第四章 数组和字符串

4.1 数组

4.2 矩阵

4.3 字符串

4.1 数 组

一、数组的存储和寻址

- 一维数组是若干个元素的有限序列。元素本身就是一个数据结构。
- 一维数组的元素必须具有相同的类型，每个数组元素都占据相同大小的存储空间。
- 一维数组采用顺序存储，数组是线性表的推广。

- 每个元素都通过一个下标来指定，故一个一维数组对应一个下标函数。
- 一维数组A[*n*]，每个数组元素占一个存储单元（不妨设为C个连续字节）. 数组元素A[0]的首地址Loc(A[0])，则对于0 ≤ *i* ≤*n*-1，有：

$$Loc(A[i])=Loc(A[0])+i\times C$$

- 可以将多维数组转化为一维数组计算元素的地址。
- 多维数组有两种存放次序：按行优先顺序和按列优先顺序存储。

- 按行优先顺序，就是将数组元素按行向量的顺序存储，第*i*+1个行向量存储在第*i*个行向量之后。BASIC、PASCAL、C/C++等程序设计语言中，数组按行优先顺序存放；
- 按列优先顺序，就是将数组元素按列向量的顺序存储，第*i*+1个列向量存储在第*i*个列向量之后。FORTRAN语言、Matlab中，数组按列优先顺序存放。

二维数组

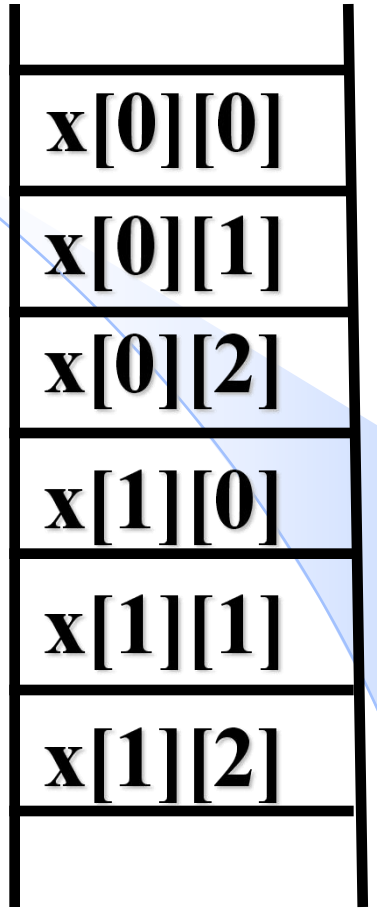
[例] int x[2][3]　　/*有2×3个数组元素*/

x[0][0]	x[0][1]	x[0][2]
x[1][0]	x[1][1]	x[1][2]

按行优先顺序存放

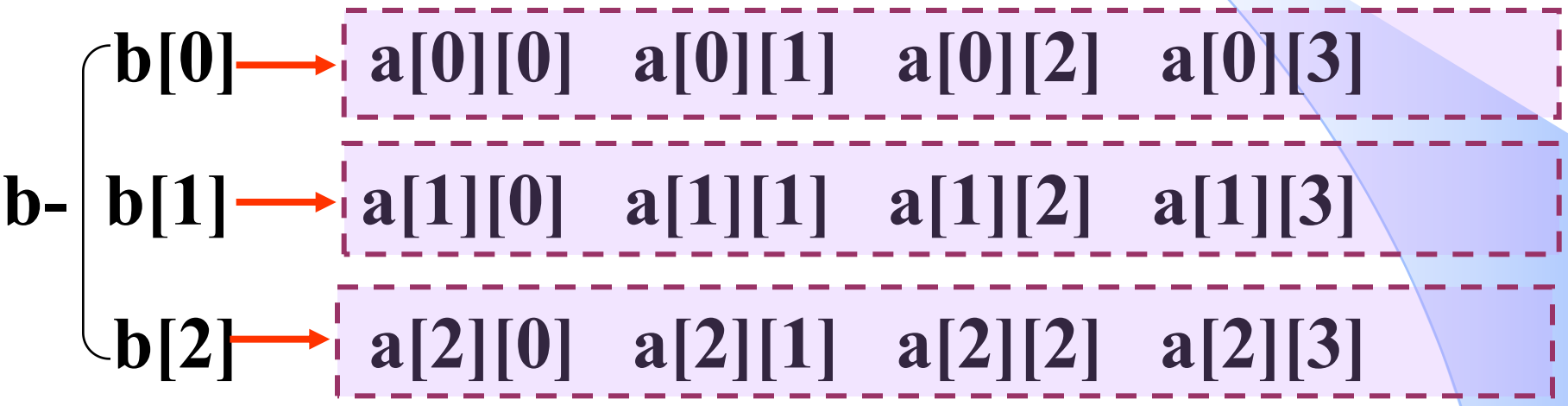
存储分配顺序为：

x[0][0]->x[0][1]->
x[0][2]->x[1][0]->
x[1][1]->x[1][2]

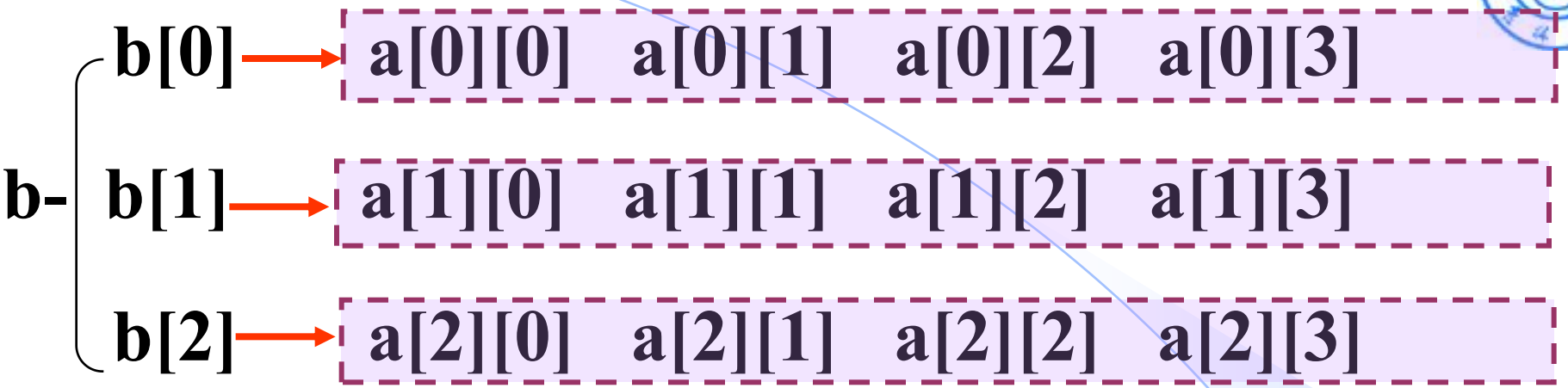


二维数组可以看作是一种特殊的一维数组。

[例] float a[3][4];



float a[3][4];



二维数组($m\times n$)中元素a[i][j]的地址：

$Loc(a[i][j]) = Loc(b[i]) + j\times C$

$Loc(b[i]) = Loc(b[0]) + i\times C'$ // $C' = n\times C$

$Loc(a[i][j]) = Loc(a[0][0]) + i\times n\times C + j\times C$

$= Loc(a[0][0]) + (i\times n + j)\times C$

[例] **float a[3][4]**

Loc(a[1][2])

= Loc(a[0][0]) +(*i*×*n*+*j*)×C

= Loc(a[0][0]) +(1×4+2)×4

= Loc(a[0][0]) +24

三维数组（按行优先顺序）

多维数组元素在内存中的排列顺序为：第一维的下标变化最慢，最右边维的下标变化最快。

[例]float a[2][3][4]

a[0][0][0]→a[0][0][1]→a[0][0][2]→a[0][0][3]→
a[0][1][0]→a[0][1][1]→a[0][1][2]→a[0][1][3]→
a[0][2][0]→a[0][2][1]→a[0][2][2]→a[0][2][3]→

a[1][0][0]→a[1][0][1]→a[1][0][2]→a[1][0][3]
a[1][1][0]→a[1][1][1]→a[1][1][2]→a[1][1][3]
a[1][2][0]→a[1][2][1]→a[1][2][2]→a[1][2][3]

三维数组A[*m*][*n*][*p*]中数组元素a[*i*][*j*][*k*]的地址计算公式为：

$$Loc(a[i][j][k])$$

$$=Loc(a[0][0][0])+i\times n\times p\times C+j\times p\times C+k\times C$$

[例] float D[3][3][4]

Loc(D[1][2][2])

$= d + i \times n \times p \times C + j \times p \times C + k \times C$

$= d + (1 \times 3 \times 4 + 2 \times 4 + 2) \times 4$

$= d + 88$

a[i_1][i_2]…[i_n]的存储地址（行优先）

$$Loc(i_1,i_2,\cdots,i_n)=Loc(0,i_2,\cdots,i_n)+i_1\times C_1$$

$$=Loc(0,0,i_3,\cdots,i_n)+i_2\times C_2+i_1\times C_1$$

$$\cdots$$

$$=Loc(0,\cdots,0)+i_n\times C+i_{n-1}\times C_{n-1}+\cdots+i_1\times C_1$$

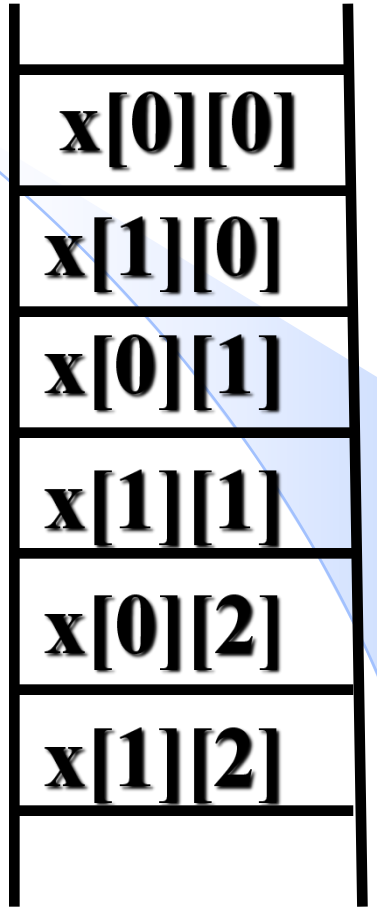
$$=Loc(0,\cdots,0)+i_n\times C+i_{n-1}\times C\times m_n+\cdots+i_1\times C\times m_n\times\cdots\times m_2$$

$$=Loc(0,\cdots,0)+\{\sum_{k=1}^{n-1}(i_k\times\prod_{p=k+1}^{n}m_p)+i_n\}\times C$$

- 如何计算按列优先顺序存储的数组元素地址？
- 将数组元素按列向量的顺序存储，第i+1个列向量存储在第i个列向量之后。

二维数组x[2][3]的按列优先存储

x[0][0]	x[0][1]	x[0][2]
x[1][0]	x[1][1]	x[1][2]



a[i_1][i_2]…[i_n]的存储地址（列优先）

$$Loc(0,\cdots,0)+\{\sum_{k=2}^{n}(\ i_k * \prod_{p=1}^{k-1} m_p)+i_1\}*C$$

对于四维数组A[3][5][11][3]，

分别给出按行优先、列优先存储下的

A[I][J][K][L]地址计算公式。

Loc(A)+(165I+33J+3K+L)×C 行优先

Loc(A)+ (165L+15K+3J+I)×C 列优先

二、数组操作

数组类型是高级程序设计语言所提供的基本数据类型之一，可定义一组相同类型的元素。但是直接创建的数组存在着一些问题，诸如：

- 无法对数组执行一些简单运算，如数组加法和数组减法等操作；
- 没有越界索引保护，不检查数组的下标索引值是否在0到arraysize-1范围内。例如一些高级程序设计语言对越界索引访问并不一定会产生异常。没有越界索引保护会直接给程序调试带来难以预料的困难。

4.2 矩 阵

矩阵是许多物理问题中出现的数学对象，是一种常用的数据组织方式。计算机工作者关心的是矩阵在计算机中如何存储，以及如何实现矩阵的基本操作。

二维数组与矩阵相比：

- 数组的基本操作是数组加减，而矩阵的基本操作还有矩阵相乘、矩阵转置等；
- 数组的下标从0开始，而矩阵的下标一般从1开始；
- 数组元素用a[*i*][*j*]表示，而矩阵元素通常用a(*i*, *j*)表示。

一、矩阵的乘法运算

对于矩阵$A_{m\times p}$和$B_{p\times n}$的乘积$C_{m\times n}$，其第i行第j列元素c_{ij}的计算公式为 $\sum_{k=1\cdots p}\left(a_{ik}\times b_{kj}\right)$，矩阵乘法运算的算法可描述如下：

算法Multi-1(A, B, m, p, n . C)

FOR i←1 TO m DO

FOR j←1 TO n DO

(C_{ij} ←0 .

FOR k←1 TO p DO

C_{ij} ← C_{ij} + A_{ik}×B_{kj} .) ▌

如何实现一维数组存放的矩阵的乘法运算。

考虑按行优先，用一维数组s存放$A_{m\times p}$，一维数组t存放$B_{p\times n}$，将A与B的乘积$C_{m\times n}$存放在一维数组w中。

cs为A中i行k列元素的下标，那么A中i行k+1列元素的下标是什么？ cs←cs+1.

ct为B中k行j列元素的下标，那么B中k+1行j列元素的下标是什么？ ct←ct+n.

算法Multi-2(s, t, *m*, *p*, *n*. w)

M1. [初始化]

cw←0 . // 初始时cw是c_{11}在一维数组w中的下标

M2. [循环]

FOR *i*←1 TO *m* DO // 第一层循环

FOR *j*←1 TO *n* DO // 第二层循环

(cs←(*i*–1)×*p* . // 确定A矩阵第*i*行的第一个元素

ct←*j*–1. // 确定B矩阵第*j*列的第一个元素

w[cw]←0 . //计算C_{ij}

FOR *k*=1 TO *p* DO // 第三层循环

// 累加$a_{ik}\times b_{kj}$ 并存于w[cw]中

(w[cw]←w[cw]+s[cs]×t[ct].

cs←cs+1. // cs为A中本行下一列元素在s中的下标

ct←ct+*n*.) // ct为B中本列下一行元素在t中的下标

cw←cw+1.) ▌

分析：

矩阵乘法算法中包含了三层**for**循环，所以其时间复杂性为**O** (***m*×*n*×*p***) .

二、特殊矩阵

前面所介绍的矩阵，是以按行优先次序将所有矩阵元素存放在一个一维数组中。但是对于特殊矩阵，如**对称**矩阵、**三角**矩阵、**对角**矩阵和**稀疏**矩阵等, 如果用一维数组来实现，那么大量的存储空间中存放的是重复信息或者是零元素，这将造成很大的空间浪费。为节省存储空间，提高算法运行效率，通常会采用压缩存储的方法。

二、特殊矩阵

1、对角矩阵的压缩存储

若$n\times n$的方阵M是对角矩阵，则对所有的$i\neq j$ $(1\leq i, j \leq n)$都有M$(i, j)=0$，即非主对角线上的元素均为0 .

对于一个$n\times n$的对角矩阵，至多只有n个非零元素，因此只需存储其n个对角元素的信息。

采用一维数组d[n]来压缩存储对角矩阵，其中d[i-1]存储M(i, i)的值。

2、三角矩阵的压缩存储

- 三角矩阵分为上三角矩阵和下三角矩阵。
- 方阵M是上三角矩阵，当且仅当$i>j$时有$M(i,j)=0$. 方阵M是下三角矩阵，当且仅当$i<j$时有$M(i,j)=0$.

$$\begin{pmatrix} 50 & 15 & 35 & 25 \\ 0 & 10 & 20 & 60 \\ 0 & 0 & 30 & 10 \\ 0 & 0 & 0 & 40 \end{pmatrix} \qquad \begin{pmatrix} 50 & 0 & 0 & 0 \\ 15 & 10 & 0 & 0 \\ 35 & 20 & 30 & 0 \\ 25 & 60 & 10 & 40 \end{pmatrix}$$

以 **下三角矩阵M**为例，讨论其压缩存储方法。

考虑一个$n\times n$维下三角矩阵，其第一行有1个非零元素，第二行有2个非零元素，…，第n行有n个非零元素，非零元素共有$(1+2+\ldots+n)=n(n+1)/2$个。可以用这样大小的一维数组d来存储下三角矩阵M的非零元素。

映射次序可采用按行优先或按列优先。假设采取按行优先，非零元素M(*i*,*j*)会映射到一维数组d中的哪个元素？

- 设元素M(*i*, *j*)前面有*k*个元素，可以计算出

  $k = 1+2+\ldots+(i-1)+(j-1) = i(i-1)/2+(j-1)$

- 设一维数组d的下标是从0开始，则M(*i*, *j*)映射到d中所对应的元素是d[*k*]．有了*k*的计算公式，可以很容易实现下三角矩阵的压缩存储。

3. 对称矩阵M的压缩存储

- 将对称矩阵映射为一个一维数组d
- d需要多少个元素？ $n(n+1)/2$
- 按行优先方式， M(*i*, *j*)的寻址方式是什么？

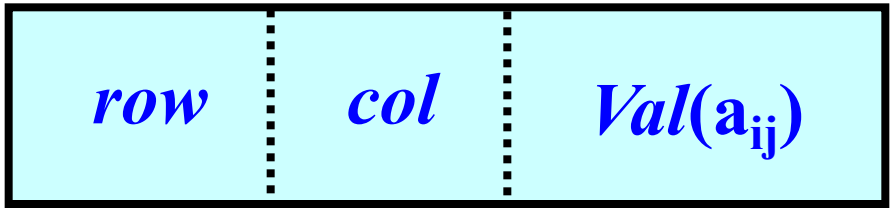
  - $i \geq j$，映射到d[k]，$k = i(i-1)/2 + (j-1)$
  - $i < j$，映射到d[q]，$q = j(j-1)/2 + (i-1)$

4、稀疏矩阵的压缩存储

定义：设矩阵 $A_{m\times n}$ 中非零元素的个数远远小于零元素的个数，则称 A 为稀疏矩阵。

- 特点：非零元素的分布一般没有规律，无法简单地利用一维数组和映射公式来实现其压缩存储。
- 作用：仅存储非零元素，节省空间。

对于矩阵 $A_{m\times n}$的每个元素a_{ij}，知道其行号i和列号j，就可以确定该元素在矩阵中的位置。因此，如果用一个结点来存储一个非零元素，那么结点可以设计为如下形式：



- 由三个域（行号、列号和元素值）构成的结点被称为三元组结点：矩阵的每个非零元素可由一个三元组结点（i, j, a_{ij}）唯一确定。
- 如何在三元组结点的基础上实现对整个稀疏矩阵的存储？
  - 顺序存储方式实现的三元组表；
  - 链接存储方式实现的十字链表。

三元组表

将表示稀疏矩阵的非零元素的三元组结点按行优先的顺序排列，得到一个线性表，将此线性表用顺序存储结构进行存储，称之为三元组表。

[例] 稀疏矩阵

$$A=\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$

三元组表

B[0]	1	1	50
B[1]	2	1	10
B[2]	2	3	20
B[3]	4	1	-30
B[4]	4	3	-60
B[5]	4	4	5

[例] 稀疏矩阵 $A=\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$

转置矩阵 $A'=\begin{pmatrix} 50 & 10 & 0 & -30 \\ 0 & 0 & 0 & 0 \\ 0 & 20 & 0 & -60 \\ 0 & 0 & 0 & 5 \end{pmatrix}$

[例] 稀疏矩阵的三元组表示

$$A = \begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$

$$A` = \begin{pmatrix} 50 & 10 & 0 & -30 \\ 0 & 0 & 0 & 0 \\ 0 & 20 & 0 & -60 \\ 0 & 0 & 0 & 5 \end{pmatrix}$$

a[0]	1	1	50
a[1]	2	1	10
a[2]	2	3	20
a[3]	4	1	-30
a[4]	4	3	-60
a[5]	4	4	5

b[0]	1	1	50
b[1]	1	2	10
b[2]	1	4	-30
b[3]	3	2	20
b[4]	3	4	-60
b[5]	4	4	5

算法Transpose(a, *count*. b) /* *m*×*n*矩阵*A*存放在三元组表a中，求*A*的转置矩阵并将其保存在三元组表b中*/

T1. [初始化]

j ← 0. // 首先赋值三元组表b的第一个结点b[0]

T2. [逐行转置]

IF *count* ≤ 0 THEN RETURN; // a为空

FOR *k*=1 TO *n* DO // 对矩阵b按行优先依次确认非零元素

FOR *i*=0 TO *count*–1 DO // 对每个*k*扫描所有非零元

IF (col(a[*i*])=*k* THEN // 处理列号为*k*的非零元素

(row(b[*j*])←*k* . // 行号应为*k*

col(b[*j*])←row(a[*i*]). // 列号应为其在a中的行号

value(b[*j*])←value(a[*i*]).

j←*j*+1.) ▌ // 赋值三元组表b的下一个结点

- 对于一个用三元组表存储的稀疏矩阵 $A_{m\times n}$，若矩阵非零元素个数为t，求 $A_{m\times n}$ 的转置矩阵的时间复杂性是多少呢？
- 观察**Transpose**算法不难发现，其中包含双重循环，第一重循环是对转置矩阵 $A'_{n\times m}$ 按行优先依次确认非零元素，故循环次数为A'的行数n；第二重循环是扫描原矩阵的三元组表，执行次数是矩阵非零元素个数t，显然，求转置矩阵的时间复杂性为$O(nt)$.
- 就计算时间而言，使用三元组表示比二维数组表示更差，为了节省空间而付出了过多时间。

稀疏矩阵的三元组表存储方式分析

相应的算法描述较为简单，但对于非零元的位置或个数经常发生变化的矩阵运算就显得不太适合。

[例] 执行将矩阵 B 相加到矩阵 A 上的运算时，某位置上的结果可能会由非零元变为零元，但也可能由零元变为非零元，这就会引起在A的三元组表中进行删除和插入操作，从而导致大量结点的移动。

矩阵相加

a[0]	1	2	6
a[1]	2	1	4
a[2]	3	2	9
a[3]	3	4	7
a[4]	4	4	8

b[0]	1	1	5
b[1]	2	2	1
b[2]	3	2	3
b[3]	3	4	-7
b[4]	4	3	4

a[0]	1	1	5
a[1]	1	2	6
a[2]	2	1	4
a[3]	2	2	1
a[4]	3	2	12
a[5]	4	3	4
a[6]	4	4	8

对此类运算采用链式存储结构为宜。

例如：每行对应一个非零元单链表

思考：稀疏矩阵的乘法

十字链表

矩阵的元素结构如下：分别表示该元素的左邻非零元素、上邻非零元素、所在的行、所在的列和数据值。

LEFT		UP
ROW	COL	VAL

例：稀疏矩阵

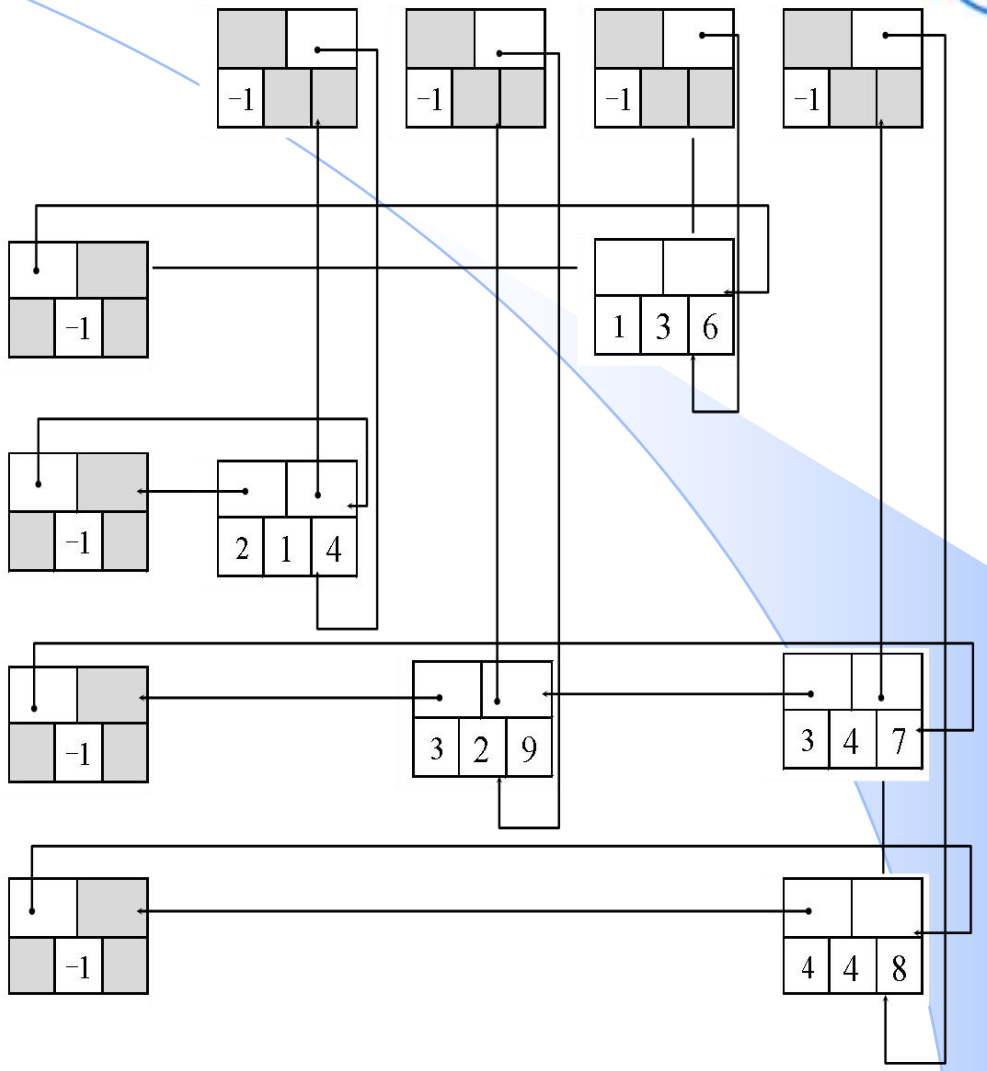
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 6 & 0 \\ 4 & 0 & 0 & 0 \\ 0 & 9 & 0 & 7 \\ 0 & 0 & 0 & 8 \end{pmatrix} \end{array}$$

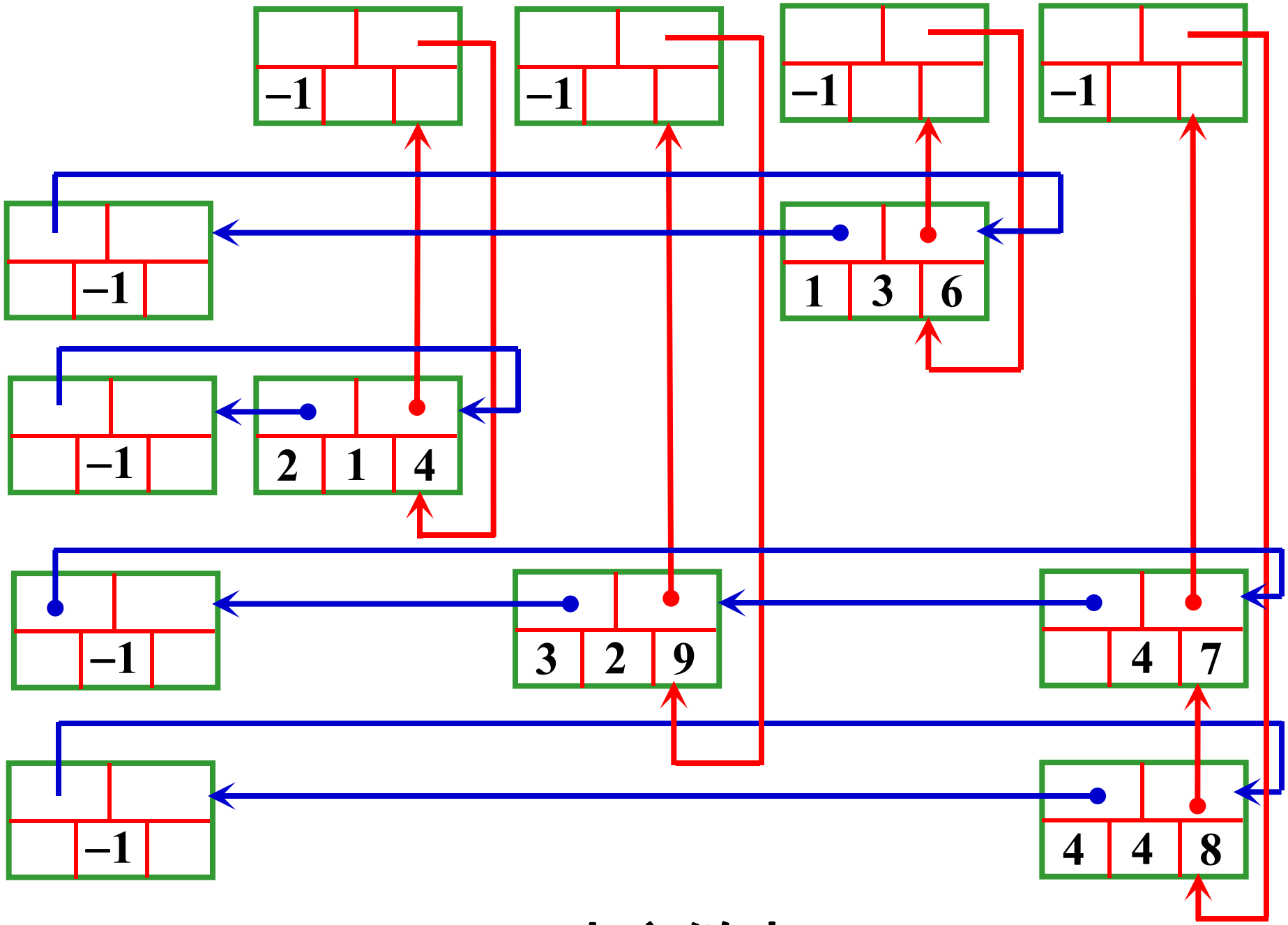
矩阵的每一行、每一列都设置为由一个表头结点引导的循环链表，并且各行和各列的表头结点有如下特点：

-1 = COL(Loc(BASEROW[i]))< 0

-1 = ROW(Loc(BASECOL[j]))< 0

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 6 & 0 \\ 4 & 0 & 0 & 0 \\ 0 & 9 & 0 & 7 \\ 0 & 0 & 0 & 8 \end{pmatrix} \end{array}$$





十字链表

若某一行没有非零元素，则

LEFT(Loc(BASEROW[*i*]))= Loc(BASEROW[*i*])

若某一列没有非零元素，则

UP(Loc(BASECOL[*i*]))=Loc(BASECOL[*i*])

- 对两个稀疏矩阵A和B，如何实现运算A=A+B呢？假设A和B都采用十字链表作存储结构，现要求将B中结点合并到A中，合并后的结果有三种可能：

  1）不变：a_{ij}（$b_{ij}=0$）

  2）插入：b_{ij}（$a_{ij}=0$）

  3）更新：$a_{ij}+b_{ij}\neq 0$

  4）删除：$a_{ij}+b_{ij}=0$

- 由此可知当将 B 加到 A 中去时，对 A 矩阵的十字链表来说，或者不变（$b_{ij}=0$），或者插入一个新结点（$a_{ij}=0, b_{ij}\neq 0$），或者是更新结点的VAL域值（$a_{ij}+b_{ij}\neq 0$），还可能是删除一个结点（$a_{ij}+b_{ij}=0$）。

- 整个运算过程可以从矩阵的第一行起逐行进行。
- 对每一行都从行表头出发分别找到A和B在该行中的最右边的非零元结点后开始比较其列值，然后按下述四种不同情况分别处理之。
- 若*pa*和*pb*分别指向A和B的十字链表中行值相同的两个结点，则四种情况描述为：

1）col(*pa*) = col (*pb*) 且val(*pa*)+val(*pb*)≠0，则只要将$a_{ij}+b_{ij}$的值赋给*pa*所指结点的值域即可，其他域的值都不变化。

2）col(*pa*)=col(*pb*)且val(*pa*)+val(*pb*)=0，则需要在A矩阵的链表中删除*pa*所指的结点。这时，需改变同一行中前驱结点的left域值，以及同一列中前驱结点的up域值。

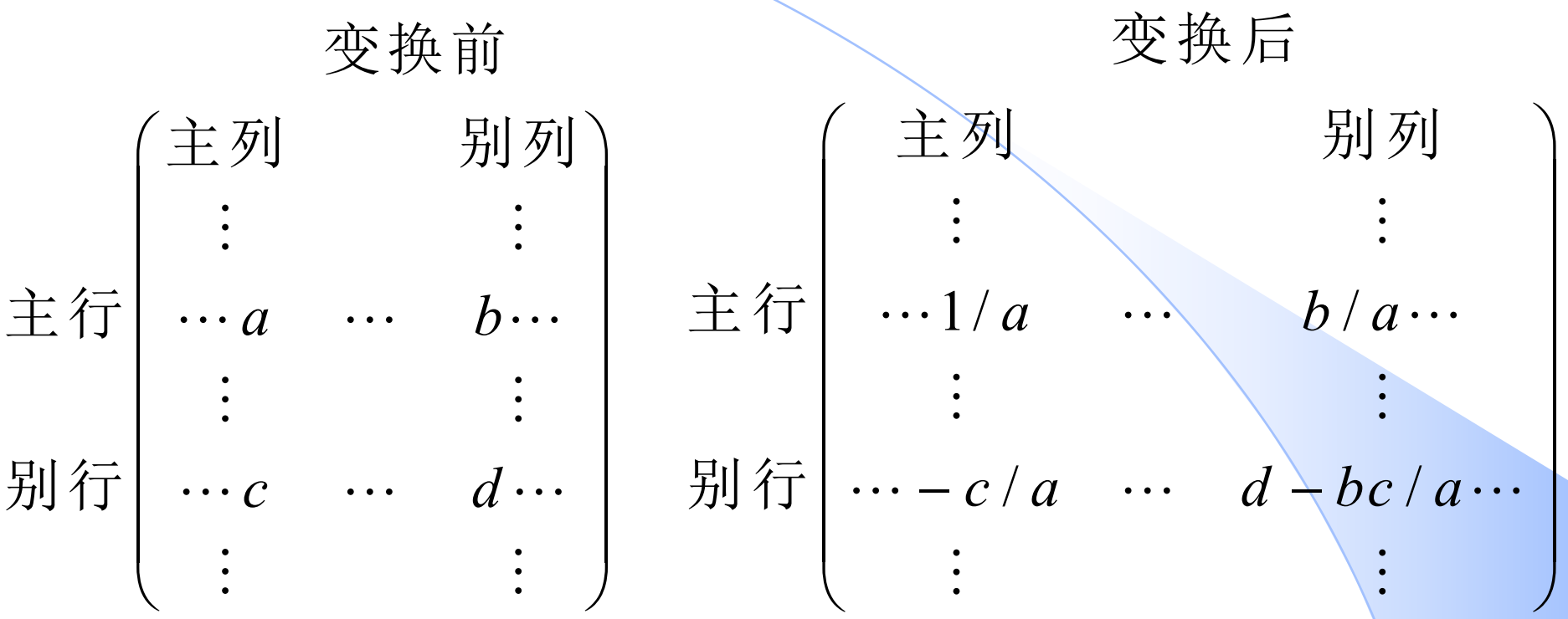
3）col(*pa*) >col(*pb*)，则只要将pa指针往左推进一步，并重新加以比较即可。

4）col(*pa*)<col(*pb*)，则需在A矩阵的链表中插入*pb*所指结点。

对于矩阵的加减运算，只需有行或列链表就可以。

对矩阵的运算实质上就是在十字链表中插入结点、删除结点以及改变某个结点的 VAL 域的值。

矩阵的主步骤操作：



选定非零元素 a为主元素，其所在行为主行，所在列为主列。

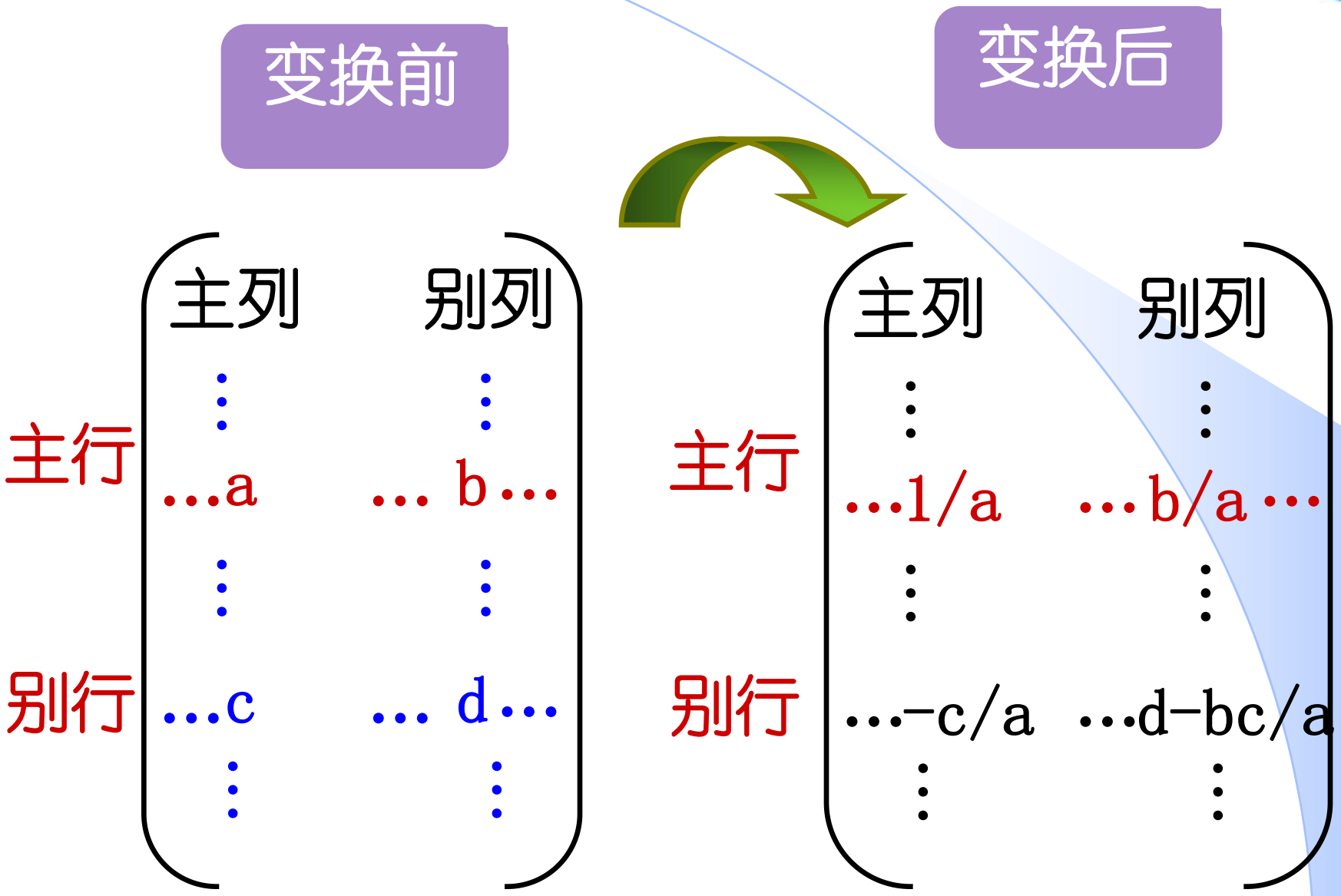
变换前，若某主行别列元素为0或某别行主列元素为0，则变换后它们仍然为零。

对别行别列元素A(i, j)，仅当其对应的主行j列和主列i行元素皆非零，它才需要被处理。

下面以矩阵A为例，选A(2, 1)=10为主元素，执行主步骤操作。

$$A=\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix}$$

“主步骤”操作：要求主行主列元素非零。



$$\left(\begin{array}{cccc} -\frac{50}{10} & 0-\frac{0\times 50}{10} & 0-\frac{20\times 50}{10} & 0-\frac{0\times 50}{10} \\ \frac{1}{10} & \frac{0}{10} & \frac{20}{10} & \frac{0}{10} \\ -\frac{0}{10} & 0-\frac{0\times 0}{10} & 0-\frac{20\times 0}{10} & 0-\frac{0\times 0}{10} \\ \frac{30}{10} & 0+\frac{0\times 30}{10} & -60+\frac{20\times 30}{10} & 5+\frac{0\times 30}{10} \end{array}\right|$$

$$A=\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix} \quad A^*=\begin{pmatrix} -5 & 0 & -100 & 0 \\ 0.1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 5 \end{pmatrix}$$

经主步骤操作后矩阵A变成A*，与A相比A*的第1行第3列的元素变成非零元，第4行第3列的非零元素变为零元；A*还是稀疏矩阵。

算法 SP（BASE，PIVOT. BASE）
// 稀疏矩阵的主步骤操作，稀疏矩阵的表示方式为正交链表，指针变量PIVOT
// 指向主元素，一维数组PTR[1:n]是指针型
SP1 [初始化，确定主行I_0，主列J_0]
I_0←ROW(PIVOT) . J_0←COL(PIVOT) .
α←1.0 / VAL(PIVOT) . VAL(PIVOT) ←1.0 .
P_0←Loc(BASEROW[I_0]) . Q_0←Loc (BASECOL[J_0]) .
SP2 [处理主行I_0]
P_0←LEFT(P_0) . J←COL(P_0) .
IF J<0 THEN GOTO SP3.
ELSE (PTR[J] ←Loc(BASECOL[J]) .
VAL(P_0) ←α* VAL(P_0) .
GOTO SP2.) .
SP3 [找新行I，并指定P_1]
Q_0←UP(Q_0) . I←ROW (Q_0) .
IF I<0 THEN RETURN .
IF I=I_0 THEN GOTO SP3 .
P←Loc(BASEROW[I]) . P_1←LEFT(P) .
SP4 [确定新列J]
P_0←LEFT(P_0) . J←COL(P_0) .
IF J<0 THEN (VAL(Q_0)← -α* VAL(Q_0) .
GOTO SP3.) .
IF J=J_0 THEN GOTO SP4 .

SP5 [P1所指元素所在的列与J列比较]

WHILE COL(P_1) > J DO

($P \leftarrow P_1$. $P_1 \leftarrow$LEFT(P)) .

IF COL(P_1) =J THEN GOTO SP7 .

SP6 [插入新元素]

WHILE ROW(UP(PTR[J])) > I DO

PTR[J]←UP(PTR[J]) .

X<=AVAIL . VAL(X) ⇐–VAL(Q_0)×VAL(P_0) .

ROW(X) ←I . COL(X) ←J .

LEFT(X) ← P_1 . UP(X) ←UP(PTR[J]) .

LEFT(P) ←X . P ←X. UP(PTR[J]) ←X . PTR[J] ←X .

GOTO SP4.

SP7 [主步骤操作]

VAL(P_1) ←VAL(P_1) - VAL(Q_0)* VAL(P_0) .

IF VAL(P_1)=0 THEN GOTO SP8 .

ELSE (PTR[J]←P_1 . P←P_1 .

P_1←LEFT(P) . GOTO SP4 .) .

SP8 [删除零元素]

WHILE UP(PTR[J]) ≠ P_1 DO

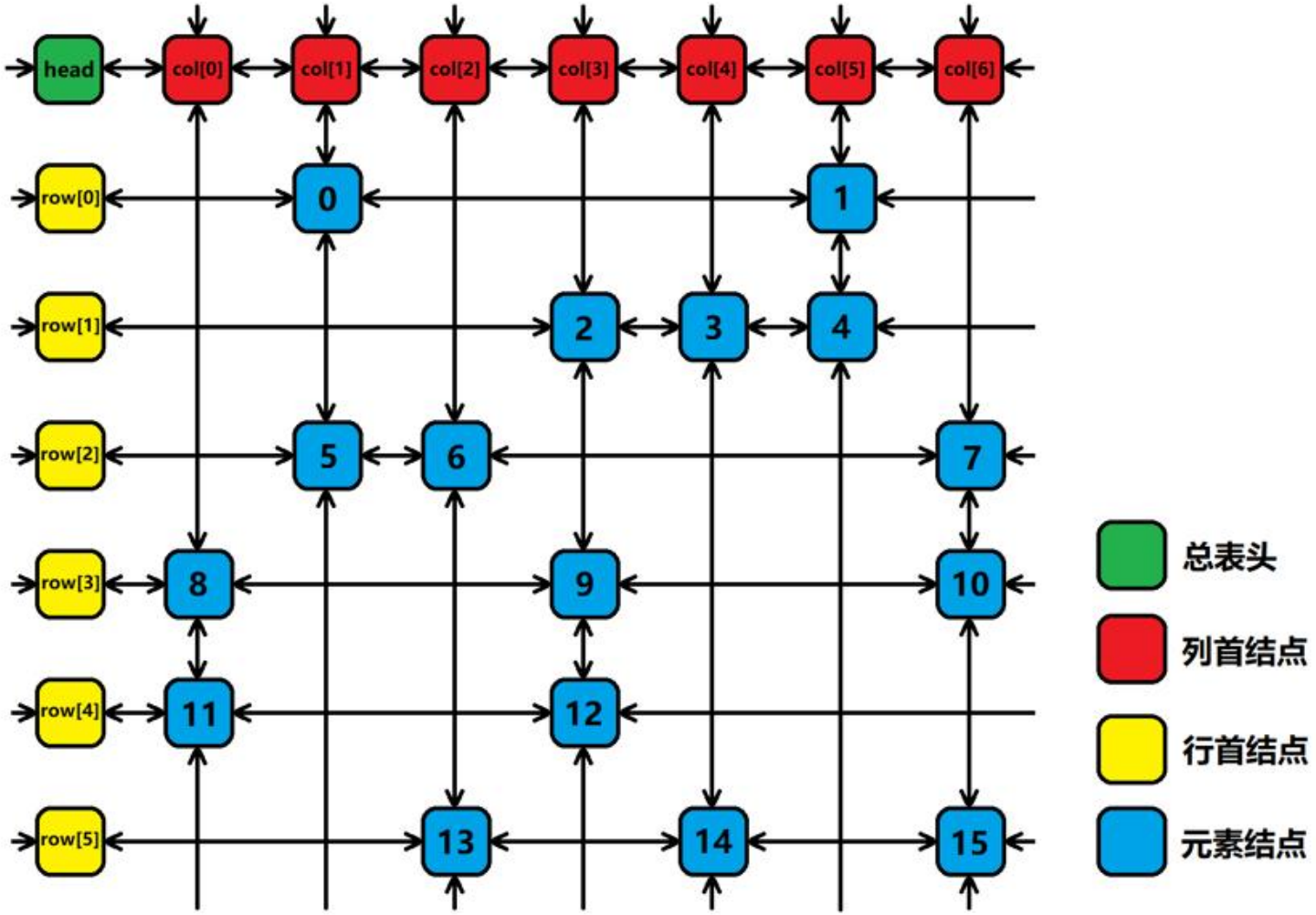
PTR[J]←UP(PTR[J]) .

UP(PTR[J])←UP(P_1) . LEFT(P)←LEFT(P_1) .

AVAIL <= P_1 . P_1 ←LEFT(P) .

GOTO SP4. ▌

双向十字链表（舞蹈链，Dancing Links）



双向十字链表（舞蹈链，Dancing Links）

- 舞蹈链：带哨兵的双向循环链表，即双向十字链表。采用双向十字链表来存储稀疏矩阵，达到优化搜索的目的。在搜索问题中，所需存储的矩阵往往随着递归的加深会变得越来越稀疏，这种情况用Dancing Links 来存储矩阵，往往可以取得非常好的效果。

- L[R[x]] ← L[x]，R[L[x]] ← R[x]；移除结点
- L[R[x]] ← x，R[L[x]] ← x；恢复结点

精确覆盖问题

- 集合X的若干子集构成集合S，精确覆盖是指S的子集S*，满足X中的每一个元素在S*中恰好出现一次。S*中子集的并集为X，两两交集为空。

精确覆盖问题的矩阵表示法

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

精确覆盖问题的矩阵表示法

采用矩阵表示法，求一个精确覆盖转化为求矩阵的若干个行的集合，使每列有且仅有一个1。

S*={B, D, F}

	1	2	3	4	5	6	7
A	1	0	0	1	0	0	1
B	1	0	0	1	0	0	0
C	0	0	0	1	1	0	1
D	0	0	1	0	1	1	0
E	0	1	1	0	0	1	1
F	0	1	0	0	0	0	1

跳舞链算法（DLX，Dancing Links X）

算法思想：回溯法

1、从矩阵中选择一行，标示矩阵中其他行的元素；

2、删除相关行和列的元素，得到新矩阵；

3、如果新矩阵是空矩阵，并且之前的一行都是1，那么求解结束，跳转到步5；新矩阵不是空矩阵，继续求解，跳转到步1；新矩阵是空矩阵，之前的一行中有0，跳转到步4；

4、说明之前的选择有误，回溯到之前的一个矩阵，跳转到步1；如果没有矩阵可以回溯，说明该问题无解，跳转到步6；

5、求解结束，输出结果；

6、求解结束，无解。

跳舞链算法（DLX，Dancing Links X）

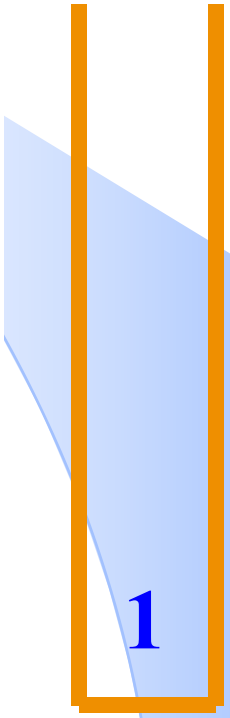
例子：

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

跳舞链算法（DLX，Dancing Links X）

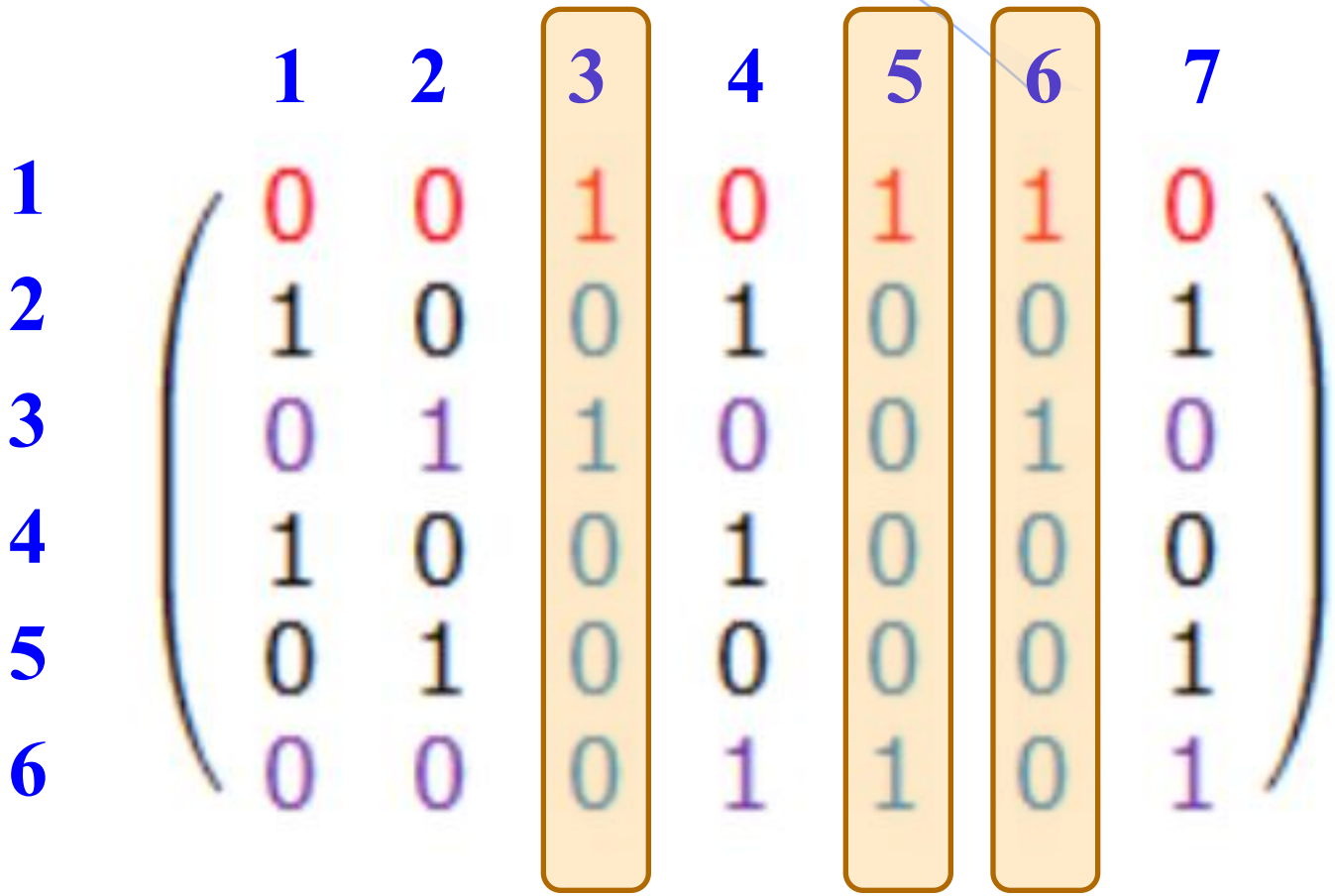
选择第1行

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



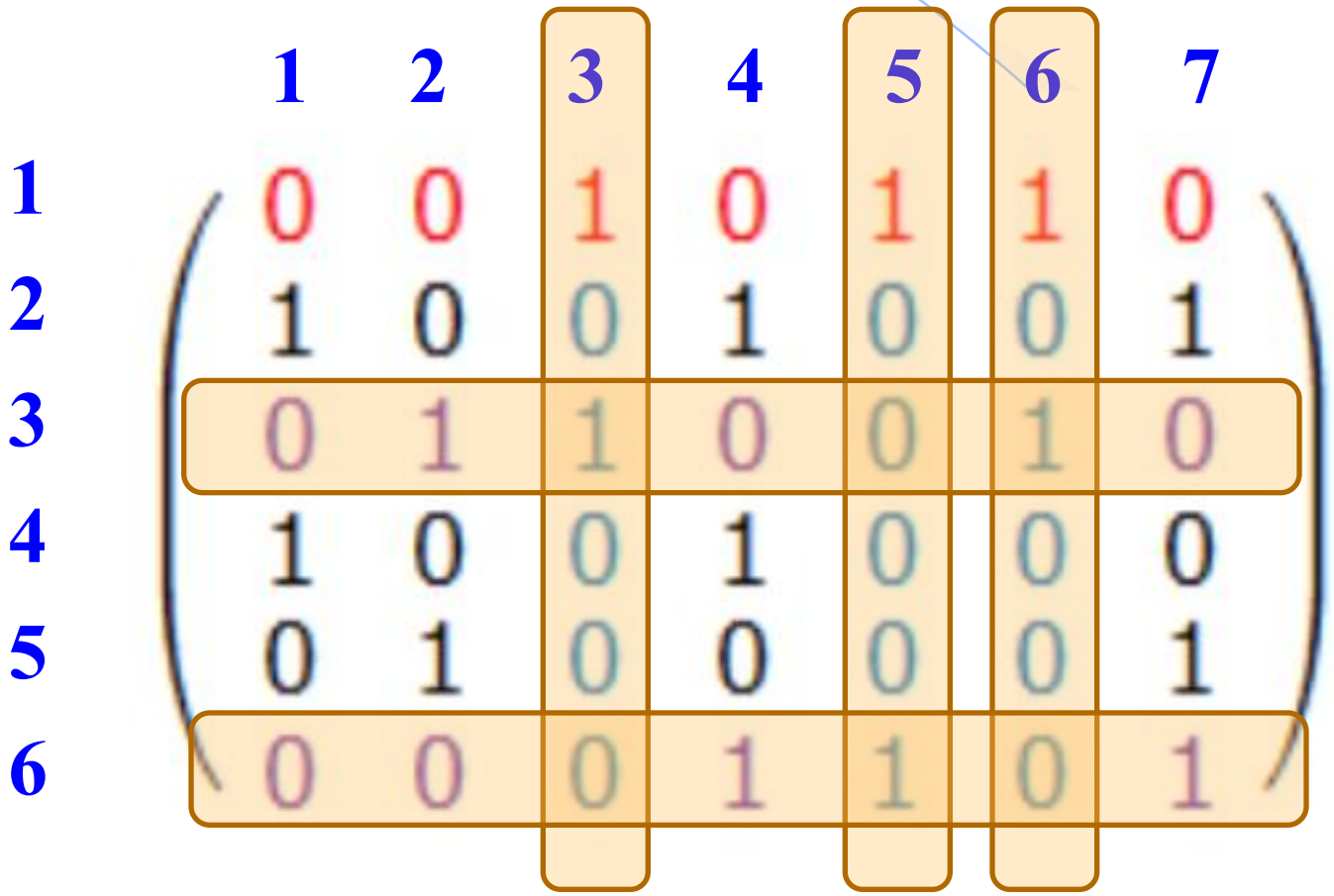
跳舞链算法（DLX，Dancing Links X）

标记包含“1”的列（已经覆盖的元素）



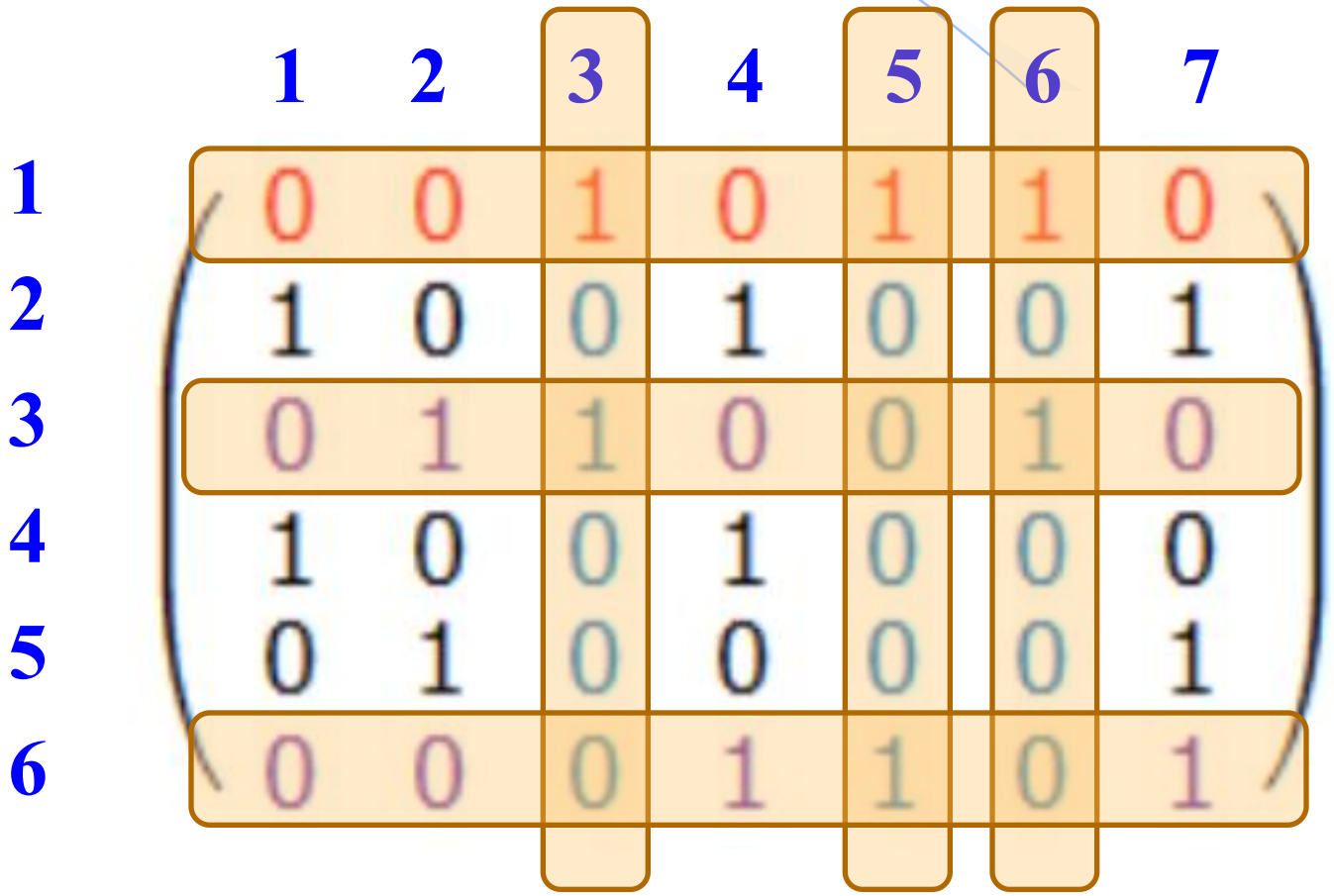
跳舞链算法（DLX，Dancing Links X）

标记冲突的行（以免重复覆盖）



跳舞链算法（DLX，Dancing Links X）

标记选择过的行（选择过的子集不再考虑）



跳舞链算法（DLX，Dancing Links X）

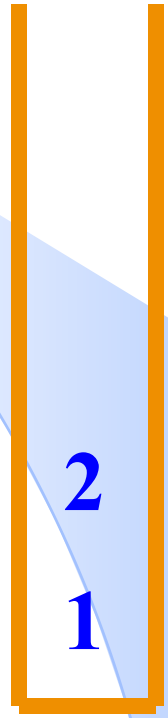
删除以上标记的元素

$$\begin{array}{c} \\ 2 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 4 & 7 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$

跳舞链算法（DLX，Dancing Links X）

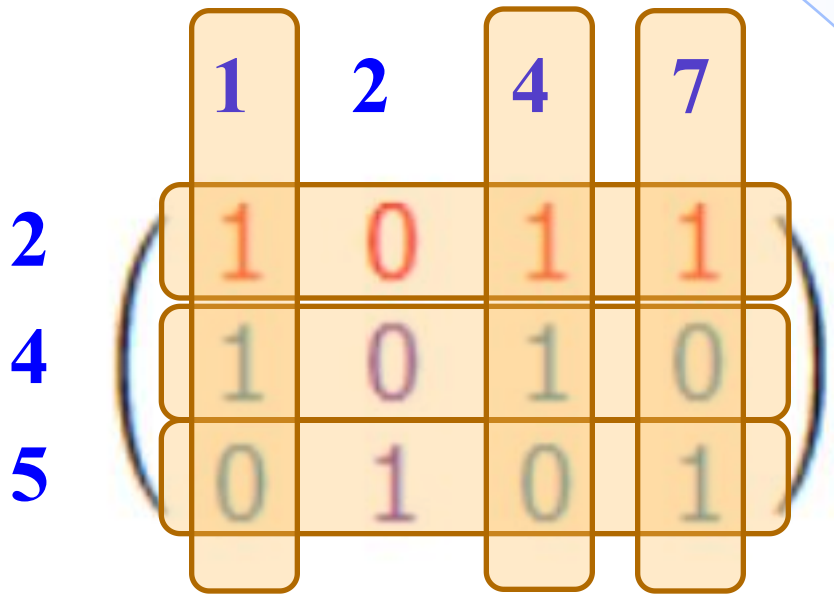
选择第2行

$$\begin{array}{c} \\ 2 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 4 & 7 \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \end{array}$$



跳舞链算法（DLX，Dancing Links X）

标记并删除相关的行和列

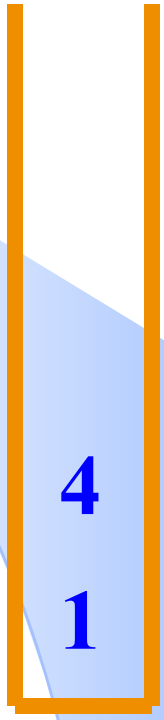


导致空矩阵产生，而红色的一行中有0（有0就说明这一列没有1覆盖）。说明选择第2行是错误的。

跳舞链算法（DLX，Dancing Links X）

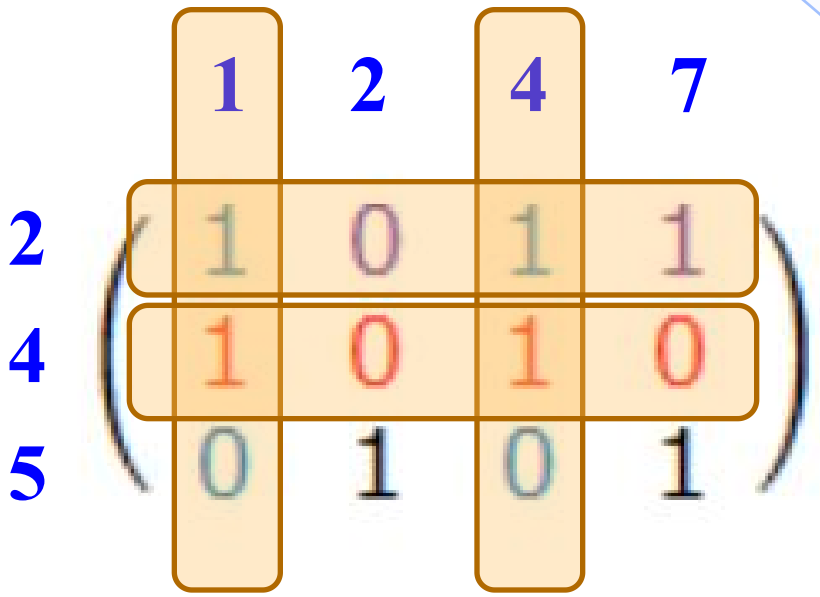
回溯，重新选择第4行

	1	2	4	7
2	1	0	1	1
4	1	0	1	0
5	0	1	0	1



跳舞链算法（DLX，Dancing Links X）

标记待删除的行和列



跳舞链算法（DLX，Dancing Links X）

删除后的矩阵

2　　7

5　　(1　　1)

5

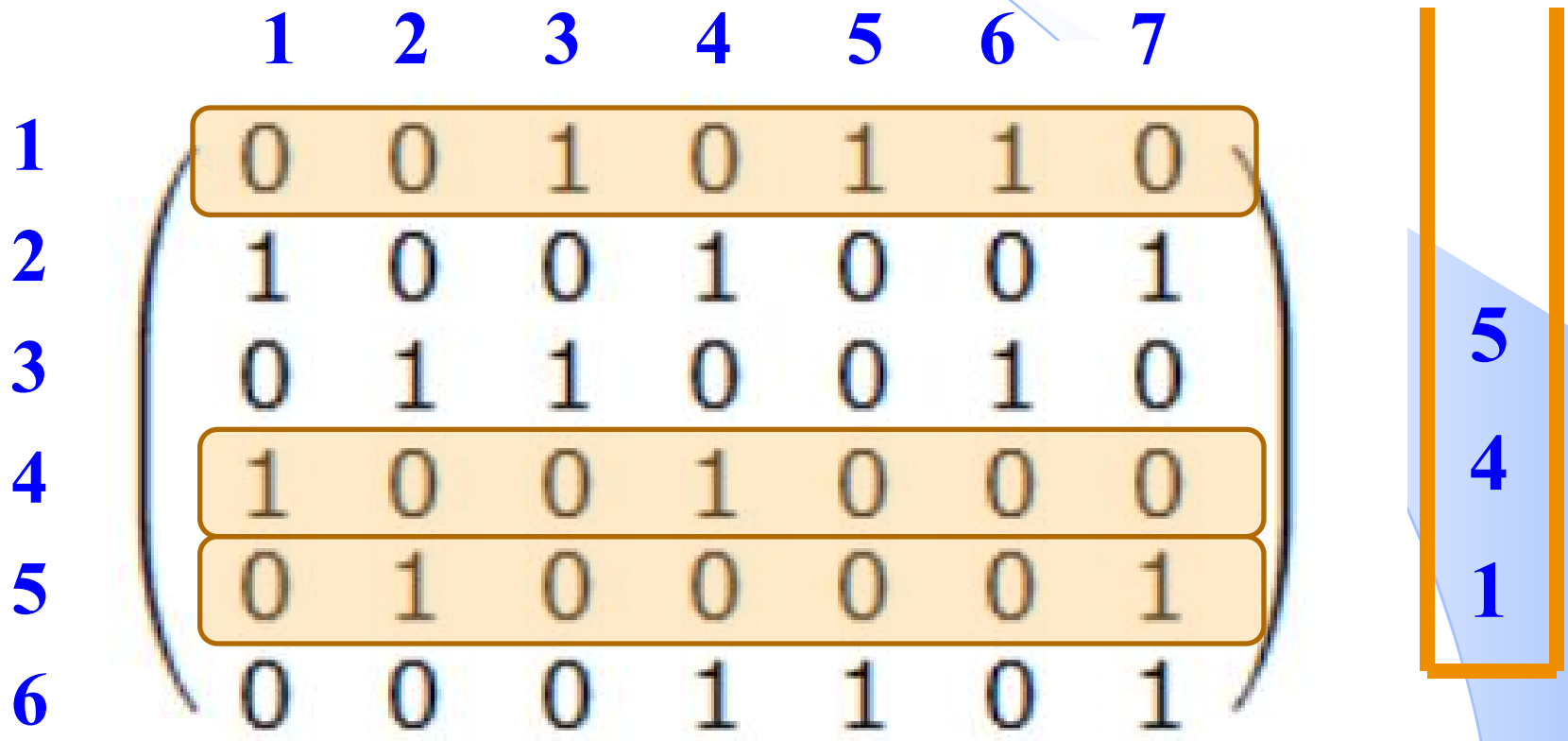
4

1

剩余矩阵只有1行，因此只能选择第5行，删除后得到空矩阵，但之前所有元素都为1，说明全部元素被覆盖，问题得到解决。

跳舞链算法（DLX，Dancing Links X）

最终选择的行



跳舞链算法（DLX，Dancing Links X）

求解的过程中有大量的探索（产生新的更稀疏的矩阵）和回溯（恢复旧矩阵）步骤。如何缓存新矩阵和如何恢复旧矩阵是一个重要问题。这可以采用双向十字链表作为数据结构加以解决。

跳舞链算法（DLX，Dancing Links X）

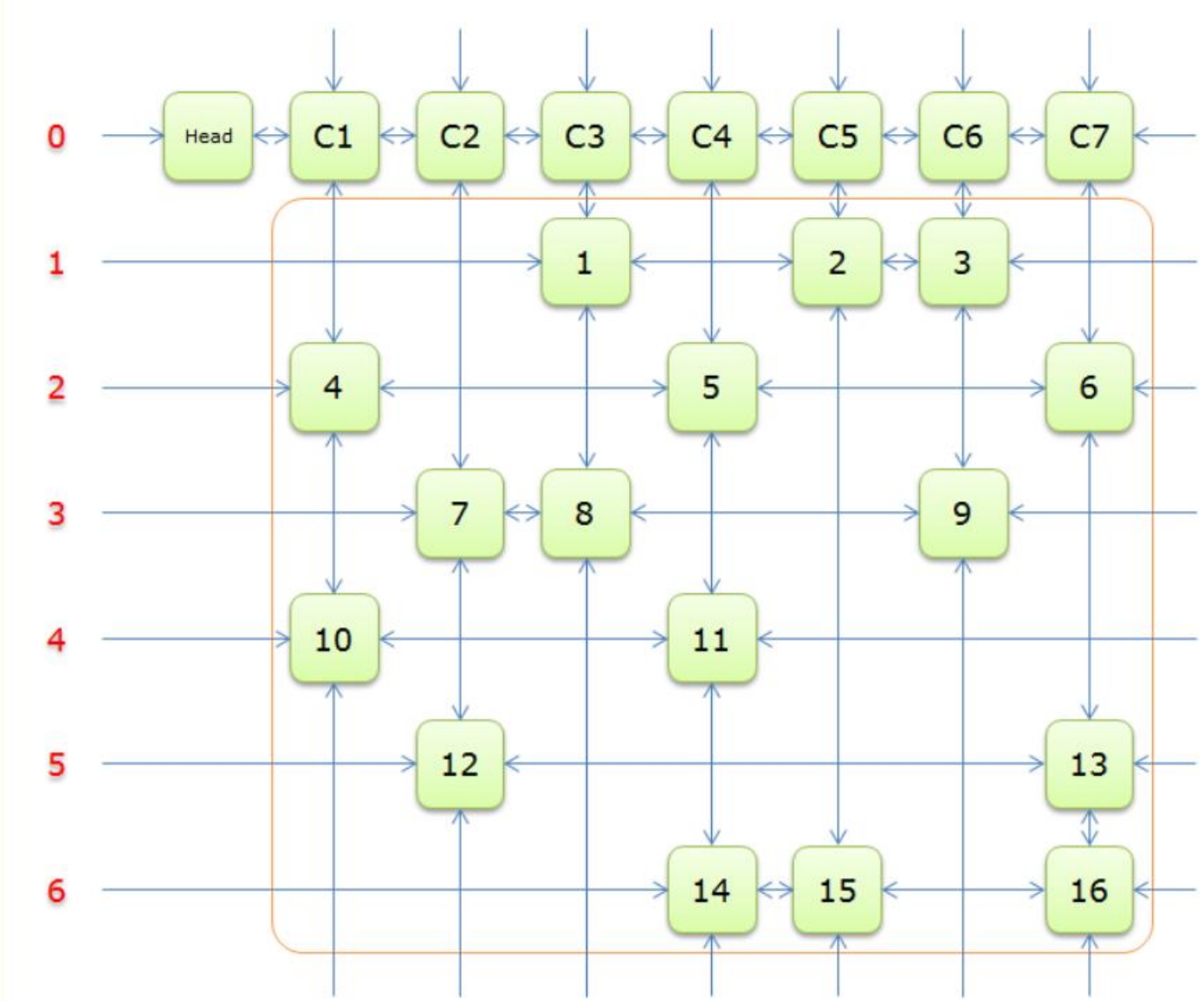
- Knuth提出的舞蹈链（Dancing Links）实际上并不是一种算法，而是一种数据结构。一种非常巧妙的数据结构，他的数据结构在缓存和回溯的过程中效率惊人，不需要额外的空间，以及近乎线性的时间。而在整个求解过程中，指针在数据之间跳跃着，就像精巧设计的舞蹈一样，故Knuth把它称为Dancing Links（舞蹈链）。
- 移除操作对应着缓存数据、恢复加入操作对应着回溯数据。
- 应用：数独问题、地砖铺放问题、*n*皇后问题、俄罗斯方块覆盖问题等。

跳舞链算法（DLX，Dancing Links X）

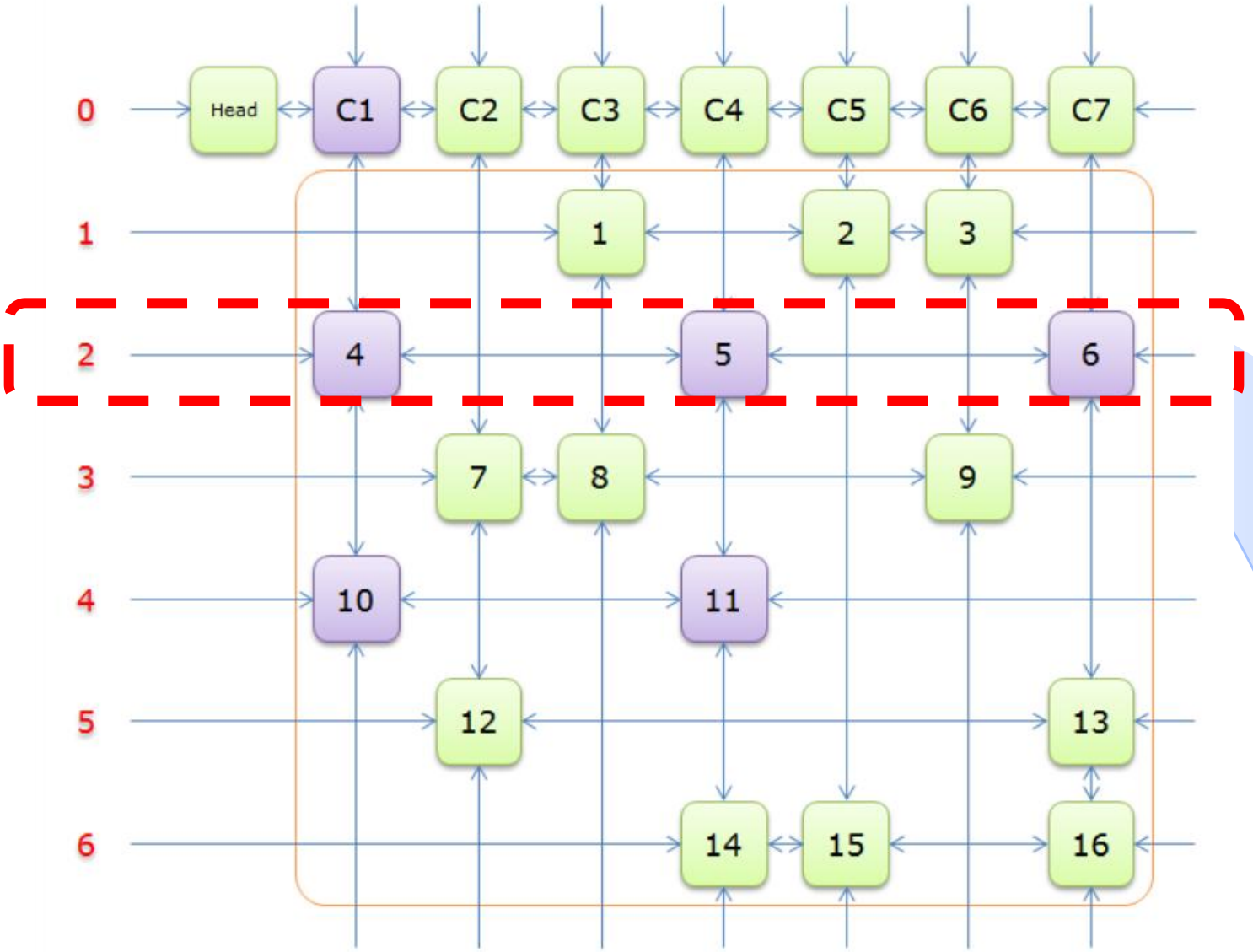
例子：

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

跳舞链（双向十字链表）

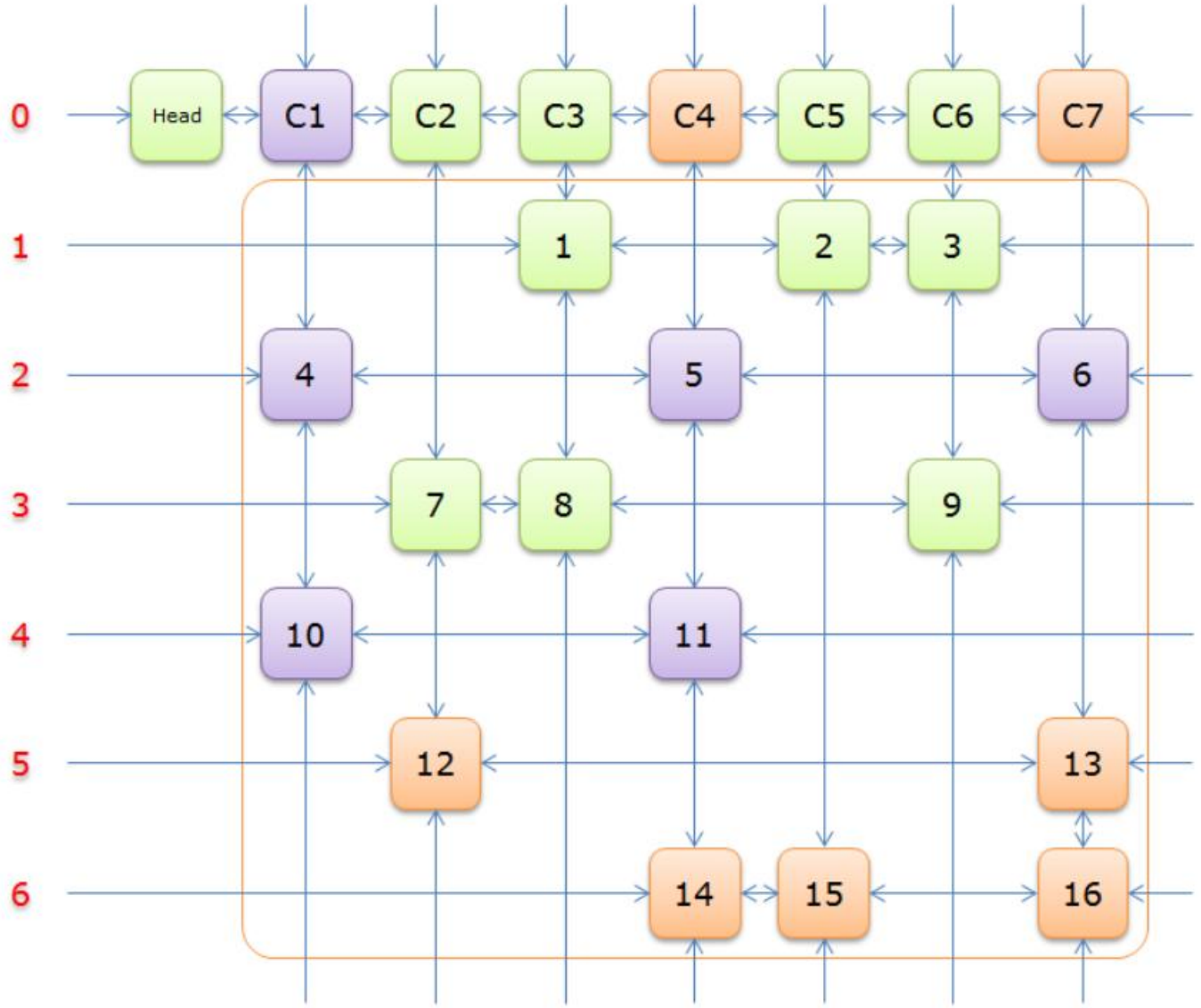


选择行（选择第2行）

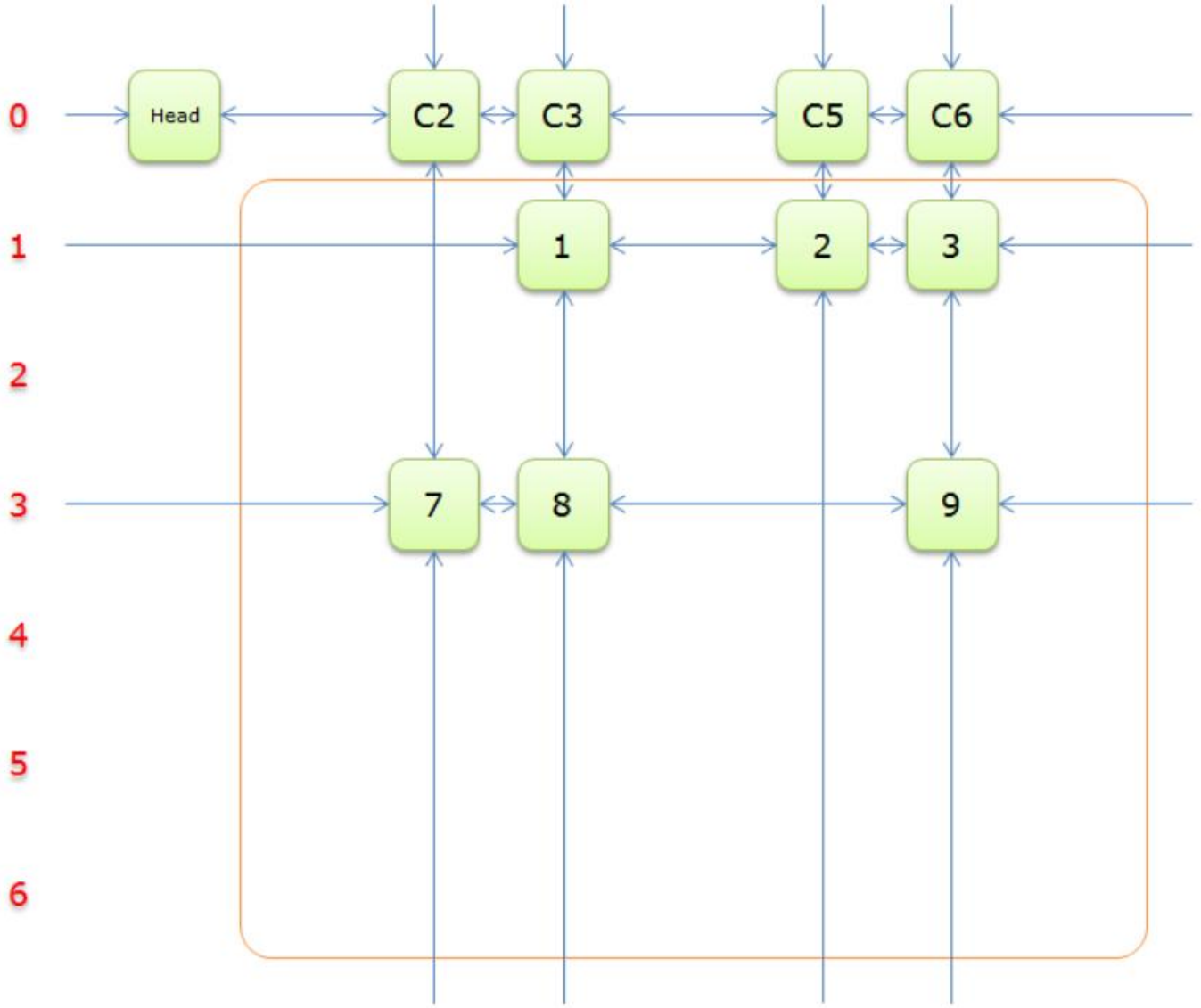


2

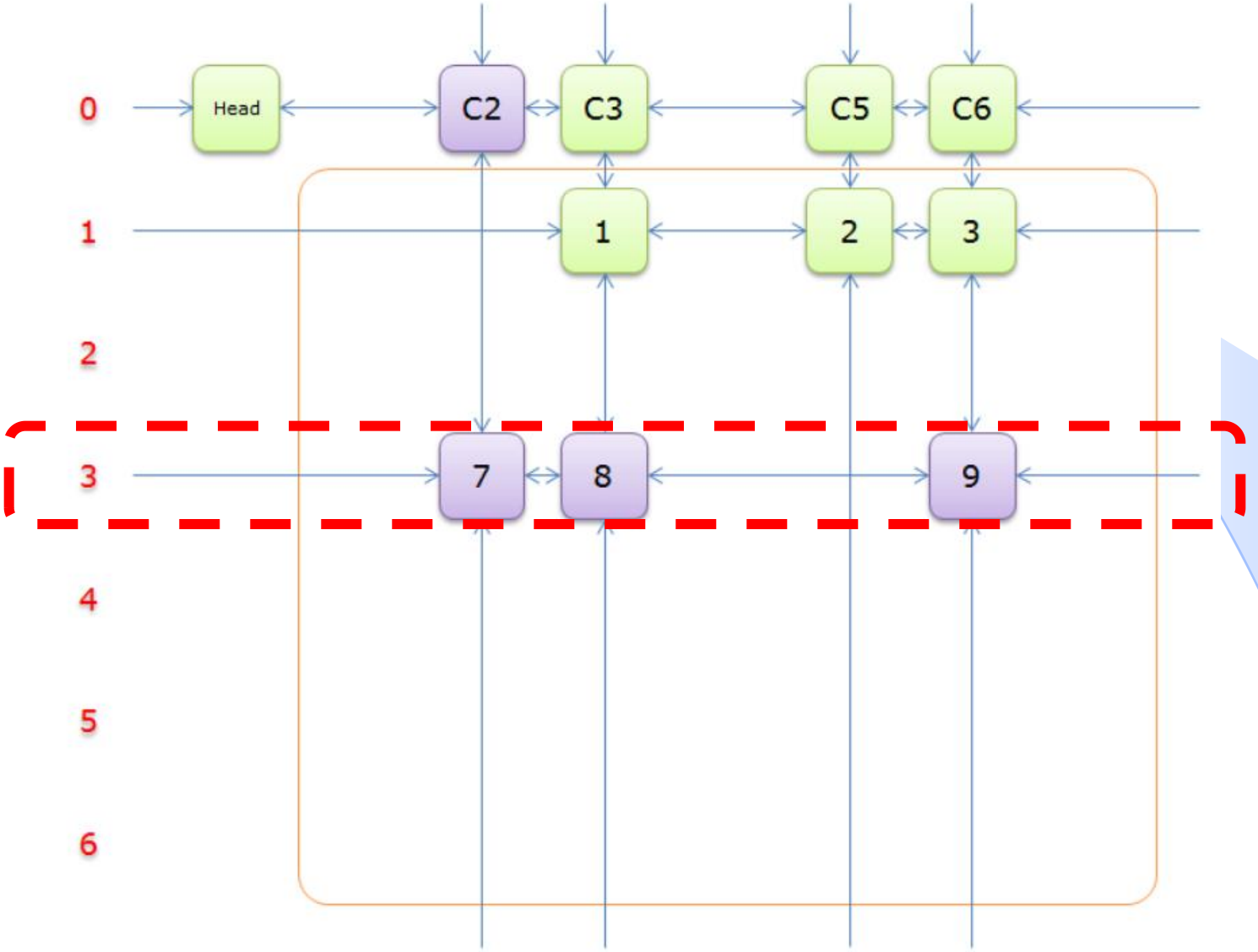
标记待删除的行和列



删除后的矩阵



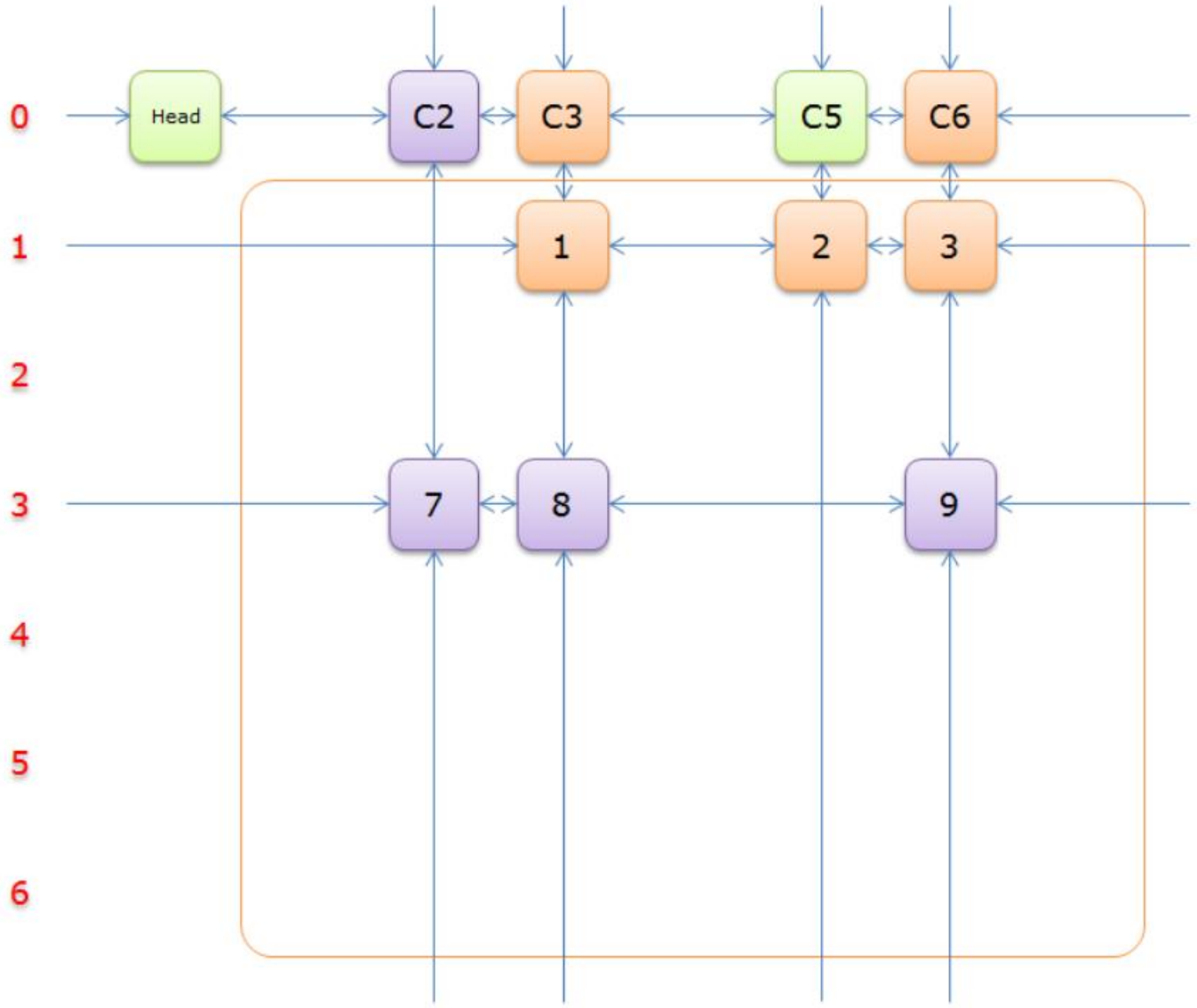
选择行（选择第3行）



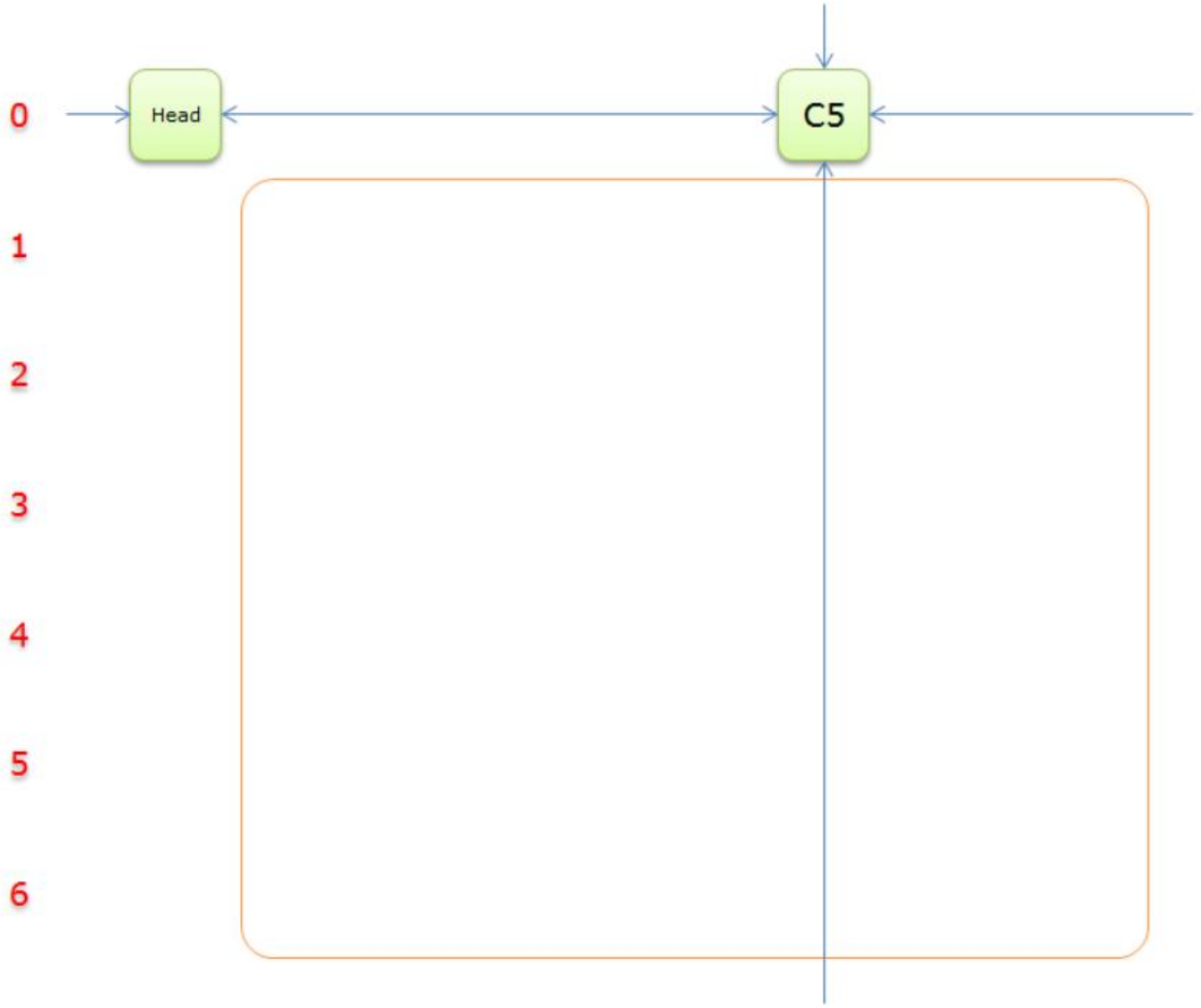
3

2

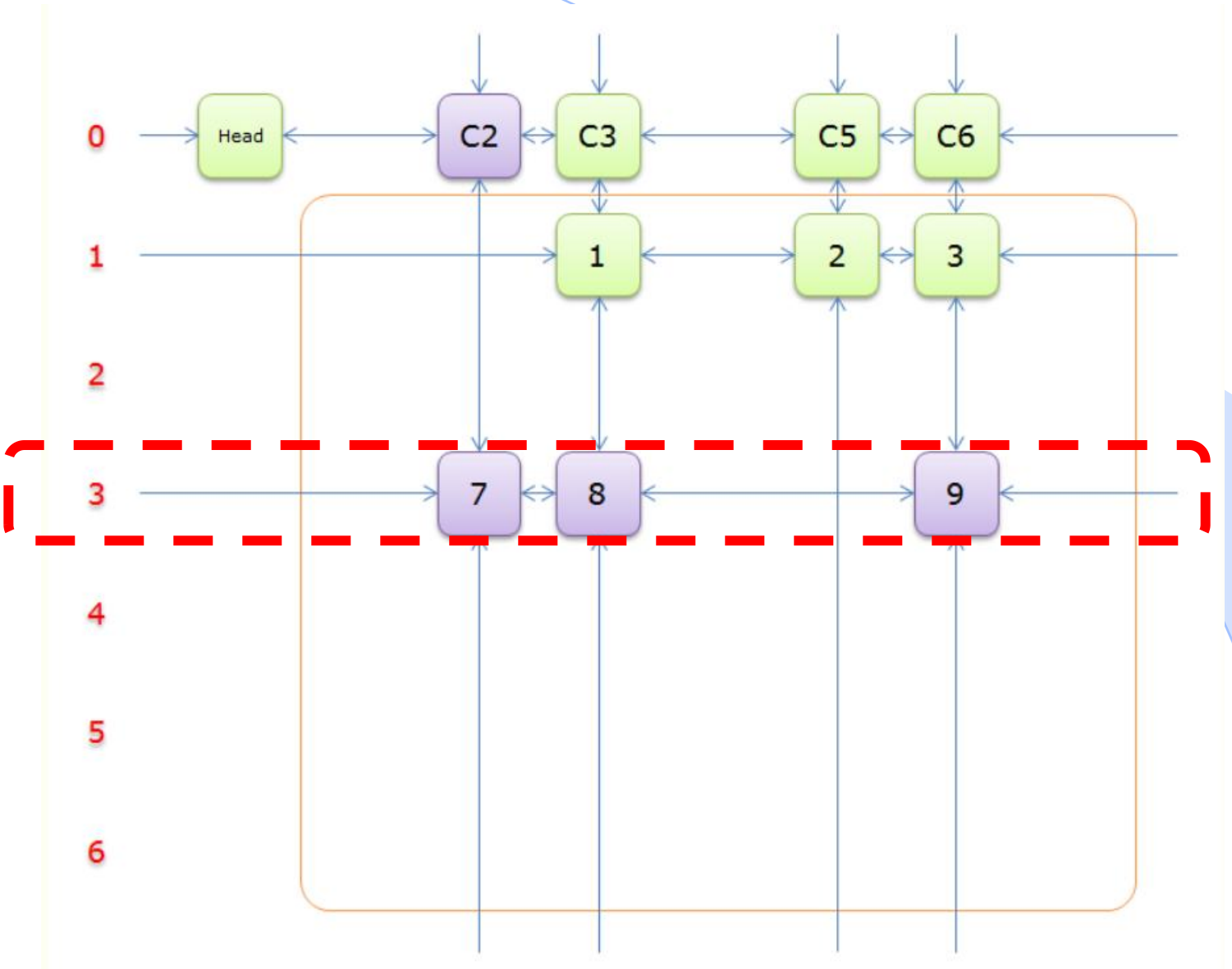
标记待删除的行和列



删除后的矩阵



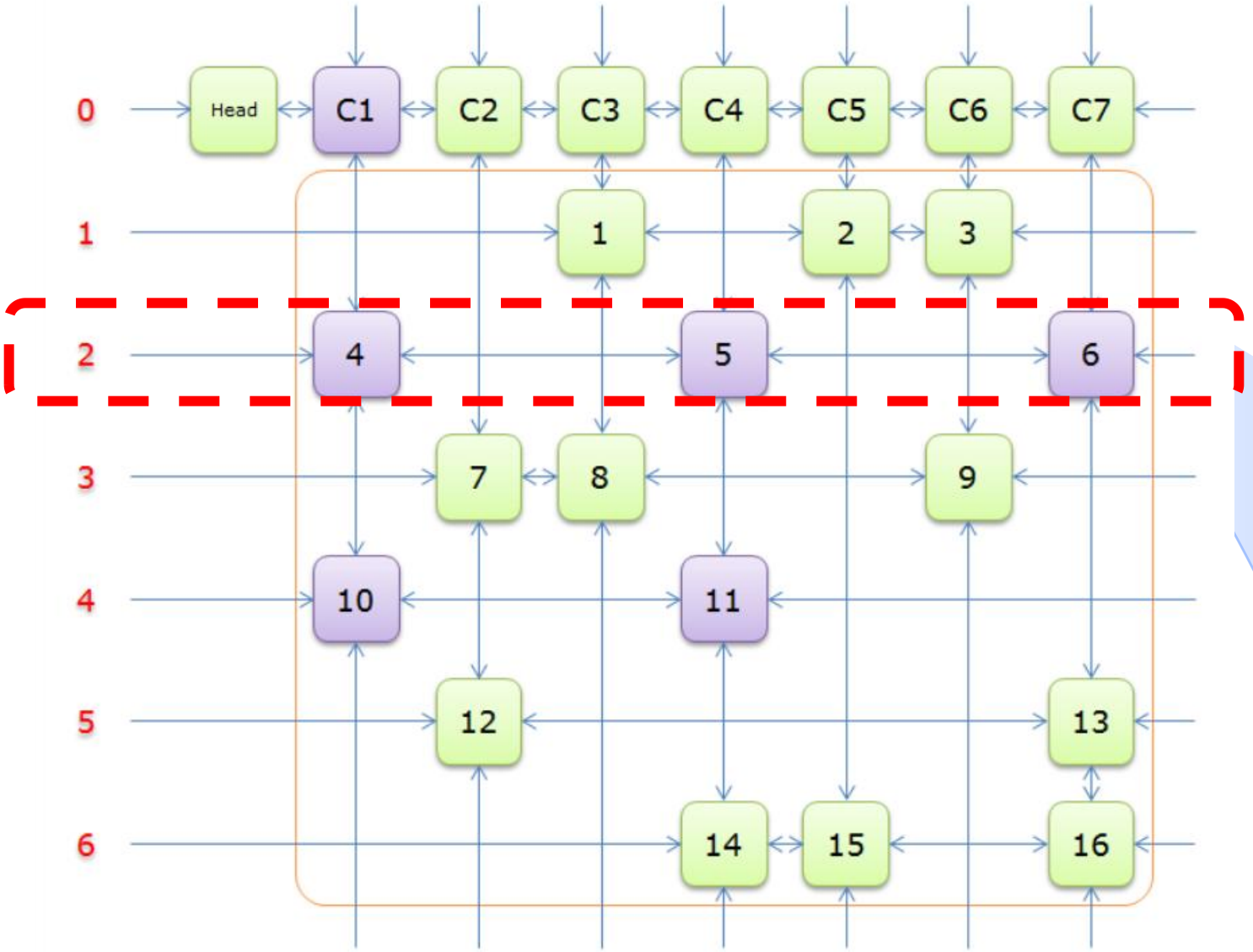
回溯过程（退回到选择2和3的状态）



3

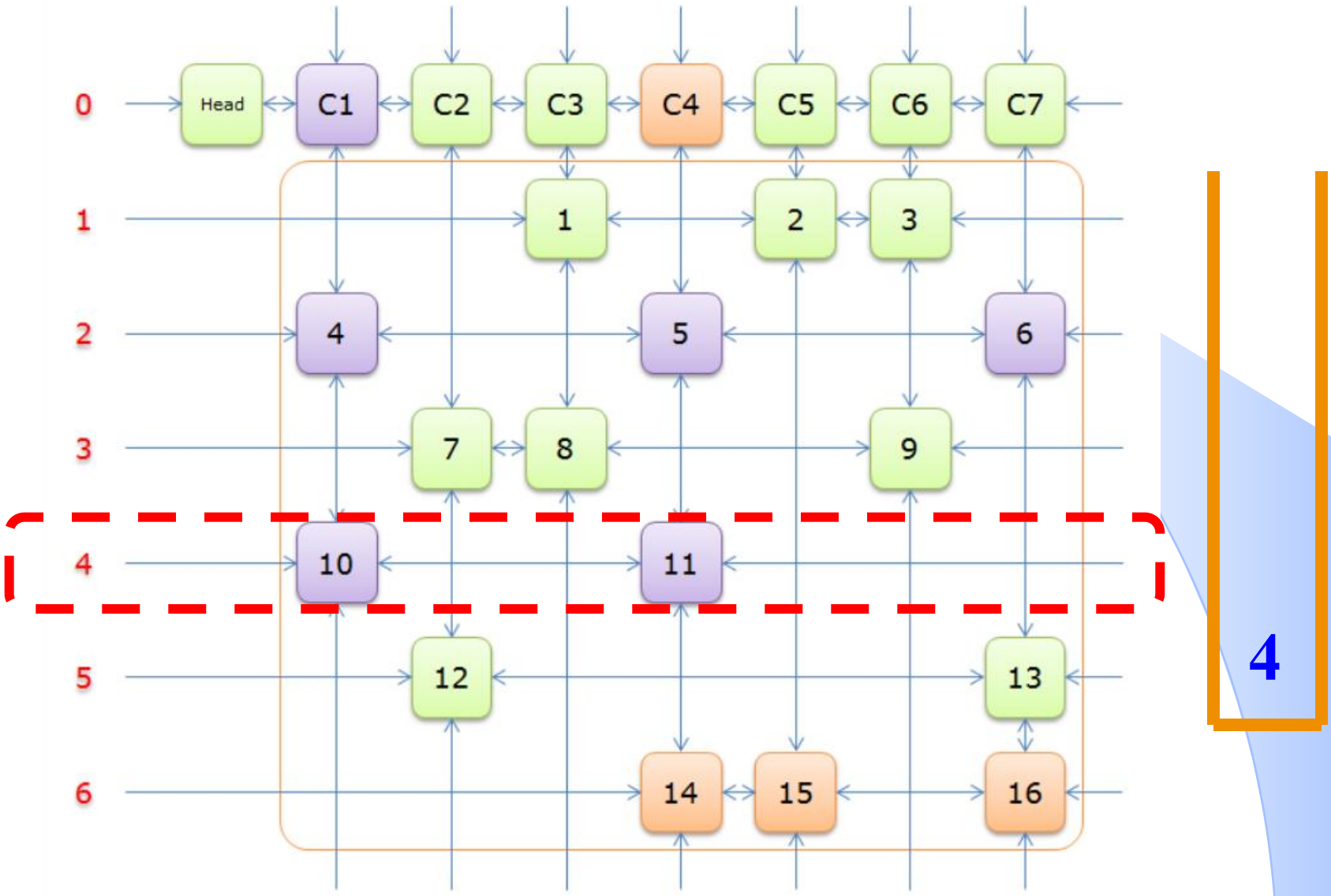
2

回溯（回溯到选择2的状态）

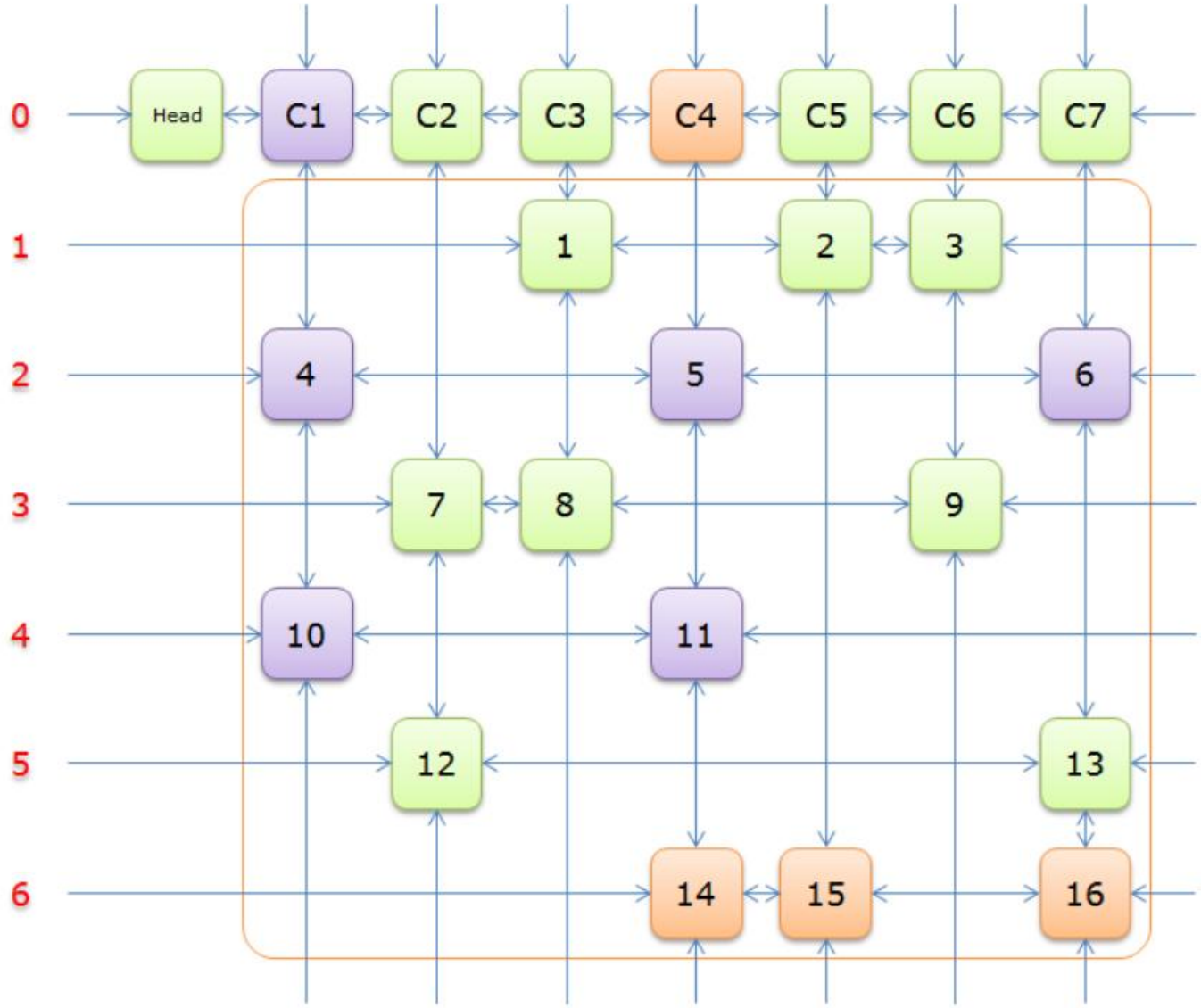


2

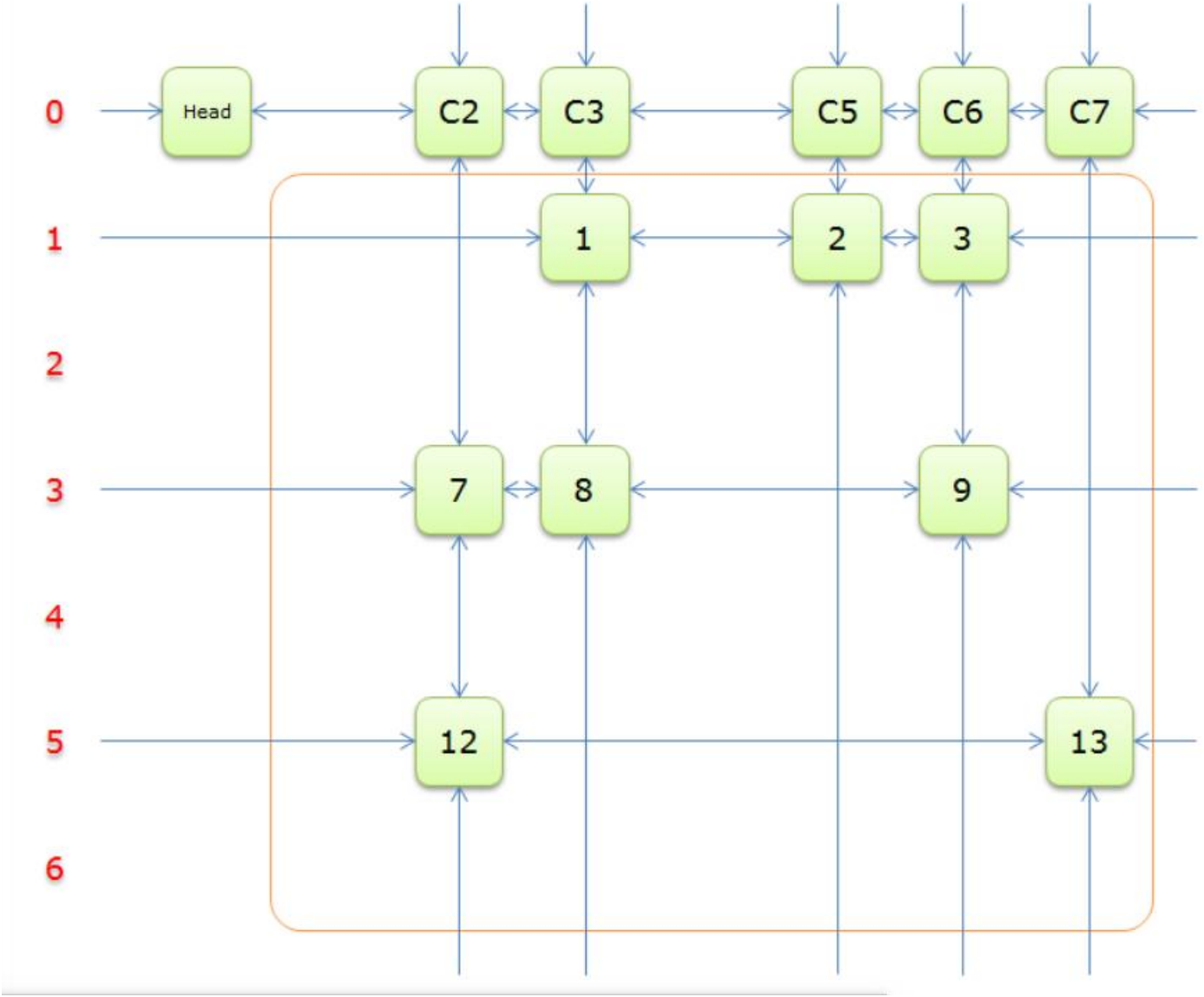
选择行（重新选择第4行）



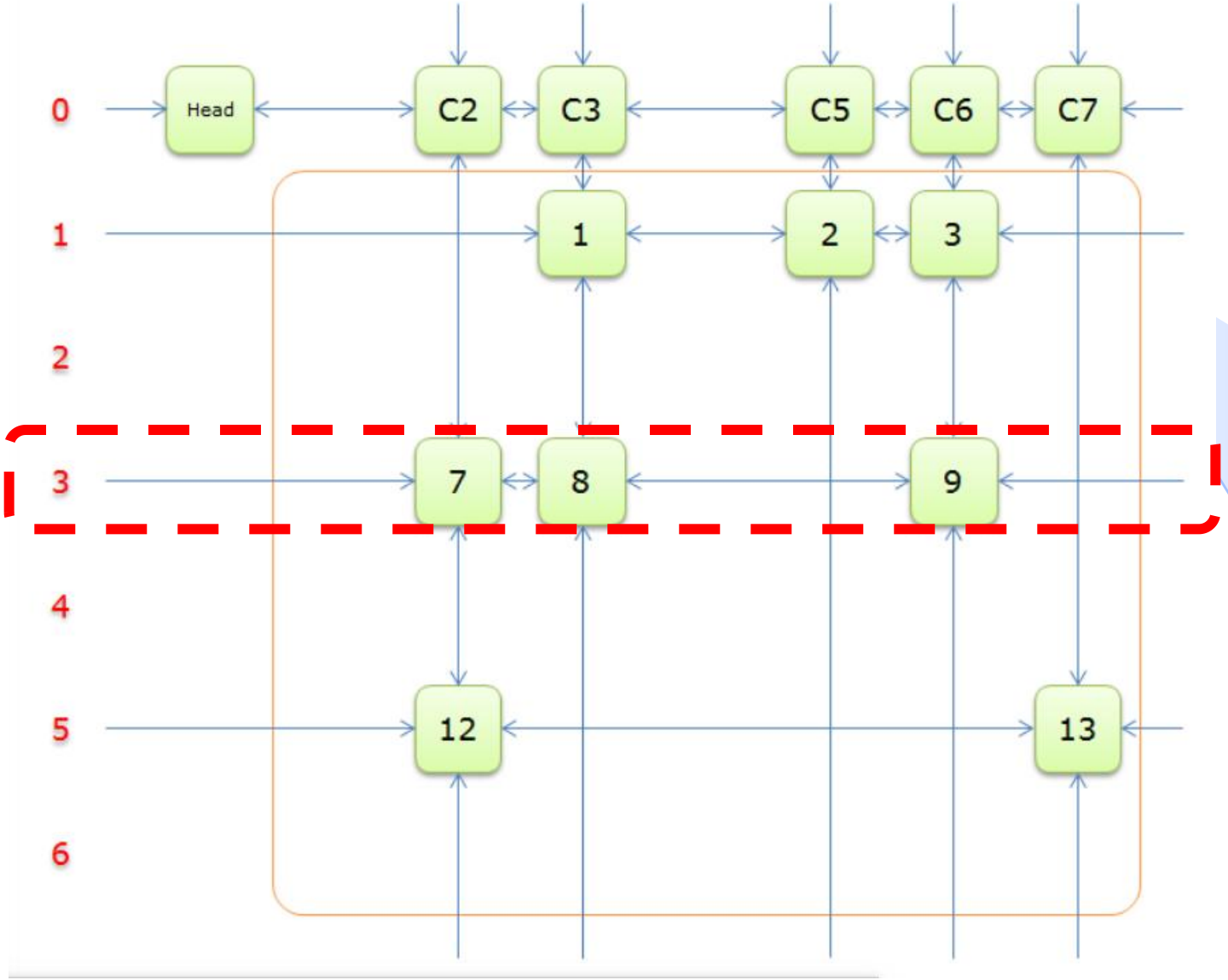
标记待删除的行和列



删除相关的行和列



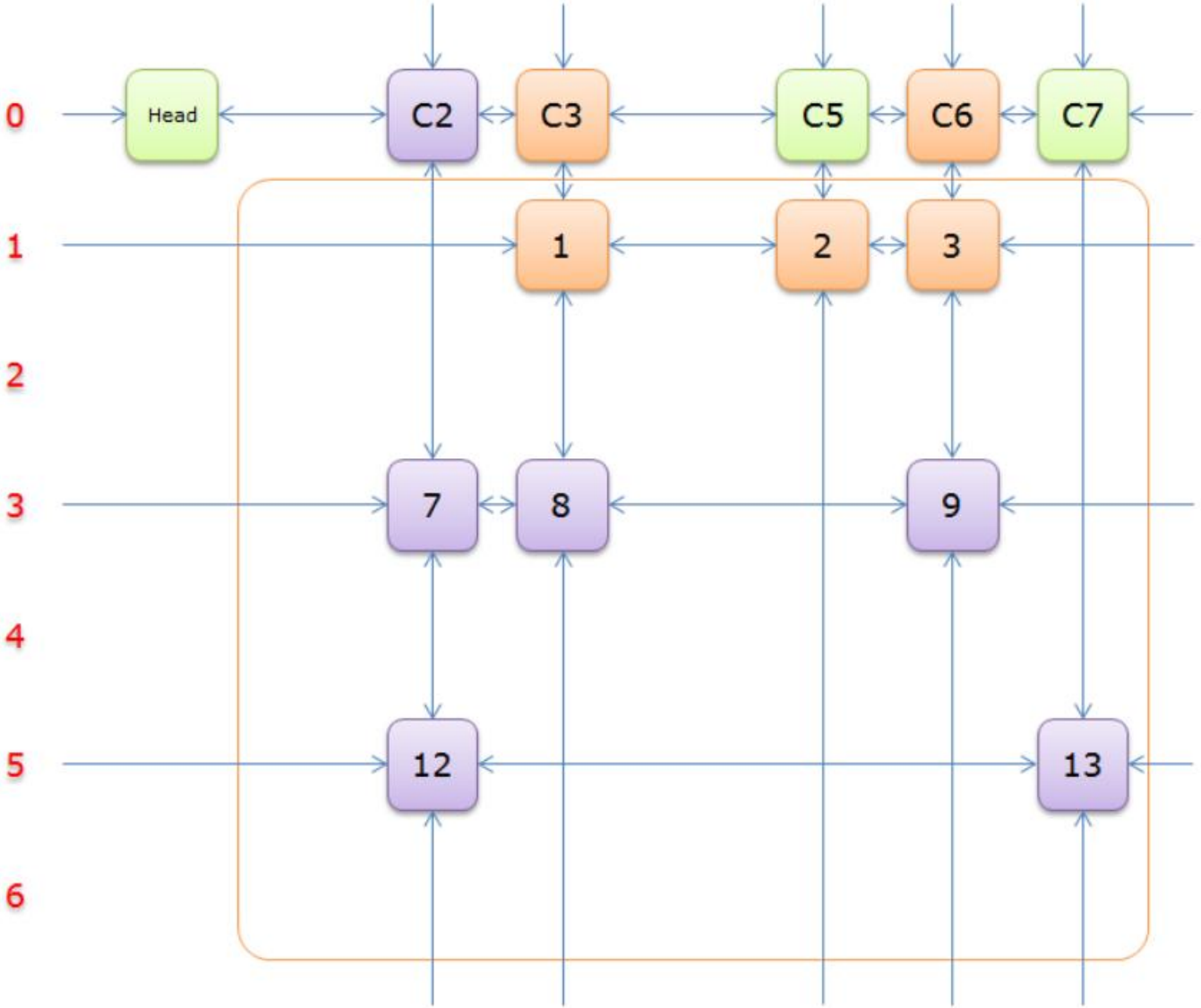
选择行（选择第3行）



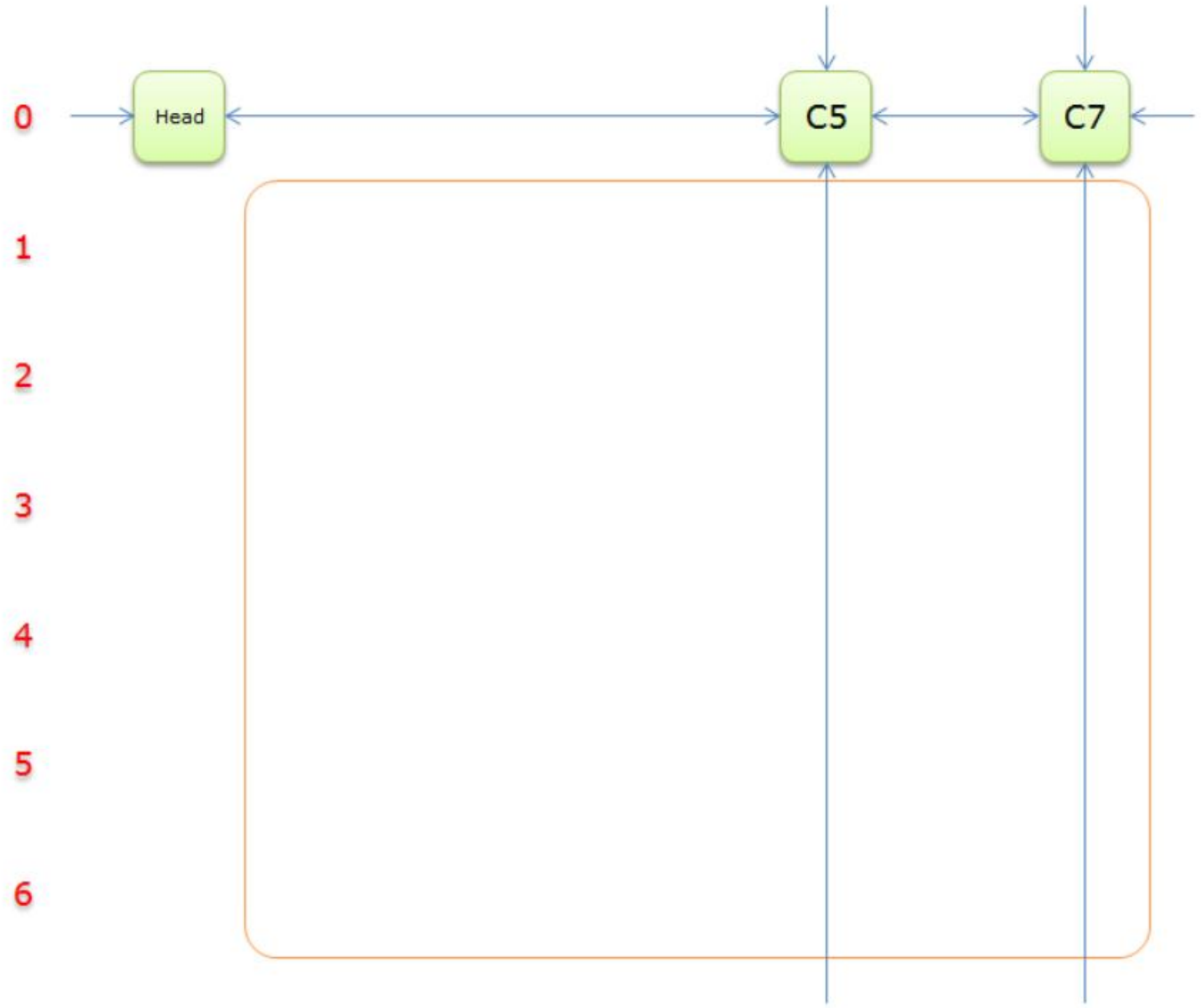
3

4

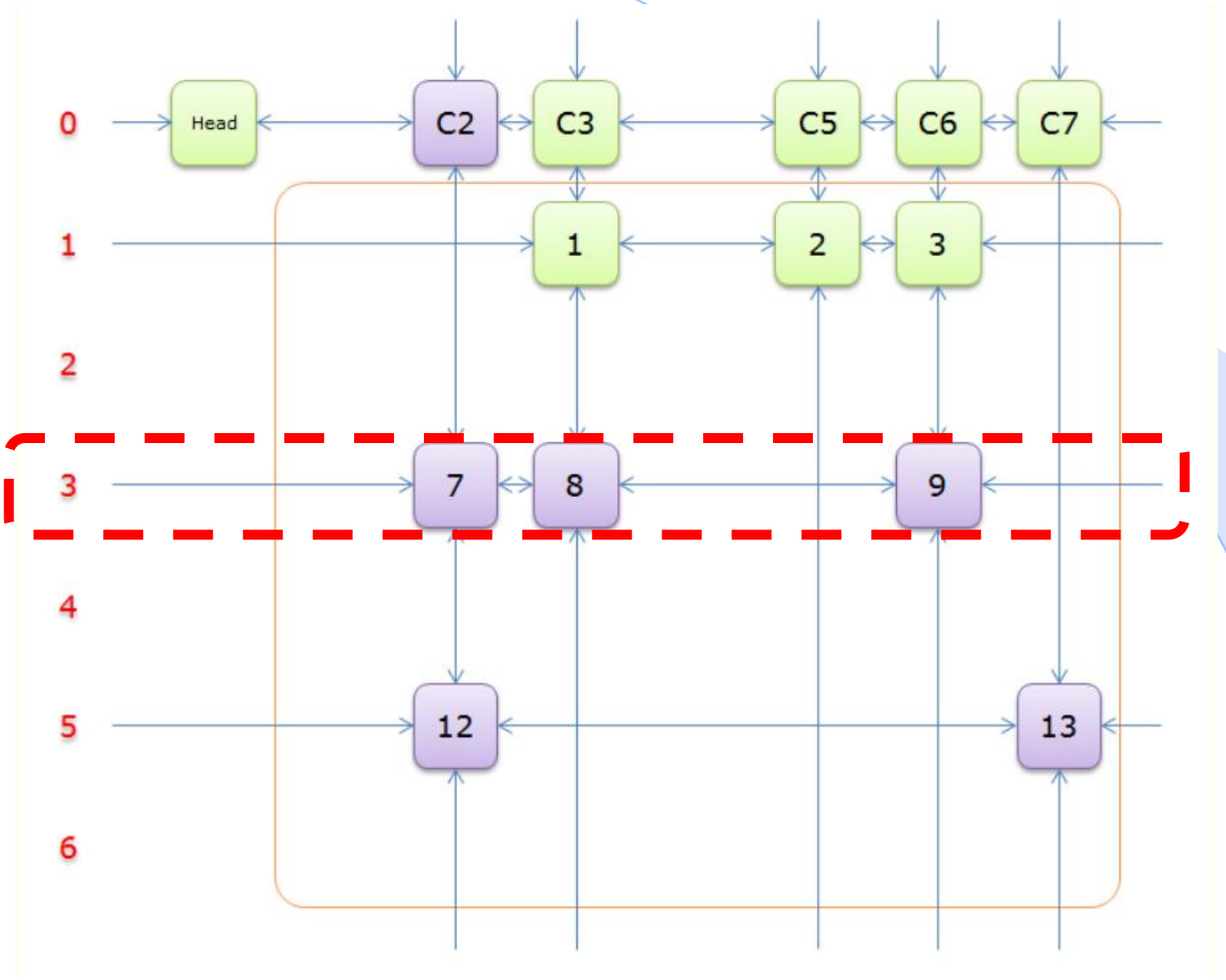
标记待删除的行和列



删除后矩阵

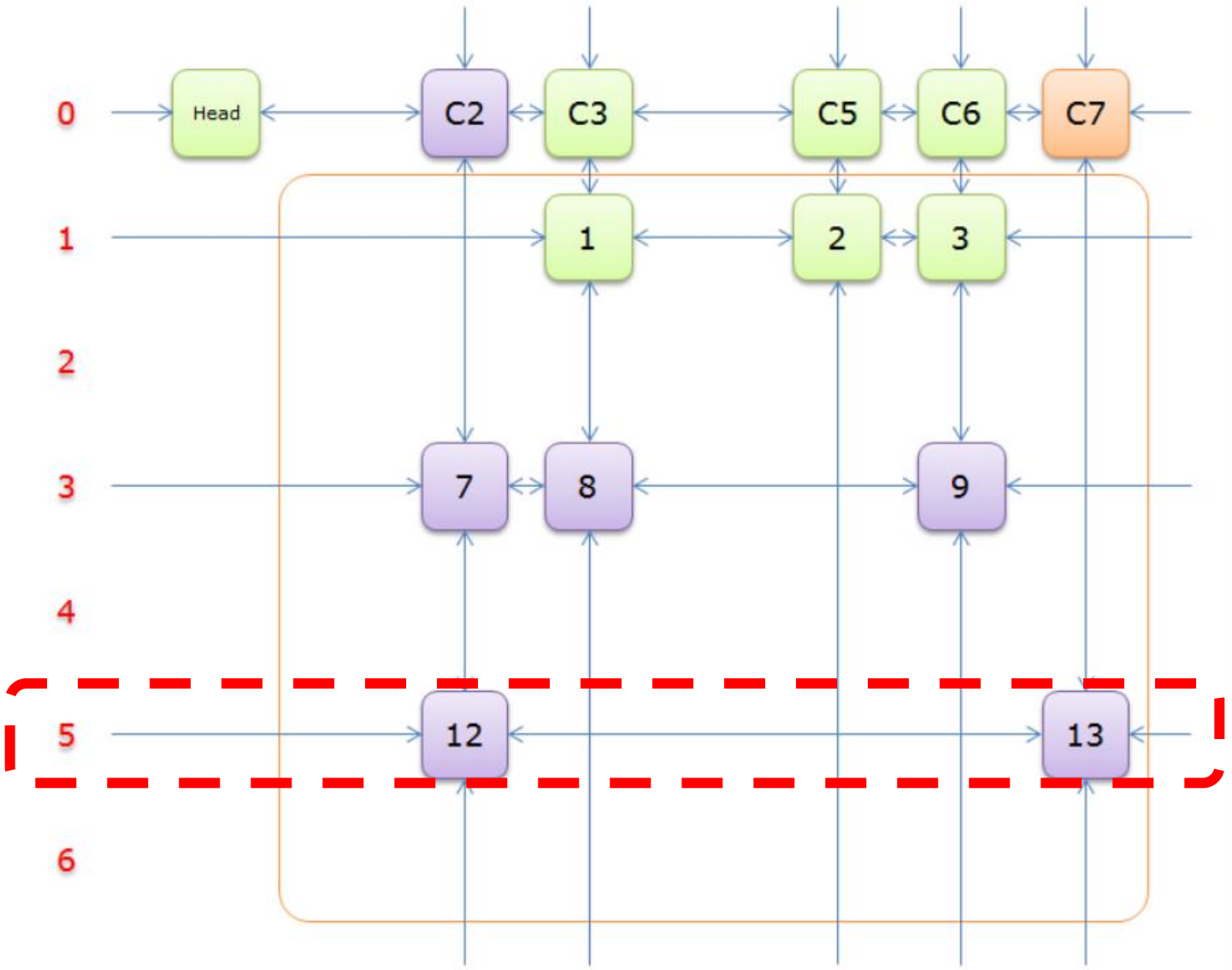


回溯（退回到选择4、3行的状态）



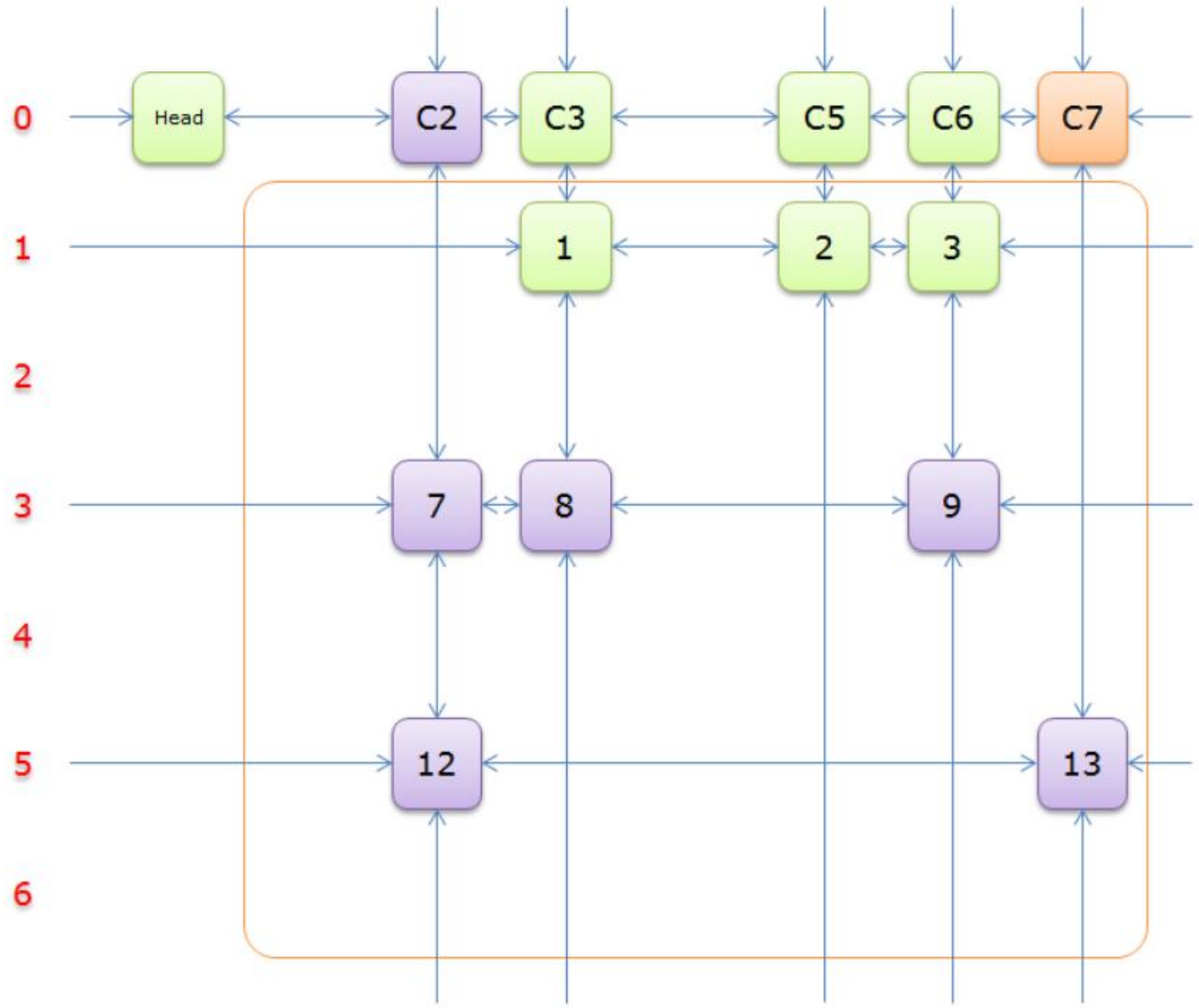
3
4

回溯（重新选择第5行）

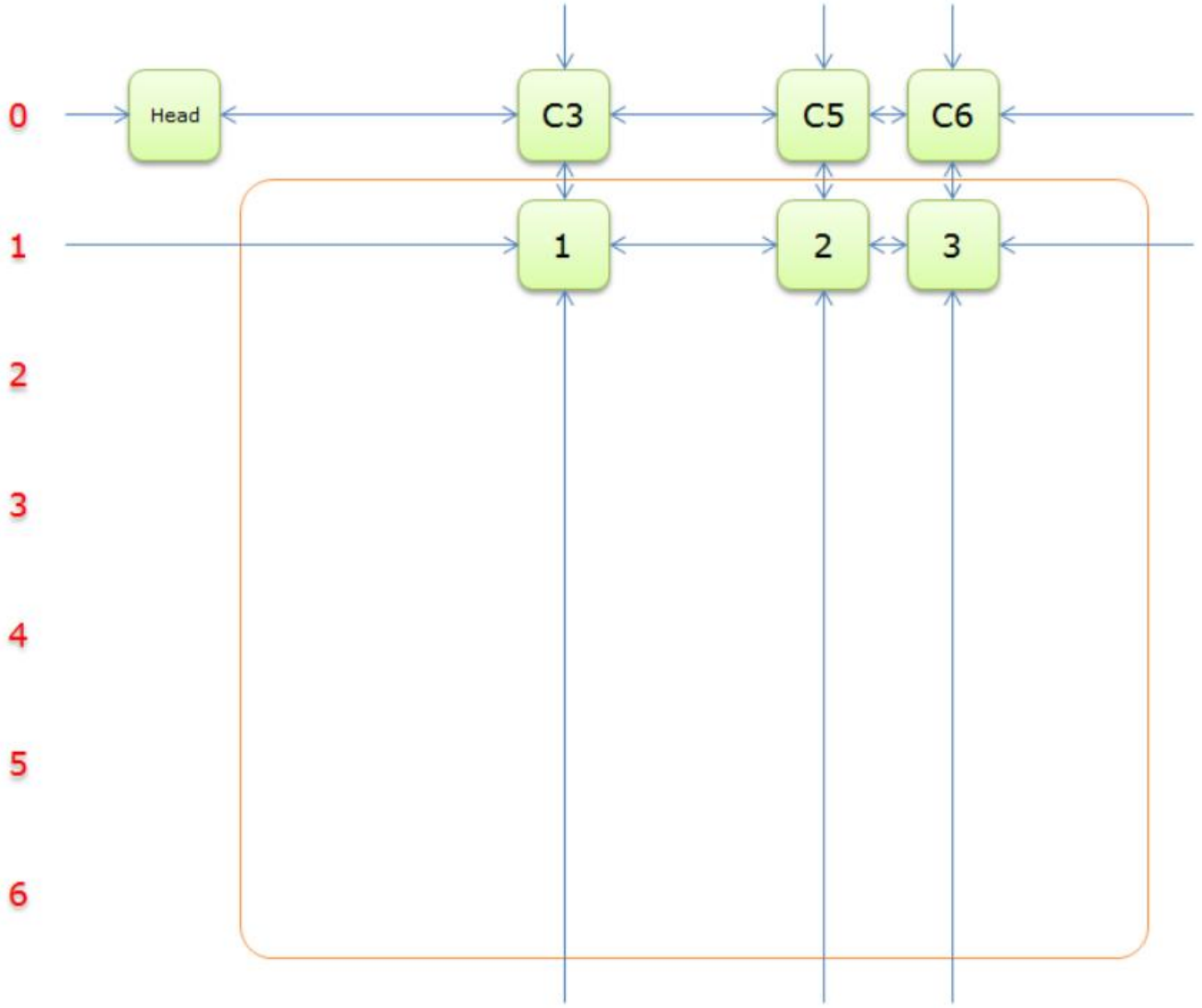


5
4

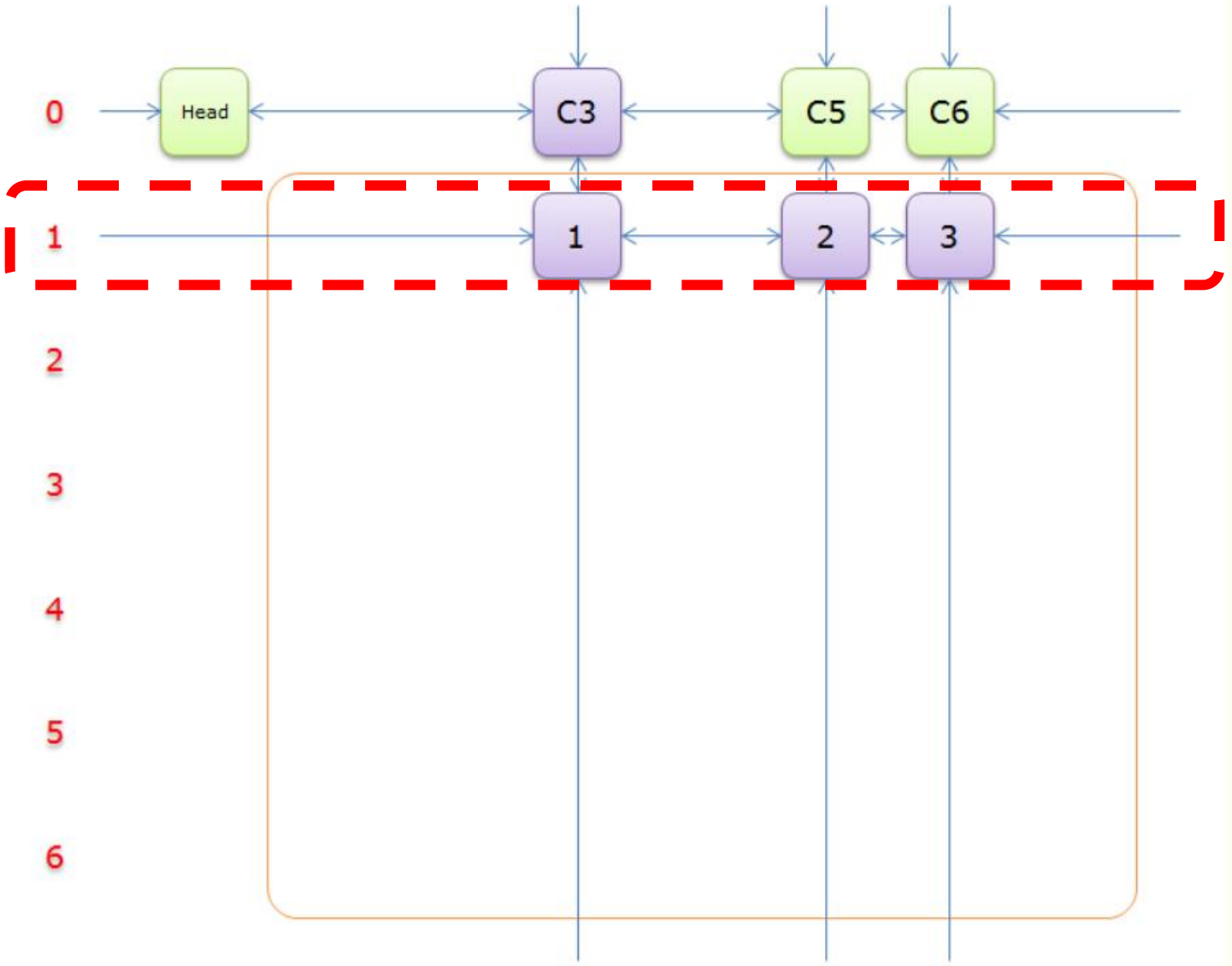
标记待删除的行和列



删除后矩阵

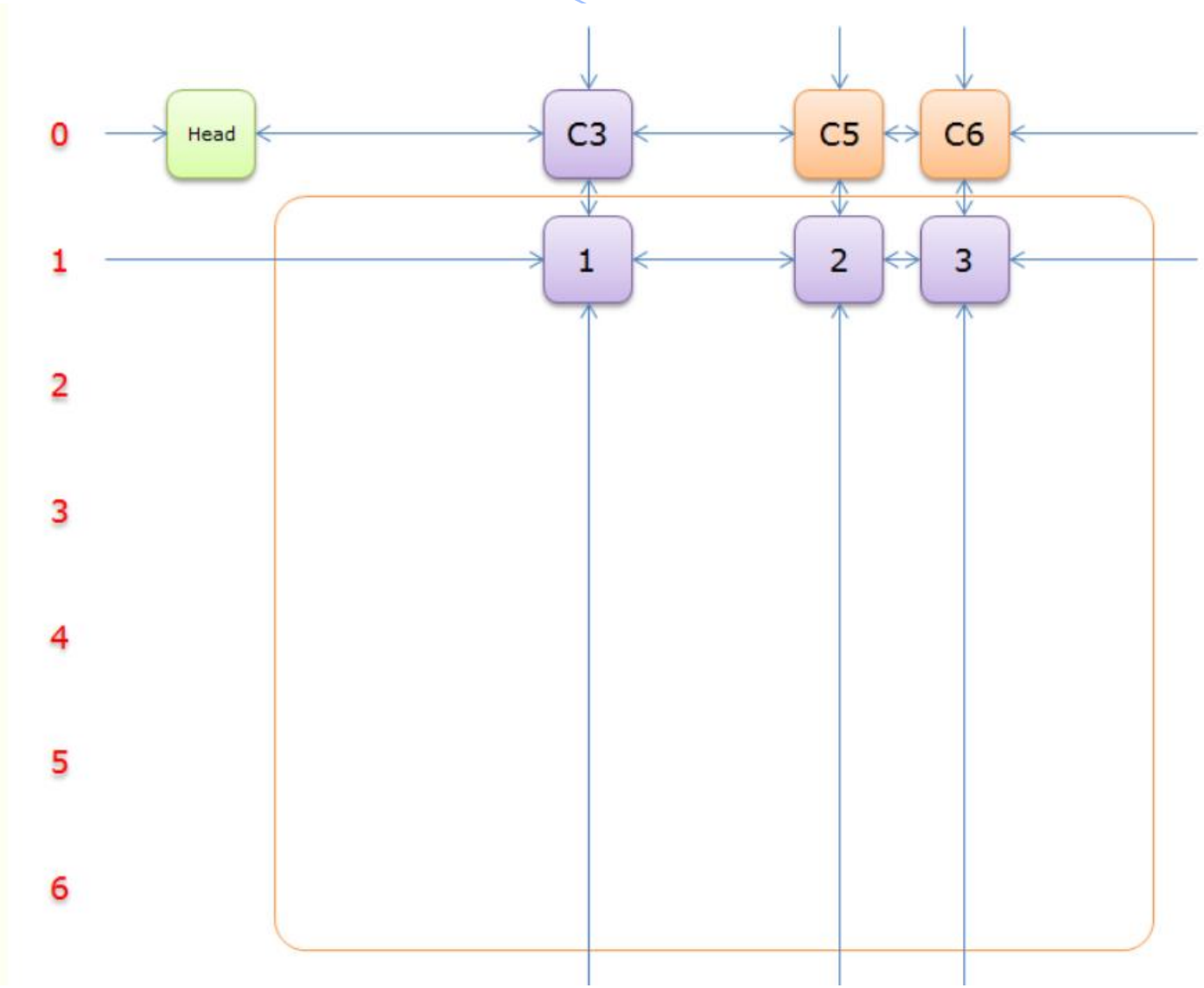


选择行

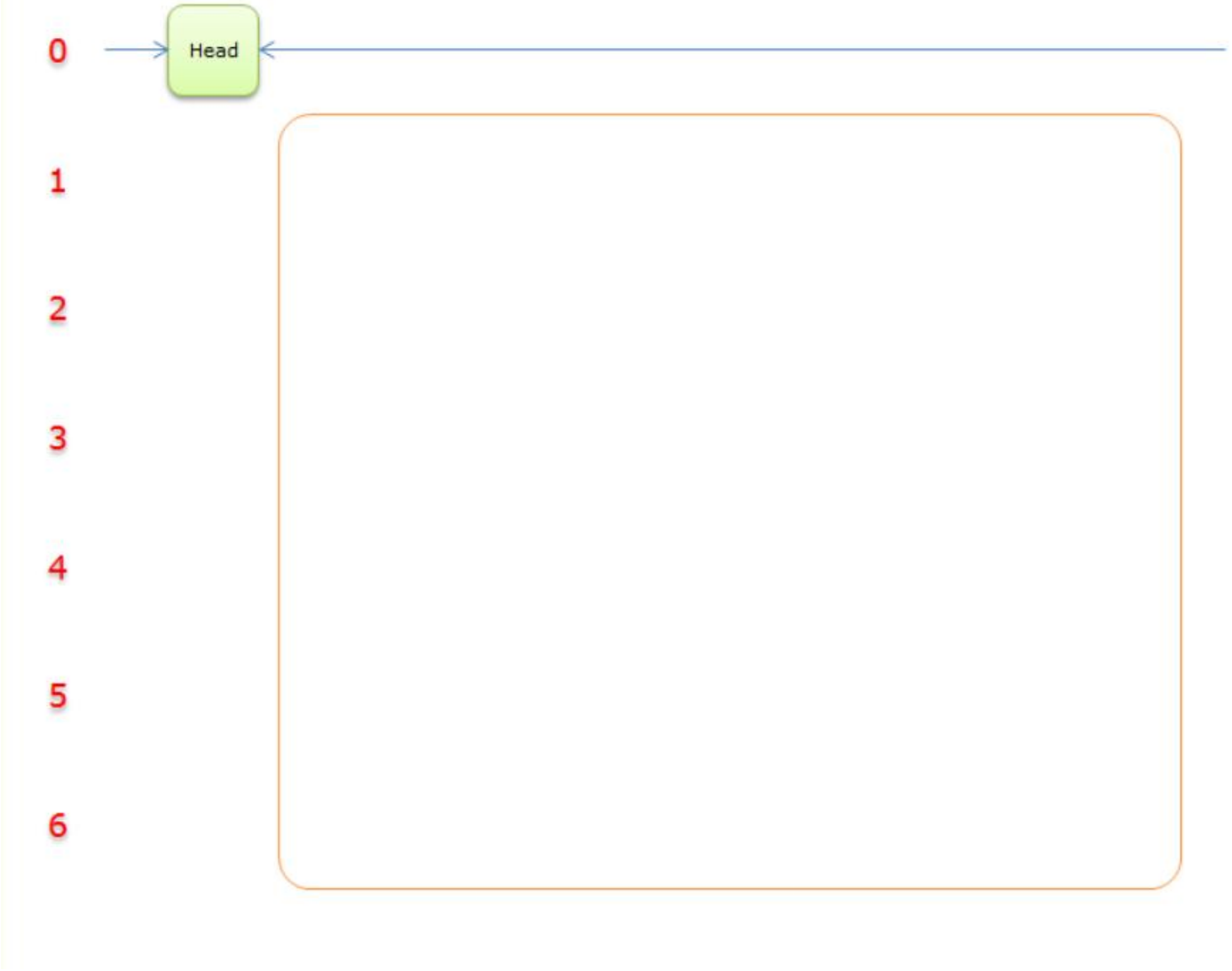


1
5
4

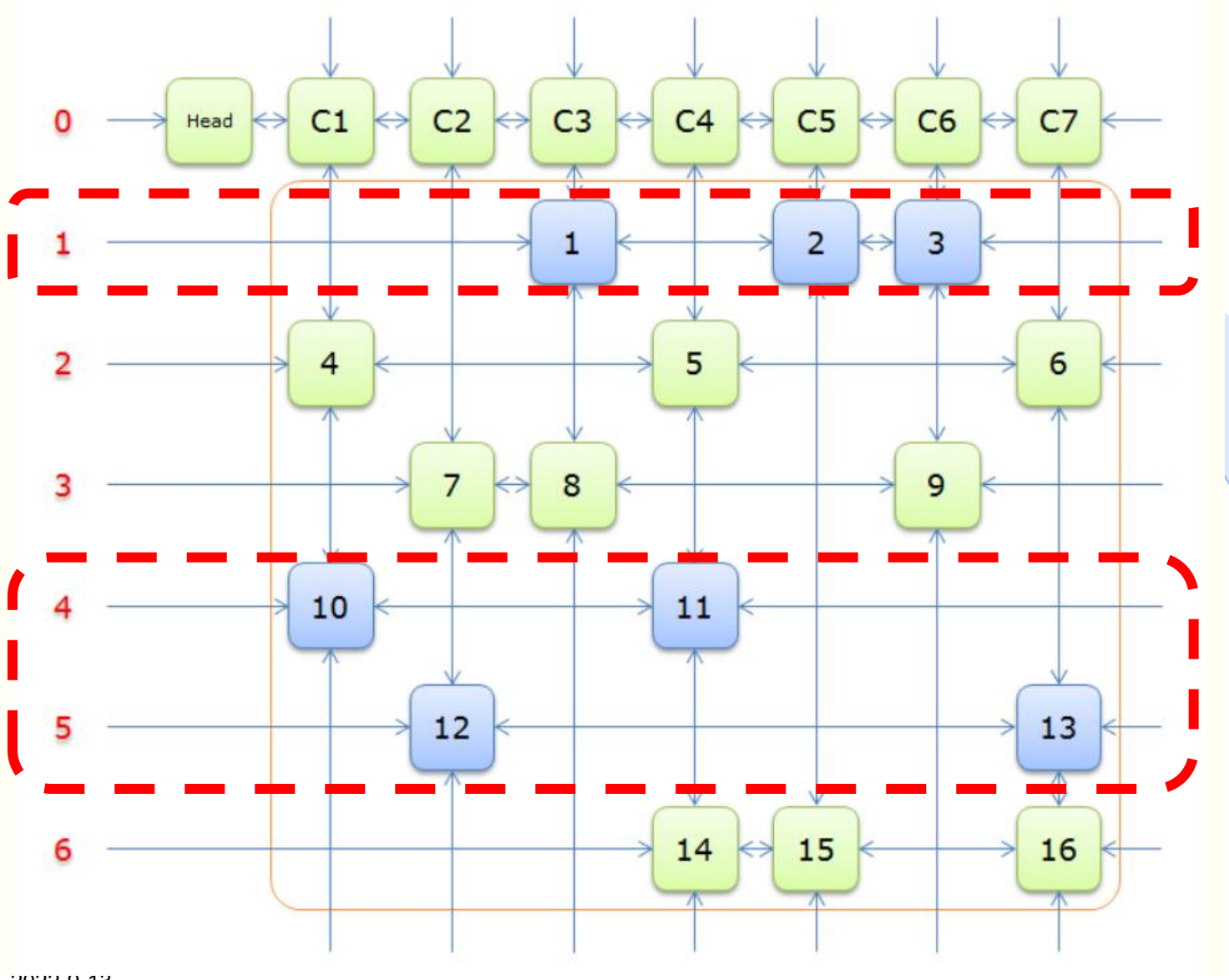
标记待删除的行和列



删除后



得到的解



1
5
4