1º Teste de Algoritmos e Estruturas de Dados

13 de Abril de 2013

Duração: 2 horas

Enunciado do Problema

Título
Percurso divertido.

Objectivo
Gerir o percurso pelas diversões dum parque, realizado por uma determinada criança.

Funcionalidades

O parque, por exemplo a Euro Disney, é constituído por diferentes diversões (ex., montanha russa, roda gigante, etc...), cada uma delas com um nome único que a identifica.

Cada criança no parque pode usufruir das diferentes diversões as vezes que quiser. A cada criança é atribuído um número único, e é sempre registada eletronicamente a sua visita a cada uma das diversões. Para além disso, a criança atribui uma pontuação à diversão numa escala de 0 a 20, cada vez que realiza uma visita a essa diversão.

Interacção com o utilizador

A primeira coisa que é registado na aplicação é o número atribuído à criança, da qual se vai gerir o seu percurso no parque. Uma vez identificada a criança, a interação com o utilizador é realizada através de um interpretador de comandos. Os comandos a processar são:

- V <nome_diversão> <pontos> -> regista a visita da criança na diversão dada, como sendo a última visita até ao momento. Isto é, se a criança já visitou n diversões, então esta será a (n + 1) visita. Para além disso atribui a essa visita a pontuação indicada.
- L -> lista pela ordem de visita da criança todos os nomes das diversões visitadas, com a respectiva pontuação.
- D <i> -> indica o nome da i-ésima diversão que a criança visitou durante o seu percurso. Esta operação só tem sucesso se "i" for menor ou igual ao número de visitas realizadas pela criança.
- M -> indica o nome da diversão visitada pela criança que tem a maior pontuação até ao momento. Em caso de existir empate nas pontuações é apresentado o nome da diversão visitada há menos tempo pela criança. Esta operação só tem sucesso se a criança visitou pelo menos uma diversão.
- N <nome_diversão> -> indica o número de visitas que a criança realizou à diversão dada.
- X -> saída da aplicação, indicando o número total de visitas realizadas pela criança.

Os antigos alunos de AED já definiram os TADs do domínio do problema que serão usados para resolver o problema "Pecurso divertido". Os TADs identificados foram:

- TAD percursoParque: representa o percurso pelas diversões dum parque, realizado por uma criança;
- TAD visita: representa a visita a uma diversão e os pontos atribuídos pela criança a essa visita.

Pegunta I (4 valores)
Implemente o TAD visita, o qual está definido no ficheiro "visita.h", da seguinte forma:

```
#ifndef _H_VISITA
#define _H_VISITA

/* Tipo do TAD visita */
typedef struct _visita * visita;

/************************************************
criaVisita - Criação da instância da estrutura associada a uma visita.
Parâmetros:
        nomeDiv - nome da diversão visitada
        pontosDiv - pontos atribuídos à visita
Retorno: apontador para a instância criada
Pré-condições:
************************************************/
visita criaVisita(char * nomeDiv, int pontosDiv);

/************************************************
destroiVisita - Liberta a memória ocupada pela estrutura associada à visita.
Parâmetros:
        v - visita a destruir
Retorno:
Pré-condições: v != NULL
************************************************/
void destroiVisita(visita v);

/************************************************
diversaoVisita - Consulta o nome da diversão da visita dada.
Parâmetros:
        v - visita
Retorno: nome da diversão
Pré-condições: v != NULL
************************************************/
char * diversaoVisita(visita v);

/************************************************
pontosVisita - Consulta os pontos atribuídos à visita dada.
Parâmetros:
        v - visita
Retorno: pontos da visita
Pré-condições: v != NULL
************************************************/
int pontosVisita(visita v);

#endif /* _H_VISITA */
```

Pergunta II

O TAD percursoParque está definido no ficheiro "percursoParque.h" da seguinte forma:

```
#ifndef _H_PERCURSOPARQUE
#define _H_PERCURSOPARQUE

/* Tipo do TAD percursoParque */
typedef struct _percurso * percursoParque;

/***********************************************
criaPercurso - Criação da instância da estrutura associada a um percursoParque.
Parâmetros:     nCrianca - número da criança no parque     nVis - número previsto de visitas no parque
Retorno: apontador para a instância criada
Pré-condições:
***********************************************/
percursoParque criaPercurso(int nCrianca, int nVis);

/***********************************************
destroiPercurso - Liberta a memória ocupada pela estrutura associada ao percursoParque dado.
Parâmetros:     p - percursoParque a destruir
Retorno:
Pré-condições: p != NULL
***********************************************/
void destroiPercurso(percursoParque p);

/***********************************************
adicionaVisPercurso - adiciona uma visita ao percursoParque dado, como sendo a última visita do percurso.
Parâmetros:     p – percursoParque     nomeDiv - nome da diversão visitada
                pontosVis - pontos atribuídos à visita
Retorno:
Pré-condições: p != NULL
***********************************************/
void adicionaVisPercurso(percursoParque p, char * nomeDiv, int pontosVis);

/***********************************************
tamanhoPercurso - Consulta o número de visistas registadas no percursoParque dado.
Parâmetros:     p - percursoParque
Retorno: número de visitas no percurso
Pré-condições: p != NULL
***********************************************/
int tamanhoPercurso(percursoParque p);

/***********************************************
visitaPercurso - Retorna a i-ésima visita realizada no percursoParque dado.
Parâmetros:     p – percursoParque     i - posição (valor inteiro)
Retorno: i-ésima visita do percurso
Pré-condições: p != NULL && i > 0 && i <= tamanhoPercurso(p)
***********************************************/
visita visitaPercurso(percursoParque p, int i);

/***********************************************
maxVisitaPercurso - Retorna a visita realizada com pontuação máxima entre todas as visitas do percursoParque dado. Em caso de empate nas pontuações das visitas, retorna a visita que tenha sido visitada há menos tempo.
Parâmetros:     p - percursoParque
Retorno: visita do percurso com pontuação máxima
Pré-condições: p != NULL
***********************************************/
visita maxVisitaPercurso(percursoParque p);
```

```
/***********************************************
numVisitaPercurso - Retorna o número de vezes que a diversão com o nome dado foi visitada no percursoParque dado.
Parâmetros:          p – percursoParque              nomeDis - nome da diversão
Retorno: número de vezes visitada
Pré-condições: p != NULL
***********************************************/
int numVisitaPercurso(percursoParque p, char * nomeDis);

/***********************************************
iteradorPercurso - Retorna um iterador para iterar/percorrer as visitas do percursoParque dado.
Parâmetros:          p - percursoParque
Retorno: iterador
Pré-condições: p != NULL && tamanhoPercurso(p) > 0
***********************************************/
iterador iteradorPercurso(percursoParque p);

#endif /* _H_PERCURSOPARQUE */
```

II.a. (5 valores)

Para implementar o TAD percursoParque, qual das opções dadas será a mais adequada para a estrutura de dados. Justifique a sua resposta.

Note que nas opções dadas são utilizados os TADs apresentados em anexo, assim como o TAD visita (pergunta I).

Opção A

```
struct _percurso{
    int numeroCrianca;
   fila percurso; // fila de visitas
   visita maxUltVisita; // última visita com máxima pontuação
}
```

Opção B

```
struct _percurso{
    int numeroCrianca;
   dicionario percurso; // dicionário de visitas
   visita maxUltVisita; // última visita com máxima pontuação
}
```

Opção C

```
struct _percurso{
    int numeroCrianca;
   sequencia percurso; // sequência de visitas
   visita maxUltVisita; // última visita com máxima pontuação
}
```

Opção D

```
struct _percurso{
    int numeroCrianca;
   conjunto percurso; // conjunto de visitas
   visita maxUltVisita; // última visita com máxima pontuação
}
```

II.b. (9 valores)
Implemente o TAD percursoParque, tendo como base a estrutura escolhida na pergunta II.a.
Caso necessite de alguma operação para tratar o tipo de dados "visita" como "void *", pode usar as seguintes operações, definidas no ficheiro "visitaGen.h":

```
#ifndef _H_VISITAGEN
#define _H_VISITAGEN

/************************************************
destroiVisitaGen - Liberta a memória ocupada pela estrutura associada à visita.
Parâmetros:
        v - visita a destruir (void *)
Retorno:
Pré-condições: v != NULL
************************************************/
void destroiVisitaGen(void * v);

/************************************************
diversaoVisitaGen - Cria e devolve uma cópia do nome da diversão (do tipo void *) associada à visita dada.
Parâmetros:
        v - visita (void *)
Retorno: nome da diversão (void *)
Pré-condições: v != NULL
************************************************/
void * diversaoVisitaGen(void * v);

/************************************************
igualVisita - Indica se as visitas dadas são iguais.
Parâmetros:
        v1 - visita (void *)
        v2 - visita (void *)
Retorno: 1 caso iguais; 0 caso contrário
Pré-condições: v1 != NULL && v2 != NULL
************************************************/
int iguaisVisitaGen(void * v1, void * v2);

#endif /* _H_VISITAGEN */
```

Pergunta III (2 valores)
No programa principal do problema "Percurso divertido" existe uma função "listaDiversoes" que implementa o comando "L". Esta função lista na consola os nomes e as pontuações atribuídas às diversões visitadas pela criança. A ordem de listagem é a ordem de visita da criança. Note que caso não existam visitas realizadas pela criança, a função escreve na consola "Sem visitas.".
Implemente a função "listaDiversoes", a qual tem o seguinte cabeçalho/protótipo:

```
void listaDiversoes(percursoParque p);
```

FOLHA DE RESPOSTAS

Nome:_____**Número:**_____

Pegunta I

FOLHA DE RESPOSTAS

Nome:_____**Número:**_____

Pegunta II.a

Pergunta II.b

FOLHA DE RESPOSTAS

Nome:_____**Número:**_____

Pegunta II.b (continuação)

FOLHA DE RESPOSTAS

Nome:_____**Número:**_____

Pegunta III

ANEXO

TAD – sequencia

```
#ifndef _H_SEQUENCIA
#define _H_SEQUENCIA

/* Tipo de dados: sequencia ---> s1, s2, s3 ....*/
typedef struct _sequencia * sequencia;

/************************************************
criaSequencia - Criacao da instancia da estrutura associada a uma sequencia.
Parametros:
        cap - capacidade prevista da sequencia
Retorno: apontador para a  instancia criada
Pre-condicoes:
************************************************/
sequencia criaSequencia(int cap);

/************************************************
destroiSequencia - Liberta a memoria ocupada pela instancia da estrutura associada à sequencia.
Parametros:
        s - sequencia a destruir
Retorno:
Pre-condicoes: s != NULL
************************************************/
void destroiSequencia (sequencia s );

/************************************************
destroiSeqEElems - Liberta a memoria ocupada pela instancia da estrutura associada à sequencia e os elementos.
Parametros:
        s - sequencia a destruir
        destroi - função para destruição os elementos contidos na sequencia
Retorno:
Pre-condicoes: s != NULL
************************************************/
void destroiSeqElems(sequencia s, void (*destroi)(void *) );

/************************************************
vaziaSequencia - Indica se a sequencia está vazia.
Parametros:
        s - sequencia
Retorno: 1- caso sequencia vazia; 0 - caso contrário
Pre-condicoes: s != NULL
************************************************/
int vaziaSequencia(sequencia s);

/************************************************
tamanhoSequencia - Consulta o numero de elementos na sequencia.
Parametros:
        s - sequencia
Retorno: numero de elementos na sequencia
Pre-condicoes: s != NULL
************************************************/
int tamanhoSequencia(sequencia s);

/************************************************
elementoPosSequencia - Consulta o elemento i-esimo na sequencia.
Parametros:
        s - sequencia
        i - posicao na sequencia
```

```
Retorno: endereco do elemento
Pre-condicoes: s != NULL && i>0 && i<= tamanhoSequencia(s)
************************************************/
void * elementoPosSequencia(sequencia s, int i);

/************************************************
adicionaPosSequencia - Adiciona o elemento dado na posicao i da sequencia.
Parametros:
        s - sequencia
        elem - endereco do elemento
        i - posicao na sequencia
Retorno:
Pre-condicoes: s != NULL && i>0 && i<= tamanhoSequencia(s)+1
************************************************/
void adicionaPosSequencia(sequencia s, void * elem, int i);

/************************************************
removePosSequencia - Remove o elemento na posicao i da sequencia.
Parametros:
        s - sequencia
        i - posicao na sequencia
Retorno: retorna o elemento
Pre-condicoes: s != NULL && i>0 && i<= tamanhoSequencia(s)
************************************************/
void * removePosSequencia(sequencia s, int i);

/************************************************
iteradorSequencia - Cria e devolve um iterador para a sequencia.
Parametros:
        s - sequencia
Retorno: iterador da sequencia
Pre-condicoes: s != NULL && vaziaSequencia(s)!=1
************************************************/
iterador iteradorSequencia(sequencia s);

#endif
```

TAD – conjunto

```
#ifndef _H_CONJUNTO
#define _H_CONJUNTO

/* Tipo de dados: conjunto ---> os elementos não podem ser repetidos com base numa função de igualdade.*/
typedef struct _conjunto * conjunto;

/************************************************
criaConjunto - Criacao da instancia da estrutura associada a um conjunto.
Parametros:
    cap - capacidade prevista do conjunto
    igual – função de igualdade de elementos (retorna 1 se iguais, 0 caso contrario
Retorno: apontador para a  instancia criada
Pre-condicoes:
************************************************/
conjunto criaConjunto(int cap, int (* igual)(void *, void *));

/************************************************
destroiConjunto - Liberta a memoria ocupada pela instancia da estrutura associada ao conjunto.
Parametros: c - conjunto a destruir
Retorno:
Pre-condicoes: c != NULL
************************************************/
```

```
void destroiConjunto (conjunto c );

/***********************************************
destroiConjEElems - Liberta a memoria ocupada pela instancia da estrutura associada ao conjunto e os elementos.
Parametros:
        c - conjunto a destruir     destroi - função para destruição os elementos
Retorno:
Pre-condicoes: c != NULL
***********************************************/
void destroiConjEElems(conjunto c, void (*destroi)(void *) );

/***********************************************
vazioConjunto - Indica se o conjunto está vazio.
Parametros:
        c - conjunto
Retorno: 1- caso conjunto vazio; 0 - caso contrário
Pre-condicoes: c != NULL
***********************************************/
int vazioConjunto(conjunto c);

/***********************************************
tamanhoConjunto - Consulta o numero de elementos no conjunto.
Parametros:
        c - conjunto
Retorno: numero de elementos no conjunto
Pre-condicoes: c != NULL
***********************************************/
int tamanhoConjunto(conjunto c);

/***********************************************
existeElemConjunto – Indica se o elemento existe no conjunto.
Parametros:
c – conjunto
e – endereço do elemento
Retorno: retorna 1 se existir, e 0, caso contrário
Pre-condicoes: c != NULL
***********************************************/
int existeElemConjunto(conjunto c, void * e);

/***********************************************
adicionaElemConjunto - Adiciona o elemento dado no conjunto,caso exista.
Parametros:
        c – conjunto
        elem - endereco do elemento
Retorno: retorna 1 se adicionar e 0, caso contrário
Pre-condicoes: c != NULL
***********************************************/
int adicionaElemConjunto(conjunto c, void * elem);

/***********************************************
removeElemConjunto - Remove o elemento no conjunto, caso exista.
Parametros:
        c – conjunto
        elem - elemento
Retorno: Retorna o elemento, caso exista, ou NULL caso contrário
Pre-condicoes: c != NULL
***********************************************/
void * removeElemConjunto(conjunto c, void * elem);

/***********************************************
```

```
iteradorConjunto - Cria e devolve um iterador para o conjunto.
Parametros:
c - conjunto
Retorno: iterador do conjunto
Pre-condicoes: c != NULL && vazioConjunto(c)!=1
************************************************/
iterador iteradorConjunto(conjunto c);

#endif
```

TAD – dicionario

```
#ifndef _H_DICIONARIO
#define _H_DICIONARIO
/* Tipo de dados: dicionario ---> os elementos não podem ser repetidos com base num identificador (chave) dos
elementos */
typedef struct _dicionario * dicionario;

/************************************************
criaDicionario - Criacao da instancia da estrutura associada a um dicionario.
Parametros:
    cap - capacidade prevista do dicionario
    chave – função que retorna o endereço de uma cópia da chave do elemento
    TipoChave – 0 – inteiro, 1 - string
Retorno: apontador para a  instancia criada
Pre-condicoes:
************************************************/
dicionario criaDicionario(int cap, void * (* chave)(void *), int tipoChave);

/************************************************
destroiDicionario - Liberta a memoria ocupada pela instancia da estrutura associada ao dicionario.
Parametros: d - dicionario a destruir
Retorno:
Pre-condicoes: d != NULL
************************************************/
void destroiDicionario (dicionario d );

/************************************************
destroiDicEElems - Liberta a memoria ocupada pela instancia da estrutura associada ao dicionario e os elementos.
Parametros:
                d - dicionario a destruir    destroi - função para destruição os elementos
Retorno:
Pre-condicoes: d != NULL
************************************************/
void destroiDicEElems(dicionario d, void (*destroi)(void *) );

/************************************************
vazioDicionario - Indica se o dicionario está vazio.
Parametros:
                d - dicionario
Retorno: 1- caso dicionario vazio; 0 - caso contrário
Pre-condicoes: d != NULL
************************************************/
int vazioDicionario(dicionario d);

/************************************************
tamanhoDicionario - Consulta o numero de elementos no dicionario.
Parametros:
                d - dicionario
Retorno: numero de elementos no dicionario
Pre-condicoes: d != NULL
```

```
************************************************/
int tamanhoDicionario(dicionario d);

/************************************************
existeElemDicionario – Indica se o elemento  com a chave dada existe no dicionario.
Parametros:
        d – dicionario
        ch – endereço da chave do elemento
Retorno: retorna 1 se existir, e 0, caso contrário
Pre-condicoes: d != NULL
************************************************/
int existeElemDicionario(dicionario d, void * ch);

/************************************************
elementoDicionario - retorna o elemento no dicionario com a chave dada
Parametros:
        d – dicionario          ch - endereco da chave do elemento
Retorno: retorna o elemento
Pre-condicoes: d != NULL
************************************************/
void * elementoDicionario(dicionario d, void * ch);

/************************************************
adicionaElemDicionario - Adiciona o elemento dado no dicionario, se não existir um elemento com a mesma chave.
Parametros:
        d – dicionario
        elem - endereco do elemento
Retorno: Retorna 1 se adicionar, e 0, caso contrário
Pre-condicoes: d != NULL
************************************************/
int adicionaElemDicionario(dicionario d, void * elem);

/************************************************
removeElemDicionario - Remove o elemento com a chave dada no dicionario, se existir.
Parametros:
        d – dicionario          ch – endereco da chave
Retorno: Retorna o elemento, caso exista ou NULL, caso contrario
Pre-condicoes: d != NULL
************************************************/
void * removeElemDicionario(dicionario d, void * ch);

/************************************************
iteradorDicionario - Cria e devolve um iterador para o dicionario.
Parametros:
d - dicionario
Retorno: iterador do dicionario
Pre-condicoes: d != NULL && vazioDicionario(d)!=1
************************************************/
iterador iteradorDicionario(dicionario d);

#endif
```

TAD – fila

```
#ifndef _H_FILA
#define _H_FILA
/* Tipo de dados: fila ---> comportamento “Primeiro a chegar, Primeiro a sair” */
typedef struct _fila * fila;

/************************************************
criaFila - Criacao da instancia da estrutura associada a uma fila.
```

```
Parametros:
    cap - capacidade prevista da fila
Retorno: apontador para a  instancia criada
Pre-condicoes:
***********************************************/
fila criaFila(int cap);

/*destroiFila - Liberta a memoria ocupada pela instancia da estrutura associada à fila.
Parametros:
                f - fila a destruir
Retorno:
Pre-condicoes: f != NULL
***********************************************/
void destroiFila (fila f );

/***********************************************
destroiFilaEElems - Liberta a memoria ocupada pela instancia da estrutura associada à fila e os elementos.
Parametros:
                f - fila a destruir             destroi - função para destruição os elementos
Retorno:
Pre-condicoes: f != NULL
***********************************************/
void destroiFilaEElems(fila f, void (*destroi)(void *) );

/***********************************************
vaziaFila - Indica se a fila está vazia.
Parametros:
                f - fila
Retorno: 1- caso fila vazia; 0 - caso contrário
Pre-condicoes: f != NULL
***********************************************/
int vaziaFila(fila f);

/***********************************************
tamanhoFila - Consulta o numero de elementos na fila.
Parametros:
                f - fila
Retorno: numero de elementos na fila
Pre-condicoes: f != NULL
***********************************************/
int tamanhoFila(fila f);

/***********************************************
elementoFila – Retorna o elemento  à mais tempo na fila.
Parametros:
f – fila
Retorno: retorna o endereco do elemento
Pre-condicoes: f != NULL && tamanhoFila(f) != 0
***********************************************/
void * elementoFila(fila f);

/***********************************************
adicionaElemFila - Adiciona o elemento dado na fila.
Parametros:
                f – fila
                elem - endereco do elemento
Retorno:
Pre-condicoes: f != NULL
***********************************************/
void adicionaElemFila(fila f, void * elem);
```

```
/***********************************************
removeElemFila - Remove o elemento à mais tempo na fila.
Parametros:
        f – fila
Retorno: retorna o elemento
Pre-condicoes: f != NULL && vaziaFila(f) != 1
***********************************************/
void * removeElemFila(fila f);

#endif
```

TAD – pilha

```
#ifndef _H_PILHA
#define _H_PILHA

/* Tipo de dados: pilha ---> comportamento "Último a chegar, Primeiro a sair" */
typedef struct _pilha * pilha;

/***********************************************
criaPilha - Criacao da instancia da estrutura associada a uma pilha.
Parametros:
    cap - capacidade prevista da pilha
Retorno: apontador para a  instancia criada
Pre-condicoes:
***********************************************/
pilha criaPilha(int cap);

/***********************************************
destroiPilha - Liberta a memoria ocupada pela instancia da estrutura associada à pilha.
Parametros:
        p - pilha - fila a destruir
Retorno:
Pre-condicoes: p != NULL
***********************************************/
void destroiPilha (pilha p );

/***********************************************
destroipilhaEElems - Liberta a memoria ocupada pela instancia da estrutura associada à pilha e os elementos.
Parametros:
        p - pilha a destruir          destroi - função para destruição os elementos
Retorno:
Pre-condicoes: p != NULL
***********************************************/
void destroiPilhaEElems(pilha p, void (*destroi)(void *) );

/***********************************************
vaziaPilha - Indica se a pilha está vazia.
Parametros:
        p - pilha
Retorno: 1- caso pilha vazia; 0 - caso contrário
Pre-condicoes: p != NULL
***********************************************/
int vaziaPilha(pilha p);

/***********************************************
tamanhoPilha - Consulta o numero de elementos na pilha.
Parametros:
        p - pilha
Retorno: numero de elementos na pilha
```

```
Pre-condicoes: p != NULL
************************************************/
int tamanhoPilha(pilha p);

/************************************************
topoPilha – Retorna o elemento  à menos tempo na pilha (no topo).
Parametros:
p – pilha
Retorno: retorna o endereco do elemento
Pre-condicoes: p != NULL && tamanhoPilha(p) != 0
************************************************/
void * topoPilha(pilha p);

/************************************************
emPilha - Adiciona o elemento dado na pilha.
Parametros:
        p – pilha
        elem - endereco do elemento
Retorno:
Pre-condicoes: p != NULL
************************************************/
void emPilha(pilha p, void * elem);

/************************************************
desemPilha - Remove o elemento à menos tempo na pilha (no topo).
Parametros:
        p – pilha
Retorno: retorna o elemento
Pre-condicoes: p != NULL && vaziaPilha(p) != 1
************************************************/
void * desemPilha(pilha p);

#endif
```

TAD – iterador

```
#ifndef _H_ITERADOR
#define _H_ITERADOR

/* Tipo de dados: iterador */
typedef struct _iterador * iterador;

/* Prototipos das funcoes associadas a um iterador */

/************************************************
criaIterador - Criacao da instancia da estrutura associada a um iterador para um vector com n elementos.
Parametros:
        vector - endereco do vector a iterar
        n - numero de elementos no vector
Retorno: apontador para a  instancia criada
Pre-condicoes:  vector != NULL && n > 0
************************************************/
iterador criaIterador(void ** vector, int n);

/************************************************
destroiIterador - Liberta a memoria ocupada pela instancia da estrutura associada ao iterador.
Parametros:
        it - iterador a destruir
Retorno:
Pre-condicoes: it != NULL
************************************************/
```

```
void destroiIterador (iterador it);

/***********************************************
temSeguinteIterador - Indica se existe mais elementos para iterar no iterador.
Parametros:
                it - iterador
Retorno: 1- caso exista mais elementos; 0- caso contrario
Pre-condicoes: it != NULL
***********************************************/
int temSeguinteIterador(iterador it);

/***********************************************
seguinteIterador - Consulta o seguinte elemento no iterador.
Parametros:
                it - iterador
Retorno: enderco do elemento
Pre-condicoes: it != NULL && temSeguinteIterador(it) == 1
***********************************************/
void * seguinteIterador(iterador it);

#endif
```