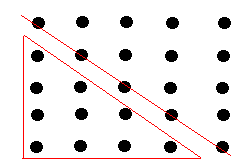
矩阵

特殊矩阵：方阵中的对称矩阵

若N阶方阵按主对角线数值对称，且主对角线上的值均为0：这种矩阵称为对称矩阵。

若对N阶对称矩阵进行全存储，明显没有意义，完全可以只存储左下半三角矩阵（或者右上半三角矩阵），若只存储左下半三角部分的数据，那么，可以提高内存利用率。

N阶对称矩阵按照压缩存储方式，仅存储其左下半三角的数据，则，需要存储的数据量为：

1+2+3+...+N-1 = (1+N-1)\*(N-1)/2 = N\*(N-1)/2

与全存储对比，是其(N\*(N-1)/2)/N\*N = ½

但是，按照这种存储方式存储数据后，失去了二维特性，事实上是以一维数组方式存储的

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

0 1 2 3 4 5 6 7 8 9

需要实现一个基本的地址映射操作：由matrix[i][j] <=> T[index]

需要得到关于index的i,j计算公式

对于m[i][j], 假设N阶，若i,j都是有效下标，需要先将其格式化成一个统一的模式：

m[i][j] => m[row][col] 其中row > col,若i == j 则，返回0

对于i,j已满足i > j, 如何计算index?

第i行前有0+1+2+...+ i-1=> (1+i-1)\*(i-1)/2;

第j列前，有j个元素;

两式求和：i\*(i-1)/2 + j;

稀疏矩阵：

若一个矩阵中，绝大部分数值都为0或者相等，只有极少数数值非零，这种矩阵被称为稀疏矩阵。衡量一个矩阵是否满足系数矩阵的一般性原则是稀疏因子：非零个数/总个数\*100% < 0.15%

从稀疏矩阵的定义来看，若使用全存储方案，明显是及其耗费存储空间的！

为此，合理的存储稀疏矩阵，尽可能节省存储空间是一个努力方向！

稀疏矩阵的压缩存储方案常见的有两种：

1，三元组法；

2，十字交叉链表法；

三元组法：

0 1 2 3 4 5 6

下标 rowcolvalue

0:(0, 1, 4)

1:(0, 5, 1)

2:(1, 4, 9)

3:(2, 1, 2)

4:(2, 3, 8)

5:(3, 1, 6)

6:(3, 4, 3)

7:(3, 6, 7)

8:(5, 4, 5)

0 0 4 0 0 0 1 0

1 0 0 0 0 9 0 0

2 0 2 0 8 0 0 0

3 0 6 0 0 3 0 7

4 0 0 0 0 0 0 0

5 0 0 0 0 5 0 0

对于矩阵中的每一个元素，记录其行，列和值这三个数据，称为三元数据；由于稀疏矩阵中存在多个非零元素，那么，这些个元素的三元数据集合称为三元组。

进一步的要求：

三元组中的数据按行值升序，行值相同的按列值升序排列。

对于上述稀疏矩阵所生成的三元组为：

0: (0, 1, 4)

1: (0, 5, 1)

2: (1, 4, 9)

3: (2, 1, 2)

4: (2, 3, 8)

5: (3, 1, 6)

6: (3, 4, 3)

7: (3, 6, 7)

8: (5, 4, 5)

借鉴堆栈，队列的实现经验，当然应该设计一个控制头，用以表示整个矩阵；

typedef struct TRIAD{

int row;

int col;

USER\_TYPE value;

} TRIAD;

typedef struct MATRIX{

TRIAD \*triad;

int count;

int Row;

int Col;

} MATRIX;

基于三元组的稀疏矩阵的转置运算

即，将上述矩阵转置成

0 1 2 3 4 5

下标 rowcolvalue

0:(1, 0, 4)

1:(1, 2, 2)

2:(1, 3, 6)

3:(3, 2, 8)

4:(4, 1, 9)

5:(4, 3, 3)

6:(4, 5, 5)

7:(5, 0, 1)

8:(6, 3, 7)

0 0 0 0 0 0 0

1 4 0 2 6 0 0

2 0 0 0 0 0 0

3 0 0 6 0 0 0

4 0 9 0 3 0 5

5 1 0 0 0 0 0

6 0 0 0 7 0 0

第一步：申请一个以maxCol+1个元素的数组

arIndex:

0 0 0123 0 01 0123 01 01

0 1 2 3 4 5 6 7

0 0 3 0 1 3 1 1

第二步：遍历原三元组，以列值+1为arIndex的下标，给arIndex的元素增1

第三步：将上述arIndex的值递增，得到：

0 1 2 3 4 5 6 7

0 0 3 3 4 7 8 9

1 4 5 8

2

3 6 9 7

0 3 3 4 7 8 9 9

同时，上述数组值的意义还有：未来转置后的矩阵，下标为index的第一个元素在目标三元组中的下标就是arIndex[index]

能直接定位转置后的三元元素在目标三元组中的下标位置！

下标 rowcolvalue

0:(0, 1, 4)

1:(0, 5, 1)

2:(1, 4, 9)

3:(2, 1, 2)

4:(2, 3, 8)

5:(3, 1, 6)

6:(3, 4, 3)

7:(3, 6, 7)

8:(5, 4, 5)

下标 rowcolvalue

0:(1, 0, 4)

1:(1, 2, 2)

2:(1, 3, 6)

3:(3, 2, 8)

4:(4, 1, 9)

5:(4, 3, 3)

6:(4, 5, 5)

7:(5, 0, 1)

8:(6, 3, 7)

转置

为实现这个目的，分成三步实施：

1、统计转置后每一行的数据个数；

2、计算转置后每一行的第一个数据前存在的数据个数，即，这一行的第一个数据应该存在于目标三元组中的起始下标。

具体操作过程：

1、先申请一个元素个数为矩阵列阶+1的全0数组；

|  |
| --- |
| 0 0 0 0 0 0 0 0  0 1 2 3 4 5 6 7 |

2、遍历原三元组，以列值+1为index下标，并对该元素+1

|  |
| --- |
| 0 1 2 3 4 5 6 7  下标 rowcolvalue  0:(0, 1, 4)  1:(0, 5, 1)  2:(1, 4, 9)  3:(2, 1, 2)  4:(2, 3, 8)  5:(3, 1, 6)  6:(3, 4, 3)  7:(3, 6, 7)  8:(5, 4, 5)  0 0 0123 0 01 0123 01 01  0 0 3 0 1 3 1 1 |

3、对index数组中的值累计和值，得到的就是最重要的下标数组！

|  |
| --- |
| 0 1 2 3 4 5 6 7  0 0 3 0 1 3 1 1  0 0 3 3 4 7 8 9 垒加后 |

4、遍历原三元组，开始转置过程。

算法时间复杂度、空间复杂度分析：

空间复杂度：列阶+1

时间复杂度：O(2\*元素个数 + 列阶)