(T)

判断二叉树的根节点是否为空，若为空则返回1，不为空则返回2。

时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（6）BiTreeDepth(T)

从根节点出发，树的深度等于左右子树深度较大值加1，若左右子树均为空，则该树深度为1；若根节点为空，则深度为0。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（7）Root(T)

返回二叉树的根节点，若该二叉树的根节点为空，则提示用户“二叉树为空！”。

时间复杂度T(n) = O(1)

空间复杂度S(n) = O(1)

（8）Value(T，e)

首先在主函数中询问用户要查找第几个结点的值（对结点编号已在CreateBiTree函数中实现），然后将二叉树中的每个结点存入顺序队列中，通过从头遍历，当存在结点的编号为e时返回该结点的值，若遍历结束后仍未找满足条件的结点，则返回0。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（9）Assign(T，&e，value)

首先在主函数中询问用户要修改第几个结点的值（对结点编号已在CreateBiTree函数中实现），然后将二叉树中的每个结点存入顺序队列中，通过从头遍历，当存在结点的编号为e时将该结点的值用value替换，若遍历结束后仍未找满足条件的结点，则返回0。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（10）Parent(T，e)

将二叉树中的所有顶点入队，并且从头遍历判断是否有某一结点的左孩子或右孩子的数据与用户所要查找的结点的值相同，若相同则返回；若遍历完整个二叉树后都没找到，则返回‘0’。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（11）LeftChild(T，e)

使用Point函数返回用户所输入的值对应结点的指针，最后返回这个指针所指向结点的左孩子。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（12）RightChild(T，e)

使用Point函数返回用户所输入的值对应结点的指针，最后返回这个指针所指向结点的右孩子。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（13）LeftSibling(T，e)

使用变量a保存函数Parent函数返回的用户输入数据e的父母的值，再使用Point函数返回指向a的指针，如果该指针所指结点的左孩子不为空且不等于e，则返回左孩子的值，若为空或等于e，则返回‘0’。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（14）RightSibling(T，e)

使用变量a保存函数Parent函数返回的用户输入数据e的父母的值，再使用Point函数返回指向a的指针，如果该指针所指结点的右孩子不为空且不等于e，则返回右孩子的值，若为空或等于e，则返回‘0’。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（15）InsertChild(T,p,LR,c)

与创建树的方式类似，同样也需要创建一个新的子树，只不过在创建之后，需要将原树对应结点的子树指针指向子树的根结点；与此同时，将子树根结点的右子树指向原来母树结点所指的树枝结点——因为整个原因，插入的子树的右子树必须为空。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（16）DeleteChild(T，p，LR)

首先通过Point函数返回用户想要删除子树结点的指针，然后根据LR等于0或1 判断是删除左子树或是右子树，最后将p的左子树或右子树的空间释放。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（17）PreOrderTraverse（T,Visit（））

采用递归算法，先访问根节点，即对于任一结点，其可看做是根结点，因此可以直接访问，访问完之后，若其左孩子不为空，按相同规则访问它的左子树；当访问其左子树时，再访问它的右子树。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（18）InOrderTraverse（T,Visit（））

采用递归算法，先访问左子树，即对于任一结点，其可看做是根结点，因此可以直接访问，访问完之后，若其左孩子为空，访问根节点，再访问它的右子树。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（19）PostOrderTraverse（T,Visit（））

采用递归算法，先访问左子树，即对于任一结点，其可看做是根结点，因此可以直接访问，访问完之后，若其左孩子为空，访问右子树，再访问它的根节点。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（20）LevelOrderTraverse（T,Visit（））

构建一个队列，按层将每一次的节点输入队列，然后使用BiTreeDepth函数得到二叉树的层数，最后使用循环将该二叉树每一层的数据输出。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（21）SaveBiTree（\*T）

将二叉树以先序序列存入文件（当某结点的左孩子或右孩子为空时，保存一个“#”到文件中）。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(1)

（22）ReadBiTree（T）

传入根节点指针，使用递归算法，将文件中的内容按字符串的形式读出，并且在创建时按照层序遍历的方式对二叉树的每个结点编号。

时间复杂度T(n) = O(n)

空间复杂度S(n) = O(n)

（程序源代码详见文件夹）

## 3.3 系统测试

3.4实验小结

在本次实验中，对二叉树的基本操作有了一个整体的理解。其中，二叉树的构造和二叉树的遍历是两个的最为重要的内容。而他们都用到了“递归”这一功能。由于之前使用的一直是循环，这次写代码的过程中给我带来了不小的困扰。

其次，在储存过程中，我还用到了队列的知识，虽然并没有做队列的相关实验，测试过程中bug频出，但最终我还是克服了困难。

## 3.5 附录C 基于二叉链表的二叉树实现源代码

见附件