

# 第三章：线性表

---

## 第一部分：线性表的顺序存储结构

---

### 1.有关线性表顺序存储结构的操作

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

//线性表顺序存储
#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 //存储空间初始分配量

typedef int Status; //Status是函数的类型，它的值是函数结果状态代码，比如ok
typedef int ElemType; //ElemType类型自己根据需求定义，这里是int

typedef struct{
    ElemType data[MAXSIZE]; //数组，存储数据元素
    int length; //线性表当前的长度
} SqList;

//打印函数
Status visit(ElemType c)
{
    printf("%d ", c);
```

```

        return OK;
    }

//初始化顺序线性表
Status InitList(SqList *L){
    L->length = 0;
    return OK;
}

//判断空表
/* 初始条件：顺序线性表L已经存在
   操作结果：若L是空表，返回TRUE；否则返回FALSE。
*/
Status ListEmpty(SqList L){
    if(L.length==0){
        return TRUE;
    }
    else{
        return FALSE;
    }
}

/* 初始条件：顺序线性表L已存在。操作结果：将L重置为空表 */
Status ClearList(SqList *L)
{
    L->length=0;
    return OK;
}

//SqList *L 和 SqList L 的区别是：*L表示要修改内容， L表示
只是访问，不修改内容

//返回元素个数
//初始条件：顺序线性表L已经存在，操作结果：返回L中元素的个数
int ListLength(SqList L){
    return L.length;
}

```

```
}
```

//返回元素的值

//初始条件：L存在

//操作结果：用e返回L中第i个元素的值，！！i指的是位置，第一个位置上的数组是从0开始的

```
Status GetElem(SqList L,int i,ElemType *e){
    if(L.length==0 || i<1 || i>L.length){
        return ERROR;
    }
    *e = L.data[i - 1];

    return OK;
}
```

//返回元素的位置

//操作结果：返回L中某个元素e的位置

```
int LocateElem(SqList L,ElemType e){
    int i;
    if(L.length==0){
        return 0;
    }
    if(i>=L.length){
        return 0;
    }
    for (i = 0; i < L.length;i++){
        if(L.data[i]==e){
            break;
        }
    }
    return i + 1;
}
```

//插入数据

//操作结果：在L中第i个位置之前插入新的数据元素e；L的长度增加1

```
Status ListInsert(SqList *L, int i,ElemType e){
```

```

int j;
if(L->length==MAXSIZE){
    return ERROR;
}
if(i<1 || i>L->length+1){
    return ERROR;
}
if(i<=L->length){
    for (j = L->length - 1; j>=i-1;j--){ //将要插入位置之后的元素向后移动一位
        L->data[j + 1] = L->data[j];
    }
}
L->data[i - 1] = e; //将新元素插入
L->length++;

return OK;
}

```

//删除元素

//操作结果：删除L的第i个元素，并用e返回它的值，L的长度减1

```

Status ListDelete(SqList *L ,int i,ElemType *e){
    int j;
    if(L->length==0){
        return ERROR;
    }
    if(i<1 || i>L->length){
        return ERROR;
    }
    *e = L->data[i - 1];
    if(i<L->length){
        for (j = i; j < L->length;j++){
            L->data[j - 1] = L->data[j];
        }
    }
    L->length--;
}

```

```

        return OK;
    }

//输出线性表
Status ListTraverse(SqList L){
    int i;
    for (i = 0; i < L.length;i++){
        visit(L.data[i]);
    }
    printf("\n");
    return OK;
}

//合并
//实现两个线性表集合A和B的并及操作
//也就是把存在集合B中但是并不存在A中的数据元素 插入到A中就可以了。

void unionL(SqList *La,SqList Lb){
    int La_len, Lb_len, i;
    ElemType e; //声明La和Lb是相同的数据元素
    La_len = ListLength(*La); //求线性表的长度
    Lb_len = ListLength(Lb);
    for (i = 1; i <= Lb_len;i++){
        GetElem(Lb, i, &e); //取Lb中第i个元素赋给e
        if(!LocateElem(*La,e)){ //La中不存在和e相同的数据元素
            ListInsert(La, ++La_len, e); //插入
        }
    }
}

int main(){
    SqList L;
    SqList Lb;

    ElemType e;
    Status i;

```

```
int j, k;
i = InitList(&L);
printf("初始化L后: L.length=%d\n", L.length);
for (j = 1; j <= 5; j++){
    i = ListInsert(&L, 1, j);
}
printf("在L的表头依次插入1-5后: L.data=");
ListTraverse(L);

printf("L.length=%d \n", L.length);
i = ListEmpty(L);
printf("L是否空: i=%d(1:是 0:否)\n", i);

i = ClearList(&L);
printf("清空L后: L.length=%d\n", L.length);
i = ListEmpty(L);
printf("L是否空: i=%d(1:是 0:否)\n", i);

for(j=1; j<=10; j++)
    ListInsert(&L, j, j);
printf("在L的表尾依次插入1~10后: L.data=");
ListTraverse(L);

printf("L.length=%d \n", L.length);

ListInsert(&L, 1, 0);
printf("在L的表头插入0后: L.data=");
ListTraverse(L);
printf("L.length=%d \n", L.length);

GetElem(L, 5, &e);
printf("第5个元素的值为: %d\n", e);
for(j=3; j<=4; j++)
{
    k = LocateElem(L, j);
    if(k)
```

```

        printf("第%d个元素的值为%d\n", k, j);
    else
        printf("没有值为%d的元素\n", j);
}

k=ListLength(L); /* k为表长 */
for(j=k+1; j>=k; j--)
{
    i=ListDelete(&L, j, &e); /* 删除第j个数据 */
    if(i==ERROR)
        printf("删除第%d个数据失败\n", j);
    else
        printf("删除第%d个的元素值为：
%d\n", j, e);
}
printf("依次输出L的元素：");
ListTraverse(L);

j=5;
ListDelete(&L, j, &e); /* 删除第5个数据 */
printf("删除第%d个的元素值为： %d\n", j, e);

printf("依次输出L的元素：");
ListTraverse(L);

//构造一个有10个数的Lb
i=InitList(&Lb);
for(j=6; j<=15; j++)
    i=ListInsert(&Lb, 1, j);

unionL(&L, Lb);

printf("依次输出合并了Lb的L的元素：");
ListTraverse(L);

```

```
    return 0;  
}
```

## 2.线性表顺序存储结构的优点和缺点

优点：

- 无须为表示表中元素之间的逻辑关系而增加额外的存储空间；
- 可以快速的存取表中任一位置的元素。

缺点：

- 插入和删除操作需要移动大量元素；
- 当线性表长度变化较大的时候，难以确定存储空间的容量；
- 造成存储空间的“碎片”

## 第二部分：线性表的链式存储结构

### 1.头指针与头结点的异同

头指针：

- 头指针是指 链表指向第一个结点的指针，若链表有头结点，则是指向头结点的指针；
- 头指针具有标识作用，所以常用头指针冠以链表的名字；
- 无论链表是否为空，头指针均不为空，头指针是链表的必要元素。

头结点：

- 头结点是为了操作的统一和方便而设立的，放在第一元素的结点之前，其数据域一般无意义（也可以放链表的长度）；



- 有了头结点，对在第一元素结点前插入结点和删除第一结点，其操作与其它结点的操作就统一了；
- 头结点不一定是链表必须要素。

## 2.有关线性表链式存储结构的操作

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

//线性表的链式存储结构（单链表）

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 20 //空间初始存储分配量

typedef int Status; //Status是函数的类型，其值是函数结果
状态代码，比如OK
typedef int ElemType; //ElemType类型根据实际情况而定，这
里一般假设为int

//单链表存储结构
typedef struct Node{
    ElemType data;
    struct Node *next;
} Node;
typedef struct Node *LinkList; //定义LinkList

//打印
Status visit(ElemType c){
    printf("%d ", c);
    return OK;
```

```

}

//初始化顺序线性表
Status InitList(LinkList *L){
    *L = (LinkList)malloc(sizeof(Node)); //产生头结
    点, 并且使L指向此头结点
    if(!(*L)){ //存储分配失败
        return ERROR;
    }
    (*L)->next = NULL;

    return OK;
}

/* 初始条件: 顺序线性表L已存在。
操作结果: 若L为空表, 则返回TRUE, 否则返回FALSE */
Status ListEmpty(LinkList L){
    if(L->next){
        return FALSE;
    }else{
        return ERROR;
    }
}

/* 初始条件: 顺序线性表L已存在。
操作结果: 将L重置为空表 */
Status ClearList(LinkList *L){
    LinkList p, q;
    p = (*L)->next; //p指向第一个结点
    while(p){ //没有到表尾
        q = p->next;
        free(p);
        p = q;
    }
    (*L)->next = NULL; //头结点指针域为空
    return OK;
}

```

```
}
```

/\* 初始条件：顺序线性表L已存在。

操作结果：返回L中数据元素个数 \*/

```
int ListLength(LinkList L){  
    int i = 0;  
    LinkList p = L->next; //p指向第一个结点  
    while(p){  
        i++;  
        p = p->next;  
    }  
    return i;  
}
```

/\* 初始条件：顺序线性表L已存在， $1 \leq i \leq \text{ListLength}(L)$  \*/

/\* 操作结果：用e返回L中第i个数据元素的值 \*/

```
Status GetElem(LinkList L, int i, ElemType *e){  
    int j;  
    LinkList p; //声明一个结点p;  
    p = L->next; //让p指向链表L的第一个结点  
    j = 1; //j为计数器  
    while(p && j < i){ //p不为空或者计数器j还没有等于i的时  
候，循环继续  
        p = p->next; //让p指向下一个结点  
        ++j;  
    }  
    if(!p || j > i){  
        return ERROR; //第i个元素不存在  
    }  
    *e = p->data; //取第i个元素的数据  
  
    return OK;  
}
```

/\* 初始条件：顺序线性表L已存在 \*/

/\* 操作结果：返回L中第1个与e满足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 \*/

```
int LocateElem(LinkList L, ElemType e){
    int i = 0;
    LinkList p = L->next;
    while(p){
        i++;
        if(p->data==e){ //找到这样的元素
            return i;
        }
        p = p->next;
    }

    return 0;
}
```

/\* 初始条件：顺序线性表L已存在,  $1 \leq i \leq \text{ListLength}(L)$ , \*/

/\* 操作结果：在L中第i个位置之前插入新的数据元素e，L的长度加1 \*/

```
Status ListInsert(LinkList *L, int i, ElemType e){
    int j;
    LinkList p, s;
    p = *L;
    j = 1;
    while(p && j<i){ //寻找第i个结点
        p = p->next;
        ++j;
    }

    if(!p || j>i){
        return ERROR; //第i个元素不存在
    }

    s = (LinkList)malloc(sizeof(Node)); //生成新的结点
    (c语言的标准函数)
    s->data = e;
    s->next = p->next; //将p的后继结点赋值给s的后继
    p->next = s; //将s赋值给p的后继
}
```

```

        return OK;
    }

/* 初始条件：顺序线性表L已存在， $1 \leq i \leq \text{ListLength}(L)$  */
/* 操作结果：删除L的第i个数据元素，并用e返回其值，L的长度减1 */
Status ListDelete(LinkList *L, int i, ElemType *e){
    int j;
    LinkList p, q;
    p = *L;
    j = 1;
    while(p->next && j<i){    //遍历寻找第i个元素
        p = p->next;
        ++j;
    }
    if(!(p->next) || j>i){
        return ERROR;    //第i个元素不存在
    }

    q = q->next;
    p->next = q->next;    //将q的后继赋值给p的后继
    *e = q->data;    //将q结点中的数据给e
    free(q);    //让系统回收此结点，释放内存

    return OK;
}

/* 初始条件：顺序线性表L已存在 */
/* 操作结果：依次对L的每个数据元素输出 */
Status ListTraverse(LinkList L){
    LinkList p = L->next;
    while(p){
        visit(p->data);
        p = p->next;
    }
}

```

```

    peprintf("\n");
    return OK;
}

//头插法!!!!
/* 随机产生n个元素的值, 建立带表头结点的单链线性表L (头插法)
*/
void CreateListHead(LinkList *L,int n){
    LinkList p;
    int i;
    srand(time(0)); //初始化随机数种子
    *L = (LinkList)malloc(sizeof(Node));
    (*L)->next = NULL; //先建立一个带头结点的单链表
    for (i = 0; i < n;i++){
        p = (LinkList)malloc(sizeof(Node)); //生成新结
        点
        p->data = rand() % 100 + 1; //随机生成100以内的
        数字
        //插入到表头
        p->next = (*L)->next;
        (*L)->next = p;
    }
}

//!!!! 尾插法
/* 随机产生n个元素的值, 建立带表头结点的单链线性表L (尾插法)
*/
void CreateListTail(LinkList *L,int n){
    LinkList p, r;
    int i;
    srand(time(0)); //初始化随机数种子
    *L = (LinkList)malloc(sizeof(Node)); //L为整个链表
    r = *L; //r为指向尾部的结点
    for (i = 0; i < n;i++){
        p = (Node *)malloc(sizeof(Node)); //生成新结
        点

```

```

        p->data = rand() % 100 + 1;    //随机生成100以
        内的数字
        r->next = p; //将表尾终端结点的指针指向新结点
        r = p;    //将当前的新结点定义为表尾终端结点
    }
    r ->next = NULL;    //表示当前链表结束
}

```

//测试

```

int main()
{
    LinkList L;
    ElemType e;
    Status i;
    int j,k;
    i=InitList(&L);
    printf("初始化L后:
    ListLength(L)=%d\n",ListLength(L));
    for(j=1;j<=5;j++)
        i=ListInsert(&L,1,j);
    printf("在L的表头依次插入1~5后: L.data=");
    ListTraverse(L);

    printf("ListLength(L)=%d \n",ListLength(L));
    i=ListEmpty(L);
    printf("L是否空: i=%d(1:是 0:否)\n",i);

    i=ClearList(&L);
    printf("清空L后:
    ListLength(L)=%d\n",ListLength(L));
    i=ListEmpty(L);
    printf("L是否空: i=%d(1:是 0:否)\n",i);

    for(j=1;j<=10;j++)
        ListInsert(&L,j,j);
}

```

```

printf("在L的表尾依次插入1~10后: L.data=");
ListTraverse(L);

printf("ListLength(L)=%d \n",ListLength(L));

ListInsert(&L,1,0);
printf("在L的表头插入0后: L.data=");
ListTraverse(L);
printf("ListLength(L)=%d \n",ListLength(L));

GetElem(L,5,&e);
printf("第5个元素的值为: %d\n",e);
for(j=3;j<=4;j++)
{
    k=LocateElem(L,j);
    if(k)
        printf("第%d个元素的值为%d\n",k,j);
    else
        printf("没有值为%d的元素\n",j);
}

k=ListLength(L); /* k为表长 */
for(j=k+1;j>=k;j--)
{
    i=ListDelete(&L,j,&e); /* 删除第j个数据 */
    if(i==ERROR)
        printf("删除第%d个数据失败\n",j);
    else
        printf("删除第%d个的元素值为:
%d\n",j,e);
}
printf("依次输出L的元素: ");
ListTraverse(L);

j=5;

```



```

ListDelete(&L, j, &e); /* 删除第5个数据 */
printf("删除第%d个的元素值为: %d\n", j, e);

printf("依次输出L的元素: ");
ListTraverse(L);

i=ClearList(&L);
printf("\n清空L后:
ListLength(L)=%d\n", ListLength(L));
CreateListHead(&L, 20);
printf("整体创建L的元素(头插法): ");
ListTraverse(L);

i=ClearList(&L);
printf("\n删除L后:
ListLength(L)=%d\n", ListLength(L));
CreateListTail(&L, 20);
printf("整体创建L的元素(尾插法): ");
ListTraverse(L);

return 0;
}

```

### 3.单链表结构与顺序存储结构优缺点

存储分配方式:

顺序存储结构用一段连续的存储单元一次存储线性表的数据元素

单链表采用链式存储结构, 用一组任意的存储单元存放线性表元素

时间性能:

## 1.查找

顺序存储结构 $O(1)$

单链表 $O(n)$

## 2.插入和删除

顺序存储结构需要平均移动表长一半的元素，时间为 $O(n)$

单链表在选出某位置的指针后，插入和删除时间仅为 $O(1)$

## 3.空间性能

顺序存储结构需要预分配存储空间，分大了，浪费，分小了易发生上溢

单链表不需要分配存储空间，只要有就可以分配，元素个数也不受限制

# 第三部分：静态链表

---

详细的部分后续再补上

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

#define OK 1
#define ERROR 0
#define TRUE 1
#define FALSE 0

#define MAXSIZE 1000 /* 存储空间初始分配量 */

typedef int Status; /* Status是函数的类型, 其值是函数结果状态代码, 如OK等 */
typedef char ElemType; /* ElemType类型根据实际情况而定, 这里假设为char */
```

```

Status visit(ElemType c)
{
    printf("%c ", c);
    return OK;
}

/* 线性表的静态链表存储结构 */
typedef struct
{
    ElemType data;
    int cur; /* 游标(Cursor) , 为0时表示无指向 */
} Component, StaticLinkList[MAXSIZE];

/* 将一维数组space中各分量链成一个备用链表, space[0].cur为
头指针, "0"表示空指针 */
Status InitList(StaticLinkList space)
{
    int i;
    for (i=0; i<MAXSIZE-1; i++)
        space[i].cur = i+1;
    space[MAXSIZE-1].cur = 0; /* 目前静态链表为空, 最后
一个元素的cur为0 */
    return OK;
}

/* 若备用空间链表非空, 则返回分配的结点的下标, 否则返回0 */
int Malloc_SSL(StaticLinkList space)
{
    int i = space[0].cur; /* 当前数
组第一个元素的cur存的值 */
    /* 就是要
返回的第一个备用空闲的下标 */

```

```

        if (space[0]. cur)
            space[0]. cur = space[i].cur;          /* 由于要
拿出一个分量来使用了,  */

                                                    /* 所以我
们就得把它的下一个 */

                                                    /* 分量用
来做备用 */
        return i;
    }

/* 将下标为k的空闲结点回收到备用链表 */
void Free_SSL(StaticLinkList space, int k)
{
    space[k].cur = space[0].cur;          /* 把第一个元素的
cur值赋给要删除的分量cur */
    space[0].cur = k;                    /* 把要删除的分量下
标赋值给第一个元素的cur */
}

/* 初始条件：静态链表L已存在。操作结果：返回L中数据元素个数
*/
int ListLength(StaticLinkList L)
{
    int j=0;
    int i=L[MAXSIZE-1].cur;
    while(i)
    {
        i=L[i].cur;
        j++;
    }
    return j;
}

/* 在L中第i个元素之前插入新的数据元素e    */

```

```

Status ListInsert(StaticLinkList L, int i, ElemType
e)
{
    int j, k, l;
    k = MAXSIZE - 1;    /* 注意k首先是最后一个元素的下标
*/
    if (i < 1 || i > ListLength(L) + 1)
        return ERROR;
    j = Malloc_SSL(L);    /* 获得空闲分量的下标 */
    if (j)
    {
        L[j].data = e;    /* 将数据赋值给此分量的data */
        for(l = 1; l <= i - 1; l++)    /* 找到第i个元素
之前的位置 */
            k = L[k].cur;
        L[j].cur = L[k].cur;    /* 把第i个元素之前的cur
赋值给新元素的cur */
        L[k].cur = j;    /* 把新元素的下标赋值给
第i个元素之前元素的ur */
        return OK;
    }
    return ERROR;
}

```

/\* 删除在L中第i个数据元素 \*/

```

Status ListDelete(StaticLinkList L, int i)
{
    int j, k;
    if (i < 1 || i > ListLength(L))
        return ERROR;
    k = MAXSIZE - 1;
    for (j = 1; j <= i - 1; j++)
        k = L[k].cur;
    j = L[k].cur;
    L[k].cur = L[j].cur;
    Free_SSL(L, j);
}

```

```

        return OK;
    }

Status ListTraverse(StaticLinkList L)
{
    int j=0;
    int i=L[MAXSIZE-1].cur;
    while(i)
    {
        visit(L[i].data);
        i=L[i].cur;
        j++;
    }
    return j;
    printf("\n");
    return OK;
}

int main()
{
    StaticLinkList L;
    Status i;
    i=InitList(L);
    printf("初始化L后: L.length=%d\n", ListLength(L));

    i=ListInsert(L, 1, 'F');
    i=ListInsert(L, 1, 'E');
    i=ListInsert(L, 1, 'D');
    i=ListInsert(L, 1, 'B');
    i=ListInsert(L, 1, 'A');

    printf("\n在L的表头依次插入FEDBA后: \nL.data=");
    ListTraverse(L);

    i=ListInsert(L, 3, 'C');

```

```
printf("\n在L的“B”与“D”之间插入“C”后: \nL.data=");  
ListTraverse(L);  
  
i=ListDelete(L,1);  
printf("\n在L的删除“A”后: \nL.data=");  
ListTraverse(L);  
  
printf("\n");  
  
return 0;  
}
```