```
/***************************************************************************
*  $MCI Módulo de implementação: LIS  Lista duplamente encadeada
*
*  Arquivo gerado:              LISTA.c
*  Letras identificadoras:      LIS
*
*  Nome da base de software:    Arcabouço para a automação de testes de
programas redigidos em C
*  Arquivo da base de software: D:\AUTOTEST\PROJETOS\LISTA.BSW
*
*  Projeto: Trabalho 2 - Programação Modular
*  Autores: avs - Arndt von Staa
*           GB - Gustavo Bach
*           JG - João Lucas Gardenberg
*           MV - Michelle Valente
*
*  $HA Histórico de evolução:
*     Versão  Autor      Data          Observações
*     5.00    GM,JG,MV   12/abr/2014   todas as funções exportadas pelo
módulo retornam
*                                      condições de retorno
*     5.00    GB,JG,MV   11/abr/2014   adicionar função de alterar valor do
elemento
*     4.00    avs        01/fev/2006   criar linguagem script simbólica
*     3.00    avs        08/dez/2004   uniformização dos exemplos
*     2.00    avs        07/jul/2003   unificação de todos os módulos em um
só projeto
*     1.00    avs        16/abr/2003   início desenvolvimento
*
***************************************************************************/

#include   <stdio.h>
#include   <string.h>
#include   <memory.h>
#include   <malloc.h>
#include   <assert.h>

#define LISTA_OWN
#include "LISTA.h"
#undef LISTA_OWN

/***************************************************************************
*
*  $TC Tipo de dados: LIS Elemento da lista
*
*
***************************************************************************/

   typedef struct tagElemLista {
```

```
         void * pValor ;
               /* Ponteiro para o valor contido no elemento */

         struct tagElemLista * pAnt ;
               /* Ponteiro para o elemento predecessor */

         struct tagElemLista * pProx ;
               /* Ponteiro para o elemento sucessor */

   } tpElemLista ;

/***************************************************************************
*
*  $TC Tipo de dados: LIS Descritor da cabeça de lista
*
*
***************************************************************************/

   typedef struct LIS_tagLista {

         tpElemLista * pOrigemLista ;
               /* Ponteiro para a origem da lista */

         tpElemLista * pFimLista ;
               /* Ponteiro para o final da lista */

         tpElemLista * pElemCorr ;
               /* Ponteiro para o elemento corrente da lista */

         int numElem ;
               /* Número de elementos da lista */

         void ( * ExcluirValor ) ( void * pValor ) ;
               /* Ponteiro para a função de destruição do valor contido em um
elemento */

   } LIS_tpLista ;

/***** Protótipos das funções encapsuladas no módulo *****/

   static void LiberarElemento( LIS_tppLista pLista ,
                                tpElemLista  * pElem   ) ;

   static tpElemLista * CriarElemento( LIS_tppLista pLista ,
                                       void *       pValor  ) ;

/*****  Código das funções exportadas pelo módulo  *****/
```

```
/***************************************************************************
*
*  Função: LIS  &Criar lista
*  ****/

   LIS_tpCondRet LIS_CriarLista( LIS_tppLista * pLista,
             void   ( * ExcluirValor ) ( void * pDado ) )
   {

      if ( *pLista != NULL )
      {
              LIS_DestruirLista( *pLista ) ;
      } /* if */

      *pLista = ( LIS_tpLista * ) malloc( sizeof( LIS_tpLista ) ) ;
      if ( *pLista == NULL )
      {
         return LIS_CondRetFaltouMemoria ;
      } /* if */

      (*pLista)->pOrigemLista = NULL ;
      (*pLista)->pFimLista = NULL ;
      (*pLista)->pElemCorr = NULL ;
      (*pLista)->numElem   = 0 ;

      (*pLista)->ExcluirValor = ExcluirValor ;

      return LIS_CondRetOK ;

   } /* Fim função: LIS  &Criar lista */

/***************************************************************************
*
*  Função: LIS  &Destruir lista
*  ****/

   LIS_tpCondRet LIS_DestruirLista( LIS_tppLista pLista )
   {

      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

        if( pLista != NULL )
        {
             LIS_tpCondRet CondRet ;
```

```
         CondRet = LIS_EsvaziarLista( pLista ) ;
             if( CondRet != LIS_CondRetOK )
             {
                   return CondRet ;
             } /* if */
         } /* if */

      free( pLista ) ;

        pLista = NULL ;

        return LIS_CondRetOK ;

   } /* Fim função: LIS  &Destruir lista */

/***************************************************************************
*
*  Função: LIS  &Esvaziar lista
*  ****/

   LIS_tpCondRet LIS_EsvaziarLista( LIS_tppLista pLista )
   {

      tpElemLista * pElem ;
      tpElemLista * pProx ;

      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

        if( pLista == NULL )
        {
              return LIS_CondRetListaNaoExiste ;
        } /* if */

      pElem = pLista->pOrigemLista ;
      while ( pElem != NULL )
      {
         pProx = pElem->pProx ;
         LiberarElemento( pLista , pElem ) ;
         pElem = pProx ;
      } /* while */

      pLista->pOrigemLista = NULL ;
      pLista->pFimLista = NULL ;
      pLista->pElemCorr = NULL ;
      pLista->numElem   = 0 ;
```

```
         return LIS_CondRetOK ;

   } /* Fim função: LIS  &Esvaziar lista */

/***************************************************************************
*
*  Função: LIS  &Inserir elemento antes
*  ****/

   LIS_tpCondRet LIS_InserirElementoAntes( LIS_tppLista pLista ,
                                           void * pValor        )
   {

      tpElemLista * pElem ;

      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

      /* Criar elemento a inerir antes */

         pElem = CriarElemento( pLista , pValor ) ;
         if ( pElem == NULL )
         {
            return LIS_CondRetFaltouMemoria ;
         } /* if */

      /* Encadear o elemento antes do elemento corrente */

         if ( pLista->pElemCorr == NULL )
         {
            pLista->pOrigemLista = pElem ;
            pLista->pFimLista = pElem ;
         } else
         {
            if ( pLista->pElemCorr->pAnt != NULL )
            {
               pElem->pAnt  = pLista->pElemCorr->pAnt ;
               pLista->pElemCorr->pAnt->pProx = pElem ;
            } else
            {
               pLista->pOrigemLista = pElem ;
            } /* if */

            pElem->pProx = pLista->pElemCorr ;
            pLista->pElemCorr->pAnt = pElem ;
         } /* if */
```

```
         pLista->pElemCorr = pElem ;

         return LIS_CondRetOK ;

   } /* Fim função: LIS  &Inserir elemento antes */

/***************************************************************************
*
*  Função: LIS  &Inserir elemento após
*  ****/

   LIS_tpCondRet LIS_InserirElementoApos( LIS_tppLista pLista ,
                                          void * pValor        )

   {

      tpElemLista * pElem ;

      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

      /* Criar elemento a inerir após */

         pElem = CriarElemento( pLista , pValor ) ;
         if ( pElem == NULL )
         {
            return LIS_CondRetFaltouMemoria ;
         } /* if */

      /* Encadear o elemento após o elemento */

         if ( pLista->pElemCorr == NULL )
         {
            pLista->pOrigemLista = pElem ;
            pLista->pFimLista = pElem ;
         } else
         {
            if ( pLista->pElemCorr->pProx != NULL )
            {
               pElem->pProx  = pLista->pElemCorr->pProx ;
               pLista->pElemCorr->pProx->pAnt = pElem ;
            } else
            {
               pLista->pFimLista = pElem ;
            } /* if */

            pElem->pAnt = pLista->pElemCorr ;
```

```
            pLista->pElemCorr->pProx = pElem ;

         } /* if */

         pLista->pElemCorr = pElem ;

         return LIS_CondRetOK ;

   } /* Fim função: LIS  &Inserir elemento após */

/***************************************************************************
*
*  Função: LIS  &Excluir elemento
*  ****/

   LIS_tpCondRet LIS_ExcluirElemento( LIS_tppLista pLista )
   {

      tpElemLista * pElem ;

      #ifdef _DEBUG
         assert( pLista  != NULL ) ;
      #endif

      if ( pLista->pElemCorr == NULL )
      {
         return LIS_CondRetListaVazia ;
      } /* if */

      pElem = pLista->pElemCorr ;

      /* Desencadeia à esquerda */

         if ( pElem->pAnt != NULL )
         {
            pElem->pAnt->pProx   = pElem->pProx ;
            pLista->pElemCorr    = pElem->pAnt ;
         } else {
            pLista->pElemCorr    = pElem->pProx ;
            pLista->pOrigemLista = pLista->pElemCorr ;
         } /* if */

      /* Desencadeia à direita */

         if ( pElem->pProx != NULL )
         {
            pElem->pProx->pAnt = pElem->pAnt ;
         } else
```

```
         {
            pLista->pFimLista = pElem->pAnt ;
         } /* if */

      LiberarElemento( pLista , pElem ) ;

      return LIS_CondRetOK ;

   } /* Fim função: LIS  &Excluir elemento */

/***************************************************************************
*
*  Função: LIS  &Obter referência para o valor contido no elemento
*  ****/

   LIS_tpCondRet LIS_ObterValor( LIS_tppLista pLista,
                                 void ** pValor         )
   {

      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

        if ( pLista == NULL )
      {
         return LIS_CondRetListaNaoExiste ;
      } /* if */

      if ( pLista->pElemCorr == NULL )
      {
          *pValor = NULL ;
        return LIS_CondRetListaVazia ;
      } /* if */

      *pValor = pLista->pElemCorr->pValor ;

        return LIS_CondRetOK ;

   } /* Fim função: LIS  &Obter referência para o valor contido no elemento
*/

/***************************************************************************
*
*  Função: LIS  &Ir para o elemento inicial
*  ****/

   LIS_tpCondRet LIS_IrInicioLista( LIS_tppLista pLista )
   {
```

```
      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

      if( pLista == NULL )
        {
              return LIS_CondRetListaNaoExiste ;
        } /* if */

        if( pLista->pElemCorr == NULL )
        {
              return LIS_CondRetListaVazia ;
        } /* if */

      pLista->pElemCorr = pLista->pOrigemLista ;

        return LIS_CondRetOK ;

   } /* Fim função: LIS  &Ir para o elemento inicial */

/***************************************************************************
*
*  Função: LIS  &Ir para o elemento final
*  ****/

   LIS_tpCondRet LIS_IrFinalLista( LIS_tppLista pLista )
   {

      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

      if( pLista == NULL )
        {
              return LIS_CondRetListaNaoExiste ;
        } /* if */

        if( pLista->pElemCorr == NULL )
        {
              return LIS_CondRetListaVazia ;
        } /* if */

      pLista->pElemCorr = pLista->pFimLista ;

        return LIS_CondRetOK ;

   } /* Fim função: LIS  &Ir para o elemento final */
```

```
/***************************************************************************
*
*  Função: LIS  &Avançar elemento
*  ****/

   LIS_tpCondRet LIS_AvancarElementoCorrente( LIS_tppLista pLista ,
                                              int numElem )
   {

      int i ;

      tpElemLista * pElem ;

      #ifdef _DEBUG
         assert( pLista != NULL ) ;
      #endif

      /* Tratar lista vazia */

         if ( pLista->pElemCorr == NULL )
         {

            return LIS_CondRetListaVazia ;

         } /* fim ativa: Tratar lista vazia */

      /* Tratar avançar para frente */

         if ( numElem > 0 )
         {

            pElem = pLista->pElemCorr ;
            for( i = numElem ; i > 0 ; i-- )
            {
               if ( pElem == NULL )
               {
                  break ;
               } /* if */
               pElem    = pElem->pProx ;
            } /* for */

            if ( pElem != NULL )
            {
               pLista->pElemCorr = pElem ;
               return LIS_CondRetOK ;
            } /* if */
```

```
            pLista->pElemCorr = pLista->pFimLista ;
            return LIS_CondRetFimLista ;

         } /* fim ativa: Tratar avançar para frente */

      /* Tratar avançar para trás */

         else if ( numElem < 0 )
         {

            pElem = pLista->pElemCorr ;
            for( i = numElem ; i < 0 ; i++ )
            {
               if ( pElem == NULL )
               {
                  break ;
               } /* if */
               pElem    = pElem->pAnt ;
            } /* for */

            if ( pElem != NULL )
            {
               pLista->pElemCorr = pElem ;
               return LIS_CondRetOK ;
            } /* if */

            pLista->pElemCorr = pLista->pOrigemLista ;
            return LIS_CondRetFimLista ;

         } /* fim ativa: Tratar avançar para trás */

      /* Tratar não avançar */

         return LIS_CondRetOK ;

   } /* Fim função: LIS  &Avançar elemento */

/***************************************************************************
*
*  Função: LIS  &Procurar elemento contendo valor
*  ****/

   LIS_tpCondRet LIS_ProcurarValor( LIS_tppLista pLista ,
                                    void * pValor       )
   {

      tpElemLista * pElem ;
```

```
      #ifdef _DEBUG
         assert( pLista  != NULL ) ;
      #endif

      if ( pLista->pElemCorr == NULL )
      {
         return LIS_CondRetListaVazia ;
      } /* if */

      for ( pElem  = pLista->pElemCorr ;
            pElem != NULL ;
            pElem  = pElem->pProx )
      {
         if ( pElem->pValor == pValor )
         {
            pLista->pElemCorr = pElem ;
            return LIS_CondRetOK ;
         } /* if */
      } /* for */

      return LIS_CondRetNaoAchou ;

   } /* Fim função: LIS  &Procurar elemento contendo valor */

/***************************************************************************
*
*  Função: LIS  &Alterar valor de um elemento
*  ****/

   LIS_tpCondRet LIS_AlterarValor( LIS_tppLista pLista,
                                   void * pValor          )
   {

         if( pLista == NULL )
         {
               return LIS_CondRetListaNaoExiste ;
         } /* if */

         if( pLista->pElemCorr == NULL || pLista->pOrigemLista == NULL )
         {
               return LIS_CondRetListaVazia ;
         } /* if */

         pLista->pElemCorr->pValor = pValor ;

         return LIS_CondRetOK ;

   } /* Fim função: LIS  &Alterar valor de um elemento */
```

```
/*****  Código das funções encapsuladas no módulo  *****/


/***************************************************************************
*
*  $FC Função: LIS  -Liberar elemento da lista
*
*  $ED Descrição da função
*     Elimina os espaços apontados pelo valor do elemento e o
*     próprio elemento.
*
***************************************************************************/

   void LiberarElemento( LIS_tppLista pLista ,
                         tpElemLista  * pElem   )
   {

      if ( ( pLista->ExcluirValor != NULL )
        && ( pElem->pValor != NULL        ))
      {
         pLista->ExcluirValor( pElem->pValor ) ;
      } /* if */

      free( pElem ) ;

      pLista->numElem-- ;

   } /* Fim função: LIS  -Liberar elemento da lista */


/***************************************************************************
*
*  $FC Função: LIS  -Criar o elemento
*
***************************************************************************/

   tpElemLista * CriarElemento( LIS_tppLista pLista ,
                                void *       pValor  )
   {

      tpElemLista * pElem ;

      pElem = ( tpElemLista * ) malloc( sizeof( tpElemLista )) ;
      if ( pElem == NULL )
      {
         return NULL ;
```

```
      } /* if */

      pElem->pValor = pValor ;
      pElem->pAnt   = NULL  ;
      pElem->pProx  = NULL  ;

      pLista->numElem ++ ;

      return pElem ;

   } /* Fim função: LIS  -Criar o elemento */

/********** Fim do módulo de implementação: LIS  Lista duplamente encadeada
**********/
```