TDA-PILHA

Exercícios

1. Construa uma operação (hipotética) int buscaNaBase(Pilha *p, info *pReg) que permite acesso à informação na base da pilha (última versão da PE e PSE).

2. Implemente a função *int inverte(descritorPSE *p)* a qual inverte a ordem dos itens em uma PSE de maneira otimizada, com o menor número de passos. Considerando uma pilha contendo "a, ..., y, z, w", ao final da inversão teremos "w, z, y, ...,a", ou seja, o último que entrou será posicionado no topo da pilha, tornando-se o primeiro a sair, o penúltimo será o segundo a sair e assim sucessivamente até o primeiro que entrou, o qual será o último a sair. Veja a ilustração de uma PSE ao lado. Não será aceita a solução que simplesmente move o topo para a base da pilha, pois isso não mantém a consistência com as demais operações (os links usados para o encadeamento – "acima" e "abaixo" – ficariam inconsistentes).

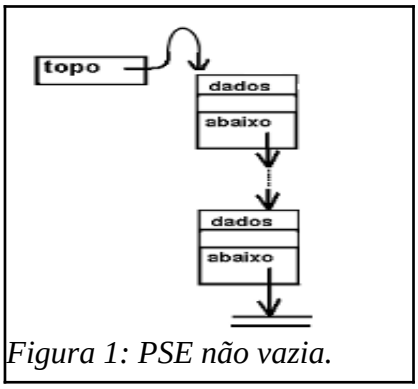

Figura 1: PSE não vazia.

3. Com base na pilha simplesmente encadeada, implemente uma Pilha Duplamente Encadeada (PDE).

4. Implemente a função *int inverte(descritorPDE *p)*, a qual inverte a ordem de uma PDE. Veja exemplo na Figura 1.

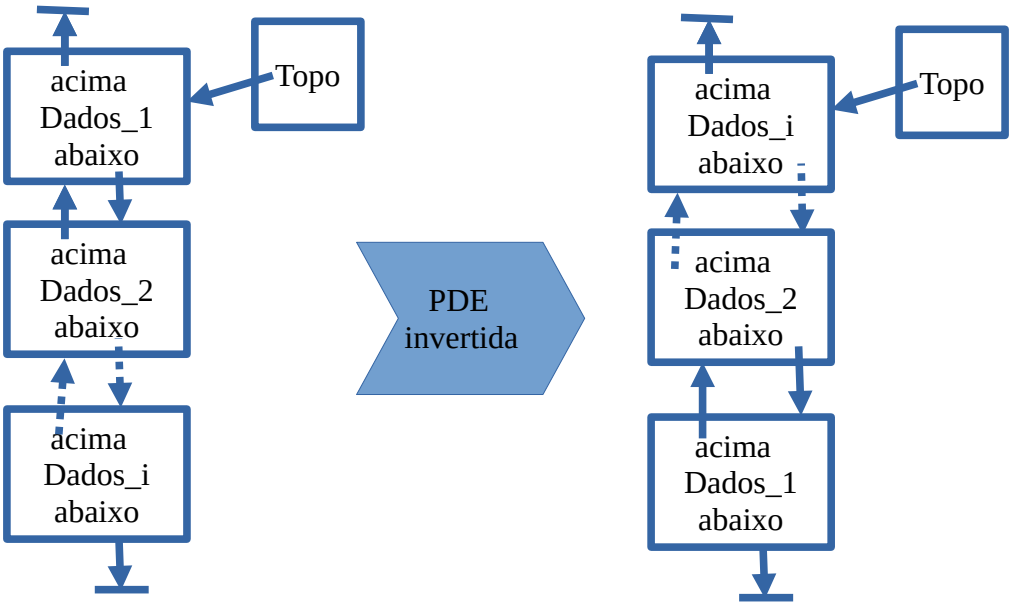

Figura 1: PDE e sua inversão.

5. Para uma pilha encadeada, construa uma função que conta a quantidade de *nós* de dados. Ao final a pilha deve preservar o mesmo estado de antes da execução da função.

6. O descritor da PE (última versão da pilha sobre vetor) não anota explicitamente o número atual de itens inseridos. Implemente uma função eficiente (com o menor número de passos) que retorna (return) o número de elementos atualmente empilhados na Pilha Estática. O estado da PE deve ser preservado. Protótipo: *int numeroDeEmpilhados(PE *p)*

7. A) Considere uma multipilha estática contendo duas pilhas em um mesmo vetor (Fig. 2):
   I) Inicialização (durante a criação) dos topos das pilhas como $topo_1$ = -1 e $topo_2$ = tamVetor;
   II) Todas as alocações de memória são realizadas na criação;
   III) As pilhas concorrem livremente pelo espaço no vetor, crescendo em direções opostas sem sobreposição de uma na outra.
   Implemente as seguintes operações: criação (instanciação), destruição (remoção da memória), reinicialização de uma pilha-alvo, empilhamento em uma pilha-alvo, desempilhamento de uma pilha-alvo, teste da condição de pilha-alvo cheia, teste da condição de pilha-alvo vazia, busca no topo de uma pilha-alvo, cálculo do número de elementos inseridos em uma pilha-alvo.

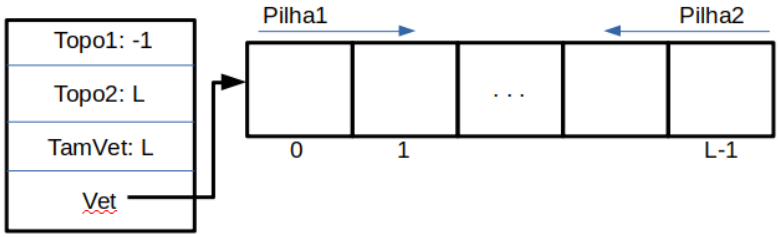

Figura 2: Duas pilhas concorrentes.

B) Considere uma multipilha estática contendo duas pilhas em um mesmo vetor (Fig. 3), porém cada pilha armazena um tipo de informação diferente:
   I) Inicialização (durante a criação) dos topos das pilhas como $topo_1$ = -1 e $topo_2$ = tamVetor;
   II) Todas as alocações de memória são realizadas na criação;
   III) As pilhas concorrem livremente pelo espaço no vetor, crescendo em direções opostas sem sobreposição uma na outra.
   Implemente as seguintes operações para esse TDA: empilhamento/desempilhamento de uma pilha, teste da condição de uma estar pilha cheia (vazia), criação, destruição, reinicialização da MpE ou de alguma pilha individual, busca em um topo, cálculo do número de elementos inseridos em uma pilha.

Você vai precisar utilizar o recurso da “union” para criar o vetor capaz de armazenar os dois modelos de dados (*structs*) definidos e manipulados pela aplicação.

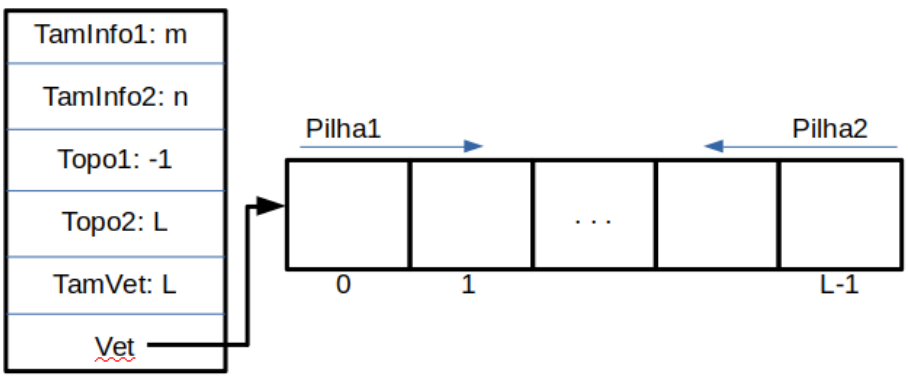

Figura 3: Duas pilhas concorrentes vazias, cada pilha é capaz de armazenar seu próprio modelo de informação (os modelos podem ser diferentes entre si).

8. Descreva o funcionamento (teste de mesa) e resultado da execução do código abaixo para a PDSE (Pilha Simplesmente Encadeada) discutida em sala, na qual o campo de ligação do elemento na base da pilha (posição oposta ao topo da pilha) aponta para NULL:

```
int xxx(descritor *p)
{
        if(p->topo == NULL)
                return FRACASSO;
        else
        { tmp=yyy(p, p→topo);
          tmp->abaixo=NULL;
        }
}
```

```
NoPDSE * yyy(descritor *p, NoPDSE *pNo)
{   if (pNo->abaixo==NULL)
        {  p->topo=pNo;
           return pNo;
        }
        else{  aux=yyy(p, pNo->abaixo);
               aux->abaixo=pNo;
               return(pNo);
            }
}
```

9. Considere uma nova proposta de PDSE, na qual o campo de ligação do elemento na base da pilha (posição oposta ao topo da pilha) aponta para ele mesmo:

   a) Descreva com ilustrações claras o comportamento da função *xxx(...)*, abaixo, considerando sua chamada após vários empilhamentos utilizando a função *int empilha(PDSE *p, info *novo)* abaixo.
   Para esta PDSE:

   (i) PDSE Vazia (*p->topo==NULL*);

   (ii) PDSE Não vazia (com *n* elementos, *n > 1* e *p->topo !==NULL*) cuja inserção ocorreu por meio da função de empilhamento abaixo.

   b) Qual o erro na função *xxx(...)* que a leva a uma contagem infinita? Como corrigir isso?

```
int xxx(PDSE *p)
{     pNoPDSE aux=NULL;
      int cont =0;
      aux=p->topo)
      while(aux != NULL)
        {
                cont++;
                aux=aux→abaixo;
        }
}
int empilha(PDSE *p, info *novo)
{
        NoPDSE *temp;
        int ret = FRACASSO;
        if((temp=(NoPDSE *)malloc(sizeof(NoPDSE)))!=NULL)
        {
                memcpy(temp->dados,novo,p->tamInfo);
                if(p->topo==NULL)
                        temp->abaixo=temp;
                else
                        temp->abaixo=p->topo;
                p->topo=temp;
                ret = SUCESSO;
        }
        else
            free(temp);
        return ret;
}
```

10. A) Implemente a função *int buscaNoTopoDeUmaPilha(MultiPilha *p, int idPilha, info *destino)* para a multipilha exibida na Figura 4, a qual copia o elemento no topo da pilha *idPilha* para o endereço na variável *destino*. A função retorna FRACASSO ou SUCESSO a depender do *status* de *idPilha* (vazia ou não vazia).

    Protótipo: *int buscaNoTopoDeUmaPilha(MultiPilha *p, int idPilha, void *destino);*

    ATENÇÃO:

    a) Há um único descritor geral;

    b) *N* corresponde ao número total de pilhas implementadas;

    c) Há um descritor individual D_i para cada pilha P_i, onde $1 \leq i \leq N$.

    d) O vetor de descritores individuais possui *N* células;

    e) *L* é o comprimento máximo de uma pilha P_i;

    f) $IniPart_i$ corresponde ao início da partição da P_i sobre o vetor de pilhas:

    $IniPart_i = (i-1)*L;$

    g) O final de uma partição não é anotado no descritor, mas pode ser calculado por meio da equação: $fimPart_i = IniPart_i + (L-1)$;

    h) Em termos de implementação, $topo_i$ é um deslocamento a partir de $IniPart_i$ que determina o topo efetivo da desejada pilha P_i. Por exemplo: o topo da P_2 estará efetivamente localizado em $vet[iniPart_2+topo_2]$

    i) $-1 \leq topo_i \leq (L-1)$, $topo_i = -1$ indica P_i vazia, $topo_i = L-1$ indica P_i cheia;

    j) Quando são possíveis, as inserções ocorrem a incrementos do $topo_i$;

    k) Quando são possíveis, as remoções ocorrem por decrementos do $topo_i$;

    l) O vetor de pilhas possui $W=N*L$ células;

    m) Cada descritor individual guarda o seu $tamInfo_i$, pois cada pilha apresenta o seu próprio tamanho de informação (tamanho em bytes da informação manipulada pela respectiva pilha P_i);

    n) Todos os campos são inicializados durante a criação/instânciação da est. de dados.

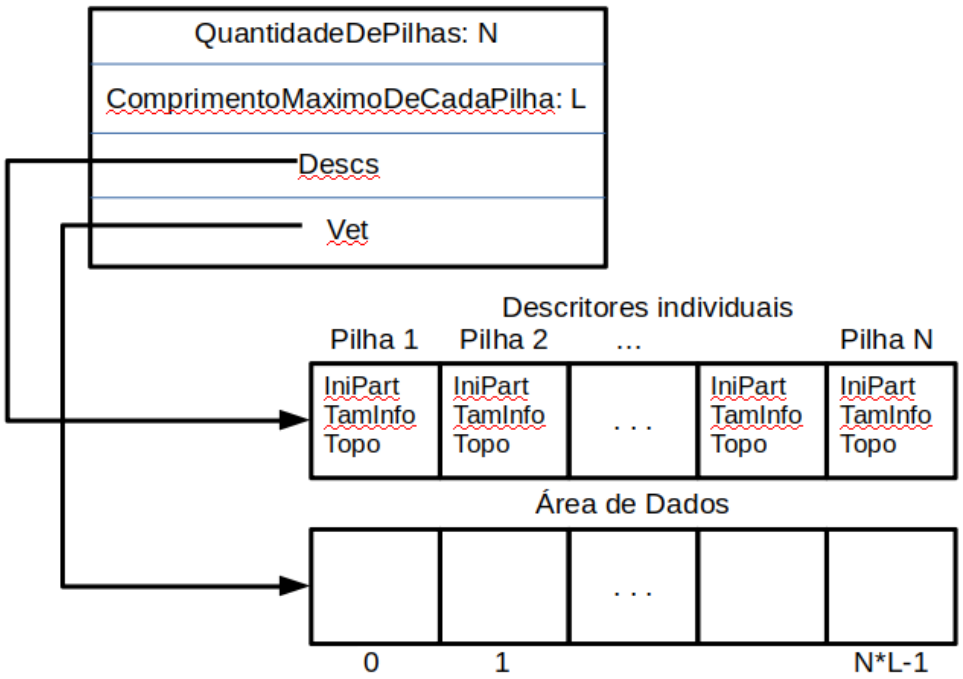

Figura 4:Uma MPE: Descritor geral apontando para vetor de descritores e vetor de dados.

B) Implemente a mesma função anterior considerando que o tamanho de partição (L) varia para cada pilha. Portanto, L tem que constar como atributo em cada descritor individual, não mais fazendo parte do descritor geral.

*int buscaNoTopoDeUmaPilha(MultiPilha *p, int idPilha, info *destino);*

11. Proponha e implemente uma multipilha dinâmica com as seguintes operações: empilhamento/desempilhamento de uma pilha, teste da condição de uma estar pilha cheia (vazia), destruição, reinicialização da MpE ou de alguma pilha individual, busca em um topo, cálculo do número de elementos inseridos em uma pilha.

12. Dada uma multipilha estática contendo N pilhas sob um mesmo vetor, o qual apresenta um conjunto de células reservadas para atender a uma das pilhas (P_i , $1 \leq i \leq N$)) que tenha ficado cheia e não possa atender a uma inserção corrente. Nesse caso, a pilha P_i poderá utilizar uma região de reserva, caso a mesma não tenha sido reclamada anteriormente. As demais pilhas poderão ser deslocadas para benefício da pilha reclamante. Cada descritor (D_i) apresenta o respectivo Início da Partição (IP_i) e Final da Partição (Fp_i).

    Implemente as seguintes operações: empilhamento/desempilhamento de uma pilha, teste da condição de uma estar pilha cheia (vazia), destruição, reinicialização da MpE ou de alguma pilha individual, busca em um topo, cálculo do número de elementos inseridos em uma pilha.

13. Utilize uma ou mais pilhas para serem aplicadas nos seguintes casos:
a) cálculo de expressões matemáticas pós-fixas. Exemplo: (*A B C ^ / D E * + A C * –*) equivalente à notação infixa (*A / B ^ C + D * E – A * C*), onde o operador x^y corresponde a x^y. (veja solução em HOROWITZ [6]).

Operador	Ordem de precedência
^	3
*, /	2
+,-	1

b) Avaliação do correto "aninhamento" de expressões delimitadas por parênteses, colchetes e chaves. Ex: ([x^{z+y}-t]*[z+w]/x).

c) Ordenação de um vetor com o auxílio de duas pilhas. Este não é um método eficiente para ordenação.

d) Construção de uma função de inserção em ordem em um vetor de inteiros, com auxílio de uma pilha.

e) Conversão de número inteiro decimal em seu equivalente binário, octal ou hexadecimal.

f) Conversão de expressões infixas em pós-fixas (veja solução em HOROWITZ [6]).

g) Discorra sobre a aplicação de pilhas para o rastreio de chamadas recursivas a uma função. A pilha deverá guardar os parâmetros importantes das chamadas.

h) Adaptado da maratona de programação, treinamento disponível no URL: http://br.spoj.com/problems/ACIDO/:

> Um novo RNA foi descoberto. Ele também é constituído de uma cadeia composta de vários intervalos chamados *trans-acting siRNAs*, abreviadamente conhecidos como TAS.
>
> Cada TAS é composto de bases identificadas como B, C, F e S e podem ser pareadas com as bases correspondentes de outro TAS, sendo que os únicos pares possíveis são formados entre as bases B-S e C-F (de TAS diferentes).
>
> O RNA é capaz de dobrar-se sobre si mesmo em intervalos (TAS), para maximizar o número de ligações entre as bases de TAS diferentes. Ao dobrar-se, todas as bases no intervalo se ligam com as suas correspondentes, formando os pares válidos entre bases (Fig. 5).

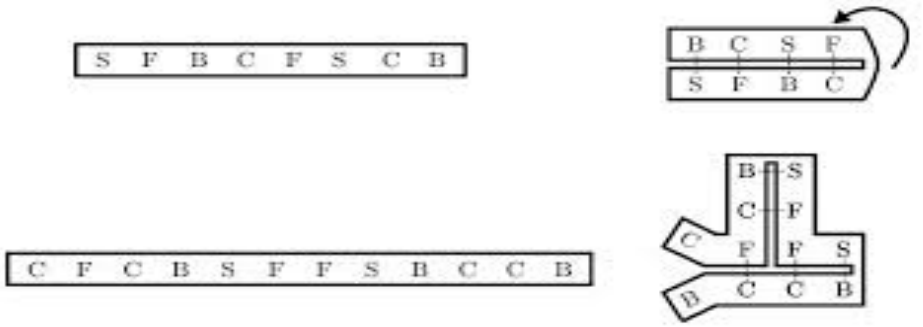

Figura 5: Dobraduras e ligações de bases entre cadeias de TAS.

Considerando que:

- As bases são apenas as quatro seguintes: B, C, F e S
- Cada base pode se ligar a apenas uma outra base (B liga com S e F com C) de outro TAS;
- Não há ordem no emparelhamento, a ligação S-B é tão válida quanto a ligação B-S, o mesmo para C-F e F-C;
- As dobraduras devem ser orientadas para a maximização do número de ligações entre bases;

Utilize uma pilha para determinar o número de ligações entre as bases de uma cadeia ativa.

Exemplos:

Entradas: cadeias	SBC	FCC	SFBC	SFBCFSCB	CFCBSFFSBCCB
Saídas: número de ligações	1	1	0	4	5

i) Um baralho contém 52 cartas distribuídas em quatro naipes: espadas (spades), paus (clubs), ouros (diamonds) e copas (harts). Cada naipe com treze cartas: *1* (*ás), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 (valete), 12 (dama)* e *13 (rei).*
Considerando a existência do TDA pilha estática, já implementado, cuja interface oferece as funções exibidas na Tabela 1, utilizando **apenas** pilhas estáticas e na **menor** quantidade possível, pede-se as seguintes aplicações para o TDA-PE:

a. Construa (em *C*) a estrutura da informação que representa a carta do baralho;
b. Construa o procedimento *PE* embaralha(PE* pBaralhoCartas)* o qual tem como entrada uma PE apontada por *pBaralhoCartas* e retorna esta mesma pilha embaralhada. Uma ideia é dividir o baralho ao meio e intercalar as cartas para prover alguma desordenação;
c. Um procedimento comum é separar as 52 cartas nas 4 sequências (correspondentes aos respectivos naipes), sendo cada sequência ordenada pelos valores das cartas:
1 (*ás), 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 (valete), 12 (dama)* e *13 (rei)*. Programe o processo de separação das cartas nos quatro naipes ordenados. Utilize o protótipo abaixo:

void separa(PE pBaralhoCartas, PE* pPEcopas,*
PE pPEpaus, PE* pPEouros, PE* pPEespadas);*

Onde o endereço em *pBaralhoCartas* corresponde à PE que representa o baralho de entrada. Os apontadores *pPEcopas, pPEpaus, pPEouros, pPEespadas* referenciam as PEs de saída, as quais representam as quatro sequências desejadas.

Criação da PE:	**PE *p cria(PE *p, int tamPilha);**
Destruição de qualquer traço da PE na memória:	**PE * destroi(PE *p);**
Força a PE ao estado de recém criada (vazia):	**int reinicia(PE *p);**
Copia a informação de topo no endereço apontado por pRegBuscado:	**int buscaNoTopo(PE *p, info *pRegBuscado);**
Copia a informação de topo no endereço apontado por pRegExcluido e desempilha atualizando o topo:	**int desempilha(PE *p, info *pRegExcluido);**
Empilha copiando o conteúdo apontado por pRegNovo para o novo topo da PE:	**int empilha(PE *p, info *pRegNovo);**
Testa a condição de PE vazia:	**int testaVazia(PE *p);**
Testa a condição de PE cheia:	**int testaCheia(PE *p);**
retorna o tamanho (número de elementos) inseridos na pilha:	**int numElementos(PE *p);**

Tabela 1: Interface do TDA PE.

Bibliografia

[1] Backus, John. Evolução das Principais Linguagens de Programação. In: Sebesta, R. W. Conceitos de Linguagens de Programação. 4ª ed. Porto Alegre: Editora Bookman, 2000. p. 49-110.

[2] Barr, Michael. Programming embedded Systems in C and C++. First Edition. Sebastopol: O'Reilly and Associates, 1999. 174p. cap 1.: Introduction: C: The least Common Denominator.

[3] Fishwick, P. A. Computer Simulation Potentials, IEEE , Volume: 15 , Issue: 1 , P:24–27.Feb.-March.1996.

[4] Gersting, Judith L. "Fundamentos Matemáticos para a Ciência da computação". Ed. LTC.

[5] Goldberg, Adele. Suporte para Programação Orientada a Objeto. In: Sebesta, R. W. Conceitos de Linguagens de Programação. 4ª ed. Porto Alegre: Editora Bookman, 2000. p. 417-466.

[6] Horowitz, E. & Sahni, S. "Fundamentos de Estruturas de Dados". Ed. Campus. 1984.

[7] Pereira, Sílvio L. Estruturas de Dados Fundamentais – Conceitos e Aplicações. 7a. ed. Érica, 2003,p. 73

[8] Parnas, D. L. On the Criteria to be Used in Decomposing systems into Modules Communications of the ACM**,** Volume 15 Issue 12. December 1972.

[9] Ritchie, Dennis. Subprogramas. In: Sebesta, R. W. Conceitos de Linguagens de Programação. 4ª ed. Porto Alegre: Editora Bookman, 2000. p. 315-359.

[10] Szwarcfiter, Jayme L. & Markenzon L. “Estruturas de Dados e Seus Algoritmos”. Rio de Janeiro. Ed. LTC. 1994.

[11] Tenembaum, Aaron M. e outros. “Estruturas de Dados Usando C”. São Paulo. Ed.Makron Books. 1995.

[12] Tremblay, J.; Bunt, R..Ciência dos Computadores. Uma abordagem algorítmica, McGraw Hill, 1983.

[13] Van Gelder, Allen. A Discipline of Data Abstraction using ANSI C. Documento disponível em http://www.cse.ucsc.edu/~avg/. 02/2005.

[14] Veloso, Paulo et al. “Estruturas de Dados”. Ed. Campus. 1984.

[15] Wirth, Niklaus. “Algorithms + Data structures = Programs”, Ed. Prentice Hall, 1976.