SCC0202 – Algoritmos e Estruturas de Dados I

Pilhas

Prof.: Dr. Rudinei Goularte

(rudinei@icmc.usp.br)

Instituto de Ciências Matemáticas e de Computação - ICMC

Sala 4-229

Conteúdo

- Conceitos Introdutórios
- Implementação Sequencial
- Implementação Encadeada
- Aplicações com Pilha

Exemplo prático

- Faça um algoritmo para converter um número decimal em sua respectiva representação binária.

Pilhas

- O que são?
  - Exemplos do mundo real...
- Para que servem?
  - Auxiliam em problemas práticos em computação

O quê são?

Exemplos do mundo real

Para que servem?

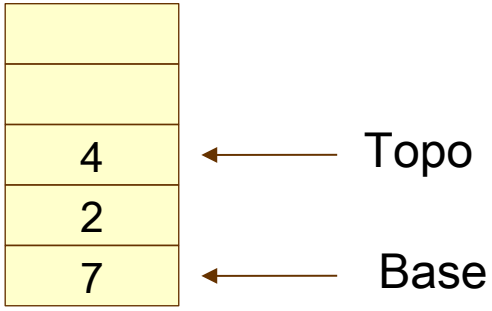
- Auxiliam em problemas práticos em computação
  - Alguns exemplos a seguir...

- Exemplos de aplicações de pilhas
  - O botão “back” de um navegador web ou a opção “undo” de um editor de textos
  - Controle de chamada de procedimentos
  - Estrutura de dados auxiliar em alguns algoritmos como a busca em profundidade

    - Veremos mais exemplos de aplicações de pilhas mais adiante.

Pilhas - definição

- Pilhas são estruturas de dados nas quais as inserções e remoções são realizadas na mesma extremidade da estrutura, chamada topo. Dessa maneira o último elemento que foi inserido é sempre o primeiro a ser removido.
  - Política Last-In/First-Out (LIFO)



Organização vs. alocação de memória

- Alocação Estática: reserva de memória em tempo de compilação
- Alocação Dinâmica: em tempo de execução

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

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

TAD Pilhas

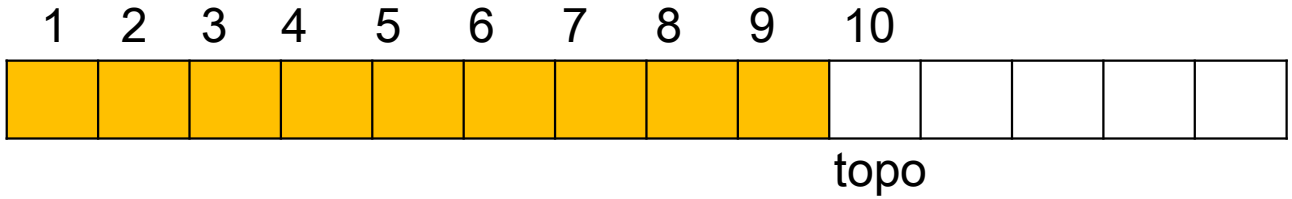
- Operações principais
  - empilhar(P,x): insere o elemento x no topo de P
  - desempilhar(P): remove o elemento do topo de P, e retorna esse elemento

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).

Implementação Sequencial

- Implementação simples
- Uma variável mantém o controle da posição do topo, e pode ser utilizada também para informar o número de elementos na pilha (tamanho)



Definição da Interface das Operações

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

- void pilha_print(PILHA *p);
- void pilha_inverter(PILHA *p);

Definição de Tipos

```
/* interface (arquivo .h). Ex.: pilha.h*/
1  #include “item.h”
2  #include <stdbool.h>
3  typedef struct pilha PILHA;
4  #define TAM 100
5
...

/* implementação (arquivo .c). Ex.: pilha.c*/
1  struct pilha {
2     ITEM *item[TAM];
3      int tamanho; /*topo da pilha*/
4  };
5 ...
/* programa cliente (arquivo .c). Ex.: decimal_binario.c*/
1  #include “pilha.h”
2 ...
30 PILHA *p;
31...
```

Implementação Sequencial

```
1 PILHA* pilha_criar(void) {
2
3    PILHA* pilha = (PILHA *) malloc(sizeof (PILHA));
4
5    if (pilha != NULL)
6       pilha->tamanho = 0;
7
8    return (pilha);
9 }
10
11 bool pilha_vazia(PILHA* pilha) {
12      if (pilha != NULL)
13         return ((pilha->tamanho == 0) ? true : false);
14      return(false);
15 }
16
17 bool pilha_cheia(PILHA *pilha) {
18    if (pilha != NULL)
19         return ((pilha->tamanho == TAM) ? true : false);
20    return(false);
21 }
22
23 int pilha_tamanho(PILHA *pilha) {
24   return ((pilha != NULL) ? pilha->tamanho: ERRO);
25 }
```

Implementação Sequencial

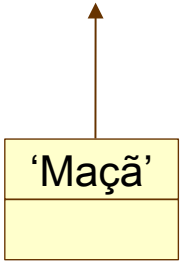
```
1 bool pilha_empilhar(PILHA *pilha, ITEM* item) {
2    if ((pilha!=NULL) && (!pilha_cheia(pilha)) {
3      pilha->item[pilha->tamanho] = item;
4      pilha->tamanho++;
5      return (true);
6    }
7    return (false);
8 }
9
10 ITEM* pilha_desempilhar(PILHA* pilha) {
11   ITEM* i;
12   if ((pilha != NULL) && (!pilha_vazia(pilha))) {
13      i = pilha_topo(pilha);
14      pilha->item[pilha->tamanho-1] = NULL;
15      pilha->tamanho--;
16      return (i);
17   }
18   return (NULL);
19 }
```

Exemplo prático

- Implemente um programa em C para converter um número decimal em sua respectiva representação binária usando o TAD pilha (sequencial).

Implementação Dinâmica Encadeada

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

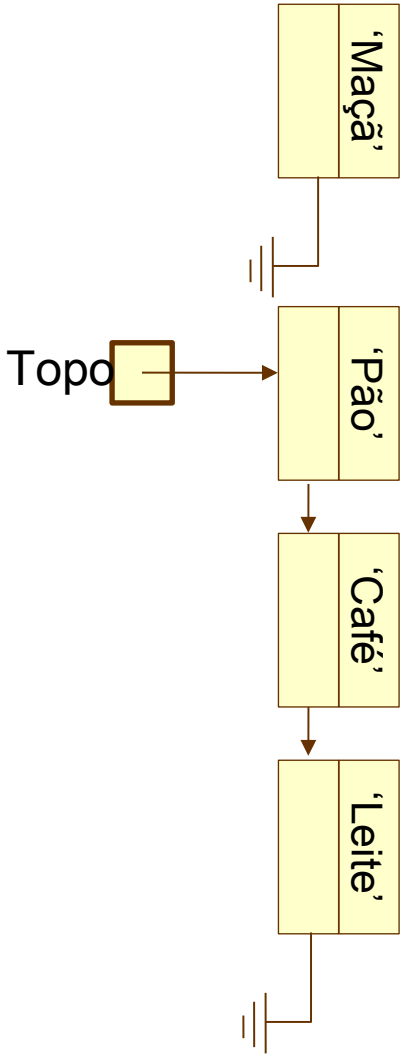


Implementação Dinâmica Encadeada

- 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

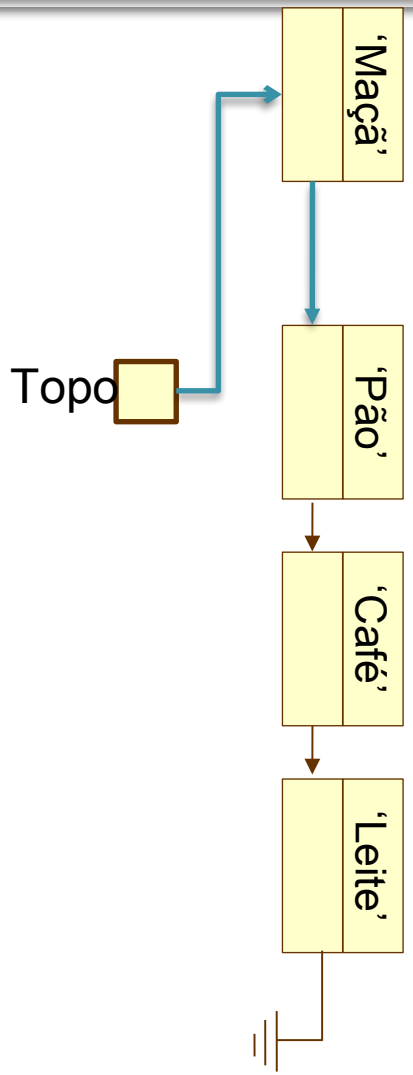
Discussão Intuitiva

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



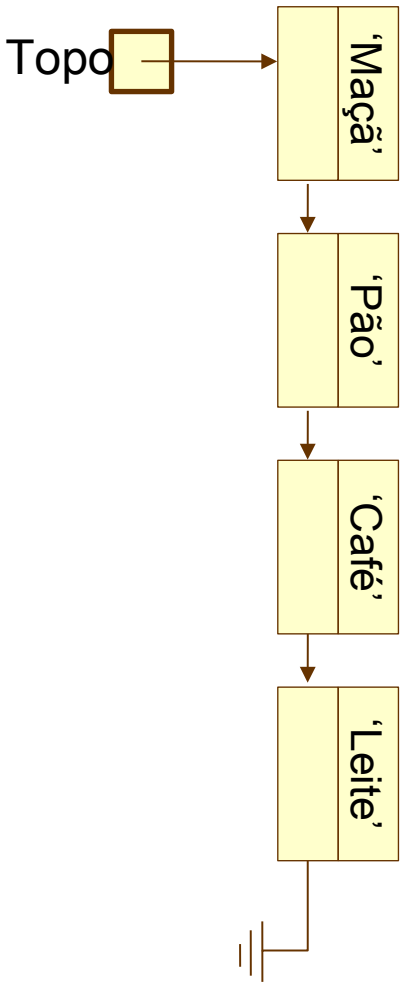
Discussão Intuitiva

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



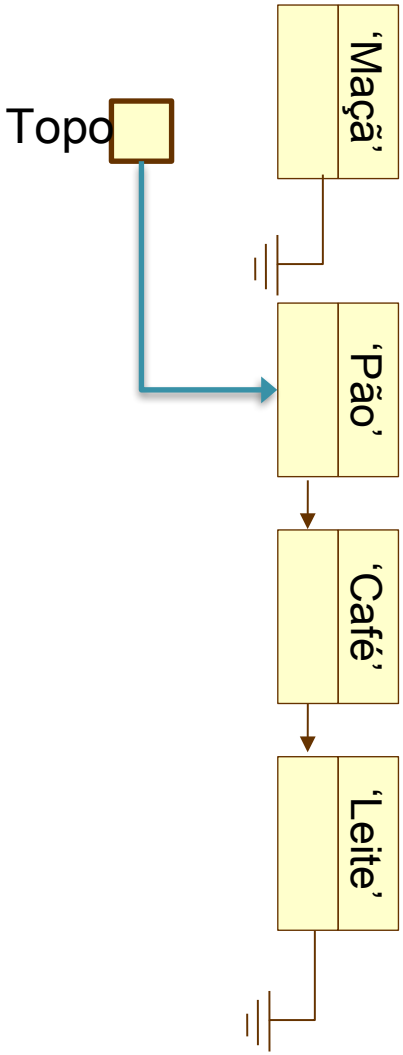
Discussão Intuitiva

- 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  #include “item.h”
2  #include <stdbool.h>
3  typedef struct pilha PILHA;
4  #define TAM 100
...

// implementação (arquivo pilha.c)
1 #include <stdio.h> …
2 #include "Pilha.h"
3 typedef struct no NO;
4 struct no {
5      ITEM* item;
6      NO* anterior;
7 };
8
9  struct pilha {
10   NO* topo;
11   int tamanho;
12 };
...
```

Implementação Dinâmica Encadeada

```
1 PILHA* pilha_criar() {
2    PILHA* pilha = (PILHA *) malloc(sizeof (PILHA));
3    if (pilha != NULL) {
4      pilha->topo = NULL;
5      pilha->tamanho = 0;
6    }
7    return (pilha);
8 }
9
10 void pilha_apagar(PILHA** pilha) {
11   NO* paux;
12   if ( ((*pilha) != NULL) && (!pilha_vazia(*pilha)) ) {
13
14      while (pilha->topo != NULL) {
15         paux = (*pilha)->topo;
16         (*pilha)->topo = (*pilha)->topo->anterior;
17         item_apagar(&paux->item);
18         paux->anterior = NULL;
19         free(paux); paux = NULL;
20      }
21   }
22   free(*pilha);
23   *pilha = NULL;
24 }
```

Implementação Dinâmica Encadeada

```
1 bool pilha_vazia(PILHA* pilha) {
2      if (pilha != NULL)
3         return ((pilha->tamanho == 0) ? true : false);
4      return(false);
5 }
6
7 bool pilha_cheia(PILHA *pilha) {
8    if (pilha != NULL)
9         NO *novo = (NO*) malloc(sizeof(NO));
10        if(novo != NULL){
11          free(novo);
12          return(false);
13        }
14       return(true);
15   }
16}
17 int pilha_tamanho(PILHA* pilha) {
18    return ((pilha != NULL) ? pilha->tamanho : ERRO);
19 }
20
21 ITEM* pilha_topo(PILHA* pilha) {
22   if ((pilha != NULL) && (!pilha_vazia(pilha)) ){
23      return (pilha->topo->item);
24   }
25   return (NULL);
26 }
```

Implementação Dinâmica Encadeada

```
1 bool pilha_empilhar(PILHA* pilha, ITEM* item) {
2    NO* pnovo = (NO *) malloc(sizeof (NO));
3    if (pnovo != NULL) {
4      pnovo->item = item;
5      pnovo->anterior = pilha->topo;
6      pilha->topo = pnovo;
7      pilha->tamanho++;
8      return (TRUE);
9    }
10   return (FALSE);
11 }
12
13 ITEM* pilha_desempilhar(PILHA* pilha) {
14   if ((pilha != NULL) && (!pilha_vazia(pilha)) ){
15     NO* pno = pilha->topo; ITEM* item = pilha->topo->item;
16     pilha->topo = pilha->topo->anterior;
17     pno->anterior=NULL; free(pno); pno=NULL;
18     pilha->tamanho--;
19     return (item);
20   }
21   return (NULL);
22 }
```

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

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]

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.

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.

Referências

- Material baseado nos originais produzidos pelos professores:
  - Gustavo E. de A. P. A. Batista
  - Fernando V. Paulovich
  - Maria das Graças Volpe Nunes