DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estrutura de Dados

Pilhas e Filas

Professores: Luiz Chaimowicz e Raquel Prates

Listas, Pilhas e Filas

- Listas:
  - Inserção: em qualquer posição
  - Remoção: em qualquer posição
- Pilhas:
  - Inserção: Topo (primeira posição na lista)
  - Remoção: Topo (primeira posição na lista)
- Filas:
  - Inserção: Trás (última posição na lista)
  - Remoção: Início (primeira posição na lista)

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)*

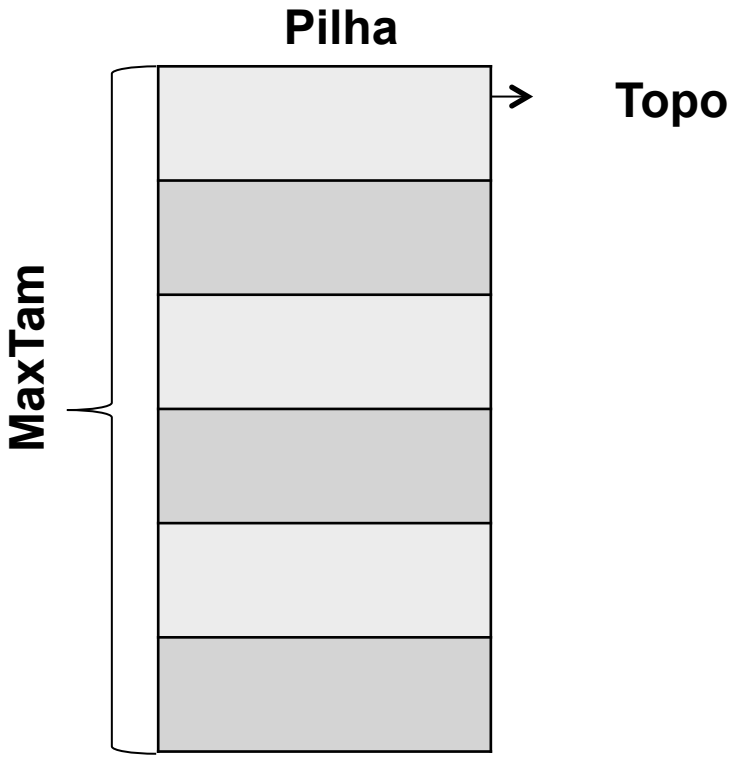
- Analogia: pilha de pratos, livros, etc
- Usos: Chamada de subprogramas, avaliação de expressões aritméticas, recursividade, etc...

TAD Pilhas

- **Conjunto de operações:**
  1) FPVazia(Pilha). Faz a pilha ficar vazia.
  2) Vazia(Pilha). Retorna true se a pilha está vazia; caso contrário, retorna false.
  3) Empilha(x, Pilha). Insere o item x no topo da pilha.
  4) Desempilha(Pilha). Retorna o item x no topo da pilha, retirando-o da pilha.
  5) Tamanho(Pilha). Esta função retorna o número de itens da pilha.

- **Representações comuns**
  - Alocação sequencial (arranjos) e apontadores

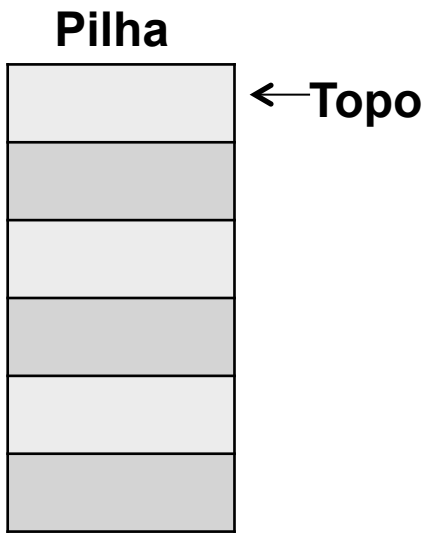
TAD Pilhas



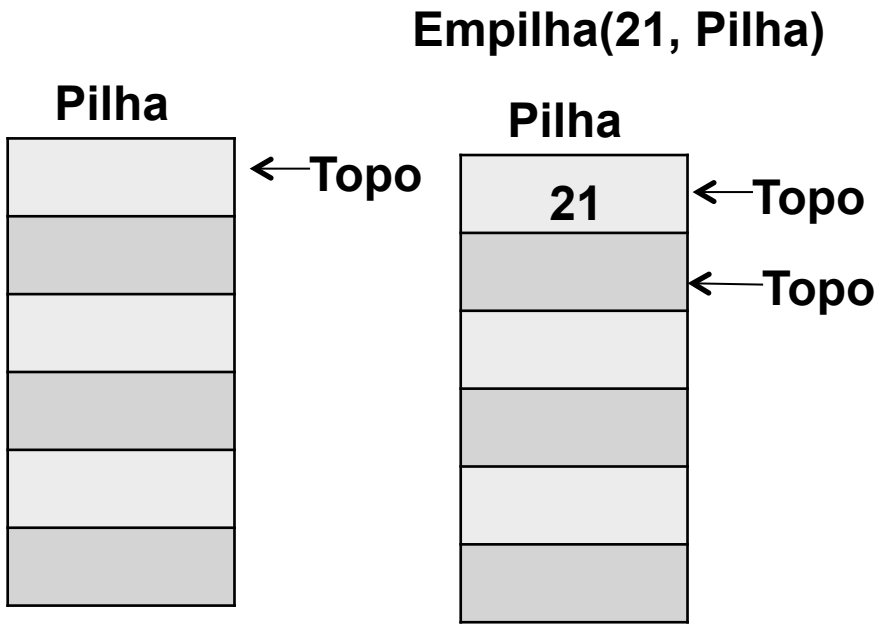
Implementação de Pilhas com Alocação Sequencial

- Os itens da pilha são armazenados em posições contíguas de memória.
- Inserções e retiradas**: apenas no topo**.
- Variável **Topo** é utilizada para controlar a posição do item no topo da pilha.

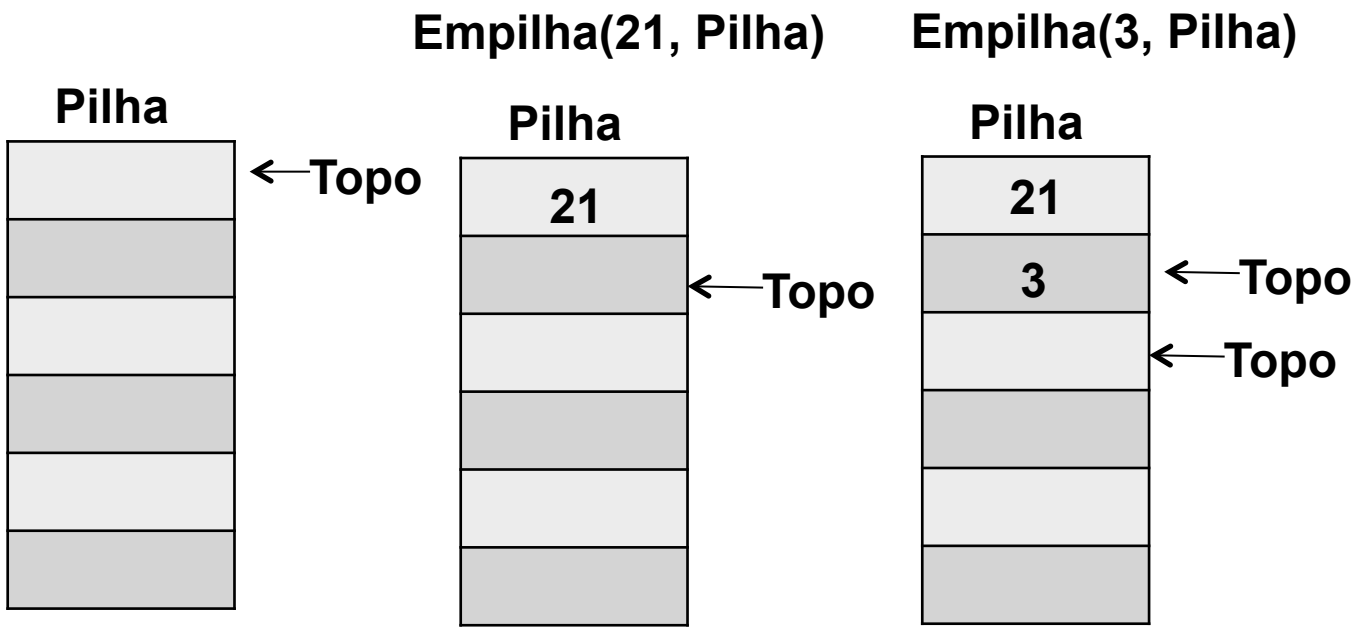
Implementação de Pilhas com Alocação Sequencial



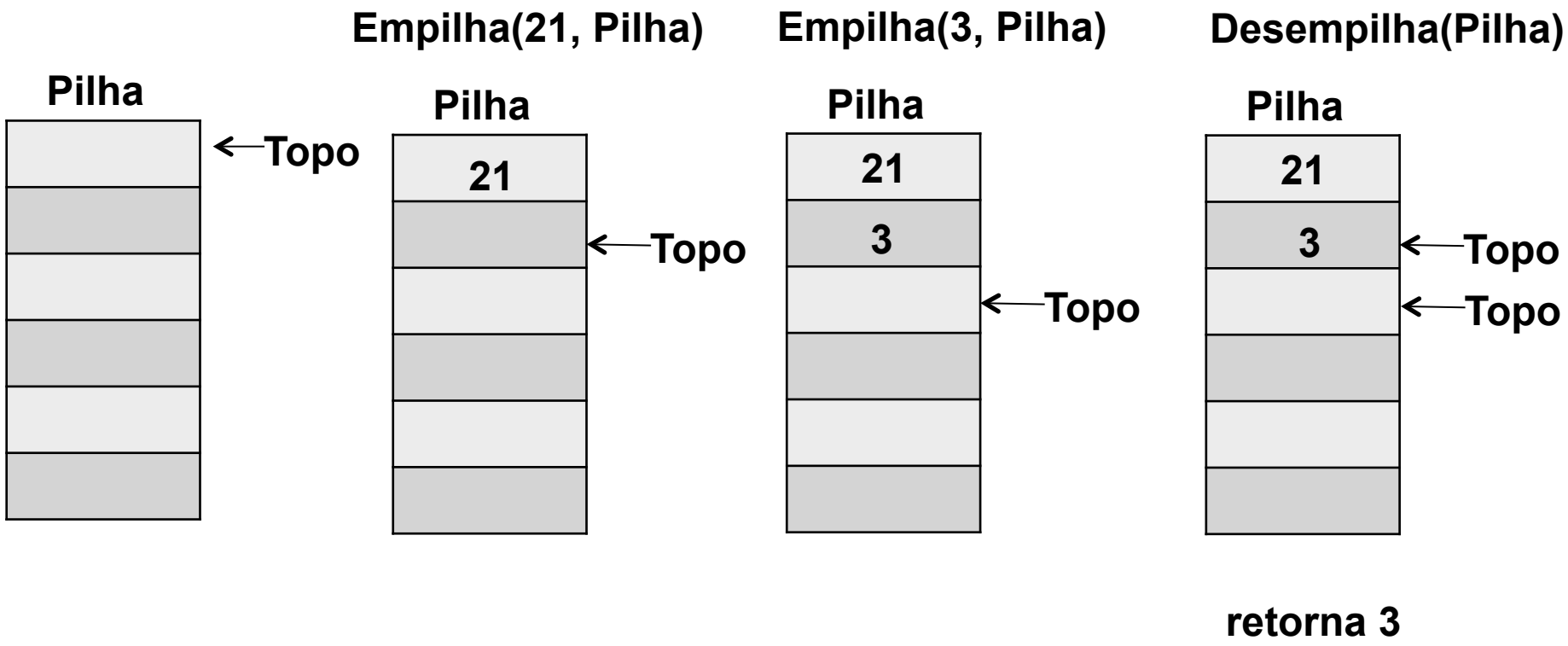
Implementação de Pilhas com Alocação Sequencial



Implementação de Pilhas com Alocação Sequencial

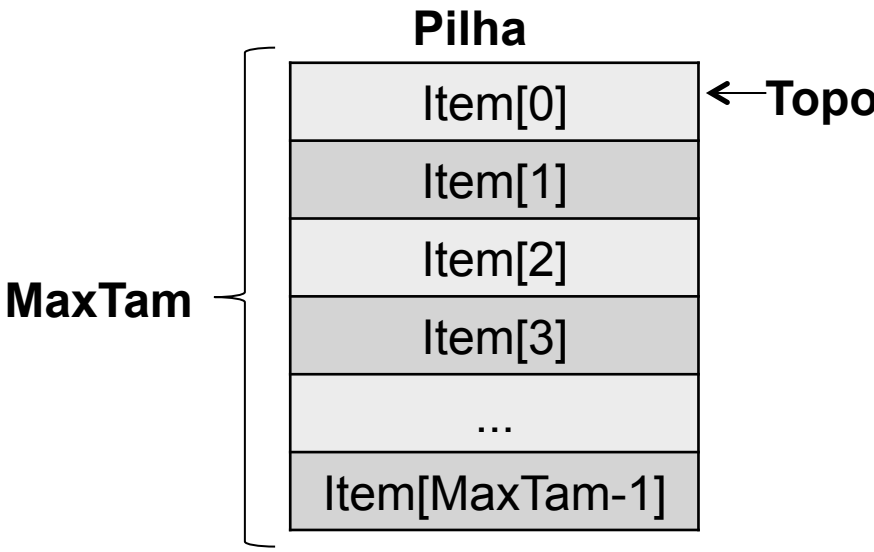


Implementação de Pilhas com Alocação Sequencial



Implementação de Pilhas com Alocação Sequencial

```
const MaxTam = 1000;

typedef int Apontador;
typedef int TipoChave;

typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;

typedef struct {
  TipoItem Item[MaxTam];
  Apontador Topo;
} TipoPilha;
```

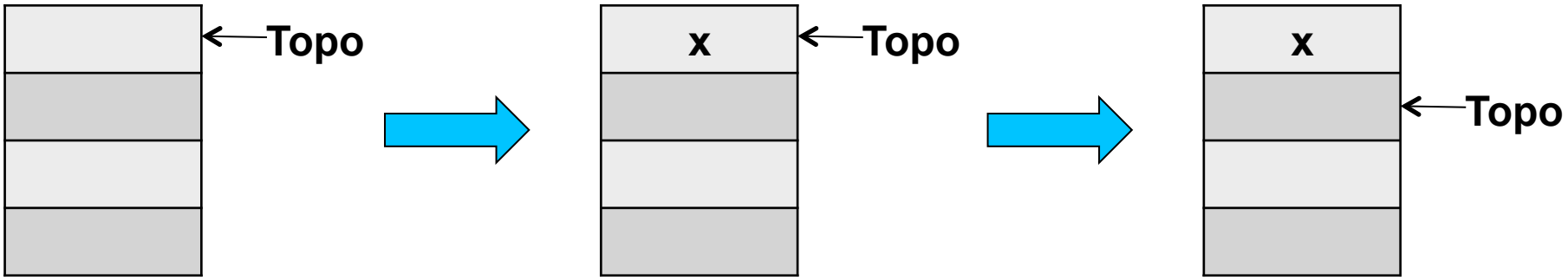


Implementação de Pilhas com Alocação Sequencial

```
void FPVazia(TipoPilha *Pilha){
  Pilha->Topo = 0;
}  /* FPVazia */


int Vazia(const TipoPilha *Pilha){
  return (Pilha->Topo == 0);
}  /* Vazia */
```

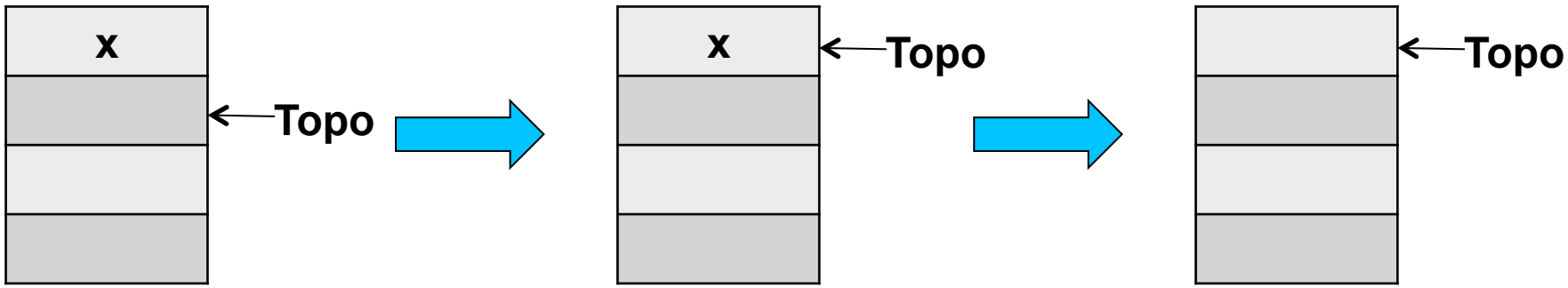
Implementação de Pilhas com Alocação Sequencial

```
void Empilha(TipoItem x, TipoPilha *Pilha){
  if (MaxTam == Pilha->Topo)
      printf("Erro: pilha está cheia\n");
  else {
      Pilha->Item[Pilha->Topo] = x;
      Pilha->Topo++;
  }
}
```



Operações sobre Pilhas com alocação Sequencial

```
void Desempilha(TipoPilha *Pilha, TipoItem *item){
  if (Vazia(Pilha))
      printf("Erro: a pilha está vazia\n");
  else {
      Pilha->Topo--;
      *item = Pilha->Item[Pilha->Topo];
  }
}
```



Operações sobre Pilhas Usando Arranjos

```
int Tamanho(const TipoPilha *Pilha){
  return Pilha->Topo;
}
```

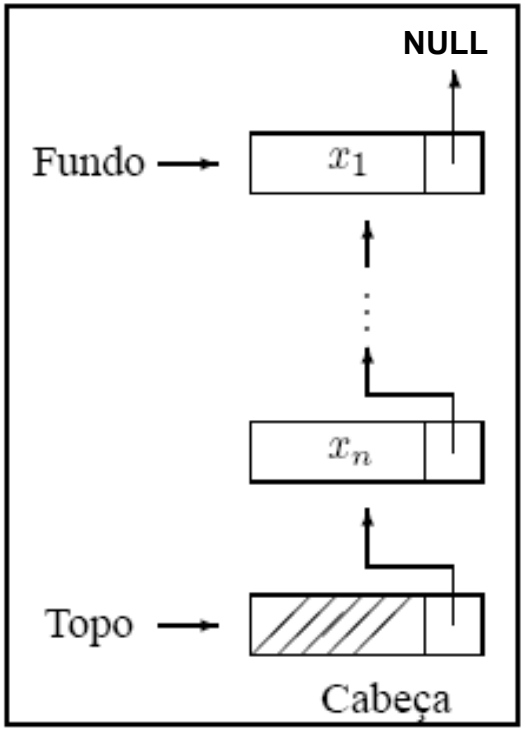
Pilhas: exercício

- Usando alocação de memória por meio de um vetor A[1..n].
- Implemente duas pilhas de tal forma que elas compartilhem o vetor A e não sofram overflow a não ser que o toal de elementos nas duas juntas seja maior do que n.
  - As operações PUSH e POP devem ter custo O(1)
  - C++ class twoStacks
    - twoStacks(int n)
    - void push1(int x), void push2(int x)
    - int pop1(), int pop2()

```
#include<iostream>
#include<stdlib.h>

using namespace std;

class twoStacks
{
    int *arr;
    int size;
    int top1, top2;
public:
   twoStacks(int n)  // constructor
   {
       size = n;
       arr = new int[n];
       top1 = -1;
       top2 = size;
   }

   // Method to push an element x to stack1
   void push1(int x)
   {
       // There is at least one empty space for new element
       if (top1 < top2 - 1)
       {
           top1++;
           arr[top1] = x;
       }
       else
       {
           cout << "Stack Overflow";
           exit(1);
       }
   }

   // Method to push an element x to stack2
   void push2(int x)
   {
       // There is at least one empty space for new element
       if (top1 < top2 - 1)
       {
           top2--;
           arr[top2] = x;
       }
       else
       {
           cout << "Stack Overflow";
           exit(1);
       }
   }

   // Method to pop an element from first stack
   int pop1()
   {
       if (top1 >= 0 )
       {
           int x = arr[top1];
           top1--;
           return x;
       }
       else
       {
           cout << "Stack UnderFlow";
           exit(1);
       }
   }

   // Method to pop an element from second stack
   int pop2()
   {
       if (top2 < size)
       {
           int x = arr[top2];
           top2++;
           return x;
       }
       else
       {
           cout << "Stack UnderFlow";
           exit(1);
       }
   }
};
```

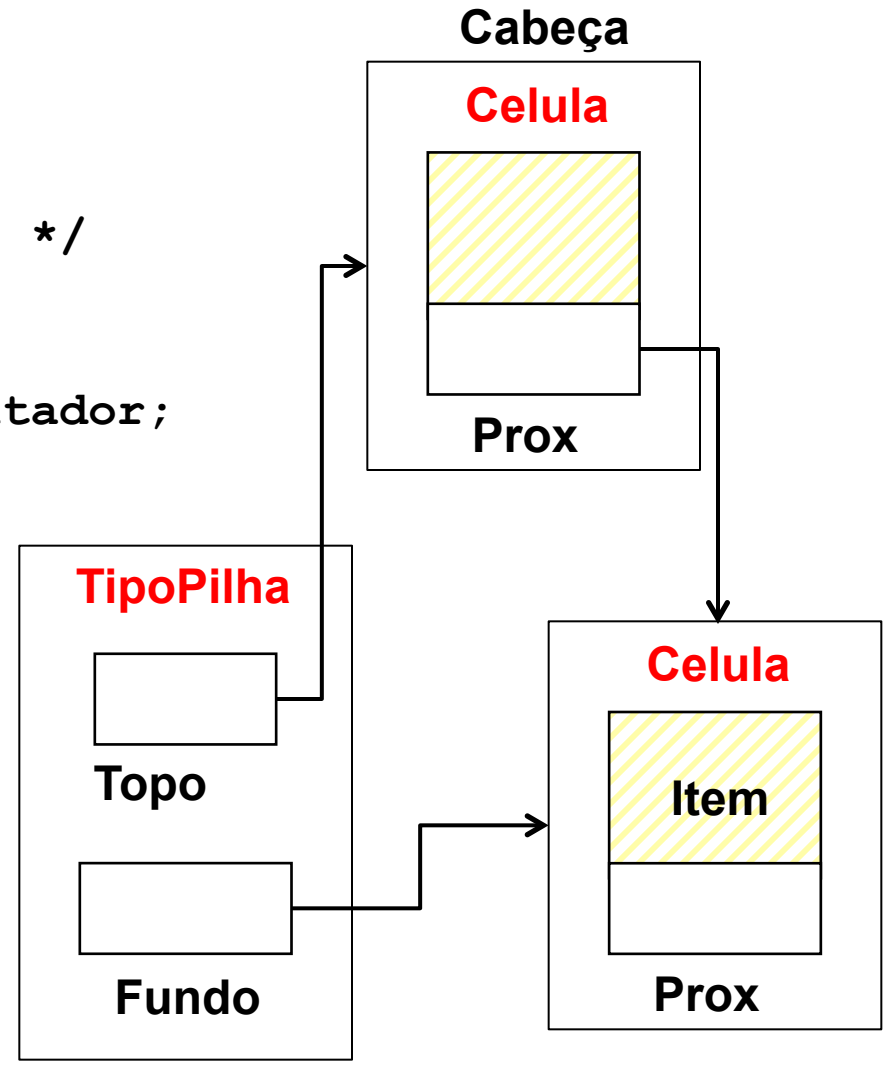
Implementação de Pilhas por meio de Apontadores

- Há uma **célula cabeça** no topo para facilitar a implementação das operações empilha e desempilha quando a pilha está vazia.
- **Desempilha**: desliga a célula cabeça da lista. A próxima célula, que contém o primeiro item, passa a ser a célula cabeça.
- **Empilha**: Cria uma nova célula cabeça e adiciona o item a ser empilhado na antiga célula cabeça.



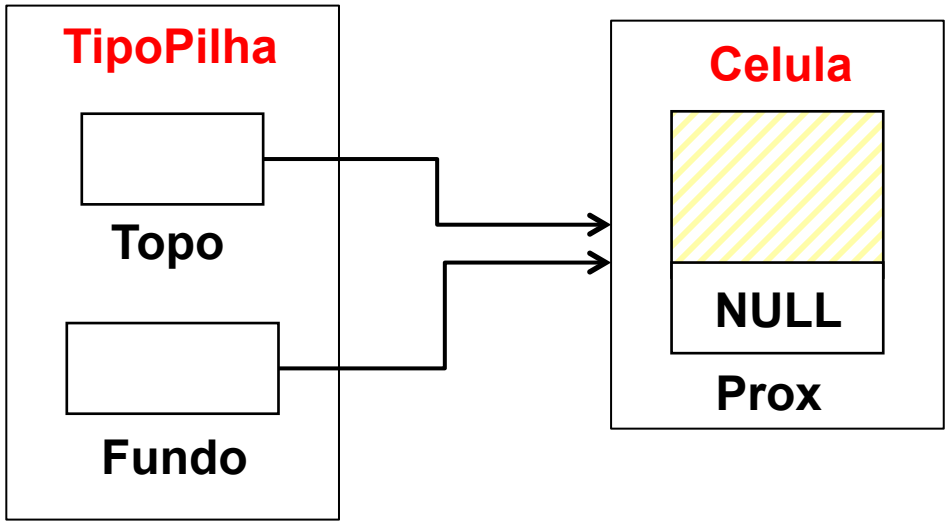
Estrutura da Pilha Usando Apontadores

```
typedef int TipoChave;
typedef struct {
  TipoChave Chave;
  /* --- outros componentes --- */
} TipoItem;

typedef struct Celula_str *Apontador;

typedef struct Celula_str {
  TipoItem Item;
  Apontador Prox;
} Celula;

typedef struct {
  Apontador Fundo, Topo;
  int Tamanho;
} TipoPilha;
```



Operações sobre Pilhas Usando Apontadores

```
void FPVazia(TipoPilha *Pilha) {
  Pilha->Topo = (Apontador) malloc(sizeof(Celula));
  Pilha->Fundo = Pilha->Topo;
  Pilha->Topo->Prox = NULL;
  Pilha->Tamanho = 0;
}  /* FPVazia */
```

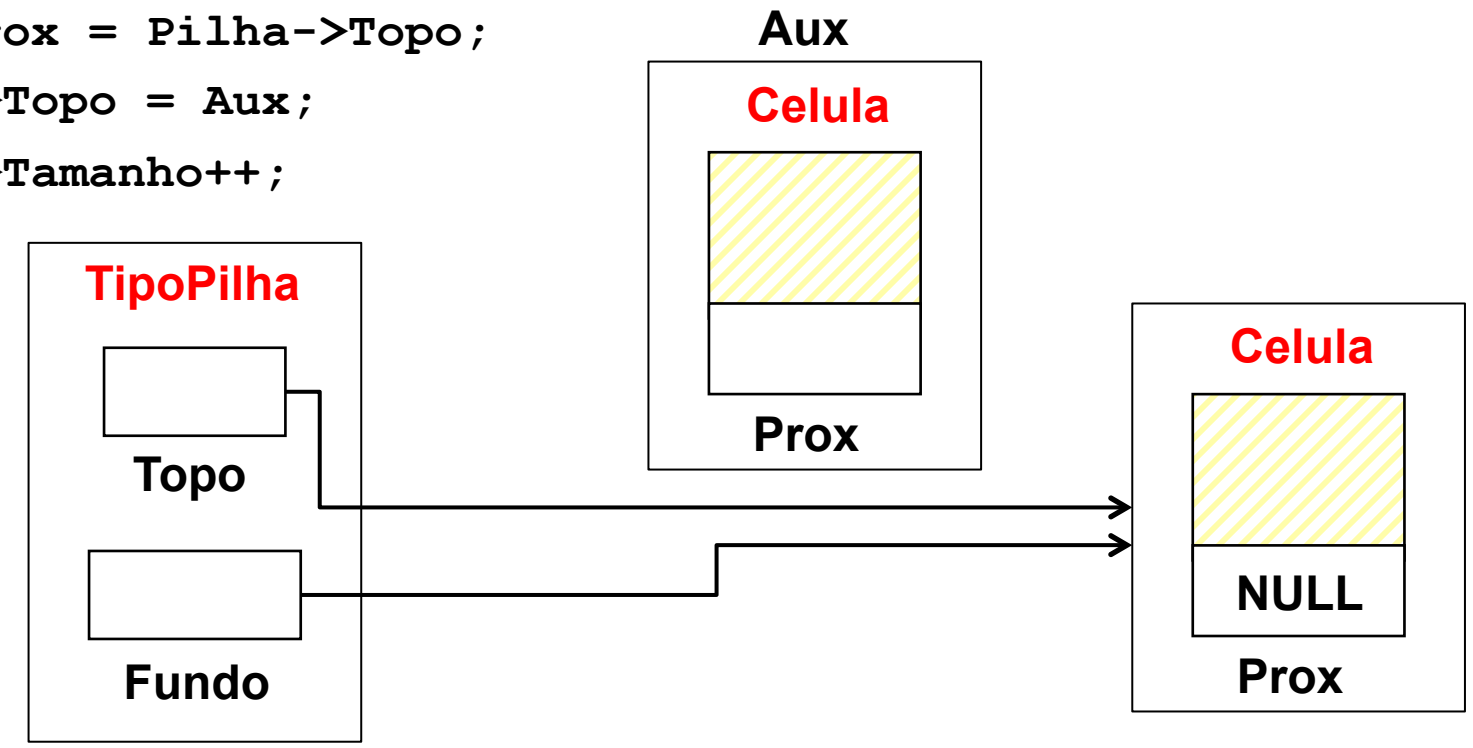


Operações sobre Pilhas Usando Apontadores

```
int Vazia(const TipoPilha *Pilha){
  return (Pilha->Topo == Pilha->Fundo);
}  /* Vazia */
```

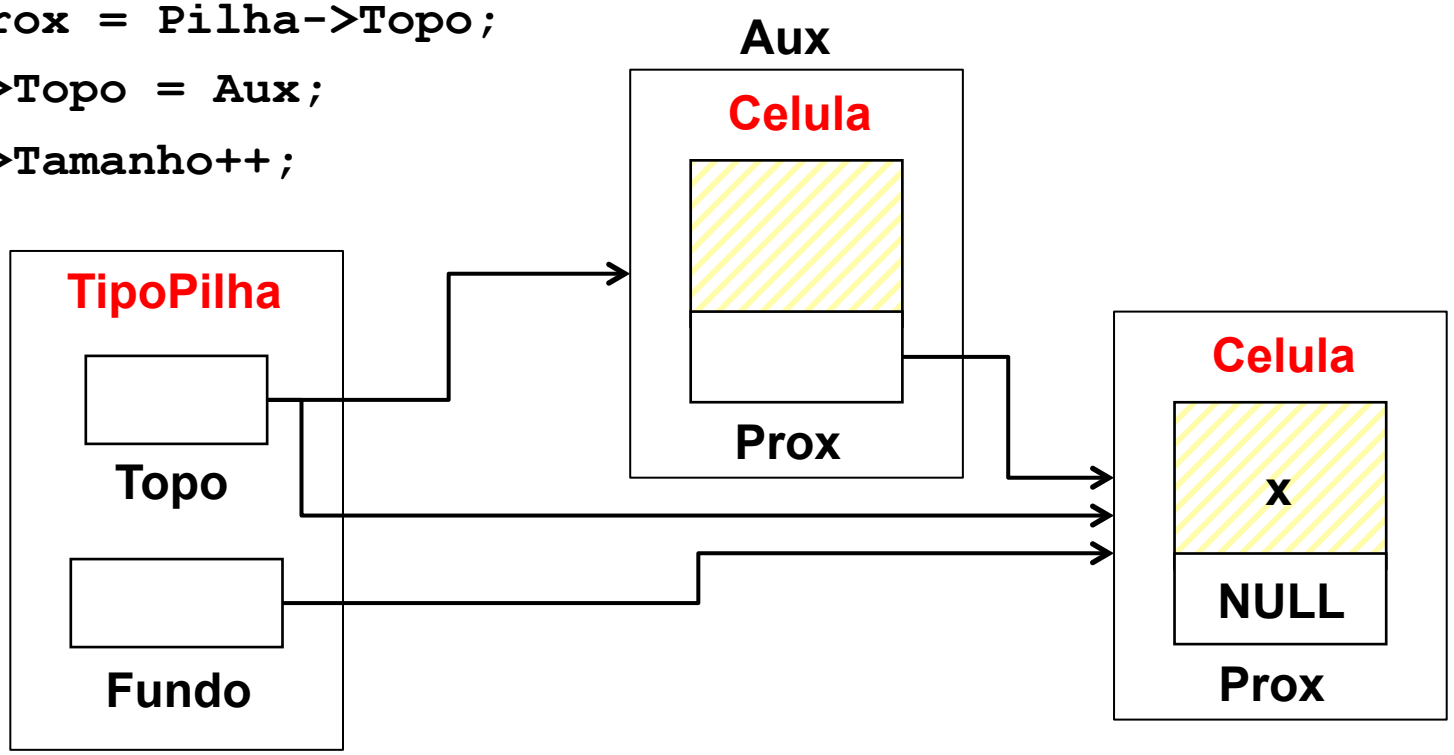
Operações sobre Pilhas Usando Apontadores

```
void Empilha(TipoItem x, TipoPilha *Pilha) {
  Apontador Aux;
  Aux = (Apontador) malloc(sizeof(Celula));
  Pilha->Topo->Item = x;
  Aux->Prox = Pilha->Topo;
  Pilha->Topo = Aux;
  Pilha->Tamanho++;
}
```



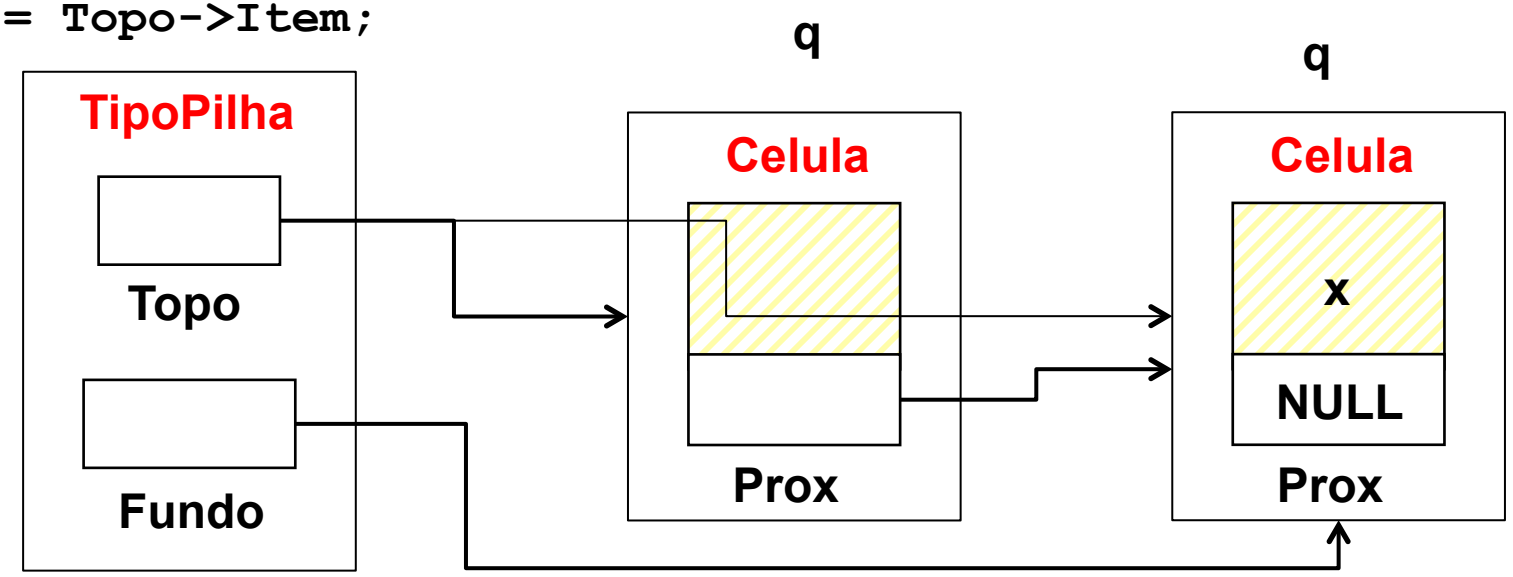
Operações sobre Pilhas Usando Apontadores

```
void Empilha(TipoItem x, TipoPilha *Pilha) {
  Apontador Aux;
  Aux = (Apontador) malloc(sizeof(Celula));
  Pilha->Topo->Item = x;
  Aux->Prox = Pilha->Topo;
  Pilha->Topo = Aux;
  Pilha->Tamanho++;
}
```



Operações sobre Pilhas Usando Apontadores

```
void Desempilha(TipoPilha *Pilha, TipoItem *item) {
  Apontador q;
  if (Vazia(Pilha)) {
          printf("Erro: pilha vazia\n");
  }
  q = Pilha->Topo;
  Pilha->Topo = q->Prox;
  free(q);
  Pilha->Tamanho--;
  *item = Topo->Item;
}
```

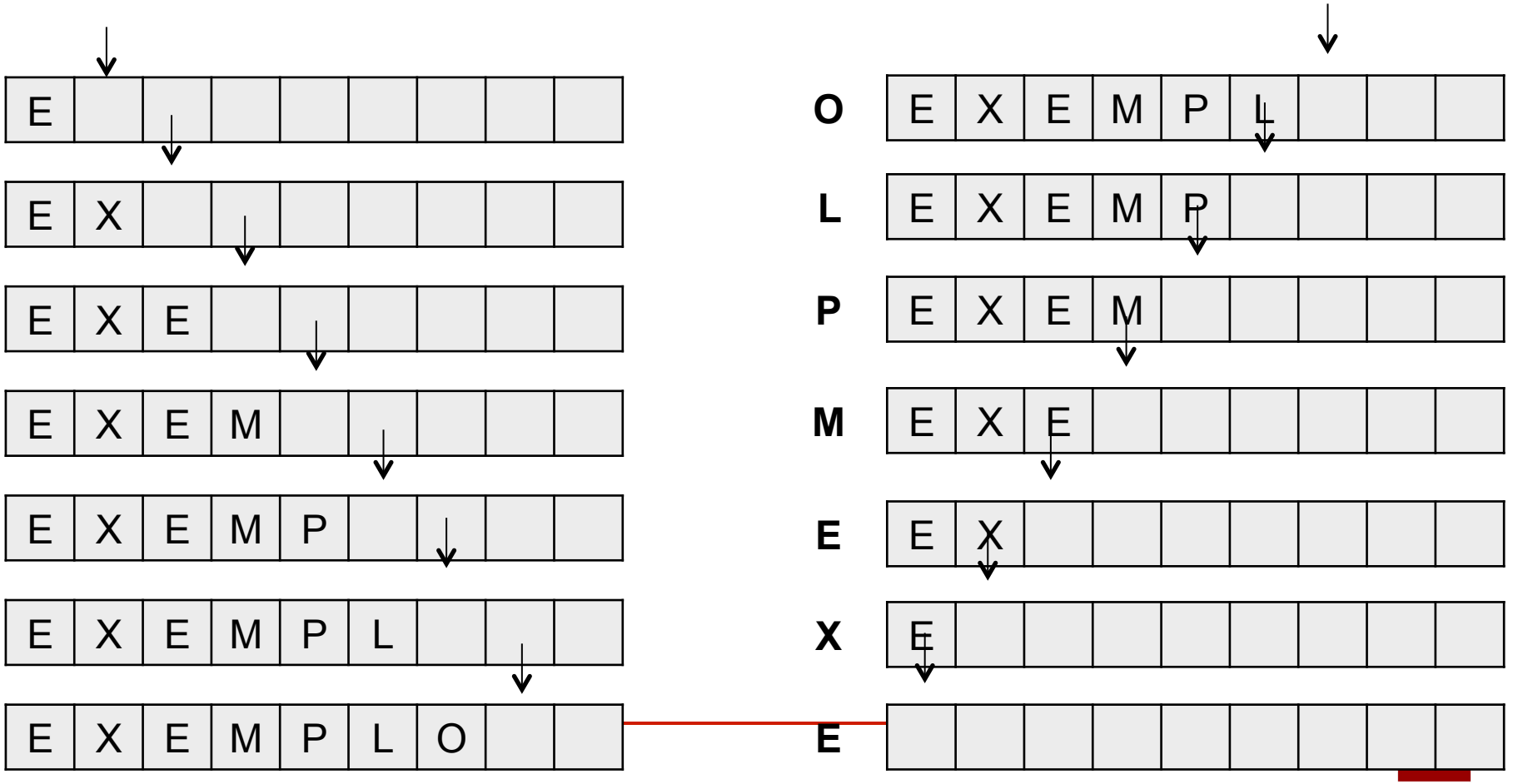


Operações sobre Pilhas Usando Apontadores

```
int Tamanho(const TipoPilha *Pilha){
  return (Pilha->Tamanho);
}  /* Tamanho */
```

Exemplos: Pilhas - Inversão de strings

- **Inverter a string “Exemplo” usando uma pilha.**
  1. **Empilha cada caractere em uma pilha vazia**
  2. **Desempilha todos elementos**



Exercício Pilhas: Conversão notação infixada pós-fixada

- Infixada: `(5 * (((9+8) * (4*6)) + 7))`
- Pós-fixada: `5 9 8 + 4 6 * * 7 + *`
  - Simplicidade: operandos precedem seus operadores
  - Não exige regras de precedência de operadores ou parênteses
- Utilizar uma pilha para converter de infixada para pós-fixada.

Exercício Pilhas: Conversão notação infixada pós-fixada

```
Converte(char *exp, TipoPilha *pilha){

  for (i = 0; i < N; i++) {
    if (exp[i] == ')')
      printf("%c ", desempilha(pilha))
    if (exp[i] == '+' || exp[i] == '*')
      empilha(exp[i], pilha);
    if (exp[i] >= '0' && exp[i] <= '9')
      printf("%c ", exp[i]);
}
```

Pilhas - Conversão notação infixada p/ pós-fixada

- **entrada:** `(5 * (((9+8) * (4*6)) + 7))`

Entrada	Pilha	saída
(		
5		5
*	*	
(((		
9		9
+	* +	
8		8
)	*	+
*	* *	
(		
4		4
*	* * *	
6		6
)	* *	*
)	*	*
+	* +	
7		7
)	*	+
)		*

saída: `5 9 8 + 4 6 * * 7 + *`

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

TAD Filas

- **Conjunto de operações:**
  1) FFVazia(Fila). Faz a fila ficar vazia.
  2) Enfileira(x, Fila). Insere o item x no final da fila.
  3) Desenfileira(Fila). Retorna o item x no início da fila, retirando-o da fila.
  4) Vazia(Fila). Esta função retorna true se a fila está vazia; senão retorna false.

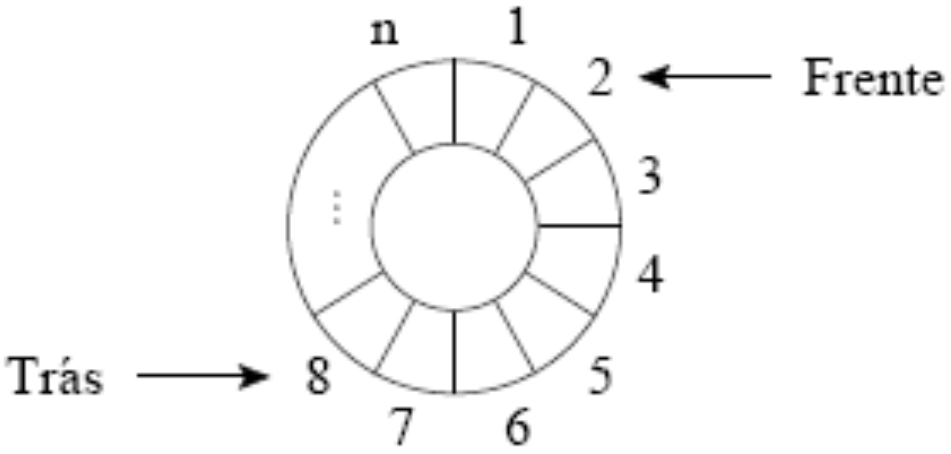
Implementação de Filas com alocação sequencial

Arranjos

- Os itens são armazenados em **posições contíguas** de memória.
- **Enfileira:** faz a parte de trás da fila expandir-se.
- **Desenfileira:** faz a parte da frente da fila contrair-se.
- A fila tende a se movimentar pela memória do computador, **ocupando espaço na parte de trás e descartando espaço na parte da frente.**

Implementação de Filas com alocação sequencial

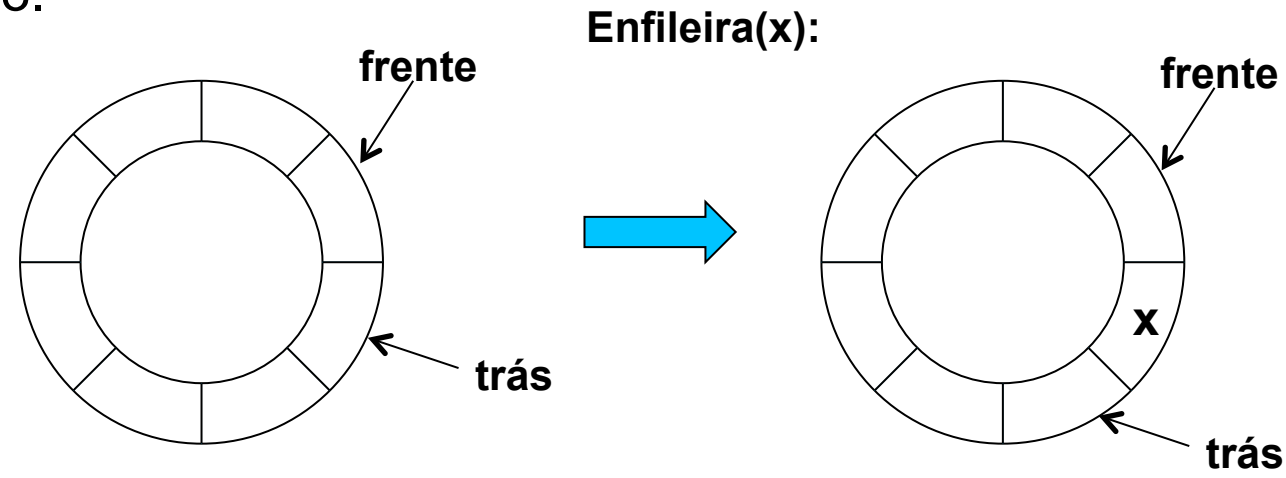
- Com poucas inserções e retiradas, a fila vai ao encontro do limite do espaço da memória alocado para ela.
- **Solução**: imaginar o vetor como um círculo. A primeira posição segue a última.



Implementação de Filas com Alocação Sequencial

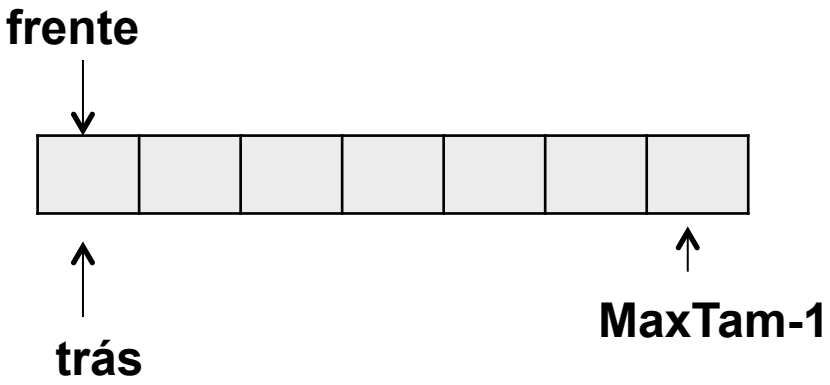
- A fila se encontra em posições contíguas de memória, em alguma posição do círculo, delimitada pelos apontadores Frente e Trás.
  - **Frente**: posição do primeiro elemento, **trás**: a primeira posição vazia)

- **Enfileirar:** mover o apontador Trás uma posição no sentido horário.
- **Desenfileirar:** mover o apontador Frente uma posição no sentido horário.



Estrutura da Fila com Alocação Sequencial

```
const MaxTam = 1000;

typedef int Apontador;
typedef int TipoChave;

typedef struct {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;

typedef struct {
  TipoItem Item[MaxTam];
  Apontador Frente, Tras;
} TipoFila;
```



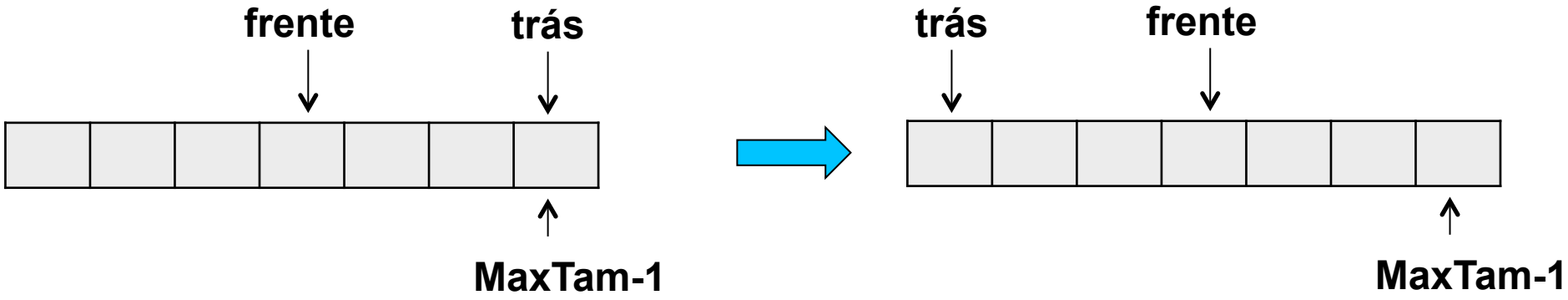
Operações sobre Filas com Alocação Sequencial

- Nos casos de fila cheia e fila vazia, os apontadores Frente e Trás apontam para a mesma posição do círculo.
- Uma saída para distinguir as duas situações é deixar uma posição vazia no array.
- Neste caso, a fila está cheia quando Trás+1 for igual a Frente.

```
void FFVazia(TipoFila *Fila){
  Fila->Frente = 0;
  Fila->Tras = Fila->Frente;
}  /* FFVazia */

int Vazia(const TipoFila *Fila){
  return (Fila->Frente == Fila->Tras);
}  /* Vazia */
```

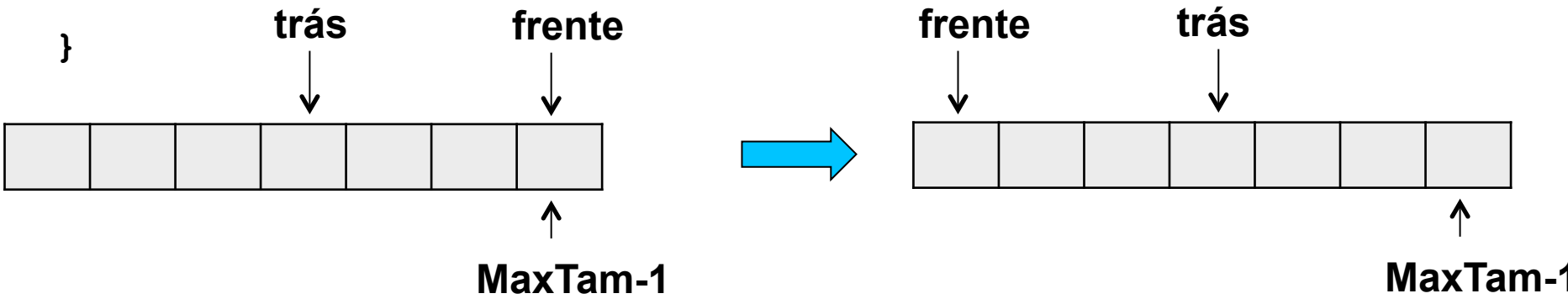
Operações sobre Filas com Alocação Sequencial

```
int Enfileira(TipoItem x, TipoFila *Fila) {
  if ((Fila->Tras+1) % MaxTam == Fila->Frente){
     printf("Erro: fila está cheia\n"); return 0;
  } else {
     Fila->Item[Fila->Tras] = x;
     Fila->Tras = (Fila->Tras + 1) % MaxTam;
  }
  return 1;
}
```



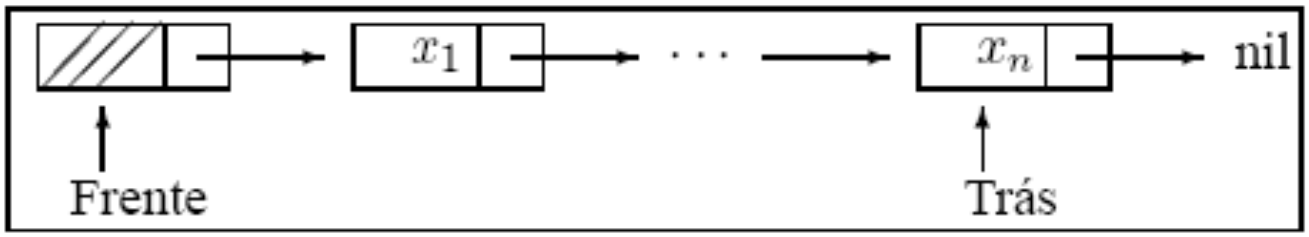
Operações sobre Filas com Alocação Sequencial

```
int Desenfileira(TipoFila *Fila, TipoItem *item) {
  if (Vazia(Fila)){
      printf("Erro: fila está vazia\n"); return 0;
  } else {
      int idx = Fila->Frente;
      Fila->Frente = (Fila->Frente + 1) % MaxTam;
      *item = Fila->Item[idx];
            return 1;
  }
}
```



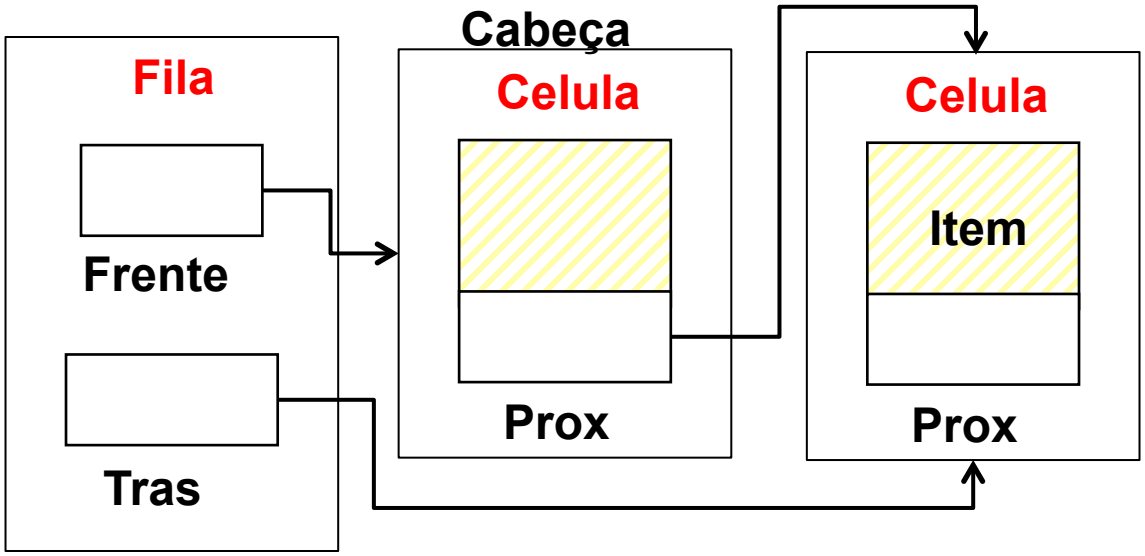
Implementação de Filas por meio de Apontadores

- Utiliza célula cabeça para facilitar a implementação das operações Enfileira e Desenfileira quando a fila está vazia.
- **Quando vazia**: apontadores Frente e Trás apontam para a célula cabeça.
- **Enfileirar novo item**: criar uma célula nova, ligá-la após a célula contendo x_n e colocar nela o novo item.
- **Desenfileirar**: desligar a célula cabeça da lista e a célula que contém x_1 passa a ser a célula cabeça.



Estrutura da Fila Usando Apontadores

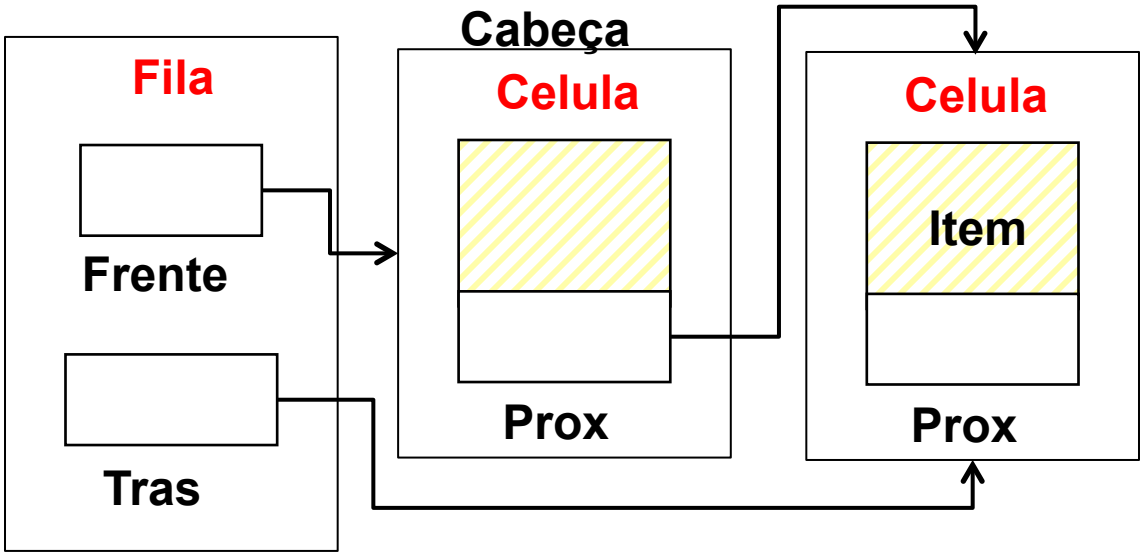
- A fila é implementada por meio de células.
- Cada célula contém um item da fila e um apontador para outra célula.
- A estrutura TipoFila contém um apontador para a frente da fila (célula cabeça) e um apontador para a parte de trás da fila.



Estrutura da Fila Usando Apontadores

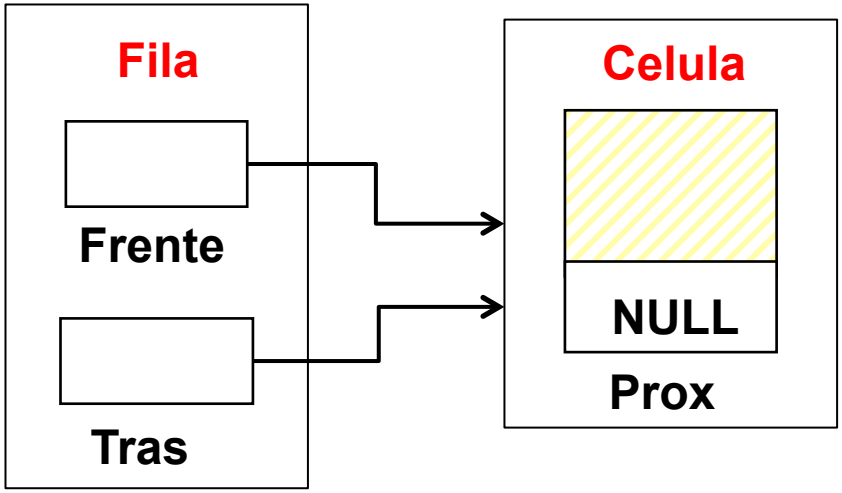
```
typedef int TipoChave;

typedef struct TipoItem {
  TipoChave Chave;
  /* outros componentes */
} TipoItem;
```

```
typedef struct Celula *Apontador;

typedef struct Celula {
  TipoItem Item;
  Apontador Prox;
} Celula;

typedef struct TipoFila {
  Apontador Frente, Tras;
} TipoFila;
```



Operações sobre Filas Usando Apontadores

```
void FFVazia(TipoFila *Fila){
  Fila->Frente = (Apontador) malloc(sizeof(Celula));
  Fila->Tras = Fila->Frente;
  Fila->Frente->Prox = NULL;
}  /* FFVazia */
```

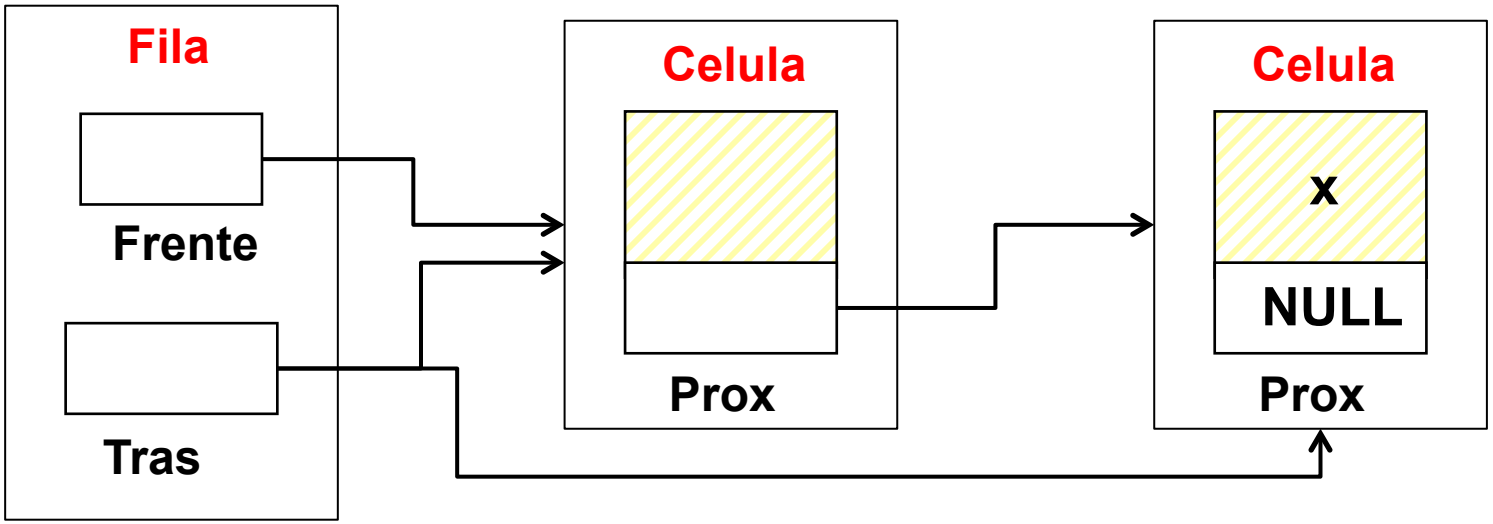


Operações sobre Filas Usando Apontadores

```
int Vazia(const TipoFila *Fila){
      return (Fila->Frente == Fila->Tras);
}  /* Vazia */
```

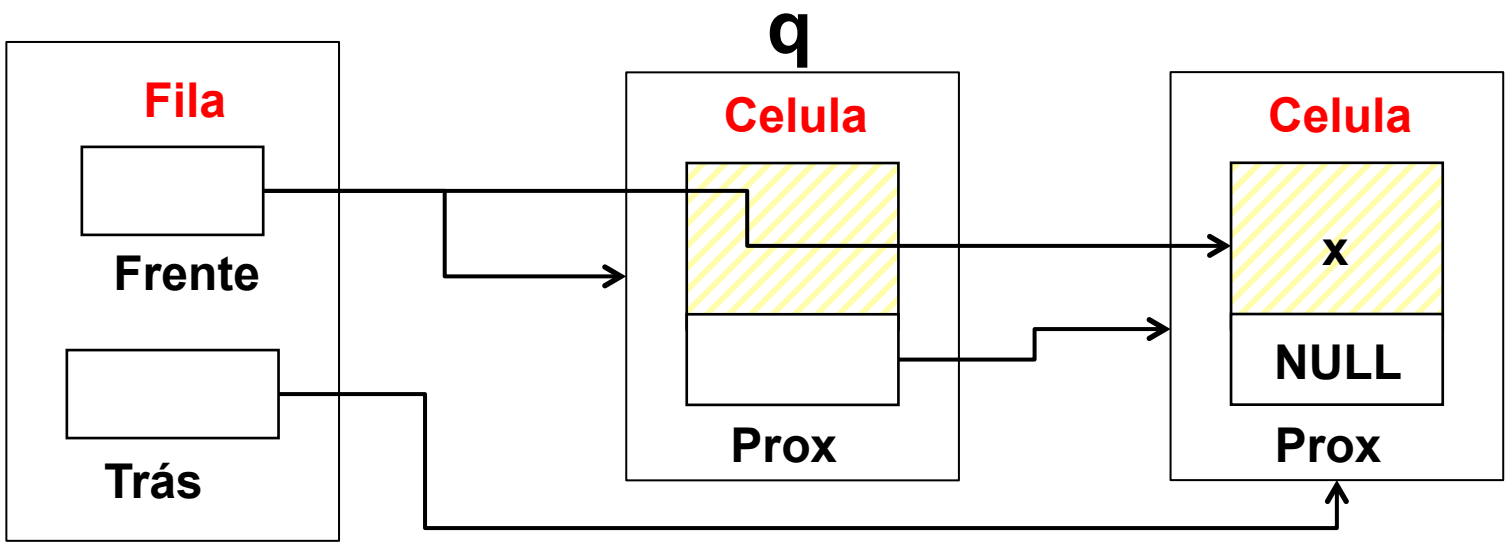
Operações sobre Filas Usando Apontadores

```
void Enfileira(TipoItem x, TipoFila *Fila){
  Fila->Tras->Prox = (Apontador) malloc(sizeof(Celula));
  Fila->Tras = Fila->Tras->Prox;
  Fila->Tras->Item = x;
  Fila->Tras->Prox = NULL;
}
```



Operações sobre Filas Usando Apontadores

```
int Desenfileira(TipoFila *Fila, TipoItem *item){
        if (Vazia(Fila)) {
                printf("Erro: fila está vazia\n");
                return 0;
        }
        Aprontador q = Fila->Frente;
        Fila->Frente = Fila->Frente->Prox;
        free(q);
        *item = Fila->Frente->Item;
        return 1;
}
```



Exercício: FIFO usando LIFO

- Você possui a estrutura de dados Pilha, com operações POP e PUSH com custo O(1).
- Implemente a estrutura de dados Fila usando os métodos de Pilha.
- Analise os custos das operações
  - void enfileira(q, x)
  - int desenfileira(q)

Exercício: FIFO com duas Pilhas

- Método 1 (enfileirar é caro, O(n))
  - void enfileira(q,x):
    - Enquanto pilha1 não vazia: pilha2.push(pilha1.pop());
    - pilha1.push(x);
    - Enquanto pilha2 não vazia: pilha1.push(pilha2.pop());
  - int desenfileira(q):
    - Se pilha1 vazia, retorne erro;
    - Retorne pilha1.pop();
- Método 2 (desenfileirar é caro, O(n))
  - void enfileira(q,x):
    - Pilha1.push(x);
  - int desenfileira(q):
    - Se pilha1 e pilha2 vazias, retorne erro;
    - Se pilha2 vazia: Enquanto pilha1 não vazia: pilha2.push(pilha1.pop());
    - Retorne pilha2.pop();