Tecnología de la Programación
TDAs Fundamentales

2020
Juan Antonio Sánchez Laguna
Grado en Ingeniería Informática
Facultad de Informática
Universidad de Murcia

TABLA DE CONTENIDOS

TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES 3

EL TIPO DE DATOS ABSTRACTO PILA 4
IMPLEMENTACIÓN CON MEMORIA CONTIGUA 5
IMPLEMENTACIÓN CON ESTRUCTURAS ENLAZADAS LINEALES 9
COMPARACIÓN ENTRE IMPLEMENTACIONES 11
EL TIPO DE DATOS ABSTRACTO COLA 12
IMPLEMENTACIÓN CON MEMORIA CONTIGUA 13
IMPLEMENTACIÓN CON ESTRUCTURAS ENLAZADAS LINEALES 17
COMPARACIÓN ENTRE IMPLEMENTACIONES 19
EL TIPO DE DATOS ABSTRACTO LISTA 21
IMPLEMENTACIÓN CON MEMORIA CONTIGUA 22
IMPLEMENTACIÓN CON ESTRUCTURAS ENLAZADA 27
MEJORAS A LA IMPLEMENTACIÓN DEL TDA LISTA CON ESTRUCTURAS ENLAZADAS 32
COMPARACIÓN ENTRE IMPLEMENTACIONES 34
EL TIPO DE DATOS ABSTRACTO CONJUNTO 35
IMPLEMENTACIÓN CON ESTRUCTURAS ENLAZADAS LINEALES 35
IMPLEMENTACIÓN CON ÁRBOLES BINARIOS DE BÚSQUEDA 41
COMPARACIÓN ENTRE IMPLEMENTACIONES 42
EJEMPLOS DE USO 44
USO DE MEMORIA CONTIGUA DINÁMICA EN LA IMPLEMENTACIÓN DE CONTENEDORES 49
USO DE ARRAYS DINÁMICOS EN LA IMPLEMENTACIÓN DE CONTENEDORES 51

Tipos de Datos Abstractos Fundamentales

En este apartado se estudian varios Tipos de Datos Abstractos que, por ser ampliamente utilizados en ciencias de la computación, se consideran fundamentales: Pilas, Colas, Listas y Conjuntos. Todos ellos son TDAs contenedores, es decir, Tipos de Datos que almacenan elementos de otro Tipo de Datos. De todos se estudiará su especificación, varias posibles implementaciones, las decisiones de diseño tomadas y cómo afectan a la eficiencia de sus operaciones.

Los tres primeros TDAs representan secuencias de elementos. Pilas y Colas son secuencias con restricciones en cuanto a las posiciones por donde se pueden insertar o extraer los elementos. Las Listas son mucho más flexibles permitiendo insertar y suprimir en cualquier posición. De los tres se estudiará cómo implementarlos usando memoria contigua y estructuras enlazadas lineales.

Las implementaciones basadas en memoria contigua estudiadas en primer lugar se basarán en arrays con tamaño máximo limitado en su declaración. A los Tipos de Datos Contenedor con esta limitación se les denomina Acotados. Las implementaciones basadas en estructuras enlazadas no tendrán dicha limitación, pero, a cambio, su implementación es algo más compleja y usarán más memoria que las primeras, pues cada nodo tendrá un puntero que lo enlace con el siguiente.

En cuanto a los Conjuntos, lo que los diferencia de los otros tres Tipos de Datos es que no contienen elementos repetidos, y de ellos se verá una implementación basada en estructuras enlazadas lineales y otra con árboles binarios de búsqueda.

En todos los casos se estudiará la eficiencia de las operaciones en las diferentes implementaciones de un mismo Tipo de Datos comparando su tiempo de ejecución. Para expresar cómo aumenta el tiempo de ejecución de un cierto algoritmo en función de los datos de entrada se usará la notación asintótica O (O grande). Con esta notación se indica una función matemática que sirva de cota superior para la función real que determina la complejidad temporal del algoritmo.

Por ejemplo, si el tiempo de ejecución de un algoritmo crece de forma lineal con el tamaño de los datos de entrada, se dirá que su tiempo de ejecución es “de orden n” y se escribirá como “de O(n)”. Si lo hace de forma cuadrática se dirá que es de $O(n^2)$ y así sucesivamente. Las funciones más usadas con esta notación van desde la que representa la máxima eficiencia: O(1) hasta la que representa un algoritmo extremadamente costoso que sería la de $O(n^n)$:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)$$

Por otro lado, la especificación de un TDA Contenedor no debería depender del Tipo de Datos de los elementos que va a contener. A esta característica se le denominada Genericidad y, de las diversas formas en las que se puede implementar en C, se ha elegido una que resulta muy sencilla, aunque, estrictamente hablando, más que genericidad lo que consigue es flexibilidad a la hora de definir los Tipos de Datos usados por los contenedores.

Consiste en definir los contenedores de modo que los elementos a almacenar sean de un Tipo de Datos denominado `Elemento`. La definición de `Elemento` se hace mediante un alias (`typedef`) en la parte pública de los módulos usados para implementar los TDAs. De este modo, es sencillo adaptarlos para que contengan cualquier Tipo de Datos, sin mas que cambiar la definición de `Elemento`.

El Tipo De Datos Abstracto Pila

En el mundo real hay múltiples ejemplos de elementos organizados en forma de pila, por ejemplo, una pila de platos, un paquete de pelotas de tenis, etc. En todos los casos lo que caracteriza a esta forma de organizar objetos es que son depositados y retirados por el mismo extremo de la fila que forman.

Las pilas en ciencias de la computación son muy parecidas a las pilas de objetos del mundo real. Normalmente una pila se representa gráficamente (Fig. 1) como una caja abierta por la parte superior y con la palabra tope se marca la posición que ocupa el último elemento insertado en la misma que, a su vez, es el único al que se puede acceder directamente.

Formalmente, el TDA Pila es una secuencia de elementos en la que las inserciones y extracciones sólo pueden hacerse por uno de los extremos, al que se le llama tope. Entre las operaciones que forman parte del interfaz público de este TDA se encuentran las siguientes:

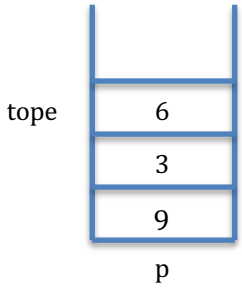

Fig. 1 Representación gráfica de una Pila

- **Inserta(x)** (push) añade el elemento x a la Pila situándolo encima del resto en la posición que constituirá el nuevo tope de la Pila.
- **Suprime()** (pop) extrae el elemento situado en el tope de la Pila, siempre que ésta no esté vacía, dejándola con un elemento menos.
- **Recupera()** (peek) devuelve el elemento situado en el tope de la Pila, siempre que ésta no esté vacía.
- **Vacía()** (empty) devuelve verdadero si la Pila está vacía y falso en caso contrario.

Las Pilas tienen múltiples aplicaciones en ciencias de la computación: posibilitan la invocación de funciones y la recursividad, permiten comprobar si una expresión tiene los paréntesis equilibrados e incluso determinar su valor, sirven para implementar la función "Atrás" de los navegadores, así como la función "Deshacer" de los editores de texto, etc.

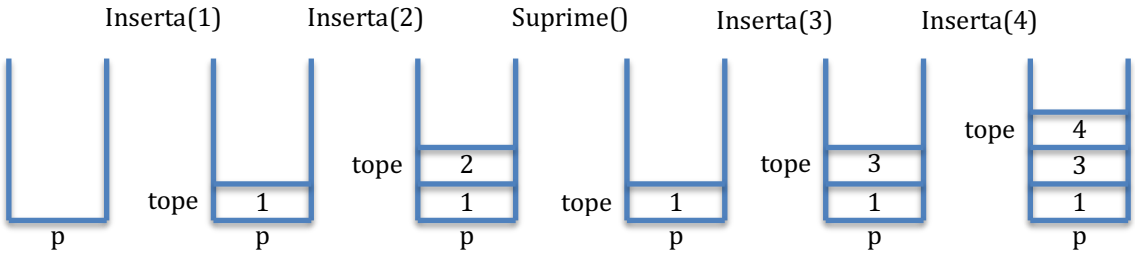

Fig. 2 Ejemplo de uso de una Pila de enteros

El ejemplo de la Fig. 2 muestra el funcionamiento de una pila de enteros. Empieza por una pila recién creada y vacía hasta que, tras realizar las operaciones Inserta(1), Inserta(2), Suprime(), Inserta(3), e Inserta(4), la pila queda con la secuencia: (1, 3, 4) con el elemento 4 situado en el tope de la misma. En este momento, la operación Recupera() devolvería el valor 4 y la operación Vacía() devolvería falso.

Al utilizar una Pila siempre se extrae, en primer lugar, el último de los elementos previamente insertados, es decir, el que lleva menos tiempo almacenado en la Pila. Por eso se dice que las Pilas tienen un modelo de acceso LIFO: Last In First Out.

Para implementar el TDA Pila es necesario elegir la estructura de datos que lo representará e implementar las operaciones descritas en su interfaz público. Hay dos estructuras de datos con las que habitualmente se representan las Pilas: los arrays (memoria contigua) y las estructuras enlazadas lineales.

Implementación con memoria contigua

Un array es una secuencia de elementos del mismo tipo almacenados en memoria de forma contigua. Por tanto, se puede utilizar un array para almacenar los datos contenidos en una Pila. El array se definirá en función del Tipo de Datos de los elementos a almacenar en la Pila, y los datos se pueden guardar (Fig. 3) en orden ascendente empezando por el principio del array. En cualquier caso, es necesario acompañar el array de un entero que indique en todo momento la posición del array que corresponda al tope de la Pila. Es decir, la posición en la que estará almacenado el último elemento que se haya insertado en la Pila.

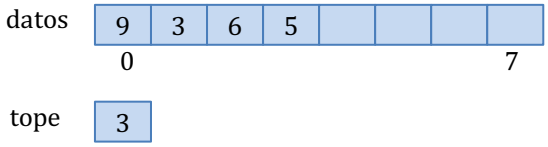

Fig. 3 Esquema de uso de un array para representar una Pila

La Fig. 4 muestra la estructura de datos necesaria para representar una Pila de enteros tal y como se acaba de describir. Contiene el array `datos` y la variable `tope`. El array se declara con un tamaño máximo limitado (`MAX`). Esta limitación no viene impuesta por la especificación general del TDA Pila, pero es la forma más sencilla de implementar una Pila con arrays. A las Pilas que tienen limitado su tamaño máximo se les denomina Pilas Acotadas. Más adelante se verán otras implementaciones que no tienen esta restricción.

```
struct PilaRep {
  Elemento datos[MAX];
  int tope;
};
```

Fig. 4. Estructura de datos usada para representar un TDA Pila con memoria contigua (arrays)

Como es habitual, la estructura de datos elegida para representar un TDA influye en la implementación de las operaciones que componen su interfaz público. Por ejemplo, el array `datos` no se puede usar para determinar si la Pila está vacía o no porque siempre tiene el mismo tamaño. Su contenido tampoco es determinante en este aspecto. Sin embargo, la variable `tope` si se puede usar para este fin. Como las posiciones válidas del array comienzan en la 0, cualquier valor negativo serviría para indicar que la Pila está vacía. Ahora bien, para decidir el valor concreto a usar es necesario estudiar primero el resto de operaciones.

Las operaciones Inserta, Suprime y Recupera deben usar y modificar el valor de `tope` de modo que, en todo momento, se cumpla con la especificación del TDA Pila. Es decir, el valor de `tope` debe indicar la posición del ultimo elemento insertado.

De este modo, al empezar a usar el array por el principio, cuando la pila contenga un único elemento, tope debería valer cero y cuando la pila tenga n elementos valdrá n-1. Además, cada vez que se produzca una inserción, la posición del array que ocupará el nuevo elemento será tope+1 y, por otro lado, cuando se suprima un elemento de la pila habrá que restarle uno a tope.

Teniendo en cuenta todo lo anterior el valor más adecuado para reflejar que la pila está vacía es el -1, pues al sumarle uno durante la primera inserción se convertirá en cero, que es la primera posición válida del array. Además, cuando se suprima el último elemento de la pila y se reste uno a tope, éste volverá a valer -1, indicando que vuelve a estar vacía.

```
Inserta( x ) {
  tope = tope + 1;
  datos[tope] = x;
}
```

```
Vacía( x ) {
  return tope == -1;
}
```

Fig. 5 Pseudocódigo de las operaciones Inserta y Vacía en el TDA Pila implementado con arrays

Nótese la importancia que tiene el orden en que se realizan las dos instrucciones de la operación Inserta (Fig. 5). Primero se incrementa el valor de tope y después se guarda el nuevo elemento en la posición indicada por el mismo. Por su parte, la operación Vacía devolverá verdadero cuando tope valga -1.

```
Recupera( x ) {
  return datos[tope];
}
```

```
Suprime( ) {
  tope = tope - 1;
}
```

Fig. 6. Pseudocódigo de las operaciones Recupera y Suprime en el TDA Pila implementado con arrays

La operación Recupera (Fig. 6) devuelve el elemento situado en la posición indicada por tope. Y la operación Suprime simplemente reduce el valor de tope. Aunque el array siga conteniendo el elemento que antes estaba en la posición del tope, este será sobrescrito la próxima vez que se inserte otro. Y si antes se llega a ejecutar la operación Recupera, el valor devuelto será el del tope actual.

De cara a implementar en C el TDA Pila se debe escribir un módulo. La Fig. 7 muestra el fichero cabecera de dicho módulo. Como puede verse, además de las operaciones descritas, es necesario incluir una operación de creación y otra de liberación que permitan crear y liberar Pilas desde el código cliente del TDA.

```
#ifndef __Pila_H__
#define __Pila_H__

typedef int Elemento;
typedef struct PilaRep * Pila;

Pila crea();
void libera( Pila p );
void inserta( Pila p, Elemento e );
void suprime( Pila p );
Elemento recupera( Pila p );
int vacia( Pila p );
int llena( Pila p );

#endif
```

Fig. 7. Fichero cabecera de un módulo que implementa el TDA Pila con memoria contigua (arrays)

Además, dado que se está implementando una Pila Acotada, es necesario añadir otra operación que devuelva verdadero cuando la Pila está llena.

El último aspecto interesante del fichero cabecera es que en él también se define el Tipo de Datos `Elemento` como un alias de `int`. De este modo, el TDA Pila implementado en este ejemplo es capaz de almacenar valores enteros.

La Fig. 8 muestra el fichero .c en el que se implementa el TDA Pila Acotada de enteros usando un array estático en la estructura que representa la Pila como acaba de comentarse.

```
#include "Pila.h"
#include <stdlib.h>
#include <assert.h>

#define MAX   50

struct PilaRep {
  Elemento datos[MAX];
  int tope;
};

Pila crea() {
  Pila nueva = malloc( sizeof( struct PilaRep ) );
  nueva->tope = -1;
  return nueva;
}

void libera( Pila p ) {
  free( p );
}

void inserta( Pila p, Elemento e ) {
  assert( p->tope < MAX-1 );
  p->tope = p->tope + 1;
  p->datos[p->tope] = e;
}

void suprime( Pila p ) {
  assert( p->tope >= 0 );
  p->tope = p->tope - 1;
}

Elemento recupera( Pila p ) {
  assert( p->tope >= 0 );
  return p->datos[p->tope];
}

int vacia( Pila p ) {
  return p->tope == - 1;
}

int llena( Pila p ) {
  return p->tope == MAX-1;
}
```

Fig. 8. Fichero .c de un módulo que implementa el TDA Pila con memoria contigua (arrays)

Al principio del fichero `.c`, se incluye el fichero de cabecera `Pila.h` para conseguir que los Tipos de Datos `Elemento` y `Pila`, y las declaraciones de las funciones, estén disponibles en este fichero. Aquí es donde se define la estructura struct `PilaRep` que representará el Tipo de Datos Pila mediante la agrupación del array `datos` y la variable `tope`.

La función `crea` reserva memoria para la estructura de datos que representa la Pila e inicializa el campo `tope` a -1 indicando que la Pila está recién creada y, por tanto, vacía. La función `libera` simplemente libera la memoria asociada a la estructura que representa la Pila.

Las funciones que implementan las operaciones Inserta, Suprime y Recupera son traducciones literales del pseudocódigo visto anteriormente con la salvedad de que las variables `datos` y `tope` son campos de la estructura struct `PilaRep`, de ahí que en todos los casos se use `p->tope` y `p->datos` para referirse a ellas.

La limitación de tamaño de la Pila hace que, al igual que pasa con las operaciones Recupera y Suprime, haya que imponer una precondición a la operación Inserta, puesto que ésta no se podrá llevar a cabo si la Pila está llena. Así pues, en la implementación de todas las operaciones que tienen precondiciones: `inserta`, `suprime` y `recupera`, se ha usado la macro `assert` para comprobarlas y detener la ejecución cuando no se cumplan, lo cual es útil durante la fase de desarrollo.

La Pila Acotada que se acaba de implementar tiene un problema de diseño importante. El tamaño máximo queda prefijado en la implementación de la misma. Por lo tanto, todas las Pilas tendrán siempre dicho tamaño y puede resultar demasiado grande en unos casos y demasiado pequeño en otros. Una posible solución sería mover la definición de MAX al fichero cabecera, pero incluso en ese caso, todas las Pilas de una misma aplicación tendrían que tener el mismo tamaño máximo, lo que puede ser poco eficiente.

La implementación del TDA Pila con estructuras enlazadas que se verá a continuación no tiene esta limitación, pero, en cualquier caso, más adelante se verán propuestas de mejora que permitan crear Pilas Acotadas implementadas con memoria contigua dinámica cuyo tamaño máximo sea especificado en el momento de su creación.

Implementación con estructuras enlazadas lineales

Una estructura enlazada lineal representa una secuencia de elementos, por lo tanto, se puede usar para almacenar los datos contenidos en una Pila. Además, en este caso, se puede aprovechar la simplicidad y eficiencia de las inserciones y supresiones de elementos por el principio para implementar las operaciones Inserta y Suprime.

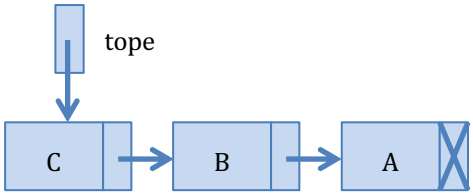

Fig. 9 Esquema de uso de una estructura enlazada lineal para representar una Pila

Cada nodo de la estructura enlazada se encargará de almacenar uno de los elementos de la secuencia que forma la Pila. Además, si las inserciones y supresiones se realizan siempre por el principio de la lista, el apuntador externo de la estructura enlazada que se use para manejarla será el equivalente al tope de la Pila y se conseguirá replicar el comportamiento LIFO propio de las Pilas.

```
struct Nodo {
  Elemento dato;
  struct Nodo * sig;
};

typedef struct Nodo * NodoPtr;

struct PilaRep {
  NodoPtr tope;
};
```

Fig. 10. Estructura de datos usada para representar un TDA Pila con estructuras enlazadas lineales

Así pues, una vez definida la estructura de datos enlazada lineal mediante la definición de los Nodos que la componen (Fig. 10), la estructura de datos para representar la Pila (`struct PilaRep`) consistirá únicamente en una variable de tipo `NodoPtr` que contendrá la dirección del primer nodo de la estructura enlazada lineal. Ese primer nodo contendrá siempre el último elemento insertado por el principio y representará el tope de la Pila. Por lo tanto, una Pila vacía será aquella cuyo campo `tope` valga `NULL`, es decir, no apunte a ningún nodo.

```
Vacía( x ) {
  return tope == NULL;
}
```

```
Recupera( ) {
  return tope->dato;
}
```

Fig. 11 Pseudocódigo de las operación Vacía en el TDA Pila implementado con estructuras enlazadas

La Fig. 11 muestra el pseudocódigo de las operaciones Vacía y Recupera. La primera simplemente compara el valor de `tope` con `NULL` ya que esa es la definición de Pila vacía que se ha decidido. La segunda, asumiendo que se cumplen las precondiciones y, por tanto, la Pila no está vacía, simplemente devuelve el valor del dato almacenado en el primer nodo de la estructura enlazada, es decir, aquel al que en ese momento esté apuntando la variable `tope`.

```
Inserta( x ) {
  NodoPtr nuevo = malloc(sizeof(struct Nodo));
  nuevo->dato = x;
  nuevo->sig = tope;
  tope = nuevo;
}
```

```
Suprime( ) {
  NodoPtr borrar = tope;
  tope = borrar->sig;
  free( borrar );
}
```

Fig. 12 Pseudocódigo de las operaciones Inserta y Suprime en el TDA Pila implementado con listas

La Fig. 12 muestra el pseudocódigo de las operaciones Inserta y Suprime. Como se puede ver, la inserción consiste básicamente en crear un nuevo nodo para guardar el dato a insertar y colocarlo como nuevo tope, pero cuidando primero de hacer que su campo `sig` apunte al que, hasta ese momento, era el primero de la lista.

La operación Suprime hace lo contrario: primero guarda la dirección del tope actual y después coloca como tope al nodo que hubiera a continuación del mismo. Por último, libera la memoria asociada al nodo al que, hasta ese momento apuntaba, `tope`.

Para llevar a cabo la implementación del TDA Pila mediante estructuras enlazadas hay que escribir un fichero `.c` como el mostrado en la Fig. 13.

La estructura `PilaRep` contiene todo lo necesario para representar al Tipo de Datos que, en este caso, únicamente es la variable `tope`. En el mismo módulo, y oculto a los clientes del mismo, se han definido la estructura `struct Nodo` y el tipo `NodoPtr` que permiten crear y manejar la estructura enlazada con la que se representará la Pila.

Las funciones que implementan las operaciones Inserta, Suprime y Recupera son traducciones literales del pseudocódigo visto anteriormente con la salvedad de que la variable `tope` es un campo de la estructura `struct PilaRep`, de ahí que en todos los casos se use `p->tope` para referirse a ella.

Por otro lado, la función `libera` se ocupa de forma iterativa de liberar la memoria asociada a todos los nodos de la estructura enlazada y, finalmente, de liberar la memoria asociada a la estructura `struct PilaRep` apuntada por el parámetro `p`.

La decisión de representar una Pila vacía mediante el valor `NULL` implica que se está usando una estructura enlazada sin cabecera. Sin embargo, en este caso no resulta problemático. Como las inserciones y supresiones se hacen siempre por el principio, ambas modifican la variable `tope` y no se requieren casos especiales en el código. Además, dicho puntero externo a la estructura enlazada está siempre accesible por ser un campo de la estructura `struct PilaRep`. En el fondo, es como si esa estructura actuara de cabecera de la estructura enlazada cuyo primer nodo con datos es el apuntado por el campo tope.

Por otro lado, esta implementación no necesita la operación Llena porque, gracias al uso de la estructura enlazada para representar los elementos almacenados, el número máximo de elementos no necesita ser conocido a priori y, en principio, no está limitado. Por lo tanto, esta implementación sí que cumple el interfaz original del TDA Pila y la función `llena` se puede quitar del fichero cabecera.

Comparación entre implementaciones

La implementación del TDA Pila con memoria contigua impone una limitación en el tamaño máximo de la Pila que la implementación basada en estructuras enlazadas no tiene. Sin embargo, ésta incurre en una pequeña sobrecarga relativa al uso de memoria porque cada nodo de la estructura enlazada incluye un puntero al siguiente.

En cuanto a la eficiencia de las operaciones, en ambas implementaciones se ha conseguido que todas las operaciones del TDA tengan un tiempo de ejecución constante e independiente del número de elementos de la Pila. La única excepción es la función libera en el caso de la implementación mediante estructuras enlazadas, pero esta operación no es la más frecuentemente usada y, por eso, queda fuera de la comparación.

```
#include "Pila.h"
#include <stdlib.h>
#include <assert.h>

struct Nodo {
  Elemento dato;
  struct Nodo * sig;
};

typedef struct Nodo * NodoPtr;

struct PilaRep {
  NodoPtr tope;
};

Pila crea() {
  Pila nueva = malloc( sizeof( struct PilaRep ) );
  nueva->tope = NULL;
  return nueva;
}

void libera( Pila p ) {
  while ( p->tope != NULL ) {
    NodoPtr borrado = p->tope;
    p->tope = borrado->sig;
    free( borrado );
  }
  free( p );
}

void inserta( Pila p, Elemento e ) {
  NodoPtr nuevo = malloc( sizeof( struct Nodo ) );
  nuevo->dato = e;
  nuevo->sig = p->tope;
  p->tope = nuevo;
}

void suprime( Pila p ) {
  assert( p->tope != NULL );
  NodoPtr borrar = p->tope;
  p->tope = borrar->sig;
  free( borrar );
}

Elemento recupera( Pila p ) {
  assert( p->tope != NULL );
  return p->tope->dato;
}

int vacia( Pila p ) {
  return p->tope == NULL;
}
```

Fig. 13. Fichero .c de un módulo que implementa el TDA Pila con estructuras enlazadas lineales

El Tipo De Datos Abstracto Cola

En el mundo real hay múltiples ejemplos de situaciones en las que es necesario utilizar una cola para racionalizar el acceso a un recurso escaso: cola para sacar entradas, para pagar, para que te firmen un autógrafo, etc. En todos los casos lo que caracteriza esta forma de ordenar personas es que el primero que llega es el primero en ser atendido, o visto de otro modo, quien lleva más tiempo esperando es atendido en primer lugar.

Las colas en ciencias de la computación son muy parecidas a las colas del mundo real. Normalmente se representan gráficamente (Fig. 14) como un par de líneas horizontales que delimitan un espacio interior donde se almacenan los datos y dos extremos abiertos: uno por donde se añaden nuevos elementos y otro por donde se extraen los existentes.

Formalmente, el TDA cola es una secuencia ordenada de elementos en donde a un extremo se le llama inicio o frente y al otro final. Las inserciones se deben hacer siempre por el final y las extracciones por el inicio. Las operaciones que forman parte del interfaz público de este TDA incluyen al menos las siguientes:

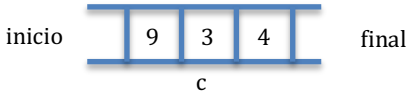

Fig. 14 Representación gráfica de una Cola

- **Inserta(x)** (enqueue) añade el elemento x a la Cola situándolo al final como último elemento de la Cola.
- **Suprime()** (dequeue) elimina el elemento situado en el inicio de la Cola, siempre que ésta no esté vacía.
- **Recupera()** (peek) devuelve el elemento situado en el inicio de la Cola, siempre que ésta no esté vacía.
- **Vacía()** (empty) devuelve verdadero si la Cola está vacía y falso en caso contrario.

Las colas tienen múltiples aplicaciones en ciencias de la computación: en comunicaciones entre procesos se usan como buffer intermedio, los sistemas operativos tienen colas de procesos esperando para acceder a ciertos recursos, se usan para realizar simulaciones de diferentes tipos, etc.

La Fig. 15 muestra un ejemplo de funcionamiento de una cola de enteros. Empieza por una cola recién creada, y por tanto, vacía hasta que, tras llevar a cabo las operaciones: Inserta(1), Inserta(2), Inserta(3), Suprime(), y Suprime(), la cola queda así : (3), con el elemento 3 situado en el frente y, al mismo tiempo, al final.

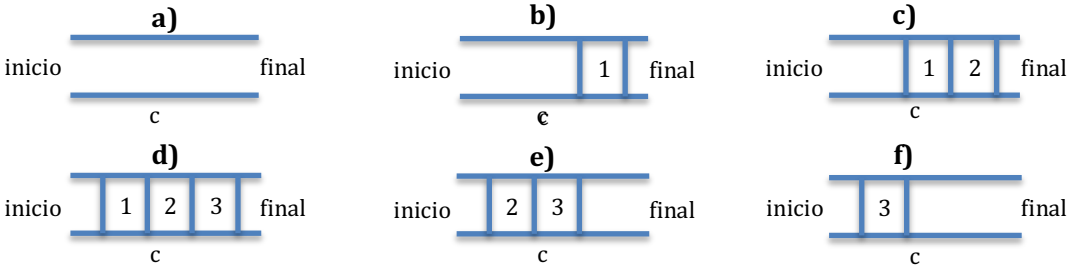

Fig. 15 Ejemplo de uso de una Cola de enteros

Al utilizar una Cola siempre se extrae, en primer lugar, el primero de los elementos previamente insertados, es decir, el que lleva más tiempo almacenado en la Cola, por eso se dice que las Colas tienen un modelo de acceso FIFO: First In First Out.

Para implementar el TDA Cola es necesario elegir la estructura de datos que lo represente e implementar las funciones descritas en su interfaz público. Hay dos estructuras de datos con las que habitualmente se representan las Colas: los arrays (memoria contigua) y las estructuras enlazadas.

Implementación con memoria contigua

Para almacenar la secuencia de elementos en que consiste una Cola se puede usar un array. Dicho array se definirá en función del Tipo de Datos de los elementos a almacenar. Los elementos se pueden guardar en orden ascendente empezando por el principio del array. Además, se debe garantizar que las inserciones y supresiones se hagan respetando el modelo de acceso impuesto por la especificación del TDA Cola. Para ello, se pueden usar dos variables de tipo entero: una que indique la posición del array correspondiente al inicio de la Cola, y otra que indique la posición correspondiente al final de la Cola. Ambas deberán estar actualizadas en todo momento.

```
struct ColaRep {
  Elemento datos[MAX];
  int inicio;
  int final;
};
```

Fig. 16. Estructura de datos propuesta para representar un TDA Cola con memoria contigua (arrays)

La Fig. 16 muestra la estructura de datos necesaria para representar una Cola tal y como se acaba de describir. Contiene el array `datos` y los enteros `inicio` y `final`. El array `datos` se declara con un tamaño máximo limitado (`MAX`). Esta limitación no viene impuesta por la especificación general del TDA Cola, pero es la forma más sencilla de implementar una Cola con arrays. A las Colas que tienen limitado su tamaño máximo se les denomina Colas Acotadas. Más adelante se verán otras implementaciones que no tienen esta restricción.

El valor de la variable `final` debería indicar la posición del array `datos` donde se llevará a cabo la siguiente inserción, y el valor de la variable `inicio` la posición donde estará almacenado el elemento añadido hace más tiempo. Por tanto, ambas variables se inicializarán a cero ya que los arrays se indexan a partir de ese valor.

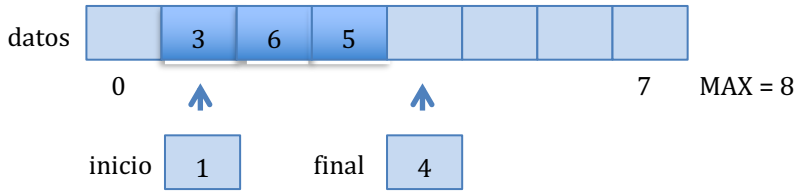

Fig. 17 Estado de las variables usadas para representar una Cola tras varias operaciones

La figura Fig. 17 muestra los valores que tendrían las tres variables usadas para representar una Cola tras haber realizado cuatro inserciones y una supresión en una Cola con capacidad para almacenar un máximo de 8 enteros.

La Fig. 18 muestra el pseudocódigo de las operaciones del TDA Cola usando dichas variables. La operación Inserta primero guarda el nuevo elemento en la posición indicada por `final` y después incrementa su valor, la operación Suprime lo único que hace es incrementar el valor de `inicio`, pues, aunque el array siga conteniendo el elemento, dicha posición ya no será usada.

La operación Recupera simplemente devuelve el elemento situado en la posición `inicio`. Y la operación Vacía consiste únicamente en comprobar si el valor de `inicio` es igual al de `final` porque la existencia de elementos en el array `datos` implicará que la posición `inicio` sea menor que la de `final`.

```
Inserta( x ) {
  datos[final] = x;
  final = final + 1;
}
```

```
Suprime( ) {
  inicio = inicio + 1;
}
```

```
Recupera( ) {
  return datos[inicio];
}
```

```
Vacía( ) {
  return inicio == final;
}
```

Fig. 18. Pseudocódigo inicial de las operaciones del TDA Cola implementado con arrays

Pero si se implementan las operaciones siguiendo estos esquemas surge un problema. Las inserciones hacen que la posición representada por `final` vaya avanzando y, en algún momento, alcanzará el máximo del array. Al mismo tiempo, la variable `inicio` también va aumentando por las sucesivas supresiones. Por tanto, puede quedar un hueco al comienzo del array que no se podrá usar.

Aplicando la técnica del array circular se puede solucionar el problema. La idea consiste en simular que tras la última posición física del array vuelve a estar la primera. Para conseguirlo, tras incrementar `inicio` o `final` su valor se corrige para mantenerlos en el intervalo `[0, MAX-1]`. Esto se hace aplicando el operador resto (`%`) antes de guardar los nuevos valores.

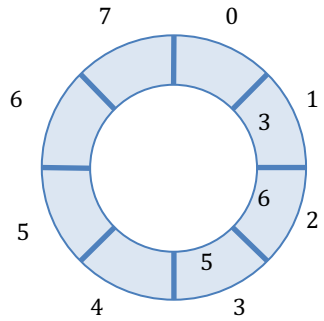

Fig. 19 Ejemplo de array circular

La Fig. 20 muestra la sencilla modificación necesaria para usar la técnica del array circular. Al aplicar esta técnica se consigue reusar las partes del array que van quedando libres. Sin embargo, el hecho de que `inicio` y `final` coincidan puede ahora significar que el array `datos` está vacío o que está lleno. Por lo tanto, es necesario añadir una nueva variable de tipo entero a la representación del TDA Cola. Esta variable, llamada `n`, llevará la cuenta del número de elementos existentes en cada momento. Se debe inicializar a cero, y mantenerla actualizada aumentando o disminuyendo su valor tras Insertar o Suprimir:

```
Inserta( ) {
  datos[final] = x;
  final = (final + 1) % MAX;
  n = n + 1;
}
```

```
Suprime( ) {
  inicio = (inicio + 1) % MAX;
  n = n - 1;
}
```

Fig. 20. Pseudocódigo de las operaciones Inserta y Suprime del TDA Cola usando arrays circulares

La operación Recupera no necesita ninguna modificación, pero la operación Vacía quedaría ahora como se muestra en la Fig. 21. Por supuesto, la estructura de datos que representa al Tipo de Datos Cola también se debe modificar incluyendo la variable `n` que se acaba de describir.

```
Recupera( ) {
  return datos[inicio];
}
```

```
Vacía( ) {
  return n == 0;
}
```

Fig. 21. Pseudocódigo de las operaciones Recupera y Vacía del TDA Cola con arrays circulares

De cara a implementar en C el TDA Cola se debe escribir un módulo. La Fig. 22 muestra el fichero cabecera de dicho módulo. Como puede verse, además de las operaciones descritas, es necesario incluir una operación de creación y otra de liberación que permitan crear y liberar Colas desde el código cliente del TDA. Además, dado que se está implementando una Cola Acotada, es necesario añadir otra operación que devuelva verdadero cuando la Cola está llena.

El último aspecto interesante del fichero cabecera es que en él también se define el Tipo de Datos `Elemento` como un alias de `int`. De este modo, el TDA Cola implementado en este ejemplo es capaz de almacenar valores enteros.

```
#ifndef __Cola_H__
#define __Cola_H__

typedef int Elemento;
typedef struct ColaRep * Cola;

Cola crea();
void libera( Cola c );
void inserta( Cola c, Elemento e );
void suprime( Cola c );
Elemento recupera( Cola c );
int vacia( Cola c );
int llena( Cola c );

#endif
```

Fig. 22. Fichero cabecera de un módulo que implementa el TDA Pila con memoria contigua (arrays)

La Fig. 23 muestra un módulo .c en el que se implementa el TDA Cola Acotada de enteros usando un array estático en la estructura que representa la Cola como acaba de comentarse.

Al principio del fichero `.c`, se incluye el fichero de cabecera `Cola.h` para conseguir que los Tipos de Datos `Elemento` y `Cola`, así como las declaraciones de las funciones, estén disponibles en este fichero. Aquí es donde se define la estructura struct `ColaRep` que representará el Tipo de Datos Cola mediante la agrupación del array `datos`, los índices `inicio` y `final`, y la variable `n` que llevará la cuenta del número de elementos contenidos en la Cola.

La función `crea` reserva memoria para la estructura de datos que representa la Cola y asigna como valor inicial un cero a los campos `inicio`, `final` y `n` indicando que la Cola está recién creada y vacía. La función `libera` simplemente libera la memoria asociada a la estructura que representa la Cola.

La limitación de tamaño de la Cola hace que, al igual que pasa con Recupera y Suprime, haya que imponer una precondición a la operación Inserta, puesto que ésta no se podrá llevar a cabo si la Cola está llena. Teniendo esto en cuenta, en la implementación de todas las operaciones que tienen precondiciones: `inserta`, `suprime` y `recupera`, se ha usado la macro `assert` para comprobarlas y detener la ejecución cuando no se cumplan, lo cual es útil durante la fase de desarrollo.

Esta implementación de Cola Acotada resulta muy poco práctica ya que todas las Colas tendrán siempre el mismo tamaño máximo: el indicado por la constante `MAX`. Más adelante se verán propuestas de mejora que permitan crear Colas Acotadas con un tamaño máximo especificado en el momento de su creación.

```
#include "Cola.h"
#include <stdlib.h>
#include <assert.h>

#define MAX   50

struct ColaRep {
  Elemento datos[MAX];
  int inicio;
  int final;
  int n;
};

Cola crea() {
  Cola nueva = malloc( sizeof( struct ColaRep ) );
  nueva->inicio = 0;
  nueva->final = 0;
  nueva->n = 0;
  return nueva;
}

void libera( Cola c ) {
  free( c );
}

void inserta( Cola c, Elemento e ) {
  assert( c->n < MAX );
  c->datos[c->final] = e;
  c->final = (c->final + 1) % MAX;
  c->n = c->n + 1;
}

void suprime( Cola c ) {
  assert( c->n > 0 );
  c->inicio = (c->inicio + 1) % MAX;
  c->n = c->n - 1;
}

Elemento recupera( Cola c ) {
  assert( c->n > 0 );
  return c->datos[c->inicio];
}

int vacia( Cola c ) {
  return c->n == 0;
}

int llena( Cola c ) {
  return c->n == MAX;
}
```

Fig. 23. Fichero .c de un módulo que implementa el TDA Cola con memoria contigua (arrays)

Implementación con estructuras enlazadas lineales

Otra forma de implementar una Cola es mediante una estructura enlazada. Pero en este caso, a diferencia de la implementación del TDA Pila con estructuras enlazadas lineales, es necesario insertar por un extremo y suprimir por el otro.

Una estructura enlazada lineal se gestiona habitualmente con un único puntero externo que apunta al primer nodo de la misma. Gracias a este puntero es fácil y eficiente insertar y suprimir por el principio de la estructura enlazada. Sin embargo, para insertar y suprimir por el final hay recorrer toda la estructura enlazada para encontrar el último nodo, con el coste que esto implica.

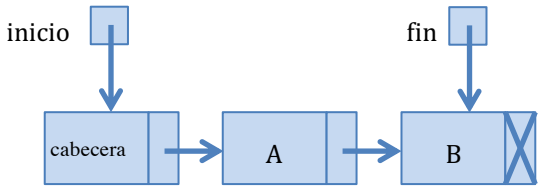

Fig. 24 Esquema de uso de una estructura enlazada lineal para representar una Cola

Ahora bien, si se usa un segundo puntero externo que se mantenga todo el tiempo actualizado apuntando al último nodo de la estructura enlazada, las inserciones por el final también serán muy eficientes: O(1). Las supresiones por el final seguirían siendo costosas: O(n), pues sigue siendo necesario recorrer toda la estructura hasta encontrar el nodo anterior al último, para poder así borrarlo. Pero como sólo se necesita insertar por un extremo y suprimir por el contrario, se puede hacer la supresión por el principio, que sí es eficiente, y la inserción por el final, que también lo es cuando se tiene el citado segundo apuntador externo.

```
struct Nodo {
  Elemento dato;
  struct Nodo * sig;
};

typedef struct Nodo * NodoPtr;

struct ColaRep {
  NodoPtr inicio;
  NodoPtr final;
};
```

Fig. 25. Estructura de datos usada para representar un TDA Cola con estructuras enlazadas lineales

Así pues, llamando `inicio` al puntero externo que apunta al primer nodo de la estructura enlazada, y `final` al nuevo puntero externo que apuntará al último de la misma, las operaciones del TDA Cola con estructuras enlazas lineales serán muy eficientes. Si además se usa la estrategia del nodo cabecera, las operaciones de inserción y supresión no tendrán casos especiales.

La Fig. 25 muestra la definición de todo lo necesario para usar una estructura enlazada lineal: la estructura `struct Nodo` y el alias `typedef NodoPtr`. Y también la estructura `struct ColaRep` que se usará para representar una Cola. En ella se agrupan los punteros `inicio` y `final` que apuntarán respectivamente a la cabecera y al nodo que lleve menos tiempo dentro de la Cola.

```
Recupera( ) {
  return inicio->sig->dato;
}
```

```
Vacía( ) {
  return inicio->sig == NULL;
}
```

Fig. 26. Pseudocódigo de las operaciones del TDA Cola implementado con estructuras enlazadas

Como se va a usar la estrategia del nodo cabecera para simplificar el código de las distintas operaciones, una Cola vacía estará representada por una secuencia de elementos vacía, es decir, sólo el nodo cabecera. Así pues, el pseudocódigo de la operación Vacía (Fig. 26) consistirá únicamente en comprobar que tras la cabecera no hay ningún otro nodo. Otra forma de verlo es que en una Cola vacía tanto `inicio` como `final` apuntan al mismo nodo, que será el nodo cabecera, por lo tanto la operación Vacía podría consistir en comprobar que sus valores son iguales.

Por su parte, la operación Recupera devolverá el dato almacenado en el nodo situado a continuación de la cabecera, es decir, el siguiente al apuntado por el puntero `inicio`.

```
Inserta( x ) {
  NodoPtr nuevo = malloc( sizeof( struct Nodo ) );
  nuevo->dato = x;
  nuevo->sig = NULL;
  final->sig = nuevo;
  final = nuevo;
}
```

Fig. 27 Pseudocódigo de la operación Inserta en el TDA Cola implementado con listas

La operación Inserta (Fig. 27) creará un nuevo nodo y lo situará como último de la lista de nodos. Es decir, a continuación del apuntado por `final` y, una vez conectado, `final` se modificará para que apunte al nuevo último nodo.

```
Suprime( ) {
  NodoPtr borrar = inicio->sig;
  inicio->sig = borrar->sig;
  if ( borrar == final ) final = inicio;
  free( borrar );
}
```

Fig. 28 Pseudocódigo de la operación Suprime en el TDA Cola implementado con listas

La operación Suprime (Fig. 28) elimina el nodo que contiene el elemento que lleva más tiempo en la Cola, es decir, el siguiente a la cabecera. Y, además, comprueba si el nodo borrado coincide con el último de la estructura enlazada. Esta coincidencia sólo se da cuando se suprime en una cola que tiene un único elemento almacenado. En este caso el campo `final` debe modificarse haciendo que apunte a la cabecera de la estructura enlazada, pues será el único nodo que quede en la misma, y por tanto, el último.

Otra forma de resolver la operación Suprime (Fig. 29) consiste en borrar la cabecera y convertir el que hasta ese momento era el primer nodo con datos en la nueva cabecera. De este modo no es necesario actualizar el campo `final`, pues si antes de la supresión apuntaba al único nodo con datos de la estructura enlazada, tras la operación que se acaba de describir, estará apuntando a la nueva cabecera.

```
Suprime( ) {
  NodoPtr borrar = inicio;
  inicio = borrar->sig;
  free( borrar );
}
```

Fig. 29 Pseudocódigo alternativo de la operación Suprime en el TDA Cola implementado con listas

Para llevar a cabo la implementación del TDA Cola mediante estructuras enlazadas hay que escribir un fichero `.c` como el mostrado en la Fig. 30.

Como es habitual, la estructura `struct ColaRep` contiene todo lo necesario para representar al Tipo de Datos que, en este caso, son las variables `inicio y final`. En el mismo módulo, y oculto a los clientes del mismo, se han definido la estructura `struct Nodo` y el alias `NodoPtr` que permiten crear y manejar la estructura enlazada con la que se representará la Cola.

La función `crea`, además de reservar memoria para la estructura `struct ColaRep`, también crea e inicializa un nuevo nodo en memoria dinámica que será la cabecera de la estructura enlazada. A continuación, inicializa las variables `inicio y final` apuntando a dicho nodo.

Las funciones que implementan las operaciones Inserta, Suprime y Recupera son traducciones literales del pseudocódigo visto anteriormente, con la salvedad de que las variables `inicio y final` son campos de la estructura `struct ColaRep`, de ahí que en todos los casos se use `c->inicio y c->final` para referirse a ellas.

Por otro lado, la función `libera` se ocupa de forma iterativa de liberar la memoria asociada a todos los nodos de la estructura enlazada y, finalmente, de liberar la memoria asociada a la estructura `struct ColaRep` apuntada por el parámetro `c`.

En la implementación de las operaciones que tienen precondiciones: `suprime` y `recupera`, se ha usado la macro `assert` para comprobarlas y detener la ejecución cuando no se cumplan, lo cual es útil durante la fase de desarrollo.

Por otro lado, esta implementación no necesita la operación Llena porque, gracias al uso de la estructura enlazada para representar los elementos almacenados, el número de estos no necesita ser conocido a priori y, en principio, no esta limitado. Por lo tanto, esta implementación sí se cumple el interfaz original del TDA Cola y la función `llena` se puede quitar del fichero cabecera.

Comparación entre implementaciones

La implementación del TDA Cola con memoria contigua impone una limitación en el tamaño máximo de la Cola que la basada en estructuras enlazadas no tiene. Sin embargo, ésta incurre en una pequeña sobrecarga relativa al uso de memoria, porque cada nodo de la estructura enlazada incluye un puntero al siguiente.

En cuanto a la eficiencia de las operaciones, en ambas implementaciones se ha conseguido que todas las operaciones del TDA tengan un tiempo de ejecución constante e independiente del número de elementos de la Cola. La única excepción es la función libera en el caso de la implementación mediante estructuras enlazadas, pero esta operación no es la más frecuentemente usada y, por eso, queda fuera de la comparación.

```
#include "Cola.h"
#include <stdlib.h>
#include <assert.h>

struct Nodo {
  Elemento dato;
  struct Nodo * sig;
};

typedef struct Nodo * NodoPtr;

struct ColaRep {
  NodoPtr inicio;
  NodoPtr final;
};

Cola crea() {
  Cola nueva = malloc( sizeof( struct ColaRep ) );
  nueva->inicio = malloc( sizeof( struct Nodo ) );
  nueva->inicio->sig = NULL;
  nueva->final = nueva->inicio;
  return nueva;
}

void libera( Cola c ) {
  while ( c->inicio != NULL ) {
    NodoPtr borrado = c->inicio;
    c->inicio = borrado->sig;
    free( borrado );
  }
  free( c );
}

void inserta( Cola c, Elemento e ) {
  NodoPtr nuevo = malloc( sizeof( struct Nodo ) );
  nuevo->dato = e;
  nuevo->sig = NULL;
  c->final->sig = nuevo;
  c->final = nuevo;
}

void suprime( Cola c ) {
  assert( c->inicio->sig != NULL );
  NodoPtr borrar = c->inicio->sig;
  c->inicio->sig = borrar->sig;
  if ( borrar == c->final ) c->final = c->inicio;
  free( borrar );
}

Elemento recupera( Cola c ) {
  assert( c->inicio->sig != NULL );
  return c->inicio->sig->dato;
}

int vacia( Cola c ) {
  return c->inicio->sig == NULL;
}
```

Fig. 30. Fichero .c de un módulo que implementa el TDA Cola con estructuras enlazadas lineales

El Tipo De Datos Abstracto Lista

Las listas en el mundo real aparecen continuamente ya que los humanos tienen tendencia a ordenar las cosas, clasificarlas y enumerarlas para luego reordenarlas, buscar en ellas o contar cuántos elementos cumplen ciertas propiedades. Se hacen listas de todo usando muy diversos criterios de ordenación.

Las Listas en ciencias de la computación son muy parecidas a las del mundo real. Se representan gráficamente como secuencias de elementos separados por comas (Fig. 31) con subíndices que indican la posición de cada uno. Por ejemplo, una lista con cuatro elementos se representaría como la secuencia: (a_1, a_2, a_3, a_4). El elemento a_i siempre precede al a_{i+1} para cualquier 1 <= i < n. Esta ordenación es independiente de las características de los mismos, es decir, el orden viene impuesto por la posición que ocupan los elementos no por sus valores.

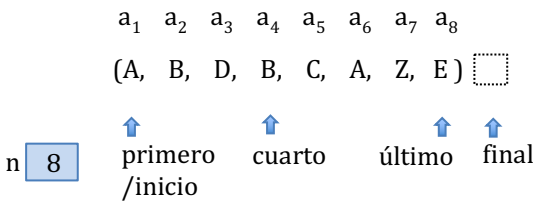

Fig. 31 Representación gráfica de una Lista

Formalmente, una Lista es una secuencia ordenada de elementos del mismo tipo en la que los accesos, inserciones y supresiones pueden realizarse en cualquier posición.

Las posiciones ocupadas por los elementos de una lista compuesta por **n** elementos van, desde la primera, o posición de inicio, correspondiente al elemento con subíndice 1, hasta la última, correspondiente al elemento con el subíndice **n**. Además, hay otra posición válida que, sin ser propiamente la de ninguno de los elementos pertenecientes a la Lista, es importante porque permite ampliarla. Se trata de la posición final que equivaldría a la **n+1**. Esta posición sería la que ocuparía un nuevo elemento añadido al "final" de la Lista.

Aunque no hay un conjunto estándar de operaciones que definan el TDA Lista, el siguiente conjunto permite realizar las operaciones más habituales directamente o mediante una combinación de varias operaciones:

- **Inserta(x, p)** (add) añade el elemento x situándolo en la posición p. El valor de p queda indefinido.
- **Suprime(p)** (remove) elimina el elemento situado en la posición p, siempre que ésta no sea la posición final. El valor de p queda indefinido.
- **Recupera(p)** (get) devuelve el elemento situado en la posición p, siempre que ésta no sea la posición final.
- **Asigna(x, p)** (set) sustituye por x el elemento situado en la posición p, siempre que ésta no sea la posición final.
- **Longitud()** (length) devuelve el número de elementos incluidos en la Lista.

Las cuatro primeras operaciones: Inserta, Suprime, Recupera y Asigna, requieren de una posición para actuar. Por tanto, además del Tipo de Datos Lista, también es necesario definir un Tipo de Datos que represente una posición dentro de una Lista.

El Tipo de Datos Posición es una referencia abstracta al lugar que ocupa un elemento dentro de una Lista, o al final de la misma, y sólo tiene dos operaciones:

- La operación **Siguiente(p)** (next) devuelve la posición siguiente a p, siempre que p no sea la posición final.
- La operación **Anterior(p)** (prev) devuelve la posición anterior a p, siempre que p no sea el inicio.

Una vez definido el Tipo de Datos Posición, es necesario añadir al interfaz del TDA Lista alguna operación que devuelva un elemento válido de este tipo. A continuación, se muestran tres operaciones que permiten obtener posiciones dentro de una Lista.

- La operación **Inicio()** (begin) devuelve la posición del primer elemento de la Lista, o la posición Final cuando la lista está vacía.
- La operación **Final()** (end) devuelve la posición correspondiente al siguiente elemento al último de la lista.
- La operación **i-esimo(i)** (nth) devuelve la posición correspondiente al i-ésimo elemento de la Lista, siempre que 1<=i<=longitud().

Una vez que se tiene una posición válida se puede obtener otra usando las operaciones: Siguiente y Anterior. Ahora bien, hay que tener en cuenta las precondiciones y los efectos de algunas de las operaciones. Por ejemplo, no es posible calcular la Posición anterior a Inicio ni la siguiente al Final. Y tras suprimir e insertar la posición utilizada deja de ser válida pues la estructura de la lista ha cambiado. Además, si la Lista está vacía las posiciones Inicio y Final coinciden.

Cualquiera de las tres operaciones por sí misma sería suficiente para poder acceder a todas las posiciones de la lista. Sin embargo, dado que los accesos por el principio y por el final son muy frecuentes resulta útil contar con las funciones Inicio y Final. Además, la función i-ésimo, sin ser esencial, es útil y permite comparar las dos implementaciones que se van a mostrar en cuanto a su eficiencia.

Implementación con memoria contigua

Para almacenar la secuencia de elementos en que consiste una Lista se puede usar un array. Dicho array se definirá en función del Tipo de Datos de los elementos a almacenar. También es necesario saber cuántas de las celdas del array contienen elementos válidos, es decir, la longitud de la Lista. Para esto se puede usar una variable de tipo entero que lleve la cuenta del número de elementos incluidos en la lista. Este número debe ser menor o igual que el tamaño máximo del array, que también será necesario conocer. Claramente, esta restricción implica que la Lista implementada será una Lista Acotada.

Además, hay que elegir una representación para el TDA Posición y, en este caso, los valores enteros asociados a los índices del array son una buena opción. Pero para que los índices se correspondan con las posiciones lógicas dentro de la Lista es necesario mantener los datos compactados. Es decir, los n datos que compongan la Lista deberán ocupar siempre las n primeras posiciones del array.

La compactación de los datos es necesaria ya que durante el uso normal de la Lista se pueden producir supresiones en cualquier posición intermedia. Los elementos eliminados dejan huecos en el array que corresponden a posiciones que no pueden quedar vacías. Por eso, cada vez que se elimine un elemento habrá que desplazar todos los elementos siguientes una celda a la izquierda, dejándolos colocados en las celdas que les corresponda estar tras la eliminación.

La Fig. 32 muestra un ejemplo en el que, tras suprimir el tercer elemento (b) se realiza una compactación (c).

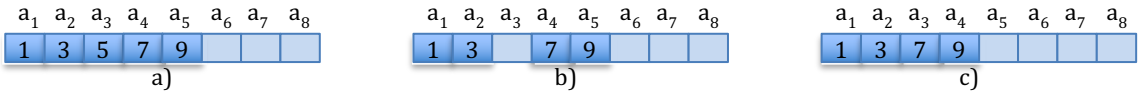

Fig. 32 Compactación de los datos en un array

Así pues, la Fig. 33 muestra la composición de la estructura `struct ListaRep` que se usará para representar al Tipo de Datos Lista Acotada con memoria contigua. En ella se incluye el array `datos` y el entero `n` que llevará la cuenta del número de elementos incluidos.

```
struct ListaRep {
  Elemento datos[MAX];
  int n;
};
```

Fig. 33. Estructura de datos para representar un TDA Lista con memoria contigua (arrays)

Con esta representación, las operaciones para obtener una posición válida, así como las de asignación, recuperación y cálculo de la longitud son muy sencillas, como se puede ver en la Fig. 34. Teniendo en cuenta que los índices ocupados por los n elementos de la lista irán desde 0 hasta `n-1`, la operación Inicio devuelve cero y Final devuelve `n`, que es el índice de un posible siguiente elemento. Claramente, como los índices empiezan en cero y las posiciones en 1, cada índice equivale a una posición una unidad mayor. Por lo tanto, la operación i-ésimo simplemente transforma de posición a índice restando uno.

```
Inicio( x ) {
  return 0;
}
```

```
Final( ) {
  return n;
}
```

```
i-ésimo( i ) {
  return i - 1;
}
```

```
Siguiente( p ) {
  return p + 1;
}
```

```
Anterior( p ) {
  return p - 1;
}
```

```
Asigna( x, p ) {
  datos[p] = x;
}
```

```
Recupera( p ) {
  return datos[p];
}
```

```
Longitud( ) {
  return n;
}
```

Fig. 34. Pseudocódigo inicial de varias operaciones del TDA Lista implementado con arrays

Las operaciones Asigna y Recupera simplemente acceden al array en la posición indicada para guardar o devolver un elemento. Siempre que se cumplan las precondiciones, estas posiciones serán un entero entre 0 y `n-1`. Y, finalmente, la operación Longitud se limita a devolver el valor de la variable n.

Sin embargo, las inserciones y supresiones implican realizar desplazamientos. La operación Inserta (Fig. 35) requiere desplazar los elementos a la derecha para hacer hueco al nuevo elemento, y la operación Suprime debe realizar desplazamientos a la izquierda para mantener los datos compactados tras la eliminación.

```
Inserta( x, p ) {
  int i = n;
  while ( i > p ) {
    datos[i] = datos[i - 1];
    i = i - 1;
  }
  datos[p] = x;
  n = n + 1;
}
```

```
Suprime( p ) {
  int i = p;
  while ( i < n - 1 ) {
    datos[i] = datos[i + 1];
    i = i + 1;
  }
  n = n - 1;
}
```

Fig. 35. Pseudocódigo de las operaciones Inserta y Suprime del TDA Lista implementado con arrays

Es interesante observar las diferencias entre los dos bucles usados para desplazar elementos. El de la operación Inserta tiene como objetivo abrir un hueco donde colocar el nuevo elemento. Para conseguirlo mueve una posición a la derecha todos los elementos situados a la derecha de la posición que se desea liberar, empezando por el ultimo de la lista: el que ocupa la posición `n-1`. Por otro lado, el bucle de la operación Suprime pretende compactar el array tras haber eliminado un elemento. Por lo tanto, mueve una posición a la izquierda todos los elementos que han quedado a la derecha del hueco, empezando por el que hay más cerca y acabando por el último de la lista. Además, ambas funciones modifican el valor de la variable `n` para mantenerla actualizada en todo momento.

De cara a implementar en C el TDA Lista se debe escribir un módulo. La Fig. 36 muestra el fichero cabecera de dicho módulo. Como puede verse además de las operaciones descritas, es necesario incluir una operación de creación y otra de liberación que permitan crear y liberar Listas desde el código cliente del TDA.

```
#ifndef __Lista_H__
#define __Lista_H__

typedef int Elemento;
typedef int Posicion;
typedef struct ListaRep * Lista;

Lista crea();
void libera( Lista l );
void inserta( Lista l, Posicion p, Elemento e );
void suprime( Lista l, Posicion p );
Elemento recupera( Lista l, Posicion p );
void asigna( Lista l, Posicion p, Elemento e );
int longitud( Lista l );
Posicion inicio( Lista l );
Posicion final( Lista l );
Posicion iesima( Lista l, int indice );
Posicion siguiente( Lista l, Posicion p );
Posicion anterior( Lista l, Posicion p );
int llena( Lista l );

#endif
```

Fig. 36. Fichero cabecera de un módulo que implementa el TDA Lista con memoria contigua (arrays)

En este módulo también se está ofreciendo un segundo Tipo de Datos que es el que sirve para representar posiciones dentro de las Listas, por lo tanto, la definición de `Posicion` y el interfaz de sus operaciones aparece en este mismo fichero.

Como ya se ha comentado, en este caso, una Posición está representada por un entero y, por simplicidad, se ha decidido no ocultar su representación. Así pues, en este caso, `Posición` es un Tipo de Datos no un Tipo de Datos Abstracto.

Además, dado que se está implementando una Lista Acotada, es necesario añadir otra operación que devuelva verdadero cuando la Lista está llena

El último aspecto interesante del fichero cabecera es que en él también se define el Tipo de Datos `Elemento` como un alias de `int`. De este modo, el TDA Lista implementado en este ejemplo es capaz de almacenar valores enteros.

La Fig. 37 muestra un módulo .c en el que se implementa el TDA Lista Acotada de enteros usando un array estático en la estructura que representa la Lista.

Al principio del fichero `.c`, se incluye el fichero de cabecera `Lista.h` para conseguir que los Tipos de Datos `Elemento`, `Posicion` y `Lista`, así como las declaraciones de las funciones, estén disponibles en este fichero. Aquí es donde se define la estructura struct `ListaRep` que representará el Tipo de Datos Lista mediante la agrupación del array `datos`, y la variable `n`.

La función `crea` reserva memoria para la estructura de datos que representa la Lista, e inicializa el campo `n` a cero indicando que la Lista está recién creada y, por tanto, vacía. La función `libera` simplemente libera la memoria asociada a la estructura que representa la Lista.

El resto de funciones son traducciones literales del pseudocódigo visto anteriormente, con la salvedad de que las variables son campos de la estructura struct `ListaRep`, de ahí que en todos los casos se use `lista->` para acceder a ellas.

La limitación de tamaño de la Lista hace que, al igual que pasa con Recupera y Suprime, haya que imponer una precondición a la operación Inserta, puesto que ésta no se podrá llevar a cabo si la Lista está llena. Teniendo esto en cuenta, en la implementación de las operaciones que tienen precondiciones: `inserta`, `suprime`, `recupera, asigna, siguiente y anterior`, se ha usado la macro `assert` para comprobarlas y detener la ejecución cuando no se cumplan, lo cual es útil durante la fase de desarrollo.

Por otro lado, las funciones `siguiente` y `anterior` no usan el parámetro `lista`, pero se incluye para que el interfaz del Tipo de Dato `Posicion` sea homogéneo. En otras implementaciones sí será necesario usar el parámetro `lista` para poder realizar alguna de esas operaciones.

Por último, hay que resaltar que la Lista Acotada que se acaba de implementar tiene un problema de diseño importante. El tamaño máximo queda prefijado en la implementación de la misma. Por lo tanto, todas las Listas tendrán siempre dicho tamaño, y puede resultar demasiado grande en unos casos y demasiado pequeño en otros. Una posible solución sería mover la definición de MAX al fichero cabecera, pero incluso en ese caso, todas las Listas de una misma aplicación tendrían que tener el mismo tamaño máximo, lo que puede ser poco eficiente. Más adelante se verán propuestas de mejora que permitan crear Listas Acotadas con un tamaño máximo especificado en el momento de su creación.

```c
#include "ListaArray.h"
#include <stdlib.h>
#include <assert.h>

#define MAX  50

struct ListaRep {
  Elemento datos[MAX];
  int n;
};

Lista crea() {
  Lista nueva = malloc( sizeof( struct ListaRep ) );
  nueva->n = 0;
  return nueva;
}

void libera( Lista lista ) {
  free( lista );
}

void inserta( Lista lista, Posicion p, Elemento e ) {
  assert( lista->n < MAX );
  int i = lista->n;
  while ( i > p ) {
    lista->datos[i] = lista->datos[i - 1];
    i = i - 1;
  }
  lista->datos[p] = e;
  lista->n = lista->n + 1;
}

void suprime( Lista lista, Posicion p ) {
  assert( p != lista->n );
  int i = p;
  while ( i < lista->n - 1 ) {
    lista->datos[i] = lista->datos[i + 1];
    i = i + 1;
  }
  lista->n = lista->n - 1;
}

Elemento recupera( Lista lista, Posicion p ) {
  assert( p < lista->n );
  return lista->datos[p];
}

void asigna(Lista lista, Posicion p, Elemento e) {
  assert( p < lista->n );
  lista->datos[p] = e;
}

int longitud( Lista lista ) {
  return lista->n;
}

int vacia( Lista lista ) {
  return lista->n == 0;
}

int llena( Lista lista ) {
  return lista->n == MAX;
}
...
```

```c
...
Posicion inicio( Lista lista ) {
  return 0;
}

Posicion final( Lista lista ) {
  return lista->n;
}

Posicion siguiente( Lista lista, Posicion p ) {
  assert( p < lista->n );
  return p + 1;
}

Posicion anterior( Lista lista, Posicion p ) {
  assert( p > 0 );
  return p - 1;
}

Posicion iesima( Lista lista, int indice ) {
  return indice – 1;
}
```

Fig. 37. Fichero .c de un módulo que implementa el TDA Lista con memoria contigua (arrays)

Implementación con estructuras enlazada

Para implementar el TDA Lista con estructuras enlazadas se puede usar cada nodo de la estructura enlazada para almacenar cada uno de los elementos de la Lista. De este modo se mantiene el orden lógico de la misma. Además, hay que elegir una representación para las posiciones, de modo que sea posible acceder a cualquier elemento de la lista, modificarlo, suprimirlo e insertar otros nuevos.

En una lista implementada mediante una estructura enlazada lineal es fácil eliminar un elemento de cualquier nodo. Basta con modificar el campo "siguiente" del nodo que hay antes para que apunte al nodo que haya a continuación del que se quiere borrar.

Del mismo modo es fácil insertar un nuevo elemento a continuación de cualquier nodo. En ambos casos lo único que se necesita es tener acceso al nodo anterior al que contiene el elemento que interesa. Por lo tanto, la posición se puede definir como la dirección del nodo anterior al que contenga el elemento en cuestión. Si además se usa la técnica del nodo cabecera, las operaciones se podrán hacer sin casos excepcionales como el de borrar el primer nodo o tratar con la lista vacía.

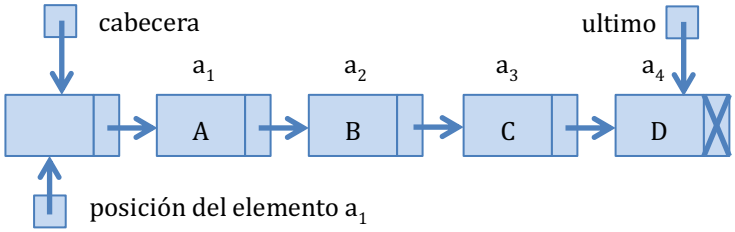

Fig. 38 Esquema de uso de una estructura enlazada lineal para representar una Lista

Para gestionar la estructura enlazada se necesita un puntero a la cabecera desde el cual se pueda acceder a todos los elementos recorriendo la lista. Este puntero externo, que podría llamarse `cabecera`, permite implementar la operación Inicio. Sin embargo, si sólo se cuenta con ese puntero externo las inserciones por el final resultarían costosas, pues sería necesario recorrer toda la estructura enlazada hasta encontrar el último nodo. Para implementar la operación Final de forma eficiente se puede usar, como en el caso del TDA Cola, un segundo puntero externo que apunte al último nodo de la lista, es decir, el anterior al que ocupará un futuro elemento insertado al "final" de la lista. Este puntero se podría llamar `ultimo`, pues apunta al nodo que contiene el último elemento, aunque "final" también sería un buen nombre, pues es la posición que representa.

```
struct PosicionRep {
  Elemento dato;
  struct PosicionRep * sig;
};

tpedef struct PosicionRep * NodoPtr;

struct ListaRep {
  NodoPtr cabecera;
  NodoPtr ultimo;
};
```

Fig. 39. Estructura de datos para representar un TDA Lista con estructuras enlazadas lineales

La Fig. 39 muestra la definición de todo lo necesario para usar una estructura enlazada lineal: la estructura `struct PosicionRep` y el alias `typedef NodoPtr`. Además, muestra la estructura `struct ListaRep` que representará una Lista agrupando los punteros `cabecera` y `ultimo`, que apuntarán respectivamente a la cabecera de la estructura enlazada y al último nodo de la misma.

En este caso, se ha sustituido el nombre `Nodo` por `PosicionRep`, de modo que cuando en el fichero cabecera se defina el tipo `Posicion` como un puntero a esta estructura no se den pistas sobre cómo está implementada la misma.

```
Inicio( x ) {
  return cabecera;
}
```

```
Final( ) {
  return ultimo;
}
```

```
Siguiente( p ) {
  return p->sig;
}
```

Fig. 40. Pseudocódigo de las operaciones Inicio, Final y Siguiente del TDA Lista implementado con estructuras enlazadas

Con la definición de Posición que se ha decidido, las operaciones Inicio y Final son directas gracias a las variables `cabecera` y `ultimo`. Y una vez que se dispone de una posición válida también es muy sencillo encontrar la siguiente, pues basta con acceder al campo `sig` del nodo en cuestión.

```
i-ésimo( indice ) {
  NodoPtr aux = cabecera;
  int i = 1;
  while ( i < indice ) {
    aux = aux->sig;
    i = i + 1;
  }
  return aux;
}
```

```
Anterior( p ) {
  NodoPtr aux = cabecera;
  while ( aux->sig != p ) {
    aux = aux->sig;
  }
  return aux;
}
```

Fig. 41. Pseudocódigo de i-ésimo y Anterior del TDA Lista implementado con estructuras enlazadas

Sin embargo, las operaciones i-ésimo y Anterior tienen un tiempo de ejecución de O(n), pues necesitan recorrer la estructura enlazada desde el principio. En el caso de i-esimo se recorre hasta el nodo anterior al que contiene el i-ésimo elemento. En el caso de la operación Anterior se recorre la estructura enlazada hasta encontrar el nodo anterior al apuntado por la posición recibida como parámetro.

```
Asigna( x, p ) {
  p->sig->dato = x;
}
```

```
Recupera( p ) {
  return p->sig->dato;
}
```

Fig. 42. Pseudocódigo de Asigna y Recupera del TDA Lista implementado con estructuras enlazadas

Las operaciones Asigna y recupera también tienen una solución directa usando el puntero recibido como parámetro que representa la posición del nodo en cuestión.

Las inserciones y supresiones dada una posición se hacen con un tiempo de ejecución de O(1), pues consisten en modificar un par de punteros pero es muy importante que, además de hacer su función principal, actualicen el puntero al último nodo cuando el nodo borrado sea el último hasta ese momento, o cuando se inserte un nuevo nodo al final de la Lista.

```
Inserta( x, p ) {
  NodoPtr nuevo = malloc(sizeof(struct PosicionRep));
  nuevo->dato = x;
  nuevo->sig = p->sig;
  p->sig = nuevo;
  if ( ultimo == p ) {
    ultimo = nuevo;
  }
}
```

Fig. 43. Pseudocódigo de la operación Inserta del TDA Lista implementado con estructuras enlazadas

En concreto, al insertar (Fig. 43) un nodo a continuación del último, es decir, cuando la posición en la que se va a insertar es la misma que la que representa el final de la lista, el nuevo nodo insertado pasará a ser el último de la estructura enlazada.

Por su parte, la operación Suprime (Fig. 44) detecta cuándo se ha borrado el último elemento y, gracias a que para hacer esto se necesita la posición del nodo anterior, es posible actualizar la variable `ultimo` con este valor.

```
Suprime( p ) {
  NodoPtr borrar = p->sig;
  p->sig = borrar->sig;
  if ( ultimo == borrar ) {
    ultimo = p;
  }
  free( borrar );
}
```

Fig. 44. Pseudocódigo de la operación Suprime del TDA Lista implementado con estructuras enlazadas

Con la representación del TDA Lista vista hasta el momento la operación Longitud (Fig. 45) tendría un tiempo de ejecución de O(n), puesto que debe recorrer toda la estructura enlazada para contar el número de elementos que contiene.

```
Longitud( ) {
  int n = 0;
  NodoPtr aux = cabecera;
  While ( aux->sig != NULL ) {
    n = n + 1;
    aux = aux->sig;
  }
  return n;
}
```

Fig. 45. Pseudocódigo de la operación Longitud del TDA Lista implementado con estructuras enlazadas

De cara a implementar en C el TDA Lista se debe escribir un módulo. En este caso, el fichero cabecera debe modificarse con respecto al que se usó en la implementación con arrays. El motivo es que la Posición ahora será un puntero a la estructura `struct PosicionRep` en lugar de un entero. La diferencia es importante pues, en este caso, sí que se consigue ocultar la representación del Tipo de Datos `Posición`, por lo que ahora sí se trata de un TDA en toda regla. La Fig. 46 muestra el fichero cabecera que, por lo demás es idéntico al anterior.

```
#ifndef __Lista_H__
#define __Lista_H__

typedef int Elemento;
typedef struct PosicionRep * Posicion;
typedef struct ListaRep * Lista;

Lista crea();
void libera( Lista l );
void inserta( Lista l, Posicion p, Elemento e );
void suprime( Lista l, Posicion p );
Elemento recupera( Lista l, Posicion p );
void asigna( Lista l, Posicion p, Elemento e );
int longitud( Lista l );
Posicion inicio( Lista l );
Posicion final( Lista l );
Posicion iesima( Lista l, int indice );
Posicion siguiente( Lista l, Posicion p );
Posicion anterior( Lista l, Posicion p );

#endif
```

Fig. 46. Fichero cabecera de un módulo que implementa el TDA Lista con estructuras enlazadas

El fichero `.c` con la implementación del módulo puede verse en la Fig. 47. Como es habitual, la estructura `struct ListaRep` contiene todo lo necesario para representar al Tipo de Datos que, en este caso, son las variables `cabecera` y `ultimo`. En el mismo fichero, y oculto a los clientes del módulo, se ha definido la estructura `struct PosicionRep` que permite crear y manejar la estructura enlazada con la que se representará la Lista, y que además es la representación del TDA Posición. También se define el alias `NodoPtr` para facilitar la escritura del código.

La función `crea`, además de reservar memoria para la estructura `struct ListaRep` también crea e inicializa un nuevo nodo en memoria dinámica que será la cabecera de la estructura enlazada. A continuación, inicializa las variables `cabecera` y `ultimo` apuntando a dicho nodo.

Las funciones que implementan las operaciones Inserta, Suprime, Recupera, Inicio y Final son traducciones literales del pseudocódigo visto anteriormente con la salvedad de que las variables `cabecera` y `ultimo` son campos de la estructura `struct ListaRep`, de ahí que en todos los casos se use `lista->cabecera` y `lista->ultimo` para referirse a ellas.

Por otro lado, la función `libera` se ocupa de forma iterativa de liberar la memoria asociada a todos los nodos de la estructura enlazada, y finalmente, de liberar la memoria asociada a la estructura `struct ListaRep` apuntada por el parámetro `lista`.

En la implementación de las operaciones que tienen precondiciones: `recupera`, `asigna`, `suprime`, `siguiente` y `anterior`, se ha usado la macro `assert` para comprobarlas y detener la ejecución cuando no se cumplan, lo cual es útil durante la fase de desarrollo.

```c
#include "Lista.h"
#include <stdlib.h>
#include <assert.h>

struct PosicionRep {
  Elemento dato;
  struct PosicionRep * sig;
};

struct ListaRep {
  Posicion cabecera;
  Posicion ultimo;
};

typedef struct PosicionRep * NodoPtr;

Lista crea() {
  Lista nueva = malloc( sizeof( struct ListaRep ) );
  nueva->cabecera = malloc( sizeof( struct PosicionRep ) );
  nueva->cabecera->sig = NULL;
  nueva->ultimo = nueva->cabecera;
  return nueva;
}

void libera( Lista lista ) {
  while ( lista->cabecera != NULL ) {
    NodoPtr borrar = lista->cabecera;
    lista->cabecera = borrar->sig;
    free( borrar );
  }
  free( lista );
}

void inserta( Lista lista, Posicion p, Elemento e ) {
  NodoPtr nuevo = malloc( sizeof( struct PosicionRep ) );
  nuevo->dato = e;
  nuevo->sig = p->sig;
  p->sig = nuevo;
  if ( lista->ultimo == p ) {
    lista->ultimo = nuevo;
  }
}

Elemento recupera( Lista lista,
                   Posicion p ) {
  assert( p != lista->ultimo  );
  return p->sig->dato;
}

void asigna( Lista lista, Posicion p,
                   Elemento e ) {
  assert( p != lista->ultimo  );
  p->sig->dato = e;
}

void suprime( Lista lista, Posicion p) {
  assert( p != lista->ultimo  );
  NodoPtr borrar = p->sig;
  p->sig = borrar->sig;
  if ( lista->ultimo == borrar ) {
    lista->ultimo = p;
  }
  free( borrar );
}

int longitud( Lista lista ) {
  int n = 0;
  NodoPtr aux = lista->cabecera;
  while ( aux->sig != NULL ) {
    n = n + 1;
    aux = aux->sig;
  }
  return n;
}
```

```c
...
Posicion inicio( Lista lista) {
  return lista->cabecera;
}

Posicion final( Lista lista) {
  return lista->ultimo;
}

Posicion siguiente(Lista lista, Posicion p) {
  assert( p != lista->ultimo  );
  return p->sig;
}

Posicion anterior(Lista lista, Posicion p) {
  assert( p != lista->cabecera  );
  NodoPtr aux = lista->cabecera;
  while ( aux->sig != p ) {
    aux = aux->sig;
  }
  return aux;
}

Posicion iesima( Lista lista, int indice ) {
  NodoPtr aux = lista->cabecera;
  int i = 1;
  while ( i < indice ) {
    aux = aux->sig;
    i = i + 1;
  }
  return aux;
}
```

Fig. 47. Fichero .c de un módulo que implementa el TDA Lista con estructuras enlazadas

Mejoras a la implementación del TDA Lista con estructuras enlazadas

La operación Longitud en la implementación del TDA Lista realizada con estructuras enlazadas se diseñó asumiendo que la representación del TDA Lista incluye únicamente las variables `cabecera` y `ultimo`.

Para mejorar la eficiencia de la operación Longitud basta con añadir a la estructura `struct ListaRep` una variable de tipo entero que lleve la cuenta de los elementos incluidos en la lista. Esta variable, que se podría llamar `n`, habrá que actualizarla tras cada inserción y supresión, pero este añadido al código mejora mucho la eficiencia de la operación Longitud, ya que su tiempo de ejecución pasa de ser de O(n) a ser de O(1) como muestra la Fig. 48.

```
Longitud( ) {
  return n;
}
```

Fig. 48. Optimización de la operación Longitud del TDA Lista con estructuras enlazadas

Por otro lado, para mejorar la eficiencia de la operación Anterior se puede recurrir a un esquema de doble enlace. En este caso, cada nodo tendría dos punteros, uno hacia el siguiente nodo de la estructura enlazada (`sig`) y otro hacia el anterior (`ant`). Como el nodo cabecera tiene la misma estructura que los demás también tendrá un campo `ant`. Este campo se puede hacer que apunte al último nodo y, al mismo tiempo, el campo `sig` del último nodo se puede hacer que apunte a la cabecera. Esta forma de actuar da como lugar una estructura enlazada de doble enlace con un esquema circular como el mostrado en la Fig. 49.

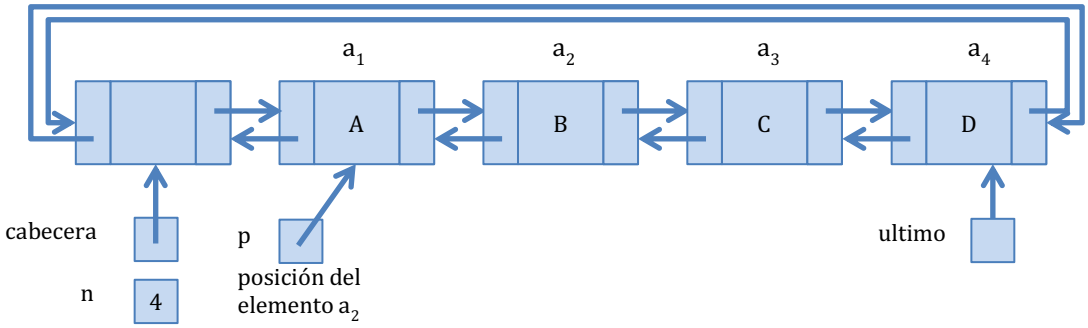

Fig. 49 Esquema de una estructura enlazada de doble enlace circular para representar el TDA Lista

Una ventaja adicional de seguir este esquema es que la operación Final se puede implementar usando el puntero anterior de la cabecera, por lo que deja de ser necesario el segundo puntero externo que se usaba para mantener la posición del último nodo de la estructura. Para que este esquema funcione basta con hacer que los punteros `sig` y `ant` de la cabecera apunten a sí misma en la creación de la Lista vacía.

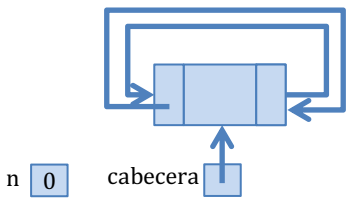

Fig. 50 Esquema de una Lista vacía implementada con estructura enlazada de doble enlace circular

```
struct PosicionRep {
  Elemento dato;
  struct PosicionRep * sig;
  struct PosicionRep * ant;
};

tpedef struct PosicionRep * NodoPtr;

struct ListaRep {
  NodoPtr cabecera;
  int n;
};
```

Fig. 51. Estructura de datos para representar un TDA Lista con estructuras enlazadas lineales de doble enlace

La Fig. 51 muestra la definición de todo lo necesario para usar una estructura enlazada lineal de doble enlace: la estructura `struct PosicionRep` y el alias `typedef NodoPtr`. Además, muestra la estructura `struct ListaRep` que representará una Lista agrupando el puntero `cabecera` y el entero `n`.

Dado que con esta representación sí es posible acceder al nodo anterior, el TDA Posición se podría haber definido como la dirección del nodo que contiene el elemento en cuestión en lugar de la del nodo anterior. Sin embargo, esto implica tratar con el caso especial que aparece cuando la Lista está vacía. Por esa razón es más adecuado mantener la definición de Posición como estaba.

Pero el doble enlace permite que la operación Anterior se pueda llevar a cabo eficientemente (Fig. 52). Bastará con acceder al campo `ant` del nodo apuntado. Y, como ya se ha comentado, la operación Final se resuelve accediendo al campo `ant` de la cabecera. Las operaciones Inicio y Siguiente quedan como estaban al igual que Asigna y Recupera.

```
Inicio( x ) {
  return cabecera;
}
```

```
Final( ) {
  return cabecera->ant;
}
```

```
Siguiente( p ) {
  return p->sig;
}
```

```
Anterior( p ) {
  return p->ant;
}
```

```
Asigna( x, p ) {
  p->sig->dato = x;
}
```

```
Recupera( p ) {
  return p->sig->dato;
}
```

Fig. 52. Pseudocódigo de las operaciones Inicio, Final, Siguiente, Anterior, Asigna y Recupera del TDA Lista implementado con estructuras enlazadas de doble enlace

Sin embargo, las operaciones Inserta y Suprime sufren cambios importantes, pues ahora se debe actualizar, además de los enlaces al siguiente nodo, los enlaces al anterior. También se debe mantener actualizada la variable `n`, pero ya no es necesario preocuparse de actualizar la variable `ultimo`, pues ya no está en la representación.

```
Inserta( x, p ) {
  NodoPtr nuevo = malloc(sizeof(struct PosicionRep));
  nuevo->dato = x;
  nuevo->sig = p->sig;
  p->sig = nuevo;
  nuevo->sig->ant = nuevo;
  nuevo->ant = p;
  n = n + 1;
}
```

Fig. 53. Operación Inserta del TDA Lista implementado con estructuras enlazadas de doble enlace

En concreto, al insertar un nuevo nodo se añade código para modificar el campo `ant` del nodo siguiente haciendo que apunte al nuevo, y, además, se debe hacer que el campo `ant` del nuevo nodo apunte al apuntado por el parámetro `p`. Finalmente se incrementa el contador de elementos sumando uno a la variable `n`.

Por su parte, la operación Suprime (Fig. 54) debe modificar el campo `ant` del nodo siguiente al que se está eliminando para que apunte al anterior del borrado. Es decir, para que apunte al mismo nodo al que apunta el parámetro `p`. Y, como siempre, el siguiente de `p` debe pasar a ser el siguiente del eliminado. Finalmente se reduce el contador de elementos restándole uno a la variable `n`.

```
Suprime( p ) {
  NodoPtr borrar = p->sig;
  p->sig = borrar->sig;
  borrar->sig->ant = p;
  free( borrar );
  n = n – 1;
}
```

Fig. 54. Operación Suprime del TDA Lista implementado con estructuras enlazadas de doble enlace

Comparación entre implementaciones

En comparación con la implementación basada en arrays, la implementación con estructuras enlazadas lineales de simple enlace, además de un tamaño máximo ilimitado y dinámico, permite inserciones y supresiones eficientes, con un tiempo de ejecución constante e independiente del número de elementos de la Lista. Sin embargo, las operaciones i-ésimo y Anterior son de O(n) cuando en la implementación con arrays son de O(1) gracias al acceso indexado que permiten los arrays. Además, se usa más memoria por elemento que con la representación basada en arrays debido a que cada elemento va acompañado de un puntero.

La implementación con estructuras enlazadas lineales de doble enlace conserva las ventajas de la de simple enlace y mejora la eficiencia de la operación Anterior. A cambio, el uso de memoria aumenta al tener dos punteros por Nodo.

Por lo tanto, la implementación con estructuras enlazadas es la más adecuada cuando la aplicación que la usa necesita hacer muchas inserciones y supresiones y no se conoce el número máximo de elementos con los que se va a trabajar. Si los recorridos hacia atrás son prioritarios, y el sobrecoste en el uso de memoria por nodo no es un problema, la mejor opción es la lista de doble enlace. Y si los accesos a posiciones aleatorios son la prioridad, se puede establecer a priori el número máximo de elementos y el consumo de memoria se debe mantener bajo, la implementación con arrays será la más adecuada.

El Tipo De Datos Abstracto Conjunto

En matemáticas se usan los conjuntos para referirse a los elementos que cumplen cierta propiedad. En la vida real también se usan los conjuntos para distinguir o clasificar elementos en función de alguna característica. En ambos casos, lo que diferencia un conjunto de otro tipo de colecciones es que sus elementos no están repetidos.

Una Lista también puede contener elementos no repetidos, pero, mientras que la Lista implica cierto orden o relación de precedencia entre sus elementos, los Conjuntos no imponen ningún orden a sus elementos. Por tanto, los Conjuntos se utilizan cuando lo único que interesa es la idea de pertenencia.

Formalmente, un conjunto es una colección de elementos no repetidos del mismo tipo. Y aunque hay muchas operaciones que se pueden hacer con Conjuntos, el interfaz que se propone aquí es reducido. Aun así, también es completo, pues permite construir cualquier otra operación utilizando sólo las incluidas.

- **Inserta(x)** (add) si x no pertenece al Conjunto lo añade.
- **Suprime(x)** (remove) si x pertenece al Conjunto lo elimina.
- **Pertenece(x)** (contains) devuelve verdadero si x está en el conjunto y falso en caso contrario.
- **Cardinalidad()** (size) devuelve la cantidad de elementos del Conjunto.
- **Enumera()** (toList) devuelve una Lista con todos los elementos del conjunto.

Implementación con estructuras enlazadas lineales

Una estructura enlazada se puede utilizar para almacenar los distintos elementos incluidos en el conjunto, pero se debe garantizar la no existencia de elementos repetidos. La decisión sobre si usar cabecera o no, y sobre la conveniencia de usar doble enlace o no, depende del uso que se vaya a hacer de la estructura enlazada, por lo que primero hay que estudiar las operaciones del TDA Conjunto.

La operación Inserta requiere hacer una búsqueda inicial, ya que sólo si no se encuentra se procederá a su inserción. La inserción del elemento se puede llevar a cabo en cualquier punto de la estructura enlazada, pues el orden no es importante.

Como es lógico, la operación Suprime también requiere de una búsqueda previa, pero la supresión habrá que hacerla allí donde se haya encontrado el elemento. Este puede ser el primero, el último u ocupar cualquier posición intermedia.

La operación Pertenece es en sí misma la citada búsqueda, y como tal, va a suponer un recorrido de la estructura enlazada, por lo que su tiempo de ejecución será de O(n). Por consiguiente, tanto Inserta como Suprime también serán de O(n).

La operación Cardinalidad se puede llevar a cabo recorriendo la estructura enlazada para contar los elementos incluidos. Sin embargo, resulta mucho más eficiente usar una variable de tipo entero que se actualice al insertar y suprimir, y que lleve así la cuenta de los elementos incluidos en el conjunto. De este modo se consigue que Cardinalidad sea una operación de O(1).

Por último, la operación Enumera es esencialmente un recorrido en el que se crea una nueva Lista conteniendo todos los elementos incluidos en el conjunto. Por tanto, tendrá un tiempo de ejecución de O(n)

Tras estudiar las operaciones parece claro que no es necesario usar una estructura doblemente enlazada, pues nunca se necesita recorrerla en sentido inverso. Pero sí conviene usar la estrategia del nodo cabecera, pues de no hacerlo, las posibles supresiones por el principio habría que tratarlas con un caso especial en el código.

```
struct Nodo {
  Elemento dato;
  struct Nodo * sig;
};

typedef struct Nodo * NodoPtr;

struct ConjuntoRep {
  NodoPtr cabecera;
  int n;
};
```

Fig. 55. Estructura de datos usada para representar un TDA Conjunto con estructuras enlazadas lineales

Así pues, la Fig. 55 muestra todo lo necesario para definir la estructura `struct ConjuntoRep` que representará un Conjunto mediante estructuras enlazadas lineales. En concreto, la estructura `struct Nodo` y el alias `typedef NodoPtr`. La estructura `struct ConjuntoRep` agrupa el puntero `cabecera` y el entero llamado `n`. El primero apuntará al nodo cabecera de la estructura enlazada que contendrá los elementos del Conjunto, y el segundo almacenará el número de elementos del Conjunto.

```
Pertenece( x ) {
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL && aux->sig->dato != x ) {
    aux = aux->sig;
  }
  return aux->sig != NULL;
}
```

Fig. 56. Operación Pertenece del TDA Conjunto implementado con estructuras enlazadas lineales

La Fig. 56 muestra el pseudocódigo de la Operación Pertenece. Como se puede ver, se trata de una búsqueda secuencial que acaba al llegar al final de la estructura enlazada o cuando `aux` apunta al nodo anterior al que contiene el dato buscado. Pero en ambos casos `aux` siempre acaba apuntando a algún nodo.

Sin embargo, el aspecto más interesante de esta búsqueda es la comparación entre elementos. Es necesario decidir cómo comparar si dos elementos son iguales o no. Es decir, los elementos deben ser distinguibles y, evidentemente, esto depende del Tipo de Datos de los elementos que se vayan a almacenar en el Conjunto.

La forma más sencilla de abordar este asunto, que es a su vez la menos genérica, consiste en incluir en la operación Pertenece el código específico para comparar valores del Tipo de Datos de los Elementos a guardar. Sin embargo, esto hace que esta implementación de TDA Conjunto sólo valga para ese Tipo de Datos en particular. De ahí que sea la solución menos genérica.

En C se podría resolver el problema con punteros a funciones, pero cuando se estudien otros lenguajes de programación que soportan Orientación a Objetos se verán soluciones mucho más elegantes y generales.

```
Inserta( x ) {
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL && aux->sig->dato != x ) {
    aux = aux->sig;
  }
  if ( aux->sig == NULL ) {
    NodoPtr nuevo = malloc( sizeof( struct Nodo ) );
    nuevo->sig = aux->sig;
    aux->sig = nuevo;
    n = n + 1;
  }
}
```

Fig. 57. Operación Inserta del TDA Conjunto implementado con estructuras enlazadas lineales

La Fig. 57 muestra el pseudocódigo de la Operación Inserta. Como se puede ver, tras realizar la misma búsqueda que en la operación Pertenece, se lleva a cabo la inserción, pero sólo si no se ha encontrado el elemento. Es decir, sólo si aux apunta al último nodo de la estructura enlazada. La inserción se hace justo a continuación de dicho nodo y, además, se incrementa el valor de la variable n.

```
Suprime( x ) {
  NodoPtr aux = cabecera;
  while ( aux->sig != NULL && aux->sig->dato != x ) {
    aux = aux->sig;
  }
  if ( aux->sig != NULL ) {
    NodoPtr borrar = aux->sig;
    aux->sig = borrar->sig;
    free( borrar );
    n = n - 1;
  }
}
```

Fig. 58. Operación Suprime del TDA Conjunto implementado con estructuras enlazadas lineales

La Fig. 58 muestra el pseudocódigo de la Operación Suprime. Como se puede ver, tras realizar la misma búsqueda que en las operaciones Pertenece e Inserta se lleva a cabo la supresión, pero sólo si se ha encontrado el elemento. Es decir, sólo si aux no apunta al último nodo. Tras eliminar el nodo se le resta uno a la variable n.

```
Enumera( x ) {
  Lista nueva = Crea();
  NodoPtr aux = cabecera->sig;
  while ( aux != NULL ) {
    Inserta( nueva, aux->dato, Final( nueva ) );
    aux = aux->sig;
  }
  return nueva;
}
```

Fig. 59. Operación Enumera del TDA Conjunto implementado con estructuras enlazadas lineales

La operación Enumera (Fig. 59) en muy simple, pero demuestra el uso del TDA Lista. Lo más importante de esta operación es entender que en ningún caso se debe devolver la estructura enlazada usada en la representación para no romper el principio de protección de los datos. De hacerlo se permitiría que el usuario del TDA Conjunto accediera a los datos sin usar las operaciones públicas del interfaz.

Finalmente, la operación Cardinalidad consiste únicamente en devolver el valor de la variable `n`. Si al crear el Conjunto se inicializa a cero y se actualiza en las inserciones y supresiones, en todo momento representará el número de elementos incluidos en el Conjunto.

Como ya se ha comentado, en C no hay una forma simple de implementar el TDA Conjunto sin depender del Tipo de Datos de los elementos a almacenar. Por lo tanto, aquí, se muestra un ejemplo concreto: el TDA Conjunto de enteros.

La Fig. 60 muestra el fichero cabecera del módulo en el que se implementará el TDA Conjunto de enteros. Como se puede ver, se incluyen dos ficheros cabecera. El del TDA Lista, puesto que se usa como valor devuelto por la función `enumera`, y uno llamado `Elemento.h`. Este fichero, mostrado en la Fig. 61, permite definir en un único punto del programa el Tipo de Datos `Elemento`, y es una práctica habitual en aplicaciones que incluyen varios TDA que dependen de este mismo Tipo de Datos.

```
#ifndef __Conjunto_H__
#define __Conjunto_H__

#include "Elemento.h"
#include "Lista.h "

typedef struct ConjuntoRep * Conjunto;

Conjunto crea_conjunto();
void libera_conjunto( Conjunto c );
void inserta_conjunto( Conjunto c, Elemento e );
void suprime_conjunto( Conjunto c, Elemento e );
int pertenece_conjunto( Conjunto c, Elemento e );
int cardinalidad_conjunto( Conjunto c );
Lista enumera_conjunto( Conjunto c );

#endif
```

Fig. 60. Fichero cabecera de un módulo que implementa el TDA Conjunto con estructuras enlazadas

Además, para evitar la existencia de identificadores duplicados se añade a cada función un sufijo que indica a qué TDA pertenece. Esto se debe hacer también en el módulo que implementa el TDA Lista. Finalmente, y como es habitual al implementar un TDA en C, se incluyen funciones de creación y liberación.

```
#ifndef __Elemento_H__
#define __Elemento_H__

typedef int Elemento;

#endif
```

Fig. 61. Fichero cabecera con la definición del Tipo de Datos Elemento

En el fichero `.c` del módulo que implementa el TDA Conjunto de enteros, mostrado en la Fig. 62, se define la estructura `struct ConjuntoRep` que contiene todo lo necesario para representar al Tipo de Datos. En este caso, las variables `cabecera` y `n`. En el mismo módulo, y oculto para los clientes del TDA, se ha definido la estructura `struct Nodo` que permite crear y manejar la estructura enlazada con la que se representará el Conjunto, así como el alias `NodoPtr` para facilitar la escritura del código.

```c
#include "Conjunto.h"
#include <stdlib.h>

struct Nodo {
  Elemento dato;
  struct Nodo * sig;
};

typedef struct Nodo * NodoPtr;

struct ConjuntoRep {
  NodoPtr cabecera;
  int n;
};

Conjunto crea_conjunto( ) {
  Conjunto nuevo = malloc( sizeof( struct ConjuntoRep ) );
  nuevo->cabecera = malloc( sizeof( struct Nodo ) );
  nuevo->cabecera->sig = NULL;
  nuevo->n = 0;
  return nuevo;
}

void libera_conjunto( Conjunto c ) {
  while ( c->cabecera != NULL ) {
    NodoPtr borrar = c->cabecera;
    c->cabecera = borrar->sig;
    free(borrar);
  }
  free( c );
}

NodoPtr busca( Conjunto c, Elemento e ) {
  NodoPtr aux = c->cabecera;
  while ( aux->sig != NULL && aux->sig->dato < e ) {
    aux = aux->sig;
  }
  return aux;
}

void inserta_conjunto( Conjunto c, Elemento e ) {
  NodoPtr aux = busca( c, e );
  if ( aux->sig == NULL || aux->sig->dato != e) {
    NodoPtr nuevo = malloc( sizeof( struct Nodo ) );
    nuevo->dato = e;
    nuevo->sig = aux->sig;
    aux->sig = nuevo;
    c->n++;
  }
}

void suprime_conjunto( Conjunto c, Elemento e ) {
  NodoPtr aux = busca( c, e );
  if ( aux->sig != NULL && aux->sig->dato == e ) {
    NodoPtr borrar = aux->sig;
    aux->sig = borrar->sig;
    free( borrar );
    c->n--;
  }
}

int pertenece_conjunto( Conjunto c, Elemento e ) {
  NodoPtr aux = busca( c, e );
  return aux->sig != NULL && aux->sig->dato == e;
}

int cardinalidad_conjunto( Conjunto c ) {
  return c->n;
}

Lista enumera_conjunto( Conjunto c ) {
  Lista lista = crea_lista();
  NodoPtr aux = c->cabecera->sig;
  while ( aux != NULL ) {
    inserta_lista( lista, final_lista( lista ), aux->dato );
    aux = aux->sig;
  }
  return lista;
}
```

Fig. 62. Fichero .c de un módulo que implementa el TDA Conjunto con estructuras enlazadas lineales

La función `crea`, además de reservar memoria para la estructura `struct ConjuntoRep`, también crea e inicializa un nuevo nodo en memoria dinámica que será la cabecera de la estructura enlazada. A continuación, inicializa la variable `cabecera`, apuntando a dicho nodo, y `n` a 0.

La función `libera` se ocupa de forma iterativa de liberar la memoria asociada a todos los nodos de la estructura enlazada, y finalmente, de liberar la memoria asociada a la estructura `struct ConjuntoRep` apuntada por el parámetro `c`.

A continuación, aparece una función privada al módulo. Se denomina `busca` y se incluye para evitar duplicar código en las funciones `inserta`, `suprime` y `pertenece`. Lo que hace es buscar el elemento pasado como parámetro y devolver la dirección del nodo anterior al que lo contenga. Si no se encuentra el elemento buscado devuelve la dirección del último nodo de la estructura enlazada, o la de la cabecera cuando el Conjunto está vacío. Por lo tanto, nunca devuelve `NULL`.

Lo más importante de esta función es que siempre devuelve la dirección de un nodo de la lista. Por lo tanto, las funciones `inserta_conjunto`, `suprime_conjunto` y `pertenece_conjunto` pueden usar el campo `sig` de dicho nodo para comprobar si el elemento en cuestión existe y en la estructura enlazada.

Además, para reducir el tiempo de ejecución de la función `busca`, y por tanto el de las tres operaciones que la usan, se ha escrito el bucle asumiendo que la estructura enlazada se mantiene ordenada de menor a mayor. De este modo, la búsqueda se puede detener al encontrar un valor mayor o igual al buscado, evitando así tener que recorrer toda la lista en los casos en que el elemento buscado no se encuentre en la misma. Evidentemente, esto sólo se puede hacer si, como en este caso, es posible establecer un orden entre los elementos del Conjunto.

Para mantener la lista ordenada hay que insertar los nuevos elementos justo en el lugar que les corresponde. Como la función `busca`, en los casos en que el elemento buscado no está en la lista, devuelve la dirección del nodo anterior al que contiene un elemento mayor que el que se desea insertar, la inserción a continuación de dicho nodo mantendrá la lista ordenada.

Pero la inserción sólo se llevará a cabo si el elemento no está ya incluido en el conjunto. Es decir, si la llamada a `busca` devolvió la dirección del último nodo, indicando que el elemento buscado no se encuentra en la lista, o si devolvió la dirección de un nodo anterior a otro cuyo elemento es distinto (sólo podría ser mayor) del que se desea insertar.

La función `suprime_conjunto` comprueba si a continuación de la dirección devuelta por la llamada a `busca` hay un nodo o no. Si lo hay, y además coincide con el que se quiere eliminar, entonces borra el nodo, en otro caso no lo hace.

La función `pertenece_conjunto` se comporta igual que `suprime_conjunto` pero, en lugar de eliminar el elemento encontrado, devuelve verdadero o falso en función del resultado. Y la función `cardinalidad_conjunto` se limita a devolver el valor de n que, tras ser inicializado a cero en la creación del Conjunto, ha sido actualizado por las funciones `inserta_conjunto` y `suprime_conjunto`.

Finalmente, la función enumera_conjunto crea una nueva Lista y recorre la estructura enlazada insertando cada elemento al final de la nueva Lista, para terminar devolviéndola.

Implementación con árboles binarios de búsqueda

Los árboles binarios de búsqueda son una estructura de datos ideal para representar conjuntos, especialmente cuando estos tienen un gran tamaño.

Para representar un conjunto mediante un árbol binario de búsqueda es necesario definir la estructura que representa cada uno de sus nodos.

```
struct Nodo {
  Elemento dato;
  struct Nodo * hijoIzquierdo;
  struct Nodo * hijoDerecho;
};

typedef struct Nodo * ABB;

struct ConjuntoRep {
  ABB raiz;
};
```

Fig. 63. Estructura de datos usada para representar un TDA Conjunto con un árbol binario de búsqueda

La Fig. 63 muestra todo lo necesario para representar un conjunto mediante un árbol binario de búsqueda. Incluye la estructura `struct Nodo`, el alias `ABB` y la estructura `struct ConjuntoRep` que contiene únicamente la variable `raiz`, que apuntará a la raíz del árbol binario de búsqueda.

Las operaciones del interfaz público del TDA Conjunto se pueden implementar casi directamente con las operaciones típicas definidas sobre un árbol binario de búsqueda. Así pues, la operación Pertenece, Inserta y Suprime se resuelven con sus equivalentes aplicadas sobre el árbol representado por la variable `raiz`.

```
Pertenece( x ) {
  return existe( raiz, x );
}
```

```
Inserta ( x ) {
  raiz = inserta( raiz, x );
}
```

```
Suprime( x ) {
  raiz = suprime( raiz, x );
}
```

Fig. 64. Operación Pertenece del TDA Conjunto implementado con estructuras enlazadas lineales

La operación Enumera se resuelve haciendo un recorrido en inorden del árbol binario de búsqueda y creando de forma recursiva la lista con todos los elementos del árbol que, además, estarán ordenados de menor a mayor gracias a la "Propiedad de los árboles binarios de búsqueda". La Fig. 65 muestra cómo realizar esta operación.

La idea es la siguiente: la enumeración de un árbol vacío es una lista vacía. Y la enumeración de un árbol no vacío es la concatenación de las enumeraciones de sus hijos, intercalando entre ambas el elemento de la raíz del árbol. Como puede verse, tras comprobar y resolver el caso base devolviendo una nueva lista vacía, se obtienen las enumeraciones de los subárboles. A la del hijo izquierdo se le añade por el final, primero el elemento de la raíz y, después, la lista obtenida del hijo derecho. Antes de devolver la lista que contiene todo se libera la del hijo derecho.

```
Lista enumera_abb( ABB a ) {
  if ( a == NULL ) return crea_lista();

  Lista li = enumera_abb( a->hijoIzquierdo );
  inserta_lista( li, final_lista ( li ), a->dato );

  Lista ld = enumera_abb( a->hijoDerecho );

  Posicion p = inicio_lista( ld );
  while ( p != final_lista( ld ) ) {
    Elemento dato = recupera_lista( ld, p );
    inserta_lista( li, final_lista( li ), dato );
    p = siguiente_posicion( ld, p );
  }
  libera_lista( ld );
  return li;
}
```

Fig. 65. Función para crear una lista ordenada con los elementos de un árbol binario de búsqueda

La operación Cardinalidad se puede resolver creando una función que cuente todos los elementos del árbol. Y también modificando su estructura, y las operaciones de inserción y supresión, para que cada nodo mantenga actualizado en todo momento el número de nodos del árbol del cual es raíz. La primera opción implica que la operación Cardinalidad tenga un tiempo de ejecución de O(n), pero es más simple de implementar. La segunda, es más compleja, pero consigue que el tiempo de ejecución se reduzca pasando a ser de O(1).

Finalmente, para implementar el TDA Conjunto de enteros usando el árbol binario es necesario escribir un módulo. En dicho módulo se incluiría, de forma privada para el usuario del TDA, todo el código necesario para definir la estructura que representa un árbol binario de búsqueda y las funciones para usarlo. También se incluye, como habitualmente, la definición de la estructura `struct ConjuntoRep` que es la que representa el Tipo de Datos Conjunto, y el código de las funciones públicas de su interfaz.

La Fig. 66 muestra cómo quedaría el fichero .c de dicho módulo. Como puede verse, las funciones del interfaz del TDA Conjunto se limitan a usar las del árbol binario de búsqueda.

Comparación entre implementaciones

La inserción, supresión y búsqueda en árboles binarios de búsqueda llegan a tener un O(log n) cuando los árboles están equilibrados. Y la operación Cardinalidad puede tener un tiempo de ejecución constante si el árbol se implementa para ello. En el peor de los casos, cuando el árbol no está equilibrado, el comportamiento sería equivalente al obtenido con la implementación basada en estructuras enlazadas lineales. Es decir, todas las operaciones tendrían un tiempo de ejecución lineal excepto Cardinalidad.

```
#include "Conjunto.h"
#include <stdio.h>
#include <stdlib.h>

/**
  * Aquí van las estructuras y funciones necesarias para trabajar con
  * un árbol binario de búsqueda.
**/

struct ConjuntoRep {
  ABB raiz;
};

Conjunto crea_conjunto( ) {
  Conjunto nuevo = malloc( sizeof( struct ConjuntoRep ) );
  nuevo->raiz = NULL;
  return nuevo;
}

void libera_conjunto( Conjunto c ) {
  libera_abb( c->raiz );
  free( c );
}

void inserta_conjunto( Conjunto c, Elemento e ) {
  c->raiz = inserta_abb( c->raiz, e );
}

void suprime_conjunto( Conjunto c, Elemento e ) {
  c->raiz = suprime_abb( c->raiz, e );
}

int pertenece_conjunto( Conjunto c, Elemento e ) {
  return existe_abb( c->raiz, e );
}

int cardinalidad_conjunto( Conjunto c ) {
  return cardinalidad_abb( c->raiz);
}

Lista enumera_conjunto( Conjunto c ) {
  return enumera_abb( c->raiz);
}
```

Fig. 66. Fichero .c de un módulo que implementa el TDA Conjunto con árboles binarios de búsqueda

Ejemplos de uso

Como sólo es posible acceder al elemento situado en el tope de una Pila, si se desea mostrar todo su contenido hay que extraerlo y volver a colocarlo donde estaba. Para eso se puede usar una segunda Pila en donde se irán almacenando los elementos extraídos de la primera. Una vez terminado este proceso se sacan de la Pila auxiliar y se vuelven a insertar en la original. Si los elementos se muestran en la primera iteración aparecerán en un orden y si se muestran en la segunda en el contrario. En la Fig. 67 se muestra el esquema de esta función.

```
void muestra( Pila p ) {
  Pila aux = crea();
  while ( !vacia( p ) ) {
    Elemento e = recupera( p );
    suprime( p );
    // Mostrar e aquí para conseguir orden inverso
    inserta( aux, e );
  }
  while ( !vacia( aux ) ) {
    Elemento e = recupera( aux );
    suprime( aux );
    // Mostrar e aquí para conseguir orden natural
    inserta( p, e );
  }
  libera( aux );
}
```

Fig. 67. Función que muestra el contenido de una Pila con ayuda de otra

Para encontrar el equivalente en binario de un número decimal basta con ir dividiendo por dos el número original y comprobando si cada división es exacta o no. Si es exacta se anota un cero y si no un uno. Al terminar se leen los números anotados en orden inverso y ese es el número binario equivalente.

Para programar este algoritmo se puede usar una Pila en la que se vayan introduciendo los restos de las sucesivas divisiones. Al terminar se extraen y muestran todos los elementos de la pila obteniendo el resultado deseado. La Fig. 68 muestra un programa completo que hace todo esto.

```
#include <stdio.h>
#include "Pila.h"

int main( int argc, char *argv[] ) {
  int numero;
  scanf( "%d", &numero );
  Pila p = crea();
  while ( numero > 0 ) {
    int resto = numero % 2;
    inserta( p, resto );
    numero = numero/2;
  }
  while ( !vacia( p ) ) {
    int bit = recupera( p );
    suprime( p );
    printf( "%d", bit );
  }
  printf("\n");
  libera( p );
}
```

Fig. 68. Programa que usa un TDA Pila de enteros para pasar un número de decimal a binario

Un ejemplo más avanzado de uso del TDA Pila es el mostrado en la Fig. 69. En él se ha programado una función que determina si los paréntesis y corchetes de una expresión aritmética están correctamente equilibrados y no hay ninguno huérfano.

Para hacerlo se usa una Pila de caracteres (modificando la definición de `Elemento` para que equivalga a `char`) en la que se insertan los símbolos de abrir paréntesis o corchetes. Cada vez que se encuentra uno de cierre se comprueba si en el tope de la Pila está el opuesto. Si la Pila está vacía o si lo que hay en su tope no coincide con lo esperado se termina inmediatamente devolviendo 0 (falso). Si se llega al final de la expresión, pero quedan elementos en la Pila significa que se han producido aperturas, pero no sus cierres correspondientes, es decir, hay aperturas huérfanas. Por eso, al final de la función se devolverá 1 (verdadero) sólo si la Pila está vacía.

El programa principal comprueba si la expresión aritmética pasada al programa como primer argumento está equilibrada. Para ello, primero comprueba si al ejecutarlo se ha pasado, al menos, un argumento y si no muestra cómo ha de usarse la aplicación.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include "Pila.h"

int equilibrada( char * expresion ) {
  Pila operadores = crea();
  int i = 0;
  while ( expresion[i] != '\0' ) {
    if ( expresion[i] == '(' || expresion[i] == '[' ) {
      inserta( operadores, expresion[i] );
    }
    if ( expresion[i] == ')' ) {
      if ( vacia( operadores ) || recupera( operadores ) != '(' ) return 0;
      suprime( operadores );
    }
    if ( expresion[i] == ']' ) {
      if ( vacia( operadores ) || recupera( operadores ) != '[' ) return 0;
      suprime( operadores );
    }
    expresion++;
  }
  int resultado = vacia( operadores );
  libera( operadores );
  return resultado;
}

int main( int argc, char * argv[] ) {
  if ( argc < 2 ) {
    printf( "Uso: %s <expresión aritmética>\n", argv[0] );
    return 1;
  }
  printf( "La expresión %s ", argv[1]);
  if ( equilibrada( argv[1] ) == 1 ) {
    printf( "está equilibrada\n" );
  } else {
    printf( "no está equilibrada\n" );
  }
}
```

Fig. 69. Función que usa un TDA Pila de caracteres para determinar si una expresión tiene bien balanceados sus paréntesis y corchetes

El TDA Conjunto que se ha estudiado no incluye en su interfaz público operaciones para hacer la unión o la intersección de conjuntos. Sin embargo, dichas operaciones se pueden construir usando las operaciones que sí incluye el interfaz. Por ejemplo, la unión de dos conjuntos se logra enumerando los elementos de ambos e insertándolos en un tercer conjunto creado a tal efecto.

```
Conjunto union( Conjunto a, Conjunto b ) {
  Conjunto u = crea_conjunto();
  Lista datos = enumera_conjunto( a );
  Posicion p = inicio_lista( datos );
  while ( p != final_lista( datos ) ) {
    int dato = recupera_lista( datos, p );
    inserta_conjunto( u, dato );
    p = siguiente_posicion( datos, p );
  }
  libera_lista( datos );
  datos = enumera_conjunto( b );
  p = inicio_lista( datos );
  while ( p != final_lista( datos ) ) {
    int dato = recupera_lista( datos, p );
    inserta_conjunto( u, dato );
    p = siguiente_posicion( datos, p );
  }
  libera_lista( datos );
  return u;
}
```

Fig. 70. Función que devuelve un conjunto resultado de unir otros dos

Del código mostrado en la Fig. 70 hay que destacar la importancia de liberar la memoria de los TDA que dejan de usarse. En concreto, la Lista `datos` se libera dos veces. Primero, justo después de haber sido usada para insertar los elementos del conjunto `a` en el conjunto `u`. La segunda liberación corresponde a la lista que devolvió la función `enumera_conjunto` aplicada al conjunto `b`.

La intersección se puede construir de una forma muy parecida. En este caso hay que añadir al nuevo conjunto sólo los elementos que están en ambos conjuntos. Así pues, como muestra la Fig. 71, se enumera uno de los conjuntos y se recorre la lista obtenida insertando en el nuevo conjunto sólo aquellos que sí existen en el segundo conjunto.

```
Conjunto interseccion( Conjunto a, Conjunto b ) {
  Conjunto i = crea_conjunto();
  Lista datos = enumera_conjunto( a );
  Posicion p = inicio_lista( datos );
  while ( p != final_lista( datos ) ) {
    int dato = recupera_lista( datos, p );
    if ( pertenece_conjunto( b, dato ) ) {
      inserta_conjunto( i, dato );
    }
    p = siguiente_posicion( datos, p );
  }
  libera_lista( datos );
  return i;
}
```

Fig. 71. Función que devuelve un conjunto resultado de hacer la intersección de otros dos

En principio, todos los TDAs Contenedores que se han visto se pueden adaptar para contener elementos de cualquier Tipo de Datos, incluidos otros TDAs. Sólo hay que modificar la definición del alias `Elemento` para conseguirlo. Ahora bien, al usarlos hay que ser cuidadoso pues ninguna operación de las incluidas en los TDA contenedores se encarga de liberar la memoria asociada a los elementos contenidos. Esto es responsabilidad del usuario del TDA contenedor que es quien se encarga de crear los elementos que se añaden al contenedor.

Por ejemplo, suponiendo que se dispusiera del TDA Punto y que este representara un punto en un plano mediante un par de coordenadas enteras, el siguiente fragmento de código muestra cómo eliminar una lista completa.

```
Posicion p = inicio_lista( puntos );
while ( p != final_lista( puntos ) ) {
  Punto punto = recupera_lista( puntos, p );
  libera_punto( punto );
  p = siguiente_posicion( puntos, p );
}
libera_lista( puntos );
```

Fig. 72. Función que devuelve un conjunto resultado de unir otros dos

Como puede verse, primero hay que recorrer la lista completa eliminando cada elemento, y después, eliminar la lista usando la función correspondiente del TDA.

Otra operación frecuente con listas es la de filtrado. Es decir, dejar en una lista aquellos elementos que cumplan cierto criterio eliminando el resto. De nuevo, hay que ser cuidadoso y no olvidar liberar la memoria de los propios elementos eliminados. Pero, además, hay que tener en cuenta que las operaciones Inserta y Suprime del TDA Lista tiene como efecto el dejar invalidada la posición usada. Por lo tanto, no es posible suprimir en mitad de un bucle y, tras la supresión, continuar recorriendo la lista usando la misma variable de tipo `Posicion` usada hasta ese momento.

```
...
Lista nueva = crea_lista();
Posicion p = inicio_lista( puntos );
while ( p != final_lista( puntos ) ) {
  Punto punto = recupera_lista( puntos, p );
  if ( criterio_permanencia( punto ) ) {
    inserta_lista( nueva, final_lista( nueva ), punto );
  } else {
    libera_punto( punto );
  }
  p = siguiente_posicion( puntos, p );
}
libera_lista( puntos );
puntos = nueva;
...
```

Fig. 73. Función que devuelve un conjunto resultado de unir otros dos

Para obtener una lista filtrada hay que crear una nueva lista y añadir en ella los elementos de la original que cumplan el criterio de permanencia. Los que no lo cumplan se van eliminando y, finalmente, tras eliminar la memoria asociada a la lista original, la variable `lista` toma el valor de la nueva lista generada.

A veces interesa recorrer un árbol por niveles y no en profundiad como con los algoritmos de recorrido preorden, inorden y postorden. Para poder hacerlo es necesario explorar todos los hijos de un nodo antes de empezar con sus nietos. Este problema se puede resolver usando una Cola de árboles.

Inicialmente se inserta la raíz del árbol original en la Cola. El algoritmo funciona extrayendo un árbol de la cola, visitando su raíz y añadiendo a la cola a sus hijos.

```
void anchura( ArbolBinario a ) {
  Cola c = crea_cola();
  cola_inserta( c, a );
  while ( !vacia_cola( c ) ) {
    ArbolBinario a = recupera_cola( c );
    suprime_cola( c );
    printf( "%d\n", a->dato ) ;
    if ( a->hijoIzquierdo != NULL ) inserta_cola( c, a->hijoIzquierdo );
    if ( a->hijoDerecho != NULL ) inserta_cola( c, a->hijoDerecho );
  }
  libera_cola( c );
}
```

Fig. 74. Función que recorren un árbol binario en anchura

La función mostrada en la Fig. 74 trabaja con un árbol binario. Para hacer lo mismo con un árbol general basta con sustituir las dos condiciones que se encargan de añadir los subárboles izquierdo y derecho, por un bucle en el que se procesan todos los hijos añadiéndolos a la cola.

Uso de memoria contigua dinámica en la implementación de Contenedores

La implementación con memoria contigua de los Tipos de Datos Pila, Cola y Lista que se ha visto se basa en el uso de arrays cuyo tamaño máximo se especifica en el código fuente. Se trata pues de arrays estáticos y su uso hace que estos TDAs contenedores tengan limitado el número máximo de elementos que pueden almacenar y, además, que este tamaño no pueda decidirlo el cliente del TDA.

Una mejora consiste en permitir que el tamaño máximo se determine en el momento de la creación del contenedor. De este modo, seguirían siendo Contenedores Acotados, pero al menos, su tamaño máximo podrá ser elegido por el usuario del TDA, dando como resultado un TDA contenedor más flexible y útil para cualquier aplicación.

La solución pasa por usar memoria dinámica, es decir, declarar el array `datos` como un puntero al Tipo de Datos de los elementos que vaya a guardar el TDA Contenedor (`Elemento * datos`) en lugar de como un array de tamaño fijo.

La función `crea` reservará memoria dinámica para guardar en ella el array `datos`. Así pues, a esta función se le debe añadir un parámetro que permita al usuario indicar el tamaño máximo del Contenedor que desee crear. Este valor se debe guardar para poder usarlo en la función `llena` y en la precondición de `inserta`. Para guardarlo se añade a la estructura que representa el Tipo de Datos un nuevo campo de tipo entero llamado `max`. La Fig. 75 muestra cómo quedaría la estructura que representaría el TDA Pila con los cambios que se acaban de comentar.

```
struct PilaRep {
  Elemento * datos;
  int tope;
  int max;
};
```

Fig. 75. Estructuras de datos usada para representar un TDA Pila con memoria contigua dinámica

La Fig. 76 muestra cómo quedaría el fichero `.c` que implementa el TDA Pila con memoria contigua dinámica. Las adaptaciones en el caso de los TDA Cola y Lista son idénticas. La función `crea` se ha modificado y ahora usa `malloc` para reservar la memoria necesaria para el array `datos`. Nótese que el número de elementos máximo indicado como parámetro se multiplica por la memoria necesaria para cada uno de ellos, y así se obtiene la memoria total necesaria para el array.

Otro cambio necesario es el de la función `libera`. Ahora, antes de liberar la memoria asociada a la estructura que representa la Pila se debe liberar la memoria asociada al array `datos`.

El resto de funciones permanece idéntico excepto que en la precondición de la función `inserta` y en el código de la función `llena` se usa el campo `max` en lugar de la constante `MAX`.

```
#include "Pila.h"
#include <stdlib.h>
#include <assert.h>

struct PilaRep {
  Elemento * datos;
  int tope;
  int max;
};

Pila crea( int max ) {
  Pila nueva = malloc( sizeof( struct PilaRep ) );
  nueva->datos = malloc( sizeof( Elemento ) * max );
  nueva->tope = -1;
  nueva->max = max;
  return nueva;
}

void libera( Pila p ) {
  free( p->datos );
  free( p );
}

void inserta( Pila p, Elemento dato ) {
  assert( p->tope < p->max-1 );
  p->tope = p->tope + 1;
  p->datos[p->tope] = dato;
}

void suprime( Pila p ) {
  assert( p->tope >= 0 );
  p->tope = p->tope - 1;
}

Elemento recupera( Pila p ) {
  assert( p->tope >= 0 );
  return p->datos[p->tope];
}

int vacia( Pila p ) {
  return p->tope == - 1;
}

int llena( Pila p ) {
  return p->tope == p->max-1;
}
```

Fig. 76. Fichero .c de un módulo que implementa el TDA Pila con memoria contigua dinámica

Uso de arrays dinámicos en la implementación de Contenedores

Gracias al concepto de array dinámico es posible implementar TDAs Contenedores no Acotados usando memoria contigua. De este modo se mantiene la ventaja del acceso directo indexado a los elementos, y el uso de memoria estrictamente necesario característico de los arrays, y al mismo tiempo, se gana la flexibilidad de no tener un tamaño máximo limitado. Obviamente, todo no serán ventajas.

Los arrays dinámicos son aquellos que pueden cambiar su tamaño durante la ejecución del programa y, normalmente, se crean en la memoria dinámica. Para usar un array dinámico se puede utilizar la misma estructura de datos usada para la implementación con memoria contigua dinámica (Fig. 75). La diferencia está en las funciones Inserta y Suprime. En ellas se determinará si es necesario y conveniente modificar el tamaño del array `datos` y, por consiguiente, actualizar el valor del campo `max`.

La Fig. 77 muestra cómo implementar el TDA Pila con arrays dinámicos. Como se puede ver, en la función `inserta` se calcula la nueva posición del `tope` y después se comprueba si éste es igual al máximo actual. Si lo es el array `datos` estará lleno y, por tanto, se procede a redimensionarlo con la función `realloc`, declarada en el fichero de cabecera `<stdlib.h>`, y a actualizar el valor del campo `max`.

Como puede verse, a la función `realloc` se le pasa el puntero que guarda la dirección de la zona de memoria que se quiere redimensionar y el nuevo tamaño en bytes. La función intenta modificar el tamaño de la zona de memoria indicada. Si no lo consigue reserva una nueva zona del tamaño apropiado y copia allí todos los datos. Finalmente devuelve la dirección de memoria de la zona donde hayan quedado los datos. Por esta razón siempre que se ejecute `realloc` se debe actualizar el valor del puntero pasado como parámetro con el valor devuelto por la función. La posible necesidad de copiar los datos hace que esta operación pueda resultar muy costosa desde el punto de vista computacional.

Teniendo esto en cuenta, cada vez que se llena el array se redimensiona doblando su tamaño máximo. Este crecimiento en progresión geométrica reduce significativamente el número de llamadas a la función `realloc`. Básicamente, si las inserciones se producen a ritmo constante cada vez que se alcanza el máximo se tarda el doble de tiempo en volver a alcanzarlo.

Del mismo modo, cuando se realizan operaciones de supresión, y la ocupación se reduce a un cuarto del tamaño máximo, se vuelve a reajustar, esta vez a la baja, la cantidad de memoria usada por el array `datos`. El espacio se reduce a la mitad, dejando aún margen para previsibles inserciones producidas a continuación, y evitando así la necesidad de volver a aumentar inmediatamente su tamaño.

Por otro lado, al no tener un tamaño máximo la operación Llena y la precondición de la Operación Inserta dejan de ser necesarias.

A pesar de la estrategia de redimensionado usada, el coste de las posibles redimensiones hace que las operaciones Inserta y Suprime no tengan un tiempo de ejecución de O(1). Normalmente, la reducción en el coste computacional suele venir acompañada de un aumento en el consumo de memoria, que es justo lo que pasa al usar una implementación basada en estructuras enlazadas. Estas consiguen reducir el tiempo de ejecución de las operaciones Inserta y Suprime a cambio de usar más memoria para guardar los datos.

```
#include "Pila.h"
#include <stdlib.h>
#include <assert.h>

struct PilaRep {
  Elemento * datos;
  int max;
  int tope;
};

Pila crea( int max ) {
  Pila nueva = malloc( sizeof( struct PilaRep ) );
  nueva->datos = malloc( sizeof( Elemento ) * max );
  nueva->tope = -1;
  nueva->max = max;
  return nueva;
}

void libera( Pila p ) {
  free( p->datos );
  free( p );
}

void inserta( Pila p, Elemento dato ) {
  p->tope = p->tope + 1;
  if ( p->tope == p->max ) {
    p->max = p->max * 2;
    p->datos = realloc( p->datos, p->max * sizeof( Elemento ) );
  }
  p->datos[p->tope] = dato;
}

void suprime( Pila p ) {
  assert( p->tope >= 0 );
  p->tope = p->tope - 1;
  if ( p->tope > 0 && p-> tope == p-> max / 4 ) {
    p->max = p->max / 2;
    p->datos = realloc( p->datos, p->max * sizeof( Elemento ) );
  }
}

Elemento Recupera( Pila p ) {
  assert( p->tope >= 0 );
  return p->datos[p->tope];
}

int vacia( Pila p ) {
  return p->tope == - 1;
}
```

Fig. 77. Fichero .c de un módulo que implementa el TDA Pila con arrays dinámicos

Las modificaciones necesarias para usar arrays dinámicos en Colas y Listas son muy similares a la mostrada en la figura anterior para el TDA Pila. Pero en el caso de las Colas, al utilizarse el array `datos` de forma circular, en lugar de usar `realloc` es más conveniente hacer una nueva reserva de memoria con `malloc` y copiar en la nueva zona de memoria los datos actuales compactándolos para que ocupen las primeras posiciones del nuevo array. Después se libera la memoria que usaba el array `datos` anterior, se actualiza la variable `datos` para que apunte a la nueva zona de memoria y también se modifican las variables `inicio` y `final` dándoles los valores 0 y `n` respectivamente, siendo n el número de elementos tras la inserción.