Algoritmos y Programación II
75.41

Cátedra: Lic. Gustavo Carolo

Guía de Estudio – Listas, Pilas y Colas

Enero 2005

Índice

Desarrollo De Tipos De Datos Abstractos

Explicación General

En la primera parte de la materia se desarrollan los tipos abstractos Listas simples, Pilas y Colas. Se definen las primitivas de estas estructuras. Luego se desarrollan dos formas de implementación: usando estructuras estáticas (cursores), y usando estructuras dinámicas (utilizando punteros):

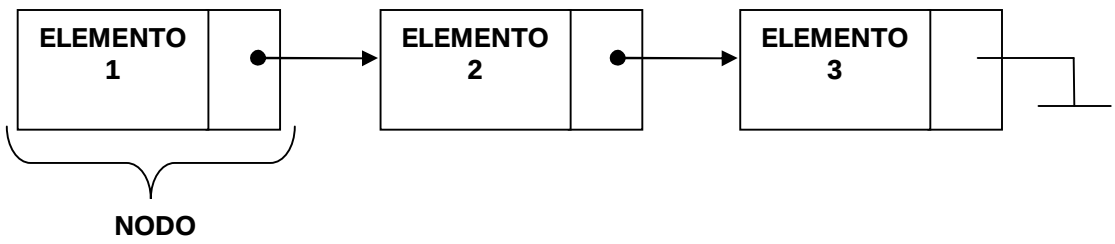
- Estructura estática con Cursores "embebidos" en la implementación de la estructura abstracta
- Estructura dinámica con punteros.

Nivel de Aplicación y Nivel de Implementación

En el nivel de aplicación de cada estructura abstracta se definen las primitivas: los tipos de elementos que usan, las pre y poscondiciones que explican su funcionamiento para el desarrollo de procedimientos abstractos a su implementación.
En el nivel de Implementación se codifican las primitivas de manera de lograr el funcionamiento esperado según las definiciones realizadas en el nivel de aplicación. Puede ser necesario definir nuevos tipos de datos y nuevas funciones y o procedimientos que solo surgen por la implementación elegida.
El alumno puede notar estas diferencias al comparar las dos implementaciones que se desarrollan para cada estructura.

Nodo

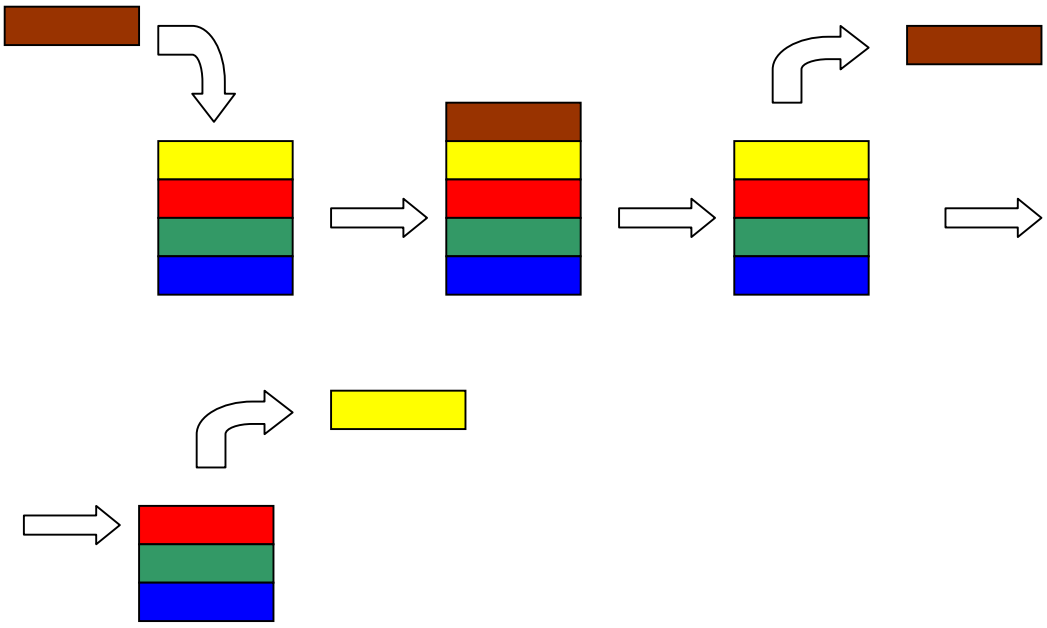
Para la implementación de TDA's como ser listas, pilas o colas, es necesario utilizar nodos. Un nodo es una estructura que contiene un elemento (El dato que se desea guardar en la estructura), y una referencia a otro nodo. De esta forma es posible encadenar los nodos pudiendo guardar una cantidad ilimitada de elementos.



Tipo De Dato Abstracto Pila

Definición

Una pila es una estructura de datos en la cual los elementos almacenados en la misma se agregan y se sacan del mismo lugar, llamado el tope de la pila. El tope es el único lugar a partir del cual se pueden acceder a los elementos de la estructura. Esta característica hace que el último elemento en ser insertado en la pila es el primero en salir. Este tipo de estructuras se denominan LIFO (Last In First Out).



Primitivas

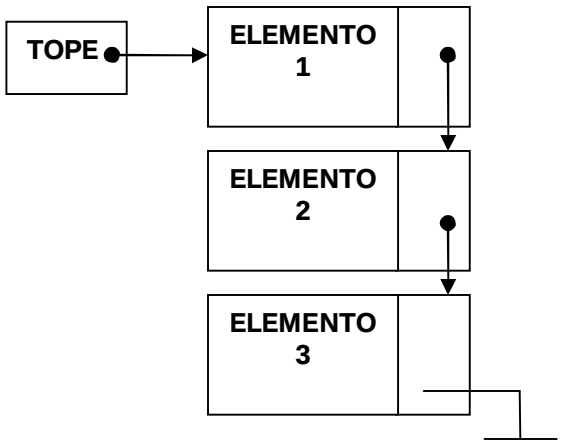
Para utilizar el Tipo de Dato Abstracto (TDA) Pila, el mismo nos proveerá de una serie de procedimientos que nos permitirán acceder o agregar elementos.
Los siguientes son los procedimientos básicos que debe contener una pila:

- P_Crear
- P_Vaciar
- P_Vacia
- P_Agregar
- P_Sacar

El usuario de una pila utilizara estos procedimientos, sin tener en cuenta como están implementados por dentro. Lo único que deberá conocer es los procedimientos y sus parámetros.

Implementación

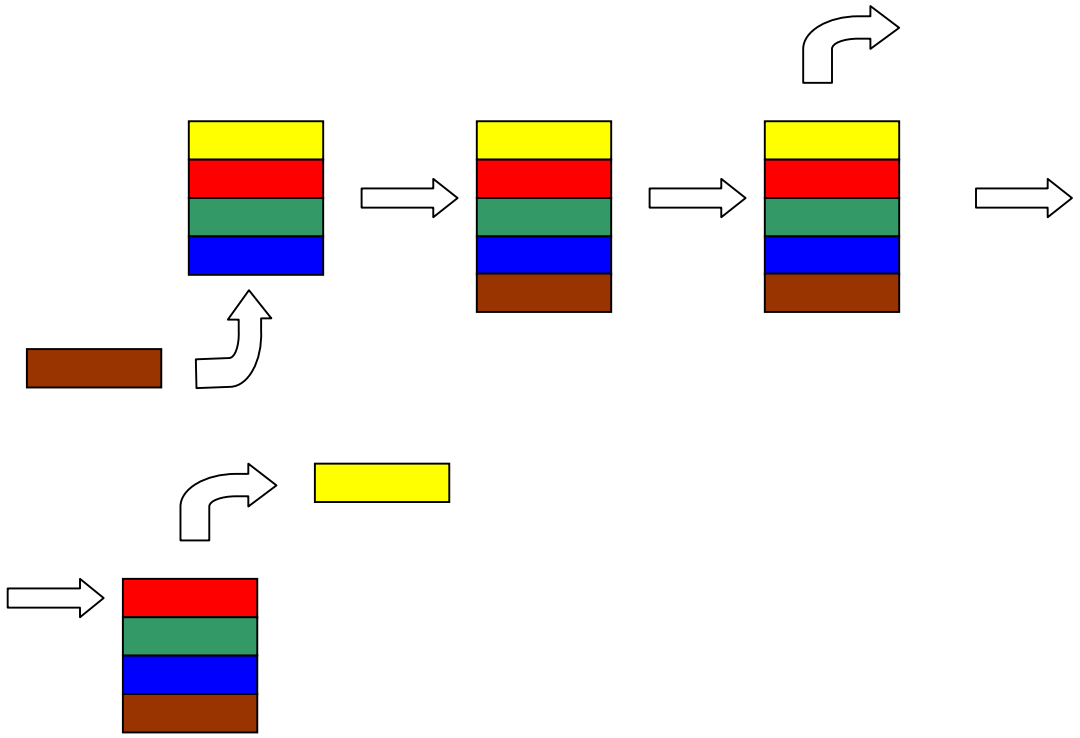
Para la implementación de la pila, es necesario que el tipo de dato contenga una referencia al nodo tope de la pila. Luego, cada nodo tendrá una referencia al nodo que le siguiente. De esta manera se formara una cadena con inicio en el nodo tope y que finaliza en el último elemento de la pila, cuyo nodo no referenciará a ningún elemento.



Tipo de Dato Abstracto Cola

Definición

Una cola es una estructura de datos en la cual los elementos almacenados en la misma se agregan al final y se sacan del principio de la cola. Esta característica hace que el primer elemento insertado en la cola es el primero en salir, como en cualquier cola de la realidad (en un banco, en el cine, en el colectivo). Este tipo de estructuras se denominan FIFO (First In First Out).



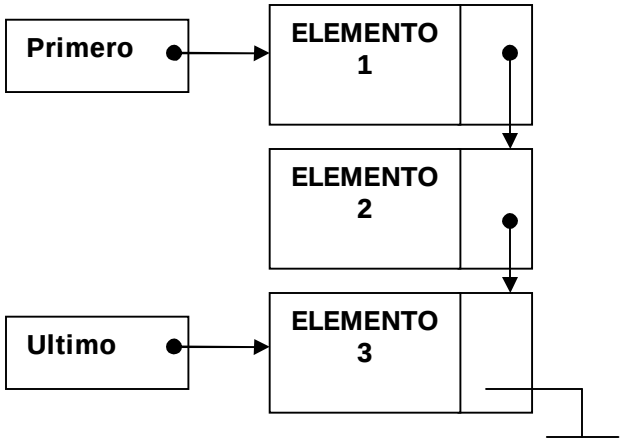
Primitivas

Para utilizar el Tipo de Dato Abstracto (TDA) Pila, el mismo nos proveerá de una serie de procedimientos que nos permitirán acceder o agregar elementos.
Los siguientes son los procedimientos básicos que debe contener una pila:

- C_Crear
- C_Vaciar
- C_Vacia
- C_Agregar
- C_Sacar

Implementación

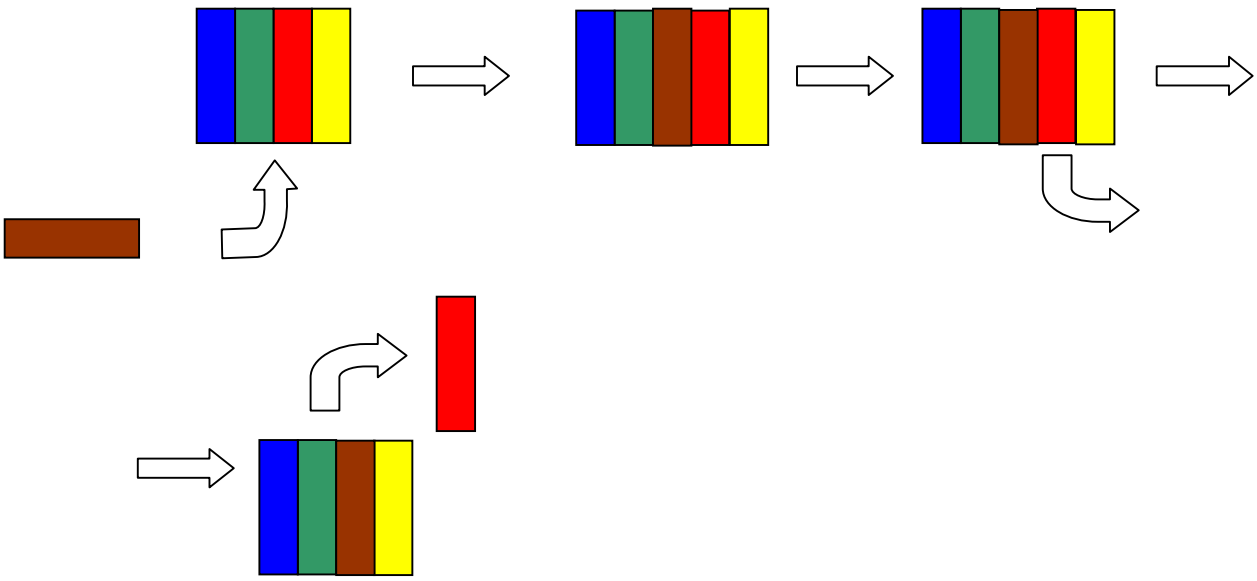
Para implementar una cola será necesario que la estructura contenga una referencia al primer nodo de la cola y otra al último nodo. Luego, desde el primer nodo de la cola, se irán encadenando los demás nodos, hasta el último.



Tipo de Dato Abstracto Lista

Definición

Una lista es una estructura de datos en la cual los elementos almacenados en la misma pueden ser agregados, borrados y accedidos sin restricciones, en cualquier punto de la estructura. A diferencia de las pilas y las colas, en las listas se pueden ver todos los elementos de la estructura, permitiendo realizar recorridos y consultas de los datos. De la estructura de una lista se distinguen dos elementos: el principio, a partir del cual se inician las búsquedas y recorridos; y el corriente, elemento de referencia en la lista, a partir del cual se realizan borrados, inserciones y modificaciones.

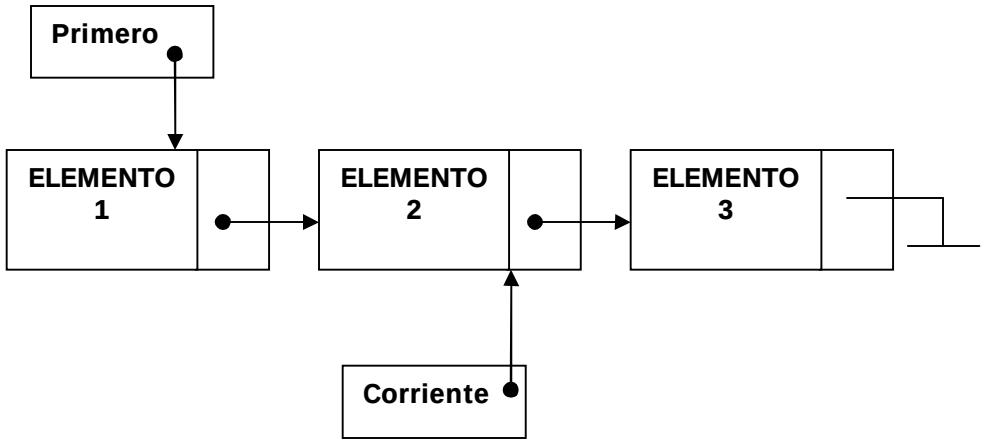


Primitivas

- ls_Crear
- ls_Vaciar
- ls_Vacia
- ls_ElemCorriente
- ls_ModifCorriente
- ls_MoverCorriente
- ls_BorrarCorriente
- ls_Insertar

Implementación

El tipo de dato abstracto lista, contendrá dos referencias a nodos de la lista. La primera corresponderá al primer nodo de la lista. La otra hará referencia al elemento actual o corriente de la lista. Luego, los nodos de la lista se irán encadenando uno a uno hasta el último elemento. Hay que tener en cuenta que el elemento corriente puede ser cualquiera de la lista, y que ira variando según la primitiva que se utilice.



Implementación De Estructuras Con Punteros

Punteros

Un puntero es una variable que contiene una dirección de memoria. Esta dirección es la posición de otra variable. Se dice que la variable puntero "apunta a la segunda". Es utilizado como los cursores para la implementación de estructuras encadenadas, por ejemplo, listas, pilas, colas, árboles, etc. A diferencia del cursor, el alocamiento de la memoria se realiza al momento de necesitarse el espacio, cuando va a ser ocupado, por eso se denomina alocamiento "dinámico".

Representación

Con cada elemento de la estructura encadenada se formará un nodo, con el dato propiamente dicho y el/los puntero/s al siguiente elemento según la lógica del encadenamiento. La estructura en general estará formada por los punteros generales necesarios para la ubicación de los primeros elementos y luego se irán encadenando. Los punteros y elementos que se incorporen a la definición de la estructura general dependerán de la lógica de la misma. Veremos a continuación la definición de listas, pilas y colas simples como ejemplos.

Implementación del TDA Pila utilizando punteros

PILA . H

```
#ifndef __PILA_H__
   #define __PILA_H__

   #if !defined(NULL)
      #define NULL 0
   #endif

   #if !defined(FALSE)
      #define FALSE 0
   #endif

   #if !defined(TRUE)
      #define TRUE 1
   #endif

   typedef struct TNodoPila
   {
      void* Elem;
      struct TNodoPila *Siguiente;
   } TNodoPila;

   typedef struct
   {
      TNodoPila *Tope;
      int TamanioDato;
   } TPila;

   /*P_Crear
   Pre: P no fue creada.
   Post: P creada y vacía. */
   void P_Crear(TPila *pP, int TamanioDato);

   /*P_Vaciar
   Pre: P creada.
   Post: P vacía. */
   void P_Vaciar(TPila *pP);

   /*P_Vacia
   Pre: P creada.
   Post: Si P está vacía devuelve TRUE, sino FALSE. */
   int P_Vacia(TPila P);

   /*P_Agregar
   Pre: P creada.
   Post: E se agregó como Tope de P.
   Devuelve FALSE si no pudo agregar E, sino devuelve TRUE.*/
   int P_Agregar(TPila *pP, void* pE);

   /*P_Sacar
   Pre: P creada y no vacia.
   Post: Se extrajo de P el tope y se devuelve en E.
   Si no pudo extraer el elemento (porque la pila estaba vacía) devuelve FALSE,
   sino devuelve TRUE.*/
   int P_Sacar(TPila *pP, void* pE);
```

```
#endif
```

PILA . C

```
#include "Pila.h"
#include <malloc.h>
#include <memory.h>

/*P_Crear
Pre: P no fue creada.
Post: P creada y vacía. */
void P_Crear(TPila *pP, int TamanioDato)
{
   pP->Tope = NULL;
   pP->TamanioDato = TamanioDato;
}

/*P_Vaciar
Pre: P creada.
Post: P vacía. */
void P_Vaciar(TPila *pP)
{
   TNodoPila *pAux = pP->Tope;
   TNodoPila *pSig;
   while (pAux)
   {
      pSig = pAux->Siguiente;
      free(pAux->Elem);
      free(pAux);
      pAux = pSig;
   }
   pP->Tope= NULL;
}

/*P_Vacia
Pre: P creada.
Post: Si P está vacía devuelve TRUE, sino FALSE. */
int P_Vacia(TPila P)
{
   return (P.Tope==NULL);
}

/*P_Agregar
Pre: P creada.
Post: E se agregó como Tope de P.
Devuelve FALSE si no pudo agregar E, sino devuelve TRUE.*/
int P_Agregar(TPila *pP, void* pE)
{
   TNodoPila *pNodo = (TNodoPila*) malloc(sizeof(TNodoPila));
   if (!pNodo)
      return FALSE;
   else
   {
      pNodo->Siguiente = pP->Tope;
      pP->Tope = pNodo;
      pNodo->Elem = malloc (pP->TamanioDato);
      memcpy(pNodo->Elem, pE, pP->TamanioDato);
      return TRUE;
   }
}
```

```c
/*P_Sacar
Pre: P creada y no vacia.
Post: Se extrajo de P el tope y se devuelve en E.
Si no pudo extraer el elemento (porque la pila estaba vacía) devuelve FALSE,
sino devuelve TRUE.*/
int P_Sacar(TPila *pP, void* pE)
{
   TNodoPila *pAux = pP->Tope;
   pP->Tope = pP->Tope->Siguiente;
   memcpy(pE, pAux->Elem, pP->TamanioDato);
   free(pAux->Elem);
   free(pAux);
   return TRUE;
}
```

Implementación del TDA Cola utilizando punteros

COLA . H

```
#ifndef __COLA_H__

   #define __COLA_H__

   #if !defined(NULL)
      #define NULL 0
   #endif

   #if !defined(FALSE)
      #define FALSE 0
   #endif

   #if !defined(TRUE)
      #define TRUE 1
   #endif

   typedef struct TNodoCola
   {
      void* Elem;
      struct TNodoCola *Siguiente;
   } TNodoCola;

   typedef struct
   {
      TNodoCola *Primero, *Ultimo;
      int TamanioDato;
   } TCola;

   /*C_Crear
   Pre: C no fue creada.
   Post: C creada y vacía. */
   void C_Crear(TCola *pC, int TamanioDato);

   /*C_Vaciar
   Pre: C creada.
   Post: C vacía. */
   void C_Vaciar(TCola *pC);

   /*C_Vacia
   Pre: C creada.
   Post: Si C está vacía devuelve TRUE, sino FALSE. */
   int C_Vacia(TCola C);

   /*C_Agregar
   Pre: C creada.
   Post: E se agregó como último elemento de C.
   Devuelve FALSE si no pudo agregar E, sino devuelve TRUE.*/
   int C_Agregar(TCola *pC, void* pE);

   /*C_Sacar
   Pre: C creada y no vacia.
   Post: Se extrajo de C el primer elemento y se devuelve en E.
   Si no pudo extraer el elemento (porque la cola estaba vacía) devuelve FALSE,
   sino devuelve TRUE.*/
   int C_Sacar(TCola *pC, void* pE);
```

```
#endif
```

COLA . C

```
#include "Cola.h"
#include <malloc.h>
#include <memory.h>

/*C_Crear
Pre: C no fue creada.
Post: C creada y vacía. */
void C_Crear(TCola *pC, int TamanioDato)
{
   pC->Primero = pC->Ultimo = NULL;
   pC->TamanioDato = TamanioDato;
}

/*C_Vaciar
Pre: C creada.
Post: C vacía. */
void C_Vaciar(TCola *pC)
{
   TNodoCola *pAux = pC->Primero;
   TNodoCola *pSig;
   while (pAux)
   {
      pSig = pAux->Siguiente;
      free(pAux->Elem);
      free(pAux);
      pAux = pSig;
   }
   pC->Primero = pC->Ultimo = NULL;
}

/*C_Vacia
Pre: C creada.
Post: Si C está vacía devuelve TRUE, sino FALSE. */
int C_Vacia(TCola C)
{
   return (C.Primero==NULL);
}

/*C_Agregar
Pre: C creada.
Post: E se agregó como último elemento de C.
Devuelve FALSE si no pudo agregar E, sino devuelve TRUE.*/
int C_Agregar(TCola *pC, void* pE)
{
   TNodoCola *pNodo = (TNodoCola*) malloc(sizeof(TNodoCola));
   if (!pNodo)
      return FALSE;
   else
   {
      if (pC->Ultimo)
         pC->Ultimo->Siguiente = pNodo;
      if (!pC->Primero) //Está vacía
         pC->Primero = pNodo;
      pNodo->Siguiente = NULL;
      pC->Ultimo = pNodo;
```

```
      pNodo->Elem = malloc (pC->TamanioDato);
      memcpy(pNodo->Elem, pE, pC->TamanioDato);
      return TRUE;
   }
}

/*C_Sacar
Pre: C creada y no vacia.
Post: Se extrajo de C el primer elemento y se devuelve en E.
Si no pudo extraer el elemento (porque la cola estaba vacía) devuelve FALSE,
sino devuelve TRUE.*/
int C_Sacar(TCola *pC, void* pE)
{
   TNodoCola *pAux = pC->Primero;
   pC->Primero = pC->Primero->Siguiente;
   if (!pC->Primero)
      pC->Ultimo = NULL;
   memcpy(pE, pAux->Elem, pC->TamanioDato);
   free(pAux->Elem);
   free(pAux);
   return TRUE;
}
```

Implementación del TDA Lista utilizando punteros

LISTASIMPLE . H

```
#ifndef __ListaSimple_h__
   #define __ListaSimple_h__

   #if !defined(NULL)
      #define NULL 0
   #endif

   #if !defined(FALSE)
      #define FALSE 0
   #endif

   #if !defined(TRUE)
      #define TRUE 1
   #endif

   typedef enum
   {
      LS_PRIMERO,
      LS_SIGUIENTE,
      LS_ANTERIOR
   } TMovimiento_Ls;

   typedef struct TNodoSimple
   {
      void* Elem;
      struct TNodoSimple *Siguiente;
   } TNodoSimple;

   typedef struct
   {
      TNodoSimple *Primero, *Corriente;
      int TamanioDato;
   } TListaSimple;

   /* ls_Crear
   Pre: Ls no fue creada.
   Post: Ls creada y vacía */
   void ls_Crear(TListaSimple *pLs, int TamanioDato);

   /* ls_Vaciar
   Pre: Ls creada.
   Post: Ls vacía.*/
   void ls_Vaciar(TListaSimple *pLs);

   /* ls_Vacia
   Pre: Ls creada.
   Post: Si Ls tiene elementos devuelve FALSE sino TRUE.*/
   int ls_Vacia(TListaSimple Ls);

   /* ls_ElemCorriente
   Pre: Ls creada y no vacía.
   Post: Se devuelve en E el elemento corriente de la lista.*/
   void ls_ElemCorriente(TListaSimple Ls, void* pE);
```

```
   /* ls_ModifCorriente
   Pre: Ls creada y no vacía.
   Post: El contenido del elemento actual quedo actualizado con E. */
   void ls_ModifCorriente(TListaSimple *pLs, void* pE);

   /* ls_MoverCorriente
   Pre: Ls creada y no vacía.
   Post: Si Ls está vacía, devuelve FALSE. Sino:
   Si M = LS_PRIMERO, el nuevo elemento corriente es el primero. Devuelve TRUE
   Si M = LS_SIGUIENTE, el nuevo elemento corriente es el siguiente al
   anterior. Si estaba en el último elemento, devuelve FALSE, sino TRUE.
   Si M = LS_ANTERIOR, devuelve FALSE. */
   int ls_MoverCorriente(TListaSimple *pLs, TMovimiento_Ls M);

   /* ls_BorrarCorriente
   Pre: Ls creada y no vacía.
   Post: Se eliminó el elemento corriente, El nuevo elemento es el siguiente o
   el anterior si el corriente era el último elemento.*/
   void ls_BorrarCorriente(TListaSimple *pLs);

   /* ls_Insertar
   Pre: Ls creada.
   Post: E se agregó a la lista y es el actual elemento corriente.
   Si M=LS_PRIMERO: se insertó como primero de la lista.
   Si M=LS_SIGUIENTE: se insertó después del elemento corriente.
   Si M=LS_ANTERIOR: se insertó antes del elemento corriente.
   Si pudo insertar el elemento devuelve TRUE, sino FALSE.*/
   int ls_Insertar(TListaSimple *pLs, TMovimiento_Ls M, void* E);

#endif
```

LISTASIMPLE.C

```
#include "ListaSimple.h"
#include <malloc.h>
#include <memory.h>

/* ls_Crear
Pre: Ls no fue creada.
Post: Ls creada y vacía */
void ls_Crear(TListaSimple *pLs, int TamanioDato)
{
   pLs->Corriente = NULL;
   pLs->Primero = NULL;
   pLs->TamanioDato = TamanioDato;
}

/* ls_Vaciar
Pre: Ls creada.
Post: Ls vacía.*/
void ls_Vaciar(TListaSimple *pLs)
{
   TNodoSimple *pNodo, *siguiente;
   for(pNodo = pLs->Primero;(pNodo);pNodo=siguiente)
   {
      siguiente = pNodo->Siguiente;
      free(pNodo->Elem);
      free(pNodo);
   }
```

```
   pLs->Primero=pLs->Corriente=NULL;
}

/* ls_Vacia
Pre: Ls creada.
Post: Si Ls tiene elementos devuelve FALSE sino TRUE.*/
int ls_Vacia(TListaSimple Ls)
{
   return (Ls.Primero == NULL);
}

/* ls_ElemCorriente
Pre: Ls creada y no vacía.
Post: Se devuelve en E el elemento corriente de la lista.*/
void ls_ElemCorriente(TListaSimple Ls, void *pE)
{
   memcpy(pE, Ls.Corriente->Elem, Ls.TamanioDato);
}

/* ls_ModifCorriente
Pre: Ls creada y no vacía.
Post: El contenido del elemento actual quedo actualizado con E. */
void ls_ModifCorriente(TListaSimple *pLs, void* pE)
{
   memcpy(pLs->Corriente->Elem, pE, pLs->TamanioDato);
}

/* ls_MoverCorriente
Pre: Ls creada y no vacía.
Post: Si Ls está vacía, devuelve FALSE. Sino:
Si M = LS_PRIMERO, el nuevo elemento corriente es el primero. Devuelve TRUE
Si M = LS_SIGUIENTE, el nuevo elemento corriente es el siguiente al
anterior. Si estaba en el último elemento, devuelve FALSE, sino TRUE.
Si M = LS_ANTERIOR, devuelve FALSE. */
int ls_MoverCorriente(TListaSimple *pLs, TMovimiento_Ls M)
{
   switch (M)
   {
      case LS_PRIMERO:   pLs->Corriente=pLs->Primero;
                         break;
      case LS_SIGUIENTE: if (pLs->Corriente->Siguiente==NULL)
                            return FALSE;
                         else
                            pLs->Corriente=pLs->Corriente->Siguiente;
                         break;
      case LS_ANTERIOR:return FALSE;
   }
   return TRUE;
}

/* ls_BorrarCorriente
Pre: Ls creada y no vacía.
Post: Se eliminó el elemento corriente, El nuevo elemento es el siguiente o
el anterior si el corriente era el último elemento.*/
void ls_BorrarCorriente(TListaSimple *pLs)
{
   TNodoSimple *PNodo=pLs->Corriente;

   if (pLs->Corriente==pLs->Primero)
   {
      pLs->Primero = pLs->Corriente->Siguiente;
      pLs->Corriente = pLs->Primero;
```

```
   }
   else
   {
      TNodoSimple *PAux=pLs->Primero;
      while (PAux->Siguiente!=pLs->Corriente)
         PAux = PAux->Siguiente;
      PAux->Siguiente=pLs->Corriente->Siguiente;
      if (PAux->Siguiente) //Si no es el último
         pLs->Corriente = PAux->Siguiente;
      else
         pLs->Corriente = PAux;  //Si es el último queda en el anterior al
                                 //borrado
   }
   free(PNodo->Elem);
   free(PNodo);
}

/* ls_Insertar
Pre: Ls creada.
Post: E se agregó a la lista y es el actual elemento corriente.
Si M=LS_PRIMERO: se insertó como primero de la lista.
Si M=LS_SIGUIENTE: se insertó después del elemento corriente.
Si M=LS_ANTERIOR: se insertó antes del elemento corriente.
Si pudo insertar el elemento devuelve TRUE, sino FALSE.*/
int ls_Insertar(TListaSimple *pLs, TMovimiento_Ls M, void* pE)
{
   TNodoSimple *pNodo = (TNodoSimple*) malloc(sizeof(TNodoSimple));
   if (!pNodo)
      return FALSE; //No hay memoria disponible
   if ((pLs->Primero == NULL) || (M==LS_PRIMERO) ||
      ((M==LS_ANTERIOR) && (pLs->Primero==pLs->Corriente)))
   {
      //Si está vacía o hay que insertar en el primero o
      //hay que insertar en el anterior y el actual es el primero
      pNodo->Siguiente = pLs->Primero;
      pLs->Primero = pLs->Corriente = pNodo;
   }
   else
   {
      if (M == LS_SIGUIENTE)
      {
         pNodo->Siguiente = pLs->Corriente->Siguiente;
         pLs->Corriente->Siguiente = pNodo;
      }
      else //LS_ANTERIOR
      {
         TNodoSimple *pAux=pLs->Primero;
         while (pAux->Siguiente!=pLs->Corriente)
            pAux = pAux->Siguiente;
         pAux->Siguiente = pNodo;
         pNodo->Siguiente = pLs->Corriente;
      }
   }
   pNodo->Elem = malloc (pLs->TamanioDato);
   memcpy(pNodo->Elem, pE, pLs->TamanioDato);
   pLs->Corriente=pNodo;
   return TRUE;
}
```

Implementación de estructuras con Cursores

Cursores

Un cursor es utilizado para la implementación de estructuras encadenadas, por ejemplo, listas, pilas, colas.
Esta formado por un vector de elementos con un campo entero normalmente llamado SIG, para realizar el encadenamiento, y un puntero entero que indica el próximo elemento disponible DISPO. Se desarrollan a continuación el funcionamiento y primitivas generales que serán usados para la implementación de las estructuras con cursores "embebidos".

Representación

Se utiliza para implementar otras estructuras, con lo cual utilizamos una general, solo mostrando la inclusión de un cursor simple en la misma. La estructura general puede ser, por ejemplo, la lista, la pila, la cola, o cualquier otra que utilice esta alternativa de cursor embebido.

```
#define POS_NULA -1
#define POS_INIC 0
#define POS_MAX 10

typedef int TIPO_POS;
typedef struct TIPO_NODO
{
   TElem elem;
   TIPO_POS sig;
} TIPO_NODO;

typedef struct T_EST_GEN /* EST_GEN puede ser pila, cola, lista, etc.*/
{
   TIPO_POS dispo;
   TIPO_NODO nodos[POS_MAX];
} T_EST_GEN;
```

Primitivas

```
void CURSOR_INICIALIZAR_NODO(T_EST_GEN *te)
{
   int i;
   te->dispo = POS_INIC;
   for (i = POS_INIC; i < POS_MAX; i++)
      te->nodos[i].sig = i + 1;
   te->nodos[POS_MAX-1].sig = POS_NULA;
}
```

```
void CURSOR_OBTENER_NODO(T_EST_GEN *te, TIPO_POS *pos_nodo)
{
   *pos_nodo = te->dispo;
   te->dispo = te->nodos[te->dispo].sig;
}

void CURSOR_LIBERAR_NODO(T_EST_GEN *te, TIPO_POS pos_nodo)
{
   te->nodos[pos_nodo].sig = te->dispo;
   te->dispo = pos_nodo;
}
```

Implementación del TDA Pila utilizando cursores embebidos

Representación

```
typedef struct TPila
{
   TIPO_POS dispo, tope;
   TIPO_NODO nodos[POS_MAX];
} TPila;
```

Primitivas

```
void P_Crear(TPila *p)
{
   p->tope = POS_NULA;
   CURSOR_INICIALIZAR_NODO(p);
}

void P_Sacar(TPila *p, TElem *e)
{
   TIPO_POS s;
   *e = p->nodos[p->tope].elem;
   s = p->nodos[p->tope].sig;
   CURSOR_LIBERAR_NODO(p,p->tope);
   p->tope = s;
}

void P_Vaciar(TPila *p)
{
   P_Crear(p);
}

void P_Agregar(TPila *p, TElem e)
{
   TIPO_POS pos;
   CURSOR_OBTENER_NODO(p,&pos);
   p->nodos[pos].elem = e;
   p->nodos[pos].sig = p->tope;
   p->tope = pos;
}
```

```
int P_Vacia(TPila p)
{
   return (p.tope == POS_NULA);
}
int P_Llena(TPila p)
{
   return (p.dispo == POS_NULA);
}
```

Implementación del TDA Cola utilizando cursores embebidos

Representación

```
typedef struct TCola
{
   TIPO_POS dispo, prim, ult;
   TIPO_NODO nodos[POS_MAX];
} TCola;
```

Primitivas

```
void C_Crear(TCola *c)
{
   c->prim = POS_NULA;
   c->ult = POS_NULA;
   CURSOR_INICIALIZAR_NODO(c);
}

void C_Sacar(TCola *c, TElem *e)
{
   TIPO_POS s;
   *e = c->nodos[c->prim].elem;
   s = c->prim;
   if (c->ult == c->prim)
   {
      c->prim = POS_NULA;
      c->ult = POS_NULA;
   }
   else
      c->prim = c->nodos[c->prim].sig;
   CURSOR_LIBERAR_NODO(c,s);
}

void C_Vaciar(TCola *c)
{
   C_Crear(c);
}
```

```
void C_Agregar(TCola *c, TElem e)
{
   TIPO_POS pos;
   CURSOR_OBTENER_NODO(c,&pos);
   c->nodos[pos].elem = e;
   if (c->prim = POS_NULA)
      c->prim = pos;
   else
      c->nodos[c->ult].sig = pos;
   c->ult = pos;
}

int C_Vacia(TCola c)
{
   return (c.prim == POS_NULA);
}

int C_Llena(TCola c)
{
   return (c.dispo == POS_NULA);
}
```

Implementación del TDA Lista utilizando cursores embebidos

Representación

```
#define PRIMERO 0
#define SIGUIENTE 1
#define ANTERIOR 2

typedef struct TLista
{
   TIPO_POS dispo, prim, cte;
   TIPO_NODO nodos[POS_MAX];
} TLista;
```

Primitivas

```
void L_Crear(TLista *l)
{
   l->prim = POS_NULA;
   l->cte = POS_NULA;
   CURSOR_INICIALIZAR_NODO(l);
}

void L_Vaciar(TLista *l)
{
   L_Crear(l);
}

int L_Vacia(TLista l)
{
   return (l.prim == POS_NULA);
}
```

```
int L_Llena(TLista l)
{
   return (l.dispo == POS_NULA);
}

void L_Elem_Cte(TLista l, TElem *e)
{
   *e = l.nodos[l.cte].elem;
}

void L_Modif_Cte(TLista *l, TElem e)
{
   l->nodos[l->cte].elem = e;
}

int L_Mover_Cte(TLista *l, int m)
{
   switch (m)
   {
   case PRIMERO:   l->cte = l->prim;
                   break;
   case SIGUIENTE: if (l->nodos[l->cte].sig != POS_NULA)
                   l->cte = l->nodos[l->cte].sig;
                   else
                      return 1;
                   break;
   case ANTERIOR:     return 1;
                   break;
   default:           return 1;
   }
   return 0;
}

void L_Borrar_Cte(TLista *l)
{
   TIPO_POS p, cte_ant;
   if (l->cte = l->prim)
      l->prim = l->nodos[l->cte].sig;
   else
   {
      p = l->prim;
      while (l->nodos[p].sig != l->cte)
      p = l->nodos[p].sig;
      l->nodos[p].sig = l->nodos[l->cte].sig;
   }

   cte_ant = l->cte;

   if (l->nodos[l->cte].sig == POS_NULA)
      l->cte = l->prim;
   else
      l->cte = l->nodos[l->cte].sig;

   CURSOR_LIBERAR_NODO(l,cte_ant);
}
```

```
void L_Insertar(TLista *l,int m, TElem e)
{
   TIPO_POS p, nuevo;
   CURSOR_OBTENER_NODO(l,&nuevo);
   l->nodos[nuevo].sig = POS_NULA;
   l->nodos[nuevo].elem = e;

   if (l->prim = POS_NULA || (l->prim = l->cte && m == ANTERIOR)
       || m == PRIMERO)
   {
      l->nodos[nuevo].sig = l->prim;
      l->prim = nuevo;
   }
   else if (m == ANTERIOR)
   {
      p = l->prim;
      while (l->nodos[p].sig != l->cte)
         p = l->nodos[p].sig;
      l->nodos[nuevo].sig = l->cte;
      l->nodos[p].sig = nuevo;
   }
   else if (m == SIGUIENTE)
   {
      l->nodos[nuevo].sig = l->nodos[l->cte].sig;
      l->nodos[l->cte].sig = nuevo;
   }
   l->cte = nuevo;
}
```