SCC 202 Prova 1

28/9/2010

Resolução e Comentários

Questão 1) (3.5) Sobre TADs.

a) O que **é** e quais são as **vantagens** de se utilizar Tipos Abstratos de Dados (TADs) no projeto de desenvolvimento de software?

- Um tipo abstrato de dado, ou TAD,
  - especifica um conjunto de operações (ou métodos) e
  - a semântica das operações (o que elas fazem),
- mas não especifica a implementação das operações. Isto é o que o faz abstrato.

- Os programas que usam o TAD não “conhecem” as implementações
  - Fazem uso do TAD através de suas operações

- A característica essencial de um **TAD** é a **separação** entre **conceito** e **implementação**.

- Ao usuário são fornecidos a descrição dos valores e o conjunto de operações do TAD,
  - mas a implementação do tipo e das operações são **invisíveis** e **inacessíveis**.

- Vantagens: estimula o **reuso de código**, pois o código de TAD deve ter sido fortemente testado, e permite **modificar as implementações** dos tipos com o **menor impacto possível** para os programas que o usam, **minimizando custos**.

b) Defina um TAD para representar um ponto no R2. Chame o tipo de **Ponto** e faça a **interface** e **implementação** das seguintes operações **em C**:

1. cria: operação que cria um ponto com coordenadas x e y;
2. libera: operação que libera a memória alocada por um ponto;
3. acessa: operação que devolve as coordenadas de um ponto;
4. atribui: operação que atribui novos valores às coordenadas de um ponto;
5. distancia: operação que calcula a distância entre dois pontos, usando o Teorema de Pitágoras.
6. soma de dois pontos, via soma termo a termo resultando em outro ponto

Interface ponto.h

```
/* TAD: Ponto (x,y) */

/* Tipo exportado */
typedef struct ponto Ponto;

/* Funções exportadas */

/* Função cria - Aloca e retorna um ponto com coordenadas (x,y).
Retorna flagErro = 1 se ponto não alocado e flagErro = 0 se sucesso. */
Ponto* pto_cria (float x, float y, int *flagErro);

/* Função libera - Libera a memória de um ponto previamente criado.
Retorna flagErro = 2 se houve problema na liberação e flagErro = 0 se sucesso.
    */
void pto_libera (Ponto* p, int *flagErro);
```

/* Função acessa - Retorna os valores das coordenadas de um ponto.
Retorna flagErro = 2 se houve problema no acesso e flagErro = 0 se sucesso */
void pto_acessa (Ponto* p, float* x, float* y, int *flagErro);

/* Função atribui - Atribui novos valores às coordenadas de um ponto.
Retorna flagErro = 2 se houve problema na atribuição e flagErro = 0 se sucesso */
void pto_atribui (Ponto* p, float x, float y, int *flagErro);

/* Função distancia - Retorna a distância entre dois pontos.
Retorna flagErro = 2 se houve problema no cálculo e flagErro = 0 se sucesso */
float pto_distancia (Ponto* p1, Ponto* p2, int *flagErro);

/* Função soma – Retorna um novo ponto cujas coordenadas são a soma dos termos das coordenadas dos pontos p1 e p2. Retorna flagErro = 2 se houve problema na soma e flagErro = 0 se sucesso */
Ponto* pto_soma(Ponto *p1, Ponto *p2, int *flagErro);

Implementação ponto.c

```
#include <stdlib.h>
#include “ponto.h"
 struct ponto {
float x;
float y;
}

Ponto* pto_cria (float x, float y, int *flagErro)
{
Ponto* p = (Ponto*) malloc(sizeof(Ponto));
if (p == NULL) {
*flagErro = 1; // ERRO_MEMORIA_INSUFICIENTE
return p;
} else
{
*flagErro = 0; // SUCESSO
p->x = x;
p->y = y;
return p; }
}
```

```
void pto_libera (Ponto* p, int *flagErro)
{
if(p != NULL){
free(p);
*flagErro = 0; //SUCESSO
} else
*flagErro = 2; // ERRO_PONTEIRO_NULO
}


void pto_acessa (Ponto* p, float* x, float* y, int *flagErro)
{
if(p != NULL){
*x = p->x;
*y = p->y;
*flagErro = 0; //SUCESSO
} else
*flagErro = 2; // ERRO_PONTEIRO_NULO
}
```

```
void pto_atribui (Ponto* p, float x, float y, int *flagErro)
{
if(p != NULL){
p->x = x;
p->y = y;
*flagErro = 0; //SUCESSO
} else
*flagErro = 2; // ERRO_PONTEIRO_NULO
}

float pto_distancia (Ponto* p1, Ponto* p2, int *flagErro)
{
if(p1 != NULL & p2!= NULL){
*flagErro = 0; //SUCESSO
float dx = p2->x – p1->x;
float dy = p2->y – p1->y;
return sqrt(dx*dx + dy*dy);
} else
{
*flagErro = 2; // ERRO_PONTEIRO_NULO
return 0;
}
}
```

```
Ponto* pto_soma(Ponto *p1, Ponto *p2, int *flagErro);
{
if(p1 != NULL & p2!= NULL){
Ponto* p = (Ponto*) malloc(sizeof(Ponto));
if (p == NULL) {
*flagErro = 1; // ERRO_MEMORIA_INSUFICIENTE
return p;
} else
{
*flagErro = 0; //SUCESSO
p->x = p1->x + p2->x;
p->y = p1->y + p2->y;
return p;
} else
{
*flagErro = 2; // ERRO_PONTEIRO_NULO
return NULL;
}
}
```

c) Escreva um **programa em C** que faça uso do TAD ponto definido acima, chamando as 6 operações acima.

```
#include <stdio.h>
#include "ponto.h"
int main (void)
{
int erro;
float x, y;

Ponto* p = pto_cria(2.0,1.0, &erro);
Ponto* q = pto_cria(3.4,2.1,&erro);
float d = pto_distancia(p,q,&erro);
printf("Distancia entre pontos: %f\n",d);

Ponto* r = pto_soma (p, q,&erro);
pto_acessa (r, &x, &y, &erro);

printf("Soma dos pontos (X,Y): %f\ - %f\n",x,y);
pto_atribui (r, y, x, &erro);

pto_libera(q,&erro);
pto_libera(p,&erro);
pto_libera(r,&erro);

return 0;
}
```

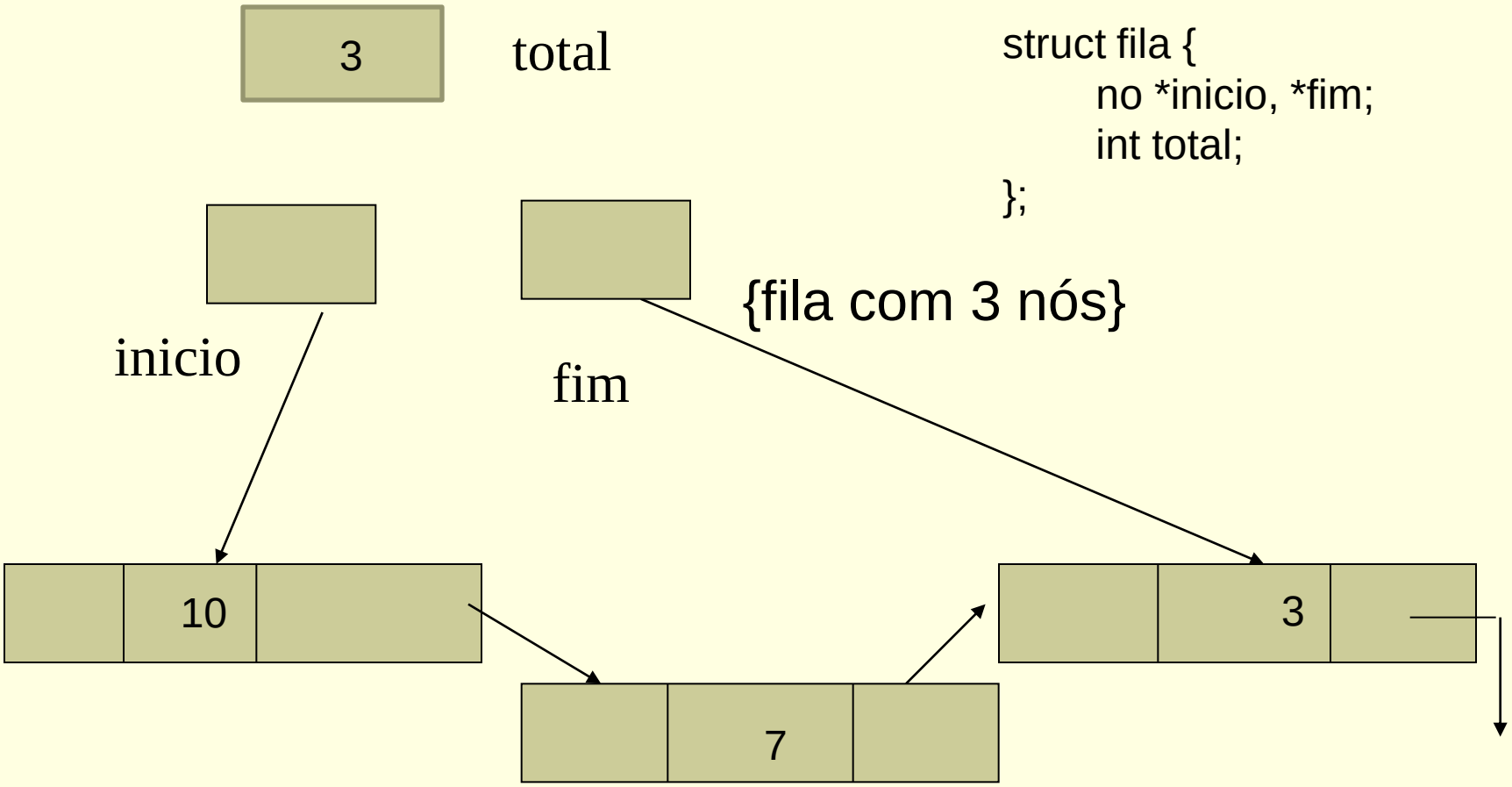
Questão 2) (1.0) Sobre Projeto de ED.

- Uma central de atendimento a clientes tem vários atendentes, mas um número muito maior de linhas telefônicas recebendo chamadas. As chamadas são colocadas em uma fila de espera segundo a ordem de chegada (e atendidas quando possível). Ocorre que algumas destas chamadas vêm de longe, e neste caso, se elas ficam esperando na linha, elas ficam causando uma despesa muito maior do que as chamadas que vem de perto.
-
- Uma solução alternativa seria colocar as chamadas em fila segundo a prioridade definida primeiramente pelo custo (as mais caras devem esperar menos) e secundariamente por ordem de chamada.
- **Projete uma estrutura para modelar essa situação alternativa. Faça um desenho, descreva o funcionamento da estrutura, via as funções principais de inserção e remoção e escreva a ED em C, justificando a escolha da REPRESENTAÇÃO e da IMPLEMENTAÇÃO.**

Desenho, ins e rem, ED

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

struct fila {
        no *inicio, *fim;
        int total;
};
```



Insere uma chamada segundo a sua prioridade; remove do início. A REP/IMP enc/din facilita a inserção de um nó segundo sua prioridade, pois não há necessidade de movimentação de itens (SEQ/EST)

Questão 3) (1.5) Sobre Listas.

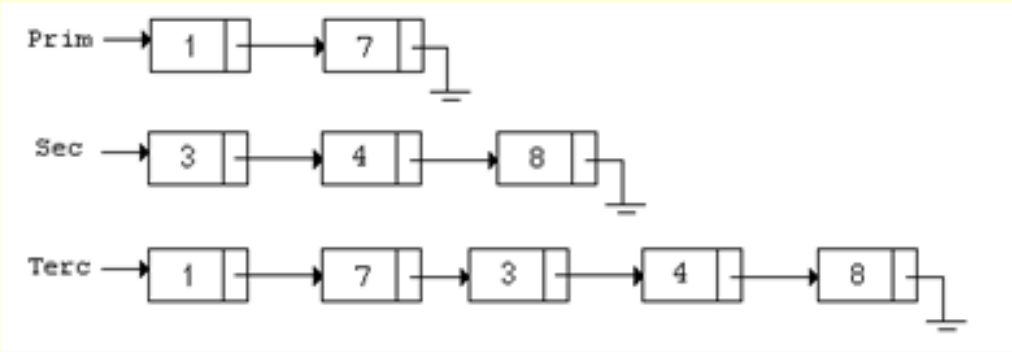
a) Usando a estrutura de dados abaixo para uma de **lista simplesmente encadeada** de inteiros sendo representada pelo ponteiro para seu primeiro elemento (Lista*):

/* declarações em Lista.h */

```
typedef int elem;
typedef struct lista Lista;
```

/* declaração em Lista.c */

```
struct lista {
    elem info;
    struct lista* prox;
};
```



Implemente uma função em C para fazer parte de um TAD listas que faça a união/concatenação das duas listas ordenadas (Prim e Sec), gerando uma terceira (Terc):

Lista *concatena(Lista* L1, Lista* L2) {

```
 Lista *p, *p1; /* aux */
if (L1 == NULL) p = L2;
else {
p = L1;
If (L2 != NULL) {
   p1 = L1;
   while (p1->prox != NULL) p1 = p1->prox;
   p1->prox = L2;
}
}
L1 = NULL; L2 = NULL;
return p;
}
```

b) Quais são as vantagens de se usar um nó cabeça de lista? Considere pelo menos 3 cenários de uso.

- Pode armazenar o número de elementos da lista de forma que uma função que retorna o tamanho não tenha que percorrer a lista contando seus elementos
- Pode guardar informações da aplicação, como:
  - Em uma fábrica, guarda-se as peças que compõem cada equipamento produzido, sendo este indicado pelo nó sentinela
  - Informações do vôo correspondente a uma fila de passageiros
- Pode guardar o item de busca (facilita se lista for circular, assim não tem perigo de acessar valor inválido)
- Aponta um nó específico numa busca para indicar a continuação da busca

Questão 4) (4.0) Sobre Pilhas e Filas. *Usando as interfaces do TAD pilha e do TAD fila dadas abaixo*.

a) Faça uma função "Imprime" que imprime os itens de uma Pilha, sem destruir a Pilha, passada como parâmetro.

b) Faça uma função "Insere_Ordenado" que insere um Item em uma Pilha obedecendo os seguintes requisitos:

b.1) o procedimento deverá receber através de parâmetro passado por referência a variável Pilha e através de parâmetro passado por valor o Item a ser inserido;

b.2) a pilha apresenta seus itens ordenados decrescentemente e o novo item deve ser inserido de forma a manter a ordem original dos itens da pilha.

c) Faça uma função "Imprime" que imprime os itens de uma Fila, sem destruir a Pilha, passada como parâmetro.

OBS: todas as funções acima são cliente, portanto são construídas usando os TAD fornecidos.

```
procedure ImprimePilha(var Pilha: TipoPilha);
var PilhaAux: TipoPilha;
          x: TipoItem;
begin
  CriaPilhaVazia(PilhaAux);

  writeln('Itens da Pilha.');
  while not PilhaVazia(Pilha) do
  begin
    Desempilha(Pilha, x);
    writeln(x);
    Empilha(x, PilhaAux);
  end;

  while not PilhaVazia(PilhaAux) do
  begin
    Desempilha(PilhaAux, x);
    Empilha(x, Pilha);
  end;
end;
```

Algoritmo

```
{ Imprime os itens de uma Fila. Note que os itens retirados do início
    da fila (desenfileira), para serem impressos, são deslocados para
    o final (enfileira); ao final do processo de impressão, os itens da
    fila completaram um ciclo retornando para as suas posições
    originais. }
procedure ImprimeFila(var Fila: TipoFila);
var x: TipoItem;
   i: integer;
begin

  writeln('Itens da fila.');
  for i:=1 to TamanhoFila(Fila) do
  begin
    Desenfileira(Fila, x);
    writeln(x);
    Enfileira(x, Fila);
  end;

end;
```

Algoritmo

```
procedure InsereNaOrdem(var Pilha: TipoPilha; x: TipoItem);
var PilhaAux: TipoPilha;
        xAux: TipoItem;
        Sair: boolean;
begin
  CriaPilhaVazia(PilhaAux);
{ procura a posição correta do item "x" }
  Sair := false;
  while (not PilhaVazia(Pilha)) and (not Sair) do
  begin
    Desempilha(Pilha, xAux);
    if xAux <= x
       then begin
              Empilha(xAux, Pilha);
              Sair := true; { encontrou a posição do item }
            end
    else Empilha(xAux, PilhaAux);
  end;

  Empilha(x, Pilha);          { insere o item na posição }

{ retorna os itens da pilha auxiliar }
  while not PilhaVazia(PilhaAux) do
  begin
    Desempilha(PilhaAux, xAux);
    Empilha(xAux, Pilha);
  end;
end;
```


Algoritmo