Universidade Federal do Ceará - Campus Quixadá
QXD0010 – Estruturas de Dados – Turma 03A
Prof. Atílio Luiz

Primeiro Projeto

A solução da questão descrita neste documento devem ser entregue até a meia-noite do dia **08/08/2021** pelo Moodle.

Leia atentamente as instruções abaixo.

Instruções:

- Este trabalho pode ser feito em **dupla** ou **individualmente** e deve ser implementado usando a linguagem de programação C++ **(Não aceitarei mais do que dois alunos por projeto)**
- Você tem até o dia 22 de julho para definir se vai fazer o trabalho em dupla ou individualmente. Me envie um email (`gomes.atilio@ufc.br`) com o nome completo e matrícula dos integrantes da sua dupla assim que ela for definida (basta um integrante me comunicar).
- Coloque a solução de cada questão em uma pasta específica. O seu trabalho deve ser compactado (.zip, .rar, etc.) e enviado para o Moodle na atividade correspondente ao Projeto 01.
- Identifique o seu código-fonte colocando o **nome** e **matrícula** dos integrantes da dupla como comentário no início de seu código.
- Indente corretamente o seu código para facilitar o entendimento. **Trabalhos com códigos maus indentados sofrerão redução na nota.**
- As estruturas de dados devem ser implementadas como TAD. Deve haver um arquivo de cabeçalho (.h) e um arquivo de implementação (.cpp) para cada estrutura de dado programada.
- Os programas-fonte devem estar devidamente organizados e documentados.
- Observação: Lembre-se de desalocar os endereços de memória alocados quando os mesmos não forem mais ser usados.
- **Observação: Qualquer indício de plágio resultará em nota ZERO para todos os envolvidos.**

DICA: COMECE O TRABALHO O QUANTO ANTES.

1 Matrizes esparsas

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

Há numerosos exemplos de aplicações que exigem o processamento de matrizes esparsas. Muitas se aplicam a problemas científicos ou de engenharia que só são facilmente entendidos por peritos. No entanto, há uma aplicação muito familiar que usa matriz esparsa: um programa de planilha. Muito embora a matriz de uma planilha comum seja muito grande, digamos 999 por 999, apenas uma porção da matriz pode realmente estar sendo usada em um dado momento. Planilhas usam a matriz para guardar fórmulas, valores e strings associados a cada posição. Em uma matriz esparsa, o armazenamento para cada elemento é alocado de um espaço de memória livre (heap) conforme se torne necessário. Embora apenas uma pequena porção dos elementos esteja realmente sendo usada, a matriz pode parecer muito grande – maior do que o que normalmente caberia na memória do computador.

Uma maneira eficiente de representar estruturas com tamanho variável e/ou desconhecido é com o emprego de alocação encadeada, utilizando listas. Vamos usar essa representação para armazenar as matrizes esparsas. Cada coluna da matriz será representada por uma **lista linear circular** com um **nó cabeça**. Da mesma maneira, cada linha da matriz também será representada por uma lista linear circular com um nó cabeça. Cada nó da estrutura, além dos nós cabeça, representará os termos diferentes de zero da matriz e deverá ser como no código abaixo:

```
1 struct Node {
2     Node *direita;
3     Node *abaixo;
4     int linha;
5     int coluna;
6     double valor;
7 };
```

O campo `abaixo` do `struct Node` deve ser usado para referenciar o próximo elemento diferente de zero na mesma coluna. O campo `direita` deve ser usado para referenciar o próximo elemento diferente de zero na mesma linha. Dada uma matriz A, para um valor $A(i, j)$ diferente de zero, deverá haver um nó com o campo `valor` contendo $A(i, j)$, o campo `linha` contendo i e o campo `coluna` contendo j. Esse nó deverá pertencer à lista circular da linha i e também deverá pertencer à lista circular da coluna j. Ou seja, cada nó pertencerá a duas listas ao mesmo tempo. Para diferenciar os nós cabeça, coloque -1 nos campos `linha` e `coluna` desses nós.

Considere a seguinte matriz esparsa:

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

A representação da matriz A pode ser vista na Figura 1.

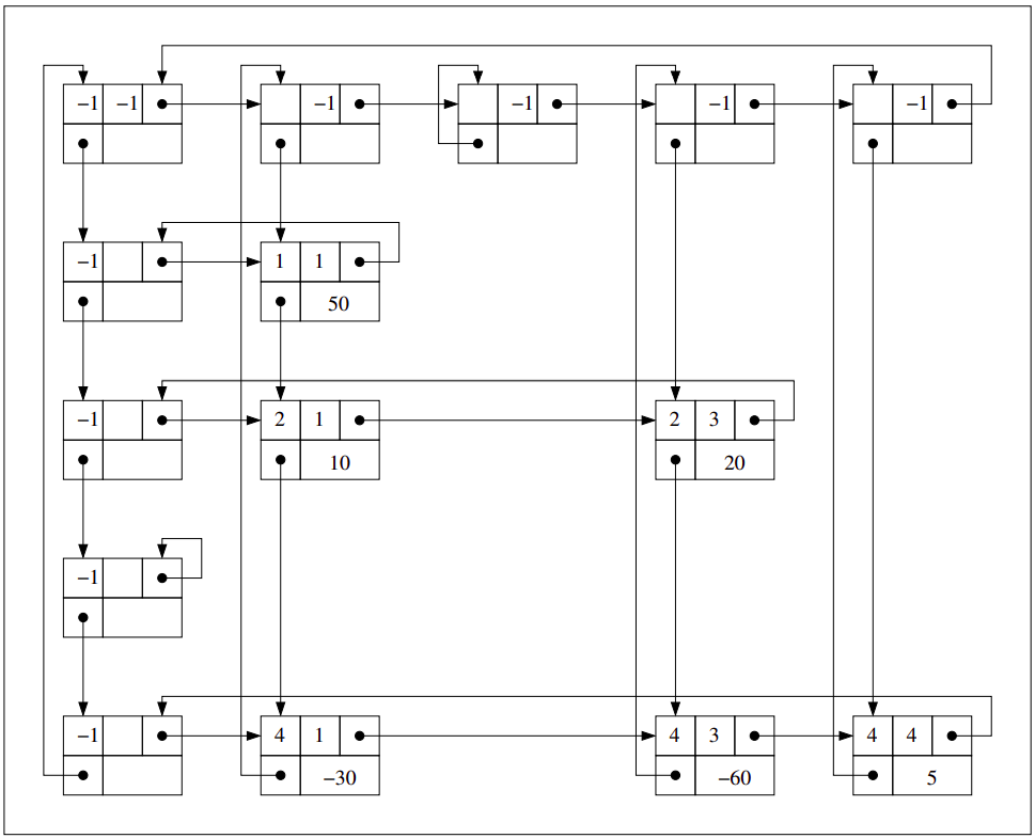

Figura 1: Exemplo de matriz esparsa.

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

O trabalho consiste em desenvolver em C++ um tipo abstrato de dados **Matriz** com as seguintes operações, conforme esta especificação:

- `Matriz(int m, int n);`
  Inicializa uma matriz esparsa vazia com capacidade para m linhas e n colunas.

- `~Matriz();`
  Destrutor que libera toda a memória que foi alocada dinamicamente para esta estrutura de dados.

- `void insert(int i, int j, double value);`
  Insere um valor na célula (i, j) da matriz, onde i é alinha e j é a coluna.

- `double getValue(int i, int j);`
  Devolve o valor na célula (i, j) da matriz, onde i é a linha e j é a coluna.

- `void print();`
  Esse método imprime a matriz A, inclusive os elementos iguais a zero.

As funções-membro descritas acima são apenas as funções necessárias para possibilitar o mínimo de funcionamento da matriz. Se você identificar outras funções-membro adicionais que facilitem o uso da estrutura de dados, você pode programá-las.

Além do TAD Matriz, você deve implementar as seguintes 3 funções adicionais (que não são funções-membro da Matriz, são funções externas à classe):

- `Matriz *soma(Matriz *A, Matriz *B);`
  Essa função recebe como parâmetro as matrizes A e B, devolvendo uma matriz C que é a soma de A com B.

- `Matriz *multiplica(Matriz *A, Matriz *B);`
  Essa função recebe como parâmetro as matrizes A e B, devolvendo uma matriz C que é o produto de A por B.

- `Matriz *lerMatrizDeArquivo(std::string nome_do_arquivo);`
  Essa função lê, de um arquivo de entrada, os elementos diferentes de zero de uma matriz e monta a estrutura especificada anteriormente, devolvendo um ponteiro para uma Matriz alocada dinamicamente. Considere que a primeira linha do arquivo de entrada consiste dos valores de m e n (número de linhas e de colunas da matriz), e as demais linhas do arquivo são constituídas de triplas $(i, j, valor)$ para os elementos diferentes de zero da matriz. Por exemplo, para a matriz A anterior, o conteúdo do arquivo de 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
  ```

Observação: É obrigatório o uso de alocação dinâmica de memória para implementar as listas de adjacência que representam as matrizes.

Além da matriz A outras matrizes podem ser lidas para testar os métodos, como, por exemplo:

$$B = \begin{pmatrix} 50 & 30 & 0 & 0 \\ 10 & 0 & -20 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -5 \end{pmatrix} \qquad C = \begin{pmatrix} 3 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$$

As funções deverão ser testadas utilizando-se um programa `main.cpp`. Um exemplo bem simples de um pedaço de main.cpp é apresentado abaixo, mas o mais recomendável seria programar um menu ou algo semelhante a um menu.

```
void main() {
   Matriz *A = lerMatrizDeArquivo("A.txt");
   Matriz *B = lerMatrizDeArquivo("B.txt");
   A->print();
   B->print();
   Matriz *C = soma(A,B);
   C->print();
   Matriz *D = multiplica(A,B);
   D->print();
   delete A;
   delete B;
   delete C;
   delete D;
   return 0;
}
```

1.1 O que deve ser submetido

- **Deverá ser submetido:**
  - Um **relatório do trabalho** realizado, contendo a especificação completa das estruturas de dados utilizadas; e as decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado. Uma seção descrevendo como o trabalho foi dividido entre a dupla, se for o caso; além das dificuldades encontradas. Referências bibliográficas, tutoriais, vídeos ou outros materiais consultados.
  - O código-fonte devidamente organizado e documentado.

- Um dos parâmetros utilizados na avaliação da qualidade de uma implementação consiste na constatação da presença ou ausência de comentários. Comente o seu código. Mas também não comente por comentar, forneça bons comentários.

- Outro parâmetro de avaliação de código é a *portabilidade*. Dentre as diversas preocupações da portabilidade, existe a tentativa de codificar programas que sejam compiláveis em qualquer sistema operacional. Como testarei o seu código em uma máquina que roda Linux, não use bibliotecas que só existem para o sistema Windows como, por exemplo, a biblioteca `conio.h` e outras tantas.