# 线性结构-线性表

## 实验简介

数据结构中的逻辑结构分为线性结构和非线性结构，这一章和下一章我们会介绍线性结构，简单地说，线性结构是n个数据元素的有序（次序）集合，它有下列几个特征：

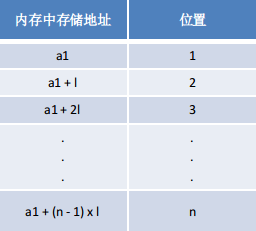
1．集合中必存在唯一的一个"第一个元素"； 2．集合中必存在唯一的一个"最后的元素"； 3．除最后元素之外，其它数据元素均有唯一的"后继"； 4．除第一元素之外，其它数据元素均有唯一的"前驱"。

这一章我们就来讲解线性结构中线性表，它是最常用且最简单的一种数据结构。线性表是一个含有n≥0个结点的有限序列，对于其中的结点，有且仅有一个开始结点没有前驱但有一个后继结点，有且仅有一个终端结点没有后继但有一个前驱结点，其它的结点都有且仅有一个前驱和一个后继结点。一般地，一个线性表可以表示成一个线性序列：k1，k2，…，kn，其中k1是开始结点，kn是终端结点。

一般线性表包含下列基本操作：初始化、销毁、重置为空表、判断是否为空、获取长度、根据位置获取对应元素、查找元素、获取指定元素的前驱和后继元素、插入元素、删除元素、遍历元素。

## 1. 线性表的顺序表示和实现

线性表的顺序表示指的是用物理上的一段连续的地址来存储数据元素，如下图所示。如果第一个元素的在内存上的地址为a1，每个元素占用的空间是l，那么第n个元素的地址就是a1+(n-1) x l。



只要确定了第一个元素的地址，那么我们可以对线性表中的任一元素随机存取，由于编程语言中的数组也有随机存取的特点，下面就用数组来描述线性表的顺序存储结构。

下面是代码实现：

#include <stdio.h>#include <stdlib.h>

#define TRUE 1#define FALSE 0#define OK 1#define ERROR 0#define INIT\_SIZE 10 //初始化表长#define INCREMENT\_SIZE 5 //分配增量

typedef int Status;

typedef int Elemtype;

/\*

\* 存储结构

\*/

typedef struct

{

Elemtype \*elem; //存储空间基址

int length; //当前长度

int size; //当前分配的表长大小

}SqList;

/\*

\* 初始化一个空的线性表

\*/Status InitList(SqList \*L){

L->elem = (Elemtype \*) malloc(INIT\_SIZE \* sizeof(Elemtype));

if (!L->elem)

{

return ERROR;

}

L->length = 0;

L->size = INIT\_SIZE;

return OK;

}

/\*

\* 销毁线性表

\*/Status DestroyList(SqList \*L){

free(L->elem);

L->length = 0;

L->size = 0;

return OK;

}

/\*

\* 清空线性表

\*/Status ClearList(SqList \*L){

L->length = 0;

return OK;

}

/\*

\* 判断线性表是否为空

\*/Status isEmpty(const SqList L){

if (0 == L.length)

{

return TRUE;

}

else

{

return FALSE;

}

}

/\*

\* 获取长度

\*/Status getLength(const SqList L){

return L.length;

}

/\*

\* 根据位置获取元素

\*/Status GetElem(const SqList L, int i, Elemtype \*e){

if (i < 1 || i > L.length)

{

return ERROR;

}

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

return OK;

}

/\*

\* 比较两个元素是否相等

\*/Status compare(Elemtype e1, Elemtype e2){

if (e1 == e2)

{

return 0;

}

else if (e1 < e2)

{

return -1;

}

else

{

return 1;

}

}

/\*

\* 查找元素

\*/Status FindElem(const SqList L, Elemtype e, Status (\*compare)(Elemtype, Elemtype)){

int i;

for (i = 0; i < L.length; i++)

{

if (!(\*compare)(L.elem[i], e))

{

return i + 1;

}

}

if (i >= L.length)

{

return ERROR;

}

}

/\*

\* 查找前驱元素

\*/Status PreElem(const SqList L, Elemtype cur\_e, Elemtype \*pre\_e){

int i;

for (i = 0; i < L.length; i++)

{

if (cur\_e == L.elem[i])

{

if (i != 0)

{

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

}

else

{

return ERROR;

}

}

}

if (i >= L.length)

{

return ERROR;

}

}

/\*

\* 查找后继元素

\*/Status NextElem(const SqList L, Elemtype cur\_e, Elemtype \*next\_e){

int i;

for (i = 0; i < L.length; i++)

{

if (cur\_e == L.elem[i])

{

if (i < L.length - 1)

{

\*next\_e = L.elem[i + 1];

return OK;

}

else

{

return ERROR;

}

}

}

if (i >= L.length)

{

return ERROR;

}

}

/\*

\* 插入元素

\*/Status InsertElem(SqList \*L, int i, Elemtype e){

Elemtype \*new;

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

{

return ERROR;

}

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

{

new = (Elemtype\*) realloc(L->elem, (L->size + INCREMENT\_SIZE) \* sizeof(Elemtype));

if (!new)

{

return ERROR;

}

L->elem = new;

L->size += INCREMENT\_SIZE;

}

Elemtype \*p = &L->elem[i - 1];

Elemtype \*q = &L->elem[L->length - 1];

for (; q >= p; q--)

{

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

}

\*p = e;

++L->length;

return OK;

}

/\*

\* 删除元素并返回其值

\*/Status DeleteElem(SqList \*L, int i, Elemtype \*e){

if (i < 1 || i > L->length)

{

return ERROR;

}

Elemtype \*p = &L->elem[i - 1];

\*e = \*p;

for (; p < &L->elem[L->length]; p++)

{

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

}

--L->length;

return OK;

}

/\*

\* 访问元素

\*/void visit(Elemtype e){

printf("%d ", e);

}

/\*

\* 遍历线性表

\*/Status TraverseList(const SqList L, void (\*visit)(Elemtype)){

int i;

for(i = 0; i < L.length; i++)

{

visit(L.elem[i]);

}

return OK;

}

/\*

\* 主函数测试

\*/int main(){

SqList L;

if (InitList(&L))

{

Elemtype e;

printf("init\_success\n");

int i;

for (i = 0; i < 10; i++)

{

InsertElem(&L, i + 1, i);

}

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

if (GetElem(L, 1, &e)) {

printf("The first element is %d\n", e);

}

else

{

printf("element is not exist\n");

}

if (isEmpty(L))

{

printf("list is empty\n");

}

else

{

printf("list is not empty\n");

}

printf("The 5 at %d\n", FindElem(L, 5, \*compare));

PreElem(L, 6, &e);

printf("The 6's previous element is %d\n", e);

NextElem(L, 6, &e);

printf("The 6's next element is %d\n", e);

DeleteElem(&L, 1, &e);

printf("delete first element is %d\n", e);

printf("list:");

TraverseList(L,visit);

if (DestroyList(&L))

{

printf("\ndestroy\_success\n");

}

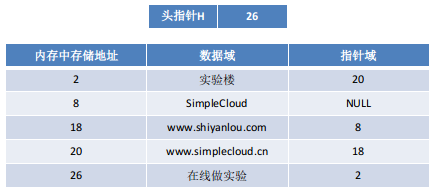
}

}

## 2. 线性表的链式表示和实现

上一节讨论了线性表的顺序表示和实现，这一节我们来讨论线性表的链式表示和实现，线性表的顺序存储结构是逻辑位置和物理位置都相邻，而链式存储结构是逻辑位置相邻，但物理位置不一定相邻，相比顺序存储结构，它不能随机存取，但在插入和删除操作时不需要移动元素，大大提高了增加和删除元素的效率。

通常链式存储结构会有一个个结点组成，结点中包含两个域一个是数据域，一个是指针域，数据域中存储数据，指针域中存储下一个后继元素的地址，如下图所示，这一个个结点组成链表，也称线性链表或单链表。



单链表的逻辑结构如下图所示

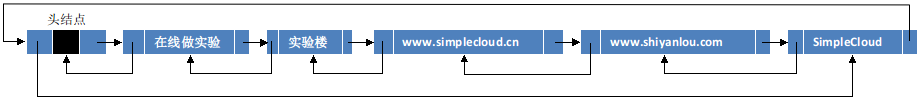
IMG_258

除了单链表之外还有循环链表和双向链表，循环链表的特点是最后一个结点的指针指向头结点，形成一个环，双向链表的特点是结点中多了一个指向前驱元素的指针，这两种链表的逻辑结构如下面两张图所示

**循环链表**



**双向链表**



这里主要代码实现一下单链表：

#include <stdio.h>#include <stdlib.h>

#define TRUE 1#define FALSE 0#define OK 1#define ERROR 0#define OVERFLOW -2

typedef int ElemType;

typedef int Status;

/\*

\* 存储结构

\*/

typedef struct LNode

{

ElemType data;

struct LNode \*next;

}LNode, \*LinkList;

/\*

\* 初始化线性表

\*/void InitList(LinkList \*L){

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

if (!L)

{

exit(OVERFLOW);

}

(\*L)->next = NULL;

}

/\*

\* 销毁线性表

\*/void DestroyList(LinkList \*L){

LinkList temp;

while (\*L)

{

temp = (\*L)->next;

free(\*L);

\*L = temp;

}

}

/\*

\* 清空线性表

\*/void ClearList(LinkList L){

LinkList p = L->next;

L->next = NULL;

DestroyList(&p);

}

/\*

\* 判断是否为空

\*/Status isEmpty(LinkList L){

if (L->next)

{

return FALSE;

}

else

{

return TRUE;

}

}

/\*

\* 获取长度

\*/int GetLength(LinkList L){

int i = 0;

LinkList p = L->next;

while (p)

{

i++;

p = p->next;

}

return i;

}

/\*

\* 根据位置获取元素

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

int j = 1;

LinkList p = L->next;

while (p && j < i)

{

j++;

p = p->next;

}

if (!p || j > i)

{

return ERROR;

}

\*e = p->data;

return OK;

}

/\*

\* 比较两个元素是否相等

\*/Status compare(ElemType e1, ElemType e2){

if (e1 == e2)

{

return 0;

}

else if (e1 < e2)

{

return -1;

}

else

{

return 1;

}

}

/\*

\* 查找指定元素的位置

\*/int FindElem(LinkList L, ElemType e, Status (\*compare)(ElemType, ElemType)){

int i = 0;

LinkList p = L->next;

while (p)

{

i++;

if (!compare(p->data, e))

{

return i;

}

p = p->next;

}

return 0;

}

/\*

\* 获取前驱元素

\*/Status PreElem(LinkList L, ElemType cur\_e, ElemType \*pre\_e){

LinkList q, p = L->next;

while (p->next)

{

q = p->next;

if (q->data == cur\_e)

{

\*pre\_e = p->data;

return OK;

}

p = q;

}

return ERROR;

}

/\*

\* 获取后继元素

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

LinkList p = L->next;

while (p->next)

{

if (p->data == cur\_e)

{

\*next\_e = p->next->data;

return OK;

}

p = p->next;

}

return ERROR;

}

/\*

\* 插入元素

\*/Status InsertElem(LinkList L, int i, ElemType e){

int j = 0;

LinkList s, p = L;

while (p && j < i - 1)

{

j++;

p = p->next;

}

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

{

return ERROR;

}

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

s->data = e;

s->next = p->next;

p->next = s;

return OK;

}

/\*

\* 删除元素并返回值

\*/Status DeleteElem(LinkList L, int i, ElemType \*e){

int j = 0;

LinkList q, p = L;

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

{

j++;

p = p->next;

}

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

{

return ERROR;

}

q = p->next;

p->next = q->next;

\*e = q->data;

free(q);

return OK;

}

/\*

\* 访问元素

\*/void visit(ElemType e){

printf("%d ", e);

}

/\*

\* 遍历线性表

\*/void TraverseList(LinkList L, void (\*visit)(ElemType)){

LinkList p = L->next;

while (p)

{

visit(p->data);

p = p->next;

}

}

int main(){

LinkList L;

InitList(&L);

ElemType e;

int i;

if (L)

{

printf("init success\n");

}

if (isEmpty(L))

{

printf("list is empty\n");

}

for (i = 0; i < 10; i++)

{

InsertElem(L, i + 1, i);

}

if (GetElem(L, 1, &e)) {

printf("The first element is %d\n", e);

}

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

printf("The 5 at %d\n", FindElem(L, 5, \*compare));

PreElem(L, 6, &e);

printf("The 6's previous element is %d\n", e);

NextElem(L, 6, &e);

printf("The 6's next element is %d\n", e);

DeleteElem(L, 1, &e);

printf("delete first element is %d\n", e);

printf("list:");

TraverseList(L,visit);

DestroyList(&L);

if (!L) {

printf("\ndestroy success\n");

}

}

## 3. 小结

这一章我们讲解了线性结构中线性表的顺序及链式的表示和实现，顺序存储结构中的元素在逻辑位置和物理位置上都相邻，链式存储结构中的元素在逻辑位置上相邻，但在物理位置上不一定相邻，顺序存储结构读取元素的效率比较高，链式存储结构添加和删除元素的效率比较高。链式存储结构除了单链表之外，还有循环链表和双向链表。

## 作业

1、使用顺序存储结构或链式存储结构实现一元多项式的加法运算。

2、编写函数，实现反转链表功能。