

# 线性表

## 线性表的定义

- 线性表是n个系统数据元素的有限序列
- 线性表示一种数据结构/逻辑结构

## 特点

- 表中元素的个数有限
- 表中元素具有逻辑上的顺序性
- 表中元素的数据类型都相同，这意味着每个元素占有相同大小的存储空间

## 运算

InitList(&L)	初始化表	构造一个空的线性表
Length(L)	求表长	返回线性表L的长度，即L中数据元素的个数
LocateElem(L,e)	按值查找操作	在表L中查找具有给定关键字值的元素
GetElem(L,i)	按位查找操作	获取表L中第i个位置的元素的值
ListInsert(&L,i,&e)	插入操作	在表中的第i个位置插入指定元素e
ListDelete(&L,i,&e)	删除操作	删除表中第i个位置的元素，并用e返回删除元素的值
PrintList(L)	输出操作	按前后顺序输出线性表的所有元素值
DestroyList(&L)	销毁操作	销毁线性表，并释放空间。

# 顺序表

## 定义

- 用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻

## 特点

- 表中元素的逻辑顺序与其物理顺序相同
- 随机访问，即通过首地址和元素序号能在时间O(1)内找到指定的元素
- 顺序表的存储密度高，每个节点只存储数据元素
- 顺序表逻辑上相邻的元素物理上也相邻，所以插入和删除需要移动大量元素

## 静态分配

## 定义

- 数组的大小和空间事先已经固定
- 一旦空间占满，再加入新的数据就会产生溢出，进而导致程序崩溃

```
#define MaxSize 50;
typedef struct{
    ElemType data[MaxSize];
    int length;
}SqList;
```

## 动态分配

### 定义

- 存储数组的空间是在程序执行过程中通过动态存储分配语句分配的
- 一旦数据空间占满，就另外开辟一块更大的存储空间，用以替换原来的存储空间

```
#define InitSize 100;
typedef struct{
    ElemType *data;
    int MaxSize,length;
}SeqList;
L.data = (ElemType *)malloc(sizeof(ElemType)*InitSize);
```

## 顺序表相关操作

### 插入

```
bool ListInsert(SqList &L,int i;ElemType e)
{
    if(i<1||i>L.length+1){
        return false;
    }
    if(L.length>=MaxSize){
        return false;
    }
    for(int j=L.length;j>=i;j--){
        L.data[j]=L.data[j-1];
    }
    L.data[i-1]=e;
    L.length++;
}
```

```
    return true;
}
```

## 删除

```
bool ListDelete(SqList &L,int i,ElemType e)
{
    if(i<1||i>L.length+1){
        return false;
    }
    e=L.data[i-1];
    for(j=i;j<L.length;j++){
        L.data[j-1]=L.data[j];
    }
    L.length--;
    return true;
}
```

## 按值查找

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

## 单链表

### 特点

- 通过“链”建立起数据元素之间的逻辑关系
- 指针的设置是任意的，可以很方便的表示各种逻辑结构
- 插入和删除操作不需要移动元素，只需要修改指针，但也会失去顺序表可随机

### 头节点和头指针

- 不管带不带头结点，头指针都始终指向链表的第一个节点

- 头结点是带头结点的链表中的第一个节点，节点内通常不存储信息

## 优点

- 单链表设置头结点的目的是方便运算的实现
  - 好处一：有头节点后，插入和删除数据元素的算法就统一了，不再需要判断是否在第一个元素之前插入或删除第一个元素
  - 好处二：不论链表是否为空，其头指针是指向头节点的非空指针，链表的头指针不变，因此空表

## 基本操作

### 采用头插法建立单链表

```
LinkedList List_HeadInsert(LinkedList &L)
{
    LNode *s; int x;
    L=(LNode *)malloc(sizeof(LNode));
    L->next =NULL;
    scanf("%d",&x);
    while(x!=9999){
        s=(LNode *)malloc(sizeof(LNode));
        s->data=x;
        s->next=L->next;
        L->next=s;
        scanf("%d",&x);
    }
    return L;
}
```

### 采用尾插法建立单链表

```
LinkedList List_TailInsert(LinkedList &L)
{
    int x;
    L=(LNode *)malloc(sizeof(LNode));
    LNode *s *r =L;
    scanf("%d",&x);
    while(x!=9999){
        s = (LNode *)malloc(sizeof(LNode));
        s->data = x;
        r->next=s;
        r=s;
    }
}
```

```

        scanf("%d",&x);
    }
    r->next=NULL;
    return L;
}

```

## 按序号查找节点值

```

LNode *GetElem(LinkList L, int i)
{
    int j = 1;        //计数，初始为1

    LNode *p = L->next; //第1个节点指针赋给p

    if (i == 0)

        return L; //若i=0，则返回头结点

    if (i < 1)

        return NULL; //若i无效，则返回NULL

    while (p && j < i)

    { //从第i个节点开始找，查找第i个节点

        p = p->next;

        j++;

    }

    return p; //返回第i个节点的指针，若i大于表长，则返回NULL
}

```

## 按值查找表节点

```

LNode *LocateElem(LinkList L, ElemType e)
{
    LNode *p = L->next;

```

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

    //从第i个节点开始查找data域为e的节点

    p = p->next;

return p;
```

## 插入节点

```
p = GetElem(L, i - 1); //查找插入位置的前驱结点

s->next = p->next;

p->next = s
```

## 删除节点

```
p = GetElem(L, i - 1); //查找删除位置的前驱结点

q = p->next;           //令q指向被删除节点

p->next = q->next;     //将*q节点从链中断开

free(q)               //释放节点的存储空间
```

### Abstract

- 1.带头结点的单链表，判断表为空的条件：head->next==NULL
- 2.不带头结点的单链表，判断表为空的条件：head==NULL

## 双链表

- 单链表的缺点：只能从头结点依次顺序地向后遍历
- 为解决这个问题引入双链表：其中有两个指针prior和next，分别指向其前驱节点和后继节点

### 双链表的插入操作

```
s->next = p->next; //将节点*s插入到*p之后

p->next->prior = s;

s->prior = p;

p->next = s
```

## 双链表的删除操作

```
p->next = q->next; //删除节点*q

q->next->prior = p;

free(q)
```

## 循环链表

### 循环单链表

- 最后一个节点指向头结点
- 可以从任意一个节点开始遍历整个链表
- 仅设置尾指针

### 循环双链表

- 最后一个节点的next指针指向头结点
- 头结点的prior指针指向最后一个节点

#### Info

- 1.判断带头结点循环单链表为空的条件:  $head \rightarrow next == head$
- 2.注意在计算线性表长度的时候, 头结点不计算在内
- 3.带头结点的双循环链表L为空的条件是:  $L \rightarrow prior == L \ \&\& \ L \rightarrow next == L$ (即头结点的prior和next都指向自己)

## 静态链表

- 借助数组来描述线性表的链式存储结构
- 节点也有数据域data和指针域next
- 这里的指针是节点的相对位置（数组下标）， 又称游标



```
#define MaxSize 50
typedef struct {
    ElemType data;
    int next;
} SLinkList[MaxSize];
```

**i Info**

1. 以next== -1 结尾
2. 静态链表需要分配较大空间，插入和删除不需要移动元素的线性表

顺序表和链表的比较

	顺序表	链表
读取方式	能随机存取	不能随机存取
逻辑结构与物理结构	相邻	不一定相邻
空间分配	需要预先按需分配存储空间	可以在需要时申请分配，只要内存有空间就可以分配
按值查找	无序为 $O(n)$ ；有序可采用折半查找 $O(\log_2^n)$	$O(n)$
按序号查找	$O(1)$	$O(n)$
插入	$O(n)$	$O(1)$
删除	$O(n)$	$O(1)$

怎样

	顺序表	链表
--	-----	----



基于存储的考虑	难以估计时不宜用顺序表，顺序表存储密度高	链表不用估计，但存储密度较低
基于运算的考虑	- 若经常按序号访问，选择顺序表	- 经常插入、删除则选择链表 - 常在最后一个元素后插入元素和删除第一个元素，考虑不带头结点且有尾指针的单循环链表 - 常删除最后一个元素，最好使用带尾节点的双链表或者带任意节点的循环双链表
基于环境的考虑	顺序表容易实现	链表是基于指针的

## 易错题

给定有  $n$  个元素的一维数组，建立一个有序单链表的最低时间复杂度是 ( )。

- A.  $O(1)$                   B.  $O(n)$                   C.  $O(n^2)$                   D.  $O(n\log_2 n)$

将长度为  $n$  的单链表链接在长度为  $m$  的单链表后面，其算法的时间复杂度采用大  $O$  形式表示应该是 ( )。

- A.  $O(1)$                   B.  $O(n)$                   C.  $O(m)$                   D.  $O(n+m)$

对于一个带头结点的循环单链表  $L$ ，判断该表为空表的条件是 ( )。

- A. 头结点的指针域为空                  B.  $L$  的值为 NULL  
C. 头结点的指针域与  $L$  的值相等                  D. 头结点的指针域与  $L$  的地址相等