# Outras Estruturas

Algoritmos e Estrutura de Dados 1
Prof. Luiz Gustavo Almeida Martins

# Fila de Prioridade

- Estrutura de Dados na qual a classificação intrínseca dos elementos determina os resultados das operações básicas.
  - **Analogia:** pode ser vista como uma mistura de **Lista** com critério de ordenação implícito e **Fila.**
  - Prioridade **afeta a inserção ou a remoção.**
- ∃ 2 tipos básicos:
  - **Fila de Prioridade Ascendente (FPA):**
    - O elemento com **MENOR** "prioridade" deve ser removido.
  - **Fila de Prioridade Descendente (FPD):**
    - O elemento com **MAIOR** "prioridade" deve ser removido.

# Fila de Prioridade

- 2 abordagens para implementar FPA ou FPD:

1. **Estrutura SEM critério de ordenação**

2. **Estrutura COM critério de ordenação**

# Fila de Prioridade

- **Fila de prioridades não-ordenada:**
  - **Inserção no final** (como na fila)
  - **Remoção de acordo com a prioridade**
    - Busca o elemento com menor/maior prioridade e o remove da estrutura
  - Demais operações básicas são **IDÊNTICAS** as usadas para a estrutura **fila**

# Fila de Prioridade

- **Fila de prioridades ordenada:**
  - **Inserção ordenada**
    - Busca a posição adequada para o elemento de acordo com a sua prioridade (valor)
  - **Remoção no início** (como na fila)
  - Demais operações básicas são **IDÊNTICAS** as usadas para a estrutura **fila**

# Fila de Prioridade

- Análise da "complexidade" das 2 abordagens:
  - **1ª Abordagem (não-ordenada):**
    - **Inserção**: 1 passo
    - **Remoção**: ***N*** passos, para fila de ***N*** elementos
  - **2ª Abordagem (ordenada):**
    - **Inserção**: ***K*** passos, com ***K*** variando de 1 a ***N***
    - **Remoção**: 1 passo

# Fila de Prioridade

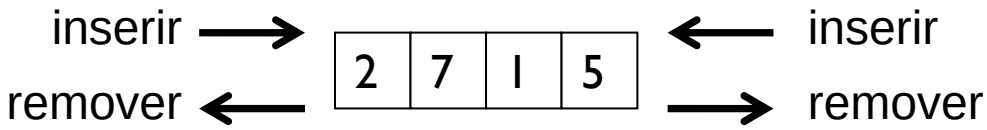
- **Notação BIG-O (análise do pior caso):**
  - **1ª Abordagem:** 1 passo inserção e ***N*** passos remoção
  - **2ª Abordagem:** ***N*** passos inserção e 1 passo remoção
- Portanto, ambas têm complexidade **O(N)**

# Fila de Prioridade

- **Análise do caso médio:**
  - **1ª Abordagem:** ***N*** passos remoção
  - **2ª Abordagem:** ***N*/2** passos inserção
- Portanto, **2ª abordagem é + vantajosa para o caso médio**

# Fila Dupla ou Deque

- **Estrutura que permite remoção/inserção nas 2 extremidades**




- É como se em uma mesma estrutura existissem **duas filas**, uma inversa da outra.

# Fila Dupla ou Deque

- **TAD: operações básica**
  - cria_deque
  - deque_vazio
  - deque_cheio
  - inserir_início
  - inserir_final
  - remover_início
  - remover_final

# Fila Dupla ou Deque

- **Técnicas de Implementação:**

  **a) Estática/Sequencial:**

  – Uso do **incremento circular** (insere_final e remove_início) e do **decremento circular** (insere_início e remove_final)

  – **Decremento circular:**

$$F\Theta 1 = \begin{cases} F-1, \text{ se } F > 0 \\ Max-1, \text{ se } F = 0 \end{cases}$$

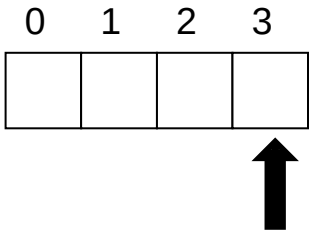
# Fila Dupla ou Deque

- **Diferenciação entre deque vazio e cheio** adota as abordagens de fila:
  - **Desprezar 1 posição:**
    - **Deque vazio:** ini = fim
    - **Deque cheio:** ini =(fim+1) % max
  - **Uso de contador:**
    - **Deque vazio:** cont = 0
    - **Deque cheio:** cont = MAX

# Fila Dupla ou Deque

- **Exemplo: Solução que despreza 1 posição**

1) Inicializa



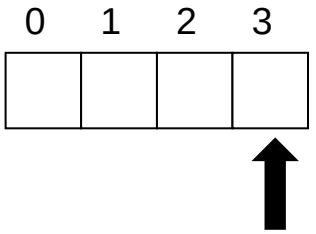

max = 4

ini = fim = max - 1

# Fila Dupla ou Deque

- **Exemplo: Solução que despreza 1 posição**

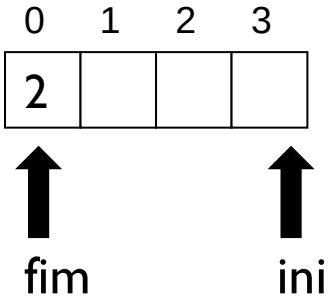
1) Inicializa




max = 4

ini = fim = max - 1

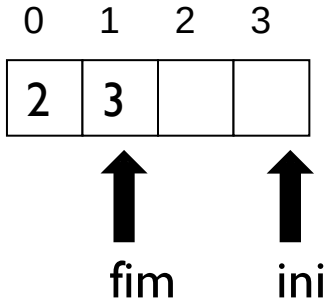
2) Insere_final(2)




fim = fim ⊕ 1
Vet[fim] = 2

# Fila Dupla ou Deque

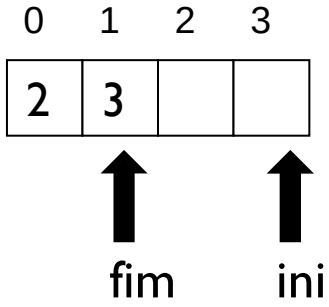
3) Insere_final(3):




fim = fim ⊕ 1
Vet[fim] = 3

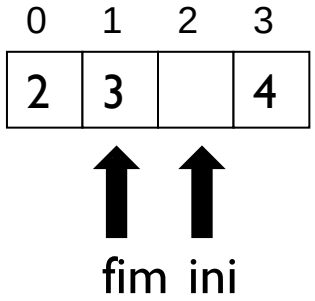
# Fila Dupla ou Deque

3) Insere_final(3):




fim = fim ⊕ 1
Vet[fim] = 3

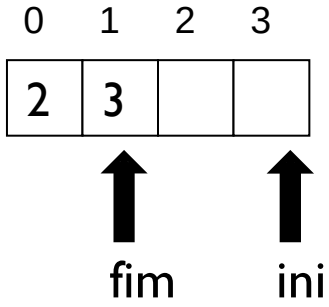
4) Insere_início(4):




Vet[ini] = 4
ini = ini ⊖ 1

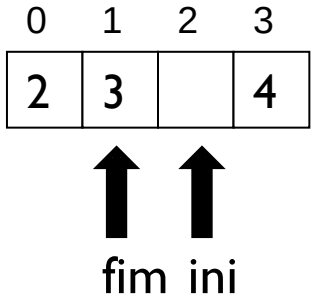
# Fila Dupla ou Deque

3) Insere_final(3):




fim = fim ⊕ 1
Vet[fim] = 3

4) Insere_início(4):

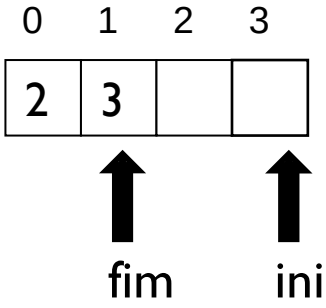



Vet[ini] = 4
ini = ini ⊖ 1

5) Insere_final(5): **Falha** – Fila Cheia (ini = fim ⊕ 1)

# Fila Dupla ou Deque

6) remove_início(&x):



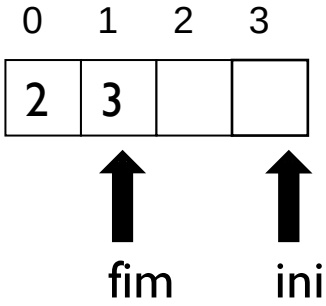

x = 4

ini = ini ⊕ 1

x = vet[ini]

# Fila Dupla ou Deque

6) remove_início(&x):

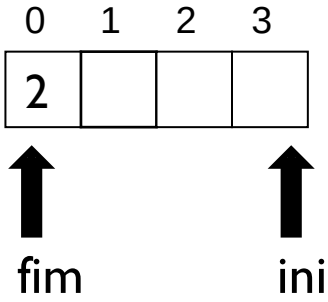



x = 4

ini = ini ⊕ 1

x = vet[ini]

7) remove_final(&x):



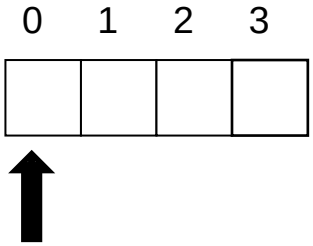

x = 3

x = vet[fim]

fim = fim ⊖ 1

# Fila Dupla ou Deque

8) remove_início(&x):



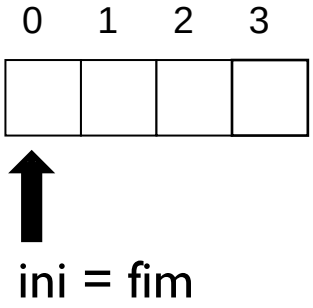

x = 2

ini = ini ⊕ 1

x = vet[ini]

# Fila Dupla ou Deque

8) remove_início(&x):




x = 2

ini = ini ⊕ 1

x = vet[ini]

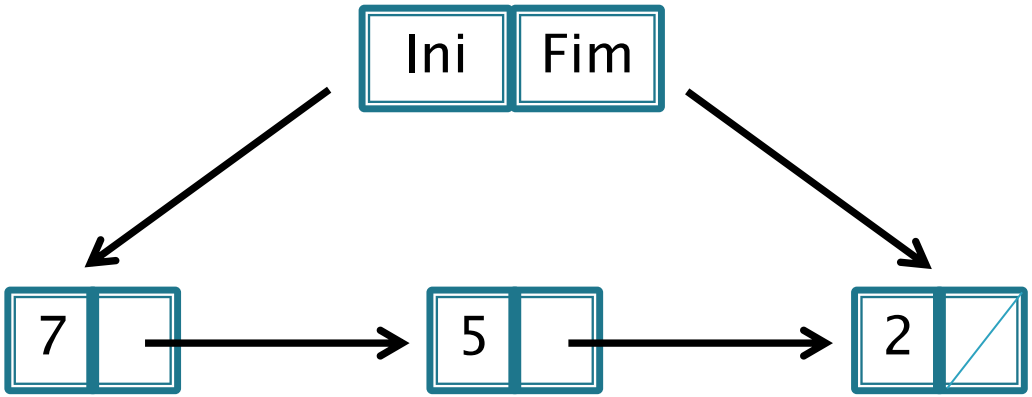
9) remove_final(&x): **Falha** – Fila vazia (ini = fim)

# Fila Dupla ou Deque

**b) Dinâmica/Encadeada:**

- **Encadeamento simples:** não é eficiente para a **remoção no final**

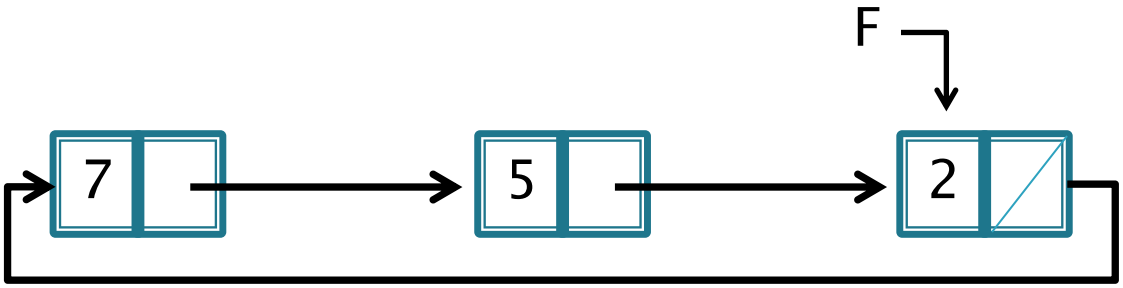
**Problema:** Qual nó antecede Fim?



# Fila Dupla ou Deque

- **Encadeamento circular**: também é ineficiente para a **remoção no final**

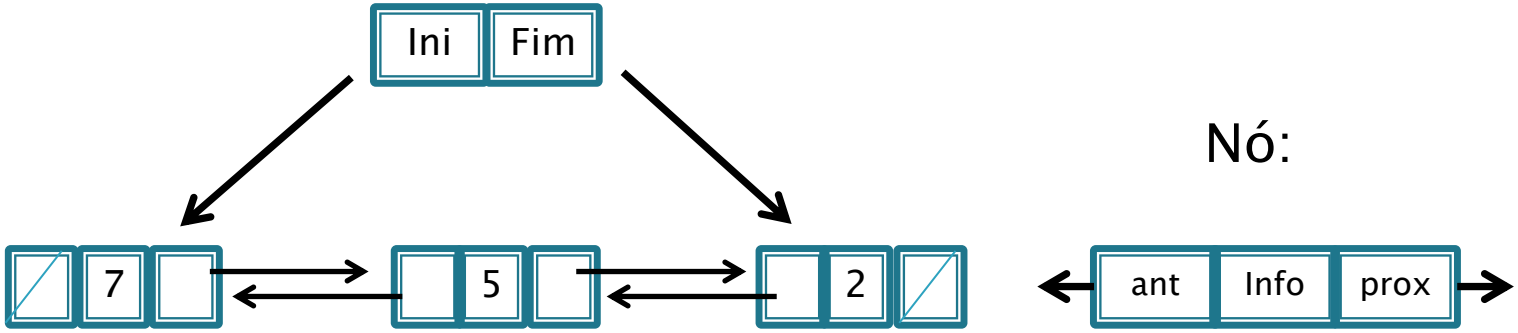
**Problema:** Qual nó antecede F?



# Fila Dupla ou Deque

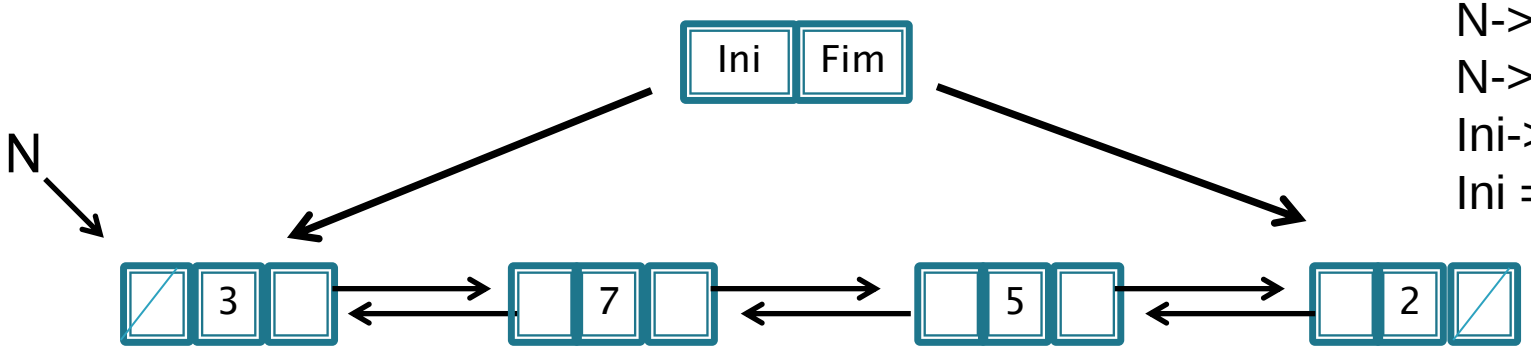
- Solução usual: USO DE ENCADEMENTO DUPLO

Ex: encadeamento duplo simples



# Fila Dupla ou Deque

Insere_início(3):

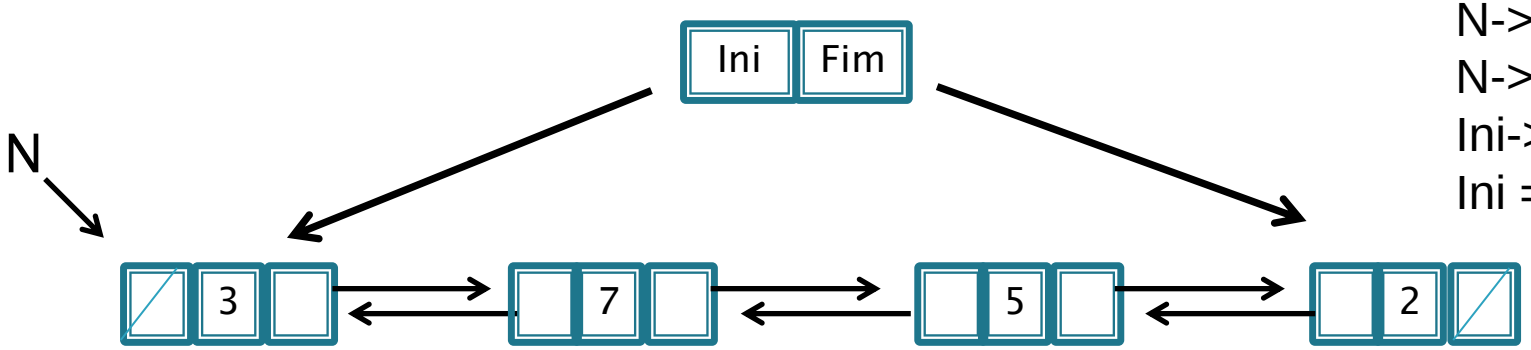



Aloca novo nó N
N->info = 3
N->prox = Ini
N->ant = NULL
Ini->ant = N
Ini = N

# Fila Dupla ou Deque

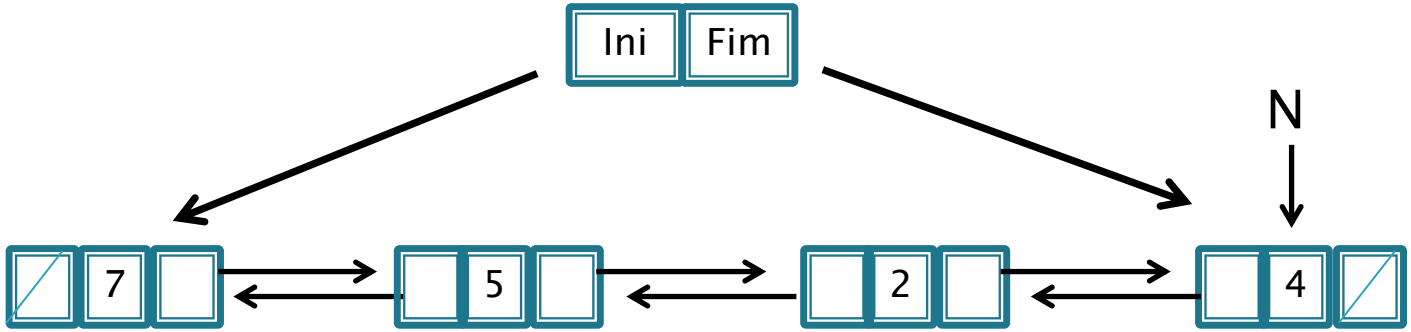
## Insere_início(3):

Aloca novo nó N
N->info = 3
N->prox = Ini
N->ant = NULL
Ini->ant = N
Ini = N




## Insere_final(4):

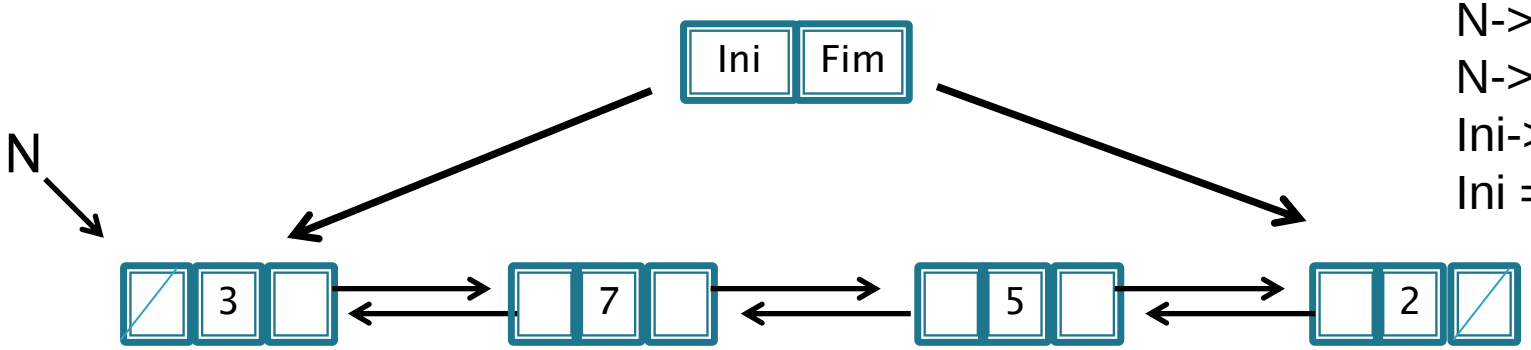
Aloca novo nó N
N->info = 4
N->prox = NULL
N->ant = Fim
Fim->prox = N
Fim = N



# Fila Dupla ou Deque

## Insere_início(3):

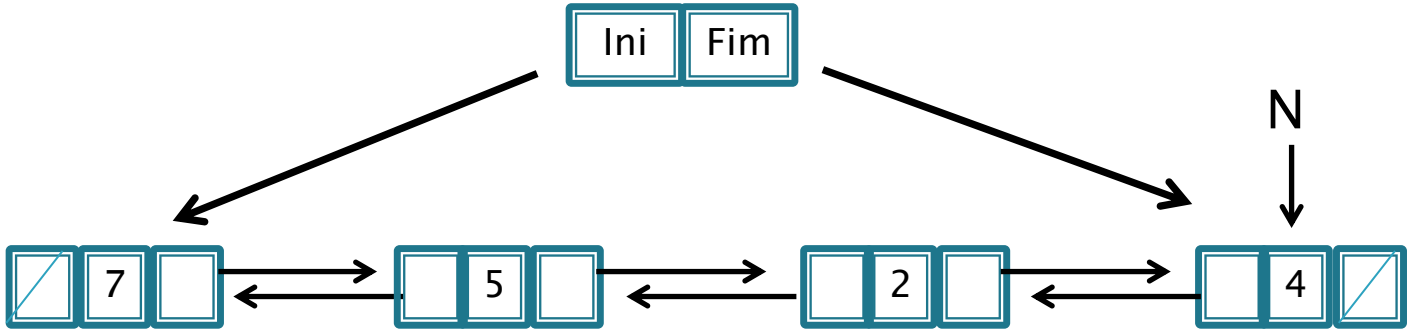
Aloca novo nó N
N->info = 3
N->prox = Ini
N->ant = NULL
Ini->ant = N
Ini = N




## Insere_final(4):

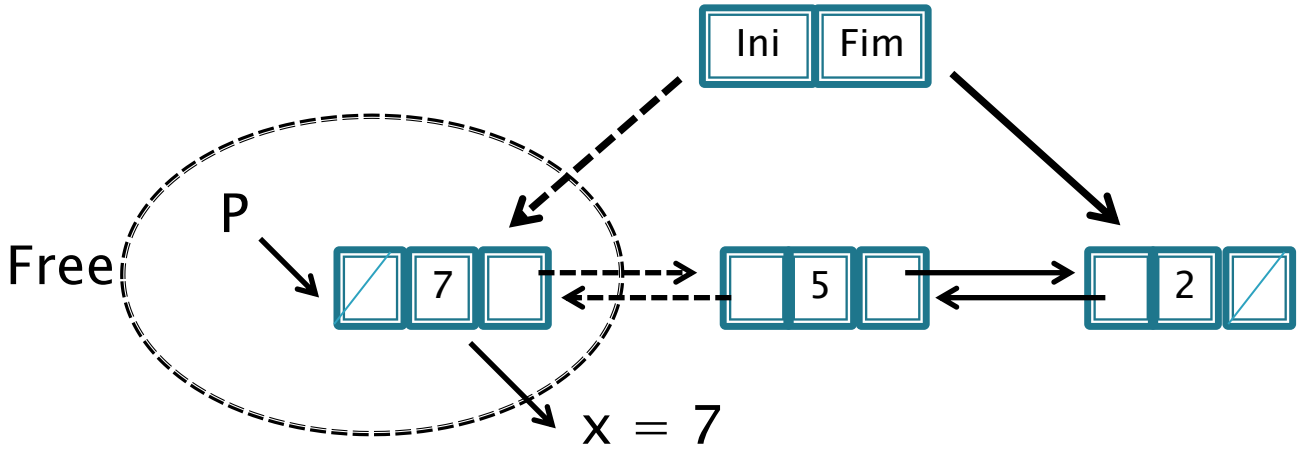
**Ambos os casos exigem tratamento especial para deque vazio**

Aloca novo nó N
N->info = 4
N->prox = NULL
N->ant = Fim
Fim->prox = N
Fim = N

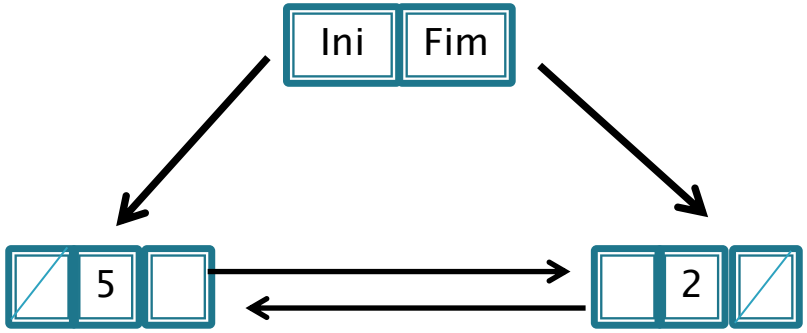


# Fila Dupla ou Deque

Remove_início(&x):



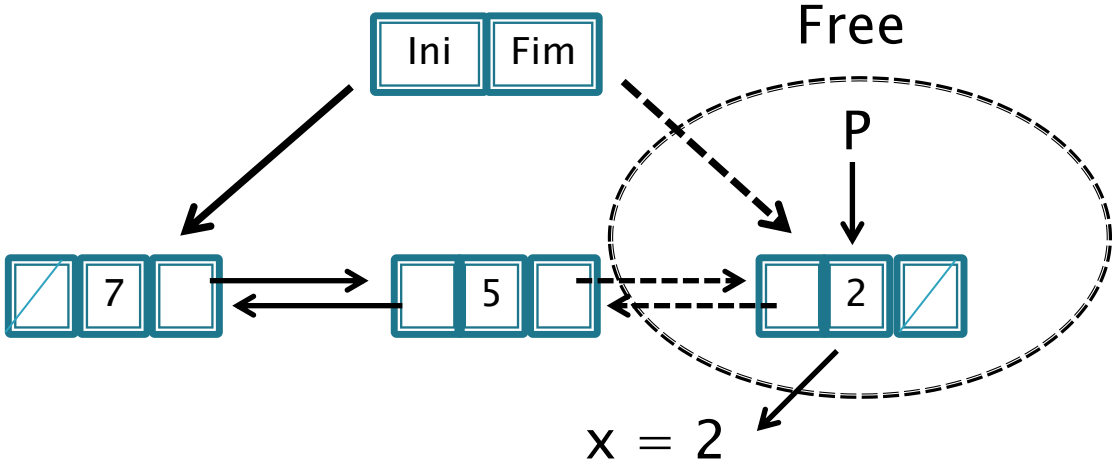

P = Ini
x = P->info
Ini = P->prox
Ini->ant = NULL
free(P)

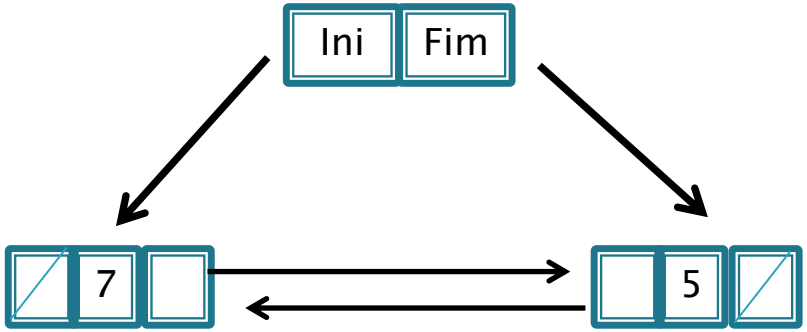


# Fila Dupla ou Deque

Remove_final (&x):

P = Fim
x = P->info
Fim = P->ant
Fim->prox = NULL
free(P)

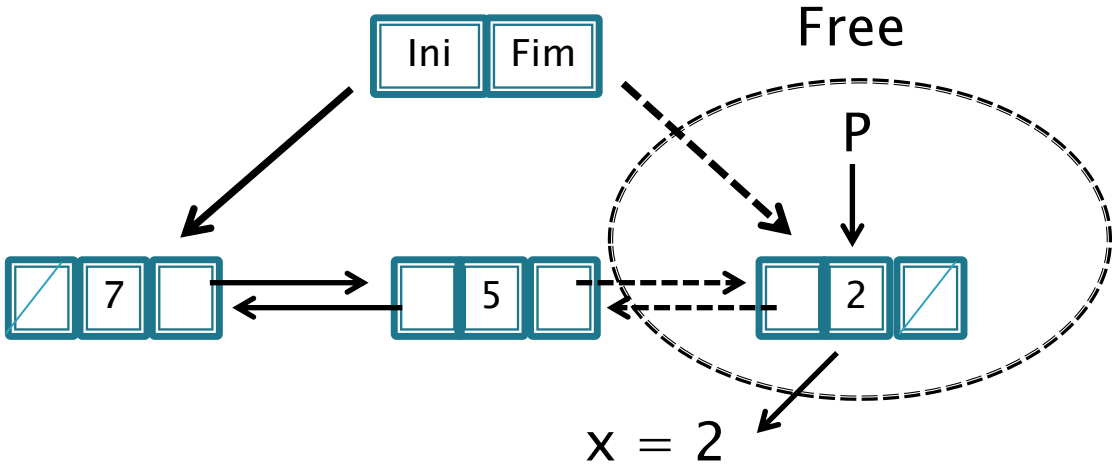


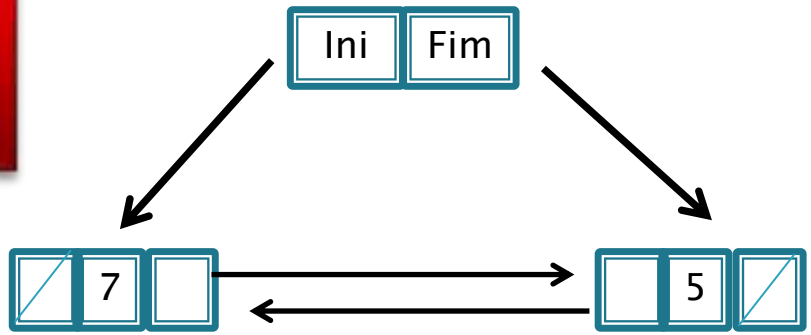
# Fila Dupla ou Deque

Remove_final (&x):

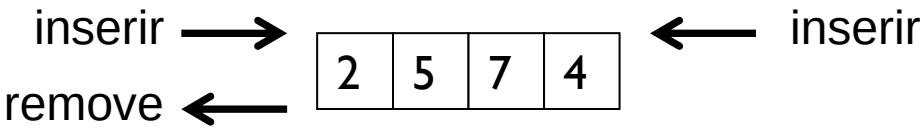
P = Fim
x = P->info
Fim = P->ant
Fim->prox = NULL
free(P)




Ambas as remoções exigem tratamento especial para **deque com um ÚNICO nó**



# Deque com saída restrita

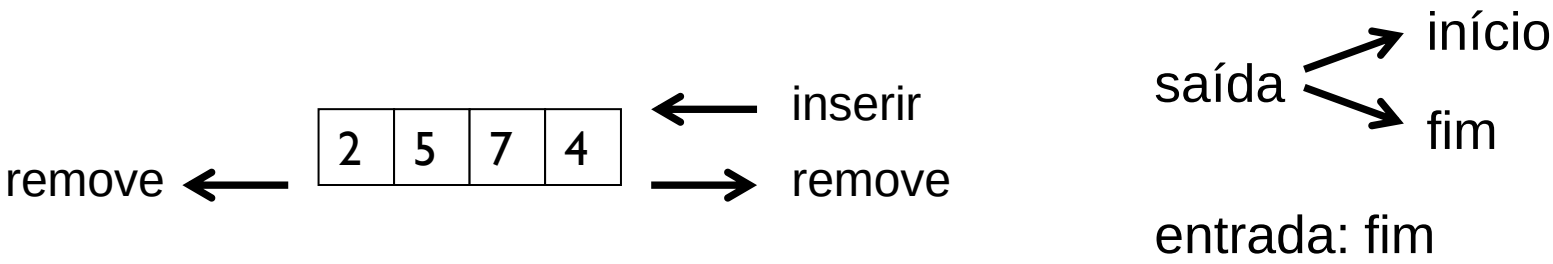



saída: início

entrada → início
entrada → fim

- Funciona como uma fila especial onde eventualmente podemos dar prioridade a um elemento, inserindo-o no início

# Deque com entrada restrita




- Funciona como uma pilha especial, onde podemos remover da base
  - **Final é o topo e o início é a base**
  - **Ex**: quando ocorrer um estouro da pilha (pilha cheia), pode ser usada a remoção da BASE para retirar o elemento **mais antigo** da pilha

# Listas Simples e Listas Estruturadas

- Maioria dos exemplos vistos trata de listas simples, onde o elemento é um único campo, mas também podemos ter elementos estruturados
  - **Ex:** lista do polinômios

```
struct no {
        int Info;
        struct no *Prox;
};
```

```
struct no {
        char Info;
        struct no *Prox;
};
```

```
struct termo {
        int Coef;
        int Expo;
        struct termo *Prox;
};
```

# Listas Simples e Listas Estruturadas

- Uma opção para se trabalhar com listas mais complexas é o uso de uma estrutura para representar o elemento

```
struct dado {
        int cod-matricula;
        char *nome;
        char *endereço;
        int idade;
};
```

Acesso: l -> Info.idade

```
struct no {
        struct dado Info;
        struct no *Prox;
};
```

OU

```
struct no {
        struct dado *Info;
        struct no *Prox;
};
```

Acesso: l -> Info -> idade

# Listas Homogêneas e Heterogêneas

- Mais usual são estruturas com elementos do mesmo tipo (ex: **lista homogênea**)
- Estruturas podem lidar com elementos de tipos diferentes (ex: **listas heterogêneas**)
- Podem ser implementadas de 2 formas:
  - Uso do **UNION**
  - Uso de ponteiro do tipo **VOID**

# Listas Homogêneas e Heterogêneas

- **Uso do UNION:**

```
struct no {
        int tipo;
        union {
                int i_int;
                char i_char;
        } info;
        struct no *Prox;
};
```

**Exemplo de acesso:**

**SE** lista->tipo = 0 **ENTÃO**
    lista->info.i_int = x;
**SENÃO**
    lista->info.i_char = x;
**FIM-SE**

# Listas Homogêneas e Heterogêneas

- **Uso do UNION:**

```
struct no {
        int tipo;
        union {
                int i_int;
                char i_char;
        } info;
        struct no *Prox;
};
```

**Exemplo de acesso:**

```
SE lista->tipo = 0 ENTÃO
        lista->info.i_int = x;
SENÃO
        lista->info.i_char = x;
FIM-SE
```

- Os membros de um union se sobrepõem
  - **Espaço é alocado para o tipo de maior tamanho**

# Listas Homogêneas e Heterogêneas

- **Uso de um ponteiro para VOID:**

```
struct no{
        int tipo;
        void *info;
        struct no *Prox;
};
```

**Exemplo de acesso:**

```
int *i;
char *c;

if (lista->tipo==0) {
        *i = x;
        lista->info = i;
}
else {
        *c = x;
        lista->info = c;
}
```

# Exercícios

1. Usando **UNION**, implemente uma **lista heterogênea de notas** para os alunos de uma turma. Dependendo do curso, essa nota pode ser um **número real** (float) **ou um conceito** (A, B, C ou D). Essa lista deve ser **estática/sequencial com no máximo 30 elementos** e contemplar as operações: criar_lista, lista_vazia, lista_cheia, insere_elem, remove_elem e obtem_elem_pos. Além disso, desenvolva um programa aplicativo que permita ao usuário criar uma lista, inserir e remover elementos e imprimir a lista.

2. Refaça o exercício anterior, usando uma implementação **dinâmica/encadeada** e **ponteiro do tipo VOID**.