CES-11

Algoritmos e Estruturas de Dados

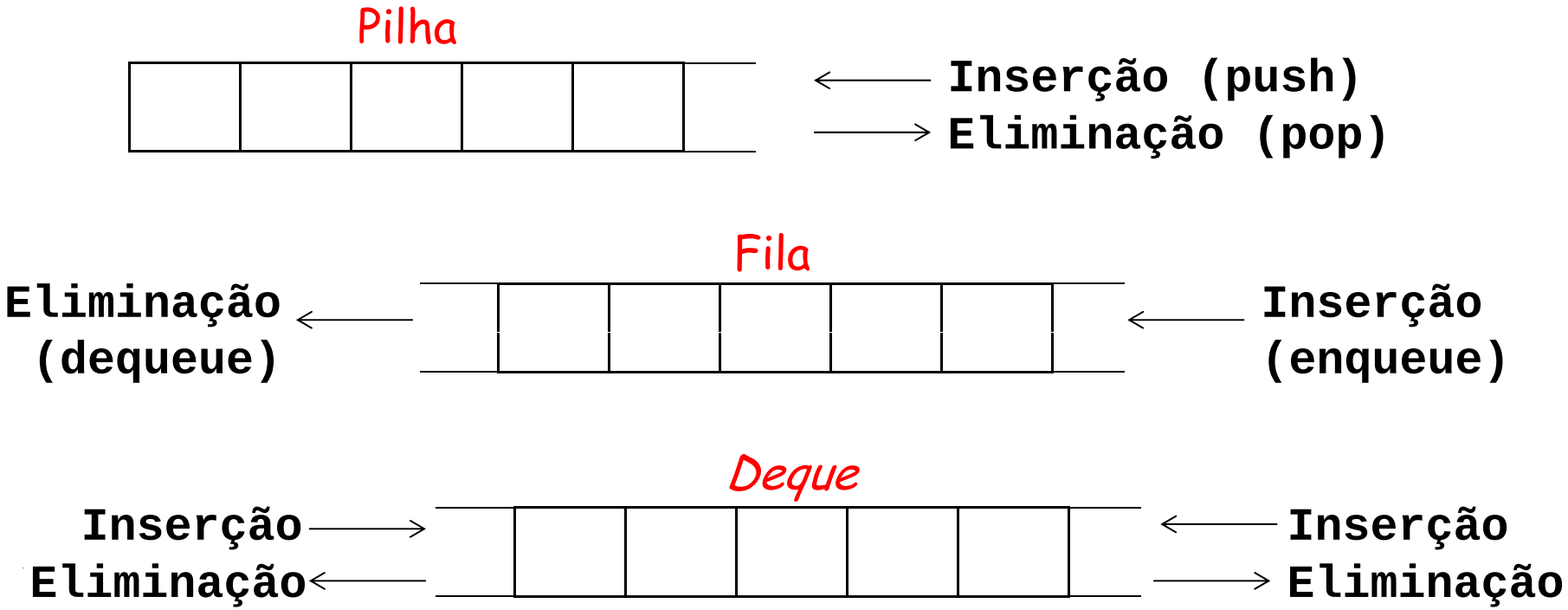
Carlos Alberto Alonso Sanches
Juliana de Melo Bezerra

CES-11

- Pilhas
- Filas
- *Deques*

Pilhas, filas e *deques*

- Vimos que as listas lineares admitem inserção e eliminação em qualquer posição.
- Pilhas, filas e *deques* (filas com duplo-fim) só admitem acessos nas suas extremidades:

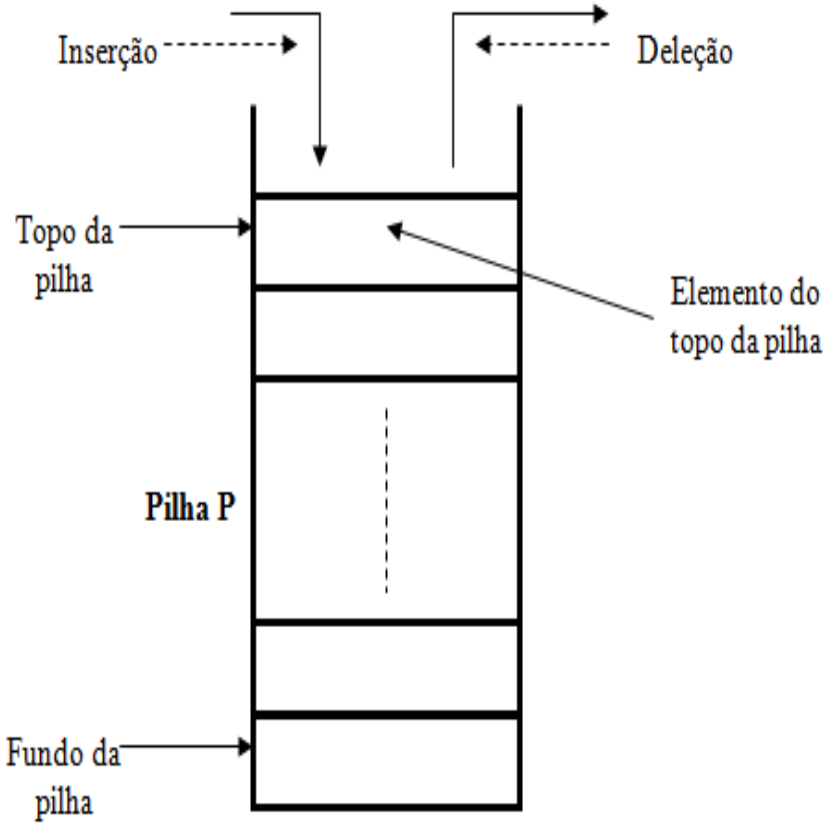


CES-11

- Pilhas
- Filas
- *Deques*

Pilhas (*stacks*)

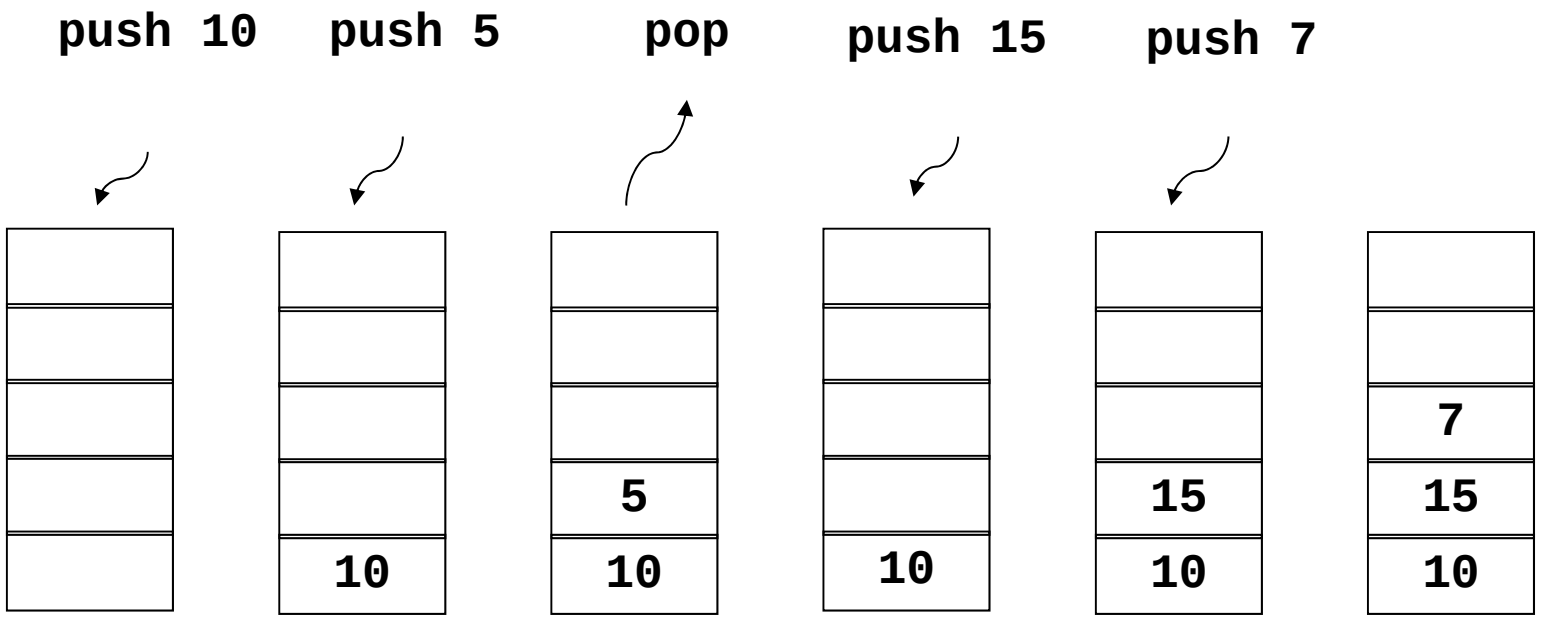
- *Pilha* é uma estrutura linear que permite acesso em somente uma extremidade.
- Por essa razão, a pilha é chamada de estrutura *LIFO* (*last in / first out*).



Pilhas (*stacks*)

- Operações (no caso, pilha de inteiros):
  - void **inicPilha**(pilha *P): inicializa a pilha
  - int **size**(pilha P): retorna o tamanho atual da pilha
  - bool **isEmpty**(pilha P): verifica se a pilha está vazia
  - int **top**(pilha P): retorna o topo sem desempilhá-lo
  - void **push**(pilha *P, int x): insere x no topo
  - void **pop**(pilha *P): retira o elemento topo

Exemplos de operações com pilhas



Uma aplicação típica

- Pilhas são utilizadas, por exemplo, para verificar o emparelhamento de delimitadores.

```
pilha P1; // suponha pilha de caracteres
inicPilha(&P1);
Leia o próximo caractere ch;
while não é fim de arquivo {
     if ch é '(', '[' ou '{'
        push(&P1,ch);
     else if ch é ')', ']' ou '}' {
             x = top(P1);
             pop(&P1);
             if ch e x não se casam
                 Erro;
     }
     Leia o próximo caractere ch;
}
if isEmpty(P1)
   sucesso;
else Erro;
```

Implementação com vetor

```
#define N 100
struct pilha{
  int S[N];
  int t;
}
```


```
void inicPilha(pilha *P){
  P->t = -1; }

int size(pilha P) {
  return P.t + 1;}

bool isEmpty(pilha P) {
  return (P.t < 0);}

int top(pilha P) {
  if (isEmpty(P)) return Erro;
  return P.S[P.t];}
```

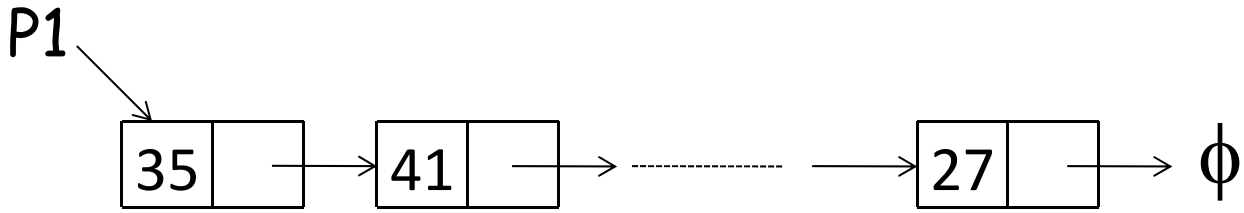
```
void push(pilha *P, int x){
  if (size(*P) == N)
    printf("Erro");
  else {
    P->t++;
    P->S[P->t] = x;
  }
}

void pop(pilha *P) {
  if (isEmpty(*P))
    printf("Erro");
  else P->t--;
}
```

Eficiência dessa implementação

- Na implementação com vetor, todas as operações anteriores podem ser executadas em tempo $O(1)$.
- Caso fosse necessário implementar uma operação de busca na pilha, a solução gastaria tempo proporcional ao tamanho dessa estrutura...

Implementação com nós encadeados

- Os elementos da pilha podem estar conectados através de uma cadeia de nós
- Exemplo: pilha de inteiros

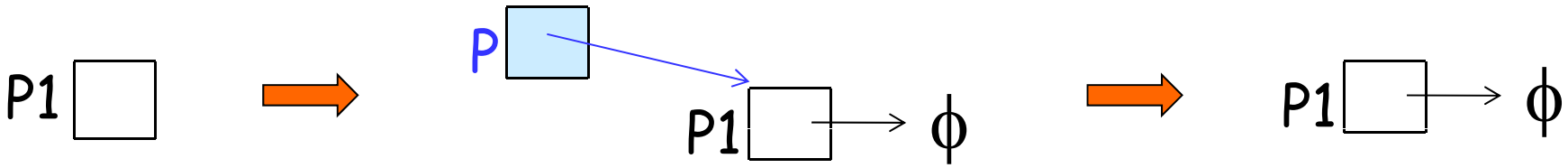



```
struct node {int elem; node *prox;};
typedef node *pilha;
pilha P1;
```

Implementação com nós encadeados

- **`inicPilha(&P1)`**: inicializa a pilha P1

```
void inicPilha (pilha *P) {
     *P = NULL;
}
```

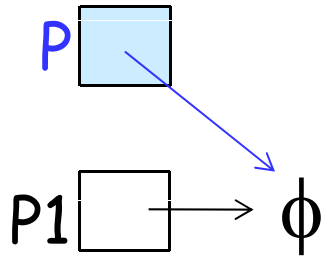


Implementação com nós encadeados

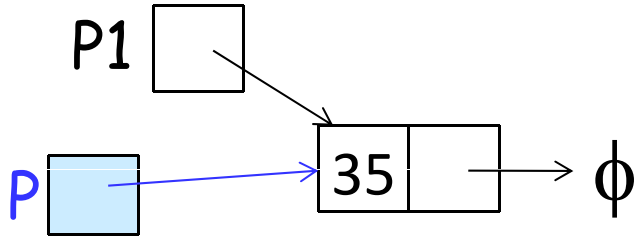
- **`isEmpty(P1)`**: verifica se a pilha P1 está vazia

```
bool isEmpty (pilha P) {
     if (P == NULL) return true;
     else return false;
}
```

Pilha vazia



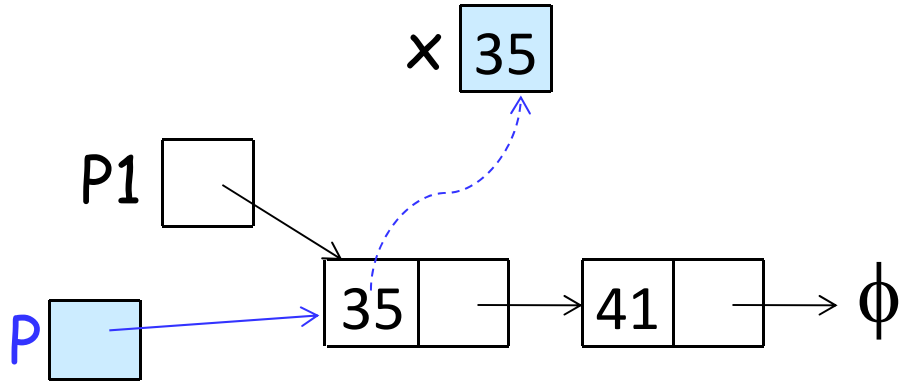
Pilha não vazia



Implementação com nós encadeados

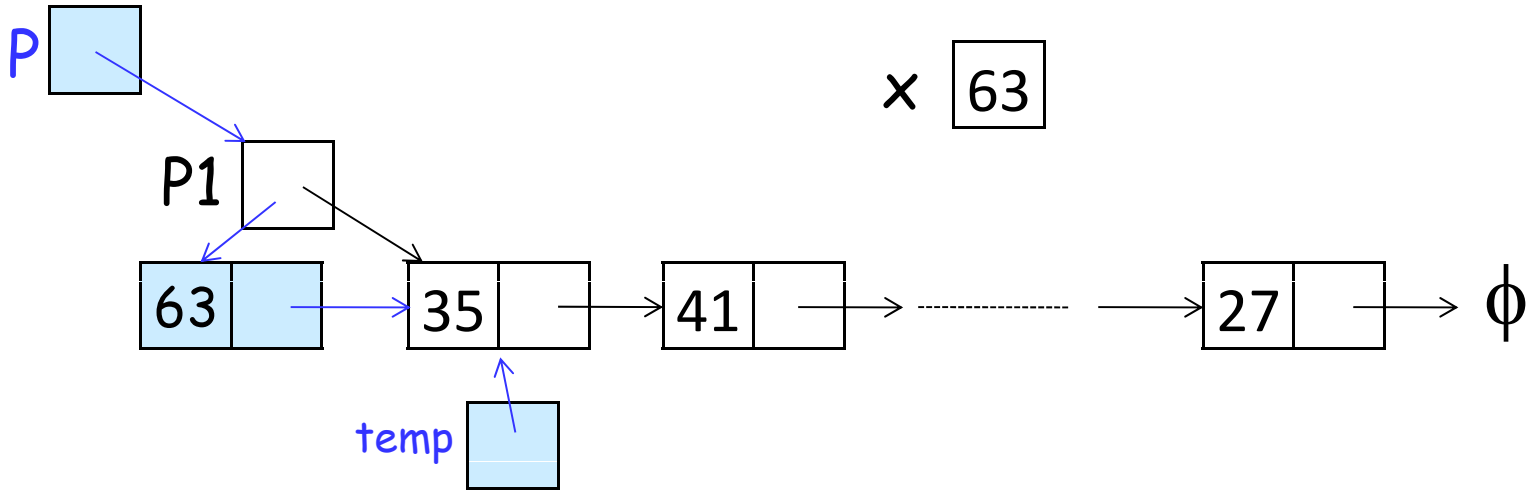
- `x = top(P1)`: x recebe o topo da pilha P1

```
int top (pilha P) {
     if (!isEmpty (P))
          return P->elem;
     else
          return Erro;
}
```



Implementação com nós encadeados

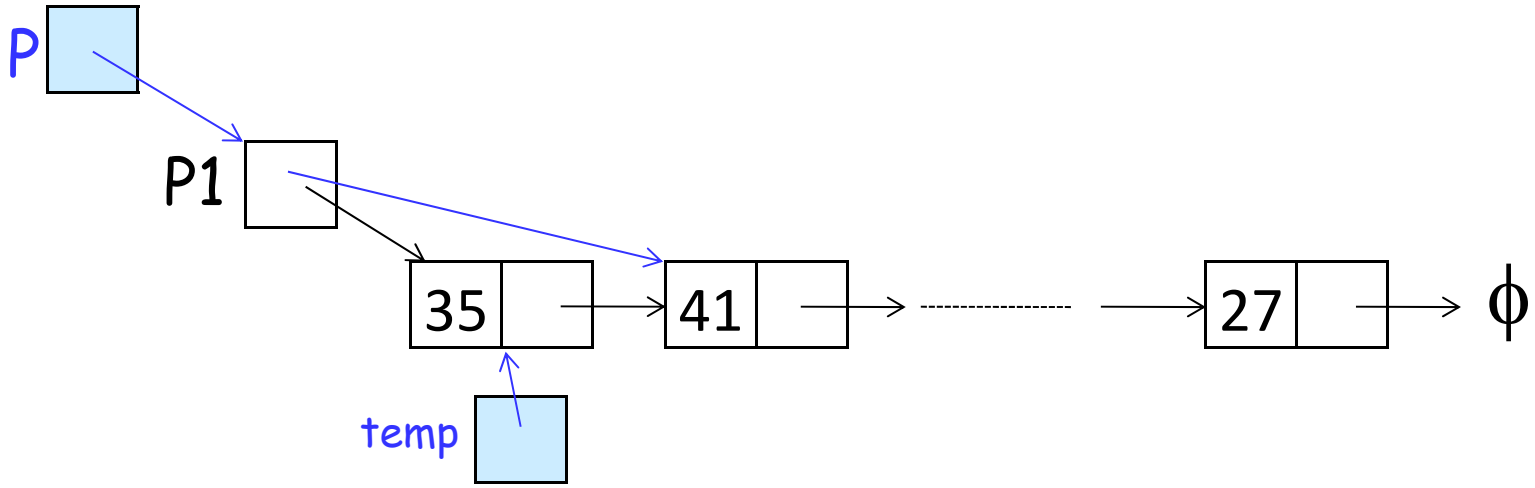
- **push(&P1,x)**: empilha o conteúdo de x na pilha P1




```
void push (pilha *P, int x) {
      node *temp;
      temp = *P;
      *P = (node *) malloc (sizeof (node));
      (*P)->elem = x;
      (*P)->prox = temp;
}
```

Implementação com nós encadeados

- **pop(&P1)**: desempilha o topo da pilha P1




```
void pop (pilha *P) {
      node *temp;
      if (!isEmpty (*P)) {
            temp = *P;
            *P = (*P)->prox;
            free (temp);  }
      else Erro ("Pilha vazia"); }
```

Eficiência dessa implementação

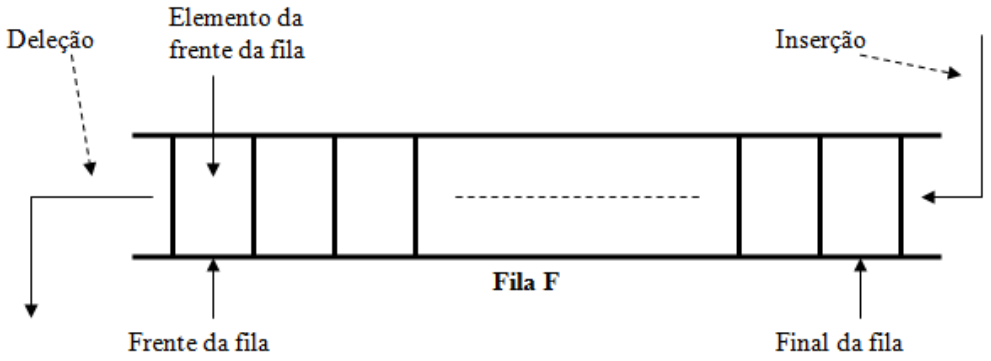
- Na implementação com nós encadeados, todas as operações anteriores podem ser executadas em tempo $O(1)$.
- Qual a vantagem em relação à implementação com vetor?
  - A memória é alocada gradativamente, apenas quando necessário.

CES-11

- Pilhas
- Filas
- *Deques*

Filas (*queues*)

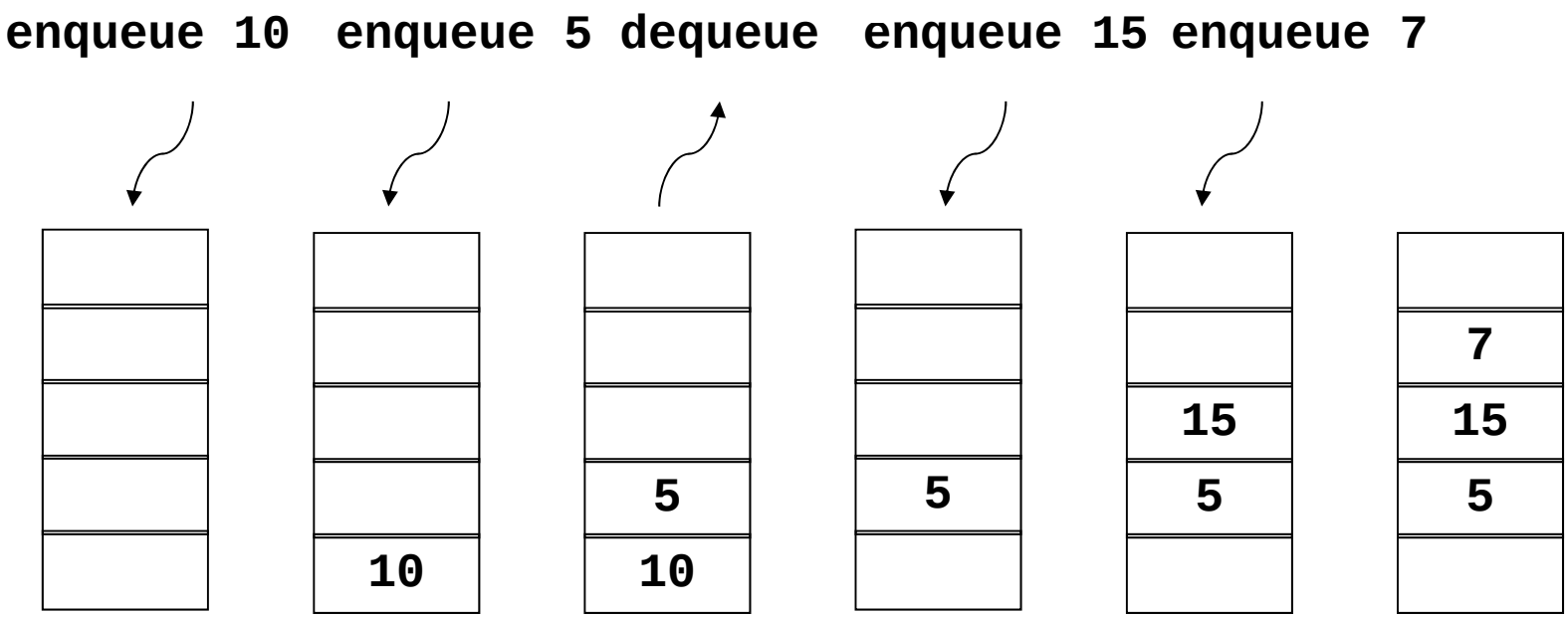
- *Filas* são simples sequências de espera: crescem através do acréscimo de novos elementos no final e diminuem com a saída dos elementos da frente.
- Os elementos são acrescentados em uma extremidade e removidos da outra.
- Em relação à pilha, a principal diferença é que a fila é uma estrutura *FIFO (first in/first out).*



Filas

- Operações (no caso, fila de inteiros):
  - void **inicFila**(fila *F): inicializa fila
  - int **size**(fila F): retorna o tamanho atual da fila
  - bool **isEmpty**(fila F): verifica se a fila está vazia
  - int **first**(fila F): retorna o primeiro elemento da fila sem removê-lo
  - void **dequeue**(fila *F): retira o primeiro elemento da fila
  - void **enqueue**(fila *F, int x): coloca x no final da fila

Exemplos de operações com filas



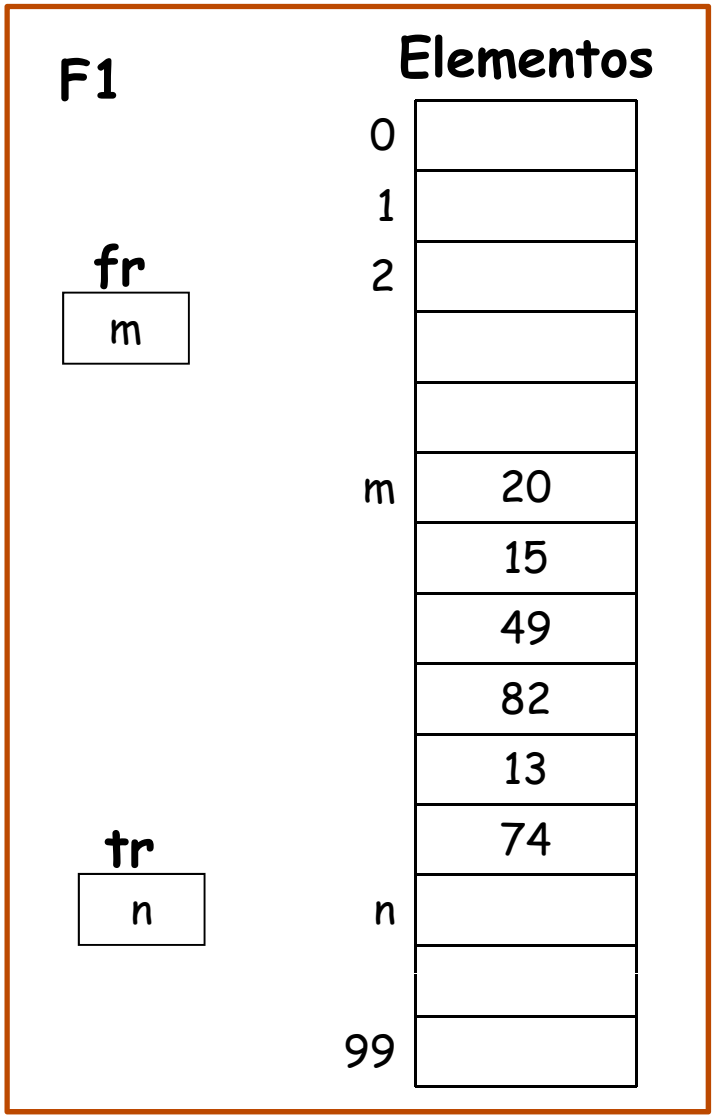
Implementação com vetor circular

- O vetor é utilizado de maneira circular.

```
const int max = 100;
typedef int vetor[max];
struct fila {
       vetor elementos;
       int fr, tr;
};
fila F1, F2, F3;
```

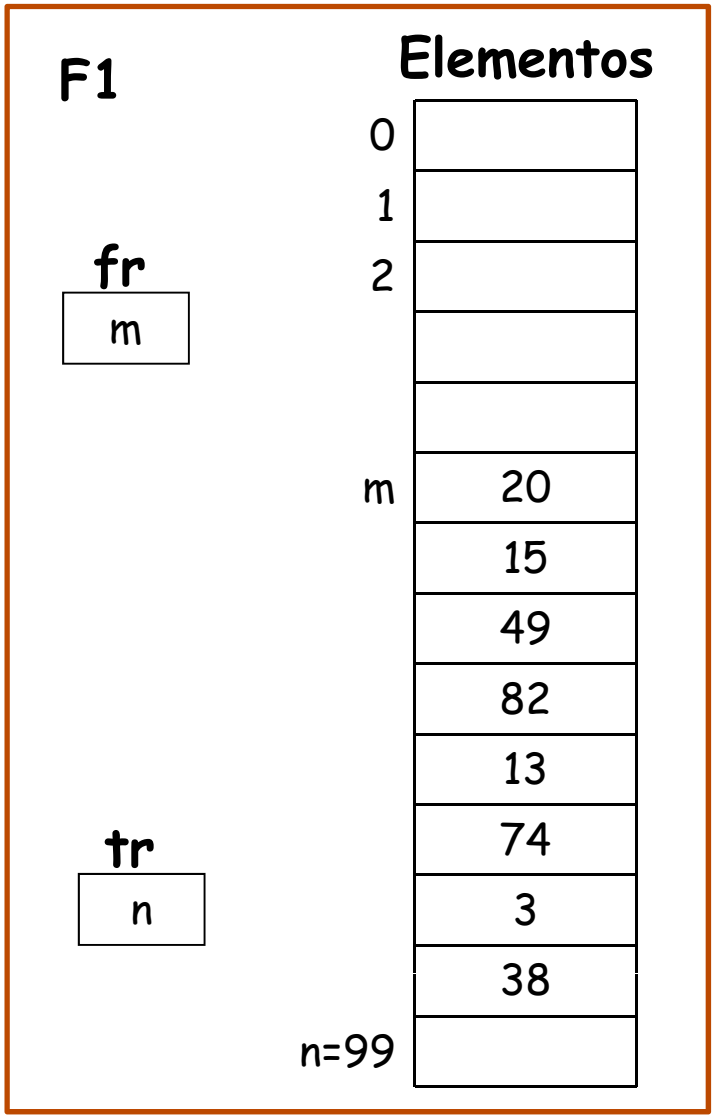
Próxima posição disponível

Posição do primeiro elemento



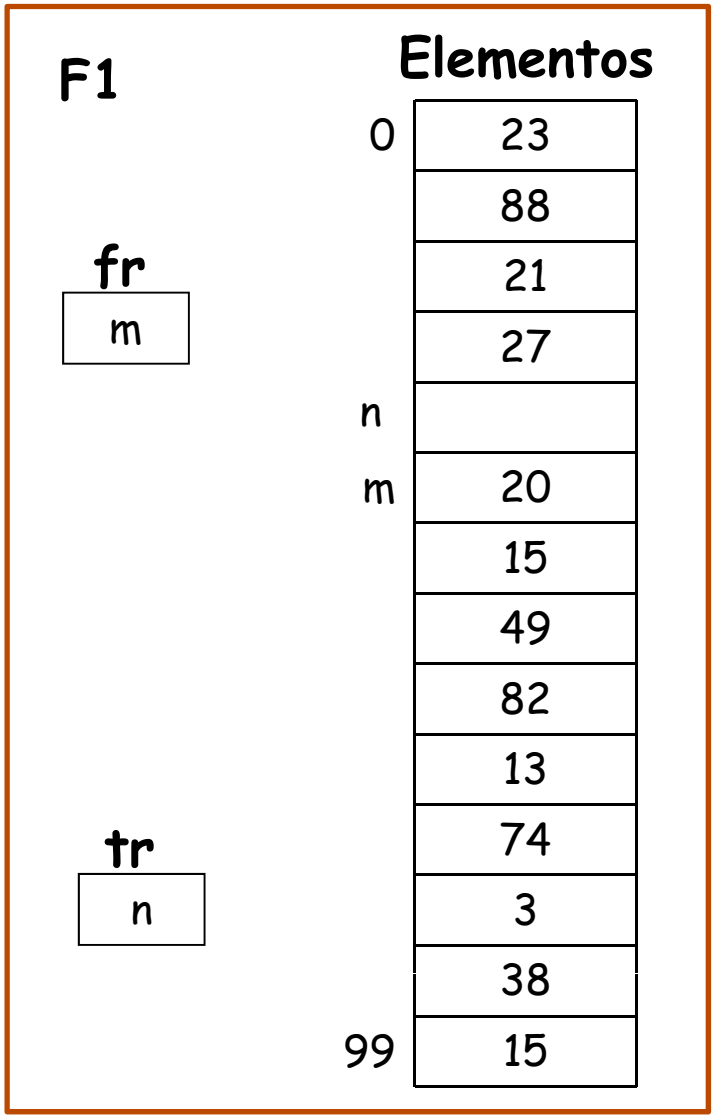
Implementação com vetor circular

- É circular: quando **tr=99**, uma inserção levará **tr** para o índice **0** (zero).
- Desse modo, não haverá necessidade de deslocar todos os elementos para cima...



Implementação com vetor circular

- Considere a situação ao lado.
- Supondo max=100, é possível admitir 100 elementos na fila?
  - Não: fila cheia seria confundida com fila vazia...
- Caso geral desta implementação: em uma fila com max posições, podemos ter no máximo max-1 elementos.

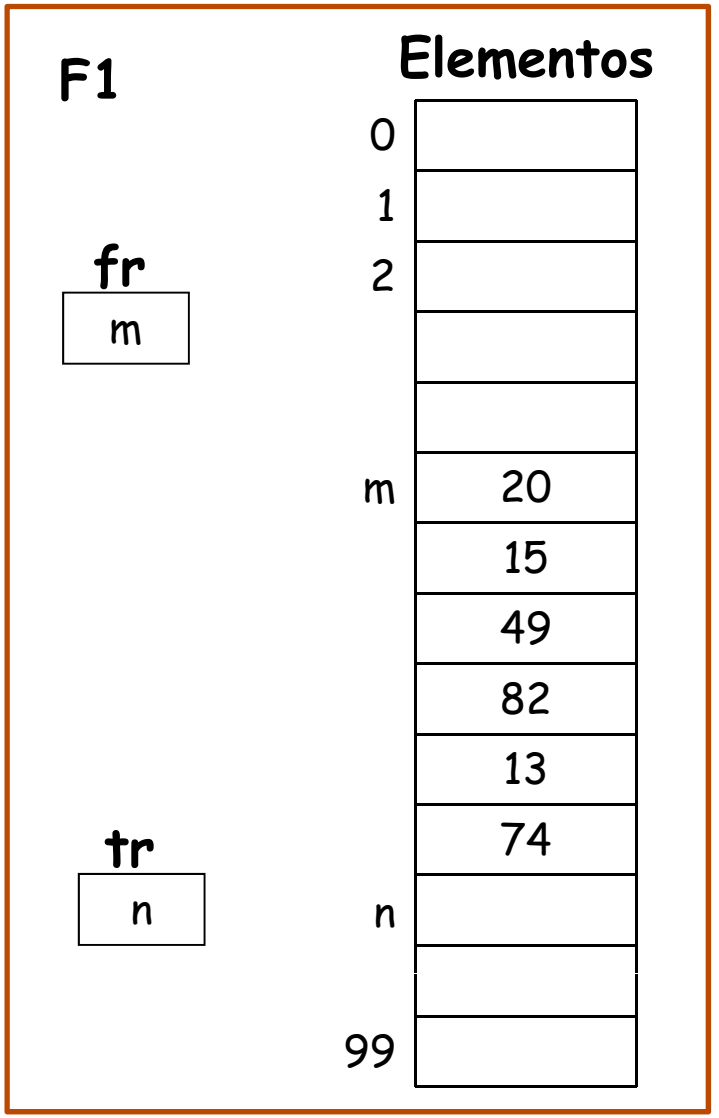


Implementação com vetor circular

```
void inicFila (fila *F) {
     F->fr = 0;
     F->tr = 0;
}
```

Função auxiliar: próximo índice de um vetor circular

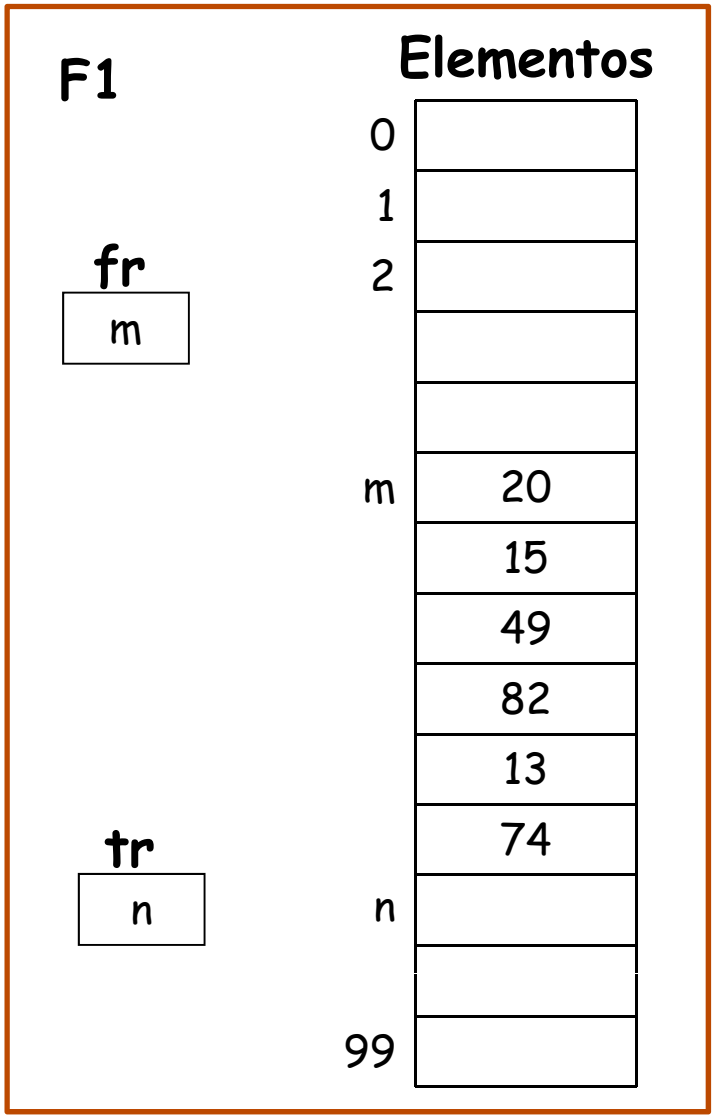
```
int prox (int i)  {
     return (i+1) % max;
}
```



Implementação com vetor circular

```
bool isEmpty (fila F) {
      if (F.tr == F.fr)
            return true;
      return false;
}
```

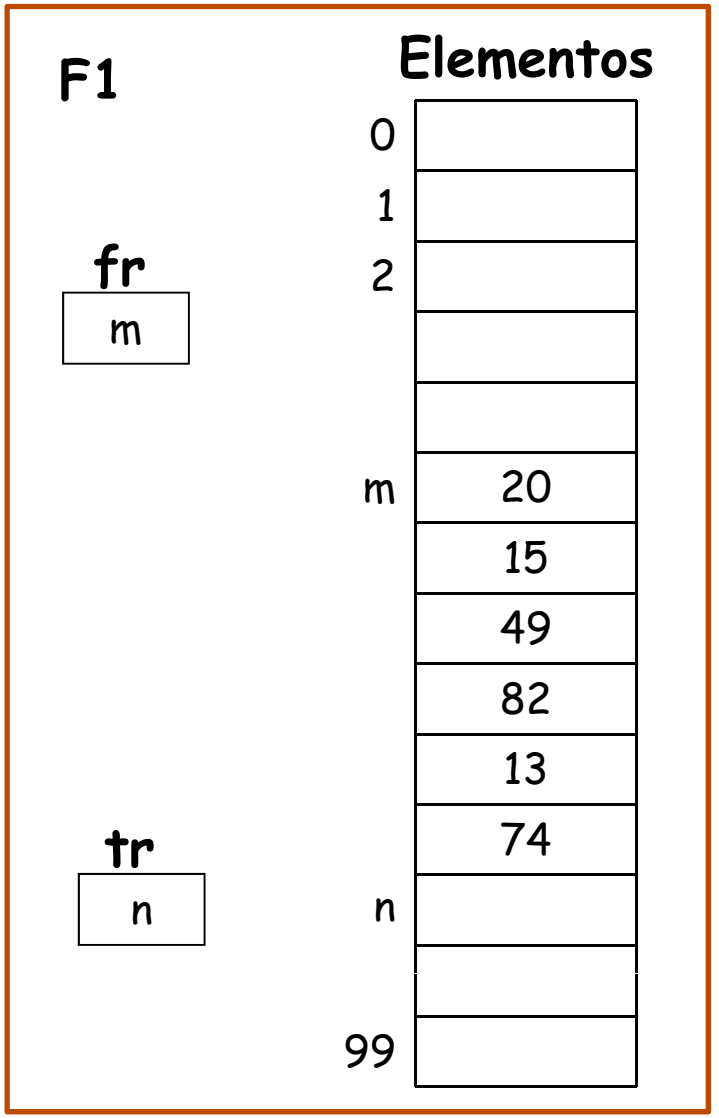
```
int first (fila F) {
   if (isEmpty(F))
      Erro (“Fila vazia”);
   else
      return F.elementos[F.fr];
}
```



Implementação com vetor circular

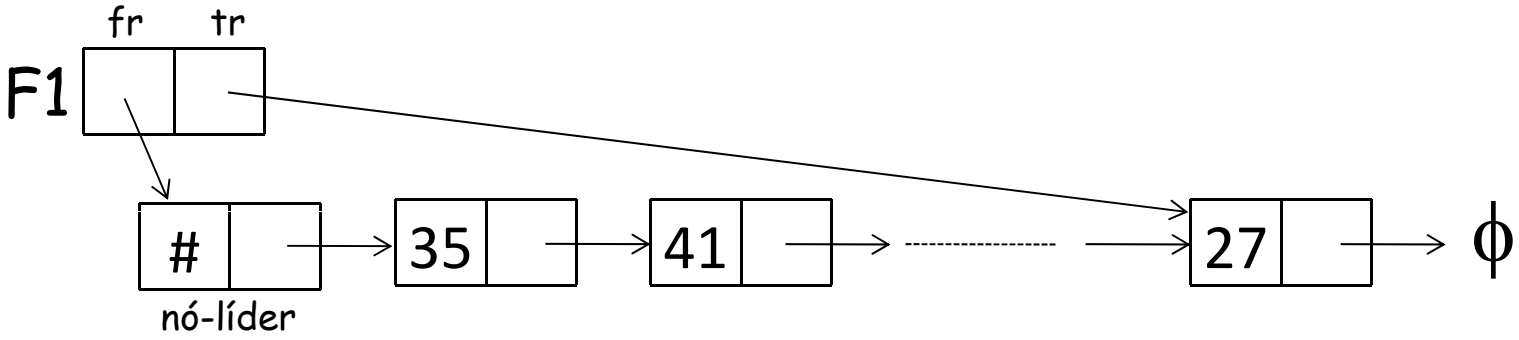
```
void enqueue (fila *F, int x)
{
    if (prox(F->tr) == F->fr)
       Erro (“Fila cheia”);
    else {
       F->elementos[F->tr] = x;
       F->tr = prox(F->tr);
    }
}
```

```
void dequeue (fila *F)
    if (isEmpty (F))
         Erro (“Fila vazia”);
    else
         F->fr = prox(F->fr);
}
```



Implementação com nós encadeados

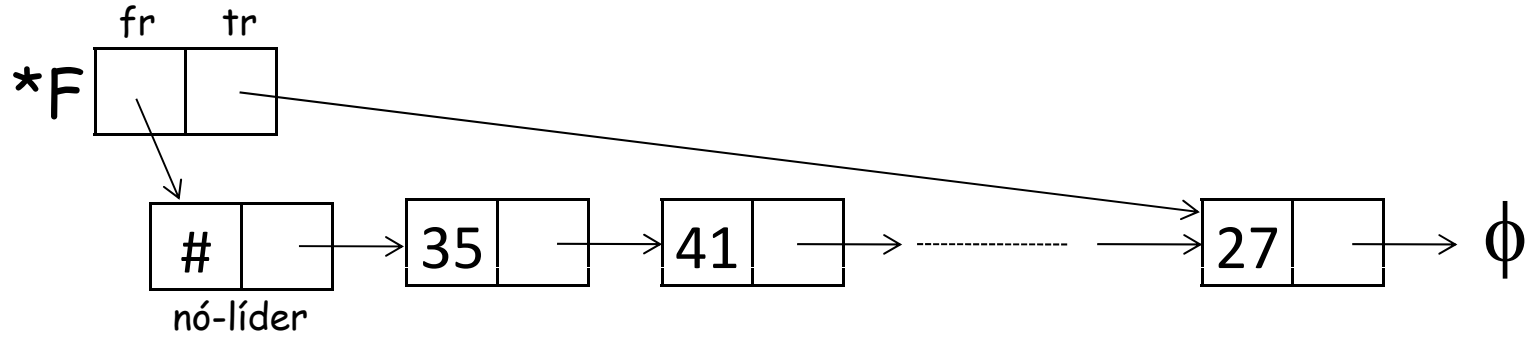
- Os elementos da fila podem estar conectados através de uma cadeia de nós.
- O uso de um nó-líder facilita a codificação.
- Exemplo:

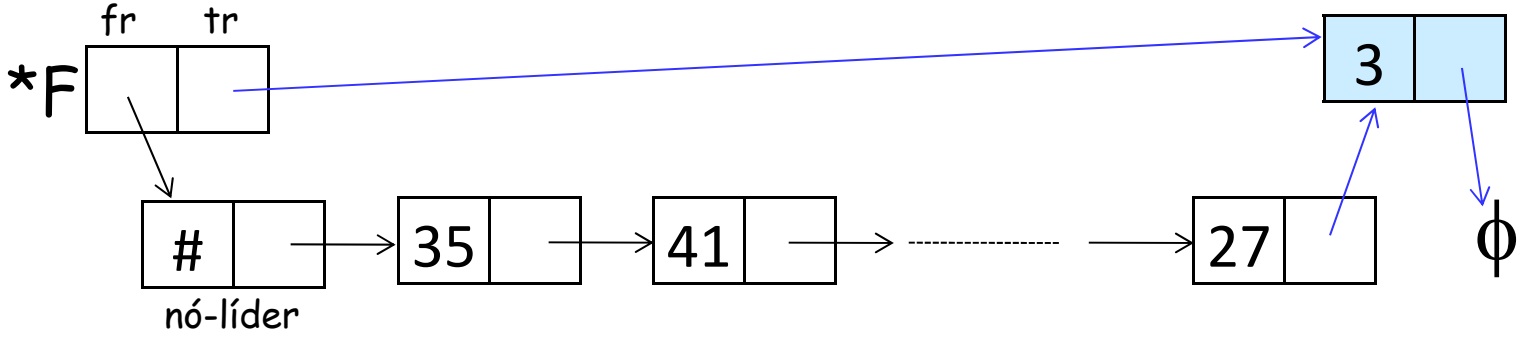



```
struct node {int elem; node *prox;};
struct fila {node *fr, *tr;};
fila F1;
```

Implementação com nós encadeados

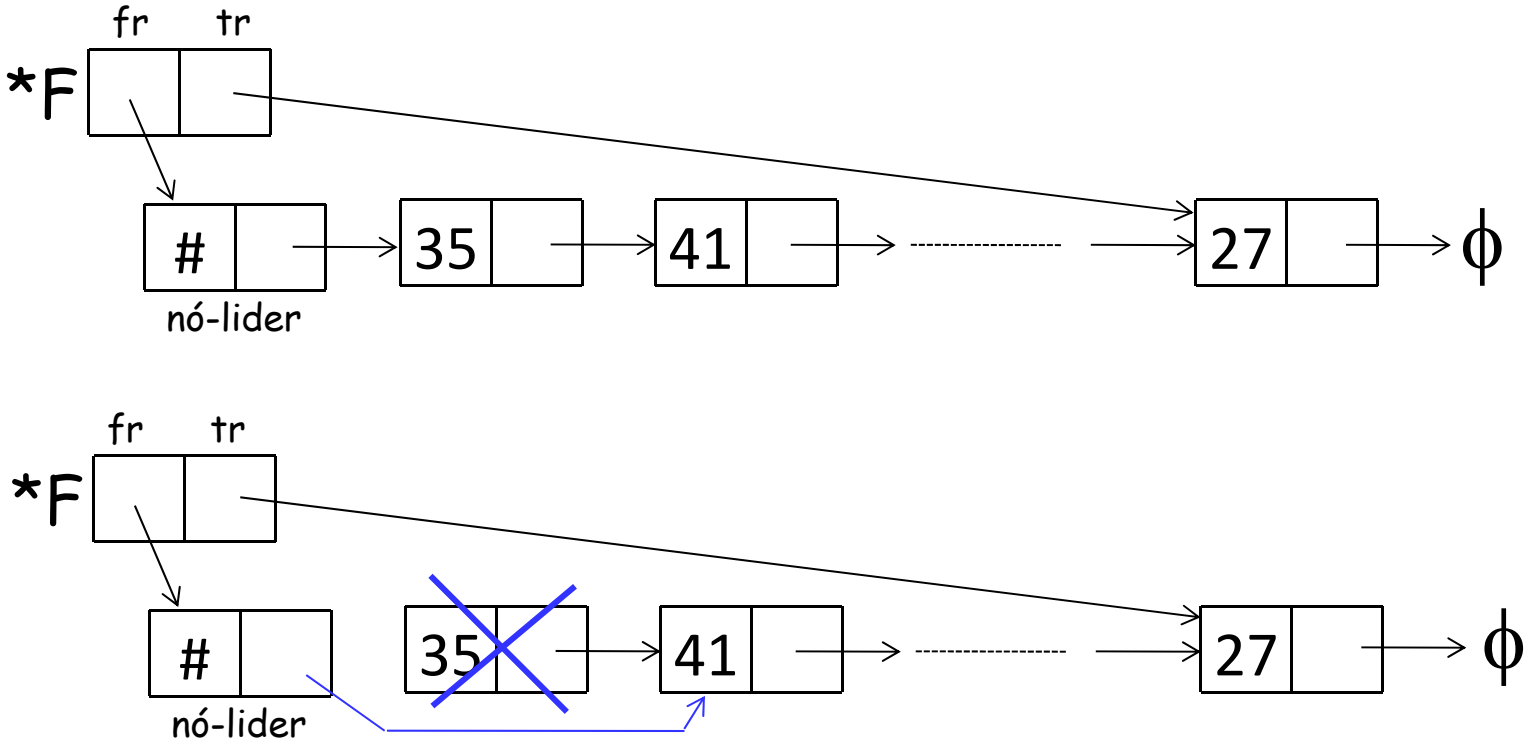
- **`void enqueue(fila *F, int x)`:**
  - Inserir um nó com valor x após o último elemento
  - Atualizar o ponteiro F->tr





Implementação com nós encadeados

- **`void dequeue(fila *F)`:**
  - Apagar o primeiro nó da cadeia
  - Se não houver mais nós, F->tr deverá apontar para o nó-líder



Implementação com nós encadeados

- Primeiro elemento da fila F
  - `int first(fila F);`
    - `F.fr->prox->elem`
- Verificação de fila F vazia
  - `bool isEmpty(fila F);`
    - `F.fr == F.tr` ou
    - `F.fr->prox == NULL`
- Esvaziamento da fila F
  - `void esvaziar(fila *F);`
    - Liberar os nós de F, exceto o líder
    - `F->fr->prox = NULL;`
    - `F->tr = F->fr`

CES-11

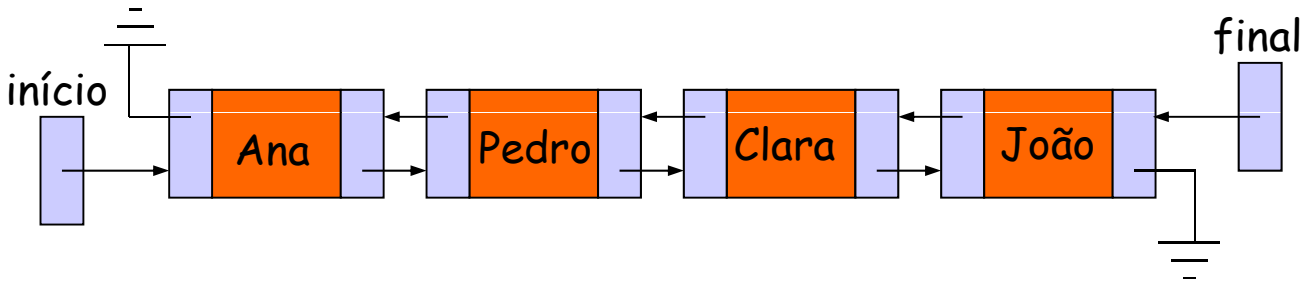
- Pilhas
- Filas
- *Deques*

Deques (filas com duplo-fim)

- Filas com duplo-fim (*doubled-ended queue*) permitem inserção e remoção, em tempo constante, tanto no início como no fim.
- Uma *deque* pode simular tanto as operações de uma pilha como as de uma fila.
- Operações (no caso, *deque* de inteiros):
  - void **inicDeque**(deque *D): inicializa o deque
  - void **insertFirst**(deque *D, int x): insere x no início
  - void **insertLast**(deque *D, int x): insere x no final
  - void **removeFirst**(deque *D): remove o primeiro elemento
  - void **removeLast**(deque *D): remove o último elemento
  - int **first**(deque D): retorna o primeiro elemento
  - int **last**(deque D): retorna o último elemento
  - int **size**(deque D): retorna o tamanho atual
  - bool **isEmpty**(deque D): verifica se está vazia

Implementação com encadeamento duplo

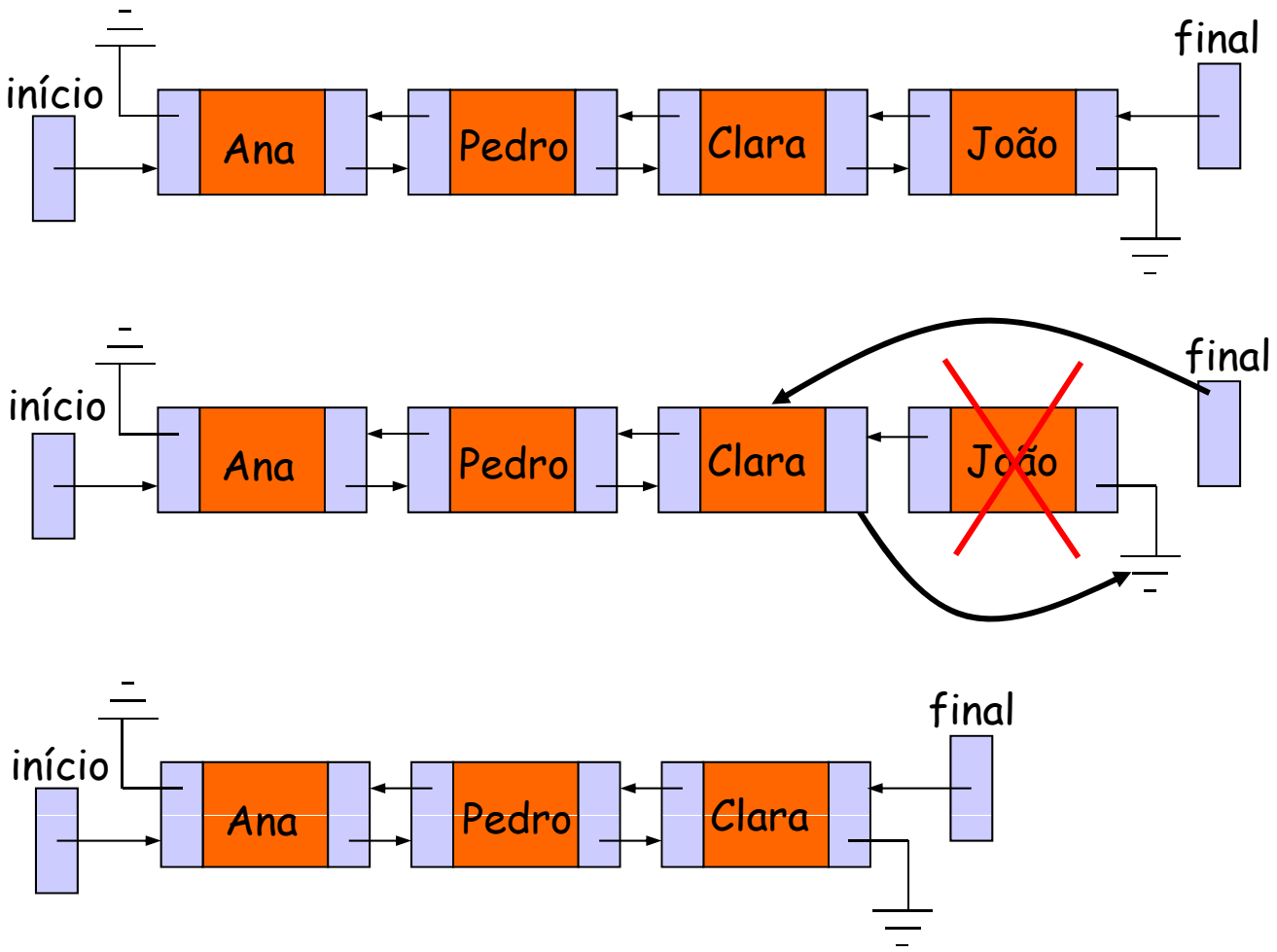
- Podemos implementar uma *deque* através de uma lista duplamente ligada.
- Exemplo de uma *deque* de *strings*:



- Cada nó tem dois ponteiros: para o próximo e para o anterior.
- Para cada *deque*, é preciso guardar ponteiros para o seu início e o seu final.

Implementação com encadeamento duplo

- Remoção do último elemento:



Implementação com vetor

- Seria possível implementar uma *deque* através de um vetor?
- Quais as vantagens e as desvantagens em relação à implementação com listas duplamente ligadas?