第 4 章 串、数组和广义表

4.1 串

串（String）是一种特殊的线性表，其特殊性体现在其元素值的类型：每个元素是一个字符。这种特殊性使得其存储结构和运算与线性表存在一定的差异。

4.1.1 串的定义和运算

串（String），或字符串，是由有限个字符 $a_1,a_2,a_3,\ldots,a_n$ 组成的序列，记作 S=”$a_1a_2a_3\ldots a_n$”。

其中 S 称作**串名**，等号右边为**串值**，元素个数 n（n≥0）称为串的**长度**，当 n 为 0 时，称串 S 为**空串**。称 S 中的一个连续段所组成的串为 S 的**子串**。

例如，设串 S1=“abcdefghijk”, S2=“cdef”，S3=“abc123”，则 S1 的长度为 11，S2 的长度为 4，S3 的长度为 6。S2 为 S1 的子串，但 S3 不是 S1 的子串。

对串通常有如下基本运算：

（1）赋值运算（S=S1）：将一个串（值）S1 传送给一个串名 S。

（2）求长度运算 strLength(S)：返回串 S 的长度值。

（3）连接运算(S1+S2)：将串 S1 和 S2 连接成为一个新串。

（4）求子串函数 substr(S,i,j)：返回串 S 中从第 i 个元素开始的 j 个元素所组成的子串。

（5）串比较：比较两个串的大小。此处所谓比较两个串的大小，是指在左对齐的情况下，按两个串的对应位字符的 ASCII 码之间大小的比较。如前例中，S1<S2, 这是因为串 S1 中的第一个字符‘a’小于串 S2 中的第一个字符‘c’。同理，S1>S3，这是因为两者前 3 个字符相同，但串 S1 中的第四个字符‘d’大于串 S2 中的第四个字符‘1’。这一运算可有两种形式的返回结果：

其一是仅比较是否相等，因此可采用函数 equal(S1,S2)返回 0 与 1 或 FALSE 与 TRUE

（分别表示相等关系的不成立和成立）的形式。

其二是不仅要能判断是否相等，还要能区分大小，因此可采用 strcmp(S1,S2)返回-1、0、1（分别表示 S1<S2、S1=S2 和 S1>S2）的形式。

上述两种形式都可能会在某种情况下被采用。

除了上述 5 个基本运算外，还有插入和删除这两个常用运算：

（6）插入运算 Insert(S,i,S1)：将子串 S1 插入到串 S 的从第 i 字符开始的位置上。

（7）删除运算 delete(S,i,j)：删除串 S 中从第 i 个字符开始的 j 个字符。

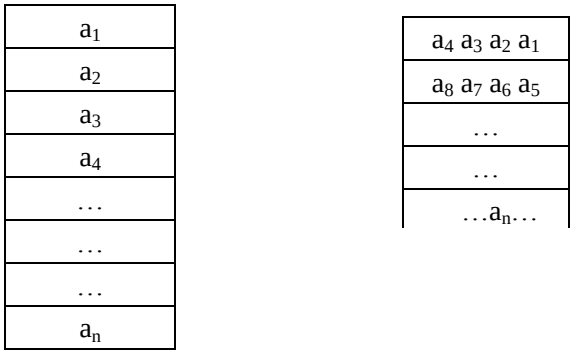
串还有一种比较重要的运算 — 串的模式匹配，限于篇幅，本书不做介绍。感兴趣的同学请参看其它资料。

4.1.2 串的存储

由于串是特殊的线性表，故可采用线性表的存储结构形式，即顺序存储形式和链式存储形式，由此而得到顺序串和链串两种存储结构，讨论如下：

1．顺序串

显然，这一存储形式类似于顺序表结构，由连续的存储空间及指示大小的变量组成，如图 4-1(a)所示。然而，由于串中的每个字符仅占用 1 个字节的存储空间，而许多计算机系统中的每个内存单元的大小可能包含多个字节，因而造成存储空间的浪费，故可采用压缩的方法来存储，将每个内存单元中尽可能多地存放字符，由此而得到紧凑格式（或称为紧缩格式）的顺序串，如图 4-1(b)所示。



(a) 非紧凑格式顺序串示意图　　(b)紧凑格式顺序串示意图

图 4-1　顺序串存储形式示意图

显然，紧凑格式的顺序串能节省存储空间，但运算不便，例如，如果删除其中的某个字符，则可能需要将该单元后面的字符（如果被删除字符不是该单元中所存储的最后字符的话）以及该单元之后的各单元的字符往前移，因而操作较麻烦。

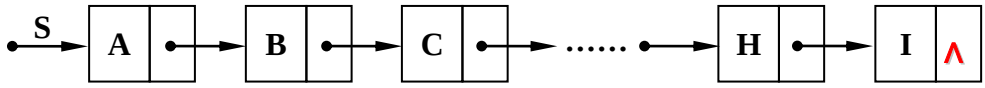
与紧缩格式相反的是，非紧凑格式的顺序串较浪费存储空间，但是运算要方便得多。

另外，无论是紧凑还是非紧凑格式，对插入和删除运算来说，都需要移动元素。运用前面顺序表插入和删除运算的分析方法可知，插入和删除一个元素平均需要移动一半的元素，因而不便于规模较大的串的存储。这样就需要采用链串来存储。

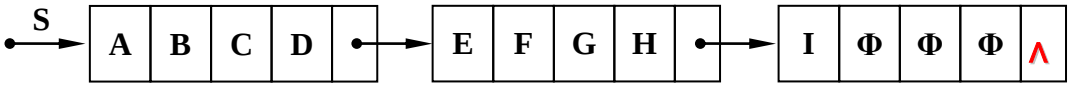
2．链串

为便于插入和删除运算的实现，需要采用链表结构来存储串。在前面讨论线性表的存储结构时，我们用链表中的每个结点存储一个元素，然而，这一方法显然不适于串的存储，因为一个结点中的指针所需要的存储空间通常要多于一个字节，例如 32 位机器是 4 个字节，由此造成存储空间的有效利用率低。为此，链串常采用**块链**结构，一个结点中可存放多个字

符。在这种情况下，将每个结点中最多能存储的元素个数定义为**结点大小**。在一个结点存放多个字符时，最后一个结点可能剩余一些空位置。如图 4-2 所示，图（a）表示结点大小为 1 的链串；图（b）表示结点大小为 4 的链串：



（a）结点大小为 1 的链串



（b）结点大小为 4 的链串

图4-2 链串结构示意图

显然，结点大小大于 1 的链串能节省存储空间，但运算不便，而结点大小为 1 的链串则较浪费存储空间，但是运算要方便得多。

【例 4.1】设计算法对两个结点大小为 1 的链串 S1 和 S2 实现运算 strcmp(S1,S2)，并根据不同情况分别返回－1、0、1（分别表示 S1<S2、S1=S2 和 S1>S2）。

【分析】因为结点大小为 1，可采用普通的单链表结构实现，既可以带头结点，也可以不带头结点，本题采用带头结点的单链表实现。使用两个指针 p1、p2 分别指向串 S1 和 S2 的结点，然后逐结点比较字符大小（ASCII 值大小），并根据不同情况作相应的处理：

如果 p1 和 p2 指示的字符不同，若 p1->data<p2->data，返回-1；否则，返回 1。

p1 和 p2 指示的字符相同，p1、p2 同时后移一个结点，继续比较。

当比较到有一个串的结尾时，比较操作结束，此时也要根据不同情况作相应的处理。若 S1 和 S2 同时结束，则 S1=S2，返回 0；S1 结束，S2 未结束，则 S1<S2，返回-1；S1 未结束，S2 结束，则 S1>S2，返回 1。

【算法描述】

```
int strcmp( node* S1,   node* S2 )
{
        node *p1=S1->next, *p2=S2->next;      //指向 2 串的第一个字符（结点）
        while( p1!=NULL && p2!=NULL )      //两串皆未结束
        {
                if(p1->data==p2->data)
                {
                        p1=p1->next;       //对应字符相同，p1、p2 同时后移一个字
符
                        p2=p2->next;
                }
                else if( p1->data<p2->data )
                        return -1;       //S1<S2
                else
                        return 1;        //S1>S2
        }
          //有一个串结束或两个串都结束的处理
        if(p1==NULL)     //S1 结束
```

```
    {
        if(p2!=NULL)
            return -1;     //S1 结束，S2 未结束
        else
            return 0;      //2 串同时结束，为相等的情况
    }
    else     //此为 S1 未结束，但 S2 已经结束，所以 S1>S2
        return 1;
}
```

串的其它基本运算请大家自行实现。

4.2 数组

这一节所讨论的数组，可以看成是前面所介绍的线性表的推广，其元素本身也是一个数据结构。数组是软件设计中应用最多的结构，在工程领域中有广泛的应用，由此而引出了一些特殊形式的数组形式。下面先讨论数组的基本结构形式，然后讨论有关特殊形式矩阵的基本内容。

4.2.1 数组的定义和运算

数组是计算机程序设计语言中常见的一种类型，几乎所有的高级程序设计语言中都有数组类型。

定义：一维数组是有限个具有相同类型的变量组成的序列。若其中每个变量本身是一维数组，则构成**二维数组**，类似地，若每个变量本身为(n-1)维数组，则构成**n 维数组**。

例如，图 4-3 是一维数组的示意图，其中共有 n 个元素。

在一维数组中，每个元素对应一个下标以标识该元素。例如，图 4-3 中一维数组的第一个元素 a_1 的下标为 1。

图 4-4 为二维数组的示意图，共有 m×n 个元素，分布于 m 行、n 列中，每个元素属于其中的某一行、某一列。若将其中的每行当作一个元素，则此二维数组也可看作是由 m 个元素组成的一维数组，只不过其元素本身是一个一维数组。与一维数组类似，在二维数组中，每个元素对应两个方向的下标以标识该元素。例如，图 4-4 中二维数组的第二行第三列的元素 a_{23} 的下标有两个，分别为 2 和 3。

（a_1，a_2，a_3，…，a_n）

图 4-3 一维数组示意图

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{i1} & a_{i2} & a_{i3} & \cdots & a_{in} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{pmatrix}$$

图 4-4 二维数组示意图

类似地，在 n 维数组中，每个元素对应 n 个方向的下标以标识该元素。

由于一维数组的线性关系，因此，一维数组中的每个元素最多有一个直接前趋和一个直接后继。而在二维数组中，每个元素分别属于两个向量(即行向量和列向量)，因此，每个元素最多有两个直接前趋和两个直接后继。类似地，在 n 维数组中，每个元素最多有 n 个直接前趋和直接后继。

对数组的运算，通常有如下两个：

(1)给定一组下标，存取相应的数组元素；

(2)给定一组下标，修改相应的元素值。

由于这两个运算在内部实现时都需要计算出给定元素的实际存储地址，因此，计算数组元素地址这一运算就成了数组中最基本的运算，在采用特定的存储结构存储数组时，都需要能实现。

4.2.2 数组的顺序存储

由于数组一般没有插入和删除运算，因此，采用顺序结构是理想的。现在的问题是：以什么次序来存储各元素的值?

由于一维数组与计算机内存存储结构一致，因此存储起来比较方便。而在多维数组中，情况就要麻烦一些。一般有两种存储方式，下面以二维数组为例来说明。

① 以行序为主序的存储（即行优先次序）：逐行地顺序存储各元素，如图 4-5 所示。在 C，Java，C#，PASCAL，COBOL，PL/1 等语言中均采用这种存储方式。

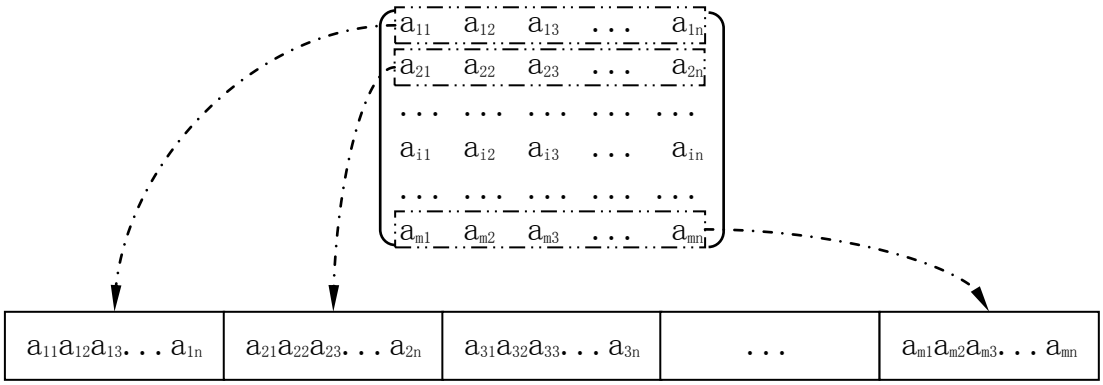


图 4-5 二维数组行优先存储示意图

② 以列序为主序的存储（即列优先次序）：逐列地顺序存储各元素，如图 4-6 所示。FORTRAN 语言中采用的就是这种方法。

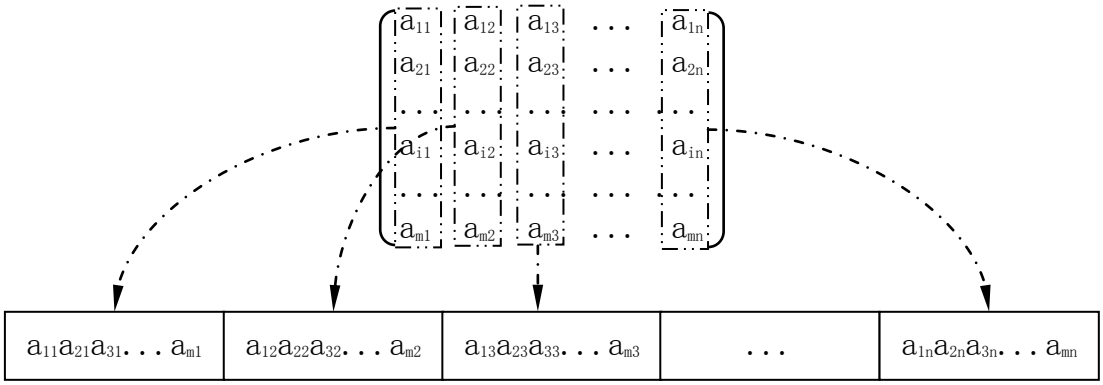


图 4-6 二维数组列优先存储示意图

由前述可知，在实现数组的运算中，涉及到求解给定元素的地址这样的问题。由于顺序存储方式的规律性，较易实现数组元素的地址的求解，简要讨论如下：

对给定的二维数组的元素 A [i,j]，在以行序为主序的存储方式中，该元素的序号为 Num(i,j)=(i-1)*n+j，而在以列序为主序的存储方式中，序号为 Num(i,j)=(j-1)*m+i。

若给定存储区的起始地址为 Addr0,每个元素占 C 个单元，则元素 A [i,j] 在内存中的地址为 Loc(i,j)=Addr0+(Num(i,j)-1)*c。

需要说明的是，此处所给出的数组的行、列下标是按序号从 1 开始的，然而，在 C、C++、Java 等语言中，数组下标是从 0 开始的，故计算公式中要略为有所变动。例如，如果行列数不变，但均是从 0 开始，则采用行优先时的序号计算公式变成 Num(i,j)=i*n+j+1。

可以将此地址计算公式推广到多维数组。为此，首先要知道相应数组分别在行优先和列优先时的元素下标的变化规律。先从二维数组存储时的下标变化规律开始：

在二维数组以行序为主序的存储方式中，列下标变化速度最快。而在以列序为主序的存储方式中，行下标变化速度最快。同理，在 n 维数组中，两种存储方式下的下标变化速度也具有这样的规律：在以行序为主序存储时，右边的下标比左边的下标变化快，其变化就象数字电表的各位数字进位那样，当低位满了，就要向前一位进位。在以列序为主序存储时，左边下标比右边下标变化快。

例如，对三维数组 A[1..3,1..3,1..3]，其行优先和列优先存储时的元素序列分别如下：

行优先：a_{111}, a_{112}, a_{113}, a_{121}, a_{122}, a_{123}, a_{131}, a_{132}, a_{133}, a_{211}, a_{212}, a_{213}, a_{221}, a_{222}, a_{223}, a_{231}, a_{232}, a_{233}, a_{311}, a_{312}, a_{313}, a_{321}, a_{322}, a_{323}, a_{331}, a_{332}, a_{333}

列优先：a_{111}, a_{211}, a_{311}, a_{121}, a_{221}, a_{321}, a_{131}, a_{231}, a_{331}, a_{112}, a_{212}, a_{312}, a_{122}, a_{222}, a_{322}, a_{132}, a_{232}, a_{332}, a_{113}, a_{213}, a_{313}, a_{123}, a_{223}, a_{323}, a_{133}, a_{233}, a_{333}

4.2.3 矩阵的压缩存储

矩阵是许多科学、工程中研究和应用的数学对象。在实际应用中经常会用到一些阶数较高的矩阵，因而要占用较大的存储空间。然而，许多所涉及到的矩阵中有较多的元素的值为 0，称这种矩阵为**稀疏矩阵**。另外，还有一些矩阵的元素值的分布有一定规律，称这类矩阵为**特殊矩阵**。为节省存储空间，可以对此类矩阵采用压缩方式来存储。此处所谓压缩是指：在不影响完整性的前提下，用更少的存储空间存储其元素。下面分别讨论这两类矩阵的压缩存储。

1．特殊矩阵的压缩存储

如前所述，所谓特殊矩阵就是元素值的分布有一定规律的矩阵。下面仅给出对称矩阵、

三角矩阵和对角矩阵等的压缩存储方法。

（1）对称矩阵和三角矩阵

若矩阵 $A_{n\times n}$ 满足 $a_{ij}=a_{ji}(1\leqslant i,j\leqslant n)$，则称 A 为**对称矩阵**。

由于对称矩阵关于对角线对称，因此，知道其对角线以下或以上的各元素的值，就能知道其另外的元素，所以我们可以考虑只存储下三角或上三角(包括对角线)部分的元素，另一部分不必存储，从而实现了压缩存储。不失一般性，我们按以行序为主序方式存储矩阵的下三角部分(共 n(n+1)/2 个元素)到数组 SA［1..n(n+1)/2］中。如图 4-7 所示。

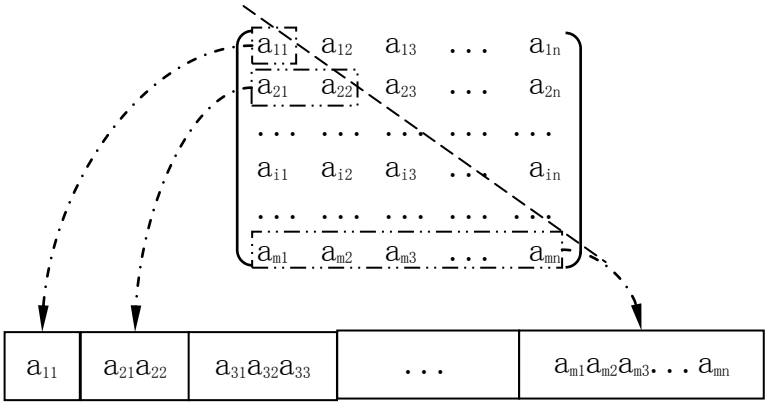


图 4-7 对称矩阵的行优先存储示意图

这样，对给定的下三角元素 a_{ij}，其在 SA 中的序号 num(i,j)可由下式来确定：

num(i,j)=1+2+3+…+(i-1)+j=i(i-1)/2+j　　(i>=j)

若元素 a_{ij} 是上三角部分的元素，即 i<j,则其序号的确定显然就是将行列互换，即计算公式为 num(i,j)=1+2+3+…+(j-1)+i=j(j-1)/2+i。

同样需要说明的是，此处行列下标均是从 1 开始，如果限定为用 C 语言中的矩阵，则下标要从 0 开始，故需要调整。相信读者不难理解，故不多述。

这种方法同样适用于三角矩阵。所谓**三角矩阵**是指对角线以上或以下的元素全为 0，或全为同一值。即元素 a_{ij} 的序号计算公式为 num(i,j)=1+2+3+…+(j-1)+i=j(j-1)/2+i (i>=j)

（2）对角矩阵

所谓**对角矩阵**，是指除了主对角线和紧靠主对角线的上下若干条对角线外，其余元素全为 0。对此，也可按某种方式来存储，如以逐行、逐列或以对角线的顺序将这几个对角上的元素存储到一维数组上，在此不妨讨论三对角矩阵的行优先方式的存储。此处所谓三对角矩阵，是指除了主对角线及其上下一条对角线上有非 0 元外，其余位置均为 0 的矩阵。其存储形式如图 4-8 所示。

在这种存储方式下，元素 a_{ij} 在矩阵中的序号 num(i,j)的计算公式：除了第一行和最后一行外，每行均存储 3 个元素，因此，其计算公式中应考虑到其前面各行中的元素个数与其在本行中的序号这两个方面，讨论过程从略，计算公式如下：

num(i,j)=[3(i-1)-1]+[j-i+2]=2i+j-2　　|i-j|<=1

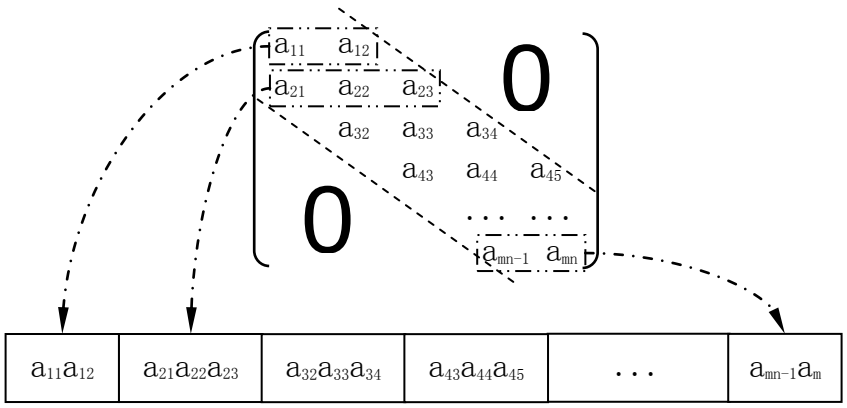


图 4-8 三对角矩阵的行优先存储示意图

4.2.3.2 稀疏矩阵的压缩存储

当数组中非零元素个数非常少时(这是一个模糊概念，一般只是凭直觉来判断)，称之为**稀疏矩阵**。在对稀疏矩阵进行压缩存储时，除了要存储非零元素的值 v 之外，还要存储其行列号 i 和 j，故每个非零元素对应一个三元组(i,j,v)。因此，整个稀疏矩阵的压缩存储可通过存储这些三元组来实现。如果将这些三元组集合以线性表的形式组织起来，则可构成**三元组表**。考虑到与矩阵的对应关系，需要在三元组表中增设元素个数、行列数以唯一确定一个稀疏矩阵。如图 4-9 为一稀疏矩阵和对应的三元组表。

$$\begin{pmatrix} 0 & 12 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 5 & 0 & 0 & 3 & 0 \\ 7 & 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

（a）稀疏矩阵

行号	列号	值
1	2	15
1	6	5
2	4	6
3	2	5
3	5	3
4	1	7
4	4	10
6	3	9

矩阵行数：7
矩阵列数：6
非 0 元素个数：8

（b）三元组表

图 4-9 稀疏矩阵及其三元组表表示示例

这种存储结构需要分两步进行描述，首先定义三元素结构，然后再定义整体表结构。

【三元组结构描述】

```
typedef struct
{
        int i;   //行号
        int j;   //列号
        elementtype v;   //元素值
} tuple;
```

【表结构描述】

```
typedef struct    //三元组表结构瀠
{
    int mu;       //行数
    int nu;       //列数
    int tu;       //非 0 元个数
    tuple data [MAXLEN];  //存放三元组的数组
} spmatrix
```

其中 MAXLEN 为一个常量。

在这种存储结构上，可实现对矩阵的转置等运算。由于篇幅所限，此处不再详细介绍。

4.3 广义表

4.3.1 广义表的基本概念

广义表作为线性表的推广，在软件设计中有重要的作用。

定义：**广义表** L 是 n 个元素 $a_1,a_2,\ldots,a_n$ 组成的有限序列，记作 $L=(a_1,a_2,\ldots,a_n)$，其中每个元素 a_i 可以是不可分割的**原子**，也可以是**广义表**（称为**子表**）。另外，称元素个数 n>=0 为**表长度**，当 n=0 时为**空表**，记作 L=（）。

约定：在书写时，一般用小写字母表示原子，用大写字母表示广义表。

由定义可知，广义表是线性表的推广，然而，两者有明显的不同：线性表中每个元素的类型相同，而广义表中每个元素可以是原子又可以是广义表。

表 4-1 所示为一些广义表的实例。

表 4-1 广义表实例

广义表	说明
A=(a,b,c)	表 A 有 3 个元素，每个元素都是原子
B=(a,b,(a,d))	表 B 有 3 个元素，第 1 和第 2 个元素为原子，第 2 个元素是子表
C=(A,B)	表 C 有 2 个元素，都是广义表（子表）
D=(d,D)	表 D 有 2 个元素，分表是原子和子表，且子表为 D 自己
E=()	表 E 为空表，没有元素
F=((a,b,c),(),(d,e,f))	表 F 有 3 个元素，皆为子表，其中第二个元素为空表

可以用图的形式来直观地描述广义表，具体方法如下：

① 用一个“点”代表一个元素，即广义表、子表或原子。

② 用箭头指示一个表的所有元素，并在“点”附近标注元素信息。

表 4-1 中各广义表的图表示如图 4-10 所示。

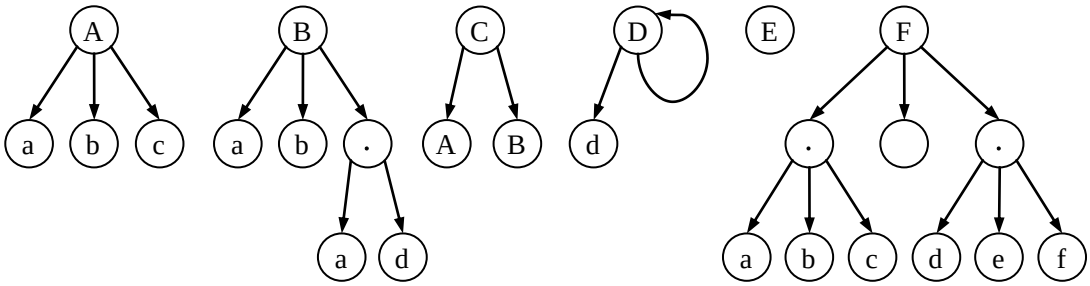


图 4-10 广义表图示法示例

4.3.2 广义表的基本运算

针对广义表可以定义多个运算，其中两个最基本的运算是**取表头**和**取表尾**运算。

① 取表头：可用函数 head(L)表示，其结果是返回广义表 L 的第一个元素。

② 取表尾：可用函数 tail(L)表示，其结果是返回广义表 L 中除去第一个元素，剩下元素构成的子表。

例如，对表 4-1 中的各个广义表进行取表头和取表尾运算，结果如表 4-2 所示。

表 4-2 对表 4-1 各广义表取表头和取表尾运算的结果

取表头	取表尾
head(A)=a	tail(A)=(b, c)
head(B)=a	tail(B)=(b, (a, d))
head(C)=A	tail(C)=(B)
head(D)=d	tail(D)=(D)
head(E)无解	tail(E)无解
head(F)=(a, b, c)	tail(F)=(( ),(d, e, f))

在此特别提醒读者注意以下两点：

① 广义表中括号用来区分元素还是广义表，是数据的一部分，不能随意增减。例如广义表(a, b)和((a, b))不是同一个广义表。

② 许多初学者对取表尾运算的理解有问题。例如，单纯从字面意义上可能错误地将取表尾运算理解为“取广义表最后一个元素”，这是不对的，应该注意。事实上，初学者可以这样来理解和记忆取表尾运算：取表尾运算就是“去表头”运算。

除了取表头和取表尾运算外，有时可能还需要其它一些运算，如“取出广义表中第 3 个元素”。如何实现这类运算？是否需要另外定义相应的运算呢？

逻辑上说，这些运算不需要另外定义新的运算函数，通过多次调用取表头和取表尾的复合运算即可实现这些功能。

例如，取出广义表 L 中第 3 个元素可以这样进行：

调用 tail(L)去掉表中第 1 个元素；对返回的子表再次调用取表尾运算，即 tail(tail(L))，去掉原表中第 2 个元素，返回的子表的表头即为原表的第 3 个元素；最后，调用取表头运算即可取出原表的第 3 个元素，即执行 head(tail(tail(L)))。显然，这个嵌套调用的次序是由内向外的。

【例 4.2】对广义表 L=(a, (b, (c, d), e))，写出运算 head(tail(head(tail(L))))的运算结果。

【解】最先执行是最内层的运算 tail(L)，返回结果是从 L 中去掉表头元素 a 后剩下部分元素形成的子表，不妨设为 L_1，即 L_1=((b, (c, d), e))。还要执行的运算为：head(tail(head(L_1)))。

第 2 步执行的运算是 head(L_1)，L_1 只有唯一的元素，所以此运算相当于去掉外层括号，取出的元素仍为一个子表，不妨设为 L_2，则 L_2=(b, (c, d), e)。还要执行的运算为：head(tail(L_2))。

第 3 步执行 tail(L_2)，去掉 L_2 的表头元素，返回的子表设为 L_3，则 L_3=((c, d), e)。还要执行的运算为：head(L_3)。

第 4 步执行 head(L_3)，返回 L_3 的表头元素(c, d)，这就是最终结果。

【思考问题】

① 如果要取出广义表 L 中第 3 个和第 4 个元素，应怎样构造复合函数？

② 对广义表 L=(a, (b, (c, d), e))，写出运算 head(tail(tail(head(tail(L)))))的运算结果。

4.3.3 广义表的存储

广义表的存储结构显然要比线性表复杂得多，原因是每个元素的类型可能是原子，也可能是一个子表。为此，需要为每个元素设置区分标志，以区分原子和子表。广义表的存储形式有多种，下面介绍两种最简单的存储形式。一种简单的存储形式是**单链表存储法**，其方法如下：

① 从总体结构上看，为广义表中的每个元素设置一个结点，并将这些结点按元素在表中的先后次序连接起来（从这一点上看就好像是一个单链表，故有此名）。

② 对表中每个结点这样设计：除了有一个指向下一个元素的后继指针（不妨设为 next）外，还要有 2 个字段：其一是区分元素还是子表的标志（不妨设为 tag，并以取 0 表示元素是原子，取 1 表示元素是子表）；其二是具体的值（元素是原子时）或指向子表的指针（当元素是子表时）。

例如，广义表 L=(a, (b, (c , d), e))的单链表存储结构如图 4-11 所示。

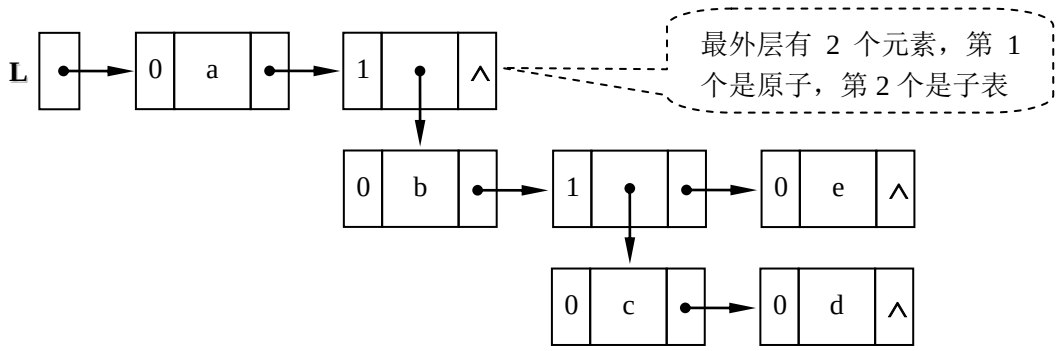


图 4-11 广义表的单链表存储结构示意图

这种表示方法的优点是比较直观，各结点的深度（指作为子表的深度）与元素在表中的括号层次数相对应，因而容易设计出运算的递归算法。例如，原子 a 的深度是 1，则在存储结构中是第 1 层，而原子 c 的深度为 3，则它位于存储结构中的第 3 层。

然而，这种存储结构也有其不足：由于表中的某个元素可能是被多处引用或指示的另外的子表，因此，当对该子表做诸如插入和删除运算时，容易造成不一致的问题。

为此，一种改进的方法是为每个子表设置一个类似于线性表中头结点的**表结点**，以取代整个子表的出现，由表结点中设置一个指针指示其具体结构。这样，对应于原子的结点可设置一个**原子结点**。显然，表结点和原子结点的结构可以不同。按照这一方法可得到前面所

述广义表 L=(a, (b, (c ,d), e))的存储结构如图 4-12 所示。

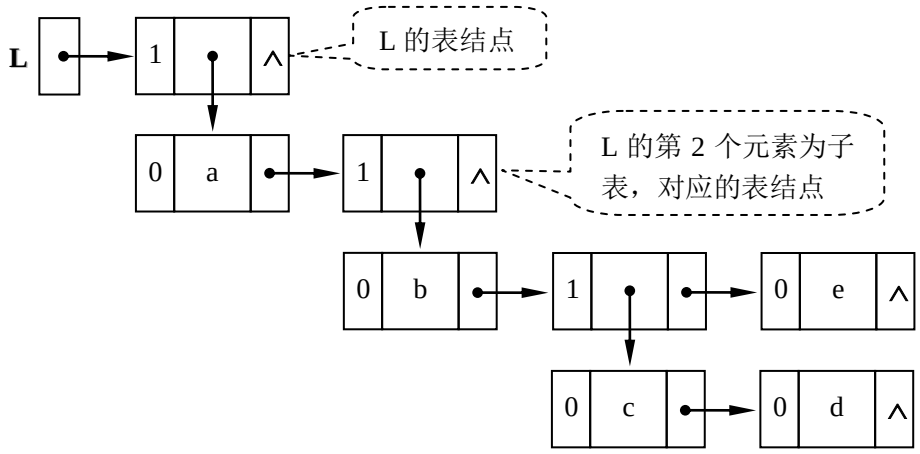


图 4-12 广义表的改进后的单链表存储结构示意图

这种表示方法的优点是便于保证运算的一致性，并且元素在结构中的层次与括号的深度一致。

上述只讨论了广义表的两种存储形式，但广义表还可有其它的形式，并且各有其特点，应根据实际需要选择。

本章小结

串是一种特殊的线性表，其运算也有较大的差异。由于每个元素仅是一个字符，使得其存储结构时与线性表又明显的不同。

数组是线性表的一种推广。数组的元素本身可能也是一个数组，因而得到多维数组。数组中每个元素的数据类型必须相同。数组的运算通常有存取元素和修改元素的值两种，并且都涉及到计算给定元素的地址的运算。由于不涉及到插入和删除元素的运算，故适于采用顺序存储方式。数组的顺序存储方式有行优先和列优先两种形式。

在实际应用中，特殊矩阵和稀疏矩阵是两种需要注意的矩阵。由于特殊矩阵的规律性，可采用压缩方式来存储以节省存储空间。由于稀疏矩阵的非 0 元个数较少，故可通过仅存储这些元素来实现压缩存储。

广义表是线性表另一种形式的推广，其中每个元素可能是不可分割的原子，也可能是子表。对广义表的基本运算有取表头和取表尾运算，通过这两种运算的组合可以得到其它许多种运算。广义表存储结构的设计比较麻烦。