**Pilhas – implementação com Arranjo**

Sumário

[**Introdução** 1](#_Toc5035456)

[**Estruturas com *restrição de acesso*** 2](#_Toc5035457)

[**Pilhas – conceitos básicos** 3](#_Toc5035458)

[**Pilhas – aplicações** 5](#_Toc5035459)

[Controle de execução de programas e subrotinas 5](#_Toc5035460)

[Balanceamento de parênteses, colchetes e chaves 6](#_Toc5035461)

[Expressões aritméticas complexas 6](#_Toc5035462)

[**Pilha – implementação com arranjo** 7](#_Toc5035463)

[Código para a operação *criaPilha*: 8](#_Toc5035464)

[Código para a operação *vazia*: 9](#_Toc5035465)

[Código para a operação *push*: 9](#_Toc5035466)

[Código para a operação *pop*: 9](#_Toc5035467)

[Código para a operação *peek*: 10](#_Toc5035468)

[**Conclusão** 11](#_Toc5035469)

# **Introdução**

Nas aula anteriores foram vistos os conceitos de *Lista* e sua implementação por meio de uma estrutura de arranjo ou vetor. Aprendeu-se que uma lista consiste em uma estrutura linear ou unidimensional, isto é, uma *lista* é uma sequência de 0 ou mais elementos organizados em uma dimensão, onde todo elemento Xi, exceto o último, tem um sucessor, Xi+1, e também um antecessor, Xi-1, exceto o primeiro elemento. Do ponto de vista de acesso aos seus elementos, foi visto que uma *lista* constitui uma estrutura bastante flexivel, onde inclusões, exclusões e consultas podem ser realizadas, quer no início, no fim ou em qualquer posição da estrutura.

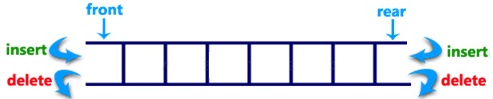
Agora será estudado o conceito de ***Pilha***, que consiste também numa estrutura linear ou unidimensional, mas com *restrição de acesso*. O que é afinal, a *restrição de acesso* numa estrutura linear?

# **Estruturas com *restrição de acesso***

Estruturas lineares com *restrição de acesso* ou *disciplina de acesso* nada mais são do que estruturas onde os acessos (inserções e retiradas) aos elementos ocorrem de forma restrita e imutável, visando a atender a uma política ou a um modelo de negócio.

Como exemplos de estruturas lineares com restrição de acesso, tem-se, das políticas mais gerais para as mais restritas:

1. **DEQUE**: acrônimo da expressão em inglês **D**ouble-**E**nded **Que**ue ou numa tradução livre, Fila de dupla extremidades. Nestas estruturas, inserções e retiradas podem ocorrer somente em ambos os lados ou extremidades, conforme ilustrado na figura a seguir.



1. **DEQUE de entrada restrita**: Nestas estruturas, as inserções podem ocorrer somente em uma das extremidades e as retiradas podem ocorrer somente em ambas as extremidades.



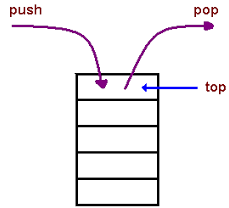
1. **DEQUE de saída restrita**: Aqui, as inserções podem ocorrer somente em ambas as extremidades da estrutura, mas as retiradas só ocorrem de um único lado.



1. **Filas**: Filas é um caso particular de DEQUE em que as retiradas são realizadas somente em uma extremidade, denominada *início*, ou *frente*, ou *front*, em inglês. E as inserções ou entradas ocorrem obrigatoriamente no lado oposto, denominado *fim*, ou *traseira*, ou em inglês, *rear*. Filas implementam uma politica de acesso denominada **FIFO**, que é um acrônimo da expressão em inglês **F**irst **I**n, **F**irst **O**ut, ou o primeiro que chega é o primeiro a ser atendido. Alguns autores denominam esta estrutura também **LILO** ou **L**ast **I**n, **L**ast **O**ut.



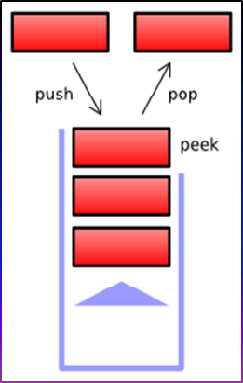
1. **Pilhas**: caso particular de DEQUE em que as inserções e as retiradas são realizadas obrigatoriamente em apenas uma das extremidades, denominada *topo*, ou *top* em inglês. Pilhas implementam uma politica de acesso denominada **LIFO**, que é um acrônimo da expressão em inglês **L**ast **I**n, **F**irst **O**ut, ou o último que chega é o primeiro a ser atendido. Também chamada alternativamente na literatura técnica de **FILO**, ou **F**irst **I**n, **L**ast **O**ut.



# **Pilhas – conceitos básicos**

Como visto, *Pilha* ou *Stack* é um tipo especial de estrutura linear com restrição de acesso, em que as inserções e retiradas são feitas em um mesmo lado da estrutura, denominado topo.

Do ponto de vista *estrutural*, uma *Pilha* é essencialmente similar a uma *Lista*: elementos são organizados em uma estrutura linear, unidimensional. A diferença entre ambas está no *comportamento*. A *Lista* não tem restrições para inserções e retiradas. Já a *Pilha* apresenta *restrição de acesso*, isto é, toda inserção (*push*), toda retirada (*pop*) e toda consulta (*peek*) ocorre sempre e obrigatoriamente de um mesmo lado da estrutura, o *topo*.



Exemplos de estruturas de *pilhas* na prática:

1. Fichas de pôquer numa mesa de jogo;



1. Pilhas de pratos em um buffet;



1. Sacas de açucar num armazém;



1. Caixas estocadas em um depósito.



1. Pilhas de livros sobre uma mesa.

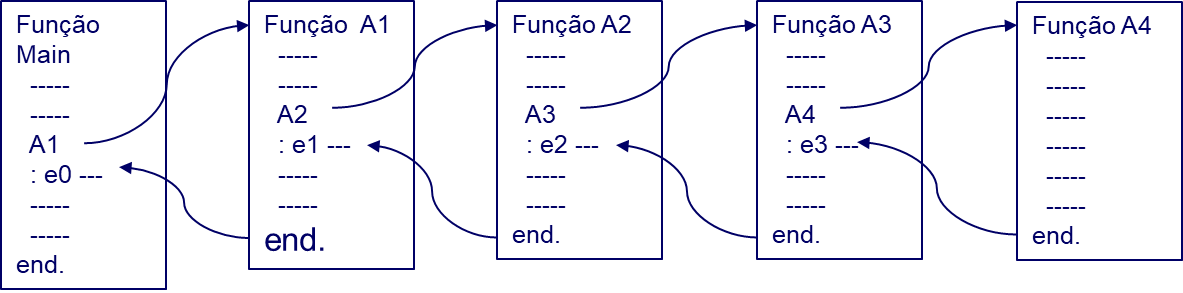


# **Pilhas – aplicações**

Pilha, por implementar a política LIFO, é solução natural para diversas situações em que se faz necessário guardar o caminho ou a trilha de processamento, para eventualmente refazer o caminho ou a trilha – estas situações em geral aplicam recursividade e são denominadas *algoritmos de backtracking*. Pilha também se aplica em situações em que há aninhamentos de estruturas, nas quais sub-estruturas mais internas precisam ser processadas antes das estruturas que a contenham. É o caso, por exemplo, de chamadas de subrotinas ou funções dentro de um programa, funções recursivas, mecanismos de *UNDO/REDO* de editores de texto, balanceamento de parênteses, colchetes e chaves em textos ou em expressões aritméticas, navegação em página web, resolução de expressões aritméticas nas notações polonesa e polonesa reversa, etc.. A seguir serão detalhadas um pouco mais algumas destas situações.

## Controle de execução de programas e subrotinas

A figura a seguir ilustra este caso.



Quando a função *Main* executa, ao chamar a função *A1*, interrompe sua execução, desvia controle da execução para *A1*, até esta concluir, quando o controle retorna para o endereço *e0* para continuar a execução de *Main*. Situação idêntica ocorre quando *A1* chama *A2*, *A2* chama *A3*, e assim sucessivamente. Também se aplica a chamadas recursivas, isto é, uma função chamando a sí própria

Como o sistema operacional controla este fluxo de chamadas e de retornos? Na verdade o S.O. utiliza uma solução de pilha – a conhecida *pilha de execução*. Ao invocar uma função, o S.O. cria um novo espaço de trabalho, com todos os parâmetros e todas as variáveis locais da função.  Esse espaço de trabalho é colocado na *pilha de execução* (sobre o espaço de trabalho que invocou a função) e a execução da função começa.  Quando a execução da função termina, o seu espaço de trabalho é removido da *pilha* e descartado.  O espaço de trabalho que estiver agora no topo da *pilha* (que é o da função que realizou a última chamada) é reativado e a execução é retomada do ponto em que havia sido interrompida.

## Balanceamento de parênteses, colchetes e chaves

Considere o problema de verificar se uma string contendo estes caracteres especiais está balanceada ou não. Veja as ilustrações a seguir:

{ ( ( [ ] { ( ) } ) ) } - sequência balanceada;

{ ( } [ ] { ( } ) ( ) } - sequência não balanceada.

O algoritmo para resolver este problema é simples desde que usando uma estrutura de *Pilha*: varrendo a *string* da esquerda para para a direita, ao encontra um caractere do tipo “abre”, um **(** ou **[** ou **{**, empilha este caractere. Ao encontrar um caractere “fecha”, **)**, **]** ou **}**, desempilha e verifica se o caractere desempilhado “bate” – faz par – com o encontrado. Ao término da varredura da string, a sequência estará balanceada se a pilha estiver vazia, ou seja, para todo empilhamento, houve um desempilhamento com o caractere correspondente e na ordem correta. Qualquer falha no processo de ‘*matching*’ ou pilha não vazia ao final da varredura, significa sequência não balanceada.

## Expressões aritméticas complexas

Uma representação para **expressões aritméticas** que seja conveniente do ponto de vista computacional é assunto de interesse, por exemplo, na área de **compiladores**. A notação tradicional (também chamada *infixa*) é ambígua e, portanto, obriga o pré-estabelecimento de **regras de prioridade**. Isso torna a tarefa computacional mais complexa. Outras notacões são apresentadas a seguir, considerando-se apenas operações binárias (com dois operandos):

* **Notação completamente Parentizada:** acrescenta-se sempre um parêntese a cada par de operandos e seu operador.

**Exemplo:** tradicional: A \* B - C / D – completamente parentizada: ((A\*B)-(C/D))

* **Notação Polonesa (ou prefixada):** os operadores aparecem imediatamente antes dos operandos. Esta notação especifica quais operadores devem ser calculados, e em que ordem. Por esse motivo dispensa o uso de parênteses, sem ambiguidades.

**Exemplo:** tradicional: A \* B - C / D – polonesa: - \* A B / C D

* **Notação Polonesa Reversa (ou posfixada):** é como a polonesa na qual os operadores aparecem após os operandos.

**Exemplo:** tradicional: A \* B - C / D – polonesa reversa: A B \* C D / -

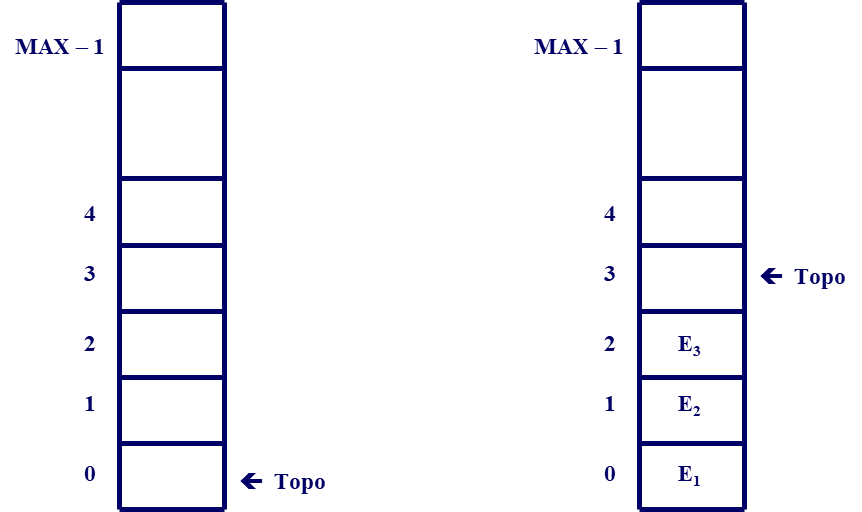
O uso de Pilha facilita o trabalho de tradução da expressão pelo compilador. Para o caso da notação posfixada, por exemplo, basta ir lendo a expressão da esquerda para a direita e todo operando encontrado é empilhado. Ao encontrar um operador, desempilha dois operandos e realiza a operação, sendo que o resultado se torna mais um operando e portanto é empilhado. Ao fim, estando a expressão correta, sobrará na pilha apenas um operando que é o resultado final da expressão.

Há algoritmos específicos para transformar uma expressão *infixa* (na notação tradicional) para uma *posfixa* ou *prefixa*. Estes algoritmos também utilizam o conceito LIFO.

# **Pilha – implementação com arranjo**

Da mesma forma que foi feito em relação a *Lista*, onde foi concebido e implementado um *TAD Lista* englobando a estrutura de armazenamento e as principais operações de uma lista abstrata, será demonstrado nesta seção uma alternativa de implementação de uma pilha usando arranjo para armazenar sua estrutura.

Do ponto de vista estrutural, uma *Pilha* é exatamente igual a uma *Lista*. A pilha terá um arranjo para armazenar os seus elementos e uma marca do *topo*. Esta marca é essencial, visto que facilita sobremaneira as operações de empilhar (*push*), desempilhar (*pop*) e espiar elemento do topo (*peek*). A figura abaixo ilustra a estrutura da Pilha. À esquerda a representação de uma pilha vazia e à direita, uma pilha com alguns elementos já inseridos.



A principal característica comportamental de Pilhas é: inserções e retiradas realizadas somente no topo da pilha. Desta forma, a marca do *Topo* move com as inserções e retiradas de elementos. Note que no modelo abstrato acima, o *Topo* aponta para a primeira posição disponível no Arranjo – no caso da pilha vazia, a posição disponível é 0. No caso de elementos já inseridos, é a posição acima da ocupada pelo elemento do topo. Note ainda que o *Topo* representa o tamanho dinâmico da pilha, ou seja, a quantidade de elementos “empilhados”: à esquerda, 0; à direita, 3.

A estrutura acima pode ser facilmente mapeada para qualquer linguagem de programação. A seguir a codificação em Português Estruturado e em Linguagem C:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| const MAX = 1000  Dados: **Registro**  ***// campos da estrutura de dados da Pilha***  **FimRegistro**  Pilha: **Registro**  vet: vetor [MAX] de Dados  topo: inteiro  **FimRegistro** | **#define** MAX 1000  **typedef struct {**  ***// campos da estrutura de dados da Pilha***  **} Dados;**  **typedef struct {**  **Dados** vet[MAX];  **int** topo;  **} Pilha;** |

Complementando a definição de um *TAD Pilha*, faz-se necessário definir as operações sobre esta estrutura. Basicamente, uma pilha possui duas operações essenciais: a inserção de um elemento no topo da pilha, operação denominada *push*, e a retirada do elemento do topo da pilha, denominada *pop*. No presente TAD Pilha, serão consideradas as seguintes operações:

1. criaPilha: inicializa uma pilha com topo = 0;
2. vazia: operação booleana que verifica se a pilha está vazia;
3. push: insere um elemento no topo da pilha (empilha);
4. pop: retira o elemento do topo da pilha (desempilha);
5. peek: espia (ou consulta) o elemento do topo da pilha (sem desempilhar).

## Código para a operação *criaPilha*:

Esta operação consiste apenas em setar o *topo* de uma variável do tipo Pilha, já criada, para 0:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** criaPilha (**var** pilha: Pilha)  **Inicio**  pilha.topo 🡨 0  **FimFuncao** | **void** criaPilha (**Pilha** \*pilha)  **{**  pilha->topo = 0;  **}** |

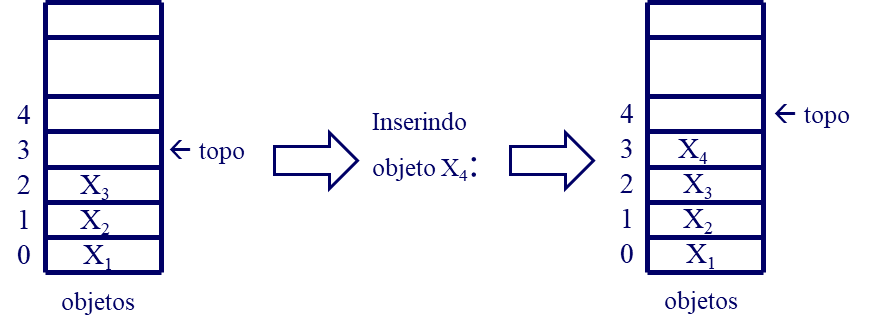
## Código para a operação *vazia*:

Esta operação *booleana*, retorna true se a pilha estiver vazia (*topo* igual a 0):

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Funcao** vazia (**var** pilha: Pilha): lógico  **Inicio**  **Retorne** (pilha.topo = 0)  **FimFuncao** | **int** vazia (**Pilha** \*pilha)  **{**  **return** (pilha->topo == 0);  **}** |

## Código para a operação *push*:

Esta operação recebe um elemento de dados pelo parâmetro *x* e insere no topo da pilha. A figura a seguir ilustra este procedimento, simulando a inserção de um 4º elemento na pilha:

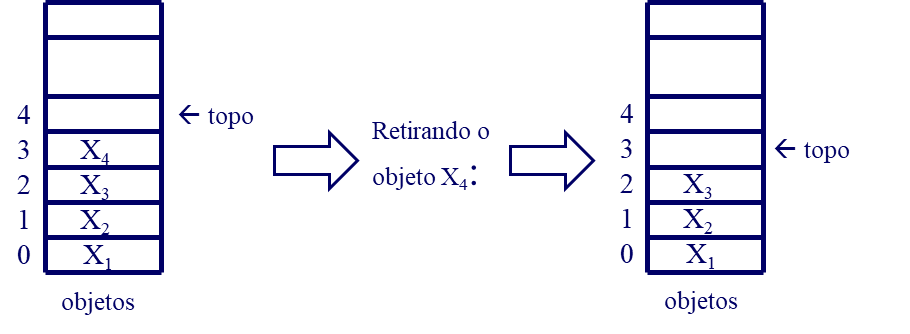


Faz-se necessário verificar se a pilha está cheia e neste caso retorna um código de erro pertinente por meio do parâmetro *codErro*. A seguir, código para esta operação:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** push (x: Dados, **var** pilha: Pilha,  **var** codErro: inteiro)  **Inicio**  codErro = 0;  **Se** (pilha.topo >= MAX) **então**  codErro 🡪99 ***// código de overflow***  **FimSe**  pilha.vet[pilha.topo] 🡨 x  pilha.topo 🡨 pilha.topo + 1  **FimFuncao** | void push (Dados x, Pilha \*pilha, int \*codErro)  {  codErro = 0;  if (pilha->topo >= MAX) {  codErro = 99; ***// código para pilha cheia***  }  pilha->vet[pilha->topo] = x;  pilha->topo++;  } |

## Código para a operação *pop*:

Nesta operação não é necessário parâmetro de entrada, *pop* retira e devolve o elemento do topo da pilha. Antes testa *underflow* da Pilha. A figura a seguir ilustra este processo, simulando a retirada do elemento do topo da pilha.

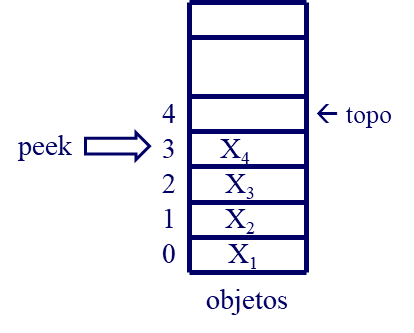


Deve-se obter e retornar ao chamador pelo parâmetro *\*x* o elemento abaixo do *Topo* e então decrementar este. A seguir, código para esta operação:

|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** pop (**var** pilha: Pilha, **var** x: Dados,  **var** codErro: inteiro)  **Inicio**  codErro = 0;  **Se** (vazia(pilha)) **então**  codErro 🡪 88 ***// código de underflow***  **FimSe**  x 🡨 pilha.vet[pilha.topo – 1]  pilha.topo 🡨 pilha.topo – 1  **FimFuncao** | void pop (Pilha \*pilha, Dados \*x, int \*codErro)  {  codErro = 0;  if (vazia(pilha)) {  codErro = 88; ***// código para pilha vazia***  }  \*x = pilha->vet[pilha->topo – 1];  pilha->topo--;  } |

## Código para a operação *peek*:

Neste método não é necessário parâmetro de entrada, *peek* devolve o elemento do topo da pilha (isto é, *Topo – 1*), sem retirá-lo. Antes testa *underflow* da Pilha. A figura a seguir ilustra a retirada.



|  |  |
| --- | --- |
| **Português Estruturado** | **Linguagem C** |
| **Procedimento** peek(**var** pilha: Pilha, **var** x: Dados,  **var** codErro: inteiro)  **Inicio**  codErro = 0;  **Se** (vazia(pilha)) **então**  codErro 🡪 88 ***// código de underflow***  **FimSe**  x 🡨 pilha.vet[pilha.topo – 1]  **FimFuncao** | void peek (Pilha \*pilha, Dados \*x, int \*codErro)  {  codErro = 0;  if (vazia(pilha)) {  codErro = 88; ***// código para pilha vazia***  }  \*x = pilha->vet[pilha->topo – 1];  } |

# **Conclusão**

Nesta apostila estudou-se o conceito de *Pilha*. Inicialmente foram apresentados os conceitos de estruturas lineares com restrição de acesso, nas quais Pilhas se enquadram. A seguir foram apresentados os conceitos básicos de uma pilha e, o mais importante, diversas situações práticas em que se aplica a política LIFO como solução trivial. Finalizando o capítulo, foi apresentada uma alternativa de implementação de pilha usando arranjo.