第4章 扩展线性表——数组与广义表

知识要点

(1) 熟悉多维定长线性表---数组的基本概念，数组的顺序存储、链式存储。多维数组与多维线性表的关系。

(2) 掌握常见特殊数组的压缩存储，了解稀疏矩阵的压缩存储方法。

(3) 熟悉多层推广线性表---广义表的基本概念，初步掌握广义表的连式存储结构。

(4) 了解数组和广义表的应用。

4.1 扩展线性表---数组*

4.1.1 数组的定义

数组(Array)是由n(n >1)个具有相同数据类型的数据元素a_0，a_2，…，a_{n-1}组成的有序序列，且该有序序列必须存储在一块地址连续的存储单元中。

数组的特点：

(1)**数组是一种“均匀”结构**，同一数组中各个数据元素必须是同一类型的；

(2)**数组是一种随机存取结构**，只要给定一组下标，就可以访问与其对应的单元。

(3)**数组中的数据元素个数是固定的**。一旦定义好了一个数组，其数据元素个数就不得有任何变化。内容可空可变。

4.1.2 数组的基本操作

(1)随机存取操作：GetArrayElement(A，SC_1，SC_2，…，SC_n)

初始条件：数组A已知，SC_1，SC_2，…，SC_n是一组没有越界的下标；

操作结果：通过给定的这组下标，在数组A中存取对应下标的数据元素。

(2)随机修改操作：ModifyArrayElement(A，SC_1，SC_2，…，SC_n)

初始条件：数组A已知，SC_1，SC_2，…，SC_n是一组没有越界的下标；

操作结果：通过给定的这组下标，修改数组A中对应下标的数据元素。

(3)数组初始化：InitialArray(A)

初始条件：无；

操作结果：在数组的维数和维长都合法的情况下，构造一个数组A。

4.1.3 数组的存储结构

静态数组（定义数组）和动态数组（指针数组）

- 对于下界为0的一个**一维数组a[n]**，给定第一个数组元素a_0的存储地址是$Loc(a_0)$，每个数据元素所占的存储单元为k，则任意数据元素a_i的存储单元地址：

$$Loc(a_i)= Loc(a_0) + i * k \quad (0 \le i < n)$$

- **二维数组推广到一般情况**：设二维数组的行下标的取值范围为rl到ru，列下标取值范围为cl到cu。
- **以行序为主序的存储方式**，则任意数据元素$a_{i,j}$的存储单元地址：

$$Loc(a_{i,j})= Loc(a_{rl,cl}) + ((i - r_l) * (cu - c_l + 1) + (j - cu)) * k$$

- **以列序为主序的存储方式**，则任意数据元素$a_{i,j}$的存储单元地址：

$$Loc(a_{i,j})= Loc(a_{rl,cl}) + ((j - c_l) * (ru - r_l + 1) + (i - r_l)) * k$$

4.1.4 矩阵的压缩存储

- **压缩存储**是指为多个值相同的矩阵元素分配一个存储空间，值为0的矩阵元素不分配空间的存储方式。
- 把矩阵元素值相同的或者的0元素分布有一定规律的矩阵，则称之为特殊矩阵。
- 如果矩阵中的0元素占绝大部分，则称此类矩阵为稀疏矩阵。

1.特殊矩阵

若n阶方阵A中的数据元素满足下列性质：

$$a_{i,j} = a_{j,i} \quad (0 \le i, j < n)$$

则称矩阵A为n阶**对称矩阵**。

$$\begin{bmatrix} 2 & 3 & 1 \\ 3 & -2 & 4 \\ 1 & 4 & 5 \end{bmatrix}$$

假设以一维数组SC[0～n(n+1)/2-1]来存储对阵矩阵A[n][n]，则SC[k]与矩阵元素a_{ij}之间存在着一一对应关系：

$k = i*(i+1)/2 + j$　　当i≥j

$k = j*(j+1)/2 + i$　　当i<=j

2.稀疏矩阵的压缩存储

假设在n*m的矩阵中，有x个数据元素的值不为0，设

$$\Gamma = \frac{x}{n \times m}$$

则称Γ为稀疏因子，如果某一矩阵的稀疏因子Γ≤0.05时，则称该矩阵为稀疏矩阵(Sparse Matrix)。

$$M = \begin{bmatrix} 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9 & 0 \end{bmatrix}$$

图 4-1 稀疏矩阵示意图

图4-1的三元组线性表为：

((0,1,2),(0,2,-1),(2,0,-1),(2,5,4),(4,1,8),(5,0,5),(6,5,9))

4.2 扩展线性表---广义表*

4.2.1广义表的定义及性质

广义表L是由n≥0个元素组成的有穷序列，即：

$$L = (a_1, a_2, \ldots, a_i, \ldots, a_n)$$

其中： 1) 广义表中的元素a_i要么是**原子**，要么是**广义表**。元素a_i可以是原子项，也可以是广义表，分别称为广义表L的**原子**(Atom)和**子表**。

2) L被称为广义表的名称。

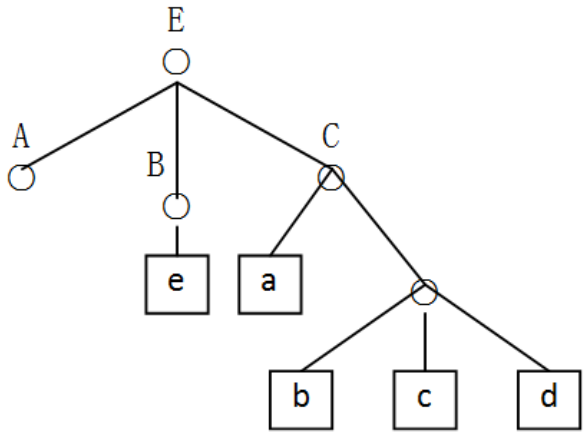
3) 在广义表的定义中，n是广义表的长度

4) 当L非空时，称第一个元素为a_1为L的**表头(Head)**，其余元素组成的表（a_2，a_3，…，a_n)为**表尾(tail)**。

很显然，在广义表的定义中又用到了广义表的概念，所以广义表是一个递归定义。

四个重要结论：

（1）**层次性**。广义表中的元素是不“均匀”的(元素类型不同)，是一个多层次的结构。右图表示的是广义表E,图中〇表示广义表，□表示原子。其中〇有几层，广义表的深度就为几层。



（2）**共享性**。如上例中，广义表A、B和C为广义表E的子表，那么，在广义表E中就不必列出子表的值，可以通过广义表的名称来引用。

（3）**递归性**。如上例中的广义表F可以得知，广义表可以是其本身的一个子表。

（4）任何非空广义表的表头可以是原子，也可以是广义表，但其**表尾必定是广义表**。

A＝()
B＝(e)
C＝(a，(b，c，d))
D＝(A，A，())
E＝(A，B，C)
F＝(a，F)

4.2.2 广义表的存储表示

1. 广义表的头尾链表存储方式

根据种类不同，广义表中的元素可以分为两类：原子结点和表结点。

- **表结点**可由三个域组成：**标识域、指示头结点的指针域和指示尾节点的指针域**；
- **原子结点**只需两个域：**标识域和值域**。

标志 tag=1	表头指针 hp	表尾指针 tp

(a) 表结点

标志 tag=0	原子的值

(b) 原子结点

算法4.2：广义表头尾链表存储表示方式的结点类

```
struct GenListNode { //广义表结点类定义
public:
    GenListNode() : utype(0), tlink(NULL), info.ref(0) {} //构造函数
    GenListNode(GenListNode<T>& RL) { //复制构造函数
        utype = RL.utype; info = RL.info;
    }
private:
    int utype; //＝0 / 1
    union { //联合
        T value; //utype=1,存放数值
        struct{
            GenListNode<T> *hp;
            GenListNode<T> *tp;
        }ptr;
    } info;
};
```

A＝()
B＝(e)
C＝(a，(b，c，d))
D＝(A，A，())
E＝(A，B，C)
F＝(a，F)

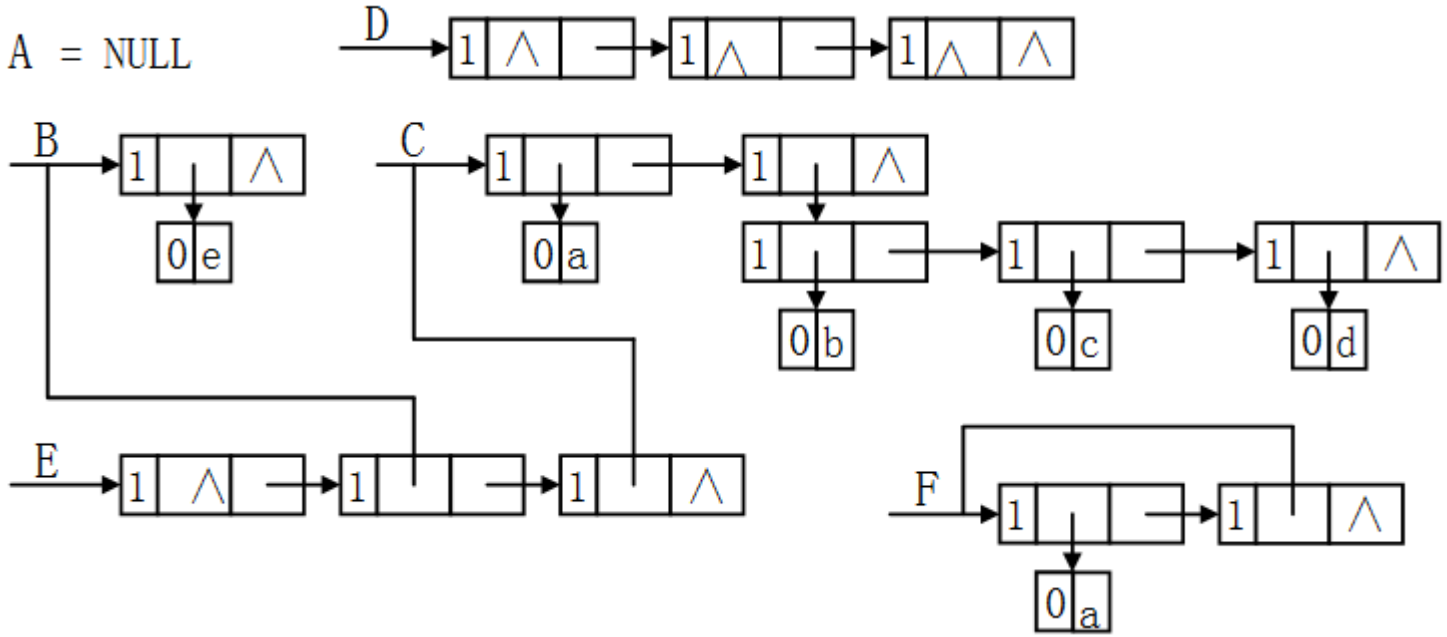


图4-9 广义表的存储结构示例

在这种存储结构中有几种情况：

（1）除空表的头指针为空外，对任何非空列表，其头指针均指向广义表表结点，且该结点中的hp指向表头（或为原子结点，或为表结点），tp指向表尾（除非表尾为空，否则必为表结点）；

（2）容易分清列表中原子和子表所在的层次。如在列表D中，原子a和e在同一层次上，而b、c和d在同一层次上且比a和e低一层，B和C是同一层次的子表；

（3）最高层的表结点个数为列表的长度。

以上三个特点在某种程度上给列表的操作带来了方便。

其**主要缺点**是：表结点过多，容易造成空间浪费。

utype = 0/1/2	ref/value/hlink	tlink
结点种类标志	信息	尾指针
（表头/元素/子表）	（引用数/元素值/头指针）	

图4-10 广义表扩展线性链表存储方式结点构造

2. 广义表的扩展线性链表存储方式

在这种存储结构中，**表结点由三个域组成**：

（1）标志域utype

它用来标明该结点是什么类型的结点。=0，是广义表专用的附加头结点；=1，是原子结点（为简化讨论，不考虑原子结点数据的不同类型）；=2，是子表结点。

（2）信息域info

不同类型的结点在这个域中存放的内容不同。当utype=0时，该信息域存放引用计数（ref）；当utype=1时，该信息域存放元素数据值（value）；当utype=2时，该信息域存放指向子表表头的指针（hlink）。

（3）尾指针域tlink

当utype=0时，该指针域存放指向该表表头元素结点的指针；

当utype≠0时，该指针域存放同一层下一个表结点的地址。

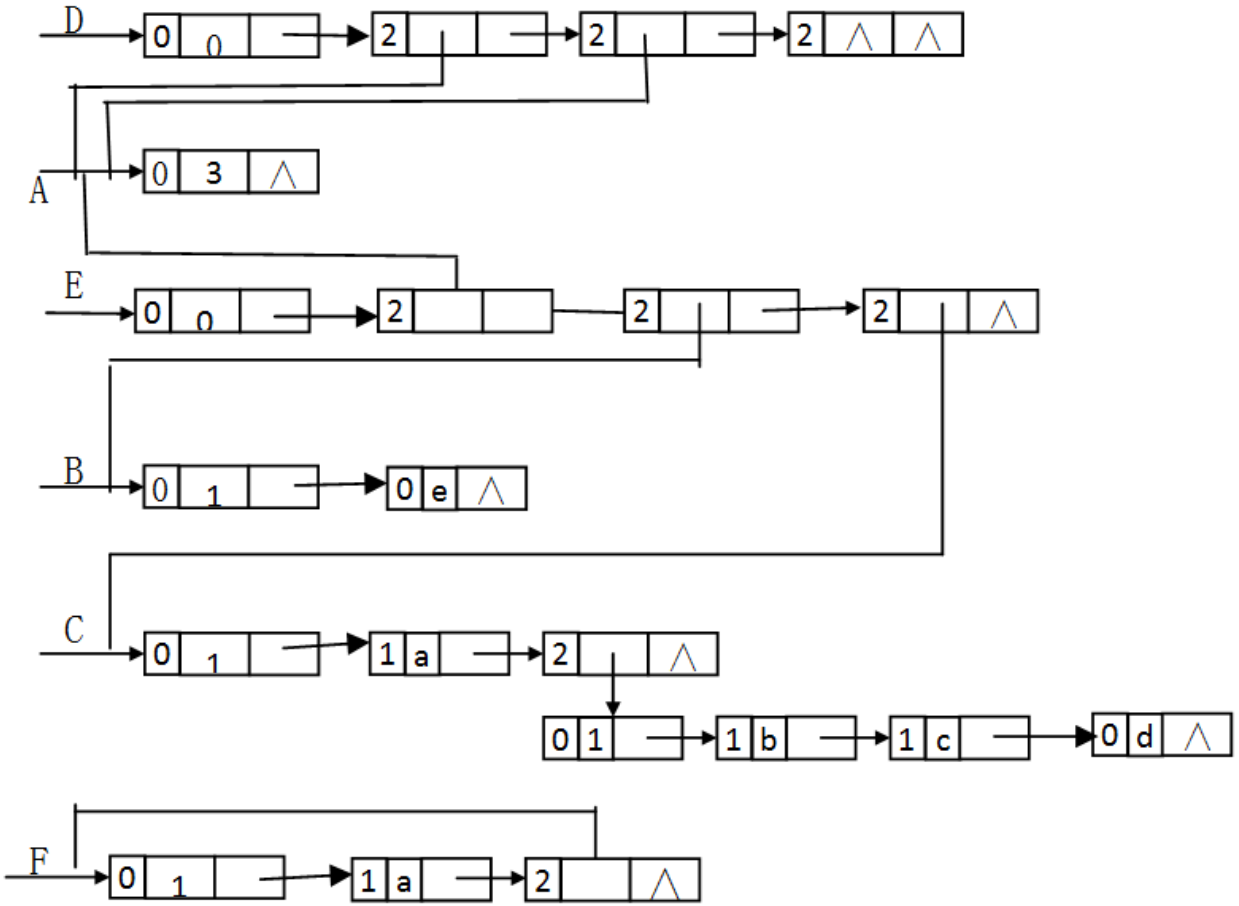


图4-11 带附加头结点的广义表存储表示

存储表有几个特点：

- **广义表中的所有子表，不论是哪一层的子表，都带有一个附加头结点，空表也不例外。**其优点是便于操作。特别是在共享表的情形，如果想要删除表中第一个元素所在结点，且表中不带附加头结点的话，必须检测所有的子表结点，逐一修改可能的指向被删结点的指针。这样修改工作量极大，很容易发生遗漏现象。如果所有子表都带有附加头结点，在删除表中第一个表元素所在结点时，不用修改任何指向该子表的指针。
- **表中结点的层次分明。**所有位于同一层的表元素，在其存储表示中也在同一层。
- **最高一层的表结点个数**(除附加头结点外)即为表的长度。

4.2.3 广义表的递归操作

广义表的主要操作是**取表头(GetHead)和取表尾(GetTail)**。

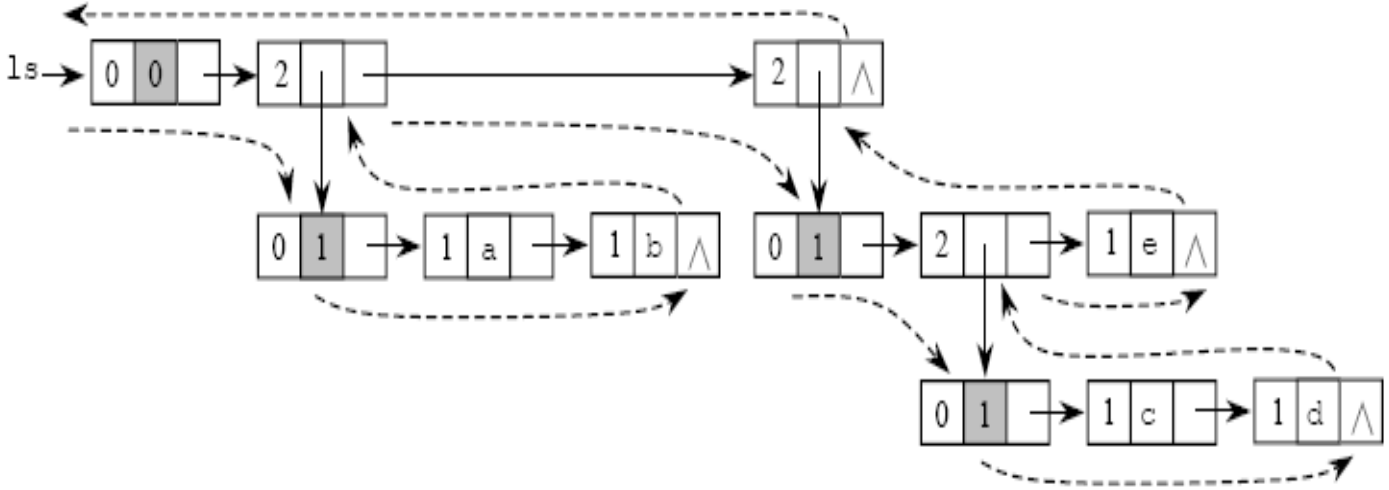
（1）**广义表的复制算法**

任何一个非空的广义表均可分为表头、表尾两个部分。因此，“一对”确定的表头和表尾可唯一地确定一个广义表。这样，复制一个广义表时，只要分别复制它的表头和表尾，然后合成就可以了。其前提是复制和被复制的广义表存在且不是共享表或递归表。

算法4.5：广义表的复制算法

```
template <class T>
void GenList<T>::Copy(const GenList<T>& R) {     //公有函数
    first = Copy(R.first);          //调用私有函数
};
template <class T>
GenListNode<T>* GenList<T>::Copy(GenListNode<T> *ls) {
    //私有函数，复制一个ls 指示的无共享子表的非递归表
    GenListNode<T> *q = NULL;
    if (ls != NULL) {
        q = new GenListNode<T>;       //处理当前的结点q
        q->utype = ls->utype;                    //复制结点类型
        switch (ls->utype) {              //根据utype 传送信息
            case 0: q->info.ref = ls->info.ref; break;   //附加头结点
            case 1: q->info.value = ls->info.value; break;          //原子结点
            case 2: q->info.hlink = Copy(ls->info.hlink); break;  //表结点
        }
        q->tlink = Copy(ls->tlink);                  //处理同一层下一结点为头的表
    }
    return q;
}
```

算法有三层考虑。首先，如果被复制结点为空，表明被复制广义表为空表，返回为空；其次，被复制结点非空，处理该结点的复制；最后，复制广义表中位于该结点之后的结点。考虑此算法的计算时间。对于空表，所花费的时间为常数值。下面分析非空表。考虑表ls = ((a, b),((c, d),e))，设其中的a,b,c,d,e是字符型数据。



（2）求广义表的长度

- 在广义表中，同一层次的每个结点是通过tlink指针链接起来的，所以可以把它看作为是由tlink链接起来的单链表。
- 求广义表的长度就是求单链表的长度
- 由于单链表的结构也是一种递归结构，即每个结点的指针域均指向一个单链表（称为该结点的后继单链表），它所指向的结点为该单链表的第一个结点（即表头结点），所以求单链表的长度也可以采用递归算法，即若单链表非空的话，其长度等于1加上表头结点的后继单链表长度，若单链表为空，则长度为0，这是递归的终止条件。

算法4.6：求广义表长度的算法

```
template <class T>
int GenList<T>::Length() {
    //共有函数，求当前广义表的长度
    return Length(first);
}
template <class T>
int GenList<T>::Length(GenListNode<T> *ls) {
    //私有函数，求以ls 为头指针的广义表的长度
    if (ls != NULL) return 1+Length(ls->tlink);
    else   return 0;
}
```

（3）求广义表的深度

- 广义表的深度定义为广义表中括号的重数。设非空广义表为LS=($\alpha_0,\alpha_1,\alpha_2,...,\alpha_{n-1}$)，其中，每个αi（0≤i≤n-1）或者是原子，或者是子表。这样，求LS的深度可分解为n个子问题，每个子问题为求αi 的深度。
- 若αi 是原子，则αi 的深度为0（没有括号）；若αi 是子表，则可继续对αi 进行分解、求解。而LS的深度为各的深度的最大值加1。空表也是广义表，其深度为1。由此可知，求广义表深度的递归过程有两个递归结束条件：原子和空表。只要能够求得各个αi 的深度，就能求得广义表LS的深度。因此，求广义表LS=($\alpha_0,\alpha_1,\alpha_2,\ldots,\alpha_{n-1}$)深度Depth(LS)的递归定义为：

$$\text{Depth(LS)}=\begin{cases}1 & \text{当 LS 为空表}\\ 0 & \text{当 LS 为原子}\\ 1+\max\limits_{1\leqslant i\leqslant n-1}\{\text{Depth}(\alpha_i)\} & \text{其他，}n\geqslant 1\end{cases}$$

算法4.7：求广义表深度的算法

```
template <class T>
    int GenList<T>::depth() {  //公有函数：计算一个非递归表的深度
    return depth(first);
}
template <class T>
int GenList<T>::depth(GenListNode<T> *ls) {       //私有函数：计算非递归广义表深度
    if (ls->tlink == NULL) return 1;          // ls->tlink ==NULL, 空表，深度为1
    GenListNode<T> *temp = ls->tlink; int m = 0, n;
    while (temp != NULL) {                    // temp 在广义表顶层横扫
        if (temp->utype == 2) {               //扫描到的结点utype 为表结点时,
            n = depth(temp->info.hlink); //计算以该结点为头的广义表深度
            if (m < n) m = n;       //取最大深度
        }
        temp = temp->tlink;
    }
    return m+1; //返回深度
};
```

广义表D(B(a,b),C(u,(x,y,z)),A())，链表结构：