SCC 202 - Algoritmos e Estruturas de Dados I

TAD Pilha

Sequencial Estática e Encadeada Dinâmica

17 e 19/8/2010

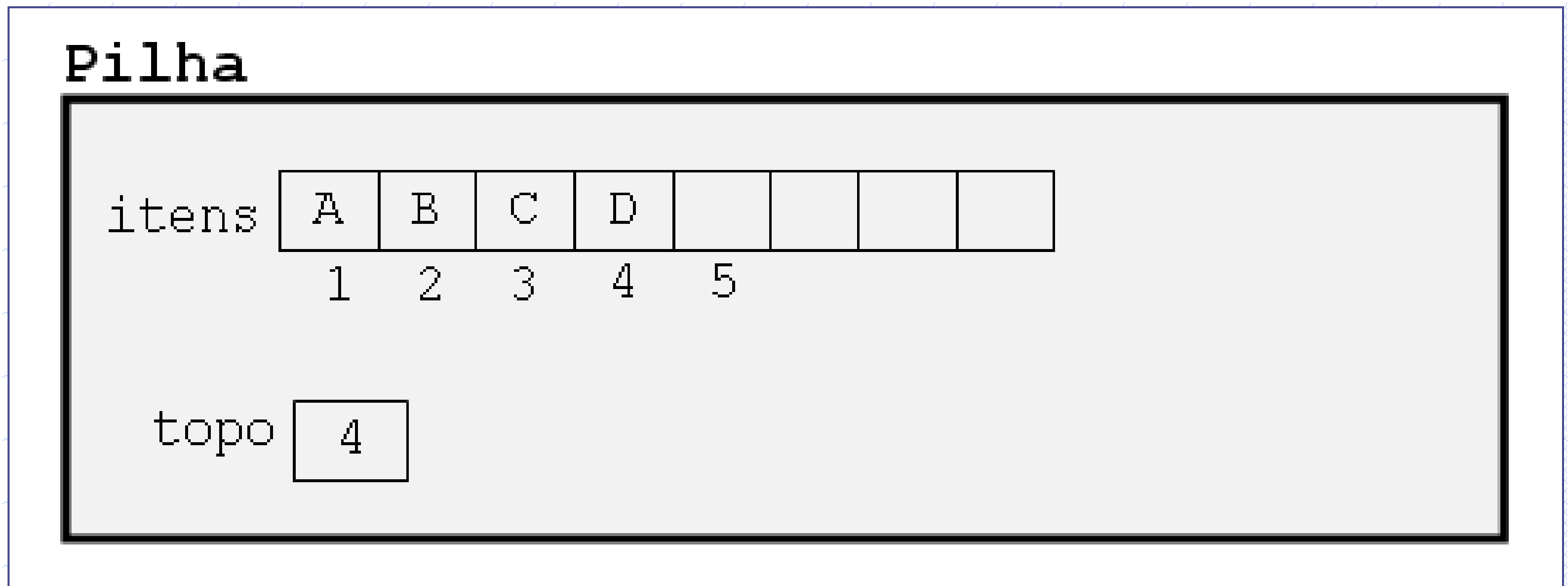
Exercício: Implementação da pilha sequencial e estática

- Declaração em C escondendo a ED do cliente ??

- Arrays são implementados internamente como ponteiros, em C.
  - Mas a idéia de chamar esta implementação de estática é que o tamanho da pilha é definido previamente
    - e não pode ser alterado nesta implementação.

REP/IMP da pilha

- Seqüencial e estática



Implementação da pilha

- Declaração em C

Usuário pode
Alterar TamPilha
e elem

```
#define TamPilha 100
typedef char elem;
typedef struct pilha Pilha;
```

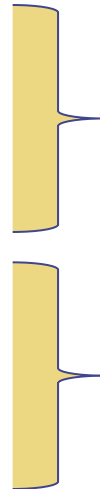
Pilha.h

```
struct pilha{
            int topo;
            elem itens[TamPilha];
};
```

Pilha.c

Exercício

- Implementar operações da pilha
  - Create
  - Empty
  - Destroy
  - Push
  - Pop
  - Top (FAZER)
  - IsEmpty
  - IsFull
  - Size



São contantes, O(1)

Auxiliares para checar erros e gerência

- Atenção: considerações sobre TAD
  - Arquivos .c e .h, parâmetros, mensagens de erro, comentários

```c
#define TamPilha 100
typedef char elem;
typedef struct pilha Pilha;
```

Em ANSI C, uma estrutura que não foi ainda definida é chamada de tipo incompleto

```c
/* cria uma pilha vazia P. Deve ser chamada antes da pilha ser
   usada */
Pilha* Create(int *flagErro);


/* esvazia uma pilha P para poder ser reusada. Retorna erro se
   Pilha não existe */
void Empty(Pilha* P, int *flagErro);


/* remove a pilha criada da memória. Retorna erro se Pilha não
   existe  */
void Destroy(Pilha *P,  int *flagErro);
```

Pilha.h

```
/* empilha o elemento X na pilha P. Se P estiver cheia
   erro = 1 e se a operação tiver sucesso, erro = 0 */
void Push(Pilha* P, elem X, int* erro);

/* desempilha P e retorna em X o valor do elemento
   que estava no topo de P. Se P estiver vazia erro = 1
   e se a operação tiver sucesso, erro = 0 */
void Pop(Pilha* P, elem* X, int* erro);

/* acessa o valor do elemento do topo de P, sem
   desempilhar.  */
void Top(Pilha* P, elem* X, int* erro);
```

```
/* retorna o número de elementos de P */
int Size(Pilha* P);

/* retorna true se a pilha estiver vazia; false caso contrário */
int IsEmpty(Pilha* P);

/* retorna true se a pilha estiver cheia; false caso contrário */
int IsFull(Pilha* P);
```

OBS:

a) Uniformizei as funções Destroy e Empty para retornarem erro se Pilha não existe.
b) O parâmetro elemento em Push é passado por valor agora

```c
#include "pilha.h"
#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */

struct pilha{
                    int topo;
                    elem itens[TamPilha];
};

Pilha* Create(int *flagErro) {
    Pilha *P = (Pilha*)malloc(sizeof(Pilha));
    if (P == NULL) {
          *flagErro = 1; // ERRO_MEMORIA_INSUFICIENTE
       return P;
    }
    else
    *flagErro = 0; // SUCESSO
    P->topo=-1;
    return P;
}
```

O tipo incompleto se torna completo na implementação, quando ele é definido

Pilha.c

```
void Empty(Pilha *P , int *flagErro) {
   if(P != NULL){
      P->topo=-1;
      *flagErro = 0; //SUCESSO
   }
   else
      *flagErro = 1; // ERRO_PONTEIRO_NULO

}

void Destroy(Pilha *P, int *flagErro) {

  if(P != NULL){
       free(P);
       *flagErro = 0; //SUCESSO
   }
   else
       *flagErro = 1; // ERRO_PONTEIRO_NULO
}
```

```
void Push(Pilha *P, elem X, int *erro) {
    if (!IsFull(P)) {
        *erro=0;
        P->topo++;
        P->itens[P->topo]=X;
    }
    else
    *erro=1;
}
```

```
void Pop(Pilha *P, elem *X, int *erro) {
    if (!IsEmpty(P)) {
        *erro=0;
        *X=P->itens[P->topo];
        P->topo--;
    }
    else
    *erro=1;
}
```

Poderia ser feita das operações já implementadas?

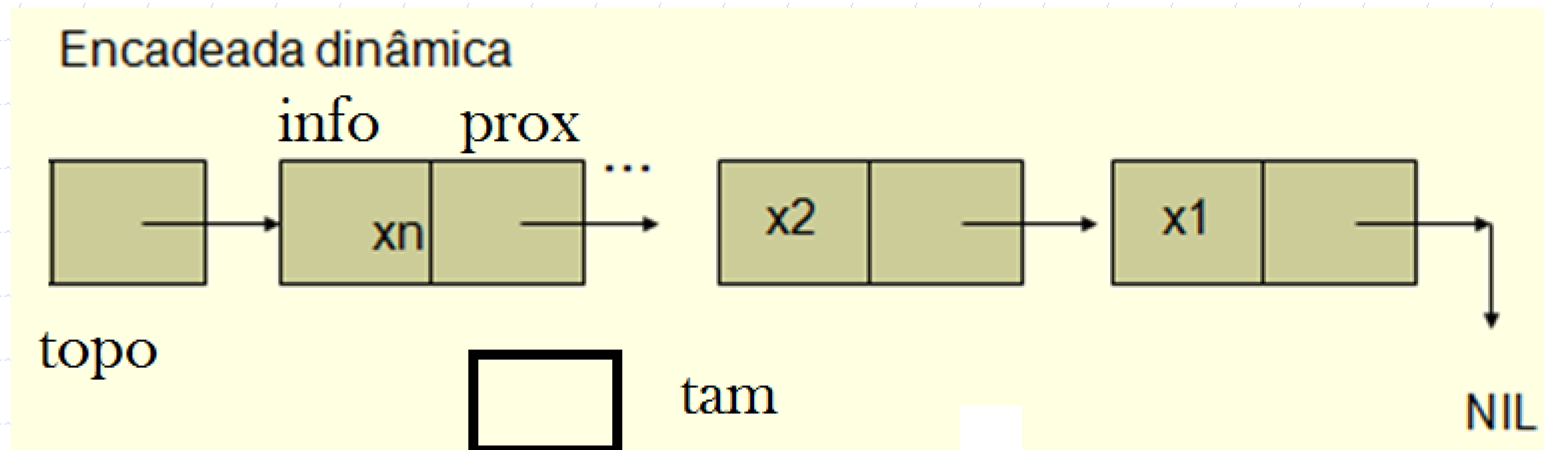
```
void Top(Pilha* P, elem* X, int* erro) {
    if (!IsEmpty(P)) {
        *erro=0;
        *X=P->itens[P->topo];
    }
    else
    *erro=1;
}
```

```
int Size(Pilha *P) {
return P->topo + 1;
}
int IsEmpty(Pilha *P) {
    if (P->topo==-1)
        return 1;
    else return 0;
}
int IsFull(Pilha *P) {
    if (P->topo==TamPilha-1)
        return 1;
    else return 0;
}
```

OBS: Troquei exit em Create para return P no caso do Ponteiro da pilha ser nulo

Exercício: Implementação da pilha encadeada dinâmica

- REP/IMP



```
typedef char elem;
typedef struct pilha Pilha;
```

Pilha.h

Não foi alterado

```
typedef struct bloco {
        elem info;
        struct bloco *prox;
}no;

struct pilha {
    no *topo;
    int tam;
};
```

Pilha.c

```c
#include "pilha.h"
#include <stdlib.h> /* malloc, free, exit */
#include <stdio.h> /* printf */


typedef struct bloco {
        elem info;
        struct bloco *prox;
}no;

struct pilha {
    no *topo;
    int tam;
};
```

Pilha.c

```
Pilha* Create(int *flagErro) {
    Pilha *P = (Pilha*)malloc(sizeof(Pilha));

    if (P == NULL) {
        *flagErro = 1; // ERRO_MEMORIA_INSUFICIENTE
        return P;
    }
    else
    *flagErro = 0; // SUCESSO
    P->topo=NULL;
    P->tam=0;
    return P;
}
```

```
// libera todos os elementos da lista e depois libera a pilha
void Destroy(Pilha *P,  int *flagErro){
   no *t,*q;

   if(P != NULL){
      q = P->topo;
      while(q!=NULL){
        t = q->prox;
        free(q);
        q = t;
      }
      free(P);
      *flagErro = 0; //SUCESSO
   }
   else
   *flagErro = 1; // ERRO_PONTEIRO_NULO
}
```

```
// libera todos os elementos da lista MAS não libera a pilha
void Empty(Pilha *P ,  int *flagErro) {
     no *t,*q;
   if(P != NULL){
   q = P->topo;
   while(q!=NULL){
        t = q->prox;
        free(q);
        q = t;
    }

    P->tam=0;
    *flagErro = 0; //SUCESSO
 }
   else
   *flagErro = 1; // ERRO_PONTEIRO_NULO
}
```

```
void Push(Pilha *P, elem X, int *erro) {
    no *pont;

    pont=(no*) malloc(sizeof(no));
    if (pont==NULL)  // Está cheia, pois não há mais espaço de memória
        *erro=1;
    else {
        *erro=0;
        pont->info=X;
        pont->prox=P->topo;
        P->topo=pont;
        P->tam = P->tam + 1;
    }
}
```

```
void Pop(Pilha *P, elem *X, int *erro) {
    no *pont;

    if (IsEmpty(P))
      *erro=1;
    else {
        *erro=0;
        pont=P->topo;
        *X=pont->info;
        P->topo=P->topo->prox;
        free(pont);
        P->tam = P->tam  - 1;
    }
}
```

```
void Top(Pilha* P, elem* X, int* erro){

    if (!IsEmpty(P)) {
        *erro=0;
        *X=P->topo->info;
    }
    else
    *erro=1;
}
```

```
int Size(Pilha *P) {
return P->tam;
}
```

```
int IsEmpty(Pilha *P) {
    if (P->tam == 0)
        return(1);
    else return 0;
}
```

```
int IsFull(Pilha* P){
 return 0;
}
```

Main.c

```
#include "pilha.h"
#include <stdio.h> /* printf */

int main(void) {
    int erro, tamanho;
    elem x;
    Pilha* P = Create(&erro);
    if (erro) printf("erro ao criar a Pilha \n");
    else {
        x='a';
        Push(P,x,&erro);
        if (erro) printf("erro ao inserir %c\n",x);
        x='b';
        Push(P,x,&erro);
        if (erro) printf("erro ao inserir %c\n",x);
        x='c';
        Push(P,x,&erro);
        if (erro) printf("erro ao inserir %c\n",x);
```

```
tamanho = Size(P);
 printf("O tamanho da pilha eh: %d\n",tamanho);
 while (!IsEmpty(P)) {
     Pop(P,&x,&erro);
     if (!erro) printf("O elemento desempilhado eh: %c\n",x);
     else printf("erro\n");
 }
 system("pause");
 Empty(P,&erro);
 x='d';
 Push(P,x,&erro);
 if (erro) printf("erro ao inserir %c\n",x);
 x='e';
 Push(P,x,&erro);
 if (erro) printf("erro ao inserir %c\n",x);
 tamanho = Size(P);
 printf("O tamanho da pilha eh: %d\n",tamanho);
```

```
while (!IsEmpty(P)) {
        Pop(P,&x,&erro);
        if (!erro) printf("O elemento desempilhado eh: %c\n",x);
        else printf("erro\n");
    }

    Destroy(P,&erro);
   }
    system("pause");
    return 0;
}
```

Exercícios

1. Desenvolva um algoritmo para testar se uma pilha P1 tem mais elementos que uma pilha P2. Considere que P1 e P2 já existem. (Mais_Elementos?)

2. Desenvolva uma operação para transferir elementos de uma pilha P1 para uma pilha P2 (cópia). (Transferir_Elementos)

3. Desenvolva um algoritmo para inverter a posição dos elementos de uma pilha P. Você pode criar pilhas auxiliares, se necessário. Mas o resultado precisa ser dado na pilha P. (Inverter)

3. Desenvolva um algoritmo para testar se duas pilhas P1 e P2 são iguais. Duas pilhas são iguais se possuem os mesmos elementos, na mesma ordem. Você pode utilizar pilhas auxiliares também, se necessário. (Iguais?)

Observações

- Em 2
  - use Create e Destroy para não perder espaço de memória.
  - Não se esqueça de que ao dar Pop em P1 está esvaziando ela. É interessante manter P1 inalterada.
- Como são funções de um programa cliente, estas não devem acessar a ED da pilha
  - (e não podem, pois não sabemos a implementação atual)

Operações Primitivas Vs. Não Primitivas

- As operações não primitivas de um Tipo Abstrato de Dados são aquelas que podem ser implementadas através do acionamento das operações primitivas.

- As operações Mais-Elementos?, Iguais?, Inverter e Transferir_Elementos são operações não primitivas.

- E a melhor forma de implementar uma operação não primitiva, visando proporcionar portabilidade e reusabilidade, é através do acionamento de operações primitivas.

Editor de texto: exercício

- Considere que um editor de texto representa os caracteres digitados como uma pilha, sendo que o último caractere lido fica no topo
- Alguns comandos apagam caracteres. Por exemplo, o *backspace* apaga o último caractere lido
- Alguns comandos apagam tudo o que já foi lido anteriormente
- Considere que, no seu editor, # representa *backspace* e @ indica “apagar tudo”
- Faça um programa que execute essas ações usando o TAD pilha

Notação posfixa: exercício

- Avaliação de expressões aritméticas
  - Às vezes, na aritmética tradicional, faz-se necessário usar parênteses para dar o significado correto à expressão
    - A*B-C/D → (A*B)-(C/D)
  - Na notação polonesa (prefixa): operadores aparecem antes dos operandos e dispensam parênteses
    - -*AB/CD
  - Na notação polonesa reversa (posfixa): operadores aparecem depois dos operandos
    - AB*CD/-

Notação posfixa: exercício

- Interpretação da notação posfixa usando pilha
  - Empilha operandos até encontrar um operador
  - Retira os operandos, calcula e empilha o resultado
  - Até que se chegue ao final da expressão

Notação posfixa: exercício

- AB*CD/-

A					
A	B				
A	B	*			
A*B					
A*B	C				
A*B	C	D			
A*B	C	D	/		
A*B	C/D				
A*B	C/D	-			
A*B-C/D					

Exercício

- Implemente uma função que calcule o valor de uma expressão posfixa passada por parâmetro utilizando o TAD pilha

Exercício – resposta algorítmica; implementem em C

```
função valor(E: expressão): retorna real;
declare x real;
declare P pilha;
início
    Create(P)
    enquanto não acabou(E) faça
          início
          x=proxsimb(E);
          se x é operando então Push(P,x)
          senão início
                    remove operandos; {dois pops, em geral}
                    calcula o resultado da operação;
                    empilhe resultado;   {push}
                   fim
          fim
    valor=Top(P);
fim
```

Critérios para a pontuação do exercício sobre o TAD Pilhas

1 ou 2 funções com erros = -0.1
3 ou 4 funções com erros = -0.2
5 ou 6 funções com erros = -0.3
7 ou + funções com erros = -0.4
(ou erros conceituais graves)

até 4 métodos faltando = -0.1
5 ou 6 métodos faltando = -0.2
7 = 0.3

sem códigos de erro/comentários = até -0.1

Agradecimentos

- Thiago Pardo por parte do material