数据结构 第九讲

数组与矩阵

刘烨斌

清华大学自动化系

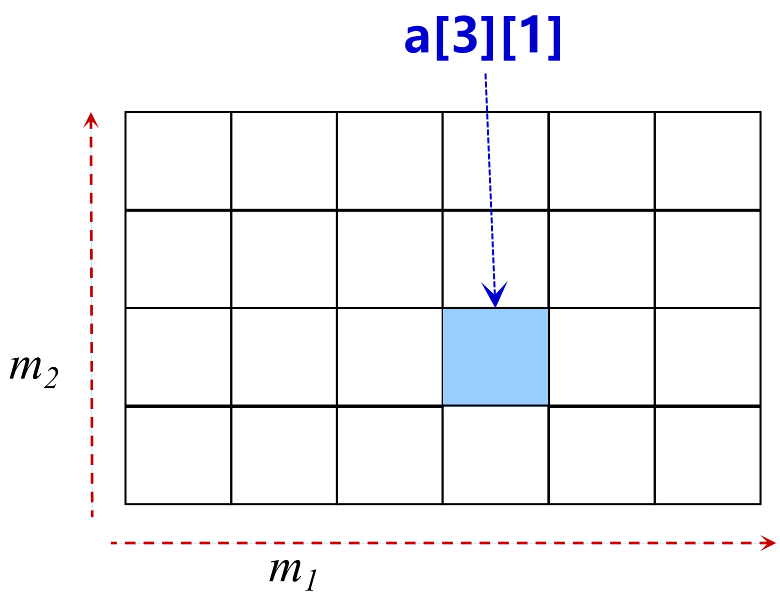
2024年5月14日

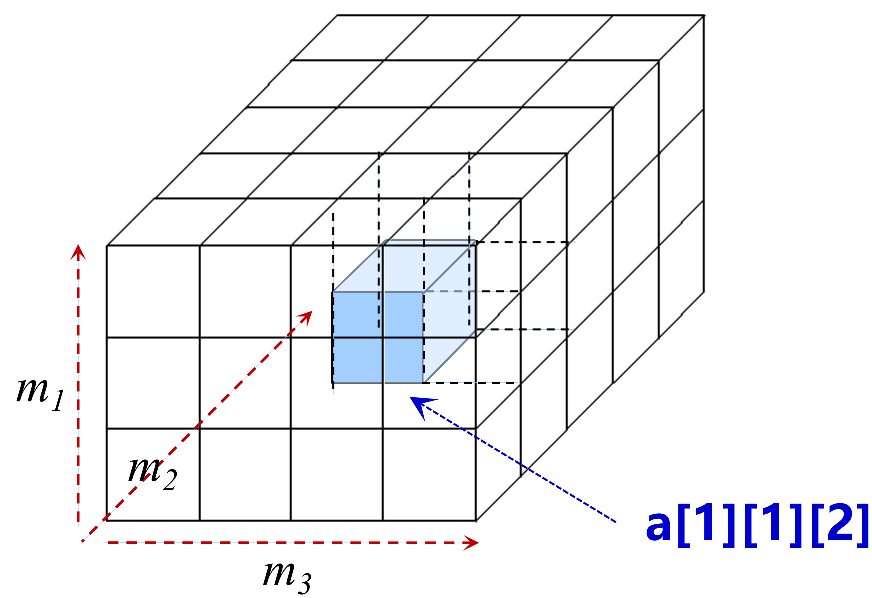
回顾：从数组到向量

数组	向量
高级程序设计语言 内置的数据类型	数组的抽象和泛化， 由模板类实现
通过下标(Index)访问	通过秩访问 (若元素e有r个前驱元素， 其秩为r)
只能读取和修改	带有很多操作接口

数组：基本概念

- **(多维)数组: 由下标(index)和值(value)组成的序列的集合**
  - ✓ C++中有静态数组和动态数组;
  - ✓ 从定义上来看，线性表和数组都是数据元素的有序集
  - ✓ 数组有维度（比如三维数组）的概念而线性表没有
  - ✓ 数组一般不进行数据插入和删除操作





数组：基本概念

- **(多维)数组: 扩展一维数组概念，可定义多维数组**
  - ✓ "数组元素为一维数组"的一维数组，可以视为二维数组
  - ✓ "数组元素为二维数组"的一维数组，可以视为三维数组

```
int a[2][3];
```

等价于

```
typedef int A[3]; //为数组定义简洁的类型名称
A a[2];
```

两个变量组成的一个数组，其中每一个变量都是数组。其中a[0]，a[1]都是数组的名字，也就是地址

a[1][2]
a[1][1]
a[1][0]
a[0][2]
a[0][1]
a[0][0]

数组：基本概念

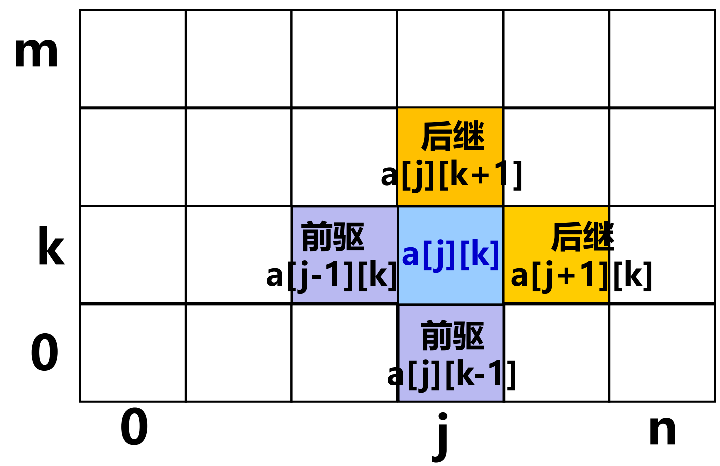
- **思考： (多维)数组与多维向量的区别?**

```
float f[5][5];

vector<vector<float>> f;
```

数组：基本概念

- 多维数组：多个前驱，多个后继



- 在一个n维数组A$[m_1][m_2]\ldots[m_n]$中，总共有$m_1 \times m_2 \times \ldots \times m_n$个数组元素，每一个元素a$[i_1][i_2]\ldots[i_n]$$(0\le i_1 \le m_1\text{-}1,\ldots, 0\le i_n \le m_n\text{-}1)$处于n个向量之中，其位置由下标的n元组$[i_1][i_2]\ldots[i_n]$唯一确定。
- 一维数组a[n]: 设起始存储地址为a，每一数组元素大小为s, 则任意数组元素的存储地址LOC(i)满足,

$$LOC(i) = LOC(i-1)+s = a+i\times s$$

多维数组的存储表示

- 数组是多维的结构，而存储空间是一维结构，对于二维数组

$$A = \begin{bmatrix} a[0][0] & a[0][1] & a[0][2] & \cdots & a[0][m-1] \\ a[1][0] & a[1][1] & a[1][2] & \cdots & a[1][m-1] \\ a[2][0] & a[2][1] & a[2][2] & \cdots & a[2][m-1] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a[n-1][0] & a[n-1][1] & a[n-1][2] & \cdots & a[n-1][m-1] \end{bmatrix}$$

- **行优先存储：大多数程序设计语言（ALGOL、PASCAL、C/C++、Basic）**

  $a[0][0], a[0][1], a[0][2] \ldots, a[0][m-1], a[1][0], a[1][1] \ldots, a[1][m-1], \ldots,$

  $a[\mathrm{n}-1][0], a[\mathrm{n}-1][1], a[\mathrm{n}-1][2] \ldots, a[\mathrm{n}-1][m-1]$

- **列优先存储：FORTAN语言**

  $a[0][0], a[1][0], a[2][0] \ldots, a[\mathrm{n}-1][0], a[0][1], a[1][1] \ldots, a[\mathrm{n}-1][1], \ldots,$

  $a[0][\mathrm{m}-1], a[1][\mathrm{m}-1], a[2][\mathrm{m}-1] \ldots, a[\mathrm{n}-1][m-1]$

多维数组的存储表示

■ 行优先二维数组地址映射

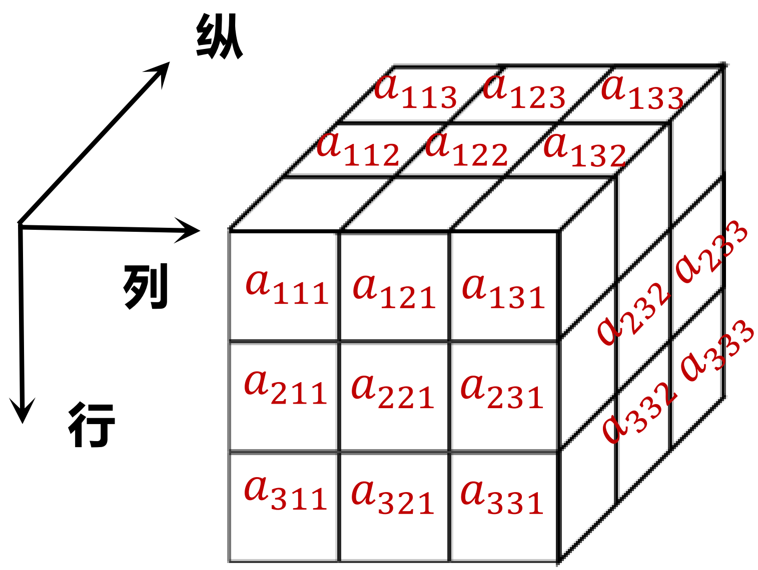
设二维数组A[n][m]**的第一个元素**a[0][0]**地址为** α **,每个元素占用** s **个元素空间，那么任一数组元素**a[j][k]**的存放位置为：**

$$LOC(j,k) = LOC(j,0) + k \times s = LOC(0,0) + j \times m \times s + k \times s = \alpha + (j \times m + k) \times s$$

■ 行优先三维数组地址映射

如果对A$_{3\times4\times2}$（下标从1开始）采用以行为主序的方法存放，即行下标变化最慢，纵下标变化最快，则顺序为：

a_{111}，a_{112}，a_{113}，a_{121}，…，a_{323}，a_{331}，a_{332}，a_{333}



多维数组的存储表示

行优先三维数组地址映射

设二维数组$A[\mathrm{m}_1][\mathrm{m}_2][\mathrm{m}_3]$的第一个元素$a[0][0][0]$地址为$a$,每个元素占用 s 个元素空间，那么任一数组元素 $a[i_1][i_2][i_3]$ 的存放位置为：

$$\mathrm{LOC}\,(\,i_1,\,i_2,\,i_3\,) = a + \,(\,i_1 * m_2 * m_3 + i_2 * m_3 + i_3\,) * s$$

n维数组地址映射

对于多维数组 $A_{b_1 b_2 \ldots b_n}$的任一元素 $a_{j_1 j_2 \ldots j_n}$

以行为主序优先存储时的存储地址计算公式为：

$$LOC(j_1, j_2, \cdots j_n) = LOC(0,0,\cdots,0) + (b_2 \times \cdots b_n \times j_1 + b_3 \times \cdots b_n \times j_2 + \cdots + b_n \times j_{n-1} + j_n)L$$

$$= LOC(0,0,\cdots,0) + \sum_{i=1}^{n} c_i j_i L$$

其中： $c_n = 1, c_{i-1} = b_i \times c_i, 1 < i \le n$

c_i 称为n维数组的映像函数

练习

- 1.二维数组A[0…3][0…3]的元素起始地址是LOC（A[0][0]）=1000,元素的长度为2，则LOC(A[2][2])为多少?

  LOC(A[2][2])= LOC（A[0][0]）+（2*4+2）*2=1020

- 2.数组A[1…10, -2…6, 2…8]以行优先顺序存取，设第一个元素的首地址为100，每个元素占3个单元的存储空间，则元素A[5][0][7]的存储地址为（）

  LOC(A[5][0][7])=100+[(5-1)*(6-(-2)+1)*(8-2+1)+(0-(-2))*(8-2+1)+(7-2)]*3=913

二维数组的抽象数据类型定义

ADT Array {

数据对象:

$D = \{a_{ij} \mid 0 \le i \le b_1-1,\ 0 \le j \le b_2-1\}$

数据关系:

$R = \{ ROW, COL \}$

$ROW = \{<a_{i,j}, a_{i,j+1}> \mid 0 \le i \le b_1-1,\ 0 \le j \le b_2-2\}$

$COL = \{<a_{i,j}, a_{i+1,j}> \mid 0 \le i \le b_1-2,\ 0 \le j \le b_2-1\}$

基本操作:

}ADT Array

填空题 1分

设置

数组A[1...10, 2...6, 2...8]以行优先顺序存取，设第一个元素的首地址为100，每个元素占3个单元的存储空间，则元素A[6][3][7]的存储地址为 [填空1]

正常使用填空题需3.0以上版本雨课堂

作答

练习

3、假设有一个8X5X9的三维数组.A_{8x5x6} => a_{ijk} 以行为主序，请回答:

$a_{2,5,6}$的地址是 LOC(a_{111})+t*L , t是多少?

第222个元素的下标是?

Loc [2,5,6] =Loc [1, 1, 1] +[(2-1)×5×9+(5-1)×9+(6-1)]*L; t=45+36+5=86

221%45=4……41 i=5

41%9=4……5 j=5

K=5+1=6

LOC[5,5,6]

矩阵的压缩存储

在高级语言编程时，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。

但对于某些矩阵，特别是高阶矩阵，若其中零元素或非零元素呈某种规律分布，或者矩阵中有大量的零元素，若仍然用常规方法存储，可能存储重复的非零元素或零元素，将造成存储空间的大量浪费。对这类矩阵进行压缩存储：

- 多个相同的非零元素只分配一个存储空间
- 零元素不分配空间

特殊矩阵：非零元素或零元素的分布有一定规律的矩阵

稀疏矩阵：存在大量零元素的矩阵，非零元素分布无规律

对称矩阵及其压缩存储

- n阶方阵$A=(a_{ij})$ 满足:$a_{ij}=a_{ji},1\leq i,j\leq n$且$i\neq j$，则A为对称矩阵

$$A=\begin{bmatrix}1&5&6&0\\5&2&7&8\\6&7&3&9\\0&8&9&4\end{bmatrix}$$

$$\begin{bmatrix}a_{11}&&&\\a_{21}&a_{22}&&\\a_{31}&a_{32}&a_{33}&\\a_{41}&a_{42}&a_{43}&a_{44}\end{bmatrix}$$

有效数据仅为下三角矩阵，共n(n+1)/2个

M=	a_{11}	a_{21}	a_{22}	a_{31}	…	a_{n1}	…	a_{nn}
k =	0	1	2	3		n(n-1)/2		n(n+1)/2

映射关系：A_{ij} = M_k

$$\mathbf{k}=\begin{cases}\dfrac{i(i-1)}{2}+j-1 & \text{当}i\geq j\\[2ex]\dfrac{j(j-1)}{2}+i-1 & \text{当}i<j\end{cases}$$

三角矩阵及其压缩存储

- **以主对角线划分，三角矩阵有上三角和下三角两种**
  - **上三角矩阵的下三角（不包括主对角线）中元素均为常数c(一般为0)**
  - **下三角矩阵正好相反，它的主对角线上方均为常数**

a_{11}	c	···	c
a_{21}	a_{22}	···	c
···	···	···	···
a_{n1}	a_{n2}	···	a_{nn}

下三角矩阵

a_{11}	a_{12}	···	a_{1n}
c	a_{22}	···	a_{2n}
···	···	···	···
c	c	···	a_{nn}

上三角矩阵

图中黄色部分为常数，一般为0

三角矩阵及其压缩存储

- 三角矩阵可用一维数组M[n×(n+1)/2+1]来存储，其中常数C放在数组的最后一个下标变量中

可得下三角矩阵的存储单元M[k] 的下标与a_{ij} 的下标i、j的对应关系为:

映射关系：$A_{ij} = M_k$

$$\mathbf{k} = \begin{cases} \dfrac{i(i-1)}{2} + j - 1 & \text{当i≥j} \\ \dfrac{n(\mathrm{n}+1)}{2} & \text{当i<j} \end{cases}$$

对角矩阵及其压缩存储

- 除了主对角线和主对角线上或下方若干条对角线上的元素之外，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

a_{11}	a_{12}	0	…	0	0
a_{21}	a_{22}	a_{23}	0	0	0
0	a_{32}	a_{33}	…	0	…
…	0	…	…	…	0
0	0	0	…	a_{n-1n-1}	a_{n-1n}
0	0	…	0	a_{nn-1}	a_{nn}

对角矩阵

如图**三对角矩阵**，非零元素仅出现在主对角(a_{ii},1≦i≦n)上、主对角线**上**的那条对角线($a_{i\ i+1}$,1≦i≦n-1)、主对角线**下**的那条对角线上($a_{i+1\ i}$,1≦i≦n-1)。

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)A是满足下述条件：

当| i-j |>(k-1)/2时， $a_{i\ j}$=0

对角矩阵及其压缩存储

- 以三对角矩阵为例，当i=1，j=1、2，或i=n， j=n-1、n或1<i<n, j=i-1、i、i+1的元素a_{ij}外，其余元素都是0
- 按“行优先顺序”存储时， 第1行和第n行是2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
- 对应的一维存储结构M中，在a_{ij}之前矩阵有i-1行,因此对应M中共有$3(i-1)-1$个非零元素；在第i行，有$j-i+1$个非零元素，这样，非零元素a_{ij}的地址为：

$$LOC[a_{ij}] = LOC[a_{11}] + [3(i-1)-1+(j-i+1)] \times L = LOC[a_{11}] + (2i+j-3) \times L$$

例如，a_{34}对应着M[7]，因 $k = 2i + j - 3 = 7$

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

填空题 1分

设置

一个10阶对称矩阵A，采用行主序方式压缩存储，a11为第一个元素，其存储地址为1，每一个元素占一个地址空间，则a94的地址为 [填空1]

作答

稀疏矩阵

- **稀疏矩阵(Sparse Matrix)**：设矩阵A是一个m×n的矩阵中有t个非零元素，设 δ=t/(m×n)，称δ为稀疏因子，通常如果某一矩阵的稀疏因子δ满足δ≦0.05时称为稀疏矩阵

	12	9					
-3							4
		24			2		
	18						
						-7	
			-6				

稀疏矩阵

稀疏矩阵压缩存储

- 仅存储非零元素
- 三元组（i,j,aij）唯一确定非零元素，使用三元组线性表
- 假设以行序为主序，顺序存储结构，可得三元组顺序表

```
template<typename E>
struct Triple        // 三元组
{
    int row, col; // 非零元素行号/列号
    E value;      //非零元素的值
    void operator = (Triple<E> &R){
       row = R.row;
       col = R.col;
       value = R.value;
    }
};
```

行	列	值

三元组结点

稀疏矩阵压缩存储

三元组顺序表定义

```
template <typename E>class SparseMatrix{ //稀疏矩阵定义
private:
    int Rows, Cols, Terms;   // 行数、列数、非零元素个数
    Triple<E> *smArray;      // 三元组顺序表
    int maxTerms;            // 最大可能非零元素个数
};
```

	12	9					
-3							4
		24			2		
	18						
						-7	
			-6				

行主序顺序存储顺序表：

**((0,1,12),(0,2,9),(2,0,-3),
(2,7,4),(3,2,24),(4,1,18),
(5,6,-7),(6,3,-6))**

稀疏矩阵压缩存储

- 稀疏矩阵三元组表存储中元素的位置和下标没有关系，因此无法依靠下标进行矩阵运算

	12	9					
-3							4
		24			2		
	18						
						-7	
			-6				

Rows, Cols, Terms

7	8	9
0	1	12
0	2	9
2	0	-3
2	7	4
3	2	24
3	5	2
4	1	18
5	6	-7
6	3	-6

原矩阵三元组表

8	7	9
0	2	-3
1	0	12
1	4	18
2	0	9
2	3	24
3	6	-6
5	3	2
6	5	-7
7	1	4

转置矩阵三元组表

需新算法进行矩阵转置、矩阵求逆、矩阵加减、矩阵乘除等

稀疏矩阵的转置（慢速转置）

- 原矩阵A行优先顺序存储压缩
- 对每项交换row(i)和col(j)值，得列优先顺序存储压缩矩阵B
- 对B中三元组表进行行优先重排列

原矩阵A三元组表(行优先)

7	8	9
0	1	12
0	2	9
2	0	-3
2	7	4
3	2	24
3	5	2
4	1	18
5	6	-7
6	3	-6

转置矩阵B三元组表(列优先)

8	7	9
1	0	12
2	0	9
0	2	-3
7	2	4
2	3	24
5	3	2
1	4	18
6	5	-7
3	6	-6

按照B的row值重排列

转置矩阵B三元组表(行优先)

8	7	9
0	2	-3
1	0	12
1	4	18
2	0	9
2	3	24
3	6	-6
5	3	2
6	5	-7
7	2	4

稀疏矩阵的转置（慢速转置）

- 实际操作时将上述两步合二为一
- 逐趟扫描三元组序列，第k趟提取col值为k的三元组，放入目标压缩矩阵B

原矩阵A
三元组表
(行优先)

7	8	9	
0	1	12	第2趟
0	2	9	第3趟
2	0	-3	第1趟
2	7	4	第8趟
3	2	24	第3趟
3	5	2	第6趟
4	1	18	第2趟
5	6	-7	第7趟
6	3	-6	第4趟

转置矩阵B
三元组表
(行优先)

8	7	9
0	2	-3
1	0	12
1	4	18
2	0	9
2	3	24
3	6	-6
5	3	2
6	5	-7
7	2	4

稀疏矩阵的转置（慢速转置）

```
template <typename E>SparseMatrix<E> SparseMatrix<E>::Transpose(){
        SparseMatrix<E> B(maxTerms,Cols,Rows);
    if (Terms > 0){
        int CurrentB = 0; int i, k;
        for (k = 0; k < Cols; k++){          //按列号扫描
            for (i = 0; i < Terms; i++){    //在数组中找列号为k的三元组
                if (smArray[i].col == k){  //这样所得转置矩阵三元组有序
                    B.smArray[CurrentB].row = k;
                    B.smArray[CurrentB].col = smArray[i].row;
                    B.smArray[CurrentB].value=smArray[i].value;
                    CurrentB++;
                }
            }
        }
    }
    B.Terms=Terms;
    return B;
}
```

时间复杂度：O(Cols×Terms)

若Terms数量级等价于Cols×Rows,则复杂度为$O(Cols^2 \times Rows)$

稀疏矩阵的转置（快速转置）

- 算法思想：顺序扫描原三元组表示，直接对每个放入正确位置
- 如何知道正确位置？预先构建**各列的非零元素个数表**

rowSize = 各列的非零元素个数
= 转置后各行的非零元素个数

```
for (i = 0; i < Cols; i++) rowSize[i] = 0;
for (i = 0; i < Terms; i++) rowSize[smArray[i].col]++;
```

rowStart = 各行的非零元素在转置矩阵的三元组表中应存放的起始位置

```
rowStart[0] = 0;
for (i = 1; i < Cols; i++)
    rowStart[i] = rowStart[i-1] + rowSize[i-1];
```

col	0	1	2	3	4	5	6	7
rowSize	1	2	2	1	0	1	1	1
rowStart	0	1	3	5	6	6	7	8

7	8	9
0	1	12
0	2	9
2	0	-3
2	7	4
3	2	24
3	5	2
4	1	18
5	6	-7
6	3	-6

A三元组表
(行优先)

8	7	9
0	2	-3
1	0	12
1	4	18
2	0	9
2	3	24
3	6	-6
5	3	2
6	5	-7
7	2	4

B三元组表
(行优先)

注意：rowStart为转置矩阵各行非零元素的起始位置

稀疏矩阵的转置（快速转置）

稀疏矩阵快速转置练习

$$M=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

col	0	1	2	3	4	5	6
rowSize	2	2	2	1	0	1	0
rowStart	0	2	4	6	7	7	8

6	7	8
0	1	12
0	2	9
2	0	-3
2	5	14
3	2	24
4	1	18
5	0	15
5	3	-7

M矩阵三元组表
(行优先)

稀疏矩阵的转置（快速转置）

稀疏矩阵快速转置示例代码

```
for (int i = 0; i < Terms; i++){ // 遍历A三元组各项
        int j = rowStart [smArray[i].col];
        // 取出起始位置
        B.smArray[j].row = smArray[i].col;
        B.smArray[j].col = smArray[i].row;
        // B矩阵行号列号对调
        B.smArray[j].value = smArray[i].value;
        // 赋值域
        rowStart [smArray[i].col]++;
        // 起始位置加1
}
```

rowStart值加1，下次遇到该列，可以当作该列的第一个元素，用相同的方法直接计算

col	0	1	2	3	4	5	6	7
rowSize	1	2	2	1	0	1	1	1
rowStart	0	1 **2**	3 **4**	5	6	6	7	8

时间复杂度：O(Cols+Terms)

7	8	9
0	1	12
0	2	9
2	0	-3
2	7	4
3	2	24
3	5	2
4	1	18
5	6	-7
6	3	-6

A三元组表
（行优先）

8	7	9
0	2	-3
1	0	12
1	4	18
2	0	9
2	3	24
3	6	-6
5	3	2
6	5	-7
7	2	4

B三元组表
（行优先）

稀疏矩阵的转置（快速转置）

- 稀疏矩阵快速转置练习

col	0	1	2	3	4	5	6
rowStart	2	4	6	7	7	8	8

M矩阵三元组表（行优先）

6	7	8
0	1	12
0	2	9
2	0	-3
2	5	14
3	2	24
4	1	18
5	0	15
5	3	-7

装置矩阵三元组表（行优先）

	7	6	8
0	0	2	-3
1	0	5	15
2	1	0	12
3	1	4	18
4	2	0	9
5	2	3	24
6	3	5	-7
7	5	2	14

填空题 2分

设置

以下稀疏矩阵进行快速转置时，构建的rowsize和rowstart表，表中rowsize各元素的和为 [填空1] ，rowstart各元素的和为 [填空2]

$$M=\begin{pmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{pmatrix}$$

正常使用填空题需3.0以上版本雨课堂

作答

稀疏矩阵乘法与行逻辑链接三元组顺序表

- 将上述辅助向量rowStart[ ]固定在稀疏矩阵的三元组表中，用来指示“行”的信息，得到另一种顺序存储结构：行逻辑链接的三元组顺序表
- 注意：前述快速矩阵转置中的rowStart为原始矩阵的转置矩阵的行起始指示，而非原始矩阵本身的

```
template <typename E>class SparseMatrix{
private:
    int Rows, Cols, Terms;  // 行数、列数、非零元素个数
    Triple<E> *smArray;    // 三元组顺序表
     int *rowStart;        // 各行第一个非零元的位置表
    int maxTerms;          // 最大可能非零元素个数
};
```

行逻辑链接三元组顺序表

row	0	1	2	3	4	5	
rowSize	2	0	2	1	1	1	
rowStart	0	2	2	4	5	6	7

6	7	7
0	1	12
0	2	9
2	0	-3
2	5	14
3	2	24
4	1	18
5	0	15

单选题 1分

设置

矩阵A,B进行经典矩阵乘法，即最原始的计算方法，A的每行与B的每列进行内积求和。设矩阵的横列分别为A.Rows, A.Cols, B.Rows, B.Cols，则经典矩阵的计算复杂度为

A. A.Rows×B.Cols

B. A.Rows×B.Cols×A.Cols

C. A.Rows×B.Cols×A.Cols×B.Rows

D. A.Rows×B.Cols× A.Rows×B.Cols

提交

稀疏矩阵乘法与行逻辑链接三元组顺序表

稀疏矩阵乘法

$$A_{3\times 4}=\begin{bmatrix}10 & 0 & 5 & 7\\ 2 & 1 & 0 & 0\\ 3 & 0 & 4 & 0\end{bmatrix}\quad B_{4\times 2}=\begin{bmatrix}2 & 0\\ 4 & 8\\ 0 & 14\\ 3 & 5\end{bmatrix}\quad C=A\times B=\begin{bmatrix}41 & 105\\ 8 & 8\\ 6 & 56\end{bmatrix}$$

核心算法： $C[i][j]=\sum_{k=0}^{n-1}A[i][k]\times B[k][j]$

推论：任意 $A[i][k]$ **必然与** $B[k][j]$ **相乘，结果累加入**$C[i][j]$

矩阵乘法核心思想：

- **遍历A中任意非零元素**$A[i][k]$**，其行列分别为**i,k**；**
- **在B中遍历搜索行号为** k **的任意元素** $B[k][j]$ **相乘，结果累加入**$C[i][j]$

遍历搜索行号为 k **的元素可由rowStart数组直接给出**

稀疏矩阵乘法与行逻辑链接三元组顺序表

```
template <typename E> SparseMatrix<E>
SparseMatrix<E>::Multiply(SparseMatrix<E> &b){
       SparseMatrix<E> result(Rows*b.Cols/2,Rows, b.Cols);
       if (Cols != b.Rows){
              cout << "Incompatible matrices" << endl;
              return result;
       }
       int *rowSize = new int[b.Rows];// 矩阵B 各行非零元素个数
       rowStart = new int[b.Rows+1];// 矩阵B 各行在三元组开始位置
       E *temp = new E[b.Cols];     // 暂存每一行计算结果
       int i, Current, lastInResult, RowA, ColA, ColB;
       for (i = 0; i < b.Rows; i++) rowSize[i] = 0;
       for (i = 0; i < b.Terms; i++) rowSize[b.smArray[i].row]++;
              rowStart[0] = 0;//B第i 行非零元素开始位置
       for (i = 1; i <= b.Rows; i++)
              rowStart[i] = rowStart[i-1]+rowSize[i-1];
       Current = 0; lastInResult = -1;//a 扫描指针及result 存指针
```

稀疏矩阵乘法与行逻辑链接三元组顺序表

```
    while (Current < Terms){//生成result 的当前行temp
        RowA = smArray[Current].row;//当前行的行号
        for (i = 0; i < b.Cols; i++) temp[i] = 0;
        while(Current<Terms && smArray[Current].row==RowA){//处理该行各元素
            ColA = smArray[Current].col; //矩阵A 当前扫描到元素的列号
           for (i = rowStart[ColA]; i < rowStart[ColA+1]; i++){
                ColB = b.smArray[i].col; //矩阵B 中相乘元素的列号
                temp[ColB] += smArray[Current].value*b.smArray[i].value;
            } //A的RowA 行与B 的ColB 列相乘
            Current++;
        }
        for (i = 0; i < b.Cols; i++){
            if (temp[i] != 0){//将temp 中的非零元素压缩到result 中去
                lastInResult++;
                result.smArray[lastInResult].row = RowA;//行号固定
                result.smArray[lastInResult].col = i;//列号对应temp下标
               result.smArray[lastInResult].value = temp[i];
            }
        }
    }
    result.Rows = Rows;result.Cols = b.Cols;result.Terms = lastInResult+1;
    delete []rowSize;delete []rowStart;delete []temp;
    return result;
    }
```

稀疏矩阵乘法与行逻辑链接三元组顺序表

- 稀疏矩阵乘法复杂度

  约为 $O(A.Terms \times B.Cols)$

- 经典矩阵乘法复杂度

```
for (int i = 0; i < A.Rows; i ++)
    for (int j = 0; j <B.Cols; j ++) {
     Q[i][j] = 0;
     for (int k = 0; k <A.Cols; k ++)
           Q[i][j] + = A[i][k] * B[k][j];
       }
```

$$O(A.Rows \times B.Cols \times A.Cols)$$

矩阵十字链表（正交链表）

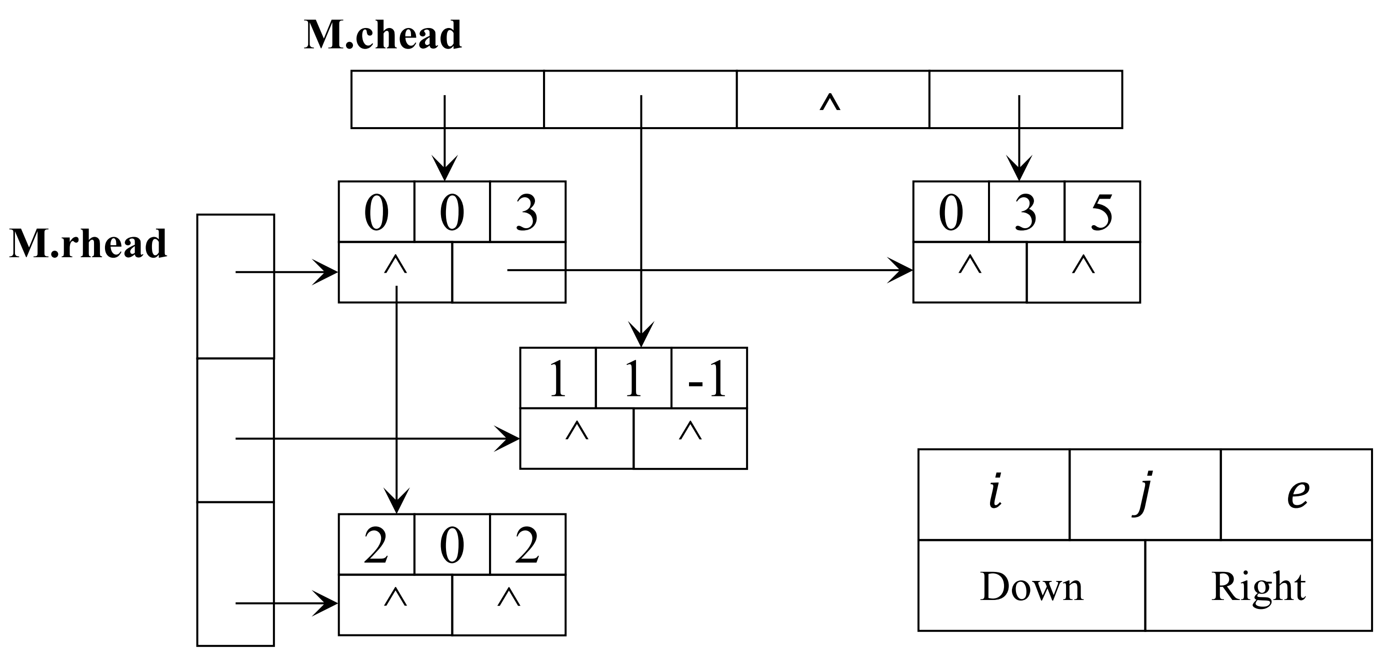
- 矩阵运算如加减乘除等运算**非零元素的位置和个数经常发生变化**，就不宜采用三元组表，此时采用链式存储结构来表示三元组方便一些
- 采用“十字链表”的链式存储结构
  - 每个非零元素结点中除了非零元素的三元组(i，j，v)外，
  - 增加了两个链域：向下域（down）和向右域（right）
  - 其中向下域用于链接同一列中的非零元素，向右域（right）用于链接同一行中的非零元素

```
typedef struct OLNode{
    int  i, j;
    ElemType e;
    struct  OLNode  *down, *right;
} OLNode, *OLink;   //结点
```

```
typedef struct {
    Olink *rhead, *chead;
    // 矩阵行头链和列头链序列
    int Rows, Cols, Terms;
}CrossList;   //十字链表
```

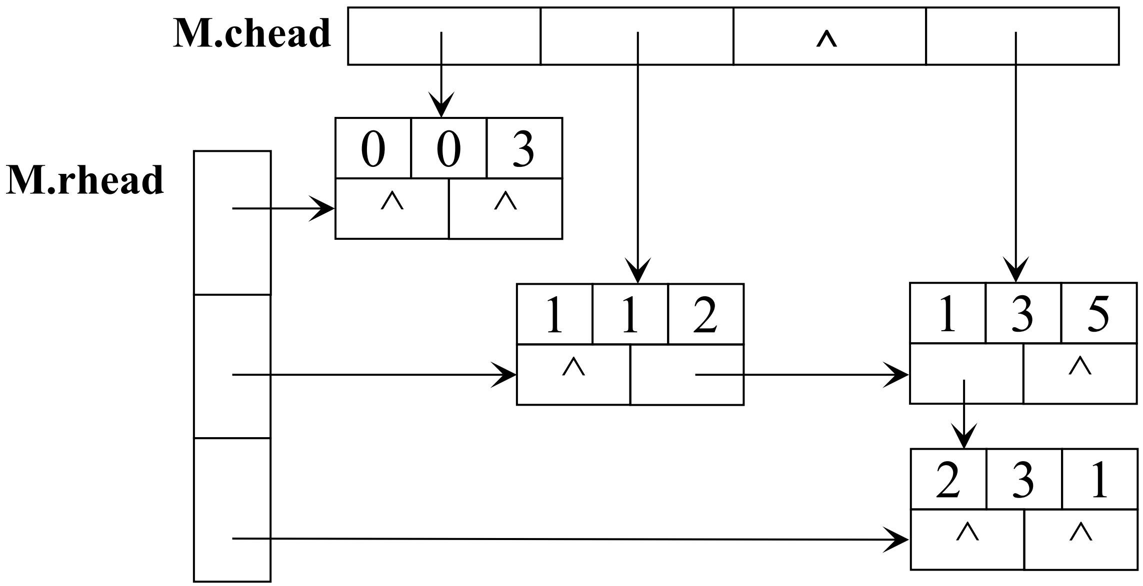
矩阵十字链表（正交链表）

例如，矩阵M = $\begin{pmatrix} 3 & 0 & 0 & 5 \\ 0 & -1 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{pmatrix}$ 的十字链表如下所示

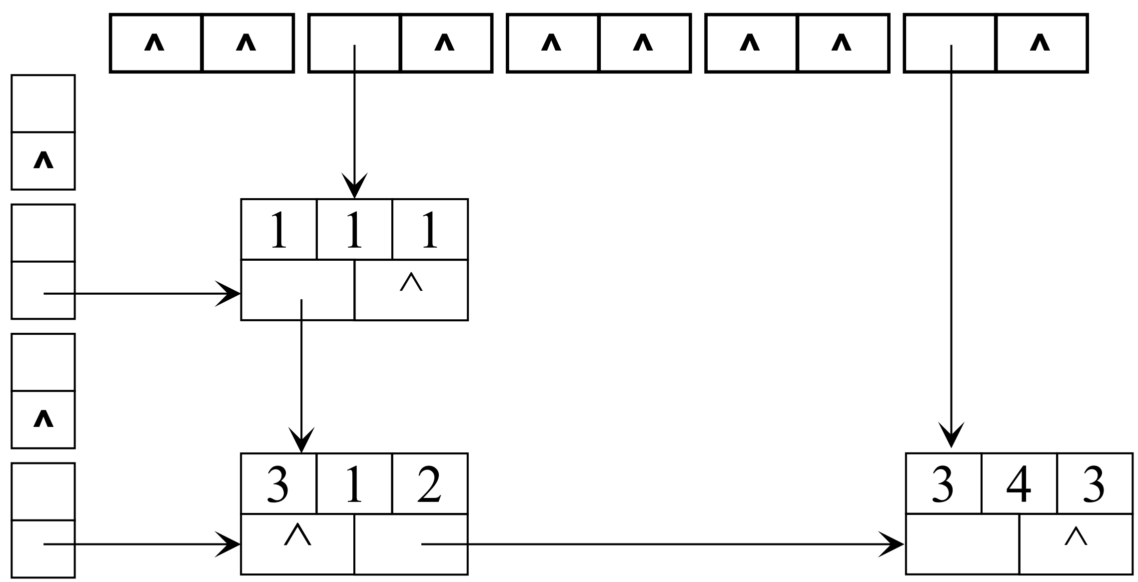


稀疏矩阵M的十字链表

矩阵十字链表（正交链表）



根据图形，写出矩阵



矩阵十字链表及稀疏矩阵加法

- 矩阵A+矩阵B=矩阵A′ ,其非零元素 a'_{ij} 可能3情况
  - $a_{ij}+b_{ij}$, 改变A中节点值($a_{ij}+b_{ij}\neq 0$)
  - a_{ij}(b_{ij}=0), 保持不变
  - b_{ij}(a_{ij}=0)，加入新节点
- 还可能是删除节点 a_{ij}， $a_{ij}+b_{ij}\neq 0$

共4种情况

假设非空指针 pa 和 pb 分别指向矩阵 A 和 B 中行值相同的两个结点，*pa*==NULL 表明矩阵 A 在该行中没有非零元，则上述 4 种情况的处理过程为：

(1) 若 pa==NULL 或 pa->j>pb->j，则需要在 A 矩阵的链表中插入一个值为 b_{ij} 的结点。此时，需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->j<pb->j，则只要将 pa 指针往右推进一步。

(3) 若 pa->j==pb->j 且 pa->e+pb->e !=0，则只要将 $a_{ij}+b_{ij}$ 的值送到 pa 所指结点的 e 域即可，其他所有域的值都不变。

(4) 若 pa->j==pb->j 且 pa->e+pb->e==0，则需要在 A 矩阵的链表中删除 pa 所指的结点。此时，需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。

矩阵十字链表及稀疏矩阵加法

■ 稀疏矩阵加法伪代码

```
令pa和pb分别指向第一行的第一个元素;
while (矩阵未处理完)   {
  while  (本行未处理完)  {
     if  (pa==NULL 或 pa->j > pb->j)
        {在A中插入pb所指结点; 然后pb指向B中本行下一个结点; }
     else  if (pa->j < pb->j)
           { 只要将pa指针往右推进一步; }
          else  if (pa->j == pb->j)  {
                 if (pa->e + pb->e !=0)
                    {将pa->e + pb->e的值送到pa->e;}
                 else
                    {在A 中删除pa所指的结点 ; }
                 pa和pb指针分别指向本行下一个结点;  }
     pa和pb分别指向下一行第一个元素;
  }  //while
}
```