

# Estruturas de Dados

## TADs: Listas, Filas, Pilhas

---

Professores: Anisio Lacerda  
Wagner Meira Jr.  
Washington Cunha

# Módulo 3 - Sumário

- Introdução
  - Tipos Abstratos de Dados
- Listas Lineares
  - Implementação por arranjos (sequencial)
  - Implementação por apontadores (encadeada)
- Filas, Pilhas
  - Implementação por arranjos (sequencial)
  - Implementação por apontadores (encadeada)

# Tipos Abstratos de Dados (TADs)

- **Construções que agrupam a estrutura de dados juntamente com as operações que podem ser feitas sobre esses dados**
- O TAD **encapsula** a estrutura de dados, fornecendo acesso apenas através de uma “interface” (conjunto de funções públicas)
  - Usuário do TAD só “enxerga” a interface, não a implementação específica

# Como implementar...

## ■ Em C++: **Classes**

- ❑ Atributos privados encapsulam os dados
- ❑ Métodos públicos fornecem a interface
- ❑ Exemplo:

```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};
        void InsereInicio(TipoItem item);
        TipoItem RemovePosicao(int pos);
        TipoItem Pesquisa(TipoChave c);
        void Imprime();

    private:
        static const int MAXTAM = 100;
        TipoItem itens[MAXTAM];
};
```

# Estruturas de Dados

## Listas Lineares

---

Professores: Anisio Lacerda  
Wagner Meira Jr.  
Washington Cunha

# Listas Lineares

- Maneira de representar um conjunto de elementos
- Itens podem ser acessados, inseridos ou retirados em qualquer posição de uma lista
- Com isso, as listas podem crescer ou diminuir de tamanho durante a execução
- Implementada na maioria das linguagens
  - STL (C++): list
  - java.util: List, ArrayList, LinkedList
  - python: lista =  $[x_1, x_2, \dots, x_n]$

# Definição formal de Listas Lineares

- **Sequência de zero ou mais itens**

- $x_1, x_2, \dots, x_n$ , na qual  $x_i$  é de um determinado tipo e  $n$  representa o tamanho da lista linear.

- **Sua principal propriedade estrutural envolve as posições relativas dos itens em uma dimensão.**

- Assumindo  $n \geq 1$ ,  $x_1$  é o primeiro item da lista e  $x_n$  é o último item da lista.
- $x_i$  precede  $x_{i+1}$  para  $i = 1, 2, \dots, n - 1$
- $x_i$  sucede  $x_{i-1}$  para  $i = 2, 3, \dots, n$
- o elemento  $x_i$  é dito estar na  $i$ -ésima posição da lista.

# TAD: Lista

## ■ Duas Implementações:

- ❑ Sequencial (uso de arranjos, alocação estática)
- ❑ Encadeada (uso de apontadores, alocação dinâmica)

## ■ Operações:

- ❑ Criar uma nova lista (construtor)
- ❑ Métodos de Acesso (Get, Set)
- ❑ Testar se é uma lista *vazia*
- ❑ Inserção: no início, no final, em uma posição  $p$
- ❑ Remoção: do início, do final, de uma posição  $p$
- ❑ Pesquisar por uma chave
- ❑ Imprimir a Lista
- ❑ Limpar a Lista

### ***Disclaimer:***

*os códigos que serão apresentados devem ser considerados como exemplos. Eles não são, necessariamente, os mais modulares ou eficientes...*



# TAD Lista

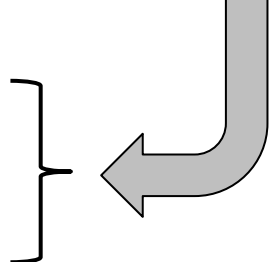
## ■ Class Lista

- ❑ Classe Abstrata: métodos implementados nas classes herdeiras
- ❑ Trata apenas o atributo *tamanho* (inicialização, acesso, teste Vazia)

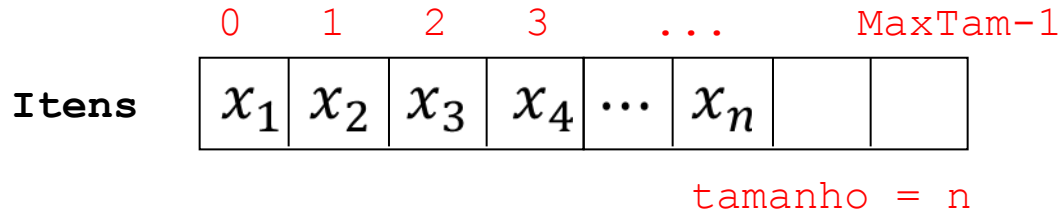
```
class Lista
{
    public:
        Lista() {tamanho = 0;};
        int GetTamanho() {return tamanho;};
        bool Vazia() {return tamanho == 0;};

        virtual TipoItem GetItem(int pos) = 0;
        virtual void SetItem(TipoItem item, int pos) = 0;
        virtual void InsereInicio(TipoItem item) = 0;
        virtual void InsereFinal(TipoItem item) = 0;
        virtual void InserePosicao(TipoItem item, int pos) = 0;
        virtual TipoItem RemoveInicio() = 0;
        virtual TipoItem RemoveFinal() = 0;
        virtual TipoItem RemovePosicao(int pos) = 0;
        virtual TipoItem Pesquisa(TipoChave c) = 0;
        virtual void Imprime() = 0;
        virtual void Limpa() = 0;

    protected:
        int tamanho;
};
```

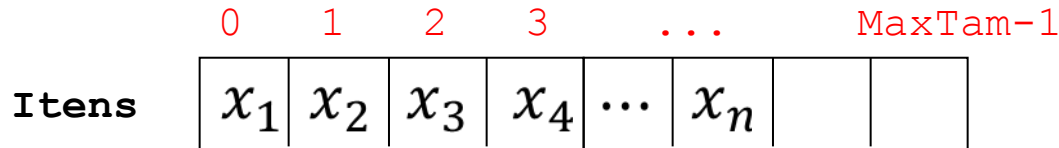


# Alocação Sequencial



- Itens da lista são armazenados em um vetor
  - ❑ Alocação Estática, com um tamanho máximo
  - ❑ Permite acesso aleatório a qualquer posição em tempo  $O(1)$
  - ❑ Permite percorrer a lista em ambas direções se necessário
  - ❑ Vetor começa em 0:  $i$ -ésimo item fica na posição  $i-1$ .
- Inserção e Remoção
  - ❑ No final tem custo constante:  $O(1)$
  - ❑ Em qualquer outra posição causa o deslocamento dos itens à frente:  $O(n)$  no pior caso

# Class Lista Arranjo



tamanho = n

```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};
        TipoItem GetItem(int pos);
        void SetItem(TipoItem item, int pos);
        void InsereInicio(TipoItem item);
        void InsereFinal(TipoItem item);
        void InserePosicao(TipoItem item, int pos);
        TipoItem RemoveInicio();
        TipoItem RemoveFinal();
        TipoItem RemovePosicao(int pos);
        TipoItem Pesquisa(TipoChave c);
        void Imprime();
        void Limpa();

    private:
        static const int MAXTAM = 100;
        TipoItem itens[MAXTAM];
};
```

# Class TipoItem

- Classe para representar os elementos da lista
  - Campo **int chave**: identificador único
  - Poderia ter outros campos, ou possuir um apontador para qualquer outro tipo de objeto
  - Possui métodos para inicialização, acesso e impressão

```
typedef int TipoChave; // TipoChave é um inteiro
```

```
class TipoItem
{
    public:
        TipoItem();
        TipoItem(TipoChave c);
        void SetChave(TipoChave c);
        TipoChave GetChave();
        void Imprime();

    private:
        TipoChave chave;
        // outros membros
};
```

# *Class* TipoItem

## Métodos para Inicialização acesso e impressão

```
TipoItem::TipoItem()
{
    chave = -1; // indica um item vazio
}

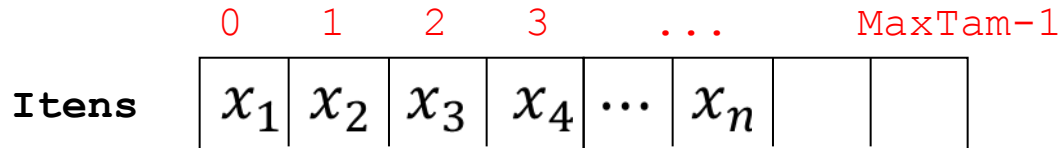
TipoItem::TipoItem(TipoChave c)
{
    chave = c;
}

void TipoItem::SetChave(TipoChave c)
{
    chave = c;
}

TipoChave TipoItem::GetChave()
{
    return chave;
}

void TipoItem::Imprime()
{
    printf("%d ", chave);
}
```

# Class Lista Arranjo



tamanho = n

```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};
        TipoItem GetItem(int pos);
        void SetItem(TipoItem item, int pos);
        void InsereInicio(TipoItem item);
        void InsereFinal(TipoItem item);
        void InserePosicao(TipoItem item, int pos);
        TipoItem RemoveInicio();
        TipoItem RemoveFinal();
        TipoItem RemovePosicao(int pos);
        TipoItem Pesquisa(TipoChave c);
        void Imprime();
        void Limpa();

    private:
        static const int MAXTAM = 100;
        TipoItem itens[MAXTAM];
};
```

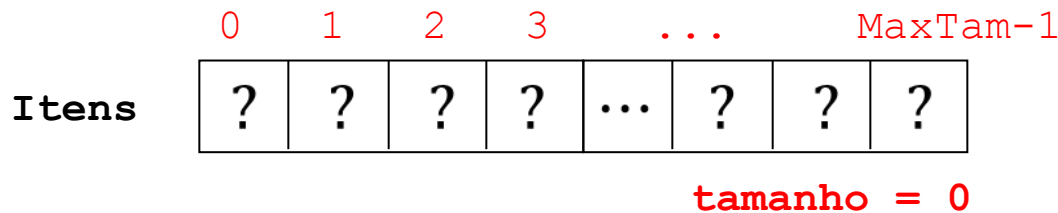
# Class Lista Arranjo - Construtor

## ■ Construtor

- ❑ Apenas chama o construtor da classe pai, que inicializa o atributo **tamanho** com o valor 0.
- ❑ O conteúdo dos elementos do vetor **itens** não importa...

```
class ListaArranjo : public Lista
{
    public:
        ListaArranjo() : Lista() {};

    ...
};
```



**ListaArranjo L;**

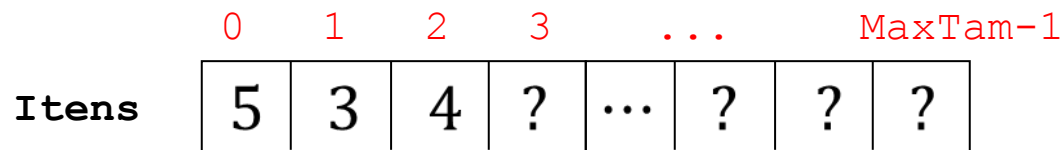
# Class Lista Arranjo – Get & Set

```
TipoItem ListaArranjo::GetItem(int pos){  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    return itens[pos-1];  
}
```

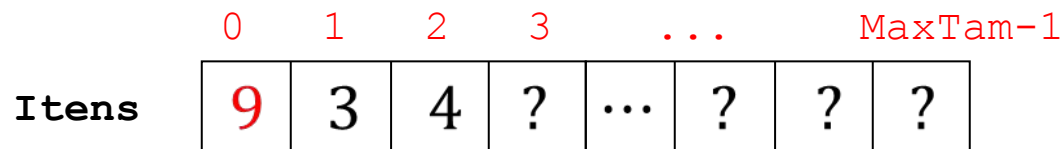
$O(1)$

```
void ListaArranjo::SetItem(TipoItem item, int pos){  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    itens[pos-1] = item;  
}
```

$O(1)$



tamanho = 3



tamanho = 3

```
ListaArranjo L;  
TipoItem x;  
...  
x.SetChave(9)  
L.SetItem(x,1);  
x = L.GetItem(3)  
x.Imprime();
```

Posição Lógica x Posição Física  
1º elemento está na posição 0

4



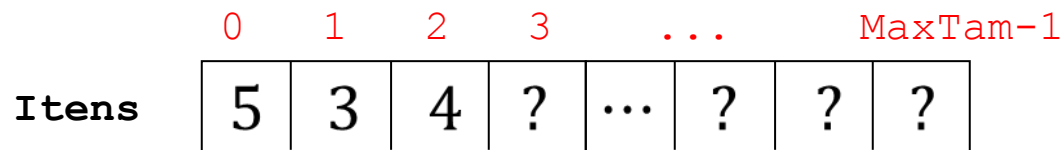
# *Class* Lista Arranjo - Inserção

- Inserção pode ser feita no início, no final, ou em uma posição  $p$  qualquer
- A inserção que não seja feita no final causa o deslocamento de todos os itens do vetor
- Deve-se testar se há espaço para a inserção do novo item (alocação estática)
  - Gera uma exceção que pode ser tratada por quem chamou o método.

# Class Lista Arranjo - Inserção

```
void ListaArranjo::InsereInicio(TipoItem item) {  
    int i;  
    if (tamanho == MAXTAM)  
        throw "ERRO: Lista cheia!";  
  
    tamanho++;  
    for(i=tamanho-1;i>0;i--)  
        itens[i] = itens[i-1];  
  
    itens[0] = item;  
};
```

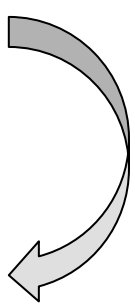
$O(n)$



tamanho = 3



tamanho = 4

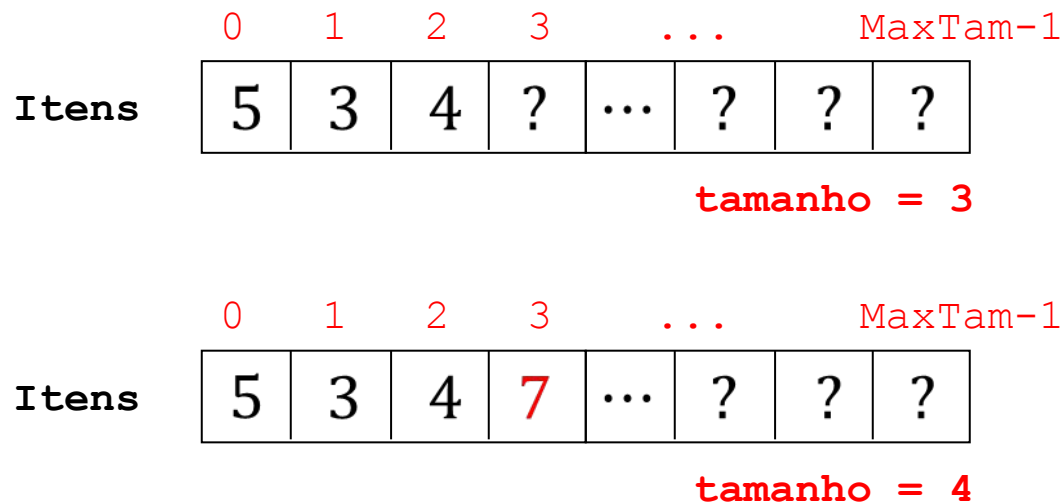


```
ListaArranjo L;  
TipoItem x;  
...  
x.SetChave(7)  
L.InsereInicio(x)
```

# Class Lista Arranjo - Inserção

```
void ListaArranjo::InsereFinal(TipoItem item) {  
    if (tamanho == MAXTAM)  
        throw "ERRO: Lista cheia!";  
  
    itens[tamanho] = item;  
    tamanho++;  
};
```

$O(1)$



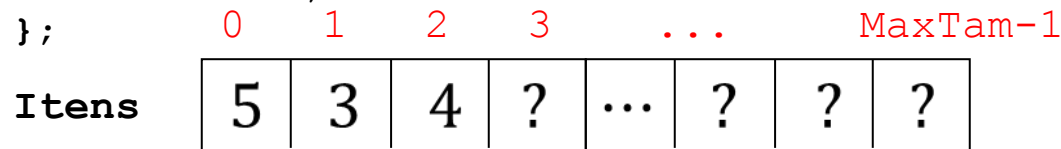
```
ListaArranjo L;  
TipoItem x;  
...  
x.SetChave(7)  
L.InsereFinal(x)
```

# Class Lista Arranjo - Inserção

```
void ListaArranjo::InserePosicao(TipoItem item, int pos) {  
    int i;  
    if (tamanho == MAXTAM)  
        throw "ERRO: Lista cheia!";  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    pos--; // posição no vetor = pos-1 (vetor começa em  
0)  
    for(i=tamanho;i>pos;i--)  
        itens[i] = itens[i-1];  
  
    itens[pos] = item;  
    tamanho++;  
};
```

Melhor  
Caso  $O(1)$

Pior  
Caso  $O(n)$



tamanho = 3



tamanho = 4

ListaArranjo L;  
TipoItem x;  
...  
**x.SetChave(7)**  
**L.InserePosicao(x,2)**

Posição Lógica x Posição Física  
2º elemento está na posição 1

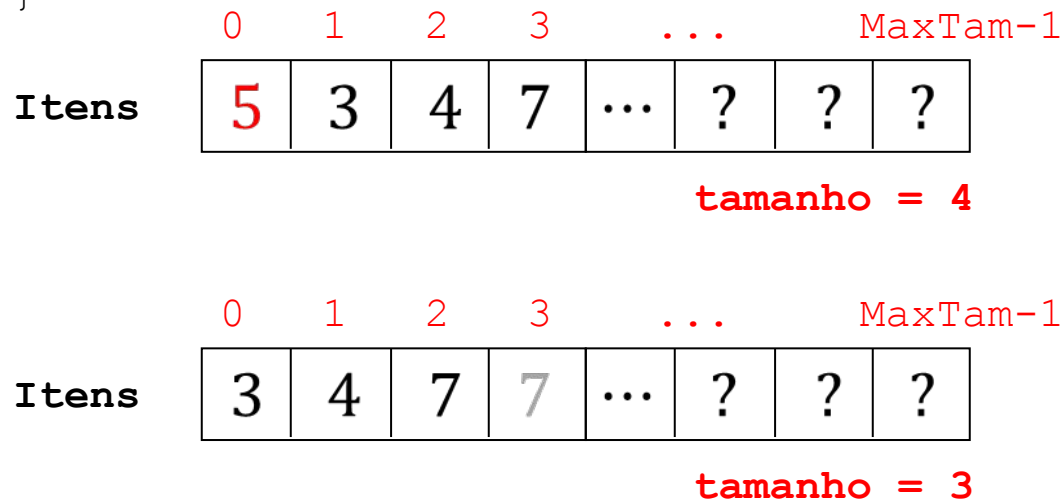
# *Class* Lista Arranjo - Remoção

- Da mesma forma, a remoção pode ser feita no início, no final, ou em uma posição  $p$  qualquer
- A remoção que não seja feita no final causa o deslocamento de todos os itens do vetor
- Deve-se verificar se há elementos e se a posição de remoção é válida
  - Gera uma exceção que pode ser tratada por quem chamou o método.
- O elemento removido é retornado pelo método

# Class Lista Arranjo - Remoção

```
TipoItem ListaArranjo::RemoveInicio() {  
    int i;  
    TipoItem aux;  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    aux = itens[0];  
    for(i=0; i<tamanho; i++)  
        itens[i] = itens[i+1];  
  
    tamanho--;  
    return aux;  
}
```

$O(n)$



```
ListaArranjo L;  
TipoItem x;  
...  
x = L.RemoveInicio();  
x.Imprime();
```

5

# Class Lista Arranjo - Remoção

```
TipoItem ListaArranjo::RemoveFinal() {  
    TipoItem aux;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    tamanho--;  
    aux = itens[tamanho];  
    return aux;  
}
```

$O(1)$



tamanho = 4



tamanho = 3

```
ListaArranjo L;  
TipoItem x;  
...  
x = L.RemoveFinal();  
x.Imprime();
```

7

# Class Lista Arranjo - Remoção

```
TipoItem ListaArranjo::RemovePosicao(int pos) {  
    int i; TipoItem aux;  
  
    if ( (pos > tamanho) || (pos <= 0) )  
        throw "ERRO: Posicao Invalida!";  
  
    pos--; // posição no vetor = pos-1 (vetor começa em  
0)  
    aux = itens[pos];  
    for(i=pos; i<tamanho; i++)  
        itens[i] = itens[i+1];  
    tamanho--;  
    return aux;  
}
```

Melhor  
Caso  $O(1)$

Pior  
Caso  $O(n)$



tamanho = 4



tamanho = 3

```
ListaArranjo L;  
TipoItem x;  
...  
x = L.RemovePosicao(2);  
x.Imprime();
```

Posição Lógica x Posição Física  
2º elemento está na posição 1

3



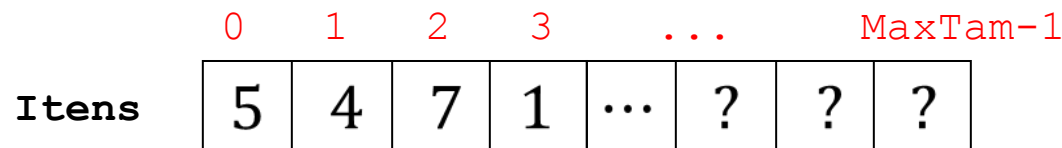
# Class Lista Arranjo - Pesquisa

- Pesquisa por um item com uma determinada chave
  - Retorna o item encontrado ou um *flag* (-1)

```
TipoItem ListaArranjo::Pesquisa(TipoChave c) {  
    int i; TipoItem aux;  
  
    if (tamanho == 0)  
        throw "ERRO: Lista vazia!";  
  
    aux.SetChave(-1); // indica pesquisa sem sucesso  
    for(i=0; i<tamanho; i++)  
        if(itens[i].GetChave() == c) {  
            aux = itens[i];  
            break;  
        }  
  
    return aux;  
};
```

Melhor Caso  $O(1)$

Pior Caso  $O(n)$



tamanho = 4

```
ListaArranjo L;  
TipoItem x;  
...  
x = L.Pesquisa(7);  
x.Imprime();
```

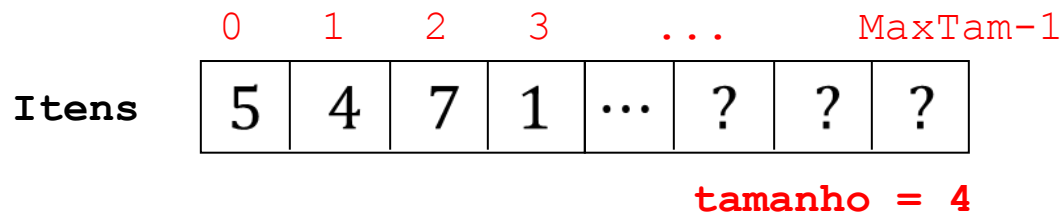
7

# Class Lista Arranjo - Imprime

## ■ Imprime todos os elementos

```
void ListaArranjo::Imprime() {  
    int i;  
  
    for(i=0;i<tamanho;i++)  
        itens[i].Imprime();  
  
    printf("\n");  
};
```

$O(n)$



```
ListaArranjo L;  
TipoItem x;  
...  
L.Imprime();
```

5 4 7 1

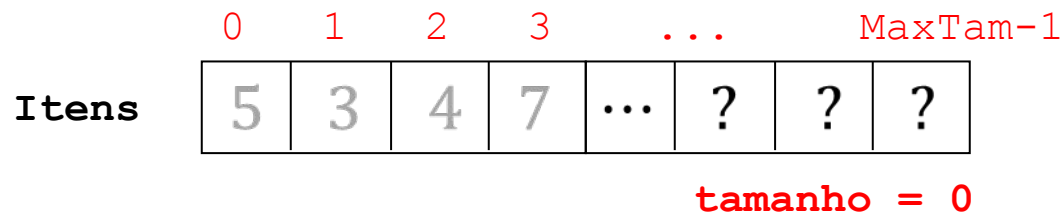
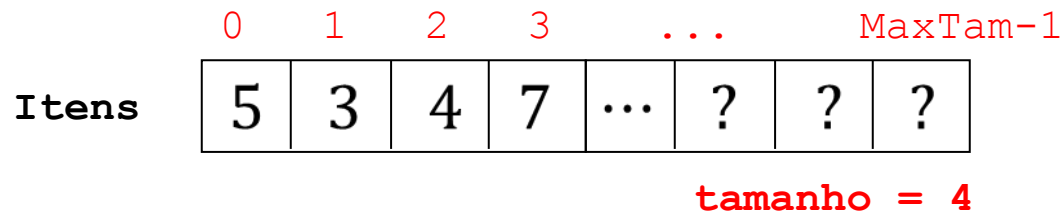
# Class Lista Arranjo - Limpa

## ■ “Limpa” a Lista

- Basta fazer o tamanho = 0

```
void ListaArranjo::Limpa() {  
    tamanho = 0;  
};
```

$O(1)$



```
ListaArranjo L;  
TipoItem x;  
...  
L.Limpa();
```

# Alocação Sequencial

- Vantagens:
  - ❑ **Não necessita de apontadores explícitos** para organizar os itens na lista
    - Economia de memória
    - Implementação mais simples
  - ❑ Permite **acesso direto aos itens** em uma determinada posição
    - Métodos *Get* e *Set* são  $O(1)$

# Alocação Sequencial

- Desvantagens:

- ❑ Custo para inserir ou retirar itens da lista, que pode causar um deslocamento de todos os itens
  - $O(n)$  no pior caso
- ❑ O tamanho máximo da lista é **fixo** e definido em tempo de compilação. Pouco prático em aplicações onde o tamanho não pode ser previsto...
  - Pode causar *overflow* se número de itens for maior que o tamanho previsto
  - Desperdício de memória se o número de itens for muito menor que o tamanho previsto

# Estrutura de Dados

## Pilhas e Filas

---

Professores: Anisio Lacerda  
Wagner Meira Jr.  
Washington Cunha

# TAD Pilhas

- Tipo Abstrato de dados com a seguinte característica:

O último elemento a ser inserido é o primeiro a ser retirado (*LIFO – Last In First Out*)

- Analogia: pilha de pratos, pilha de livros, etc
- Usos: chamada de subprogramas, avaliação de expressões aritméticas, caminhamento em árvores, etc...

# TAD: Pilha

- Duas Implementações:

- ❑ Sequencial (uso de arranjos, alocação estática)
- ❑ Encadeada (uso de apontadores, alocação dinâmica)

- Operações:

- ❑ Criar uma nova pilha (construtor)
- ❑ Testar se a pilha está *vazia*
- ❑ **Empilhar** um item
- ❑ **Desempilhar** um item
- ❑ Limpar a pilha

*Disclaimer: os códigos que serão apresentados devem ser considerados como exemplos. Eles não são, necessariamente, os mais modulares ou eficientes...*



# TAD Pilha

## ■ Class Pilha

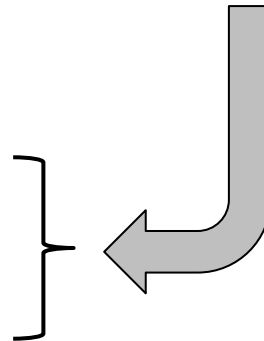
- ❑ Classe Abstrata: métodos implementados nas classes herdeiras
- ❑ Trata apenas o atributo *tamanho* (inicialização, acesso, teste Vazia)

```
class Pilha
{
    public:
        Pilha() { tamanho = 0; };
        int GetTamanho() { return tamanho; };
        bool Vazia() { return tamanho == 0; };

        virtual void Empilha(TipoItem item) = 0;
        virtual TipoItem Desempilha() = 0;
        virtual void Limpa() = 0;

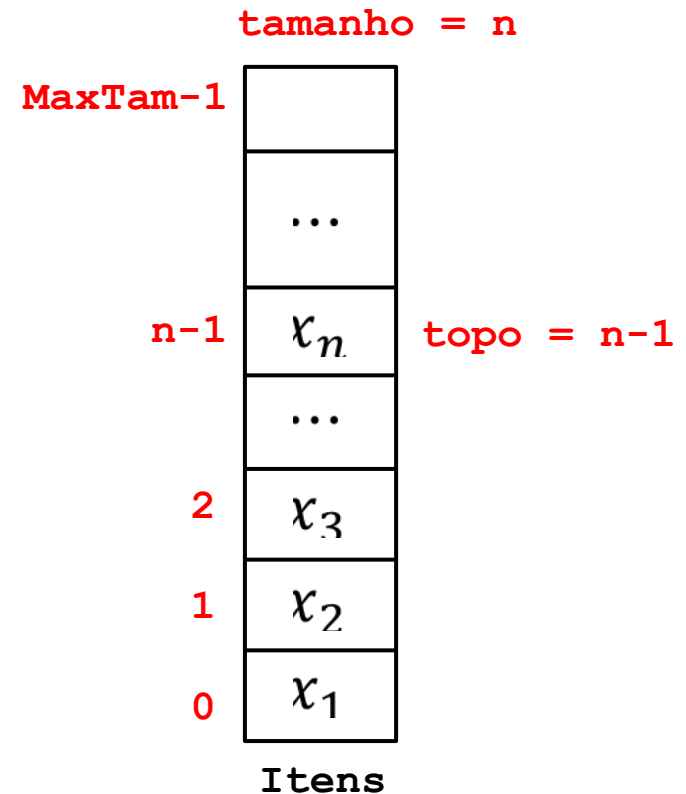
    protected:
        int tamanho;

};
```



# Alocação Sequencial

- Itens da pilha são armazenados em um vetor
  - Alocação Estática, com um tamanho máximo
  - Vetor começa em 0:  $i$ -ésimo item fica na posição  $i-1$ .
  - **topo** armazena o índice da posição onde está o topo da pilha
- Inserções e Retiradas em apenas um extremo do vetor
  - **Empilha**: incrementa o topo e coloca um novo elemento
  - **Desempilha**: retira o elemento do topo e decrementa

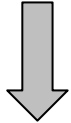


# Class Pilha Arranjo

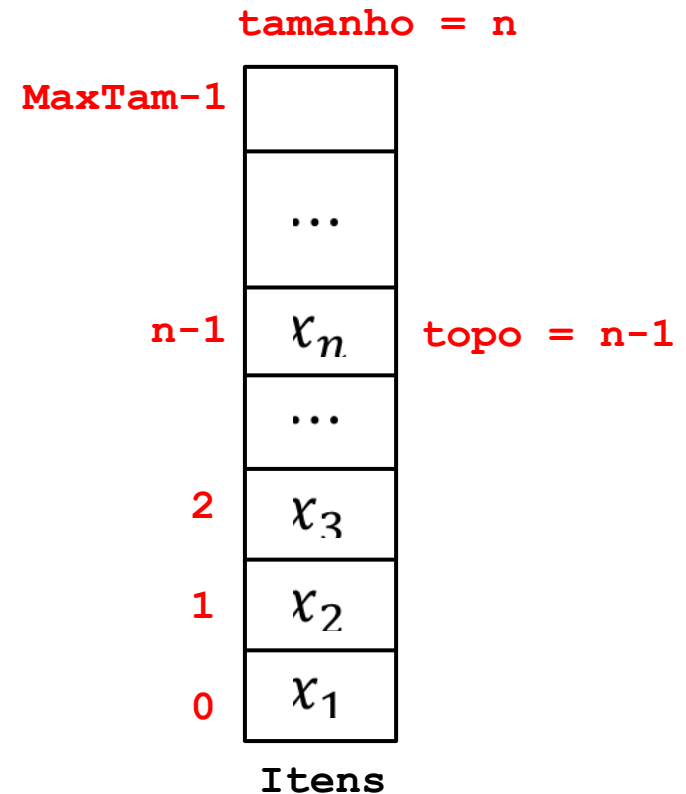
```
class PilhaArranjo : public Pilha
{
    public:
        PilhaArranjo();

        void Empilha(TipoItem item);
        TipoItem Desempilha();
        void Limpa();

    private:
        int topo;
        static const int MAXTAM = 100;
        TipoItem itens[MAXTAM];
};
```



Classe TipoItem é o mesmo da Lista



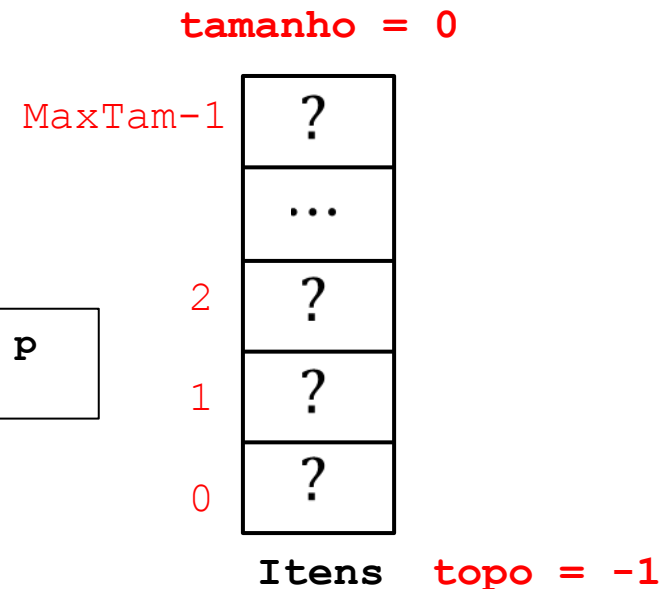
# Class Pilha Arranjo - Construtor

## ■ Construtor

- ❑ Chama o construtor da classe pai, que inicializa o atributo **tamanho** com o valor 0 e inicializa **topo** com o valor -1
- ❑ O conteúdo dos elementos do vetor **itens** não importa...

```
PilhaArranjo::PilhaArranjo() : Pilha()  
{  
    topo = -1;  
}
```

PilhaArranjo p  
...



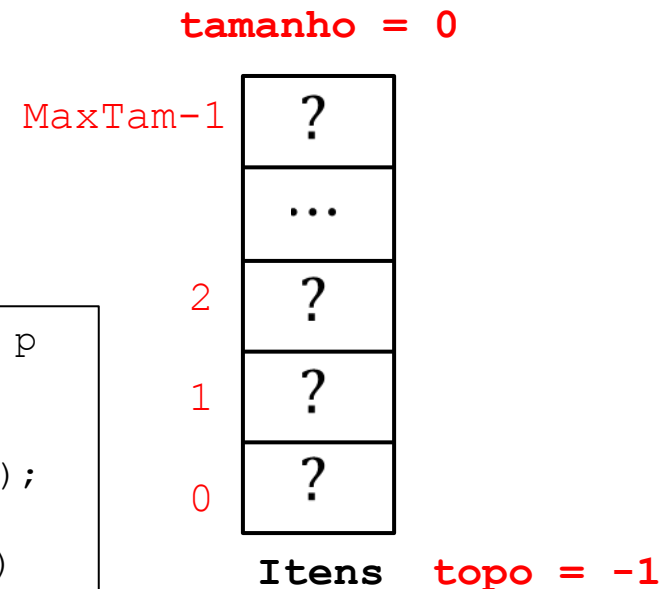
# Class Pilha Arranjo - Empilha

## ■ Empilha

- ❑ Testa se a Pilha está cheia, gerando uma exceção
- ❑ Incrementa o valor do topo e coloca o elemento na posição
- ❑ Incrementa o tamanho

```
void PilhaArranjo::Empilha(TipoItem item){  
    if(tamanho == MAXTAM)  
        throw "A pilha está cheia!";  
  
    topo++;  
    itens[topo] = item;  
    tamanho++;  
  
};
```

```
PilhaArranjo p  
TipoItem x;  
  
x.SetChave(5);  
p.Empilha(x)  
x.SetChave(2)  
p.Empilha(x)
```



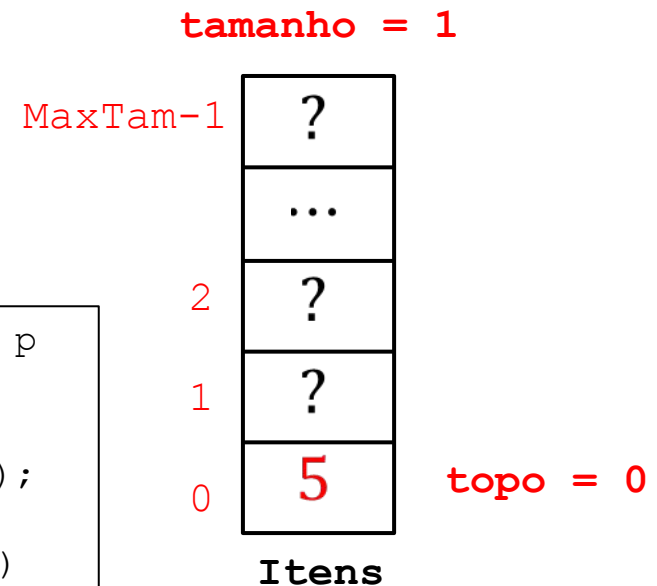
# Class Pilha Arranjo - Empilha

## ■ Empilha

- ❑ Testa se a Pilha está cheia, gerando uma exceção
- ❑ Incrementa o valor do topo e coloca o elemento na posição
- ❑ Incrementa o tamanho

```
void PilhaArranjo::Empilha(TipoItem item){  
    if(tamanho == MAXTAM)  
        throw "A pilha está cheia!";  
  
    topo++;  
    itens[topo] = item;  
    tamanho++;  
  
};
```

```
PilhaArranjo p  
TipoItem x;  
  
x.SetChave(5);  
p.Empilha(x)  
x.SetChave(2)  
p.Empilha(x)
```



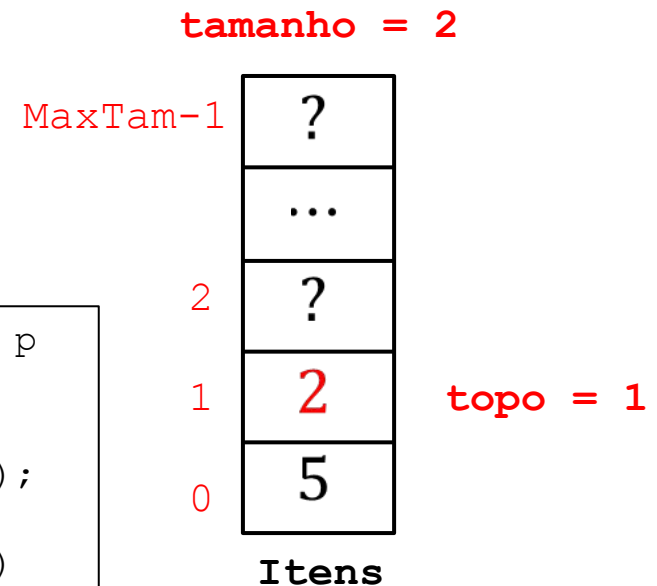
# Class Pilha Arranjo - Empilha

## ■ Empilha

- ❑ Testa se a Pilha está cheia, gerando uma exceção
- ❑ Incrementa o valor do topo e coloca o elemento na posição
- ❑ Incrementa o tamanho

```
void PilhaArranjo::Empilha(TipoItem item){  
    if(tamanho == MAXTAM)  
        throw "A pilha está cheia!";  
  
    topo++;  
    itens[topo] = item;  
    tamanho++;  
  
};
```

```
PilhaArranjo p  
TipoItem x;  
  
x.SetChave(5);  
p.Empilha(x)  
x.SetChave(2)  
p.Empilha(x)
```



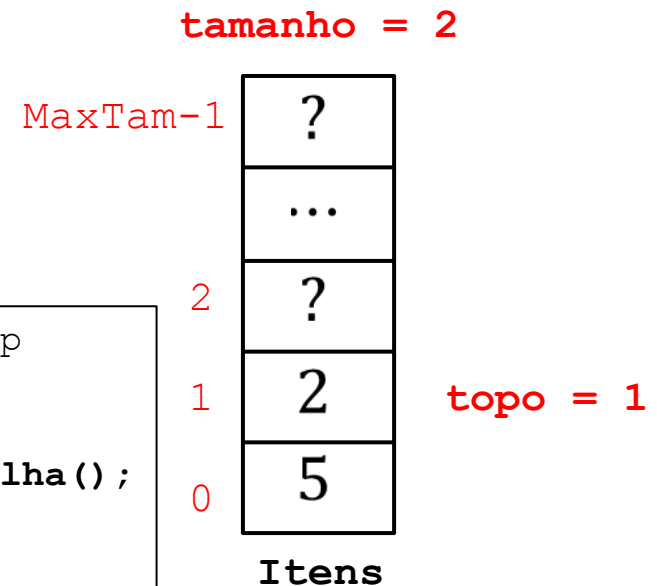
# Class Pilha Arranjo - Desempilha

## ■ Desempilha

- ❑ Testa se a pilha está vazia, gerando uma exceção
- ❑ Pega o valor que está no topo
- ❑ Decrementa o valor do topo e o tamanho
- ❑ Retorna o elemento

```
TipoItem PilhaArranjo::Desempilha() {  
    TipoItem aux;  
  
    if(tamanho == 0)  
        throw "A pilha está vazia!";  
  
    aux = itens[topo]  
    topo--;  
    tamanho--;  
    return aux;  
};
```

```
PilhaArranjo p  
TipoItem x;  
  
x = p.Desempilha();  
x.Imprime();  
x =  
p.Desempilha();  
x.Imprime();
```





# Class Pilha Arranjo - Desempilha

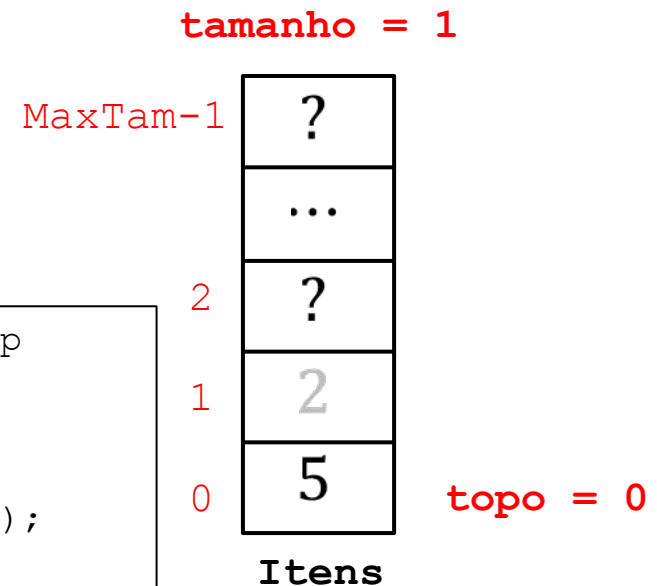
## ■ Desempilha

- ❑ Testa se a pilha está vazia, gerando uma exceção
- ❑ Pega o valor que está no topo
- ❑ Decrementa o valor do topo e o tamanho
- ❑ Retorna o elemento

```
TipoItem PilhaArranjo::Desempilha() {  
    TipoItem aux;  
  
    if(tamanho == 0)  
        throw "A pilha está vazia!";  
  
    aux = itens[topo]  
    topo--;  
    tamanho--;  
    return aux;  
};
```

2

```
PilhaArranjo p  
TipoItem x;  
  
x =  
p.Desempilha();  
x.Imprime();  
x = p.Desempilha();  
x.Imprime();
```



# Class Pilha Arranjo - Desempilha

## ■ Desempilha

- ❑ Testa se a pilha está vazia, gerando uma exceção
- ❑ Pega o item que está no topo e coloca em aux
- ❑ Decrementa o valor do topo e o tamanho
- ❑ Retorna o item

```
TipoItem PilhaArranjo::Desempilha() {  
    TipoItem aux;  
  
    if(tamanho == 0)  
        throw "A pilha está vazia!";  
  
    aux = itens[topo]  
    topo--;  
    tamanho--;  
    return aux;  
};
```

5

```
PilhaArranjo p  
TipoItem x;  
  
x =  
p.Desempilha();  
x.Imprime();  
x =
```

```
p.Desempilha();  
x.Imprime();
```

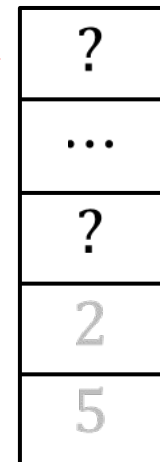
tamanho = 0

MaxTam-1

2

1

0



Itens topo = -1

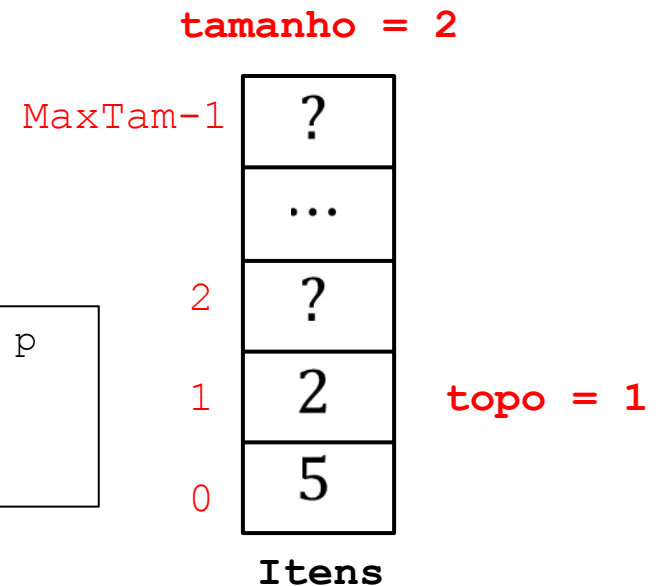
# Class Pilha Arranjo - Limpa

## ■ Limpa

- Apenas seta as variáveis tamanho para 0 e topo para -1

```
void PilhaArranjo::Limpa() {  
    topo = -1  
    tamanho = 0;  
};
```

```
PilhaArranjo p  
...  
p.Limpa();
```



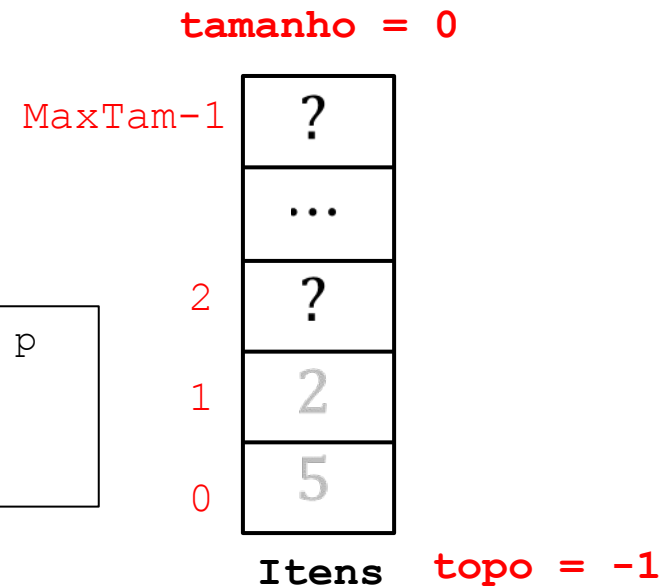
# Class Pilha Arranjo - Limpa

## ■ Limpa

- Apenas seta as variáveis tamanho para 0 e topo para -1

```
void PilhaArranjo::Limpa() {  
    topo = -1  
    tamanho = 0;  
};
```

```
PilhaArranjo p  
...  
p.Limpa();
```



# TAD Filas

- Tipo Abstrato de dados com a seguinte característica:

**O primeiro elemento a ser inserido é o primeiro a ser retirado (*FIFO – First In First Out*)**

- Analogia: fila bancária, fila do cinema
- Usos: Sistemas operacionais: fila de impressão, processamento; Simulação

# TAD: Fila

## ■ Duas Implementações:

- ❑ Sequencial (uso de arranjos, alocação estática)
- ❑ Encadeada (uso de apontadores, alocação dinâmica)

## ■ Operações:

- ❑ Criar uma nova fila (construtor)
- ❑ Testar se a fila está *vazia*
- ❑ **Enfileirar** um item: colocar um item no final da fila
- ❑ **Desenfileirar** um item: retirar um item do início da fila
- ❑ Limpar a fila

*Disclaimer: os códigos que serão apresentados devem ser considerados como exemplos. Eles não são, necessariamente, os mais modulares ou eficientes...*

# TAD Fila

## ■ Class Fila

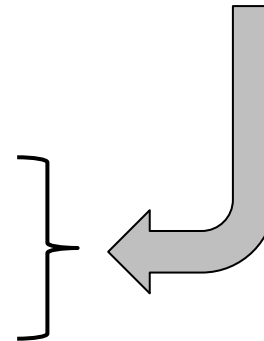
- ❑ Classe Abstrata: métodos implementados nas classes herdeiras
- ❑ Trata apenas o atributo *tamanho* (inicialização, acesso, teste Vazia)

```
class Fila
{
    public:
        Fila() {tamanho = 0;};
        int GetTamanho() {return tamanho;};
        bool Vazia() {return tamanho == 0;};

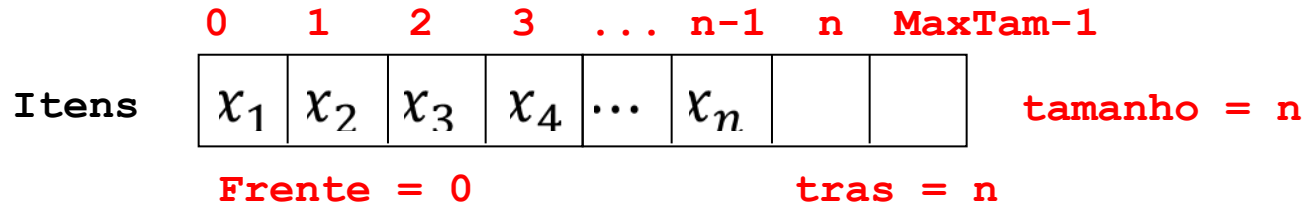
        virtual void Enfileira(TipoItem item) = 0;
        virtual TipoItem Desenfileira() = 0;
        virtual void Limpa() = 0;

    protected:
        int tamanho;

};
```



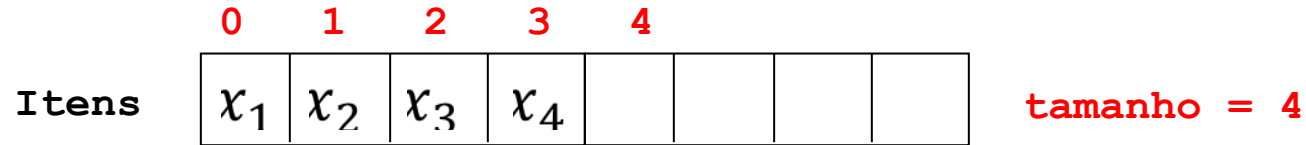
# Alocação Sequencial



- Itens da pilha são armazenados em um vetor
  - Alocação Estática, com um tamanho máximo
  - Vetor começa em 0: i-ésimo item fica na posição i-1.
  - **frente** armazena o índice da posição onde está o primeiro item
  - **tras** armazena o índice da primeira posição vazia no final da fila
- Operações
  - **Enfileira**: coloca um item no final da fila e incrementa **tras**
  - **Desenfileira**: retira o elemento da frente da fila, e... 2 opções:
    - Desloca todos os itens para frente:  $O(n)$
    - Incrementa o atributo frente: caminha no vetor... Pode “acabar” o espaço.

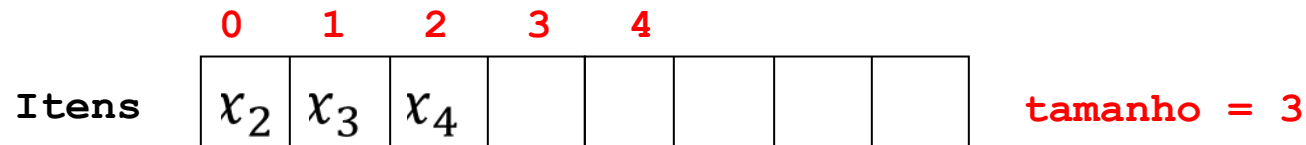


## Desenfileira: Desloca todos itens para frente:



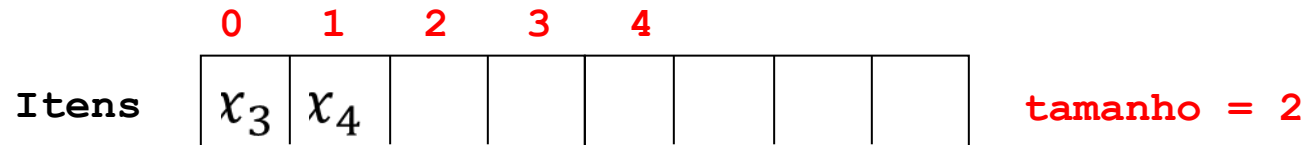
Frente = 0

tras = 4



Frente = 0

tras = 3

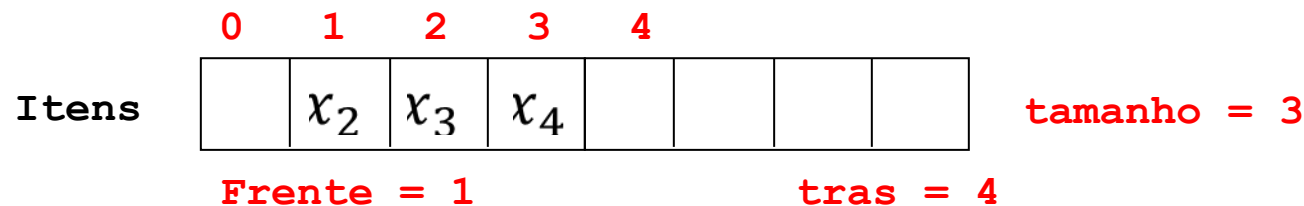
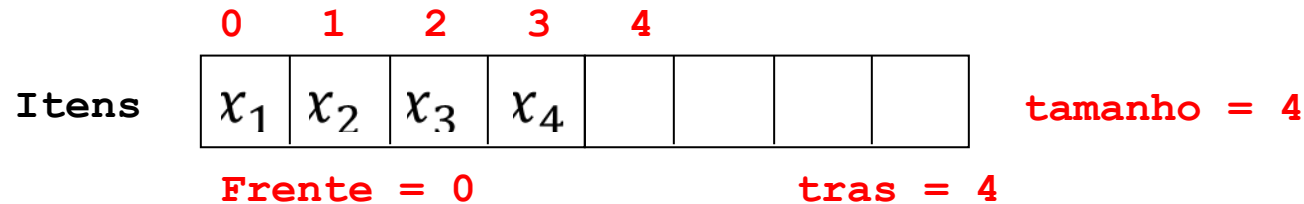
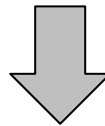
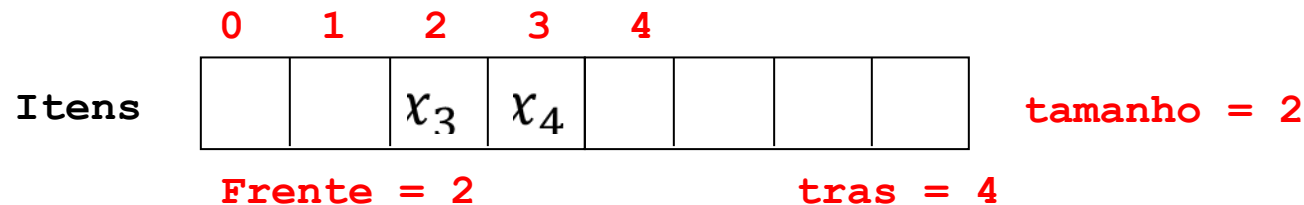


Frente = 0

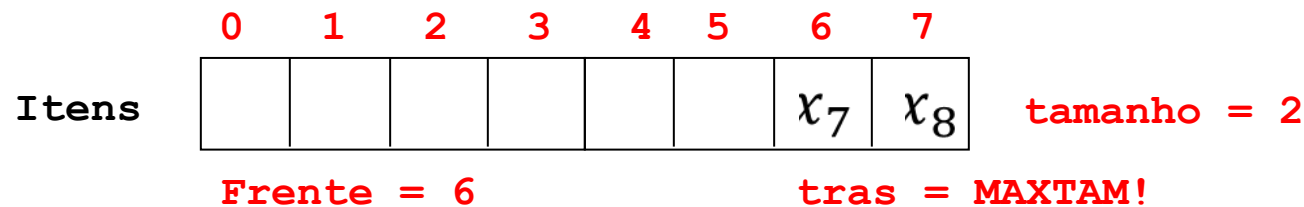
tras = 2

$$\text{Custo} = \mathcal{O}(n)$$

## Desenfileira: “Anda” com a frente da fila

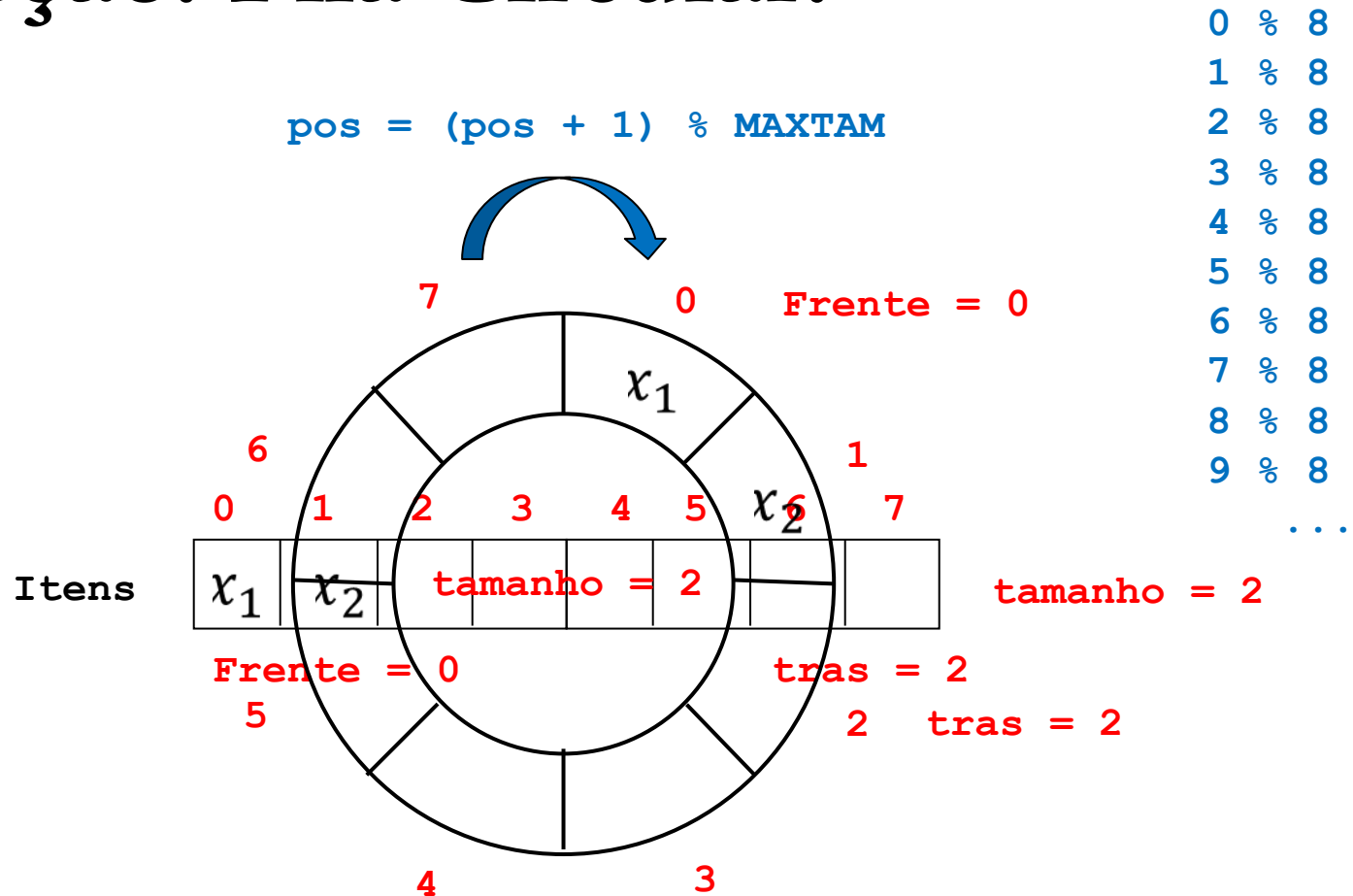
 $O(1)$ 

Problema: depois de algumas chamadas de enfileira e desenfileira...



# Solução: Fila Circular!

## Aritmética Modular



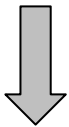
0	%	8	=	0
1	%	8	=	1
2	%	8	=	2
3	%	8	=	3
4	%	8	=	4
5	%	8	=	5
6	%	8	=	6
7	%	8	=	7
8	%	8	=	0
9	%	8	=	1
...				

# Class Fila Arranjo

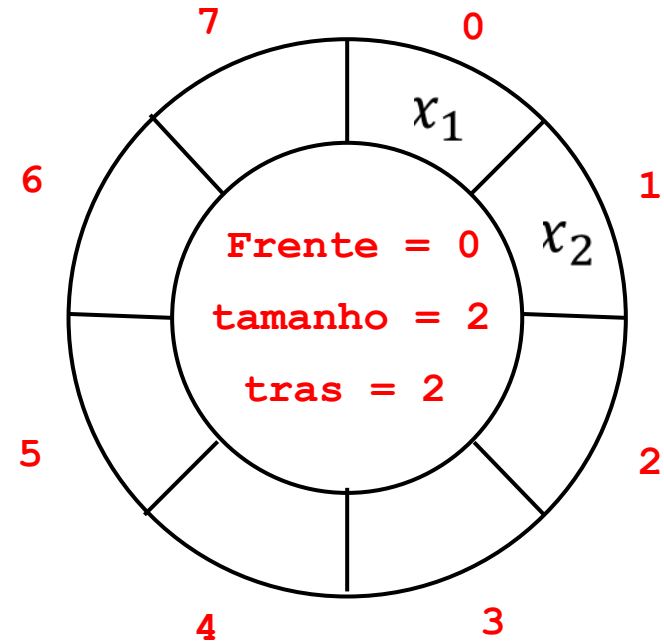
```
class FilaArranjo : public Fila
{
    public:
        FilaArranjo();

        void Enfileira(TipoItem item);
        TipoItem Desenfileira();
        void Limpa();

    private:
        int frente;
        int tras;
        static const int MAXTAM = 8;
        TipoItem itens[MAXTAM];
};
```



Classe TipoItem é o mesmo da Lista

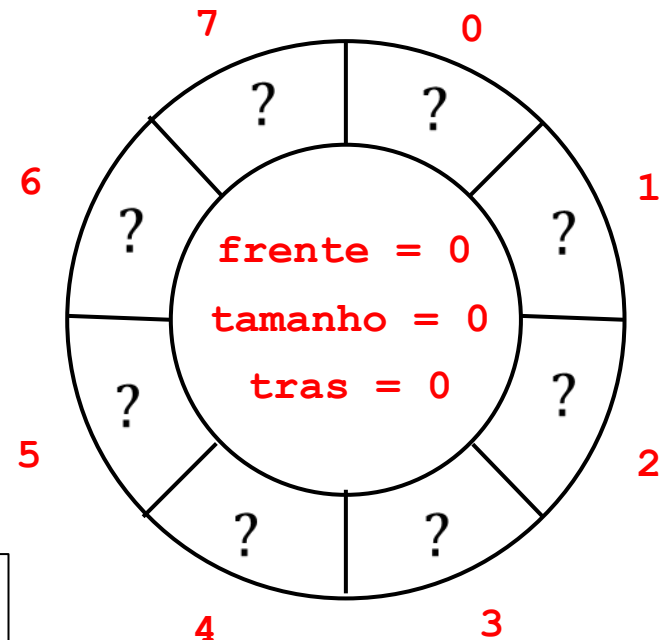


# Class Fila Arranjo - Construtor

## ■ Construtor

- ❑ Chama o construtor da classe pai, que inicializa o atributo **tamanho** com o valor 0 e inicializa **frente** e **tras** com o valor 0
- ❑ O conteúdo dos elementos do vetor **itens** não importa...

```
FilaArranjo::FilaArranjo() : Fila()  
{  
    frente = 0;  
    tras = 0;  
}
```



```
FilaArranjo f  
...
```

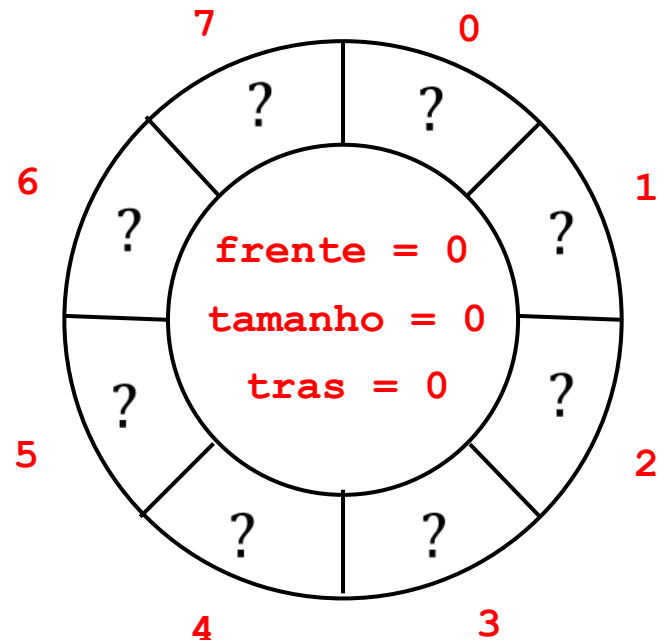
# Class FilaArranjo - Enfileira

## ■ Enfileira

- ❑ Testa se a Fila está cheia, gerando uma exceção
- ❑ Coloca o item na posição indicada por tras
- ❑ Incrementa tras usando aritmética modular:  $\% \text{MAXTAM}$  ( $\%8$ )
- ❑ Incrementa o tamanho

```
void FilaArranjo::Enfileira(TipoItem item) {  
    if (tamanho == MAXTAM)  
        throw "Fila Cheia!";  
  
    itens[tras] = item;  
    // fila circular  
    tras = (tras + 1) % MAXTAM;  
    tamanho++;  
}
```

```
FilaArranjo f  
TipoItem x;  
  
x.SetChave(5);  
f.Enfileira(x)
```



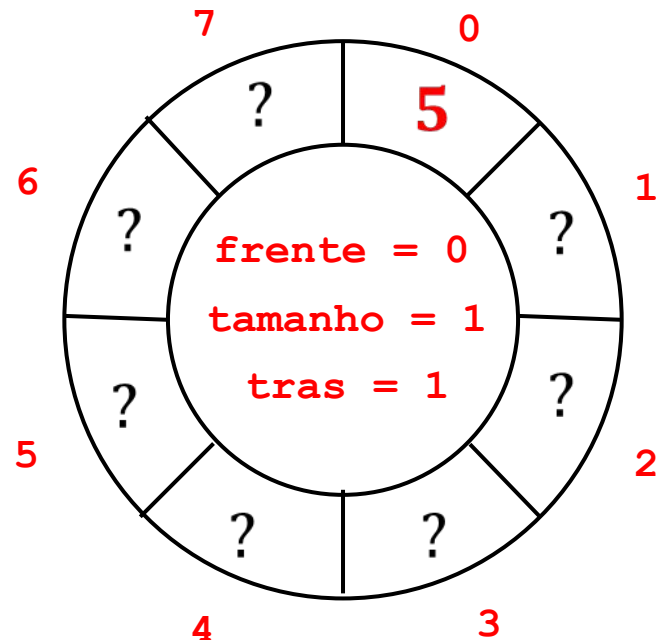
# Class FilaArranjo - Enfileira

## ■ Enfileira

- ❑ Testa se a Fila está cheia, gerando uma exceção
- ❑ Coloca o item na posição indicada por tras
- ❑ Incrementa tras usando aritmética modular:  $\% \text{MAXTAM} (\%8)$
- ❑ Incrementa o tamanho

```
void FilaArranjo::Enfileira(TipoItem item) {  
    if (tamanho == MAXTAM)  
        throw "Fila Cheia!";  
  
    itens[tras] = item;  
    // fila circular  
    tras = (tras + 1) % MAXTAM;  
    tamanho++;  
}
```

```
FilaArranjo f  
TipoItem x;  
  
x.SetChave(5);  
f.Enfileira(x)
```



# Class FilaArranjo - Enfileira

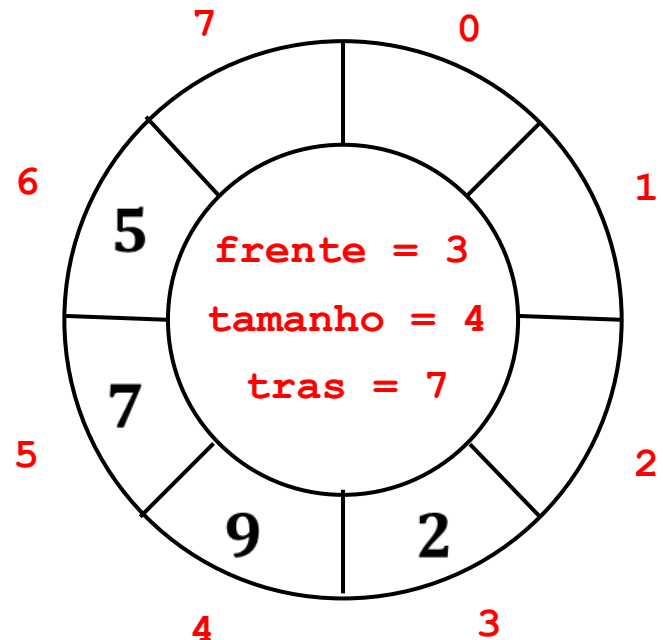
## ■ Enfileira

- ❑ Testa se a Fila está cheia, gerando uma exceção
- ❑ Coloca o item na posição indicada por tras
- ❑ Incrementa tras usando aritmética modular:  $\% \text{MAXTAM} (\%8)$
- ❑ Incrementa o tamanho

```
void FilaArranjo::Enfileira(TipoItem item) {  
    if (tamanho == MAXTAM)  
        throw "Fila Cheia!";  
  
    itens[tras] = item;  
    // fila circular  
    tras = (tras + 1) % MAXTAM;  
    tamanho++;  
}
```

**tras = (7+1)%8 = 0**

```
FilaArranjo f  
TipoItem x;  
...  
x.SetChave(4);  
f.Enfileira(x)
```





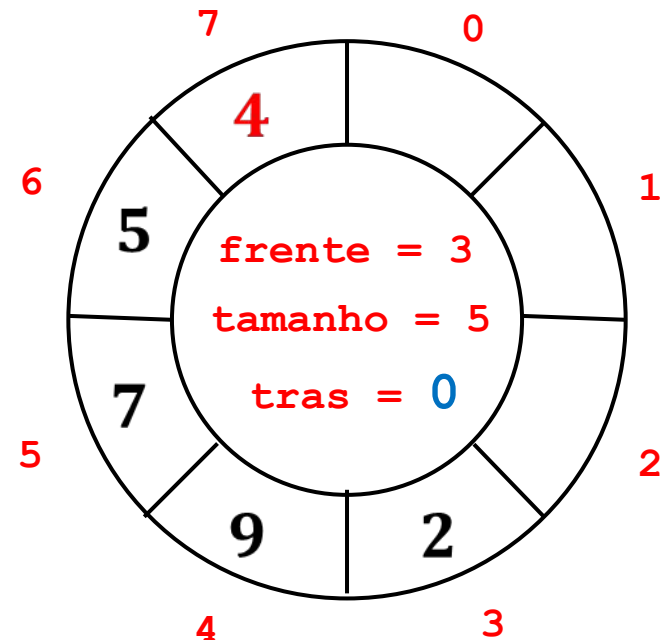
# Class FilaArranjo - Enfileira

## ■ Enfileira

- ❑ Testa se a Fila está cheia, gerando uma exceção
- ❑ Coloca o item na posição indicada por tras
- ❑ Incrementa tras usando aritmética modular:  $\% \text{MAXTAM} (\%8)$
- ❑ Incrementa o tamanho

```
void FilaArranjo::Enfileira(TipoItem item) {  
    if (tamanho == MAXTAM)  
        throw "Fila Cheia!";  
  
    itens[tras] = item;  
    // fila circular  
    tras = (tras + 1) % MAXTAM;  
    tamanho++;  
}
```

```
FilaArranjo f  
TipoItem x;  
...  
x.SetChave(4);  
f.Enfileira(x)
```



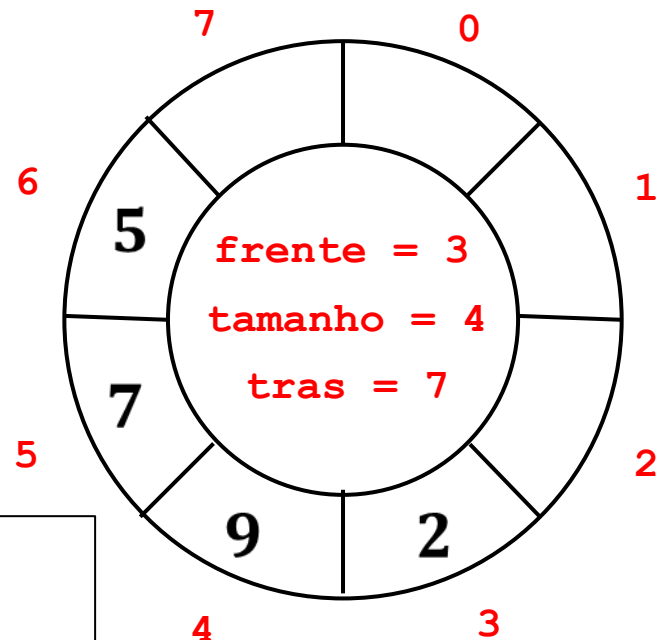
# Class FilaArranjo - Desenfileira

## ■ Desenfileira

- ❑ Testa se a Fila está vazia, gerando uma exceção
- ❑ Armazena o item da frente em aux
- ❑ Incrementa frente usando aritmética modular:  $\% \text{MAXTAM} (\%8)$
- ❑ decrementa o tamanho e retorna aux

```
TipoItem FilaArranjo::Desenfileira() {  
    TipoItem aux;  
  
    if (tamanho == 0)  
        throw "Fila está vazia!";  
  
    aux = itens[frente];  
    // fila circular  
    frente = (frente + 1) % MAXTAM;  
    tamanho--;  
  
    return aux;  
}
```

```
FilaArranjo f  
TipoItem x;  
...  
f.Desenfileira(x)  
x.Imprime();
```



# *Class* FilaArranjo - Desenfileira

## ■ Desenfileira

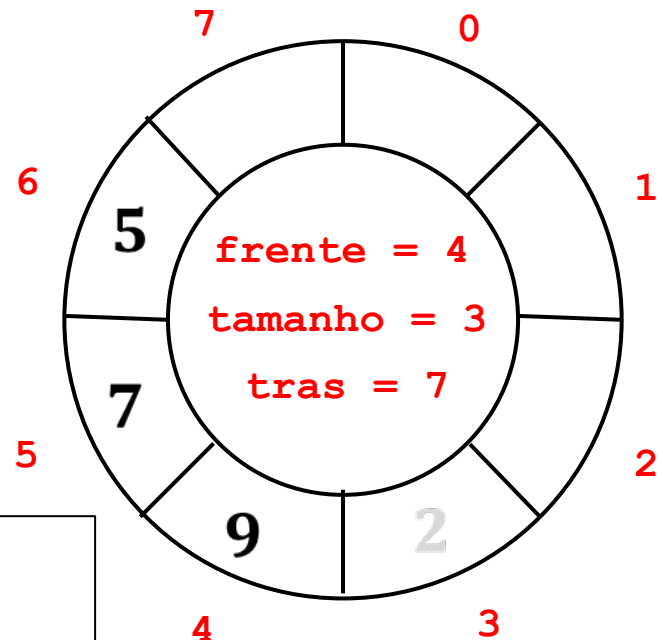
- ❑ Testa se a Fila está vazia, gerando uma exceção
- ❑ Armazena o item da frente em aux
- ❑ Incrementa frente usando aritmética modular:  $\% \text{ MAXTAM } (\%8)$
- ❑ decrementa o tamanho e retorna aux

```
TipoItem FilaArranjo::Desenfileira() {
    TipoItem aux;

    if (tamanho == 0)
        throw "Fila está vazia!";

    aux = itens[frente];
    // fila circular
    frente = (frente + 1) % MAXTAM;
    tamanho--;

    return aux;
}
```



```
FilaArranjo f
TipoItem x;
...
f.Desenfileira(x)
x.Imprime();
```

# Class FilaArranjo - Desenfileira

## ■ Desenfileira

- ❑ Testa se a Fila está vazia, gerando uma exceção
- ❑ Armazena o item da frente em aux
- ❑ Incrementa frente usando aritmética modular:  $\% \text{MAXTAM} (\%8)$
- ❑ decrementa o tamanho e retorna aux

```
TipoItem FilaArranjo::Desenfileira() {  
    TipoItem aux;
```

```
    if (tamanho == 0)  
        throw "Fila está vazia!";
```

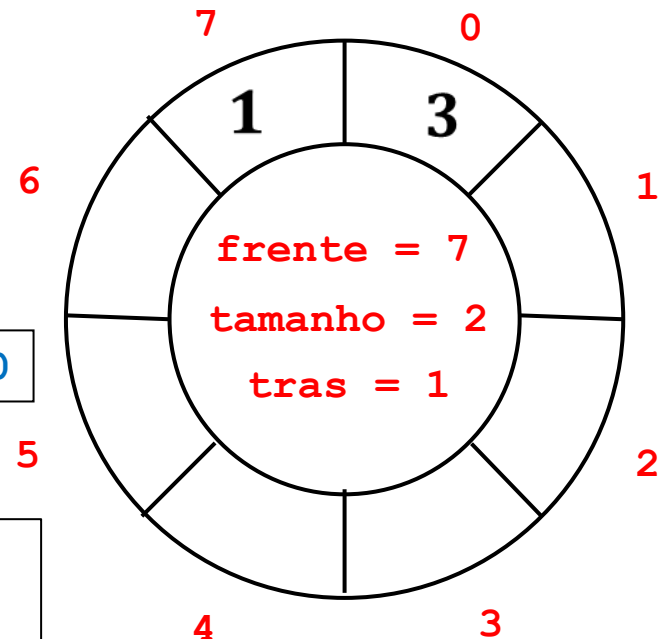
```
    aux = itens[frente];  
    // fila circular  
    frente = (frente + 1) % MAXTAM;  
    tamanho--;
```

```
    return aux;
```

```
}
```

**frente = (7+1)%8 = 0**

```
FilaArranjo f  
TipoItem x;  
...  
f.Desenfileira(x)  
x.Imprime();
```



# Class FilaArranjo - Desenfileira

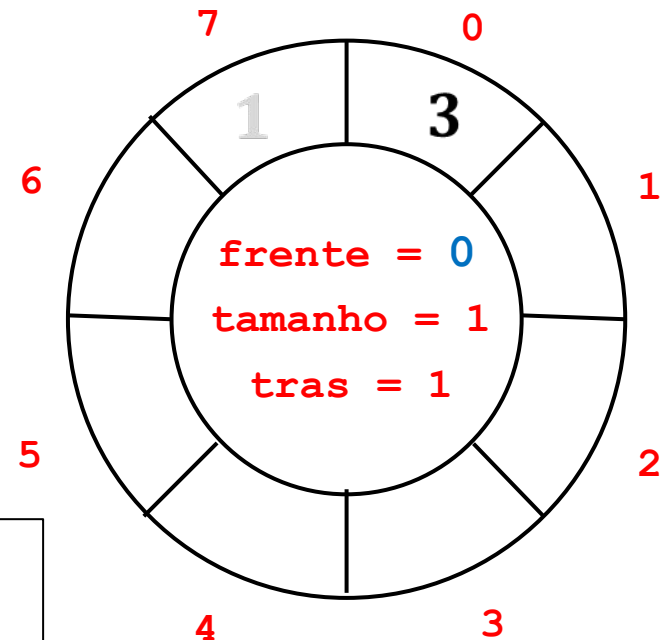
## ■ Desenfileira

- ❑ Testa se a Fila está vazia, gerando uma exceção
- ❑ Armazena o item da frente em aux
- ❑ Incrementa frente usando aritmética modular:  $\% \text{MAXTAM} (\%8)$
- ❑ Decrementa o tamanho e retorna aux

```
TipoItem FilaArranjo::Desenfileira() {  
    TipoItem aux;  
  
    if (tamanho == 0)  
        throw "Fila está vazia!";  
  
    aux = itens[frente];  
    // fila circular  
    frente = (frente + 1) % MAXTAM;  
    tamanho--;  
  
    return aux;  
}
```

1

```
FilaArranjo f  
TipoItem x;  
...  
f.Desenfileira(x)  
x.Imprime();
```



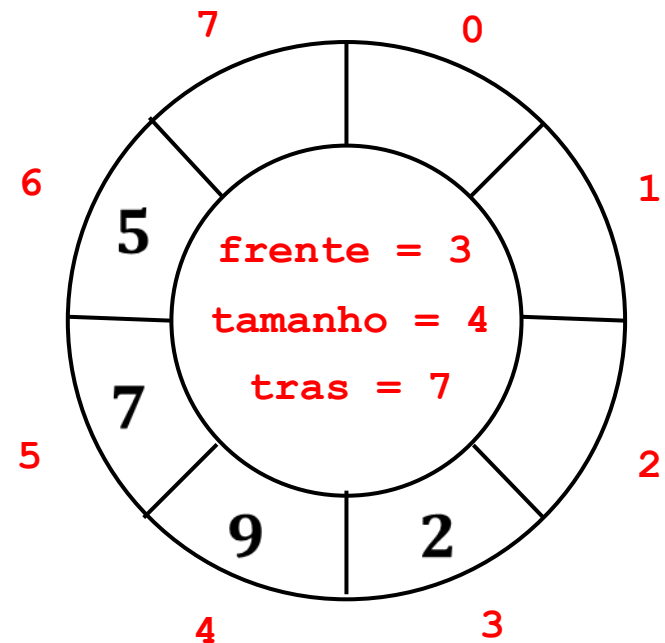
# Class FilaArranjo - Limpa

## ■ Limpa

- Apenas seta as variáveis tamanho, frente e tras para 0

```
void FilaArranjo::Limpa() {  
    frente = 0;  
    tras = 0;  
    tamanho = 0;  
}
```

```
FilaArranjo f  
...  
f.Limpa();
```



# Class FilaArranjo - Limpa

## ■ Limpa

- Apenas seta as variáveis tamanho, frente e tras para 0

```
void FilaArranjo::Limpa() {  
    frente = 0;  
    tras = 0;  
    tamanho = 0;  
}
```

```
FilaArranjo f  
...  
f.Limpa();
```

