

Matrizes

Estrutura de Dados

Universidade Federal de Juiz de Fora
Departamento de Ciência da Computação

Conteúdo

- ▶ Introdução vetores e matrizes
- ▶ Representações de matrizes
- ▶ TADs
 - ▶ TAD Vetor
 - ▶ TAD Vetor Flexível
 - ▶ TAD Matriz
 - ▶ TAD Matriz - Representação Linear
 - ▶ TAD Matriz Flexível - Representação Linear
- ▶ Matrizes especiais
 - ▶ Diagonal
 - ▶ Triangular inferior (ou superior)
 - ▶ Simétrica
 - ▶ Anti-simétrica

Introdução

- ▶ Em computação, as matrizes são representadas por meio de estruturas de dados conhecidas como **vetores** ou **arrays**, onde cada posição/valor pode ser referenciada por um ou mais índices (dependendo da quantidade de dimensões da matriz);
- ▶ Enquanto na matemática uma matriz possui sempre duas dimensões, na computação chama-se qualquer vetor de matriz, podendo possuir uma ou mais dimensões.
 - ▶ Matrizes unidimensionais (ou vetores)
 - ▶ Matrizes bidimensionais
 - ▶ Matrizes N-dimensionais

Introdução

- ▶ Quanto de memória ocupa uma matriz 5000×5000 de valores reais?
 - ▶ Um valor real (`float`) ocupa 4 bytes;
 - ▶ Então, essa matriz ocupa aproximadamente 100MB.
- ▶ E se somente alguns poucos elementos dessa matriz fossem diferentes de zero?
- ▶ Seria possível reduzir a sua representação de forma que ela passasse a ocupar menos espaço de memória?
- ▶ Veremos como representar essas matrizes de forma compacta:
 - ▶ Matrizes Diagonais;
 - ▶ Matrizes Triangulares;
 - ▶ Matrizes Esparsas;
 - ▶ Etc...

TADs

- ▶ TAD Vetor (matriz unidimensional);
- ▶ TAD Vetor Flexível;
- ▶ TAD Matriz bidimensional.

TAD Vetor

- Seja o TAD Vetor de n elementos reais, representado na classe em C++ a seguir:

```
class Vetor
{
public:
    Vetor(int tam);
    ~Vetor();
    float get(int indice);
    void set(int indice, float valor);

private:
    int n;           // tamanho do vetor
    float *vet;      // array que armazena n floats

    bool verifica(int indice);
};
```

TAD Vetor

- ▶ Implementar o TAD Vetor anterior para um vetor de reais, de acordo com as seguintes especificações:
 - ▶ O tamanho do vetor deve ser definido em tempo de execução. Assim, o construtor deve alocar memória de acordo com o tamanho especificado pelo seu parâmetro `tam`;
 - ▶ Ao acessar ou modificar um elemento do vetor, verificar a validade do índice;
- ▶ Em seguida, desenvolver um programa (implementar `main()`) para testar o TAD Vetor, usando um vetor de 60 elementos reais.

TAD Vetor

► Construtor e destrutor

```
Vetor::Vetor(int tam) {  
    // inicializa a variavel interna n e  
    // aloca memoria para o vetor vet  
    n = 0;  
    if (n > 0)  
        n = tam;  
    vet = new float[n];  
  
    // opcional: inicializar vet com zeros  
    for(int i=0; i<n; i++)  
        vet[i] = 0.0;  
}  
  
Vetor::~~Vetor() {  
    // desaloca a memoria alocada no construtor  
    delete [] vet;  
}
```


TAD Vetor

- ▶ A função privada `verifica()` analisa a validade do índice do vetor.
- ▶ O índice pode assumir um valor entre 0 e $n-1$ (padrão C/C++):

```
bool Vetor::verifica(int indice)
{
    // verifica validade de indice
    if(indice >= 0 && indice < n)
        return true;
    else
        return false;
};
```

TAD Vetor

```
float Vetor::get(int indice)
{
    if ( verifica(indice) )
        return vet[indice];
    else {
        cout << "Indice invalido: get" << endl;
        exit(1) ; // finaliza o programa
    }
}

void Vetor::set(int indice, float valor)
{
    if ( verifica(indice) )
        //armazena valor na posicao indice de vet
        vet[indice] = valor;
    else
        cout << "Indice invalido: set" << endl;
}
```

TAD Vetor

- Programa que usa o TAD Vetor criando um vetor de 60 elementos reais:

```
#include "Vetor.h"

int main() {
    int tam = 60;
    Vetor v(tam);                      // aloca vet[60]

    for(int i=0; i<tam; i++)           // armazena seq
        v.set(i,i+1);                  // de 1 a 60

    for(int i=0; i<tam; i++)
    {
        float val = v.get(i) ;
        cout << val << endl;
    }
    return 0;
}
```

Vetor com índices flexíveis

- ▶ Sabendo que na linguagem C/C++, o índice de um vetor de tamanho n é um valor inteiro entre 0 e $n - 1$, pede-se:
- ▶ Desenvolver um **TAD** para possibilite criar vetores cujos índices podem assumir seus valores em intervalos inteiros e quaisquer, como por exemplo entre -10 e 45 .
- ▶ Desenvolver uma **aplicação** para testar o TAD anterior, criando um vetor de 60 elementos reais numerados de -29 (limite inferior) a 30 (limite superior).
- ▶ Os valores limites (inferior e superior) do intervalo do índice devem ser definidos na aplicação, em **tempo de execução**. Assim, o construtor deve alocar memória dinamicamente para o vetor, de acordo com a definição desses limites.
- ▶ Verificar a validade do índice, quando necessário.

Vetor com índices flexíveis

Programa

```
VetorFlex v(-5, 6);
```

C

-5 -4 -3 -2 -1 0 1 2 3 4 5 6

F

TAD VetorFlex

C/C++

índices

0 1 2 3 4 5 6 7 8 9 10 11

valores

2	1	3	9	6	-5	-3	25
---	---	---	---	---	----	-----	-----	-----	-----	----	----

TAD VetorFlex

```
class VetorFlex
{
private:
    int n;          // tamanho do vetor
    float *vet;     // array que armazena n floats
    int c, f        // c: limite inferior do indice
                    // f: limite superior do indice

    int detInd(int indice); // operador privado

public:
    VetorFlex(int a, int b);
    ~VetorFlex();
    float get(int indice);
    void set(int indice, float valor);
};
```

TAD VetorFlex

```
// construtor
VetorFlex::VetorFlex(int cc, int ff)
{
    c = cc;
    f = ff;
    n = f - c + 1;
    vet = new double[n];
}

// destrutor
VetorFlex::~~VetorFlex()
{
    delete [] vet;
}
```

TAD VetorFlex

- ▶ A função `detInd(int i)` é privada, isto é, só pode ser utilizada dentro da classe `VetorFlex`.
- ▶ A função `detInd(int i)` verifica a validade do índice de `vet`, isto é, se $c \leq i \leq f$.
- ▶ Se for válido, retorna o valor do índice de acordo com o padrão C/C++, isto é, o valor correspondente a índice dentro do intervalo de 0 a $n-1$.
- ▶ Senão (se for inválido), retorna -1 .

TAD VetorFlex

```
int VetorFlex::detInd(int indice)
{
    if(c <= indice && indice <= f)
        return (indice - c);
    else
        return -1;
};
```

- ▶ Até então só foram utilizados atributos (variáveis membro) como membros privados.
- ▶ Por que criar uma função privada em uma classe?
 - ▶ Para realizar alguma tarefa que só é de interesse da classe.
 - ▶ Nesse exemplo, o usuário do TAD `VetorFlex` não precisa saber se essa verificação é feita ou como ela é feita antes de acessar ou modificar um elemento do vetor.

TAD VetorFlex

```
float VetorFlex::get(int indice) {
    int i = detInd(indice);
    if(i != -1)
        return vet[i];
    else {
        cout << "Indice invalido: get\n";
        exit(1);
    }
}

void VetorFlex::set(int indice, double val) {
    int i = detInd(indice);
    if(detInd(indice) != -1)
        vet[i] = val;
    else {
        cout << "Indice invalido: set\n";
        exit(1);
    }
}
```

Vetor com índices flexíveis

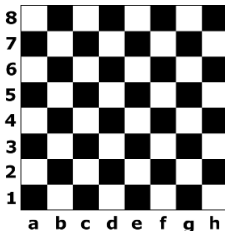
Aplicação

```
#include "VetorFlex.h"

int main()
{
    int cc = -29; int ff = 30;
    VetorFlex v(cc, ff);
    for(int i = cc; i <= ff; i++)
    { // valores no intervalo 1...60
        double val = i - cc + 1;
        v.set(i, val);
    }
    for(int i = cc; i <= ff; i++)
    {
        double val = v.get(i);
        cout << val << endl;
    }
    return 0;
}
```

Matrizes

- ▶ Matrizes com mais de uma dimensão.
- ▶ As principais operações são de atribuição e consulta;
- ▶ O projeto do TAD Matriz é idêntico ao do TAD Vetor, devendo-se utilizar tantos índices quantas forem as dimensões da matriz considerada;
- ▶ Exemplos de aplicações:

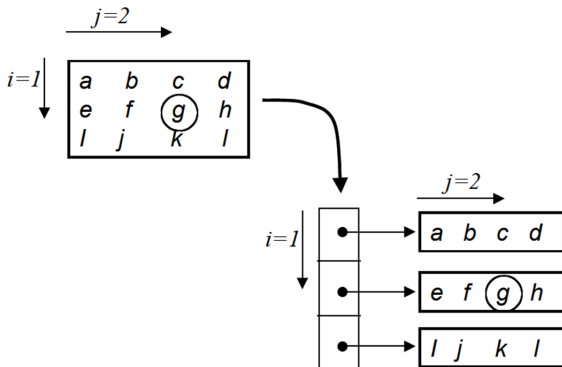


Matrizes

- ▶ Serão estudadas 2 formas diferentes de representar matrizes:
 1. Ponteiro de ponteiro (ou vetor de vetores): TAD Matriz2D
 2. Representação linear: TAD Matriz

Matrizes

- ▶ TAD Matriz2D
- ▶ Representação por ponteiro de ponteiro (ou vetor de vetores).
- ▶ Esquema:



TAD Matriz2D

- Classe para o TAD Matriz de 2 dimensões:

```
class Matriz2D
{
    public:
        Matriz2D(int nml, int nnc);
        ~Matriz2D();

        float get(int i, int j);
        void set(int i, int j, float valor);

    private:
        int nl;           // numero de linhas
        int nc;           // numero de colunas
        float **mat;      // array com nl*nc floats

        bool verifica(int i, int j);
};
```

TAD Matriz2D

► Construtor e destrutor.

```
Matriz2D::Matriz2D(int nml, int nnc)
{
    nl = nml;
    nc = nnc;
    // aloca o vetor de vetores
    mat = new float*[nl];
    // aloca cada um dos vetores (cada linha)
    for(int i = 0; i < nl; i++)
        mat[i] = new float[nc];
}

Matriz2D::~~Matriz2D()
{
    // desaloca a memoria alocada no construtor
    for(int i = 0; i < nl; i++)
        delete [] mat[i];
    delete [] mat;
}
```


TAD Matriz2D

- ▶ A função privada `verifica()` analisa a validade dos índices i e j seguindo o padrão C/C++;
- ▶ O índice i pode assumir valor entre 0 e $nl - 1$;
- ▶ e índice j entre 0 e $nc - 1$.

```
bool Matriz2D::verifica(int i, int j)
{
    if(i >= 0 && i < nl && j >= 0 && j < nc)
        return true;
    else
        return false; // indice invalido
};
```

TAD Matriz2D

```
float Matriz2D::get(int i, int j)
{
    if ( verifica(i, j) )
        return mat[i][j];
    else {
        cout << "Erro: indice invalido" << endl;
        exit(1);
    }
}

void Matriz2D::set(int i, int j, float valor)
{
    if ( verifica(i, j) )
        mat[i][j] = valor;
    else {
        cout << "Erro: indice invalido" << endl;
        exit(1);
    }
}
```

TAD Matriz2D

- ▶ Desenvolver um programa que usa o TAD Matriz2D para:
 - a) Ler 25 valores reais e gerar uma matriz 5x5, linha por linha.
 - b) Imprimir a 4ª coluna da matriz.
 - c) Ler índice de linha, índice de coluna e um valor real e alterar a posição correspondente da matriz.
 - d) Determinar e imprimir a transposta da matriz.
 - e) Determinar o maior valor da diagonal secundária da matriz.

Aplicação com o TAD Matriz2D

```
#include "Matriz2D.h"

int main() {
    Matriz2D mat(5,5);

    for(int i=0; i<5; i++)
        for(int j=0; j<5; j++) {
            float val;
            cin >> val;
            mat.set(i,j,val);
        }

    for(int i=0; i<5; i++) {
        cout << mat.get(i,3) << endl;
    }

    // etc ...

    return 0;
}
```

Matriz

- ▶ No TAD Matriz2D, apresentado anteriormente, usamos um `array bidimensional float **mat` para representar a matriz e o acesso era realizado com a seguinte operação: `mat[i][j]`.
- ▶ Seja a matriz A (3×4) de inteiros

$$A = \begin{bmatrix} 5 & 9 & 6 & 7 \\ -3 & 2 & 0 & 4 \\ 1 & 8 & 3 & -5 \end{bmatrix}$$

- ▶ Também pode-se armazenar a matriz A na memória usando um único array unidimensional `float *mat`, assim todos os elementos de A serão armazenados em posições consecutivas de memória a partir de um endereço base.
- ▶ Essa forma é conhecida como `representação linear`.

Matriz

Representação linear

- ▶ Para o exemplo anterior, considere que
 - ▶ o índice da linha L varia de 0 a 2;
 - ▶ o índice da coluna C varia de 0 a 3;
 - ▶ a matriz seja percorrida **linha por linha** para ser armazenada em memória.
- ▶ Visão da representação linear na memória

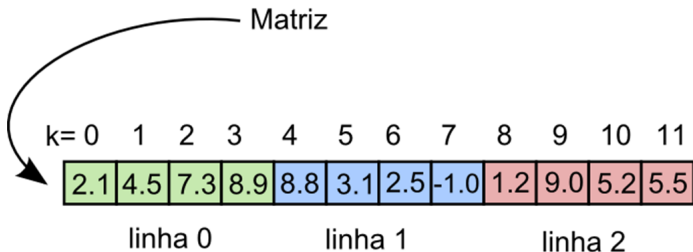
		linha 0				linha 1				linha 2				
Endereços		b+0	b+1	b+2	b+3	b+4	b+5	b+6	b+7	b+8	b+9	b+10	b+11	
Valor	...	5	9	6	7	-3	2	0	4	1	8	3	-5	...
L	...	0	0	0	0	1	1	1	1	2	2	2	2	...
C	...	0	1	2	3	0	1	2	3	0	1	2	3	...

Matriz

Representação linear

	j=0	j=1	j=2	j=3
i=0	2.1	4.5	7.3	8.9
i=1	8.8	3.1	2.5	-1.0
i=2	1.2	9.0	5.2	5.5

M=3 linhas
N=4 colunas



$$k = i * N + j$$

Matriz

Representação linear

- ▶ Desta forma, a matriz bidimensional $A(3 \times 4)$ é representada linearmente por um vetor V do tipo:

```
float V[12];
```

- ▶ Isto é, V é a representação linear de A .
- ▶ Para acessar um elemento de A em V , é necessário relacionar o índice k de V com os índices i e j da matriz A .
- ▶ Isso é feito através da seguinte relação:

$$k = 4i + j$$

Matriz

Representação linear

- ▶ Assim, dados:
 - ▶ o vetor V (representação linear da matriz A)
 - ▶ um par de índices válidos i e j de A
- ▶ Para obter o valor do elemento $A[i,j]$, deve-se acessar o elemento $V[k]$, sendo

$$k = 4i + j$$

- ▶ Notar que a matriz A está armazenada na memória através de sua **representação linear** V .
- ▶ O que se deseja é consultar o valor de $A[i, j]$ a partir de V .

Matriz

Representação linear

- ▶ Representação Linear de Matrizes
- ▶ Desenvolver o TAD Matriz para uma matriz $m \times n$ de elementos reais.
- ▶ **Observações:**
 - ▶ A representação interna da matriz deve ser **linear**.
 - ▶ O número de linhas m e o de colunas n devem ser definidos em tempo de execução.
 - ▶ Assim, o construtor deve alocar memória de acordo com o tamanho da matriz especificado pelos parâmetros m e n ;
 - ▶ Verificar a validade dos índices;
 - ▶ Desenvolver um programa para testar o TAD Matriz, que usa uma matriz 7×11 de elementos reais.

TAD Matriz

```
class MatrizLin
{
public:
    MatrizLin(int m, int n);
    ~MatrizLin();

    float get(int i, int j);
    void set(int i, int j, float val);

private:
    int nl, nc;    // numero de linhas e colunas
    float *vet;    // vetor de tamanho nl*nc

    int detInd(int linha, int coluna);
};
```

TAD Matriz - Representação linear

► Construtor e destrutor

```
MatrizLin::MatrizLin(int m, int n)
{
    // inicializa as variaveis internas
    // e aloca memoria de vet (representacao linear)
    nl = m;
    nc = n;
    vet = new float[nl*nc];
}

MatrizLin::~~MatrizLin()
{
    // desaloca a memoria alocada no construtor
    delete [] vet;
}
```

TAD Matriz - Representação linear

- ▶ A função privada `detInd` converte os índices linha e coluna da matriz no índice k do vetor `vet`.
- ▶ Além disso, verifica validade de linha e coluna. Todos os índices (linha, coluna e k) variam a partir de 0 (padrão C/C++):

```
int MatrizLin::detInd(int i, int j)
{
    if(i >= 0 && i < nl && j >= 0 && j < nc)
        return i*nc + j;
    else
        return -1; // indice invalido
};
```

TAD Matriz - Representação linear

```
float MatrizLin::get(int i, int j)
{
    int k = detInd(i, j);
    if(k != -1)
        return vet[k];
    else {
        cout << "Erro: get" << endl;
        exit(1);
    }
}

void MatrizLin::set(int i, int j, float valor)
{
    int k = detInd(i, j);
    if(k != -1)
        vet[k] = valor;
    else {
        cout << "Erro: set" << endl; exit(1);
    }
}
```

Aplicação com o TAD Matriz

```
#include "MatrizLin.h"

int main() {
    int m = 7, n = 11;
    MatrizLin mat(m,n);

    for(int i=0; i<m; i++)
        for(int j=0; j<n; j++)
        {
            float val = j + n*i;
            mat.set(i,j,val);
        }

    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++)
        {
            float val = mat.get(i,j) ;
            cout << val << "\t";
        }
        cout << endl;
    }

    return 0;
}
```

Matriz - Representação linear

- ▶ Vamos considerar agora uma situação mais geral na qual os elementos de uma matriz possuem índices quaisquer (similar ao TAD `VetorFlex`).
- ▶ Seja A uma matriz com $m \times n$ de elementos de um tipo qualquer. Os índices das linhas L e colunas C são:
 - ▶ $L = c_1 \dots f_1$
 - ▶ $C = c_2 \dots f_2$
- ▶ A matriz A possui um total de:
 - ▶ linhas: $m = f_1 - c_1 + 1$
 - ▶ colunas: $n = f_2 - c_2 + 1$
- ▶ O índice k da representação linear V que corresponde ao elemento $A[L, C]$ é dado por

$$I = (C - c_2) + n(L - c_1).$$

TAD Matriz Flexível - Representação linear

- ▶ Representação Linear de Matrizes
- ▶ Desenvolver o TAD MatrizFlex para uma matriz $m \times n$ de elementos reais com índices quaisquer.
- ▶ **Observações:**
 - ▶ A representação interna da matriz deve ser linear;
 - ▶ Os limites dos intervalos dos índices de linha e de coluna devem ser arbitrários e definidos em tempo de execução. Assim, o construtor deve alocar memória de acordo com esses limites;
 - ▶ Verificar a validade dos índices, quando necessário;
 - ▶ Desenvolver um programa que use o TAD MatrizFlex e crie uma matriz de elementos reais com os intervalos de linha = -2..7 e de coluna = 0..5.

TAD MatrizFlex

```
class MatrizFlex
{
public:
    MatrizFlex(int cc1, int ff1, int cc2, int ff2);
    ~MatrizFlex();

    float get(int i, int j);
    void set(int i, int j, float val);

private:
    float *vet; // representacao linear da matriz
    int m, n;   // numero de linhas e colunas
    int c1;     // limite inicial da linha
    int c2;     // limite inicial da coluna
    int f1;     // limite final da linha
    int f2;     // limite final da coluna
    int detInd(int linha, int coluna);
};
```

TAD MatrizFlex

► Construtor e destrutor

```
MatrizFlex::MatrizFlex(int cc1, int ff1,  
                       int cc2, int ff2)  
{  
    // inicializa os limites  
    c1 = cc1;  
    c2 = cc2;  
    f1 = ff1;  
    f2 = ff2;  
    m = f1 - c1 + 1; // calcula o numero de linhas  
    n = f2 - c2 + 1; // calcula o numero de colunas  
    vet = new float[m*n] ;  
}  
  
MatrizFlex::~~MatrizFlex()  
{  
    delete [] vet;  
}
```

TAD MatrizFlex

- A função privada `detInd` converte os índices linha e coluna da matriz no índice k do vetor `vet`. Além disso, verifica validade de linha e coluna.

```
int MatrizFlex::detInd(int i, int j)
{
    if(i >= c1 && i <= f1 && j >= c2 && j <= f2)
        return (j - c2) + n*(i - c1);
    else
        return -1;
};
```

TAD MatrizFlex

```
float MatrizFlex::get(int i, int j)
{
    int k = detInd(i, j) ;
    if(k != -1)
        return vet[k];
    else
        cout << "Indice invalido: get" << endl;
    exit(1);
}

void MatrizFlex::set(int i, int j, float val)
{
    int k = detInd(i, j) ;
    if(k != -1)
        vet[k] = valor;
    else
        cout << "Indice invalido: set" << endl;
    exit(1);
}
```

TAD MatrizFlex

```
#include "MatrizFlex.h"

int main(){
    int c1 = -2, f1 = 7;
    int c2 = 0, f2 = 5;

    MatrizFlex mat(c1,f1,c2,f2);

    // atribui valores a matriz mat
    for(int i=c1; i<=f1; i++){
        for(int j=c2; j<=f2; j++){
            float val = (f2-c2+1)*(i-c1) + j - c2; // 0...(n*m-1)
            mat.get(i,j,val);
        }

    // imprime a matriz mat
    for(int i=c1; i<=f1; i++) {
        for(int j=c2; j<=f2; j++) {
            float val = mat.set(i,j);
            cout << val << "\t";
        }
        cout << endl;
    }
    return 0;
}
```

Matrizes Especiais

Matrizes Especiais

- ▶ Matriz Diagonal
- ▶ **Matriz Triangular Inferior**
- ▶ Matriz Triangular Superior
- ▶ Matriz Simétrica
- ▶ Matriz Anti-Simétrica
- ▶ Matriz Tridiagonal

Matrizes Especiais

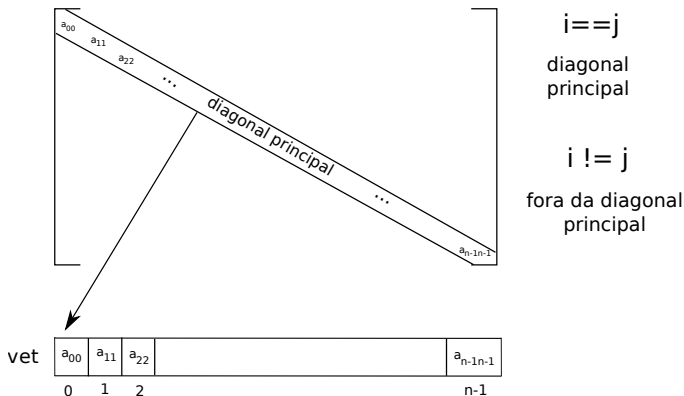
- ▶ Serão apresentados diferentes tipos de matrizes especiais.
- ▶ A **representação linear** será usada para representar cada tipo de matriz.
- ▶ Um TAD fundamental será desenvolvido com as seguintes operações:
 - ▶ Construtor
 - ▶ Destrutor
 - ▶ Função para consultar (`get`) um elemento A_{ij}
 - ▶ Função para alterar o valor de um elemento A_{ij}

Introdução

Representação Linear

- ▶ A representação linear (ou vetorial) é sempre usada pelo computador para o armazenamento de uma matriz com duas ou mais dimensões.
- ▶ Para usar uma representação linear V da matriz A , deve-se realizar três tarefas:
 1. Definir quais e de que forma os elementos de A serão alocados em V ;
 2. Dimensionar V ;
 3. Relacionar os índices $[i, j]$ de A com o índice k de V .
- ▶ Para atender a restrição da linguagem C/C++, será considerada a variação dos índices a partir de 0 (zero) nos casos que se seguem.

Matriz Diagonal



- Implemente o TAD MatrizDiagonal como exercício.

Matriz Triangular

- ▶ Matriz triangular inferior

$$l_{ij} = 0, \quad \forall i < j, \quad \text{Exemplo: } \mathbf{L} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix}$$

- ▶ Matriz triangular superior

$$u_{ij} = 0, \quad \forall i > j, \quad \text{Exemplo: } \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Matriz Triangular Inferior

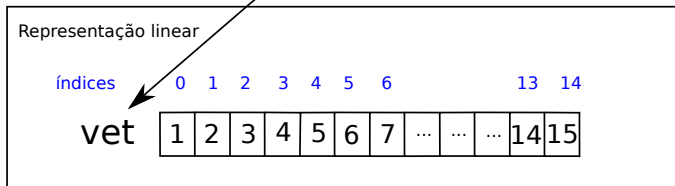
- ▶ Vamos estudar como representar uma matriz triangular inferior de $n \times n$ elementos reais.
- ▶ O procedimento é similar para uma matriz triangular superior.

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ \vdots & & & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix}$$

- ▶ Questões importantes:
 1. Quantos elementos armazenar?
 2. Representação linear: como armazenar os elementos?
 3. Como acessar/modificar um elemento?

Matriz Triangular Inferior

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 7 & 8 & 9 & 10 & 0 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$$



Matriz Triangular Inferior

- ▶ Quantos elementos a matriz triangular inferior L de dimensão n possui?

$$L = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \dots & 0 \\ \vdots & & & \ddots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix}$$

- ▶ Linha por linha:
 - ▶ Na linha 1 temos 1 elemento
 - ▶ Na linha 2 temos 2 elementos
 - ▶ Na linha 3 temos 3 elementos
 - ▶ ...
 - ▶ Na linha n temos n elementos
- ▶ Total = $1 + 2 + 3 + \dots + n$ elementos

Matriz Triangular Inferior

- ▶ **Quantos elementos armazenar?**
- ▶ Total = $1 + 2 + 3 + \dots + n$ elementos.
- ▶ Progressão aritmética (PA) de razão $r = 1$.
- ▶ Soma dos n termos de uma PA:

$$S_n = \frac{(a_1 + a_n)n}{2}$$

- ▶ Portanto, a matriz possui um total de $\frac{(n+1)n}{2}$ elementos.

Matriz Triangular Inferior

- ▶ **Como armazenar os elementos?**
- ▶ Utilizando uma representação linear.
- ▶ Armazenar os elementos da matriz no vetor `vet` da seguinte forma:
 - ▶ Linha por linha;
 - ▶ Da esquerda para a direita.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 7 & 8 & 9 & 10 & 0 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$$

índices	0	1	2	3	4	5	6	13	14				
vet	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Matriz Triangular Inferior

- ▶ **Como acessar/modificar um elemento?**
- ▶ Para acessar o elemento na posição (i, j) :
 - ▶ Contar quantos elementos tem antes da linha i :

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 7 & 8 & 9 & 10 & 0 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$$

índices 0 1 2 3 4 5 6 ...

vet

1	2	3	4	5	6	7	8	9	...
---	---	---	---	---	---	---	---	---	-----

- ▶ Número de elementos até a linha i :

$$\frac{(i+1)i}{2}$$

- ▶ Para acessar o elemento (i, j) basta somar j , isto é

$$k = \frac{(i+1)i}{2} + j$$

Matriz Triangular Inferior

```
class MatrizTriInf
{
public:
    MatrizTriInf(int ordem);
    ~MatrizTriInf();

    float get(int i, int j);
    void set(int i, int j, float valor);

private
    int n;           // ordem da matriz triangular
    float *vet;      // representacao linear

    bool verifica(int i, int j);
};
```

Matriz Triangular Inferior

```
MatrizTriInf::MatrizTriInf(int ordem)
{
    n = ordem;
    int tam = n*(n + 1)/2;
    vet = new float[tam];
}

MatrizTriInf::~~MatrizTriInf()
{
    delete [] vet;
}

bool MatrizTriInf::verifica(int i, int j)
{
    if(i >= 0 && i < n && j >= 0 && j < n)
        return true;
    else
        return false;
};
```

Matriz Triangular Inferior

```
float MatrizTriInf::get(int i, int j)
{
    if( verifica(i, j) )
    {
        if(i >= j)
        {
            int k = i*(i + 1)/2 + j;
            return vet[k];
        }
        else
            return 0.0;
    }
    else
        cout << "Erro: indice invalido\n";
    exit(1);
}
```

Matriz Triangular Inferior

```
void MatrizTriInf::set(int i, int j, float val)
{
    if( verifica(i, j) )
    {
        if(i >= j)
        {
            int k = i*(i + 1)/2 + j;
            vet[k] = valor;
        }
        else
            if(valor != 0.0)
                cout << "Elemento fora da parte "
                    << "triangular inferior\n";
    }
    else
        cout << "Erro: indices invalidos\n";
}
```

Matriz Triangular Superior

- Para uma matriz triangular superior...

$$U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

- Questões importantes:
 1. Quantos elementos armazenar?
 2. Representação linear: como armazenar os elementos?
 3. Como acessar/modificar um elemento?

Matriz Simétrica

- ▶ Definição:

$$a_{ij} = a_{ji}, \quad \forall i, j$$

- ▶ Exemplo

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{12} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{13} & a_{23} & a_{33} & \dots & a_{3n} \\ \vdots & & & \ddots & \\ a_{1n} & a_{2n} & a_{3n} & \dots & a_{nn} \end{bmatrix}$$

- ▶ Representação: matriz triangular superior (ou triangular inferior).
- ▶ Se a representação for por uma matriz triangular superior:
 - ▶ Se $i < j$, retorna o elemento $k = \frac{(j+1)j}{2} + i$.
 - ▶ Senão, retorna o elemento $k = \frac{(i+1)i}{2} + j$.

Matriz Anti-Simétrica

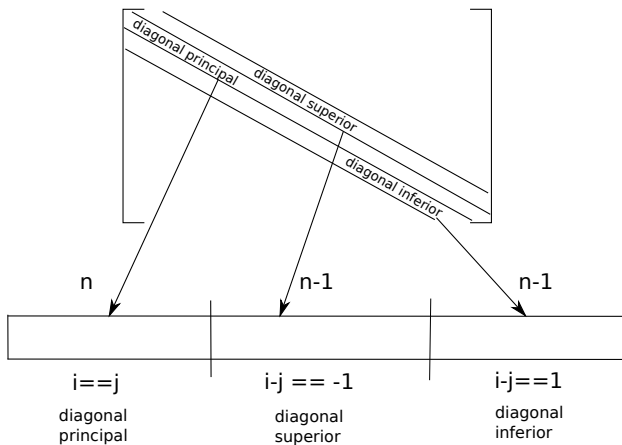
- Definição:

$$a_{ij} = -a_{ji}, \quad \forall i, j$$

- Exemplo

$$A = \begin{bmatrix} 0 & a_{12} & a_{13} & \dots & a_{1n} \\ a_{12} & 0 & a_{23} & \dots & a_{2n} \\ a_{13} & a_{23} & 0 & \dots & a_{3n} \\ \vdots & & & \ddots & \\ a_{1n} & a_{2n} & a_{3n} & \dots & 0 \end{bmatrix}$$

Matriz Tridiagonal



Exercícios

1. Desenvolver um TAD para matrizes **diagonais**.
2. Desenvolver um TAD para matrizes **triangulares superiores**.
2. Desenvolver um TAD para matrizes **simétricas**.
3. Desenvolver um TAD para matrizes **anti-simétricas**.
4. Desenvolver um TAD para matrizes **tridiagonais**.
 - * Em cada caso desenvolver uma aplicação para testar e usar o TAD da matriz desenvolvido.