

中国风



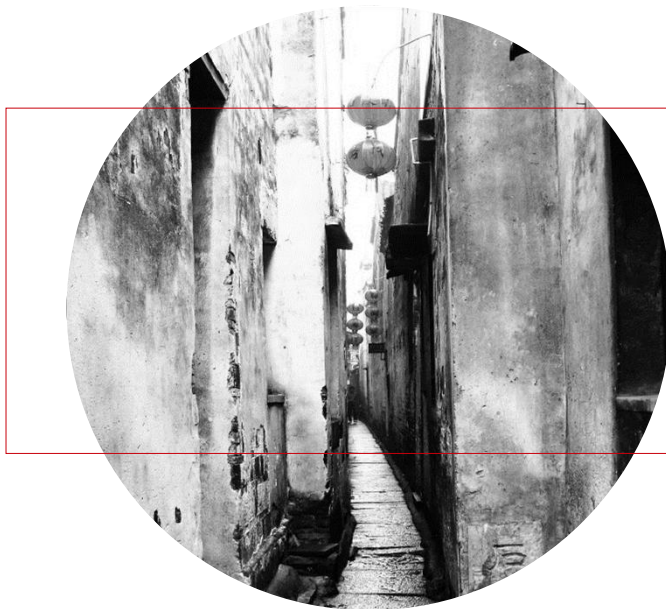
数据结构实验课



实验2.1内容算法提示

题目：稀疏矩阵转置

实现矩阵转置的前提是将矩阵存储起来，数据结构中提供了 3 种存储矩阵的结构，分别是三元组顺序表、行逻辑链接的顺序表和十字链表。（此题目是用三元组方式存）



稀疏矩阵转置（三元组存矩阵）

实验2.1 内容算法提示

三元组的概念：

设 $m \times n$ 矩阵中有 t 个非零元素且 $t \ll m \times n$ ，这样的矩阵称为稀疏矩阵。很多科学管理及工程计算中，常会遇到阶数很高的大型稀疏矩阵。如果按常规分配方法，顺序分配在计算机内，那将是相当浪费内存的。

为此提出另外一种存储方法，**仅仅存放非零元素**。但对于这类矩阵，通常零元素分布没有规律，为了能找到相应的元素，所以仅**存储非零元素的值**是不够的，还要记下**它所在的行和列**。于是采取如下方法：**将非零元素所在的行、列以及它的值构成一个三元组 (i, j, v)** ，然后再按某种规律存储这些三元组，这种方法可以节约存储空间。

实验2.1内容算法提示

三元组的概念：

将三元组按行优先的顺序，同一行中列号从小到大的规律排列成一个线性表，称为三元组表，采用顺序存储方法存储该表。如图5.11 稀疏矩阵对应的三元组表为图5.12。

$$A = \begin{pmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

图 5.11 稀疏矩阵

	i	j	v
1	1	1	15
2	1	4	22
3	1	6	-15
4	2	2	11
5	2	3	3
6	3	4	6
7	5	1	91

图 5.12 三元组表

实验2.1内容算法提示

三元组的概念：

显然，要唯一的表示一个稀疏矩阵，还需要存储三元组表的同时存储该矩阵的行、列，为了运算方便，矩阵的非零元素的个数也同时存储。这种存储的思想实现如下：

```
define SMAX 1024 /*一个足够大的数*/
typedef struct
{ int i,j; /*非零元素的行、列*/
  datatype v; /*非零元素值*/
}SPNode; /*三元组类型*/

typedef struct
{ int mu,nu,tu; /*矩阵的行、列及非零元素的个数*/
  SPNode data[SMAX]; /*三元组表*/
} SPMatrix; /*三元组表的存储类型*/
```

实验2.1内容算法提示

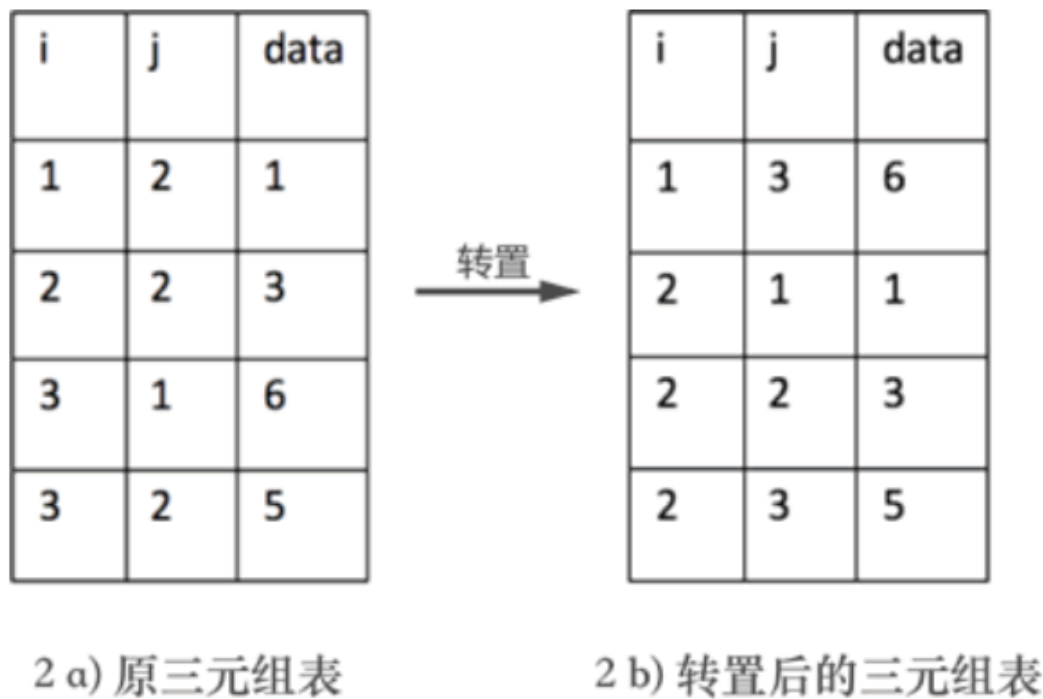


图 2a) 表示的是转置之前矩阵的三元组表，

2b) 表示的是矩阵转置后对应的三元组表。

不仅如此，如果矩阵的行数和列数不等，也需要将它们互换。

因此通过左图分析，矩阵转置的实现过程需完成以下 3 步：

- 1.将矩阵的行数和列数互换；
- 2.将三元组表（存储矩阵）中的 i 列和 j 列互换，实现矩阵的转置；
- 3.以 j 列为序，重新排列三元组表中存储各三元组的先后顺序；

实验2.1内容算法提示

矩阵转置的实现思路是：不断遍历存储矩阵的三元组表，每次都取出表中 j 列最小的那一个三元组，互换行标和列标的值，并按次序存储到一个新三元组表中。

例如，将图 3a) 三元组表存储的矩阵进行转置的过程为：新建一个三元组表（用于存储转置矩阵），并将原矩阵的行数和列数互换赋值给新三元组；遍历三元组表，找到表中 j 列最小值 1 所在的三元组 (3,1,6)，然后将其行标和列标互换后添加到一个新的三元组表中，如图 3 所示：

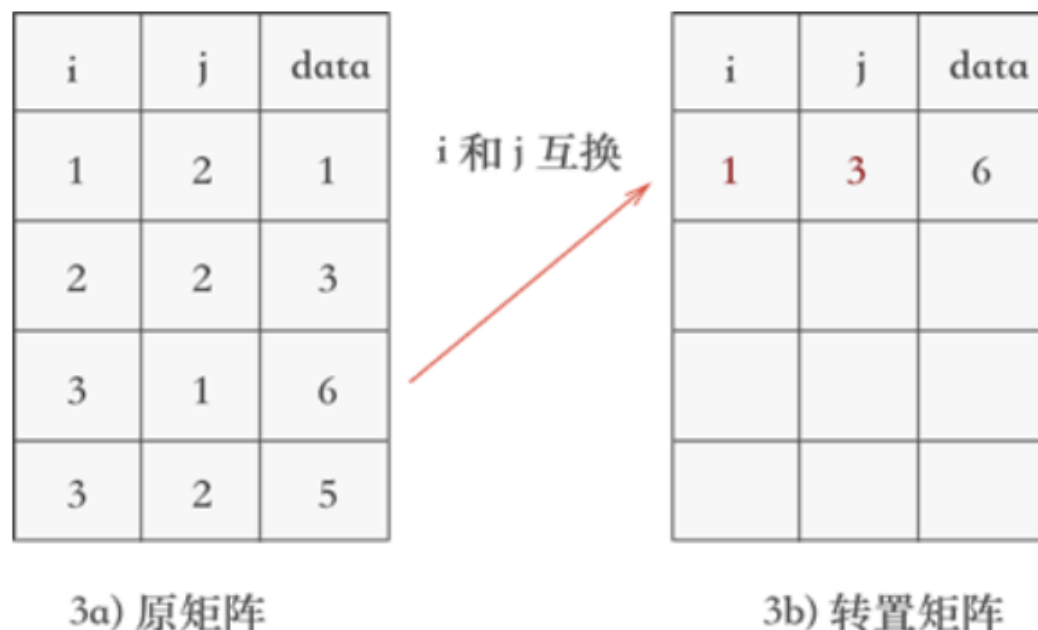
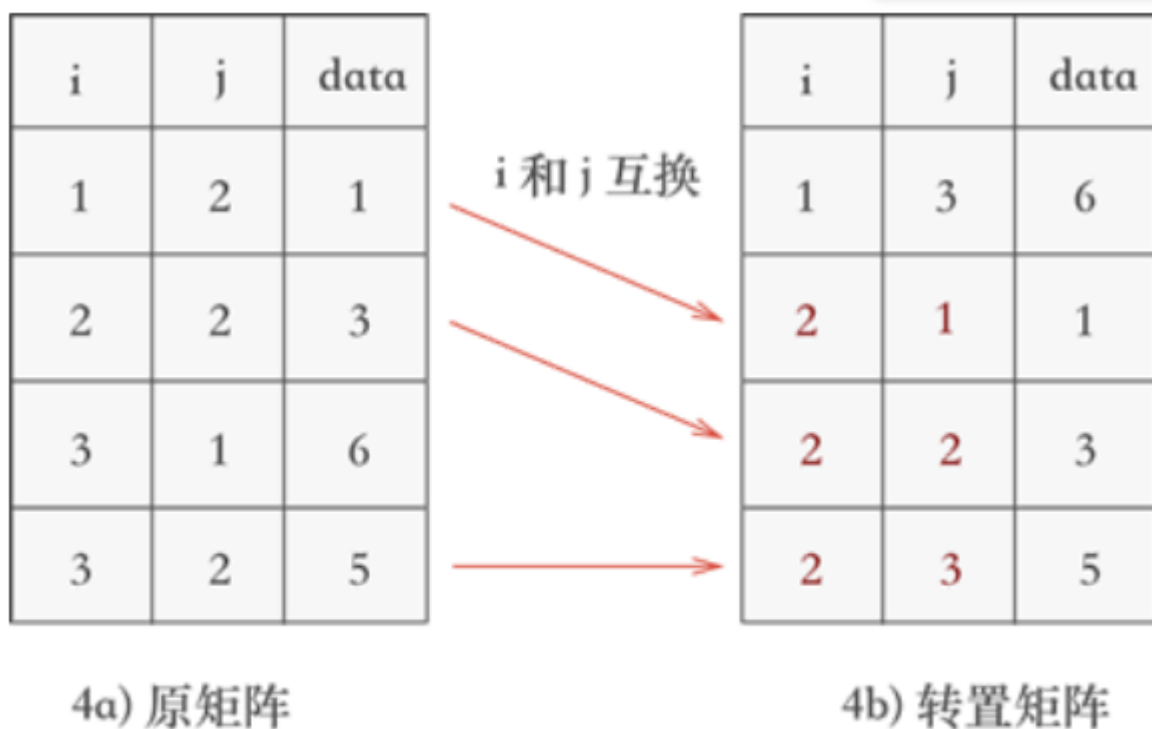


图 3 矩阵转置的第一个过程

实验2.1内容算法提示

继续遍历三元组表，找到表中 j 列次小值为 2 的三元组，分别为 (1,2,1)、(2,2,3) 和 (3,2,5)，根据找到它们的先后次序将各自的行标和列标互换后添加到新三元组表中，如图 4 b所示：



对比图 4 和图 2b) 可以看到，矩阵被成功地转置。

此算法中可以嵌套使用了两个 for 循环完成，时间复杂度为 $O(n^2)$ 。

图 4 矩阵转置的第二个过程

实验2.2内容算法提示

题目：输入两个稀疏矩阵，输出它们相加的结果。 $C = A + B$

Input: 第一行输入四个正整数，分别是两个矩阵的行 m 、列 n 、第一个矩阵的非零元素的个数 t_1 和第二个矩阵的非零元素的个数 t_2 。

接下来的 t_1+t_2 行是三元组，分别是第一个矩阵的数据和第二个矩阵的数据。三元组的第一个元素表示行号，第二个元素表示列号，第三个元素是该项的值。



矩
阵
相
加

实验2.2内容算法提示

方法一：

简单粗暴，给每个矩阵创建 $m*n$ 个三元组进行存储，直接进行相加。

三元组示例：

```
typedef struct {  
    int i,j;  
    int e;  
}Triple;
```

```
typedef struct {  
    Triple data[MAXSIZE+1];  
    int mu,nu,tu;  
}TSMatrix;
```

空间复杂度： $m*n$

时间复杂度： $m*n$



矩
阵
相
加

实验2.2内容算法提示

方法二:

三元组中仅存储非零数字

三元组示例:

```
typedef struct {
```

```
    int i,j;
```

```
    int e;
```

```
}Triple;
```

```
typedef struct {
```

```
    Triple data[MAXSIZE+1];
```

```
    int mu,nu,tu;
```

```
}TSMatrix;
```

空间复杂度: t_1+t_2

时间复杂度: t_1+t_2



矩
阵
相
加

实验2.2内容算法提示

方法二注意点:

1. 大循环 (countA < A.tu && countB < B.tu)

结束后, 要分别对A B矩阵检查是否全部计算。

2. t1 + t2个元素都要进行计算, 不能多次计算, 也不可遗漏。


3. 当两元素位置相同才能进行加减

$A[i] == B[i] \ \&\& \ A[j] == B[j]$

需要两次判断。

右图介绍 $A[i] > B[i]$ 的情况, 其他情况举一反三。

当行数相等之后, 对列数要进行同样的判断。



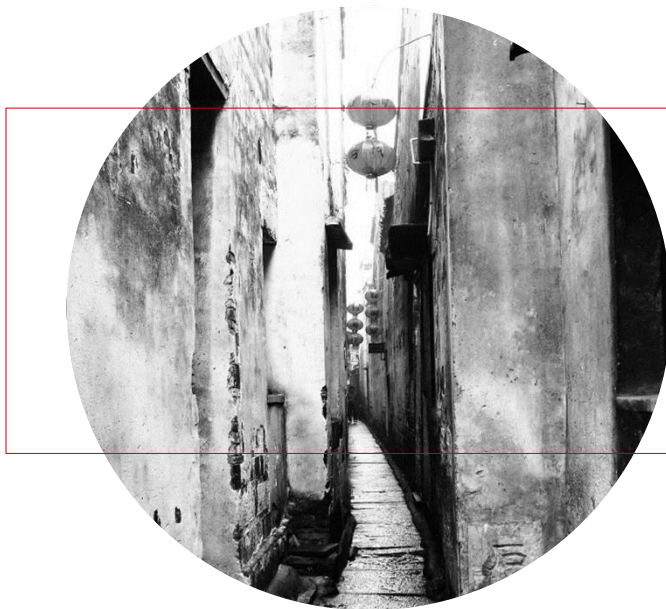
```
while(countA <= A.tu && countB <= B.tu){
    ai = A.data[countA].i;
    bi = B.data[countB].i;
    if(ai > bi){ //A的行数 > B的行数
        ci = bi; //ci为该对象的行数
        while(ci == B.data[countB].i){ //若B的行数一直不变
            C.data[countC].i = ci; //行数
            C.data[countC].j = B.data[countB].j; //列数
            C.data[countC].e = B.data[countB].e; //数值
            ++countB;
            ++countC;
        }
    }
    else if(ai < bi){
        //
    }
    else if(ai == bi){
        //
    }
}
```

实验2.2内容算法提示

方法二:

空间复杂度: t_1+t_2

时间复杂度: t_1+t_2

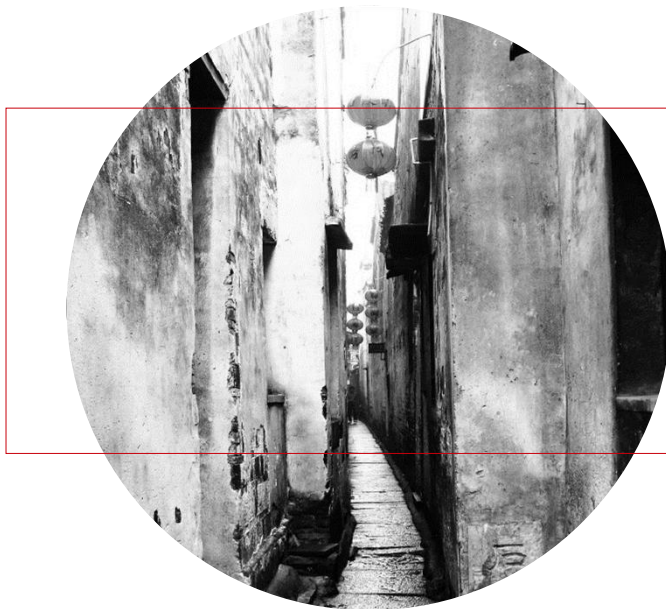


矩
阵
相
加

实验2.3内容算法提示

题目：用十字链表实现两个稀疏矩阵加法。

总体思路：用十字链表分别作为两个稀疏矩阵的存储结构A，B，将B中元素逐个相加到A的对应位置中，实现 $A=A+B$ 。



稀疏矩阵加法（十字链表）

实验2.3内容算法提示

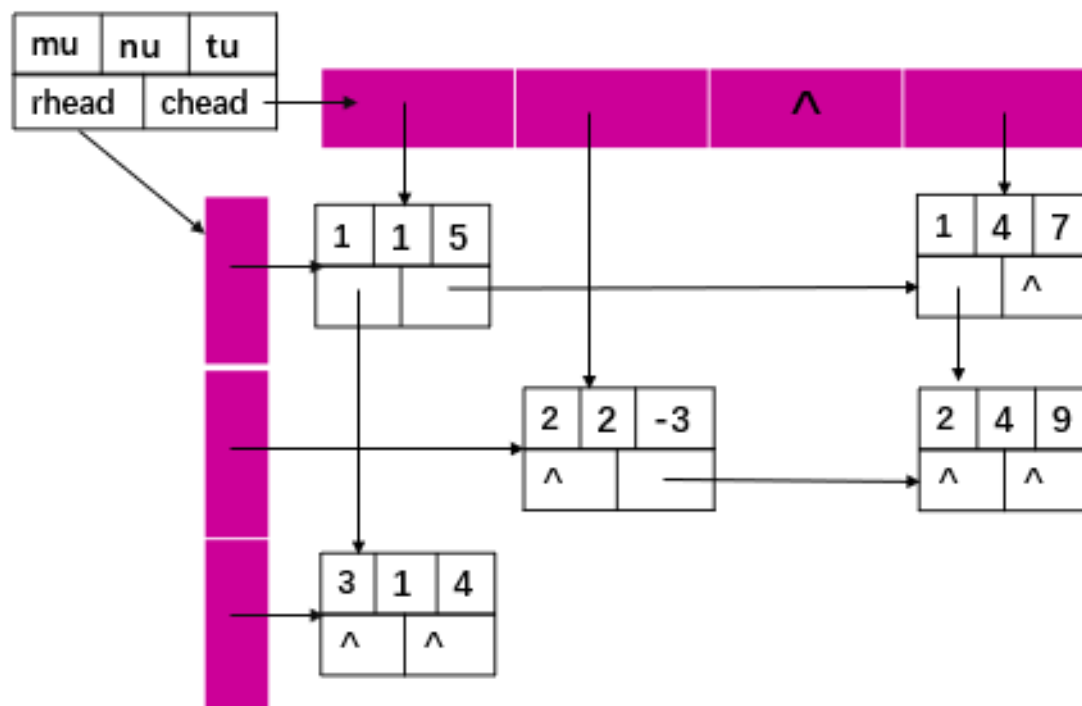
算法主要实现：

(1) 用十字链表存储稀疏矩阵A, B, 矩阵的每一行和每一个列都可以看作是一个单独的链表, 而之所以能够表示矩阵, 是因为行链表和列链表都分别存储在各自的数组中。

(2) 按照行列对两个矩阵对应节点相加, 若A (i, j)不存在元素, 而B (i, j) 存在元素, 则在表示矩阵A的十字链表中插入此节点, 否则, 对应元素相加 (若相加为0, 则删除此节点), 实现A=A+B。

(3) 按照行输出运算完成后矩阵A。

$$A = \begin{Bmatrix} 5 & 0 & 0 & 7 \\ 0 & -3 & 0 & 9 \\ 4 & 0 & 0 & 0 \end{Bmatrix}$$



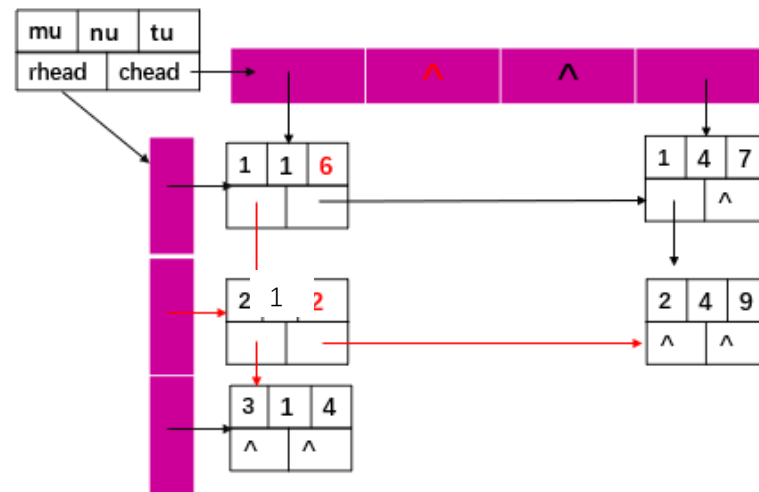
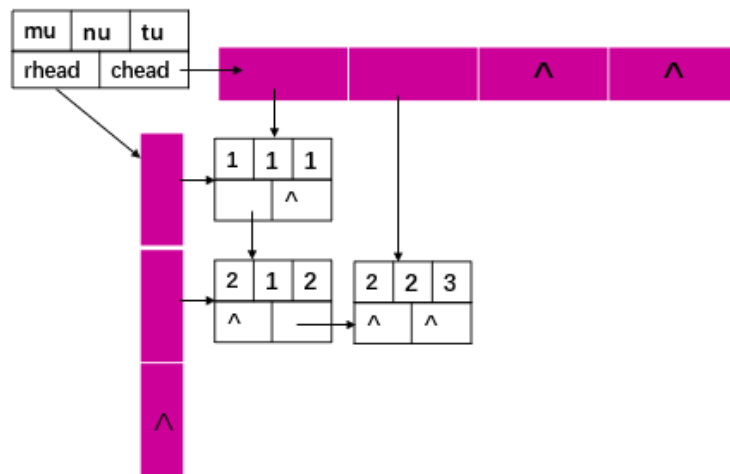
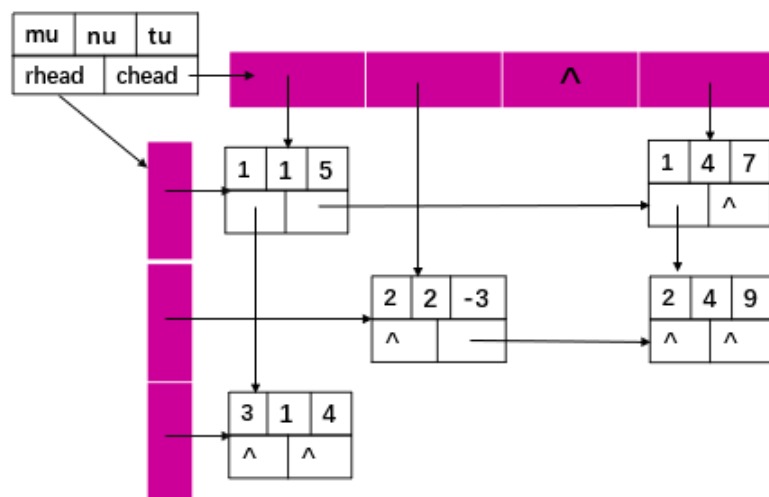
稀疏矩阵加法（十字链表）

实验2.3内容算法演示

$$A = \begin{Bmatrix} 5 & 0 & 0 & 7 \\ 0 & -3 & 0 & 9 \\ 4 & 0 & 0 & 0 \end{Bmatrix}$$

$$B = \begin{Bmatrix} 1 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{Bmatrix}$$

$$A = A + B = \begin{Bmatrix} 6 & 0 & 0 & 7 \\ 2 & 0 & 0 & 9 \\ 4 & 0 & 0 & 0 \end{Bmatrix}$$



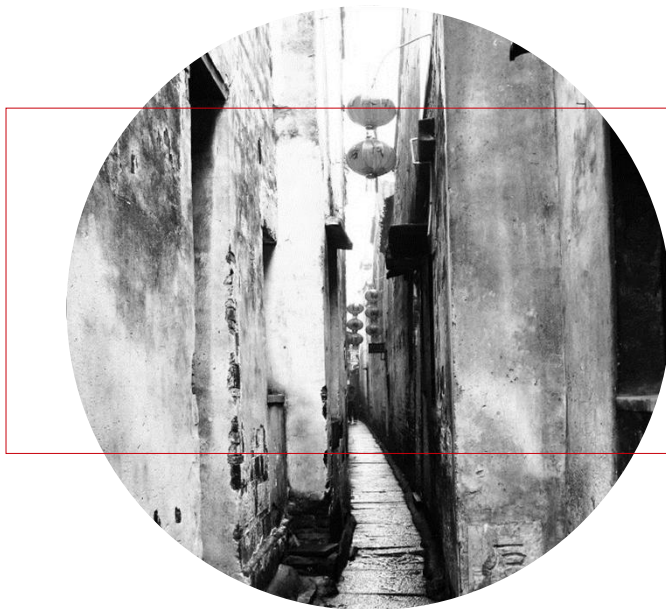
总结:

1. $A(i, j)$, $B(i, j)$ 都存在元素, 且相加不为0, 则只改变值域, 否则, 删除节点。
2. $A(i, j)$ 不存在元素, 而 $B(i, j)$ 存在元素, 则改变指针域。
3. $A(i, j)$ 存在元素, 而 $B(i, j)$ 不存在元素, 无需改变。

实验2.4内容算法提示

题目：稀疏矩阵的乘法

计算两个稀疏矩阵的乘法



稀疏矩阵的乘法

实验2.4内容算法提示

两个矩阵相乘也是矩阵的一种常用的运算。设矩阵M是 $m1 \times n1$ 矩阵，N是 $m2 \times n2$ 矩阵；若可以相乘，则必须满足矩阵M的列数 $n1$ 与矩阵N的行数 $m2$ 相等，才能得到结果矩阵 $Q=M \times N$ （一个 $m1 \times n2$ 的矩阵）。

数学中矩阵Q中的元素的计算方法如下：

$$Q[i][j] = \sum_{k=1}^{n1} M[i][k] \times N[k][j]$$

其中： $1 \leq i \leq m1, 1 \leq j \leq n2$ 。

实验2.4内容算法提示

根据数学上矩阵相乘的原理， 我们可以得到矩阵相乘的经典算法：

```
for(i=1; i<=m1; i++)
```

```
    for(j=1; j<=n2; j++)
```

```
        {Q [i] [j] =0;
```

```
            for(k=1; k<=n1; k++)
```

```
                Q [i] [j] =Q [i] [j] +M [i] [k] *N [k] [j] ;
```

```
        }
```

实验2.4内容算法提示

经典算法中，不论 $M[i][k]$ 、 $N[k][j]$ 是否为零，都要进行一次乘法运算，而实际上，这是没有必要的。采用三元组表的方法来实现时，因为三元组只对矩阵的非零元素做存储所以可以采用固定三元组表 a 中的元素 (i, k, M_{ik}) ($1 \leq i \leq m_1, 1 \leq k \leq n_1$)，在三元组表 b 中找所有行号为 k 的对应元素 (k, j, N_{kj}) ($1 \leq k \leq m_2, 1 \leq j \leq n_2$)进行相乘、累加，从而得到 $Q[i][j]$ ，即以三元组表 a 中的元素为基准，依次求出其与三元组表 b 的有效乘积。

实验2.4内容算法提示

已知稀疏矩阵A($m_1 \times n_1$)和B($m_2 \times n_2$), 求乘积C($m_1 \times n_2$)。稀疏矩阵A、B、C及它们对应的三元组表A.data、B.data、C.data示。

$$A = \begin{bmatrix} 3 & 0 & 0 & 7 \\ 0 & 0 & 0 & -1 \\ 0 & 2 & 0 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 4 & 1 \\ 0 & 0 \\ 1 & -1 \\ 0 & 2 \end{bmatrix}$$

$$C = \begin{bmatrix} 12 & 17 \\ 0 & -2 \\ 0 & 0 \end{bmatrix}$$

	i	j	v
1	1	1	3
2	1	4	7
3	2	4	-1
4	3	2	2

A.data

	i	j	v
1	1	1	4
2	1	2	1
3	3	1	1
4	3	2	-1
5	4	2	2

B.data

	i	j	v
1	1	1	12
2	1	2	17
3	2	2	-2

C.data

解题思路：在矩阵乘积运算中，只有A元素的列与B元素的行相等的两项（即 $a_j=b_i$ ）才有相乘的机会，因而在程序中：if $a_j=b_i$, 值相乘。

实验2.4内容算法提示

因为现在按三元组表存储，三元组表是按行为主序存储的，在B.data 中，同一行的非零元素其三元组是相邻存放的，同一列的非零元素其三元组并未相邻存放，因此在B.data 中反复搜索某一列的元素是很费时的，因此改变一下求值的顺序。

以求c11 和c12 为例，因为

c11=	c12=	解释
$a_{11} * b_{11} +$	$a_{11} * b_{12} +$	a11 只与 B 中 1 行元素相乘
$a_{12} * b_{21} +$	$a_{12} * b_{22} +$	a12 只与 B 中 2 行元素相乘
$a_{13} * b_{31} +$	$a_{13} * b_{32} +$	a13 只与 B 中 3 行元素相乘
$a_{14} * b_{41}$	$a_{14} * b_{42}$	a14 只与 B 中 4 行元素相乘

即a11 只可能和B 中第1 行的非零元素相乘，a12 只可能和B 中第2 行的非零元素相乘，...，而同一行的非零元是相邻存放的，所以求c11 和c12 同时进行：求 $a_{11} * b_{11}$ 累加到c11，求 $a_{11} * b_{12}$ 累加到c12，再求 $a_{12} * b_{21}$ 累加到c11，再求 $a_{12} * b_{22}$ 累加到c22., ...，当然只有aik 和 bkj(列号与行号相等)且均不为零（三元组存在）时才相乘，并且累加到cij 当中去。

为了运算方便，设一个累加器：datatype temp[n+1]; 用来存放当前行中cij 的值，当前行中所有元素全部算出之后，再存放到C.data 中去。

实验2.4内容算法提示

为了便于B.data 中寻找B 中的第k 行第一个非零元素，与前面类似，在此需引入num和rpot 两个向量。

num[k]表示矩阵B 中第k 行的非零元素的个数；rpot[k]表示第k 行的第一个非零元素在B.data 中的位置。

于是有

$$\text{rpot}[1]=1$$

$$\text{rpot}[k]=\text{rpot}[k-1]+\text{num}[k-1] \quad 2 \leq k \leq n$$

对于矩阵B 的num 和rpot 如图

col	1	2	3	4
num[col]	2	0	2	1
rpot[col]	1	3	3	5

根据以上分析，稀疏矩阵的乘法运算的粗略步骤如下：

(1)初始化。清理一些单元，准备按行顺序存放乘积矩阵；

(2)求B 的num, rpot;

(3)做矩阵乘法。将A.data 中三元组的列值与B.data 中三元组的行值相等的非零元素相乘，并将具有相同下标的乘积元素相加。

实验2.4内容算法提示

```
#define MAXSIZE 1000 /*非零元素的个数最多为1000*/
#define MAXROW 1000 /*矩阵最大行数为1000*/
typedef struct
{
    int row, col; /*该非零元素的行下标和列下标*/
    ElementType e; /*该非零元素的值*/
}Triple;
typedef struct
{
    Triple data [MAXSIZE+1]; /* 非零元素的三元组表, data [0] 未用*/
    int rpot [MAXROW+1]; /* 三元组表中各行第一个非零元素所在的位置 */
    int m, n, len; /*矩阵的行数、列数和非零元素的个数*/
}TriSparMatrix;
```

该算法的时间主要耗费在乘法运算及累加上，其时间复杂度为 $O(A.len \times B.n)$ 。当 $A.len$ 接近于 $A.m \times A.n$ 时，该算法时间复杂度接近于经典算法的时间复杂度 $O(A.m \times A.n \times B.n)$ 。



再
会

