Pilhas Encadeadas

Prof.: Leonardo Tórtoro Pereira
leonardop@usp.br

Baseado nos slides do Prof. Rudinei Goularte

Métodos de implementação de Pilha

Organização vs. alocação de memória

1. Sequencial e estática: Uso de arrays
2. Encadeada e estática: Array simulando Mem. Princ.
3. Sequencial e dinâmica: Alocação dinâmica de Array
4. Encadeada e dinâmica: Uso de ponteiros

Alocação da memória \ Organização da memória:	Sequencial	Encadeada
Estática	1	2
Dinâmica	3	4

Implementando um TAD de Pilha

TAD Pilhas

- Operações auxiliares
  - criar(P): cria uma pilha P vazia
  - apagar(P): apaga a pilha P da memória
  - topo(P): retorna o elemento do topo de P, sem remover
  - tamanho(P): retorna o número de elementos em P
  - vazia(P): indica se a pilha P está vazia
  - cheia(P): indica se a pilha P está cheia (útil para implementações sequenciais).

Definição da Interface das Operações

- PILHA *pilha_criar(void);
- void pilha_apagar(PILHA **pilha);
- int pilha_vazia(PILHA *pilha);
- int pilha_cheia(PILHA *pilha);
- int pilha_tamanho(PILHA *pilha);
- ITEM *pilha_topo(PILHA *pilha);
- int pilha_empilhar(PILHA *pilha, ITEM *item);
- ITEM *pilha_desempilhar(PILHA *pilha);
- void pilha_print(PILHA *p);
- void pilha_inverter(PILHA *p);

Aplicação de Pilhas

- Avaliação de expressões aritméticas
  - Notação infixa é ambígua
    - A + B * C =?
    - Necessidade de precedência de operadores ou utilização de parênteses
- Entretanto existem outras notações...

Aplicação de Pilhas

- Notação polonesa (prefixa)
  - Operadores precedem os operandos
  - Dispensa o uso de parênteses
  - - * AB/CD = (A * B) - (C/D)
- Notação polonesa reversa (posfixa)
  - Operadores sucedem os operandos
  - Dispensa o uso de parênteses
  - AB * CD/- = (A * B) - (C/D)

Aplicação de Pilhas

- Expressões na notação posfixa podem ser avaliadas utilizando uma pilha
  - A expressão é avaliada de esquerda para a direita
  - Os operandos são empilhados
  - Os operadores fazem com que: dois operandos sejam desempilhados, o cálculo seja realizado e o resultado empilhado

Aplicação de Pilhas

➔ Por exemplo: 6 2 / 3 4 * + 3 - = 6 / 2 + 3 * 4 - 3

Símbolo	Ação	Pilha
6	empilhar	P[6]
2	empilhar	P[2, 6]
/	desempilhar, aplicar operador e empilhar	P[(6/2)] = P[3]
3	empilhar	P[3, 3]
4	empilhar	P[4, 3, 3]
*	desempilhar, aplicar operador e empilhar	P[(3*4), 3] = P[12, 3]
+	desempilhar, aplicar operador e empilhar	P[3 + 12] = P[15]
3	empilhar	P[3, 15]
-	desempilhar, aplicar operador e empilhar	P[(15 – 3)] = P [12]
	final, resultado no topo da pilha	P[12]

Aplicação de Pilhas

Considere o problema de decidir se uma dada sequência de parênteses e chaves é bem formada. Por exemplo, a sequência abaixo:

(( ) { ( ) })

é bem-formada, enquanto a sequência

({) }

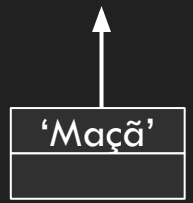
é malformada.

Aplicação de Pilhas

Suponha que a sequência de parênteses e chaves está armazenada em uma cadeia de caracteres s . Escreva uma função bem_formada() que receba a cadeia de caracteres s e devolva 1 se s contém uma sequência bem-formada de parênteses e chaves e devolva 0 se a sequência está malformada.

Implementação Dinâmica Encadeada

- Ponteiros podem ser usados para construir estruturas, tais como Pilhas, a partir de componentes simples chamados nós

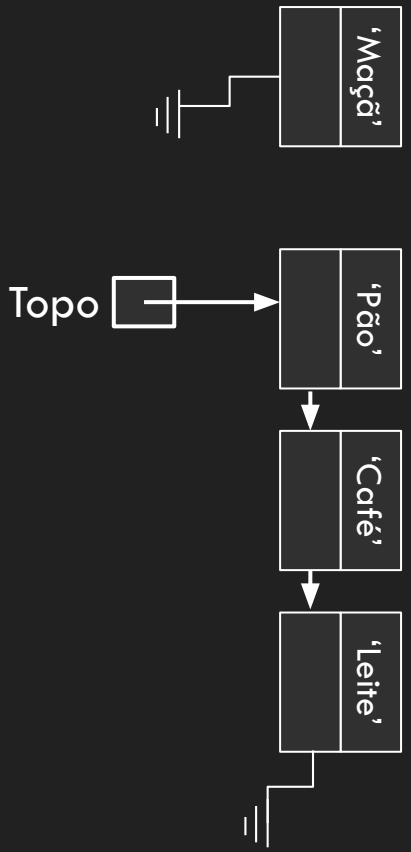


Implementação Dinâmica Encadeada

- Encadeamentos são úteis pois podem ser utilizadas para implementar o TAD pilha
- Uma vantagem é o fato de não ser necessário informar o número de elementos em tempo de compilação

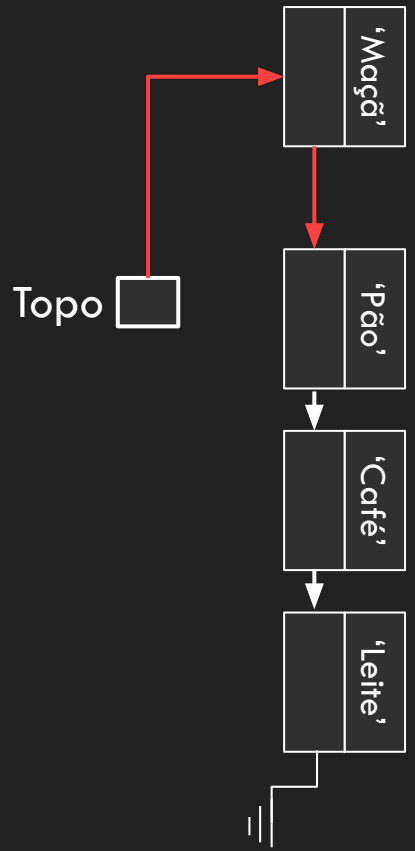
Discussão Intuitiva

- Por exemplo, uma operação de **empilhar** pode ser feita da seguinte maneira



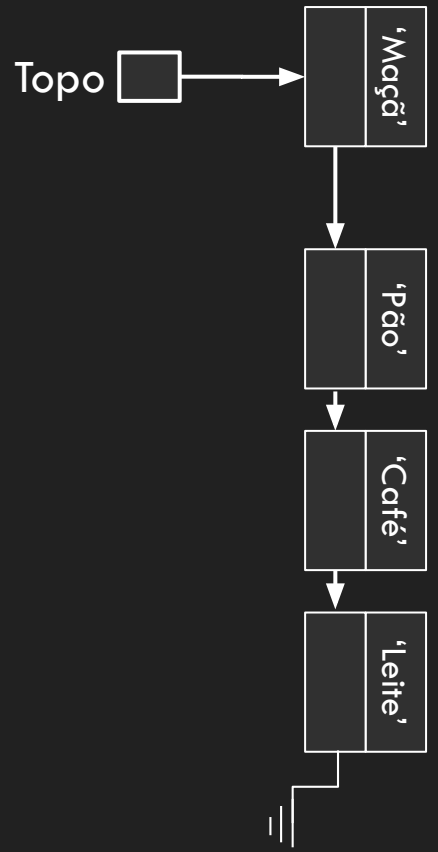
Discussão Intuitiva

- Por exemplo, uma operação de **empilhar** pode ser feita da seguinte maneira



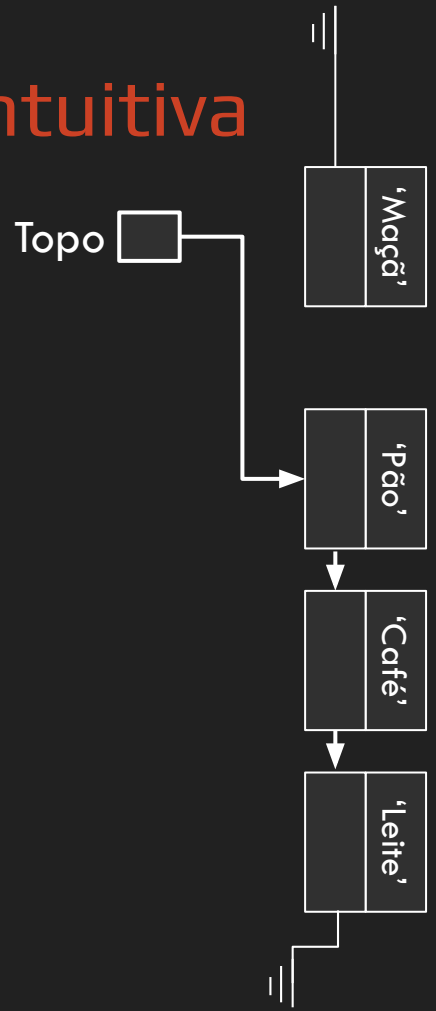
Discussão Intuitiva

- Por exemplo, uma operação de **desempilhar** pode ser feita da seguinte maneira



Discussão Intuitiva

- Por exemplo, uma operação de **desempilhar** pode ser feita da seguinte maneira



Implementação Dinâmica Encadeada

- O topo é o início
- * Pode-se implementar as operações utilizando a abordagem de lista ligada simples
- Utilizando o TAD Pilha, a interface e o programa cliente não mudam, apenas a implementação (Pilha.c) muda!!!

  - * Problema do ovo e da galinha! Optamos por estudar Pilhas primeiro.

Definição de Tipos

```
// interface (arquivo pilha.h)
1 typedef struct pilha PILHA;
2 #define TAM 100
3 #define ERRO -32000
...
// implementação (arquivo pilha.c)
2 #include "Pilha.h"
3 typedef struct no NO;
4 struct no {
5     ITEM* item;
6     NO* anterior;
7 };
9  struct pilha {
10   NO* topo;
11   int tamanho;
12 };
...
```

Implementação Dinâmica Encadeada

```
PILHA* pilha_criar() {
   PILHA* pilha = (PILHA *) malloc(sizeof (PILHA));
   if (pilha != NULL) {
     pilha->topo = NULL;
     pilha->tamanho = 0;
   }
   return (pilha);
}
```

Implementação Dinâmica Encadeada

```
void pilha_apagar(PILHA** pilha) {
    NO* paux;
    if ( ((*pilha) != NULL) && (!pilha_vazia(*pilha)) ) {

        while (pilha->topo != NULL) {
            paux = (*pilha)->topo;
            (*pilha)->topo = (*pilha)->topo->anterior;
            item_apagar(&paux->item);
            paux->anterior = NULL;
            free(paux); paux = NULL;
        }
    }
    free(*pilha);
    *pilha = NULL;
}
```

Implementação Dinâmica Encadeada

```
1 int pilha_vazia(PILHA* pilha) {
2    return ((pilha != NULL) ? pilha->tamanho == 0 : ERRO);
3 }
4
5 int pilha_tamanho(PILHA* pilha) {
6    return ((pilha != NULL) ? pilha->tamanho : ERRO);
7 }
8
9 ITEM* pilha_topo(PILHA* pilha) {
10   if ((pilha != NULL) && (!pilha_vazia(pilha)) ){
11      return (pilha->topo->item);
12   }
13   return (NULL);
14 }
```

Implementação Dinâmica Encadeada

```
int pilha_empilhar(PILHA* pilha, ITEM* item) {
    NO* pnovo = (aNO *) malloc(sizeof (NO));
    if (pnovo != NULL) {
        pnovo->item = item;
        pnovo->anterior = pilha->topo;
        pilha->topo = pnovo;
        pilha->tamanho++;
        return (1);
    }
    return (ERRO);
}
```

Implementação Dinâmica Encadeada

```
ITEM* pilha_desempilhar(PILHA* pilha) {
    if ((pilha != NULL) && (!pilha_vazia(pilha)) ){
        NO* pno = pilha->topo;
        ITEM* item = pilha->topo->item;
        pilha->topo = pilha->topo->anterior;
        pno->anterior=NULL;
        free(pno);
        pno=NULL;
        pilha->tamanho--;
        return (item);
    }
    return (NULL);
}
```

Sequencial versus Encadeada

Operação	Sequencial	Encadeada
Criar	O(1)	O(1)
Apagar	O(n)*	O(n)
Empilhar	O(1)	O(1)
Desempilhar	O(1)	O(1)
Topo	O(1)	O(1)
Vazia	O(1)	O(1)
Tamanho	O(1)	O(1) (com contador)

* Do modo como foi implementado, o TAD pilha é um array de ponteiros para TADs ITEM. Isto é, cada posição do array aponta para um item, o qual, por sua vez, foi alocado dinamicamente. Assim, ao apagar a pilha é necessário percorrer o array usando as referências (ponteiros) para desalocar (free) os itens!! => O(n).

Sequencial versus Encadeada

- Sequencial
  - Implementação simples
  - Tamanho da pilha definido *a priori*
- Encadeada
  - Alocação dinâmica permite gerenciar melhor estruturas cujo tamanho não é conhecido *a priori* ou que variam muito de tamanho

Referências

- ZIVIANI, N. Projeto de Algoritmos, Thomson, 2a. Edição, 2004.