Tópicos de Programação

Arthur Casals
(arthur.casals@usp.br)

IME - USP

Aula 6:

- Fundamentos de Estruturas de Dados

Na aula passada...

Listas lineares duplamente ligadas:

- Motivação:
  - Algumas vezes pode ser necessário (em termos de eficiência) possuir fácil acesso a todos os nós adjacentes de cada nó na lista
  - Exemplos práticos: lista de músicas, históricos de navegadores

Na aula passada...

Listas lineares duplamente ligadas:

- Vantagens sobre listas ligadas:
  - Pode ser percorrida em ambos os sentidos
  - Operação de exclusão pode ser mais eficiente

Na aula passada...

Listas lineares duplamente ligadas:

› Desvantagens em relação a listas ligadas:

- Cada elemento requer espaço extra para ponteiro adicional
- Ponteiro adicional tem que ser mantido em todas as operações

Na aula passada...

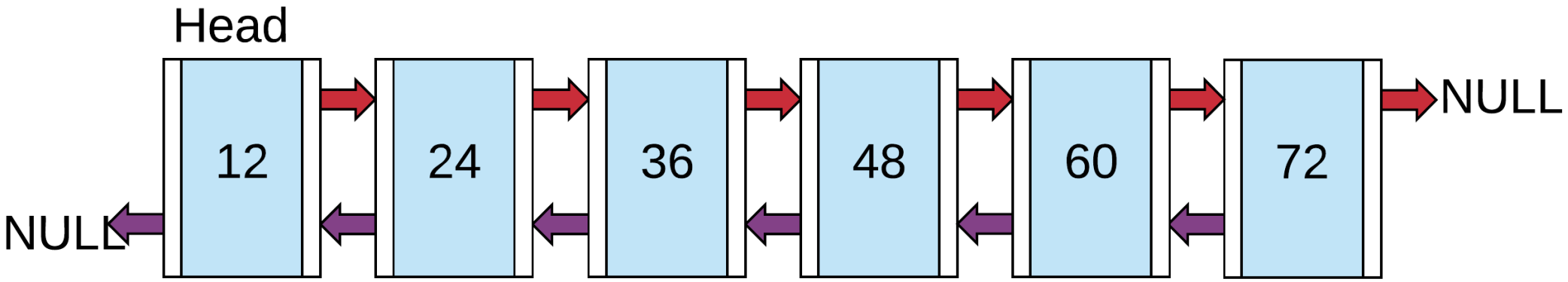
Listas lineares duplamente ligadas:

- Estrutura:
  - Uma lista ligada é formada por nós
  - Cada lista possui um nó inicial (*head*)
  - Cada nó possui uma referência (apontador) para o próximo nó da lista
  - Cada nó possui uma referência (apontador) para o nó anterior da lista

Na aula passada...

Listas lineares duplamente ligadas:

› Representação:



Na aula passada...

Operações em listas duplamente ligadas:

› Inserção de elementos

› Remoção de elementos

› Tamanho

› Busca (verificar se um elemento pertence à lista)

› Troca de posições de nós (sem trocar os dados)

Na aula passada...

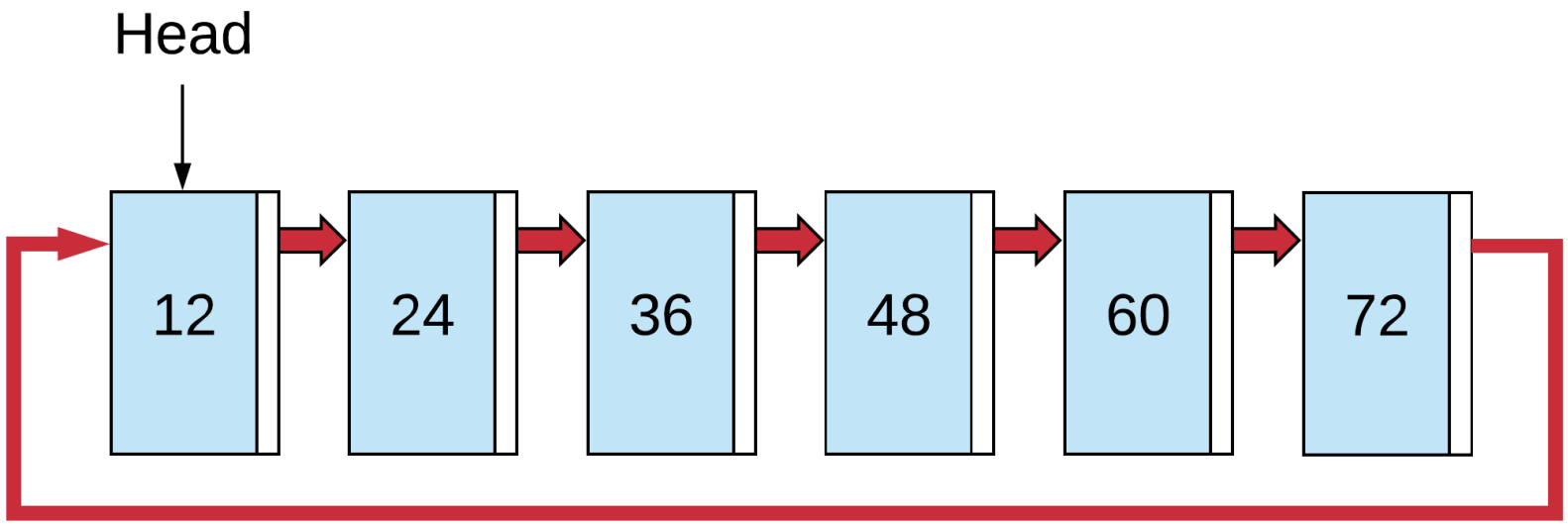
Listas circulares:

- Motivação:
  - Alocação de memória previsível (evita *malloc/free*)
  - É mais simples percorrer uma lista circular
  - Sequenciamento de dados limitados
  - Exemplo prático: de quem é a vez?, listas de música

Na aula passada...

Listas circulares ligadas:

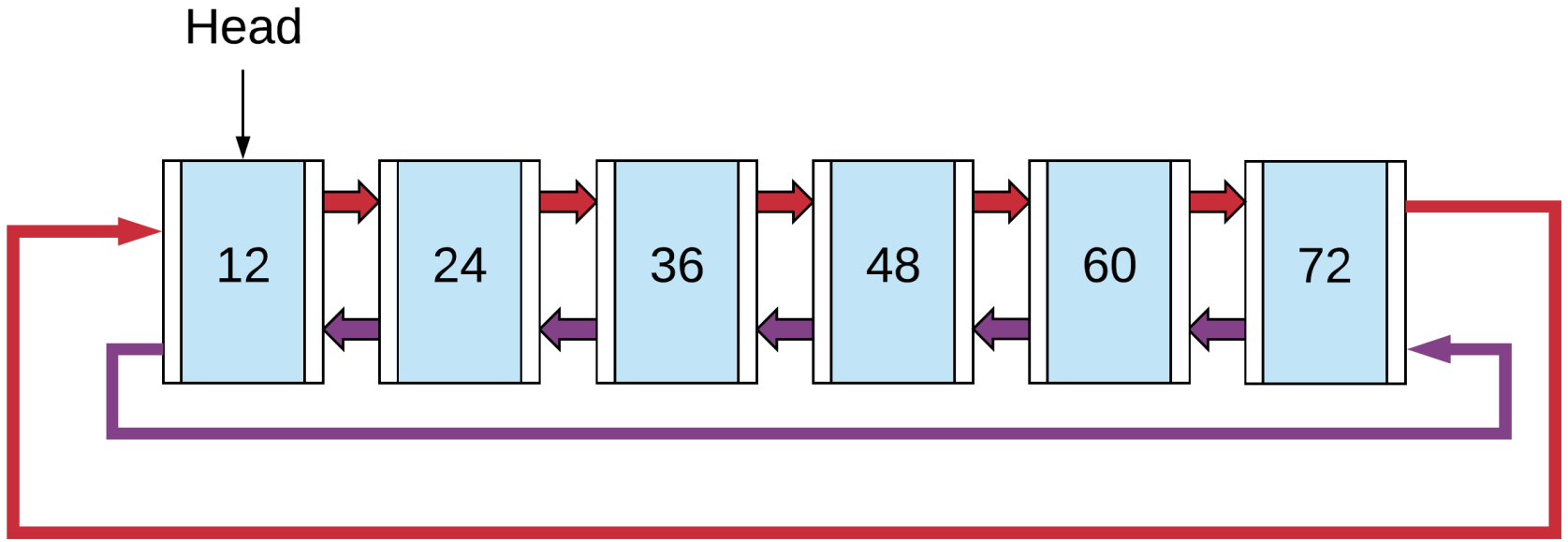
› Representação:



Na aula passada...

Listas circulares duplamente ligadas:

› Representação:



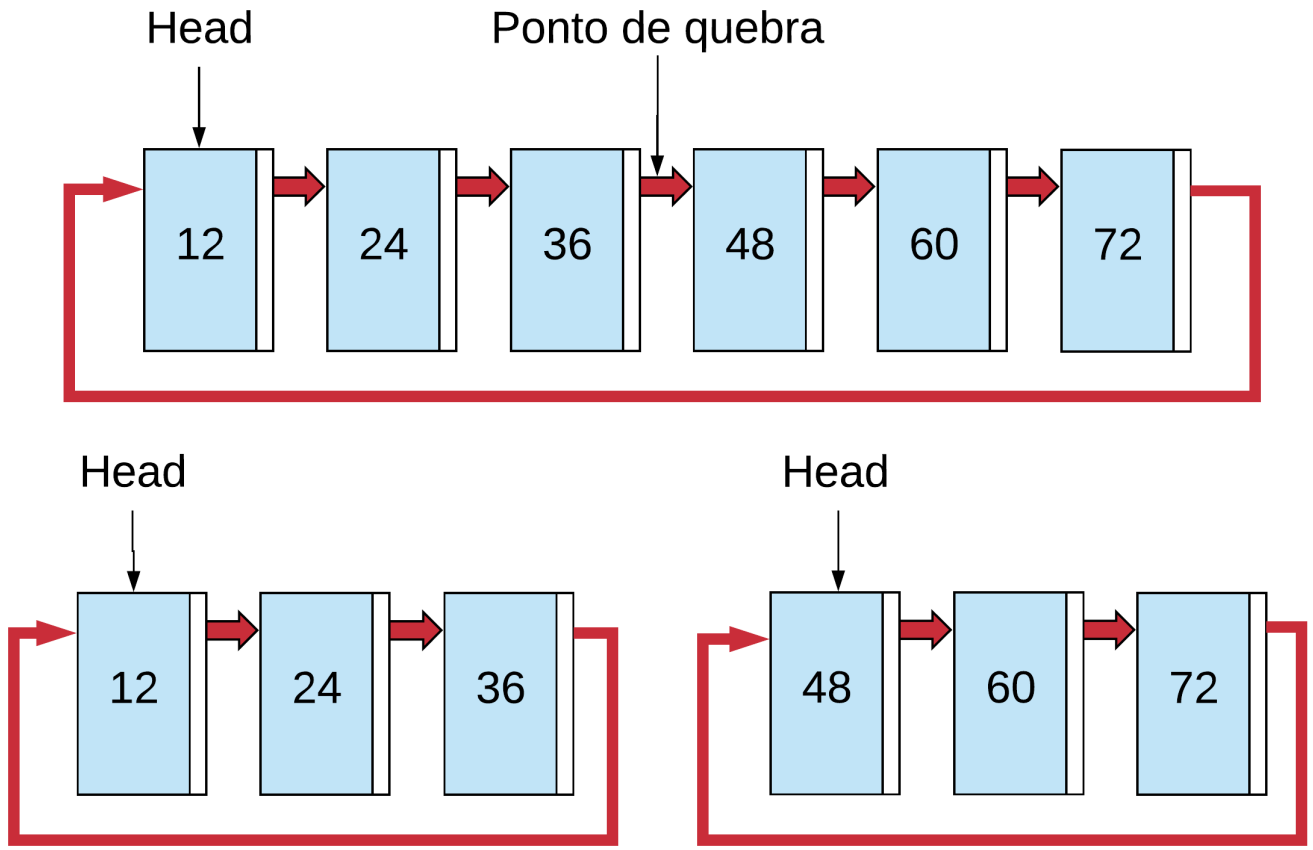
Na aula passada...

Outras operações em listas circulares:

- Dividir uma lista circular em duas listas circulares menores
- Repetir uma operação *n* vezes
- Auto-organização

Na aula passada...

Dividir uma lista circular em duas menores:



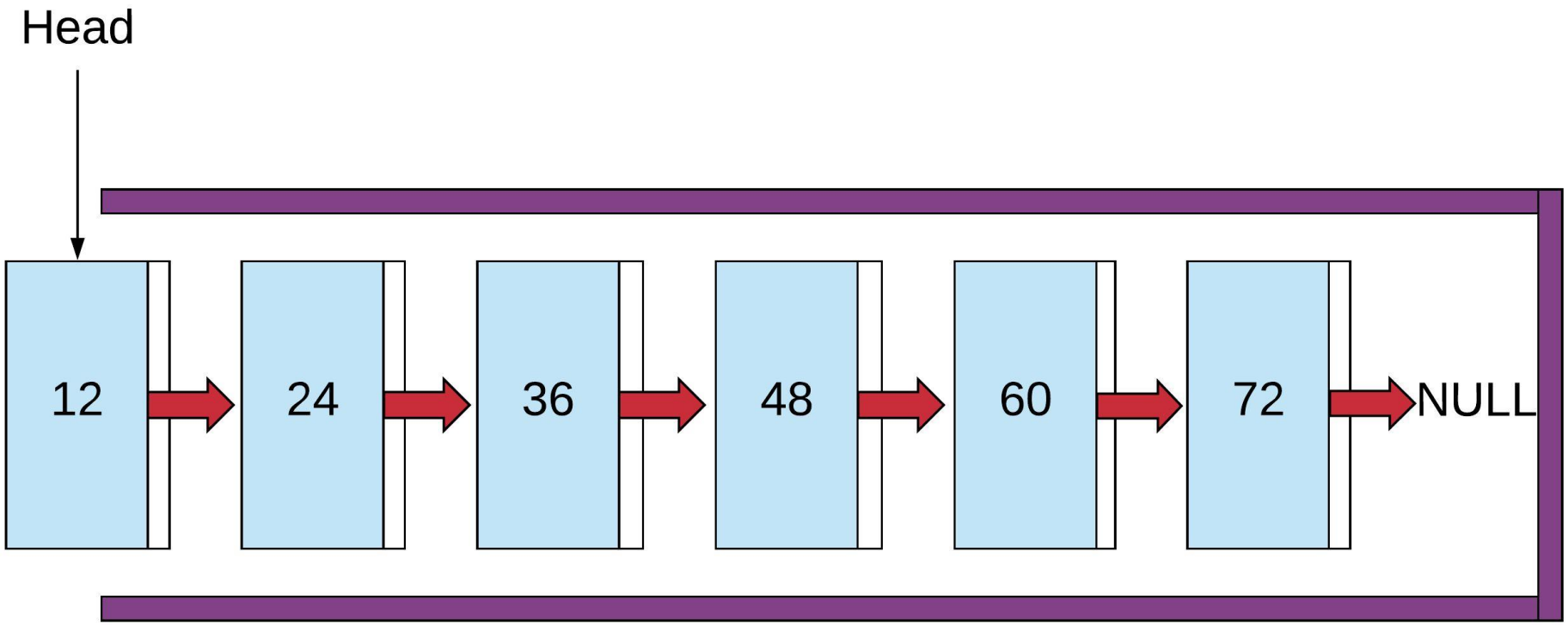
Fundamentos de estruturas de dados

Pilha:

› "Pode ser vista como uma **lista linear ligada** na qual **todos os acessos** são realizados **somente** em uma das extremidades (**topo**)"

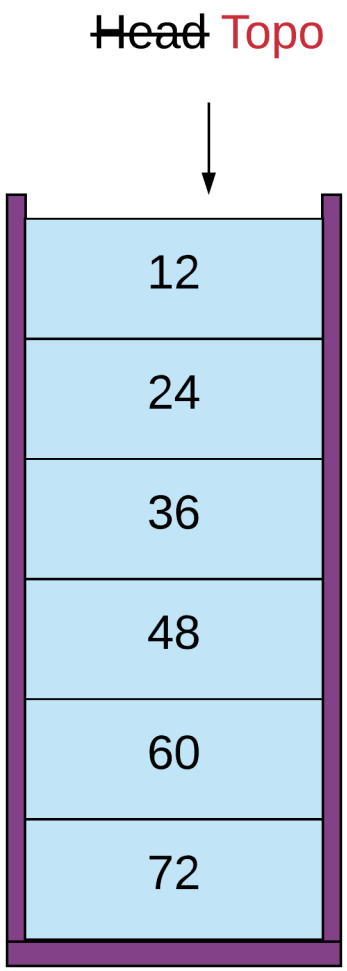
Fundamentos de estruturas de dados

Pilha:



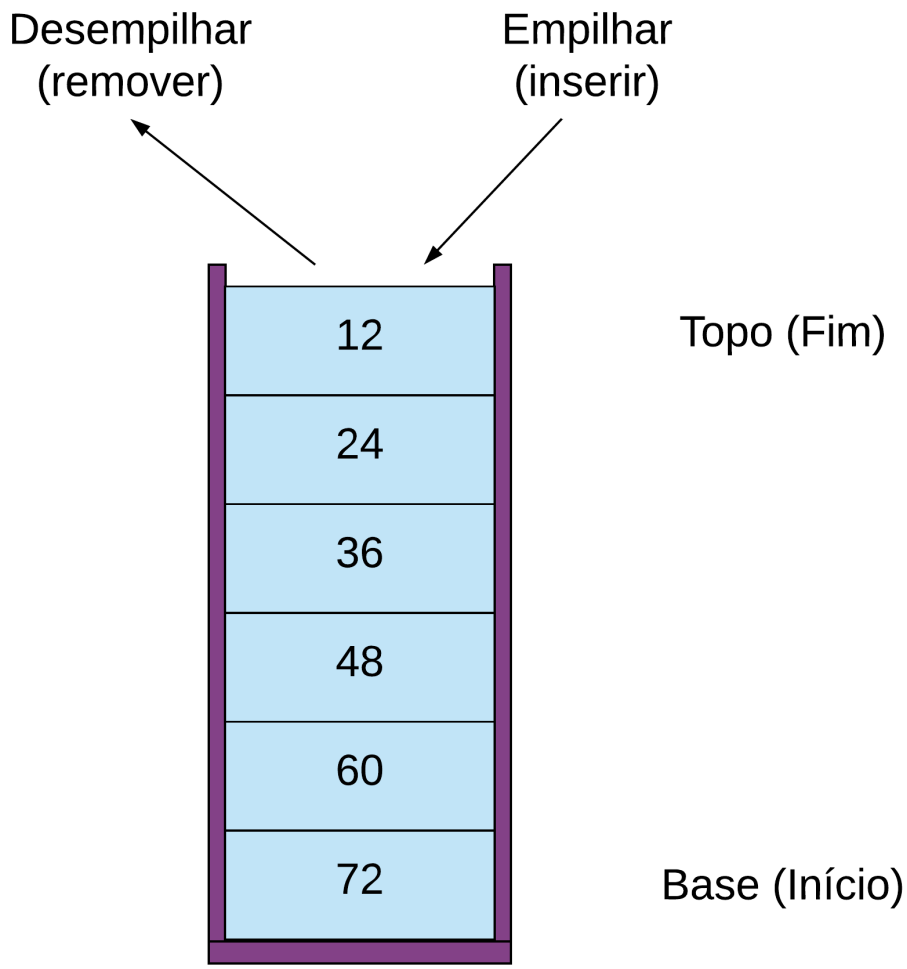
Fundamentos de estruturas de dados

Pilha:



Fundamentos de estruturas de dados

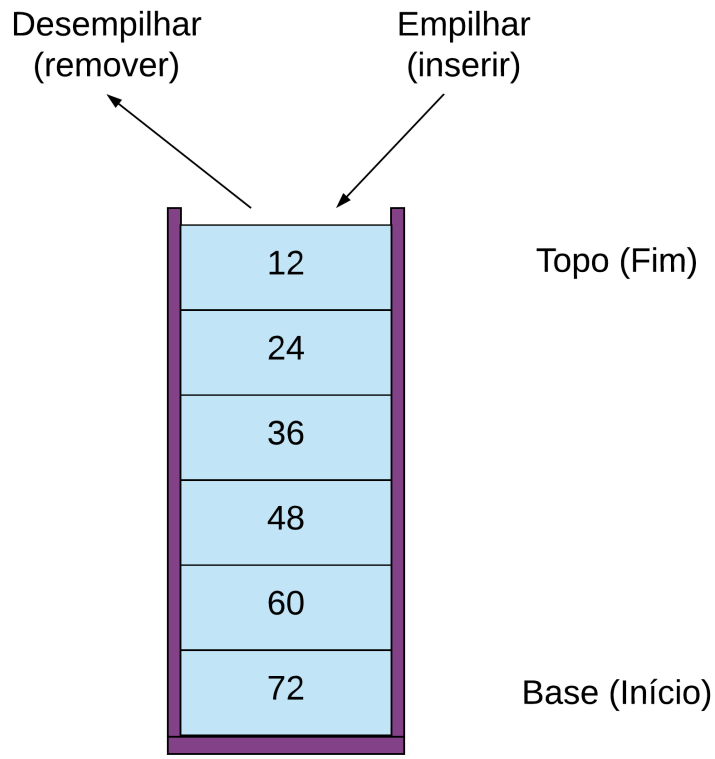
Pilha:



Fundamentos de estruturas de dados

Pilha:

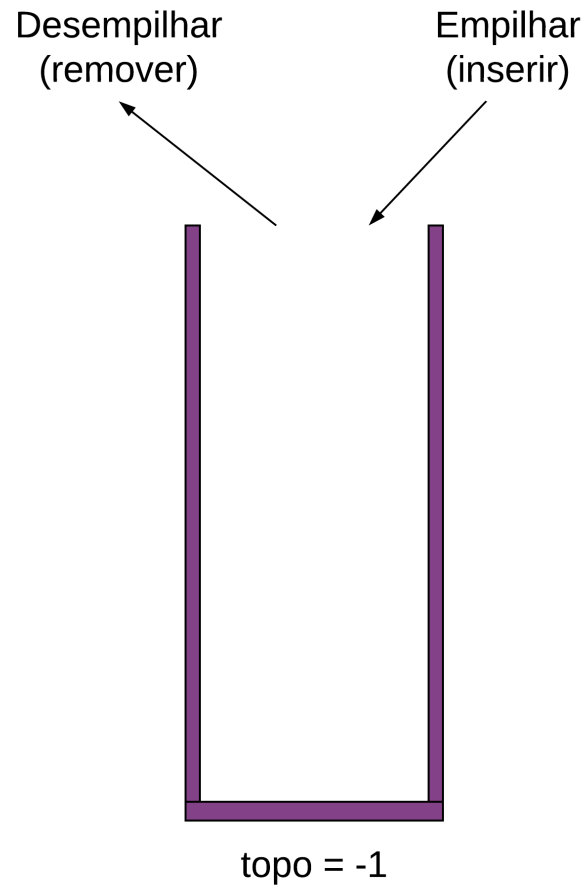
- O último elemento a ser inserido é o primeiro a ser removido: **LIFO** (*Last In, First Out*)
- Também pode ser representada utilizando-se uma lista linear (*array*)



Fundamentos de estruturas de dados

Pilha – representação utilizando *arrays* :

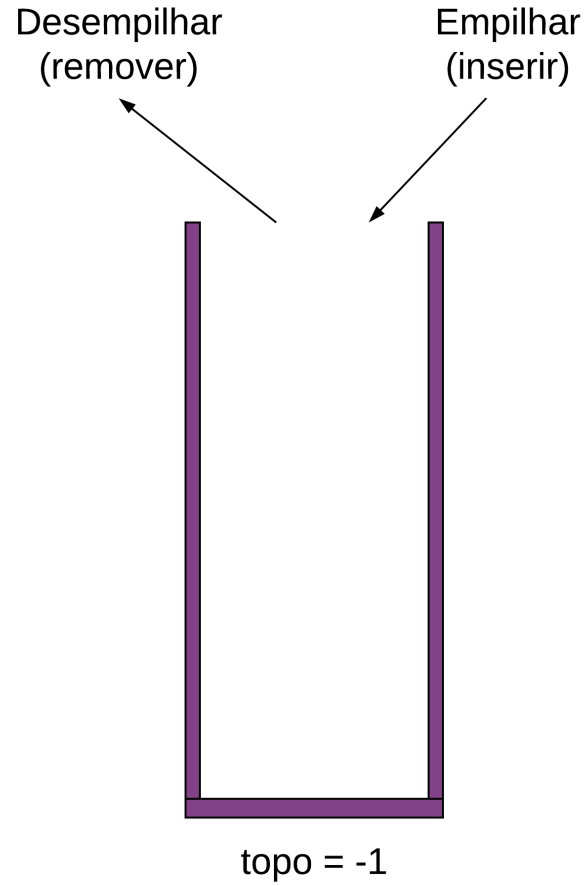
```
#define MAXPILHA 8

struct pilha {

        int topo;

        int item[MAXPILHA];

}
```



Fundamentos de estruturas de dados

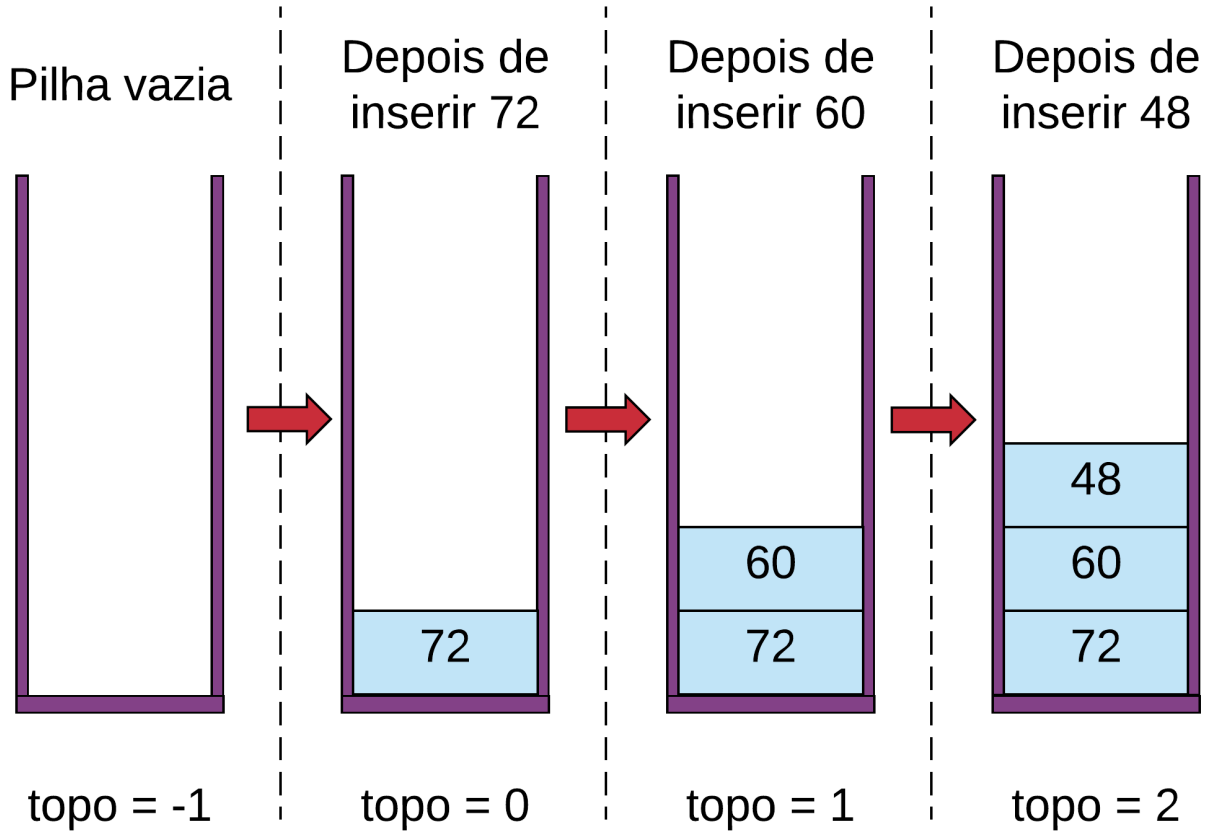
Pilha – representação utilizando *arrays* :

```
//iniciando uma pilha

int main() {

        //...

        struct pilha *p;

        p->topo = -1;

}
```



Fundamentos de estruturas de dados

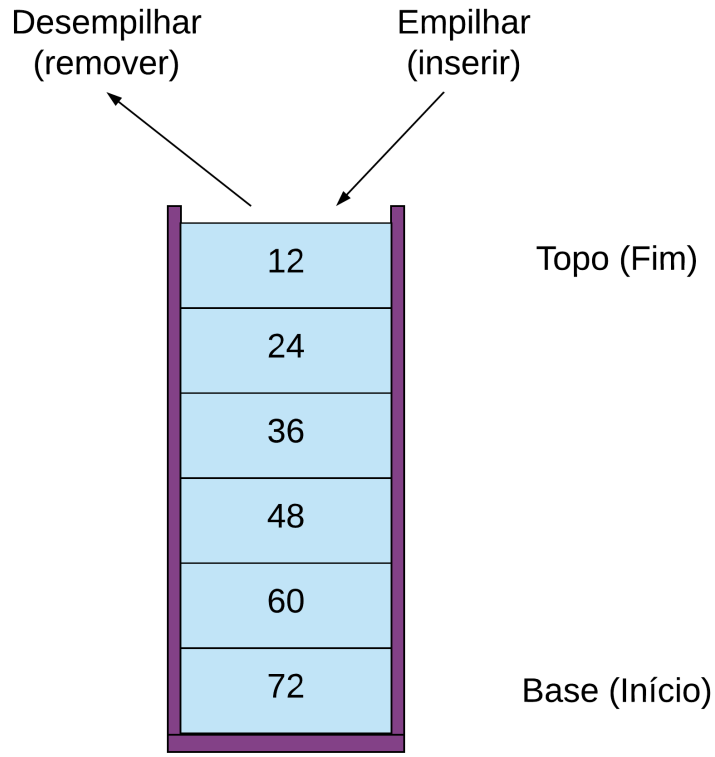
Pilha – representação utilizando *arrays* :



Fundamentos de estruturas de dados

Pilha:

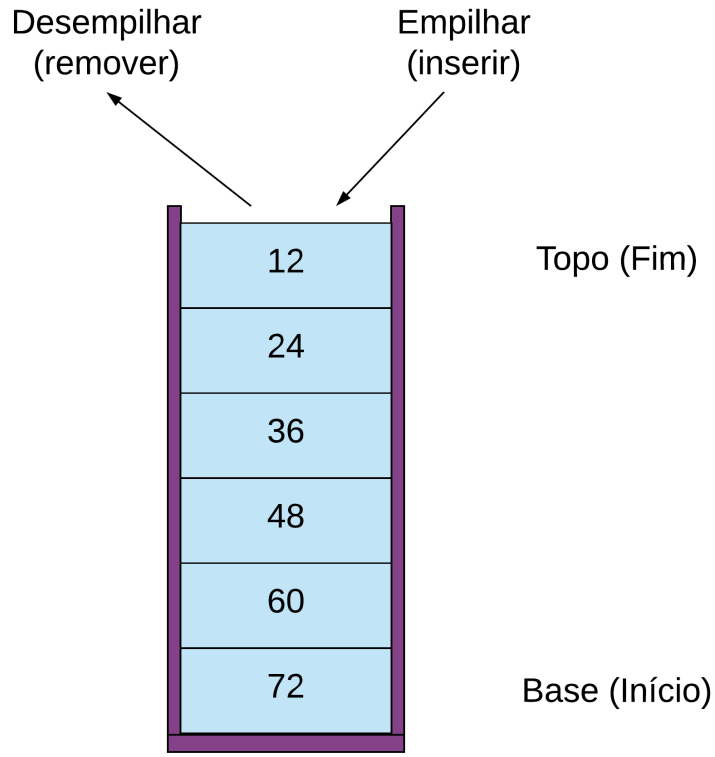
- É uma estrutura de dados; logo, possui um *tipo* para os dados que armazena, assim como *operações* definidas



Fundamentos de estruturas de dados

Operações em uma pilha **p**:

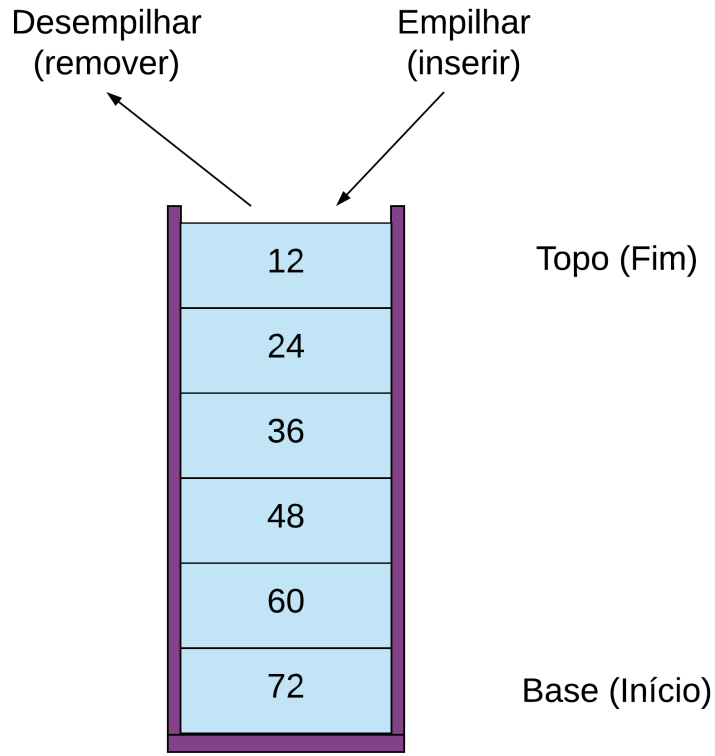
- Topo(**p**)
- EsvaziaPilha(**p**)
- PilhaVazia(**p**)
- PilhaCheia(**p**)
- Empilha(**d,p**)
- Desempilha(**d,p**)



Fundamentos de estruturas de dados

Topo(**p**): retorna a posição do topo da pilha

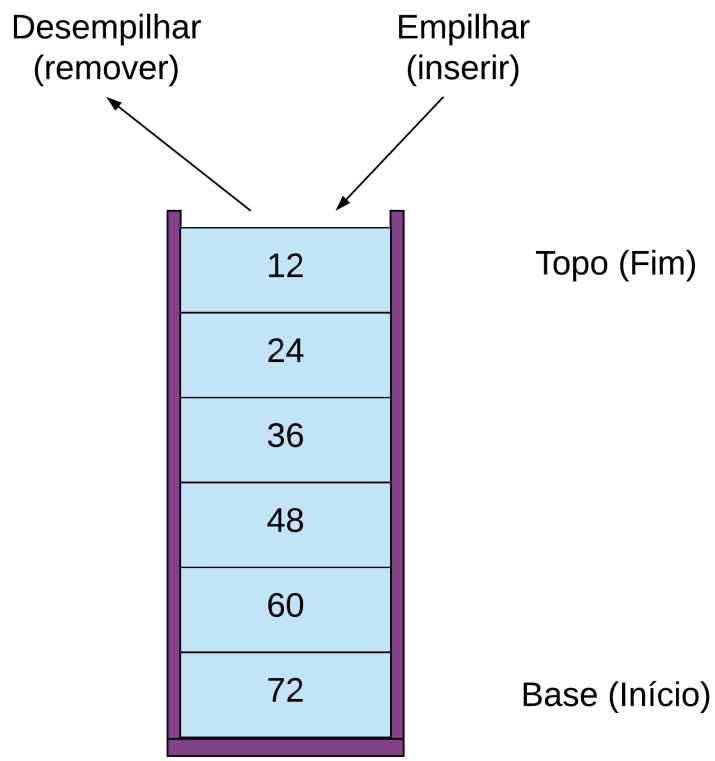
```
int topo(struct pilha *p) {
        return p->topo;
}
```



Fundamentos de estruturas de dados

EsvaziaPilha(**p**): remove todos os elementos da pilha

```
int esvaziaPilha(struct pilha *p) {
        while (p->topo > -1) {
                p->item[p->topo] = NULL;
                p->topo = p->topo – 1;
        }
        return 0;
}
```

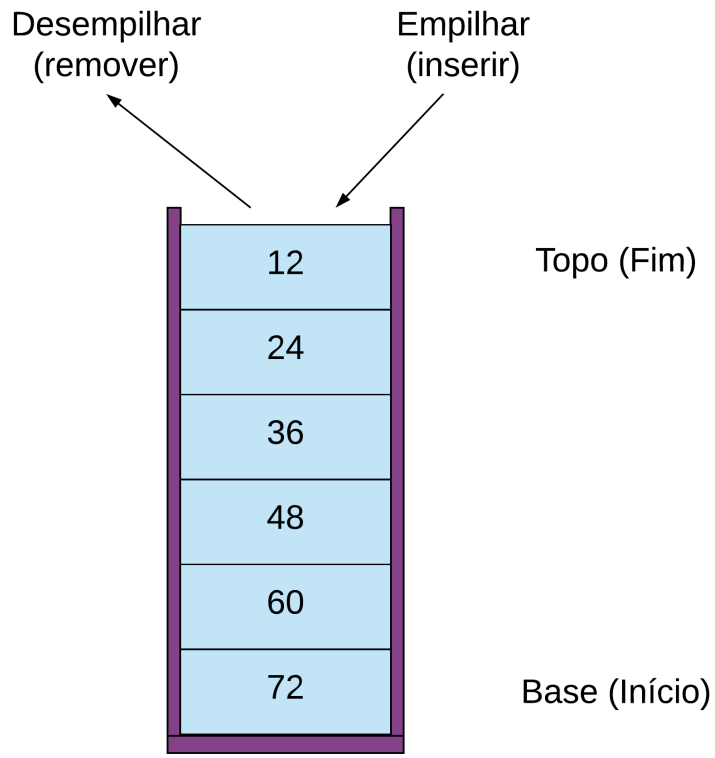


Fundamentos de estruturas de dados

EsvaziaPilha(**p**): remove todos os elementos da pilha

```
while (p->topo > -1)
```

- Enquanto houver elementos na pilha



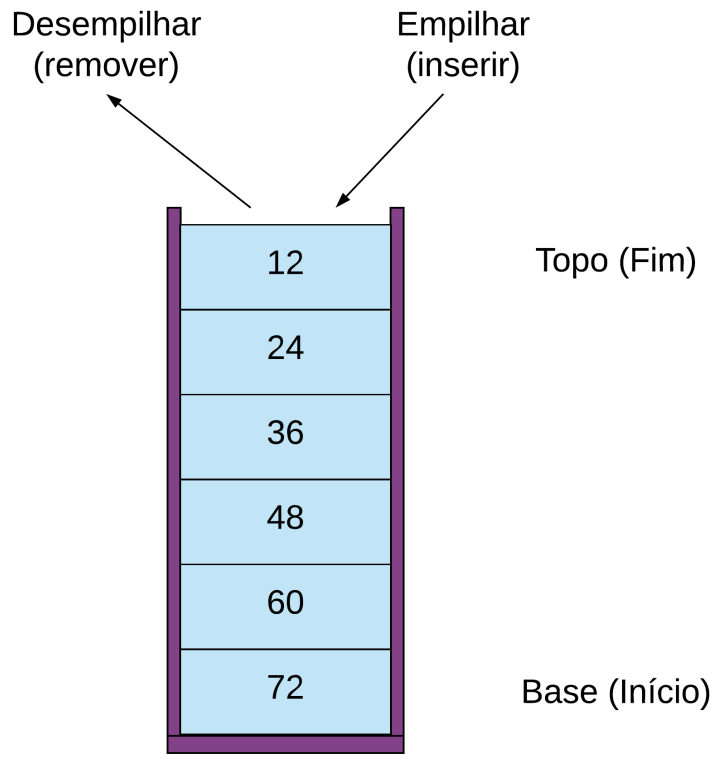
Fundamentos de estruturas de dados

EsvaziaPilha(**p**): remove todos os elementos da pilha

p->item[p->topo] = NULL;

- O elemento na posição do topo recebe valor nulo

- Posição do topo: p->topo

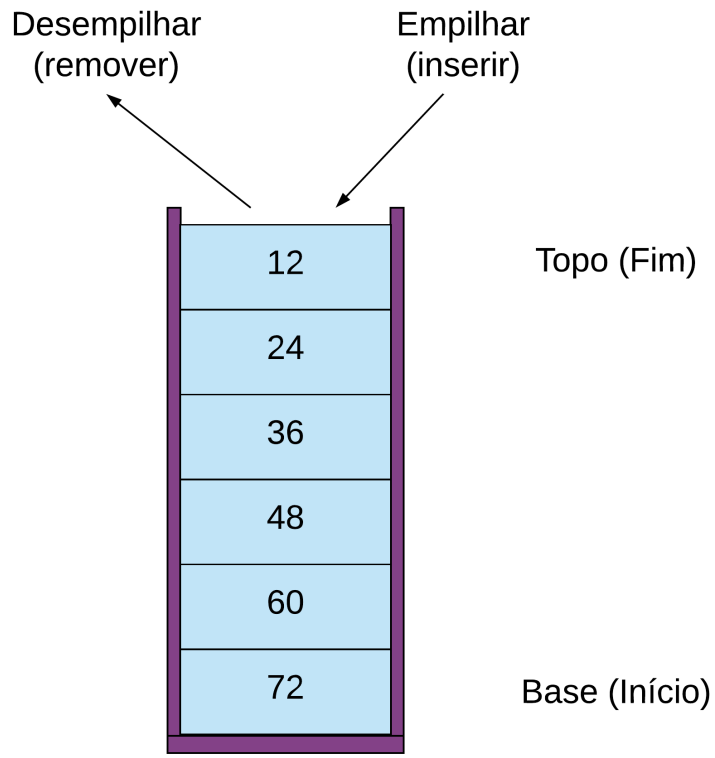


Fundamentos de estruturas de dados

EsvaziaPilha(**p**): remove todos os elementos da pilha

```
p->topo = p->topo – 1;
```

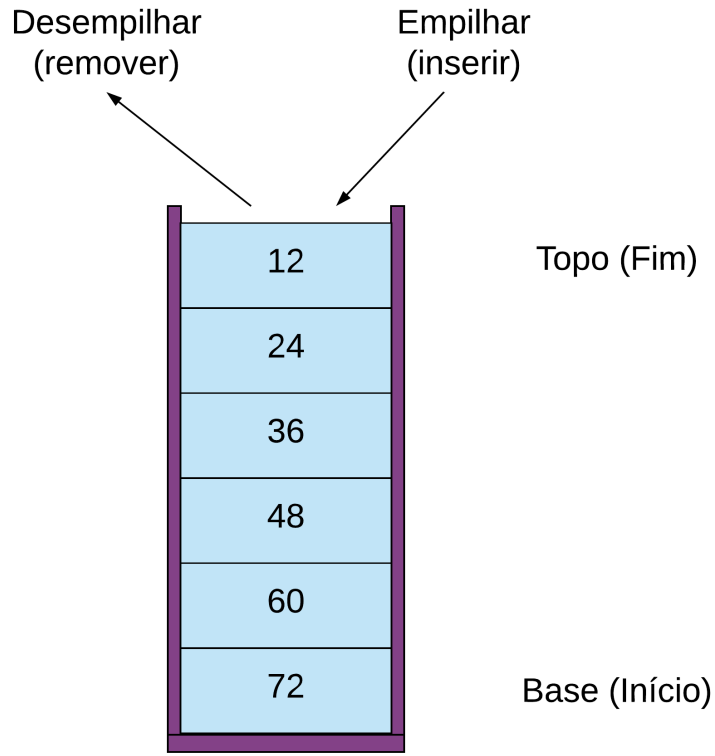
- Depois de removido o elemento do topo, a pilha diminui de tamanho (posição *topo* decresce)



Fundamentos de estruturas de dados

PilhaVazia(**p**): verifica se a pilha **p** está vazia

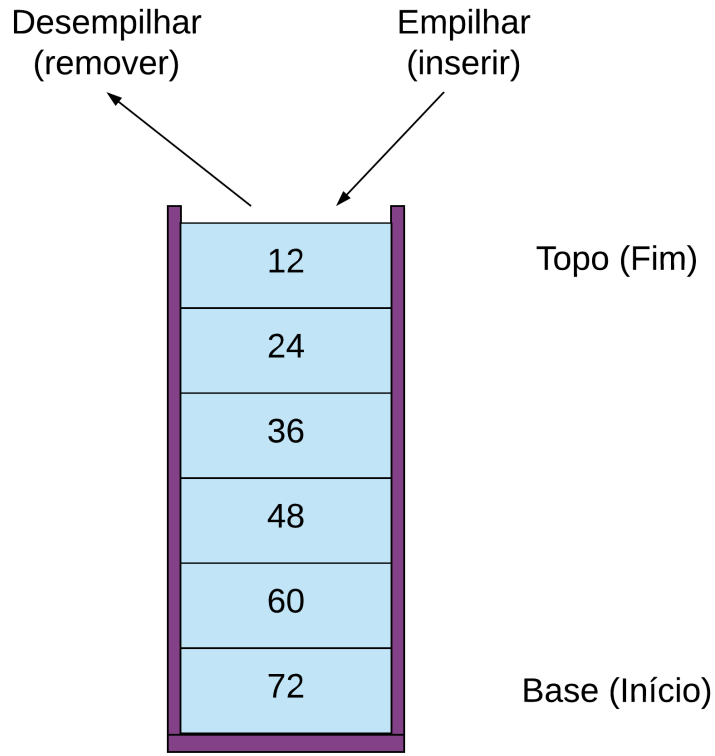
```
_Bool pilhaVazia(struct pilha *p) {
        if(p->topo == -1)
                return true;
        else
                return false;
}
```



Fundamentos de estruturas de dados

PilhaCheia(**p**): verifica se a pilha **p** está cheia

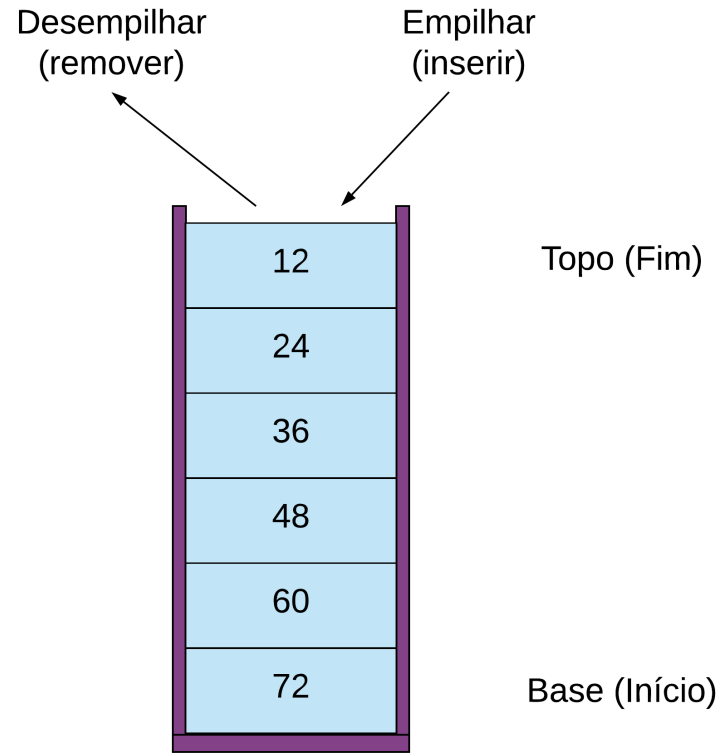
```
_Bool pilhaCheia(struct pilha *p) {
        if(p->topo+1 == MAXPILHA)
                return true;
        else
                return false;
}
```



Fundamentos de estruturas de dados

Empilha(**d,p**): empilha (insere) o elemento **d** na pilha **p**

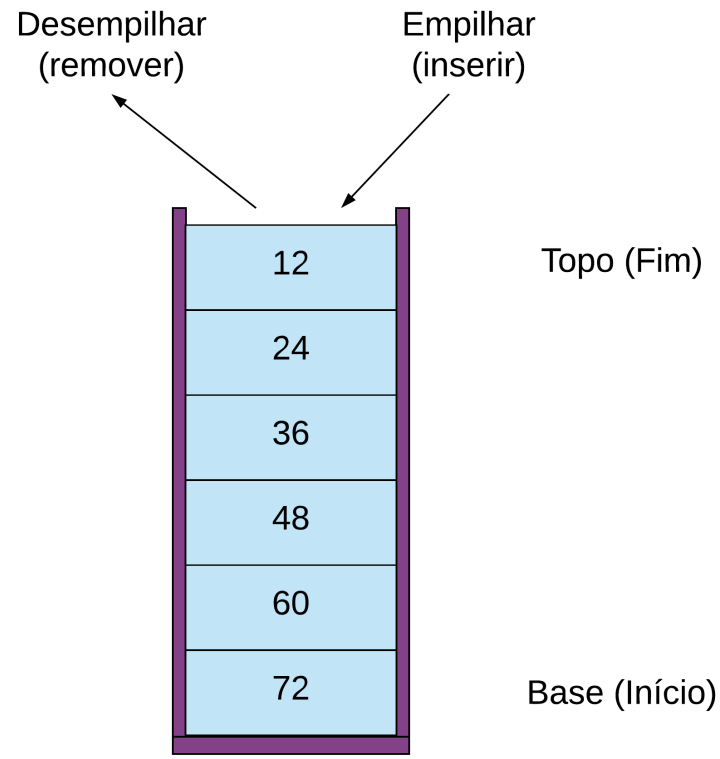
```
int empilha(int d, struct pilha *p) {
        if(pilhaCheia(p) == false) {
                p->item[p->topo+1] = d;
                p->topo = p->topo + 1;
}
```



Fundamentos de estruturas de dados

Desempilha(**d,p**): desempilha (remove) o elemento **d** da pilha **p**

- O elemento **d** existe na pilha?
- E se o elemento **d** estiver na pilha, mas não estiver no topo?



Fundamentos de estruturas de dados

Primeiro exercício para 18/01:

- Implementar um algoritmo para desempilhar um elemento qualquer da pilha **p**
  - Criar uma pilha
  - Preencher com os elementos: 12, 24, 36, 48, 60, 72
  - O algoritmo será testado para elementos que podem ou não existir na pilha
  - Caso o elemento a ser desempilhado não exista na pilha, o algoritmo deve retornar a mensagem "erro" (tudo em minúsculo)
  - Implementação pode ser em C ou JAVA

Fundamentos de estruturas de dados

Utilizando pilhas para tratar expressões matemáticas

- Expressões válidas:

A + 2*B
(A + 2)*B

- Expressões inválidas:

A + B)

Fundamentos de estruturas de dados

Utilizando pilhas para tratar expressões matemáticas

- Verificando parênteses:
  - Número de parênteses abertos igual ao número de parênteses fechados;
  - Antes de um parêntese de fechamento, deve haver um parêntese de abertura

Fundamentos de estruturas de dados

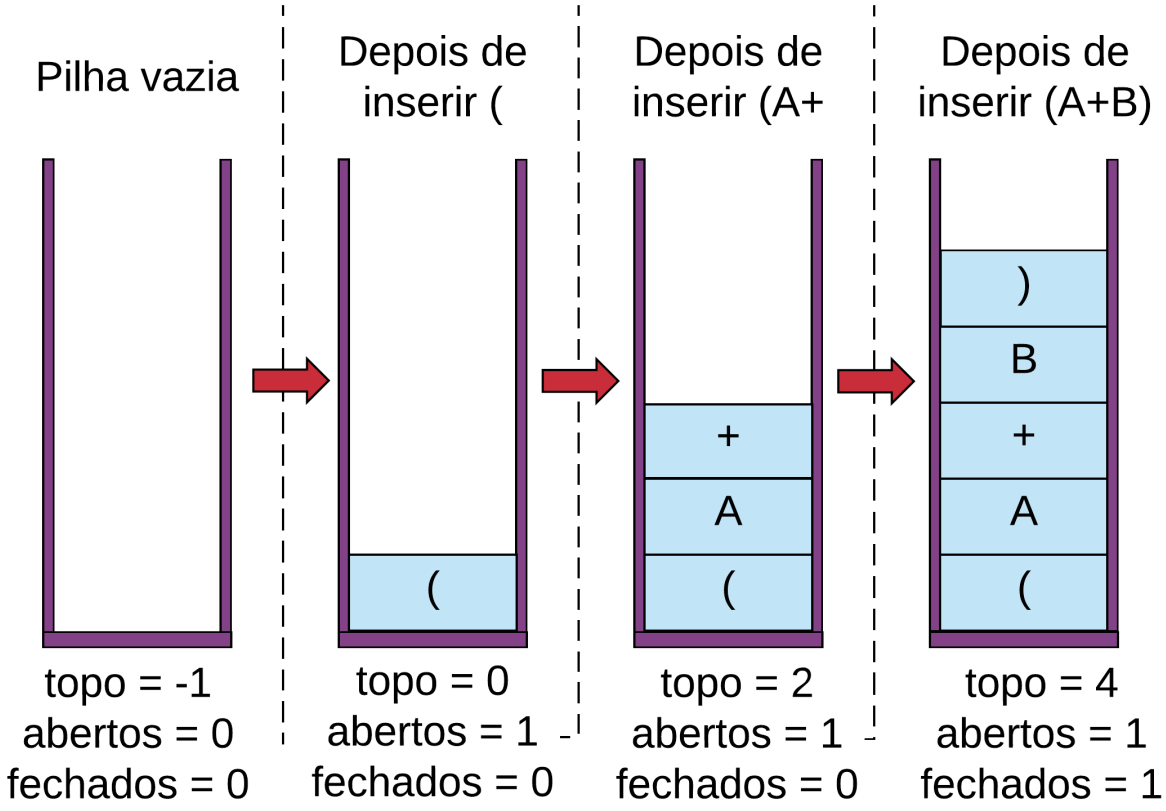
Utilizando pilhas para tratar expressões matemáticas

› Verificando parênteses:

- Número de parênteses abertos igual ao número de parênteses fechados: no final da expressão, **abertos – fechados = 0**
- Antes de um parêntese de fechamento, deve haver um parêntese de abertura: em qualquer momento, **abertos >= fechados**

Fundamentos de estruturas de dados

Empilhando a expressão: (A + B)



Fundamentos de estruturas de dados

Utilizando pilhas para tratar expressões matemáticas

- A expressão pode ser validada antes da expressão ser processada
- Expressões parciais podem ser tratadas sequencialmente

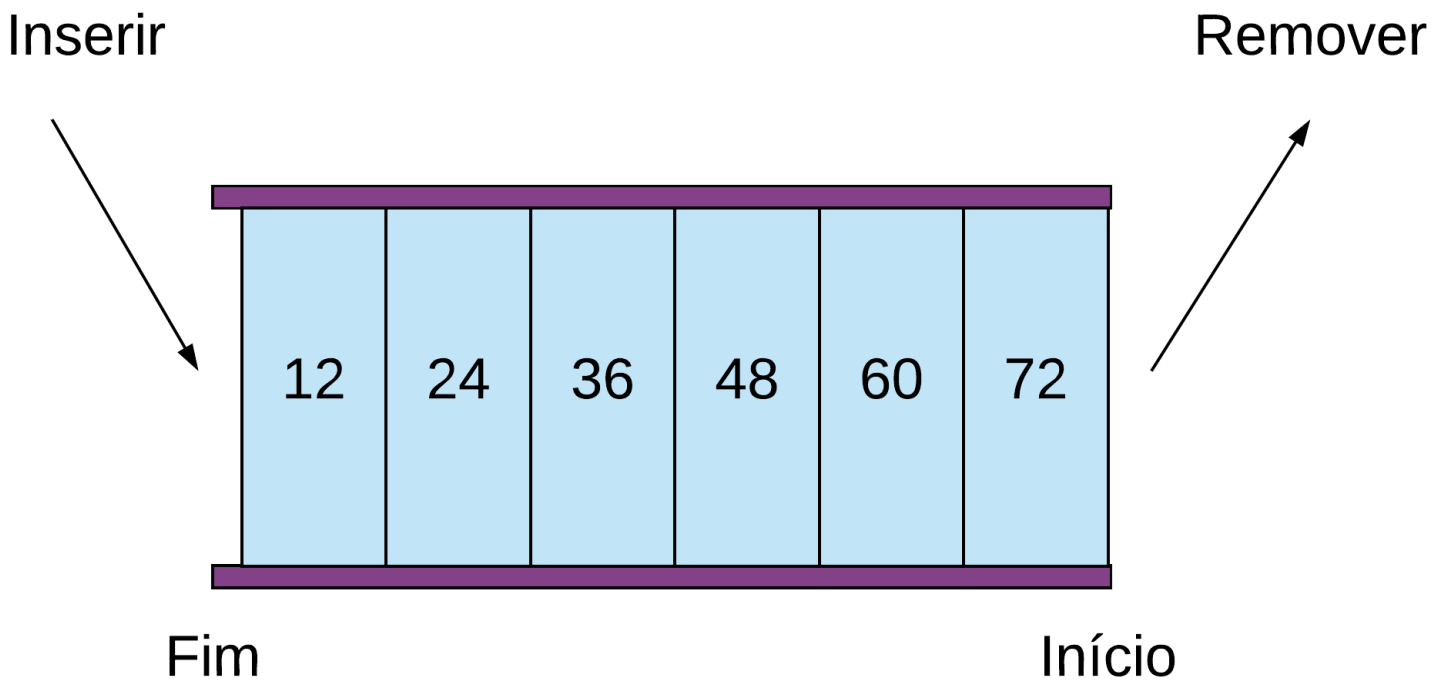
Fundamentos de estruturas de dados

Fila:

› "Pode ser vista como uma **lista linear** na qual **todas as inserções** são realizados **somente** em uma das extremidades (**fim**), enquanto **todas as remoções e acessos** são realizados na outra extremidade (**início**)."

Fundamentos de estruturas de dados

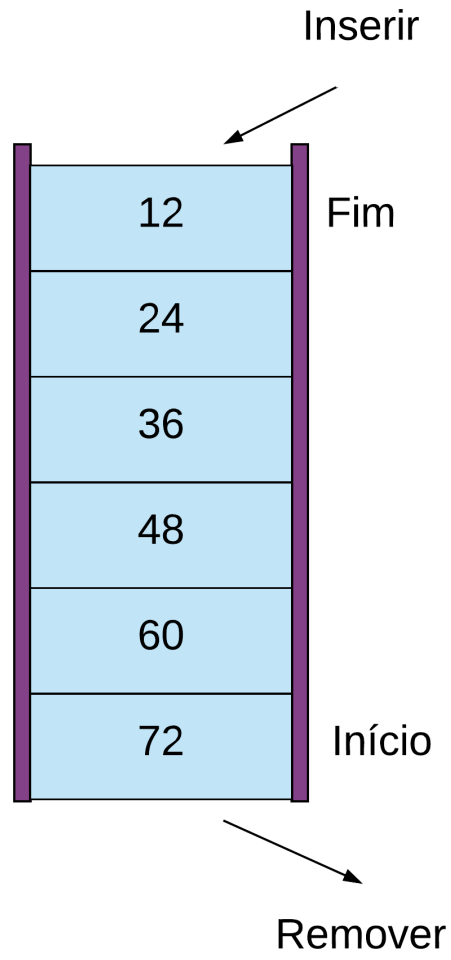
Fila:



Fundamentos de estruturas de dados

Fila:

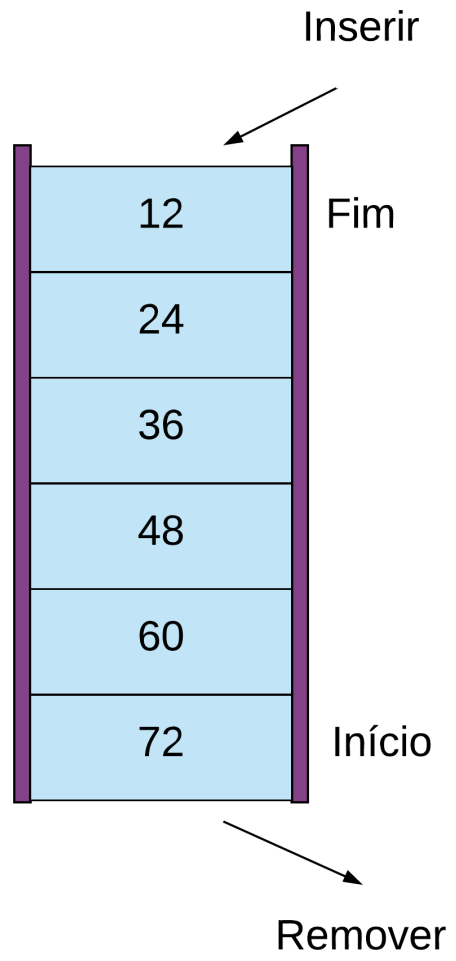
- O primeiro elemento a ser inserido também é o primeiro a ser removido: **FIFO** (*First In, First Out*)
- Também pode ser representada utilizando-se uma lista linear (*array*)



Fundamentos de estruturas de dados

Fila – representação utilizando *arrays* :

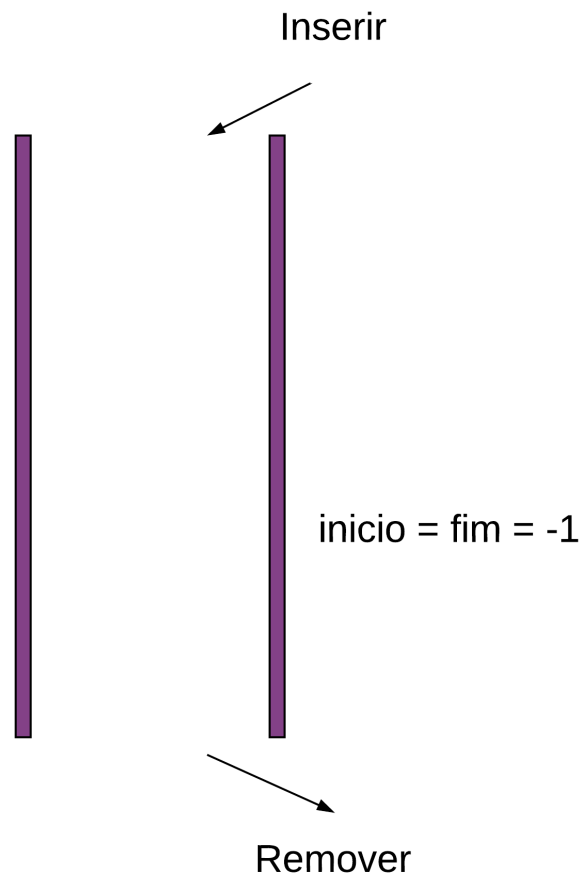
```
#define MAXFILA 8

struct fila {

        int inicio, fim;

        int item[MAXFILA];

}
```



Fundamentos de estruturas de dados

Fila – representação utilizando *arrays* :

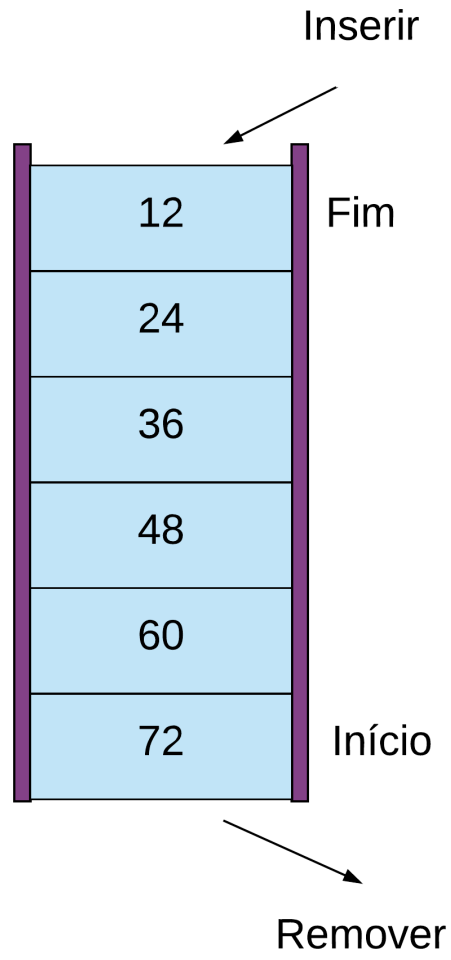
```
//iniciando uma fila

int main() {

        //...

        struct fila *f;

        f->inicio = -1;

        f->fim = -1;

}
```



Fundamentos de estruturas de dados

Fila:

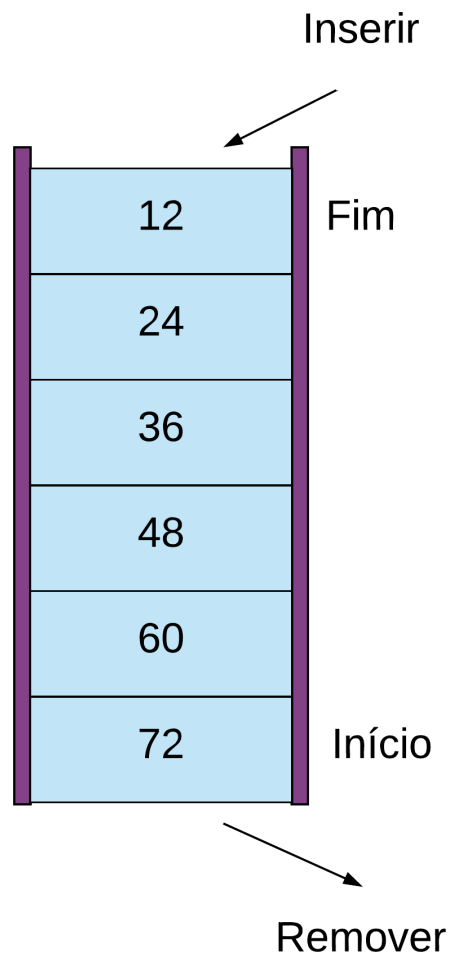
- É uma estrutura de dados; logo, possui um *tipo* para os dados que armazena, assim como *operações* definidas



Fundamentos de estruturas de dados

Operações em uma fila **f**:

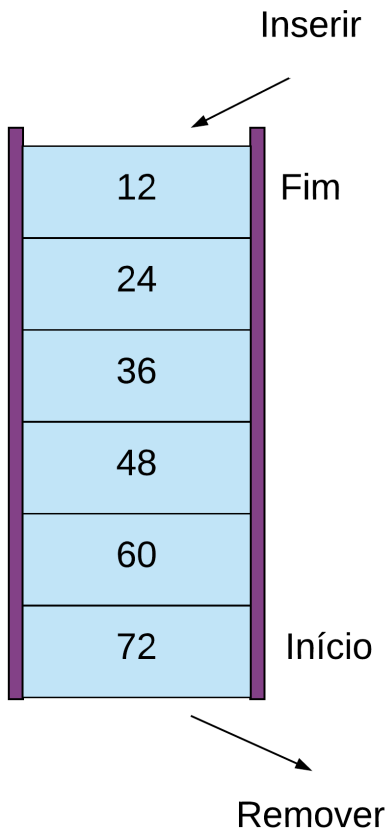
- EsvaziaFila(**f**)
- FilaVazia(**f**)
- FilaCheia(**f**)
- Insere(**d,f**)
- Remove(**d,f**)



Fundamentos de estruturas de dados

Insere(**d,f**): insere um elemento **d** na fila **f**

```
int insere(int d, struct fila *f) {
        if(f->inicio == -1) {
                f->inicio = 0; f->fim = 0;
        }
        else {
                if((f->inicio+1) < MAXFILA)
                        f->inicio = f->inicio + 1;
                while... //desloca elementos em direção ao inicio
        }
        f->item[f->fim] = d;
}
```

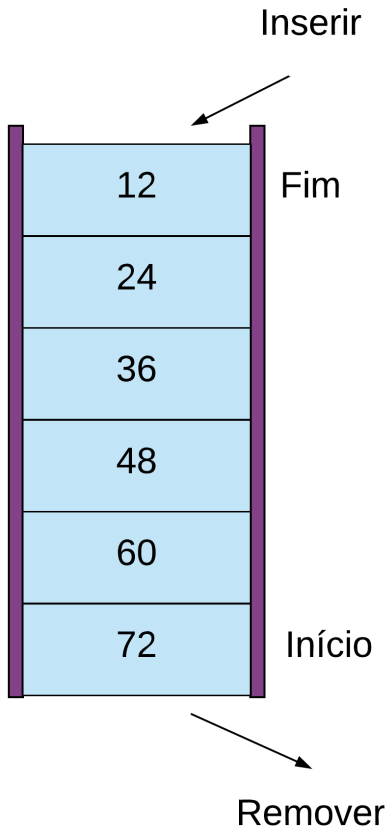


Fundamentos de estruturas de dados

Insere(**d**,**f**): insere um elemento **d** na fila **f**

```
        if(f->inicio == -1) {
                f->inicio = 0; f->fim = 0;
        }
```

- Se a fila estiver vazia, deve ser preparada para receber o primeiro elemento

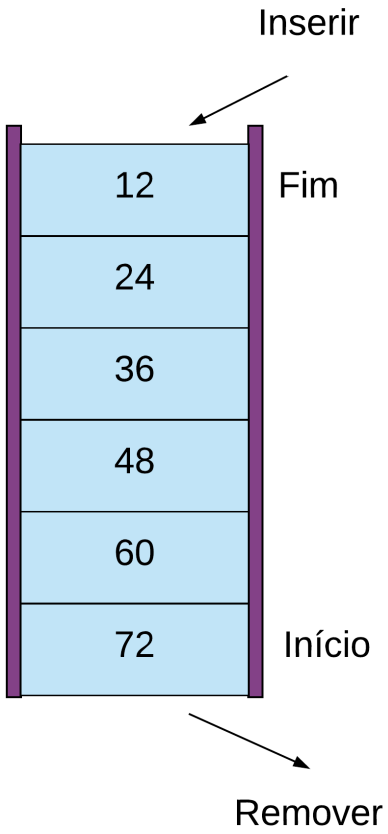


Fundamentos de estruturas de dados

Insere(**d**,**f**): insere um elemento **d** na fila **f**

```
        else {
                if((f->inicio+1) < MAXFILA)
                        f->inicio = f->inicio + 1;
                while... //desloca elementos
        }
```

- Caso contrário, a fila é deslocada em direção ao início para poder receber o novo elemento

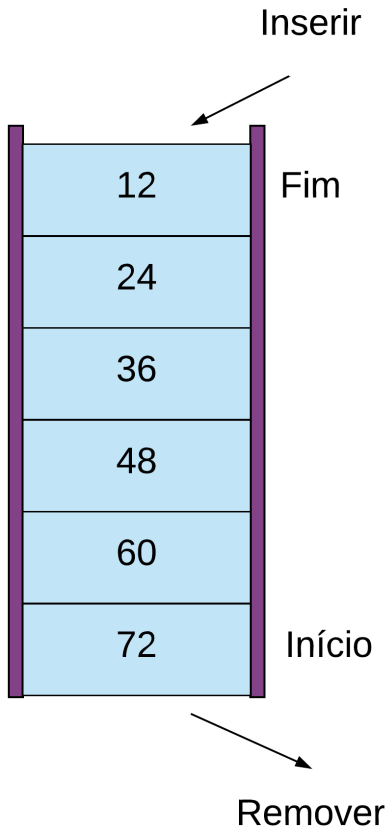


Fundamentos de estruturas de dados

Insere(**d**,**f**): insere um elemento **d** na fila **f**

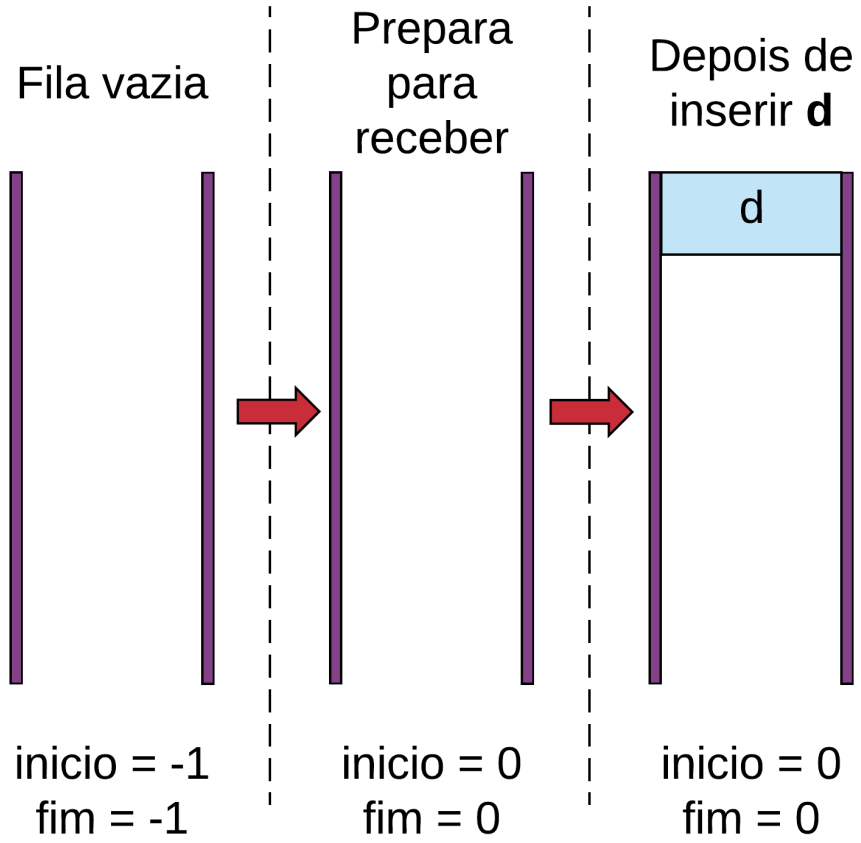
```
f->item[f->fim] = d;
```

- Novo elemento é inserido no fim da fila



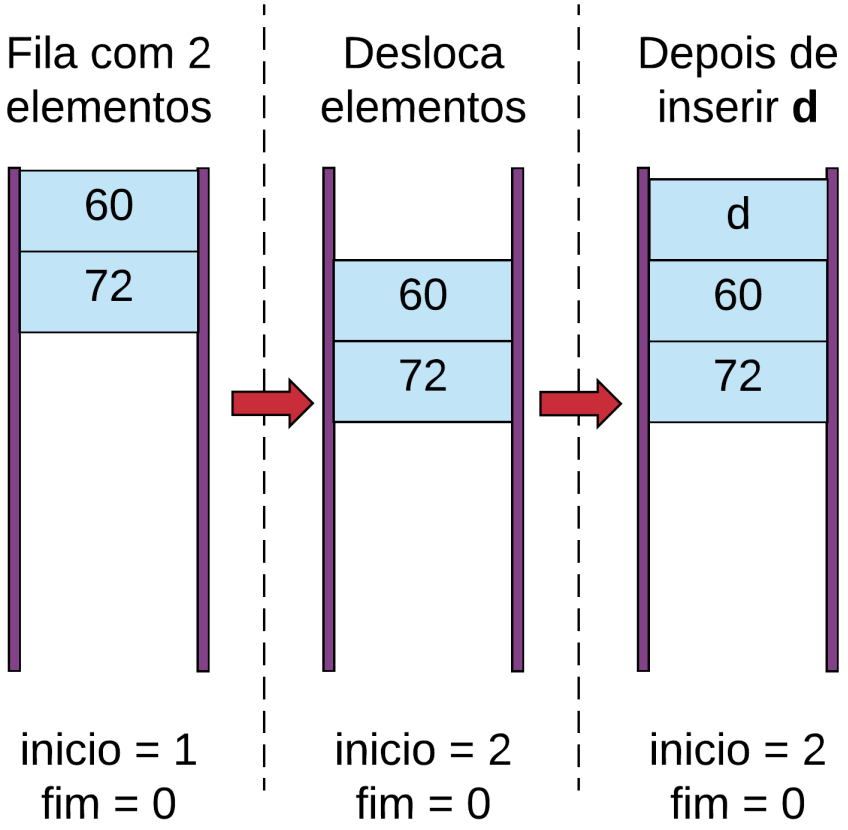
Fundamentos de estruturas de dados

Insere(**d**,**f**): insere um elemento **d** na fila **f** ***vazia***



Fundamentos de estruturas de dados

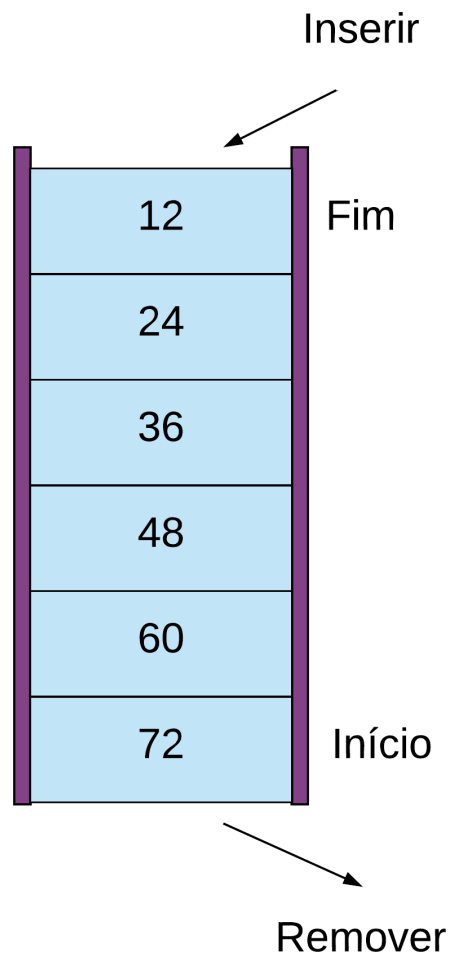
Insere(**d**,**f**): insere um elemento **d** na fila **f** ***não vazia***



Fundamentos de estruturas de dados

Operações em uma fila **f**:

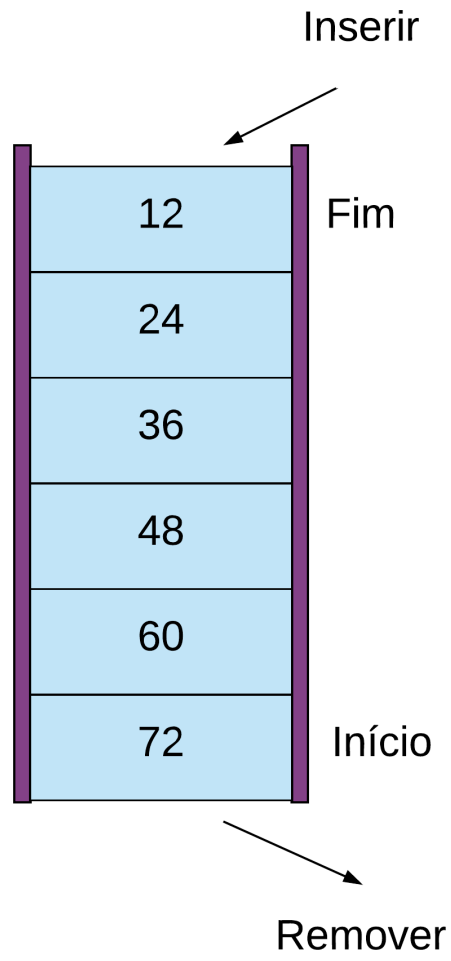
- EsvaziaFila(**f**)
- FilaVazia(**f**)
- FilaCheia(**f**)
- Insere(**d**,**f**)
- Remove(**d**,**f**)



Fundamentos de estruturas de dados

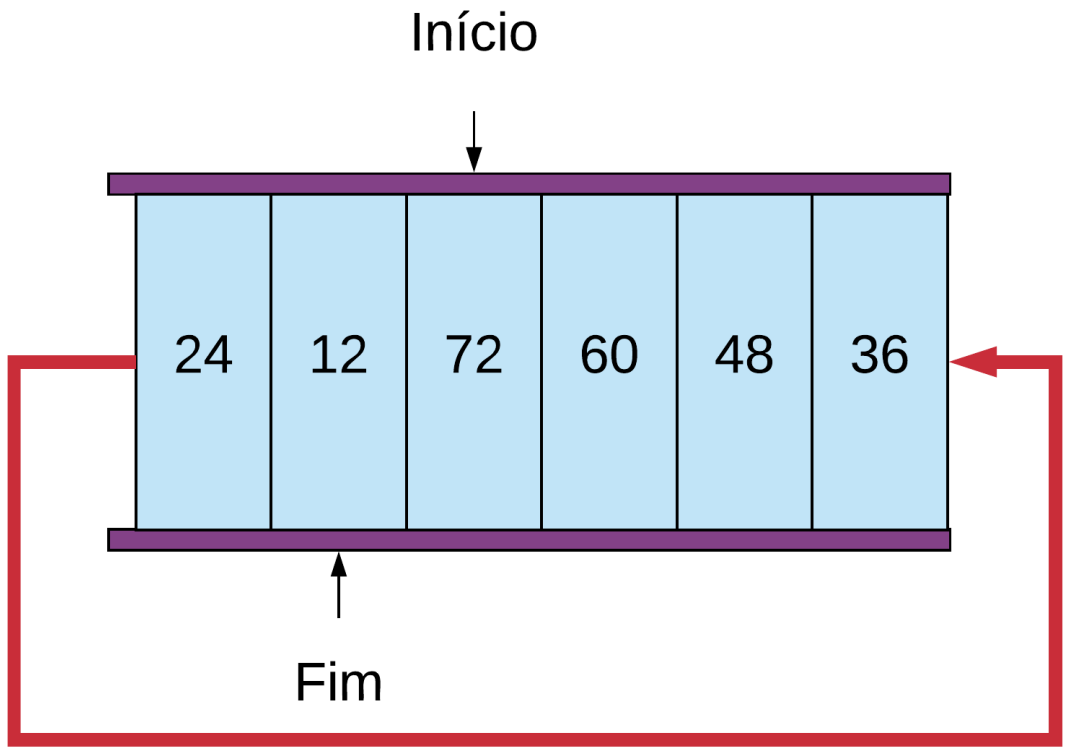
Operações em uma fila **f**:

- Complexidade de inserção ou remoção depende de como a fila é inicializada



Fundamentos de estruturas de dados

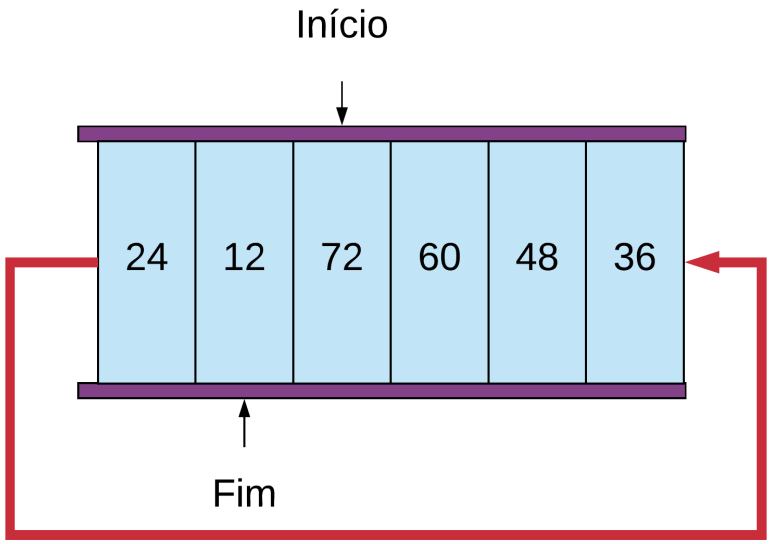
Filas circulares:



Fundamentos de estruturas de dados

Filas circulares:

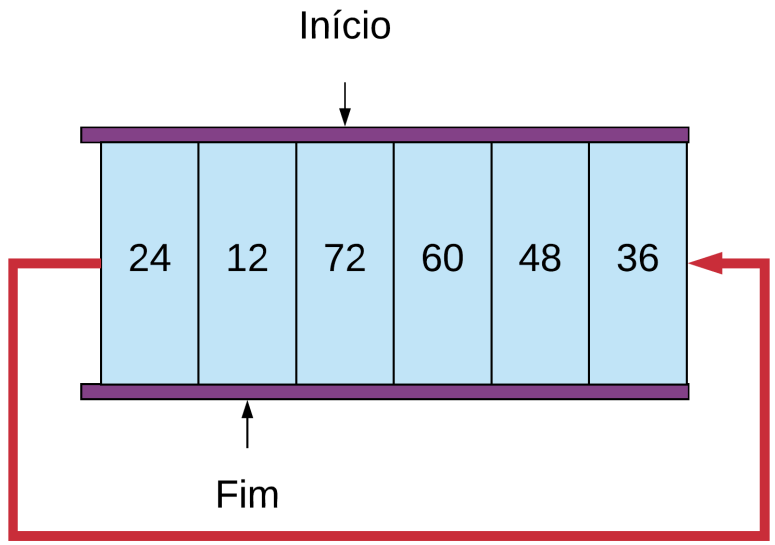
- Ideia: armazenar os elementos na fila como se esta fosse um círculo
- Mitiga problemas de complexidade nas operações de inserção e remoção de elementos
- *Primeiro elemento da fila vem logo depois do último*



Fundamentos de estruturas de dados

Fila circular – representação utilizando *arrays* :

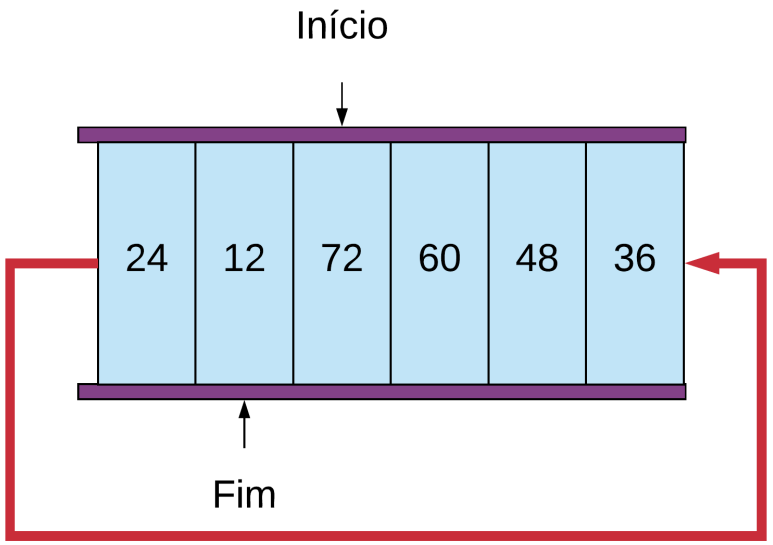
```
#define MAXFILA 8

struct fila {

        int inicio, fim;

        int item[MAXFILA];

}
```



Fundamentos de estruturas de dados

Fila circular – representação utilizando *arrays* :

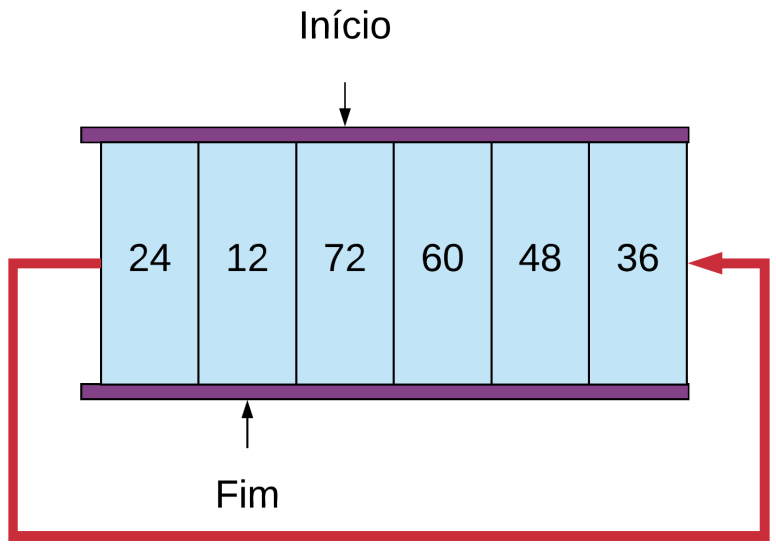
```
//iniciando uma fila circular

int main() {

        //...

        struct fila *f;

        f->inicio = MAXFILA-1;

        f->fim = MAXFILA-1;

}
```



Fundamentos de estruturas de dados

Operações em uma fila circular **f**:

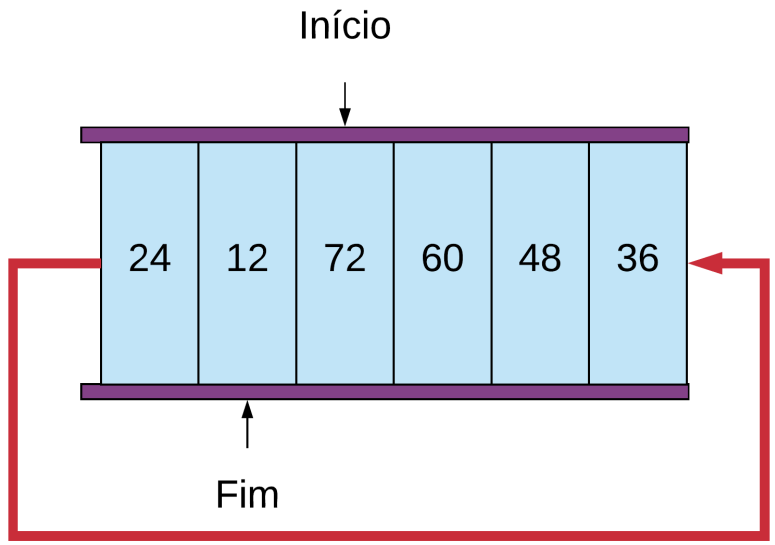
- FilaVazia(**f**)
- FilaCheia(**f**)
- EsvaziaFila(**f**)
- Insere(**d,f**)
- Remove(**d,f**)



Fundamentos de estruturas de dados

FilaVazia(**f**): verifica se a fila circular **f** est vazia

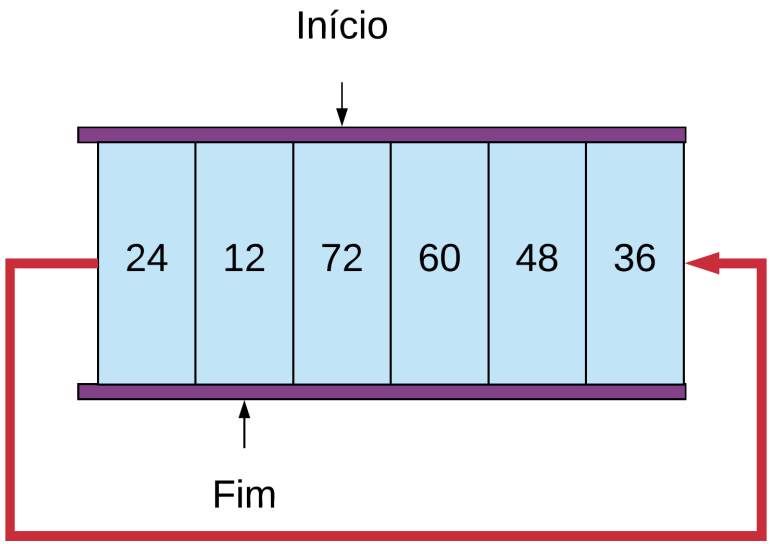
```
int filaVazia(struct fila *f) {
        if(f->inicio == f->fim)
                return 1; //verdadeiro
        else
                return 0;
}
```



Fundamentos de estruturas de dados

FilaCheia(**f**): verifica se a fila circular **f** está cheia

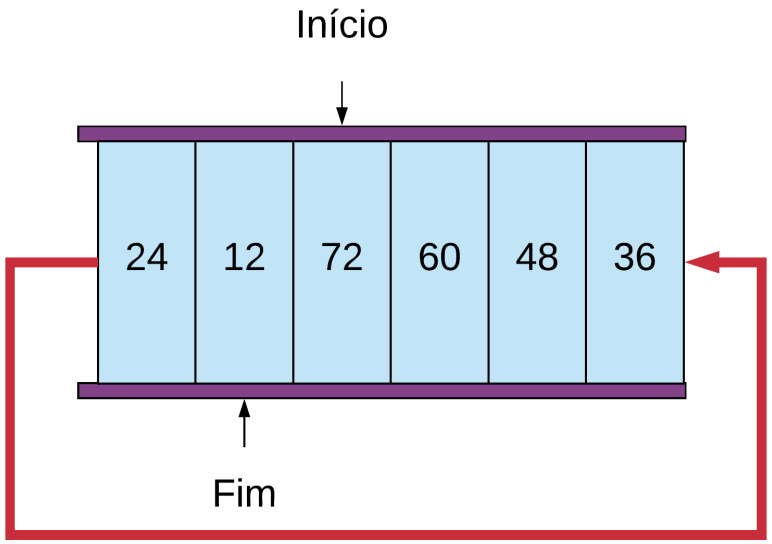
```
int filaVazia(struct fila *f) {
        if(f->inicio == f->fim + 1)
                return 1;
        else
                return 0;
}
```



Fundamentos de estruturas de dados

Insere(**d**,**f**): insere um elemento **d** na fila circular **f**

```
void insere(int d, struct fila *f) {
        if(filaCheia(f) != 0) {
                if(f->fim==MAXFILA-1)
                        f->fim = 0;
                else
                        f->fim = f->fim + 1;
                f->item[f->fim]  = d;
        }
}
```

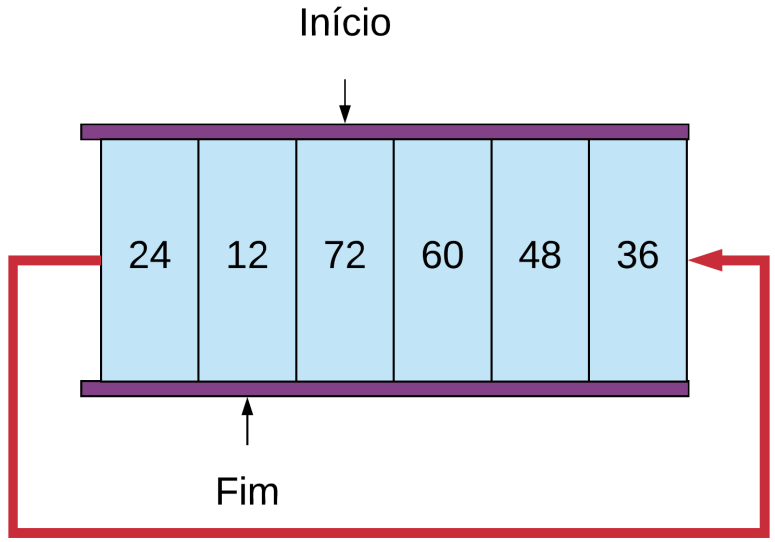


Fundamentos de estruturas de dados

Remove(**d,f**): remove um elemento **d** de uma fila circular **f**

- O elemento **d** existe na fila?

- Se existir **e não estiver no início**, o que acontece com o resto da fila?



Fundamentos de estruturas de dados

Segundo exercício para 18/01 (valendo 2 pontos):

- Implementar um algoritmo para remover um elemento conhecido de uma fila circular **f**
  - Criar uma fila circular
  - Preencher com os elementos: 12, 24, 36, 48, 60, 72, de forma que o **início** da fila (primeiro elemento que entrou na fila) seja o número 72
  - O algoritmo deverá remover o elemento passado como parâmetro SOMENTE se este elemento estiver no início da fila
    - Verificar se o elemento passado como parâmetro é o elemento que se encontra no início da fila
    - Em caso afirmativo, remover o elemento
    - Fazer as atualizações necessárias em INICIO e FIM
  - Implementação pode ser em C ou JAVA