Ingeniería Técnica en
Informática de Gestión

Estructuras de Datos I

José Antonio Alonso de la Huerta
José Fidel Argudo Argudo
Mª Teresa García Horcajadas

Departamento de Lenguajes y Sistemas Informáticos

Estructuras de Datos I

José Antonio Alonso de la Huerta
joseantonio.alonso@uca.es

José Fidel Argudo Argudo
josefidel.argudo@uca.es

Mª Teresa García Horcajadas
mayte.garcia@uca.es

Edita: Departamento de Lenguajes y Sistemas Informáticos.
Universidad de Cádiz

ISBN: 84-89867-41-0

Primera edición – Marzo 2003

CONTENIDO

ÍNDICE DE FIGURAS

1

TIPOS ABSTRACTOS DE DATOS

1.1. Conceptos, terminología y ejemplos

La **abstracción** es un mecanismo de la mente humana fundamental para la comprensión de fenómenos o situaciones que involucren una gran cantidad de detalles.

Como define Wulft: “Los humanos hemos desarrollado una técnica excepcionalmente potente para tratar la complejidad, abstraernos de ella. Incapaces de dominar en su totalidad los objetos complejos, se ignoran los detalles no esenciales, tratando en su lugar con el modelo ideal del objeto y centrándonos en el estudio de sus aspectos esenciales".

La abstracción es un proceso mental que consta de dos aspectos complementarios:

- Destacar los detalles relevantes del objeto en estudio.
- Ignorar los detalles irrelevantes del objeto (en ese nivel de abstracción).

La abstracción permite estudiar fenómenos complejos siguiendo un método jerárquico, por sucesivos niveles de detalle.

Los problemas que suelen resolverse mediante programas de orde-

nador son complejos, luego es un campo ideal en el que aplicar la abstracción.

El uso de la abstracción nos permite generar un modelo abstracto del problema.

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de programas pueden implementar dichos modelos abstractos.

La abstracción es, por tanto, clave para diseñar software de calidad, es decir, nos permite diseñar programas más cortos, legibles, fáciles de mantener y fiables.

La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías:

- La *abstracción operacional*: se basa en la utilización de procedimientos o funciones sin preocuparse de cómo se implementan. Se destaca qué hace y se ignora cómo lo hace.
- La *abstracción de datos*: es la técnica de programación que permite inventar o definir nuevos tipos de datos adecuados a la aplicación que se desea realizar.

Haciendo uso de la abstracción operacional, el programador, usuario del procedimiento, sólo necesita conocer la especificación del mismo, pudiendo ignorar los detalles de implementación. La abstracción provoca, por tanto, un **ocultamiento de información**.

La abstracción de datos nos lleva a la construcción de **Tipos Abstractos de Datos** (TAD), que es otro mecanismo de ocultación de información, ya que existe un TAD si:

- Existe una construcción del lenguaje que le permite asociar la representación de los datos con las operaciones que los manipulan.
- La representación del nuevo tipo de dato está oculta a las unidades del programa que lo utilizan.

1.2. Tipos Abstractos de Datos

Un tipo abstracto es una colección de valores y de operaciones que se definen mediante una especificación independiente de cualquier representación.

El calificativo abstracto expresa precisamente esta cualidad de **independencia de la representación**.

Para definir un nuevo tipo, el programador deberá decidir en primer lugar qué operaciones le parecen relevantes y útiles para operar con las valores pertenecientes al mismo, es decir, establecer la interfaz.

Una vez establecida la interfaz, el programador es libre para escoger la representación que más se adecúe al problema. Basándose en esto el programador puede definir diferentes estructuras de datos para un mismo tipo abstracto de datos, pero la utilización de este TAD se basa en su especificación, es decir en sus operaciones y propiedades funcionales, y es independiente de la estructura de datos que lo soporta y de la implementación elegida para las operaciones (figura **??**).

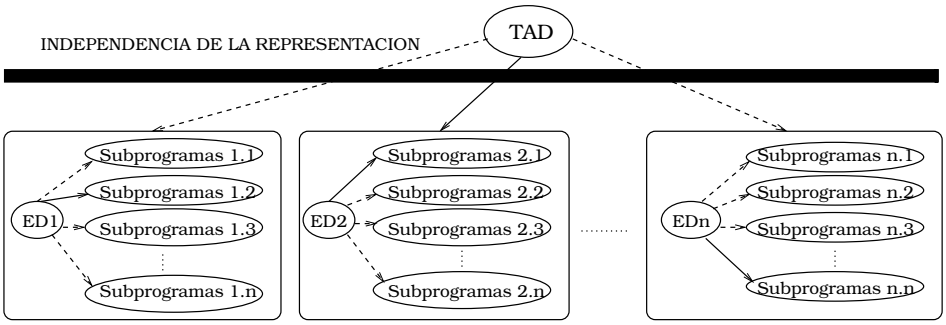


Figura 1.1: Diferentes representaciones para un TAD

El tratamiento dado por el lenguaje en los tipos definidos por el programador sería equivalente al que da a sus propios tipos: privacidad de la representación y protección (sólo se pueden utilizar las operaciones previstas en la especificación).

Consideraciones sobre el concepto de TAD:

- Un tipo no es simplemente una colección de valores (dominio), tan importantes o más que éstos son las operaciones que podemos realizar con los mismos.
- La colección de operaciones ha de permitir generar cualquier valor del tipo, mediante un conjunto de operaciones generadoras.
- El diseñador de un TAD ha de crear dos piezas de documentación: la especificación del tipo y su implementación. La especificación es la única parte que conoce el usuario del mismo y contiene el nombre del tipo y la especificación de las operaciones. La implementación es conocida sólo por el diseñador del TAD y contiene su

representación (**estructura de datos**) y la implementación de las operaciones.

- Un tipo abstracto representa una abstracción en el sentido de que, se destacan los detalles de la especificación (comportamiento observable del tipo) cuyo aspecto es bastante estable durante la vida útil del programa; y se ocultan los detalles de la implementación, cuyo aspecto es propenso a cambios.

Los TAD proporcionan numerosos beneficios al programador:

- Permiten una mejor conceptualización y modelado del mundo real. Mejora la representación y la comprensibilidad. Clarifica los objetos basados en características y comportamientos comunes.
- Mejoran la robustez del sistema.
- Mejoran el rendimiento del desarrollo y mantenimiento del software.
- Separan la implementación de la especificación. Permiten la modificación y mejora de la implementación sin afectar al interfaz público del TAD.
- Permiten la extensibilidad del sistema. Los componentes de software reutilizables son más fáciles de crear y mantener.
- Recogen mejor la semántica del tipo. Los TAD agrupan o localizan las operaciones y la representación de atributos.

Estas propiedades hacen que el tipo abstracto sea el concepto ideal alrededor del cual basar la descomposición en módulos de un programa grande.

1.3. Modularidad

El concepto de TAD proporciona una base ideal para la descomposición de un programa.

La descomposición del problema conduce a dividir el programa en componentes llamados módulos.

Un **módulo** es una unidad de organización de un programa que engloba a un conjunto de entidades (**encapsulamiento**), tales como datos y operaciones, y que controla lo que pueden ver y utilizar los usuarios

externos al módulo. De esta forma, un módulo se puede utilizar para ocultar en su interior los detalles de implementación de un TAD (ocultación de información) y ofrecer las operaciones de uso externo a través de su interfaz.

Desde el punto de vista de la ingeniería del software, un módulo ha de cumplir ciertos requisitos interesantes para una correcta división del trabajo entre los programadores y para facilitar el posterior mantenimiento del producto. Dichos requisitos son:

- Las conexiones del módulo con el resto del programa han de ser pocas y simples. De este modo se espera lograr una relativa independencia en el desarrollo de cada módulo con respecto a los otros.
- La descomposición en módulos ha de ser tal, que la mayor parte de los cambios y mejoras del programa impliquen modificar sólo un módulo, o un número muy pequeño de ellos.
- El tamaño de un módulo ha de ser el adecuado: si es demasiado grande, será difícil realizar cambios en él, si es demasiado pequeño, no es rentable.

1.4. Uso de TAD

Para definir un TAD de manera que se pueda usar necesitamos realizar una especificación del mismo y para ello hay que tener en cuenta:

- El dominio donde tomarán valores los objetos pertenecientes al TAD.
- Cómo usar los objetos, para ello necesitaremos una **especificación sintáctica**, que nos indique las reglas a seguir para realizar una operación, y una **especificación semántica** que exprese el significado de cada operación.

Para identificar y definir el dominio se puede hacer de varias formas: enumerándolo, mediante referencia a otros dominios conocidos, mediante otras definiciones que pueden ser recursivas.

La especificación sintáctica consiste en determinar la forma en que se han de escribir las operaciones, dando el orden y el tipo de operandos y resultado.

En nuestro caso la sintaxis de las operaciones vendrá descrita por las cabeceras de las funciones correspondientes a cada una de ellas.

La especificación semántica, se puede expresar de múltiples formas:

- Una de ellas es mediante el lenguaje natural aunque puede dar lugar a ambigüedades.
- Otro tipo de especificación es la algebraica que consiste en dar un conjunto de axiomas que verifican las operaciones asociadas a los objetos en cuestión. Este tipo de especificación es útil para los tipos simples o primitivos.
- Otro método es mediante el uso de modelos abstractos, donde modelizamos el dominio y las operaciones de un tipo, usando el dominio y las operaciones de algún tipo o tipos previamente definidos.

En cualquier caso, la especificación semántica la hacemos mediante el uso de **precondiciones** y **postcondiciones** del siguiente modo: bajo el cumplimiento de las precondiciones, se puede realizar la operación y después de la ejecución de la misma, se deben cumplir las postcondiciones. No se afirma nada en el caso de que no se cumplan las precondiciones, puede dar un resultado indeterminado o error. Lo ideal sería que diese un error.

Para hacer uso de los TAD en programación, el proceso a seguir ante cualquier problema sería el siguiente (figura **??**):

- Determinar los objetos candidatos a ser tipos de datos, estén o no en el lenguaje de partida.
- Identificar las operaciones básicas (primitivas) de dichos objetos.
- Especificar sintáctica y semánticamente dichas operaciones.
- Implementar, de la forma más eficiente posible; los tipos y operaciones que no estén en el lenguaje de partida, usando los elementos de dicho lenguaje.
- Construir un programa, que usando esos tipos y operaciones, resuelva el problema original.

Con esta metodología se logra disminuir la complejidad inherente a la tarea de construir programas separando dos problemas que se resuelven de forma independiente: construir un programa a partir de unos objetos adecuados a las características particulares del problema que se quiere resolver e implementar estos objetos en base a los elementos del lenguaje.

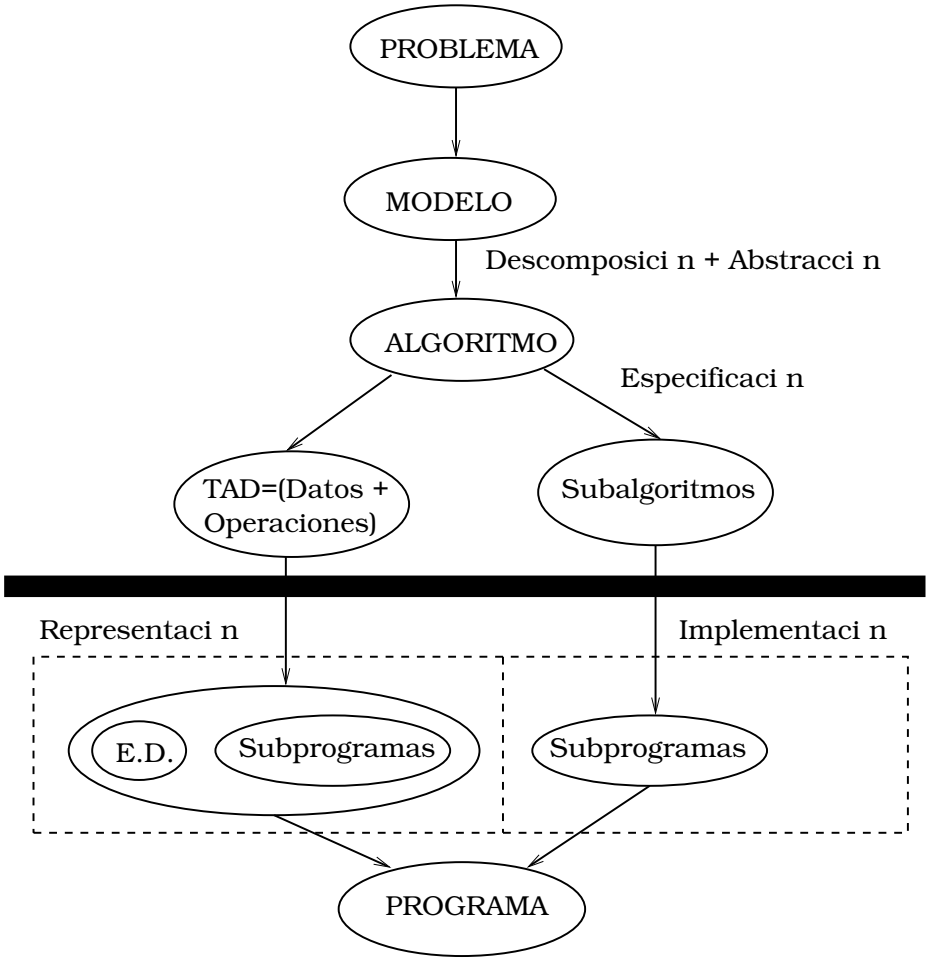


Figura 1.2: Resolución de un problema mediante el uso de TAD.

1.5. Ejemplo: TAD *número racional*

Definición:

Un elemento del TAD *número racional* es un número de la forma n/d, donde n y d son números enteros tales que $-2^{31} < n \leq 2^{31} - 1$ y $0 < d \leq 2^{31} - 1$. A n se le llama numerador y a d denominador.

Especificación de las operaciones del TAD:

racional Asignar (long n, long d)

Precondiciones: $d > 0$

Postcondiciones: Devuelve el número racional n/d.

long Numerador (racional r)

Precondiciones: r está inicializado.

Postcondiciones: Devuelve el numerador de r.

long Denominador (racional r)
Precondiciones: r está inicializado.
Postcondiciones: Devuelve el denominador de r.

int Signo (racional r)
Precondiciones: r está inicializado.
Postcondiciones: Devuelve 1 si r es positivo y 0 en caso contrario.

int Equiv (racional r, racional s)
Precondiciones: r y s están inicializados.
Postcondiciones: Devuelve 1 si r y s son equivalentes y 0 en otro caso.

racional Suma (racional r, racional s)
Precondiciones: r y s están inicializados.
Postcondiciones: Devuelve $r + s$.

racional Producto (racional r, racional s)
Precondiciones: r y s están inicializados.
Postcondiciones: Devuelve $r * s$.

racional Opuesto (racional r)
Precondiciones: r está inicializado.
Postcondiciones: Devuelve $-r$.

racional Inverso (racional r)
Precondiciones: $r \neq 0$.
Postcondiciones: Devuelve $1/r$.

*void Reducir (racional *r)*
Precondiciones: r está inicializado.
Postcondiciones: Transforma r en el número racional irreducible equivalente.

Implementación de las operaciones del TAD:

```
/*-----------------------------------------------*/
/* racional.h  - Interfaz del TAD racional       */
/*-----------------------------------------------*/

#ifndef _RACIONAL_
#define _RACIONAL_
   typedef struct {
              long num,
                   den;
          } racional;
```

```
   racional Asignar (long n, long d);
   void Reducir (racional *r);
   long Numerador (racional r);
   long Denominador (racional r);
   int Signo (racional r);
   int Equiv (racional r, racional s);
   racional Suma (racional r, racional s);
   racional Producto(racional r, racional s);
   racional Opuesto (racional r);
   racional Inverso (racional r);
#endif


/*-----------------------------------------------*/
/* racional.c  - Implementación del TAD racional */
/*-----------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "racional.h"

/* Operaciones internas */

static void Error (const char *s)
{
   printf("%s\n", s);
   exit(1);
}

static long Max (long x, long y)
{
   return x < y ? y : x;
}

static long mcd (long x, long y)
{
   long a;

   a = Max(x, y);
   if (x != a)
   {
      y = x;
```

```
      x = a;
   }
   while (x % y != 0)
   {
      a = x % y;
      x = y;
      y = a;
   }
   return y;
}

static long mcm (long x, long y)
{
   long i, aux;

   i = 2;
   aux = x;
   while (aux % y != 0)
   {
      aux = x * i;
      i = i + 1;
   }
   return aux;
}

/* Operaciones públicas */

racional Asignar (long n, long d)
{
   racional r;

   if (d <= 0)
      Error("Asignar: denominador <= 0");

   r.num = n;
   r.den = d;
   return r;
}

void Reducir (racional *r)
{
   long m;

   if (r->num != 0)
```

```
   {
      m = mcd (labs(r->num), r->den);
      r->num = r->num / m;
      r->den = r->den / m;
   }
}

long Numerador (racional r)
{
   return r.num;
}

long Denominador (racional r)
{
   return r.den;
}

int Signo (racional r)
{
   return r.num > 0;
}

int Equiv (racional r, racional s)
{
   return (r.num % s.num == 0) &&
          (r.den % s.den == 0) &&
          (r.num / s.num == r.den / s.den) ||
          (s.num % r.num == 0) &&
          (s.den % r.den == 0) &&
          (s.num / r.num == s.den / r.den);
}

racional Suma (racional r, racional s)
{
   racional t;

   if (r.num != 0 && s.num != 0)
   {
      Reducir(&r);
      Reducir(&s);
      t.den = mcm(r.den, s.den);
      t.num = r.num * t.den / r.den +
              s.num * t.den / s.den;
      Reducir(&t);
```

```
   }
   else
   {
      if (r.num != 0)
         t.den = r.den;
      else
         t.den = s.den;
      t.num = r.num + s.num;
   }
   return t;
}

racional Producto (racional r, racional s)
{
   racional t;
   long a, b;

   if (r.num != 0 && s.num != 0)
   {
      Reducir(&r);
      Reducir(&s);
      a = mcd(labs(r.num), s.den);
      b = mcd(r.den, labs(s.num));
      t.num = r.num / a * s.num / b;
      t.den = r.den / b * s.den / a;
   }
   else
   {
      t.num = 0;
      t.den = 1;
   }
   return t;
}

racional Opuesto (racional r)
{
   racional t;

   t.num = -1 * r.num;
   t.den = r.den;
   return t;
}

racional Inverso (racional r)
```

```
{
   racional t;

   if (r.num == 0)
      Error("Inverso: numerador = 0");

   if (r.num > 0)
   {
      t.num = r.den;
      t.den = r.num;
   }
   else
   {
      t.num = -1 * r.den;
      t.den = -1 * r.num;
   }
   return t;
}
```

1.6. Ejemplo: Uso del TAD *número racional*

```
/*-----------------------------------------------*/
/* sistecu.c  - Resolución de un sistema de 2    */
/*              ecuaciones con 2 incógnitas y    */
/*              coeficientes racionales.         */
/*-----------------------------------------------*/

#include <stdio.h>
#include "racional.h"

void LeerEcu (racional *a, racional *b, racional *c);
int SistEcu (racional a1, racional b1, racional c1,
             racional a2, racional b2, racional c2,
             racional *x, racional *y);
racional Det (racional a11, racional a12,
              racional a21, racional a22);

void main()
{
   racional a1, b1, c1, a2, b2, c2,
            x, y;
   int s;
```

```
   /* Entrada de datos */
   printf("1ª ecuación (a1*x + b1*y = c1):\n\n");
   LeerEcu(&a1, &b1, &c1);
   printf("\n");
   printf("2ª ecuación (a2*x + b2*y = c2):\n\n");
   LeerEcu(&a2, &b2, &c2);
   printf("\n");

   /* Cálculo de las soluciones */
   s = SistEcu(a1, b1, c1, a2, b2, c2, &x, &y);

   /* Salida de datos */
   if (s == 0)
   {
      printf("Solución:\n\n");
      printf("x = %ld / %ld\n",
             Numerador(x), Denominador(x));
      printf("y = %ld / %ld\n",
             Numerador(y), Denominador(y));
   }
   else if (s == -1)
      printf("Sistema incompatible");
   else /* s == 1 */
      printf("Infinitas soluciones");
}

void LeerEcu (racional *a, racional *b, racional *c)
{
   long num, den;

   printf("Coeficiente de x:\n");
   printf("  Numerador = "); scanf("%ld", &num);
   printf("  Denominador = ");scanf("%ld",&den);
   *a = Asignar(num, den);
   printf("Coeficiente de y:\n");
   printf("  Numerador = "); scanf("%ld", &num);
   printf("  Denominador = ");scanf("%ld",&den);
   *b = Asignar(num, den);
   printf("Término independiente:\n");
   printf("  Numerador = "); scanf("%ld", &num);
   printf("  Denominador = ");scanf("%ld",&den);
   *c = Asignar(num, den);
}
```

```
int SistEcu (racional a1, racional b1, racional c1,
             racional a2, racional b2, racional c2,
             racional *x, racional *y)
{
   racional A, B, C;

   A = Det(a1, b1, a2, b2);
   B = Det(c1, b1, c2, b2);
   C = Det(a1, c1, a2, c2);
   if (Numerador(A) != 0)
   {
      A = Inverso(A);
      *x = Producto(B, A);
      *y = Producto(C, A);
      return 0;
   }
   else if (Numerador(B) != 0 && Numerador(C) != 0)
      return -1; /* Sistema incompatible */
   else
      return 1; /* Infinitas soluciones */
}

racional Det (racional a11, racional a12,
              racional a21, racional a22)
{
   return Suma(Producto(a11, a22),
               Opuesto(Producto(a21, a12)));
}
```

2

TIPO ABSTRACTO DE DATOS PILA

2.1. Concepto de pila

Una **pila** es una secuencia de elementos en la que todas las operaciones se realizan por un extremo de la misma. Dicho extremo recibe el nombre de tope, cima, cabeza ...

En una pila el último elemento añadido es el primero en salir de ella, por lo que también se les conoce como estructuras LIFO (*Last Input First Output*).

El modelo intuitivo de una pila es un conjunto de objetos apilados, de forma que al añadir uno se coloca encima del último añadido, y para quitar un objeto del montón hay que quitar previamente los que están por encima de él (figura **??**).

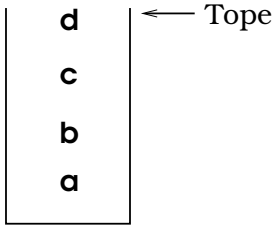


Figura 2.1: Modelo de una pila.

2.2. Especificación del TAD PILA

Definición:

Una pila es una secuencia de elementos de un tipo determinado, en la cual se pueden añadir y eliminar elementos sólo por uno de sus extremos llamado tope o cima.

Operaciones:

Pila CrearPila ()
Postcondiciones: Crea y devuelve una pila vacía.

int PilaVacia (Pila P)
Precondiciones: P está creada.
Postcondiciones: Indica si P está vacía.

tElemento PilaTope (Pila P)
Precondiciones: P no está vacía.
Postcondiciones: Devuelve el elemento del tope de la pila P.

void PilaPop (Pila P)
Precondiciones: P no está vacía.
Postcondiciones: Elimina el elemento del tope de la pila P y el siguiente se convierte en el nuevo tope.

void PilaPush (tElemento x, Pila P)
Precondiciones: P está creada y no está llena.
Postcondiciones: Inserta el elemento x en el tope de la pila P y el antiguo tope pasa a ser el siguiente.

void DestruirPila (Pila P)
Precondiciones: P está creada.
Postcondiciones: Libera la memoria ocupada por la pila P. Para usarla otra vez, hay que volver a crearla con la operación *CrearPila()*.

2.3. Implementación del TAD PILA

Existen dos formas de representar las pilas:

a) Mediante el uso de arrays unidimensionales (vectores).

b) Mediante el uso de estructuras enlazadas.

Una posible **representación vectorial** consistiría tan sólo en un vector, asumiendo que el tope de la pila se encuentra en la primera posición del mismo. Obviamente, una inserción implica el desplazamiento

hacia la derecha de los elementos contenidos en el vector, y un borrado, implicaría un desplazamiento hacia la izquierda. Esta opción de representación de la pila es bastante ineficiente debido al excesivo movimiento de datos.

Otra alternativa, que soluciona el problema anterior (excesivo movimiento de datos), consiste en representar la pila mediante un registro que contiene dos campos: uno con la posición del tope de la pila y otro, un vector que contiene los elementos de la misma (figura **??**). De esta forma, el tope de la pila se desplaza hacia la derecha cuando se añaden nuevos elementos y hacia la izquierda cuando se extraen elementos de ella.

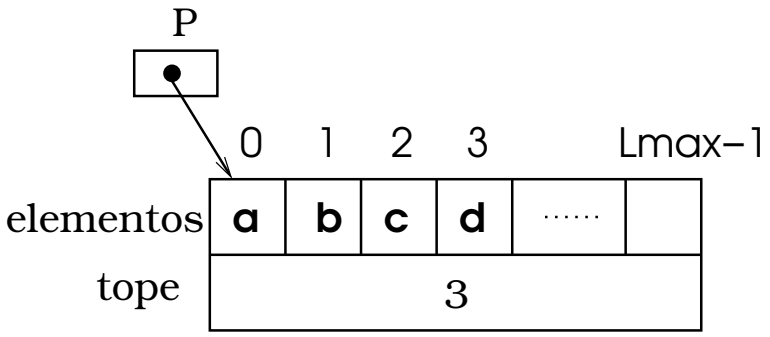


Figura 2.2: Representación vectorial estática de una pila.

Obsérvese que en la figura **??** realmente se representa la pila como un puntero `P` al registro, en lugar de hacerlo simplemente mediante el registro. Se ha añadido este puntero porque la estructura enlazada, que se explica más adelante, hay que transferirla por referencia a las operaciones del TAD y, por tanto, para preservar el principio de **independencia de la representación** esta estructura estática hay que transferirla del mismo modo. Una manera de hacer este paso de parámetros por referencia es definiendo el tipo pila como un puntero a la estructura de datos que representa a una pila.

Esta representación vectorial, al estar basada en una estructura de datos **estática**, exige fijar el tamaño máximo de la pila en tiempo de compilación, por lo que ese tamaño tendrá que ser una constante y por tanto, el mismo para todas las pilas de nuestra aplicación. Cuando sea conveniente emplear pilas de diferentes tamaños máximos en un programa, podemos optar por otra representación vectorial, a la que llamaremos **pseudoestática**, en la que el tamaño máximo de la pila no es el mismo para todas las pilas y se fijará en tiempo de ejecución en el momento de crear la misma, sin embargo es importante señalar que

este tamaño será constante durante el tiempo de vida de cada pila. En este caso, el registro que representa a la pila, además de los dos campos que almacenan el vector y la posición del tope, requiere un tercer campo para almacenar el tamaño máximo de ésta (figura **??**).

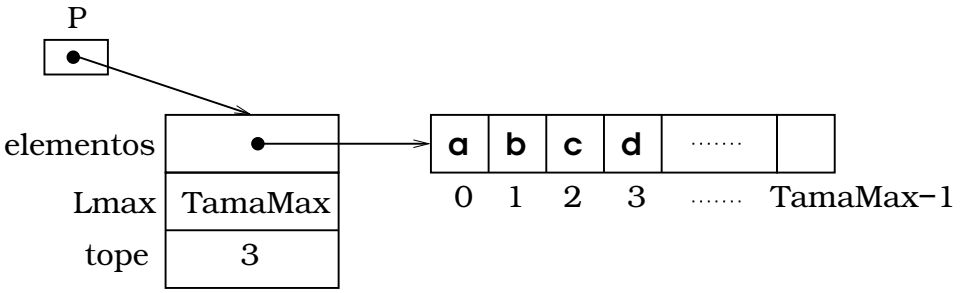


Figura 2.3: Representación vectorial pseudoestática de una pila.

La **representación mediante estructuras enlazadas** es una estructura de datos **dinámica** en la que el tamaño de la misma es variable, es decir, siempre ocupará el espacio justo para almacenar los elementos que contenga la pila en cada momento. Consistirá en una secuencia de registros enlazados con punteros, en la que cada registro almacenará un elemento, y la pila será un puntero al primer elemento de la secuencia (el tope), por donde se realizan las inserciones y borrados en esta estructura de datos (figura **??**).

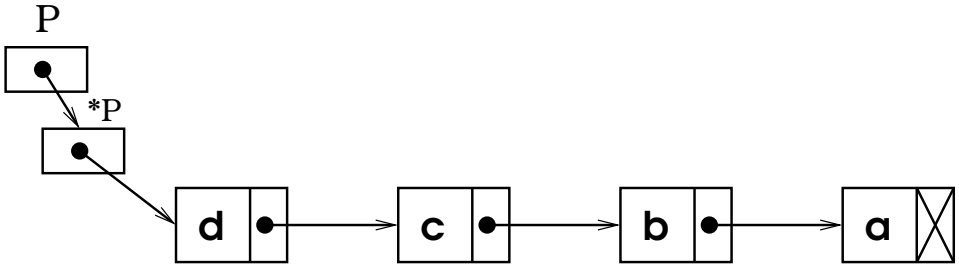


Figura 2.4: Representación dinámica de una pila.

En realidad la pila se ha representado en las figuras como un puntero `P` a las estructuras de datos diseñadas. La razón para introducir este puntero es hacer que el paso de parámetros a las operaciones del TAD sea por referencia, de modo que los cambios en los parámetros formales sean reflejados en los parámetros reales. Cuando se pasa una pila a algunas operaciones, como por ejemplo *PilaPush()* y *PilaPop()*, y se está utilizando una estructura enlazada (figura **??**) es necesario modificar el puntero al tope de la pila (`*P`) para hacer que apunte al nuevo tope, por tanto ese puntero hay que pasarlo por referencia.

Por otra parte, en las estructuras vistas anteriormente (figuras **??** y **??**) el registro que representa a la pila no se modifica al añadir o quitar elementos de ella y por eso no es necesario pasarlo por referencia. Sin embargo, en virtud del principio de **independencia de la representación**, los parámetros de tipo pila deberán ser transferidos del mismo modo, es decir por referencia, en las diferentes implementaciones que se proponen. Una forma de hacerlo es definir el tipo pila como un puntero a las estructuras de datos diseñadas. De este modo se obtienen además dos ventajas adicionales: [1]

- El paso por referencia queda oculto en la implementación del TAD y es transparente para el usuario del mismo.
- El espacio de memoria ocupado en la trasferencia de parámetros es menor cuando se utiliza una representación estática o pseudoestática. La razón es que cuando se ejecuta la llamada a una función, la transferencia de parámetros se hace siempre por valor, lo cual provoca que los parámetros reales se copien en los parámetros formales, de modo que todos estos datos están duplicados en la memoria mientras se ejecuta la función. Al definir el tipo pila como un puntero al registro que representa a la pila, sólo se hace una copia de este puntero y no del registro completo.

2.3.1. Implementación matricial de pilas

Representación estática

```
/*-----------------------------------------------*/
/* pilamat0.h                                    */
/*-----------------------------------------------*/

#define LMAX 100 /* Tamaño máximo de pila */

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento; /* Por ejemplo */
#endif
#ifndef _PILA_
```

[1] En otros lenguajes de programación diferentes de C, como Pascal o Módula, la transferencia de parámetros se puede hacer por valor o por referencia, a elección del diseñador, pero la llamada a la función se hace de la misma manera en ambos casos, con lo cual los parámetros de tipo pila se pueden pasar por referencia sin ser necesario introducir este puntero para ocultar el paso por referencia o para ahorrar memoria.

```
#define _PILA_
   typedef struct {
              tElemento elementos[LMAX];
              int tope;
            } tipoPila;
   typedef tipoPila *Pila;

   Pila CrearPila ();
   int PilaVacia (Pila P);
   tElemento PilaTope (Pila P);
   void PilaPop (Pila P);
   void PilaPush (tElemento x, Pila P);
   void DestruirPila (Pila P);
#endif


/*-------------------------------------------------*/
/* pilamat0.c                                      */
/*-------------------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "pilamat0.h"

Pila CrearPila ()
{
   Pila P;

   P = (Pila) malloc(sizeof(tipoPila));
   if (P == NULL)
      ERROR("CrearPila: No hay memoria");
   P->tope = -1;
   return P;
}
int PilaVacia (Pila P)
{
   return (P->tope == -1);
}

tElemento PilaTope (Pila P)
{
   if (Vacia(P))
      ERROR("PilaTope: Pila vacía");
```

```
   return (P->elementos[P->tope]);
}

void PilaPop (Pila P)
{
   if (Vacia(P))
      ERROR("Pop: Pila vacía");
   P->tope--;
}

void PilaPush (tElemento x, Pila P)
{
   if (P->tope == LMAX - 1)
      ERROR("Push: Pila llena");
   P->tope++;
   P->elementos[P->tope] = x;
}

void DestruirPila (Pila P)
{
   free(P);
}
```

Representación pseudoestática

```
/*-----------------------------------------------*/
/* pilamat1.h                                    */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento; /*Por ejemplo*/
#endif
#ifndef _PILA_
#define _PILA_
   typedef struct {
              tElemento *elementos;
              int Lmax;
              int tope;
            } tipoPila;
   typedef tipoPila *Pila;

   Pila CrearPila (int TamaMax);
```

```
   int PilaVacia (Pila P);
   tElemento PilaTope (Pila P);
   void PilaPop (Pila P);
   void PilaPush (tElemento x, Pila P);
   void DestruirPila (Pila P);
#endif


/*-----------------------------------------------*/
/* pilamat1.c                                    */
/*-----------------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "pilamat1.h"

Pila CrearPila (int TamaMax)
{
   Pila P;

   P = (Pila) malloc(sizeof(tipoPila));
   if (P == NULL)
      ERROR("CrearPila: No hay memoria");
   P->Lmax = TamaMax;
   P->tope = -1;
   P->elementos = (tElemento *)
                  malloc(TamaMax*sizeof(tElemento));
   if (P->elementos == NULL)
      ERROR("CrearPila: No hay memoria");
   return P;
}

int PilaVacia (Pila P)
{
   return (P->tope == -1);
}

tElemento PilaTope (Pila P)
{
   if (Vacia(P))
      ERROR("PilaTope: Pila vacía");
   return (P->elementos[P->tope]);
}
```

```
void PilaPop (Pila P)
{
   if (Vacia(P))
      ERROR("Pop: Pila vacía");
   P->tope--;
}
void PilaPush (tElemento x, Pila P)
{
   if (P->tope == P->Lmax-1)
      ERROR("Push: Pila llena");

   P->tope++;
   P->elementos[P->tope] = x;
}

void DestruirPila (Pila P)
{
   free(P->elementos);
   free(P);
}
```

2.3.2. Implementación de pilas mediante celdas enlazadas

```
/*-----------------------------------------------*/
/* pilaenla.h                                    */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento; /*Por ejemplo*/
#endif
#ifndef _PILA_
#define _PILA_
   typedef struct nodo {
              tElemento elto;
              struct nodo *sig;
            } tipoNodo;
   typedef tipoNodo **Pila;

   Pila CrearPila ();
   int PilaVacia (Pila P);
   tElemento PilaTope (Pila P);
```

```
   void PilaPop (Pila P);
   void PilaPush (tElemento x, Pila P);
   void DestruirPila (Pila P);
#endif


/*-----------------------------------------------*/
/* pilaenla.c                                    */
/*-----------------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "pilaenla.h"

Pila CrearPila ()
{
   Pila P;

   P = (Pila) malloc(sizeof(tipoNodo *));
   if (P == NULL)
      ERROR("CrearPila: No hay memoria");
   *P = NULL;
   return P;
}

int PilaVacia (Pila P)
{
   return (*P == NULL);
}

tElemento PilaTope (Pila P)
{
   if (Vacia(P))
      ERROR("PilaTope: Pila vacía");
   return ((*P)->elto);
}

void PilaPop (Pila P)
{
   tipoNodo *q;

   if (Vacia(P))
      ERROR("Pop: Pila vacía");
   q = *P;
```

```
   *P = q->sig;
   free(q);
}

void PilaPush (tElemento x, Pila P)
{
   tipoNodo *q;

   q = (tipoNodo *)
   malloc(sizeof(tipoNodo));
   if (q == NULL)
      ERROR("Push: No hay memoria");
   q->elto = x;
   q->sig = *P;
   *P = q;
}

void DestruirPila (Pila P)
{
   while (!Vacia(P))
      Pop(P);
   free(P);
}
```

2.4. Aplicaciones de las pilas

Básicamente, las aplicaciones de la pilas están relacionadas con su estructura LIFO. Se utilizan en problemas que se enuncian en un sentido y se resuelven en sentido inverso. Un ejemplo típico sería la suma de dos números enteros. Los números se representan escribiéndolos de izquierda a derecha empezando por su cifra más significativa. Sin embargo, a la hora de efectuar la operación (resolver el problema) debemos empezar por las cifras menos significativas (unidades) e ir avanzando hasta las más significativas (es decir, de derecha a izquierda). También son muy utilizadas en sistemas operativos cuando se hacen llamadas a una función, cuando se utiliza la recursividad, etc.

3

TIPO ABSTRACTO DE DATOS COLA

3.1. Concepto de cola

Una **cola** es una secuencia de elementos en la que las eliminaciones se realizan por un extremo, llamado inicio, frente o principio de la cola, y los nuevos elementos son añadidos por el otro extremo, llamado fondo o final de la cola (figura **??**).

En una cola el primer elemento añadido es el primero en salir de ella, por lo que también se les conoce como estructuras FIFO (*First Input First Output*).

El modelo intuitivo del TAD *cola*, coincide con el concepto de "cola" que utilizamos en la vida cotidiana. Pensemos por ejemplo en la cola de un cine. La adición de un elemento a la cola se produciría cuando una persona nueva se añade al fin de la misma, y el borrado de un elemento, ocurriría cuando un cliente, ya con sus entradas en la mano, abandona la cola.

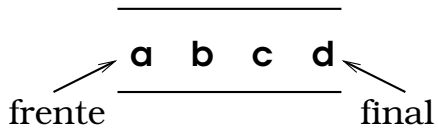


Figura 3.1: Modelo de una cola.

3.2. Especificación del TAD COLA

Definición:

Una cola es una secuencia de elementos de un tipo determinado, en la cuál se pueden añadir elementos sólo por un extremo, al que llamaremos fin, y eliminar por el otro, al que llamaremos inicio.

Operaciones:

Cola CrearCola ()
Postcondiciones: Crea y devuelve una cola vacía.

int ColaVacia (Cola C)
Precondiciones: C está creada.
Postcondiciones: Indica si C está vacía.

int ColaLlena (Cola C)
Precondiciones: C está creada.
Postcondiciones: Indica si C está llena.

tElemento ColaFrente (Cola C)
Precondiciones: C no está vacía.
Postcondiciones: Devuelve el elemento del frente de la cola C.

void ColaPop (Cola C)
Precondiciones: C no está vacía.
Postcondiciones: Elimina el elemento del frente de la cola C y el siguiente se convierte en el nuevo frente.

void ColaPush (tElemento x, Cola C)
Precondiciones: C está creada y no está llena.
Postcondiciones: Inserta el elemento x al final de la cola C.

void DestruirCola (Cola C)
Precondiciones: C está creada.
Postcondiciones: Libera la memoria ocupada por la cola C. Para usarla otra vez se debe volver a crear con la operación *CrearCola()*.

3.3. Implementación del TAD COLA

Análogamente al caso de las pilas, existen dos formas de representar las colas:

a) Mediante el uso de vectores.

b) Mediante el uso de estructuras enlazadas.

Una **representación vectorial** consistiría en un vector para almacenar los elementos y, por otro lado, un entero que indique la posición del fin de la cola. En esta representación no sería necesario guardar la posición del frente de la cola ya que se asumiría que es la primera posición del vector (figura **??**).[1] El mayor inconveniente de esta representación surge en la operación de eliminación de un elemento de la cola, la cuál implicaría mover todos y cada uno de los elementos de la misma, lo que podría resultar muy costoso en función del número y tamaño de los elementos involucrados.

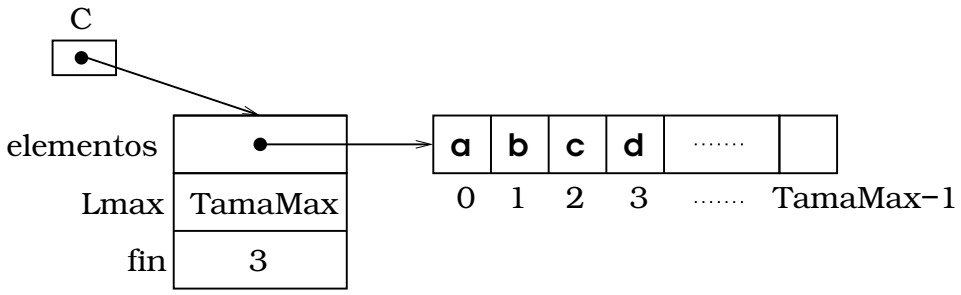


Figura 3.2: Representación vectorial pseudoestática de una cola.

Podría pensarse que una solución obvia al excesivo (y probablemente innecesario) movimiento de datos en la representación anterior sería el uso de dos índices, indicando el principio y fin de la cola. Sin embargo, esta representación nos provoca un problema aún mayor; las sucesivas inserciones y eliminaciones en la cola irán incrementando las posiciones de inicio y fin de la misma, provocando un desplazamiento de los elementos hacia el final del vector. Siguiendo esta estrategia puede ocurrir que los elementos de la cola lleguen a ocupar las últimas posiciones del vector, de manera que se hace imposible añadir nuevos elementos a la misma, cuando realmente el vector no está completamente ocupado, ya que pueden quedar posiciones libres al principio del mismo, lo cual descarta absolutamente esta posibilidad.

Analizando más profundamente la opción anterior, se deduce que el problema de la misma no radica en la movilidad de los índices (que ahorra movimiento de datos), sino en que el extremo final del vector y su extremo inicial no son contiguos, lo que nos impide aprovechar los espacios libres del comienzo del vector. Este razonamiento nos lleva a una representación en la que la primera posición del vector sigue a la última, es decir, se unen los dos extremos del vector creando un vector circular. Para añadir un elemento a la cola se mueve el índice del fin

[1] El tipo cola realmente se define como un puntero a la estructura de datos diseñada. La razón por la que se hace así es la misma que en el caso del TAD *pila* (véase la sección **??**, pág. **??**).

una posición en el sentido de las agujas del reloj y en ella se almacena el elemento. Para suprimir un elemento, basta con mover el índice de inicio de la cola una posición en el mismo sentido. De esta forma, la cola se mueve siempre en el mismo sentido, tanto para inserciones como para borrados (figura **??**). Pero, ¿cómo conseguir que un vector lineal tenga un comportamiento circular? Un nuevo elemento se coloca en la posición $(fin + 1)\ mod\ Lmax$ y para suprimir un elemento $inicio$ se avanza una posición mediante $(inicio+1)\ mod\ Lmax$, donde $incio$ y fin corresponden a las posiciones del primer y último elemento de la cola respectivamente y $Lmax$ es el tamaño del vector.

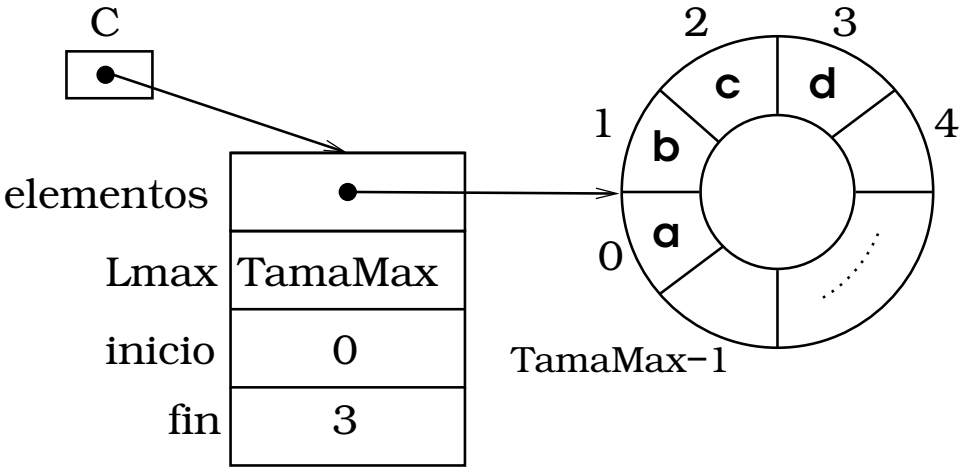
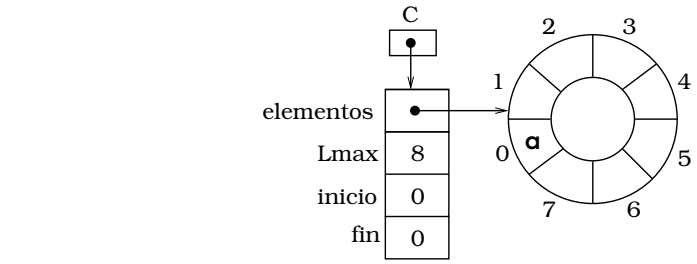


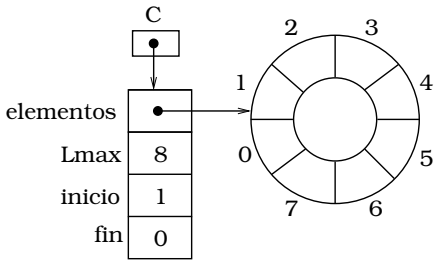
Figura 3.3: Representación vectorial circular de una cola.

Esta representación nos plantea el problema de distinguir entre una cola vacía y una cola llena, ya que la posición relativa de los índices $inicio$ y fin sería la misma en ambos casos (el frente estaría en la posición siguiente al final de la cola, $inicio = (fin+1)\ mod\ Lmax$). Veámoslo con un ejemplo: Supongamos una cola de tamaño máximo 8 que tiene un solo elemento a en la posición 0 del vector, con lo cual $inicio = 0$ y $fin = 0$ (figura **??**). Si este elemento es sacado de la cola, entonces queda vacía e $incio = 1$ y $fin = 0$ (figura **??**). Ahora, supongamos que la cola se llena añadiendo consecutivamente 8 elementos (a, b, c, d, e, f, g y h). En cada inserción el final de la cola avanza una posición y por tanto, da una vuelta completa al vector y se mueve desde 0 hasta volver nuevamente a 0, es decir $inicio = 1$ y $fin = 0$ (figura **??**). Como puede observarse la posición de $inicio$ y fin cuando la cola está vacía es la misma que cuando está llena, lo cual impide saber si hay espacio para nuevos elementos o no.

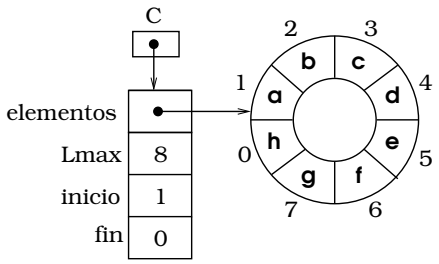
Existen dos soluciones a este problema. Una posibilidad sería añadirle un indicador a la representación de la cola, el cual indicaría si la



(a) Cola con un único elemento.



(b) Cola vacía.



(c) Cola llena.

Figura 3.4: Cola vacía vs. cola llena con un vector circular.

cola está o no vacía. Otra alternativa puede ser emplear un vector con una posición más y sacrificar una de estas posiciones, es decir, no ocupar nunca todas las celdas del vector, sino dejar por lo menos una vacía. De esta forma, la posición relativa de los índices para el caso de cola vacía y cola llena es ahora diferente, ya que estará vacía cuando $inicio$ siga a fin, y estará llena cuando entre ambas posiciones quede una libre, la que hemos sacrificado. Ésta es la estructura de datos que se utiliza en la implementación que aparece en la siguiente sección (pág. **??**).

La **representación** de una cola mediante **estructuras enlazadas** podría realizarse utilizando un solo puntero a uno de los extremos de la misma, o mediante dos punteros, apuntando a cada uno de los extremos.

Con un solo puntero la primera idea que surge es apuntar al inicio de la cola, representada mediante celdas enlazadas en las que cada elemento apunta al siguiente de la misma. Rápidamente notamos que está representación es claramente ineficiente ya que, aunque el acceso al inicio de la cola es de coste $O(1)$, el acceso al fin de la misma es de coste $O(n)$ (para acceder al fin de la cola se debe recorrer toda ella).

Una posible mejora a esta representación, utilizando tan sólo un puntero, sería apuntar al fin de la misma, en lugar de al inicio, y enlazar el último elemento con el primero, de forma que se accedería al fin de la cola en coste $O(1)$, y al inicio en coste $O(2)$ (figura **??**).

Otra posibilidad, acorde con los accesos al TAD *cola*, es representarla mediante dos punteros que apunten al inicio y fin de la cola (figura **??**). Su ventaja es que el acceso a ambos extremos de la cola se consigue en coste $O(1)$, y su inconveniente es que requiere dos punteros en vez de uno para representar la misma.

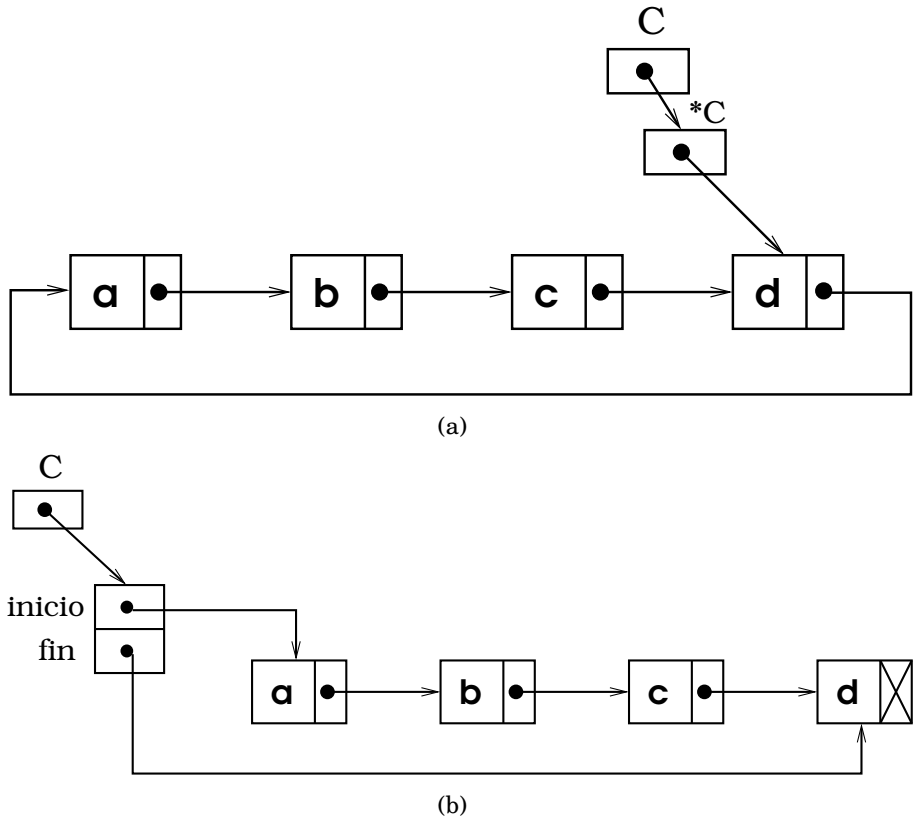


Figura 3.5: Representaciones enlazadas de una cola.

3.3.1. Implementación vectorial de colas

Representación lineal

```
/*-----------------------------------------------*/
/* colamat.h                                     */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento; /*Por ejemplo*/
```

```
#endif
#ifndef _COLA_
#define _COLA_
   typedef struct {
               tElemento *elementos;
               int Lmax;
               int fin;
            } tipoCola;
   typedef tipoCola *Cola;

   Cola CrearCola (int TamaMax);
   int ColaVacia (Cola C);
   int ColaLlena (Cola C);
   tElemento ColaFrente (Cola C);
   void ColaPop (Cola C);
   void ColaPush (tElemento x, Cola C);
   void DestruirCola (Cola C);
#endif


/*-------------------------------------------------*/
/* colamat.c                                       */
/*-------------------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "colamat.h"

Cola CrearCola (int TamaMax)
{
   Cola C;

   C = (Cola) malloc(sizeof(tipoCola));
   if (C == NULL)
      ERROR("CrearCola: No hay memoria");
   C->elementos = (tElemento *)
                  malloc(TamaMax*sizeof(tElemento));
   if (C->elementos == NULL)
      ERROR("CrearCola: No hay memoria");
   C->Lmax = TamaMax;
   C->fin = -1;
   return C;
}
```

```
int ColaVacia (Cola C)
{
   return (C->fin == -1);
}

int ColaLlena (Cola C)
{
   return (C->fin == C->Lmax - 1);
}

tElemento ColaFrente (Cola C)
{
   if (ColaVacia(C))
      ERROR("ColaFrente: Cola vacía");
   return *(C->elementos);
}

void ColaPop (Cola C)
{
   int i;

   if (ColaVacia(C))
      ERROR("ColaPop: Cola vacía");

   for(i = 0; i < C->fin; i++)
      C->elementos[i] = C->elementos[i+1];
   C->fin--;
}

void ColaPush (tElemento x, Cola C)
{
   if (ColaLlena(C))
      ERROR("ColaPush: Cola llena");

   C->fin++;
   C->elementos[C->fin] = x;
}

void DestruirCola (Cola C)
{
   free(C->elementos);
   free(C);
}
```

Representación circular

```
/*-----------------------------------------------*/
/* colacir.h                                     */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento;  /*Por ejemplo*/
#endif
#ifndef _COLA_
#define _COLA_
   typedef struct {
              tElemento *elementos;
              int Lmax;
              int inicio, fin;
            } tipoCola;
   typedef tipoCola *Cola;

   Cola CrearCola (int TamaMax);
   int ColaVacia (Cola C);
   int ColaLlena (Cola C);
   tElemento ColaFrente (Cola C);
   void ColaPop (Cola C);
   void ColaPush (tElemento x, Cola C);
   void DestruirCola (Cola C);
#endif

/*-----------------------------------------------*/
/* colacir.c                                     */
/*-----------------------------------------------*/
#include <stdlib.h>
#include "error.h"
#include "colacir.h"

Cola CrearCola (int TamaMax)
{
   Cola C;
   C = (Cola) malloc(sizeof(tipoCola));
   if (C == NULL)
      ERROR("CrearCola: No hay memoria");
   C->elementos = (tElemento *)
             malloc((TamaMax+1)*sizeof(tElemento));
   if (C->elementos == NULL)
```

```
      ERROR("CrearCola: No hay memoria");
   C->Lmax = TamaMax + 1;
   C->inicio = 0;
   C->fin = C->Lmax - 1;
   return C;
}

int ColaVacia (Cola C)
{
   return (C->fin+1) % C->Lmax == C->inicio;
}

int ColaLlena (Cola C)
{
   return (C->fin+2) % C->Lmax == C->inicio;
}

tElemento ColaFrente (Cola C)
{
   if (ColaVacia(C))
      ERROR("ColaFrente: Cola vacía");
   return C->elementos[C->inicio];
}

void ColaPop (Cola C)
{
   if (ColaVacia(C))
      ERROR("ColaPop: Cola vacía");
   C->inicio = (C->inicio+1) % C->Lmax;
}

void ColaPush (tElemento x, Cola C)
{
   if (ColaLlena(C))
      ERROR("ColaPush: Cola llena");
   C->fin = (C->fin+1) % C->Lmax;
   C->elementos[C->fin] = x;
}

void DestruirCola (Cola C)
{
   free(C->elementos);
   free(C);
}
```

3.3.2. Implementación de colas mediante estructuras enlazadas

```
/*-----------------------------------------------*/
/* colaenla.h                                    */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento;  /*Por ejemplo*/
#endif
#ifndef _COLA_
#define _COLA_
   typedef struct nodo {
              tElemento elto;
              struct nodo *sig;
           } tipoNodo;
   typedef struct {
              tipoNodo *inicio,
                       *fin;
           } tipoCola;
   typedef tipoCola *Cola;

   Cola CrearCola ();
   int ColaVacia (Cola C);
   int ColaLlena (Cola C);
   tElemento ColaFrente (Cola C);
   void ColaPop (Cola C);
   void ColaPush (tElemento x, Cola C);
   void DestruirCola (Cola C);
#endif


/*-----------------------------------------------*/
/* colaenla.c                                    */
/*-----------------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "colaenla.h"

Cola CrearCola ()
{
   Cola C;
```

```
   C = (Cola) malloc(sizeof(tipoCola));
   if (C == NULL)
      ERROR("CrearCola: No hay memoria");
   C->inicio = C->fin = NULL;
   return C;
}

int ColaVacia (Cola C)
{
   return (C->inicio == NULL);
}

int ColaLlena (Cola C)
{
   return 0;
}

tElemento ColaFrente (Cola C)
{
   if (ColaVacia(C))
      ERROR("ColaFrente: Cola vacía");
   return C->inicio->elto;
}

void ColaPop (Cola C)
{
   tipoNodo *q;

   if (ColaVacia(C))
      ERROR("ColaPop: Cola vacía");
   q = C->inicio;
   C->inicio = q->sig;
   free(q);
}

void ColaPush (tElemento x, Cola C)
{
   tipoNodo *q;

   q = (tipoNodo *) malloc(sizeof(tipoNodo));
   if (q == NULL)
      ERROR("ColaPush: No hay memoria");
   q->elto = x;
```

```
   q->sig = NULL;
   if (ColaVacia(C))
      C->inicio = q;
   else
      C->fin->sig = q;
   C->fin = q;
}

void DestruirCola (Cola C)
{
   while (!ColaVacia(C))
      ColaPop(C);
   free(C);
}
```

3.4. Aplicaciones de las colas

Las aplicaciones de este tipo abstracto de dato tienen que ver con su estructura FIFO. Dado que el TAD, y el concepto en la vida cotidiana de la cola representan lo mismo, cualquier simulación por ordenador de una típica cola del mundo real, utilizaría este TAD.

Particularmente, en el mundo de la computación, podemos encontrar numerosos ejemplos en los que intervienen colas, a saber, las tareas a realizar por una impresora, acceso a almacenamiento de disco, el uso de la CPU en sistemas de tiempo compartido, etc. En definitiva, las colas son utilizadas para modelizar el reparto de un recurso entre diferentes procesos

4

TIPO ABSTRACTO DE DATOS LISTA

4.1. Concepto de lista

Una **lista** es una secuencia de elementos de un tipo determinado. Se suele representar como una sucesión de elementos $(a_1, a_2, \ldots, a_n)$, donde $n \geq 0$ es el número de elementos que tiene. Se denomina *longitud* de la lista a este número. Si $n = 0$, es decir, si la lista no tiene elementos, entonces se denomina *lista vacía*.

Cada elemento ocupa una *posición* en la lista, de modo que los elementos están ordenados de forma lineal según las posiciones que ocupan. Todos los elementos, salvo el *primero*, tienen un único *predecesor* y todos, excepto el *último*, tienen un único *sucesor*.

Las operaciones sobre listas permiten acceder, insertar y suprimir elementos en cualquier posición de ellas.

Según estas definiciones, las pilas y las colas no son más que tipos especiales de listas en las que las operaciones están restringidas a los elementos situados en los extremos de la secuencia. Si los elementos se pueden añadir y eliminar sólo por un extremo, se trata de una pila, si sólo se pueden añadir por un extremo y suprimir por el otro, entonces tenemos una cola.

4.2. Especificación del TAD LISTA

Para formar un tipo abstracto de datos a partir del concepto de lista, se debe definir un conjunto de operaciones con objetos de tipo *Lista*.

Como sucede con todos los TAD, ningún conjunto de operaciones es adecuado para todas las aplicaciones. Además, debido a la flexibilidad de las listas en particular, este conjunto puede ser muy amplio. Por ello, analizaremos un conjunto representativo de operaciones básicas o primitivas que nos permitan definir a partir de ellas otras más complejas que nos puedan hacer falta en una aplicación concreta.

Convendremos que una lista tiene una posición ***Fin*** que sucede a la del último elemento. Esta posición no ocupada por ningún elemento nos será útil, por ejemplo, para conocer la longitud de la lista o para añadir un elemento al final.

Definición:

Una lista es una secuencia de elementos de un tipo determinado. Se puede representar de la forma

$$L = (a_1, a_2, \ldots, a_n)$$

donde $n \geq 0$ y $longitud = n$.

Si $n = 0$, entonces es una *lista vacía*.

Posición: lugar que ocupa un elemento en la lista.

Los elementos están ordenados de forma lineal según las posiciones que ocupan. Todos los elementos, salvo el *primero*, tienen un único *predecesor* y todos, excepto el *último*, tienen un único *sucesor*.

Posición *Fin*: Posición siguiente a la del último elemento. Esta posición no está ocupada nunca por ningún elemento.

Operaciones:

Lista CrearLista ()

Postcondiciones: Crea y devuelve una lista vacía.

void Insertar (tElemento x, posicion p, Lista L)

Precondiciones: $L = (a_1, a_2, \ldots, a_n)$, $1 \leq p \leq n+1$

Postcondiciones: $L = (a_1, \ldots, a_{p-1}, x, a_p, \ldots, a_n)$

void Eliminar (posicion p, Lista L)

Precondiciones: $L = (a_1, a_2, \ldots, a_n)$, $1 \leq p \leq n$

Postcondiciones: $L = (a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n)$

tElemento Recuperar (posicion p, Lista L)
Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$1 \leq p \leq n$
Postcondiciones: Devuelve el elemento que ocupa la posición p (a_p) de la lista L.

posicion Buscar (tElemento x, Lista L)
Precondiciones: L está creada.
Postcondiciones: Devuelve la posición de la primera ocurrencia de x en al lista L. Si x no pertenece a L, devuelve la posición *Fin*.

posicion Siguiente (posicion p, Lista L)
Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$1 \leq p \leq n$
Postcondiciones: Devuelve la posición siguiente a p de la lista L.

posicion Anterior (posicion p, Lista L)
Precondiciones: $L = (a_1, a_2, \ldots, a_n)$
$2 \leq p \leq n + 1$
Postcondiciones: Devuelve la posición anterior a p de la lista L.

posicion Primera (Lista L)
Precondiciones: L está creada.
Postcondiciones: Devuelve la primera posición de la lista L. Si la lista está vacía devuelve la posición *Fin*.

posicion Fin (Lista L)
Precondiciones: L está creada.
Postcondiciones: Devuelve la última posición de la lista L, la siguiente a la del último elemento. Esta posición siempre está vacía, no existe ningún elemento que la ocupe.

void DestruirLista (Lista L)
Precondiciones: L está creada.
Postcondiciones: Libera la memoria ocupada por la lista L. Para usarla otra vez se debe volver a crear con la operación *CrearLista()*.

4.3. Implementación del TAD LISTA

4.3.1. Implementación vectorial

Los elementos de una lista se pueden almacenar en celdas contiguas de una matriz o vector de un tamaño dado. Además, será necesario guar-

dar la longitud que tenga la lista en cada momento, o bien el índice de la celda que contenga el último elemento (figura **??**).[1]

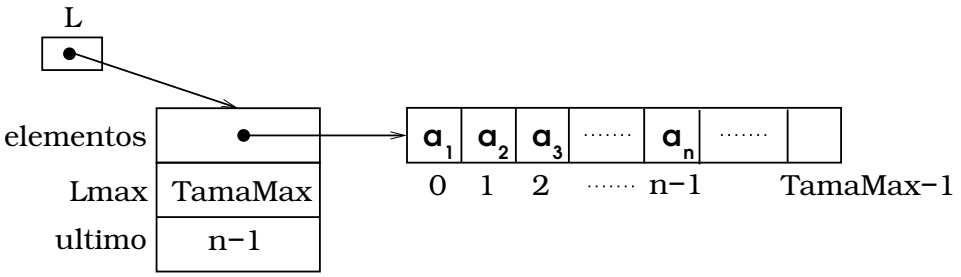


Figura 4.1: Representación vectorial de una lista.

El i-ésimo elemento de la lista está en la i-ésima celda del vector, la cual tiene el índice $i - 1$. De esta forma, las posiciones se pueden representar mediante enteros; la ***i*-ésima posición** mediante el entero ***i* – 1**, para $1 \leq i \leq longitud$, y la posición ***Fin***, mediante ***longitud***.

Esta estructura de datos permite el acceso directo al i-ésimo elemento de una lista a través del índice del vector. Sin embargo, para insertar o eliminar un elemento es necesario desplazar todos los elementos que siguen a éste una posición hacia delante o hacia atrás, respectivamente. En el caso más desfavorable, es decir, al insertar o eliminar al principio de la lista, habrá que mover todos los elementos, lo cuál podría ser muy costoso en función del número y tamaño de los elementos (como ocurría en el TAD *cola* al eliminar un elemento).

Otro inconveniente de esta estructura de datos es que exige especificar el tamaño máximo de una lista. En muchos casos la elección es difícil, ya que la cantidad de datos que habrá que manejar es impredecible. Si elegimos un tamaño muy grande, estaremos desperdiciando memoria la mayor parte del tiempo. Si, por el contrario, tratamos de ahorrar espacio y elegimos un tamaño pequeño, entonces corremos el riesgo de que la lista se llene.

```
/*-----------------------------------------------*/
/* listamat.h                                    */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento;
```

[1] Al igual que en los capítulos anteriores, el tipo lista se definirá como un puntero a la estructura de datos en la que se almacena. Véase la sección **??**, pág. **??**, para una explicación detallada de las razones por las que se hace así.

```
#endif
#ifndef _LISTA_
#define _LISTA_
   typedef struct {
              tElemento *elementos;
              int Lmax;            /* Tamaño máx */
              int ultimo;    /* pos. último elto */
           } tipoLista;
   typedef tipoLista *Lista;
   typedef int posicion;

   Lista CrearLista (int TamaMax);
   void Insertar (tElemento x, posicion p, Lista L);
   void Eliminar (posicion p, Lista L);
   tElemento Recuperar (posicion p, Lista L);
   posicion Buscar (tElemento x, Lista L);
   posicion Siguiente (posicion p, Lista L);
   posicion Anterior (posicion p, Lista L);
   posicion Primera (Lista L);
   posicion Fin (Lista L);
   void DestruirLista (Lista L);
#endif


/*-----------------------------------------------*/
/* listamat.c                                    */
/*-----------------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "listamat.h"

/* Operaciones internas */

static int Igual (tElemento a, tElemento b)
/* Compara dos datos, a y b, de tipo tElemento y
   devuelve 0 si son distintos o cualquier otro
   valor si son iguales. */
{
   if (a == b) return 1;
   else return 0;
}

/* Operaciones públicas */
```

```
Lista CrearLista (int TamaMax)
{
   Lista L;

   L = (Lista) malloc(sizeof(tipoLista));
   if (L == NULL)
      ERROR("CrearLista: No hay memoria");
   L->elementos = (tElemento *)
                  malloc(TamaMax*sizeof(tElemento));
   if (L->elementos == NULL)
      ERROR("CrearLista: No hay memoria");
   L->Lmax = TamaMax;
   L->ultimo = -1;
   return L;
}

void Insertar (tElemento x, posicion p, Lista L)
{
   posicion q;

   if (p < 0 || p > L->ultimo+1)
      ERROR("Insertar: Posición incorrecta");
   if (L->ultimo >= L->Lmax-1)
      ERROR("Insertar: Lista llena");
   /* Desplaza los eltos en p, p+1, ...
      a la siguiente posición */
   for (q = L->ultimo; q >= p; q--)
      L->elementos[q+1] = L->elementos[q];
   L->ultimo++;
   L->elementos[p] = x;
}

void Eliminar (posicion p, Lista L)
{
   posicion q;

   if (p < 0 || p > L->ultimo)
      ERROR("Eliminar: Posición incorrecta");
   /* Desplaza los elementos en p+1, p+2, ...
      a la posición anterior */
   L->ultimo--;
   for (q = p; q <= L->ultimo; q++)
      L->elementos[q] = L->elementos[q+1];
```

```
}

tElemento Recuperar (posicion p, Lista L)
{
   if (p < 0 || p > L->ultimo)
      ERROR("Recuperar: Posición incorrecta");
   return L->elementos[p];
}

posicion Buscar (tElemento x, Lista L)
{
   posicion q;
   int encontrado;

   q = 0;
   encontrado = 0;
   while (q <= L->ultimo && !encontrado)
      if (Igual(L->elementos[q], x))
         encontrado = 1;
      else q++;
   return q;
}

posicion Siguiente (posicion p, Lista L)
{
   if (p < 0 || p > L->ultimo)
      ERROR("Siguiente: Posición incorrecta");
   return p+1;
}

posicion Anterior (posicion p, Lista L)
{
   if (p <= 0 || p > L->ultimo+1)
      ERROR("Anterior: Posición incorrecta");
   return p-1;
}

posicion Primera (Lista L)
{
   return 0;
}

posicion Fin (Lista L)
{
```

```
   return L->ultimo+1;
}

void DestruirLista (Lista L)
{
   free(L->elementos);
   free(L);
}
```

4.3.2. Implementación mediante celdas enlazadas

Esta estructura de datos consiste en una secuencia enlazada de nodos que almacenan elementos consecutivos. Cada nodo almacena un elemento y un puntero al nodo que contiene el siguiente elemento. El nodo que contiene el último elemento posee un puntero nulo (`NULL`). Así pues, podríamos pensar en una lista como un puntero al primer elemento y en una lista vacía como un puntero `NULL`. Sin embargo, cuando se realice alguna operación que modifique el primer elemento será necesario cambiar ese puntero inicial para que apunte al nuevo primer elemento. Por tanto, dicho puntero es necesario pasarlo por referencia a las operaciones del TAD que lo modifiquen. Una forma de hacerlo es utilizando otro puntero que apunte a éste. En consecuencia, una **lista** será un **puntero (`L`) a un puntero (`*L`) al primer elemento** y una lista vacía se representará como un puntero a un puntero `NULL` (figura **??**).

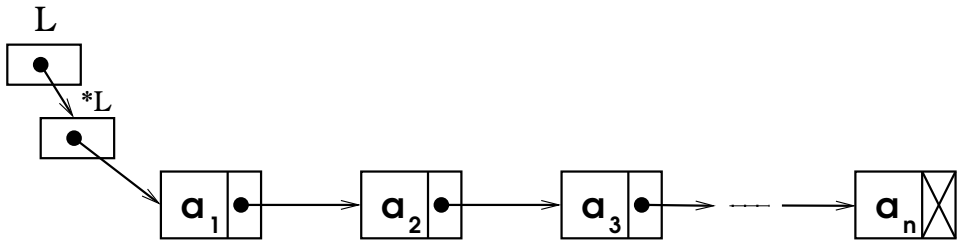


Figura 4.2: Representación enlazada de una lista.

Un nodo se crea cuando se inserta un nuevo elemento en la lista y se suprime cuando se elimina el elemento. Con esta implementación, se elude la reserva de un espacio de memoria de tamaño fijo y, también, el desplazamiento de elementos para hacer inserciones o para rellenar los huecos al eliminar elementos. A cambio se ocupa espacio adicional para almacenar los punteros.

Definiremos la **posición de un elemento como un puntero al nodo que lo contiene**. Así que la primera posición será un puntero al primer nodo de la lista (si la lista está vacía, éste será `NULL`) y la última

posición, es decir, la que hemos llamado *Fin*, corresponde al valor del puntero almacenado en el último nodo de la lista, o sea NULL.

```
/*-----------------------------------------------*/
/* listenla.h                                    */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento;
#endif
#ifndef _LISTA_
#define _LISTA_
   typedef struct nodo {
              tElemento elemento;
              struct nodo *sig;
           } tipoNodo;
   typedef tipoNodo **Lista;
   typedef tipoNodo *posicion;

   Lista CrearLista ();
   void Insertar (tElemento x, posicion *p, Lista L);
   void Eliminar (posicion *p, Lista L);
   tElemento Recuperar (posicion p, Lista L);
   posicion Buscar (tElemento x, Lista L);
   posicion Siguiente (posicion p, Lista L);
   posicion Anterior (posicion p, Lista L);
   posicion Primera (Lista L);
   posicion Fin (Lista L);
   void DestruirLista (Lista L);
#endif
```

Implementación de algunas operaciones

```
Lista CrearLista ()
{
   Lista L;

   L = (Lista) malloc(sizeof(tipoNodo *));
   if (L == NULL)
      ERROR("CrearLista: No hay memoria");
   *L = NULL;
   return L;
```

```
}

void DestruirLista (Lista L)
{
   posicion p;

   while (*L != NULL)
   {
      p = *L;
      *L = p->sig;
      free(p);
   }
   free(L);
}

void Insertar (tElemento x, posicion *p,Lista L)
{
   posicion q;

   if (*p == *L) /* inserción al ppio */
   {
      *L = (tipoNodo *) malloc(sizeof(tipoNodo));
      if (*L == NULL)
         ERROR("Insertar: No hay memoria");
      q = *L;
   }
   else /* cualquier otra posición */
   {
      q = *L;
      while (q->sig != *p)
         q = q->sig;
      q->sig = (tipoNodo *) malloc(sizeof(tipoNodo));
      if (q->sig == NULL)
         ERROR("Insertar: No hay memoria");
      q = q->sig;
   }
   q->elemento = x;
   q->sig = *p;
   *p = q;
}
```

Como podemos ver, la operación de inserción es poco eficiente, pues para insertar un nuevo elemento en la posición representada por un puntero `*p` es necesario recorrer la lista con un puntero auxiliar `q` hasta

colocarlo en el nodo anterior. Esto es debido a que es necesario cambiar el puntero `sig` de este nodo para que apunte al nuevo y la única forma de llegar hasta él es recorriendo la lista desde el inicio.

La operación de inserción se puede implementar de forma que se evite tener que recorrer la lista siempre. Para ello, se crea un nuevo nodo en el que se copia el que está en la posición `*p` y se sustituye el elemento de esta posición por el nuevo.

```
void Insertar (tElemento x, posicion *p, Lista L)
{
   posicion q;

   if (*p != NULL) /* *p no es la pos Fin */
   {
      q = (tipoNodo *) malloc(sizeof(tipoNodo));
      if (q == NULL)
         ERROR("Insertar: No hay memoria");
      *q = **p; /* copia el elto **p en *q */
      (*p)->elemento = x;
      (*p)->sig = q;
   }
   else /* inserción al final */
   {
      if (*L == NULL) /* lista vacía */
      {
         *L = (tipoNodo *) malloc(sizeof(tipoNodo));
         if (*L == NULL)
           ERROR("Insertar: No hay memoria");
         q = *L;
      }
      else
      {
         q = *L;
         while (q->sig != NULL)
            q = q->sig;
         q->sig = (tipoNodo *)
                          malloc(sizeof(tipoNodo));
         if (q->sig == NULL)
           ERROR("Insertar: No hay memoria");
         q = q->sig;
      }
      q->elemento = x;
      q->sig = NULL;
```

```
      *p = q;
   }
}
```

A pesar de todo, para insertar un nuevo elemento al final de la lista, es decir, en la posición *Fin*, sigue siendo necesario recorrerla completamente para situar un puntero `q` en el último nodo. Por lo que, en el peor caso, ambos procedimientos consumen el mismo tiempo. Además, si el elemento es grande, también hay que tener en cuenta que la copia necesitará bastante tiempo. En definitiva, dado que la segunda versión de la función *Insertar()* se hace más difícil de leer y que la ganancia de tiempo puede no ser importante, optaremos por la primera versión.

Obsérvese que la función *Insertar()* modifica el puntero `*p`. Esto es porque el nuevo elemento ocupa ahora la posición que representa `*p`, así que por consistencia hay que hacer que `*p` apunte al nuevo elemento. Vamos a verlo con un ejemplo: Supóngase que se tiene una lista con tres elementos $L = (a, b, c)$ y una variable `q` de tipo posición que representa la posición que ocupa b en la lista, es decir, en realidad `q` apunta al nodo que contiene b. Si se llama a la función `Insertar(x, q, L)`, la lista se convertiría en $L = (a, x, b, c)$ y `q` no debería seguir apuntando a b, ya que en realidad b ha cambiado de posición. Es lógico que `q` apunte a x, que ahora ocupa la posición en la que estaba b.

En la eliminación de elementos es inevitable recorrer la lista desde el principio (en el peor caso, habrá que recorrerla entera), pues es necesario cambiar el puntero del nodo anterior al que se suprime para hacer que apunte al siguiente.

Igual que ocurre al insertar, al suprimir un elemento también hay que modificar el puntero `*p` para hacer que apunte al nodo que pasa a ocupar la posición afectada por la operación.

```
void Eliminar (posicion *p, Lista L)
{
   posicion q;

   if (*p == NULL)
      ERROR("Eliminar: Elemento inexistente");
   if (*L == *p) /* primera pos */
   {
      *L = (*p)->sig;
      free(*p);
      *p = *L;
   }
```

```
   else
   {
      q = *L;
      while (q->sig != *p)
         q = q->sig;
      q->sig = (*p)->sig;
      free(*p);
      *p = q->sig;
   }
}
```

Podemos observar que las inserciones y eliminaciones en la primera posición de la lista hay que tratarlas de forma especial, porque el primer nodo no tiene ninguno que le preceda. Esto complica algo los dos algoritmos.

Una última observación que hacer a esta implementación es que la utilización del TAD *lista* no es independiente de la representación, puesto que las llamadas a las funciones *Insertar()* y *Eliminar()* no se ajustan a la especificación del TAD. Es decir, cuando se utilice esta implementación mediante celdas enlazadas, las llamadas a las funciones para insertar y eliminar un elemento serían de la forma

```
Insertar(x, &p, L);
Eliminar(&p, L);
```

y no de la forma especificada

```
Insertar(x, p, L);
Eliminar(p, L);
```

En la siguiente sección se propone una estructura de datos mejorada para resolver este problema.

En cuanto a la elección de la representación de listas adecuada para una aplicación concreta, va a depender de las operaciones que se deseen realizar, o de las que se realicen con mayor frecuencia, y de la longitud que pueda llegar a tener la lista. En general, habrá que atenerse a los siguientes criterios:

1. La representación con vectores requiere fijar el tamaño máximo de la lista, lo cuál implica que se ocupará todo el espacio reservado, independientemente de la longitud que tenga la lista en un momento dado. Si el espacio reservado es mucho más grande que el

tamaño medio de la lista, se malgastará espacio casi todo el tiempo. Si, por ahorrar espacio, se fija un tamaño máximo demasiado justo, se corre el riesgo de llenar la lista y que no sea posible almacenar todos los elementos necesarios en un momento dado.

2. La representación con punteros utiliza sólo el espacio necesario para almacenar los elementos que tiene la lista en cada momento, pero necesita espacio adicional para guardar el puntero a cada nodo. Si el espacio ocupado por los punteros en relación al espacio requerido para almacenar los datos es muy pequeño, quizás interese la implementación con punteros.

3. *Insertar()* y *Eliminar()* son más lentas con vectores, porque necesitan mover los elementos dentro del vector. Cuando se utilizan punteros, sólo hay que cambiar el valor de unos pocos punteros y se evita el movimiento de los elementos que almacena la lista.

4. El acceso a una posición anterior es menos eficiente en la implementación mediante punteros, pues requiere recorrer la lista desde el principio. Por el contrario, al utilizar vectores, el acceso a una posición anterior es directo.

4.4. Otras estructuras enlazadas

4.4.1. Listas con cabecera

Para resolver el problema de la dependencia de la representación (en la implementación de listas con celdas enlazadas) introduciremos algunas modificaciones en la estructura de datos.

Una posibilidad consiste en considerar que la posición del i-ésimo elemento es un puntero al nodo anterior, en lugar de un puntero al i-ésimo nodo. Con esto, también conseguimos no tener que recorrer la lista para situar un puntero en el nodo anterior, al insertar y eliminar elementos, pues ahora la posición de un elemento es este puntero.

Sin embargo, también hemos visto que el hecho de que el primer nodo de una lista no tenga predecesor obliga a un tratamiento especial de las inserciones y supresiones en la primera posición. Este segundo problema no sólo no está resuelto, sino que se ha agravado. Si el primer elemento no tiene predecesor, ¿cómo representamos la primera posición de una lista? Podemos colocar un nodo cabecera al principio de la lista en el que no almacenaremos ningún elemento, simplemente, lo utilizaremos para representar la posición del primer elemento. También,

se podría utilizar para guardar información de la lista como, por ejemplo, la longitud. Con este nodo cabecera, conseguimos también que para cualquier elemento de la lista exista un nodo anterior, con lo cual las inserciones y eliminaciones al principio de la misma se harán exactamente igual que en cualquier otra posición. En la figura **??** se muestra una lista representada mediante esta nueva estructura de datos.

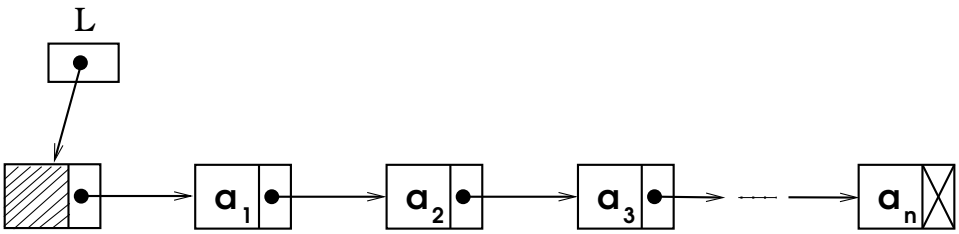


Figura 4.3: Representación enlazada con cabecera de una lista.

Obsérvese que en este caso la lista es un puntero `L` al nodo cabecera y no un puntero a un puntero, ya que, una vez creada la lista, `L` no es necesario modificarlo para llevar a cabo ninguna operación del TAD y por tanto, puede ser un parámetro pasado por valor.

```
/*-----------------------------------------------*/
/* listenla.h                                    */
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento;
#endif
#ifndef _LISTA_
#define _LISTA_
   typedef struct nodo {
              tElemento elemento;
              struct nodo *sig;
           } tipoNodo;
   typedef tipoNodo *Lista;
   typedef tipoNodo *posicion;

   Lista CrearLista ();
   void Insertar (tElemento x, posicion p, Lista L);
   void Eliminar (posicion p, Lista L);
   tElemento Recuperar (posicion p,Lista L);
   posicion Buscar (tElemento x, Lista L);
   posicion Siguiente (posicion p, Lista L);
```

```
   posicion Anterior (posicion p, Lista L);
   posicion Primera (Lista L);
   posicion Fin (Lista L);
   void DestruirLista (Lista L);
#endif
```

Implementación de algunas operaciones

```
Lista CrearLista ()
{
   Lista L;

   L = (Lista) malloc(sizeof(tipoNodo));
   if (L == NULL)
      ERROR("CrearLista: No hay memoria");
   L->sig = NULL;
   return L;
}

void DestruirLista (Lista L)
{
   posicion p;

   while (L != NULL)
   {
      p = L;
      L = p->sig;
      free(p);
   }
}

void Insertar (tElemento x, posicion p, Lista L)
{
   posicion q;

   q = (posicion) malloc(sizeof(tipoNodo));
   if (q == NULL)
      ERROR("Insertar: No hay memoria");
   q->elemento = x;
   q->sig = p->sig;
   p->sig = q;
}
```

```
void Eliminar (posicion p, Lista L)
{
   posicion q;

   if (p->sig == NULL)
      ERROR("Eliminar: Elemento inexistente");
   q = p->sig;
   p->sig = q->sig;
   free(q);
}

posicion Anterior (posicion p, Lista L)
{
   posicion q;

   if (p == L)
      ERROR("Anterior: Posición incorrecta");

   q = L;
   while (q->sig != p)
      q = q->sig;
   return q;
}

posicion Fin (Lista L)
{
   posicion p;

   p = L;
   while (p->sig != NULL)
      p = p->sig;
   return p;
}
```

Nótese que ahora, en las operaciones *Insertar()* y *Eliminar()*, el puntero p no hay que cambiarlo y se transfiere por valor, porque la posición que representa cuando se llama a estas funciones es la misma que ocupará el nuevo elemento que se inserta o el elemento siguiente al que se suprime después de ejecutar la operación. Por tanto, con la introducción del nodo cabecera hemos conseguido que estas operaciones se ajusten a la especificación dada.

A excepción de las operaciones que se refieren a una lista globalmente, como son *CrearLista()*, *DestruirLista()*, *Buscar()*, *Primera()* y *Fin()*,

las demás no necesitan el parámetro `L` de tipo `Lista`, pero es imprescindible incluirlo para mantener la homogeneidad con las otras implementaciones del TAD *lista*. Si suprimimos `L` en los argumentos, la especificación y utilización del TAD dejan de ser independientes de la implementación.

4.4.2. Listas doblemente enlazadas

Son listas en las que cada nodo tiene dos enlaces, uno al nodo anterior y otro al siguiente (figura **??**). Son útiles cuando es necesario recorrer eficientemente una lista, tanto hacia delante como hacia atrás, o también, cuando se quiera que la operación *Anterior()* sea rápida.

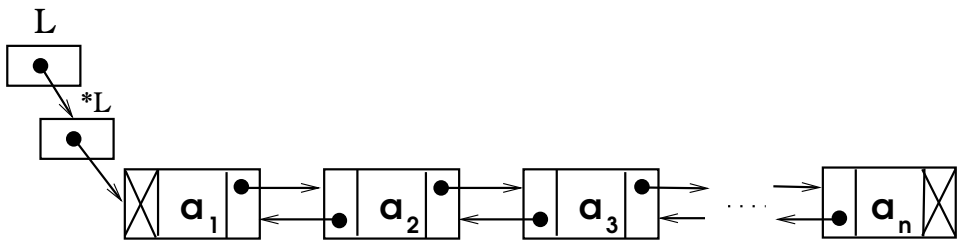


Figura 4.4: Representación doblemente enlazada de una lista.

Esta estructura de datos se define como sigue:

```
#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento;
#endif
#ifndef _LISTA_
#define _LISTA_
   typedef struct nodo {
              tElemento elemento;
              struct nodo *ant, *sig;
           } tipoNodo;
   typedef tipoNodo **Lista;
   typedef tipoNodo *posicion;
#endif
```

Eliminación en una lista doblemente enlazada. Versión 1.

```
void Eliminar (posicion *p, Lista L)
{
```

```
   posicion q;

   if (*p == NULL)
      ERROR("Eliminar: Elemento inexistente");
   q = (*p)->sig;
   if ((*p)->ant == NULL) /* es el primero */
      *L = (*p)->sig;
   else
      (*p)->ant->sig = q;
   if ((*p)->sig != NULL) /*no es el último*/
      (*p)->sig->ant = (*p)->ant;
   free(*p);
   *p = q; /* el sigte. queda en pos. p */
}
```

Esta versión tiene dos inconvenientes:

- La cabecera de la función no se corresponde con la especificación del TAD.
- Hay que tratar como casos especiales la eliminación del primer y último nodo.

Eliminación en una lista doblemente enlazada. Versión 2.

Para evitar los inconvenientes de la versión 1 se realizan dos modificaciones en la estructura de datos (figura **??**):

- Introducción de un nodo cabecera.
- Representación de la posición de un elemento como un puntero al nodo anterior.

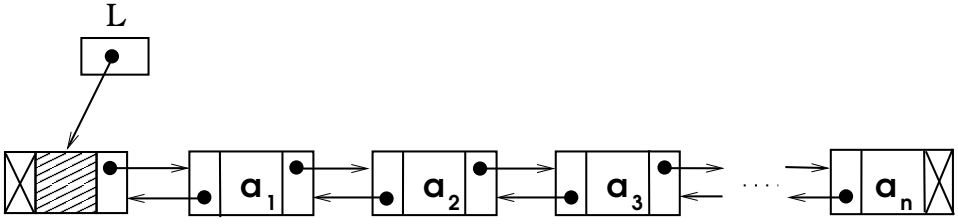


Figura 4.5: Representación doblemente enlazada con cabecera de una lista.

La definición de tipos se hace de la siguiente forma:

```
typedef struct nodo {
           tElemento elemento;
           struct nodo *ant, *sig;
        } tipoNodo;
typedef tipoNodo *Lista;
typedef tipoNodo *posicion;
```

Y como podemos ver la implementación de *Eliminar()* es más sencilla, aunque hay que tener en cuenta que el último elemento se debe seguir eliminando de forma diferente al resto.

```
void Eliminar (posicion p, Lista L)
{
   posicion q;

   if (p->sig == NULL)
      ERROR("Eliminar: Elemento inexistente");
   q = p->sig;
   p->sig = q->sig;
   if (q->sig != NULL) /* no es el último */
      q->sig->ant = p;
   free(q);
}
```

Lista doblemente enlazada y circular

Para evitar tratar como caso especial la eliminación del último elemento de la lista, se puede hacer ésta doblemente enlazada circular. Esto es, el campo `sig` del último nodo apunta a la cabecera y el campo `ant` de la cabecera apunta al último nodo (figura **??**).

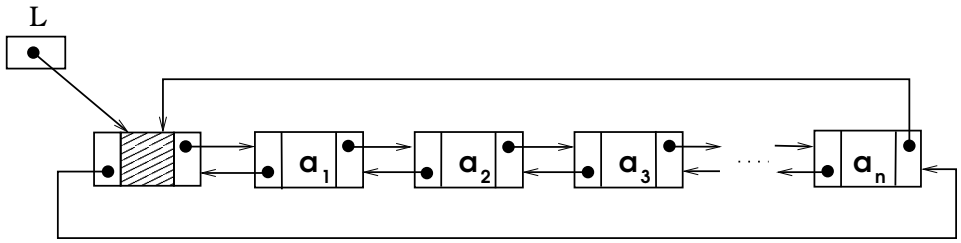


Figura 4.6: Representación doblemente enlazada y circular de una lista.

```
/*-----------------------------------------------*/
/* lisdoble.h                                    */
```

```
/*-----------------------------------------------*/

#ifndef _tElemento_
#define _tElemento_
   typedef int tElemento;
#endif
#ifndef _LISTA_
#define _LISTA_
   typedef struct nodo {
               tElemento elemento;
               struct nodo *ant, *sig;
            } tipoNodo;
   typedef tipoNodo *Lista;
   typedef tipoNodo *posicion;

   Lista CrearLista ();
   void Insertar (tElemento x, posicion p, Lista L);
   void Eliminar (posicion p, Lista L);
   tElemento Recuperar (posicion p, Lista L);
   posicion Buscar (tElemento x, Lista L);
   posicion Siguiente (posicion p, Lista L);
   posicion Anterior (posicion p, Lista L);
   posicion Primera (Lista L);
   posicion Fin (Lista L);
   void DestruirLista (Lista L);
#endif


/*-----------------------------------------------*/
/* lisdoble.c                                    */
/*-----------------------------------------------*/

#include <stdlib.h>
#include "error.h"
#include "lisdoble.h"

/* Operaciones internas */

int Igual (tElemento a, tElemento b)
/* Compara dos datos, a y b, de tipo tElemento y
   devuelve 0 si son distintos o cualquier otro
   valor si son iguales. */
{
   if (a == b)
```

```
      return 1;
   else
      return 0;
}

/* Operaciones públicas */

Lista CrearLista ()
{
   Lista L;

   L = (Lista) malloc(sizeof(tipoNodo));
   if (L == NULL)
      ERROR("CrearLista: No hay memoria");
   L->ant = L;
   L->sig = L;
   return L;
}

void DestruirLista (Lista L)
{
   posicion p;

   while (L->sig != L)
   {
      p = L->sig;
      L->sig = p->sig;
      free(p);
   }
   free(L);
}

void Insertar(tElemento x,posicion p,Lista L)
{
   posicion q;

   q = (posicion) malloc(sizeof(tipoNodo));
   if (q == NULL)
      ERROR("Insertar: No hay memoria");
   q->elemento = x;
   q->ant = p;
   q->sig = p->sig;
   p->sig->ant = q;
   p->sig = q;
```

```
}

void Eliminar (posicion p, Lista L)
{
   posicion q;

   if (p == L->ant)
      ERROR("Eliminar: Elemento inexistente");
   q = p->sig;
   p->sig = q->sig;
   q->sig->ant = p;
   free(q);
}

tElemento Recuperar (posicion p, Lista L)
{
   if (p == L->ant)
      ERROR("Recuperar: Elemento inexistente");
   return p->sig->elemento;
}

posicion Buscar (tElemento x, Lista L)
{
   posicion p;
   int encontrado;

   p = L;
   encontrado = 0;
   while (p != L->ant && !encontrado)
      if (Igual(p->sig->elemento, x))
         encontrado = 1;
      else
         p = p->sig;
   return p;
}

posicion Siguiente (posicion p, Lista L)
{
   if (p == L->ant)
      ERROR("Siguiente: Posición incorrecta");
   return p->sig;
}

posicion Anterior (posicion p, Lista L)
```

```
{
   if (p == L)
      ERROR("Anterior: Posición incorrecta");
   return p->ant;
}

posicion Primera (Lista L)
{
   return L;
}

posicion Fin (Lista L)
{
   return L->ant;
}
```

4.5. TAD LISTA CIRCULAR

Existen ciertos problemas en los que el TAD *lista* (lineal) no es el más apropiado para resolverlos. Se trata de problemas en los que el primer y el último elemento de la lista no se distinguen del resto, todos ellos forman una secuencia, pero no están claramente definidos los extremos, ya que cada elemento está precedido de uno anterior y seguido de otro posterior. Para este tipo de problemas es adecuado un nuevo TAD que denominaremos TAD *lista circular*.

Las operaciones de este TAD pueden ser prácticamente las mismas que las del TAD *lista*. La diferencia radica en que una lista circular no tiene extremos, es decir, no existen las posiciones que hemos llamado *primera* y *fin*, en consecuencia, las operaciones que devuelven estas posiciones no forman parte del TAD. No obstante, para recorrer la lista en un sentido o en el otro y poder situarnos en cualquier elemento, necesitamos una operación que devuelva una posición desde la cual comenzar el recorrido. A esta operación que nos permite inicializar una variable de tipo *posición* la llamaremos *LCCrearPos()*. Según estas consideraciones, definiremos el TAD *lista circular* mediante la siguiente especificación:

Definición:

Una lista circular es una secuencia de elementos de un mismo tipo, en la que todos tienen un predecesor y un sucesor. La *longitud* o *tamaño* de la lista coincide con el número de elementos que la forman; si es 0, entonces la lista está vacía. Una lista circular de

longitud n se puede representar de la forma

$$L = (a_1, a_2, \ldots, a_n, a_1)$$

donde repetimos a_1 después de a_n para indicar que el elemento que sigue a a_n es a_1 y el anterior a éste es a_n.

Definimos una *posición* como el lugar que ocupa un elemento en la lista. La constante *POS_NULA* denota una posición inexistente.

Operaciones:

LCircular CrearLCircular ()
Postcondiciones: Crea y devuelve una lista circular vacía.

void LCInsertar (tElemento x, posicion p, LCircular L)
Precondiciones: $L = ()$ y p es irrelevante o bien,
$L = (a_1, a_2, \ldots, a_n, a_1)$ y $1 \leq p \leq n$
Postcondiciones: Si $L = ()$, entonces $L = (x, x)$ (lista circular con un único elemento); en caso contrario,

$$L = (a_1, \ldots, a_{p-1}, x, a_p, \ldots, a_n, a_1)$$

void LCEliminar (posicion p, LCircular L)
Precondiciones: $L = (a_1, a_2, \ldots, a_n, a_1)$
$1 \leq p \leq n$
Postcondiciones: $L = (a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n, a_1)$

tElemento LCRecuperar (posicion p, LCircular L)
Precondiciones: $L = (a_1, a_2, \ldots, a_n, a_1)$
$1 \leq p \leq n$
Postcondiciones: Devuelve a_p, el elemento que ocupa la posición p en la lista L.

posicion LCBuscar (tElemento x, LCircular L)
Precondiciones: L está creada.
Postcondiciones: Devuelve la posición de una ocurrencia de x en al lista L. Si x no pertenece a L, devuelve `POS_NULA`.

posicion LCCrearPos (LCircular L)
Precondiciones: L está creada.
Postcondiciones: Devuelve una posición indeterminada de la lista L. Si la lista está vacía devuelve `POS_NULA`. Esta operación se utilizará para inicializar una variable de tipo *posicion*.

posicion LCSiguiente (posicion p, LCircular L)
Precondiciones: $L = (a_1, a_2, \ldots, a_n, a_1)$
$1 \leq p \leq n$
Postcondiciones: Devuelve la posición siguiente a p en L.

posicion LCAnterior (posicion p, LCircular L)
Precondiciones: $L = (a_1, a_2, \ldots, a_n, a_1)$
$1 \leq p \leq n$
Postcondiciones: Devuelve la posición anterior a p en L.

void DestruirLCircular (LCircular L)
Precondiciones: L está creada.
Postcondiciones: Libera la memoria ocupada por la lista L. Para usarla otra vez se debe volver a crear con la operación *CrearLCircular()*.

La estructura de datos que mejor se adapta a este TAD es una lista enlazada circular, la cual se forma enlazando el último nodo de la cadena con el primero. De esta forma, cualquier nodo de la estructura está seguido (y por tanto precedido) de otro, por lo que desde un nodo cualquiera de la misma se puede acceder a cualquier otro, simplemente hay que seguir los enlaces hasta alcanzar el nodo deseado. Esta característica permite representar la posición de un elemento de una lista circular como un puntero al nodo anterior, sin necesidad de añadir a la estructura ningún nodo extra, como es el nodo cabecera en el caso de las listas lineales.

4.6. Aplicaciones de las listas

Como ocurría en el caso de la colas, el concepto representado por el TAD lista, coincide plenamente con el concepto de una lista en la vida cotidiana. Hay numerosos ejemplos: la lista de la compra, los 40 principales, ingredientes de una comida, etc. A diferencia de las colas, como ya se ha indicado previamente, las listas permiten realizar accesos, adiciones y borrados en cualquier posición de la misma, igual que en la vida real. Por tanto, en cualquier aplicación informática que intente modelizar una "lista real", surge el TAD lista.

5

FICHEROS

5.1. Introducción

En la sociedad actual, el hombre tiene cada vez una mayor necesidad de consultar una gran cantidad de información para poder desarrollar sus actividades diarias. La creciente avalancha de información hace necesario que ésta tenga que ser almacenada y organizada cuidadosamente para poder acceder a la misma en el futuro.

Según lo visto hasta ahora, la única forma que el ordenador tiene de almacenar información es utilizando variables. El problema de la utilización de variables es que su ámbito de vida se limita (en el caso más amplio) a la ejecución del programa o subprograma en que fueron declaradas; ello implica que cuando el programa o subprograma finaliza su ejecución, o cuando el ordenador se apaga, por ejemplo, desaparecen. En el mundo real los datos no desaparecen al acabar la ejecución del programa o subprograma o al apagar el ordenador. Dado que nuestro objetivo es modelizar la realidad, no parecen ser las variables la opción más adecuada. Por lo tanto, si deseamos tener acceso en cualquier momento a una determinada información, necesitaremos almacenarla sobre un soporte físico que sea capaz de mantenerla de forma permanente. Dicho de otra forma, necesitamos ampliar el ámbito de nuestras "variables" más allá de la mera ejecución del programa o subprograma en el que fueron declaradas o definidas.

Otro problema adicional es el escaso tamaño que suele tener la memoria principal, lo que provoca que, frecuentemente, no podamos al-

macenar al mismo tiempo todos nuestros datos en ella. Son estos dos problemas, (el ámbito temporal de las variables, y la limitación de espacio de la memoria principal), las que han provocado la inclusión en el presente libro de una estructura de datos diseñada para almacenar información en memoria secundaria, esto es, ficheros. Estudiaremos aquí únicamente los ficheros secuenciales, ya que el motivo de la presente publicación son las estructuras de datos lineales.

Después de lo dicho, resulta obvio que esta información, con pretensiones de cierta permanencia, y, al mismo tiempo, demasiado amplia para la memoria principal, no va a almacenarse en ésta sino que debe ser guardada en disquetes, discos duros, cintas magnéticas, CD-ROMs, etc., capaces de almacenar información de un cierto tamaño de forma permanente. A estos dispositivos se les denomina dispositivos de almacenamiento secundario.

Si comparamos estos dispositivos con los soportes de memoria principal conocidos, sus mayores ventajas consisten en su capacidad de almacenamiento por unidad monetaria (vamos, que son mas baratos por byte almacenado) y que, como ya hemos dicho, pueden almacenar la información de una manera permanente. Si nuestro análisis acabase aquí, podríamos plantearnos seriamente el porqué de la existencia de la memoria principal (volátil, mucho más cara por byte que la secundaria y en consecuencia, mucho más pequeña). Parecería mucho más razonable utilizar la memoria secundaria para almacenar y manipular variables. La respuesta a nuestro pequeño dilema es fácil: La memoria principal es mucho más rápida que la memoria secundaria; ello justifica que un soporte tan caro y consecuentemente tan pequeño esté presente en todos y cada uno de los ordenadores utilizados habitualmente (arquitectura de Von Newman). Cuando decimos mucho más rápida, queremos decir “muchísimo más rápida”. Es habitual hablar de tiempos de acceso a los discos duros, por ejemplo (memoria secundaria bastante rápida), del orden de milisegundos, mientras que en el caso de memoria principal tiende a hablarse de nanosegundos. Así pues, hablamos de diferencias en rapidez de un millón a uno. Ello explica la existencia de la memoria principal. Sin su tecnología nuestros ordenadores serían tan lentos que no serían operativos en el sentido actual de la palabra. La interactividad de la que disfrutamos hoy día sería simple ciencia-ficción. Así pues, la tecnología que nos ofrece la memoria principal, aunque cara por unidad de almacenamiento, es, simplemente, imprescindible. Generalizando, podríamos decir, que en un programa “normal”, los datos de entrada se obtendrán de dispositivos de almacenamiento secundario, serán almacenados en memoria principal a través de variables, el programa manipulará dichas variables, y los resultados obtenidos (salida) serán almacenados de nuevo en memoria secundaria.

5.2. Conceptos básicos

El concepto intuitivo de lo que es un fichero, es conocido por todos. Cuando, por ejemplo, buscamos un libro en una biblioteca, consultamos el fichero de libros, cuando queremos acceder a los datos de un trabajador de una empresa acudimos al fichero de personal, etc. Básicamente la idea intuitiva es que un fichero es un lugar, en el que, con una cierta organización, se encuentra información sobre un tema en concreto. Obviamente aquí desarrollaremos el concepto de fichero desde un punto de vista informático, es decir, nos plantearemos modelizar el concepto de fichero que hay en el mundo real, al mundo de la informática.

Para desarrollar el concepto "informático" de fichero vamos a definir una serie de conceptos que nos resultaran muy útiles.

Registro: Un registro es una colección de información relativa a una entidad particular. Por lo tanto, el registro contendrá todos aquellos **campos** lógicamente relacionados, referentes a la misma entidad, y que pueden ser tratados conceptualmente al mismo tiempo por un programa. Por ejemplo, la información de un alumno de la carrera de Informática en particular, puede tener los campos *NIF*, *nombre*, *apellido1*, *apellido2*, *curso*, etc. Así pues, un registro es un conjunto de campos referentes a una entidad en particular y un campo será un identificador que representa un determinado aspecto (información concreta y diferenciada del resto) o item de esa entidad en concreto.

Fichero: Un fichero es un conjunto de registros homogéneos, almacenados en un soporte externo, que presentan entre sí una relación lógica y que pueden ser consultados individualmente de forma iterativa y sistemática.

Un ejemplo de fichero podría ser el fichero de alumnos de la asignatura *Estructuras de Datos I* (figura **??**), compuesto por diferentes registros, cada uno de los cuales correspondería a un alumno (entidad) en particular, que a su vez estaría compuesto por diferentes campos, cada uno de los cuales representaría una información concreta y diferenciada del resto para esa entidad particular (nombre, dirección, etc).

Aquí es importante destacar de mayor a menor (en orden inverso de inclusión) los conceptos de fichero, registro y campo.

Fichero externo vs. fichero interno: El primero es la estructura de datos utilizada para almacenar información en memoria secundaria. Para poder procesar dicha información en un programa, es

Código	Nombre	Dirección	Teléfono	Edad
C1	Lizondo, José Antonio	Arenas, 45	956432816	18
C2	García, Fernando	Liebre, 25	956186349	21
C3	Martínez, Evaristo	Teruel, 24	679154682	19
C4	Abbasi, Rafael	Idiomas, 10	954824735	17
C5	Rumeu, Ana	Percebe, 13	689457614	20
C6	Santos, César	Mesones, 37	956152981	19
C7	García, Carmen	Larga, 62	629295541	21
⋮	⋮	⋮	⋮	⋮
C799	Armas, Lola	Toledo, 129	954567812	18
C800	Soldevilla, Dolores	Lagasca, 5	956638218	20

Figura 5.1: Fichero de alumnos.

necesario utilizar una variable que represente esta estructura en memoria principal. A dicha variable se le denomina **fichero interno**. Como cualquier otra variable, lleva asociado un tipo, y es posible realizar sobre ella una serie de operaciones. Será necesario establecer un "enlace" entre los ficheros externo e interno. Dicho enlace se establecerá durante la operación de apertura del fichero.

Organización: La organización de un fichero es la forma particular en que los datos (los registros) son almacenados en el soporte de almacenamiento. El tipo de organización se decide durante la creación del fichero. El concepto de organización es absolutamente básico en su asociación al concepto de fichero. Nadie pensaría, por ejemplo, que una habitación llena de papeles por el suelo, fuera remotamente un fichero (ni informático ni no informático). Existen varios tipos de organizaciones, en este capítulo veremos la *organización secuencial*, *organización directa* y *organización secuencial indexada*.

Organización secuencial: Este tipo de organización se caracteriza porque los registros que forman un fichero que la utiliza (fichero secuencial), se escriben o se graban lógicamente (no tiene por qué ser físicamente) en posiciones contiguas en la misma secuencia u orden en que han sido introducidos. El concepto de organización es más lógico que físico. A un nivel puramente lógico los diferentes registros están colocados secuencialmente, y para acceder al registro con posición n (al n-ésimo registro del fichero) será necesario haber accedido previamente a los $n-1$ registros anteriores (acceso secuencial al fichero, concepto que surgirá próximamente).

Organización directa: Un fichero con organización directa se caracteriza porque es posible acceder a cualquier registro directamente mediante la posición que ocupa dentro del fichero. La posición en la que se almacena un registro viene determinada por los valores de los campos del mismo mediante una función matemática (hashing). Debido a esto es posible que los registros no estén almacenados en posiciones contiguas y existan huecos entre ellos.

Organización secuencial indexada: En un fichero con esta organización los registros se organizan de forma secuencial, pero además se utiliza un índice que nos permite obtener la ubicación de un registro dentro del fichero. Esto permite localizar un registro sin tener que leer previamente todos los que le preceden (a diferencia de la organización secuencial) ni conocer a priori la posición exacta del registro dentro del fichero (al contrario que la organización directa).

Denominación de ficheros : La denominación de un fichero procede de su organización. De este modo llamamos *fichero secuencial* a un fichero cuya organización es secuencial, *fichero directo* o *relativo* si su organización es directa y análogamente *fichero secuencial indexado*.

Modo de acceso: El modo de acceso a un fichero es la manera de acceder a los registros para extraer información (leer) que pueda ser procesada posteriormente, o para grabar información nueva (escribir/añadir) en el fichero.

Acceso secuencial: En este modo de acceso, la única forma de acceder al registro con posición n es recorrer previamente los $n-1$ anteriores. La idea básica consiste en que la información necesaria para encontrar el registro con posición n no está disponible si no hemos recorrido (leído) previamente el registro $n-1$. Si no hemos accedido (leído) al registro de posición n, simplemente, no sabemos dónde está el de posición $n+1$, así de simple.

Acceso directo: : El acceso a un registro se hace por la posición que ocupa dentro del fichero. Esto implica que se puede acceder a un registro cualquiera sin que ello requiera acceder a los precedentes.

Soporte de almacenamiento: Dispositivo sobre el que está almacenada físicamente la información.

Soportes secuenciales : En ellos los bloques de datos están dispuestos físicamente uno a continuación de otro de forma lineal. Esto implica que para acceder a un bloque en concreto es necesario recorrer previamente todos los anteriores a dicho bloque.

Soportes direccionables : Permiten acceder directamente a un bloque por la posición que ocupa en el soporte de almacenamiento.

Operaciones con ficheros vs. organización : Las típicas operaciones de ficheros (alta, baja, consulta y modificación) se realizarán de diferente forma en función de la organización.

- **Organización secuencial:**
  - Alta o inserción: Sólo es posible añadir detrás del último registro del fichero.
  - Baja o eliminación: No es posible borrar físicamente un registro. Hay que marcar los registros (borrado lógico) y periódicamente procesar el fichero para eliminar físicamente los registros marcados.
  - Modificación y consulta: Requiere una lectura previa de los registros anteriores al que se quiere consultar o modificar.
- **Organización directa:**
  - Alta o inserción: Se puede insertar un nuevo registro en cualquier posición del fichero.
  - Baja o eliminación: Se puede realizar borrados lógicos en cualquier posición del fichero.
  - Modificación: Se puede reescribir cualquier registro del fichero accediendo por su posición.
  - Consulta: Se puede consultar cualquier registro conociendo su posición.
- **Organización secuencial indexada:**
  - Alta o inserción: La adición de un registro al fichero implicará, en la mayoría de los casos, una reorganización del índice del fichero.
  - Baja o eliminación: Se realizan borrados lógicos. Cada eliminación puede provocar una reorganización del índice del fichero.
  - Modificación: Se puede reescribir el contenido de un registro, siempre que no cambiemos el valor de su campo clave (campo por el que está indexado el fichero).
  - Consulta: Se accede a los registros por el valor del campo clave. La lectura puede ser secuencial siguiendo el orden de la clave o directa a través de su valor.

5.3. Especificación del TAD Fichero Secuencial

Definición:

Un fichero interno secuencial es un conjunto de registros homogéneos, que presentan entre sí una relación lógica y en el que su organización sólo permite el modo de acceso secuencial.

Operaciones:

*FichSec AbrirFichSec (const char *nomfich, char modo)*

Precondiciones: El parámetro *nomfich* es un puntero a una cadena de caracteres que contiene el nombre de un fichero externo secuencial y *modo* es un carácter que indica cómo se debe abrir el fichero. Los modos de apertura son los siguientes:

- **'I' (inicio):** El fichero está en disposición de escribir el primer registro. Se puede añadir. No se puede leer ni modificar.
- **'C' (consulta):** El fichero está en disposición de leer el primer registro. Se puede leer. No se puede añadir ni modificar.
- **'A' (actualización):** El fichero está en disposición de leer el primer registro. Se puede leer y modificar. No se puede añadir.
- **'E' (extensión):** El fichero está en disposición de escribir a continuación del último registro. Se puede añadir. No se puede leer ni modificar.

Si el modo de apertura es 'C', 'A' o 'E', el fichero debe existir.

Postcondiciones: Abre el fichero externo secuencial *nomfich* y devuelve un fichero interno secuencial al que queda enlazado. Si el modo de apertura es 'I' y el fichero externo no existe, lo crea; pero si existe, borra todos los datos que contenga.

void CerrarFichSec (FichSec fsec)

Precondiciones: El fichero interno *fsec* está abierto.

Postcondiciones: Cierra el fichero interno secuencial *fsec*. Toda la información que todavía se encuentre en el buffer del disco se escribe en el fichero externo asociado a *fsec*.

registro LeerFichSec (FichSec fsec)

Precondiciones: El fichero *fsec* está abierto en un modo compatible con la lectura ('C' y 'A'). No se encuentra en la posición de fin de fichero.

Postcondiciones: Devuelve una variable de tipo registro con el

contenido de la posición de *fsec* en que se encuentre y actualiza dicha posición al valor siguiente.

void AñadirFichSec (FichSec fsec, registro reg)
Precondiciones: El fichero *fsec* está abierto en un modo compatible con la escritura ('I' y 'E').
Postcondiciones: Escribe el registro *reg* al final del fichero *fsec*.

void ModificarFichSec (FichSec fsec, registro reg)
Precondiciones: El fichero *fsec* está abierto en modo 'A'. El registro *reg* debe haber sido leído previamente.
Postcondiciones: Sustituye el valor anterior de *reg* por un nuevo valor del mismo. Si se añade un registro nulo, equivaldría a un borrado.

int FinFichSec (FichSec fsec)
Precondiciones: El fichero *fsec* está abierto.
Postcondiciones: Devuelve 0 si no se ha llegado a la posición de fin de fichero y 1 en caso contrario.

int EstadoFichSec (FichSec fsec)
Postcondiciones: Nos devuelve un entero que indica si el fichero, tras la última operación realizada con él, se encuentra en perfecto estado. (0, OK; un valor distinto, diferentes tipos de error). Debe utilizarse después de cada operación si se desea estar seguro de que no hubo ningún tipo de error.

5.4. Implementación del TAD Fichero Secuencial

```
/*-----------------------------------------------*/
/* Fichsec.h                                     */
/*-----------------------------------------------*/

#ifndef _REGISTRO_
#define _REGISTRO_

   typedef struct {
              char campo1;
              float campo2;
              long campo3;
              int borrado;
            } registro; /* Por ejemplo */
#endif
```

```
#ifndef _FICHSEC_
#define _FICHSEC_
   typedef struct {
              FILE *fichero;
              char modoAp;
            } tipoFichSec;
   typedef tipoFichSec *FichSec;

   FichSec AbrirFichSec(const char *nomfich,
                        char modo);
   void CerrarFichSec (FichSec fsec);
   registro LeerFichSec (FichSec fsec);
   void AnyadirFichSec (FichSec fsec, registro reg);
   void ModificarFichSec (FichSec fsec, registro reg);
   int FinFichSec (FichSec fsec);
   int EstadoFichSec (FichSec fsec);
#endif

/*-------------------------------------------------*/
/* Fichsec.c                                       */
/*-------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "error.h"
#include "fichsec.h"

/* Operaciones privadas */

static int ExisteFichDir (const char *nomfich)
{
   FILE *f;

   if ((f = fopen(nomfich, "rb")) == NULL)
      return 0;
   else {
      fclose(f);
      return 1;
   }
}

/* Operaciones públicas */
```

```
FichSec AbrirFichSec (const char *nomfich, char modo)
{
   FichSec fsec;

   fsec = (tipoFichSec *) malloc(sizeof(tipoFichSec));
   if (fsec == NULL)
      ERROR("AbrirFichSec: Memoria insuficiente");

   if ((modo == 'C' || modo == 'A' || modo == 'E') &&
       !ExisteFichSec(nomfich))
      ERROR("AbrirFichSec: Fichero inexistente. "
            "Modo incompatible");

   switch (modo)
   {
      case 'I': /* Crear vacío para escritura */
                fsec->fichero = fopen(nomfich,"wb");
                break;
      case 'C': /* Abrir para lectura */
                fsec->fichero = fopen(nomfich,"rb");
                break;
      case 'A': /* Abrir para lectura/escritura */
                fsec->fichero = fopen(nomfich,"rb+");
                break;
      case 'E': /* Abrir para adición */
                fsec->fichero = fopen(nomfich,"ab");
                break;
      default : ERROR("AbrirFichSec: "
                      "Modo de apertura incorrecto");
   }
   fsec->modoAp = modo;
   return fsec;
}

void CerrarFichSec (FichSec fsec)
{
   fclose(fsec->fichero);
   free(fsec);
}

registro LeerFichSec (FichSec fsec)
{
   registro reg;
```

```
   if (fsec->modoAp != 'C' && fsec->modoAp != 'A')
      ERROR("LeerFichSec: Modo incompatible");
   if (feof(fsec->fichero))
      ERROR("LeerFichSec: Fin de fichero");

   fread(&reg,sizeof(registro),1,fsec->fichero);
   return reg;
}

void AnyadirFichSec (FichSec fsec, registro reg)
{
   if (fsec->modoAp != 'I' && fsec->modoAp != 'E')
      ERROR("AnyadirFichSec: Modo incompatible");

   fwrite(&reg,sizeof(registro),1,fsec->fichero);
}

void ModificarFichSec (FichSec fsec, registro reg)
{
   if (fsec->modoAp != 'A')
      ERROR("ModificarFichSec: Modo incompatible");

   if (fseek(fsec->fichero,-1 * sizeof(registro),
             SEEK_CUR) == 0)
      fwrite(&reg,sizeof(registro),1,fsec->fichero);
}

int FinFichSec (FichSec fsec)
{
   return feof(fsec->fichero);
}

int EstadoFichSec (FichSec fsec)
{
   return ferror(fsec->fichero);
}
```

5.5. Especificación del TAD Fichero Directo

Definición:

Un fichero interno directo o relativo puede contener hasta un número máximo (N) de registros homogéneos, a cada uno de los cuales se puede acceder de modo directo por la posición relativa que

ocupa dentro del fichero. Los registros constan de un campo lógico que se utiliza para marcar los registros como borrados o activos. Un registro nulo tiene en este campo el valor `BORRADO`, en caso contrario este valor será `ACTIVO`.

Operaciones:

*FichDir AbrirFichDir (const char *nomfich, char modo, long N)*

Precondiciones: El parámetro *nomfich* es un puntero a una cadena de caracteres que contiene el nombre de un fichero externo directo. El parámetro *modo* es un carácter que indica cómo se debe abrir el fichero. Los modos de apertura son los siguientes:

'I' (inicio): El fichero se crea nuevo para lectura y escritura.

'C' (consulta): El fichero se abre sólo para lectura.

'A' (actualización): El fichero se abre para lectura y escritura. Se pueden leer y escribir registros en cualquier posición.

El parámetro *N* sólo se utiliza si el modo de apertura es 'I'.

Postcondiciones: Abre el fichero externo directo *nomfich* y devuelve un fichero interno directo al que queda enlazado. Si el modo de apertura es 'I', lo crea con *N* posiciones vacías. Si previamente existía, primero lo borra. La posición activa para la primera lectura secuencial queda establecida en el primer registro ocupado del fichero. Si el fichero está vacío, la posición activa es fin de fichero.

void CerrarFichDir (FichDir fdir)

Precondiciones: El fichero interno *fdir* está abierto.

Postcondiciones: Cierra el fichero interno directo *fdir*. Toda la información que todavía se encuentre en el buffer del disco se escribe en el fichero externo asociado a *fdir*.

int PosLibreFichDir (FichDir fdir, long p)

Precondiciones: El fichero *fdir* está abierto. La posición *p* está en el rango $[1, N]$, donde N es el número de posiciones del fichero.

Postcondiciones: Devuelve el estado del registro almacenado en la posición *p* del fichero: 0 activo, 1 borrado o posición libre.

registro LeerFichDir (FichDir fdir, long p)

Precondiciones: El fichero *fdir* está abierto. La posición *p* está en el rango $[1, N]$, donde N es el número de posiciones del fichero. La posición *p* no está libre.

Postcondiciones: Devuelve una variable de tipo *registro* con el contenido de la posición *p* de *fdir*. La posición activa avanza a la primera ocupada posterior a *p* o fin de fichero si no existe.

void EscribirFichDir (FichDir fdir, registro reg, long p)
Precondiciones: El fichero *fdir* está abierto en modo de inicio ('I') o actualización ('A'). La posición *p* está en el rango $[1, N]$, donde N es el número de posiciones del fichero.
Postcondiciones: Escribe el registro *reg* en la posición *p* del fichero *fdir*. Si la posición *p* está ocupada, sobrescribe el contenido con el nuevo registro *reg*, si este es el registro nulo equivale a un borrado. La posición activa queda establecida en la siguiente ocupada o fin de fichero si ésta no existe.

registro LeerSecFichDir (FichDir fdir)
Precondiciones: El fichero *fdir* está abierto. La posición activa no es fin de fichero.
Postcondiciones: Devuelve una variable de tipo registro con el contenido de la posición activa de *fdir* y actualiza dicha posición a la siguiente posición ocupada o fin de fichero, si ésta no existe.

int FinFichDir (FichDir fdir)
Precondiciones: El fichero *fdir* está abierto.
Postcondiciones: Devuelve 0 si no se ha llegado a la posición de fin de fichero y 1 en caso contrario.

long TamanioFichDir (FichDir fdir)
Precondiciones: El fichero *fdir* está abierto.
Postcondiciones: Devuelve el número de posiciones del fichero.

int EstadoFichDir (FichDir fdir)
Postcondiciones: Nos devuelve un entero que indica si el fichero, tras la última operación realizada con él, se encuentra en perfecto estado. (0, OK; un valor distinto, diferentes tipos de error). Debe utilizarse después de cada operación si se desea estar seguro de que no hubo ningún tipo de error.

5.6. Implementación del TAD Fichero Directo

```
/*-----------------------------------------------*/
/* FichDir.h                                     */
/*-----------------------------------------------*/

#ifndef _REGISTRO_
#define _REGISTRO_
#define BORRADO 0
#define ACTIVO 1
```

```
   typedef struct {
              char campo1;
              float campo2;
              long campo3;
              int borrado;
            } registro; /* Por ejemplo */
#endif

#ifndef _FICHDIR_
#define _FICHDIR_
   typedef struct {
              FILE *fichero;
              char modoAp;
              long tama;
              int *activo;
              long posact;
            } tipoFichDir;
   typedef tipoFichDir *FichDir;

   FichDir AbrirFichDir(const char *nomfich,char modo,
                        long N);
   void CerrarFichDir (FichDir fdir);
   int PosLibreFichDir (FichDir fdir, long p);
   registro LeerFichDir (FichDir fdir, long p);
   void EscribirFichDir (FichDir fdir, registro reg,
                         long p);
   registro LeerSecFichDir (FichDir fdir);
   int FinFichDir (FichDir fdir);
   long TamanioFichDir (FichDir fdir);
   int EstadoFichDir (FichDir fdir);
#endif



/*-----------------------------------------------*/
/* FichDir.c                                     */
/*-----------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include "error.h"
#include "fichdir.h"

/* Operaciones privadas */
```

```
static int ExisteFichDir (const char *nomfich)
{
   FILE *f;

   if ((f = fopen(nomfich, "rb")) == NULL)
      return 0;
   else {
      fclose(f);
      return 1;
   }
}

/* Operaciones públicas */

FichDir AbrirFichDir(const char *nomfich, char modo,
                     long N)
{
   FichDir fdir;
   long i;
   registro reg;

   fdir = (tipoFichDir *) malloc(sizeof(tipoFichDir));
   if (fdir == NULL)
      ERROR("AbrirFichDir: Memoria insuficiente");
   if ((modo == 'C' || modo == 'A') &&
       !ExisteFichDir(nomfich))
      ERROR("AbrirFichDir: Fichero inexistente. "
            "Modo incompatible");

   switch (modo)
   {
      case 'I': /*Crear vacío para lectura/escritura*/
                fdir->fichero = fopen(nomfich,"wb+");
                break;
      case 'C': /* Abrir para lectura */
                fdir->fichero = fopen(nomfich,"rb");
                break;
      case 'A': /* Abrir para lectura/escritura */
                fdir->fichero = fopen(nomfich,"rb+");
                break;
      default: ERROR("AbrirFichDir: "
                     "Modo de apertura incorrecto");
   }
```

```
   fdir->modoAp = modo;
   if (modo == 'I') {
      fdir->tama = N;
      fdir->activo = (int *) calloc(N+1, sizeof(int));
      if (fdir->activo == NULL)
         ERROR("AbrirFichDir: Memoria insuficiente");
      /* Escribir N registros borrados */
      reg.borrado = BORRADO;
      for (i = 0; i < fdir->tama; i++)
        fwrite(&reg,sizeof(registro),1,fdir->fichero);
   }
   else { /* modo 'C' o 'A' */
      fseek(fdir->fichero, 0, SEEK_END);
      fdir->tama = ftell(fdir->fichero) /
                   sizeof(registro);
      fseek(fdir->fichero, 0, SEEK_SET);
      fdir->activo = (int *)
                     calloc(fdir->tama+1,sizeof(int));
      if (fdir->activo == NULL)
         ERROR("AbrirFichDir: Memoria insuficiente");
      /* Leer estado registros */
      for (i = 1; i <= fdir->tama; i++) {
         fread(&reg,sizeof(registro),1,fdir->fichero);
         fdir->activo[i] = reg.borrado;
      }
   }
   fdir -> posact = 1;
   return fdir;
}

void CerrarFichDir (FichDir fdir)
{
   fclose(fdir->fichero);
   free(fdir->activo);
   free(fdir);
}

int PosLibreFichDir (FichDir fdir, long p)
{
   if (p < 1 || p > fdir->tama)
      ERROR("PosLibreFichDir: Posición no válida");
   return !fdir->activo[p];
}
```

```
registro LeerFichDir (FichDir fdir, long p)
{
   registro reg;

   if (p < 1 || p > fdir->tama)
      ERROR("LeerFichDir: Posición no válida");
   if (fdir->activo[p] == BORRADO)
      ERROR("LeerFichDir: Posición libre");

   fseek(fdir->fichero, (p-1)*sizeof(reg), SEEK_SET);
   fread(&reg,sizeof(registro),1,fdir->fichero);
   fdir->posact = p+1;
   while (fdir->activo[fdir->posact] == BORRADO &&
          fdir->posact <= fdir->tama)
      fdir->posact++;
   return reg;
}

void EscribirFichDir(FichDir fdir,registro reg,long p)
{
   if (fdir->modoAp == 'C')
      ERROR("EscribirFichDir: Modo incompatible");
   if (p < 1 || p > fdir->tama)
      ERROR("EscribirFichDir: Posición no válida");

   fseek(fdir->fichero, (p-1)*sizeof(reg), SEEK_SET);
   fwrite(&reg,sizeof(registro),1,fdir->fichero);
   fdir->activo[p] = reg.borrado;
   fdir->posact = p+1;
   while (fdir->activo[fdir->posact] == BORRADO &&
          fdir->posact <= fdir->tama)
      fdir->posact++;
}

registro LeerSecFichDir (FichDir fdir)
{
   registro reg;
   if (fdir->posact > fdir->tama)
      ERROR("LeerSecFichDir: Fin de fichero");

   fseek(fdir->fichero,
         (fdir->posact-1)*sizeof(reg), SEEK_SET);
   fread(&reg,sizeof(registro),1,fdir->fichero);
   fdir->posact++;
```

```
   while (fdir->activo[fdir->posact] == BORRADO &&
          fdir->posact <= fdir->tama)
      fdir->posact++;
   return reg;
}

int FinFichDir (FichDir fdir)
{
   return (fdir->posact > fdir->tama);
}

long TamanioFichDir (FichDir fdir)
{
   return fdir->tama;
}

int EstadoFichDir (FichDir fdir)
{
   return ferror(fdir->fichero);
}
```

5.7. Especificación del TAD Fichero Secuencial Indexado

Definición:

Un fichero interno secuencial indexado contiene un conjunto de registros homogéneos a los cuales se puede acceder en modo secuencial, siguiendo el orden del campo clave, o directo utilizando dicho campo.

Operaciones:

*FichSecInd AbrirFichSecInd (const char *nomfich, char modo)*

Precondiciones: El parámetro *nomfich* es un puntero a una cadena de caracteres que contiene el nombre de un fichero externo secuencial indexado.
El parámetro *modo* es un carácter que indica cómo se debe abrir el fichero. Los modos de apertura son los siguientes:

'I' (inicio): El fichero se crea nuevo para lectura y escritura.

'C' (consulta): El fichero se abre sólo para lectura.

'A' (actualización): El fichero se abre para lectura y escritura.

Postcondiciones: Abre el fichero externo secuencial indexado *nomfich* y devuelve un fichero interno secuencial indexado al que queda enlazado. Si el modo de apertura es 'I', crea un fichero vacío. Si previamente existía, primero lo borra. La posición activa para la primera lectura secuencial queda establecida en el primer registro (por orden de clave) del fichero. Si el fichero está vacío, la posición activa es fin de fichero.

void CerrarFichSecInd (FichSecInd fsecind)

Precondiciones: El fichero interno *fsecind* está abierto.
Postcondiciones: Cierra el fichero interno secuencial indexado *fsecind*. Toda la información que todavía se encuentre en el buffer del disco se escribe en el fichero externo asociado a *fsecind*.

*int LeerFichSecInd (FichSecInd fsecind, tclave c, registro *reg)*

Precondiciones: El fichero *fsecind* está abierto.
Postcondiciones: Localiza en *fsecind* el registro con clave *c* y devuelve 1 si lo encuentra ó 0 en caso contrario. Si *c* existe, en **reg* devuelve el contenido del registro de *fsecind* con valor *c* en el campo clave. La posición activa pasa a ser la del registro con clave siguiente a *c* o fin de fichero.

void EscribirFichSecInd (FichSecInd fsecind, registro reg)

Precondiciones: El fichero *fsecind* está abierto en modo de inicio ('I') o actualización ('A').
Postcondiciones: Escribe el registro *reg* en el fichero *fsecind*. Si ya existe un registro con la misma clave que *reg*, lo sobrescribe. La posición activa queda establecida en la del registro con clave siguiente a la de *reg* o fin de fichero.

int EliminarFichSecInd (FichSecInd fsecind, tclave c)

Precondiciones: El fichero *fsecind* está abierto en modo 'I' o 'A'.
Postcondiciones: Si existe el registro con clave *c* en el fichero *fsecind*, lo elimina y devuelve 1. En caso contrario simplemente devuelve 0. La posición activa queda establecida en el registro de clave siguiente a *c* o fin de fichero.

registro PrimeroSecFichSecInd (FichSecInd fsecind)

Precondiciones: El fichero *fsecind* está abierto y no está vacío.
Postcondiciones: Devuelve el registro de *fsecind* que tiene la primera clave y actualiza la posición activa a la del registro con la clave siguiente o fin de fichero.

registro LeerSecFichSecInd (FichSecInd fsecind)

Precondiciones: El fichero *fsecind* está abierto. La posición activa no es fin de fichero.

Postcondiciones: Devuelve una variable de tipo registro con el contenido de la posición activa de *fsecind* y actualiza dicha posición a la del registro con la clave siguiente o fin de fichero.

int FinFichSecInd (FichSecInd fsecind)

Precondiciones: El fichero *fsecind* está abierto.
Postcondiciones: Devuelve 0 si no se ha llegado a la posición de fin de fichero y 1 en caso contrario.

int EstadoFichSecInd (FichSecInd fsecind)

Postcondiciones: Nos devuelve un entero que indica si el fichero, tras la última operación realizada con él, se encuentra en perfecto estado. (0, OK; un valor distinto, diferentes tipos de error). Debe utilizarse después de cada operación si se desea estar seguro de que no hubo ningún tipo de error.

5.8. Implementación del TAD Fichero Secuencial Indexado

```
/*-----------------------------------------------*/
/* FichSecInd.h                                  */
/*-----------------------------------------------*/
#ifndef _REGISTRO_
#define _REGISTRO_
   typedef int tclave; /* Por ejemplo */
   typedef struct {
              tclave clave;
              char campo1;
              float campo2;
              long campo3;
           } registro; /* Por ejemplo */
#endif
#ifndef _FICHDIR_
#define _FICHDIR_
   typedef struct {
              tclave clave;
              long pos;
           } tipoIndice;
   typedef struct {
              FILE *fichero;
              tipoIndice *indice;
              char *nombre;
              int tama;
              char modoAp;
              long posact;
           } tipoFichSecInd;
```

```
   typedef tipoFichSecInd *FichSecInd;

   FichSecInd AbrirFichSecInd(const char *nomfich,
                              char modo);
   void CerrarFichSecInd (FichSecInd fsecind);
   int LeerFichSecInd (FichSecInd fsecind, tclave c,
                       registro *reg);
   void EscribirFichSecInd (FichSecInd fsecind,
                            registro reg);
   int EliminarFichSecInd (FichSecInd fsecind,
                           tclave c);
   registro PrimeroSecFichSecInd (FichSecInd fsecind);
   registro LeerSecFichSecInd (FichSecInd fsecind);
   int FinFichSecInd (FichSecInd fsecind);
   int EstadoFichSecInd (FichSecInd fsecind);
#endif


/*-----------------------------------------------*/
/* FichSecInd.c                                  */
/*-----------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "error.h"
#include "fichsecind.h"

/* Operaciones privadas */

static int ExisteFichSecInd (const char *nomfich)
{
   FILE *f;

   if ((f = fopen(nomfich, "rb")) == NULL)
      return 0;
   else {
      fclose(f);
      return 1;
   }
}
static int CompararClaves (const void *c1,
                           const void *c2)
{
   return (*(tclave *)c1 - *(tclave *)c2);
}
```

```
static void Indexar (FichSecInd fsecind)
{
   int i;
   registro reg;

   fseek(fsecind->fichero, 0, SEEK_END);
   fsecind->tama = ftell(fsecind->fichero) /
                   sizeof(registro);
   fsecind->indice = (tipoIndice *)
      calloc(fsecind->tama, sizeof(tipoIndice));
   if (fsecind->indice == NULL)
      ERROR("Indexar: Memoria insuficiente");
   fseek(fsecind->fichero, 0, SEEK_SET);
   /* Indexar registros */
   for (i = 0; i < fsecind->tama; i++) {
      fread(&reg,sizeof(registro),1,fsecind->fichero);
      fsecind->indice[i].clave = reg.clave;
      fsecind->indice[i].pos = i;
   }
   qsort(fsecind->indice, fsecind->tama,
         sizeof(tipoIndice), CompararClaves);
}

static int BuscarBinClave (tipoIndice *indice,
                           int ini, int fin, tclave c)
{
   int med, rescomp;

   if (ini <= fin) {
      med = (ini+fin)/2;
      rescomp = CompararClaves(&c,
                               &(indice[med].clave));
      if (rescomp == 0)
         return med;
      else if (rescomp < 0)
         return BuscarBinClave(indice, ini, med-1, c);
      else
         return BuscarBinClave(indice, med+1, fin, c);
   }
   else
      return -1;
}

/* Operaciones públicas */
```

```
FichSecInd AbrirFichSecInd(const char *nomfich,
                           char modo)
{
   FichSecInd fsecind;

   fsecind = (tipoFichSecInd *)
             malloc(sizeof(tipoFichSecInd));
   if (fsecind == NULL)
      ERROR("AbrirFichSecInd: Memoria insuficiente");

   if ((modo == 'C' || modo == 'A') &&
       !ExisteFichSecInd(nomfich))
      ERROR("AbrirFichSecInd: Fichero inexistente. "
            "Modo incompatible");

   switch (modo)
   {
      case 'I': /*Crear vacío para lectura/escritura*/
                fsecind->fichero=fopen(nomfich,"wb+");
                break;
      case 'C': /* Abrir para lectura */
                fsecind->fichero=fopen(nomfich,"rb");
                break;
      case 'A': /* Abrir para lectura/escritura */
                fsecind->fichero=fopen(nomfich,"rb+");
                break;
      default: ERROR("AbrirFichSecInd: "
                     "Modo de apertura incorrecto");
   }
   fsecind->nombre = (char *)
                malloc(strlen(nomfich)*sizeof(char));
   if (fsecind->nombre == NULL)
      ERROR("AbrirFichSecInd: Memoria insuficiente");
   strcpy(fsecind->nombre, nomfich);
   fsecind->modoAp = modo;
   if (modo == 'I') {
      fsecind->indice = NULL;
      fsecind->tama = 0;
   }
   else /* modo 'C' o 'A' */
      Indexar(fsecind);
   fsecind->posact = 0;
   return fsecind;
}
```

```
void CerrarFichSecInd (FichSecInd fsecind)
{
   FILE *f;
   registro reg;
   int i;

   if (fsecind->modoAp == 'C')
      fclose(fsecind->fichero);
   else { /* Quitar registros borrados */
      f = fopen("temp", "wb");
      for (i = 0; i < fsecind->tama; i++) {
         fseek(fsecind->fichero,
               fsecind->indice[i].pos*sizeof(reg),
               SEEK_SET);
         fread(&reg,sizeof(reg),1,fsecind->fichero);
         fwrite(&reg,sizeof(reg),1,f);
      }
      fclose(f);
      fclose(fsecind->fichero);
      remove(fsecind->nombre);
      rename("temp", fsecind->nombre);
   }
   free(fsecind->indice);
   free(fsecind->nombre);
   free(fsecind);
}

int LeerFichSecInd (FichSecInd fsecind, tclave c,
                    registro *reg)
{
   int i;

   if (fsecind->indice == NULL)
     return 0;
   else {
     i = BuscarBinClave(fsecind->indice, 0,
                        fsecind->tama-1, c);
     if (i == -1) /* No existe la clave c */
       return 0;
     else {
       fseek(fsecind->fichero,
             fsecind->indice[i].pos*sizeof(registro),
             SEEK_SET);
```

```
        fread(reg,sizeof(registro),1,fsecind->fichero);
        fsecind->posact = i+1;
        return 1;
      }
   }
}


void EscribirFichSecInd (FichSecInd fsecind,
                         registro reg)
{
   int i, j, tama,
       encontrado;

   if (fsecind->modoAp == 'C')
      ERROR("EscribirFichSecInd: Modo incompatible");

   i = 0; encontrado = 0;
   while (i < fsecind->tama && !encontrado)
      if (fsecind->indice[i].clave < reg.clave)
         i++;
      else
         encontrado = 1;

   if (!encontrado ||
       fsecind->indice[i].clave > reg.clave) {
      /* Insertar clave en índice y
         escribir registro en fin de fichero */
      fsecind->indice = (tipoIndice *)
         realloc(fsecind->indice,
                (fsecind->tama+1)*sizeof(tipoIndice));
      if (fsecind->indice == NULL)
         ERROR("EscribirFichSecInd: "
               "Memoria insuficiente");
      for (j = fsecind->tama; j > i; j--)
         fsecind->indice[j] = fsecind->indice[j-1];
      fsecind->indice[i].clave = reg.clave;
      fseek(fsecind->fichero, 0, SEEK_END);
      tama = ftell(fsecind->fichero) / sizeof(reg);
      fsecind->indice[i].pos = tama;
      fwrite(&reg,sizeof(reg),1,fsecind->fichero);
      fsecind->tama++;
   }
```

```
   else { /* ya existe la clave, sobreescribir */
      fseek(fsecind->fichero,
            fsecind->indice[i].pos*sizeof(reg),
            SEEK_SET);
      fwrite(&reg,sizeof(reg),1,fsecind->fichero);
   }
   fsecind->posact = i+1;
}


int EliminarFichSecInd (FichSecInd fsecind, tclave c)
{
   int i, j, tama;

   if (fsecind->modoAp == 'C')
      ERROR("EliminarFichSecInd: Modo incompatible");

   i = BuscarBinClave(fsecind->indice, 0,
                      fsecind->tama-1, c);
   if (i == -1) /* No existe la clave c */
      return 0;
   else {
      for (j = i; j < fsecind->tama-1; j++)
         fsecind->indice[j] = fsecind->indice[j+1];
      fsecind->tama--;
      fsecind->posact = i;
      fsecind->indice = (tipoIndice *)
         realloc(fsecind->indice,
                (fsecind->tama)*sizeof(tipoIndice));
      if (fsecind->indice==NULL && fsecind->tama > 0)
         ERROR("EliminarFichSecInd: "
               "Memoria insuficiente");
      return 1;
   }
}


registro PrimeroSecFichSecInd (FichSecInd fsecind)
{
   registro reg;

   if (fsecind->tama == 0)
      ERROR("PrimeroSecFichSecInd: Fichero vacío");
```

```
   fseek(fsecind->fichero,
         fsecind->indice[0].pos*sizeof(reg),SEEK_SET);
   fread(&reg, sizeof(reg), 1, fsecind->fichero);
   fsecind->posact = 1;
   return reg;
}

registro LeerSecFichSecInd (FichSecInd fsecind)
{
   registro reg;

   if (fsecind->posact >= fsecind->tama)
      ERROR("LeerSecFichSecInd: Fin de fichero");

   fseek(fsecind->fichero,
         fsecind->indice[fsecind->posact].pos *
         sizeof(reg), SEEK_SET);
   fread(&reg, sizeof(reg), 1, fsecind->fichero);
   fsecind->posact++;
   return reg;
}

int FinFichSecInd (FichSecInd fsecind)
{
   return (fsecind->posact >= fsecind->tama);
}

int EstadoFichSecInd (FichSecInd fsecind)
{
   return ferror(fsecind->fichero);
}
```