# 数组的定义

## 数组的基本概念

**数组：按一定格式排列起来的具有相同类型的数据元素的集合。**

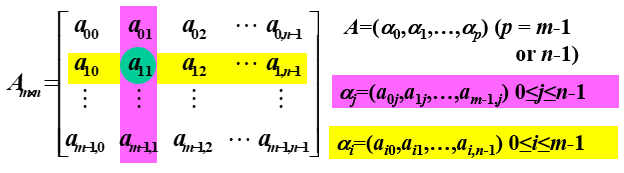
**一维数组**：若线性表中的数据元素为非结构的简单元素，则称为一维数组。

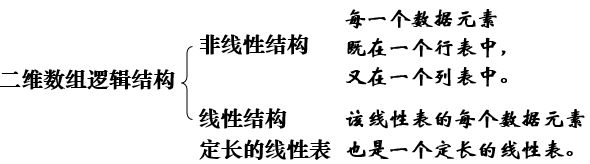
一维数组的逻辑结构：**线性结构。定长的线性表**。

**声明格式：**数据类型 变量名称[长度]；

例：**int num[5] = {0，1，2，3，4};**

**二维数组**：若一维数组中的数据元素**又是一维数组结构**，则称为二维数组。





声明格式：数据类型 变量名称[行数] [列数]

**例：int num[5] [8]**

在 C 语言中，一个二维数组类型**也可以定义为一维数组类型**（其分量类型为一维数组类型），即：

**typedef elemtype array2[m][n];**

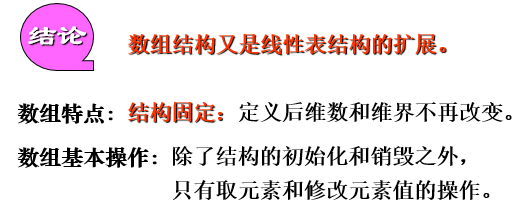
等价于：

**typedef elemtype array1[n];**

**typedef array1 array2[m];**

**三维数组：**若二维数组中的元素**又是一个一维数组结构**，则称作三维数组。

***n* 维数组：**若 ***n* -1 维**数组中的元素又是一个**一维数组**结构，则称作 *n* 维数组。



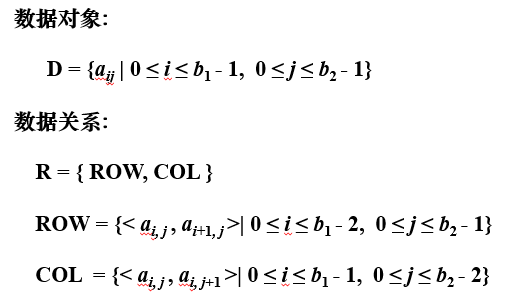
## 数组的抽象数据类型的定义

**ADT Array {**  
**数据对象：**D={*aj*1 *j*2 … *jn* | *ji*= 0, ..., *bi* -1, *i* = 1, 2, .., *n*, *n* (>0) 称为数组的维数，*bi* 是数组第 *i* 维的长度， *ji* 是数组元素的第 *i* 维下标，*a j*1 *j*2 … *jn*∈ElemSet }

**数据关系**：R＝{R1, R2, ..., R*n*}

R*i*＝{<*a*j1*...*j*i...*j*n* , *a*j1*...*j*i+*1*...*j*n* > |0≤*jk*≤*bk* -1, 1≤*k*≤*n* 且 *k*≠*i*, 0≤*ji*≤*bi* - 2, *a*j1…j*i*…j*n*, *a*j1…j*i*+1…j*n*∈D, i = 2, ..., *n* }

**二维数组的抽象数据类型的数据对象和数据关系的定义**



**基本操作：**

**InitArray(&A, *n*, bound1, ..., bound*n*)**

操作结果：若维数 *n* 和各维长度合法，则构造相应的数组 A，并返回 OK。

**DestroyArray(&A)**

操作结果：销毁数组 A。

**Value(A, &*e*, index1, ..., index*n*)**

初始条件：A 是 *n* 维数组，*e* 为元素变量。

操作结果：若各下标不超界，则 *e* 赋值为所指定的 A 的元素值，并返回 OK。

**Assign(&A, *e*, index1, ..., index*n*)**

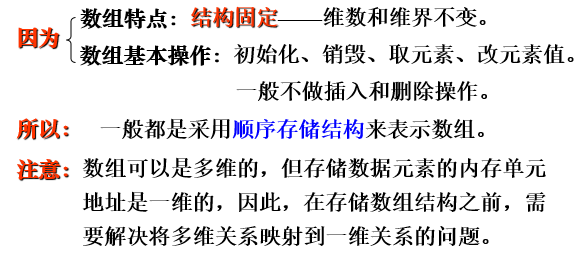
初始条件：A 是 *n* 维数组，*e* 为元素变量。

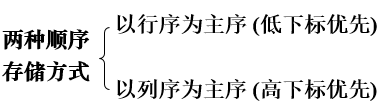
操作结果：若下标不超界，则将 *e* 的值赋给所指定的A 的元素，并返回 OK。

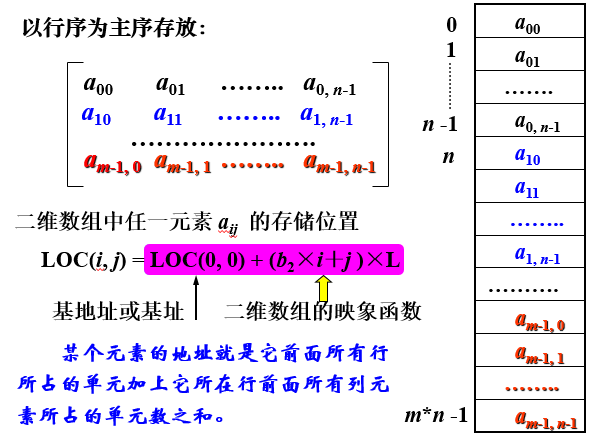
} ADT Array

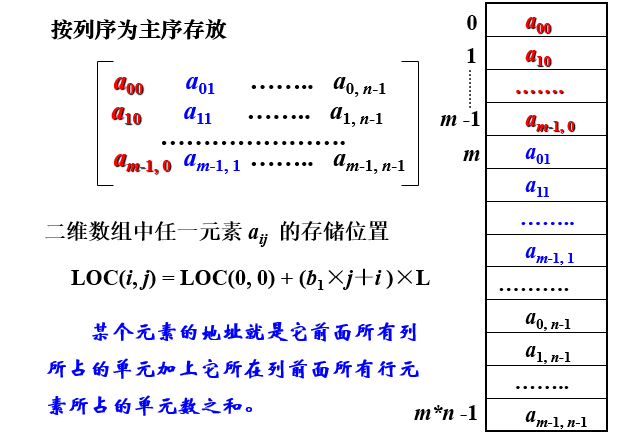
# 数组的顺序表示和实现

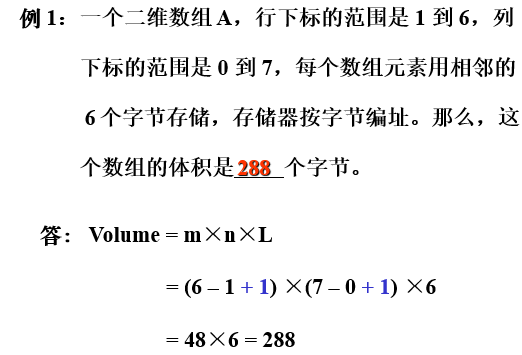
## 数组的顺序表示和实现

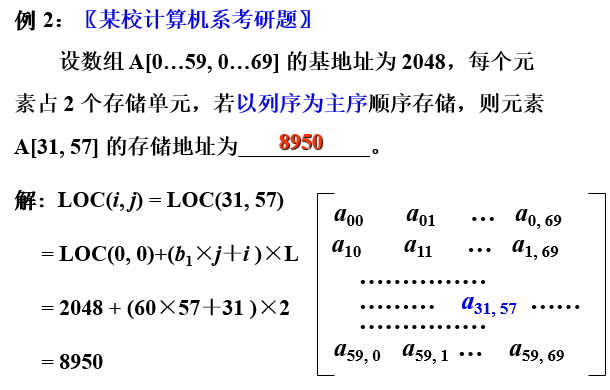


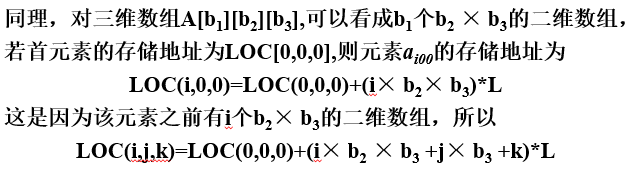












## 数组的顺序表示及相关说明

typedef struct{

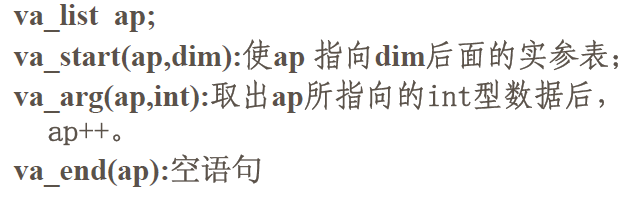
ElemType \*base; //存放元素的基址

int dim; //维数

int \*bounds; //等价整形数组，存各维长度

int \*constants; //每变化一维的跨度

}Array;



# 广义表的概念、性质

## 广义表的定义

**1、广义表的定义**

广义表（又称列表 **Lists**）是***n*≥0**个元素 *a*1, *a*2, …, *an*的有限序列，其中每一个*ai* 或者是**原子**（单个元素），或者是一个**子表**。

**例：**中国举办的国际足球邀请赛，参赛队名单可表示如下：

（阿根廷，巴西，德国，法国，（），西班牙， 意大利，英国，（国家队，建业））

在这个表中，韩国队应排在法国队后面，但由于其水平低未敢参加，成为**空表**。国家队、建业队、实德队均作为东道主的参赛队参加，**构成一个小的线性表，成为原线性表的一个数据元素。这种拓宽了的线性表就是广义表。**

**2、广义表的表示**

**广义表通常记作： *LS* = (*a*1，*a*2，…，*an*)**

其中： *LS* 为**表名**， *n* 为表的**长度**， 每一个 ***ai*** 为表的元素。

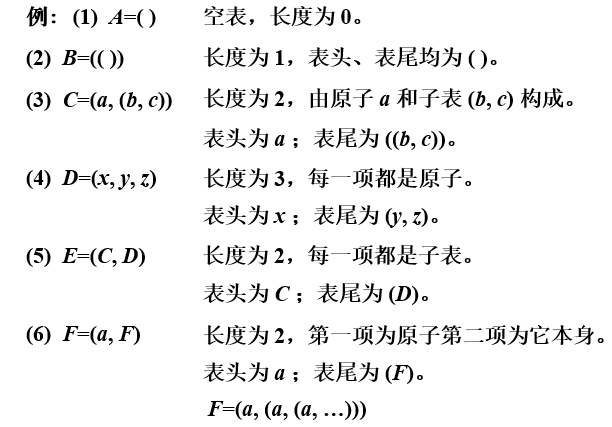
**习惯上，一般用大写字母表示广义表，小写字母表示原子。**

**3、表头和表尾**

**表头：**若 *LS* **非空** (*n*≥1 )，则其**第一个元素 *a*1** 就是表头。记作**head(*LS*)** = *a*1。**注：表头可是原子，也可是子表。**

**表尾：除表头之外的其它元素组成的表。记作 tail(*LS*) = (*a*2, ..., *an*)。**

**注：表尾不是最后一个元素，而是一个子表。**



注意：对于**非空表**才有表头和表尾的概念

## 广义表的性质

**(1) 广义表中的数据元素有相对次序；**

**(2)** 广义表的**长度**定义为**最外层所包含元素的个数**；

**如： *C*=(*a*, (*b*, *c*)) 是长度为 2 的广义表。**

**(3) 广义表的深度定义为该广义表展开后所含括号的重数；**

***A* = (*b*, *c*) 的深度为 1，*B* = (*A*, *d*) 的深度为 2，*C* = (*f*, *B*, *h*) 的深度为 3。**

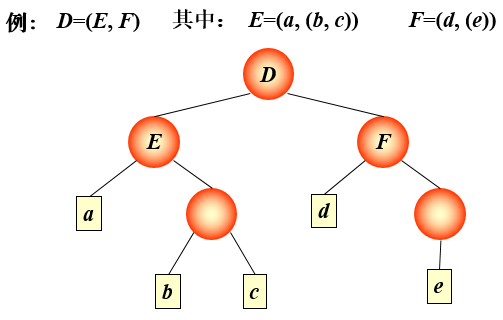
**注意：“原子”的深度为 0 ; “空表”的深度为 1 。**

**(4) 广义表可以为其他广义表共享；**如：广义表 *B* 就共享表 *A*。在 *B* 中不必列出 *A* 的值，而是通过**名称**来引用

**(5) 广义表可以是递归的表。如：*F*=(*a*, *F*)=(*a*, (*a*, (*a*, …)))**

**注意：递归表的深度是无穷值，长度是有限值。**

**(6) 广义表是多层次结构，**广义表的元素可以是**单元素**， 也可以是**子表**，而子表的元素还可以是子表，…。 可以用图形象地表示。

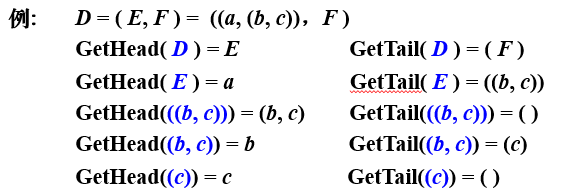


## 广义表基本运算

**取表头运算 GetHead 和取表尾运算 GetTail**

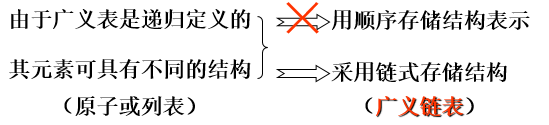
**若广义表 *LS*=(*a*1, *a*2, …, *an*)， 则 GetHead(*LS*) = *a*1  GetTail(*LS*) = (*a*2, …, *an*)。**

**注意：取表头得到的结果可以是原子，也可以是一个子表。取表尾得到的结果一定是一个子表。**

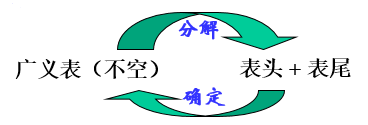


# 广义表的存储结构、相关递归算法

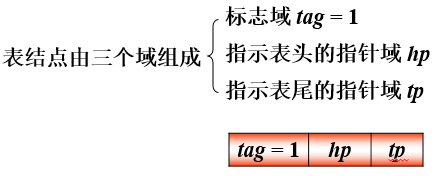
## 广义表的存储结构

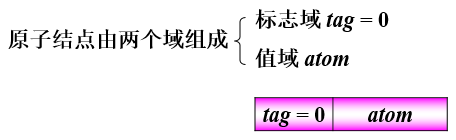


**首尾链表**



**结点的结构形式:**





typedef enum {ATOM, LIST} ElemTag; // ATOM=0：单元素；LIST=1：子表

typedef struct GLNode {

Elemtag tag; // 标志域，用于区分元素结点和表结点

union { // 元素结点和表结点的联合部分

Atomtype atom; // atom 是原子结点的值域

struct

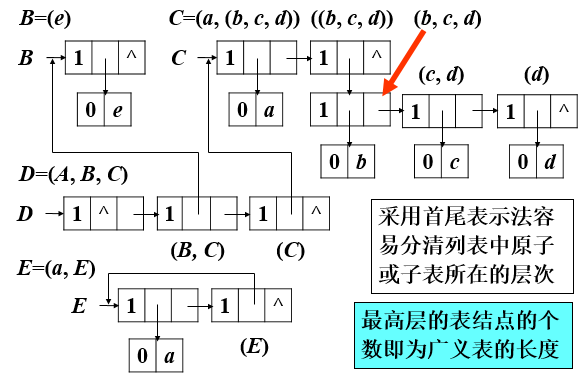
{struct GLNode \*hp, \*tp; }ptr; // ptr是表结点的指针域，ptr.hp 和 ptr.tp分别 // 指向表头和表尾

};

}\*GList; // 广义表类型

**结点的链接**





## 广义表的递归算法

**求深度算法**

**1、广义表的深度=Max {子表的深度} +1**

**2、**可以直接求解的两种简单情况为:

**空表的深度 = 1**

**原子的深度 = 0**

int GlistDepth(Glist L)

{

// 返回指针L所指的广义表的深度

if (!L) return 1; //空表深度为1

if (L->tag == ATOM) return 0; //原子深度为0

for (max=0, pp=L; pp; pp=pp->ptr.tp)

{

dep = GlistDepth(pp->ptr.hp);

if (dep > max)

max = dep;

}

return max + 1;

} // GlistDepth