链滴

摸一摸数据结构（线性表）

作者：stillwarter

原文链接：https://ld246.com/article/1630476900426

来源网站：链滴

线性结构

我们讨论的是线性结构，注意与数学上的线性结构区分。

其特点是：

1. 存在唯一的一个被称为“第一个”的数据元素
2. 存在唯一的一个被称为“最后一个”的数据元素
3. 除第一个之外，集合中每个数据元素均只有一个前驱
4. 出最后一个外，集合中每个数据元素均只有一个后继

线性表概念（定义）

简单说，一个线性表示 n 个相同数据类型数据元素的有限序列。例如一副扑克牌；英文字母表（A.....）；

在复杂一点的线性表里，一个数据元素可以由若干**数据项**组成，在这种情况下，常把数据元素称为记，含有大量记录的线性表又称为文件。

线性表是一种逻辑结构，表示元素之间的一对一关系；顺序表和链表是存储结构，两者不属于同一层。

这里我们把线性表的抽象数据类型（ADT）略去

线性表的实现

ps：所用代码均为github上的代码，猫猫只是重新写了一遍。

顺序表实现与操作

这是头文件的（.h）的引入与定义

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "Status.h"
#define listinitsize 100
#define listincrement 10
typedef int lelemtype_sq;
typedef struct{
    lelemtype_sq *elem;
    int length;
    int listsize;
}sqlist;
```

初始化

创建一个函数，传入值是指针类型的 sqlist，我们为数据项 elem 开辟空间存放 elem 数据（因为在

构体力 elem 是指针类型，不能直接赋值，需要为其指定一块空间将数据存放到这里），此时再对结体的其他数据项赋值

```
Status initlist(sqlist *L){
    (*L).elem=(lelemtype_sq*)malloc(listinitsize*sizeof(lelemtype_sq));
    if(!(*L).elem) exit(OVERFLOW);
    (*L).length=0;
    (*L).listsize=listinitsize;
    return OK;
};
```

清空

```
void clearlist(sqlist *L){
    (*L).length=0;
};
```

此清空只是将长度赋值为 0，我们无法访问到长度以外的数据，但物理存储上的数据还在。

销毁

```
void Destroylist(sqlist *L){
    free((*L).elem);
    (*L).elem=NULL;
    (*L).length=0;
    (*L).listsize=0;
};
```

我们用 free 函数来释放内存，和清空不同，销毁不仅是将线性表数据长度限制，而且将其数据存储存释放（设置为 NULL）。

判空和表长

```
Status listempty(sqlist L){
    return L.length==0 ? TRUE : FALSE;
};
int listlen(sqlist L){
    return L.length;
};
```

很简单不多 bb

根据位次得到数据

建立函数，传入线性表 L，位次 i，指针类型 e （存放表数据）；

首先判断传入值是否合理，这里主要针对 i 值，因为表的长度一般都是大于 0 小于某个数的（线性表有限的数据）。

然后根据位次 i 直接访问对应下标值的数据

```
Status getelem(sqlist L,int i,lelemtype_sq *e){
```

```
    if(i<1 || i>L.length) return ERROR;
    else *e=L.elem[i-1];
    return OK;
};
```

根据数值得到位次

建立函数，传入线性表 L 和数据 e，传入自定义函数 compare，来比较 2 个数值；解释一下，这里的 compare 是用于比较传入 e 和线性表数据的关系，根据这个关系来返回值。

```
int location(sqlist L,lelemtype_sq e,Status(Compare)(lelemtype_sq,lelemtype_sq)){
    //返回顺序表L中首个与e满足Compare(相等或其他)关系的元素位序
    int i=1;
    while (i<=L.length && !Compare(e,L.elem[i-1])) ++i;
    if(i<=L.length) return i;
    else return 0;
};
```

后继

建立函数，传入线性表 L 和数据 cur 和指针 pre（指向后继结点的数据），cur 用来定位；用 pre 来储。

首先找到 cur 对应的下标，根据下标输出后继，注意最后一个结点是没有后继的。

```
Status next(sqlist L,lelemtype_sq cur,lelemtype_sq *next){
    //用next接收cur的后继
    int i=0;
    while(i<L.length && L.elem[i]!=cur){
        ++i;
    }//先遍历线性表找当前值对应下标
    if(i<L.length-1){
        *next=L.elem[i+1];
        return OK;
    }//最后一个结点无后继，所以条件是length减一
    return ERROR;
};
```

前驱

与后继类似，但注意首结点是没有前驱的

```
Status prio(sqlist L,lelemtype_sq cur,lelemtype_sq *pre){
    int i=1;
    if(L.elem[0]!=cur){
        while(i<L.length && L.elem[i]!=cur) ++i;
        if(i<L.length){
            *pre=L.elem[i-1];
            return OK;
        }
    }//因为第一个结点无前驱，所以条件需要判定当前结点是否为第一个结点
    return ERROR;
};
```

增加

传入指向顺序表的指针 L，位次 i，数据元素 e；

为新的结点分配空间；考虑到顺序表的长度，若超出最大长度要重新为顺序表增加空间（用 realloc 数）；之后直接插入数据，位次为 i-1，然后对后续结点进行右移。

```
Status listinsert(sqlist *L,int i,lelemtype_sq e){
    lelemtype_sq *newbase;
    lelemtype_sq *p,*q;
    if(i<1 ||i>(*L).length+1) return ERROR;
    if((*L).length>=(*L).listsize){
        newbase=(lelemtype_sq*)realloc((*L).elem,((*L).listsize+listincrement)*sizeof(lelemtype_s
));
        if(!newbase) exit(OVERFLOW);

        (*L).elem=newbase;
        (*L).listsize+=listincrement;
    }

    q=&(*L).elem[i-1];//将结点插入到位次i的前一格
    for(p=&(*L).elem[(*L).length-1];p>=q;--p) *(p+1)=*p;
    //对插入位置后面的元素进行右移动
    *q=e;//填充数据e
    (*L).length++;
    return OK;
};
```

删除

和增加相同，需要对传入值进行判断，然后改变位次，左移结点。

```
Status listdel(sqlist *L,int i,lelemtype_sq *e){
    lelemtype_sq *p,*q;
    if(i<1 || i>(*L).length) return ERROR;//传入的i值不合法（不在线性表长度范围）

    p=&(*L).elem[i-1];//p为被删除元素的位置
    *e=*p;
    q=(*L).elem+(*L).length-1;

    for(++p;p<=q;++p) *(p-1)=*p;//被删除元素左移
    (*L).length--;
    return OK;
};
```

遍历

```
Status listtra(sqlist L,void(Visit)(lelemtype_sq)){
    int i;
    for(i=0;i<L.length;i++) Visit(L.elem[i]);
    return OK;

};
```

```
void printl(lelemtype_sq e)
{
    printf("%d ", e);
};
```

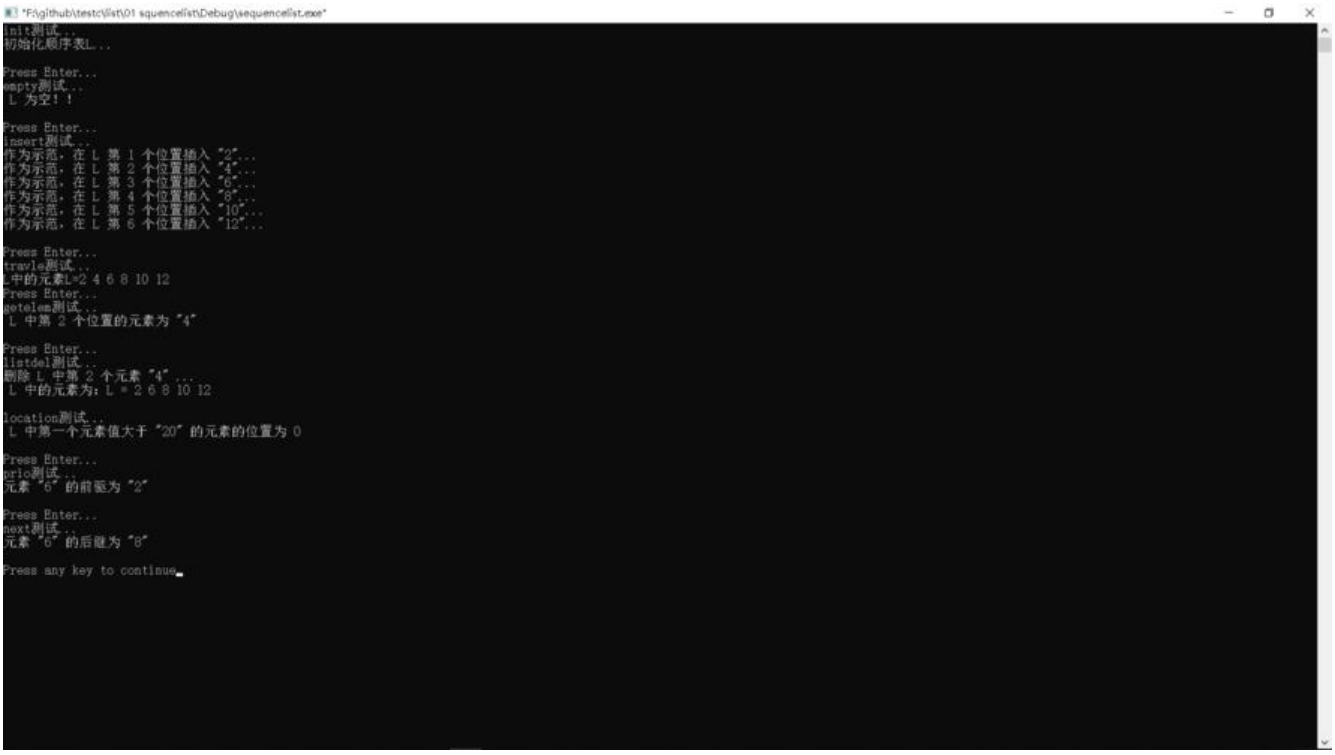
这里的 visit 是一个函数指针，实际传入值是一个自定义函数。比如我这里的就是函数 printl。

顺序表实现测试

在主程序(.c)里我们完成测试

```
#include <stdio.h>
#include "sequencelist.h"

Status CmpGreater(lelemtype_sq e, lelemtype_sq data){
    if(data > e) return TRUE;
    return FALSE;
};
int main(){

    sqlist L;
    int i=0;
    lelemtype_sq e=0;
    printf("init测试...\n");
    {
        printf("初始化顺序表L...\n");
        initlist(&L);
        printf("\n");
    }
    PressEnter;

    printf("empty测试...\n");
    {
        listempty(L) ? printf(" L 为空！！\n") : printf(" L 不为空！\n");
        printf("\n");
    }
    PressEnter;

    printf("insert测试...\n");
    {
        for(i=1; i<=6; i++)
        {
            printf("作为示范，在 L 第 %d 个位置插入 \"%d\"...\n", i, 2*i);
            listinsert(&L, i, 2*i);
        }
        printf("\n");
    }
    PressEnter;

    printf("clearlist测试...\n");
    {
        clearlist(&L);
        e=L.elem[1];
```

```
        printf("%d",e);
    }

    printf("travle测试...\n");
    {
        printf("L中的元素L=");
        listtra(L,printl);
        printf("\n");
    }
    PressEnter;
/*
    printf("getelem测试...\n");
    {
        getelem(L,2,&e);
        printf(" L 中第 2 个位置的元素为 \"%d\" \n", e);
        printf("\n");
    }
    PressEnter;*/
/*
    printf("listdel测试...\n");
    {
        listdel(&L,2,&e);
        printf("删除 L 中第 2 个元素 \"%d\" ...\n", e);
        printf(" L 中的元素为：L = ");
        listtra(L, printl);
        printf("\n\n");
    }
    */
    printf("location测试...\n");
    {
        i = location(L, 20, CmpGreater);
        printf(" L 中第一个元素值大于 \"20\" 的元素的位置为 %d \n", i);
        printf("\n");
    }
    PressEnter;

    printf("prio测试...\n");
    {
        prio(L, 2, &e);
        printf("元素 \"2\" 的前驱为 \"%d\" \n", e);
        printf("\n");
    }
    PressEnter;

    printf("next测试...\n");
    {
        next(L, 2, &e);
        printf("元素 \"2\" 的后继为 \"%d\" \n", e);
        printf("\n");
    }
    return 0;
}
```



后续的什么链表，循环链表等等我们以后补充（大概）

单链表实现与操作

数据结构

```
#include <stdio.h>
#include <stdin.h>
#include "Status.h"
#include "Scanf.h"

typedef int lemletype_l;
typedef struct lnode{
    lelemtype_l data;
    struct lnode* next;
}lnode;
typedef lnode* linklist;//指向单链表结点的指针
```

单链表结点是一个具有数据和指向下一结点的结构，所以其构成的链式的。很容易想像一条链子互相接的场景，相应的，因为每个结点（除头结点）都是有一个指针用于连接，所以链式的物理存储可以一个随机的地方，有着指针的指向我们可以从头开始依次找到每个结点的位置。但也正因如此，其时复杂度一般是链表的长度 n，而顺序表可以随机访问所以其时间复杂度是 1。

注意这里的语法在 typedef struct 这后面是不跟结构名的，但是因为我们需要在数据项用到其本身，于 c 的顺序读取，所以我们需要在这里带上结构名。

初始化

```
Status linklistl(linklist *L){
    (*L)=(linklist)malloc(sizeof(lnode));
```

```
    if(!(*L)) exit(OVERFLOW);
    (*L)->next=NULL;
    return OK;
};
```

传入一个指向链式结点的指针（这里只要传入头结点就相当于传入整个链表）。为其开辟空间存放数，判断是否开辟成功，然后给其后继赋值魏 NULL，不然就是一个野指针。

创建（头插法和尾插法）

```
Status creatlistlh(FILE *fp,linklist *L,int n){//头插法
    int i;
    linklist p;
    lelemtype_l tmp;
    *L=(linklist)malloc(sizeof(lnode));
    if(!(*L)) exit(OVERFLOW);

    (*L)->next=NULL;
    for(i=1;i<n;i++){
        if(Scanf(fp,"%d",&tmp)==1){
            p=(linklist)malloc(sizeof(lnode));
            if(!p) exit(OVERFLOW);
            p->data=tmp;
            p->next=(*L)->next;
            (*L)->next=p;
        }
        else
            return ERROr;
    }
};
```

传入数据文件，链表指针，和链表长度。

初始化变量 i，p，tmp（temp 意思是备份）后，对链表分配空间。

分配成功后对其后继赋值为空（避免变成野指针）。

然后根据长度循环输入数据，扫描文本数据，对其整形数据赞存在 tmp，若数据文件有整型数据，就该数据开辟一块空间用于存放。

开辟成功后，对其数据进行赋值。注意这里如何钩链（一般是先右后左）。

p 是创建的新结点，其后继是头结点的下一个（指向头结点的下一个），然后更新头的后继为 p（头点的下一个为 p），这样 p 就嵌入到链表里面了。

因为每次都是在头结点后面插入，所以当输入顺序为 1，2，3 时，其链表存储为：头-3-2-1。

```
Status creatlistll(FILE *fp,linklist *L,int n){//尾插法
    int i;
    linklist p,q;
    lelemtype_l tmp;
    *L=(linklist)malloc(sizeof(lnode));
```

```
    if(!(*L)) exti(OVERFLOW);
    (*L)->next=NULL;
    for(i=1;q=*L;i<n;++i){
        if(Scanf(fp,"%d",&tmp)==1){
            p=(linklist)malloc(sizeof(lnode));
            if(!P) exit(VOERFLOW);
            p->data=tmp;
            q-next=p;
            q=q->next;
            p->next=NULL;
        }
        else
            return ERROR;
    }
};
```

和头插法大体类似，但是钩链的实现不一样，而且需要一个备用变量（q）存储数据（链表头结点）。

创建新结点后对结点赋值，q 的后继是 p（q 存储了头结点信息，头结点后面跟 p）,然后更新 q 为当的 p 所指向的结点（所以每次插入都是在上一个结点的后面）。

之所以使用 q 这个变量，是因为指向头结点的指针是不能改变的，其改变之后，链表的信息也改变了。

清空与销毁

```
Status clearlistl(linklist L){
    linklist next,p;
    if(!L) return ERROR;

    next=L->next;
    while(next){
        p=pre->next;
        free(p);
        pre=p;··
    }
    L->next=NULL;
    return OK;
};
void destroylistl(linklist *L)
{
    linklist p = *L;

    while(p)
    {
        p = (*L)->next;
        free(*L);
        (*L) = p;
    }
}
```

遍历链表，若链表（头指针）存在下一个结点，那么更新 p 然后释放这个结点，最后给头指针的后继值为空。注意两者的区别。

判空与链长

```
Status listemptyl(linklist L){
    if(L!=NULL && L->next==NULL)
        return TRUE;
    else
        return FALSE;
};

int listlenl(linklist L){
    linklist p;
    int i;

    if(L)
    {
        i = 0;
        p = L->next;
        while(p)
        {
            i++;
            p = p->next;
        }
    }

    return i;
};
```

略

得到结点数据

```
Status geteleml(linklist L,int i,lelemtype_l *e){
    int j;
    linklist p=L->next;
    j=1;
    p=L->next;
    while(p&&j<i){
        j++;
        p=p->next;
    }
    if(!p||j>i) return ERROR;
    *e=p->data;
    return OK;
};
```

设置变量 p 用于存放头结点的下一个（也就是第一个结点）。

当 p 存在且 j（计数器）是小于传入值的为此（第几个结点），我们就循环更新 p 结点指导目标结点此时我们要做一次判断（判断结点是否存在和 j 与 i 的值是否相等），最后我们输出 对应的数据就行。

根据数值找结点位次

```
int locateeleml(linklist L,lelemtype_l e,Status(compare)(lelemtype_l,lelemtype_l)){
    int i;
    linklist p;
```

```
    i=-1;//表示一种链表不存在的状态，是没有对应量的
    if(L){
        i=0;
        p=L->next;
        while(p){
            i++;
            if(!Compare(e,p->data)){
                p=p->next;
                if(p==NULL) i=0;
            }
            else break;
        }
    }
    return i;
};
```

设置变量 p 为第一个结点，让后遍历链表对比数据，若直到最后一个也没有相等的值，则返回 0。

前一个结点和后一个结点

```
Status proeleml(linklist L,lelemtype_l cur,lelemtype_l *pre){
    linklist p,suc;
    if(L){
        p=L->next;
        if(p->data!=cur){
            while(p->next){
                suc=p->next;
                if(suc->data==cur)
                {
                    *pre=p->data;
                    return OK;
                }
                p=suc
            }
        }
    }
    return ERROR;
};

Status nexteleml(linklist L,lelemtype_l cur,lelemtype_l *next){
    linklist p,suc;
    if(L){
        p=L->next;
        while(p&&p->next){
            suc=p->next;
            if(suc&&p->data==cur){
                *next=suc->data;
                return OK;
            }
            if(suc) p=suc;
            else break;
        }
    }
    return ERROR;
```

```
};
```

和根据数值定位结点类似，但是这次比较的不是当前结点的值，而是前一个结点的值或者后一个结点值。就是这个思想，细节不多说了。

插入和删除

```
Status listinsertl(linklist L,int i,lelemtype_l e){
    linklist p,s;
    int j;
    p=L;
    j=0;
    while(p&&j<i-1){
        p=p->next;
        ++j;
    }
    if(!p||j>i-1) return ERROR;
    s=(linklist)malloc(sizeof(lnode));
    if(!s) exit(OVERFLOW);

    s->data=e;
    s->next=p->next;
    p->next=s;
    return OK;

};

Status listdell(linklist L,int i,lelemtype_l *e){
    linklist pre,p;
    int j;
    pre = L;
    j = 1;

    while(pre->next && j<i)          //寻找第i个结点，并令pre指向其前驱
    {
        pre = pre->next;
        ++j;
    }

    if(!pre->next || j>i)          //删除位置不合理
        return ERROR;

    p = pre->next;
    pre->next = p->next;
    *e = p->data;
    free(p);

    return OK;

};
```

插入一个结点，需要新建一个结点和开辟新空间；对应的删除则需要释放空间。

传入值确定了新建结点（s）的位次，所以我们只需要进行该位次的遍历（使指针指向对应位次的结

（p）），然后勾链，s 的后继指向 p 的后继（p 的下一个结点），然后 p 的后继更新为 s（也就是由 p-q 变为 p-s-q）。

删除同理。

遍历

```
Status listtral(linklist L,void(Visit)(lelemtype_l)){
    linklist p;
    if(!L)
        return ERROR;
    else
        p = L->next;

    while(p)
    {
        Visit(p->data);
        p = p->next;
    }

    return OK;
};
```

略

测试

```
#include <stdio.h>
#include "singlylinklist.h"                    //**▲02 线性表**//

/* 函数原型 */
Status CmpGreater(lelemtype_l e, lelemtype_l data);    //判断data是否大于e //若data大于e，返TRUE
void PrintElem(lelemtype_l e);                    //测试函数，打印整型

int main(int argc, char **argv)
{
    linklist L;
    int i;
    lelemtype_l e;

    printf("InitList_L 测试...\n");            //1.函数InitList_L测试
    {
        printf("初始化单链表 L ...\n");
        linklistl(&L);
        printf("\n");
    }
    PressEnter;

    printf("ListEmpty_L 测试...\n");        //4.函数ListEmpty_L测试
    {
        listemptyl(L) ? printf(" L 为空！！\n") : printf(" L 不为空！\n");
        printf("\n");
```

```
}
PressEnter;

printf("ListInsert_L 测试...\n");        //10.函数ListInsert_L测试
{
    for(i=1; i<=6; i++)
    {
        printf("在 L 第 %d 个位置插入 \"%d\" ...\n", i, 2*i);
        listinsertl(L, i, 2*i);
    }
    printf("\n");
}
PressEnter;

printf("ListTraverse_L 测试...\n");    //12.函数ListTraverse_L测试
{
    printf(" L 中的元素为：L = ");
    listtral(L, PrintElem);
    printf("\n\n");
}
PressEnter;

printf("ListLength_L 测试...\n");        //5.函数ListLength_L测试
{
    printf(" L 的长度为 %d \n", listlenl(L));
    printf("\n");
}
PressEnter;

printf("ListDelete_L 测试...\n");        //11.函数ListDelete_L测试
{
    listdell(L, 6, &e);
    printf("删除 L 中第 6 个元素 \"%d\" ...\n", e);
    printf(" L 中的元素为：L = ");
    listtral(L, PrintElem);
    printf("\n\n");
}
PressEnter;

printf("GetElem_L 测试...\n");            //6.函数GetElem_L测试
{
    geteleml(L, 4, &e);
    printf(" L 中第 4 个位置的元素为 \"%d\" \n", e);
    printf("\n");
}
PressEnter;

printf("LocateElem_L 测试...\n");        //7.函数LocateElem_L测试
{
    i = locateeleml(L, 13, CmpGreater);
    printf(" L 中第一个元素值大于 \"7\" 的元素的位置为 %d \n", i);
    printf("\n");
}
PressEnter;
```

```
    printf("PriorElem_L 测试...\n");        //8.函数PriorElem_L测试
    {
        proeleml(L, 6, &e);
        printf("元素 \"6\" 的前驱为 \"%d\" \n", e);
        printf("\n");
    }
    PressEnter;

    printf("NextElem_L 测试...\n");            //9.函数NextElem_L测试
    {
        nexteleml(L, 6, &e);
        printf("元素 \"6\" 的后继为 \"%d\" \n", e);
        printf("\n");
    }
    PressEnter;
    /*
    printf("▼2\n▲函数 ClearList_L 测试...\n");        //2.函数ClearList_L测试
    {
        printf("清空 L 前：");
        listemptyl(L) ? printf(" L 为空！！\n") : printf(" L 不为空！\n");
        clearlistl(L);
        printf("清空 L 后：");
        listemptyl(L) ? printf(" L 为空！！\n") : printf(" L 不为空！\n");
        printf("\n");
    }
    PressEnter;*/
/*
    printf("▼3\n▲函数 DestroyList_L 测试...\n");        //3.函数DestroyList_L测试
    {
        printf("销毁 L 前：");
        L ? printf(" L 存在！\n") : printf(" L 不存在！！\n");
        destorylistl(&L);
        printf("销毁 L 后：");
        L ? printf(" L 存在！\n") : printf(" L 不存在！！\n");
        printf("\n");
    }
    PressEnter;*/

    printf("CreateList_HL 测试...\n");        //13.函数CreateList_HL测试
    {
        FILE *fp;
        linklist l;
        printf("头插法建立单链表 L = ");
        fp = fopen("TestData_HL.txt", "r");                //文件指针，指向数据源
        creatlistlh(fp, &l, 5);
        fclose(fp);
        listtral(l, PrintElem);
        printf("\n\n");
    }
    PressEnter;

    printf("CreateList_TL 测试...\n");        //14.函数CreateList_TL测试
    {
```

```
        FILE *fp;
        linklist l;
        printf("尾插法建立单链表 L = ");
        fp = fopen("TestData_TL.txt", "r");              //文件指针，指向数据源
        creatlistll(fp, &l, 5);
        fclose(fp);
        listtral(l, PrintElem);
        printf("\n\n");
    }
    PressEnter;

    return 0;
}

Status CmpGreater(lelemtype_l e, lelemtype_l data)
{
    return data>e ? TRUE : FALSE;
}

void PrintElem(lelemtype_l e)
{
    printf("%d ", e);
}
```

小结

单链表是基础，是理解更复杂数据结构的基石。为什么设计出链表，因为它可以解决顺序表无法解决拓展性的问题。单链表的元素随机分布在存储空间中，是一种“非随机存储”的存取存储结构。我们要搞清楚头插尾插的区别以及各个功能函数的实现（主要理解结点是如何相连的，画图）。

循环链表实现与操作

数据结构

```
/* 双循环链表类型定义 */
typedef int LElemType_DC;
typedef struct DuLNode
{
    LElemType_DC data;
    struct DuLNode *prior;
    struct DuLNode *next;
}DuLNode;
typedef DuLNode* DuLinkList;            //指向双循环链表结构的指针
```

对于循环链来说，其数据项有指向上一个和下一个的指针。所以其首位是相连的，类似与一个圆一样。

初始化

```
Status InitList_DuL(DuLinkList *L)
{
```

```
    *L = (DuLinkList)malloc(sizeof(DuLNode));
    if(!(*L))
        exit(OVERFLOW);

    (*L)->next = (*L)->prior = *L;

    return OK;
}
```

为链表的第一个结点分配空间，然后将 next 和 pro 都指向自己（该结点）。

清空与销毁

```
Status ClearList_DuL(DuLinkList L)
{
    DuLinkList p, q;

    p = L->next;

    while(p!=L)
    {
        q = p->next;
        free(p);
        p = q;
    }

    L->next = L->prior = L;

    return OK;
}

void DestroyList_DuL(DuLinkList *L)
{
    ClearList_DuL(*L);

    free(*L);

    *L = NULL;
}
```

链表的清空与销毁都需要 free 释放空间。但是清空不需要将头结点释放，而销毁是在清空后再对链的头结点空间释放（实际上也就是销毁链表）

判空和链长度

```
Status ListEmpty_DuL(DuLinkList L)
{
    if(L && L->next==L && L->prior==L)
        return TRUE;
    else
        return FALSE;
}
```

```
int ListLength_DuL(DuLinkList L)
{
    DuLinkList p;
    int count;

    if(L)
    {
        count = 0;
        p = L;                  //p指向头结点

        while(p->next!=L)       //p没到表头
        {
            count++;
            p = p->next;
        }
    }

    return count;
}
```

对于循环链表来说，链表是否为世界上就是判断链表的头结点是否与自己相连（也就是头结点是否有继或者前驱）；

而输出循环的长度，不仅需要遍历链表，而且要判断最后遍历到的结点是不是头结点（从头结点开始也从头结点结束）。

得到数据元素

```
Status GetElem_DuL(DuLinkList L, int i, LElemType_DC *e)
{
    DuLinkList p;
    int count;

    if(L)
    {
        count = 1;
        p = L->next;

        while(p!=L && count<i)
        {
            count++;
            p = p->next;
        }

        if(p!=L)
        {
            *e = p->data;
            return OK;
        }
    }

    return ERROR;
}
```

传入值是链表 L，位次 i，数据指针 e（用于存放链表取到的数据）。

在函数内声明了结点 p 用于遍历，判断链表是否存在后，对 p 进行更新然后循环访问下一个结点（要满足相应的条件），根据遍历到对应结点后，判断当前结点是否是头结点然后输出数据。

根据数据得到位次

```
int LocateElem_DuL(DuLinkList L, LElemType_DC e, Status(Compare)(LElemType_DC, LElemTy
e_DC))
{
    DuLinkList p;
    int count;

    if(L)
    {
        count = 1;
        p = L->next;

        while(p!=L && !Compare(e, p->data))
        {
            count++;
            p = p->next;
        }

        if(p!=L)
            return count;
    }

    return 0;
}
```

输出链表 L，数据 e，和函数指针。

判断链表存在后，对链表进行遍历；根据数据使用传入的函数指针进行比较，直到找到那个结点，判是否是头结点后输出 count。

前驱和后继

```
Status PriorElem_DuL(DuLinkList L, LElemType_DC cur_e, LElemType_DC *pre_e)
{
    DuLinkList p;

    if(L)
    {
        p = L->next;

        while(p!=L && p->data!=cur_e)
            p = p->next;

        if(p!=L && p->prior!=L)              //p不为首结点
        {
            *pre_e = p->prior->data;
```

```
            return OK;
        }
    }

    return ERROR;
}

Status NextElem_DuL(DuLinkList L, LElemType_DC cur_e, LElemType_DC *next_e)
{
    DuLinkList p;

    if(L)
    {
        p = L->next;

        while(p!=L && p->data!=cur_e)
            p = p->next;

        if(p!=L && p->next!=L)
        {
            *next_e = p->next->data;
            return OK;
        }
    }


    return ERROR;
}
```

传入链表 L 和当前结点的数据以及数据指针（存放上一个或下一个结点的数据）。

判断链表存在后进行遍历，判断当前结点的下一个结点的数据是否等于 cur，不是则更新；在找到后断该结点是否是头结点。后继也一样。

得到对应位次的指针 p

```
DuLinkList GetElemPtr_DuL(DuLinkList L, int i)
{
    int count;
    DuLinkList p;

    if(L && i>0)
    {
        count = 1;
        p = L->next;

        while(p!=L && count<i)
        {
            count++;
            p = p->next;
        }

        if(p!=L)
```

```
        return p;
    }

    return NULL;
}
```

传入 L 和 i;

在判断 L 存在和数据 i 的合理性后，对 p 进行更新然后根据条件判断整个链表的数据。最后若 p 不头结点，则返回 p 不是返回 NULL;

插入和删除

```
Status ListInsert_DuL(DuLinkList L, int i, LElemType_DC e)
{
    DuLinkList p, s;

    if(i<1 || i>ListLength_DuL(L)+1)    //先对i做出限制
        return ERROR;

    p = GetElemPtr_DuL(L, i);           //确定第i个结点指针
    if(!p)                              //此处若p=NULL，说明i = ListLength_DuL(L)+1
        p = L;                          //令p指向头指针

    s = (DuLinkList)malloc(sizeof(DuLNode));
    if(!s)
        exit(OVERFLOW);
    s->data = e;

    s->prior = p->prior;
    p->prior->next = s;
    s->next = p;
    p->prior = s;

    return OK;
}

/*═══════╗
║ 算法2.19 ║
╚═══════*/
Status ListDelete_DuL(DuLinkList L, int i, LElemType_DC *e)
{
    DuLinkList p;

    if(!(p=GetElemPtr_DuL(L, i)))       //i值不合法
        return ERROR;

    *e = p->data;
    p->prior->next = p->next;
    p->next->prior = p->prior;

    free(p);
    p = NULL;
```

```
    return OK;
}
```

传入值是 L，位次 i 和要插入的数据。

在对输入进行合理化判定（长度）后，我们直接用 getelem 函数根据 i 找到对应结点 m。然后分配空间在 m，注意勾连的顺序。

删除也类似。

遍历

```
void ListTraverse_DuL(DuLinkList L, void(Visit)(LElemType_DC))
{
    DuLinkList p;

    p = L->next;                    //p指向头结点，正向访问链表

    while(p!=L)
    {
        Visit(p->data);
        p = p->next;
    }
}
```

略

测试

略

静态链表实现与操作

静态链表主要注意与顺序表有什么不同，其功能实现的思想和链表相似，这里我们就不过多赘述。

数据结构

```
#define MAXSIZE 1000                    //静态链表的最大长度

/* 静态链表类型定义 */
typedef int SLinkList;                  //静态链表类型
typedef int LElemType_S;
typedef struct
{
    LElemType_S data;
    int cur;            //cur是游标，做指针用，区别于数组下标
}Component[MAXSIZE];                    //链表空间类型

/* 全局变量 */
Component SPACE;
```

静态链表是带有下表的数组。对于静态链表来说是无法给链表添加元素的。在编码上我们用了一个备

空间，这个空间用于链表结点的空间申请，也就是说链表结点的数据是在这里存储，每次申请一个都要在这个空间和删除表示该空间已经被使用。

初始化

```
void InitSpace_SL()                //首先初始化备用空间
{
   int i;              //0号单元做备用空间的起始结点

   for(i=0; i<MAXSIZE-1; i++)          //各空间结点首尾相接
      SPACE[i].cur = i + 1;

   SPACE[MAXSIZE-1].cur = 0;
}
```

对该数组（备份空间）进行初始化，每个结点的数据填充为 当前下标加一，最后一个位置是链表的长。

申请空间

```
int Malloc_SL()                    //从备用空间申请结点空间
{
   int i;

   i = SPACE[0].cur;

   if(SPACE[0].cur)
   {
      SPACE[0].cur = SPACE[i].cur;
//将申请到的空间从备用空间中删去
      return i;              //返回新申请结点下标
   }
   else
      return 0;              //申请失败返回0
}
```

申请的空间从第一个开始申请，返回申请的下标。

i 值是备用空间的第一个的下标（1），当此下标存，在那么 数组的 0 号空间下标更新为 i 值对应的次的下表值（这一步是将申请到的空间从备用空间删除，那么你下次申请就无法申请到 1 号下标对应数组空间了），返回 i 值。

分配也都是从 0 号分配，所以每次分配后 0 号空间的下标就要更新为 1 号空间的下标。这样每次分都会是分配一个新空间不会冲突。

回收 k 结点空间

```
void Free_SL(int k)                    //回收k结点空间
{
   SPACE[k].cur = SPACE[0].cur;
   SPACE[0].cur = k;
}
```

传入值是 整型 k，代表结点 k 所占据的空间，结点 k 所占据的数组空间下标跟新为当前备用空间数的第一个空间对应的下标（这里意味着该结点 k 对应的下标值是备用空间的第一个，而第一个通常是使用中的，所以 k 的空间是已被占用的），然后第一个下标值跟新为 k。（将备用空间数组的一个更为一个空闲状态，这个下标 k 是没有被使用的）

换个理解就是这个函数是释放结点空间的，依据的是结点对应的下标值，所以该下标就会回到备用空内。这里将结点下标更新为 0 号也就是头结点的下标，也意味着该结点被删除了。

比如我链表的结点 5（假设下标为 6）删除，则该下标值更新为当前 0 号空间的下标值（加上为 7）这个时候 space[0].cur=5，space[5].cur=7；在备用空间里，现在 5 号空间就又可以被使用了。

初始化静态链表

```
Status InitList_SL(SLinkList *H)       //H为头结点指针
{
   *H = Malloc_SL();                  //创建头结点
   if(!(*H))
      exit(OVERFLOW);                  // “内存”已满

   SPACE[*H].cur = 0;                  //头结点游标置为0

   return OK;
}
```

这里我们设置一个头结点方便操作，我们初始化链表就是生成一个头指针。

清空和销毁

```
Status ClearList_SL(SLinkList H)
{
   int p, q;

   if(!H)
      return ERROR;

   p = SPACE[H].cur;           //p指向第一个结点

   while(p)           //从链表首结点开始删除
   {
      SPACE[H].cur = SPACE[p].cur;
      Free_SL(p);
      p = SPACE[H].cur;
   }

   return OK;
}

void DestroyList_SL(SLinkList *H)
{
   ClearList_SL(*H);           //清空静态链表

   Free_SL(*H);               //释放头结点
```

```
    *H = 0;
}
```

注意清空和销毁的区别，对于销毁来说，它将链表的头指针也进行 free，意味着存放头指针的空间没了，也自然没有这个数组；而清空只是将头指针的下标，也就是头指针的下一个结点进行清空。

判空和链长

```
Status ListEmpty_SL(SLinkList H)
{
    if(H && !SPACE[H].cur)                //只有头结点
        return TRUE;
    else
        return FALSE;
}

int ListLength_SL(SLinkList H)
{
    int count;                            //计数器
    int p;

    if(!H)
        exit(OVERFLOW);

    count = 0;
    p = SPACE[H].cur;

    while(p)
    {
        count++;
        p = SPACE[p].cur;
    }

    return count;
}
```

略

根据位次得到元素

```
Status GetElem_SL(SLinkList H, int i, LElemType_S *e)
{
    int count, p;

    if(!H || i<1 || i>MAXSIZE-2)
        return ERROR;

    count = 0;
    p = SPACE[H].cur;

    while(p)
    {
        count++;
```

```
        if(count==i)
        {
            *e = SPACE[p].data;
            return OK;
        }

        p = SPACE[p].cur;
    }
}
```

略

根据数据元素得到位次

```
int LocateElem_SL(SLinkList H, LElemType_S e)
{
    int k, count;

    count = 1;
    if(H && SPACE[H].cur)
    {
        k = SPACE[H].cur;

        while(k && SPACE[k].data!=e)
        {
            count++;
            k = SPACE[k].cur;
        }

        if(k)
            return count;
    }

    return 0;
}
```

略

前驱和后继

```
Status NextElem_SL(SLinkList H, LElemType_S cur_e, LElemType_S *next_e)
{
    int p;

    if(H)
    {
        p = SPACE[H].cur;

        while(p && SPACE[p].data!=cur_e)
            p = SPACE[p].cur;

        if(p && SPACE[p].cur)    //找到了cur_e且不是最后一个结点
```

```
        {
            p = SPACE[p].cur;
            *next_e = SPACE[p].data;
            return OK;
        }
    }

    return ERROR;
}

Status ListInsert_SL(SLinkList H, int i, LElemType_S e)
{
    int count, k, p;

    if(!H)                //链表不存在
        return ERROR;

    if(i>0)
    {
        count = 0;
        k = H;            //k指向头结点

        while(k && count<i-1)    //寻找插入位置的前一个位置
        {
            count++;
            k = SPACE[k].cur;
        }

        if(k)        //找到了第i-1个元素的位置
        {
            p = Malloc_SL();
            if(!p)            //申请空间失败
                return ERROR;

            SPACE[p].data = e;    //插入元素e
            SPACE[p].cur = SPACE[k].cur;
            SPACE[k].cur = p;

            return OK;
        }
    }

    return ERROR;
}
```

删除结点

```
Status ListDelete_SL(SLinkList H, int i, LElemType_S *e)
{
    int count, k, p;

    if(!H)                    //链表不存在
        return ERROR;
```

```
    if(i>0)
    {
        count = 0;
        k = H;              //k指向头结点

        while(k && count<i-1)       //寻找插入位置的前一个位置
        {
            count++;
            k = SPACE[k].cur;
        }

        if(k && SPACE[k].cur)
//找到了第i-1个元素的位置且不是最后一个元素
        {
            p = SPACE[k].cur;//p指向要被删除的结点
            *e = SPACE[p].data;
            SPACE[k].cur = SPACE[p].cur;
            Free_SL(p);

            return OK;
        }
    }

    return ERROR;
}
```

遍历结点

```
Status ListTraverse_SL(SLinkList H, void(Visit)(LElemType_S))
{
    int p;

    if(!H)
        return ERROR;                   //静态链表不存在或为空

    p = SPACE[H].cur;

    while(p)
    {
        Visit(SPACE[p].data);
        p = SPACE[p].cur;
    }

    return OK;
}
```

扩展的单链表

```
typedef struct ELNode                   //结点类型
{
    LElemType_E data;
    struct ELNode *next;
```

```
}ELNode;
typedef ELNode* Link;              //指向结构的指针
typedef ELNode* PositionPtr;
typedef struct                     //链表类型
{
    Link head, tail;        //分别指向线性链表中的头结点和尾结点
    int len;            //指示线性链表中数据元素的个数
}ELinkList;
```

这里我们只给出数据结构，扩展的单链表很特殊，因为其链表不仅有头结点，还有尾结点

思考

1. 顺序存储结构的优点是什么?
2. 线性表的顺序存储结构是随机的还是顺序的?

ps: 摸一摸数据结构(目录)