

# Pilhas e Filas

Estrutura de Dados

Prof. Anselmo C. de Paiva

Departamento de Informática – Núcleo de Computação Aplicada NCA-UFMA

# Operações sobre Coleções

- ▶ Inserir um elemento
- ▶ Remover um elemento
- ▶ Buscar por um elemento
- ▶ Enumerar um elemento
- ▶ Operações complexas
  - ▶ União/interseção/diferença de coleções
  - ▶ Nem sempre necessárias para todas as aplicações
- ▶ Estrutura mais simples com restrições de acesso são úteis

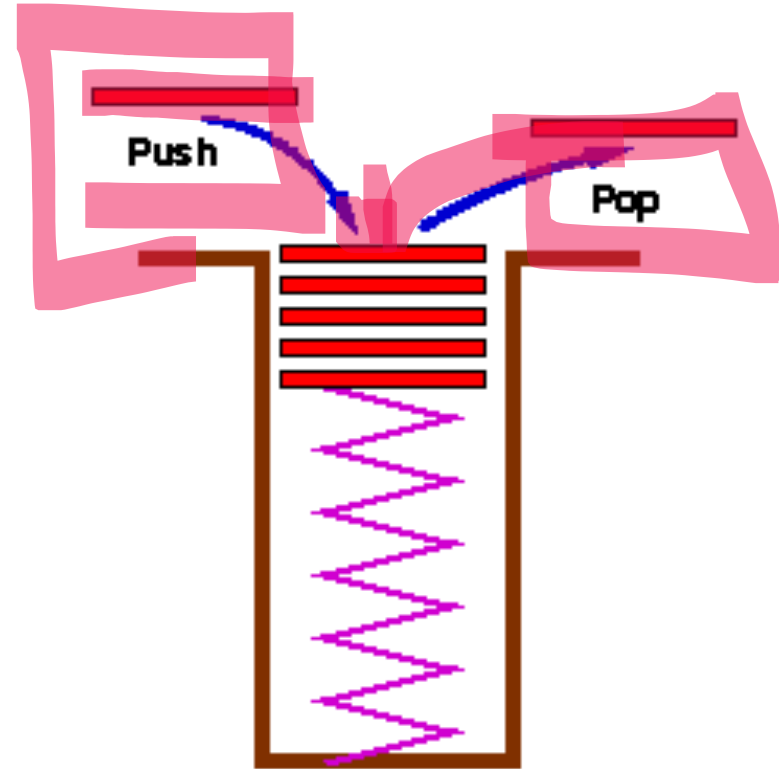
## Exemplos de limitações de acesso:

- ▶ Inserção acontece somente em posições conhecida
- ▶ Listagem dos elementos deve obedecer ordem específica
- ▶ Busca pode acontecer simplesmente em uma posição específica

▶ FIFO (first in first out) e LIFO (last in first out)

# Estrutura de Dados - Pilha

- ▶ Operações afetam somente
  - ▶ elemento mais recente
- ▶ Disciplina de acesso **FIFO**
  - ▶ Primeiro que entra é o primeiro que sai
- ▶ Operações Básicas:
  - ▶ Criar
  - ▶ Inserção – push
  - ▶ Remoção – pop
  - ▶ Destruir
- ▶ Operações Adicionais:
  - ▶ Primeiro
  - ▶ EstaVazia

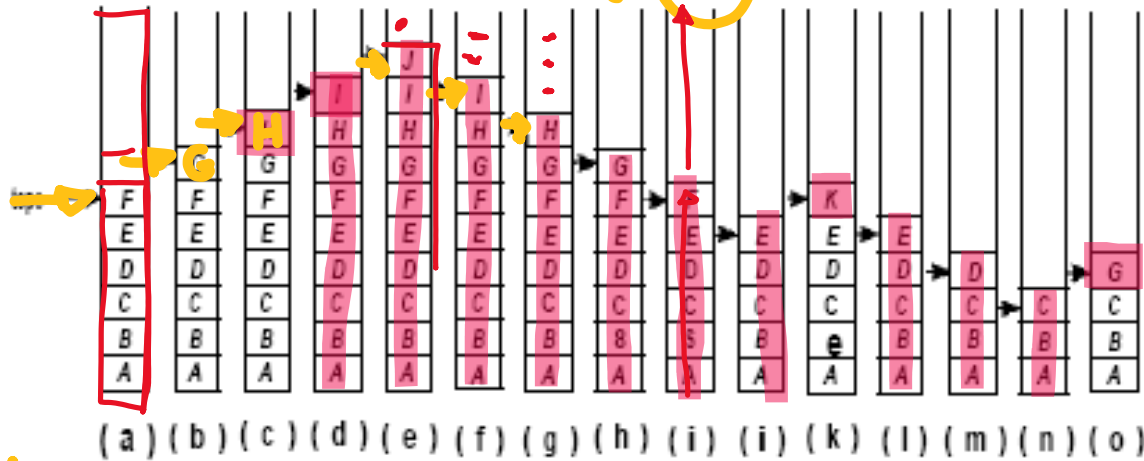


# Pilha - Exemplo

top

0 — S → top

iti



```

push (s, H);      (quadro (c))
push (s, I);      (quadro (d))
push (s, J);      (quadro (e))
pop (s);          (quadro (f))
pop (s);          (quadro (g))
pop (s);          (quadro (h))
pop (s);          (quadro (i))
pop (s);          (quadro (j))
push (s, K);      (quadro (k))
pop (s);          (quadro (l))
pop (s);          (quadro (m))
pop (s);          (quadro (n))
push (s, G);      (quadro (o))
    
```

top++  
[top]

Push

Pop ← [top]  
top--

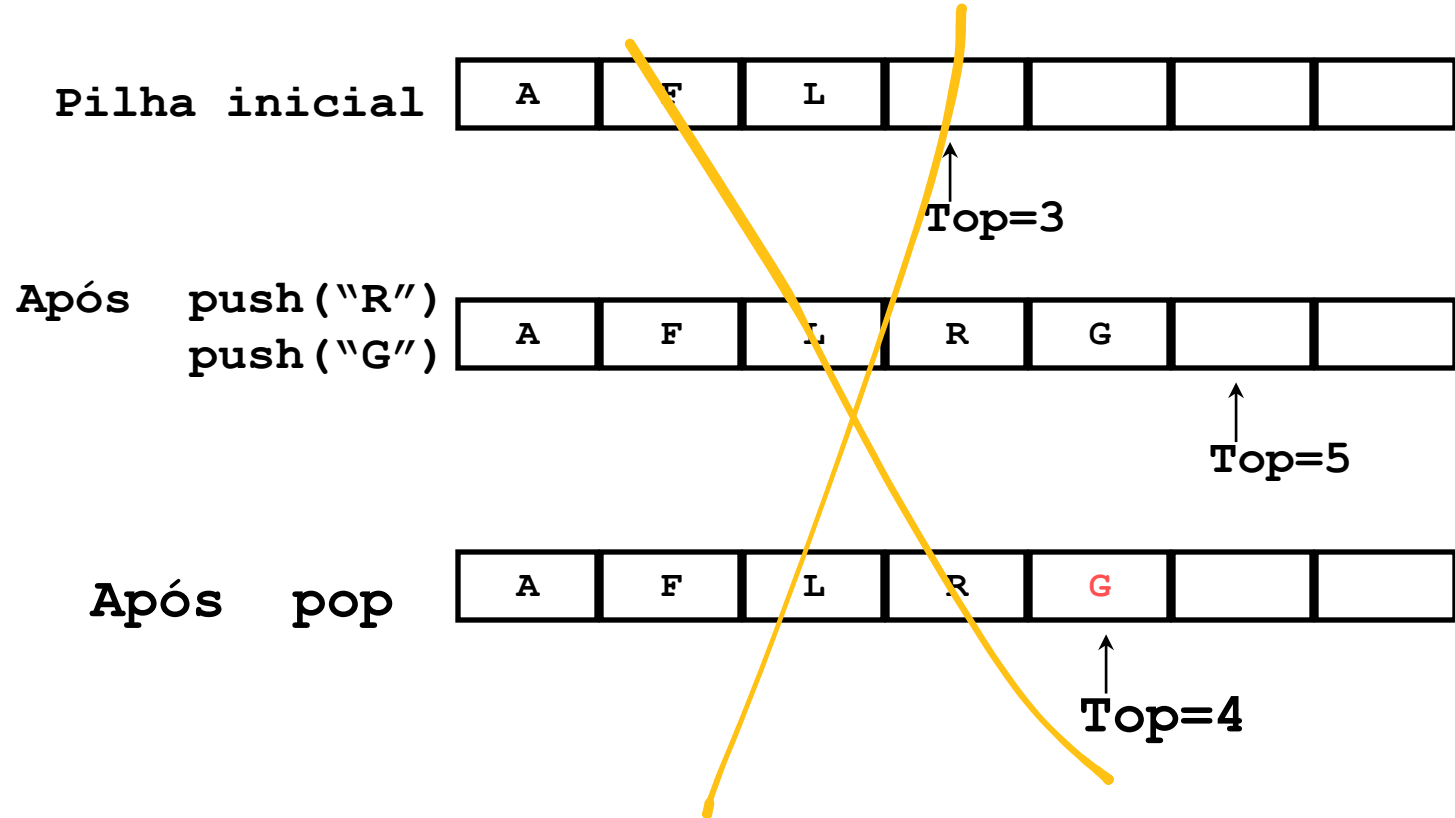
# Pilha - Especificação Formal

- ▶ `/* Construir uma nova pilha`
- ▶ Pre-condição: `(max_itens > 0) && (item_size > 0)`
- ▶ Pos-condição: `retorna um ponteiro para uma pilha vazia */`
- ▶ `/* Retira o elemento do topo da pilha`
- ▶ Pre-condição: `(s foi criada por stkCria) && (cabe um elemento na pilha) && (item != NULL)`
- ▶ Pos-condição: `item foi adicionado ao topo da pilha s */`
- ▶ `/* Retira o elemento no topo da pilha`
- ▶ Pre-condição: `(s foi criada por stkCria) && (existe pelo menos um item na pilha)`
- ▶ Pos-condição: `elemento do topo da pilha foi removido */`

## Pilha - Implementação em Vetor

- ▶ Elementos são armazenados em um vetor  $S$ .
- ▶ O final da pilha fica na posição 0 -  $S[0]$ .  $\text{Max}[tm - 1]$
- ▶ A variável  $top$  armazena o número de elementos na pilha
- ▶ Push armazena um item em  $S[top]$ , e incrementa  $top$
- ▶ Pop decrementa  $top$  e retorna  $S[top]$

## Pilha - Funcionamento em um vetor





## Pilha - Implementação – stack.h

- ▶ typedef struct \_stack\_ Stack;
- ▶ Stack \*stkCreate ( int max);
- ▶ int    stkPush( Stack \*s, void \*elm );
- ▶ void \*stkPop( Stack \*s );
- ▶ void \*stkTop( Stack \*s );
- ▶ int    stkIsEmpty( Stack \*s );
- ▶ int    stkDestroy (Stack \*s);
  
- ▶ Implementar as funções

## STACK.C

```
typedef struct _stack_  
{  
    int maxElms;  
    int top;  
    void **elm;  
} Stack;
```

```
Stack *stkCreate( int max)  
{  
    Stack *s;  
    if( max > 0){  
        s = (Stack *)malloc(sizeof(Stack));  
        if( s != NULL){  
            s->elm = (void **)malloc(sizeof(void *)*max);  
            if(s->elm != NULL){  
                s->maxElms = max;  
                s->top = -1;  
                return s;  
            }  
            free (s);  
        }  
    }  
    return NULL;  
}
```

```
int stkDestroy( void)  
{  
  
    if( s != NULL){  
        if (s->top < 0){  
            free(s->elm);  
            free(s);  
            return TRUE;  
        }  
    }  
    return FALSE;  
}
```

```
int    stkPush( Stack *s, void *elm )
{
    if (s != NULL){
        if (s->top < s->maxElms) {
            s->top++;
            s->elm[s->top] = elm;
            return TRUE;
        }
    }
    return FALSE;
}
```

```
void *stkPop( Stack *s )
{
    void *aux;
    if (s != NULL){
        if (s->top >= 0) {
            aux = s->elm[s->top];
            s->top--;
            return aux;
        }
    }
    return NULL;
}
```

```
void *stkTop( Stack *s )
{
    void *aux;
    if (s !=NULL){
        if (s->top >= 0) {
            aux = s->elm[s->top];
            return aux;
        }
    }
    return NULL;
}
```

```
int    stkIsEmpty( Stack *s )
{
    if (s !=NULL){
        if (s->top < 0) {
            return TRUE;
        }
    }
    return FALSE;
}
```

# Aplicação de Pilha – Avaliação de Expressão Aritmética

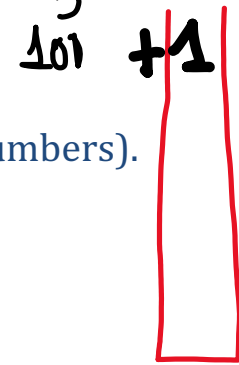
## ▶ Como avaliar

- ▶  $1 + ((2 + 3) * 4 * 5)$  **INFIXA**
- ▶ expressão com todos os parênteses (fully parenthesized)
- ▶ expression composta de parenteses, operadores e operandos (numbers).

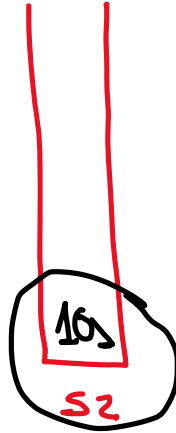
## ▶ Algoritmo:

- ▶ Leia a expressão da esquerda para a direita um caractere por vez
- ▶ Push operandos na pilha de operandos
- ▶ Push operadores na pilha de operadores
- ▶ Ignore fecha parenteses.
- ▶ Ao encontrar um fecha parentese
  - ▶ pop um operador,
  - ▶ pop o número adequado de operandos
  - ▶ push na pilha de operandos o resultado da aplicação do operador aos operandos
- ▶ No final a pilha de operandos possui um único valor com o resultado da expressão

$$\begin{array}{rcl} 3 + 2 & = & 5 \\ 5 * 4 & = & 20 \\ 20 * 5 & = & 100 \end{array}$$

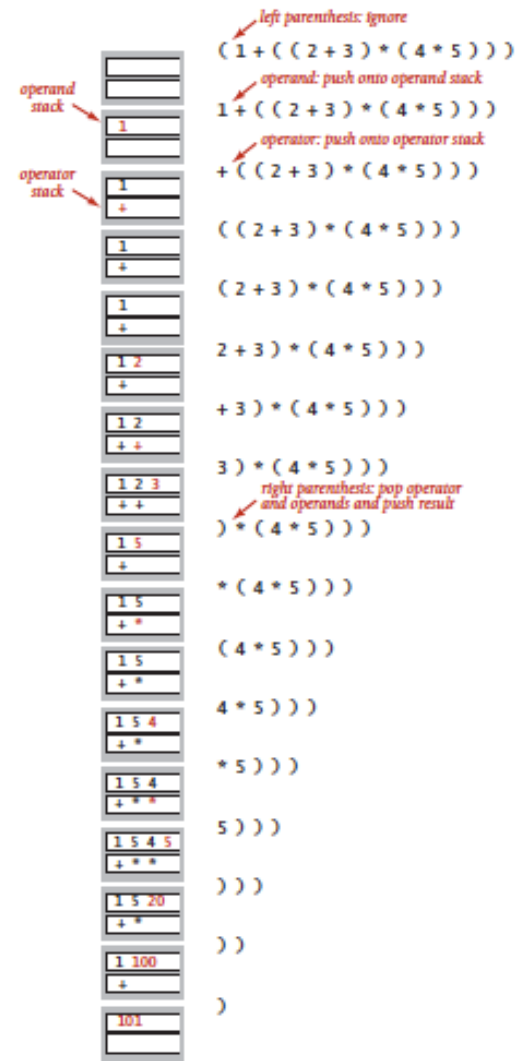


**S1**  
**OPERADORES**



**S2**  
**OPERANDOS**

# Aplicação de Pilha – Avaliação de Expressão Aritmética



# Notação de Expressões Aritméticas

## ► Posição relativa do operador em relação aos operandos

► Prefixa: + A B

► Infixa: A + B

► Posfixa: A B +



## ► Exemplos

Forma Infixa

$$\frac{A + B}{(A + B) - C}$$

$$(A + B) * (C - D)$$

$$A \$ B * C - D + E / F / (G + H)$$

$$((A + B) * C - (D - E)) \$ (F + G)$$

$$A - B / (C * D \$ E)$$

Forma Posfixa

$$\frac{AB +}{\boxed{AB + C} -}$$

$$AB + CD - *$$

$$AB \$ C * D - EF / GH + / +$$

$$AB + C * DE - - FG + \$$$

$$ABCDE \$ * / -$$

Forma Prefixa

$$\frac{+ AB}{- (+ ABC) -}$$

$$* + AB - CD$$

$$+ - * \$ ABCD // EF + GH$$

$$\$ - + ABC - DE + FG$$

$$- A / B * C \$ DE$$

# Avaliação Expressão Posfixa

6 2 3 ⊕ - 3 8 2 / ± \* 2 \$ 3 +

```

opndstk = a pilha vazia;
/* verifica a primeira string lendo um */
/* elemento por vez para symb */
while (nao terminar a entrada) {
    symb = proximo caractere de entrada;
    if (symb eh um operando)
        push(opndstk, symb);
    else { /* symb eh um operador */
        opnd2 = pop(opndstk);
        opnd1 = pop(opndstk);
        value = resultado de aplicar symb a opnd1 e opnd2;
        push(opndstk, value);
    } /* fim else */
} /* fim while */
return(pop(opndstk));
    
```

7  
1

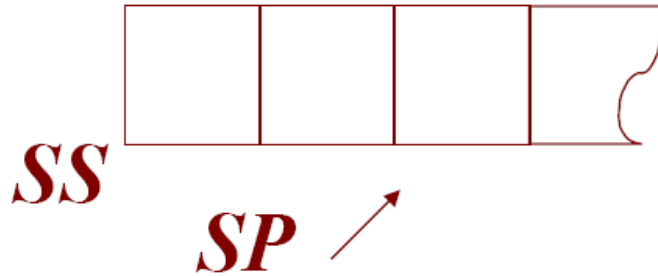
3+4=7  
8/2=4  
6-5  
3+2=5

| symb | opnd1 | opnd2 | value | opndstk |
|------|-------|-------|-------|---------|
| 6    |       |       |       | 6       |
| 2    |       |       |       | 6,2     |
| 3    |       |       |       | 6,2,3   |
| +    | 2     | 3     | 5     | 6,5     |
| -    | 6     | 5     | 1     | 1       |
| 3    | 6     | 5     | 1     | 1,3     |
| 8    | 6     | 5     | 1     | 1,3,8   |
| 2    | 6     | 5     | 1     | 1,3,8,2 |
| /    | 8     | 2     | 4     | 1,3,4   |
| +    | 3     | 4     | 7     | 1,7     |
| *    | 1     | 7     | 7     | 7       |
| 2    | 1     | 7     | 7     | 7,2     |
| \$   | 7     | 2     | 49    | 49      |
| 3    | 7     | 2     | 49    | 49,3    |
| +    | 49    | 3     | 52    | 52      |



# Aplicações de Pilha

- *Hardware do PC*  
*Seção de Pilha (SS)*



\* *PUSH AX*  
*MOV [SS], AX*  
*INC SP*  
\* *POP AX*

## - Pilha de Registro de Ativação



1. void main (void) {

2.     →

3.     →

4. → função1();

5.     .

6.     for (. 0000

7.         funcao2();

8.     if ....

▶ .



E D C B A

→ função1()  
12

14  
→ 17

funcao2()  
|||  
|||  
|||

A B C D E

# Verificação de Parênteses

$() \quad [] \quad \{\} \quad \{[()]\} \quad [(())]$

*Parêntese à esquerda: abre escopo*

*Parêntese à direita: fecha escopo*

*Profundidade de aninhamento: abertos*

*Exemplo(1): Um único tipo de parêntese*

$7 - (x * ((x + y) / (x - 3) + y) / 2)$   
 $\begin{matrix} 0 & 1 & 2 & & 3 & 4 & & 3 & 4 & & 3 & 2 & & 1 & & 0 \end{matrix}$

*O.k.*

$(A + B) ) - (C + D$   
 $\begin{matrix} 1 & & 0 & -1 & \leftarrow & 0 \end{matrix}$

$((A + B)$   
 $\begin{matrix} 1 & 2 & & 1 \end{matrix}$

*ñ O.k.*  
*ñ O.k.*

*É suficiente um programa com incremento / decremento para parênteses.*

## Verificação de parênteses de vários tipos

*Algoritmo básico:*

Até final da expressão

*1: Se início do escopo empilha.*

*Guarda, na ordem do último aberto, os escopos*

*2: Se fim de escopo desempilha*

*Examina se o finalizador atual corresponde ao último aberto*

*3: Se pilha vazia ou não  
coincidir → erro.*

---

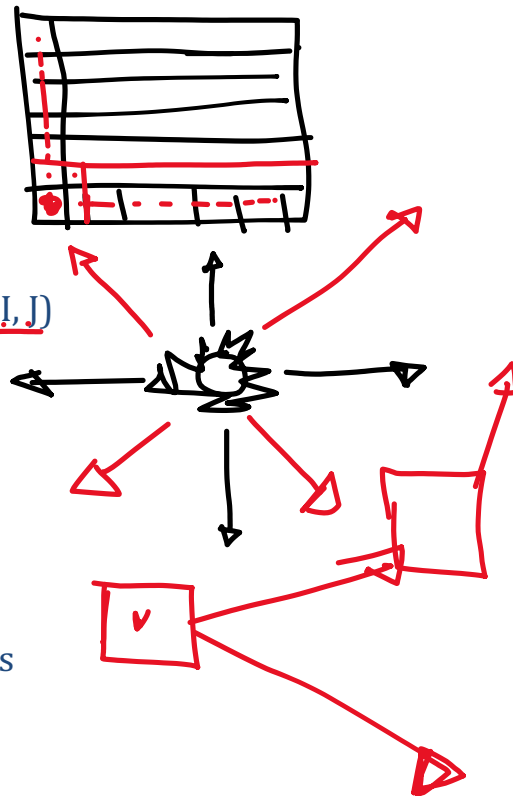
## *Pseudo código*

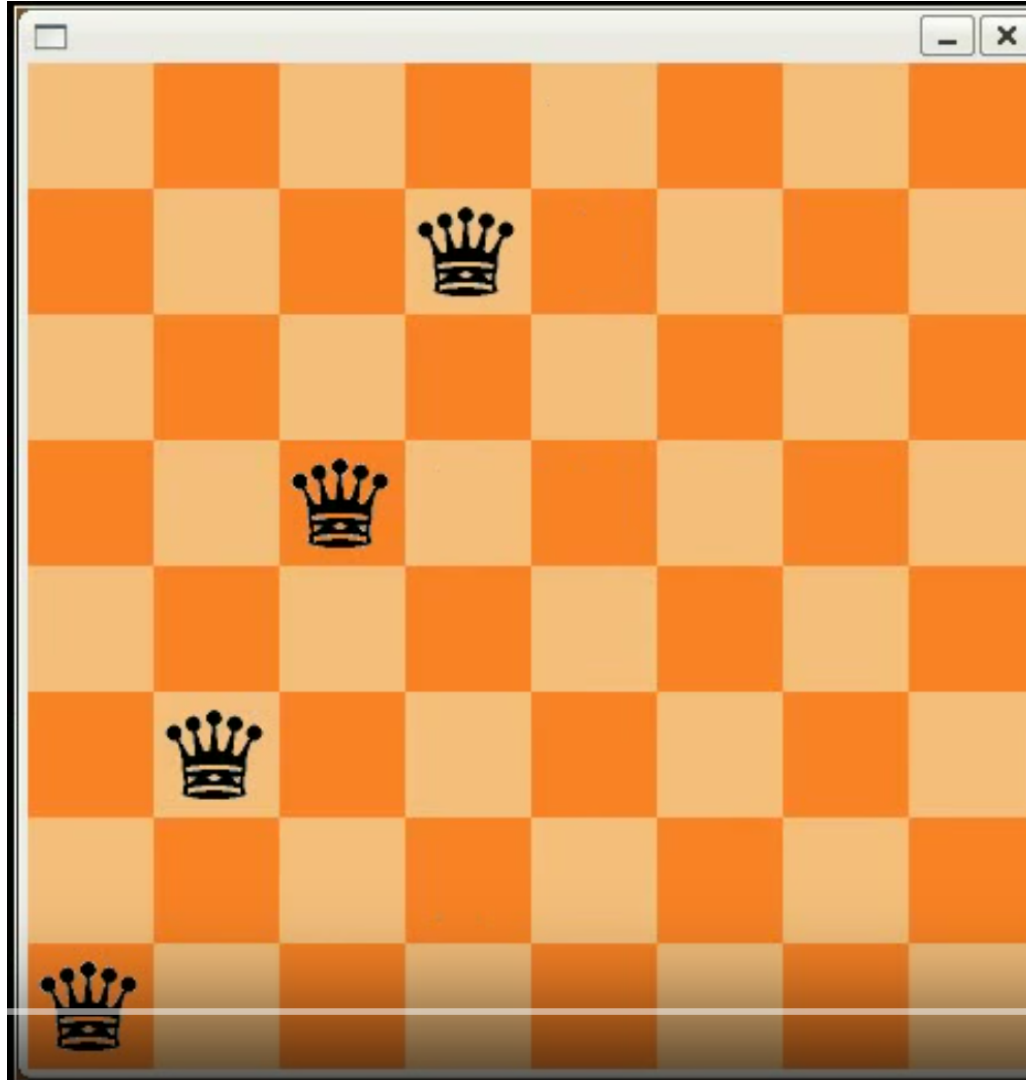
```
válido := TRUE;
inicializa pilha S;
enquanto (não fim de expressão) {
    lê_próximo (símbolo) da expressão
    se (símbolo = '(' ou símbolo='[' ou símbolo = '{') faça
        push (S, símbolo);
    se (símbolo = ')' ou símbolo=']' ou símbolo = '}') faça
        se (empty (S)) faça
            válido := FALSE;
        senão {
            I = pop (S);
            se (I não corresponde a símbolo) faça
                válido := FALSE;
        }
}
se (não empty (S)) válido := FALSE;
se (válido) ok
```

---

# Problema das N-Rainhas

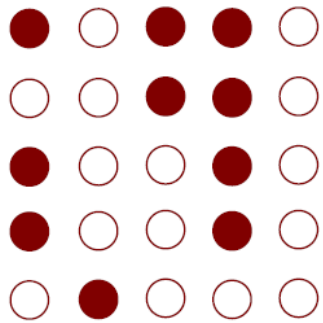




- ▶ colocar 8 rainhas em tabuleiro de xadrez sem se atacarem
- ▶ Genérico: Colocar N Rainhas num tabuleiro N x N.
  - ▶ usa pilha para guardar onde cada rainha é colocada. (numrainha, I, J)
  - ▶ Guarda quantas linhas foram preenchidas até o momento (L).
  - ▶ Para cada nova rainha
    - ▶ Coloca a posição da nova rainha na pilha
    - ▶ Enquanto há conflito muda para a próxima coluna à direita.
    - ▶ Quando não há mais conflitos,  $L++$
  - ▶ Se não for possível colocar uma rainha, após vários deslocamentos
    - ▶ posicionamento atual das demais rainhas está impedindo
    - ▶ Fazer backtracking (retrocesso) para a última rainha colocada
      - Tentar coloca-la numa nova posição
    - ▶ continuar o restante do processo a partir dessa nova posição.
- ▶ Este raciocínio é seguidamente repetido até que encontremos uma configuração sem ataques.







# Preenchimento de Regiões

|   |                                    |                      |
|---|------------------------------------|----------------------|
|  | <b>1 0 1 1 0</b>                   | <b>2 0 4 6 0</b>     |
|  | <b>0 0 1 1 0</b>                   | <b>0 0 1 4 0</b>     |
|  | <b>1 0 0 1 0</b>                   | <b>5 0 0 8 0</b>     |
|  | <b>1 0 0 1 0</b>                   | <b>1 0 0 8 0</b>     |
|  | <b>0 1 0 0 0</b>                   | <b>0 1 0 0 0</b>     |
| <i>imagem formada<br/>por <b>PIXELS</b></i>                                       | <i>matricial<br/>monocromática</i> | <i>policromática</i> |

---

*Pixels conectados: vizinhos com mesma cor*

*Para pintar uma região:*

- Um ponto inicial dentro da região*
- Uma nova cor*



# Preenchimento de Regiões

## Algoritmo de pintura

*Sugestão: Usar pilha para armazenar temporariamente pontos a serem pintados*

```
pinta (int x, int y, int cor)  
  push inicial  
  guardar cor_antiga  
  enquanto pilha não vazia  
    pop ponto p  
    se cor do ponto_p = cor_antiga  
      push 4-conectados  
      pinta ponto_p com cor  
    fim  
fim
```

# Notação Infixa, Posfixa, Prefixa

- Referente à posição do operador em relação aos operandos

- $A+B$   $+AB$   $AB+$

- $\text{add}(A, B)$

- Conversão de notação

- Infixa para posfixa

|   |   |
|---|---|
| $\begin{array}{c} A + B * C \\ \underbrace{\hspace{1.5cm}} \\ A \quad BC^* + \end{array}$ | $\begin{array}{c} (A + B) * C \\ \underbrace{\hspace{1.5cm}} \\ AB + C^* \end{array}$ |
|---|---|

- Operações com maior precedência convertidas antes
- Desnecessário uso de parênteses
- Exponenciação  $^$  Multiplicação & Divisão  $*$  / Adição & Subtração  $+$  -

→ Infixa para prefixa

|                |               |
|----------------|---------------|
| $A + B * C$    | $(A + B) * C$ |
| $+A \quad *BC$ | $*+ABC$       |

1.  $A ^ B * C - D + E / F / ( G + H )$

|                              |            |       |
|------------------------------|------------|-------|
| $AB^{\wedge}$                | $EF/$      | $GH+$ |
| $AB^{\wedge}C^{*}$           | $EF/GH+ /$ |       |
| $AB^{\wedge}C^{*}D-EF/GH+/+$ |            |       |
| <b>Posfixa.</b>              |            |       |

|                              |           |       |
|------------------------------|-----------|-------|
| $^{\wedge}AB$                | $/EF$     | $+GH$ |
| $^{*^{\wedge}}ABC$           | $//EF+GH$ |       |
| $+^{-*^{\wedge}}ABCD//EF+GH$ |           |       |
| <b>Prefixa.</b>              |           |       |

## Avaliando uma expressão posfixa

- ▶ Se for operando, empilha.



- ▶ se operador, desempilha os 2 operandos da pilha, aplica o operador e empilha o resultado para os próximos operadores.

► Ex.: 6 2 3 + - 3 8 2 / + \* 2 ^ 3 +

| <i>termo</i> | <i>ação</i>                 | <i>pilha</i> |
|--------------|-----------------------------|--------------|
| 6            | <i>empilha</i>              | 6 ←          |
| 2            | <i>Empilha</i>              | 6 2 ←        |
| 3            | <i>Empilha</i>              | 6 2 3 ←      |
| +            | <i>desempilha e soma</i>    | 6 5 ←        |
| -            | <i>desempilha e subtrai</i> | 1 ←          |
| 3            | <i>Empilha</i>              | 1 3 ←        |
| 8            | <i>Empilha</i>              | 1 3 8 ←      |
| 2            | <i>Empilha</i>              | 1 3 8 2 ←    |
| /            | <i>desempilha e divide</i>  | 1 3 4 ←      |

## Algoritmo de avaliação de expressão posfixa.

```
str := expressão_posfixa  
s := pilha_vazia  
enquanto (não fim str)  
    elemento := le_elemento (str)  
    se elemento é operando  
        empilha (s, elemento)  
    senão  
        operando1 := desempilha (S)  
        operando2 := desempilha (S)  
        valor := aplica (operando1, operando2, elemento)  
        empilha (S, valor)  
    fim_se  
fim_enquanto  
resultado := desempilha (S)
```



- Conversão de infixa para posfixa



*Se precedência do operador\_infixa for maior que operador\_pilha então empilha operador\_infixa*


**Exemplo:**  $A + B * C$

| <i>termo</i> | <i>posfixa</i> | <i>pilha</i>       |
|--------------|----------------|--------------------|
| $A$          | $A$            | $\cdot \leftarrow$ |
| $+$          | $A$            | $+ \leftarrow$     |
| $B$          | $AB$           | $+ \leftarrow$     |
| $*$          | $AB$           | $+* \leftarrow$    |
| $C$          | $ABC$          | $+* \leftarrow$    |
| $\cdot$      | $ABC*$         | $+ \leftarrow$     |
| $\cdot$      | $ABC*+$        | $\cdot \leftarrow$ |

## Algoritmo de conversão infix a posfixa

```
s := pilha_vazia  
infixa := expressao_infixa  
enquanto (nao fim infix a)  
    simbolo := le_infixa()  
    se simbolo for operando  
        inclui em posfixa  
    senão  
        enquanto (nã o pilha vazia & precedência > (stacktop (s), simbolo))  
            simbolo_topo := pop (s)  
            inclui simbolo_topo em posfixa  
        fim_enq  
        push (s, simbolo)  
    fim_se  
fim_enq  
enquanto ( nã o empty (s) )  
    simbolo_topo := pop (s)  
    inclui simbolo_topo em posfixa  
fim_enq
```

*Enquanto (operador topo precede operador  
símbolo) desempilha)*



## Infixa para posfixa com parênteses

### *Alterações no algoritmo anterior*

- *Função precedência > (pilha, símbolo)*
- *Quando símbolo = '(' ele é introduzido na pilha. Para isso:  
Precedência > (pilha, '(') - .F.  
i.e., qualquer elemento na pilha é inferior a '('*
- *Quando símbolo = ')' retira todos da pilha até o '('.* Basta:  
*Precedência > (pilha, ')') - .V.  
i.e., qualquer elemento na pilha é superior a ')'. Exceto '('*

Ex.:  $\Rightarrow (A+B)*C$

| <i>termo</i> | <i>posfixa</i> | <i>pilha</i> |
|--------------|----------------|--------------|
| (            | .              | ( ←          |
| <i>A</i>     | <i>A</i>       | ( ←          |
| +            | <i>A</i>       | ( + ←        |
| <i>B</i>     | <i>AB</i>      | ( + ←        |
| )            | <i>AB+</i>     | . ←          |
| *            | <i>AB+</i>     | * ←          |
| <i>C</i>     | <i>AB+C</i>    | * ←          |
| .            | <i>AB+C*</i>   | . ←          |

**Infixa para posfixa com parêntese**

**substituição de *push (S, simbolo)* por:**

```
if ( empty (S) ou simbolo ≠ ')' )  
    push (S, simbolo)  
else  
    pop (S)
```

# Exemplo

**Ex.:**  $((A-(B+C)) * D) ^ E + F$

| <i>item</i> | <i>posfixa</i> | <i>pilha</i> |
|-------------|----------------|--------------|
| (           | .              | (            |
| (           | .              | ((           |
| A           | A              | ((           |
| -           | A              | ((-          |
| (           | A              | ((-(         |
| B           | AB             | ((-(         |
| +           | AB             | ((-(+        |
| C           | ABC            | ((-(+        |
| )           | ABC+           | ((-          |
| )           | ABC+ -         | (            |
| *           | ABC+ -         | (*           |
| D           | ABC+ - D       | (*           |
| )           | ABC+ - D *     | .            |
| ^           | ABC+ - D *     | ^            |
| E           | ABC+ - D * E   | ^            |

# Recursividade

- ▶ Técnica muito útil
  - ▶ Usada na definição de várias funções matemáticas
    - ▶ fatorial
    - ▶ Fibonacci
    - ▶ Maior denominador comum de Euclides
  - ▶ Definição de estruturas de dados
    - ▶ estruturas recursivas são naturalmente processadas por funções recursivas
  - ▶ Resolução de problemas como Jogos
    - ▶ Torres de Hanoi (simples)
    - ▶ Xadrez (complexos)
      - resolvendo o programa recursivamente, a estrutura recursiva registra os passos executados para chegar na solução



# Recursão - Exemplo

- ▶ Números de Fibonacci
  - ▶ Método de Cálculo

```
fib( n ) = if ( n = 0 ) then 1  
           else if ( n = 1 ) then 1  
           else fib(n-1) + fib(n-2)
```

Implementação em C

```
int fib( n ) {  
    if ( n < 2 ) return 1;  
    else return fib(n-1) + fib(n-2);  
}
```

***Solução simples e elegante, mas ineficiente,  
No entanto, em muitos outros casos ( e.g. busca  
binária) solução eficiente também***

# Aspectos da Recursividade

- ▶ Todo processo recursivo:
  - ▶ Caso base: resolvido sem recursividade;
  - ▶ Método geral que reduz um caso particular a um ou mais casos base.

# Recursividade – Torres de Hanoi

## ► Problema:

- Considere 3 torres ( 1, 2 e 3) e 64 discos com diâmetros decrescentes colocados na torre 1;
- Mover os 64 discos para a torre 3 (usando 2 como suporte), com a restrição que um disco maior não pode ficar sobre um menor;
- Mover (64, 1, 3, 2)

## 5. Torres de Hanói



*Dado 3 pinos A, B, C com  $n$  discos de diferentes tamanhos, inicialmente em A, movê-los para C. Restrição:*

**Objetivo:**

*Mover todos os discos de  $A \rightarrow C$ , usando o auxiliar B*

$N = 1 \quad A \ B \ C \rightarrow A \ B \ C$

$N = 2 \quad \overline{A} \ B \ C \rightarrow \overline{A} \ B \ C \rightarrow A \ B \ \overline{C} \rightarrow A \ B \ \overline{C}$

$N = 3 \quad \overline{\overline{A}} \ B \ C \rightarrow \overline{\overline{A}} \ B \ C \rightarrow \overline{\overline{A}} \ \overline{B} \ C \rightarrow \overline{\overline{A}} \ \overline{B} \ C$

$A \ \overline{B} \ \overline{\overline{C}} \rightarrow A \ \overline{B} \ \overline{\overline{C}} \rightarrow A \ B \ \overline{\overline{C}} \rightarrow A \ B \ \overline{\overline{C}}$

## Recursividade – Torres de Hanoi

```
void move ( int n, int de, int para, int temp)
{
    if ( n>0) {
        move ( n-1, de, temp, para)
        printf("Move %d discos de %d para %d",
               n, de, para)
        move ( n-1, temp, para, de)
    }
}
```

## Algoritmo

```
programa Hanói;  
  lê número discos ( n > 0);  
  fanoi ( 'A', 'B', 'C', N);  
fim;  
  
fanoi (origem, aux, destino, n)  
  caracter: origem, aux, destino;  
  integer: n;  
  {   se (n = 1)  
      escreva (origem, 'para', destino);  
  senão {  
      fanoi (origem, destino, aux, n - 1);  
      fanoi (origem, aux, destino, 1);  
      fanoi (aux, origem, destino, n-1);  
  }  
fim;
```

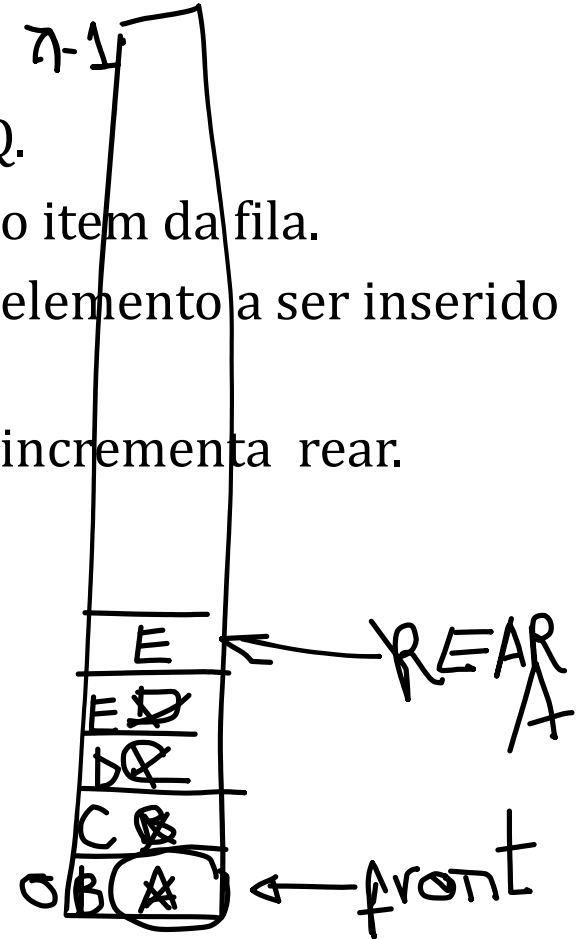
# Fila



- ▶ Coleção em que a inserção é feita numa extremidade e a eliminação na outra. (FIFO: first in, first out).
- ▶  $(a_1, a_2, \dots, a_n)$  eliminações no início
- ▶ inserções no final
- ▶ Operações associadas:
  - ▶ qCreate (F) - criar uma fila F vazia
  - ▶ qEnqueue (x, F) - insere x no fim de F (enqueue)
  - ▶ qIsEmpty (F) - testa se F está vazia
  - ▶ qFirst (F) - acessa o elemento do início da fila
  - ▶ qDequeue (F) - elimina o elemento do início da fila (dequeue)

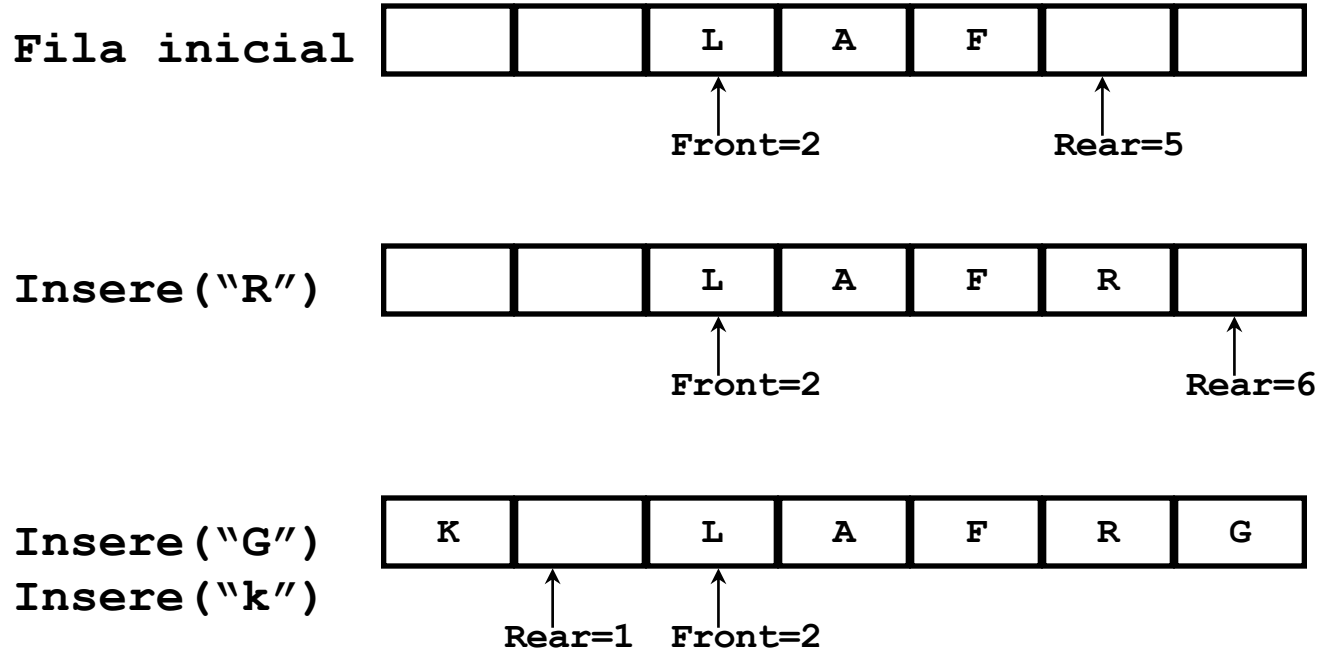
## Fila - Implementação em um Vetor

- ▶ Elementos são armazenados em um vetor Q.
- ▶ A variável `front` guarda o índice do primeiro item da fila.
- ▶ A variável `rear` guarda o índice do próximo elemento a ser inserido na fila.
- ▶ Inserir armazena o elemento em `Q[rear]`, e incrementa `rear`.
- ▶ Elimina incrementa `front`.





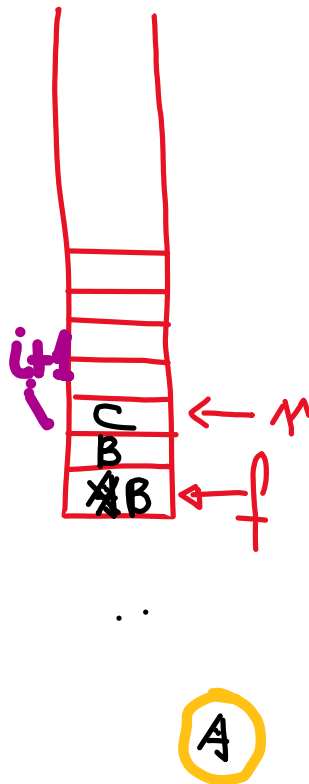
## Fila - Exemplo em um vetor



# Fila - Implementação em Vetor

```
typedef struct _queue_ {
    int front,
        rear;
    int maxItens;
    void **elm;
} Queue;

/* Criar (q) - criar uma fila q vazia */
Queue *qCreate(int max)
{
    Queue *q;
    if( max>0) {
        q = (Queue *)malloc( sizeof(Queue));
        if ( q != NULL ) {
            q->elm = (void **) malloc(sizeof(void *)*max);
            if(q->elm!= NULL) {
                q->front = 0;
                q->rear = -1;
                q->maxItens = max;
                return q;
            }
        }
    }
    return NULL;
}
```



```
/* Inserir (x, F) - insere x no fim de q */
int qEnqueue (Queue *q, void *elm)
{
    if (q != NULL) {
        if (q->rear < q->maxItens) {
            q->rear ++;
            q->elm[q->rear] = elm;
            return TRUE;
        }
    }
    return FALSE;
}

/* Vazia (q) - testa se q está vazia */
int qIsEmpty (Queue *q)
{
    if (q != NULL) {
        if ( q->rear < 0 ) {
            return TRUE;
        }
    }
    return FALSE;
}
```

# Fila - Implementação em Vetor

/\* Primeiro (q) - \*\* acessa o elemento do início da fila \*/

void \*qFirst (Queue \*q )

```
{
    if( q != NULL ) {
        if (q->rear >=0) {
            return q->elm[q->front];
        }
    }
    return NULL;
}
```

int qDestroy(Queue \*q)

```
{
    if( q != NULL ) {
        if (q->rear < 0) {
            free(q->elm);
            free(q);
            return TRUE;
        }
    }
    return FALSE
}
```

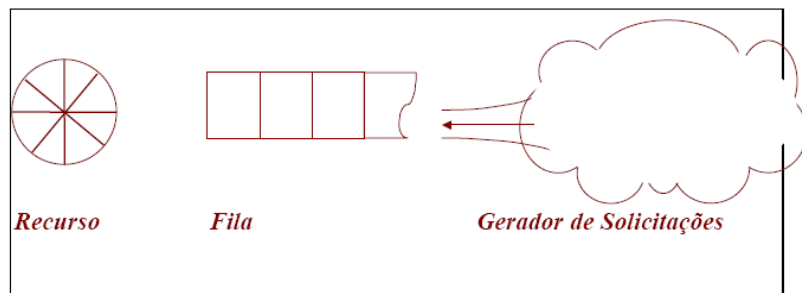
/\* Elimina (F)

\*\* elimina o elemento do início da fila \*/

void \*qDequeue (Queue \*q)

```
{
    void *aux;
    if( q != NULL ) {
        if (q->rear >=0) {
            aux = q->elm[q->front];
            for(i=q->front; i<q->rear; i++) {
                q->elm[i] = q->elm[i+1];
            }
            q->rear--;
            return aux;
        }
    }
    return NULL;
}
```

# Aplicação: Contenção de recursos



Inicializa parâmetros

$$\left\{ \begin{array}{l} prob(chegada) \\ prob(saida) \\ tempodesimulação \end{array} \right.$$

*Enquanto durar simulação faça*

*Chegada (prob-chegada)*

*Se chegou então*

*Se recurso-ocupado então*

*insere na fila*

*Senão utiliza-Recurso ( ),*

*Se recurso-ocupado então*

*Saída (prob-saída)*

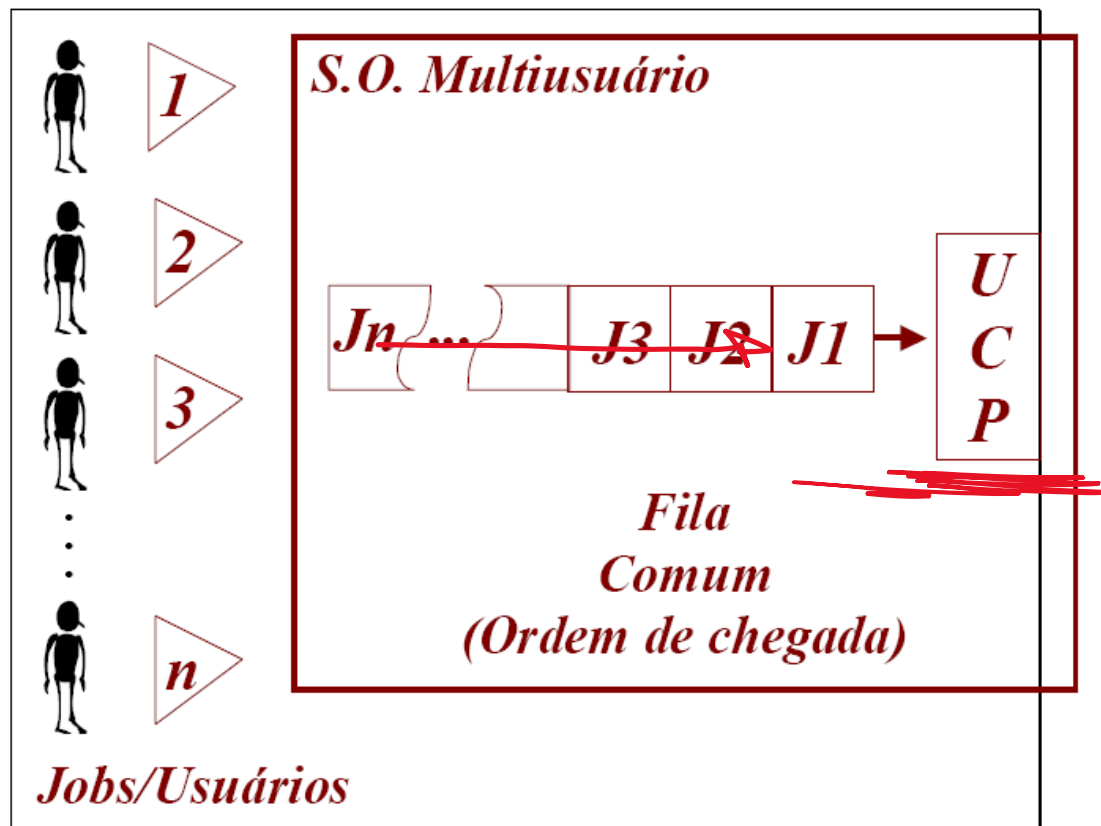
*Se saiu se fila não vazia*

*Remove da Fila*

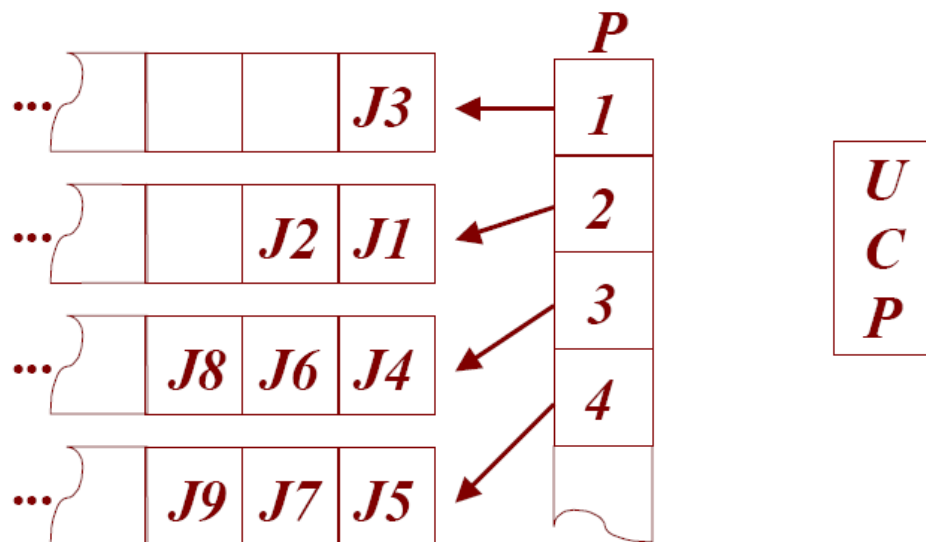
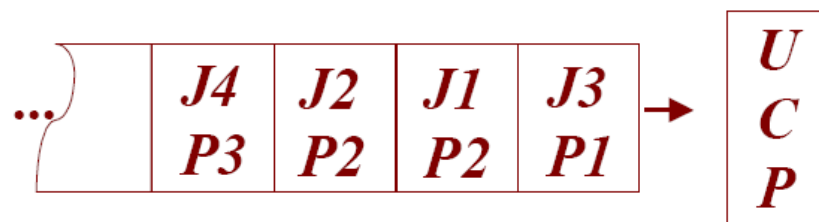
*utiliza recurso ( )*

*Fim*

# Escalonamento de Processos



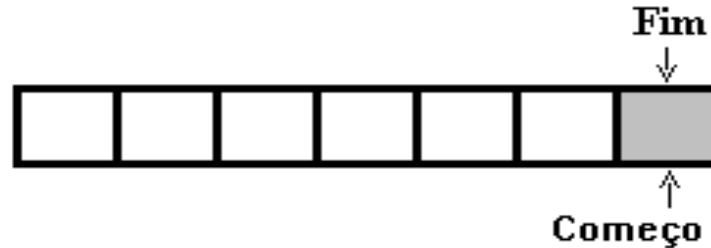
*Ou considerando prioridades*



# Fila - Problemas de Implementação

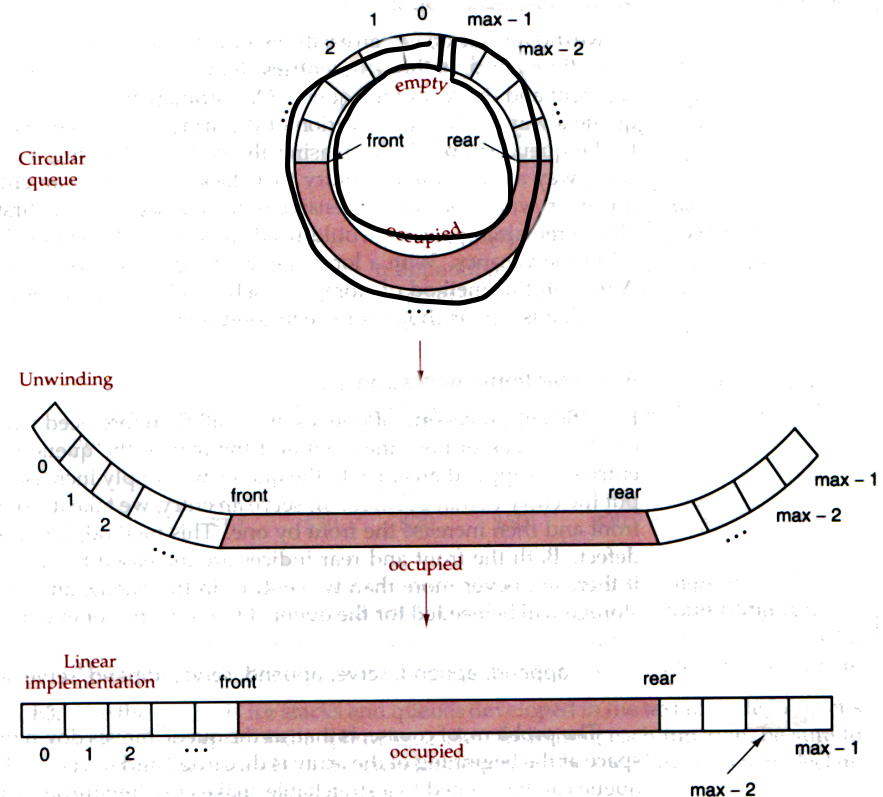
- ▶ Considere a seguinte sequência
  - ▶ QDQDQDQDQD. (Q - queue e D - dequeue)
  - ▶ fila terá sempre 1 ou 0 elementos
  - ▶ num certo instante UM elemento na fila, que ocupa última posição
- ▶ A fila está vazia, mas não cabe mais nenhum elemento
- ▶ Opção: na eliminação(Dequeue) após o incremento de front, verificar se fila vazia

```
if (IsEmpty(q) ) {  
    q->front = -1;  
    q->rear = 0;  
}
```



# Fila - Problemas de Implementação

- ▶ Alternativa:
  - ▶ Forçar rear a usar o espaço liberado por front (Fila Circular)





# Fila - Implementação em Vetor Circular

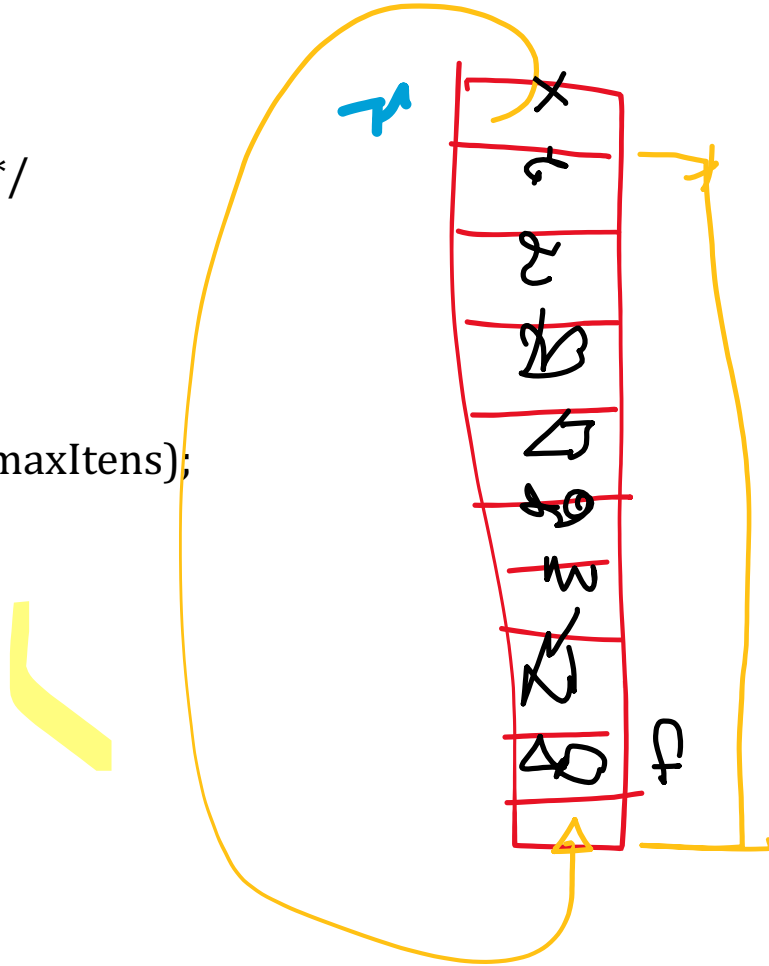
```
typedef struct _queue_ {
    int front,
        rear;
    int maxItens;
    int nElms;
    void **elms;
}Queue;

/* Criar (q) - criar uma fila q vazia */
Queue *qcCreate(int max) {
    Queue *q = (Queue *)malloc( sizeof(Queue));
    if ( max > 0 ) {
        if ( q != NULL ) {
            q->elms = (void **) malloc(sizeof(void *)*max);
            if(q->elms != NULL) {
                q->front = 0;
                q->rear = -1;
                q->maxItens = max;
                q->nElms=0;
                return q;
            }
        }
    }
    return NULL;
}
```

```

/* Inserir (x, F) - insere x no fim de q */
int qcEnqueue (Queue *q, void *elm)
{
    if ( q != NULL )
        if (q->nElms < q->maxItens ) {
            q->rear = qIncrCirc( q->rear, q->maxItens);
            q->elms[q->rear] = elm;
            q->nElms++;
            return TRUE;
        }
        return FALSE;
}

```



## Fila - Implementação em Vetor

```
/* Primeiro (q) -  
**  acessa o elemento do início da fila */  
void *qcFirst (Queue *q ) {  
    void *aux;  
    if (q != NULL) {  
        if(q->nElms>0) {  
            aux = q->elms[q->front];  
            return aux;  
        }  
    }  
    return NULL;  
}
```

```
/* Remove (F)  
**  retira da fila o elemento do início da fila */  
int qcDequeue (Queue *q) {  
    if (q!= NULL){  
        if (q->nElms >0) {  
            aux = q->elms[q->front];  
            q->front = qIncrCirc(q->front, q->MaxItens);  
            return aux;  
        }  
    }  
    return NULL  
}
```

```
▶ int qcDestroy ( Queue *q)
▶ {
▶     if ( q!= NULL) {
▶         if ( q->nElms == 0 ) {
▶             free( q->intens);
▶             free(q);
▶             return TRUE;
▶         }
▶     }
▶     return FALSE;
▶ }
```

```
▶ int qcDecrCirc( int i, int n)
▶ {
▶     if ( i>0 ) {
▶         i--;
▶         return i;
▶     } else {
▶         i = n-1;
▶         return i;
▶     }
▶ }
```

# Fila - Implem. de Fila Circular

- ▶ Como definir fila cheia e fila vazia ????

- ▶ Contador de número de elementos na fila

- ▶ typedef struct \_queue\_ {

- ▶ int front,

- ▶ rear;

- ▶ int nElms;

- ▶ int maxItens;

- ▶ void \*\*qArray;

- ▶ }Queue;

- ▶ Ao inserir e retirar algum item da fila incrementa ou decrementa o número de itens na fila( numItens), que foi inicializado com zero.

- ▶ Os algoritmos de verificação de fila cheia e fila vazia ficam:

```
int IsFull( Queue *q)
```

```
{  
    if ( q->nElms < q->maxItens ) {  
        return FALSE;  
    } else {  
        return TRUE;  
    }  
}
```

```
int IsEmpty (Queue *q)
```

```
{  
    if ( q->nEms == 0) {  
        return TRUE;  
    } else{  
        return FALSE;  
    }  
}
```

- ▶ Considere uma fila representada em um vetor circular. E Implemente as seguintes operações:

A) Promover um elemento que esta no final da fila colocando-o n posições pra frente.

```
int qPromoveUltimo( Queue *q, int n)
```

B) Punir o primeiro elemento da fila colocando-o n posições para tras.

```
int qPunePrimeiro ( Queue *q, int n)
```

## Fila – Implem. de Fila Circular

- ▶ Vetor com m elementos: índices de 0 a m-1
- ▶ Extremidades da fila: front e rear ( front = rear = 0
- ▶ qdo rear = m-1, próximo elemento é inserido na posição 0 (se vazia)

```
int Enqueue (Queue *q, void *b) {  
    if (IsFull(q)) {  
        return FALSE;  
    } else {  
        if (q->rear == (q->maxItens-1) {  
            q->qArray[q->rear] = b;  
            q->rear = 0;  
        } else {  
            q->rear = q->rear + 1;  
        }  
        return TRUE;  
    }  
}
```

```
int Dequeue (Queue *q) {  
    if (IsEmpty(q)) {  
        return FALSE;  
    } else {  
        q->front = (q->front + 1 )% q-  
            >maxItens;  
        return TRUE;  
    }  
}
```

Maneiras diferentes de fazer a mesma coisa:  
ao chegar ao final do vetor retornar pra posição zero

# Fila - Implem. de Fila Circular

- ▶ Como definir fila cheia e fila vazia ????

- ▶ Contador de número de elementos na fila

- ▶ typedef struct \_queue\_ {

- ▶ int front,

- ▶ rear;

- ▶ int numItens;

- ▶ int maxItens;

- ▶ void \*\*qArray;

- ▶ }Queue;

- ▶ Ao inserir e retirar algum item da fila incrementa ou decrementa o número de itens na fila( numItens), que foi inicializado com zero.

- ▶ Os algoritmos de verificação de fila cheia e fila vazia ficam:

```
int IsFull( Queue *q)
```

```
{  
    if ( q->numItens < q->maxItens ) {  
        return FALSE;  
    } else {  
        return TRUE;  
    }  
}
```

```
int IsEmpty (Queue *q)
```

```
{  
    if ( q->numItens == 0 ) {  
        return TRUE;  
    } else {  
        return FALSE;  
    }  
}
```