第5章

数组、矩阵和广义表

CHAPTER 5

本章学习目标

- 理解数组、矩阵和广义表的基本概念及存储结构
- 掌握广义表的基本操作实现
- 学会利用广义表解决应用问题

5.1　数组

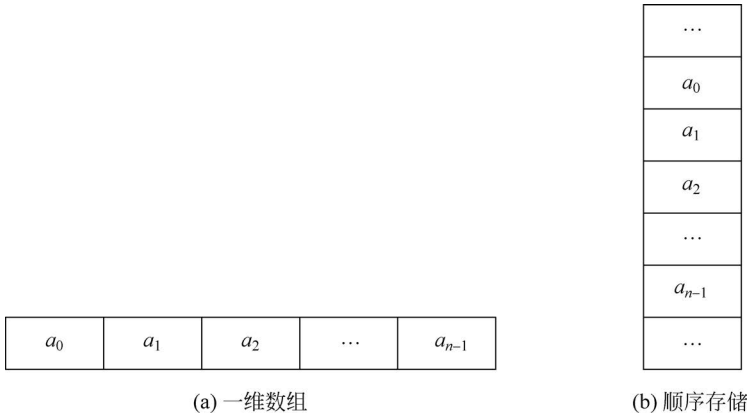

视频讲解

5.1.1　数组的定义

数组是一种比较常用的数据结构，几乎所有的程序设计语言都支持这种数据结构或将这种数据结构设定为语言的固有类型。数组可以看成线性表的推广。Java 语言支持数组，且对数组的维数没有严格的界限，但是三维以上的数组基本不常使用，使用最多的是一维、二维数组。此外，Java 语言也可以支持更复杂的数组。

5.1.2　数组的存储

数组一般不进行插入和删除操作，也就是说，数组一旦建立，结构中的元素个数和元素间的关系就不再发生变化。因此，一般都是采用顺序存储的方法来表示数组。一维数组的存储相对来说比较简单，即将数组元素 $a_0 \sim a_{n-1}$ 依次放在一组地址连续的存储单元中，如图 5-1 所示。



(a) 一维数组　　(b) 顺序存储

图 5-1　一维数组的存储

计算机的内存结构是一维(线性)地址结构，将多维数组存放(映射)到内存一维结构时，会出现次序约定问题。因此，必须按某种次序将多维数组元素排成线性序列，然后将该线性序列存放到内存中。

二维数组是最简单的多维数组，以图 5-2(a)为例说明多维数组存放(映射)到内存一维结构时的次序约定问题。二维数组通常有以下两种顺序存储方式。

(1) 行优先顺序(以行序为主序)。先存储第 1 行的元素，再存储第 2 行的元素，最后存储第 m 行的元素。存储次序为 a_{00}，a_{01}，…，$a_{0(n-1)}$，a_{10}，a_{11}，…，$a_{1(n-1)}$，…，$a_{(m-1)0}$，$a_{(m-1)1}$，…，$a_{(m-1)(n-1)}$，如图 5-2(b)所示。在 PASCAL、C 语言中，二维数组是按行优先顺序存储的。

(2) 列优先顺序(以列序为主序)。先存储第 1 列的元素，再存储第 2 列的元素，最后存储第 n 列的元素。存储次序为 a_{00}，a_{10}，…，$a_{(m-1)0}$，a_{01}，a_{11}，…，$a_{(m-1)1}$，…，$a_{0(n-1)}$，$a_{1(n-1)}$，…，$a_{(m-1)(n-1)}$，如图 5-2(c)所示。在 FORTRAN 语言中，数组是按列优先顺序存储的。

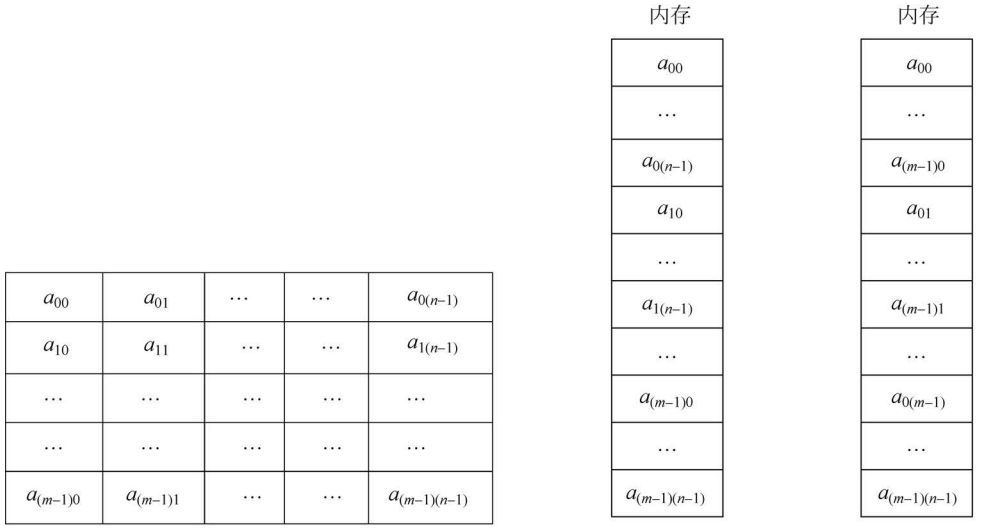


图 5-2 二维数组及其顺序存储图形式

由此可见,对于数组,一旦规定了它的维数和各维的长度,便可为它分配存储空间。反之,只要给出一组下标即可求得相应数组元素的存储位置。下面用以行序为主序的存储结构为例加以说明。

假设二维数组 $a[m..n]$的每个元素只占 L 个存储单元,“按行优先”存放数组,且首元素 a_{00} 的地址为 LOC(0, 0),求任意元素 a_{ij} 的地址。

求数组元素地址的基本原理:

a_{ij} 的起始地址=第一个元素的起始地址+该元素前面的元素个数×单位长度

第一个元素的起始地址是已知的,且每个元素的单位长度 L 也是已知的。此时,要解决的问题是 a_{ij} 前面的元素个数。

a_{ij} 排在第 $i+1$ 行、第 $j+1$ 列,前面的 i 行有 $n\times i$ 个元素。在第 $i+1$ 行,第 $j+1$ 个元素 a_{ij} 前面还有 j 个元素,则 a_{ij} 前面的元素有 $n\times i+j$ 个。由此得到如下地址计算公式:

$$\mathrm{LOC}(i,j)=\mathrm{LOC}(0,0)+(n\times i+j)L \tag{5-1}$$

推广到一般情况,可得 n 维数组的数据元素存储位置的计算公式:

$$\begin{aligned}\mathrm{LOC}(j_1,j_2,\cdots,j_n)&=\mathrm{LOC}(0,0,\cdots,0)+(b_2\times\cdots\times b_n\times j_1+\\&\quad b_3\times\cdots\times b_n\times j_2+\cdots+b_n\times j_{n-1}+j_n)L\\&=\mathrm{LOC}(0,0,\cdots,0)+\left(\sum_{i=1}^{n-1}j_i\prod_{k=i+1}^{n}b_k+j_n\right)L\end{aligned} \tag{5-2}$$

上式可缩写成:

$$\mathrm{LOC}(j_1,j_2,\cdots,j_n)=\mathrm{LOC}(0,0,\cdots,0)+\sum_{i=1}^{n}c_ij_i \tag{5-3}$$

其中,$c_n=L$,$c_{i-1}=b_i\times c_i$,$1<i\leqslant n$。

式(5-3)称为 n 维数组的映像函数。容易看出,数组元素的存储位置是其下标的线性函数,一旦确定了数组的各维的长度,c_i 就是常数。由于计算各个元素存储位置的时间

相等，所以存取数组中任意元素的时间也相等。将具有这一特点的存储结构称为随机存储结构。

从内存的角度来说，Java 是没有二维（多维）数组的。所谓的 Java 二维数组的本质是存放数组的数组，所以二维数组的本质还是一维数组，只是数组元素是引用，数组中的每一个元素都指向了另一个一维数组而已。

```
int[ ][ ] arr = new int[3][ ];
arr[0] = new int[3];
arr[1] = new int[5];
arr[2] = new int[4];
```

以上 4 条语句创建了一个逻辑意义上的二维数组，该数组共 3 行。其中，第 1 行为 3 个数据元素，第 2 行为 5 个数据元素，第 3 行为 4 个数据元素，arr[1][2]代表数组第 2 行、第 3 列的数据元素。但是，arr 实质上是一个一维数组，3 个数组元素都是引用，分别指向了 3 个一维数组，如图 5-3 所示。在图 5-3 中，d69c、e922、154f 等 16 进制数表示内存单元的编号，即引用。

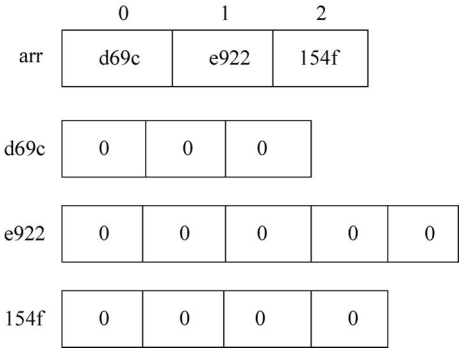


图 5-3　Java 中的二维数组

5.2　矩阵

在科学与工程计算问题中，矩阵是一种常用的数学对象。在使用高级语言进行编程时，通常将一个矩阵描述为一个二维数组。这样，可以对其元素进行随机存取，各种矩阵运算也非常简单。

在一些特殊矩阵中，元素呈某种规律分布或者矩阵中有大量的零元素（如对称矩阵、三角矩阵、对角矩阵、稀疏矩阵等），如果仍用二维数组存储，则会造成极大的浪费（尤其是在处理高阶矩阵时）。为了节省存储空间，可以对这类特殊矩阵进行压缩存储。

5.2.1　特殊矩阵的压缩存储

1. 对称矩阵的压缩存储

若 n 阶方阵 $\boldsymbol{A}$ 中的元素满足特性 $a_{ij}=a_{ji}$，其中 $1\leqslant i,j\leqslant n$，则称 $\boldsymbol{A}$ 为 n 阶对称矩阵，一个 5 阶对称矩阵如图 5-4 所示。

$$\begin{bmatrix} 1 & 5 & 1 & 3 & 7 \\ 5 & 0 & 8 & 0 & 0 \\ 1 & 8 & 9 & 2 & 6 \\ 3 & 0 & 2 & 5 & 1 \\ 7 & 0 & 6 & 1 & 3 \end{bmatrix}$$

图 5-4　5 阶对称矩阵

对称矩阵关于主对角线对称，只需存储上三角或下三角中的元素。在此假设存储下三角中的元素，则对称矩阵的压缩存储就是将矩阵下三角中的元素按行优先（也可以按列优

先)的顺序存储到一维数组中,如图 5-5 所示。

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

图 5-5 对称矩阵的压缩存储

对于对称矩阵的压缩存储,需要探讨以下 3 个问题。

(1) 假设对称矩阵 $\boldsymbol{A}$ 有 n 行 n 列,对称矩阵 $\boldsymbol{A}$ 的压缩存储需要一个多大的一维数组?

对称矩阵 $\boldsymbol{A}$ 的下三角中有多少元素就需要一个多大的一维数组。那么对称矩阵 $\boldsymbol{A}$ 的下三角中有多少元素呢? 第 1 行 1 个元素,第 2 行 2 个元素,第 3 行 3 个元素,……,第 i 行 i 个元素,第 n 行 n 个元素,共有 $1+2+\cdots+n=n(n+1)/2$ 个元素。因此,对称矩阵 $\boldsymbol{A}$ 的压缩存储需要一个 $n(n+1)/2$ 长度的一维数组,如图 5-6 所示。

$$
\begin{array}{llllllll}
a_{11} & & & & & & & \\
a_{21} & a_{22} & & & & & & \\
a_{31} & a_{32} & a_{33} & & & & & \\
\cdots & \cdots & \cdots & \cdots & & & & \\
a_{i1} & \cdots & a_{ij} & \cdots & a_{ii} & & & \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & & \\
a_{n1} & a_{n2} & \cdots & \cdots & \cdots & \cdots & a_{nn} &
\end{array}
$$

a_{11}	a_{21}	a_{22}	a_{31}	…	a_{n1}	…	a_{nn}

图 5-6 n 阶对称矩阵的下三角及其一维数组

(2) 假设对称矩阵中的元素定义为 double 型,压缩存储可以节省多少存储空间?

非压缩存储(即用二维数组存储)所用存储空间为 $8\times n\times n$ 字节,压缩存储所用存储空间$=8\times n(n+1)/2=4n(n+1)$字节,节省的存储空间$=8\times n\times n-4n(n+1)=4n(n-1)$字节。

(3) 对称矩阵中的元素 a_{ij}($i\geqslant j$ 时)在一维数组中的下标是什么?

要想知道元素 a_{ij} 在一维数组中的下标,必须先知道元素 a_{ij} 前面有多少个元素。元素 a_{ij} 前面有 $i-1$ 行,共有 $1+2+\cdots+i-1=i(i-1)/2$ 个元素;第 i 行前面有 $j-1$ 个元素,共有 $i(i-1)/2+j-1$ 个元素。因此,元素 a_{ij} 是一维数组的第 $i(i-1)/2+j$ 个元素,它的下标是 $i(i-1)/2+j-1$。

压缩对称矩阵的代码实现如下。

```
public class symmetrymatrix {
    private double[] A;
    // A 是一维数组,存放对称矩阵的下三角(包括对角线)中的数据元素
    private int n;
    // n 是对称矩阵阶数
    public symmetrymatrix(int n) {
        // 构造一个数据元素全为 0 的 n 阶对称矩阵
        this.n = n;
        A = new double[n * (n + 1) / 2];
    }
    public int getDimension() {
        return n;
    }
    public double[] getArray() {
        return A;
```

```
    }
    public double get(int i, int j) {
        // i,j 合法取值范围为[1,n]
        if (i < 0 || i > n)
            throw new IllegalArgumentException("非法参数");
        if (j < 0 || j > n)
            throw new IllegalArgumentException("非法参数");
        if (i >= j)
            return A[i * (i - 1) / 2 + j - 1];
        else
            return A[j * (j - 1) / 2 + i - 1];
    }
    public void set(int i, int j, double s) {
        // i,j 合法取值范围为[1,n]
        if (i < 0 || i > n)
            throw new IllegalArgumentException("非法参数");
        if (j < 0 || j > n)
            throw new IllegalArgumentException("非法参数");
        if (i >= j)
            A[i * (i - 1) / 2 + j - 1] = s;
        else
            A[j * (j - 1) / 2 + i - 1] = s;
    }
    public symmetrymatrix plus(symmetrymatrix B) {
        // 对称矩阵相加
        checkMatrixDimensions(B);
        symmetrymatrix X = new symmetrymatrix(n);
        double C[] = X.getArray();
        for (int i = 0; i < n * (n + 1) / 2; i++) {
            C[i] = A[i] + B.A[i];
        }
        return X;
    }

    public symmetrymatrix minus(symmetrymatrix B) {
        // 对称矩阵相减
        checkMatrixDimensions(B);
        symmetrymatrix X = new symmetrymatrix(n);
        double C[] = X.getArray();
        for (int i = 0; i < n * (n + 1) / 2; i++) {
            C[i] = A[i] - B.A[i];
        }
        return X;
    }
    private void checkMatrixDimensions(symmetrymatrix B) {
        if (B.n != n) {
            throw new IllegalArgumentException("非法参数");
        }
    }
}
```

除了矩阵的加减外，矩阵的运算还包括矩阵的转置、矩阵求逆、矩阵的乘除等。可以将矩阵的每一种运算作为一个方法，并分别加到类中。例如，在对称矩阵压缩存储结构下实现两个对称矩阵相乘，代码如下。

```
    public double[][] multiply(symmetrymatrix B) {
```

```
        // 对称矩阵相乘
        checkMatrixDimensions(B);
        double c[][] = new double[n][n];
        // 对称矩阵相乘结果不一定是对称矩阵,积矩阵用二维数组存储
        for (int i = 1; i <= n; ++i)
            for (int j = 1; j <= n; ++j) {
                c[i - 1][j - 1] = 0;          // 二维数组中的行下标、列下标从 0 开始
                for (int k = 1; k <= n; ++k)
                    c[i - 1][j - 1] = c[i - 1][j - 1] + this.get(i, k) * B.get(k, j);
            }
        return c;
    }
```

其余运算的代码实现,读者可作为练习自行完成。

2. 三角矩阵的压缩存储

若 n 阶方阵的下(上)三角(不包括对角线)中的元素均为常量 c,则称为上(下)三角矩阵。n 阶下三角矩阵如图 5-7 所示。

$$\begin{bmatrix} a_{11} & c & c & c & c & c & c \\ a_{21} & a_{22} & c & c & c & c & c \\ a_{31} & a_{32} & a_{33} & c & c & c & c \\ \cdots & \cdots & \cdots & \cdots & c & c & c \\ a_{i1} & \cdots & a_{ij} & \cdots & a_{ii} & c & c \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & c \\ a_{n1} & a_{n2} & \cdots & \cdots & \cdots & \cdots & a_{nn} \end{bmatrix}$$

图 5-7 下三角矩阵

本书以下三角矩阵为例探讨三角矩阵的压缩存储。下三角矩阵的压缩存储就是将矩阵下三角中的元素按行优先(也可以按列优先)的顺序存储到一维数组中,并将常量 c 存储到数组的最后一个位置。

a_{11}	a_{21}	a_{22}	a_{31}	…	a_{n1}	…	a_{nn}	c

对于下三角矩阵的压缩存储,同样需要探讨以下 3 个问题。

(1) 假设下三角矩阵 $\boldsymbol{A}$ 有 n 行 n 列,下三角矩阵 $\boldsymbol{A}$ 的压缩存储需要一个多大的一维数组?

下三角矩阵 $\boldsymbol{A}$ 的下三角(包括对角线)中的非常量元素个数:第 1 行 1 个元素,第 2 行 2 个元素,第 3 行 3 个元素,……,第 i 行 i 个元素,第 n 行 n 个元素,共有 $1+2+\cdots+n=n(n+1)/2$ 个元素。因此,下三角矩阵 $\boldsymbol{A}$ 的压缩存储需要一个 $n(n+1)/2+1$ 长度的一维数组。

(2) 如果下三角矩阵中的元素为 double 型,下三角矩阵的压缩存储节省了多少存储空间?

非压缩存储(即用二维数组存储)所用存储空间为 $8\times n\times n$ 字节,压缩存储所用存储空间为 $4n(n+1)+8$ 字节,可节省 $4n(n-1)-8$ 字节的存储空间。

(3) 下三角矩阵中的元素 a_{ij}($i\geqslant j$ 时)在一维数组中的下标是什么?

元素 a_{ij} 前面有 $i-1$ 行,共有 $1+2+\cdots+i-1=i(i-1)/2$ 个元素;第 i 行前面有 $j-1$ 个元素,共有 $i(i-1)/2+j-1$ 个元素。因此,元素 a_{ij} 是一维数组的第 $i(i-1)/2+j$

个元素，它的下标是 $i(i-1)/2+j-1$。常量 c 是一维数组中的最后一个元素，它的下标是 $n(n+1)/2$。

压缩下三角矩阵的代码实现如下。

```
public class lowertriangularmatrix {
    private double[] A;
    // A 是一维数组,存放矩阵下三角(包括对角线)中的数据元素及上三角区域的常量
    private int n;        // n 是下三角矩阵阶数
    public lowertriangularmatrix(int n) {
        // 构造一个数据元素全为 0 的 n 阶下三角矩阵
        this.n = n;
        A = new double[n * (n + 1) / 2 + 1];
    }
    public int getDimension() {
        return n;
    }
    public double get(int i, int j) {
        // i,j 合法取值范围为[1,n]
        if (i < 0 || i > n)
            throw new IllegalArgumentException("Illegal Argument");
        if (j < 0 || j > n)
            throw new IllegalArgumentException("Illegal Argument");
        if (i >= j)
            return A[i * (i - 1) / 2 + j - 1];
        else
            return A[n * (n + 1) / 2];
    }
    public void set(int i, int j, double s) {
        // i,j 合法取值范围为[1,n]
        if (i < 0 || i > n)
            throw new IllegalArgumentException("Illegal Argument");
        if (j < 0 || j > n)
            throw new IllegalArgumentException("Illegal Argument");
        if (i >= j)
            A[i * (i - 1) / 2 + j - 1] = s;
        else
            A[n * (n + 1) / 2] = s;
    }
    public double[] getArray() {
        return A;
    }

    public lowertriangularmatrix plus(lowertriangularmatrix B) {
         //相加
        checkMatrixDimensions(B);
        lowertriangularmatrix X = new lowertriangularmatrix(n);
        double C[] = X.getArray();
        for (int i = 0; i <= n * (n + 1) / 2; i++) {
            C[i] = A[i] + B.A[i];
        }
        return X;
    }
    public lowertriangularmatrix minus(lowertriangularmatrix B) {
         //相减
        checkMatrixDimensions(B);
        lowertriangularmatrix X = new lowertriangularmatrix(n);
```

```
            double C[] = X.getArray();
            for (int i = 0; i <= n * (n + 1) / 2; i++) {
                C[i] = A[i] - B.A[i];
            }
            return X;
        }
        private void checkMatrixDimensions(lowertriangularmatrix B) {
            if (B.n != n) {
                throw new IllegalArgumentException("Matrix dimensions must agree.");
            }
        }
    }
```

在下三角矩阵压缩存储结构下可以实现矩阵的转置、矩阵求逆、矩阵的乘除等运算,读者可作为练习自行完成。

3. 对角矩阵的压缩存储

在对角矩阵中,除了主对角线和主对角线上方或下方若干条对角线上的元素外,其余元素皆为零,即所有的非零元素集中在以主对角线为中心的带状区域中,如图 5-8 所示。在如图 5-8 所示的矩阵中,只有主对角线及其上、下两条对角线的数据元素不为 0,该矩阵为三对角矩阵。

$$\begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & \cdots & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & \cdots & 0 \\ 0 & 0 & \cdots & \cdots & \cdots & 0 & 0 \\ 0 & \cdots & 0 & a_{i(i-1)} & a_{ii} & a_{i(i+1)} & 0 \\ 0 & \cdots & 0 & 0 & a_{(n-1)(n-2)} & a_{(n-1)(n-1)} & a_{(n-1)n} \\ 0 & \cdots & 0 & 0 & 0 & a_{n(n-1)} & a_{nn} \end{bmatrix}$$

图 5-8　三对角矩阵

在该三对角矩阵中,非零元素仅出现在主对角线上(a_{ii},$1 \leqslant i \leqslant n$)、主对角线上方的对角线上($a_{i(i+1)}$,$1 \leqslant i \leqslant n-1$)、主对角线下方的对角线上($a_{(i+1)i}$,$1 \leqslant i \leqslant n-1$)。显然,当 $|i-j|>1$ 时,元素 $a_{ij}=0$。

由此可知,一个 k 对角矩阵(k 为奇数)$\boldsymbol{A}$ 满足条件:当 $|i-j|>(k-1)/2$ 时,$a_{ij}=0$。

对角矩阵可按行优先顺序或对角线顺序将其压缩存储到一个数组中,并且也能找到每个非零元素和数组下标的对应关系。仍然以三对角矩阵为例进行讨论。

当 $i=1$,$j=1$、2,或 $i=n$,$j=n-1$、n,或 $1<i<n$ 时,除 $j=i-1$、i、$i+1$ 的元素 a_{ij} 外,其余元素都是 0。

对于这种矩阵,当以“按行优先顺序”压缩存储时,第 1 行和第 n 行均有两个非零元素,其余每行的非零元素都是 3 个,则需存储的元素个数为 $3n-2$。三对角矩阵的压缩存储形式如图 5-9 所示。

k	0	1	2	3	4	5	6	7	…	$3n-4$	$3n-3$
a	a_{11}	a_{12}	a_{21}	a_{22}	a_{23}	a_{32}	a_{33}	a_{34}	…	$a_{n(n-1)}$	a_{nn}

图 5-9　三对角矩阵的压缩存储

数组 a 中的元素 $a[k]$与三对角矩阵中的元素 a_{ij} 存在一一对应的关系。在 a_{ij} 之前有 $i-1$ 行,有 $3i-4$ 个非零元素,在第 i 行有 $j-i+1$ 个非零元素,共有 $2i+j-3$ 个非零元

素。这样，非零元素 a_{ij} 在数组中的下标为 $2i+j-3$。在图 5-9 中，a_{34} 在数组中的下标 $k=2\times3+4-3=7$，对应 $a[7]$。

压缩三对角矩阵的代码实现，读者可作为练习自行完成。

上述各种特殊矩阵的非零元素的分布都是有规律的，因此总能找到一种方法将它们压缩存储到一个一维数组中，并且一般都能找到矩阵中的元素与该一维数组元素的对应关系，通过这个关系仍能对矩阵的元素进行随机存取。

5.2.2　稀疏矩阵的压缩存储


视频讲解

如果矩阵中只有少量的非零元素，并且这些非零元素在矩阵中的分布没有规律，则称为随机稀疏矩阵，简称为稀疏矩阵。稀疏矩阵示例如图 5-10 所示。

假设在 $m\times n$ 的矩阵中有 t 个非零元素，令 $\delta=t/mn$，称 δ 为矩阵的稀疏因子。图 5-10 所示矩阵的稀疏因子为 $7/30\approx0.23=23\%$，该矩阵并不属于严格意义上的稀疏矩阵，只有稀疏因子小于 5%的矩阵才能称为稀疏矩阵。

稀疏矩阵的压缩存储：只存储非零元素。

描述非零元素的信息：该非零元素所在的行数，该非零元素所在的列数，该非零元素的值。

整个稀疏矩阵的表示：每个非零元素由(row，col，value)唯一确定，整个矩阵可表示为一个三元组表，如图 5-11 所示。

$$\begin{bmatrix} 3 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 \\ 1 & 0 & 4 & 0 & 6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \end{bmatrix}$$

图 5-10　稀疏矩阵示例

row	col	value
1	1	3
1	4	5
2	3	−2
3	1	1
3	3	4
3	5	6
5	3	−1

图 5-11　稀疏矩阵三元组表

1. 三元组顺序表

在 Java 语言中，三元组表的顺序存储就是将三元组表存放到一维数组中，该种存储结构简称为三元组顺序表。

假设图 5-10 所示的稀疏矩阵 ***A*** 的数据元素是 double 型(8 字节)，row、col 是 int 型(4 字节)，value 是 double 型(8 字节)，则稀疏矩阵 ***A*** 采用三元顺序表存储节省了多少存储空间？

稀疏矩阵 ***A*** 采用二维数组存储所用的存储空间=5×6×8=240 字节，采用三元顺序表存储所用的存储空间=7×16=112 字节，则节省的存储空间=240−112=128 字节。

由此可知，稀疏矩阵稀疏因子越小，节省的存储空间越多。

如何表示三元组表的一行？三元组表的每一行都是由 3 个数据项组成的，考虑用类表示，代码如下。

```
class Triple {      // 三元组表的每一行都是由 3 个数据项组成的,考虑用类表示
    public int row, col;
```

```
    public double value;
    public Triple() {
        row = 0;
        col = 0;
        value = 0;
    }
    public Triple(int row, int col, int value) {
        this.row = row;
        this.col = col;
        this.value = value;
    }
}
```

三元组顺序表类的代码实现如下。

```
class TMatrix {
    public Triple[] data;          // 非零元素三元组表
    public int rn, cn, tn;         // 矩阵的行数、列数和非零元素个数
    public TMatrix(int size) {
        data = new Triple[size];
        for (int i = 0; i < size; i++) {
            data[i] = new Triple();
        }
    }
    public TMatrix(int rn, int cn, int size) {
        // 目前无非零元素,全是零
        data = new Triple[size];
        for (int i = 0; i < size; i++) {
            data[i] = new Triple();
        }
        this.rn = rn;
        this.cn = cn;
        this.tn = 0;
    }
    public double get(int i, int j) {
        // i 的合法取值范围为[1,rn]
        // j 的合法取值范围为[1,cn]
        if (i < 1 || i > rn || j < 1 || j > cn)
            throw new IllegalArgumentException("行标列标非法");
        int k;
        for (k = 0; k < tn; k++) {
            if (data[k].row == i && data[k].col == j)
                break;
        }
        if (k < tn)                // 非 0
        {
            return data[k].value;
        } else
            return 0;
    }
    public void set(int i, int j, double s) {
        if (i < 1 || i > rn || j < 1 || j > cn)
            throw new IllegalArgumentException("行标列标非法");
        // i 的合法取值范围为[1,rn]
        // j 的合法取值范围为[1,cn]
        int k = 0;
        for (k = 0; k < tn; k++) {
```

```
            if (data[k].row == i && data[k].col == j)
                break;
        }
        // 分 4 种情况
        // (1)s != 0 &&k < tn,表示矩阵中第 i 行第 j 列的元素非零,新设置数值 s 非零,此时只要
        // 令 data[k].value = s 即可
        // (2)s != 0 &&k == tn,表示矩阵中第 i 行第 j 列的元素是零,新设置数值 s 非零,此时需
        // 要插入( i, j, s)到数组中
        // (3)s == 0 &&k < tn,表示矩阵中第 i 行第 j 列的元素非零,新设置数值是零,此时需要从
        // 数组中删除 data[k],即( i, j,value)从 data[k + 1]到 data[tn - 1]依次上移
        // (4)s == 0 &&k == tn,表示矩阵中第 i 行第 j 列的元素是零,新设置数值 s 也是零,此时
        // 不用做任何处理
        if (s != 0 && k < tn)
            data[k].value = s;
        if (s != 0 && k == tn) {
            // 判断数组中有没有空间
            if (tn == data.length)     // 如果没有空间,则抛出异常; 否则,进行插入
            {
                throw new RuntimeException("已满,不能插入");
            } else {
                int position = 0;
                int k1;
                for (k1 = 0; k1 < tn; k1++) {
                    if (data[k1].row < i)
                        position++;
                    if (data[k1].row == i && data[k1].col < j)
                        position++;
                    if (data[k1].row > i)
                        break;
                }
                // 找到合适的插入位置
                // 从 tn - 1 至 position 依次后移一位
                for (int k2 = tn - 1; k2 >= position; k2 -- )
                    data[k2 + 1] = data[k2];
                data[position].row = i;
                data[position].col = j;
                data[position].value = s;
                tn++;
            }
        }
        if(s == 0 &&k < tn){
            //从数组中删除 data[k]
            for (int k2 = k; k2 < tn - 1; k2++)
                data[k2] = data[k2 + 1];
            tn -- ;
        }
    }
```

当稀疏矩阵采用三元组表存储结构时,不能对矩阵的元素进行随机存取,此时需要利用类中的 set 和 get 方法对矩阵的数据元素进行存取。下面讨论在这种压缩存储结构下的矩阵转置运算。

假设有一个 $m \times n$ 的矩阵 $\boldsymbol{A}$,它的转置 $\boldsymbol{B}$ 是一个 $n \times m$ 的矩阵,且 $\boldsymbol{B}[i][j]=\boldsymbol{A}[j][i]$ $(0 \leqslant i \leqslant n, 0 \leqslant j \leqslant m)$,即 $\boldsymbol{B}$ 的行是 $\boldsymbol{A}$ 的列,$\boldsymbol{B}$ 的列是 $\boldsymbol{A}$ 的行。设稀疏矩阵 $\boldsymbol{A}$ 是按行优先顺序压缩存储在三元组表 $\boldsymbol{A}$.data 中的,若只是简单地通过交换 $\boldsymbol{A}$.data 中 i 和 j 的内容得到

三元组表 ***B***.data,则 ***B***.data 将是一个按列优先顺序存储的稀疏矩阵 ***B***。要得到按行优先顺序存储的 ***B***.data,就必须重新排列三元组表 ***B***.data 中元素的顺序。

求转置矩阵的基本算法思想如下。

① 将矩阵的行、列下标值交换,即将三元组表中的行、列位置值 i、j 相互交换。

② 重排三元组表中元素的顺序,即交换后仍然是按行优先顺序排序的。

(1) 方法一。

算法思想:根据稀疏矩阵 ***A*** 的三元组表 ***A***.data 中的列次序,依次找到相应的三元组,并将其存入 ***B***.data 中。每次寻找三元组都需要完整扫描三元组表 ***A***.data,找到之后自然就成为按行优先的转置矩阵的压缩存储表示。

按方法一求转置矩阵的算法实现如下,其中 this 代表当前矩阵 ***A***。

```
TMatrix TransMatrix() {
    int p, q, col;
    TMatrix B = new TMatrix(this.data.length);
    B.rn = this.cn;
    B.cn = this.rn;
    B.tn = this.tn;
    /* 设置三元组表 B.data 的行数、列数和非零元素个数 */
    if (B.tn == 0)
        System.out.println("The Matrix = 0\n");
    else {
        q = 0;
        for (col = 1; col <= this.cn; col++)
            /* 每循环一次即找到转置后的一个三元组 */
            for (p = 0; p < B.tn; p++)
                /* 循环次数是非零元素个数 */
                if (this.data[p].col == col) {
                    B.data[q].row = this.data[p].col;
                    B.data[q].col = this.data[p].row;
                    B.data[q].value = this.data[p].value;
                    q++;
                }
    }
    return B;
}
```

算法分析:本算法的主要工作是在 p 和 col 的两个循环中完成的,故算法的时间复杂度为 $O(\text{cn}\times\text{tn})$,即与矩阵的列数和非零元素个数的乘积成正比。

稀疏矩阵如果采用普通存储方式(即二维数组),则其转置算法代码如下。

```
for(int col = 0; col < cn ;++col)
    for(int row = 0 ; row < rn ;++row)
          b[col][row] = a[row][col] ;
```

此时,其时间复杂度为 $O(\text{cn}\times\text{rn})$。

当非零元素的个数 tn 与 $\text{rn}\times\text{cn}$ 为相同数量级时,算法 TransMatrix 的时间复杂度为 $O(\text{rn}\times\text{cn}^2)$。

由此可见,该算法虽然节省了存储空间,但时间复杂度却大大增加,所以算法 TransMatrix 只适合于稀疏矩阵中非零元素的个数 tn 远远小于 $\text{rn}\times\text{cn}$ 的情况。

（2）方法二。

算法思想：直接按照稀疏矩阵 ***A*** 的三元组表 ***A***.data 的次序依次顺序转换，并将转换后的三元组放置于三元组表 ***B***.data 的恰当位置。

算法分析：若能预先确定原矩阵 ***A*** 中每一列的（即 ***B*** 中每一行）第一个非零元素在 ***B***.data 中应有的位置（下标值，从 0 开始计数），则在转置时就可以直接放在 ***B***.data 中的恰当位置。因此，应先求得 ***A*** 中每一列的非零元素个数。

附设两个辅助数组：num[ ]和 cpot[ ]。

① num[col]：统计 ***A*** 的第 col 列中非零元素的个数。

② cpot[col]：指示 ***A*** 的第 col 列中第一个非零元素在 ***B***.data 中的恰当位置（下标值，从 0 开始计数）。

显然有以下关系成立。

$$\begin{cases} \text{cpot}[1]=0 \\ \text{cpot}[\text{col}]=\text{cpot}[\text{col}-1]+\text{num}[\text{col}-1], 2\leqslant \text{col}\leqslant \boldsymbol{A}.\text{cn} \end{cases}$$

对于图 5-12 中的稀疏矩阵 ***A***，可以求得 num[col]和 cpot[col]的值如表 5-1 所示。

$$\boldsymbol{A}=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 24 & 0 & 0 & 2 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -7 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 & 0 & 0 \end{bmatrix}$$

7			rm行数
8			cn列数
9			tn元素个数
1	2	12	
1	3	9	
3	1	−3	
3	8	4	
4	3	24	
4	6	2	
5	2	18	
6	7	−7	
7	4	−6	
row	col	val	

(a) 稀疏矩阵示例　　(b) 矩阵***A***的三元组顺序表

图 5-12　稀疏矩阵及三元组顺序表

表 5-1　num[col]和 cpot[col]的值表

col	1	2	3	4	5	6	7	8
num[col]	1	2	2	1	0	1	1	1
cpot[col]	0	1	3	5	6	6	7	8

快速转置算法代码如下，其中 this 代表当前矩阵 ***A***。

```
TMatrix FastTransMatrix() {
    int p, q, col, k;
    int[] num = new int[this.cn];
    int[] cpot = new int[this.cn];
    TMatrix B = new TMatrix(this.data.length);
    B.rn = this.cn;
    B.cn = this.rn;
```

```
        B.tn = this.tn;
        /* 设置三元组表 B.data 的行数、列数和非零元素个数 */
        if (B.tn == 0)
            System.out.println("The Matrix = 0\n");
        else {
            for (col = 0; col < this.cn; ++col)
                num[col] = 0;
            /* 下标 col 从 0 开始计数 */
            /* 向量 num[]初始化为 0 */
            for (k = 0; k < this.tn; ++k)
                ++num[this.data[k].col - 1];
            /* 求原矩阵中每一列的非零元素个数 */
            for (cpot[0] = 0, col = 1; col < this.cn; ++col)
                cpot[col] = cpot[col - 1] + num[col - 1];
            /* 求第 col 列中第一个非零元素在 B.data 中的序号 */
            for (p = 0; p < this.tn; ++p) {
                col = this.data[p].col - 1;
                q = cpot[col];
                B.data[q].row = this.data[p].col;
                B.data[q].col = this.data[p].row;
                B.data[q].value = this.data[p].value;
                ++cpot[col];
            }
        }
        return B;
    }
```

该算法比起前一个算法多用了两个辅助数组。从时间上看,算法中有 4 个并列的单循环,循环次数分别是 cn 和 tn,因而其时间复杂度为 $O(cn+tn)$。当矩阵的非零元素个数 tn 与 cn×rn 为相同数量级时,算法的时间复杂度为 $O(cn\times rn)$。

在三元组顺序表存储结构下,可以进行矩阵求逆、矩阵的加减、矩阵的乘除等运算,读者可作为练习自行实现。

2. 行逻辑链接的三元组顺序表

为了便于随机存取任意一行的非零元素,需要知道每一行的第一个非零元素在三元组表中的位置(下标值,从 0 开始计数)。因此,可将指示每行第一个非零元素在三元组表中位置的数组固定在稀疏矩阵的三元组表中,以此指示“行”的信息,从而得到另一种顺序存储结构:行逻辑链接的三元组顺序表,其类型描述如下。

```
public class RLSMatrix {
    public Triple[] data;           /* 非零元素的三元组表 */
    public int[] rpos;              /* 各行第一个非零元素的位置表 */
    public int rn, cn, tn;          /* 矩阵的行数、列数和非零元素个数 */

    public RLSMatrix(int size, int rn, int cn) {
        data = new Triple[size];
        for (int i = 0; i < size; i++)
            data[i] = new Triple();
        this.rn = rn;
        this.cn = cn;
        this.tn = 0;
        rpos = new int[this.rn];
    }

    public RLSMatrix(Triple[] data, int rn, int cn, int tn) {
```

```
        this.data = data;
        this.rn = rn;
        this.cn = cn;
        this.tn = tn;
        rpos = new int[rn];
        int[] num = new int[rn];    // 各行的非零元素个数
        int i;
        if (tn == 0)
            System.out.println("The Matrix A = 0\n");
        else {
            for (i = 0; i < rn; ++i)
                num[i] = 0;
            /* 将向量 num[]初始化为 0 */
            for (int k = 0; k < tn; ++k)
                ++num[data[k].row - 1];
            /* 求矩阵中每一行非零元素的个数 */
            for (rpos[0] = 0, i = 1; i < rn; ++i)
                rpos[i] = rpos[i - 1] + num[i - 1];
            /* 求第 i 行中第一个非零元素在 data 中的序号 */
        }
    }
    public double get(int i, int j) {
        // i 的合法取值范围为[1,rn]
        // j 的合法取值范围为[1,cn]
        if (i < 1 || i > rn || j < 1 || j > cn)
            throw new IllegalArgumentException("行标列标非法");
        int k;
        for (k = rpos[i - 1]; k < rpos[i]; k++) {
            if (data[k].col == j)
                break;
        }
        if (k < rpos[i])              // 非零
            return data[k].value;
        else
            return 0;
    }

    public void set(int i, int j, double s) {
        if (i < 1 || i > rn || j < 1 || j > cn)
            throw new IllegalArgumentException("行标列标非法");
        // i 的合法取值范围为[1,rn]
        // j 的合法取值范围为[1,cn]
        int k;
        for (k = rpos[i - 1]; k < rpos[i]; k++) {
            if (data[k].col == j)
                break;
        }
        // 分 4 种情况
        // (1)s != 0 &&k < rpos[i],表示矩阵中第 i 行第 j 列的元素非零,新设置数值 s 非零,此时
        // 只要令 data[k].value = s 即可
        // (2)s != 0 &&k == rpos[i],表示矩阵中第 i 行第 j 列的元素是零,新设置数值 s 非零,此
        // 时需要插入(i, j, s)到数组中,并修改 rpos
        // (3)s == 0 &&k < rpos[i],表示矩阵中第 i 行第 j 列的元素非零,新设置数值是零,此时需
        // 要从数组中删除(i, j, value)并修改 rpos
        // (4)s == 0 &&k == rpos[i],表示矩阵中第 i 行第 j 列的元素是零,新设置数值 s 也是零,
        // 此时不用做任何处理
        if (s != 0 && k < rpos[i])
            data[k].value = s;            // 修改此元素的值
```

```
            if (s != 0 && k == rpos[i]) {
                // 判断数组中有没有空间
                if (tn == data.length)          // 如果没有空间,则抛出异常; 否则,进行插入
                {
                    throw new RuntimeException("已满,不能插入");
                } else {
                    int position = rpos[i - 1];
                    int k1;
                    for (k1 = rpos[i - 1]; k1 < rpos[i]; k1++) {
                        if( data[k1].col < j)
                            position++;
                        if (data[k1].col > j)
                            break;
                    }
                    // 找到合适的插入位置
                    // 从 tn - 1 至 position 依次后移一位
                    for (int k2 = tn - 1; k2 >= position; k2 -- )
                        data[k2 + 1] = data[k2];
                    data[position].row = i;
                    data[position].col = j;
                    data[position].value = s;
                    tn++;
                    for(int k3 = i;k3 < rn;k3++)
                        rpos[k3]++;
                }
            }
            if (s != 0 && k == rpos[i])
            {
                //从数组中删除 data[k]
                for (int k2 = k; k2 < tn - 1; k2++)
                    data[k2] = data[k2 + 1];
                tn -- ;
                for(int k3 = i;k3 < rn;k3++)
                    rpos[k3] -- ;
            }
        }
    }
```

当稀疏矩阵采用这种行逻辑链接的三元组顺序表存储结构时,同样不能对矩阵的元素进行随机存取,此时利用类中的 set 和 get 方法对矩阵的数据元素进行存取。下面讨论在这种行逻辑链接的三元组顺序表存储结构下的矩阵相乘运算。

矩阵的乘法定义如下。

设有两个矩阵:$\boldsymbol{A}=(a_{ij})_{m\times n}$,$\boldsymbol{B}=(b_{ij})_{n\times t}$,则$\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}=(c_{ij})_{m\times t}$,其中,$c_{ij}=\sum a_{ik}\times b_{kj}$,$1\leqslant k\leqslant n$,$1\leqslant i\leqslant m$,$1\leqslant j\leqslant t$。

如果矩阵采用二维数组存储结构(行、列下标均从 0 开始),则可用的经典算法是三重循环,代码如下。

```
for(i = 1;i <= m;++i)
    for(j = 1;j <= t;++j){
    c[i - 1][j - 1] = 0;
    for (k = 1; k <= n; ++k)
        c[i - 1][j - 1] = c[i - 1][j - 1] + a[i - 1][k - 1] * b[k - 1][j - 1];
}
```

假设两个稀疏矩阵 $\boldsymbol{A}=(a_{ij})_{m\times n}$ 和 $\boldsymbol{B}=(b_{ij})_{n\times t}$ 采用行逻辑链接的三元组顺序表存储

结构，即 ***A***.data 按行优先顺序存储矩阵 ***A*** 中的非零元素，***A***.tn 表示矩阵 ***A*** 中的非零元素个数，***A***.rn＝m 表示稀疏矩阵 ***A*** 的行数，***A***.cn＝n 表示稀疏矩阵 ***A*** 的列数，***A***.rpos 存储每一行的第一个非零元素在 ***A***.data 数组中的下标；***B***.data 按行优先顺序存储矩阵 ***B*** 中的非零元素，***B***.tn 表示矩阵 ***B*** 中的非零元素个数，***B***.rn＝n 表示稀疏矩阵 ***B*** 的行数，***B***.cn＝t 表示稀疏矩阵 ***B*** 的列数，***B***.rpos 存储每一行的第一个非零元素在 ***B***.data 数组中的下标。

稀疏矩阵相乘算法思想如下。

对于 ***A*** 中的每个元素 ***A***.data[p](p＝0，1，…，***A***.tn－1)，找到 ***B*** 中所有满足条件 ***A***.data[p].col＝***B***.data[q].row 的元素 ***B***.data[q]，求得 ***A***.data[p].value×***B***.data[q].value，该乘积是积矩阵元素 c_{ij} 中(i 的值是 ***A***.data[p].row，j 的值是 ***B***.data[q].col)的一部分，求得所有这样的乘积并累加求和就能得到 c_{ij}。

为得到非零的乘积，只要对 ***A***.data[]中每个元素(i，k，a_{ik})(1≤i≤***A***.rn，1≤k≤***A***.cn)，找到 ***B***.data[ ]中所有相应的元素(k，j，b_{kj})(1≤k≤***B***.rn，1≤j≤***B***.cn)相乘即可。因此必须知道矩阵 ***B*** 中第 k 行的所有非零元素，而 ***B***.rpos[]向量中提供了相应的信息。

B.rpos[row－1]指示了矩阵 ***B*** 的第 row 行中第一个非零元素在 ***B***.data[ ]中的位置(数组元素下标，从零开始计数)，***B***.rpos[row＋1－1]－1 指示了第 row 行中最后一个非零元素在 ***B***.data[ ]中的位置(数组元素下标，从 0 开始计数)。显然，最后一行中最后一个非零元素在 ***B***.data[]中的位置就是 ***B***.tn－1。

假设两个稀疏矩阵相乘的结果矩阵还是稀疏矩阵，仍旧采用行逻辑链接的三元组顺序表存储结构，两个稀疏矩阵相乘的算法代码如下。

```
static void MultsMatrix(RLSMatrix A, RLSMatrix B, RLSMatrix C)
/* 求矩阵 A 、B 的积 C = A * B,采用行逻辑链接的顺序表 */
{
    float[] ctemp = new float[B.cn];        // 行累加器
    int p, q, arow, ccol, brow, i, t, ta;   // ta 循环的终止条件
    if (A.cn != B.rn) {
        System.out.println("Error\n");
        return;
    } else {
        C.rn = A.rn;
        C.cn = B.cn;
        C.tn = 0;                           /* 初始化 C */
        if (A.tn * B.tn != 0)               /* C 是非零矩阵 */
        {
            for (arow = 1; arow <= A.rn; ++arow) {
                for (i = 0; i < B.cn; i++)
                    ctemp[i] = 0;
                /* 当前行累加器数组清零 */
                C.rpos[arow - 1] = C.tn;
                if (arow == A.rn)
                    ta = A.tn;
                else                        // ta 是循环的终止条件
                    ta = A.rpos[arow - 1 + 1]; //
                for (p = A.rpos[arow - 1]; p < ta; ++p)
                /* 遍历第 arow 行的每一个非零元素 */
                {
                    brow = A.data[p].col;
                    /* 找到元素在 B.data[]中的行号 */
                    if (brow < B.cn)
```

```
                        t = B.rpos[brow - 1 + 1];
                    else
                        t = B.tn;          // 设置循环变量的终止值
                    q = B.rpos[brow - 1];
                    for (; q < t; ++q) {   // 循环变量的起始值
                        ccol = B.data[q].col;
                        /* 积元素在 C 中的列号 */
                        ctemp[ccol - 1] += A.data[p].value * B.data[q].value;
                    }
                } /* 求出 C 中第 arow 行中的非零元素 */
                for (ccol = 1; ccol <= C.cn; ++ccol)
                    if (ctemp[ccol - 1] != 0) {
                        C.data[C.tn].row = arow;
                        C.data[C.tn].col = ccol;
                        C.data[C.tn].value = ctemp[ccol - 1];
                        C.tn++;
                    }
            }
        }
    }
}
```

如果能事先估算出乘积矩阵不是稀疏矩阵,则可以采用二维数组存储乘积矩阵,算法将更加简单。

对于行逻辑链接的三元组顺序表存储结构下的矩阵转置、矩阵求逆、矩阵加减等其他运算,读者可作为练习自行实现。

3. 十字链表

对于稀疏矩阵,当非零元素的个数和位置在操作过程中变化较大时,不宜采用三元组表的顺序存储结构,采用三元组表的链式存储结构会更恰当。

在链式存储中,矩阵的每一个非零元素用一个结点表示,该结点除了(row,col,value)数据部分外,还有行引用和列引用,如图 5-13 所示。其中,列引用指示同一列中的下一个非零元素,行引用指示同一行中的下一个非零元素。

矩阵 $\boldsymbol{B}$ 的链式存储结构如图 5-14 所示。

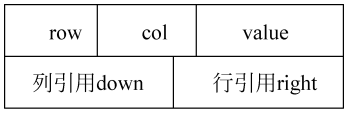


图 5-13 稀疏矩阵的链式存储结点

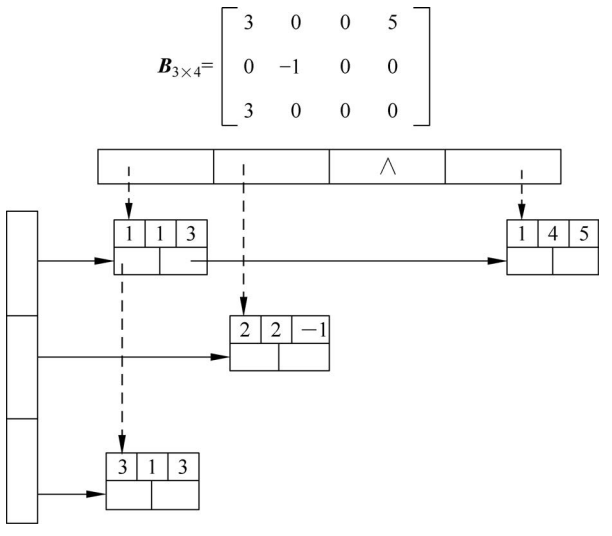


图 5-14 稀疏矩阵的链式存储结构

在链式存储结构中，每行有一个头引用，头引用指向各行的第一个非零元素结点，如果该行中没有非零元素结点，则头引用是空；每列有一个头引用，头引用指向各列的第一个非零元素结点，如果该列中没有非零元素结点，则头引用是空。

例如，矩阵 ***B*** 的第一行第一列的数据元素值是 3，列引用 down 指向同一列中的下一个非零元素 3，行引用 right 指向同一行中的下一个非零元素。

因为每个非零元素结点既处于某一行的单链表中，又处于某一列的单链表中，就像一个交叉的十字路口，所以将稀疏矩阵的这种链式存储结构称为十字链表。

结点类的定义如下。

```
class OLNode {                    /* 非零元素结点 */
    public int row, col;          /* 行号和列号 */
    public double value;          /* 元素值 */
    public OLNode down, right;
    public OLNode() {
        row = 0;
        col = 0;
        value = 0;
        down = null;
        right = null;
    }
    public OLNode(int row, int col, int value) {
        this.row = row;
        this.col = col;
        this.value = value;
        down = null;
        right = null;
   }
    public OLNode(int row, int col, int value, OLNode down, OLNode right){
        this.row = row;
        this.col = col;
        this.value = value;
        this.down = down;
        this.right = right;
    }
}
```

十字链表类 CrossLinkedListSparseMatrix 的成员变量及构造方法如下。

```
public class CrossLinkedListSparseMatrix {
    public int rn;                /* 矩阵的行数 */
    public int cn;                /* 矩阵的列数 */
    public int tn;                /* 非零元素总数 */
    public OLNode[] rhead;        // 行头结点数组
    public OLNode[] chead;        // 列头结点数组
    // 无参构造器
    public CrossLinkedListSparseMatrix() {
        rn = 0;
        cn = 0;
        tn = 0;
        rhead = new OLNode[rn];
        for (int i = 0; i < rn; ++i) {
            rhead[i] = new OLNode();
            rhead[i] = null;
        }
```

```
            chead = new OLNode[cn];
            for (int j = 0; j < cn; ++j) {
                chead[j] = new OLNode();
                chead[j] = null;
            }
        }
        public CrossLinkedListSparseMatrix(int row, int col) {
            rn = row;
            cn = col;
            tn = 0;
            rhead = new OLNode[rn];
            for (int i = 0; i < rn; ++i) {
                rhead[i] = new OLNode();
                rhead[i] = null;
            }
            chead = new OLNode[cn];
            for (int j = 0; j < cn; ++j) {
                chead[j] = new OLNode();
                chead[j] = null;
            }
        }
        public CrossLinkedListSparseMatrix(int row, int col, int tn, OLNode[ ] rhead, OLNode[ ]
    chead) {
            rn = row;
            cn = col;
            this.tn = tn;
            this.rhead = new OLNode[rn];
            int i, j;
            for (i = 0; i < rn || i < rhead.length; ++i) {
                this.rhead[i] = rhead[i];
            }
            while (i < rn) {
                rhead[i] = new OLNode();
                rhead[i] = null;
            }
            this.chead = new OLNode[cn];
            for (j = 0; j < cn || j < chead.length; ++j) {
                this.chead[j] = chead[j];
            }
            while (j < cn) {
                chead[j] = new OLNode();
                chead[j] = null;
            }
        }
    }
```

当稀疏矩阵采用十字链表存储结构时,矩阵元素失去了可随机存取的优点,十字链表存储结构下的数据元素存取方法如下。

```
    public double get(int i, int j) {
        // 判断行标和列标是否合法,若不合法则抛出异常
        if (i <= 0 || i > rn || j <= 0 || j > cn) {
            throw new RuntimeException("行标和列标非法");
        }
        // i 的合法值是大于 0 且小于 rn 的整数
        // j 的合法值是大于 0 且小于 cn 的整数
        OLNode temp = rhead[i - 1];
```

```
    if (temp == null) {
        return 0;                    // 此行链表为空,表示该行没有非零元素
    }

    while (temp.col < j && temp.right != null) {
        temp = temp.right;           //循环执行完毕后,temp.col 值大于或等于 j
    }
    if (temp.col == j)
        return temp.value;
    else
        return 0;
}
public void set(int i, int j, double s) {
    // 判断行标和列标是否合法,若不合法则抛出异常
    if (i <= 0 || i > rn || j <= 0 || j > cn) {
        throw new RuntimeException("行标和列标非法");
    }
    // 在第 i 行链表查找列下标为 j 的元素结点
    OLNode cq;
    OLNode rowprior, colprior;
    for (cq = rhead[i - 1]; cq != null; cq = cq.right) {
        if (cq.col == j)
            break;
        // 稀疏矩阵第 i 行第 j 列的元素非零,跳出循环
    }
    // 分 4 种情况
    // (1)s != 0 &&cq!= null,表示矩阵中第 i 行第 j 列的元素非零,新设置数值 s 非零,此时只要令
    // cq.value = s 即可
    // (2)s != 0 &&cq == null,表示矩阵中第 i 行第 j 列的元素是零,新设置数值 s 非零,此时插入
    // (i, j, s)到行列链表中
    // (3)s == 0 &&cq!= null,表示矩阵中第 i 行第 j 列的元素非零,新设置数值 s 是零,此时需要
    // 从行列链表中删除结点(i, j, value)
    // (4)s == 0 &&cq == null,表示矩阵中第 i 行第 j 列的元素是零,新设置数值 s 也是零,此时不
    // 用做任何处理
    if (s != 0 && cq != null)
        cq.value = s;
    if (s != 0 && cq == null) {
        OLNode p = new OLNode();  // 建立新结点
        p.row = i;
        p.col = j;
        p.value = s;              // 给新结点赋值

        if ((rhead[i - 1] == null) || (rhead[i - 1].col > j))
        // 行中没有结点或者该结点应插入行链表中的第一个位置
        {
            p.right = rhead[i - 1];
            rhead[i - 1] = p;
        } else {
            // 寻找在行表中的插入位置
            for (rowprior = rhead[i - 1]; rowprior.right != null
                    && rowprior.right.col < j; rowprior = rowprior.right)
                ;
            // 此循环执行完毕,rowprior 指向新结点在行链表中的前驱结点
            p.right = rowprior.right;
            rowprior.right = p;
        }
        // 在列链表中插入新结点
```

```
            if ((chead[j - 1] == null) || (chead[j - 1].col > i))
            // 列中没有结点或者该结点应插入列链表中的第一个位置
            {
                p.right = chead[j - 1];
                chead[j - 1] = p;
            } else {
                // 寻找在列表中的插入位置
                for (colprior = chead[j - 1]; colprior.down != null && colprior.down.row < i;
    colprior = colprior.down)
                    ;
                // 此循环执行完毕,colprior 指向新结点在列链表中的前驱结点
                p.down = colprior.down;
                colprior.down = p;
            }

            tn++;
        }
        if (s == 0 && cq != null) {
            // 从行列链表中删除当前 cq 结点
            if (rhead[i - 1] == cq)
                rhead[i - 1] = cq.right;
            else {
                // 如果不是行链表的第一个结点,则找到行链表中的前驱结点,令前驱结点的 right
                // 指向 cq.right
                for (rowprior = rhead[i - 1]; rowprior.right != null && rowprior.right.col < j;
    rowprior = rowprior.right)
                    ;
                // 此循环执行完毕,rowprior 指向 cq 的行链表中的前驱结点
                rowprior.right = cq.right;
            }
            if (chead[j - 1] == cq)
                chead[j - 1] = cq.down;
            else {
                // 如果不是列链表的第一个结点,则找到列表中的前驱结点,令前驱结点的 down 指向
                // cq.down
                for (colprior = chead[j - 1]; colprior.down != null && colprior.down.row < i;
    colprior = colprior.down)
                    ;
                // 此循环执行完毕,rowprior 指向 cq 的行链表中的前驱结点
                colprior.down = cq.down;
            }
            tn--;
        }
    }
```

行、列链表中均不存在头结点,处理行或列中的第一个数据元素时存在特殊性。由于 set 方法的实现代码比较烦琐,读者可尝试更改存储结构,在行、列链表中加上头结点后,比较一下不同存储结构下 set 方法的实现。

对于稀疏矩阵,建立并显示十字链表存储结构的实现算法如下。

```
public static void createspmatrixol() {
    Scanner reader = new Scanner(System.in);
    // 实例化 Scanner 类对象 reader
    System.out.println("请输入稀疏矩阵的行数、列数");
    int m = reader.nextInt();           // 矩阵的行数
    int n = reader.nextInt();           // 矩阵的列数
```

```
    CrossLinkedListSparseMatrix Matrix = new CrossLinkedListSparseMatrix(m, n);
    // 各行、列链表均为空链表
    // 按任意次序输入非零元素
    int k;
    while (true)                          // k = 0 表示输入结束
    {
        System.out.println("请输入数据元素序号,0 表示结束输入");
        k = reader.nextInt();
        if (k == 0)
            break;
        System.out.println("请输入数据元素的行标、列标、值");
        int i = reader.nextInt();         // 非零元素行标
        int j = reader.nextInt();         // 非零元素列标
        int e = reader.nextInt();         // 非零元素
        OLNode p = new OLNode();          // 建立新结点
        OLNode q = new OLNode();          // 建立新结点 q,q 是一个替代变量
        p.row = i;
        p.col = j;
        p.value = e;                      // 给结点赋值
        // 将新结点 p 插入行、列链表中
        if ((Matrix.rhead[i - 1] == null) || (Matrix.rhead[i - 1].col > j))
        // 行中没有结点或者该结点应插入行链表中的第一个位置
        {
            p.right = Matrix.rhead[i - 1];
            Matrix.rhead[i - 1] = p;
        } else {                          // 寻找在行表中的插入位置
            for (q = Matrix.rhead[i - 1]; q.right != null && q.right.col < j; q = q.right);

                p.right = q.right;
                q.right = p;
             // 完成行插入
        }
        if (Matrix.chead[j - 1] == null || Matrix.rhead[j - 1].row > i)
        // 列中没有结点或者该结点应插入列链表中的第一个位置
        {
            p.down = Matrix.chead[j - 1];
            Matrix.chead[j - 1] = p;
        } else {                          // 寻找在列表中的插入位置
            for (q = Matrix.chead[j - 1]; q.down != null && q.down.row < i; q = q.down);

                p.down = q.down;
                q.down = p;
             // 完成列插入
        }
        Matrix.tn++;
    }
    Matrix.print();
}
public void print() {                     // 显示十字链表存储结构
    OLNode p = new OLNode();
    for (int i = 0; i < rn; ++i) {
        p = rhead[i];
        System.out.println((i + 1) + "行元素");
        while (p != null) {
            System.out.println(p);
            System.out.println(p.row);
            System.out.println(p.col);
```

```
                System.out.println(p.value);
                System.out.println(p.down);
                System.out.println(p.right);
                System.out.println();
                p = p.right;
            }
        }
    }
```

对于 m 行 n 列且有 t 个非零元素的稀疏矩阵,上述建立算法 createspmatrixol 的执行时间为 $O(t\times s)$,其中 $s=\max\{m,n\}$。这是因为每建立一个非零元素的结点,都要寻找它在行表和列表中的插入位置,此算法对非零元素输入的先后次序没有任何要求。若按以行序为主序的次序依次输入三元组,则可将建立十字链表的算法改写成 $O(t)$ 数量级(t 为非零元素的个数)。

下面讨论在使用十字链表表示稀疏矩阵时,如何实现两个矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的相加运算。

两个矩阵相加与第 2 章中讨论的两个一元多项式相加极为相似,所不同的是一元多项式中只有一个变元(即指数项),而矩阵相加有两个变元(行值和列值)。矩阵中的每个结点既在行表又在列表中,使得插入和删除时引用的修改稍微复杂,因此需要更多的辅助引用。

利用原矩阵 $\boldsymbol{A}$ 的十字链表,转换为和矩阵的十字链表。设两个矩阵相加后的结果为 $\boldsymbol{A}'$,则和矩阵 $\boldsymbol{A}'$中的非零元 $a'_{\text{row,col}}$只可能有 3 种情况,即 $a_{\text{row,col}}+b_{\text{row,col}}$,$a_{\text{row,col}}$($b_{\text{row,col}}=0$ 时),$b_{\text{row,col}}$($a_{\text{row,col}}=0$ 时)。因此,当将 $\boldsymbol{B}$ 加到 $\boldsymbol{A}$ 上时,对矩阵 $\boldsymbol{A}$ 的十字链表来说,可能是改变结点的 value 域值($a_{\text{row,col}}+b_{\text{row,col}}!=0$)、不变($b_{\text{row,col}}=0$)或插入一个新结点($a_{\text{row,col}}=0$)。此外,还有一种可能的情况是:与矩阵 $\boldsymbol{A}$ 中的某个非零元素相对应,和矩阵 $\boldsymbol{A}'$中是零元素($a_{\text{row,col}}+b_{\text{row,col}}=0$),即对 $\boldsymbol{A}$ 的操作是删除一个结点。由此可知,整个运算过程从矩阵的第一行起逐行进行,从每一行的表头出发分别找到 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 在该行中的第一个非零元素结点并开始比较,然后按上述 4 种不同情况分别处理。

假设非空引用 pa 和 pb 分别指向 $\boldsymbol{A}$、$\boldsymbol{B}$ 中行值相同的两个结点,pa==null 表明矩阵 $\boldsymbol{A}$ 在该行中没有非零元素,则上述 4 种情况的具体处理过程如下。

(1) 若 pa==null 或 pa.col>pb.col,则需要在矩阵 $\boldsymbol{A}$ 的链表中插入一个值为 $b_{\text{row,col}}$ 的结点。此时需要改变同一行中前一结点的 right 域值,以及同一列中前一结点的 down 域值。

(2) 若 pa.col<pb.col,则只需将 pa 引用往右推进一步。

(3) 若 pa.col==pb.col 且 pa.value+pb.value!=0,则只需将 $a_{\text{row,col}}+b_{\text{row,col}}$ 的值送到 pa 所指结点的 value 域即可,其他所有域的值都不变。

(4) 若 pa.col==pb.col 且 pa.value+pb.value==0,则需要在矩阵 $\boldsymbol{A}$ 的链表中删除 pa 所指的结点。此时需要改变同一行中前一结点的 right 域值,以及同一列中前一结点的 down 域值。

为了便于插入和删除结点,还需要设立一些辅助引用。①在 $\boldsymbol{A}$ 的行链表上设 pre 引用,指示 pa 所指结点的前驱结点;②在 $\boldsymbol{A}$ 的每一列的链表上设一个引用 hl[col-1],它的初值与列链表的头引用相同,即 hl[col-1]=chead[col-1](数组下标从 0 开始计数)。

下面简要描述将矩阵 $\boldsymbol{B}$ 加到矩阵 $\boldsymbol{A}$ 上的操作过程。

(1) 令 pa 和 pb 初始化,分别指向 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的第一行的第一个非零元素的结点。

```
pa = A.rhead[0];pb = B.rhead[0];pre = null;
```

且令 hl 初始化：

```
for(col = 1;col <= A.cn;col++)hl[col - 1] = A.chead[col - 1];
```

(2) 重复本步骤，依次处理本行结点，直到 B 中无非零元素的结点，即 pb==null 为止。

① 若 pa==null 或 pa.col>pb.col(即 ***A*** 的这一行的非零元素已处理完)，则需在 ***A*** 中插入一个 pb 所指结点的复制结点。假设新结点的地址为 p，则 ***A*** 的行表中的引用变化如下。

```
if (pre == null)
   A.rhead[p.row - 1] = p;
else
   pre.right = p;
p.right = pa;pre = p;
```

A 在列表中的引用也要进行相应的改变。首先需要从 hl[p.col−1]开始找到新结点在同一列中的前驱结点，并让 hl[p.col−1]指向它，然后在列链表中插入新结点。

```
if(!A.chead[p.col - 1]||A.chead[p.col - 1].row > p.row){
   p.down = A.chead[p.col - 1];A.chead[p.col - 1] = p;
}
else{
   p.down = hl[p.col - 1].down;hl[p.col - 1].down = p;
}
hl[p.col - 1] = p;
```

② 若 pa!=null 且 pa.col<pb.col，则令 pa 指向本行下一个非零元素结点。

```
pre = pa;pa = pa.right;
```

③ 若 pa.col==pb.col，则将 ***B*** 中当前结点的值加到 ***A*** 中当前结点上。

```
pa.value = pa.value + pb.value;
```

此时，若 pa.value!=0，则引用不变；否则，删除 ***A*** 中该结点。

```
if (pre == null)
   A.rhead[pa.row - 1] = pa.right;
else
   pre.right = pa.right;
p = pa;pa = pa.right;
```

同时，为了改变列表中的引用，需要先找到同一列中的前驱结点，且让 hl[pa.col]指向该结点，然后修改相应引用如下。

```
if (A.chead[p.col - 1] == p)
   A.chead[p.col - 1] = hl[p.col - 1] = p.down;
else
   hl[p.col - 1].down = p.down;
```

(3) 若本行不是最后一行，则令 pa 和 pb 指向下一行的第一个非零元素结点，然后转到步骤(2)；否则，算法结束。

通过对该算法的分析可以得出结论：从一个结点来看，进行比较、修改引用所需的时间是一个常数；整个运算过程的关键在于对 ***A*** 和 ***B*** 的十字链表逐行扫描，其循环次数主要取决于矩阵 ***A*** 和 ***B*** 中非零元素的个数 ta 和 tb；所以该算法的时间复杂度为 $O(ta+tb)$。

对于十字链表存储结构下的矩阵转置、矩阵求逆、矩阵加减等其他运算，读者可作为练习自行实现。


视频讲解

5.3 广义表

5.3.1 广义表的定义

广义表是线性表的推广和扩充,在人工智能领域中的应用十分广泛。

线性表(Linear List)是由 $n(n\geqslant 0)$个类型相同的数据元素组成的有限序列,通常记作 $L=(a_1,a_2,\cdots,a_{i-1},a_i,a_{i+1},\cdots,a_n)$。而广义表(Generalized List)是由 $n(n\geqslant 0)$个数据元素组成的有限序列,通常记作 $LS=(\alpha_1,\alpha_2,\cdots,\alpha_i,\cdots,\alpha_n)$。

关于广义表的几点说明:

(1) α_i 与 α_j 类型可以不同。α_i 可以是原子元素,也可以是子表元素。例如,$C=(a,(b,c,d))$是一个广义表,第一个数据元素是一个原子元素 a,第二个数据元素是子表 (b,c,d)。

(2) 广义表的长度。广义表的长度是指广义表中的数据元素个数。例如,广义表 C 的长度是 2,广义表 $A=()$的长度是 0。

(3) 广义表的深度。广义表的深度是指广义表展开式所含括号的重数。例如,广义表 C 的深度是 2。又如,广义表 $B=(A,C)$的深度需要依据广义表 B 的展开式$((),(a,(b,c,d)))$进行判断,展开式括号的重数是三重,则广义表 B 的深度是 3。

(4) 广义表的表头(head)和表尾(tail)。广义表 $LS=(\alpha_1,\alpha_2,\cdots,\alpha_i,\cdots,\alpha_n)$的表头是 α_1,表尾是$(\alpha_2,\cdots,\alpha_i,\cdots,\alpha_n)$。广义表 $C=(a,(b,c,d))$的表头是 a,表尾是$((b,c,d))$。广义表 $D=(e)$的表头为 e,表尾为空表()。

5.3.2 广义表的抽象数据类型

```
ADT GLlist
{
数据对象: D = {e_i | i = 1,2,…,n,n≥0; e_i ∈ AtomSet 或 e_i ∈ GList,AtomSet 为某个数据对象}
数据关系: R = {< e_{i-1},e_i >| e_{i-1},e_i ∈ D,2≤i≤n}
基本操作:
(1) CreateGList(String s)。
初始条件: s 是广义表的书写形式串。
操作结果: 由 s 创建广义表 L。
(2) GListLength()。
操作结果: 求广义表 L 的长度,即元素个数。
(3) GListDepth()。
操作结果: 求广义表 L 的深度。
(4) GListEmpty()。
操作结果: 判定广义表 L 是否为空。
(5) GetHead()。
操作结果: 取广义表 L 的表头。
(6) GetTail()。
操作结果: 取广义表 L 的表尾。
```

(7) insertFirstGL(e)。
操作结果：插入元素 e 作为广义表 L 的第一元素。
(8) DeleteFirstGL)。
初始条件：广义表不为空。
操作结果：删除广义表 L 的第一元素，并用 e 返回其值。
(9) Traverse()。
操作结果：遍历广义表 L。
}

5.3.3　广义表的存储结构

由于广义表数据元素可以具有不同结构，因此难以用顺序方式存储。一般用链接方式存储广义表，称为广义链表。广义表 LS=(α_1,α_2,…,α_i,…,α_n)的元素 α_i 可以是原子，也可以是子表。因此，广义链表中有两种不同类型的结点：原子结点和子表结点。

1. 第一种存储结构

原子数据元素结点由两部分组成：标志域 0 和原子元素的值。子表数据元素结点由 3 部分组成：标志域 1、子表表头引用 hp 和子表表尾 tp 引用。原子结点和子表结点如图 5-15 所示。

广义表 $L=(a,(b))$的第一种链式存储结构如图 5-16 所示。

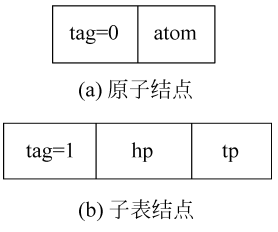


图 5-15　第一种存储结构的原子结点和子表结点

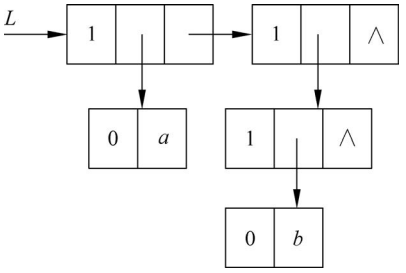


图 5-16　广义表的第一种链式存储结构

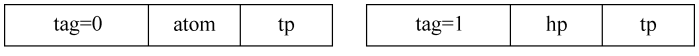
由于 Java 不支持共用体，因此原子结点和子表结点用一个类来表示，该类共有 4 个成员变量，通过 tag 值是 0 或 1 来区分是原子结点还是子表结点。原子结点的 tag 值为 0，并且原子结点的 hp 引用域和 tp 引用域一定为空；子表结点的 tag 值为 1，并且子表结点的 atom 域一定为空。

该链式存储结构结点的 Java 描述如下。

```
class GLnode {
    public int tag;
    public String atom;
    public GLnode hp;
    public GLnode tp;
}
```

2. 第二种存储结构

广义表也可采用另一种形式的存储结构，原子结点和子表结点如图 5-17 所示。



(a) 原子结点　　(b) 子表结点

图 5-17　第二种存储结构的原子结点和子表结点

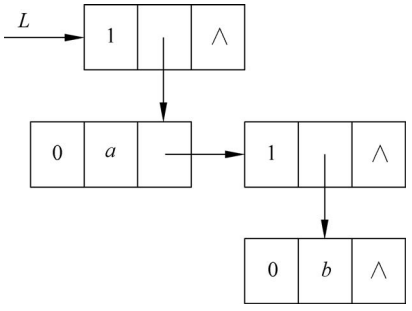


图 5-18　广义表的第二种链式存储结构

原子数据元素结点由 3 部分组成：标志域 0 和原子元素的值 atom 和下一个元素引用 tp。子表数据元素结点也由 3 部分组成：标志域 1、子表表头引用 hp 和下一个元素引用 tp。

广义表 $L=(a,(b))$的第二种链式存储结构如图 5-18 所示。

由于 Java 不支持共用体，因此原子结点和子表结点用一个类来表示，该类共有 4 个成员变量，通过 tag 值是 0 或 1 来区分是原子结点还是子表结点。原子结点的 tag 值为 0，并且原子结点的 hp 引用域一定为空；子表结点的 tag 值为 1，并且子表结点的 atom 域一定为空。

该链式存储结构结点的 Java 描述如下。

```
class GLnode {
    public int tag;
    public String atom;
    public GLnode hp;
    public GLnode tp;
}
```

5.3.4　求广义表深度基本操作的实现

首先约定所讨论的广义表都是非递归表且无共享子表。广义表的深度定义为广义表中括弧的重数，是广义表的一种量度。例如，多元多项式广义表的深度为多项式中变元的个数。

设非空广义表为 $LS=(\alpha_1,\alpha_2,\cdots,\alpha_i,\cdots,\alpha_n)$，其中 $\alpha_i(i=1,2,\cdots,n)$为原子或 LS 的子表，则求 LS 的深度可分解为 n 个子问题，每个子问题为求 α_i 的深度。若 α_i 是原子，则由定义可知其深度为零；若 α_i 是广义表，则同样进行分解处理，而 LS 的深度为 $\alpha_i(i=1,2,\cdots,n)$的深度的最大值加 1。空表也是广义表，并由定义可知空表的深度为 1。

由此可见，求广义表的深度的递归算法有两个终结状态：空表和原子。只要求得 α_i $(i=1,2,\cdots,n)$的深度，广义表的深度就容易求得了，显然应比子表深度的最大值多 1。

广义表 $LS=(\alpha_1,\alpha_2,\cdots,\alpha_n)$的深度 DEPTH(LS)的递归定义如下。

(1) 基本项：DEPTH(LS)=1，当 LS 为空表时；

　　DEPTH(LS)=0，当 LS 为原子时。

(2) 归纳项：DEPTH(LS)=1+MAX{DEPTH(α_i)}。

由此定义容易写出求深度的递归函数。假设 L 是 GLnode 型的变量(采用 5.3.3 节中的第一种存储结构)，则 L=null 表明广义表为空表，L.tag=0 表明是原子。在其他情况下，L 指向表结点，该结点中的 hp 引用指向表头，即为 L 的第一个子表，而结点中的 tp 所

指表尾结点中的 hp 指向 L 的第二个子表，在第一层中由 tp 相连的所有尾结点中的 hp 均指向 L 的子表。由此，求广义表深度的递归方法实现如下。

```
public int Depth(GLnode L) {            // 采用头尾链表存储结构,求广义表 L 的深度
    GLnode pp = new GLnode();
    if (L == null)
        return 1;                       // 空表深度为 1
    if (L.tag == 0)
        return 0;                       // 原子深度为 0
    else {
        int max = 0;
        for (pp = L; pp != null; pp = pp.tp) {
            int dep = Depth(pp.hp);  // 求以 pp.hp 为头引用的子表深度
            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
    }
}// GListDepth
```

上述算法的执行过程实质上是遍历广义表的过程，在遍历中首先求得各子表的深度，然后综合得到广义表的深度。例如，图 5-19 展示了求广义表 D 的深度的过程，其中用虚线示意遍历过程中引用 L 的变化状况，在指向结点的虚线旁标记的是将要遍历的子表，而在从结点射出的虚线旁标记的数字是刚求得的子表的深度。由图 5-20 可见，广义表 $D=(A,B,C)=((),(e),(a,(b,c,d)))$的深度为 3。若按递归定义分析广义表 D 的深度，则有：

$\mathrm{DEPTH}(D)=1+\mathrm{MAX}(\mathrm{DEPTH}(A),\mathrm{DEPTH}(B),\mathrm{DEPTH}(C))$

$\mathrm{DEPTH}(B)=1+\mathrm{MAX}(\mathrm{DEPTH}(e))=1+0=1$

$\mathrm{DFPTH}(C)=1+\mathrm{MAX}(\mathrm{DEPTH}(a),\mathrm{DEPTH}((b,c,d)))=2$

$\mathrm{DEPTH}(a)=0$

$\mathrm{DEPTH}((b,c,d))=1+\mathrm{MAX}(\mathrm{DEPTH}(b),\mathrm{DEPTH}(c),\mathrm{DEPTH}(d))=1+0=1$

由此，$\mathrm{DEPTH}(D)=1+\mathrm{MAX}(1,1,2)=3$。

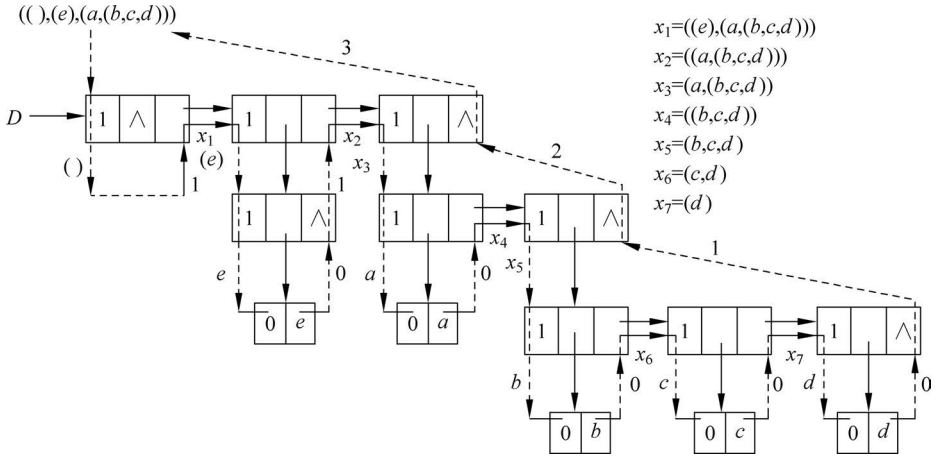


图 5-19　求广义表 *D* 的深度的过程

利用面向对象思想思考问题时,GLnode 型的引用可以确定整个广义表。因此,这里定义广义表类 GL,将 GLnode 型的引用作为类 GL 的成员变量,定义 GListDepth()作为类 GL 的方法,返回值是广义表的深度。在广义表的链式存储结构下实现求深度操作的代码如下。

```
class GL {
    GLnode L;                                   //成员变量
    public int GListDepth() {                   // 采用头尾链表存储结构,求广义表 L 的深度
        return Depth(L);
    }// GListDepth
    public int Depth(GLnode L) {                // 采用头尾链表存储结构,求广义表 L 的深度
        GLnode pp = new GLnode();
        if (L == null)
            return 1;                           // 空表深度为 1
        if (L.tag == 0)
            return 0;                           // 原子深度为 0
        else {
            int max = 0;
            for (pp = L; pp != null; pp = pp.tp) {
                int dep = Depth(pp.hp);         // 求以 pp.hp 为头引用的子表深度
                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
        }
    }//Depth
}
```

在对广义表的操作进行递归定义时,可以有两种分析方法。一种是将广义表分解成表头和表尾两部分;另一种是将广义表看成是含有 n 个并列子表(假设原子也视作子表)的表。在讨论建立广义表的存储结构时,这两种分析方法均可使用。

假设将广义表的书写形式看成是一个字符串 s,则当 s 为非空白串时广义表非空。此时可以利用 5.3.2 节中定义的取列表表头 GetHead 和取列表表尾 GetTail 两个方法建立广义表的链表存储结构,读者可自行实现该算法。

下面就第二种分析方法进行讨论。

广义表字符串 s 可能有两种情况:①s="()"(带括号的空白串);②$s=(\alpha_1,\alpha_2,\cdots,\alpha_n)$,其中 $\alpha_i(i=1,2,\cdots,n)$是 s 的子串。对应于第一种情况 s 的广义表为空表;对应于第二种情况 s 的广义表中含有 n 个子表,每个子表的书写形式即为子串 $\alpha_i(i=1,2,\cdots,n)$。此时可类似于求广义表的深度,分析由 s 建立的广义表和由 $\alpha_i(i=1,2,\cdots,n)$建立的子表之间的关系。假设按图 5-17 所示的结点结构建立广义表的存储结构,则含有 n 个子表的广义表中有 n 个表结点序列。第 i 个表结点中的表尾引用指向第 $i+1$ 个表结点,第 n 个表结点的表尾引用为 null。如果将原子也看成是子表,则第 i 个表结点的表头引用 hp 指向由 α_i 建立的子表。因此,由 s 建立广义表的问题可转化为由 $\alpha_i(i=1,2,\cdots,n)$建立子表的问题。此时,α_i 可能有以下 3 种情况。

(1) 带括号的空白串;

(2) 长度为 1 的单字符串;

(3) 长度大于 1 的字符串。

显然，前两种情况为递归的终结状态，子表为空表或只含一个原子结点，后一种情况为递归调用。因此，在不考虑输入字符串可能出错的前提下，可以得到下列建立广义表链表存储结构的递归定义。

(1) 基本项：置空广义表，当 s 为空表串时；

建立原子结点的子表，当 s 为单字符串时。

(2) 归纳项：假设 sub 为删去 s 中最外层括号后的字符串，记为“$s_1,s_2,\cdots,s_n$”，其中 $s_i(i=1,2,\cdots,n)$为非空字符串。对每一个 s_i 建立一个表结点，并令其 hp 域的引用为由 s_i 建立的子表的头引用，除最后建立的表结点的尾引用为 null 外，其余表结点的尾引用均指向其后建立的表结点。

假定方法 string[] sever(str)的功能为：从字符串 str 中取出第一个“,”之前的字串并赋给 hsub，使 str 成为删去子串 hstr 和“,”之后的剩余串；若串 str 中没有字符“,”，则操作后的 hsub 即为操作前的 str，而操作后的 str 为空串""；返回 hsub 和 str(将二者存放在一个长度为 2 的字符串数组中，并返回该字符串数组)。根据上述递归定义可得创建广义表存储结构的递归算法如下。

```
public GLnode createGList(String s) {
    // 采用头尾链表存储结构,由广义表的书写形式串 s 创建广义表 L,设 emp = "()"
    String emp = "()";                          // 建立字符串类 emp
    String hsub = "";
    String[] str = new String[2];
    GLnode L1 = new GLnode();                   // 创建表结点
    if (s == emp)
        L1 = null;                              // 创建空表
    else {
        if (s.length() == 1) {
            L1.tag = 0;
            L1.atom = s;
        } // 创建单原子广义表
        else {
            L1.tag = 1;
            GLnode p = L1;
            GLnode q;
            String sub = s.substring(1, (s.length() - 1));       // 删外层括号
            do {                                // 重复创建 n 个子表
                str = sever(sub);               // 从 sub 中分离出表头串 hsub 和表尾串
                hsub = str[0];                  // hsub 初始值为空
                sub = str[1];                   // sub 初始值为空
                p.hp = createGList(hsub);
                q = p;
                if (!sub.equals("")) {          // 表尾不空
                    // 新建结点
                    p = new GLnode();
                    p.tag = 1;
                    q.tp = p;

                } // if
            } while (!sub.equals(""));
```

```
                q.tp = null;
            } // else
        } // else
        return L1;
    } // CreateGList
    public String[] sever(String str) {
        // 将非空串 str 分割成两部分:hsub 为第一个外层','之前的子串(表头),str 为之后的子串
        // 在 str = "((5,6),(7,8,9))"调用方法中先删外层括号 str = "(5,6),(7,8,9)",则 hsub = "(5,6)",
        // str = "(7,8,9)"
        // 在 str = "((5,6))"调用方法中先删外层括号 str = "(5,6)",则 hsub = "(5,6)",str = ""
        // 在 str = "(5)"调用方法中先删外层括号 str = "5",则 hsub = "5",str = ""
        String hsub = "";                          // hsub 初始值为空
        int n = str.length();
        int i = -1;
        int k = 0;                                 // k 为尚未配对的左括号个数
        String ch;
        do {                                       // 搜索最外层的第一个逗号
            i++;
            ch = str.substring(i, i + 1);
            if (ch.equals("(")) {
                k++;
            } else if (ch.equals(")")) {
                k--;
            }
        } while ((i < (n - 1)) && (!ch.equals(",") || k != 0));
        if (i < n - 1) {
            hsub = str.substring(0, i);
            str = str.substring(i + 1, n);
        } else {
            hsub = str;
            str = "";
        }
        String[] str1 = new String[2];
        str1[0] = hsub;
        str1[1] = str;
        return str1;
    }// sever
```

调用方法代码如下。

```
public static void main(String[] args) {
    GL List1 = new GL();
    List1.L = List1.createGList("(((((6)),7,8),9),8)");
    System.out.println(List1.GListDepth());
}
```

5.3.5 m 元多项式的表示

在一般情况下使用的广义表大多既不是递归表,也不为其他表所共享。对广义表可以这样理解,广义表中的一个数据元素可以是另一个广义表,m 元多项式的表示就是这种应用的典型实例。在第 2 章中,作为线性表的应用实例讨论了一元多项式,一个 n 项

一元多项式可以用一个长度为 n 且每个数据元素有两个数据项(系数项和指数项)的线性表进行表示。

下面将讨论如何表示 m 元多项式。

如果用线性表进行表示,则每个数据元素需要 $m+1$ 个数据项,以此存储一个系数值和 m 个指数值。这将产生两个问题:一是无论多项式中各项的变元数是多少,若都按 m 个变元分配存储空间,则将造成浪费;反之,若按各项实际的变元数分配存储空间,则会造成结点的大小不匀,给操作带来不便。二是对 m 值不同的多项式,线性表中的结点大小也不同,这同样会引起存储管理的不便。

由于 m 元多项式中每一项的变元数目的不均匀性和变元信息的重要性,因此不适于用线性表表示。例如,三元多项式

$$P(x,y,z)=x^{10}y^3z^2+2x^6y^3z^2+3x^5y^2z^2+x^4y^4z+6x^3y^4z+2yz+15$$

其中,各项的变元数目不尽相同,y^3,z^2 等因子多次出现。将该多项式改写为

$$P(x,y,z)=((x^{10}+2x^6)y^3+3x^5y^2)z^2+((x^4+6x^3)y^4+2y)z+15$$

此时,多项式 P 是变元 z 的多项式,即 $Az^2+Bz+15z^0$,其中 A 和 B 本身又是一个 (x,y) 的二元多项式,15 是 z 的零次项的系数。进一步考察 $A(x,y)$,也可将其看成是 y 的多项式。例如,Cy^3+Dy^2,其中 C 和 D 为 x 的一元多项式。

任何一个 m 元多项式都可以先分解出一个主变元,然后再分解出第二个变元等。因此,一个 m 元的多项式首先是它的主变元的多项式,而其系数 R 是第二变元的多项式,由此可用广义表表示 m 元多项式。例如,上述三元多项式可用下面的广义表表示,广义表的深度即为变元个数。

$$P=z((A,2),(B,1),(15,0))$$

其中,$A=y((C,3),(D,2))$。

$C=x((1,10),(2,6))$

$D=x((3,5))$

$B=y((E,4),(F,1))$

$E=x((1,4),(6,3))$

$F=x((2,0))$

类似于广义表的第二种存储结构,定义表示 m 元多项式的广义表的存储结构,链表的结点结构如图 5-20 所示。

(a) 子表结点　　(b) 原子结点

图 5-20　m 元多项式的链表结点结构

$P=z((A,2),(B,1),(15,0))$的广义表的存储结构如图 5-21 所示,在每一层上增设一个表头结点并利用 exp 指示该层的变元,可用一维数组存储多项式中的所有变元,故 exp 域存储的是该变元在一维数组中的下标。头引用 p 所指表结点中 exp 的值 3 为多项式中变元的个数。由此可见,这种存储结构可以表示任何元的多项式。

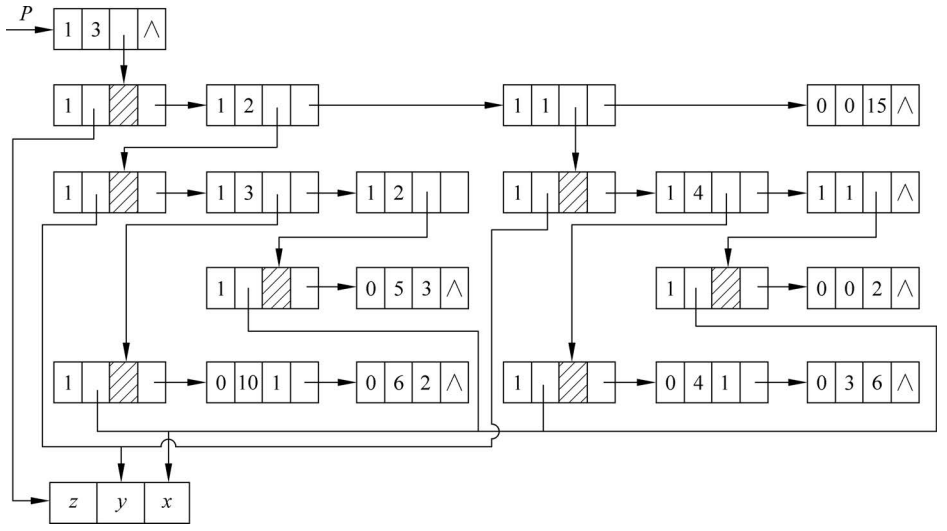


图 5-21 三元多项式 $P(x,y,z)$的存储结构

本章小结

本章内容分为 3 部分：数组、矩阵和广义表。第一部分介绍了一维数组和二维数组的存储。第二部分介绍了几种特殊矩阵：对称矩阵、三角矩阵、对角矩阵和稀疏矩阵，对这些特殊矩阵仍采用二维数组存储会造成浪费，因此研究了这些特殊矩阵的压缩存储，并探讨了这些特殊矩阵在不同的存储结构下各种矩阵运算的实现。第三部分介绍了广义表，首先介绍了广义表的定义、广义表抽象数据类型，然后介绍了广义表的两种链式存储结构及 Java 语言描述，接着基于第一种链式存储结构——头尾链表实现了抽象数据类型广义表中的求长度操作，最后介绍了广义表的应用——利用广义表表示 m 元多项式。

在线测试

习题 5

一、选择题

1. 设广义表 $L=((a,b,c))$，则 L 的长度和深度分别为(　　)。

A. 1 和 1　　B. 1 和 3　　C. 1 和 2　　D. 2 和 3

2. 广义表$((a),a)$的表尾是(　　)。

A. a　　B. (a)　　C. ()　　D. $((a))$

3. 一个非空广义表的表头(　　)。

A. 不可能是子表　　B. 只能是子表

C. 只能是原子　　D. 可以是子表或原子

4. 数组 $A[0..5,0..6]$的每个元素占 5 字节，将其按列优先次序存储在起始地址为 1000 的内存单元中，则元素 $A[5][5]$的地址是(　　)。

A. 1175　　B. 1180　　C. 1205　　D. 1210

5. 广义表 $G=(a,b(c,d,(e,f)),g)$的长度是(　　)。

A. 3　　B. 4　　C. 7　　D. 8

6. 对一些特殊矩阵采用压缩存储的目的主要是为了(　　)。

A. 使表达变得简单　　B. 使矩阵元素的存取变得简单

C. 去掉矩阵中的多余元素　　D. 减少不必要的存储空间的开销

7. 设矩阵 $\boldsymbol{A}$ 是一个对称矩阵,为了节省存储空间,将其下三角部分按行序存放在一维数组 $B[1..n(n-1)/2]$中,对下三角部分中的任意元素 $a_{i,j}(i\geqslant j)$,在一维数组 B 的下标位置 k 的值是(　　)。

A. $i(i-1)/2+j-1$　　B. $i(i-1)/2+j$

C. $i(i+1)/2+j-1$　　D. $i(i+1)/2+j$

8. 以下有关广义表的说法,正确的是(　　)。

A. 由 0 个或多个原子或子表构成的有限序列

B. 至少有一个元素是子表

C. 不能递归定义

D. 不能为空表

二、填空题

1. 由于计算机内存中的存储单元是一个一维的存储结构,因此要想按顺序将多维数组存储到计算机存储单元中就必须解决排列顺序问题。对于二维数组,有两种排列形式:一种是_____;另一种是_____。

2. 零元素数目远远多于非零元素数目,并且非零元素的分布没有规律的矩阵称为_____。

3. 在一个 n 阶方阵 $\boldsymbol{A}$ 中,若元素满足性质:$a_{ij}=a_{ji}(0\leqslant i,j\leqslant n-1)$,则称 $\boldsymbol{A}$ 为 n 阶_____。

4. _____运算是矩阵运算中最基本的一项,它是将一个 $m\times n$ 的矩阵变成另外一个 $n\times m$ 的矩阵,同时使原来矩阵中元素的行、列位置互换而值保持不变。

5. 广义表是 $n(n\geqslant 0)$个元素的序列,记作 $A=(a_1,a_2,\cdots,a_n)$。其中,A 是广义表的_____,n 是它的_____,当 $n=0$ 时称为_____。

6. 在一个非空的广义表中,其元素 a_i 可以是某一确定类型的单个元素,称为_____;也可以是一个广义表,称为_____。

7. 广义表的定义是一种递归的定义,广义表是一种递归的数据结构。当广义表非空时,称第一个元素 a_1 为广义表 A 的_____,其余元素组成的表$(a_2,a_3,\cdots,a_n)$是 A 的_____。

8. 已知二维数组 $\boldsymbol{A}[m][n]$采用行序为主方式存储,每个元素占 k 个存储单元,并且第一个元素的存储地址是 LOC($\boldsymbol{A}$[0][0]),则 $\boldsymbol{A}[i][j]$的地址是_____。

9. 稀疏矩阵常用的压缩存储方式有_____和_____。

三、判断题

1. 稀疏矩阵在压缩存储后必然会失去随机存取功能。　(　　)

2. 若一个广义表的表头为空表,则此广义表为空表。　(　　)

3. 广义表是线性表的推广,是一类线性数据结构。　(　　)

4. 假设一个稀疏矩阵 $\boldsymbol{A}_{m\times n}$ 采用三元组形式表示,若将三元组中有关行下标与列下标的值互换,并将 m 和 n 的值互换,则可完成 $\boldsymbol{A}_{m\times n}$ 的转置运算。 ( )

5. 数组中的元素必定具有相同的数据类型。 ( )

6. 如果广义表中的每个元素都是原子,则该广义表即为线性表。 ( )

7. 广义表中的原子个数即为广义表的长度。 ( )

8. 广义表是一种多层次的数据结构,其元素可以是单原子或子表。 ( )

四、综合题

1. 现有一个稀疏矩阵如图 5-22 所示,请给出它的三元组顺序表和十字链表。

$$\begin{bmatrix} 0 & 3 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & -2 & 0 \end{bmatrix}$$

图 5-22 稀疏矩阵示例

五、实验题

1. 请开发一个对角矩阵运算系统,基于对角矩阵的压缩方式实现对角矩阵的相加、相乘、转置等运算。

2. 请开发一个稀疏矩阵运算系统,实现稀疏矩阵的相加、相乘、转置等运算,并尝试在二维数组、三元组顺序表、十字链表等不同的存储结构下实现。