Introdução

Matrizes esparsas são matrizes nas quais a maioria das posições são preenchidas por zeros. Para estas matrizes, podemos economizar um espaço significativo de memória se apenas os termos diferentes de zero forem armazenados. As operações matemáticas usuais sobre estas matrizes (somar, multiplicar, inverter, pivotar) também podem ser feitas em tempo muito menor se não armazenarmos as posições que contêm zeros.

O objetivo deste trabalho é implementar e testar uma forma particular de representação para armazenar e manipular as matrizes esparsas: listas encadeadas. Elas se contrapõem a outros formatos também usuais para representar matrizes esparsas, tipicamente baseados em conjuntos de vetores de índices e valores (como CRS, CCS etc). Esses formatos fornecem uma maneira eficiente de implementar operações matemáticas usuais sobre matrizes, mas não são tão eficientes para representar estruturas com tamanho variável e/ou desconhecido. É nesse último tipo de cenário que a representação por listas encadeadas apresenta vantagens.

Representação por Listas Encadeadas

Na representação por listas encadeadas, cada coluna da matriz é representada por uma lista linear circular com uma célula cabeça. Da mesma maneira, cada linha da matriz também é representada por uma lista linear circular com uma célula cabeça. Cada célula da estrutura, além das células-cabeça, representa os termos diferentes de zero da matriz. Um exemplo de tipo para essas células é ilustrado abaixo:

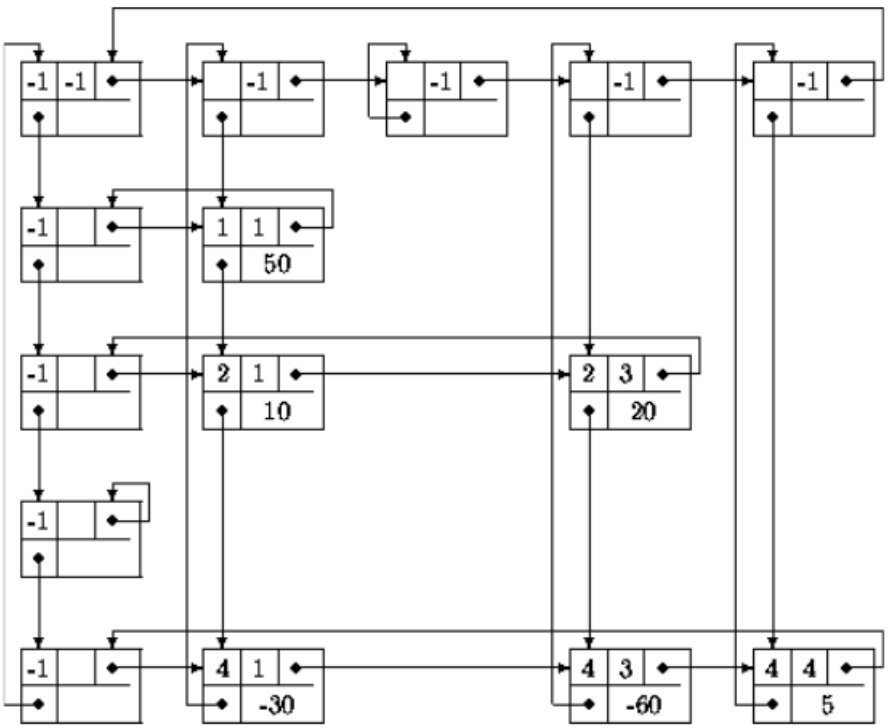
```
public class Cell {
  private int line;
  private int column;
  private float info;

  private Cell right;
  private Cell bellow;

  /*****************************************************************
  ************* Getters e Setters
  *****************************************************************/
}
```

O campo **below** é usado para apontar o próximo elemento diferente de zero na mesma coluna. O campo **right** é usado para apontar o próximo elemento diferente de zero na mesma linha. Dada uma matriz A, para um valor $A(i, j)$ diferente de zero, há uma célula com o campo **info** contendo $A(i, j)$, o campo *line* contendo i e o campo *column* contendo j. Esta célula pertence à lista circular da linha i e também à lista circular da coluna j. Ou seja, cada célula pertence a duas listas ao mesmo tempo. Para diferenciar as células cabeça, valores inválidos (por exemplo, -1) podem ser usados nos campos *line* e *column* destas células.

Como exemplo, considere a matriz esparsa

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

Uma possível representação dessa matriz usando listas encadeadas pode ser vista na Figura .



Com esta representação, uma matrix esparsa $m \times n$ com r elementos diferentes de zero gastará $(m+n+r)$ células. É bem verdade que cada célula ocupa vários bytes na memória, no entanto, o total de memória usado será menor do que as $m \times n$ posições de memória necessárias para representar a matriz toda, desde que r seja suficientemente pequeno.

Descrição

Dada a forma de representação acima, o trabalho consiste em desenvolver um TAD para criação e manipulação de matrizes. O TAD deverá se chamar Matrix e esse identificador deverá ser usado para referenciar a forma de representação acima. Toda operação deve retornar 0 em caso de sucesso e $\neq 0$ em caso de erro.

`Matrix()`

lê de stdin os elementos diferentes de zero de uma matriz e monta a estrutura especificada acima **para listas encadeadas**, retornando a matriz

criada. A entrada consiste dos valores de m e n (número de linhas e de colunas da matriz) seguidos de triplas (i, j, $valor$) para os elementos diferentes de zero da matriz. Por exemplo, para a matriz da Figura , a entrada seria:

```
4  4
1  1   50.0
2  1   10.0
2  3   20.0
4  1  -30.0
4  3  -60.0
4  4    5.0
0
```

Para facilitar a leitura de stdin, deve-se usar "0" como marcador especial de fim de matriz após a última tripla (vide exemplo acima).

`int matrix_print()`

imprime a matriz para stdout no mesmo formato usado no construtor;

`int matrix_add(Matrix m, Matrix n, Matrix r)`

recebe como parâmetros as matrizes m e n, retornando em r a soma das mesmas (a estrutura da matriz retornada deve ser alocada dinamicamente pela própria operação);

`int matrix_multiply(Matrix m, Matrix n, Matrix r)`

recebe como parâmetros as matrizes m e n, retornando em r a multiplicação das mesmas (a estrutura da matriz retornada deve ser alocada dinamicamente pela própria operação);

`int matrix_transpose(Matrix m, Matrix r)`

recebe como parâmetro a matriz m, retornando em r a matriz m^T - a transposta de m (a estrutura da matriz retornada deve ser alocada dinamicamente pela própria operação);

`int matrix_getelem(int x, int y, float elem)`

retorna o valor do elemento (x, y) da matriz m em elem;

`int matrix_setelem(int x, int y, float elem)`

atribui ao elemento (x, y) da matriz m o valor elem.

Resolução e Testes

A resolução deve ser testada, **minimamente**, com o programa de teste a seguir, usando os pares de matrizes A e B abaixo como entradas:

$$A = \begin{bmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{bmatrix}, B = \begin{bmatrix} 50 & 30 & 0 & 0 \\ 10 & 0 & -20 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{bmatrix} \text{ e}$$

$$A = \begin{bmatrix} 1 & 0 & 0 & 6 & 0 \\ 0 & 10.5 & 0 & 0 & 0 \\ 0 & 0 & 0.015 & 0 & 0 \\ 0 & 250.5 & 0 & -280 & 33.32 \\ 0 & 0 & 0 & 0 & 12 \end{bmatrix},$$

$$B = \begin{bmatrix} 1 & 0 & 0 & 6 \\ 0 & 10.5 & 0 & 0 \\ 0 & 0 & 0.3 & 0 \\ 0 & 100 & 0 & 30 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$