mente pedagógica, desarrollándose con el fin de facilitar el estudio y el autoaprendizaje. De este modo todos los conceptos, algoritmos, tipos de datos abstractos, etc., se presentan ilustrados con ejemplos. Dichos ejemplos son descritos paso a paso y con detalle, con el fin de minimizar el esfuerzo de comprensión.

En el mismo sentido los contenidos se han estructurado de manera que sean evaluables, separando lo fundamental de lo accesorio, lo formativo de lo informativo. Así por ejemplo, los algoritmos de clasificación se presentan de manera que los detalles de implementación, que requieren un esfuerzo especial ya que los procedimientos pueden ser relativamente largos, no interfieran con los conceptos básicos que definen el algoritmo. El hecho de presentar las implementaciones completamente desarrolladas responde a la necesidad, que todos tenemos, de disponer del "programa" final para comprobar su "funcionamiento". En cualquier caso lo fundamental, tal y como se insiste a lo largo de todo el texto, son las ideas básicas y los algoritmos, no los programas. Por ello, y con el fin de que el alumno no "pierda el tiempo" en escribir los programas, se incluyen en CDROM las fuentes tanto en Pascal como en Modula. Esto no quiere decir que sólo los conocedores de estos lenguajes de alto nivel pueden comprender este texto. De hecho todos los contenidos del mismo se presentan insistiendo claramente en la diferencia entre algoritmo y programa, Tipo de Datos Abstracto y su implementación. Por tanto, los conceptos básicos son generales e independientes de las implementaciones utilizadas así como del lenguaje utilizado para realizarlas.

Los Autores.

Madrid, Junio de 2000.

CAPÍTULO 1

Introducción y conceptos fundamentales

1.1 ESTRUCTURAS DE DATOS

En general puede entenderse que todo sistema informático puede realizar dos tareas básicas: cálculos o gestión de información. Deliberadamente se ha prescindido de la expresión tratamiento de información por ser un aspecto más general que incluye al cálculo como caso particular. En una tarea de cálculo, función que explica el término de origen anglosajón *computador* (computer), el objetivo fundamental será la resolución de problemas numéricos tal que a partir de un conjunto de datos dados se obtengan los resultados requeridos; mientras que en la tarea de gestión de información, descrita por el vocablo que procede del francés *ordenador* (ordenateur) se almacenan datos con el fin de organizarlos y recuperarlos cuando sean precisos.

Dependiendo del ámbito específico de aplicación, se utilizará predominantemente una u otra capacidad, o ambas. Así por ejemplo, el cálculo predomina en la actividad científica y técnica de manera que el computador se ha convertido en una herramienta habitual en cualquier centro de investigación y desarrollo. Por otro lado, el tratamiento de información prevalece en las aplicaciones de gestión mediante las cuales los sistemas informáticos han irrumpido en la sociedad desde hace algunos años, aplicándose a labores que van desde la elaboración del censo hasta el inventario de un pequeño comercio. Más recientemente, el computador

forma parte de la vida cotidiana para un sinfín de tareas personales, desde la consulta de enciclopedias multimedia a la comunicación telemática a través de Internet.

En cualquier caso, independientemente de la tarea específica a la que vaya a ser dedicado un sistema informático, siempre se va a tener la necesidad de almacenar y manipular datos. Para ello se realizarán programas, es decir, una secuencia de instrucciones que indica las acciones que han de ser ejecutadas por el sistema programable.

La metodología de diseño de un programa ha ido evolucionando a lo largo de los años. En un principio era de hecho un arte, en el que la habilidad del programador para utilizar las posibilidades que le ofrecían los primeros lenguajes de programación era una componente fundamental. Además, el código de un programa medianamente complejo solía ser laborioso y difícil de seguir debido a que la 'herramienta' comúnmente utilizada era la conocida sentencia '`go to`', que obligaba a una planificación de acciones ciertamente intrincada. Desde entonces el arte de programar ha ido conceptualizándose, desarrollando técnicas para una programación sistemática y bien organizada. Así surgió la programación estructurada en la que las tareas se dividen y agrupan en funciones o procedimientos de manera que por un lado se atiende a los detalles de programación de éstos y posteriormente se utilizan en el programa principal que los requiera.

La programación estructurada, también llamada programación orientada a procedimientos, es bien conocida en la actualidad por todo aquel que se haya iniciado en programación con cualquier lenguaje de alto nivel, y constituye un claro ejemplo de *abstracción*. Los detalles de una tarea específica se encuentran en una función que posteriormente será vista como algo que tiene unas entradas y ofrece unas salidas determinadas. De este modo, puede observarse que la abstracción permite una programación más sencilla y sistemática, siendo la operación subyacente a todos los conceptos que serán presentados a lo largo del texto.

En sentido estricto, abstraer es la operación mediante la cual la inteligencia llega a formar conocimiento conceptual común a un conjunto de entidades, separando de ellos los datos contingentes e individuales para atender a lo que los constituye esencialmente. Es decir: aislar mentalmente o considerar por separado las cualidades de un objeto. En general, cuando un estudiante de cualquier disciplina, escucha que debe abstraer unos conocimientos dados, suele pensar que es una tarea difícil, poco tangible y nada práctica. Esto suele ser debido a que se confunde con 'imaginar algo inexistente'. Abstraer debe entenderse como 'extraer y agrupar', es decir, descubrir las cualidades comunes y englobarlas bajo un mismo concepto. Esta capacidad de la mente se pone de manifiesto constantemente en la vida cotidiana; de hecho, el lenguaje es el exponente máximo de la abstracción. En la programación, la acción de abstraer se realiza constantemente y a todos los niveles. Incluso cuando se trata con ceros y unos se están ignorando las características propias de cada uno, tales como los valores reales de las tensiones que los representan internamente en la máquina y su nivel de ruido.

Atendiendo a estas consideraciones la metodología de programación ha ido avanzando en el proceso de abstracción. En la programación orientada a procedimientos este hecho es claramente reflejado por la expresión del profesor N. Wirth:

Estructuras de datos + Algoritmos = Programas

A lo largo de este capítulo serán analizados los conceptos fundamentales de las dos columnas básicas en las que se basa la programación: las *estructuras de datos* y los *algoritmos*. Ambos están íntimamente relacionados, hecho que se manifestará a lo largo de todo el texto.

Finalmente debe destacarse el hecho de que en el último nivel de abstracción aparece una nueva metodología conocida como programación orientada a objetos, la cual, parafraseando la expresión del profesor N. Wirth, podría expresarse como:

Tipos de datos abstractos $\cup$ *Algoritmos = Objeto*

A esta metodología se dedica el último capítulo a modo de introducción.

1.2 ESTRUCTURAS DE DATOS Y TIPOS DE DATOS ABSTRACTOS

1.2.1 CONCEPTO

Aunque los conceptos englobados en los términos 'estructuras de datos' y 'tipos de datos abstractos' son sencillos, en ocasiones puede producirse cierta confusión debido a que en algunos textos puede encontrarse el término 'estructuras de datos' referido a lo que en otros son los 'tipos de datos abstractos', y en gran número de ellos se utilizan los términos 'tipo de datos' y 'tipo de datos abstracto' indistintamente. Con el fin de precisar los términos, en principio se distinguirá entre 'tipo', 'tipo de datos abstracto' y 'estructura de datos'.

Definición 1.1: Un tipo de datos es una colección de valores.

Definición 1.2: Un Tipo de Datos Abstracto (TDA) es un tipo de datos definido de forma única mediante un tipo y un conjunto dado de operaciones definidas sobre el tipo.

Definición 1.3: Una estructura de datos es la implementación física de un tipo de datos abstracto.

A la vista de estas definiciones cabe preguntarse qué son los enteros, caracteres, etc., conocidos en programación básica y generalmente predefinidos en los lenguajes de alto nivel. Por ejemplo, el tipo Booleano está formado por los dos valores

`True` y `False` pero no es un tipo de datos abstracto ya que sobre él no se han definido las operaciones para manipularlo. Sin embargo, el tipo Booleano junto con los operadores booleanos (AND, OR, NOT) constituye un tipo de datos abstracto. Del mismo modo los enteros son un tipo formado por el conjunto de valores denominados por las matemáticas 'enteros'. Este tipo junto con las operaciones definidas sobre ellos forman un tipo de datos abstractos.

Es importante considerar estas definiciones y tenerlas en mente a lo largo de todo el texto, no olvidando nunca que las operaciones que manipulan los objetos están incluidas en la especificación de un tipo de datos abstracto y que no interviene ninguna consideración sobre la implementación. Ésta es expresa y conscientemente ocultada, hecho que es conocido como encapsulación de datos (Figura 1.1 a). El hecho de que se oculte la información no significa que sea inaccesible. Se trata de disponer únicamente de lo necesario para realizar las operaciones, funciones de acceso y variables de entrada y salida. Esto se realiza una vez asegurado que la implementación es correcta y adecuada.

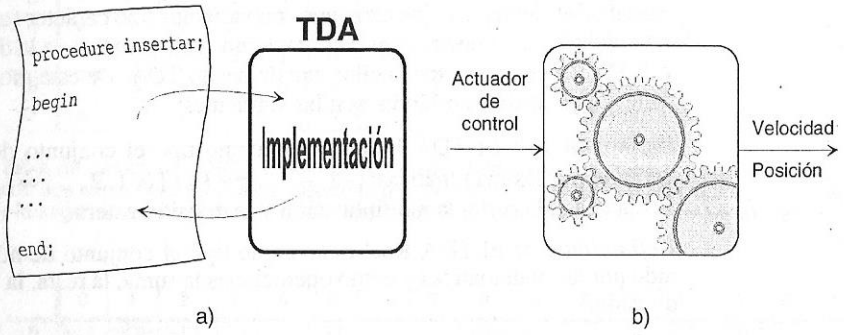

Figura 1.1 a) Encapsulación de datos b) Motor.

El encapsulado, consecuencia misma del proceso de abstracción, se utiliza en todos los ámbitos de la ciencia y de la ingeniería. Por ejemplo, se utilizan diagramas de bloques caracterizados por las entradas y salidas de los sistemas físicos. De este modo un *motor* puede representarse por la entrada proveniente del actuador de control, y la salida (velocidad y/o posición) tal y como se ilustra en la Figura 1.1.b, sin atender a los detalles físicos del mismo (escobillas o no, inducido, eléctrico o de combustión, etc.). Del mismo modo, esta idea se utiliza en otros ámbitos de la informática, como por ejemplo en la técnica de monitorización en el desarrollo de sistemas operativos.

Los conocedores del C estándar y de los primeros compiladores de este lenguaje recordarán que el tipo de datos abstracto `Boolean` no era un tipo predefinido, mientras que en Pascal por ejemplo si lo era. Pero de este hecho es un error deducir que el tipo de datos `Boolean` no existe en C estándar. Lo cierto es que no está implementado en los tipos básicos, hecho superable sencillamente definiendo un tipo con los valores `True` y `False` y asociándole los operadores AND, OR y NOT. Otro ejemplo característico es el de los enteros. Como es bien sabido, todo computador presenta limitaciones para la representación de los números enteros de manera que el rango de posibles valores que se adaptan a este tipo dependen del computador. En general la implementación de un entero utiliza una palabra para su almacenamiento. Así, en un IBM PC con palabras de 16 bits se utilizan dos bytes para almacenar cada entero, con lo que el rango de posibles valores va desde -32768 hasta +32767. Pero otras máquinas ofrecen rangos diferentes. Los clásicos DEC VAX y los procesadores de la familia i86 de Intel (a partir del 80386) utilizan 32 bits, por ejemplo. Es más, en lenguaje C es posible implementar los enteros mediante diversas formas, aumentando o disminuyendo el rango de valores. Así, si el entero mediante la declaración `int` está representado por 2 bytes, con la declaración `long int` se almacena en 4 bytes; si `int` almacena en 4 bytes, la declaración `short int` los almacenará en 2 bytes. Es decir, las implementaciones de algo tan sencillo como son los enteros pueden ser distintas. Pero lo que es importante resaltar es que el tipo de datos abstractos denominado 'enteros' es siempre el mismo. Si en un caso dado la implementación con el rango de 2 bytes es insuficiente será preciso cambiarla, pero esto no afecta al tipo de datos abstracto ni al uso que se haya hecho de él a lo largo del programa elaborado. Lo mismo sucede con otras estructuras más complejas que se presentarán en temas posteriores. Por ejemplo, supóngase que en un programa se utiliza una pila, y en su momento el diseñador decidió utilizar una implementación estática mediante un arreglo de 100 elementos. Pasado el tiempo se observa que es necesario un tamaño mayor. En ese momento basta con atender a los detalles de la implementación de la estructura, aumentando el número de elementos del arreglo o cambiando a una pila implementada dinámicamente. Estas modificaciones no requieren un gran esfuerzo ya que el tipo de datos abstracto es el mismo, y por tanto, la pila y su funcionalidad siguen siendo las mismas.

Los anteriores comentarios no implican que las implementaciones sean poco importantes o irrelevantes. De hecho son fundamentales y la elección de una u otra implementación para un TDA dado deberá analizarse cuidadosamente. Lo importante es entender la diferencia entre un TDA y su implementación, y considerar cada uno de ellos en su momento. De esta manera se facilita el proceso de diseño así como sus posibles modificaciones y mejoras.

Como puede observarse de las consideraciones anteriores, el término 'estructura de datos' hace referencia, según la Definición 1.3, a la implementación del TDA. En este sentido podría entenderse que los numerosos textos titulados *Estructuras de datos y algoritmos* hacen referencia a implementaciones de TDA y a algoritmos. Sin embargo, al manejar cualquiera de ellos, inmediatamente se precisan los términos, introduciendo los conceptos de TDA y estableciendo su significado. El término 'estructura de datos' se reserva para las representaciones físicas de los datos.

Con todo lo dicho, la selección de un TDA para resolver un problema dado deberá realizarse analizando en primer lugar el problema para determinar las restric-

ciones y especificaciones que debe satisfacer la solución, seguidamente se deberán determinar las operaciones básicas a realizar y por último se seleccionará la más adecuada. Evidentemente la más adecuada vendrá dada en función de alguna característica relevante, para lo cual habrá que establecer algún parámetro objetivo que proporcione una medida de su coste o de sus beneficios. A lo largo de todo el texto se presentarán numerosos TDA, y una vez comprendidos tanto su concepto como su funcionalidad, serán analizados con detalle.

Desgraciadamente no existe un único TDA capaz de ser el mejor en todos los casos, y todos ellos presentan inconvenientes y ventajas que los hacen adecuados o no para resolver un problema específico. Así por ejemplo, se presentarán los árboles de búsqueda. Una vez analizados se obtiene la conclusión de que en el peor de los casos degenerarán en una lista lineal, perdiendo sus ventajas y haciendo que la búsqueda sea muy costosa. Buscando una solución mejor se introducen los árboles AVL, se analizan y se concluye que en el peor de los casos la operación de búsqueda tiene un coste menor. Sin embargo, esto no significa que sean los mejores; en una aplicación dada es posible que nunca se dé el peor de los casos y por tanto, puede que los árboles de búsqueda sean una mejor solución para ese problema en concreto.

El coste del TDA no viene determinado por un único factor sino que deberá ser analizado desde diversos puntos de vista. En general deberán considerarse siempre dos aspectos. El primero, en relación con el almacenamiento que será establecido por los datos y la cantidad de ellos que deberán ser manejados. El segundo aspecto, igualmente importante, será el coste de las operaciones básicas. En el caso de los árboles, por ejemplo, serán los costes de las operaciones de inserción, búsqueda y eliminación. En general estas operaciones se realizan mediante algoritmos, con lo que su análisis se realizará utilizando las técnicas de análisis de algoritmos. Como puede observarse, estos dos aspectos consideran las dos restricciones fundamentales que deben tenerse en cuenta en el diseño de todo programa: las limitaciones en el espacio de almacenamiento y el tiempo de ejecución necesario para realizar una tarea.

En definitiva, debido al proceso de abstracción, el diseño deberá realizarse siguiendo tres pasos fundamentales:

1. Análisis de datos y operaciones.
2. Elección del Tipo de Datos Abstracto.
3. Elección de la implementación.

En los capítulos siguientes se presentarán diversos TDA básicos y bien conocidos. En todos ellos se presentará el concepto que incorporan, su utilidad y su coste. Sin embargo, no sólo son interesantes con el fin de conocerlos para utilizarlos si fuese necesario, sino que fundamentalmente son útiles como ejemplo de metodología de diseño de TDA. De este modo, si para un problema específico no es útil ninguno de los TDA básicos, se podrá definir y realizar un TDA siguiendo la misma metodología.

Para finalizar esta sección un breve comentario sobre terminología. En la mayoría de la literatura se muestra una diferencia entre los términos 'tipo de datos'

y 'tipo de datos abstracto' [Shaffer, 1997], [Heileman, 1998]. Cuando esto sucede, el concepto de TDA sigue siendo el dado en la Definición 1.3 mientras que se hace referencia a 'tipo de datos' como una descripción matemática y formal de un tipo y un conjunto de operaciones. En otros textos [Dale, 1989] se utiliza el término 'tipo de datos' como TDA. En cualquier caso, desde el punto de vista de la metodología de programación el concepto fundamental que debe considerarse siempre y comprender sin ambigüedades es el de TDA.

Con el fin de tener una perspectiva de conjunto, en el siguiente apartado se realiza una breve presentación de los tipos de datos conocidos de programación básica, pero presentados como TDA.

1.2.2 TIPOS DE DATOS BÁSICOS

En un primer acercamiento a la programación, independientemente del lenguaje utilizado, es habitual entender que los tipos de datos son herramientas o utilidades para almacenar los datos necesarios para resolver el problema. Así, se suele entender que los enteros se almacenan en variables tipo Entero (`int` en C, `Integer` en Pascal o Modula2, ...) , los caracteres en variables tipo carácter (`char` en C, en Pascal o Modula2,...). Es decir, en este contexto no suele atenderse a la diferencia entre tipo y TDA. Sin embargo, todos ellos son de hecho TDA. De este modo las definiciones de los tipos de datos estándar son las siguientes:

Definición 1.4: El TDA Entero tiene como tipo el conjunto de números enteros definido por las matemáticas $\{-1, -2, \ldots, \infty\} \cup \{0, 1, 2, \ldots, \infty\}$, y como operaciones la suma, la resta, la multiplicación y la división entera.

Definición 1.5: El TDA Real tiene como tipo el conjunto de números reales definido por las matemáticas y como operaciones la suma, la resta, la multiplicación y la división.

Definición 1.6: El TDA Booleano tiene como tipo el conjunto de valores booleanos {True, False} y como operaciones las definidas por el álgebra de Boole (AND, OR, NOT).

Definición 1.7: El TDA Carácter tiene como tipo el conjunto de caracteres definido por un alfabeto dado y como operaciones todos los operadores relacionales ($<$, $>$, $=$, $\geq$, $\leq$, $=$, $<>$).

En el apartado anterior ya se realizaron algunos comentarios sobre la implementación de los TDA Entero y Booleano. Del mismo modo el TDA Real tiene en su implementación diferentes soluciones, en función de la representación que se realice de ellos, que depende de diversos factores. En coma flotante debe considerarse la representación de la mantisa (complemento a 2, magnitud-signo,...), la del exponente (polarizado, no polarizado,...), la localización de ambos, etc. Estas implementaciones están normalizadas. Por ejemplo, el Institute of Electrical and Electronics Engineers (IEEE) ha propuesto la norma IEEE 754-1985 para la representación de números en coma flotante para las operaciones aritméticas en mini y microcom-

putadores, en la que se contemplan tres formatos básicos, denominados de simple, doble y cuádruple precisión (véase [Dormido, 2000]).

Análogamente el TDA Carácter se implementará de diferentes formas dependiendo del alfabeto utilizado. En general, todos los alfabetos vendrán caracterizados por una secuencia de cotejo, es decir, por la secuencia ordenada de los caracteres que constituyen el conjunto. En todos los alfabetos, como es natural, las letras se encuentran ordenadas entre sí, al igual que los dígitos. Así 'a' < 'b' < 'c'..., y '1' < '2' < '3'. Los tres conjuntos de caracteres más conocidos son el CDC-Científico, el EBCDIC (*Extended Binary Coded Decimal Interchange Code*) y el ASCII (*American Standard Code for Information Interchange*). Las Tablas 1.1, 1.2 y 1.3 muestran las secuencias de cotejo de los tres.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	E.B.	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Tabla 1.1 Tabla de caracteres ASCII. El número correspondiente de la secuencia de cotejo, se forma con dos cifras hexadecimales, siendo la primera cifra, la que aparece en la primera columna y la segunda la que aparece en la primera fila. E.B. indica el carácter espacio en blanco.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX		PT			GE				FF	CR		
1	DLE	SBA	EUA	1C		NL				EM			DUP	SF	FM	ITB
2							ETB	ESC						ENQ		
3			SYN					EOT					RA	NAK		
4	E.B.										¢	.	<	(	+	
5	&										!	$	*	)	;	¬
6	-	/									|	,	%	_	>	?
7											:	#	@	'	=	"
8		a	b	c	d	e	f	g	h	i						
9		j	k	l	m	n	o	p	q	r						
A		~	s	t	u	v	w	x	y	z						
B																
C	{	A	B	C	D	E	F	G	H	I						
D	}	J	K	L	M	N	O	P	Q	R						
E	\		S	T	U	V	W	X	Y	Z						
F	0	1	2	3	4	5	6	7	8	9						

Tabla 1.2 Tabla de caracteres EBCDIC. La celda con E.B. indica el espacio en blanco.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	:	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	P	Q	R	S	T	U	V	W	X	Y	Z	0	1	2	3	4
2	5	6	7	8	9	+	-	*	/	(	)	$	=	E.B.	,	.
3	≡	[	]	%	≠	↱	∨	∧	↑	↓	<	>	≤	≥	¬	;

Tabla 1.3 Secuencia de cotejo CDC-Científico. La celda con E.B. indica el espacio en blanco.

En estas tablas, las celdas con más de un carácter indican caracteres de control que aparecen representados por su correspondiente mnemotécnico. Como puede observarse en el código ASCII los números van antes que las letras, mientras que en los otros dos códigos van después. Las minúsculas van antes que las mayúsculas en EBCDIC, después en ASCII, y en CDC-Científico no forman parte del conjunto de valores.

Los tipos de datos definidos hasta ahora se conocen como tipos de datos simples o atómicos debido a que no son divisibles, es decir, no tienen componentes que puedan ser accedidos independientemente. Todos ellos excepto los reales son tipos ordenados, y a todos los tipos ordenados se les puede aplicar los operadores relacionales. En este contexto se definen los 'tipos de datos escalares' de la siguiente forma:

Definición 1.8: Se denominan *tipos de datos escalares* a aquellos en los que el conjunto de valores está ordenado y cada valor es atómico.

Ejemplos de ellos son los tipos de datos Carácter, Entero, Real y Booleano. Son escalares ya que todos ellos son de tipo atómico y están ordenados (los enteros y reales según indican las matemáticas, los caracteres según su secuencia de cotejo y los booleanos tal que False es menor que True).

Algunos tipos escalares presentan una propiedad adicional, la ordinalidad.

Definición 1.9: Se denominan *tipos de datos ordinales* a aquellos en los que cada valor tiene un único predecesor (excepto el primero), y un único sucesor (excepto el último).

Los tipos Carácter, Entero y Booleano son ordinales, pero como es bien conocido de matemáticas el tipo Real no lo es. Los lenguajes de alto nivel suelen presentar tres funciones adicionales asociadas a los tipos de datos ordinales: una que devuelve la posición de un elemento (`Ord`), otra que devuelve el predecesor (`Pred`), y otra que devuelve el sucesor (`Succ`).

Es bien sabido de la matemática, que los números reales no forman un conjunto ordenado al no ser numerable. Es bien cierto que a los números reales se les puede aplicar los operadores relacionales pero no se puede determinar su predecesor o su sucesor de forma única. Pensemos por ejemplo en los números 5.1 y 5.2; está claro que 5.1<5.2 y podríamos concluir erróneamente que 5.1 es el predecesor de 5.2 y este último el sucesor de 5.1. Sin embargo si consideramos dos cifras decimales, el

Como puede observarse, la lista es una estructura dinámica desde el punto de vista lógico, ya que su longitud dependerá del número de elementos que tenga, aumentará al insertar y se reducirá al suprimir.

El TDA lista puede implementarse de formas muy diferentes. La más inmediata a partir de los TDA presentados hasta ahora es mediante arreglos. Así, la lista, que es dinámica independientemente de su implementación, se realiza mediante una implementación estática. Sin embargo también es posible implementarla dinámicamente, mediante TDA basados en punteros.

En general, los tipos de datos presentados hasta ahora, tanto simples como compuestos, son bien conocidos por aquellos iniciados en las técnicas básicas de programación. No ocurre lo mismo con otros dos TDA fundamentales: la Secuencia y los Punteros. El primero de ellos debido a que como claro ejemplo de abstracción suele confundirse con alguna de sus implementaciones (ficheros o archivos, buffers de memoria, ...), mientras que el segundo suele estar poco dominado por considerarse un aspecto 'avanzado' de programación básica. Por ello se tratan de forma algo más detallada en las siguientes secciones.

1.2.3 TIPOS DE DATOS ABSTRACTOS: PUNTERO Y LISTAS ENLAZADAS

En principio los punteros son materia conocida de las nociones básicas de programación. En esta sección se recuerdan los conceptos esenciales de los mismos dado que su manejo será fundamental para todas las estructuras de datos dinámicas presentadas en el texto.

En principio un puntero es un tipo de datos simple cuyo valor es la dirección de una variable de otro tipo, denominada variable referenciada. La Figura 1.2 muestra la representación gráfica característica de estos tipos.

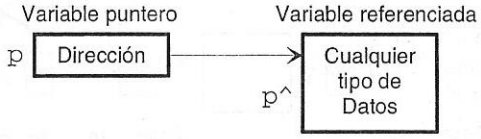

Figura 1.2 Concepto de punteros y variables referenciadas.

La utilidad de estos tipos radica en que permiten realizar una reserva dinámica de memoria. Para comprender este concepto es importante recordar cómo trabaja un programa compilado. Recuérdese que en cualquier programa de alto nivel lo primero que debe realizarse es la declaración de constantes (`Const`), tipos (`Type`) y variables (`Var`) que el programa va a necesitar. Cuando se ejecuta el programa lo primero que se hace es reservar memoria para todas estas variables. El sistema operativo es quien se encarga de gestionar la memoria primaria, y lleva el control de la memoria que está reservada porque es necesaria para los programas que se están ejecutando y la que el sistema tiene libre. A estas variables se las denomina *variables estáticas*, ya que 'existirán', en el sentido de que disponen de memoria reservada, a lo largo de toda la ejecución del programa.

Las variables referenciadas son *variables dinámicas*, es decir, se crean en tiempo de ejecución. Recuérdese que 'crear' una variable significa reservar espacio en memoria para dicha variable y asignar una etiqueta a ese espacio, es decir, tomar la memoria libre del sistema necesaria e indicar que está ocupada por dicha variable. Para las variables estáticas, esta indicación se realiza en la sección de declaraciones de forma que en tiempo de ejecución la memoria está reservada siempre, se utilice o no. Sin embargo, la memoria requerida por las variables dinámicas se reserva, como memoria ocupada, durante la ejecución del programa y sólo cuando es necesaria. Cuando no hace falta se libera o 'destruye', es decir, se indica que es memoria libre y el sistema podrá reutilizarla para otras necesidades.

En la mayoría de los computadores el valor de un puntero es un número entero, debido a que las posiciones de memoria van desde cero hasta el tamaño de la memoria menos uno. Sin embargo, este valor no es de tipo entero sino una dirección. Es más, en algunos tipos de máquinas, la memoria es accedida a través de páginas o segmentos y la dirección puede tener una estructura compleja formada por un selector o indicador de segmento y un desplazamiento dentro del segmento, alejándose considerablemente del concepto de tipo entero. El contenido de la variable referenciada puede ser cualquier tipo de dato: registro, otro puntero, etc. (Figura 1.2).

La declaración de los tipos puntero se realiza en la sección `TYPE`. En Modula2, por ejemplo, la sintaxis es la siguiente:

```
TYPE
   Tipo_puntero = POINTER TO Tipo_variable_referenciada;
```

donde `Tipo_variable_referenciada` es el tipo de datos al que pertenecerán las variables referenciadas. La declaración de la variable puntero se realiza en la sección `VAR`, y su sintaxis en Modula2 es:

```
VAR
   variable_puntero : Tipo_puntero;
```

La variable referenciada es accedida con el puntero mediante el operador apuntador, cuyo símbolo en Modula2 es el acento circunflejo: '^' .

Como ya se ha explicado detalladamente, la memoria necesaria para la variable puntero es reservada y ocupada de la misma forma que las variables estáticas. Sin embargo, la variable referenciada (por ejemplo un registro con varios campos) no existe hasta que no ha sido creada. Para crearla en Modula2 se dispone del procedimiento `Allocate`, cuya sintaxis es:

```
Allocate(variable_puntero, SIZE(Tipo_variable_referenciada));
```

Este procedimiento realiza dos acciones. En primer lugar toma la memoria libre necesaria para la variable referenciada. Esto lo realiza mediante la función `SIZE` que determina el tamaño de la variable referenciada con el fin de reservar únicamente la cantidad de memoria imprescindible. En segundo lugar asigna la dirección de memoria que ha reservado al puntero, `variable_puntero`. Tras este procedimiento la variable referenciada existe en el sentido de que hay memoria asignada para ella y además puede ser accedida por el puntero.

Una vez que una variable referenciada ya no va a ser utilizada por el programa, será destruida o liberada. El procedimiento en Modula2 que realiza esta tarea es `Deallocate`, cuya sintaxis es:

```
Deallocate(variable_puntero, SIZE(Tipo_referenciada));
```

```
MODULE ejemplo;
TYPE
   puntero = POINTER TO integer;
VAR
1  p,q : puntero;

BEGIN
2  ALLOCATE(p,SIZE(integer));
3  p^:=8;
4  q:=p;
5  q^:=5;
6  DEALLOCATE(p,SIZE(integer))
END.
```

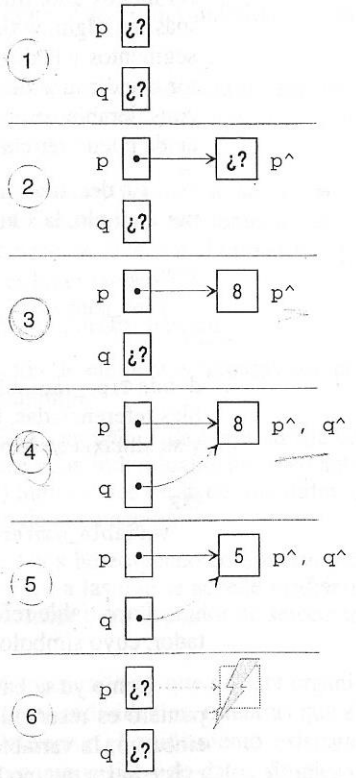

Figura 1.3 Ejemplo de programa mostrando las operaciones básicas con punteros.

Los comentarios anteriores muestran las características del tipo puntero, pero para poder establecerlo como TDA es imprescindible analizar las operaciones asociadas a los mismos. Éstas son la *asignación* y la *comparación*.

Recuérdese que los valores de los punteros son las direcciones a las que apuntan y por tanto, estas operaciones afectan a dichas direcciones. La Figura 1.3 ilustra estas operaciones y las anteriores consideraciones. En primer lugar se declara un tipo puntero a entero, de manera que la variable referenciada será de tipo entero. Seguidamente se declaran dos variables puntero de este tipo, `p` y `q`. Con ello el valor de estas variables no se asigna de ninguna manera, por lo que su contenido debe considerarse desconocido. En este sentido su comportamiento es el mismo que cualquier otra variable estática.

Cuando durante la ejecución del programa se requiera utilizar las variables referenciadas, éstas se crean mediante las sentencias: `Allocate(p,SIZE(integer))` y `Allocate(q,SIZE(integer))` (o también: `New(p)` y `New(q)`). Entonces cada puntero tendrá como valor la dirección de su variable referenciada de forma que apunta hacia ella. En esta situación las variables referenciadas recién creadas no tienen ningún valor (están sin inicializar). Para asignarlo se accede a la variable referenciada mediante el nombre del puntero que la apunta y el operador de acceso '^', realizándose la asignación mediante su operador, ':='. Debe resaltarse el hecho de que esta asignación no es la operación asociada al TDA puntero, sino que es simplemente la asignación de valores a variables. El hecho de que éstas sean variables referenciadas no implica diferencia alguna. En la Figura 1.3 se muestra la asignación del valor `8` a la variable referenciada (`p^:=8`).

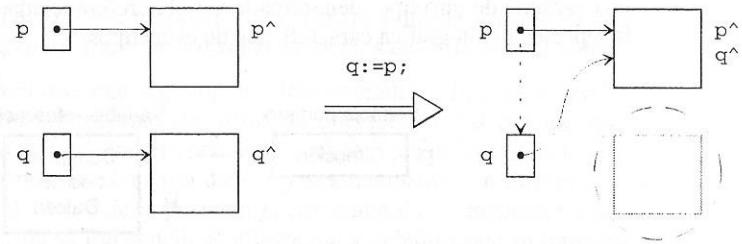

Figura 1.4 Pérdida de memoria debido a una incorrecta asignación entre punteros.

La asignación de punteros consiste en la asignación de valores puntero, es decir direcciones, entre variables puntero. Así `q:=p` tiene como resultado que el valor de `q` sea la dirección almacenada en `p`, es decir, `q` apunta a donde apunta `p`. Debe tenerse en cuenta que la asignación incorrecta de punteros puede tener un efecto muy negativo. Este efecto consiste en dejar una variable refenciada sin tener acceso a ella, tal y como se ilustra en la asignación de punteros de la Figura 1.4. Obsérvese cómo la variable que estaba apuntada por `q` queda sin apuntar por nadie (señalada con un círculo punteado en la Figura 1.4). Por tanto, se está ocupando memoria que no podrá ser reutilizada. Por ello, antes de asignar punteros debe tenerse en cuenta que si la

variable referenciada va a ser necesitada posteriormente tiene que estar apuntada por algún puntero, y si no lo va a ser entonces debe ser liberada, es decir, debe ser devuelta a la memoria libre del sistema. Esto se realiza en Modula2 mediante los procedimientos `Deallocate` o `Dispose`.

Del mismo modo, la asignación entre variables referenciadas no es más que una asignación entre variables del tipo de la misma. Así, según se muestra en la Figura 1.5, la asignación `q^:=p^`, asignaría el valor de `p^` (25), a `q^`, en el supuesto de que se hubiesen creado las correspondientes variables referenciadas.

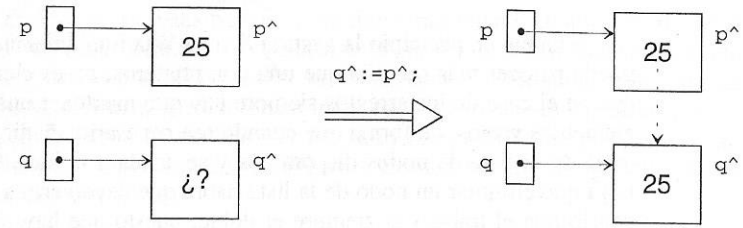

Figura 1.5 Asignación entre variables referenciadas.

La segunda operación asociada a los punteros es la comparación. Al comparar se está comprobando si sus direcciones son iguales o no, es decir, si apuntan o no a la misma dirección de memoria. En la Figura 1.6 se muestra cómo cuando apuntan a diferentes direcciones el resultado de la comparación es Falso (aunque los contenidos de las variables referenciadas sean idénticos), mientras que es Verdadero si apuntan a la misma dirección.

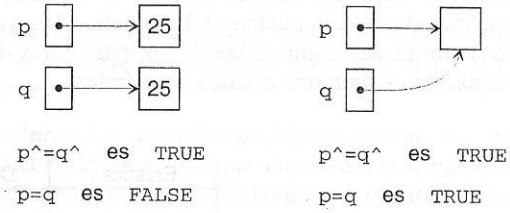

Figura 1.6 Operaciones de comparación entre punteros y variables referenciadas.

Es importante determinar cuándo un puntero no señala a ninguna variable referenciada. Para ello, en Modula2 por ejemplo, se dispone de la palabra clave `NIL` que representa una constante y en otros lenguajes se dispone de constantes similares. Así, la asignación de esta constante a un puntero indica que éste no apunta a nada. Su representación gráfica se muestra en la Figura 1.7. Esta constante tiene la particularidad de ser universal y poder ser asignada a cualquier tipo de puntero, independientemente del tipo de variable referenciada al que apunte.

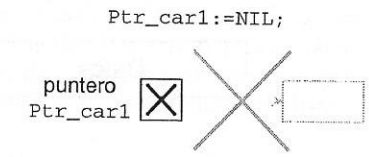

Figura 1.7 Puntero que no apunta a ninguna variable referenciada.

Atendiendo a estas consideraciones el TDA puntero puede definirse de la siguiente manera:

Definición 1.15: El TDA puntero es un tipo de datos simple cuyos valores son direcciones de memoria, y sus operadores asociados son la asignación de punteros y la comparación de punteros.

Por último, debe tenerse en cuenta el ahorro de almacenamiento que supone la utilización de punteros. La reserva de memoria realizada estáticamente sólo afecta a las variables puntero mientras que las referenciadas se crean y se destruyen de manera que sólo se reserva memoria cuando es necesitada. En este sentido, el ejemplo de la Figura 1.3 no es en absoluto ilustrativo de este hecho, y debe considerarse un ejemplo académico con el único propósito de ilustrar de forma sencilla las operaciones básicas y los operadores asociados a este TDA. En efecto, obsérvese que estáticamente se reservan dos punteros, y si cada puntero requiere cuatro bytes (como sucede habitualmente en las máquinas de 32 bits), ocupan lo mismo que podrían requerir cuatro variables de tipo entero en una máquina que representa los enteros con 16 bits. Por tanto, al crear las variables referenciadas se estarían utilizando un total de doce bytes, ocho reservados estáticamente para los punteros y cuatro dinámicamente para las variables referenciadas, cuando con reservar cuatro de forma estática sería suficiente para almacenar los dos enteros deseados. Sin embargo el ahorro de memoria y su apropiada utilización es evidente cuando la memoria a utilizar por la variable referenciada es superior a la necesaria para almacenar un puntero, o una misma memoria debe ser utilizada para distintos fines a lo largo de la ejecución de un programa; o se utiliza, como se va a describir en el siguiente apartado con las listas enlazadas, para la creación de estructuras de datos dinámicas.

Los punteros son la base de construcción de todas las denominadas estructuras dinámicas, como los árboles que se presentarán en este texto, o las listas enlazadas. Seguidamente se recuerdan los aspectos fundamentales de estas últimas.

1.2.4 LISTAS ENLAZADAS

Para almacenar de una forma organizada un conjunto de datos homogéneos se dispone de los arreglos. Esta estructura, como se dijo, es estática y está incorporada en los lenguajes de alto nivel. En este apartado se presentan las listas enlazadas. Su finalidad, como en el caso de los arreglos, es almacenar organizadamente datos del mismo tipo.

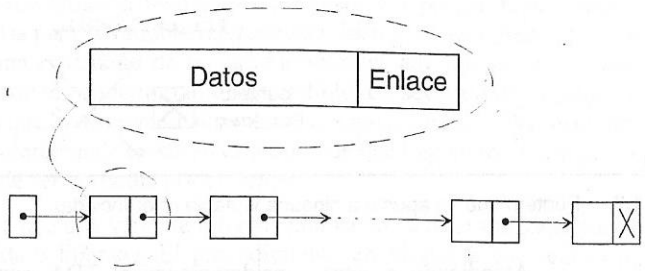

Figura 1.8 Representación esquemática de una lista enlazada y de uno cualquiera de sus nodos. El campo destinado a almacenar la información (Datos) es normalmente de un tamaño mucho mayor que el campo enlace.

Las listas enlazadas son tipos de datos dinámicos que se construyen con nodos. Un *nodo* es un registro con dos campos, uno de ellos contiene las componentes y se le denominará 'datos' (o data) y el otro es un valor que señala al siguiente nodo y se le denominará enlace (o next). El campo datos debe entenderse conceptualmente, es decir, en una implementación concreta pueden ser varios los campos que se consideran como datos. Por ejemplo, podrían ser los campos: Nombre, Apellidos, Dirección, DNI, etc.

La definición del TDA lista enlazada es la siguiente:

Definición 1.16: El TDA lista enlazada es una colección de nodos ordenada según su posición, tal que cada uno de ellos es accedido mediante el campo enlace del nodo anterior.

La Figura 1.8 ilustra el concepto de lista enlazada.

Haciendo uso de los punteros, es decir, definiendo los valores del campo `Enlace` del tipo `nodo` como punteros, las listas enlazadas se implementan aprovechando todas las ventajas de la asignación dinámica de memoria. Ésta es, de hecho, la forma habitual y más eficaz de implementar la lista enlazada (lo que explica que la representación gráfica de una lista enlazada se realice en términos de direccionamientos dinámicos, tal y como muestra la Figura 1.8).

Sin embargo, ésta no es la única implementación posible. Obsérvese que las listas enlazadas pueden implementarse estáticamente, sin más que definir el tipo del arreglo como nodos y utilizar los índices en el campo enlace para determinar el siguiente nodo. La Figura 1.9 muestra un ejemplo de este tipo de implementación de lista enlazada.

En esta figura, cada nodo está almacenado en una posición del arreglo y está formado por un registro con dos campos, al igual que la implementación dinámica basada en punteros. Por una parte está el campo de datos que contiene la información almacenada en la lista, y por otra el campo enlace, con la diferencia de que ahora en lugar de almacenar un puntero, es un valor ordinal que apunta a la posición del arreglo en la que está el siguiente elemento.

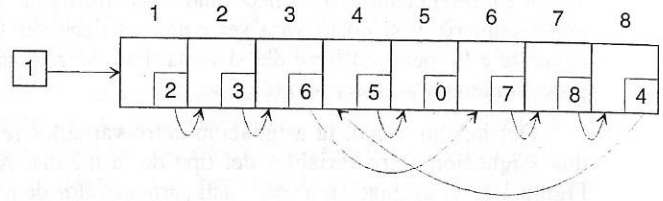

Figura 1.9 Implementación estática de una lista enlazada mediante un arreglo de registros.

Aunque en principio la gestión de una lista implementada mediante un arreglo pueda parecer más sencilla que una con punteros, no es cierto. Téngase en cuenta que en el caso de los arreglos siempre hay que mantener una lista enlazada con los elementos vacíos, de forma que cuando sea necesario añadir un nuevo elemento, se tome de la lista de nodos disponibles y se añada a la lista. Y a la inversa, cuando haya que eliminar un nodo de la lista habrá que devolverlo a la lista de disponibles, con lo que el trabajo es siempre el doble, puesto que hay que mantener dos listas sobre el mismo arreglo. En el caso de la lista con punteros, esta tarea la realiza el sistema automáticamente.

En este punto hay que marcar la distinción entre un tipo de datos abstracto y la naturaleza de su implementación. Ambos conceptos pueden ser considerados de forma estática o dinámica. Una variable de tipo 'array' (arreglo) es una estructura típicamente estática, pero puede almacenarse en memoria de forma estática (declarada en una zona de declaración de variables) o de forma dinámica (mediante un puntero). Lo mismo sucede, por ejemplo, con una pila o una lista, que son por naturaleza estructuras típicamente dinámicas pero que pueden implementarse con asignación de memoria estática (dentro de un array) o con asignación de memoria dinámica (basada en punteros). La Figura 1.10 muestra varios ejemplos con las cuatro posibilidades según se combinen estructuras de datos estáticas o dinámicas con asignación de memoria estática o dinámica.

Asignación de memoria	TDA: Estáticas	TDA: Dinámicas
Estática	integer real record array	pilas, colas, etc. con arrays
Dinámica	^integer ^real ^record ^array	pilas, colas, lista, árboles, etc. con punteros

Figura 1.10 Ejemplos de tipos de datos y de sus posibles implementaciones.

Comparando las definiciones de los TDA Lista y Lista enlazada es evidente que una forma natural de implementar una Lista en memoria primaria es mediante una Lista enlazada, y así se hace habitualmente. Sin embargo, esto no debe llevar a confundir ambos conceptos. El TDA Lista puede implementarse perfectamente mediante arreglos. De hecho, en antiguos lenguajes de programación, tales como Algol o el antiguo FORTRAN, no se disponía de implementaciones para punteros, hecho que no impedía que se implementasen listas (mediante cursores, véase detalles en [Aho, 1988], por ejemplo).

A la vista de estas consideraciones es importante reflexionar sobre las características de los TDA Arreglo, Lista enlazada y Lista. Obsérvese en primer lugar que los arreglos y las listas tienen como tipo componente cualquiera, mientras que las listas enlazadas tienen como componentes nodos. Los arreglos son estructuras estáticas, mientras que las otras dos son dinámicas. El arreglo accede a sus elementos, de forma aleatoria, mediante una indexación de los mismos, mientras que la lista y la lista enlazada lo realizan secuencialmente; y la lista enlazada necesariamente mediante el campo Enlace. La Figura 1.11 muestra una comparativa de las características de estos TDA.

	Acceso	Tipo de datos
Arreglo	Aleatorio	Cualquiera
Lista	Secuencial	Cualquiera
Lista enlazada	Secuencial	Nodos

Figura 1.11 Comparativa de los TDA Arreglo, Lista y Lista enlazada.

Seguidamente se presenta la implementación dinámica de la lista enlazada. Como se ha dicho, las variables dinámicas pueden crearse y liberarse durante la ejecución del programa.

Haciendo uso de las variables dinámicas podrán construirse estructuras de datos con un número de elementos que varía durante la ejecución del programa. El campo de enlace será un puntero y los nodos son variables referenciadas. Con esto, la declaración de tipos para los nodos en Modula2 es:

```
TYPE Ptr_Nodo = POINTER TO Nodo;
   Nodo = RECORD
             datos : Tipo_datos
             enlace : Ptr_Nodo;
          END;
```


Como se ha dicho, una lista enlazada es un conjunto organizado de componentes en los que el orden se establece mediante el campo enlace de cada nodo. Para acceder a la lista se tiene un puntero al primer nodo que se denomina puntero externo. La declaración de la lista se realizará mediante la del puntero externo, que por economía del lenguaje se le suele denominar 'lista'. En Modula2 sería:

```
VAR lista : Ptr_Nodo;
```

Para ganar claridad, también puede definirse un nuevo tipo `Tlista` y declarar las variables de este nuevo tipo:

```
TYPE
   Tlista = Ptr_Nodo;
VAR
   lista : Tlista;
```

En cualquier caso, sobre este TDA se puede realizar cualquier función necesaria gestionando adecuadamente los punteros. Seguidamente se presentan algunas de las funciones más elementales.

Inserción por la cabeza

Al ser ésta la primera operación se va a comentar detenidamente para ilustrar el manejo de punteros. Las siguientes operaciones se presentarán de forma resumida.

Para insertar un nuevo elemento en la cabeza de una lista ya existente se deberá crear un nuevo nodo, seguidamente habrá que introducir el nuevo dato a almacenar en el campo de datos y finalmente realizar los enlaces adecuados para recolocar los nodos en la lista. Es decir:

1. Crear el nuevo nodo (reservar memoria).
2. Almacenar el dato en el campo correspondiente (datos).
3. Como este nuevo nodo va a ser el primero, su campo enlace deberá apuntar al hasta ahora primer nodo.
4. Por último el enlace externo (lista), no perteneciente a ningún nodo y que apunta al primer nodo de la lista, deberá apuntar ahora al nuevo nodo.

El manejo de punteros correspondiente a esta secuencia de pasos se ilustra en la Figura 1.12. Los números indican el orden en el que debe efectuarse cada enlace, y el nodo rodeado por un círculo punteado es el nuevo nodo. Obsérvese que el orden de las acciones es fundamental. Si primero se realizase la asignación del puntero externo al nuevo nodo, cuando se desease enlazar este nuevo nodo con la lista no se podría, pues su dirección estaba en el puntero lista. La lista se ha perdido. La gestión de punteros debe realizarse de forma cuidadosa. Podría pensarse en almacenar primero todas las direcciones que posiblemente se podrán necesitar, y no prestar

mucha atención al orden de los enlaces. Sin embargo, esta 'solución' no es aceptable (aunque 'funcione'), primero porque se utilizan variables innecesarias, segundo porque la lista se manejaría de forma ineficaz y tercero porque se complica la legibilidad del código. El Programa 1.1 muestra el código necesario para la inserción por la cabeza de una lista enlazada.

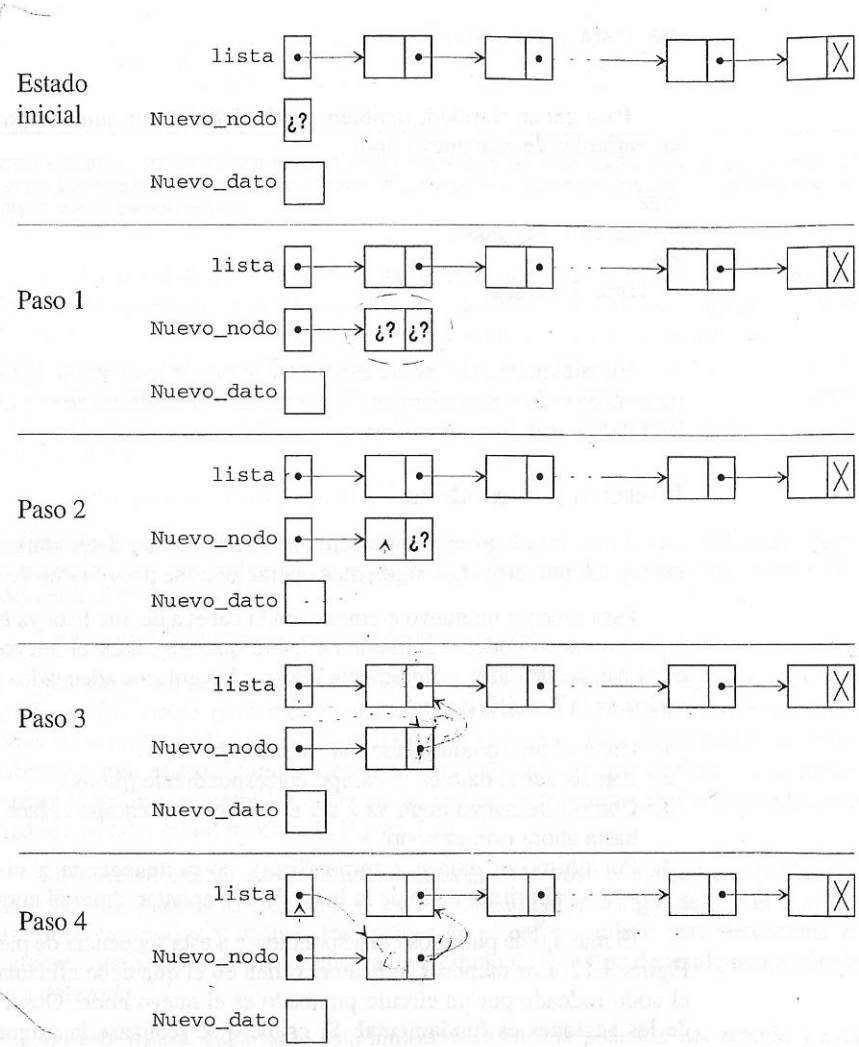

Figura 1.12 Pasos necesarios para realizar una inserción por la cabeza. Las flechas de trazo continuo indican punteros, las de trazo discontinuo indican trasvase de información correspondiente a una sentencia de asignación.

Programa 1.1 Inserción por la cabeza de una lista enlazada.

```
PROCEDURE Insertar_cabeza(VAR lista:Ptr_Nodo;nuevo_dato:Tipo_datos);
VAR
    Nuevo_nodo : Ptr_Nodo;
BEGIN
    ALLOCATE(Nuevo_nodo,SIZE(Nodo));
1   Nuevo_nodo^.datos := nuevo_dato;
2   Nuevo_nodo^.enlace :=lista;
3   lista := Nuevo_nodo;
4   END Insertar_cabeza;
```

Inserción por el final

Para insertar por el final deberemos llegar hasta él y hacer la inserción. En primer lugar se crea un nuevo nodo, se introducen los nuevos datos en el campo de datos y como a partir de ahora será el último elemento se asigna `NIL` a su campo `enlace`. Para llegar hasta el final se hace uso de un puntero auxiliar (`Actual`) que irá recorriendo la lista hasta que localice el final. Para hacer esto podemos elegir una estructura iterativa `While`, antes de la cual habrá que inicializar adecuadamente el puntero temporal (`Actual`) para que apunte al primer elemento de la lista (`Actual:=lista`). Dentro del bucle lo único que hay que hacer es *avanzar* este puntero (`Actual`) para que recorra toda la lista pasando de un elemento al siguiente (`Actual:= Actual^.enlace`). Cuando `Actual` apunte al último nodo, `Actual^.enlace` será `NIL`. Es entonces cuando hay que salir de la iteración, con lo que la condición del bucle será `Actual^.enlace<>NIL` y el último nodo estaría siendo apuntado por `Actual`.

Por último, se realiza el enlace entre el último elemento de la lista y el nuevo nodo (`Actual^.enlace:=Nuevo_nodo`).

No obstante, la condición del bucle es válida únicamente para una lista no vacía, por lo que habrá que comprobar este hecho antes de hacer una llamada a este procedimiento. Si la lista está vacía, el puntero Actual se inicializaría a NIL, y por tanto, la condición del bucle estaría basada en una variable referenciada inexistente. Sin embargo, el caso de inserción por el final de una lista enlazada vacía es el mismo que el de inserción por el principio, problema que ya está resuelto.

En la Figura 1.13 se muestran los pasos necesarios para una inserción por el final. Como en el resto de las secuencias de figuras que se muestran en este capítulo, cada una muestra la situación de la estructura después de ejecutar el paso correspondiente que está marcado con un número a la izquierda del código asociado al algoritmo. En esta Figura 1.13, el paso 4 indica la situación de los punteros después de salir del bucle `While` que aparece en el Programa 1.2. El paso etiquetado en las figuras con el número 0 indica el estado inicial antes de comenzar la ejecución de la inserción o eliminación, por lo que no aparece en el código correspondiente.

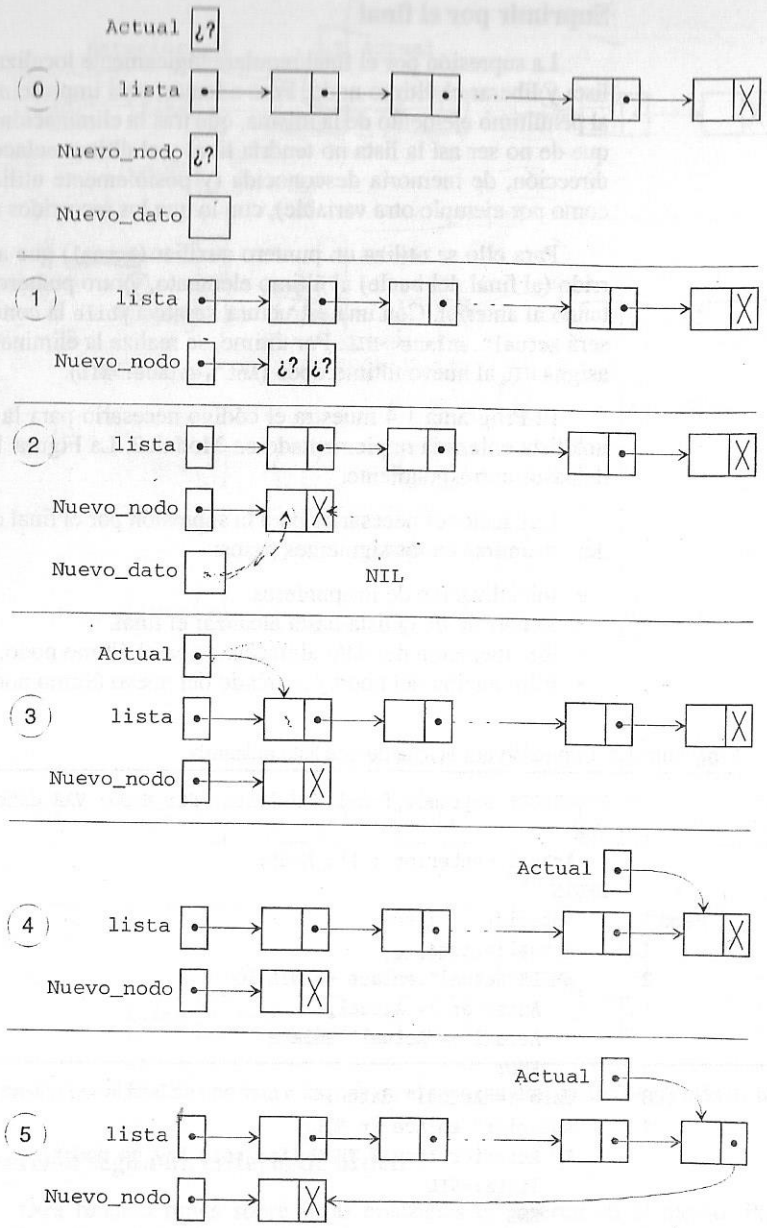

Figura 1.13 Inserción por el final de una lista enlazada.

Programa 1.2 Inserción por el final de una lista enlazada.

```
PROCEDURE Insertar_final(VAR lista:Ptr_Nodo; nuevo_dato:Tipo_datos);
VAR
   Nuevo_nodo, Actual : Ptr_Nodo;
BEGIN
   IF lista=nil THEN Insertar_cabeza(lista,nuevo_dato) ELSE
Paso 1   ALLOCATE(Nuevo_nodo,SIZE(Nodo));
2        Nuevo_nodo^.datos := nuevo_dato;
2        Nuevo_nodo^.enlace := NIL;
3        Actual := lista;
4        WHILE Actual^.enlace <> NIL DO
            Actual := Actual^.enlace;
            END;
         Actual^.enlace := Nuevo_nodo
5        END
END Insertar_final;
```

Suprimir por la cabeza

La eliminación del elemento del principio de la lista consta de cuatro pasos:

1. En primer lugar se almacena la dirección de este primer elemento, apuntado por la variable `lista`, en un puntero auxiliar (`Aux`), `Aux:=lista;`
2. En segundo lugar, habrá que asignar al parámetro formal `dato`, el valor almacenado en el elemento a eliminar.
3. Seguidamente se realiza la asignación de la lista al nuevo primer elemento (`lista:=lista^.enlace`).
4. Por último, se libera la memoria del nodo que ya no pertenece a la lista.

Debe tenerse muy en cuenta que el no realizar este último paso, es decir, suprimir únicamente mediante la asignación `lista:=lista^.enlace`, es un error muy grave. Estaría ocupándose memoria no necesaria y a lo largo de la ejecución del programa el sistema podría quedarse sin memoria libre pese a que en realidad no se utiliza la memoria que se está ocupando. Recuérdese los comentarios realizados previamente y que se ilustraban en la Figura 1.4. En el caso de las listas enlazadas el problema es especialmente grave pues puede resultar reiterativo debido a las numerosas inserciones y eliminaciones que normalmente se realizan en este tipo de estructuras.

Obsérvese también que el puntero auxiliar es imprescindible para liberar finalmente el nodo. El código necesario para realizar la supresión por la cabeza se muestra en el Programa 1.3 y la Figura 1.14 ilustra los pasos necesarios.

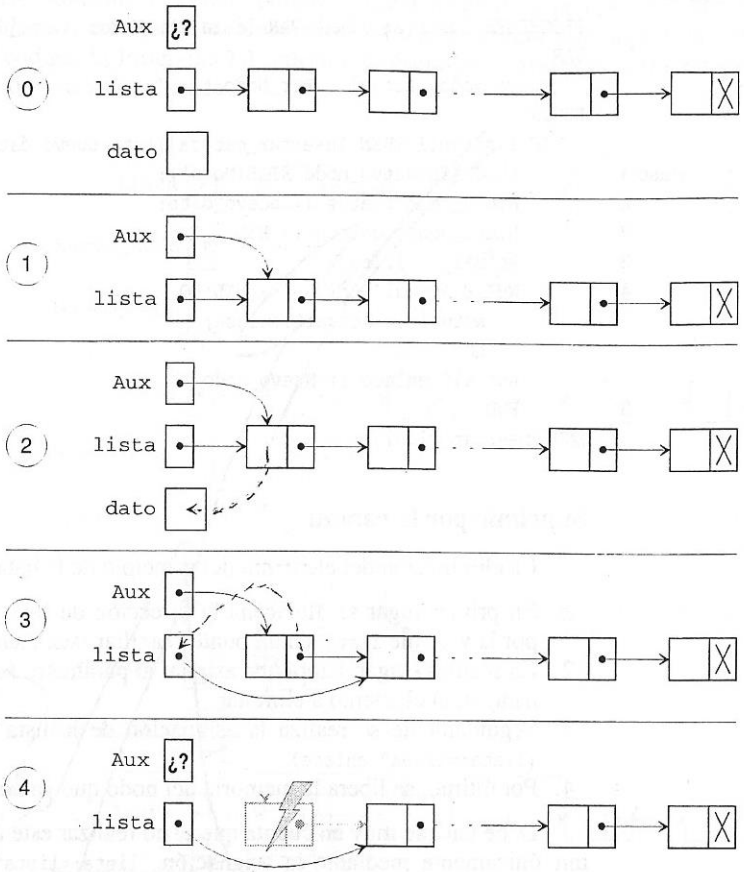

Figura 1.14 Supresión por la cabeza de una lista enlazada.

Programa 1.3 Supresión por la cabeza de una lista enlazada.

```
        PROCEDURE Suprimir_cabeza(VAR lista: Ptr_Nodo;VAR dato: Tipo_datos);
        VAR
           Aux : Ptr_Nodo;
        BEGIN
Paso 1     Aux := lista;
2          dato := lista^.datos;
3          lista:=lista^.enlace;
4          DEALLOCATE(Aux,SIZE(Nodo));
        END Suprimir_cabeza;
```


Suprimir por el final

La supresión por el final requiere lógicamente localizar la última posición de la lista y liberar el último nodo. Pero además, será imprescindible asignar el valor NIL al penúltimo elemento de la misma, que tras la eliminación pasará a ser el último, ya que de no ser así la lista no tendría final y el último enlace estaría apuntando a una dirección, de memoria desconocida (y posiblemente utilizada para otro cometido, como por ejemplo otra variable), con lo que los recorridos sobre ella no tendrían fin.

Para ello se utiliza un puntero auxiliar (`Actual`) que apuntará al final del recorrido (al final del bucle) al último elemento, y otro puntero (`Ant`) que quedará apuntando al anterior. Con una estructura iterativa `While` la condición de salida del bucle será `Actual^.enlace<>NIL`. Por último, se realiza la eliminación del último nodo y se asigna `NIL` al nuevo último nodo (`Ant^.enlace:=NIL`).

El Programa 1.4 muestra el código necesario para la supresión por el final de una lista enlazada implementado en Modula2. La Figura 1.15 muestra la secuencia de pasos correspondiente.

Las acciones necesarias para la supresión por el final de una lista enlazada pueden resumirse en los siguientes pasos:

- Inicialización de los punteros.
- Recorrido de la lista hasta alcanzar el final.
- Recuperación del dato almacenado en el último nodo.
- Eliminación del nodo y marcado del nuevo último nodo.

Programa 1.4 Supresión por el final de una lista enlazada.

```
        PROCEDURE Suprimir_final(VAR lista: Ptr_Nodo; VAR dato: Tipo_datos);
        VAR
           Actual, Anterior : Ptr_Nodo;
        BEGIN
Paso 1     Anterior := lista;
1          Actual:=lista;
2          WHILE Actual^.enlace <> NIL DO
              Anterior := Actual;
              Actual := Actual^.enlace
              END;
3          dato := Actual^.datos;
4          Anterior^.enlace := NIL;
           IF Anterior=Actual THEN (*si sólo hay un nodo*)
              lista:=NIL
              END
5          DEALLOCATE(Actual,SIZE(Nodo))
        END Suprimir_final;
```

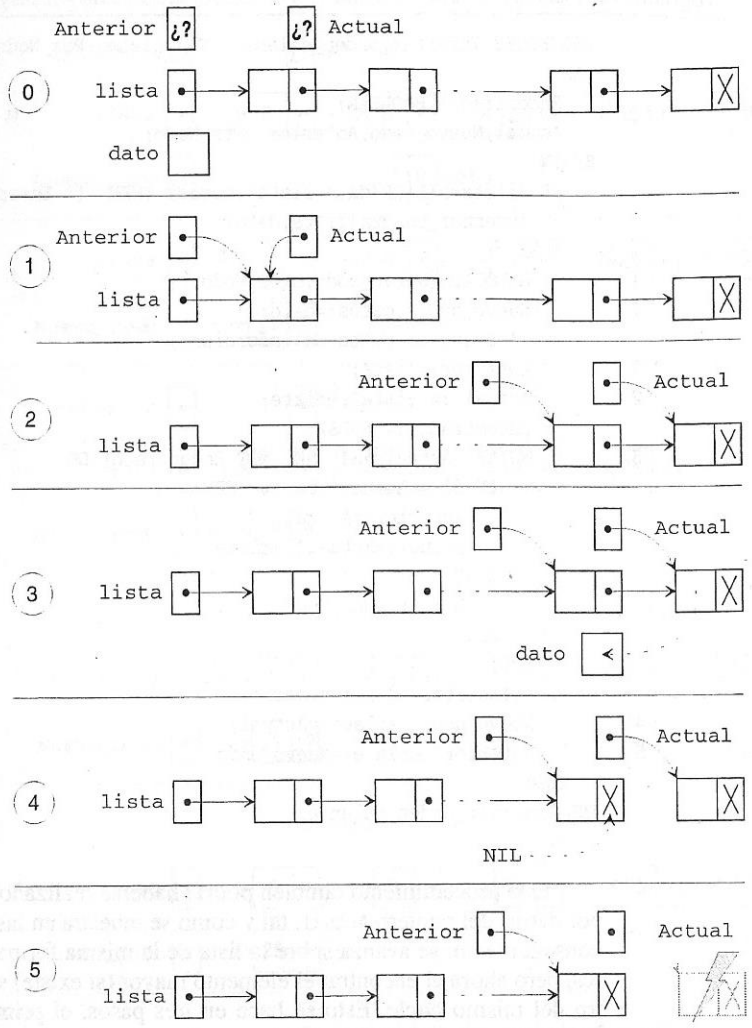

Figura 1.15 Supresión por el final de una lista enlazada en el caso general de que tenga más de un nodo.

Inserción según un criterio de orden

Otra función típica sobre listas enlazadas es insertar en el medio. Para ello habrá que hacerlo según un criterio. Supóngase que la lista se utiliza para almacenar números enteros (`Tipo_datos=INTEGER`) y que la lista está ordenada en sentido creciente. El algoritmo que se describe a continuación, inserta un nuevo número entero en su lugar y la Figura 1.16 muestra los pasos necesarios.

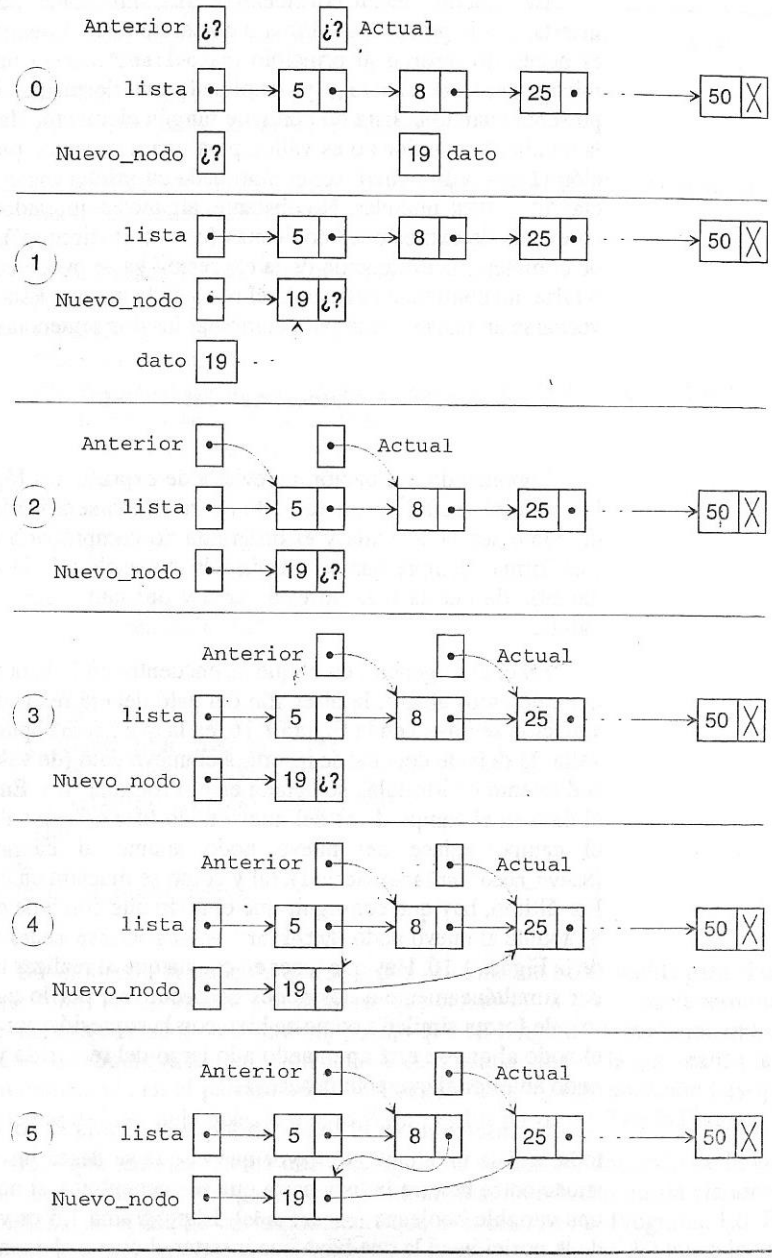

Figura 1.16 Inserción en orden en una lista enlazada delante de `Actual`.

En general, deben considerarse tres situaciones posibles: que la posición a insertar sea la primera, la última o una entre éstas. Localizada la situación en la que es necesario insertar al principio (dato<lista^.datos) puede utilizarse el procedimiento Insertar_cabeza() ya explicado anteriormente. Una situación idéntica se presenta cuando la lista no contiene ningún elemento. Hay que tener en cuenta que la condición anterior no es válida para una lista vacía, por lo que esta última situación (lista=NIL) deberá ser comprobada en primer lugar dando lugar a dos sentencias IF _ THEN anidadas. No obstante, algunos compiladores permiten la evaluación abreviada de expresiones booleanas (con 'cortocircuito'), de tal forma que si antes de completar la evaluación de la expresión ya se puede concluir que será verdadera o falsa, no continúan evaluando el resto de la misma. Esta característica puede aprovecharse en nuestro caso para combinar las dos sentencias IF _ THEN en una sola:

```
IF (lista=NIL) OR (dato<lista^.datos) THEN ...
```

Suponiendo evaluación abreviada de expresiones lógicas en esta expresión, si la lista está vacía la primera parte de la misma será verdadera independientemente de cómo sea la segunda y el programa no comprobaría la segunda condición. De esta forma, siempre que se acceda a la evaluación de la segunda parte se tendrá la garantía de que la lista no está vacía y por tanto, lista^.datos es una referencia válida.

En el caso general en el que se encuentra en la lista un elemento mayor que el que queremos añadir, la inserción del dato deberá realizarse por delante de él. Esta situación se ilustra en la Figura 1.16, en la que Actual apunta al nodo que contiene el valor 25 delante del cual se insertará el nuevo dato (de valor 19 en la figura). El procedimiento en Modula2 se detalla en el Programa 1.5. En primer lugar se almacena el dato en el campo datos del nuevo nodo (Nuevo_nodo^.datos:=dato) y se hace que el campo enlace del nuevo nodo apunte al elemento siguiente al nuevo (Nuevo_nodo^.enlace:=Actual), tal y como se muestra en el paso 4 de la Figura 1.16. Por último, hay que conseguir que el nodo que contiene el elemento anterior (valor 8) apunte al nuevo nodo (Anterior^.enlace:=Nuevo_nodo) como aparece en el paso 5 de la Figura 1.16. Hay que tener en cuenta que al realizar la inserción debemos acceder simultáneamente a dos nodos consecutivos, por lo que necesitamos dos punteros, de forma similar a como se hizo con la supresión por el final: Actual que indica el nodo al que se está apuntando a lo largo del recorrido y Anterior que apuntará al nodo anterior al que apunta Actual.

La inserción por el final habrá que realizarla si no se encuentra a lo largo de toda la lista un elemento mayor que el que se desea insertar. Para determinar esta situación se recorre la lista hasta que se encuentre, y si no es así se indica mediante una variable booleana (encontrado). El programa 1.5 es válido independientemente de la posición en la que haya que insertar el nuevo elemento.

Programa 1.5 Inserción en orden en una lista enlazada, (delante de Actual).

```
        PROCEDURE Insertar_orden_delante (VAR lista: Ptr_Nodo; dato : Tipo_datos);
        VAR
           Encontrado: BOOLEAN;
           Actual,Nuevo_nodo,Anterior: Ptr_Nodo;
        BEGIN
           IF (lista=NIL)OR(dato<lista^.datos) THEN  (* Inserción al principio *)
              Insertar_cabeza(lista,dato)
           ELSE
1             ALLOCATE(Nuevo_nodo,SIZE(Nodo));
1             Nuevo_nodo^.datos:=dato;
              (* Busca el punto de insercion*)
2             Anterior:=lista;
2             Actual := lista^.enlace;
              Encontrado := FALSE;
3             WHILE (Actual<>NIL)AND(NOT Encontrado) DO
                 IF dato>Actual^.datos THEN
                    Anterior:=Actual;
                    Actual:=Actual^.enlace
                 ELSE
                    Encontrado:=TRUE;
                 END;
              END;
              (*Insertar el nuevo nodo*)
4             Nuevo_nodo^.enlace:=Actual;
5             Anterior^.enlace:=Nuevo_nodo
           END;
        END Insertar_orden_delante;
```

Este procedimiento también podría haberse realizado insertando el nuevo nodo por detrás del puntero Actual, tal y como se muestra en las Figuras 1.17 y 1.18. Para conseguir esto, se avanza sobre la lista de la misma forma que en los casos anteriores, pero ahora al encontrar el elemento mayor (si existe) se realiza la inserción dentro del mismo bucle. Esto se hace en tres pasos, el primero consiste en copiar el contenido del nodo referenciado por el puntero de recorrido (Actual^) en el nuevo nodo (paso 4 de la Figura 1.17). A continuación, el enlace del nodo al que apunta Actual se hace que apunte al nuevo nodo (Actual^.enlace:=Nuevo_nodo), con lo que la inserción se produce efectivamente detrás del puntero Actual. Por último, se asigna el valor del nuevo dato al campo datos del nodo apuntado por Actual (Nuevo_nodo^.datos:=dato). El paso 5 de la Figura 1.17 ilustra estas dos últimas acciones y el Programa 1.6 muestra el código correspondiente.

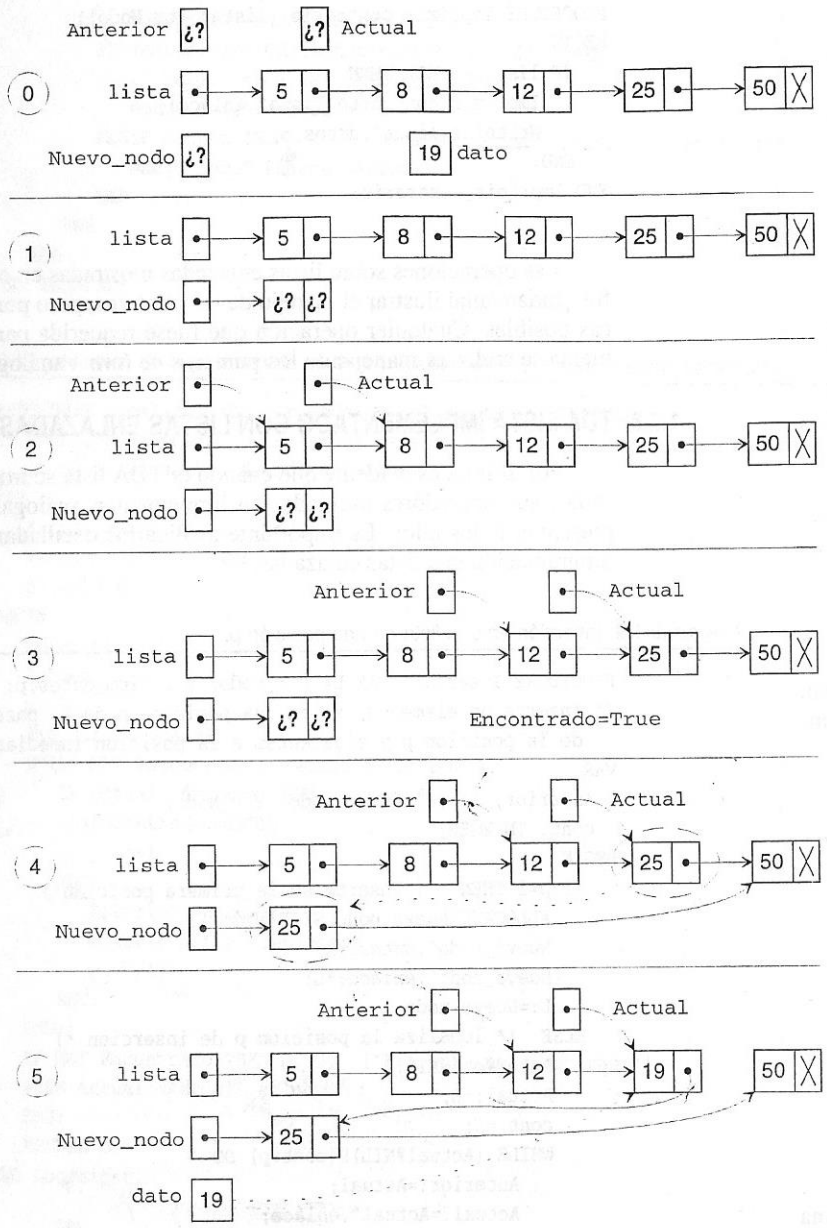

Figura 1.17 Inserción en orden en una lista enlazada (detrás de `Actual`).

Programa 1.6 Inserción en una lista enlazada según un criterio (detrás de `Actual`).

```
        PROCEDURE Insertar_orden_detras (VAR lista: Ptr_Nodo; dato : Tipo_datos);
        VAR
           Encontrado : BOOLEAN;
           Actual, Nuevo_nodo, Anterior : Ptr_Nodo;
        BEGIN
           IF (lista=NIL) OR (dato<lista^.datos) THEN  (* Insercion al principio *)
              Insertar_cabeza(lista,dato)
           ELSE
Paso 1        ALLOCATE(Nuevo_nodo,SIZE(Nodo));
2             Anterior:=lista;
2             Actual:=lista^.enlace;
              Encontrado:=FALSE;
3             WHILE (Actual<>NIL) AND (NOT Encontrado) DO (* Ins. en medio *)
                 IF dato > Actual^.datos THEN
                    Anterior:=Actual;
                    Actual:=Actual^.enlace
                 ELSE
4                   Nuevo_nodo^:=Actual^;
5                   Actual^.enlace:=Nuevo_nodo;
5                   Actual^.datos:=dato;
                    Encontrado:=TRUE;
                 END;
              END;
              IF NOT Encontrado THEN     (* Insercion al final *)
6                Nuevo_nodo^.datos:=dato;
6                Nuevo_nodo^.enlace:=NIL;
7                Anterior^.enlace:=Nuevo_nodo;
              END
           END
        END Insertar_orden_detras;
```

Sin embargo, la solución que se acaba de comentar no es válida para el caso en el que todos los elementos son menores que el nuevo, ya que el bucle terminará sin que se haya realizado la inserción, puesto que en ninguna de las iteraciones habrá dejado de cumplirse la condición `dato>Actual^.datos` que es la que realiza la inserción comentada en el párrafo anterior. Por este motivo, esta situación hay que tratarla aparte, fuera del bucle, lo que se ilustra en los pasos 6 y 7 de la Figura 1.18.

Hay que tener en cuenta que esta figura no es una continuación de la anterior, sino una alternativa a los pasos 4 y 5 de la misma. Este hecho queda claramente de manifiesto en el código correspondiente que se muestra en el Programa 1.6. En este programa se muestra que si se ejecutan los pasos 4 y 5 dentro del bucle la variable booleana `Encontrado` toma el valor `True` lo que impide que se ejecuten los pasos 6 y 7 externos al bucle.

Análogamente puede desarrollarse un procedimiento para suprimir según un criterio dado.

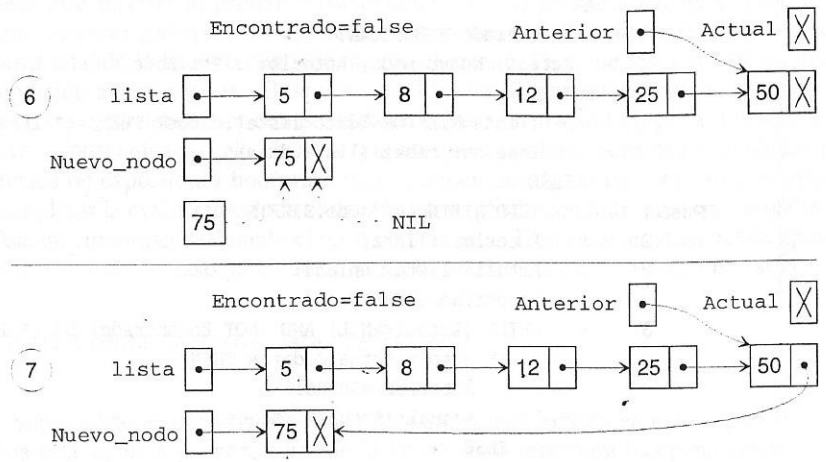

Figura 1.18 Inserción en orden en una lista enlazada por detrás del puntero de recorrido, en el caso en que el nuevo elemento sea mayor que todos los de la lista.

Imprimir una lista enlazada

Evidentemente imprimir los datos de la lista enlazada será una tarea habitual. Esto puede realizarse en orden, es decir, de manera que aparezcan los datos de principio a fin de la lista tal y como muestra el Programa 1.7

Programa 1.7 Imprimir una lista enlazada en orden.

```
PROCEDURE Imprimir_lista(lista : Ptr_Nodo);
VAR
  Aux : Ptr_Nodo;
BEGIN
  Aux := lista;
  WHILE Aux <> NIL DO
    WriteInt(Aux^.datos,7);
    Aux := Aux^.enlace;
  END;
END Imprimir_lista;
```

Sin embargo, también es posible imprimirla en orden inverso recorriendo una sola vez la lista mediante una implementación recursiva, tal y como se muestra en el Programa 1.8.

Programa 1.8 Imprimir una lista enlazada en orden inverso.

```
PROCEDURE Imprimir_contrario (lista: Ptr_Nodo);
BEGIN
  IF lista <> NIL THEN
    Imprimir_contrario(lista^.enlace);
    WriteInt(lista^.datos,6);
  END;
END Imprimir_contrario;
```

Las operaciones sobre listas enlazadas mostradas en este apartado tienen como fin fundamental ilustrar el manejo de los punteros, pero por supuesto no son las únicas posibles. Cualquier operación que fuese requerida para la solución de un problema se realizará manejando los punteros de forma análoga a la aquí presentada.

1.2.5 TDA LISTA IMPLEMENTADO CON LISTAS ENLAZADAS

Por último, es evidente que cuando el TDA lista se implementa con listas enlazadas, sus operadores asociados se implementan análogamente. Seguidamente se presentan todos ellos. Es importante analizarlos detalladamente como ejercicio de programación con listas enlazadas.

Programa 1.9 Inserción en una lista en una posición p.

```
PROCEDURE Insertar (VAR L: Ptr_Nodo; x : Tipo_datos;p: INTEGER);
(* Inserta un elemento, x, en una posicion p de L, pasando los elementos
   de la posicion p y siguientes a la posicion inmediatamente posterior *)
VAR
  Anterior, Actual,Nuevo_nodo: Ptr_Nodo;
  cont: INTEGER;
BEGIN
  IF p=1 THEN   (*Inserta en la primera posicion*)
    ALLOCATE(Nuevo_nodo,SIZE(Nodo));
    Nuevo_nodo^.datos:=x;
    Nuevo_nodo^.enlace:=L;
    L:=Nuevo_nodo
  ELSE  (* localiza la posicion p de insercion *)
    Anterior:=NIL;
    Actual:=L;
    cont:=1;
    WHILE (Actual#NIL)&(cont#p) DO
      Anterior:=Actual;
      Actual:=Actual^.enlace;
      cont:=cont+1
    END;
    IF p<cont+1  THEN (* p menor que el numero de elementos de L *)
```

```
            ALLOCATE(Nuevo_nodo,SIZE(Nodo));
            Nuevo_nodo^.datos:=x;
            Anterior^.enlace:=Nuevo_nodo;
         IF (Actual=NIL)OR((Actual=L)&(cont=p-1)) THEN
            (* insertar en la ultima posicion *)
            Nuevo_nodo^.enlace:=NIL
         ELSIF cont=p THEN          (*insertar en posicion intermedia*)
            Nuevo_nodo^.enlace:=Actual;
         END
      END
   END
END Inserar;
```

Programa 1.10 Localización de la posición de un determinado elemento pasado como parámetro.

```
PROCEDURE Localizar (L: Ptr_Nodo; x : Tipo_datos):INTEGER;
(* Localiza la posicion en la que se encuentra un elemento dado x,
   si no lo encuentra devuelve 0 *)
VAR
   Actual,Anterior: Ptr_Nodo;
   Encontrado: BOOLEAN;
   p: INTEGER;
BEGIN
   Anterior:=NIL;
   Actual:=L;
   p:=0;
   Encontrado:=FALSE;
   WHILE NOT Encontrado & (Actual#NIL) DO
      IF Actual^.datos=x THEN
         Encontrado:=TRUE;
         p:=p+1
      ELSE
         Anterior:=Actual;
         Actual:=Actual^.enlace;
         p:=p+1;
      END;
   END;
   IF NOT Encontrado THEN p:=0;  (*Elemento no encontrado*)
   ELSE Actual:=Actual^.enlace;
   END;
   RETURN p
END Localizar;
```


Programa 1.11 Obtención de la posición de un elemento dado como parámetro.

```
PROCEDURE Recuperar (L: Ptr_Nodo; VAR x : Tipo_datos;p: INTEGER;
VAR encontrado:BOOLEAN);
(* Encuentra el elemento x que esta en la posicion p, si la posicion p es
   mayor que el numero de elementos de L, devuelve encontrado a FALSE*)
VAR
   Actual: Ptr_Nodo;
   cont:INTEGER;
BEGIN
   encontrado:=FALSE;
   Actual:=L;
   cont:=1;
   WHILE (Actual#NIL)&(cont<p) DO
      Actual:=Actual^.enlace;
      cont:=cont+1
   END;
   IF (cont=p)&(Actual#NIL) THEN
      x:=Actual^.datos;
      encontrado:=TRUE;
   END
END Recuperar;
```

Programa 1.12 Eliminación del elemento de una posición dada.

```
PROCEDURE Suprimir (VAR L: Ptr_Nodo; p: INTEGER);
(*elimina de L el elemento de la posicion p*)
VAR
   Anterior,Actual: Ptr_Nodo;
   cont:INTEGER;
BEGIN
   Anterior:=NIL;
   Actual:=L;
   cont:=1;
   (*Localizar la posicion p*)
   WHILE (Actual^.enlace#NIL)&(cont<p) DO
      Anterior:=Actual;
      Actual:=Actual^.enlace;
      cont:=cont+1
   END;
   IF cont=p THEN      (*Eliminar*)
      IF cont=1 THEN    (*eliminar el primer elemento*)
         L:=Actual^.enlace
```

```
      ELSIF Actual^.enlace=NIL THEN  (*eliminar el ultimo*)
         Anterior^.enlace:=NIL
      ELSE (*intermedio*)
         Anterior^.enlace:= Actual^.enlace;
      END;
      DEALLOCATE(Actual,SIZE(Nodo))
   END
END Suprimir;
```

Programa 1.13 Eliminación de todas las apariciones de un determinado valor.

```
PROCEDURE Suprimir_dato (VAR L: Ptr_Nodo; x; Tipo_datos);
(*elimina de L todas las apariciones del elemento x*)
VAR
   Anterior,Actual,Aux: Ptr_Nodo;
   Encontrado: BOOLEAN;
BEGIN
   Anterior:=NIL;
   Actual:=L;
   WHILE Actual#NIL DO  (*busqueda de todas las apariciones*)
      Encontrado:=FALSE;
      WHILE NOT Encontrado & (Actual#NIL) DO
         IF Actual^.datos=x THEN Encontrado:=TRUE
         ELSE
            Anterior:=Actual;
            Actual:=Actual^.enlace;
         END;
      END;
      IF Encontrado THEN  (*Eliminar*)
         IF Actual=L THEN    (*eliminar el primer elemento*)
            L:=Actual^.enlace;
            Aux:=L;
         ELSIF (Actual^.enlace=NIL)&(Anterior#NIL) THEN
            (* eliminar el ultimo *)
            Anterior^.enlace:=NIL;
            Aux:=NIL;
         ELSE (*intermedio*)
            Anterior^.enlace:= Actual^.enlace;
            Aux:=Actual^.enlace;
         END;
         DEALLOCATE(Actual,SIZE(Nodo));
         Actual:=Aux;
      END;
   END;
END Suprimir_dato;
```


Programa 1.14 Vaciado completo de la lista.

```
PROCEDURE Anula (VAR L: Ptr_Nodo);
(*ocasiona que L se vacie*)
VAR
   Actual: Ptr_Nodo;
BEGIN
   IF L#NIL THEN
      Actual:=L;
      WHILE Actual^.enlace#NIL DO
         L:=Actual^.enlace;
         DEALLOCATE(Actual,SIZE(Nodo));
         Actual:=L;
      END;
      DEALLOCATE(L,SIZE(Nodo));
      L:=NIL;
   END;
END Anula;
```

Programa 1.15 Obtención del primero y último elementos de la lista.

```
PROCEDURE Primero (L: Ptr_Nodo): Tipo_datos;
(*proporciona el primer elemento de L*)
BEGIN
   IF L#NIL THEN RETURN L^.datos
   END;
END Primero;

(*****************************************************************)

PROCEDURE Fin (L: Ptr_Nodo): Tipo_datos;
(*proporciona el ultimo elemento de L*)
VAR
   Anterior, Actual: Ptr_Nodo;
BEGIN
      Anterior:=NIL;
      Actual:=L;
      WHILE Actual#NIL DO
         Anterior:=Actual;
         Actual:=Anterior^.enlace
      END;
      IF Anterior#NIL THEN RETURN Anterior^.datos
      END;
END Fin;
```

Las tablas hash clásicas tienen como inconveniente fundamental su carácter estático. Por ello, cuando el número de llaves a almacenar llena la tabla, es necesario un proceso de redimensionamiento de la tabla, cuyo coste es elevado. Para superar estos inconvenientes se introducen las tablas hash dinámicas, de aplicación inmediata en lenguajes de sistemas de desarrollo de bases de datos, como por ejemplo SQL. Un estudio detallado de estas posibilidades puede encontrarse en [Weiss, 1997].

CAPÍTULO 4

Tipos de datos abstractos dinámicos lineales

4.1 INTRODUCCIÓN

En el Capítulo 1 se presentaron los TDA básicos, bien conocidos de programación estructurada. Recuérdese que un TDA es una colección de componentes cuya organización se caracteriza por las funciones de acceso que se usan para almacenar los elementos individuales. Así por ejemplo, los arreglos unidimensionales se definen como una colección estructurada de elementos del mismo tipo, identificada con un mismo nombre, y tal que se accede a cada componente mediante un índice que indica su posición dentro de la colección. Del mismo modo, el registro es un tipo de datos estructurado compuesto por un número fijo de componentes no necesariamente del mismo tipo (denominadas campos del registro), y tal que a cada componente del registro se accede mediante un selector de campo. En Modula2, este selector está formado por el nombre del registro seguido de un punto y del nombre del identificador de campo.

Estas estructuras de datos tienen dos características reseñables. En primer lugar, son estructuras incorporadas en los lenguajes de alto nivel, es decir, estos lenguajes disponen de palabras clave para su declaración y expresiones sintácticas determinadas para el acceso a sus elementos. En segundo lugar, son TDA estáticos. Recuérdese que un TDA es estático cuando el número de elementos que lo componen es fijo y es dinámico si es variable. Para los estáticos la memoria se reserva en tiempo de compilación. Por ejemplo, los arreglos unidimensionales son TDA estáticos por su propia definición ya que en ellos el número de elementos es fijo y viene

determinado por el número de valores diferentes que puede tomar su índice. Análogamente, los registros son TDA estáticos puesto que el número de campos es fijo. Debe observarse que la naturaleza estática o dinámica de un TDA se debe a su propia definición y es independiente del uso que se haga de ella.

Otros TDA presentados en el Capítulo 1 tenían carácter dinámico. En ellos la memoria se reserva a medida que se va necesitando a lo largo de la ejecución del programa. La asignación dinámica de memoria permite construir otros tipos de datos abstractos generalizando el concepto de nodo. Así por ejemplo, en algunas aplicaciones suele ser un inconveniente el hecho de que las listas enlazadas sólo se puedan recorrer en un sentido. Si se requiere recorrer las listas en ambos sentidos puede definirse un TDA dinámico basado en un tipo similar al nodo de la lista enlazada pero que incorpore dos punteros, tal que uno apunte al predecesor y otro al sucesor. O puede ser necesario mantener, además del puntero externo a la cabeza de la lista, otro puntero al final de la misma. En conclusión, es posible definir los TDA en función de las necesidades de la aplicación.

En este capítulo y en los dos siguientes se presentan los TDA dinámicos fundamentales para la organización de datos. En este capítulo se presentarán los TDA dinámicos lineales, es decir, aquellos que mantienen una organización secuencial con un predecesor y un sucesor. En primer lugar se introducen algunos TDA similares a las listas enlazadas, como ejemplo de construcción de TDA dinámicos. Seguidamente se presentan los dos TDA dinámicos lineales más característicos: las pilas y las colas.

4.2 EJEMPLOS DE TDA DINÁMICOS LINEALES

Como ya se ha explicado anteriormente, el TDA dinámico más elemental es la lista enlazada, en la que los enlaces se realizan en un único sentido, pero generalizando el campo enlace del tipo nodo visto en el primer capítulo (Figura 1.8), puede definirse cualquier TDA dinámico con diversos enlaces. Así por ejemplo, cuando se necesita recorrer la estructura en ambos sentidos puede definirse el tipo nodo de manera que el campo enlace tenga dos punteros tal que uno apunta al sucesor y otro al predecesor de la lista. La Figura 4.1 muestra un nodo de este tipo.

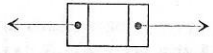

Figura 4.1 Representación esquemática de un nodo con enlace al nodo siguiente y al anterior.

De este modo puede definirse el TDA *lista doblemente enlazada* de la siguiente forma:

Definición 4.1: El TDA *lista doblemente enlazada* es una colección de nodos ordenada según su posición, tal que cada uno de ellos es accedido mediante el puntero

anterior del campo enlace del nodo siguiente y por el puntero siguiente del campo enlace del nodo anterior.

La Figura 4.2 ilustra el concepto de lista doblemente enlazada.

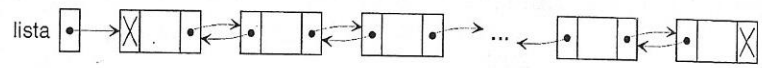

Figura 4.2 Lista doblemente enlazada.

Sobre este TDA pueden definirse funciones análogas a las realizadas sobre listas enlazadas, como insertar, suprimir, etc. El Programa 4.1 muestra la declaración de tipo en Modula2 para las listas doblemente enlazadas y la implementación de la función de inserción por el principio. Los Programas 4.2, 4.3 y 4.4 muestran respectivamente tres funciones típicas: Insertar en orden, Suprimir un elemento dado e Imprimir en sentido inverso; el procedimiento para la impresión en sentido directo es idéntico al de las listas enlazadas. Las Figuras 4.3 y 4.4 ilustran los pasos necesarios para la realización de estas dos primeras funciones básicas.

Programa 4.1 Declaración de la estructura de la lista doblemente enlazada e inserción por el principio.

```
TYPE
   tipo_datos= INTEGER;
   tipo_doble= POINTER TO nododoble;
   nododoble= RECORD
                 datos: tipo_datos;
                 sig,ant: tipo_doble
              END;
   tlista=tipo_doble;
VAR
   lista : tlista;

PROCEDURE ins_ini(VAR lista:tlista; nuevo:tipo_datos);
VAR
   aux: tipo_doble;
BEGIN
   ALLOCATE(aux, SIZE(nododoble));
   aux^.datos:=nuevo;
   aux^.sig:=lista;
   aux^.ant:=NIL;
   IF lista#NIL THEN lista^.ant:=aux; END;
   lista:=aux;
END ins_ini;
```

Programa 4.2 Inserción en orden en una lista doblemente enlazada.

```
    PROCEDURE ins_orden(VAR lista:tlista;nuevo: tipo_datos);
    VAR
      aux,ant,tempo: tipo_doble;
    BEGIN
      IF lista=NIL THEN ins_ini(lista,nuevo)
      ELSE
1       ALLOCATE(tempo,SIZE(nododoble));
2       tempo^.datos:=nuevo;
3       aux:=lista; ant:=NIL;
4       WHILE (aux#NIL) AND (nuevo>aux^.datos)DO
          ant:=aux;
          aux:=aux^.sig;
        END;
5       tempo^.sig:=aux;
6       tempo^.ant:=ant;
7       IF ant=NIL THEN lista:=tempo ELSE ant^.sig:=tempo END;
8       IF aux#NIL THEN aux^.ant:=tempo; END;
      END;
    END ins_orden;
```

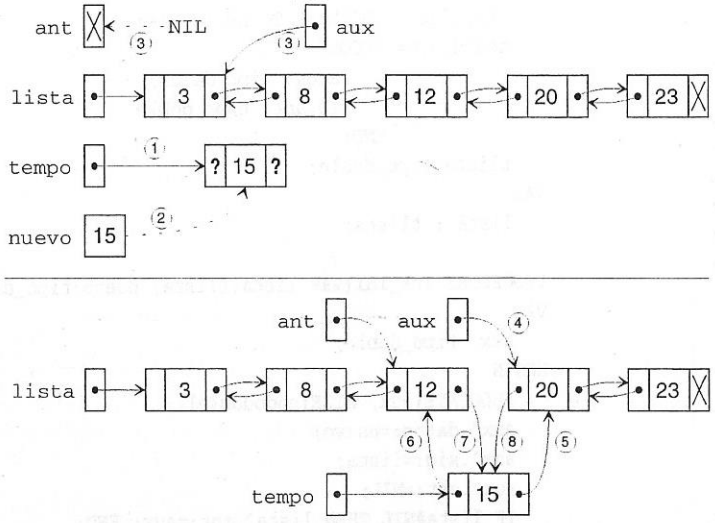

Figura 4.3 Inserción en orden en una lista doblemente enlazada. Los números muestran las acciones correspondientes a las sentencias del Programa 4.2. Estas figuras muestran el caso general de inserción por el centro pero las sentencias condicionales 7 y 8 del mencionado programa garantizan la inserción al principio y al final de la lista.

Programa 4.3 Eliminación de la primera aparición de un elemento dado en una lista doblemente enlazada.

```
    PROCEDURE sup_nodo(VAR lista:tlista; dato:tipo_datos);
    VAR
      aux: tipo_doble;
    BEGIN
1     aux:=lista;
2     WHILE (aux#NIL) & (aux^.datos#dato) DO
        aux:=aux^.sig;
      END;
      IF aux#NIL THEN   (* Si existe el nodo *)
3       IF aux^.sig#NIL THEN   (* Si tiene siguiente *)
          aux^.sig^.ant:=aux^.ant;
        END;
4       IF aux^.ant#NIL THEN    (* Si tiene anterior *)
          aux^.ant^.sig:=aux^.sig
        ELSE      (* Si es el primero  *)
          lista:=aux^.sig
        END;
5       DEALLOCATE(aux,SIZE(nododoble));
      END;
    END sup_nodo;
```

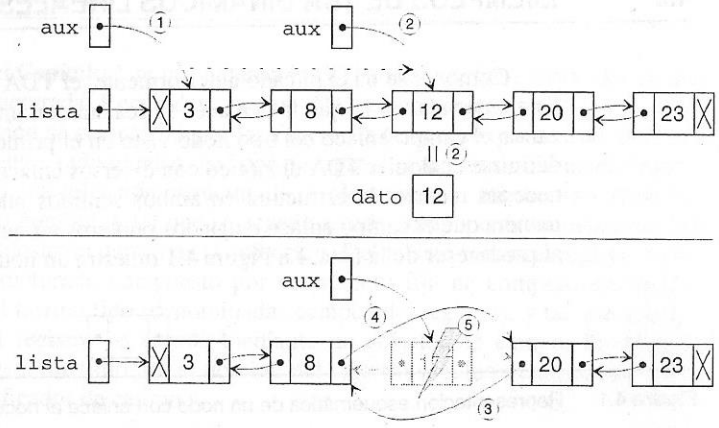

Figura 4.4 Eliminación de un nodo en una lista enlazada. Los números hacen referencia a las correspondientes sentencias del Programa 4.3. De nuevo se muestra únicamente el caso general, pero las sentencias condicionales 3 y 4 aseguran la eliminación del primer y último nodo, aunque sea el único (primero y último simultáneamente).

Programa 4.4 Impresión en sentido inverso de una lista doblemente enlazada.

```
PROCEDURE Imprimir(lista: tlista);
(*Imprime lista en orden inverso*)
VAR
  aux: tipo_doble;
BEGIN
   aux:=lista;
   IF lista#NIL THEN
   WHILE aux^.sig#NIL DO
      aux:=aux^.sig
   END;
   WriteInt(aux^.datos,4); (* último *)
   WHILE aux^.ant#NIL DO
      WriteInt(aux^.ant^.datos,4);
      aux:=aux^.ant
   END;
   END;
   WriteLn;
END Imprimir;
```

En principio, podía pensarse que las listas doblemente enlazadas son mejores que las listas enlazadas habituales ya que permiten mayor flexibilidad en su manejo y menor coste. Sin embargo, esto sólo es aparentemente. En primer lugar, es evidente que requieren más memoria, un puntero más por nodo, y como el número de nodos será elevado, éste no es un hecho despreciable. Además, el coste de las funciones sobre ellas será mayor, ya que se requerirá el doble de actualizaciones de valores de los enlaces. Por tanto, estos dos incrementos en el coste, tanto de almacenamiento como computacional, tendrán que estar justificados en función de la aplicación requerida. Supóngase, por ejemplo, que la aplicación requiere una lista y que sobre ella se suele necesitar acceder a un elemento y a su predecesor, además de las funciones típicas de una lista enlazada. En este caso no se justificaría la utilización de la lista doblemente enlazada ya que se podría implementar una función de búsqueda en la que se mantengan dos punteros, `Actual` y `Anterior` por ejemplo, con lo que el problema estaría resuelto.

Una solución similar podría realizarse si se requiriese encontrar un elemento y su quinto predecesor. Entonces, ¿cuándo se justificaría la utilización de una lista doblemente enlazada? Evidentemente cuando la necesidad de recorrer la estructura hacia los predecesores sea frecuente. Por ejemplo, si en función de un valor en un nodo se necesita volver a examinar sus predecesores.

Como ejemplo de aplicación supóngase que por un dispositivo serie se reciben cadenas de caracteres de dos periféricos tal que el primero las envía en orden y el segundo en orden inverso. El inicio de cadena se identifica mediante el carácter '1' si es del primer periférico y '2' si es del segundo y el final de cadena se produce cuando

se encuentra la marca que indica el comienzo de otra. Si por ejemplo, el periférico 1 envía las cadenas 'HOLA', 'BIEN' y 'ADIOS', y el periférico 2 envía las cadenas 'BUENA' y 'SUERTE', la secuencia de caracteres que llega será:

1	H	O	L	A	1	B	I	E	N	2	A	N	E	U	B	1	A	D	I	O	S	2	E	T	R	E	U	S

Para imprimir en pantalla los mensajes de ambos periféricos habría que recorrer la lista e ir imprimiendo los caracteres en el sentido de los sucesores siempre que el identificador de cadena encontrado fuese '1', mientras que si se encuentra un '2' habría que recorrer la lista hasta encontrar el fin de la cadena (un nuevo identificador '1' o '2') y recorrerla hacia los predecesores, imprimiendo los caracteres, hasta encontrar el principio de la cadena (identificador '2'). Esto podría resolverse también insertando las cadenas del segundo periférico en una pila. Ya se verá más adelante que este tipo de estructura devuelve la información almacenada en ella en orden inverso a como se introdujo.

Otro ejemplo para listas doblemente enlazadas podría ser una lista enlazada con fichas de libros. Supongamos que éstos están ordenados por temas. Una vez localizado un libro determinado puede ser interesante recorrer las fichas adyacentes por delante y por detrás de ésta para acceder a libros de temas similares.

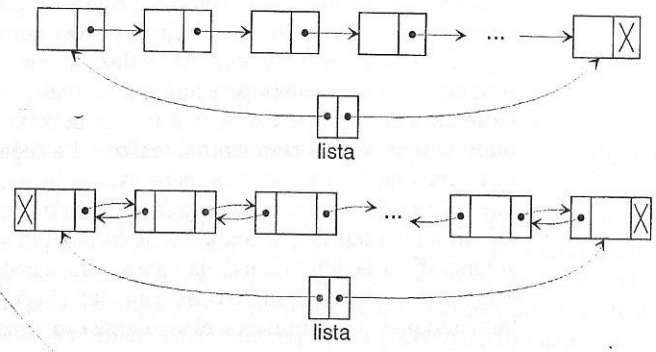

Figura 4.5 Lista enlazada y lista doblemente enlazada con puntero al final de la lista.

Al igual que se han definido las listas doblemente enlazadas puede establecerse cualquier otra que sea necesaria. Por ejemplo, en numerosas aplicaciones suele bastar una lista enlazada pero además se necesita tener un acceso rápido al último elemento. Con las listas enlazadas habituales sería necesario recorrer toda la lista para llegar al mismo. Entonces puede definirse una *lista enlazada con puntero al final de la lista* sin más que añadir un puntero externo que mantenga la dirección del último nodo. La Figura 4.5 muestra este TDA. Evidentemente las funciones realizadas sobre este TDA deben actualizar el puntero final de la lista. En esta figura se muestra también una lista doblemente enlazada con puntero al final, situación muy habi-

tual debido a que de esta forma se obtiene una estructura completamente simétrica desde ambos extremos y se aprovecha mejor la circunstancia del doble enlace de cada nodo.

Hay aplicaciones en las que resulta interesante la utilización de listas enlazadas 'con memoria'. En este tipo de listas se añade un puntero externo adicional, denominado índice, que está apuntando siempre al último elemento accedido. Esto permite detener el recorrido sobre la lista para realizar cualquier otra tarea y continuar después en el nodo donde se paró, sin necesidad de comenzar el recorrido desde el principio. Obviamente, también podemos definir listas doblemente enlazadas con memoria, con lo que después de detener el recorrido, podríamos continuar el mismo hacia delante o hacia detrás del nodo en el que se detuvo éste. En este tipo de listas, podemos definir nuevas operaciones como son:

- Avanzar y Retroceder: hacen que el puntero índice pase a apuntar al elemento siguiente o al anterior al que está apuntando en ese momento.
- Ir al Inicio o al Final. Colocar el puntero índice apuntando al primer o último elemento.
- Consultar el elemento apuntado por el índice.
- Comprobar si el índice está en el primer o el último elemento que podría utilizarse para detener un recorrido externo de la lista.

Este tipo de listas con memoria, permiten recorridos externos de la lista, de forma que éstos lo va realizando el programa que la utiliza. Esto puede ser útil por ejemplo para aplicar un determinado procesamiento a todos los elementos de la lista. Para hacer esto con una lista enlazada convencional, tenemos dos opciones: la primera sería crear una operación que lo realizase y que debería formar parte de la definición del tipo de datos, con lo que debería cambiarse esta definición si cambiase el tipo de procesamiento a realizar. La segunda opción sería ir solicitando los elementos de forma sucesiva, pero esto obligaría a que para cada elemento habría que recorrer la lista nuevamente desde el principio. Las listas enlazadas con memoria eliminan estas dos dificultades de forma que se podría cambiar el procesamiento sin modificar la definición de la lista y para acceder a cada elemento sólo habría que asignar un puntero, siempre claro está, que el acceso sea secuencial. Este tipo de tratamiento puede resumirse en el siguiente esquema:

- Poner el índice al principio (final) de la lista.
- Mientras el índice sea distinto de NIL:
  - Consultar la lista para obtener el elemento apuntado por el índice.
  - Procesar el dato.
  - Avanzar (retroceder) a la posición siguiente (anterior).

Lo que realiza este esquema también se podría hacer si una lista enlazada convencional dispusiera de una función de recorrido que admitiese como parámetro el procedimiento de procesamiento, funcionalidad que permiten la mayor parte de lenguajes de alto nivel. Pero en este caso el recorrido completo sería incondicional y no podría alterarse (parar o retroceder) en función del contenido de los elementos.

La Figura 4.6 muestra una lista doble enlazada con memoria. Obviamente, también se puede diseñar una lista simplemente enlazada con memoria, pero dada la funcionalidad de las listas con memoria es habitual hacerlo con listas doblemente enlazadas y punteros al inicio y al final.

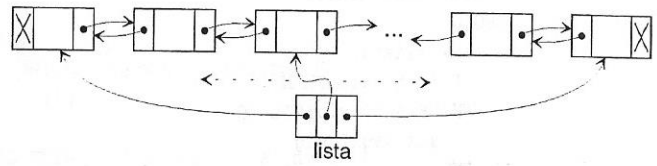

Figura 4.6 Esquema de una lista doblemente enlazada con memoria.

Una variación habitual en algunas aplicaciones de las listas doblemente enlazadas son las conocidas como *listas circulares doblemente enlazadas*. En ellas, el último nodo de la lista tiene como sucesor al primero, y el primero tiene como predecesor al último. La Figura 4.7 muestra este TDA, incluyendo el caso especial en el que la lista contiene un único nodo y es a la vez predecesor y sucesor de sí mismo. Obsérvese que en estas listas ningún nodo tiene un enlace con el valor `NIL`. Esto tiene como consecuencia que los recorridos que se realicen sobre la estructura deberán hacerse mediante un contador u observando el contenido de los distintos nodos.

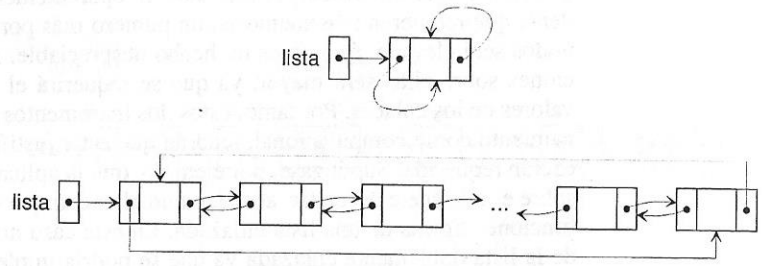

Figura 4.7 Lista circular doblemente enlazada, mostrando el caso general con varios elementos y el caso particular de elemento único. La situación de lista vacía es la habitual: `lista=NIL`.

Un tipo de estructura especialmente interesante son las listas multireferenciadas, ya que explotan especialmente las características de la asignación dinámica de memoria. En estas listas, el nodo que vertebra la estructura no contiene la información directamente sino un puntero a la misma, tal y como se muestra en la Figura 4.8. La información se almacena en una variable referenciada a la que se añade un contador que lleva la cuenta de cuántos punteros (listas) están apuntando en cada momento a la mencionada información.

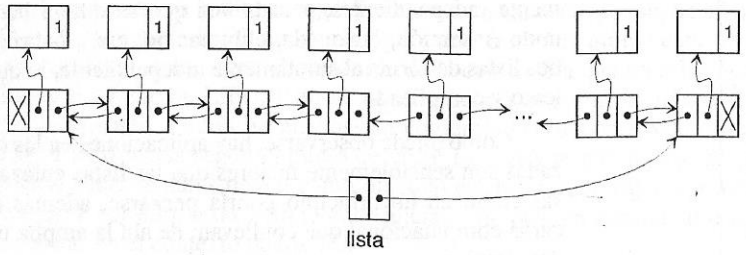

Figura 4.8 Lista doblemente enlazada donde cada nodo almacena un puntero a la información, en lugar de la información en sí. El registro referenciado incluye un contador para tener bajo control el número de punteros que en cada momento apuntan a la información que contiene.

El interés de este tipo de estructura está en aquellas aplicaciones en las que una misma información puede pertenecer simultáneamente a más de una lista. Podemos imaginar una aplicación en la que en una lista enlazada de este tipo tenemos almacenadas las fichas correspondientes a una colección de libros. Si hacemos una búsqueda, por ejemplo todos los libros de un determinado autor, podría ser interesante crear una nueva lista que contuviese sólo los libros encontrados. De esta forma podríamos operar con ella como una lista completamente independiente, sin necesidad de acceder y recorrer la lista completa, que será normalmente mucho más larga. Si ahora deseamos todos los libros que incluyan en su título la palabra 'algoritmo', crearíamos otra lista con los libros que cumplen esta segunda condición y así sucesivamente para distintas búsquedas. Una vez hecho esto podríamos realizar búsquedas combinadas, sin necesidad de recorrer la lista completa atendiendo sólo a las nuevas listas obtenidas que, en el caso general, tendrán una longitud sensiblemente inferior que la lista original.

Sin embargo, con este tipo de técnica estaríamos 'devorando' memoria de forma considerable si empleamos un tipo de lista convencional, ya que habría que duplicar la información en las distintas listas. De esta forma, si la ficha de un libro pertenece en un momento dado a siete listas, se multiplica por siete el espacio de almacenamiento necesario para esa ficha. Otro problema que se plantea es el de la inconsistencia de la información; si se modifica la ficha de un libro, habría que asegurar que la modificación se realiza en todas las listas.

Con una lista multireferenciada como la que se muestra en la Figura 4.8, se reduce considerablemente el problema del almacenamiento y se elimina el de la consistencia, ya que cuando creamos una nueva lista, no se duplica la información, que sigue siendo única, sino que basta con incrementar el contador de referencias de la misma. Este contador es necesario, porque al eliminar un elemento de una de las listas, la memoria sólo deberá ser liberada y devuelta al espacio libre, si es la única lista que contiene ese elemento. Es decir, al eliminar un elemento de la lista, lo primero que habrá que hacer será reducir su contador en una unidad y sólo si después de esto el contador vale cero se producirá una liberación de memoria. La Figura 4.9

muestra un ejemplo en el que una misma información está siendo compartida por varias listas. En esta figura se puede apreciar como el contador mencionado anteriormente lleva el control del número de referencias que tiene la información referenciada. Aunque en la Figura 4.9 se muestran listas doblemente enlazadas con punteros a los dos extremos, no tiene por qué ser así y puede adaptarse a cualquier tipo de listas.

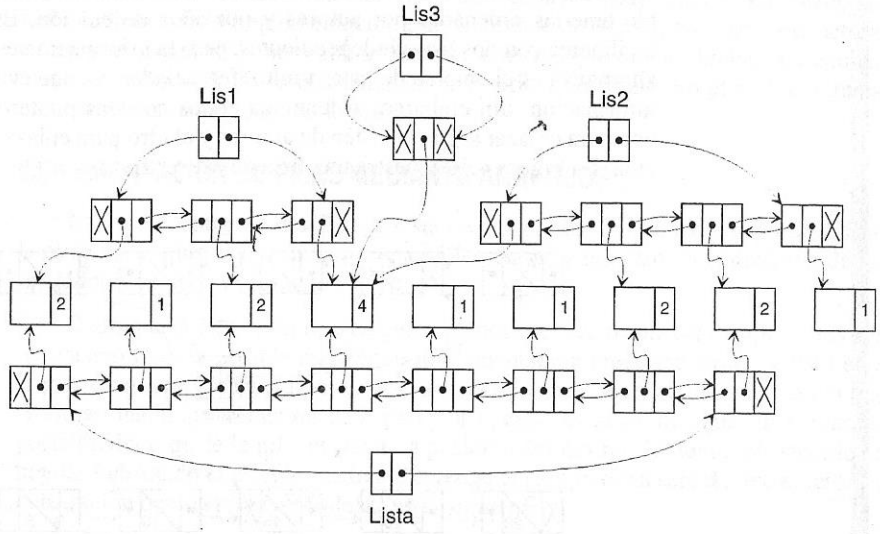

Figura 4.9 Ejemplo de lista multireferenciada con varias listas compartiendo una misma información.

Un sencillo cálculo nos permite valorar la ventaja en cuanto al espacio de almacenamiento de este tipo de listas. En el ejemplo mencionado supongamos que todos los campos necesarios para almacenar la información de cada libro necesitan 1000 bytes, cada puntero requiere 4 bytes y los enteros ocupan dos. Con una lista doblemente enlazada normal, cada nodo requeriría 1008 bytes (un campo de información y dos punteros de enlace). Con una lista multireferenciada requeriría, sin embargo, 1014 bytes (1000 de la información, 4 del puntero de referencia, 8 de los punteros de enlace y 2 del contador. El incremento de memoria necesario es <1% (6 bytes) que es un valor pequeño. Por otra parte, cada vez que un nodo que ya existe deba añadirse a una nueva lista, el espacio necesario será de sólo 12 bytes (1 puntero a la información y dos de enlace), mientras que con una lista convencional se requerirían otros 1008 bytes además del tiempo necesario para reservar la memoria y copiar la información. El interés de este tipo de lista será tanto mayor cuanto más grande sea el tamaño de las fichas y/o cuanto mayor sea el número de listas que deben compartir información. La Figura 4.9 no muestra claramente la ventaja de este tipo de listas porque los tamaños no están dibujados de forma proporcional, pero téngase en cuenta que, con los valores mencionados, que no son nada desproporcionados, un

sólo nodo convencional ocupa el equivalente a más de 100 nodos de una lista de este tipo. Si en lugar de considerar listas doblemente enlazadas, fuesen listas con un solo enlace, el ahorro sería aún mayor.

Las listas que incluyen más de un puntero en cada nodo, no siempre se enlazan simultáneamente con el anterior y el posterior según un criterio de ordenación único. Imaginemos que una misma información la queremos mantener ordenada según dos criterios distintos. En el ejemplo de las fichas de libros podría ser deseable tenerlas ordenadas por autores y por años de edición. Esto puede resolverse fácilmente con dos listas independientes, pero la información estaría duplicada. Otra alternativa es el empleo de listas multireferenciadas, ya que evitan la duplicación de información. Sin embargo, si tenemos nodos con dos punteros, podemos emplear uno para enlazar según el orden de autores y el otro para enlazar según el año de edición. La Figura 4.10 muestra una lista de este estilo.

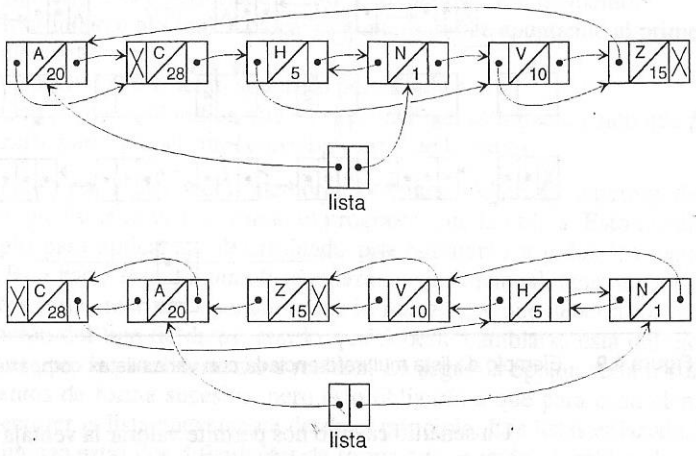

Figura 4.10 Lista con dos punteros que enlazan los elementos según dos criterios distintos. Las dos gráficas muestran la misma lista enlazada pero ordenando la representación gráfica de sus nodos según cada uno de los criterios.

Esta estructura puede generalizarse a cualquier número de punteros, tantos como criterios de ordenación distintos queramos mantener. En el caso extremo de que quisiéramos ordenar por todos los posibles campos de la información almacenada, habría un puntero por cada uno de estos campos.

Debe tenerse en cuenta que aunque sean listas doblemente enlazadas, son bastante distintas a las mostradas en los ejemplos anteriores, ya que ahora no se puede acceder desde un nodo a su predecesor, sino a dos sucesores. Cada nodo de esta lista indica quién es el nodo que le sigue según cada uno de los criterios pero no mantiene ninguna información sobre quién le precede, para ello sería necesario otro par de punteros adicional. Este tipo de listas deben considerarse como listas absolutamente independientes, y cada vez que se quiera hacer una manipulación con un nodo (inserción, búsqueda, eliminación, etc.), habría que tenerlo en cuenta en las dos listas de forma absolutamente independiente, lo que hace el tratamiento bastante lento y complicado.

Como puede observarse, hay aplicaciones en las que las listas doblemente enlazadas son sensiblemente mejores que las listas enlazadas, pero no son tan numerosas como en un principio podría pensarse, además del exceso de complejidad y carga computacional que conllevan; de ahí la amplia utilización de las listas enlazadas simples.

En conclusión, será la aplicación la que determine la necesidad de definir un TDA dinámico u otro, pero al hacerlo deben considerarse detenidamente las ventajas e inconvenientes que pueden aportar. Se deja como ejercicio la realización de las operaciones básicas sobre los distintos tipos de listas enlazadas comentadas en este apartado.

4.3 PILAS

Una pila se suele definir simplemente como una estructura de datos en la que el último elemento en entrar es el primero en salir. A estas estructuras se les denomina LIFO (*Last In, First Out*).

Evidentemente es una estructura en la que los elementos están ordenados y se añaden y suprimen únicamente por un único extremo, conocido como *tope o cabeza de la pila*. Una estructura de tipo pila responde fielmente al concepto cotidiano de pila al que estamos muy acostumbrados en la vida real. Un ejemplo típico es una pila de cajas. Sólo se puede coger la caja de arriba de la pila, y sólo ahí se puede dejar una caja. Obsérvese que esta disposición de las cajas es una pila porque se actúa así sobre ella. Efectivamente, podría pensarse en suprimir una caja que no es la primera o se podría ser lo suficientemente habilidoso como para introducir una entre dos ya colocadas. En este caso la disposición de las cajas no es una estructura pila, será otra cosa (una lista enlazada, por ejemplo).

Como puede observarse por otro lado, la estructura pila es dinámica ya que el número de elementos que la componen es variable. Por ello una pila puede encontrarse definida en la literatura, usualmente en la dedicada a principios básicos de programación estructurada, como una lista enlazada en la que el último elemento en entrar es el primero en salir.

Sin embargo, deben realizarse algunas consideraciones acerca de esta sencilla definición. La definición inicial de la pila sólo resalta la característica más sobresaliente del TDA pila, su acceso LIFO, y definirla como una lista enlazada sólo indica una manera de implementarla. Es decir, ninguna de ellas es una definición rigurosa de la pila como TDA. Esto no sólo tiene importancia formal o académica. Las definiciones de los TDA precisan los términos y las operaciones que pueden realizarse sobre los datos evitando ambigüedades. En el ejemplo de las pilas de cajas, ¿podría