# 线性表的概念

## 线性表的基本概念

文字定义：一个线性表是n个数据元素的有限序列。

一个数据元素可以由**若干个数据项**组成，这时，也可以称数据元素为**记录**。含有大量记录的线性表又称“**文件**”。

例1：26 个英文字母组成的字母表：（*A*, *B*, *C*, …, *Z*） （其中数据元素是字符）

例2：学生成绩表（90, 97, 60, 75,…,84）（其中数据元素是整数）

例3：学生健康情况登记表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **姓 名** | **学 号** | **性 别** | **年龄** | **健康情况** |
| **王小林** | **790631** | **男** | **18** | **健康** |
| **陈 红** | **790632** | **女** | **20** | **一般** |
| **刘建平** | **790633** | **男** | **21** | **健康** |
| **张立立** | **790634** | **男** | **17** | **神经衰弱** |
| **……..** | **……..** | **…….** | **…….** | **…….** |

其中整个登记表就是一个文件，其中的记录（数据元素）就是每个学生的情况。

因此：

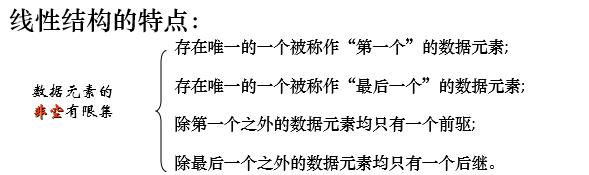
* **线性表中的数据元素可以是各种各样的，但同一线性 表中的元素必定具有相同特性（属于同一数据对象）。**
* **线性表中的数据元素之间存在着序偶关系 <*ai*–1, *ai*>** 。

## 线性表的四个“惟一”

最常用且最简单的一种数据结构。



特点：4个“惟一”。





## 线性表的ADT定义

ADT List {

数据对象：D＝{ ai | ai ∈ElemSet, i=1,2,...,n, n≥0 }

数据关系：R1＝{ <ai-1 ,ai >|ai-1 ,ai∈D, i=2,...,n }

基本操作：

结构初始化操作

结构销毁操作

引用型操作（**操作的结果不改变线性表中的数据元素，也不改变数据元素之间的关系）**

加工型操作 （**操作的结果或修改表中的数据元素，或修改元素之间的关系）**

} ADT List

**{ 结构初始化操作 }**

**InitList ( &L )**　　操作结果：构造一个空的线性表 L 。

**{ 销毁结构 }**

**DestroyList( &L )**   
　　初始条件：线性表 L 已存在。   
　　操作结果：销毁线性表 L 。

**{ 引用型操作 }**

**ListEmpty( L )（判断是否为空）**  
　　　初始条件：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。

**ListLength( L )（求线性表的长度）**  
 　初始条件：线性表 L 已存在。  
　　 操作结果：返回 L 中元素个数。

**PriorElem( L, cur\_e, &pre\_e )（返回前驱）**  
　 　初始条件：线性表 L 已存在。

操作结果：若 cur\_e 是 L 中的数据元素，则用 pre\_e 返回它的前驱，否 则操作失败，pre\_e 无定义。

**NextElem( L, cur\_e, &next\_e )（返回后继）**　　 初始条件：线性表 L 已存在。

操作结果：若 cur\_e 是 L 中的数据元素，则用 next\_e 返回它的后继，否则操作失败，next\_e 无定义。

**GetElem( L, i, &e )（返回第i个元素）**  
　　初始条件：线性表 L 已存在，1≤i≤LengthList(L)。   
　　操作结果：用 e 返回 L 中第 i 个元素的值。

**LocateElem( L, e, compare( ) )（对某一个元素进行定位，返回位序）**　　初始条件：线性表 L 已存在，compare( ) 是判定函数。

　　操作结果：返回 L 中第 1 个与 e 满足关系 compare( ) 的

元素的位序。若这种元素不存在，则返回 0。

**ListTraverse(L, visit( ))（对线性表遍历）**  
　　初始条件：线性表 L 已存在，visit( ) 为访问函数。   
　　操作结果：依次对 L 的每个元素调用函数 visit( )。  
　　　　　　　一旦 visit( ) 失败，则操作失败。

**{ 加工型操作 }**

**ClearList( &L )（与destory的区别在于线性表仍然存在，只不过为空）**初始条件：线性表 L 已存在。   
　　操作结果：将 L 重置为空表。

**PutElem( &L, i, e )（赋值）**初始条件：线性表 L 已存在，1≤i≤LengthList(L)。   
　　操作结果：L 中第 i 个元素赋值为 e 的值。

**ListInsert( &L, i, e )（插入）**初始条件：线性表 L 已存在，1≤i≤LengthList(L)+1。  
　　操作结果：在 L 的第 i 个元素之前插入新的元素 e，L 的长度增 1。

**ListDelete( &L, i, &e )（删除）**初始条件：线性表 L 已存在且非空，1≤i≤LengthList(L)。  
　　操作结果：删除 L 的第 i 个元素，并用 e 返回其值， L 的长度减 1。

## 基本操作的应用举例

例2-1：已知集合 A 和 B，求这两个集合的并集， 使 A＝A∪B，且 B 不再单独存在。

以线性表 LA 和 LB 分别表示集合 A 和 B，两个线性表的数据元素分别为集合 A 和 B 中的成员。

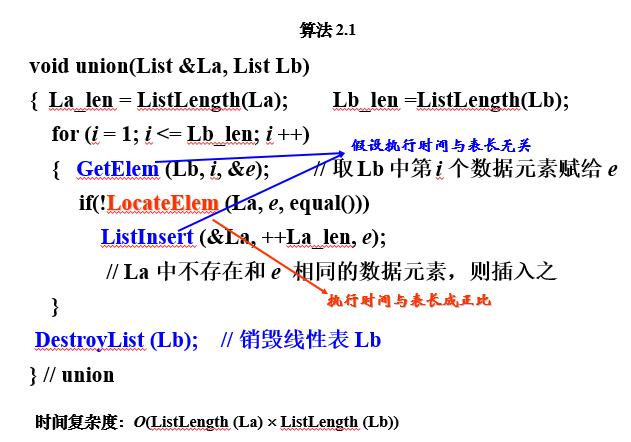
**由此，上述集合求并的问题便可演绎为：**

扩大线性表 LA， 将存在于线性表LB 中而不存在于线性表 LA 中的数据元素插入到线性表 LA 中去。

**解题思路：**

1．从 Lb 中取出一个数据元素；   
2．依次在 La 中进行查询；   
3. 若不存在，则插入之。

重复上述三步直至 Lb 中的数据元素取完为止。



例2.2 合并两个有序表

LA = (3,5,8,11)

LB = (2,6,8,9,11,15,20)

LC = (2,3,5,6,8,8,9,11,11,15,20)

思路：  
1．分别从 La 和 Lb 中取得当前元素 *ai* 和 *bj* ；  
2．若 *ai* ≤ *bj*，则将 *ai* 插入到 Lc 中，否则将 *bj* 插入到 Lc 中。



# 线性表的顺序表示和实现

## 顺序存储结构的概念

* 在计算机中**用一组地址连续的存储单元**依次存储线性表的各个数据元素，称作线性表的顺序存储结构或顺序映象。用这种方法存储的线性表称作顺序表。



假设线性表的每个元素需占 *l* 个存储单元，则第 *i* + 1 个 元素的存储位置和第 *i* 个元素的存储位置之间满足关系：

***LOC*(*ai* +1) = *LOC*(*ai* ) + *l***

**由此，所有数据元素的存储位置均可通过基地址得到：**

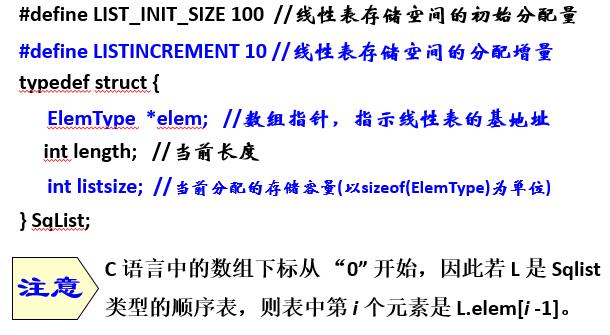
***LOC*(*ai* ) *= LOC*(*a*1) + (*i* -1) × *l***

**特点：**以物理位置相邻表示逻辑关系；任一元素**均可随机存取**。**因此在查找的时候时间复杂度与n没有关系，因此是o（1）。**

**结论：已知位置、获取该位置上的元素非常方便，与该线性表的长度无关 *。***

## 动态分配顺序存储结构

**考虑到线性表因插入元素而使存储空间不足的问题，应允许数组容量进行动态扩充。（静态顺序存储->动态顺序存储）**



## 线性表的操作举例

### 初始化操作

bool InitList(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;

}

### 插入操作

线性表的插入运算是指在表的第 *i* (1 ≤ *i* ≤ *n* +1) 个位置上， 插入一个新结点 *b*，使长度为 *n* 的线性表 (*a*1, …, *ai* –1, *ai*, …, *an*) 变成长度为*n* + 1的线性表 (*a*1, …, *ai* –1, *b*, *ai*, …, *an*)

算法思想：  
1）检查 *i* 值是否超出所允许的范围 (1 ≤ *i* ≤ *n* + 1) ，若超出，则进行“超出范围”错误处理；  
2）将线性表的第 *i* 个元素和它后面所有元素均后移一个位置；   
3）将新元素写入到空出的第 *i* 个位置上；  
4）使线性表的长度增 1。

Status ListInsert\_Sq(Sqlist &L,int i,ElemType e)

{

If (i<1||i>L.length+1) return ERROR;

If (L.length>=L.listsize)

{ //当前存储已满,增加分配

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;

//插入位置之后元素后移

\*q=e;

++L.length;

return OK;

}

注意：

1. realloc的语法：

**格式**：

[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)名=（数据类型\*）realloc（要改变内存大小的指针名，新的大小）。

**返回值**：

如果重新分配成功则返回指向被分配内存的[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)，否则返回空指针NULL。

Realloc产生两种结果，一种是直接在原有数组的后面直接添加新的内存空间，另一种就是要重新开辟原来的数据重新进行分配。（因此为避免出错，将重新分配的空间的地址赋值给一个新的变量）

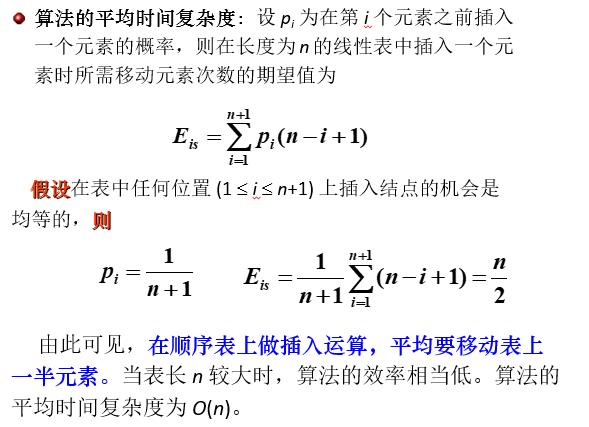
1. 在循环过程中，除了上面的代码可以实现，也可以这样操作：

for(int j = L.length-1;j >= i-1;j--)

L.elem[j+1] = L.elem[j];

1. 时间复杂度：

* 问题规模是表的长度，设它的值为 *n*。
* 算法的时间主要花费在向后移动元素的 for 循环语句上。该语句的循环次数为 **(*n*– *i* +1)**。由此可看出，所需移动结点的次数不仅依赖于表的长度 *n*，而且还与插入位置 *i* 有关。
* 当插入位置在表尾 (*i*=*n* +1) 时，不需要移动任何元素；这是最好情况，**其时间复杂度 *O*(1)**。
* 当插入位置在表头 (*i* = 1) 时，所有元素都要向后移动，循环 语句执行 *n* 次，这是最坏情况，**其时间复杂度 *O*(*n*)**。



### 删除操作

线性表的删除运算是指将线性表的第 *i* (1 ≤ *i* ≤ *n*) 个结点 删除，使长度为 *n* 的线性表(*a*1, …, *ai* –1, *ai*, *ai* +1, …, *an*) 变成长度为 *n* -1 的线性表 (*a*1, …, *ai* –1, *ai* +1, …, *an*)

算法思想：  
1) 检查 *i* 值是否超出所允许的范围 (1 ≤ *i* ≤ *n*)，若超出，则进行“超出范围”错误处理；  
2) 将线性表的第 *i* 个元素后面的所有元素均前移一个位置；  
3) 使线性表的长度减 1。

Status ListDelete\_Sq(SqList &L, int *i*, ElemType &*e*) {  
 if ((*i* <1) || (*i* > L.length)) return ERROR; // 删除位置不合法  
 *p* = &(L.elem[*i* -1]); // *p*为被删除元素的位置  
 *e* = \**p*; // 被删除元素的值赋给 *e*  
 *q* = L.elem+L.length-1; // 表尾元素的位置  
 for (++*p*; *p* <= *q*; ++*p*) \*(*p*-1) = \**p*;

// 被删除元素之后的元素左移

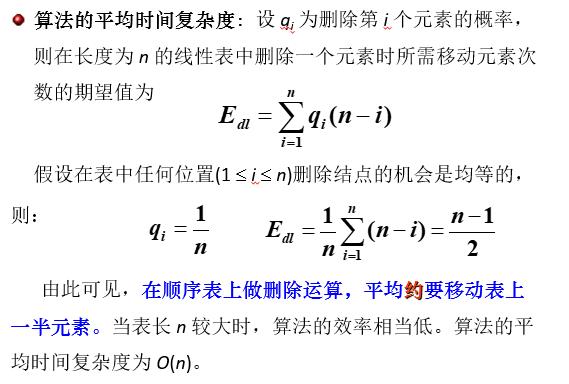
--L.length; //表长减 1

return OK;

}//ListInsert\_sq

**删除算法的复杂度分析**

* 问题规模是表的长度，设它的值为 *n*。
* 算法的时间主要花费在向前移动元素的 for 循环语句上。 该语句的循环次数为 (*n* – *i*)。由此可看出，所需移动结点的次数不仅依赖于表的长度 *n*，而且还与删除位置 *i* 有关。
* 当删除位置在表尾 (*i* = *n*) 时，不需要移动任何元素；这是最好情况，其时间复杂度 *O*(1)。
* 当删除位置在表头 (*i* = 1) 时，有 *n* -1 个元素要向前移动，循环语句执行 *n* -1 次，这是最坏情况其时间复杂度 *O*(*n*)。



***/\*\*\****

***\*DynaSeqList.cpp - 动态顺序表，即顺序表的动态数组实现***

***\****

***\****

***\*题目：实验2-1 线性表的动态顺序存储实现***

***\****

***\*\*\*\*/***

***#include*** <stdlib.h>

***#include*** <malloc.h>

***#include*** <memory.h>

***#include*** <assert.h>

***#include*** "DynaSeqList.h"

**const** **int** LIST\_INIT\_SIZE = 100; ***// 表初始分配的最大长度***

**const** **int** LISTINCREMENT = 10; ***// 分配内存的增量***

***/\*------------------------------------------------------------***

***操作目的： 初始化顺序表***

***初始条件： 无***

***操作结果： 构造一个空的线性表***

***函数参数：***

***SqList \*L 待初始化的线性表***

***返回值：***

***bool 操作是否成功***

***------------------------------------------------------------\*/***

**bool** **InitList**(**SqList** \*L)

{

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

**if** (NULL == L->elem)

**return** **false**;

L -> length = 0;

L -> listsize = LIST\_INIT\_SIZE;

**return** **true**;

}

***/\*------------------------------------------------------------***

***操作目的： 销毁顺序表***

***初始条件： 线性表L已存在***

***操作结果： 销毁线性表L***

***函数参数：***

***SqList \*L 待销毁的线性表***

***返回值：***

***无***

***------------------------------------------------------------\*/***

**void** **DestroyList**(**SqList** \*L)

{

free(L->elem);

L->length = 0;

L->listsize = 0;

}

***/\*------------------------------------------------------------***

***操作目的： 判断顺序表是否为空***

***初始条件： 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***

***函数参数：***

***SqList L 待判断的线性表***

***返回值：***

***bool 是否为空***

***------------------------------------------------------------\*/***

**bool** **ListEmpty**(**SqList** L)

{

**if** (L.length==0)

**return** **true**;

**else**

**return** **false**;

}

***/\*------------------------------------------------------------***

***操作目的： 得到顺序表的长度***

***初始条件： 线性表L已存在***

***操作结果： 返回L中数据元素的个数***

***函数参数：***

***SqList L 线性表L***

***返回值：***

***int 数据元素的个数***

***------------------------------------------------------------\*/***

**int** **ListLength**(**SqList** L)

{

**return** L.length;

}

***/\*------------------------------------------------------------***

***操作目的： 得到顺序表的第i个元素***

***初始条件： 线性表L已存在，1<=i<=ListLength(L)***

***操作结果： 用e返回L中第i个数据元素的值***

***函数参数：***

***SqList L 线性表L***

***int i 数据元素的位置***

***ElemType \*e 第i个数据元素的值***

***返回值：***

***bool 操作是否成功***

***------------------------------------------------------------\*/***

**bool** **GetElem**(**SqList** L, **int** i, **ElemType** \*e)

{

**if** (i<=L.length && i>=1)

{

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

**return** **true**;

}

**else**

{

**return** **false**;

}

}

***/\*------------------------------------------------------------***

***操作目的： 得到顺序表指定元素的位置***

***初始条件： 线性表L已存在***

***操作结果： 返回L中第一个与e满足关系compare()的数据元素的位序。***

***若这样的元素不存在则返回0。***

***函数参数：***

***SqList L 线性表L***

***ElemType e 数据元素e***

***int (\*fp)() 用于比较相等的函数指针***

***返回值：***

***int 与e满足关系compare()的数据元素的位序***

***------------------------------------------------------------\*/***

**int** **LocateElem**(**SqList** L, **ElemType** e, **int** (\*fp)(**ElemType**, **ElemType**))

{

**ElemType** a;

**ElemType** i;

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

{

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

**break**;

}

a = i;

**return** (\*fp)(a,0);

}

***/\*------------------------------------------------------------***

***操作目的： 得到顺序表指定元素的前驱***

***初始条件： 线性表L已存在***

***操作结果： 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***

***函数参数：***

***SqList L 线性表L***

***ElemType cur\_e 数据元素cur\_e***

***ElemType \*pre\_e 前驱数据元素***

***返回值：***

***bool 操作是否成功***

***------------------------------------------------------------\*/***

**bool** **PriorElem**(**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];

**return** **true**;

}

**else**

{

**return** **false**;

}

}

}

**if** (i >= L.length)

{

**return** **false**;

}

}

***/\*------------------------------------------------------------***

***操作目的： 得到顺序表指定元素的后继***

***初始条件： 线性表L已存在***

***操作结果： 若cur\_e是L的数据元素，且不是最后一个，则用nxt\_e返***

***回它的后继，否则操作失败，nxt\_e无定义***

***函数参数：***

***SqList L 线性表L***

***ElemType cur\_e 数据元素cur\_e***

***ElemType \*nxt\_e 后继数据元素***

***返回值：***

***bool 操作是否成功***

***------------------------------------------------------------\*/***

**bool** **NextElem**(**SqList** L, **ElemType** cur\_e, **ElemType** \*nxt\_e)

{

**int** i;

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

{

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

{

**if** (i < L.length - 1)

{

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

**return** **true**;

}

**else**

{

**return** **false**;

}

}

}

**if** (i >= L.length)

{

**return** **false**;

}

}

***/\*------------------------------------------------------------***

***操作目的： 遍历顺序表***

***初始条件： 线性表L已存在***

***操作结果： 依次对L的每个元素调用函数fp***

***函数参数：***

***SqList L 线性表L***

***void (\*fp)() 访问每个数据元素的函数指针***

***返回值：***

***无***

***------------------------------------------------------------\*/***

**void** **ListTraverse**(**SqList** L, **void** (\*fp)(**ElemType**))

{

**int** i;

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

{

(\*fp)(L.elem[i]);

}

}

***/\*------------------------------------------------------------***

***操作目的： 清空顺序表***

***初始条件： 线性表L已存在***

***操作结果： 将L置为空表***

***函数参数：***

***SqList \*L 线性表L***

***返回值：***

***无***

***------------------------------------------------------------\*/***

**void** **ClearList**(**SqList** \*L)

{

L->length = 0;

}

***/\*------------------------------------------------------------***

***操作目的： 在顺序表的指定位置插入结点，插入位置i表示在第i个***

***元素之前插入***

***初始条件： 线性表L已存在，1<=i<=ListLength(L) + 1***

***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***

***函数参数：***

***SqList \*L 线性表L***

***int i 插入位置***

***ElemType e 待插入的数据元素***

***返回值：***

***bool 操作是否成功***

***------------------------------------------------------------\*/***

**bool** **ListInsert**(**SqList** \*L, **int** i, **ElemType** e)

{

**ElemType** \*newbase;

**ElemType** \*q,\*p;

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

**return** **false**;

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

{

newbase = (**ElemType** \*)realloc(L->elem,(L->listsize+LISTINCREMENT)\***sizeof**(**ElemType**));

**if** (NULL == newbase)

**return** **false**;

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** **true**;

}

***/\*------------------------------------------------------------***

***操作目的： 删除顺序表的第i个结点***

***初始条件： 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，并用e返回其值，L的长度减1***

***函数参数：***

***SqList \*L 线性表L***

***int i 删除位置***

***ElemType \*e 被删除的数据元素值***

***返回值：***

***bool 操作是否成功***

***------------------------------------------------------------\*/***

**bool** **ListDelete**(**SqList** \*L, **int** i, **ElemType** \*e)

{

**ElemType** \*p,\*q;

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

**return** **false**;

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

\*e = \*p;

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

**for** (++p;p <= q; ++p)

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

--L->length;

**return** **true**;

}

# 线性表的链式表现和实现

用一组**物理位置任意**的存储单元来存放线性表的数据元素。 这组存储单元既可以是连续的，**也可以是不连续的**，甚至是零散分布在内存中的任意位置上的。因此，链表中元素的逻辑次序和物理次序不一定相同。

## 线性链表

单链表是由头指针唯一确定，因此单链表可以用头指针的名字来命名。

typedef struct Lnode{

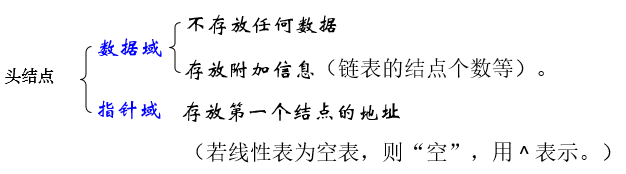
//声明结点的类型和指向结点的指针类型

ElemType data; //数据元素的类型

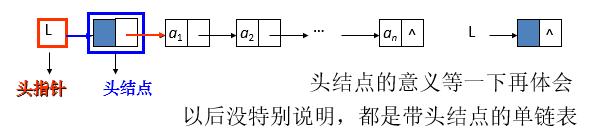
struct Lnode \*next; //指示结点地址的指针

}Lnode, \*LinkList;

头结点：在单链表的第一个结点之前人为地附设的一个结点。



头指针存放头结点的地址。

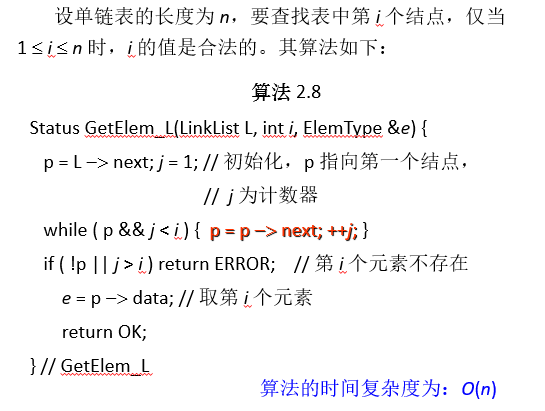


## 单链表的基本操作

### 1、查找运算

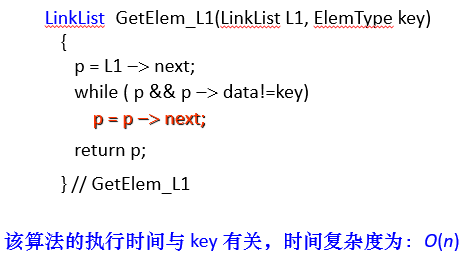
* 按序号查找（GetElem(L, *i*, &*e*)在链表中的实现）

在单链表中，即使知道被访问结点的序号 *i* ，也不能像顺序表中那样直接按序号 *i* 访问结点，而只能从头指针出发，顺链域next逐个结点往下搜索，直到搜索到第 *i* 个结点为止。因此，单链表是非随机存取的存储结构。

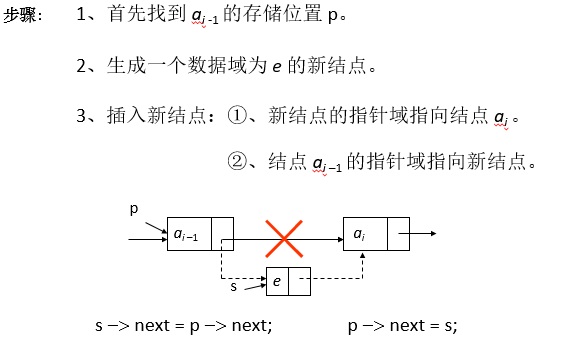


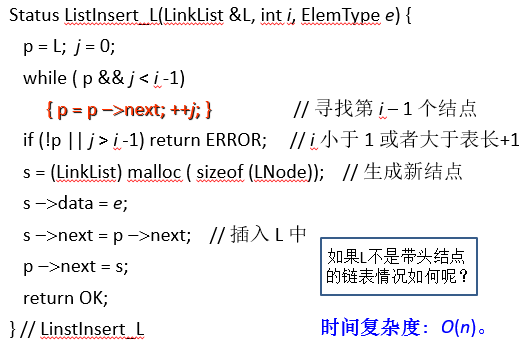
* 按值查找（LocateElem( L, e) 在链表中的实现）

按值查找是在单链表中查找结点值等于给定值 key的结点，若有的话，则返回首次找到的其值为 key 的结点的存储位置；否则返回 NULL。其算法如下：



### 2、插入运算（ListInsert(&L, *i*, *e*)在链表中的实现）

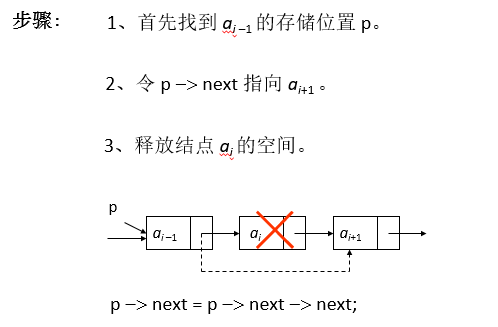




不带头结点的情况：需要对k==1的情况进行分开讨论

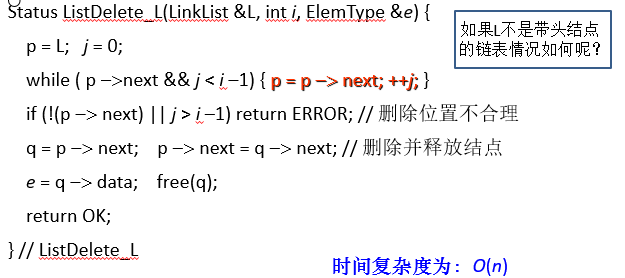
这个时间复杂度是在进行寻找结点的身上，在p指向的结点进行插入操作的时候时间复杂度是o(1)，空链表也可以进行插入

### 3、删除运算（ListDelete(&L, *i*, &*e*)在链表中的实现）



头结点的作用是对链表进行操作时，可以对空表、非空表的情况以及对首元结点进行统一处理，编程更方便。

头结点与头指针：头结点就是第一个结点之前的结点，头指针是指向第一个结点的指针。

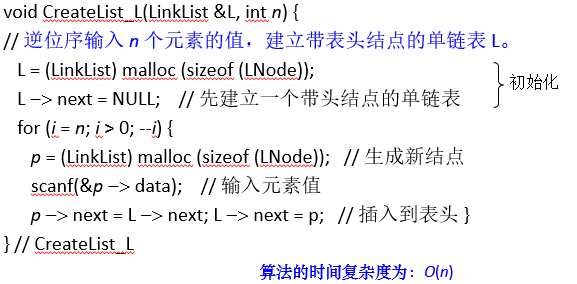


不带头结点的情况：需要对k==1的情况进行分开讨论

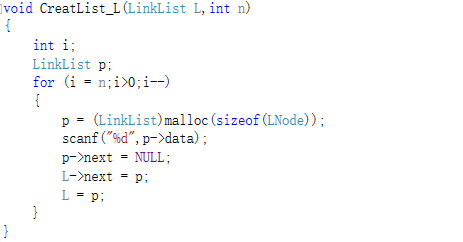
### 4、创建链表的操作

因为每个新生成的结点的插入位置在表尾，则算法中必须维持**一个始终指向已建立的链表表尾的指针**。

* 头插法



* 尾插法



### 5、链表的逆置

### 6、链表的排序

### 7、链表的合并

void MergeList\_L(LinkList La,LinkList Lb,LinkList Lc)

{

LinkList pa,pb,pc;

pa = La->next;pb = Lb->next;

Lc = pc = La;

while(pa&&pb)

{

if (pa->data<=pb->data)

{

pc->next = pa;

pc =pa;

pa = pa->next;

}

else

{

pc->next = pb;

pc =pb;

pb = pb->next;

}

}

pc->next = pa?pa:pb;

free(Lb);

}

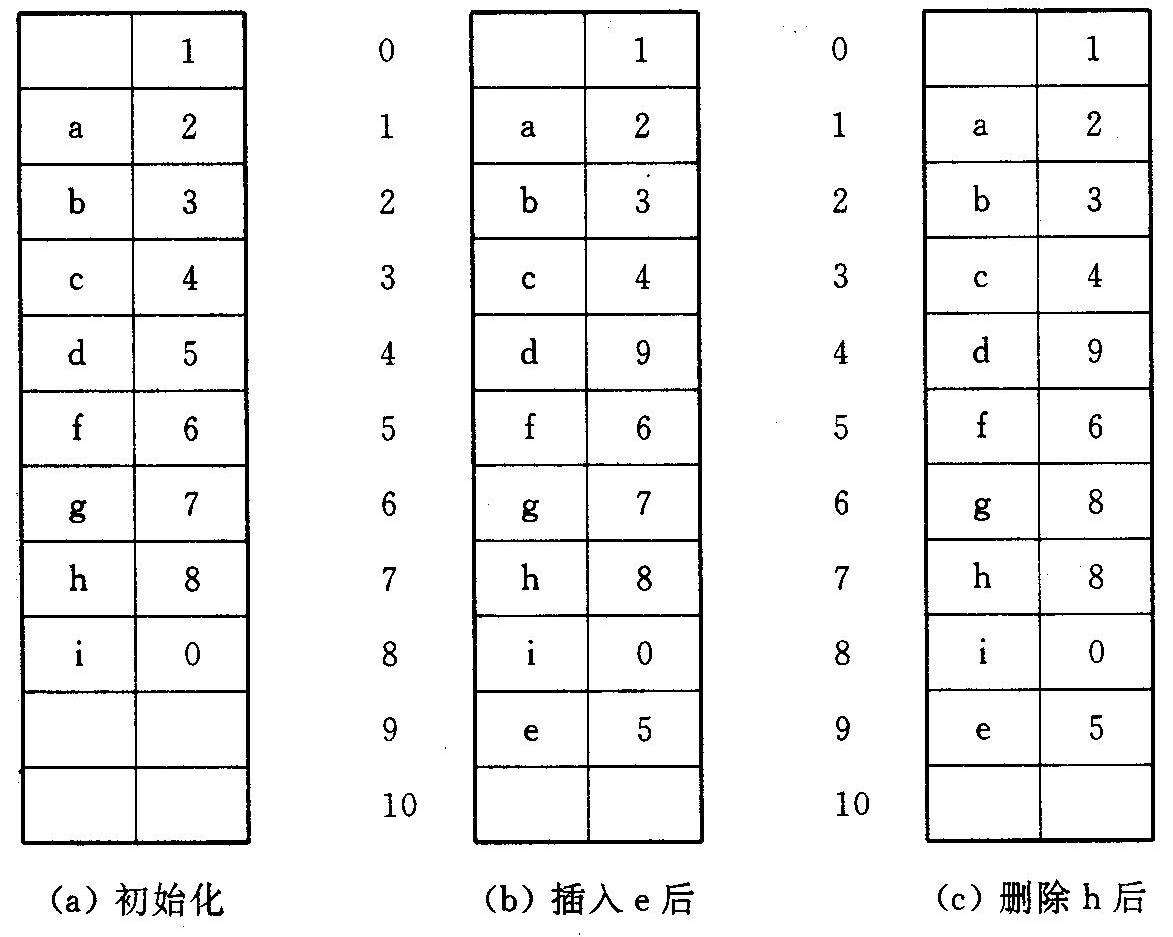
## LinkList L和LinkList \*L的区别

对于LinkList L: L是指向定义的node结构体的指针,可以用->运算符来访问结构体成员,即L->elem,而(\*L)就是个Node型的结构体了,可以用点运算符访问该结构体成员,即(\*L).elem;

对于LinkList \*L:L是指向定义的Node结构体指针的指针,所以(\*L)是指向Node结构体的指针,可以用->运算符来访问结构体成员,即(\*L)->elem,当然,(\*\*L)就是Node型结构体了,所以可以用点运算符来访问结构体成员,即(\*\*L).elem;

在链表操作中,我们常常要用链表变量作物函数的参数,这时,用LinkList L还是LinkList \*L就很值得考虑深究了,一个用不好,函数就会出现逻辑错误,**其准则是**:  
如果函数会改变指针L的值,而你希望函数结束调用后保存L的值,那你就要用LinkList \*L,这样,向函数传递的就是指针的地址,结束调用后,自然就可以去改变指针的值;  
而如果函数只会修改指针所指向的内容,而不会更改指针的值,那么用LinkList L就行了;

# 静态链表



#define MAXSIZE 1000 //链表的最大长度

typedef struct

{

ElemType data;

int cur;

}component, SLinkList[MAXSIZE];