

# Pilha Estática

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados I (AE22CP)  
Engenharia de Computação  
Departamento Acadêmico de Informática (Dainf)  
Universidade Tecnológica Federal do Paraná (UTFPR)  
Campus Pato Branco

- Pilha
  - Exemplo
- Pilhas Estáticas
- TAD Pilhas Estáticas
- Expressões Matemáticas

- Lista

...	$x_1$	$x_2$	...	$x_n$	...
-----	-------	-------	-----	-------	-----

- Listas estáticas: alocação contígua na memória

**Endereço na memória**  
**Conteúdo na memória**

	3000	3001	3003	3004	
...					...

- Tipos especiais de listas:
  - Fila
  - Pilha

**Pilha**

- É uma lista linear em que os elementos são inseridos e removidos em uma de suas extremidades
- *Last-in, first-out* (LIFO)
- A inserção de novos itens e a remoção é sempre no topo da estrutura

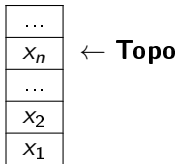


- Todas as operações em uma pilha podem ser imaginadas como as que ocorre, por exemplo, em uma pilha de pratos
- Aplicações
  - Avaliação de expressões numéricas
  - Processamento de linguagens
  - Mecanismo de fazer/desfazer em editores de texto
  - Mecanismo de avançar/retornar em páginas web
  - Execução de programas
  - Etc

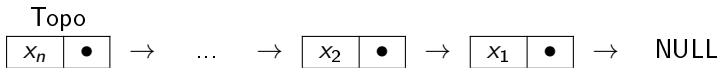
- Principais operações em pilhas
  - Criar
  - Verificar se a pilha está cheia
  - Empilhar
  - Desempilhar
  - Verificar o item que está no topo
  - Liberar



- Representação
  - Alocação contígua (estática)



- Alocação encadeada



# Pilha

## Exemplo

```
# include <stdio.h>

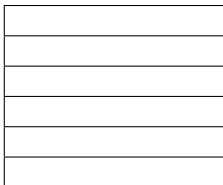
int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: inicialmente vazia



# Pilha

## Exemplo

```
# include <stdio.h>

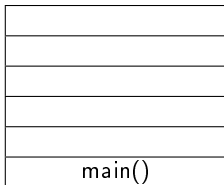
int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: a função *main* é chamada



# Pilha

## Exemplo

```
# include <stdio.h>

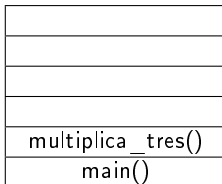
int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: a função *main* chama *multiplica\_tres*



# Pilha

## Exemplo

```
# include <stdio.h>

int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: a função *multiplica\_tres* chama *multiplica\_dois*

multiplica_dois()
multiplica_tres()
main()

# Pilha

## Exemplo

```
# include <stdio.h>

int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: a função *multiplica\_dois* retorna o valor 20

20
multiplica_tres()
main()

# Pilha

## Exemplo

```
# include <stdio.h>

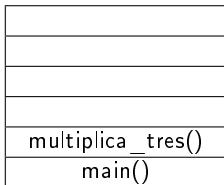
int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: a função *multiplica\_dois* termina o seu trabalho e é desempilhada



# Pilha

## Exemplo

```
# include <stdio.h>

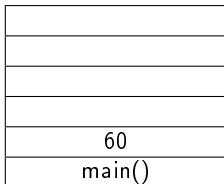
int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: a função *multiplica\_tres* retorna 60





# Pilha

## Exemplo

```
# include <stdio.h>

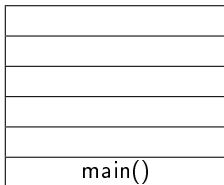
int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: a função *multiplica\_tres* termina o seu trabalho e é desempilhada



# Pilha

## Exemplo

```
# include <stdio.h>

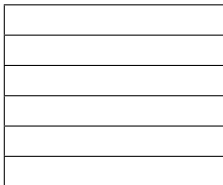
int multiplica_tres(int x, int y, int z){
    return multiplica_dois(x, y) * z;
}

int multiplica_dois(int a, int b){
    return a * b;
}

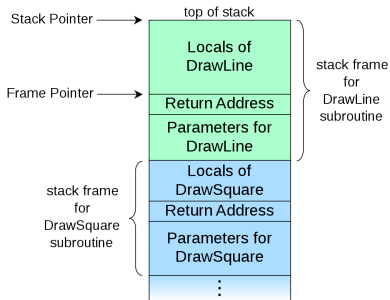
int main(){
    printf("%d * %d * %d = %d", 4, 5, 3, multiplica_tres(5, 4, 3));

    return 0;
}
```

- Pilha de execução: após a finalização do programa, a pilha encontra-se novamente vazia



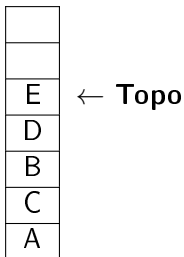
- No exemplo anterior, o processo de empilhar e de desempilhar foi apresentado de forma simplificada
- Na realidade, em cada chamada de função, um quadro com parâmetros da função, variáveis locais e endereço de retorno é empilhado



## Pilhas Estáticas

# Pilhas Estáticas

- Implementação semelhante ao da lista estática
  - Uso de vetores
- Há um cursor para controlar a posição do topo
  - Uma variável na *struct* da pilha pode ser usada para armazenar a posição do topo



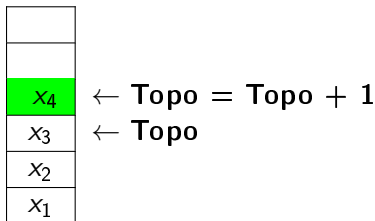
# Pilhas Estáticas

- Criar a pilha
  - Gera e inicializa a pilha com um tamanho determinado
  - A variável topo é inicializada com  $-1$  (pilha vazia)

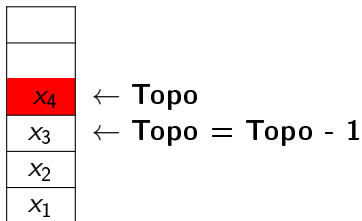


Topo = -1

- Empilhar (*push*)
  - Primeiramente, deve ser verificado se a pilha está cheia (*stack overflow*)
  - Ao empilhar o novo item, a variável *topo* é incrementada em 1



- Desempilhar (*pop*)
  - Primeiramente, deve ser verificado se a pilha já está vazia (*stack underflow*)
  - Ao desempilhar um item, a variável *topo* é decrementada em 1





- Exemplo de pilha estática vazia

	<b>Topo = -1</b>									
<b>Índice</b>	0	1	2	3	4	5	6	7	8	9
<b>Pilha</b>										

- Empilhar 11

	Topo = 0									
Índice	0	1	2	3	4	5	6	7	8	9
Pilha	11									

# Pilhas Estáticas

- Empilhar 8

	Topo = 1									
Índice	0	1	2	3	4	5	6	7	8	9
Pilha	11	8								

- Empilhar 15

	Topo = 2									
Índice	0	1	2	3	4	5	6	7	8	9
Pilha	11	8	15							

- Empilhar 16

	Topo = 3									
Índice	0	1	2	3	4	5	6	7	8	9
Pilha	11	8	15	16						

- Desempilhar

	Topo = 2									
Índice	0	1	2	3	4	5	6	7	8	9
Pilha	11	8	15	16						

- Empilhar 22

	Topo = 3									
Índice	0	1	2	3	4	5	6	7	8	9
Pilha	11	8	15	22						

## **TAD Pilhas Estáticas**



- Operações básicas para uma pilha
  - Criar uma pilha
  - Verificar se a pilha está vazia
  - Verificar se a pilha está cheia
  - Empilhar
  - Desempilhar
  - Imprimir pilha
  - Liberar a pilha

- Primeiro passo: definir arquivo .h

```
// Pilha.h
#define TAM_MAX 100 // tamanho máximo da pilha

typedef struct{
    int item[TAM_MAX];
    int topo;
}Pilha;

Pilha* criar_pilha();

int pilha_cheia(Pilha *p);

int pilha_vazia(Pilha *p);

int empilhar(Pilha *p, int key);

int desempilhar(Pilha *p);

void imprimir_pilha(Pilha *p);

void liberar(Pilha *p);
```

- Segundo passo: definir arquivo .c

```
#include "Pilha.h"

Pilha* criar_pilha(){
    Pilha *p = (Pilha *) malloc(sizeof(Pilha));

    p->topo = -1;

    return p;
}

int pilha_cheia(Pilha *p){
    if (p == NULL)
        return -1;
    else if (p->topo >= (TAM_MAX - 1))
        return 1;
    else
        return 0;
}
```

# TAD Pilhas Estáticas

```
int pilha_vazia(Pilha *p){
    if (p == NULL)
        return -1;
    else if (p->topo < 0)
        return 1;
    else
        return 0;
}

int empilhar(Pilha *p, int key){
    if (!pilha_cheia(p)){
        p->topo++;
        p->item[p->topo] = key;

        return 1;
    }

    return 0;
}
```

# TAD Pilhas Estáticas

```
int desempilhar(Pilha *p){
    int item = INT_MIN;
    if (!pilha_vazia(*p)){
        item = p->item[p->topo];
        p->topo--;
    }
    return item;
}

void imprimir_pilha(Pilha *p){
    Pilha aux = *p; // cópia da pilha

    while (!pilha_vazia(&aux)){
        item = desempilhar(&aux);
        printf("%d\n", item);
    }
}

void liberar(Pilha *p){
    if (!pilha_vazia(p));
    free(p);
}
```

- Exercício 1: aproveitando a TAD anterior, faça:
  - Altere a TAD de forma que os itens possam operar com caracteres
- Exercício 2: Utilizando uma pilha, escreva um método que receba um número inteiro positivo no formato decimal e converte este número para o formato binário.

## Expressões Matemáticas

- Pilhas são muito usadas no processamento de linguagens
  - Compiladores
- Uma das aplicações é a conversão e avaliação de expressões algébricas/numéricas
  - Consistência de parênteses: verificar a existência de fechamento de parênteses para cada abertura
  - Notação infixa: o operador está entre operandos ( $A + B$ )
  - Notação pré-fixa (polonesa): o operador precede os operandos ( $+AB$ )
  - Notação pós-fixa (polonesa inversa): o operador procede os operandos ( $AB+$ )



- Consistência de parênteses
  - Recebe uma expressão algébrica com letras e símbolos
  - Durante o processamento da *string*, caso o caractere "(" seja lido, o mesmo é colocado na pilha
  - Caso o caractere ")" seja lido, é removido o item no topo da pilha
  - Os demais caracteres são ignorados
  - A função retorna 1 se a operação for bem-sucedida (pilha vazia) ou 0, caso contrário (pilha com item ou *underflow*)

# Expressões Matemáticas

- Consistência de parênteses
  - Exemplo para a string `"((a+b)-c)*(a)"`

Posição na String	Caractere	Pilha	Operação
0	(	(	Empilhar
1	(	((	Empilhar
2	a	((	-
3	+	((	-
4	b	((	-
5	)	(	Desempilhar
6	-	(	-
7	c	(	-
8	)		Desempilhar
9	*		-
10	(	(	Empilhar
11	a	(	-
12	)		Desempilhar
Expressão consistente!			

- Notação infixa
  - Convencional
  - Pode ser necessário o uso de parênteses
    - $A + B * C$
    - $(A + B) * C$
    - $A + (B * C)$

- Notação pré-fixa (polonesa)
  - Operadores antes dos operandos
  - Determina os operadores e a respectiva ordem para o cálculo de uma expressão
  - Não há necessidade de uso de parênteses
  - Exemplos infixo  $\times$  pré-fixo
    - $A + B - C : - + ABC$
    - $(A + B) * C : * + ABC$
    - $A + B * C : + A * BC$
    - $A * B - C / D : - * AB / CD$

- Notação pós-fixa (polonesa reversa)
  - Operadores após os operandos
  - Utilizado em vários equipamentos eletrônicos: calculadoras e computadores
  - A ordem dos operandos na notação infixa e na notação polonesa (reversa ou não) é idêntica
  - Os operadores aparecem na ordem em que devem ser calculados
  - Exemplos infixo  $\times$  pós-fixo
    - $A + B - C : AB + C -$
    - $(A + B) * C : AB + C *$
    - $A + B * C : ABC * +$
    - $A * B - C / D : AB * CD / -$

- Processamento de expressões na notação pós-fixa
  - ① Cada operando é empilhado
  - ② Processamento de cada operador
    - ① Dois operandos são desempilhados
    - ② A operação é executada
    - ③ O resultado da operação é empilhado
  - ③ Retorne o resultado da operação

- Exemplo de processamento para a expressão  $7 - (6 + 2)/4 + 3$ 
  - Notação pós-fixa:  $762 + 4/ - 3 +$

Valor lido	Operação	Pilha
7	empilhar	7
6	empilhar	7, 6
2	empilhar	7, 6, 2
+	somar	7, 8
4	empilhar	7, 8, 4
/	dividir	7, 2
-	subtrair	5
3	empilhar	5, 3
+	somar	8
	<b>resultado</b>	<b>8</b>

- Conversão de infixa para pós-fixa
  - A expressão infixa deve ser percorrida da esquerda para a direita
  - Uma pilha é utilizada para armazenar operadores e os caracteres '(', '{' e '['
  - Operadores que possuem precedências iguais:
    - $+$  e  $-$
    - $*$  e  $/$
  - Os operadores  $*$  e  $/$  possuem maior precedência em relação aos  $+$  e  $-$
  - O operador  $^$  (e.g.  $2^3$ , que significa "2 elevado a 3") possui maior precedência em relação aos operadores  $*$ ,  $/$ ,  $+$  e  $-$



- Conversão de infixa para pós-fixa
  - 1 A expressão infixa deve ser percorrida da esquerda para a direita
  - 2 Se um operando é encontrado, o mesmo é colocado na saída (string que representa a expressão na notação pós-fixa)
  - 3 Se um operador é encontrado:
    - 1 Se a pilha de estiver vazia ou o operador possui maior precedência em relação ao topo da pilha (ou no topo estiver o caractere '(', '{' ou '[' ), empilhe-o
    - 2 Caso contrário, desempilhar todos os operadores com precedência maior ou igual do operador encontrado e colocá-los na saída. Após, o operador lido é empilhado
  - 4 Se '(', '{' ou '[' for encontrado, empilhe-o
  - 5 Se ')', '}' ou ']' for encontrado, desempilhe os elementos até chegar nos caracteres '(', '{' ou '[' e descarte-os
  - 6 Repita os passos anteriores (de 2 a 5) até toda a expressão ser percorrida
  - 7 Esvaziar a pilha, colocando os operadores na saída

# Expressões Matemáticas

- Conversão de infixa para pós-fixa
  - Exemplo:  $A - B * C + D$

Entrada	Pilha	Saída	Descrição
A		A	Impressão do operando A. A pilha permanece vazia.
-	-	A	O operador '-' é empilhado.
B	-	AB	O operando B é impresso. Não há alteração na pilha.
*	- *	AB	O operador "*" é empilhado, já que possui maior precedência em relação ao que estava no topo.
C	- *	ABC	O operando C é impresso.
+	+	ABC*-	O operador possui precedência menor ou igual ao que está no topo da pilha. Então, desempilhar "*" e "-", colocando-os na saída. Empilhar o caractere "+".
D	+	ABC*-D	O operando D é impresso.
		ABC*-D+	Como a expressão terminou de ser percorrida, a pilha deve ser esvaziada

- Conversão de infix para pré-fixa
  - 1 Inverter a expressão infix: '(' passa a ser ')' e ')' passa a ser '('
    - Exemplo 1:  $a + b - c$  passa a ser  $c - b + a$
    - Exemplo 2:  $a + (b * c)$  passa a ser  $(c * b) + a$
  - 2 Converter a expressão para notação pós-fixa
  - 3 Inverter a expressão pós-fixa

- Conversão de infixa para pré-fixa
  - Exemplo:  $A - B * C + D$ 
    - 1 Inverter a expressão infixa:  $D + C * B - A$
    - 2 Converter a expressão para notação pós-fixa:  $DCB*+A-$
    - 3 Inverter a expressão pós-fixa:  $-A+*BCD$





- Adaptação de uma *struct* no TAD de pilha

```
typedef struct {  
    char key;  
}Item;  
  
typedef struct{  
    Item item[TAM_MAX];  
    int topo;  
}Pilha;
```

- ou

```
typedef struct{  
    char item[TAM_MAX];  
    int topo;  
}Pilha;
```

- Algumas funções também devem ser modificadas para suportar a pilha de caracteres

-  Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S.  
*Algoritmos: teoria e prática.*  
Elsevier, 2012.
-  Pereira, S. L.  
*Estrutura de Dados e em C: uma abordagem didática.*  
Saraiva, 2016.
-  Szwarcfiter, J.; Markenzon, L.  
*Estruturas de Dados e Seus Algoritmos.*  
LTC, 2010.
-  Tenenbaum, A.; Langsam, Y.  
*Estruturas de Dados usando C.*  
Pearson, 1995.



Ziviani, M.

*Projetos de Algoritmos: com implementações em Pascal e C.*  
Thomson, 2004.