Acadêmicos: Iuri Almeida Pereira e Luis Henrique Da Silva Resende
Disciplina: Estrutura de Dados
Docente: Thiago Naves

Trabalho Encadeamento

1. **Estrutura de dados do aluno**
   a. tem 5 variáveis, a primeira variável do tipo **inteiro** que recebe a matricula do aluno, a segunda do tipo **char(caractere)** onde vai receber o nome do aluno e as outras 3 é do tipo **float** recebendo as 3 notas do aluno.

```c
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};
```

2. **Estrutura elemento** tem como variáveis a estrutura aluno que vai armazenar dentro os dados dessa estrutura, tem 4 estruturas da própria estrutura elemento que vai armazenar 4 ponteiros, e duas variáveis do tipo inteiro que vai armazenar a coordenada linha e coluna daquele elemento/nó inserido. Apelidamos a estrutura elemento para um ponteiro **Matriz,** esse ponteiro guarda a cabeça da matriz (posição inicial).
   a. **dir** aponta para a direita do elemento/nó;
   b. **esq** aponta para a esquerda do elemento/nó;
   c. **cima** aponta para cima do elemento/nó;
   d. **baixo** que aponta para baixo do elemento/nó.

```c
struct elemento{
    struct aluno dados;
    struct elemento *dir;
    struct elemento *esq;
    struct elemento *cima;
    struct elemento *baixo;
    int lin, col;
};
typedef struct elemento Elemento;
typedef struct elemento* Matriz;
```

3. **Função inserir_dados do tipo void**
   a. Recebe como parâmetro a estrutura aluno em forma de ponteiro, mas o intuito desta função é só que quando seja chamada, ela vai pedir ao usuário que entre com os dados do aluno. Ela é do tipo ponteiro, porque quando for armazenar essas informações, vai ser buscado na memória o que foi armazenado.

```c
// Função para inserir os dados da struct aluno
void inserir_dados(struct aluno *al){
    system( Command: "cls");
    printf( format: "\n\tMatricula...: ");
    scanf( format: "%d", &al->matricula);
    printf( format: "\tNome...: ");
    scanf( format: " %[^\n]", al->nome);
    printf( format: "\tNota 1...: ");
    scanf( format: "%f", &al->n1);
    printf( format: "\tNota 2...: ");
    scanf( format: "%f", &al->n2);
    printf( format: "\tNota 3...: ");
    scanf( format: "%f", &al->n3);
/* função que recebe como parâmetro um struct, e
}
```

4. **Função cria_matriz do tipo Matriz**
    a. A função **cria_matriz** do tipo apelidado ponteiro **Matriz** pois ele vai retornar a matriz criada ao final. Ela recebe como parâmetro duas variáveis do tipo **inteiro** que o usuário irá informar, esses dois parâmetros é que vai definir o tamanho da matriz que o usuário vai querer.

```
Matriz* cria_matriz(int lin, int col){
```

    b. Primeiro criamos um ponteiro chamado **matriz** do tipo apelidado **Matriz**, depois alocamos memória para ele, esse ponteiro é a cabeça da matriz (posição inicial). Verificamos se o ponteiro criado é **NULO,** caso ele não seja, passamos para ele um valor **NULO.** Criamos um ponteiro do tipo elemento com o nome **aux.** A funcionalidade é que ele seja sempre o elemento anterior a um outro elemento dentro da matriz.

```
Matriz* matriz = (Matriz*) malloc( Size: sizeof(Matriz));
if(matriz != NULL){
    *matriz = NULL;
}
Elemento *aux;
```

Pensamos em alocar linhas e colunas da matriz começando de uma posição que só vai servir de referência, mas que não podem ser acessadas, essas posições inicias são negativas, posições anteriores a posição **0,** que no caso é a posição **-1.** Como foi dito essas posições negativas só irá servir como referência da matriz, elas só ficaram ao redor das informações, porque essas informações ficaram nas posições positivas.

Para alocar a matriz e inicializa-la, fizemos um laço de repetição com **for** para alocar as **linhas** e outro com **for** para alocar as **colunas**.

    c. **Alocando linhas**
        I. Na alocação das linhas o limite de linhas alocadas será o que vai ser informado pelo usuário e recebido na função como parâmetro.
        II. Criamos e alocamos memória para um ponteiro do tipo **elemento** chamado **no** dentro desse **for**. Esse ponteiro será as posições de referência da matriz, as posições negativas como foi informado acima. Dentro do **for** criamos 3 condições dentro:

```
for(int l = -1; l < lin; l++){
    Elemento *no = (Elemento*) malloc( Size: sizeof(Elemento));
```

d. Na primeira condição caso a linha seja a posição inicial **(-1) primeira linha**, posição referência:

- **I.** A cabeça passa a ser esse **nó** criado.
- **II.** A posição linha desse **nó** vai ser as posições criadas que o **for** percorreu.
- **III.** A posição coluna desse **nó** ainda é **-1** porque não criada as colunas ainda.
- **IV.** O ponteiro cima e o ponteiro baixo aponta para **NULO,** porque ainda não contém ninguém em cima nem embaixo, então recebe **NULO.**
- **V.** O ponteiro direito e esquerda apontam para o **nó** criado, fazendo com que fique linha circular .
- **VI.** E o ponteiro **aux** aponta para o **nó** criado.

```
if(l == -1){
    (*matriz) = no;
    no->lin =  l;
    no->col = -1;
    no->cima = NULL;
    no->baixo = NULL;
    no->dir = no;
    no->esq = no;
    aux = no;
```

e. Na segunda condição é se caso seja a última linha:

- **I.** Como na primeira condição, as variáveis de linha e coluna do **nó** criado recebe a linha alocada e a coluna continua ainda recebendo a posição **-1,** pois só temos as posições das linhas alocadas.
- **II.** O ponteiro cima passa apontar para o elemento **aux** no caso, esse elemento **aux** é o anterior em cima do **nó** atual criado, porque é a criação de linhas, então é uma por cima da outra.
- **III.** O ponteiro baixo passa aponta para a cabeça da matriz, fazendo com que fique de forma circular. Aqui o ponteiro baixo está apontando para a cabeça, porque a condição é se caso for a última linhas.
- **IV.** Como no tópico anterior, para que fique circular a cabeça passa apontar para o **nó,** que no caso é a última linha criada.
- **V.** O ponteiro **aux** passa apontar para baixo dele que é esse no criado, porque como foi informado antes, o ponteiro **aux** sempre será o elemento anterior ao **nó** inserido.
- **VI.** Para que também continue circular, a direita aponta para o **nó** criado e esquerda também aponta para esse **nó,** continuando assim circular.

```
}else if(l == lin-1){
    no->lin =  l;
    no->col = -1;
    no->cima = aux;
    no->baixo = *matriz;
    (*matriz)->cima = no;
    aux->baixo = no;
    no->esq = no;
    no->dir = no;
```

f. A terceira condição é caso não aconteça as duas primeiras, essa condição é alocando as linhas no meio da matriz:

- **I.** Como na primeira condição e na segunda, as variáveis de linha e coluna do **nó** criado recebe a linha alocada e a coluna continua ainda recebendo a posição **-1,** pois só temos as posições das linhas alocadas.
- **II.** O ponteiro cima passa apontar para o elemento anterior que é o **aux.**
- **III.** O ponteiro baixo do **nó** anterior antes desse ser criado passa apontar para o atual que está sendo criado.
- **IV.** O ponteiro **aux** passa apontar para o anterior antes desse ser criado, fazendo com que **aux** continue sempre sendo o anterior.
- **V.** E a direita e a esquerda do **nó** continua apontando para ele mesmo para que continue sendo circular.

```
} else {
    no->lin =  l;
    no->col = -1;
    no->cima = aux;
    no->cima->baixo = no;
    aux = no;
    no->dir = no;
    no->esq = no;
}
```

g. Alocando colunas

- **I.** Na alocação das colunas é igual a alocação de linhas, mas só com duas condições dentro, o limite de colunas alocadas será o que vai ser informado pelo usuário e recebido na função como parâmetro.
- **II.** Criamos e alocamos memória para um ponteiro do tipo **elemento** chamado **no** dentro do **for**. Esse ponteiro será as posições de referência da matriz, as posições negativas como foi informado acima.

```
//Alocando as colunas
aux = *matriz;
for(int c = 0; c < col; c++){
    Elemento *no = (Elemento*) malloc( Size: sizeof(Elemento));
```

h. A primeira condição é caso a coluna a ser inserida for a última:

- **I.** A esquerda desse **nó** passa apontar para **aux** que no caso é o elemento anterior, porque como informamos algumas vezes atrás, **aux** serve para identificar o elemento anterior ao **nó** que está sendo criado.
- **II.** Como essa condição é se a coluna for a última, o lado direito do **nó** criado passa à apontar para a cabeça da matriz, fazendo com que também fique circular nas colunas criadas.

III. E logo a cabeça do lado esquerdo passa à apontar para o **nó** que foi criado por último, fazendo com que também continue circular.

IV. O ponteiro **aux (anterior)** a sua direita passa à apontar para o **nó** que foi criado, fazendo com que **aux** continue sempre sendo anterior.

V. O ponteiro **nó** em cima passa à apontar para o próprio **nó** criado e ponteiro **nó** baixo também passa apontar para o próprio, fazendo com que ela fique circular.

VI. O ponteiro recebe como coordenada da sua coluna, a coluna que foi criada.

VII. E a linha continua sendo a posição **-1** que a posição de referência da matriz.

```
if(c == col-1){
    no->esq = aux;
    no->dir = *matriz;
    (*matriz)->esq = no;
    aux->dir = no;
    no->cima = no;
    no->baixo = no;
    no->col = c;
    no->lin = -1;
```

i. A segunda condição é caso a primeira não ocorra. Essa condição é alocando no meio:

I. Como na condição anterior o ponteiro recebe como coordenada da sua coluna, a coluna que foi criada e a linha recebe a posição de referência da matriz que é **-1.**

II. A esquerda do **nó** passa à apontar para anterior que é **aux.**

III. O ponteiro baixo do **nó** anterior antes desse ser criado passa apontar para o atual que está sendo criado .

IV. O ponteiro **aux** passa apontar para o anterior antes desse ser criado, fazendo com que **aux** continue sempre sendo o anterior.

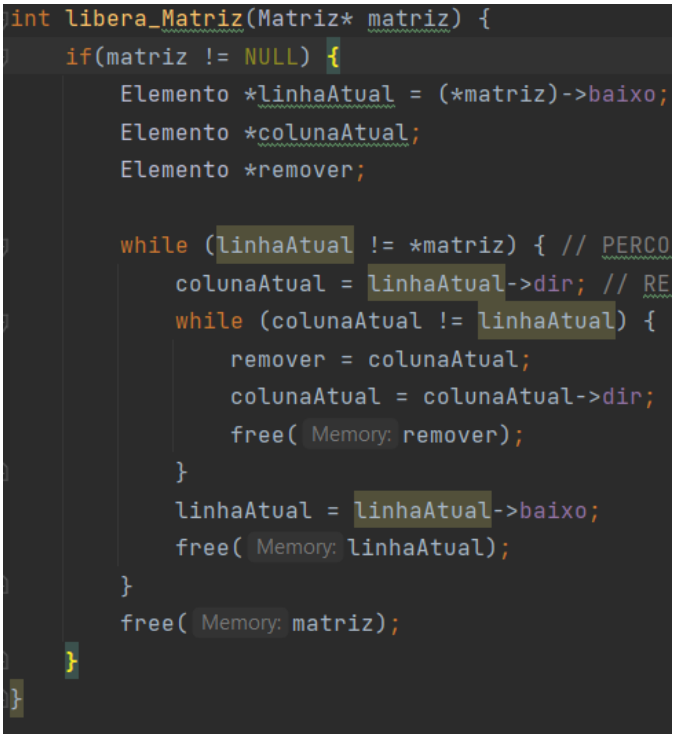
V. O ponteiro **nó** em cima passa à apontar para o próprio **nó** criado e ponteiro **nó** baixo também passa apontar para o próprio, fazendo com que ela fique circular.

```
} else {
    no->col = c;
    no->lin = -1;
    no->esq = aux;
    no->esq->dir = no;
    aux = no;
    no->baixo = no;
    no->cima = no;
}
```

j. Ao final a função retorna **matriz.**

5. **Função libera_Matriz do tipo inteiro**
    a. Recebe como parâmetro a matriz que vai ser desalocada.
    b. Faz o teste de criação da matriz, caso a matriz seja diferente de **NULO**, cria três ponteiros do tipo elemento, **linhaAtual, colunaAtual e remover**. O primeiro recebe a cabeça apontando para baixo, o segundo não é inicializado, porque ele vai receber **linhaAtual->dir** e o terceiro é o que vai removendo cada elemento da matriz**.**

        I. Inicializa um laço de repetição que vai percorrer linha por linha. Dentro desse laço **colunaAtual** recebe a **linhaAtual->dir.**
        II. Inicializa um segundo laço de repetição dentro do primeiro laço, porque esse laço vai percorrer cada coluna de **linhaAtual.** Em cada elemento percorrido, remover vai recebendo esse elemento e vai removendo, ao termino do laço, desaloca a cabeça da matriz.



6. **Função quantLinhas_matriz do tipo inteiro**
    a. Recebe como parâmetro a **matriz** criada.
    b. Cria o um ponteiro do tipo elemento chamado **no** que passa a apontar para a cabeça da matriz.
    c. Cria uma variável do tipo inteiro chamado **quant** e inicializa com **0.**
    d. Faz um laço de repetição onde o **nó** criado percorre as linhas até que encontre a cabeça da matriz novamente. Enquanto ele percorre as linhas vai incrementando na variável **quant**.
    e. Ao final retorna a variável criada **quant.**

```
int quantLinhas_matriz(Matriz* cabeca){
    Elemento *no = *cabeca;
    int quant = 0;
    while(no->baixo != *cabeca){
        quant++;
        no = no->baixo;
    }

    return quant;
}
```

7. **Função quantColunas_matriz do tipo inteiro**
    a. Recebe como parâmetro a **matriz** criada.
    b. Cria o um ponteiro do tipo elemento chamado **no** que passa a apontar para a cabeça da matriz.
    c. Cria uma variável do tipo inteiro chamado **quant** e inicializa com **0.**
    d. Faz um laço de repetição onde o **nó** criado percorre as colunas até que encontre a cabeça da matriz novamente. Enquanto ele percorre as linhas vai incrementando na variável **quant**.
    e. Ao final retorna a variável criada **quant.**

```
int quantColunas_matriz(Matriz* cabeca){
    Elemento *no = *cabeca;
    int quant = 0;
    while(no->dir != *cabeca){
        quant++;
        no = no->dir;
    }

    return quant;
}
```

8. **Função insere_matriz_byPos do tipo inteiro**
    a. Recebe como parâmetro a matriz, duas variáveis do tipo inteiro que serão as coordenadas informadas pelo usuário e os dados do aluno que também é informado pelo usuário.

```
int insere_matriz_byPos(Matriz* matriz, int l, int c, struct aluno al){
```

    b. Cria duas variáveis do tipo inteiro:
        I. **maxL,** recebe a função quantLinhas_matriz, onde ela vai armazenar a quantidades de linhas que a matriz contém.

II. **maxC,** recebe a função quantColunas_matriz, onde ela vai armazenar a quantidade de colunas que a matriz contém.

```
int maxL = quantLinhas_matriz( cabeca: matriz);
int maxC = quantColunas_matriz( cabeca: matriz);
```

c. Condição verificando se as posições informadas pelo usuário não ultrapassaram o limite ou foram menores que os limite da matriz.

```
if((l+1 > maxL || c+1 > maxC) || (l < 0 || c < 0)){
    return 0;
}
```

d. Cria três ponteiros do tipo elemento:

I. O primeiro ponteiro **no** passa apontar para a cabeça da matriz.
II. O segundo ponteiro **noNovo** é alocado dinamicamente.
III. O terceiro ponteiro **aux** incialmente aponta para nulo.

```
Elemento *no = *matriz; //DEFININDO O NÓ = CABEÇA DA MATRIZ
Elemento *noNovo = (Elemento*) malloc( Size: sizeof(Elemento)); //ALOCANDO MEMÓRIA PARA O NOVO NÓ
Elemento *aux = NULL; //APONTANDO AUX PARA NULL
```

e. Cria duas variáveis do tipo inteiro:

I. A primeira variável **linha** recebe a posição **l** informada pelo usuário.
II. A segunda variável **coluna** recebe a posição **c** informada pelo usuário.

```
int linha = l, coluna = c;
```

f. Faz um laço de repetição que vai obter o elemento anterior que ficará em acima do novo nó que está sendo inserido:

I. **Aux** vai percorrer a coluna através da posição que foi informado pelo usuário, ele vai receber o elemento através da função consulta_matriz_byPos.
II. Vai decrementando da linha até que chegue na posição anterior ao **nó novo** que está sendo inserido.

```
while(aux == NULL){// OBTENDO O ELEMENTO QUE FICARÁ ACIMA DO NOVO NÓ QUE ESTÁ SENDO INSERIDO
    linha--;
    aux = (Elemento*) consulta_matriz_byPos(matriz, l: linha, c: coluna); // BUSCANDO O ELEMENTO
}
```

k. O **nó novo** que está sendo inserido passa a apontar para o elemento que fica em cima dele, elemento esse encontrado anteriormente.

```
noNovo->cima = aux;
```

g. A variável **linha** criado anteriormente volta a receber o valor inserido pelo usuário referente a **l.** E **aux** volta a receber nulo para que agora possa verificar quem vai ficar abaixo do **novo nó.**

```
linha = l;
aux = NULL;
```

h. Faz outro laço de repetição, mas que agora vai encontrar o elemento que vai ficar abaixo do **nó novo:**

I. **Aux** vai percorrer a coluna através da posição que foi informado pelo usuário, ele vai receber o elemento através da função consulta_matriz_byPos.

II. Vai incrementando na linha que no caso vai ser inserida dentro da função de consulta.

III. Tem uma condição que se caso seja ao final da coluna, **aux** passa a receber o primeiro da coluna.

```
while(aux == NULL){ // OBTENDO O ELEMENTO QUE FICARÁ ABAIXO DO NÓ
    linha++;
    aux = (Elemento*) consulta_matriz_byPos(matriz, l: linha, c: coluna);
    if(aux == NULL && linha == maxL){
        aux = (Elemento*) consulta_matriz_byPos(matriz, l: -1, c: coluna);
    }
}
```

i. O **nó novo** que está sendo inserido passa a apontar para o elemento encontrado abaixo dele, recebe os dados do aluno, armazena a coordenada da linha e armazena a coordenada da coluna.

```
noNovo->baixo = aux;
noNovo->dados = al;
noNovo->lin = l; // A
noNovo->col = c; // A
```

j. **Aux** aponta para a cabeça da matriz e **linha** recebe a posição de referência.

```
aux = *matriz;
linha = -1; //
```

k. Faz outro laço de repetição que vai andar as posições para baixo da matriz (linhas):

I. Tem uma condição de parada, onde caso a **linha** seja igual a **l** que foi informado pelo usuário, o laço para de percorrer.

II. Vai incrementando na variável **linha.**

```
while(no->baixo != aux){
    if(linha == l){
        break;
    }
    no = no->baixo;
    linha++;
}
```

l. Agora **aux** passa a apontar para o **nó** que foi alocado para outra posição no laço anterior e **coluna** recebe a posição de referência da matriz.

```
aux = no;
coluna = -1;
```

m. Faz mais um laço de repetição que vai percorrer as colunas até a posição desejada pelo usuário:

I. Tem uma condição de parada, onde caso a **coluna** seja igual a **c** que foi informado pelo usuário, o laço para de percorrer.

II. Vai incrementando na variável **coluna.**

```
while(no->dir != aux && no->dir->col < noNovo->col){
    if(coluna == c){
        break;
    }
    no = no->dir;
    coluna++;
}
```

n. Faz três condições:

I. Caso o **nó** seja a própria cabeça da matriz

1. **Nó novo** aponta para **nó** a sua esquerda, que é a cabeça da matriz.
2. Também aponta para **nó** ao lado direito, para que continue como circular.
3. E **nó** a sua direita passa a apontar para **nó novo** que é o que está sendo inserido.
4. Tem uma condição caso **nó** seja o primeiro e o último, para que continue circular, a esquerda de **nó** passa a apontar para o **nó novo** que está sendo inserido.

```
if(no == aux) { // APONTAME
    noNovo->esq = no;
    noNovo->dir = no;
    no->dir = noNovo;
    if(no->esq == no){
        no->esq = noNovo;
    }
```

II. Caso as coordenadas informadas pelo usuário sejam existentes, ele vai sobrescrever os dados do aluno

1. **Nó** dados passa a receber os novos dados do **nó novo.**

```
}else if(noNovo->lin == no->lin && noNovo->col == no->col){
    no->dados = noNovo->dados;
```

III. Caso nenhuma das duas acima aconteça, e **nó** esteja no meio

1. **Nó novo** a sua esquerda passa a apontar para o **nó.**
2. **Nó novo** a sua direita passa a apontar para a direita que o **nó** aponta.
3. **Nó** a sua direita agora passa a apontar para **nó novo** que está sendo inserido.

```
} else { // APONTAMENTO PA
    noNovo->esq = no;
    noNovo->dir = no->dir;
    no->dir = noNovo;
}
```

IV. Ao final a função retorna 1, caso chegue ao final.

9. **Função consulta_matriz_byPos do tipo Elemento**
   a. Recebe como parâmetro a matriz e duas variáveis do tipo inteiro. Essas variáveis irão servir para consultar a posição informada pelo usuário que ele quer consultar.

```
Elemento* consulta_matriz_byPos(Matriz* matriz, int l, int c){
```

   b. Cria um ponteiro do tipo elemento chamado **no,** esse ponteiro passa a apontar para a cabeça da matriz e cria duas variáveis do tipo inteiro **maxL e maxC** que recebem a quantidade de linhas e a quantidade colunas que a matriz tem.

```
Elemento *no = *matriz;

int maxL = quantLinhas_matriz( cabeca: matriz);
int maxC = quantColunas_matriz( cabeca: matriz);
```

   c. Faz uma condição que retorna **nulo,** caso as coordenadas informadas excederem o limite da matriz.

```
if(l+1 > maxL || c+1 > maxC){
    return NULL;
}
```

   d. Cria duas variáveis do tipo inteiro **linha e coluna,** as duas recebem os valores de referência da matriz, as posições inicias.

```
int linha = -1, coluna = -1;
```

   e. Faz um laço de repetição que irá percorrer a linha com condição de que enquanto **no->baixo** não encontre a cabeça da matriz:
      I. Tem uma condição de parada, onde caso a **linha** seja igual a **l** que foi informado pelo usuário, o laço para de percorrer.
      II. Vai incrementando na variável **linha.**

```
while(no->baixo != *matriz){
    if(linha == l){
        break;
    }
    no = no->baixo;
    linha++;
}
```

f. Cria-se um ponteiro do tipo elemento chamado **aux** que passa a ser o primeiro elemento da linha e faz um laço de repetição que irá percorrer a coluna enquanto **no** a sua direita não encontre a cabeça que é **aux** nesse laço:

I. Tem uma condição de parada, onde caso a **coluna** seja igual a **c** que foi informado pelo usuário, o laço para de percorrer.

II. Vai incrementando na variável **coluna.**

```
Elemento *aux = no; //
while(no->dir != aux){
    if(coluna == c){
        break;
    }
    coluna++;
    no = no->dir;
}
```

g. Ao final, depois de chegar na posição, cria-se uma condição que irá retornar o **nó** caso o valor da coluna informada pelo usuário seja igual a coluna que está armazenada no elemento que está naquela linha. E retorna **NULO** caso não tenha aquela coluna.

```
if(c == no->col){
    return no;
} else {
    return NULL;
}
```

10. Função consulta_matriz_matricula do tipo Elemento

a. Recebe como parâmetro a matriz e uma variável do tipo inteiro, essa variável é o valor da matricula que o usuário quer consultar dentro da matriz.

b. Faz o teste de criação da matriz, caso a matriz não foi criada retorna **NULO,** caso seja bem sucedido a criação continua a função. Cria dois ponteiros do tipo elemento, **linhaAtual e colunaAtual**. O primeiro recebe a cabeça apontando para baixo e o segundo não é inicializado, porque ele vai receber **linhaAtual->dir.**

I. Inicializa um laço de repetição que vai percorrer linha por linha. Dentro desse laço **colunaAtual** recebe a **linhaAtual->dir.**

II. Inicializa um segundo laço de repetição dentro do primeiro laço, porque esse laço vai percorrer cada coluna de **linhaAtual.** Caso encontre a matricula que o usuário procura, a função retorna o **colunaAtual** que é o elemento que se encontra os dados daquela matricula informada pelo usuário.

III. Caso não encontre, a função retorna **NULO.**

```c
Elemento* consulta_matriz_Matricula(Matriz* matriz, int matricula){
    if(matriz == NULL) // TESTE DE CRIAÇÃO DA MATRIZ
        return NULL;
    Elemento* linhaAtual = (*matriz)->baixo; // RECEBE A CABEÇA APOT
    Elemento* colunaAtual;

    while (linhaAtual != *matriz) { // PERCORRE LINHA POR LINHA
        colunaAtual = linhaAtual->dir; // RECEBE A LINHA PARA QUE PO
        while (colunaAtual != linhaAtual) { // PERCORRE TODA A LINHA
            if(colunaAtual->dados.matricula == matricula) { // CASO
                return colunaAtual;
            }
            colunaAtual = colunaAtual->dir;
        }
        linhaAtual = linhaAtual->baixo;
    }
    return NULL;
}
```

11. Função imprime_matriz do tipo inteiro

- **a.** Recebe como parâmetro a matriz a ser impressa.
- **b.** Ela é igualmente elaborada de acordo a função anterior **consulta_matriz_Matricula** a diferença é que agora não tem condição nenhuma dentro do laço de repetição, porque enquanto ele percorre as linhas e as colunas da matriz, a função vai imprimindo cada informação que tem dentro da matriz armazenado.
- **c.** Retorna 0, caso a matriz esteja vazia

```c
int imprime_matriz(Matriz* matriz) {
    if(matriz == NULL) // TESTE DE CRIAÇÃO DA MATRIZ
        return 0;
    Elemento* linhaAtual = (*matriz)->baixo;
    Elemento* colunaAtual;

    while (linhaAtual != *matriz) {
        colunaAtual = linhaAtual->dir;
        while (colunaAtual != linhaAtual) {
            printf( format: "\n\tCoordenada [%d][%d]", linhaAtual->lin, colunaAtual->col);
            printf( format: "\n\tMatricula: %d\n", colunaAtual->dados.matricula);
            printf( format: "\tNome: %s\n", colunaAtual->dados.nome);
            printf( format: "\tNotas: %.2f %.2f %.2f\n", colunaAtual->dados.n1,
                    colunaAtual->dados.n2,
                    colunaAtual->dados.n3);
            printf( format: "\t--------------------------------\n");
            colunaAtual = colunaAtual->dir;
        }
        linhaAtual = linhaAtual->baixo;
    }
    return 0;
}
```

12. Função imprime_vizinhos do tipo inteiro

- **a.** Recebe como parâmetro a matriz e duas variáveis do tipo inteiro que será as coordenadas do elemento que quer imprimir os seus vizinhos.
- **b.** A elaboração dessa função é a mesma lógica das duas anteriores, pois faz o teste de criação, depois cria dois ponteiros do tipo elemento que vai percorrer a matriz e tem dois lações de repetição igualmente criado nas duas funções anteriores. A diferença é que dentro do segundo laço verifica se as coordenadas pelo usuário foram encontradas.

c. Caso seja encontrado, cria várias condições para poder imprimir os vizinhos, onde verifica se a posição do vizinho é maior do que a posição de referência da matriz, que é a posição **-1.** Caso seja verdadeiro, imprime os dados de cada vizinho.

d. Ao final retorna 0 se caso não encontrar a posição na matriz.

```
int imprime_vizinhos(Matriz* matriz, int lin, int col) {
    if(matriz == NULL)
        return 0;
    Elemento* linhaAtual = (*matriz)->baixo;
    Elemento* colunaAtual;

    while (linhaAtual != *matriz) {
        colunaAtual = linhaAtual->dir;
        while (colunaAtual != linhaAtual) {
            if(colunaAtual->lin == lin && colunaAtual->col == col) {
                if(colunaAtual->esq->lin > -1 && colunaAtual->esq->col > -1) {
                    printf( format: "\n\tVizinho da esquerda: ");
                    imprime_um_elemento( matriz: &colunaAtual->esq);
                } else { printf( format: "\n\tNão tem vizinho há esquerda\n"); }

                if(colunaAtual->cima->lin > -1 && colunaAtual->cima->col > -1) {
                    printf( format: "\n\tVizinho de cima: ");
                    imprime_um_elemento( matriz: &colunaAtual->cima);
                } else { printf( format: "\n\tNão tem vizinho há cima\n"); }

                if(colunaAtual->dir->lin > -1 && colunaAtual->dir->col > -1) {
                    printf( format: "\n\tVizinho da direita: ");
                    imprime_um_elemento( matriz: &colunaAtual->dir);
                } else { printf( format: "\n\tNão tem vizinho há direita\n"); }

                if(colunaAtual->baixo->lin > -1 && colunaAtual->baixo->col > -1) {
                    printf( format: "\n\tVizinho da baixo: ");
                    imprime_um_elemento( matriz: &colunaAtual->baixo);
                } else { printf( format: "\n\tNão tem vizinho há baixo\n"); }

            }
            colunaAtual = colunaAtual->dir;
        }
        linhaAtual = linhaAtual->baixo;
    }
    return 0;
}
```

13. Função imprime_um_elemento do tipo void

a. Recebe como parâmetro a matriz e só foi criada para poder fazer a impressão dos dados de cada elemento da matriz, por isso que ela é do tipo void, porque não retorna nada.

```
void imprime_um_elemento(Matriz* matriz) {
    Elemento* aux = *matriz;
    printf( format: "\n\tCoordenada [%d][%d]", aux->lin, aux->col);
    printf( format: "\n\tMatricula: %d\n", aux->dados.matricula);
    printf( format: "\tNome: %s\n", aux->dados.nome);
    printf( format: "\tNotas: %.2f %.2f %.2f\n", aux->dados.n1,
            aux->dados.n2,
            aux->dados.n3);
    printf( format: "\t-----------------------------\n");
}
```

Arquivo matriz.h

Todas as funções e structs criadas no arquivo matriz.c.

```c
struct aluno{
    int matricula;
    char nome[30];
    float n1,n2,n3;
};

struct elemento{
    struct aluno dados;
    struct elemento *dir;
    struct elemento *esq;
    struct elemento *cima;
    struct elemento *baixo;
    int lin, col;
};
typedef struct elemento Elemento;
typedef struct elemento* Matriz;

void inserir_dados(struct aluno *al);
Matriz* cria_matriz(int lin, int col);
int libera_Matriz(Matriz* matriz);
int matriz_vazia(Matriz* matriz);
int quantLinhas_matriz(Matriz* cabeca);
int quantColunas_matriz(Matriz* cabeca);
int insere_matriz_byPos(Matriz* matriz, int l, int c, struct aluno al);
Elemento* consulta_matriz_byPos(Matriz* matriz, int l, int c);
Elemento* consulta_matriz_Matricula(Matriz* matriz, int matricula);
int imprime_matriz(Matriz* matriz);
int imprime_vizinhos(Matriz* matriz, int lin, int col);
void imprime_um_elemento(Matriz* matriz);
```

Dificuldades

No início, na criação da estrutura da matriz e na alocação dos ponteiros e dos seus nós fizeram com que tivéssemos um pouco de dificuldade, pois o mais importante é ter que iniciar a estrutura, porque ela é primeiro passo e o passo mais importante, porque vamos armazenar os elementos nessa estrutura.

Outra dificuldade que levou um tempinho bom, foi na criação da função de inserir os nós dentro da matriz, porque como estávamos mexendo com uma matriz bi dimensional, tínhamos que controlar 4 ponteiros dentro da matriz.

Enquanto estávamos fazendo essas funções criamos vários **printf** para que pudéssemos enxergar os erros e os locais que o código não estava compilando.

Desafios

Estávamos destinados a fazer a matriz da segunda maneira pedida pelo professor, porque nos sentimos desafiados, pois era uma matriz que chamou a atenção pela complexidade e pelo tamanho do desafio que ela iria trazer para gente e também o aprendizado que seria maior ainda.

Na criação das duas primeiras funções serviram de muito aprendizado, porque fez com que entendêssemos mais ainda como manipular e controlar os ponteiros de uma memória. Entendemos como realmente é controlar a memória de um dispositivo.