第5章 数组与广义表

【本章学习目标】

多维数组是一种最简单的非线性结构，它的存储结构也很简单，绝大多数高级语言采用顺序存储方式表示数组，存放顺序有的是行优先，有的是列优先。在多维数组中，使用最多的是二维数组，它和科学计算中广泛出现的矩阵相对应。对于某些特殊的矩阵，用二维数组表示会浪费空间，本章介绍了它的压缩存储方法。对元素分布有一定规律的特殊矩阵，通常将其压缩存储到一维数组中，还可以利用该矩阵和二维数组间元素下标的对应关系，容易、直接地算出元素的存储地址。对于稀疏矩阵，通常采用三元组表或十字链表来存放元素。

广义表是一种复杂的非线性结构，是线性表的推广。这里简要介绍了它的概念、基本运算和存储结构。

通过本章的学习，要求：

- 理解数组中元素逻辑存储结构；
- 理解数组元素寻址公式的含义；
- 掌握特殊矩阵压缩存储方法；
- 掌握稀疏矩阵压缩存储表示下转置和相乘运算的实现；
- 掌握广义表的概念和它的定义方式、表头和表尾的定义；
- 掌握广义表的性质；
- 掌握广义表的存储结构；
- 掌握广义表的基本操作。

5.1 数组的定义和运算

数组的特点是每个数据元素可以又是一个线性表结构。因此，数组结构可以简单地定义如下。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，即为向量；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；以此类推，若二维数组中的元素又是一个一维数组结构，则称为三维数组。

结论：线性表结构是数组结构的一个特例，而数组结构又是线性表结构的扩展(见图5.1)。

$$A_{m\times n}=\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix}$$

图5.1 二维数组的逻辑结构

其中,A是数组结构的名称,整个数组元素可以看成是由 m 个行向量和 n 个列向量组成的,其元素总数为 $m \times n$。在C语言中,二维数组中的数据元素可以表示成a[表达式1][表达式2],表达式1和表达式2分别被称行下标表达式和列下标表达式,例如a[i][j]。

5.1.1 数组的定义

ADT Array {

数据对象:$j_i=0,\cdots,b_i-1,i=1,2,\cdots n$,

$D=\{a_{j_1 j_2 \cdots j_n} \mid n(>0)$称为数组的维度,$b_i$ 是数组第 i 维的长度,

j_i 是数组元素的第 i 维下标,$a_{j_1 j_2 \cdots j_n} \in \text{ElemSet}\}$

数据关系:$R=\{R_1,R_2,\cdots,R_n\}$

$R_i=\{<a_{j_1 \cdots j_i \cdots j_n},a_{j_1 \cdots j_i+1 \cdots j_n}> \mid$

$0 \leqslant j_k \leqslant b_k-1,1 \leqslant k \leqslant n$ 且 $k \neq i$,

$0 \leqslant j_i \leqslant b_i-2$,

$a_{j_1 \cdots j_i \cdots j_n},a_{j_1 \cdots j_i+1 \cdots j_n} \in D,i=2,\cdots,n\}$

} ADT Array

5.1.2 数组的基本运算

```
ADT Array {
InitArray(&A, n, bound1, …, boundn)
```

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, …, indexn)
```

初始条件:A是 n 维数组,e为元素变量,随后是 n 个下标值。

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, …, indexn)
```

初始条件:A是 n 维数组,e为元素变量,随后是 n 个下标值。

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
```

5.2 数组的顺序存储结构

直观上看,数组可看成一组数对,即下标和值。对每一个有定义的下标,都有一个与该下标相关联的值。一维数组的值对应一个下标,二维数组则对应两个,以此类推,n 维数组的值对应 n 个下标。

数组是一种“均匀”结构,即数组中每个元素必须具有同样的类型。数组又是一种随机存取结构,只要给定一组下标,就可以访问与这组下标相关联的值。就数组而言,我们所关

心的操作主要是值检索和值存储。

在内存中，数组元素是连续存储的。数组的第一个元素的地址称为基地址，也称为数组的起始地址。当用一组连续的存储单元存放多维数组的元素时会遇到次序问题。例如，对于二维数组 a[2][3]，如果每个元素占用 1 字节，那么分配给数组 a 的内存空间就有 6 字节，那么数组 a 的 6 个元素 a[0][0]、a[0][1]、a[0][2]、a[1][0]、a[1][1]和 a[1][2]在内存中如何排列呢？

5.2.1 行优先存储

一种方法是先放第一行的三个元素，再放第二行的三个元素。这种方式称为以行为主序的存储方式，按这种方法得到的排列次序如图 5.2 所示

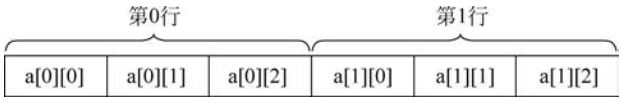

图 5.2 二维数组以行为主序的存储方式

5.2.2 列优先存储

另一种方法是先放第一列元素的两个元素，再放第二列的元素，最后放第三列的三个元素，这种方式称为以列为主序的存储方式，按这种方法得到的排列次序如图 5.3 所示。

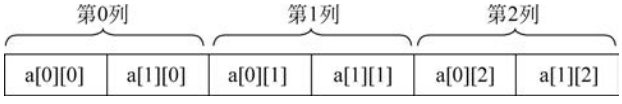

图 5.3 二维数组以列为主序的存储方式

如果数组 a[2][3]采用以行为主序的存储方式，并假定其基地址为 100，即元素 a[0][0]存储于单元 100 中，则 a[0][1]存储在单元 101 中，a[0][2]存储在单元 102 中，a[1][2]存储在单元 105 中。这个地址是很容易算出来的。其实，只要知道数组的维数，数组每个元素的地址可以很容易地通过数组的基地址得到。

对于一维数组 a[n]，假定每个元素占 1 字节，α 是数组的基地址。则元素 a[0]的地址是 α，元素 a[1]的地址是 $\alpha+1$。由于在第 $i+1$ 个元素之前有 i 个元素，因此任一元素 a[i]的地址是 $\alpha+i$，元素与其地址的关系如下：

数组元素	a[0]	a[1]	a[2]	…	a[i]	…	a[$n-1$]
地　　址	α	$\alpha+1$	$\alpha+2$	…	$\alpha+i$	…	$\alpha+n$

对于二维数组 a[m][n]，我们可以理解为 a 是一维数组，它有 m 个元素，分别是行 0、行 1……行 $m-1$，而每一行又由 n 个元素组成，如图 5.4 所示。

图 5.4 二维数组元素排列方式

若α是数组a的基地址,也就是其元素a[0][0]的地址,则第1行的第1个元素的地址为$\alpha+1\times n$,第2行的第1个元素的地址为$\alpha+2\times n$。因为在第$i+1$行的第一个元素之前有i行,每行有n个元素,所以第$i+1$行的第一个元素a[i][0]的地址是$\alpha+i\times n$,知道了a[i][0]的地址,那么a[i][j]的地址就是$\alpha+i\times n+j$。因此二维数组中元素a[i][j]的地址为:

$$\alpha+i\times n+j$$

对于三维数组a[m][n][p],可以看成m个大小为$n\times p$的二维数组,如图5.5所示。

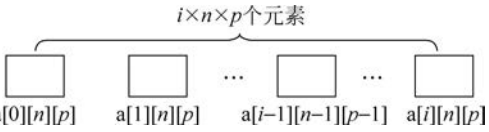

图5.5 三维数组的以行为主的顺序表示法

因为在元素a[i][0][0]之前有i个大小为$n\times p$的二维数组,所以元素a[i][0][0]的地址为$\alpha+i\times n\times p$,这也是第i个二维数组的基地址,由a[i][0][0]的地址及二维数组的寻址公式,可求得三维数组元素a[i][j][k]的寻址公式为(以行优先为例):

$$\alpha+i\times n\times p+j\times p+k$$

综上所述,可以看出数组的两个特点:第一,对同一数组的任何一个元素,由下标求存储地址的运算时间是一样的,也就是说对任何一个数组元素的访问过程是平等的,这是随机存取结构的一个优点;第二,为了在内存中给数组开辟足够的存储单元,数组的维数和大小必须事先给出。这对于数组大小不能预先确定的问题就很不方便,数组大小规定大了会浪费空间;规定小了,运行时会出现越界,使程序无法运行下去。这是数组这种数据结构的一个缺点。不过,对于许多应用程序来说,数组仍然是完成任务的最合适的工具。

5.3 矩阵的压缩存储

矩阵是许多科学与工程计算问题中经常出现的数学对象。这里,我们感兴趣的不是矩阵本身,我们所关心的是表示矩阵的方法,以使对矩阵的各种运算能有效地完成。数学上,一个矩阵一般由m行和n列元素组成,如图5.6所示。其中,矩阵元素都是数字。矩阵$\boldsymbol{M}_1$有3行4列共12个元素,矩阵$\boldsymbol{M}_2$有5行5列共25个元素。

3	2	7	8
7	0	2	0
8	1	0	6

(a) $\boldsymbol{M}_1$

7	6	5	9	0
6	0	3	0	7
5	3	2	1	0
9	0	1	5	2
0	7	0	2	8

(b) $\boldsymbol{M}_2$

图5.6 两个矩阵$\boldsymbol{M}_1$与$\boldsymbol{M}_2$

5.3.1 特殊矩阵——对称矩阵和三角矩阵

一般地,我们把一个m行n列的矩阵表示成$m\times n$,矩阵的元素个数总计为$m\times n$个。如果m等于n,则称该矩阵为方阵。对于一个$m\times n$的矩阵,用二维数组,例如matrix[m][n]

来表示最简单不过了，这种表示法需要的存储空间是 $m \times n$。在这种表示方法下，通过 matrix[i][j]可以快速找到矩阵中的任意元素。然而，这种存储表示也存在一些问题，如果观察图 5.7(b)所示的矩阵 $\boldsymbol{M}_2$，就能够发现，该矩阵中的元素是对称的，即矩阵中第 i 行第 j 列与第 j 行第 i 列元素的值相等，这种矩阵称为对称矩阵。对于 $n \times n$ 的对称矩阵，我们只需要为每一对对称元素分配一个存储空间，这样就只需要存储其下三角或上三角(包括对角线)中的元素即可，如图 5.7 中的阴影部分所示。

7	6	5	9	0
6	0	3	0	7
5	3	2	1	0
9	0	1	5	2
0	7	0	2	8

(a) 下三角元素

7	6	5	9	0
6	0	3	0	7
5	3	2	1	0
9	0	1	5	2
0	7	0	2	8

(b) 上三角元素

图 5.7　矩阵 $\boldsymbol{M}_2$ 的下三角和上三角元素

因为 $n \times n$ 矩阵的下三角或上三角的元素有 $n(n+1)/2$ 个，所以可以将 n^2 个元素压缩存储到 $n(n+1)/2$ 个存储单元中。假设以一维数组 a[$n(n+1)/2$]来存储 $n \times n$ 对称矩阵的下三角元素，那么矩阵中第 i 行第 j 列元素应该放在数组 a 中的哪一个位置呢？注意，矩阵的第 0 行只存储一个元素，第 1 行存储 2 个元素，以此类推。这样，第 $i+1$ 行前面共有 i 行，因此总共要存储 $1+2+3+\cdots+i=i \times (1+i)/2$ 个，反之，对所有的 $k=0,1,2,\cdots,n(n+1)/2-1$，都能确定 a[$k$]中的元素在矩阵中的位置($i$,$j$)。图 5.8 给出了图 5.6(b)所示矩阵 $\boldsymbol{M}_2$ 的压缩存储形式。

7	6	0	5	3	2	9	0	1	5	0	7	0	2	8
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

图 5.8　对称矩阵 $\boldsymbol{M}_2$ 的压缩存储

当一个 $n \times n$ 矩阵的主对角线上方或下方的所有元素皆为 0 时，称该矩阵为三角矩阵。下三角矩阵中的右上角总是 0，如图 5.9(a)所示。而图 5.9(b)所示的是一个上三角矩阵。对于这样的三角矩阵，同样也可采用对称矩阵的压缩存储方式将其上三角或下三角的元素存储在一维数组中，以达到节约存储空间的目的。

a_{00}	0	0	0
a_{10}	a_{11}	0	0
a_{20}	a_{21}	a_{22}	0
a_{30}	a_{31}	a_{32}	a_{33}

(a) 下三角矩阵

a_{00}	a_{01}	a_{02}	a_{03}
0	a_{11}	a_{12}	a_{13}
0	0	a_{22}	a_{23}
0	0	0	a_{33}

(b) 上三角矩阵

图 5.9　三角矩阵

5.3.2　稀疏矩阵

除了上述介绍的对称矩阵和三角矩阵等特殊矩阵外，在实际应用中我们还经常遇到这样一种矩阵。观察如图 5.10(a)所示的矩阵，就可发现矩阵中的很多元素为 0，这样的矩阵称为稀疏矩阵。至于矩阵中 0 元素多少才能算作稀疏矩阵，并没有确切定义。这只是一个直观上的概念。在图 5.10(a)所示的矩阵的 42 个元素中，只有 8 个是非 0 元素，就可以认为

这个矩阵是稀疏矩阵。

	0	1	2	3	4	5
0	5	0	0	2	0	8
1	0	6	9	0	0	0
2	0	0	0	3	0	0
3	0	0	0	0	0	0
4	9	0	0	0	0	0
5	0	0	2	0	0	0
6	0	0	0	0	0	0

(a) 稀疏矩阵M

	行	列	值
0	7	6	8
1	0	0	5
2	0	3	2
3	0	5	8
4	1	1	6
5	1	2	9
6	2	3	3
7	4	0	9
8	5	2	2

(b) 稀疏矩阵M的三元组表

图 5.10　稀疏矩阵示例

稀疏矩阵要求我们考虑一种新的表示方法。之所以如此,一方面是因为在实际应用中许多矩阵都是大型的,而同时又是稀疏的。例如 1000×1000 的矩阵,一百万个元素中只有 1000 个是非 0 元素。另一方面,矩阵中的 0 元素在矩阵运算中是没有意义的。例如在矩阵转置和矩阵相乘运算中,0 元素的运算可不必考虑。如果采用二维数组表示矩阵既浪费了大量的存储单元来存储 0 元素,又要花大量的时间进行 0 元素的运算。所以我们要为稀疏矩阵寻找一种新的存储其元素的方法,这种表示法将只存储矩阵中的非 0 元素。

由于稀疏矩阵中的非 0 元素的位置是没有规定的,因此为了表示一个矩阵的非 0 元素 M_{ij},我们必须存储该元素所在的行、列和其值。这样可将一个稀疏矩阵中的非零元素按三元组(行,列,值)存储。为操作方便起见,按照行号递增、任意行按照列号递增的方式组织三元组,最终形成三元组表结构。于是,图 5.10(a)所示的稀疏矩阵 $\boldsymbol{M}$ 就可存储在三元组表 a[t+1][3]中,如图 5.10(b)所示。其中 t=8 是稀疏矩阵中非 0 元素的个数,三元组表中 a[0][1]、a[0][2]和 a[0][3]分别表示稀疏矩阵的行数、列数和非 0 元素个数。

在矩阵上最小的操作集合包括创建、转置、加法、减法和乘法。下面介绍矩阵转置和矩阵相乘算法。

1. 稀疏矩阵的转置算法

对于一个 $m\times n$ 的矩阵 $\boldsymbol{M}$,它的转置矩阵 $\boldsymbol{T}$ 是一个 $n\times m$ 的矩阵,而且原矩阵 $\boldsymbol{M}$ 的任意元素 $M[i][j]$应该成为其转置矩阵中的元素 $T[j][i]$。换句话说,就是把矩阵的行与列对换。图 5.10(a)所示矩阵 $\boldsymbol{M}$ 的转置矩阵 $\boldsymbol{T}$ 和 $\boldsymbol{T}$ 对应的三元组表 b 如图 5.11(a)和图 5.11(b)所示。

如果矩阵采用二维数组表示,转置算法很容易实现。对于一个 $m\times n$ 的矩阵 $\boldsymbol{M}$,得到其转置矩阵 $\boldsymbol{T}$ 的关键代码为

```
for (j = 0; j < n; j++)
    for (i = 0; i < m; i++)
        T[j][i] = M[i][j];
```

由于稀疏矩阵是用三元组表表示的,很显然不能使用上述算法进行转置。那么对于图 5.10(b)中的三元组表来说,如何得到图 5.11(b)所示的三元组表呢? 一个简单的想法是,顺序地取出 a 中的元素然后顺序地放到 b 中。例如,取出 a 中的第 1 个元素(0,0,5)变成(0,0,5)放到 b 中的第 1 行,取出 a 中的第 2 个元素(0,3,2)变成(3,0,2)放到 b 中的第 2

5	0	0	0	9	0	0
0	6	0	0	0	0	0
0	9	0	0	0	2	0
2	0	3	0	0	0	0
0	0	0	0	0	0	0
8	0	0	0	0	0	0

(a) 转置矩阵T

	0	1	2
0	6	7	8
1	0	0	5
2	0	4	9
3	1	1	6
4	2	1	9
5	2	5	2
6	3	0	2
7	3	2	3
8	5	0	8

(b) T的三元组表

图 5.11　矩阵转置示例

行，a 的第 3 个元素(0,5,8)变成(5,0,8)放到 b 中的第 3 行，以此类推。但是观察图 5.10(b)和图 5.11(b)就能够发现，这种做法是错误的，因为我们看到 a 中元素(0,3,2)变成(3,0,2)并不是放在 b 中的第 2 行，而是放在 b 中的第 6 行。

正确的方法应该是将原矩阵 $\boldsymbol{M}$ 中的元素按每一列从上而下进行转置。为此，应对三元组表 a 中的第 2 列元素进行多次反复扫描，第一次取出矩阵 $\boldsymbol{M}$ 中列号为 0 的所有元素，将它们顺序放入 b 中，第二次取出矩阵 $\boldsymbol{M}$ 中列号为 1 的元素，将它们顺序放入 b 中。如此进行下去，直到 a 中所有元素均放入 b 中，转置过程结束。

以图 5.10(a)所示的稀疏矩阵 $\boldsymbol{M}$ 为例。

第一次，挑出第 2 列所有数值为 0 者，即在矩阵 $\boldsymbol{M}$ 中的第 0 列者，逐个放入数组 b 中。a 中第 1 行元素(0,0,5)和第 7 行元素(4,0,9)的列号为 0，因此将它们分别放入 b 的第 1 行和第 2 行。三元组表 b 如图 5.12 所示。

	0	1	2
0	6	7	8
1	0	0	5
2	0	4	9
3	1	1	6
4			
5			
6			
7			
8			

图 5.12　稀疏矩阵 $\boldsymbol{M}$ 的三元组表 1(第 2 列所有数值为 0 者)

第二次，挑出第 2 列所有数值为 1 的元素，只有第 4 行元素(1,1,6)的列号为 1，将其放到 b 的第 3 行。此时的三元组表 b 如图 5.13 所示。

第三次，找列号为 2 的元素进行转置。列号为 2 的元素有两个，即(1,2,9)和 (5,2,2)，将 a 中这两个元素分别放到 b 的第 4 和第 5 行。三元组表 b 如图 5.14 所示。

如此做下去，直到 a 中所有元素都放到 b 中为止，转置结束。最后的三元组表 b 如图 5.11(b)所示。需要注意的是，当 a 的某行放到 b 中时，a 的第 1 列数值放到 b 的第 2 列，

	0	1	2
0	6	7	8
1	0	0	5
2	0	4	9
3			
4			
5			
6			
7			
8			

图 5.13　稀疏矩阵 ***M*** 的三元组表 2(第 2 列所有数值为 1 者)

	0	1	2
0	6	7	8
1	0	0	5
2	0	4	9
3	1	1	6
4	2	1	9
5	2	5	2
6			
7			
8			

图 5.14　稀疏矩阵 ***M*** 的三元组表 3(列号为 2 的元素转置后)

a 的第 2 列数值放到 b 的第 1 列。

实现上述矩阵转置算法的 C 语言函数 transpose 的算法如下所示。其中 m 和 n 分别为矩阵 ***M*** 的行数和列数,t 为 ***M*** 的非 0 元素个数。p 表示 a 中某行的下标,变量 q 指向转置矩阵三元组表 b 中下一个元素插入的位置。col 表示矩阵 ***M*** 中某列的列号。

```
void transpose(a,b)
{  m = a[0][0];
   n = a[0][1];
   t = a[0][2];
   b[0][0] = n;
   b[0][1] = m;
   b[0][2] = t;
   if(t == 0) exit;
   q = 1;                                  /* 从 b 的第一行做起 */
   for(col = 0;col < n;col++)              /* 按 M 的列进行转置 */
       for(p = 1;p <= t;p++)               /* 对所有非 0 元素 */
           if(a[p][1] == col)              /* 挑出 M 中列号等于 col 的元素 */
               { b[q][0] = a[p][1];
                 b[q][1] = a[p][0];
                 b[q][2] = a[p][2];
                 q++;
                 }
   }
```

以上转置算法主要的运算时间花在两个 for 循环上,外循环共执行 n 次,内循环需要执

行 t 次，所以上述算法的时间复杂度是 $O(n\times t)$。除了 a 和 b 所需空间外，该算法只需固定量的额外空间，即变量 m、n、t、p、col 和 q 所用的空间。

回顾上面我们介绍的把矩阵表示成二维数组时，矩阵转置算法可以在 $O(n\times m)$的时间内完成矩阵转置。当矩阵中非 0 元素个数 t 的数量级为 $n\times m$ 时，算法 transpose 的时间复杂度 $O(n\times t)$便成为 $O(n\times m^2)$。这显然比用二维数组时的时间 $O(n\times m)$更差。

如果能预先确定 a 中每一个元素在 b 中的位置，那么就可以按 a 中元素的顺序逐个将元素放到 b 中。例如，如果知道 a 中元素(0,0,5)在 b 中的位置是 1，就能很容易将元素(0,0,5)取出放在 b 中的相应位置。同样，如果知道 a 中元素(0,3,2)在 b 中的位置为 6，那么从 a 中取出元素(0,3,2)时，很清楚该元素变成(3,0,2)后，应放在 b 中的第 6 个位置。采用这种思想的转置算法称为快速转置。

为了确定 a 中元素在 b 中的位置，需要知道 $\boldsymbol{M}$ 的每一列中非 0 元素的个数，有了这个信息，就可容易地获得 $\boldsymbol{M}$ 中每一列的第一个非 0 元素在 b 中的起始位置。设一维数组 S 用来存放矩阵 $\boldsymbol{M}$ 中每一列的非 0 元素个数，S[j]表示矩阵 $\boldsymbol{M}$ 的第 j 列非 0 元素的个数。一维数组 T 存放矩阵 $\boldsymbol{M}$ 的每一列的第一个非 0 元素的起始位置，T[j]表示 $\boldsymbol{M}$ 中第 j 列第一个非 0 元素在 b 中的起始位置。T[i]的值可通过下述公式推出：

$$\mathrm{T}[0]=1$$

$$\mathrm{T}[i]=\mathrm{T}[i-1]+\mathrm{S}[i-1],\quad i\geqslant 1$$

图 5.10(a)所示矩阵的 S 和 T 的值如图 5.15 所示。

i	0	1	2	3	4	5
S	2	1	2	2	0	1
T	1	3	4	6	8	8

图 5.15　S 和 T 的值

因为 T[0]的值为 1，所以当从 a 中取出元素(0,0,5)时，该元素应放在 b 中 T[0]所表示的位置，即 b[1]。同样 a 中元素(0,3,2)变换为(3,0,2)后，应放到 b 中第 6 个位置，这是因为 T[3]的值为 6。因为 T[5]的值为 8，所以 a 中元素(0,5,8)变为元素(5,0,8)，应放到 b 中最后一个位置。

下面的算法给出的是快速转置算法实现。第 17 行、第 18 行计算矩阵 $\boldsymbol{M}$ 的每一列的非 0 元素个数，S[i]的值通过统计 a 中第 2 列 i 的出现次数得到。第 19～21 行计算矩阵 $\boldsymbol{M}$ 中每一列第一个非 0 元素在 b 中的起始位置。第 22～28 行对 a 中的元素依此进行转置。首先取出元素的列号(第 23 行)，然后根据其 T 的值，将其放到 b 中的相应位置。第 27 行计算同一行的下一个元素的位置。

```
#define  MAXCOL 100
int a[][3] = {{7,6,8},{0,0,5},{0,3,2},{0,5,8},{1,1,6},{1,2,9},{2,3,3},{4,0,9},{5,2,2}};
int b[9][3];
FastTranspose(int a[][3], int b[][3])
{
   int  m,n,tn,i,j;
   int  S[MAXCOL], T[MAXCOL];
   m = a[0][0];
   n = a[0][1];
   tn = a[0][2];
   b[0][0] = n;
```

```
    b[0][1] = m;
    b[0][2] = tn;
    if(tn <= 0)   return;
    for(i = 1;i < n;i++)
       S[i] = 0;
    for(i = 1;i <= tn;i++)
       S[a[i][1]] = S[a[i][1]] + 1;
    T[0] = 1;
    for(i = 1;i < n;i++)
       T[i] = T[i - 1] + S[i - 1];
    for(i = 1;i <= tn;i++)
      {  j = a[i][1];
         b[T[j]][0] = a[i][1];
         b[T[j]][1] = a[i][0];
         b[T[j]][2] = a[i][2];
         T[j] = T[j] + 1;
      }
    }
main()
 { FastTranspose(a,b);}
```

上述快速矩阵转置算法中有4个循环,它们分别执行n、t、$n-1$和t次。这些循环每重复一次,只占用一个常量时间,故此算法的时间复杂度为$O(n+t)$。

2. 稀疏阵的乘法

设有矩阵$\boldsymbol{A}$和$\boldsymbol{B}$,其中$\boldsymbol{A}$矩阵大小为$m \times n$,矩阵$\boldsymbol{B}$大小为$n \times p$,$\boldsymbol{A}$和$\boldsymbol{B}$的乘积矩阵$\boldsymbol{C}$是$m \times p$矩阵,该矩阵的i、j元素定义为:

$$c_{ij} = \sum_{0 \leqslant k < n} a_{ik} \times b_{kj}$$

其中,$0 \leqslant i < m$,$0 \leqslant j < p$。

如果矩阵$\boldsymbol{A}$、$\boldsymbol{B}$和$\boldsymbol{C}$都采用二维数组表示,则两个矩阵相乘的经典算法是:

```
for (i = 0;i < m; i++)
    for (j = 0;j < p;j++) {
        c[i][j] =0;
        for (k = 0;k < n;k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];  }      }
```

这个算法的时间复杂度为$O(m \times n \times p)$。

在矩阵相乘的经典算法中,不论a[i][k]和b[k][j]的值是否为0,都要进行乘法运算。而实际上我们知道,a[i][k]和b[k][j]有一个值为0时,其乘积也为0。对于图5.16所示的稀疏矩阵$\boldsymbol{A}$和$\boldsymbol{B}$而言,矩阵$\boldsymbol{A}$中的非0元素a[0][0]只需与矩阵$\boldsymbol{B}$中的非0元素b[0][1]相乘。矩阵$\boldsymbol{A}$中的元素a[1][1]不需要与矩阵$\boldsymbol{B}$中的任何元素相乘。

0	2
0	0
2	0
0	1

(a) 稀疏矩阵$\boldsymbol{A}$

0	8
0	0
0	4

(b) 稀疏矩阵$\boldsymbol{B}$

3	0	0	2
0	1	0	0
2	0	0	0

(c) 矩阵乘积$\boldsymbol{C}$

图5.16　两个稀疏矩阵相乘

因此，当 ***A*** 和 ***B*** 是稀疏矩阵并用三元组表表示时，就不能套用上述算法。在对稀疏矩阵进行乘法运算时，不应该考虑值为 0 的元素。换句话说，稀疏矩阵相乘时，只需进行非 0 元素之间的运算。

对于三元组表表示的稀疏矩阵，如果从三元组表 a 中取出某元素，它在矩阵 ***A*** 中的行号为 u=a[i][0]，列号为 k=a[i][1]，则需要在三元组表 b 中找出矩阵 ***B*** 中所有行号为 k 的非 0 元素并相乘。如果某元素在矩阵 ***B*** 中的行号为 k，列号为 v=b[j][1]，则它应与 a 中取出的该非 0 元素相乘后，乘积应加到矩阵 ***C*** 的第 u 行第 v 列元素上。这是因为矩阵 ***A*** 中第 u 行的其他非 0 元素与矩阵 ***B*** 中第 v 列的相应元素相乘，其乘积也会加到矩阵 ***C*** 的这个元素上。

以图 5.17 所示的三元组表为例。对三元组表 a 中的元素 a[1]来说，它表示矩阵 ***A*** 的非 0 元素(0,0,3)，其列号为 0，该元素只需要与矩阵 ***B*** 的第 0 行的非 0 元素相乘。由于矩阵 ***B*** 的第 0 行只有一个非 0 元素(0,1,2)，其在三元组表中的位置是 b[1]，因此两者的乘积为 6，该乘积是元素 c[0][1]的一部分。a[2]表示的矩阵 ***A*** 的非 0 元素(0,3,2)应与 b[3]表示的矩阵 ***B*** 非 0 元素(3,1,1)相乘，其乘积也是 c[0][1] 的一部分，应累加到 c[0][1] 上，因此 c[0][1]的值为 8。而 a 中的元素 a[3]表示矩阵 ***A*** 的非 0 元素(1,1,1)应与矩阵 ***B*** 的第 1 行非 0 元素相乘，由于 ***B*** 中没有行号为 1 的非 0 元素，因此，该元素不需要和 b 中的任何元素相乘。

	0	1	2
0	3	4	4
1	0	0	3
2	0	3	2
3	1	1	1
4	2	0	2

(a) 稀疏矩阵*A*的三元组表a

	0	1	2
0	4	2	3
1	0	1	2
2	2	0	2
3	3	1	1

(b) 稀疏矩阵*A*的三元组表b

图 5.17　矩阵的三元组表

根据上面的分析，稀疏矩阵相乘的基本操作是：逐个从三元组表 a 中取出元素，按其列号 j 在 b 中找出矩阵 ***B*** 中所有行号为 j 的非 0 元素，将它们分别与 ***A*** 中取出的非 0 元素相乘，将乘积累加到 ***C*** 中相应的元素中去。为便于操作，应对每个乘积设一个累计和的变量，其初值为 0。

一个实现上述稀疏矩阵相乘的算法如下所示。假设 ***A*** 是一个 $m\times p$ 稀疏矩阵，非 0 元素个数为 ta，***B*** 是一个 $p\times n$ 稀疏矩阵，非零元素个数为 tb，且 ***A*** 与 ***B*** 已表示成三元组表的形式，***A*** 与 ***B*** 的乘积为 ***C***，表示成 $m\times n$ 的二维数组形式。如果矩阵 ***C*** 是稀疏的，还需要进行进一步的运算，将它表示成三元组表的形式。

```
#define  M  3
#define  P  4
#define  N  2
int a[0][3] = {{3,4,},{0,0,3},{0,3,2}{1,1,1},{2,0,2}};
int b[0][3] = {{4,3,2},{0,1,2},{2,0,2},{3,1,1}};
int c[M][N];
MatrixMultiply()
{
   int m,n,ta,tb,i,j,p,k;
```

```
    int S[P],T[P];
    m = a[0][0];
    n = a[0][1];
    ta = a[0][2];
    if(n = b[0][0])
      { p = b[0][1];
        Tb = b[0][2];
       }
    if(ta * tb == 0) return;
    for(i = 0;i < n;i++)
      S[i] = 0;
    for(i = 0;i < = tb;i++)
      S[b[i]][0] = S[b[i]][0] + 1;
    T[0] = 1;
    for(i = 0;i < = n;i++)
      T[i] = T[i - 1] + S[i - 1];
    for(i = 0;i < = ta;i++)
      { k = a[i][1];                                    /* a 中列号为 k 的非 0 元素 */
       for(j = T[k];j < T[k + 1] - 1;j++)               /* b 中第 k 行所有非 0 元素 */
         c[a[i][0]][b[j][1]] = c[a[i][0]][b[j][1]] + a[i][2] * b[j][2];
       }
}
main()
{
      MatrixMultiply();
}
```

5.4 广义表的定义与性质

5.4.1 广义表的定义

广义表(Lists,又称列表)是线性表的推广。在第2章中,我们把线性表定义为 $n \geqslant 0$ 个元素 $a_1, a_2, \cdots, a_n$ 的有限序列。线性表的元素仅限于原子项,原子是作为结构上不可分割的成分,它可以是一个数或一个结构,若放松对表元素的这种限制,容许它们具有其自身结构,这样就产生了广义表的概念。

广义表是 $n(n \geqslant 0)$ 个元素 $a_1, a_2, \cdots, a_n$ 的有限序列,其中 a_i 或者是原子项,或者是一个广义表。通常记作 $LS=(a_1, a_2, \cdots, a_n)$。LS是广义表的名字,$n$ 为它的长度。若 a_i 是广义表,则称它为LS的子表。

通常用圆括号将广义表括起来,用逗号分隔其中的元素。为了区别原子和广义表,书写时用大写字母表示广义表,用小写字母表示原子。若广义表 $LS(n \geqslant 1)$ 非空,则 a_1 是LS的表头,其余元素组成的表 $(a_2, a_3, \cdots, a_n)$ 称为LS的表尾。

显然广义表是递归定义的,这是因为在定义广义表时又用到了广义表的概念。广义表的例子如下。

(1) A=():A是一个空表,其长度为0。

(2) B=(e):表B只有一个原子e,B的长度为1。

(3) C=(a,(b,c,d)):表C的长度为2,两个元素分别为原子a和子表(b,c,d)。

(4) D=(A,B,C)：表 D 的长度为 3，三个元素都是广义表。显然，将子表的值代入后，则有 D=((),(e),(a,(b,c,d)))。

(5) E=(E)：这是一个递归的表，它的长度为 2，E 相当于一个无限的广义表 E=(a,(a,(a,(a,…))))。

从上述定义和例子可推出广义表的三个重要结论分别如下。

(1) 广义表的元素可以是子表，而子表的元素还可以是子表。由此，广义表是一个多层次的结构，可以用图形象地表示。

(2) 广义表可为其他表所共享。例如在上述例(4)中，广义表 A、B、C 为 D 的子表，则在 D 中可以不必列出子表的值，而是通过子表的名称来引用。

(3) 广义表的递归性。

综上所述，广义表不仅是线性表的推广，也是树的推广。

5.4.2 广义表的性质

从上述广义表的定义和例子可以得到广义表的下列重要性质。

(1) 广义表是一种多层次的数据结构。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。

(2) 广义表可以是递归的表。广义表的定义并没有限制元素的递归，即广义表也可以是其自身的子表。例如，表 E 就是一个递归的表。

(3) 广义表可以为其他表所共享。例如，表 A、表 B、表 C 是表 D 的共享子表。在表 D 中可以不必列出子表的值，而用子表的名称来引用。

广义表的上述特性对于它的使用价值和应用效果起到了很大的作用。

广义表可以看成线性表的推广，线性表是广义表的特例。广义表的结构相当灵活，在某种前提下，它可以兼容线性表、数组、树和有向图等各种常用的数据结构。

当二维数组的每行(或每列)作为子表处理时，二维数组即为一个广义表。

另外，树和有向图也可以用广义表来表示。

由于广义表不仅集中了线性表、数组、树和有向图等常见数据结构的特点，而且可有效地利用存储空间，因此在计算机的许多应用领域都有成功使用广义表的实例。

5.5 广义表的存储结构

由于广义表中的数据元素可以具有不同的结构，因此难以用顺序的存储结构来表示。而链式的存储结构分配较为灵活，易于解决广义表的共享与递归问题，所以通常采用链式的存储结构来存储广义表。在这种表示方式下，每个数据元素可用一个结点表示。

按结点形式的不同，广义表的链式存储结构又可以分为两种不同的存储方式：一种称为头尾表示法；另一种称为孩子兄弟表示法。

5.5.1 头尾表示法

若广义表不空，则可分解成表头和表尾；反之，一对确定的表头和表尾可唯一地确定一个广义表。头尾表示法就是根据这一性质设计而成的一种存储方法。

由于广义表中的数据元素既可能是列表也可能是单元素,相应地在头尾表示法中结点的结构形式有两种:一种是表结点,用以表示列表;另一种是元素结点,用以表示单元素。在表结点中应该包括一个指向表头的指针和指向表尾的指针;而在元素结点中应该包括所表示单元素的元素值。为了区分这两类结点,在结点中还要设置一个标志域,如果标志为1,则表示该结点为表结点;如果标志为0,则表示该结点为元素结点,其形式定义说明如下。

```
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 */
       datatype   data;                       /* data 是元素结点的值域 */
       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 */
```

头尾表示法的结点形式如图5.18所示。

(a) 表结点　　(b) 元素结点

图5.18　头尾表示法的结点形式

对于5.4.1节所列举的广义表A、B、C、D、E、F,若采用头尾表示法的存储方式,其存储结构如图5.19所示。

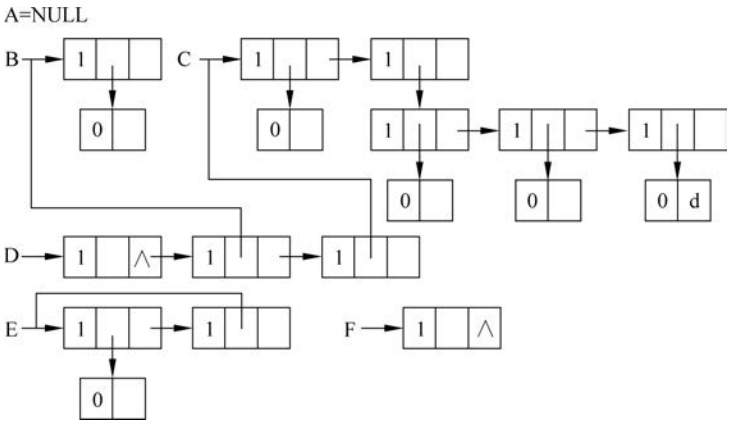

图5.19　广义表的头尾表示法存储结构示例

从上述存储结构示例中可以看出,采用头尾表示法容易分清列表中单元素或子表所在的层次。例如,在广义表D中,单元素a和e在同一层次上,而单元素b、c、d在同一层次上,且比a和e低一层,子表B和C在同一层次上。另外,最高层的表结点的个数即为广义表的长度。例如,在广义表D的最高层有3个表结点,其广义表的长度为3。

5.5.2 孩子兄弟表示法

广义表的另一种表示法称为孩子兄弟表示法。在孩子兄弟表示法中，也有两种结点形式：一种是有孩子结点，用以表示列表；另一种是无孩子结点，用以表示单元素。在有孩子结点中包括一个指向第一个孩子(长子)的指针和一个指向兄弟的指针；而在无孩子结点中包括一个指向兄弟的指针和该元素的元素值。为了能区分这两类结点，在结点中还要设置一个标志域。如果标志为 1，则表示该结点为有孩子结点；如果标志为 0，则表示该结点为无孩子结点。其形式定义说明如下：

```
typedef   enum {ATOM, LIST} Elemtag;          /* ATOM = 0: 单元素; LIST = 1: 子表 */
typedef   struct   GLENode {
    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 */
           datatype   data;                   /* 元素结点的值域 */
           struct GLENode   * hp;             /* 表结点的表头指针 */
         };
    struct GLENode   * tp;                    /* 指向下一个结点 */
  } * EGList;                                 /* 广义表类型 */
```

孩子兄弟表示法的结点形式如图 5.20 所示。

(a) 有孩子结点　　(b) 无孩子结点

图 5.20　孩子兄弟表示法的结点形式

对于 5.4.1 节中所列举的广义表 A、B、C、D、E、F，若采用孩子兄弟表示法的存储方式，其存储结构如图 5.21 所示。

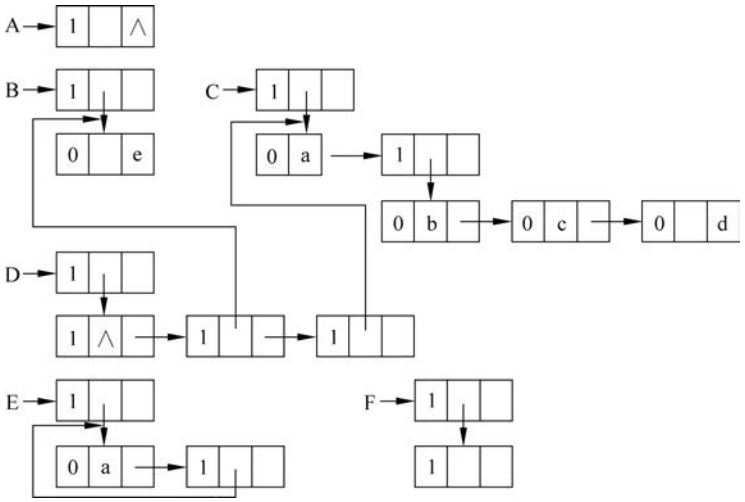

图 5.21　广义表的孩子兄弟表示法存储结构

从图 5.21 的存储结构示例中可以看出，采用孩子兄弟表示法时，表达式中的左括号“(”对应存储表示中的 tag＝1 的结点，且最高层结点的 tp 域必为 NULL。

5.6 广义表的基本操作

以头尾表示法存储广义表,讨论广义表的有关操作的实现。由于广义表的定义是递归的,因此相应的算法一般也都是递归的。

1. 广义表的取头、取尾操作

```
GList Head(GList ls)
  {
   if (ls->tag == 1)
      p = ls->hp;
   return  p;
  }

GList Tail(GList ls)
  {
   If( ls->tag == 1)
      p = ls->tp;
   return  p;
  }
```

2. 建立广义表的存储结构

算法 1:

```
int  Create(GList *ls, char *S)
 { Glist  p;  char  *sub;
   if (StrEmpty(S)) *ls = NULL;
   else {
    if (!(*ls = (GList)malloc(sizeof(GLNode))))  return  0;
    if (StrLength(S) == 1) {
      (*ls)->tag = 0;
      (*ls)->data = S;
     }
    else {
      (*ls)->tag = 1;
      p = *ls;
      hsub = SubStr(S,2,StrLength(S) - 2);
      do {
        sever(sub,hsub);
        Create(&(p->ptr.hp), sub);
        q = p;
        if (!StrEmpty(sub)){
          if (!(p = (GList)malloc(sizeof(GLNode))))  return 0;
          p->tag = 1;
          q->ptr.tp = p;
         }
       }while (!StrEmpty(sub));
       q->ptr.tp = NULL;
      }
     }
   return 1;
  }
```

算法 2：

```
int  sever(char * str, char * hstr)
{
 int  n = StrLength(str);
    i = 1; k = 0;
   for (i = 1, k = 0; i <= n || k != 0; ++i)
     {
      ch = SubStr(str,i,1);
      if (ch == '(')  ++k;
      else  if (ch == ')')  --k;
     }
   if (i <= n)
    {
     hstr = SubStr(str,1,i - 2);
     str = SubStr(str,i,n - i + 1);
     }
   else {
     StrCopy(hstr,str);
     ClearStr(str);
       }
 }
```

3. 以表头、表尾建立广义表

```
int  Merge(GList ls1,GList ls2, Glist * ls)
{
  if (!( * ls = (GList)malloc(sizeof(GLNode))))  return 0;
   * ls -> tag = 1;
   * ls -> hp = ls1;
   * ls -> tp = ls2;
  return 1;
}
```

4. 求广义表的深度

```
int Depth(GList ls)
{
   if (!ls)
     return  1;                          /* 空表深度为 1 */
   if (ls -> tag == 0)
     return  0;                          /* 单元素深度为 0 */
   for (max = 0,p = ls; p; p = p -> ptr.tp)
   {
     dep = Depth(p -> ptr.hp);           /* 求以 p -> ptr.hp 为头指针的子表深度 */
     if (dep > max)  max = dep;
    }
   return max + 1;                       /* 非空表的深度是各元素的深度的最大值加 1 */
}
```

5. 复制广义表

```
int  CopyGList(GList ls1, GList * ls2)
{
   if (!ls1)  * ls2 = NULL;              /* 复制空表 */
   else
    {
```

```
    if (!( * ls2 = (Glist)malloc(sizeof(Glnode)))) return 0;        / * 建立表结点 * /
    ( * ls2) -> tag = ls1 -> tag;
    if (ls1 -> tag == 0) ( * ls2) -> data = ls1 -> data;            / * 复制单元素 * /
    else
     {
      CopyGList(&(( * ls2) -> ptr.hp), ls1 -> ptr.hp);      / * 复制广义表 ls1 -> ptr.hp 的
                                                              一个副本 * /
      CopyGList(&(( * ls2) -> ptr.tp) , ls1 -> ptr.tp);     / * 复制广义表 ls1 -> ptr.tp 的
                                                              一个副本 * /
     }
   }
  return 1;
 }
```

5.7 数组的应用

请看下面的例题,希望能给读者带来一些指导和启发,在枯燥的学习之余给读者带来一些愉悦和收获!

【例 5.1】 请利用栈结构、利用非递归算法实现广义表的存储结构。

```
struct stack
{ struct GLNode elem[255];
  int top;
}S;                                                    //定义用于存储结点的栈
//pop(S) 和 push(S, p)表示数据的出栈和入栈
struct GLNode * bulid_ptr()                            //建立表结点函数
{
  struct GLNode * p1;
  p1 = (struct GLNode * )malloc(sizeof(struct GLNode));
  p1 -> tag = 1;                                  //表示 p 结点是表结点,并初始化两个指针变量
  p1 -> atom_ptr.hp = NULL;
  p1 -> atom_ptr.tp = NULL;
  return p1;
}
struct GLNode * bulid_atom(char c)                     //建立原子结点函数
{
  struct GLNode * p1;
  p1 = (struct GLNode * )malloc(sizeof(struct GLNode));
  p1 -> tag = 0;
  p1 -> atom_ptr.atom = c;
  return p1;
}
```

该算法的主体如下:

```
S.top = -1;
i = 0;
while(str[i]!= '/0')
{
  switch(str[i])
  {
  case '(': p = build_ptr();                           //建立表结点
         if(S.top > -1)                                //栈不为空,p 赋给栈顶表结点的 hp
```

```
            S.elem[S.top]->atom_ptr.hp=p;
        push(S,p);                                  //p结点入栈
    case ',': p=build_ptr();                        //建立表结点
        q=pop(S);                                   //S的栈顶元素出栈,并赋给q
        q->atom_ptr.tp=p;                           //q的tp指向p
        push(S,p);                                  //p结点入栈
    case ')': pop(S);                               // S的栈顶元素出栈
        default : p=build_atom(str[i]);             //建立原子结点
        S.elem[S.top]->atom_ptr.hp=p;               // p赋给栈顶表结点的hp
     }
}
```

【例 5.2】 已知 Ackerman 函数定义如下：

(1) 根据定义,写出它的递归求解算法；

(2) 利用栈,写出它的非递归求解算法。

(1) 已知函数本身是递归定义的,所以可以用递归算法来解决：

```
unsigned akm(unsigned m, unsigned n)
{
  if (m == 0) return n+1;                           //m == 0
  else if (n == 0) return akm (m-1, 1);             //m>0, n == 0
   else return  akm (m,n-1);                        //m>0, n>0
}
```

(2) 为了将递归算法改成非递归算法,首先改写原来的递归算法,将递归语句从结构中独立出来：

```
unsigned akm(unsigned m, unsigned n)
{
  unsigned v;
  if (m == 0) return n+1;                           //m == 0
  if (n == 0) return akm (m-1, 1);                  //m>0, n ==0
  v = akm(m, n-1);                                  //m>0, n>0
  return akm(m-1, v);
}
```

用一个栈记录每次递归调用时的实参值,每个结点有两个域{vm,vn}。对以上实例,栈的变化相应算法如下：

```
#include
#include "stack.h"
#define maxSize 3500;
unsigned akm(unsigned m, unsigned n)
{
  struct node { unsigned vm, vn; }
  stack st(maxSize);node w; unsigned v;
  w.vm = m; w.vn = n; st.Push(w);
  do {
      while (st.GetTop().vm>0)
        {                                           //计算 akm(m-1, akm(m, n-1) )
      while (st.GetTop().vn>0)                      //计算 akm(m, n-1),直到 akm(m, 0)
        { w.vn--; st.Push(w); }
      w = st.GetTop(); st.Pop(w0;                   //计算 akm(m-1, 1)
      w.vm--; w.vn = 1; st.Push(w);                 //直到 akm(0, akm(1, * ) )
```

```
        w = st.GetTop(); st.Pop(); v = w.vn++;  //计算 v = akm(1, * ) + 1
        if (st.IsEmpty() == 0)                  //如果栈不空, 改栈顶为(m - 1, v)
          { w = st.GetTop(); st.Pop(); w.vm -- ; w.vn = v; st.Push(w); }
        }
      } while (st.IsEmpty() == 0);
    return v;
}
```

【例 5.3】 背包问题：设有一个背包可以放入的物品的重量为 s，现有 n 件物品，重量分别为 w[1]，w[2]，…，w[n]。问能否从这 n 件物品中选择若干件放入此背包中，使得放入的重量之和正好为 s。如果存在一种符合上述要求的选择，则称此背包问题有解(或称其解为真)；否则称此背包问题无解(或称其解为假)。试用递归方法设计求解背包问题的算法。

根据递归定义，可以写出递归的算法。

```
enum boolean { False, True }
boolean Knap(int s, int n) {
if (s == 0) return True;
if (s < 0 || s > 0 && n < 1) return False;
if (Knap(s - W[n], n - 1) == True)
    { printf(" % f",W[n]);return True; }
return Knap(s, n - 1);
}
```

若设 w={0,1,2,4,8,16,32}，s=51，n=6，则递归执行过程如下：

```
递归 Knap(51, 6)
return True;                                //完成
Knap(51 - 32, 5)
return True;                                //打印 32
Knap(19 - 16, 4)
return True;                                //打印 16
Knap(3 - 8, 3)
return False;
Knap(3,3)
return True;                                //无动作 s = - 5 < 0
return False;
Knap(3 - 4, 4)
return False;
Knap(3,2)
return True;                                //无动作 s = - 1 < 0
return False ;
Knap(3 - 2, 1)
return True;                                //打印 2
Knap(1 - 1, 0)
return True;                                //打印 1 s = 0
return True
```

【例 5.4】 八皇后问题：设在初始状态下在国际象棋棋盘上没有任何棋子(皇后)。然后顺序在第 1 行，第 2 行，…，第 8 行上布放棋子。在每一行中有 8 个可选择位置，但在任一时刻，棋盘的合法布局都必须满足三个限制条件，即任何两个棋子不得放在棋盘上的同一行、同一列，或者同一斜线上，试编写一个递归算法，求解并输出此问题的所有合法布局。(提示：用回溯法。在第 n 行第 j 列安放一个棋子时，需要记录在行方向、列方向、正斜线方

向、反斜线方向的安放状态，若当前布局合法，可向下一行递归求解，否则可移走这个棋子，恢复安放该棋子前的状态，试探本行的第 $j+1$ 列）。

此为典型的回溯法问题。

在解决八皇后问题时，采用回溯法。在安放第 i 行皇后时，需要在列 j 的方向从 1 到 n 试探（$j=1,2,\cdots,n$）。首先在第 j 列安放一个皇后，如果在列、主对角线、次对角线方向有其他皇后，则出现攻击，撤销在第 j 列安放的皇后。如果没有出现攻击，则在第 j 列安放的皇后不动，递归安放第 $i+1$ 行皇后。

解题时设置如下 4 个数组。

（1）col[n]：col[i] 标识第 i 列是否安放了皇后；

（2）md[$2n-1$]：md[k] 标识第 k 条主对角线是否安放了皇后；

（3）sd[$2n-1$]：sd[k] 标识第 k 条次对角线是否安放了皇后；

（4）q[n]：q[i] 记录第 i 行皇后在第 n 列。

利用行号 i 和列号 j 计算主对角线编号 k 的方法是 $k=n+i-j-1$；计算次对角线编号 k 的方法是 $k=i+j-n$。八皇后问题解法如下：

```
void Queen(int i) {
for (int j = 0; j < n; j++) {
    if (col[j] == 0 && md[n+i-j-1] == 0 && sd[i+j] == 0) {    //第 i 行第 j 列没有攻击
            col[j] = md[n+i-j-1] = sd[i+j] = 1; q[i] = j;   //在第 i 行第 j 列安放皇后
            if (i == n)                                       //输出一个布局
                  for (j = 0; j < n; j++) printf("%d", q[j]);
                        printf("\n");
}
    else { Queen(i+1);                                        //在第 i+1 行安放皇后
            col[j] = md[n+i-j-1] = sd[i+j] = 0; [i] = 0;    //撤销第 i 行第 j 列的皇后
}
}
```

【例 5.5】 如果矩阵 $\boldsymbol{A}$ 中存在这样的一个元素 A[i][j]满足条件：A[i][j]是第 i 行中值最小的元素，且又是第 j 列中值最大的元素，则称为该矩阵的一个马鞍点。编写一个函数计算出 $1\times n$ 的矩阵 $\boldsymbol{A}$ 的所有马鞍点。

算法思想：依题意，先求出每行的最小值元素，放入 min[m]中，再求出每列的最大值元素，放入 max[n]中，若某元素既在 min[m]中又在 max[n]中，则该元素 A[i][j]便是马鞍点。找出所有这样的元素，即找到了所有马鞍点。因此，实现本题功能的程序如下。

```
#include <stdio.h>
#define m 3
#define n 4
void minmax(int a[m][n])
{
int i1,j,have = 0;
int min[m],max[n];
for(i1 = 0; i1 < m; i1++)                          /*计算出每行的最小值元素,放入 min[m]中*/
  {
    min[i1] = a[i1][0];
    for(j = 1;j < n;j++)
      if(a[i1][j]< min[i1]) min[i1] = a[i1][j];
  }
for(j = 0;j < n;j++)                               /*计算出每列的最大值元素,放入 max[n]中*/
```

```
    {
      max[j] = a[0][j];
      for(i1 = 1; i1 < m; i1++)
          if(a[i1][j]> max[j]) max[j] = a[i1][j];
    }
  for(i1 = 0; i1 < m; i1++)
    for(j = 0;j < n;j++)
      if(min[i1] == max[j])
      {
          printf("( %d, %d): %d\n", i1,j,a[i1][j]);
          have = 1;
    }
  if(!have) printf("没有马鞍点\n");
  }
```

【例5.6】 利用广义表的head和tail操作写出函数表达式,把以下各题中的单元素banana从广义表中分离出来:

(1) L1(apple, pear, banana,orange);

(2) L2((apple, pear), (banana, orange));

(3) L3(((apple), (pear), (banana), (orange)));

(4) L4((((apple))), ((pear)), (banana), orange);

(5) L5((((apple), pear), banana), orange);

(6) L6(apple, (pear, (banana), orange))。

函数表达式如下:

(1) Head(Tail(Tail(L1)));

(2) Head(Head(Tail(L2)));

(3) Head(Head(Tail(Tail(Head(L3)))));

(4) Head(Head(Tail(Tail(L4))));

(5) Head(Tail(Head(L5)));

(6) Head(Head(Tail(Head(Tail(L6)))))。

【例5.7】 编写下列程序:

(1) 求广义表表头和表尾的函数head和tail。

(2) 计算广义表原子结点个数的函数count_GL。

(3) 计算广义表所有原子结点数据域(设数据域为整型)之和的函数sum_GL。

```
#include "stdio.h"
#include "malloc.h"
typedef struct node
{ int tag;
union
  { struct  node  * sublist;
    char data;
  }dd;
struct  node  * link;
}NODE;
NODE  * creat_GL(char ** s)
{
   NODE * h;
   char ch;
```

```
   ch = *(*s);
   (*s)++;
 if(ch!= '\0')
 {
    h = (NODE *)malloc(sizeof(NODE));
    if(ch == '(')
        {
              h->tag = 1;
              h->dd.sublist = creat_GL(s);
        }
      else
        {
              h->tag = 0;
              h->dd.data = ch;
        }
      }
    else
        h = NULL;
ch = *(*s);
(*s)++;
if(h!= NULL)
    if(ch == ',')
        h->link = creat_GL(s);
    else
        h->link = NULL;
return(h);
}
void prn_GL(NODE *p)
{
if(p!= NULL)
{
    if(p->tag == 1)
{
printf("(");
if(p->dd.sublist == NULL)
        printf(" ");
else
      prn_GL(p->dd.sublist);
}
else
      printf("%c",p->dd.data);
if(p->tag == 1)
      printf(")");
if(p->link!= NULL)
{
      printf(",");
      prn_GL(p->link);
}
}
}
NODE *copy_GL(NODE *p)
{
NODE *q;
if(p == NULL)  return(NULL);
q = (NODE *)malloc(sizeof(NODE));
q->tag = p->tag;
if(p->tag)
```

```
    q->dd.sublist = copy_GL(p->dd.sublist);
else
    q->dd.data = p->dd.data;
q->link = copy_GL(p->link);
return(q);
}
NODE *head(NODE *p)                          /*求表头函数*/
{
return(p->dd.sublist);
}
NODE *tail(NODE *p)                          /*求表尾函数*/
{
return(p->link);
}
int sum(NODE *p)                             /*求原子结点的数据域之和函数*/
{ int m,n;
if(p==NULL) return(0);
else
    {  if(p->tag==0) n=p->dd.data;
       else
           n=sum(p->dd.sublist);
       if(p->link!=NULL)
           m=sum(p->link);
           else m=0;
return(n+m);
    }
}
int depth(NODE  *p)                          /*求表的深度函数*/
{
int  h, maxdh;
NODE *q;
if(p->tag==0) return(0);
else  if(p->tag==1&&p->dd.sublist==NULL) return 1;
      else
         {
           maxdh=0;
           while(p!=NULL)
             {
                if(p->tag==0) h=0;
                else
                    {  q=p->dd.sublist;
                       h=depth(q);
                    }
                if(h>maxdh)maxdh=h;
                       p=p->link;
             }
           return(maxdh+1);
      }
}
main()
{
NODE  *hd, *hc;
char s[100], *p;
p=gets(s);
hd=creat_GL(&p);
prn_GL(head(hd));
prn_GL(tail(hd));
```

```
hc = copy_GL(hd);
printf("copy after:");
prn_GL(hc);
printf("sum: %d\n",sum(hd));
printf("depth: %d\n",depth(hd));
}
```

本章小结

(1) 了解数组的两种存储表示方法,并掌握数组在以行为主的存储结构中的地址计算方法。

(2) 掌握对特殊矩阵进行压缩存储时的下标变换公式。

(3) 了解稀疏矩阵的两类压缩存储方法的特点和适用范围,领会以三元组表示稀疏矩阵时进行矩阵运算采用的处理方法。

(4) 掌握广义表的结构特点及其存储表示方法,读者可根据自己的习惯熟练掌握任意一种结构的链表,学会对非空广义表进行分解的两种分析方法,即可将一个非空广义表分解为表头和表尾两部分或者分解为 n 个子表。

(5) 学习利用分治算法的设计思想编制递归算法。

知识拓展

编程语言中的数组下标为何从 0 开始

在大多数编程语言中,数组的下标均从 0 而不是从 1 开始编号,刚刚学习数组的读者很容易弄错。那你有没有想过这是为什么呢?

从存储数组的内存模型来看,“下标”的确切含义应该是“偏移”(offset)。对于数组 A,如果它的每个元素占用 d 个存储单元,那么它的 a[0]就相于首地址偏移为 0 的内存地址,也就是数组 A 的起始存储地址,一般也称为基地址。a[i]则是相对于首地址偏移 i×d 个存储单元的内存地址。数组下标从 0 开始,计算 a[i]的内存地址只需要使用如下公式:

$$LOC(a[i]) = LOC(a[0]) + i \times d$$

但是,如果数组下标从 1 开始,计算 a [i]的内存地址的公式就会变为如下形式:

$$LOC(a[i]) = LOC(a[0]) + (i-1) \times d$$

对比上面两个公式,不难发现,如果数组下标从 1 开始,每次按照下标访问数组元素,就会多进行一次减法运算。数组是最常用的基础的数据结构,通过下标访问数组元素又是其基础的操作,效率的优化就要尽可能做到极致。因此,为了减少一次减法操作,数组的下标选择了从 0 开始编号,而不是从 1 开始编号。

也许这个理由还不够充分,数组的下标从 0 开始编号还可能与历史原因有关。最初,C 语言设计者用 0 作为数组的起始下标,目的是在一定程度上降低 C 语言程序员学习其他编程语言的成本,之后的 Java、JavaScript 等效仿了 C 语言,沿用了这一方式。

当然,并不是所有编程语言中的数组下标都从 0 开始,如 Pascal、FORTRAN、MATLAB。甚至有些语言支持负数下标,如 Python。