MCTA028-15: Programação Estruturada

Aula 13: Pilhas
(Segunda Parte)

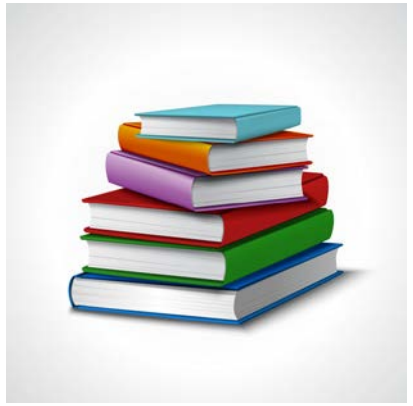
Wagner Tanaka Botelho
wagner.tanaka@ufabc.edu.br / wagtanaka@gmail.com
Universidade Federal do ABC (UFABC)
Centro de Matemática, Computação e Cognição (CMCC)

Introdução

Introdução

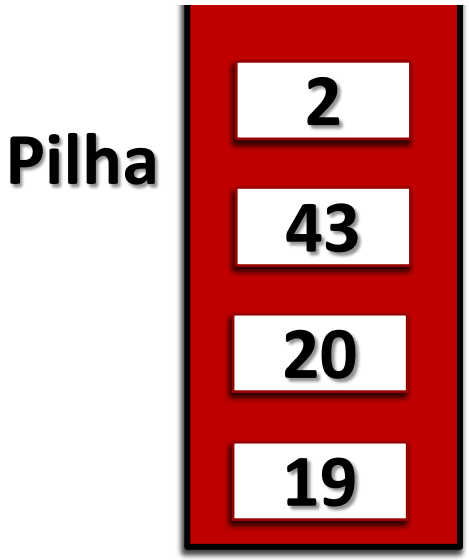
- Diferente das listas, os itens de uma pilha se encontram dispostos uns sobre os outros:
  - Assim, pode-se inserir um novo item na pilha se o colocarmos ACIMA dos demais e apenas removeremos o item que estiver no TOPO da pilha.

- Se quiser acessar determinado elemento da pilha, deve-se REMOVER todos os que estiverem sobre ele:
  - Portanto, as pilhas são conhecidas como estruturas do tipo ÚLTIMO a ENTRAR, PRIMEIRO a SAIR ou *Last In First Out* (LIFO).



Introdução

- Na Ciência da Computação:
  - Uma pilha é uma estrutura de dados linear utilizada para armazenar e organizar dados em um computador;
  - Uma estrutura do tipo pilha é uma sequência de elementos do MESMO TIPO.



Tipos de Pilhas

Tipos de Pilhas

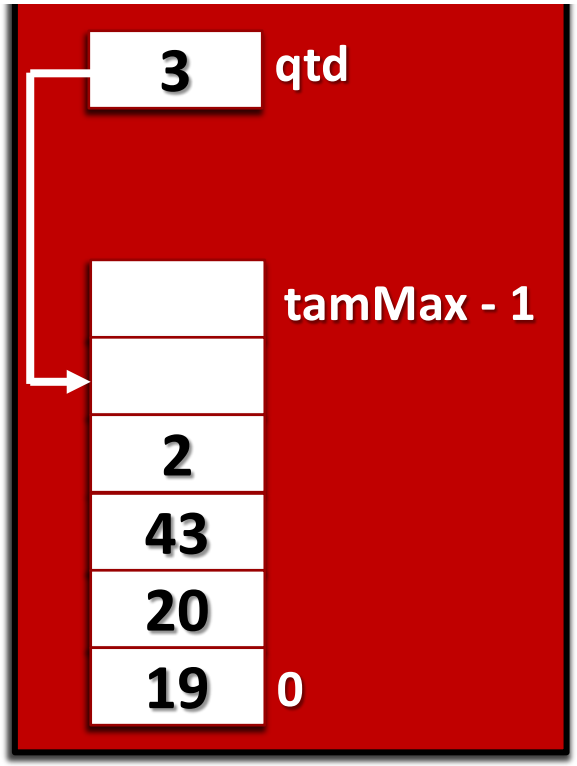
- Alocação estática com acesso sequencial:
  - Espaço de memória é alocado no momento da compilação do programa, ou seja, é necessário definir o número máximo de elementos da pilha.

- Alocação dinâmica com acesso encadeado:
  - Espaço de memória é alocado em TEMPO DE EXECUÇÃO:
    - A PILHA CRESCE à medida que NOVOS elementos são ARMAZENADOS;
    - A PILHA DIMINUI à medida que elementos são REMOVIDOS.
  - Cada elemento pode estar em uma área distinta da memória:
    - Cada elemento da pilha deve armazenar, além da sua informação, o ENDEREÇO DE MEMÓRIA onde se encontra o próximo elemento;
    - Para acessar um elemento, é preciso PERCORRER todos os seus antecessores na pilha.

Pilha Sequencial Estática

Pilha Sequencial Estática

- Tipo mais simples de pilha, definida utilizando um *array* e um CAMPO ADICIONAL (**qtd)** que serve para indicar o quanto do *array* está ocupado pelos elementos inseridos na pilha.



Definindo o Tipo

*Ex_03.c

```c
#include<stdio.h>
#include<stdlib.h>

#define tamMax 20

struct pilha{
    int qtd;
    int num[tamMax];
};

typedef struct pilha Pilha;
```

Definindo uma constante (tamanho do *array*).

Definindo o tipo pilha com dois campos:
+ qtd (int): indica quantidade de elementos inseridos na pilha;
+ *array* num do tipo int: tipo de dado a ser armazenado na pilha.

Redefinindo a *struct* para encurtar o comando.

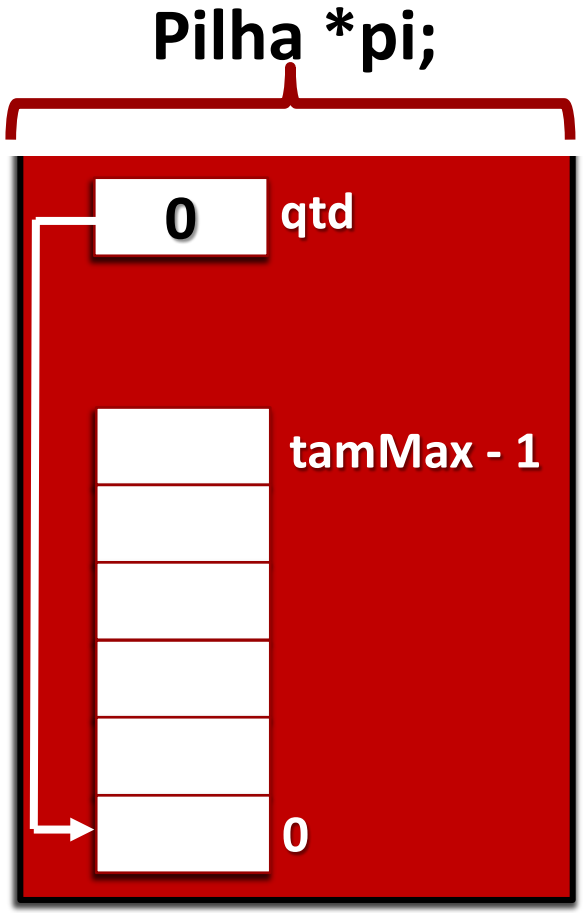
Criando a Pilha

```
13  Pilha* cria_Pilha(){
14      Pilha *pi;
15
16      pi = (Pilha *) malloc(sizeof(Pilha));
17
18      if(pi != NULL){
19          pi->qtd = 0;
20      }
21
22      return pi
23  }
```

Ponteiro para estrutura pilha.

Alocando a área de memória para a pilha.

Armazena a quantidade de elementos inseridos na lista.

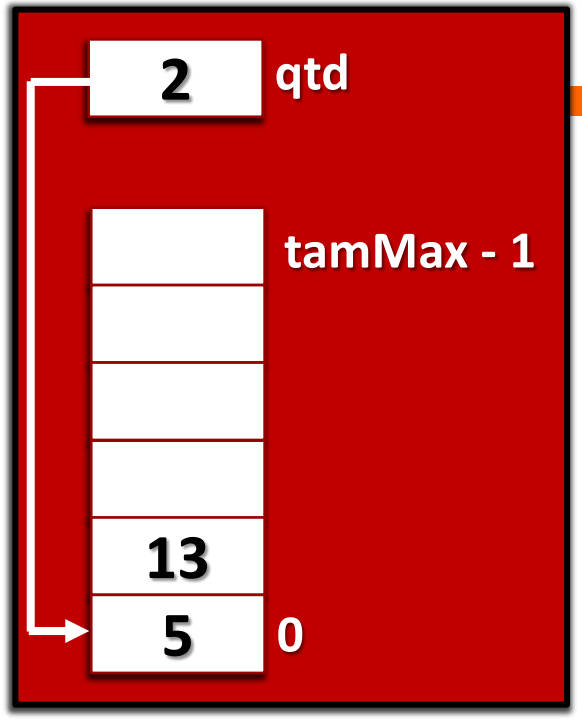


“Destruindo” a Pilha

```
25  void libera_Pilha(Pilha *pi){
26      free(pi);
27  }
```

Liberando a memória alocada para a estrutura que representa a pilha.

Tamanho da Pilha



`tam = pi->qtd;` ➡ Indica o quanto do ***array*** já está ocupado pelos elementos inseridos na pilha.

Para saber o tamanho da pilha, basta retornar o valor de `qtd`.

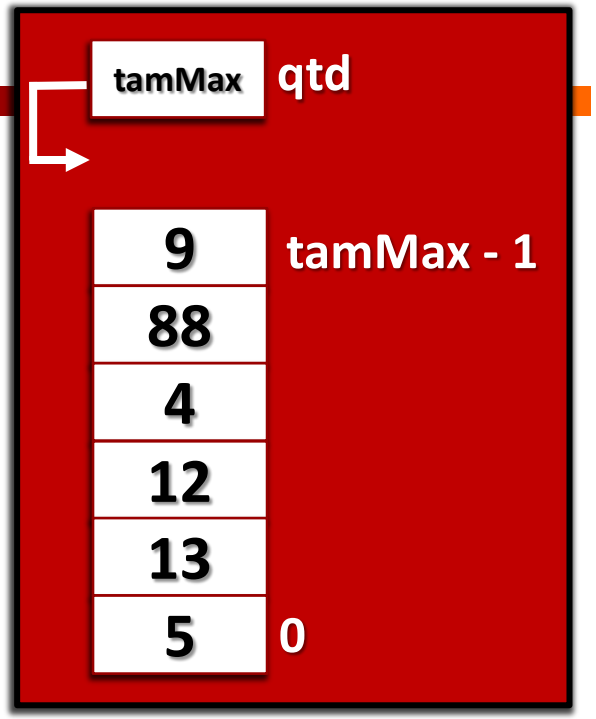
```
29  int tamanho_Pilha(Pilha *pi){
30      int tam = 0;
31
32      if(pi == NULL){
33          return -1;
34      }
35      else{
36          tam = pi->qtd;
37
38          return tam;
39      }
40  }
```

➡ Se for verdade, algum problema aconteceu na criação da pilha.

➡ `tam` recebe o valor de `qtd`, ou seja, o tamanho da pilha.

Pilha Cheia

Pilha Cheia:
pi->qtd == tamMax



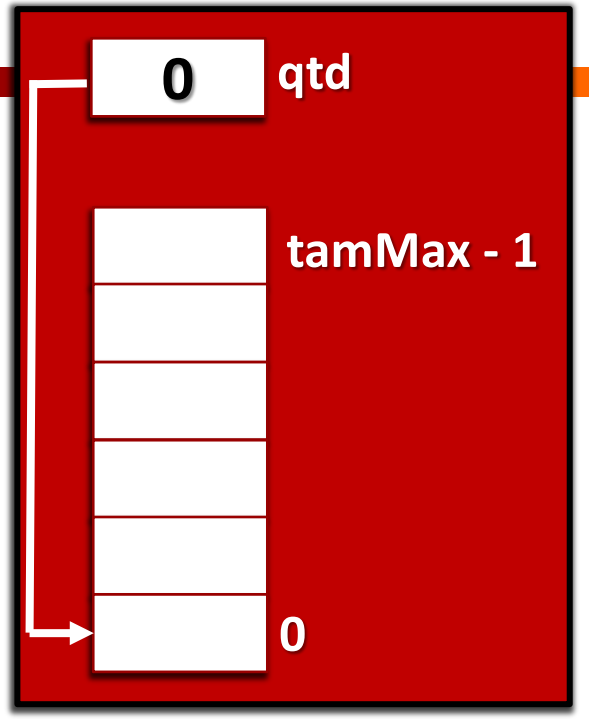

```
42  int pilha_Cheia(Pilha *pi){
43      if(pi == NULL){
44          return -1;
45      }
46      else{
47          if(pi->qtd == tamMax){
48              return 1;
49          }
50          else{
51              return 0;
52          }
53      }
54  }
```

Se ocorreu algum problema na criação da pilha, o valor retornado será -1.

A variável **qtd** também é utilizada para saber se a pilha está cheia. Basta verificar se **qtd =** ao tamanho do *array* (**tamMax**).

Se a pilha não estiver cheia, o valor retornado será 0.

Pilha Vazia



Pilha Vazia: pi->qtd == 0

```
56  int pilha_Vazia(Pilha *pi){
57      if(pi == NULL){
58          return -1;
59      }
60      else{
61          if(pi->qtd == 0){
62              return 1;
63          }
64          else{
65              return 0;
66          }
67      }
68  }
```

Se ocorreu algum problema na criação da pilha, o valor retornado será -1.

A variável **qtd** também é utilizada para saber se a pilha está vazia. Basta verificar se **qtd = 0**.

Se a pilha não estiver vazia, o valor retornado será 0.

Inserindo na Pilha

```
70  int insere_Pilha(Pilha *pi, int num){
71      if(pi == NULL){
72          return 0;
73      }
74      else{
75          if(pi->qtd == tamMax){
76              return 0;
77          }
78          else{
79              pi->num[pi->qtd] = num;
80              pi->qtd++;
81              return 1;
82          }
83      }
84  }
```

Se ocorreu algum problema na criação da pilha, o valor retornado será 0.

Se a pilha estiver cheia, o valor retornado será 0.

Inserindo o elemento (num) no topo da pilha.

Incrementando a quantidade de elementos da pilha.

insere_Pilha(pPilha, 12);

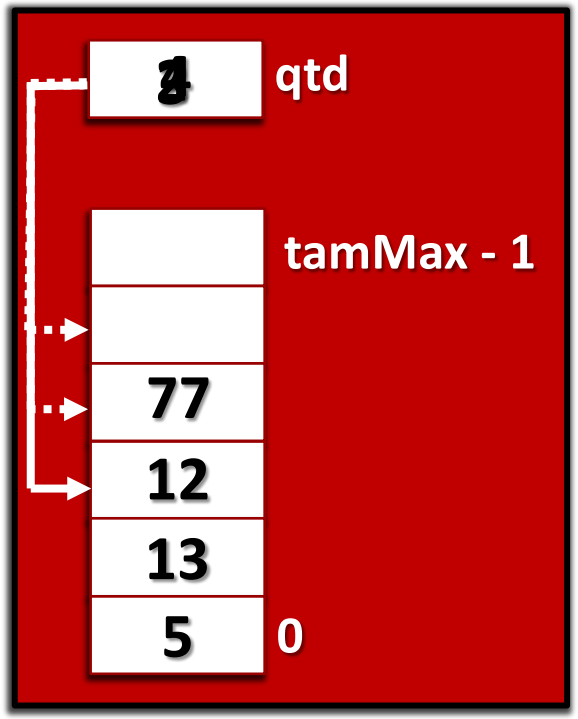
pi->num[2] = 12

pi->qtd = 3

insere_lista(pPilha, 77);

pi->num[3] = 77

pi->qtd = 4



Removendo um Elemento da Pilha

```
86  int remove_Pilha(Pilha *pi){
87      if(pi == NULL || pi->qtd == 0){
88          return 0;
89      }
90      else{
91          pi->qtd--;
92          return 1;
93      }
94  }
```

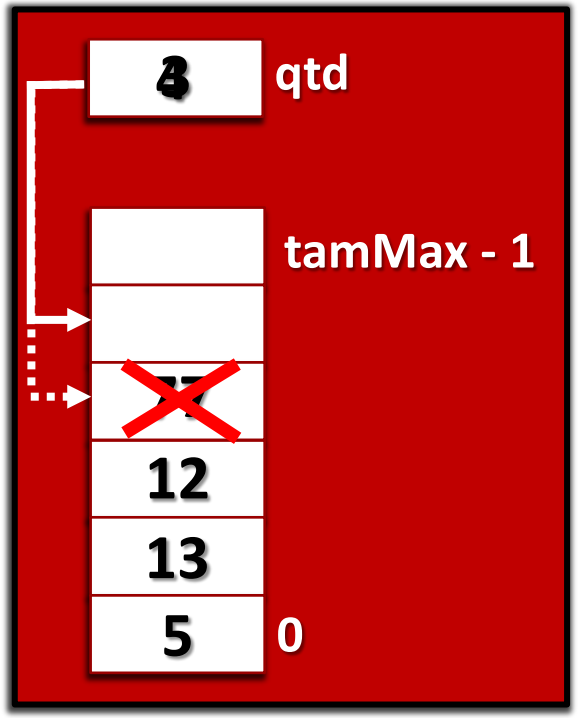
Se ocorreu algum problema na criação da pilha ou se estiver vazia, o valor retornado será 0.

Diminuindo em uma unidade a quantidade de elementos armazenados na pilha.

remove_Pilha(pPilha);

pi->qtd = 3

O topo da pilha fica duplicado. Entretanto, a posição é considerada NÃO ocupada por elementos na pilha.



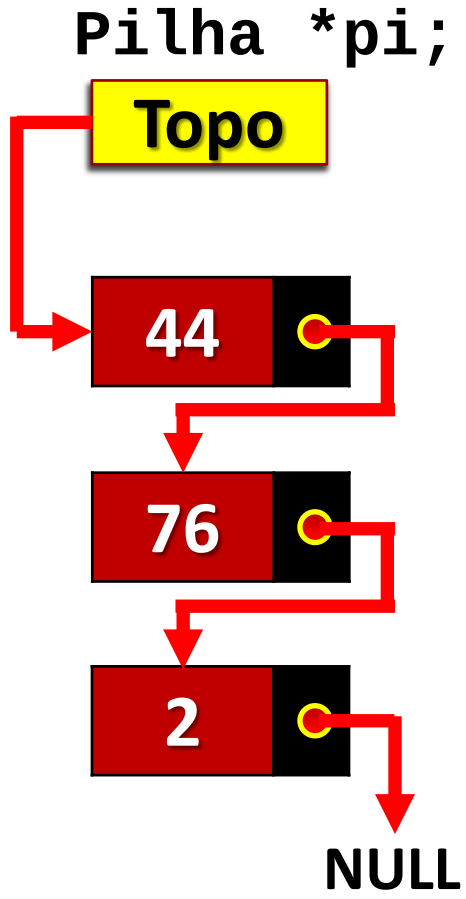
Pilha Dinâmica Encadeada

Pilha Dinâmica Encadeada

- Pilha definida utilizando ALOCAÇÃO DINÂMICA e ACESSO ENCADEADO dos elementos;
- Cada elemento da pilha e ALOCADO DINÂMICAMENTE:
  - À medida que os elementos são inseridos no **TOPO** da pilha;
  - Memória liberada, à medida que são removidos;
  - É um ponteiro para uma estrutura que contém DOIS campos de informação:
    - DADO: utilizado par armazenar a informação inserida na pilha;
    - Prox: ponteiro que indica o próximo elemento na pilha.

Pilha Dinâmica Encadeada

- Além da estrutura que define seus elementos, a pilha utiliza ponteiro para ponteiro para guardar o primeiro elemento ou "TOPO" da pilha:
  - Mesma ideia utilizada na implementação da lista dinâmica encadeada;
  - A diferença é que uma pilha apenas permite um ÚNICO tipo de inserção e remoção:
    - No **TOPO** da pilha.



Definindo o Tipo

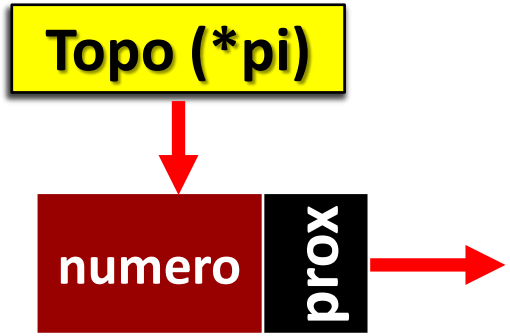
*Ex_04.c

```c
#include<stdio.h>
#include<stdlib.h>

struct elemento{
    int numero;
    struct elemento *prox;
};

typedef struct elemento Elem;

typedef struct elemento *Pilha;
```

Definindo o tipo que descreve cada elemento da pilha.
+ *prox: ponteiro para o próximo elemento da pilha;
+ numero: tipo de dado (int) a ser armazenado na pilha.

Redefinindo a *struct* para encurtar o comando.

Pilha *pi; → **É um ponteiro para Pilha que já é um ponteiro para a struct elemento. Portanto, pi é um ponteiro para ponteiro. Por ser ponteiro para ponteiro, pi armazena o endereço de um ponteiro.**



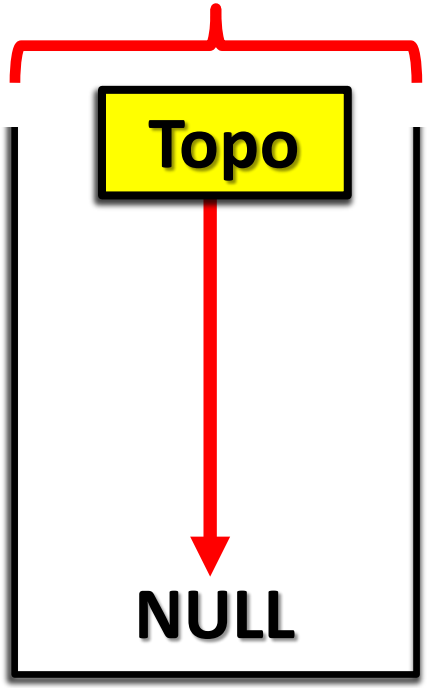
Criando a Pilha

```
13  Pilha* cria_Pilha(){
14      Pilha *pi = NULL;
15
16      pi = (Pilha*) malloc(sizeof(Pilha));
17
18      if(pi != NULL){
19          pi = NULL;
20      }
21
22      return pi;
23  }
```

Alocando uma área de memória para armazenar o endereço do TOPO da pilha.
*pi é um ponteiro para ponteiro

Se o conteúdo de pi é diferente de NULL, a alocação da memória foi realizada.

Pilha *pi;

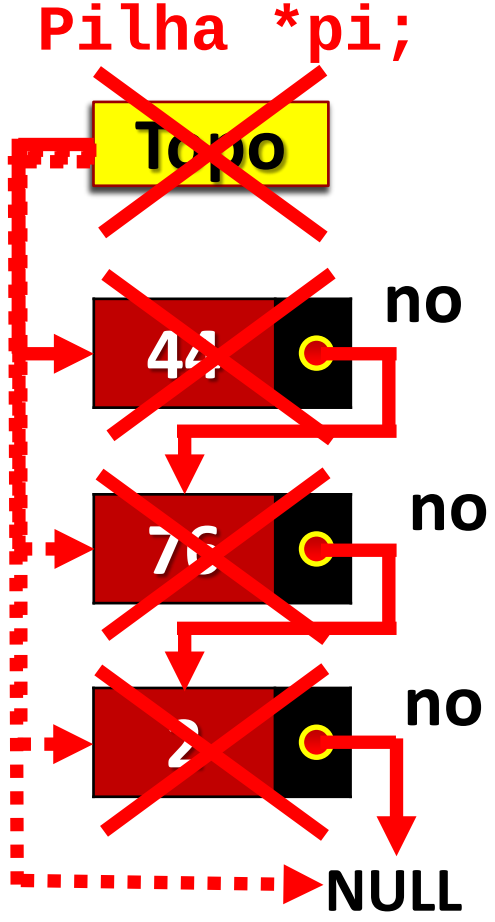


“Destruindo” a Pilha

```
25  void libera_Pilha(Pilha *pi){
26      if(pi != NULL){
27          Elem *no = NULL;
28          while((*pi) != NULL){
29              no = *pi;
30              *pi = (*pi)->prox;
31              free(no);
32          }
33          free(pi);
34      }
35  }
```

Se a pilha foi criada com sucesso.

Cada elemento da pilha é percorrido, enquanto conteúdo do TOPO da pilha for diferente de NULL.



Tamanho da Pilha

Tamanho da Pilha

- Diferente da pilha sequencial estática, para saber o tamanho da pilha dinâmica, é preciso PERCORRER toda a pilha, CONTANDO os elementos inseridos nela, até encontrar o seu final (NULL).

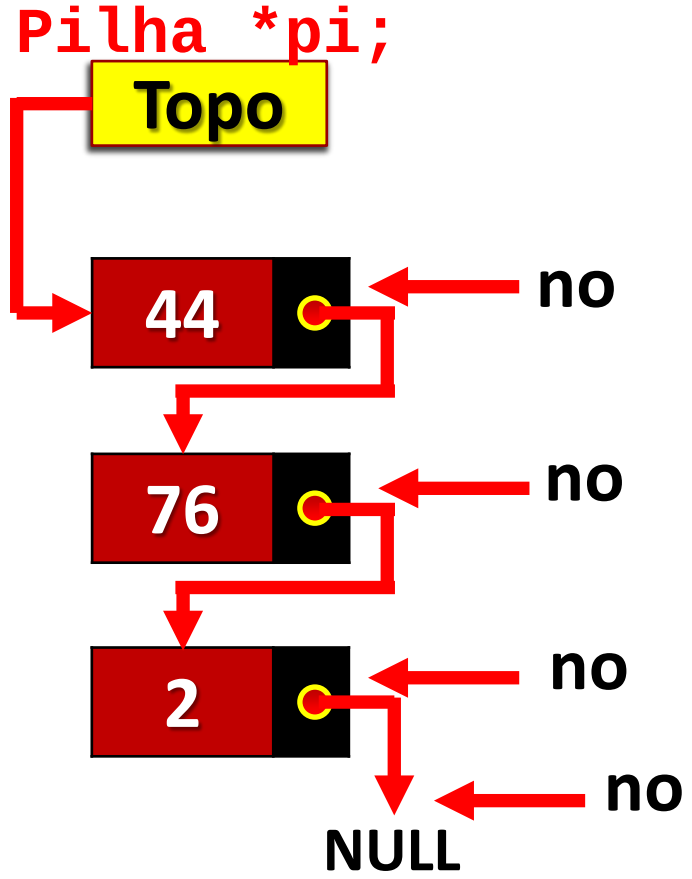
```
37  int tamanho_Pilha(Pilha *pi){
38      int cont = 0;
39      if(pi == NULL){
40          return 0;
41      }
42      else{
43          Elem *no = NULL;
44          no = *pi;
45
46          while(no != NULL){
47              cont++;
48              no = no->prox;
49          }
50          return cont;
51      }
52  }
```

Verifica se a pilha foi criada com sucesso.

Nó aponta para o topo da pilha (primeiro elemento).

Percorre a pilha até o valor do nó for diferente de NULL (fim da pilha).

cont = 3



Pilha Cheia

Pilha Cheia

- Na pilha dinâmica encadeada somente será considerada **CHEIA** quando **NÃO** tiver mais memória disponível para alocar novos elementos:
  - Apenas ocorrerá quando a chamada da função **`malloc()`** retornar **NULL**.

Pilha Vazia

Pilha Vazia

- Uma pilha dinâmica encadeada é considerada **VAZIA** sempre que o conteúdo do "**TOPO**" apontar para a constante **NULL**.

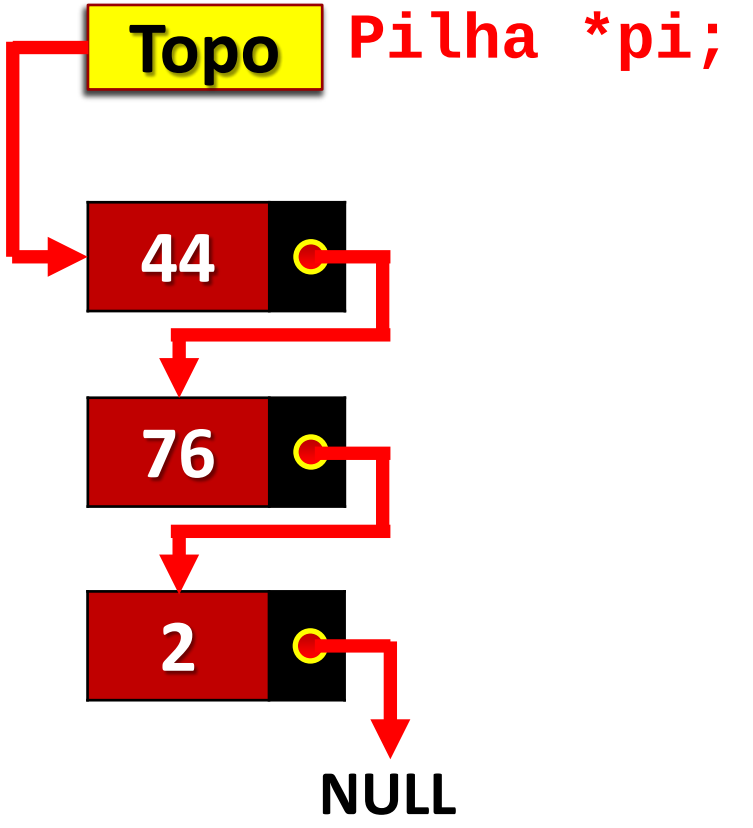
```
54  int pilha_Vazia(Pilha *pi){
55      if(pi == NULL){
56          return -1;
57      }
58      else{
59          if(*pi == NULL){
60              return 1;
61          }
62          else{
63              return 0;
64          }
65      }
66  }
```

Verifica se a pilha foi criada com sucesso.

Acessa o conteúdo do topo (*pi) para comparar com NULL.

Se a pilha estiver vazia.

Pilha não vazia.



Inserindo um Elemento na Pilha

```
68  int insere_Pilha(Pilha *pi, int num){
69      if(pi == NULL){
70          return 0;
71      }
72      else{
73          Elem *no = NULL;
74          no = (Elem*) malloc (sizeof(Elem));
75
76          if(no == NULL){
77              return 0;
78          }
79          else{
80              no->numero = num;
81              no->prox = (*pi);
82              *pi = no;
83              return 1;
84          }
85      }
86  }
```

Verifica se a pilha foi criada com sucesso.

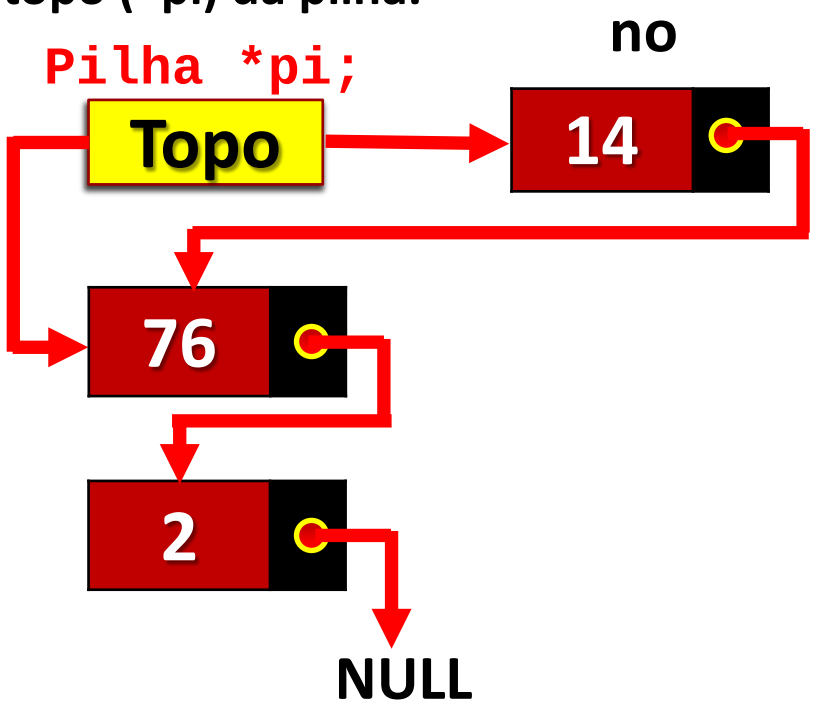
Alocando espaço na memória.

Se não foi possível alocar memória.

Copiando (para o nó) o número recebido como parâmetro.

Nó está apontando para o topo (*pi) da pilha.

O topo (*pi) da pilha aponta para o nó.



Removendo um Elemento da Pilha

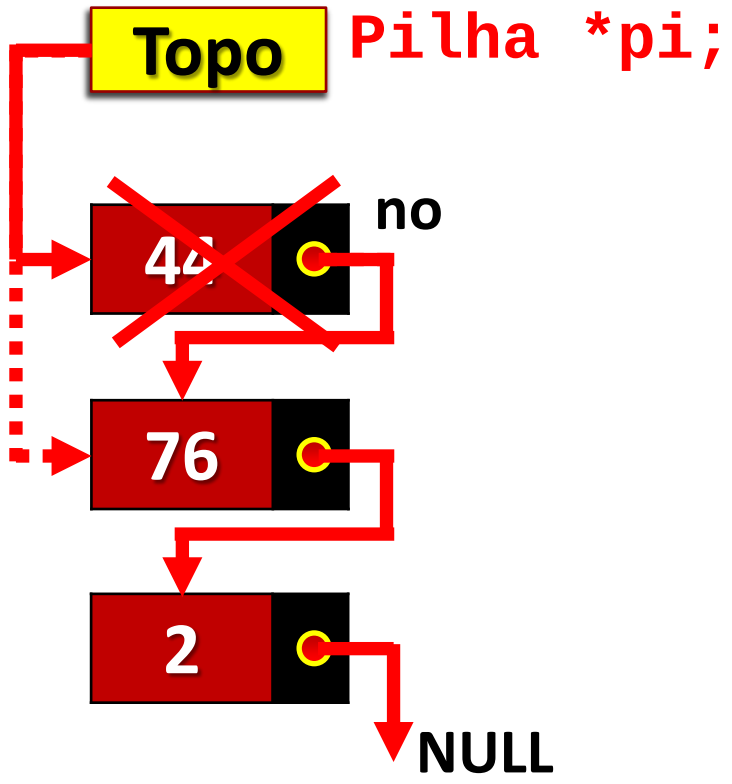
```
88  int remove_Pilha(Pilha *pi){
89      if(pi == NULL){
90          return 0;
91      }
92      else{
93          if((*pi) == NULL){
94              return 0;
95          }
96          else{
97              Elem *no = NULL;
98              no = *pi;
99              *pi = no->prox;
100
101             free(no);
102
103             return 1;
104         }
105     }
106 }
```

Verificar se a remoção é possível.

Ponteiro no aponta para o topo da pilha (pi).

pi está apontando para o próximo elemento da lista.

Liberando o nó.



Referências

- Slides do Prof. Luiz Rozante;
- SALES, André Barros de; AMVAME-NZE, Georges. Linguagem C: roteiro de experimentos para aulas práticas. 2016;
- BACKES, André. Linguagem C Completa e Descomplicada. Editora Campus. 2013;
- SCHILDT, Herbert. C Completo e Total. Makron Books. 1996;
- DAMAS, Luís. Linguagem C. LTC Editora. 1999;
- DEITEL, Paul e DEITEL, Harvey. C Como Programar. Pearson. 2011;
- BACKES, André. Estrutura de Dados Descomplicada em Linguagem C. GEN LTC. 2016.