MC-202 — Aula 4
Pilhas, Filas e Aplicações

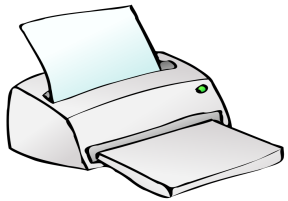
Lehilton Pedrosa

Instituto de Computação – Unicamp

Segundo Semestre de 2015

Roteiro

Introdução



Situação 1

- Uma impressora é compartilhada por alunos em um laboratório

Introdução

Situação 1

- Uma impressora é compartilhada por alunos em um laboratório
- Vários alunos enviam documentos *quase* ao mesmo tempo

Introdução

Situação 1

- Uma impressora é compartilhada por alunos em um laboratório
- Vários alunos enviam documentos *quase* ao mesmo tempo

Problema: Como gerenciar (de forma justa) a lista de tarefas de impressão?

Introdução

Situação 2

- Numa academia as anilhas são guardadas umas sobre as outras
- Você deseja obter uma anilha de um peso específico

Introdução

Situação 2

- Numa academia as anilhas são guardadas umas sobre as outras
- Você deseja obter uma anilha de um peso específico

Pergunta: Como pegar a anilha correta? Que dificuldades há?

Coleções dinâmicas

Coleções dinâmicas

São coleções variáveis de objetos guardados em uma estrutura de dados com operações inserir e remover um objeto (documento, anilha etc.).

Coleções dinâmicas

Coleções dinâmicas

São coleções variáveis de objetos guardados em uma estrutura de dados com operações inserir e remover um objeto (documento, anilha etc.).

Dois tipos fundamentais

1. **Filas**
   - Remove primeiro objetos inseridos há mais tempo
2. **Pilhas**
   - Remove primeiro objetos inseridos mais recentemente

Coleções dinâmicas

Coleções dinâmicas

São coleções variáveis de objetos guardados em uma estrutura de dados com operações inserir e remover um objeto (documento, anilha etc.).

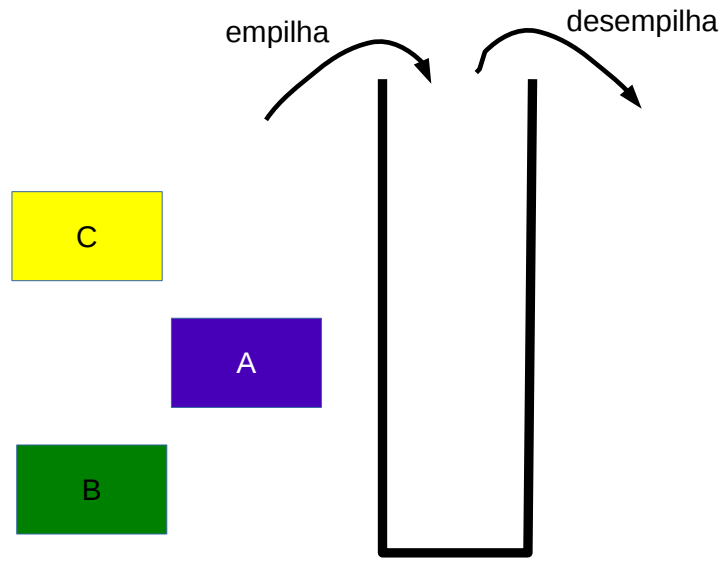
Dois tipos fundamentais

1. **Filas**
   - Remove primeiro objetos inseridos há mais tempo
2. **Pilhas**
   - Remove primeiro objetos inseridos mais recentemente

Observação: Alguns termos de contabilidade:

- **FIFO** para filas: primeiro a entrar é primeiro a sair (*first-in first-out*)
- **LIFO** para pilhas: último a entrar é primeiro a sair (*last-in first-out*)

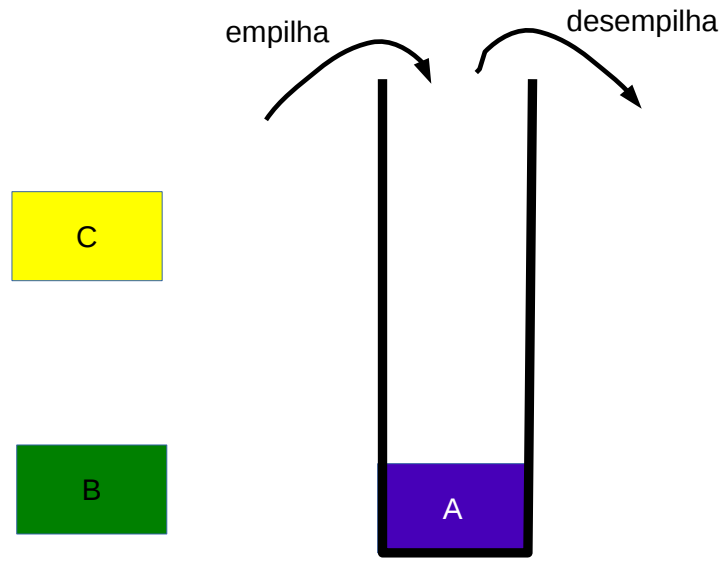
Pilha



Operações:

- **Empilha** (*push*): adiciona no topo

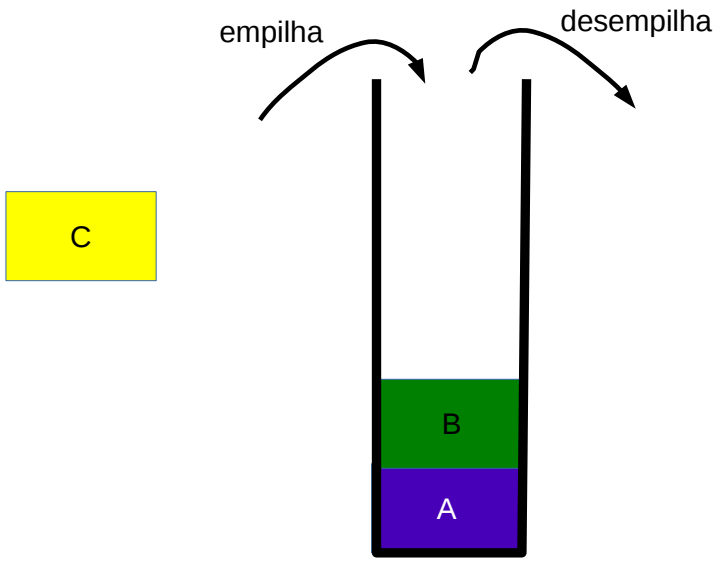
Pilha



Operações: Empilhou A

- **Empilha** (*push*): adiciona no topo

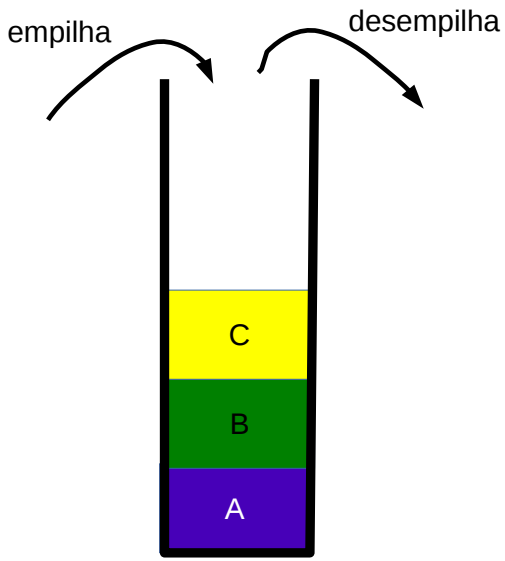
Pilha



Operações: Empilhou B

- **Empilha** (*push*): adiciona no topo

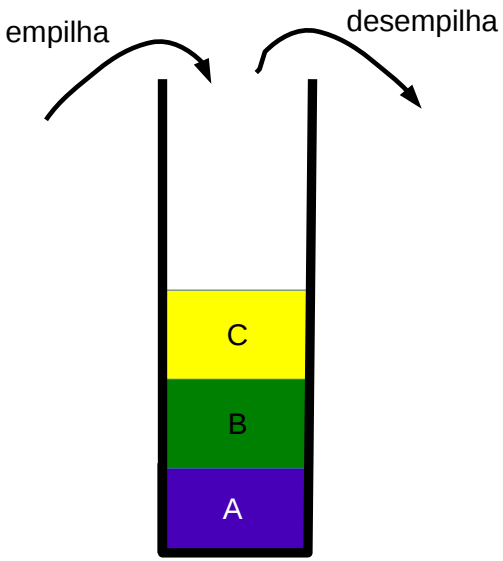
Pilha



Operações: Empilhou C

- **Empilha** (*push*): adiciona no topo

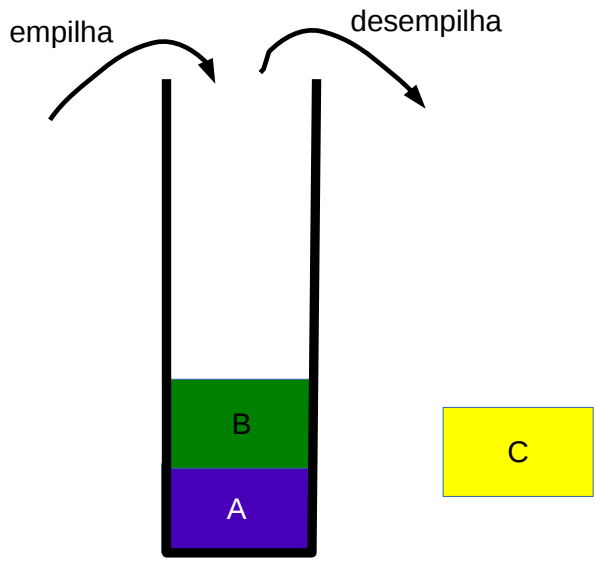
Pilha



Operações:

- **Empilha** (*push*): adiciona no topo
- **Desempilha** (*pop*): remove do topo

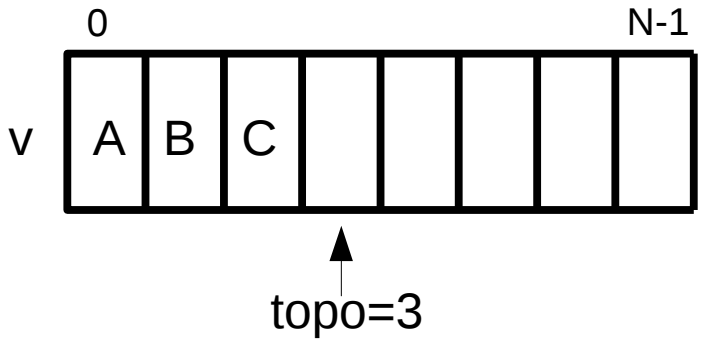
Pilha



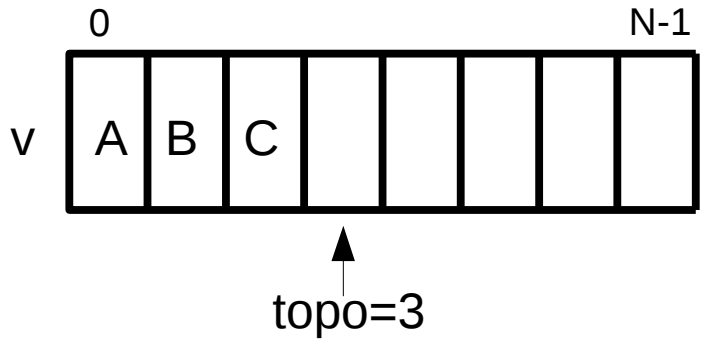
Operações: Desempilhou

- **Empilha** (*push*): adiciona no topo
- **Desempilha** (*pop*): remove do topo

Pilha: com vetor



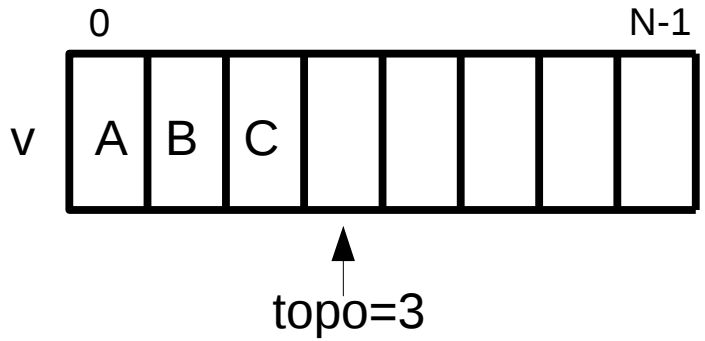
Pilha: com vetor



Definição

```
#define N 1000
typedef struct { Item v[N];  int topo;} Pilha;
```

Pilha: com vetor



Definição

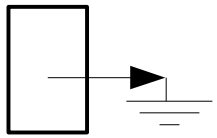
```
#define N 1000
typedef struct { Item v[N];  int topo;} Pilha;
```

```
void empilhar(Pilha *p, Item i) {
  p->v[p->topo] = i;
  (p->topo)++;
}
```

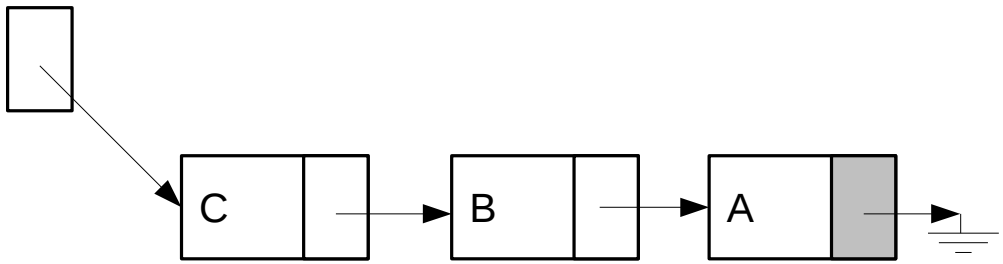
```
Item desempilhar(Pilha *p) {
  (p->topo)--;
  return p->v[p->topo];
}
```

Pilha: com lista ligada

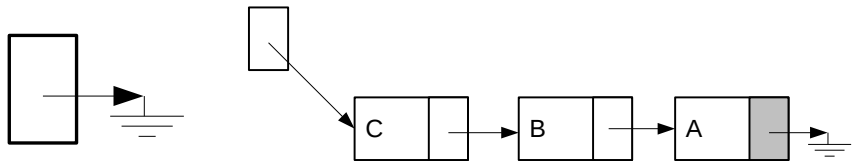
Pilha vazia:



Após inserir A, B, C:



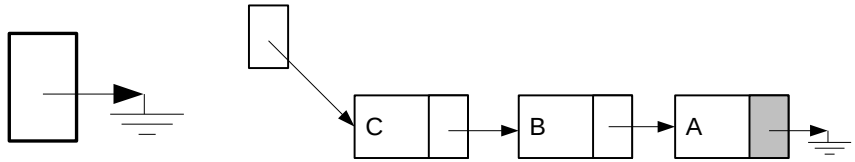
Pilha: com lista ligada



Definição

```
typedef struct NoPilha { Item dado;  struct NoPilha *prox; } NoPilha;
```

Pilha: com lista ligada



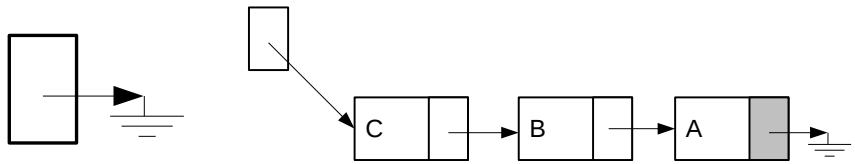
Definição

```
typedef struct NoPilha { Item dado;  struct NoPilha *prox; } NoPilha;
```

```
void empilhar(NoPilha **p, Item i) {
  NoPilha *q = malloc(sizeof(NoPilha));
  q->dado = i;
  q->prox = *p;
  *p = q;
}
```

```
Item desempilhar(NoPilha **p) {
  NoPilha *q = *p;
  Item i = q->dado;
  *p = q->prox;
  free(q);
  return i;
}
```

Pilha: com lista ligada



Definição

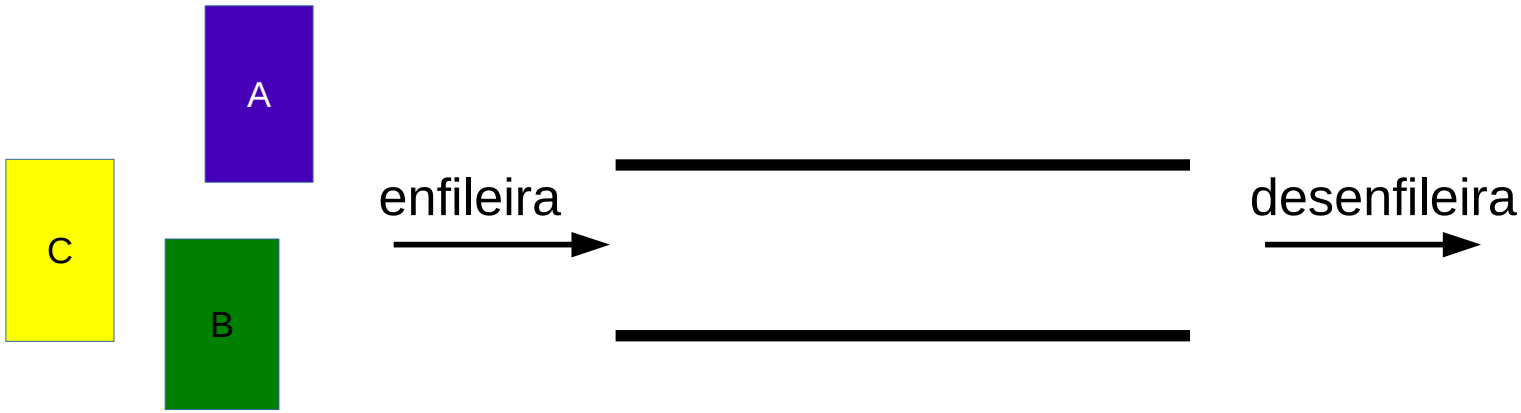
```
typedef struct NoPilha { Item dado;  struct NoPilha *prox; } NoPilha;
```

```
void empilhar(NoPilha **p, Item i) {
  NoPilha *q = malloc(sizeof(NoPilha));
  q->dado = i;
  q->prox = *p;
  *p = q;
}
```

```
Item desempilhar(NoPilha **p) {
  NoPilha *q = *p;
  Item i = q->dado;
  *p = q->prox;
  free(q);
  return i;
}
```

Exercício: corrija os casos de erro

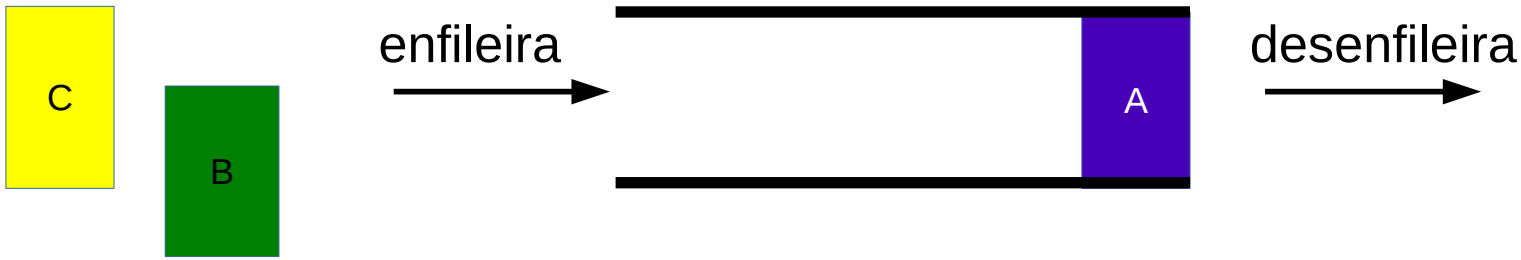
Fila



Operações:

- **Enfileira** (*queue*): adiciona item no "fim"

Fila



Operações: Enfileirou A

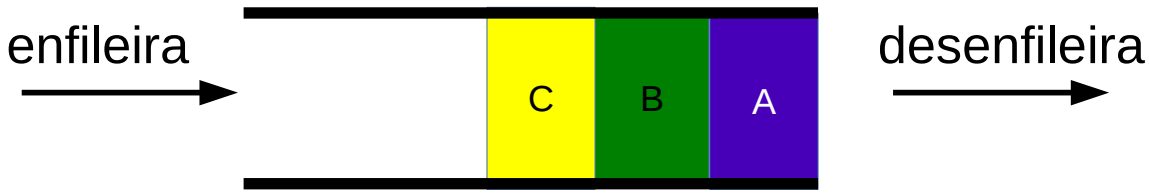
- **Enfileira** (*queue*): adiciona item no "fim"

Fila

Operações: Enfileirou B

- **Enfileira** (*queue*): adiciona item no "fim"

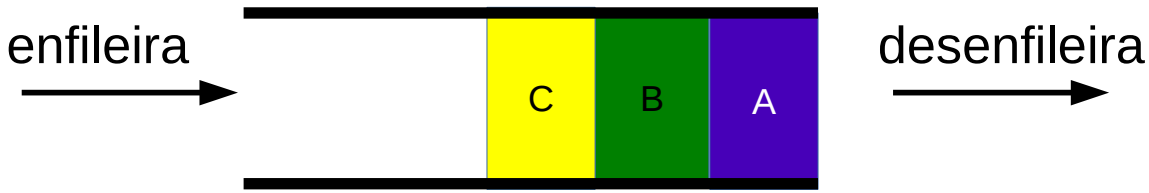
Fila



Operações: Enfileirou C

- **Enfileira** (*queue*): adiciona item no "fim"

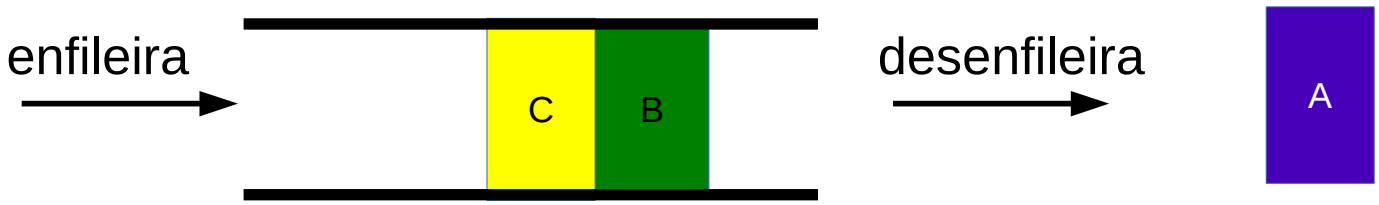
Fila



Operações:

- **Enfileira** (*queue*): adiciona item no "fim"
- **Desenfileira** (*dequeue*): remove item do "início"

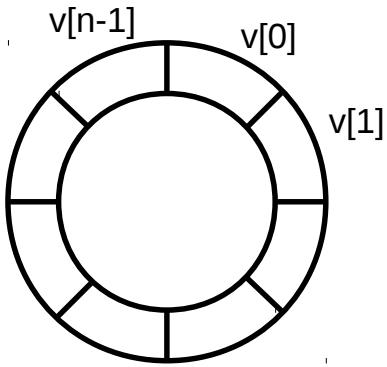
Fila



Operações: Desenfileirou

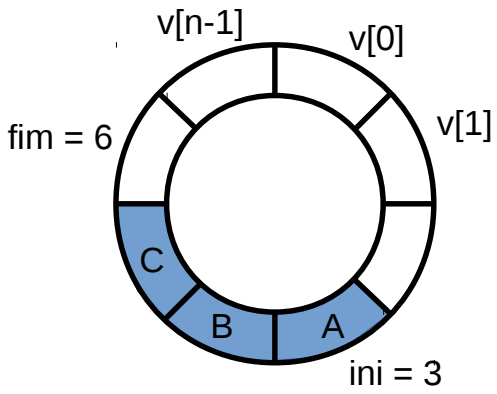
- **Enfileira** (*queue*): adiciona item no "fim"
- **Desenfileira** (*dequeue*): remove item do "início"

Fila: com vetor (fila circular)



Interpretar vetor (operações mod n)

Fila: com vetor (fila circular)

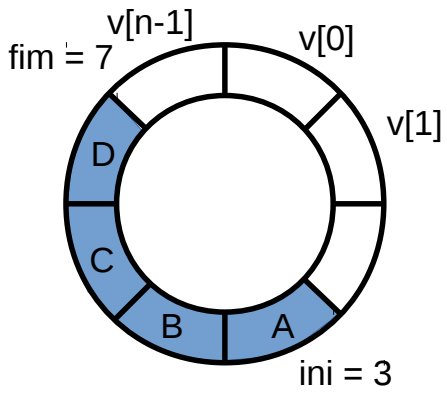


Primeira tentativa

Definição

```
#define N 1000
typedef struct { Item v[N];  int ini, fim; } Fila;
```

Fila: com vetor (fila circular)

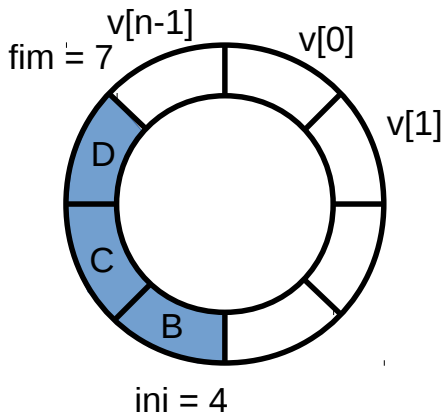


Enfileirou D

Definição

```
#define N 1000
typedef struct { Item v[N];  int ini, fim; } Fila;
```

Fila: com vetor (fila circular)

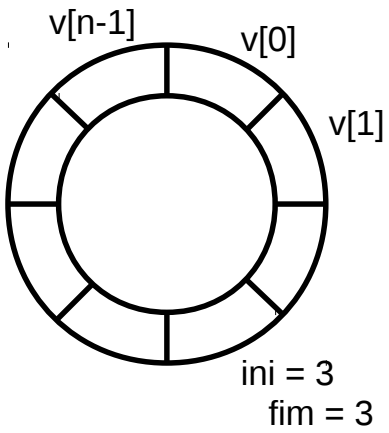


Desenfileirou (A)

Definição

```
#define N 1000
typedef struct { Item v[N];  int ini, fim; } Fila;
```

Fila: com vetor (fila circular)

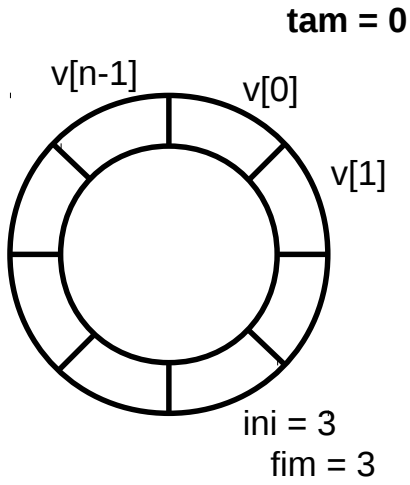


Está cheia ou vazia?

Definição

```
#define N 1000
typedef struct { Item v[N];  int ini, fim; } Fila;
```

Fila: com vetor (fila circular)

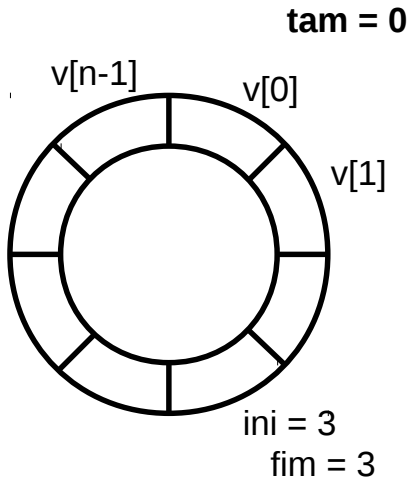


corrigida

Definição

```
#define N 1000
typedef struct { Item v[N];  int ini, fim, tam; } Fila;
```

Fila: com vetor (fila circular)



Definição

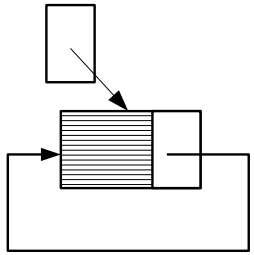
```
#define N 1000
typedef struct { Item v[N];  int ini, fim, tam; } Fila;
```

```
void enfileirar(Fila *p, Item i) {
    p->v[p->fim] = i;
    p->fim = (p->fim + 1) % N;
    (p->tam)++;
}
```

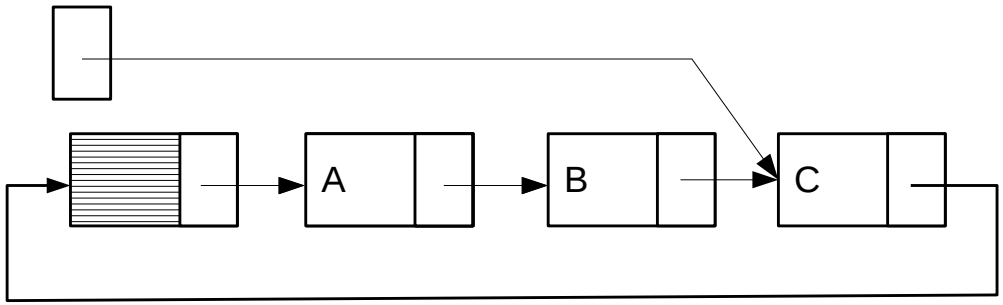
```
Item desenfileirar(Fila *p) {
    Item i = p->v[p->ini];
    p->ini = (p->ini + 1) % N;
    (p->tam)--;
    return i;
}
```

Fila: com lista ligada

Fila vazia:

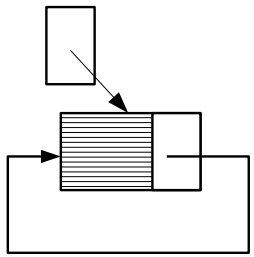


Após inserir A, B, C:

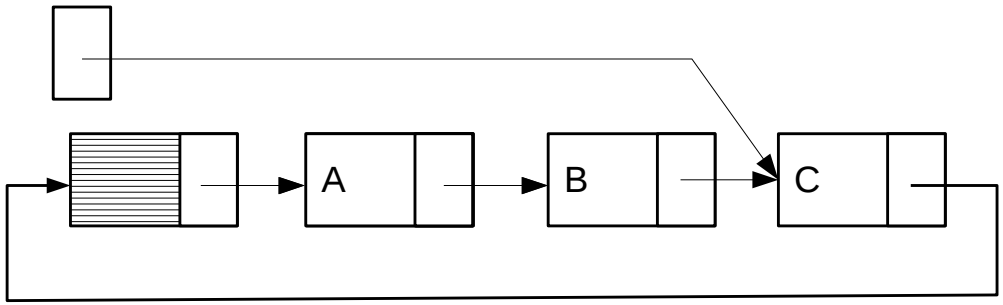


Fila: com lista ligada

Fila vazia:



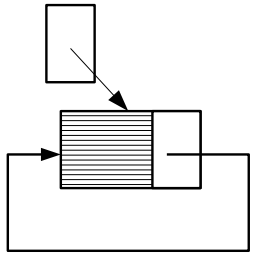
Após inserir A, B, C:



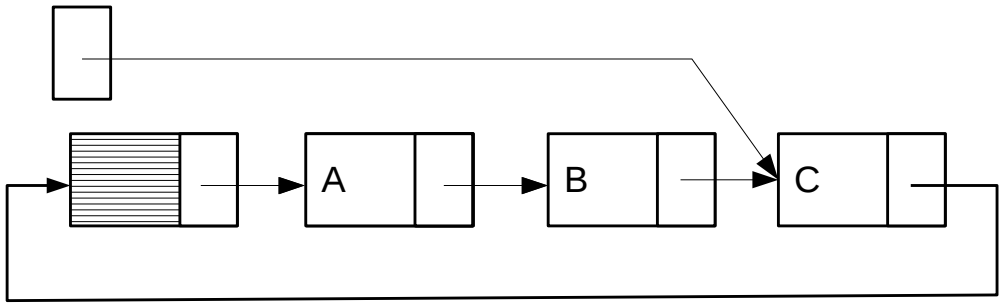
Alternativa: usar dois pointeiros `frente` e `fim`, ou usar `NULL` para vazia

Fila: com lista ligada

Fila vazia:



Após inserir A, B, C:



Alternativa: usar dois pointeiros `frente` e `fim`, ou usar `NULL` para vazia

Exercício: implementar filas com lista ligada em C

Aplicações

Exemplos de aplicações

Algumas aplicações de filas

- Percurso de estruturas de dados complexas (grafos etc.)
- Gerenciamento de fila de impressão
- Buffer do teclado etc.
- Escalonamento de processos...

Exemplos de aplicações

Algumas aplicações de filas

- Percurso de estruturas de dados complexas (grafos etc.)
- Gerenciamento de fila de impressão
- Buffer do teclado etc.
- Escalonamento de processos...

Algumas aplicações de pilhas

- Percurso de estruturas de dados complexas (grafos etc.) (**também!**)
- Balanceamento de parênteses
  - expressões matemáticas, linguagens de programação, HTML...
- Cálculo e conversão de notações
  - pré-fixa, pós-fixa, infixa (com parênteses)
- Recursão

Exemplos de aplicações

Algumas aplicações de filas

- Percurso de estruturas de dados complexas (grafos etc.)
- Gerenciamento de fila de impressão
- Buffer do teclado etc.
- Escalonamento de processos...

Algumas aplicações de pilhas

- Percurso de estruturas de dados complexas (grafos etc.) (**também!**)
- Balanceamento de parênteses
  - expressões matemáticas, linguagens de programação, HTML...
- Cálculo e conversão de notações
  - pré-fixa, pós-fixa, infixa (com parênteses)
- Recursão

Balanceamento de parênteses

Uma sequência de parênteses é válida se é:

- *vazia*
- [*sequência válida*]
- (*sequência válida*)

Exemplos

Válidos	**Inválidos**
	([]
[[]]	[[))
([()([])])	([)]

Balanceamento de parênteses

Uma sequência de parênteses é válida se é:

- *vazia*
- [*sequência válida*]
- (*sequência válida*)

Exemplos

Válidos	**Inválidos**
	([
[[]]	[[))
([()([])])	([)]

Para testar, ler cada símbolo e se:

1. **leu (ou [**: empilha lido
2. **leu]**: desempilha [
3. **leu)**: desempilha (

Balanceamento de parênteses

Uma sequência de parênteses é válida se é:

- *vazia*
- [*sequência válida*]
- (*sequência válida*)

Exemplos

Válidos	**Inválidos**
	([]
[[]]	[[))
([()([])])	([)]

Para testar, ler cada símbolo e se:

1. **leu (ou [**: empilha lido
2. **leu]**: desempilha [
3. **leu)**: desempilha (

Se alguma etapa falhar ou pilha *não* terminar vazia, sequência é inválida!

Implementação em C

```
int eh_balanceada(char *str) {
    NoPilha *pilha;   // pilha de caracteres
    int i, ok = 1;
    char par;
    iniciar_pilha(&pilha);
    for (i = 0; ok && str[i] != '\0'; i++) {  // para cada caractere
        if (str[i] == '['  || str[i] == '(') {
            empilhar(&pilha, str[i]);
        } else {
            par = desempilhar(&pilha);
            if (str[i] == ']' && par != '[') ok = 0;
            if (str[i] == ')' && par != '(') ok = 0;
        }
    }
    if (!eh_vazia(pilha)) ok = 0;  // deve terminar vazia
    destruir_pilha(pilha);
    return ok;
}
```

Implementação em C

```
int eh_balanceada(char *str) {
    NoPilha *pilha;   // pilha de caracteres
    int i, ok = 1;
    char par;
    iniciar_pilha(&pilha);
    for (i = 0; ok && str[i] != '\0'; i++) {  // para cada caractere
        if (str[i] == '['  || str[i] == '(') {
            empilhar(&pilha, str[i]);
        } else {
            par = desempilhar(&pilha);
            if (str[i] == ']' && par != '[') ok = 0;
            if (str[i] == ')' && par != '(') ok = 0;
        }
    }
    if (!eh_vazia(pilha)) ok = 0;  // deve terminar vazia
    destruir_pilha(pilha);
    return ok;
}
```

Se só houver (e), tem como melhorar?

Implementação em C

```
int eh_balanceada(char *str) {
    NoPilha *pilha;   // pilha de caracteres
    int i, ok = 1;
    char par;
    iniciar_pilha(&pilha);
    for (i = 0; ok && str[i] != '\0'; i++) {  // para cada caractere
        if (str[i] == '['  || str[i] == '(') {
            empilhar(&pilha, str[i]);
        } else {
            par = desempilhar(&pilha);
            if (str[i] == ']' && par != '[') ok = 0;
            if (str[i] == ')' && par != '(') ok = 0;
        }
    }
    if (!eh_vazia(pilha)) ok = 0;  // deve terminar vazia
    destruir_pilha(pilha);
    return ok;
}
```

Se só houver (e), tem como melhorar?

E se usássemos `return 0` dentro do `for`?

Notação de expressões

Exemplos

Infixa	*Pré-fixa*	*Pós-fixa*
a + b	+ a b	a b +
5 * ((9 + 8) * 4 * 6 + 7)	* 5 + * + 9 8 * 4 6 7	5 9 8 + 4 6 * * 7 + *

Notação de expressões

Exemplos

Infixa	*Pré-fixa*	*Pós-fixa*
a + b	+ a b	a b +
5 * ((9 + 8) * 4 * 6 + 7)	* 5 + * + 9 8 * 4 6 7	5 9 8 + 4 6 * * 7 + *

Notação de expressões aritméticas

1. **Infixa**: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade

Notação de expressões

Exemplos

Infixa	*Pré-fixa*	*Pós-fixa*
a + b	+ a b	a b +
5 * ((9 + 8) * 4 * 6 + 7)	* 5 + * + 9 8 * 4 6 7	5 9 8 + 4 6 * * 7 + *

Notação de expressões aritméticas

1. **Infixa**: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. **Pré-fixa**: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos

Notação de expressões

Exemplos

Infixa	*Pré-fixa*	*Pós-fixa*
a + b	+ a b	a b +
5 * ((9 + 8) * 4 * 6 + 7)	* 5 + * + 9 8 * 4 6 7	5 9 8 + 4 6 * * 7 + *

Notação de expressões aritméticas

1. **Infixa**: é a notação cotidiana
   - Ordem normal de leitura, com parênteses para evitar ambiguidade
2. **Pré-fixa**: é a notação polonesa do lógico Jan Lukasiewicz
   - Operador precede operandos
3. **Pós-fixa**: é notação polonesa reversa (RPN), das calculadoras HP,...
   - Operador sucede operandos

Usando expressões pós-fixas

Importância da notação pós-fixa

- É fácil calcular o valor da expressão: usando pilhas
- É similar às máquinas-a-pilhas (*stack machines*):
  - Java Virtual Machine (JVM)
  - Código de compilação intermediário, ...

Usando expressões pós-fixas

Importância da notação pós-fixa

- É fácil calcular o valor da expressão: usando pilhas
- É similar às máquinas-a-pilhas (*stack machines*):
  - Java Virtual Machine (JVM)
  - Código de compilação intermediário, ...

Exemplo: calculando uma expressão no quadro

Infixa: `2 * ( (2 + 1) * 4 + 1)` $= 26$
Pós-fixa: `2 2 1 + 4 * 1 + *`

Calculando expressões pós-fixas

Pseudocódigo

1. Para cada elemento lido:
   - Se for número n:
     - empilha n
   - Se for operador $\oplus$:
     - desempilha $operando_1$
     - desempilha $operando_2$
     - empilha $operando_1 \oplus operando_2$
2. Desempilha (único) valor da pilha e retorna.

Convertendo de infixa para pós-fixa

Objetivo

$$1 + 2 * 3 \hat{} 4 * 5 - 6 \Rightarrow 1\ 2\ 3\ 4 \hat{} *\ 5\ *\ +\ 6\ -$$

Convertendo de infixa para pós-fixa

Objetivo

$$1 + 2 * 3 \hat{} 4 * 5 - 6 \Rightarrow 1\ 2\ 3\ 4 \hat{} *\ 5\ *\ +\ 6\ -$$

Ideias

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - se o operador no topo tem prioridade maior ou igual ao operador de entrada, desempilhamos e copiamos na saída
  - empilhamos o operador novo

Convertendo de infixa para pós-fixa

Objetivo

$$1 + 2 * 3 \hat{} 4 * 5 - 6 \Rightarrow 1\ 2\ 3\ 4 \hat{} *\ 5\ *\ +\ 6\ -$$

Ideias

- Copiamos os números diretamente na saída
- Quando aparece um operador na entrada
  - se o operador no topo tem prioridade maior ou igual ao operador de entrada, desempilhamos e copiamos na saída
  - empilhamos o operador novo

Observações:

- Usamos uma nova operação do TAD: "olhar topo da pilha".
  É possível fazer sem?
- Como generalizar para o caso em que a expressão tem parênteses?

Exercícios

1. Complete as implementações (com vetor e lista ligada) de pilhas e filas com as operações auxiliares: `inicializa`, `eh_vazia`, `eh_cheia` (para vetor).
2. Um *deque* (*double-ended queue*) é um conjunto dinâmico com operações: `insere_inicio`, `insere_fim`, `remove_inicio`, `remove_fim`. Implemente um *deque* utilizando listas ligadas.
3. Implemente um programa que leia uma expressão aritimética infixada possívelmente com parênteses que reconheça os operadores binários $+$, -, *, $\hat{}$ e $/$ e em seguida imprima
   - a versão pós-fixa
   - a versão pré-fixa
   - o resultado da expressão