2 TADes LINEALES.

A partir de ahora nos vamos a centrar en tipos abstractos de datos que no tienen sentido por sí solos, sino que establecen unas conexiones entre otros elementos que son los que contienen la información que a nosotros nos interesa. Así, por un lado se tendrán los elementos que queremos estructurar, y por otro el TAD que nos dé la estructura que en cada momento se desee: Pila de elementos, en la que en cada momento sólo se puede extraer el último que se metió; Cola de elementos, en la que los elementos que entran se van poniendo en cola para poder salir; Anillo de elementos, que no es más que una cola sin fin en la que el primer elemento espera detrás del último, etc.. Como vemos, en todos ellos se trabaja con Elementos de información, y lo que varía es la forma de distribuirlos, lo que da lugar a los diferentes TADes. Por supuesto, podemos hacer que el tipo base de un TAD, sea a su vez otro TAD.

En concreto, en este capítulo vamos a ver aquellos TADes en los que la relación que se establece entre los elementos de información es de sucesión y precedencia (independientemente de cual sea el elemento de información), de ahí el nombre de Tipos Abstractos de Datos Lineales.

Una estructura lineal es aquella en la que cada elemento tiene un solo elemento sucesor, (excepto el último, que no tiene sucesor), y un solo predecesor, (excepto el elemento primero). Así, un árbol o un grafo no conforman estructuras lineales, puesto que, en un árbol, un nodo puede tener varios sucesores o hijos, y en un grafo, un nodo determinado puede tener varios padres y/o varios hijos.

Desde un punto de vista formal, los tipos lineales se corresponden básicamente con una sucesión finita de elementos a_1, a_2, a_3, ... a_n , donde $a_i \in E$, siendo E el tipo base (el mismo para todos los elementos).

En todos los TADes Lineales, existe una sucesión distinguida, que se corresponde con la sucesión vacía, o sea, con n=0, y que la usaremos como operación generadora constante, siendo por tanto una forma o término canónico por sí sólo. El resto de las sucesiones se formarán a partir de esta sucesión vacía, mediante los operadores correspondientes de incorporación de nuevos elementos del tipo base, existiendo al menos, las siguientes tres operaciones básicas para todo TAD Lineal, pero cuyo comportamiento será diferente dependiendo del TAD de que se trate:

- **Agregar** un elemento a la sucesión.
- **Eliminar** un elemento de la sucesión.
- **Consultar** un elemento de la sucesión. El elemento que en cada momento se consulta suele ser el mismo que aquél que en cada momento se puede eliminar. Será una operación parcial, de manera que no podrá actuar con la sucesión vacía.

Como se ha dicho, el comportamiento de estas operaciones, y las posibles operaciones adicionales que se pueden derivar de dichos comportamientos, establecen las diferencias entre los distintos tipos lineales.

Los TADes Lineales, así como los Árboles y Grafos que se verán más adelante, presentan unos comportamientos característicos independientes del tipo base de los elementos que integran; esta independencia posibilita la especificación genérica del TAD, con independencia de cual sea

el tipo base.

No obstante, si se quieren incorporar al TAD operaciones de equivalencia o de orden entre estructuras de dicho TAD, será necesario que el tipo base también las posea. P. ej., si queremos crear la operación <= entre listas, pues, en general, será necesario que el tipo base de tales listas posea una relación de orden completa, (que el tipo base sea números, y no colores, p.ej., ya que entre estos últimos no cabe orden alguno, al menos a priori).

2.1 Especificaciones de TADes lineales.

Las Pilas son una de las estructuras de datos más primitivas y más antiguas en la ciencia de los ordenadores. A pesar de ser tan simple, es esencial en todos los compiladores, SS.OO., y en la práctica totalidad de las aplicaciones.

Podemos representar una pila como una lista lineal que crece y decrece por el mismo extremo, hecho que se resume en la siglas LIFO (Last In First Out). Así, el último elemento que se incorpora a la estructura, es el único que se puede consultar y eliminar.

La especificación algebraica es la siguiente:

tipo Pila
dominios Elemento, Lógico
generadores
Crear: ⟶ Pila
Apilar: Elemento × Pila ⟶ Pila
constructores
Desaplilar: Pila ⟶ Pila
selectores
Cima: Pila ⟶ Elemento
Es_Vacia: Pila ⟶ Lógico
precondiciones p:Pila
Cima(p): not Es_Vacía(p)
ecuaciones p:Pila, e:Elemento
Desapilar(Crear) == Crear
Desapilar(Apilar(e, p)) == p
Es_Vacía(Crear) == V
Es_Vacía(Apilar(e, p)) == F
Cima(Apilar(e, p)) == e

2.1.1 Estructuras acotadas.

Tal y como hemos visto la definición anterior, nos puede surgir el problema de que, como la capacidad de memoria de nuestro ordenador es limitada, se pueda crear una estructura Pila de un tamaño tal, que no pueda ser contenida físicamente en memoria. Por tanto, es necesario poner

un límite o **cota superior** a la cantidad de elementos del tipo base que se pueden almacenar en un TAD, (p.ej., en una pila). Esto implica que las funciones generadores que hacen extender el tamaño de la estructura, pasen a ser operaciones parciales.

Así pues, para la especificación de los tipos correspondientes a estructuras lineales acotadas hay que ampliar la especificación formal con las especificaciones siguientes:

- Una constante TMáx, que representa el tamaño máximo de la estructura. La unidad empleada suele ser el elemento del tipo base.
- Una función Longitud, o Tamaño, que determina el tamaño de una estructura. La unidad en que se expresa ha de ser congruente con la constante anterior.
- Un predicado (función que retorna un valor lógico) Está_Lleno, que comprueba si una estructura tiene el tamaño máximo o no, esto es, si la Longitud() coincide con TMáx.

Así, la especificación de la pila pasa a ser:

tipo Pila
dominios Elemento, Lógico, *Natural*
generadores
Crear: $\longrightarrow$ Pila
Apilar: Elemento × Pila $\not\longrightarrow$ Pila
constructores
Desaplilar: Pila $\longrightarrow$ Pila
selectores
Cima: Pila $\not\longrightarrow$ Elemento
Es_Vacia: Pila $\longrightarrow$ Lógico
Longitud: Pila $\longrightarrow$ *Natural*
Esta_Llena: Pila $\longrightarrow$ *Lógico*
operaciones
TMáx: $\longrightarrow$ *Natural*
precondiciones *e:Elemento*, p:Pila
Cima(p): not Es_Vacía(p)
Apilar(e, p): not Está_Llena(p)
ecuaciones p:Pila, e:Elemento
TMáx == $suc^n(0)$
Desapilar(Crear) == Crear
Desapilar(Apilar(e, p)) == p
Es_Vacía(p) == =(Longitud(p), 0)
Cima(Apilar(e, p)) == e
Longitud(Crear) == 0
Longitud(Apilar(e, p)) == suc(Longitud(p))
Está_Llena(p) == =(Longitud(p), TMáx)

Nótese que tras la incorporación del selector Longitud(), es más fácil la especificación de la operación Es_Vacía(). Véase también que NO es lo mismo Es_Vacía() que not(Está_Llena):

$$Es\_Vacía(p) \neq \neg (Está\_Llena(p))$$

2.1.2 Colas.

La cola es una abstracción muy útil, que posee muchos paralelismos en la vida real: colas de espera en bancos, supermercados, etc.. Ello hace que su importancia sea crucial en simulaciones. Computacionalmente hay muchos casos en los que aparecen colas de espera: Procesos que esperan a ser ejecutados en sistemas multitarea, trabajos de impresión que comparten la misma impresora, procesos batch de cálculo intensivo, etc..

Este tipo de estructura se caracteriza por su forma de explotación tipo FIFO (First In First Out), que indica que el primer elemento que se incorporó a la estructura, será el primero que salga en una operación de extracción (desapareciendo ya de ella, y pasando su sucesor a ser el primero en espera), y el único que se puede consultar en un momento dado.

La especificación algebraica es la siguiente:

tipo Cola
dominios Elemento, Lógico
generadores
Crear: $\longrightarrow$ Cola
Insertar: Elemento × Cola $\longrightarrow$ Cola
constructores
Extraer: Cola $\longrightarrow$ Cola
selectores
Frente: Cola $\not\longrightarrow$ Elemento
Es_Vacía: Cola $\longrightarrow$ Lógico
precondiciones c:Cola
Frente(c): not(Es_Vacía(c))
ecuaciones e, e_1, e_2:Elemento, c:Cola
Es_Vacía(Crear) == V
Es_Vacía(Insertar(e, c)) == F
Extraer(Crear) == Crear
Extraer(Insertar(e, Crear)) == Crear
Extraer(Insertar(e_1, Insertar(e_2, c))) == Insertar(e_1, Extraer(Insertar(e_2, c)))
Frente(Insertar(e, c)) == SI Es_Vacía(c) ENTONCES
e
SINO
Frente(c)

Aquí, se introduce una nueva forma de ecuación más relajada, en la que un término se puede reducir de diferentes formas SEGÚN que cumpla unas determinadas características.

Análogamente, tenemos el problema más realista de qué hacer con una memoria limitada. Así, suponiendo que la cota superior viene dada por TMáx:

tipo Cola
dominios Elemento, Lógico, Natural
generadores
Crear: $\longrightarrow$ Cola
Insertar: Elemento × Cola $\not\longrightarrow$ Cola
constructores
Extraer: Cola $\longrightarrow$ Cola
selectores
Frente: Cola $\not\longrightarrow$ Elemento
Es_Vacía: Cola $\longrightarrow$ Lógico
Longitud: Cola $\longrightarrow$ Natural
Está_Llena: Cola $\longrightarrow$ Lógico
operaciones
TMáx: $\longrightarrow$ Natural
precondiciones c:Cola
Frente(c): not(Es_Vacía(c))
Insertar(e, c): not(Está_Llena(c))
ecuaciones e, e_1, e_2:Elemento, c:Cola
TMáx == $suc^n(0)$
Longitud(Crear) == 0
Longitud(Insertar(e, c)) == suc(Longitud(c))
Es_Vacía(c) == =(Longitud(c), 0)
Está_Llena(c) == =(Longitud(c), TMáx)
Extraer(Crear) == Crear
Extraer(Insertar(e, Crear)) == Crear
Extraer(Insertar(e_1, Insertar(e_2, c))) == Insertar(e_1, Extraer(Insertar(e_2, c)))
Frente(Insertar(e, c)) == SI Es_Vacía(c) ENTONCES
e
SINO
Frente(c)

También disponemos de las colas con prioridad. En éstas, el tipo base consta de al menos un elemento que posee una relación de orden total. Los objetos del tipo base se introducen de igual forma que en los casos anteriores; la diferencia aparece en el caso de la extracción y en la consulta del siguiente objeto a extraer. En estos casos, el elemento que se saca no es el primero que se introdujo, sino el menor de los que se han introducido según la relación de orden total de que se hablaba al principio. Si hay varios elementos menores, con el mismo valor, el que se saca es aquél que se introdujo primero.

tipo Cola
dominios Elemento, Lógico
generadores
Crear: $\longrightarrow$ Cola
Insertar: Elemento × Cola $\longrightarrow$ Cola

constructores

Frente: Cola $\not\longrightarrow$ Cola

Extraer: Cola $\longrightarrow$ Cola

selectores

Es_Vacía: Cola $\longrightarrow$ Lógico

precondiciones c:Cola

Frente(c): not(Es_Vacía(c))

ecuaciones e, e_1, e_2:Elemento, c:Cola

```
Es_Vacía(Crear) == V
Es_Vacía(Insertar(e, c))
Frente(Insertar(e, c)) ==
        SI Es_Vacía(c) ENTONCES e
        SI NO  SI Frente(c) <= e ENTONCES Frente(c)
               SI NO e
Extraer(Crear) == Crear
Extraer(Insertar(e, c)) ==      SI Es_Vacía(c) ENTONCES
                                        c
                                SI NO  SI Frente(c) <= e ENTONCES
                                               Insertar(e, Extraer(c))
                                       SI NO
                                               c
```

2.1.3 Anillos.

Un anillo es una secuencia de elementos dispuestos de forma circular. Los elementos pueden añadirse y eliminarse desde un único punto llamado **cabeza del anillo**. El círculo de elementos se puede rotar en ambos sentidos, de manera que los diferentes elementos van pasando por la cabeza del anillo.

Un anillo es, pues, una estructura en la que todo elemento tiene un sucesor y un predecesor, y hay una posición distinguida llamada cabeza del anillo.

Estructuras con características parecidas se presentan a distintos niveles, desde las redes de ordenadores con configuraciones de anillo y comunicación mediante *paso de mensajes* hasta las interfaces de usuario, en los intérpretes de comandos, que suelen utilizar un buffer de entrada estructurado como un anillo de líneas que se puede recorrer para recuperar líneas introducidas con anterioridad.

Las características de estas estructuras permiten que cualquier anillo se pueda considerar una estructura lineal cuyo primer elemento es el situado en la posición actual y continúa en el sentido del recorrido terminando en la posición anterior a la posición actual, e interpretar la operación de recorrido (a dcha.), como la generación de una nueva lista obtenida por un desplazamiento de los elementos de la lista argumento, con lo que cualquier valor de esta abstracción se podrá construir utilizando únicamente los constructores básicos de los tipos lineales. En concreto, la caracterización de este tipo de estructura se podrá dar mediante las

operaciones.

tipo Anillo
dominios Anillo, Elemento
generadores
Crear: $\longrightarrow$ Anillo
Insertar: Elemento × Anillo $\longrightarrow$ Anillo
constructores
Eliminar: Anillo $\longrightarrow$ Anillo
Rotar_Dcha: Anillo $\longrightarrow$ Anillo
Rotar_Izqd: Anillo $\longrightarrow$ Anillo
selectores
Cabeza: Anillo $\not\longrightarrow$ Elemento
Es_Vacío: Anillo $\longrightarrow$ Lógico
auxiliares
A_la_cola: Elemento × Anillo $\longrightarrow$ Anillo
Elim_Cola: Anillo $\longrightarrow$ Anillo
Cola: Anillo $\not\longrightarrow$ Elemento
precondiciones a:Anillo
Cabeza(a): not(Es_Vacío(a))
Cola(a): not(Es_Vacío(a))
ecuaciones a:Anillo, e, e_2:Elemento
Eliminar(Crear) == Crear
Eliminar(Insertar(e, a)) == a
Es_Vacío(Crear) == V
Es_Vacio(Insertar(e, a)) == F
Cabeza(Insertar(e, a)) == e
Rotar_Dcha(Crear) == Crear
Rotar_Dcha(Insertar(e, a)) == A_la_cola(e, a)
A_la_cola(e, Crear) == Insertar(e, Crear)
A_la_cola(e, Insertar(e_2, a)) == Insertar(e_2, A_la_cola(e, a))
Rotar_Izqd(Crear) == Crear
Rotar_Izqd(Insertar(e, c)) == Insertar(Cola(Insertar(e, c)), Elim_Cola(Insertar(e, c)))
Cola(Insertar(e, a))) == SI Es_Vacío(a) ENTONCES
e
SINO
Cola(a)
Elim_Cola(Crear) == Crear
Elim_Cola(Insertar(e, a)) == SI Es_Vacío(a) ENTONCES
Crear
SINO
Insertar(e, Elim_Cola(a))

Insertar() y A_la_cola() son operaciones recíprocas. Lo mismo ocurre con Eliminar() y

Elim_cola(), y con Rotar_Dcha() y Rotar_Izqd().

NOTA: También podemos considerar el TAD Cola Doble. Esta es una cola en la que se pueden insertar y eliminar elementos tanto por un extremo por el otro. De la descripción algebraica del anillo, se puede extraer que una cola doble no es más que un anillo en el que se han eliminado las operaciones Rotar_Dcha y Rotar_Izqd.

2.1.4 LISTA FUNCIONAL.

La *lista* es una secuencia lineal formada por cualquier cantidad de elementos del mismo tipo base, junto con las operaciones necesarias para tratarlos.

La lista es uno de los TADes más fundamentales y versátiles, y abundan en situaciones de la vida real, así como en computación. Por ejemplo, un fichero de texto en Modula-2 consiste en una lista de caracteres. Las operaciones particulares sobre dicha lista, la convierten en un TAD específico: el TAD fichero secuencial.

Por tanto, podemos considerar que todos los TADes lineales que hemos visto, son particularizaciones de la lista, cuya definición es la más general de todas.

Vamos a ver dos formas de representar una lista: la forma funcional, y la forma posicional. En la funcional, la posición de un elemento dentro de la secuencia viene dada por la cantidad de constructores que se hallan efectuado antes que él, y, por tanto, cada posible estado en que se puede encontrar una lista, tiene un sólo representante canónico, o sea, una sola forma canónica que la representa.

En la forma posicional, la inserción de un elemento *a posteriori* puede dar lugar a una reordenación en los elementos ya introducidos en la lista, de manera que una misma lista puede tener varias representaciones canónicas.

En este apartado veremos la lista funcional:

tipo Lista
dominios Elemento, Lógico
generadores
Crear: $\longrightarrow$ Lista
Construir: Elemento × Lista $\longrightarrow$ Lista
constructores
Cola: Lista $\longrightarrow$ Lista
selectores
Cabeza: Lista $\not\longrightarrow$ Elemento
Es_Vacia: Lista $\longrightarrow$ Lógico
precondiciones l:Lista
Cabeza(l): not Es_Vacía(l)
ecuaciones l:Lista, e:Elemento

Cola(Crear) == Crear
Cola(Construir(e, l)) == l
Es_Vacía(Crear) == V
Es_Vacía(Construir(e, l)) == F
Cabeza(Construir(e, l)) == e

A esta definición básica (nótese que coincide plenamente con el concepto de Pila), podemos añadirle todas las operaciones que se deseen; en cualquier caso, el TAD obtenido se seguirá llamando Lista. P.ej.:

operaciones
Longitud: Lista ⟶ Natural
Concatenar: Lista × Lista ⟶ Lista
Invertir: Lista ⟶ Lista
Ordenar_Lista: Lista ⟶ Lista
auxiliares
Insertar_en_orden: Elemento x Lista ⟶ Lista
ecuaciones
Longitud(l) == SI Es_Vacía(l) ENTONCES 0
SI NO suc(Longitud(Cola(l))
Concatenar(l_1, l_2) ==
SI Es_Vacía(l_1) ENTONCES l_2
SI NO Construir(Cabeza(l_1), Concatenar(Cola(l_1), l_2)
Invertir(l) ==
SI Es_Vacía(l) ENTONCES l
SI NO Concatenar(Invertir(Cola(l)),Construir(Cabeza(l), Crear))
Ordenar(l) ==
SI Es_Vacía(l) ENTONCES l
SI NO Insertar_en_orden(Cabeza(l), Ordenar_Lista(Cola(l)))
Insertar_en_orden(e, l) ==
SI Es_Vacía(l) ENTONCES Construir(e, Crear)
SI NO SI <=(e, Cabeza(l)) ENTONCES Construir(e, l)
SI NO Construir(Cabeza(l), Insertar_en_orden(e, Cola(l)))

Veamos un pequeño caso en el que analizaremos el papel de las ecuaciones como elemento para la reducción. Consideremos las siguientes sentencias de Modula-2:

(S_1) s:=Crear;
(S_2) s:=Construir(3, Construir(4, s));
(S_3) x:=Cabeza(s);
(S_4) t:=Cola(s);
(S_5) y:=Cabeza(t);

Para analizar el contenido de cada variable al final de esta secuencia, lo haremos pero al final de cada una de las sentencias que la componen:

(S_1) s=Crear
(S_2) s=Construir(3, Construir(4, Crear))
(S_3) x=Cabeza(Construir(3, Construir(4, Crear)))=3
(S_4) t=Cola(Construir(3, Construir(4, Crear)))=Construir(4, Crear)
(S_5) y=Cabeza(Construir(4, Crear)=4

Esta especificación es precisa y no ambigua, y, por tanto correcta; además es completa en el sentido de que todas las operaciones quedan perfectamente definidas, y no dejan lugar a dudas sobre la tarea que desempeñan. Si faltase cualquier axioma, el programador no sabría el comportamiento de la operación en ese caso particular.

2.1.5 LISTA POSICIONAL

Vamos a ver ahora la lista posicional, en la que cada vez que se inserta un elemento se especifica la posición, dentro de la lista ya existente, en que se quiere introducir.

tipo Lista
dominios Lógico, Natural, Elemento
generadores
Crear : $\longrightarrow$ Lista
Insertar: Lista × $\mathbb{N}$ × Elemento $\not\longrightarrow$ Lista
constructores
Eliminar: Lista × $\mathbb{N}$ $\not\longrightarrow$ Lista
Cambiar: Lista × Elemento × $\mathbb{N}$ $\not\longrightarrow$ Lista
selectores
Elemento: Lista × $\mathbb{N}$ $\not\longrightarrow$ Elemento
Es_Vacia: Lista $\longrightarrow$ B
Longitud: Lista $\longrightarrow$ $\mathbb{N}$
precondiciones l : Lista p : $\mathbb{N}$ e : Elemento
Insertar(l, p, e) : ($1 \le p \le$ Longitud(l)+1)
Eliminar(l,p) : ($1 \le p \le$ Longitud(l))
Cambiar(l, e, p) : ($1 \le p \le$ Longitud(l))
Elemento(l, p) : ($1 \le p \le$ Longitud(l))
ecuaciones
Longitud(Crear()) = 0
Longitud(Insertar(l,p,e)) = Longitud(l) + 1
Es_Vacia(Crear) = V
Es_Vacia(Insertar(l, p, e)) = F
Eliminar(Insertar(l,p_1,e),p_2) = SI p_1 = p_2 ENTONCES
l
SI NO SI p_1 <= p_2 ENTONCES
Insertar(Eliminar(l,p_2-1),p_1,e))
SI NO
Insertar(Eliminar(l,p_2),p_1-1,e))

Cambiar(Insertar(l,p_1,e_1),e_2,p_2) = SI p_1 = p_2 ENTONCES
Insertar(l,p_1,e_2)
SI NO SI p_1 <= p_2 ENTONCES
Insertar(Cambiar(l,e_2,p_2-1),p_1,e_1))
SI NO
Insertar(Cambiar(l,e_2,p_2),p_1,e_1))

Elemento(Insertar(l,p_1,e),p_2) = SI p_1 = p_2 ENTONCES
e
SI NO SI p_1 <= p_2 ENTONCES
Elemento(l,p_2-1)
SI NO
Elemento(l,p_2))

Como se dijo, este tipo de lista tiene la desventaja de que permite especificar el mismo objeto con varios términos canónicos. esto también puede verse como una lista con memoria, que recuerda el orden en que fueron introducidos los elementos, lo cual en la mayoría de los casos es irrelevante. P.ej.:

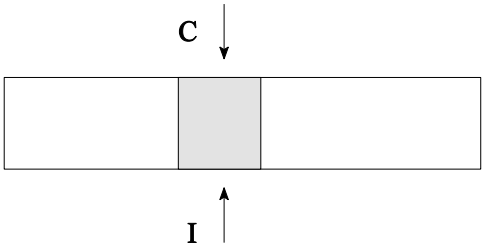
Insertar(Insertar(Insertar(Insertar(Crear, 1, 'a'), 1, 'l'), 1, 'o'), 1, 'H')

se refiere a la misma lista que:

Insertar(Insertar(Insertar(Insertar(Crear, 1, 'H'), 2, 'o'), 3, 'l'), 4, 'a')

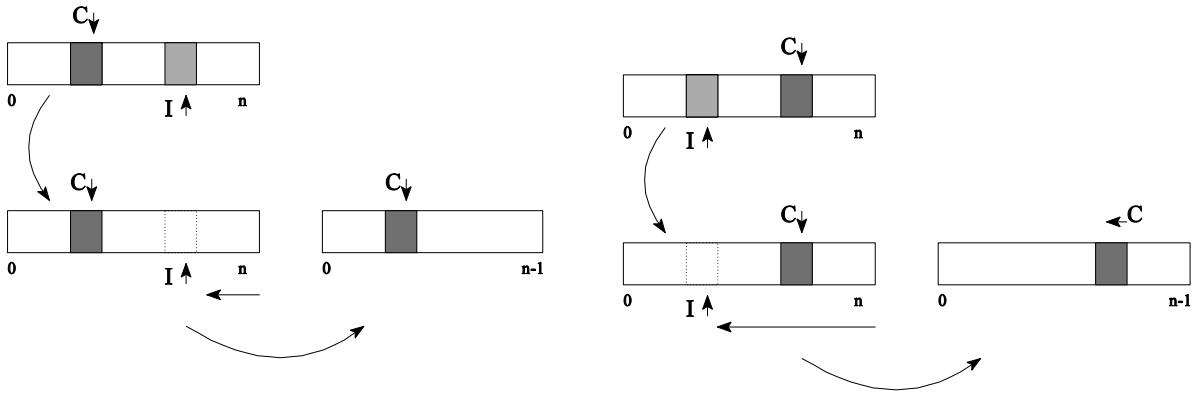
Veamos más gráficamente el funcionamiento de estas tres operaciones. La barra especifica una lista; I indica el elemento insertado con la operación Insertar involucrada, y C ó E, a las operaciones Cambiar, Extraer, o Eliminar.

Comencemos con Cambiar:



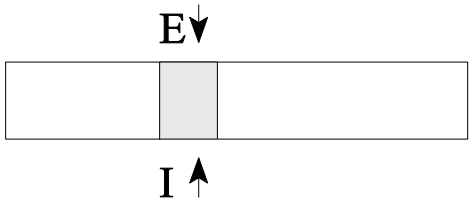
Dado que la operación Cambiar e Insertar tienen lugar sobre la misma posición, se efectúa el cambio y ya está.
Si se ha insertado en una posición previa a la que se quiere cambiar, será necesario usar recursividad: Sacar la operación Insertar, y seguir trabajando con la lista en que se insertaba.

Por tanto, la posición en que se efectúa el Cambiar debe retrasarse en 1.
En caso de que la posición a cambiar esté antes de la que se inserta, podemos efectuar la recursión sin necesidad de variar las posiciones.

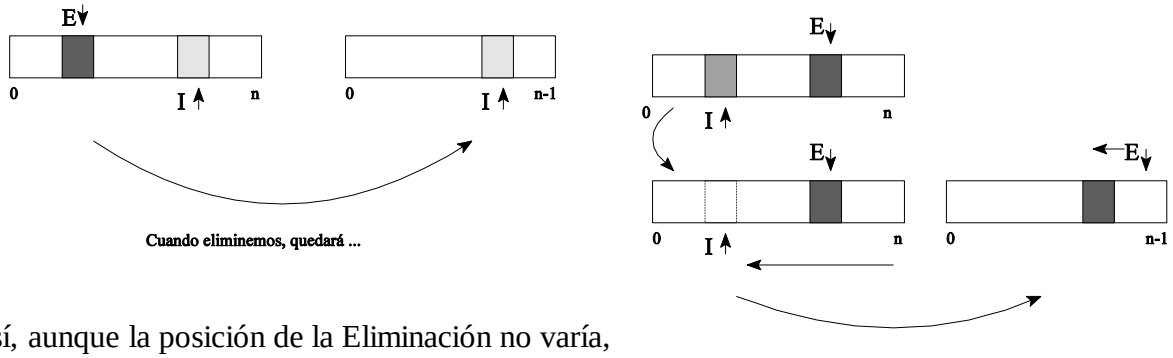


Veamos el caso de las Eliminaciones (la extracción de elementos con Elemento() viene a ser igual que Cambiar()):

En este caso, quitamos la operación de Inserción, y ya está. Nótese que, por las operaciones de los casos que veremos a continuación, el resto de Insertar() no pueden hacer referencia a posiciones de memoria mayores de n-1, puesto que se han ido ajustando de forma recursiva. Los Insertar() más internos tampoco hacen referencia a posiciones mayores de n-1, puesto que por el hecho de ser más internos, se hicieron antes del Insertar() que estamos eliminado ahora, y, por tanto, cuando sólo había n-1 elementos.



El Insertar() seguirá en la misma posición, pero al introducir el Eliminar() de forma recursiva, habrá que restarle 1 a la posición a que hace referencia.



Así, aunque la posición de la Eliminación no varía, al reinsertar el Insertar extraído por la aplicación de la recursividad, hay que hacerlo en una posición menor.

2.1.6 CADENA DE CARACTERES.

Las cadenas de caracteres de un cierto alfabeto, $\mathcal{A}$, tradicionalmente se consideran secuencias de cero o más caracteres en las que se contemplan nociones como: posición de un carácter en una cadena, número de repeticiones de un carácter y subcadena de una cadena.

Formalmente estas estructuras se han considerado estructuras lineales con un valor distinguido, la cadena vacía ε, una serie de valores constituidos por un único carácter de $\mathcal{A}$, y una operación de concatenación para generar las restantes cadenas y que facilita el cálculo con subcadenas; por lo que se ajustarían al patrón

$$\text{Str} = \{\varepsilon\} \mid \mathcal{A} \mid (\text{Str} \times \text{Str})$$

con las operaciones de construcción

$$\varepsilon: \longrightarrow \text{Str};$$
$$\text{'\_'}: \mathcal{A} \longrightarrow \text{Str};$$
$$\text{\_\&\_}: \text{Str} \times \text{Str} \longrightarrow \text{Str};$$

NOTA: El subrayado (), representa la situación en que debe aparecer el parámetro cuando no se usa notación polaca (funcional).

caracterizadas por las ecuaciones (s, s_1, s_2, s_3 : Str)

$$\varepsilon \,\&\, s == s$$
$$s \,\&\, \varepsilon == s$$
$$(s_1 \,\&\, s_2) \,\&\, s_3 == s_1 \,\&\, (s_2 \,\&\, s_3)$$

con una serie de operaciones adicionales para determinar la longitud de una cadena, la posición de un carácter, extraer caracteres y subcadenas, comparar cadenas, etc. La última ecuación de todas indica que no importa el orden en el que se apliquen las operaciones, sino sólo el orden el que intervienen los operandos.

Sin embargo, las cadenas también se pueden considerar tipos lineales ajustados al patrón

$$\text{Str} = \{\varepsilon\} \mid \mathcal{A} \times \text{Str}$$

con dos operaciones de construcción

$$\varepsilon : \longrightarrow \text{Str}$$
$$\text{Cons}: \mathcal{A} \times \text{Str} \longrightarrow \text{Str};$$

y el predicado

$$\text{EsVacia}: \text{Str} \longrightarrow \text{B}$$

lo cual no es más que la definición básica de lista funcional; respecto a esta especificación, las operaciones anteriores se pueden expresar en la forma

a: $\mathcal{A}$; s, s_1, s_2, s_3: Str
'a' == Cons(a, ε)
'a' & s == Cons(a, s);

'a' & (s_1 & s_2) == Cons(a, (s_1 & s_2));

y las restantes ecuaciones que suministran la potencia necesaria a estas estructuras son

Longitud : Str $\longrightarrow \mathbb{N}$
Carácter : $\mathbb{N}$ × Str $\not\longrightarrow \mathscr{C}$
Posición: $\mathscr{C}$ × Str $\longrightarrow \mathbb{N}$
Subcadena : $\mathbb{N}$ × $\mathbb{N}$ × Str $\longrightarrow$ Str
_=_, _<=_, _EsSubcadena_ : Str × Str $\longrightarrow$ B

La semántica operacional correspondiente a estas operaciones sería la siguiente:

Longitud(s)
PRE $s = <a_1, a_2, .. a_n>$
POST Longitud(s)=n
Carácter(i, s)
PRE $s = <a_1, a_2, .. a_n>$, $1 \le i \le n$
POST Carácter(i, s)=a_i
Posición(a, s)
PRE $s=<a_1, a_2, .. a_n>$
POST Carácter(a,s)= $\begin{cases} i & si\ \exists i,\ a_i=a \\ 0 & en\ otro\ caso \end{cases}$
Subcadena(i, j, s)
PRE $s = <a_1, a_2, .. a_n>$, $1 \le i, j, i+j-1 \le n$
POST Subcadena(i, j, s) = $<a_i, a_{i+1}, .. a_{i+j-1}>$

2.1.7 FICHERO SECUENCIAL

Los ficheros secuenciales, asociados normalmente al almacenamiento de datos en memoria secundaria, también son estructuras constituidas por registros en una disposición ordenada en secuencia, que disponen de una operación de recorrido como los anillos, mediante una *ventana* que permite ver el contenido de un registro y que puede situarse en un extremo del fichero, y desplazarse siempre en la misma dirección, registro a registro, a lo largo del fichero; presentan dos posiciones límite: el principio y el final del fichero. La existencia de esta ventana limita la operación de recorrido, impidiendo que los ficheros se puedan tratar como listas genéricas.

Existen varios comportamientos para estos ficheros según las operaciones que se puedan realizar con ellos; basta con revisar los distintos lenguajes de programación que permiten la manipulación de ficheros secuenciales, para tener idea de distintas formas de tratamiento.

Las formas canónicas vendrán dadas por la constante Crear (fichero vacío), la operación Escribir(), (que escribe al comienzo del fichero), y la operación Reset(), que especifica la posición de la ventana.

Desgraciadamente es muy difícil hacer que las formas canónicas obtenidas por reducción,

consten a lo sumo de una operación reset(). Por tanto, si en un término canónico se tienen varios reset, la ventana se considerará posicionada bajo el último que se halla hecho.

tipo Fichero
dominios Fichero, Elemento, Lógico
generadores
Crear: $\longrightarrow$ Fichero
Escribir: Elemento × Fichero $\longrightarrow$ Fichero
Reset: Fichero $\longrightarrow$ Fichero
constructores
Insertar: Elemento × Fichero $\longrightarrow$ Fichero
Avanzar: Fichero $\longrightarrow$ Fichero
Extraer: Fichero $\longrightarrow$ Fichero
selectores
Actual: Fichero $\not\longrightarrow$ Elemento
Es_Vacío: Fichero $\longrightarrow$ Lógico
Fin_de_fichero: Fichero $\longrightarrow$ Lógico
precondiciones
Actual(f) : not(Fin_de_fichero(f))
ecuaciones
Reset(Crear) == Crear
Reset(Reset(f)) == Reset(f)
Insertar(e, Crear) == Reset(Escribir(e, Crear))
Insertar(e, Escribir(e_2, f)) == Escribir(e_2, Insertar(e, f))
Insertar(e, Reset(f)) == Reset(Escribir(e, f))
Avanzar(Crear) == Crear
Avanzar(Escribir(e, f)) == SI Fin_de_fichero(f) ENTONCES
Escribir(e, f)
SI NO
Escribir(e, Avanzar(f))
Avanzar(Reset(Escribir(e, f))) == Escribir(e, Reset(f))
Extraer(Crear) == Crear
Extraer(Escribir(e, f)) == Escribir(e, Extraer(f))
Extraer(Reset(Escribir(e, f))) == Reset(f)
Actual(Escribir(e, f)) == Actual(f)
Actual(Reset(Escribir(e, f))) == e
Es_Vacío(Crear) == V
Es_Vacío(Escribir(e, f)) == F
Es_Vacío(Reset(f)) == Es_Vacío(f)
Fin_de_fichero(Crear) == V
Fin_de_fichero(Escribir(e, f)) == Fin_de_fichero(f)
Fin_de_fichero(Reset(f)) == Es_Vacío(f)

Hay casos especiales de ficheros en los que no sólo es posible efectuar el avance del punto

de interés (la ventana), sino también su retroceso. Por ejemplo, los lenguajes que permiten la gestión de ficheros clásicos de forma extendida, como puede ser Clipper, poseen instrucciones para avanzar y retroceder un número de registros determinado: *SKIP n*. Si **n** toma un valor negativo, se retrocede la posición actual. De hecho, el comportamiento de una estructura como esta se asemeja mucho a un anillo no circular, o sea a una estructura lineal en la que podemos avanzar y retroceder la ventana, pero con las limitaciones de tener extremos que impiden ir más allá del comienzo o del final de la lista.

Es más, al igual que existe un predicado que nos indica cuando estamos al final del fichero (EOF), será necesario un predicado que indique cuando estamos situados al comienzo del fichero (BOF: Begin Of File). La comprobación de este predicado nos impedirá intentar retroceder más acá del comienzo del fichero.

Podemos hacer una operación Retroceder, y otra Inicio_de_fichero, que tendrán las siguientes especificaciones:

operación

Retroceder: Fichero: ⟶ Fichero

Inicio_de_fichero: Fichero ⟶ Lógico

ecuaciones

Retroceder(Crear) == Crear

Retroceder(Reset(f)) == Reset(f)

Retroceder(Escribir(e, Reset(f))) == Reset(Escribir(e, f))

Retroceder(Escribir(e_1, Escribir(e_2, f)) == Escribir(e_1, Retroceder(Escribir(e_2, f)))

Retroceder(Escribir(e, Crear)) == Escribir(e, Crear)

Inicio_de_fichero(Reset(f)) == V

Inicio_de_fichero(Escribir(e, f)) == F

Inicio_de_fichero(Crear) == V

2.2 ESTRUCTURAS DINÁMICAS (No acotadas)

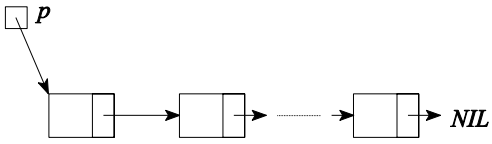
Vamos a establecer a continuación las diferentes formas de almacenar una *lista*. Recordemos que una lista es la forma básica de cualquier TAD Lineal, y que, por tanto, cualquier otro TAD puede hacerse en base a éste.

2.2.1 Sin cabecera

2.2.1.1 Estructura lineal simplemente encadenada.

Aquí, la variable p de tipo Lista, no es más que un puntero a un registro. Cada registro posee dos campos:

- Contenido, donde se almacena un valor del tipo base.
- Siguiente, que no es más que otro puntero de tipo Lista.

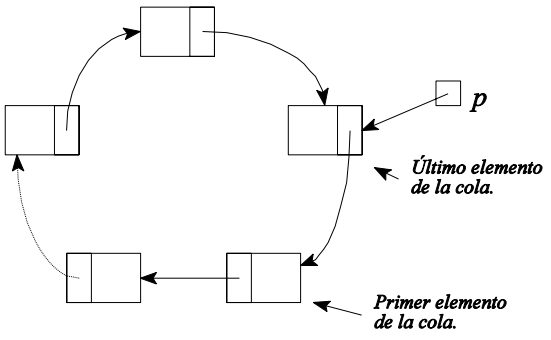


Así, vemos que estamos creando una estructura recursiva, donde una variable distinguida (p, en el ejemplo), que corresponde al tipo que estamos definiendo, apunta a la cabeza de la lista. A su vez, esta cabeza apunta a la cabeza de otra lista, y así sucesivamente, de manera que el último nodo apunta a la lista vacía, denotada por NIL.

```
TYPE
    LISTA=POINTER TO NODO;
    NODO=RECORD
            CONTENIDO:ELEMENTO;
            SIGUIENTE:LISTA;
        END;
```

Este tipo de almacenamiento se presta muy bien para el TAD Pila, ya que en cada momento, a través de la variable p sólo se tiene acceso a la cabeza de la Lista. Como en una Pila, las inserciones y eliminaciones se hacen en la cabeza, pues este método resulta el más idóneo.

2.2.1.2 Estructura circular simplemente encadenada.



Esta estructura es especialmente útil para anillos y colas simples.

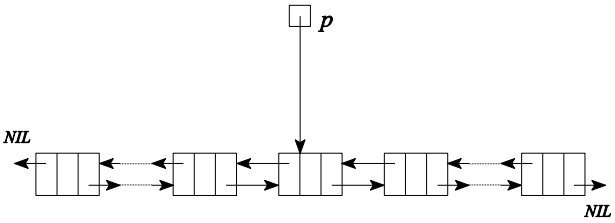
La rotación a la derecha (en el sentido de las agujas del reloj), es directa, pues basta seguir el sentido de los punteros. Las operaciones Rotar_Izqd(), A_la_cola(), y Elim_cola() no son directas, debido a la unidireccionalidad del encadenamiento simple.

Para poder hacer fácilmente la operación de Insertar() (recordemos que Insertar(), introduce un nuevo elemento que pasa a ser la cabeza del anillo), y dado que el último elemento del anillo debe apuntar al primero, es necesario que el puntero **p** no apunte a la cabeza, pues en tal caso, al insertar, sería complicado el conseguir que el último elemento del anillo apuntese al nuevo que se inserta. Para hacer esta operación más fácil, hacemos que *p* no apunte a la cabeza, sino más bien al último elemento, de forma que p^.contenido será el último elemento, y p^.siguiente^.contenido, será la cabeza del anillo. Lo mismo ocurre con las operaciones Insertar() y Extraer() de la cola simple.

Por otro lado, nótese que la estrcutura de datos definida en Modula-2 es la misma:

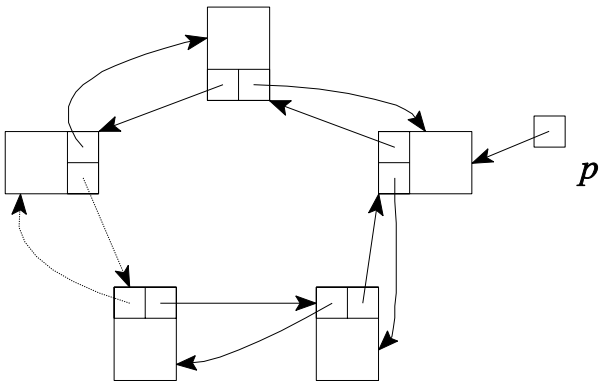
```
TYPE
        LISTA=POINTER TO NODO;
        NODO=RECORD
                        CONTENIDO:ELEMENTO;
                        SIGUIENTE:LISTA;
                END;
```

2.2.1.3 Estructura lineal doblemente encadenada.



Este tipo de estructura permite solucionar los mismo problemas que la *Simplemente encadenada con acceso a punto de interés*, con la desventaja de que es necesaria más memoria para guardar un puntero adicional. Los tipos necesarios coinciden también con los de la estructura siguiente: *Estructura circular doblemente encadenada*.

2.2.1.4 Estructura circular doblemente encadenada.



Este método, permite la implementación eficiente de todas las operaciones del anillo normal, y de la cola doble. **p** puede apuntar ahora sin problemas a la cabeza del anillo, ya que se puede acceder sin problemas y de forma directa al último elemento. Las rotaciones tanto a derecha como a izqd. no suponen tampoco complicación alguna, ya que tenemos punteros en ambas direcciones.

La estructura en Modula-2 sería:

```
TYPE
        LISTA=POINTER TO NODO;
        NODO=RECORD
                        CONTENIDO:ELEMENTO;
                        SIGUIENTE,
                        ANTERIOR:LISTA;
                END;
```

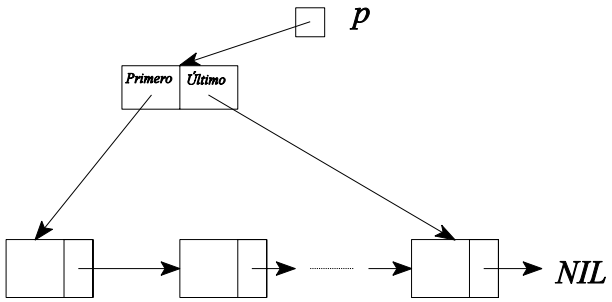
2.2.2 Con cabecera.

En los casos anteriores, definíamos el tipo Lista como un puntero a un nodo que contenía la información del tipo base que se deseaba almacenar. Hay casos en los que se desea obtener información de forma más rápida y eficiente sobre la Lista. P.ej. el cálculo de la longitud de una lista simplemente encadenada obliga a recorrer todos los elementos de la lista con el consiguiente gasto de tiempo.

Para evitar esto, vamos a crear un registro previo a la estructura Lista en sí, que contendrá información adicional. A este regitro le llamaremos Cabecera de la Lista. Las ventajas de esta cabecera son:

- Permite tener información global sobre la estructura.
- Obliga a un acceso forzoso a través del manejador de la cabecera.
- Permite agrupar información de acceso a varios puntos de la estructura.

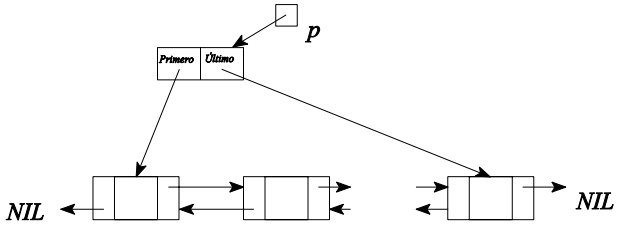
2.2.2.1 Estructura lineal simplemente encadenada con acceso a extremos.



Esta estructura tiene las misma propiedades que la cerrada simplemente encadenada. Por tanto, es útil para las colas y anillos simples. La definición en Modula-2 sería:

```
TYPE
        LISTA=POINTER TO CABECERA;
        CABECERA=RECORD
                                PRIMERO,
                                ULTIMO: POINTER TO NODO;
                        END;
        NODO=RECORD
                        CONTENIDO:ELEMENTO;
                        SIGUIENTE: POINTER TO NODO;
                END;
```

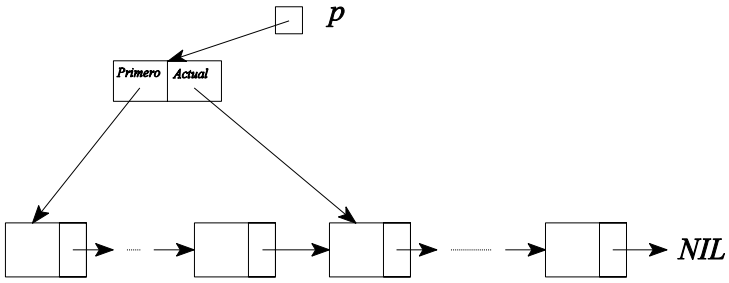
2.2.2.2 Estructura lineal doblemente encadenada con acceso a extremos.



Esta otra estructura es prácticamente igual que la lineal doblemente encadenada, excepto por el hecho de que lleva asociada una cabecera que puede contener información adicional.

```
TYPE
        LISTA=POINTER TO CABECERA;
        CABECERA=RECORD
                        PRIMERO,
                        ULTIMO: POINTER TO NODO;
                END;
        NODO=RECORD
                CONTENIDO:ELEMENTO;
                SIGUIENTE,
                ANTERIOR: POINTER TO NODO;
            END;
```

2.2.2.3 Estructura lineal simplemente encadenada con acceso a un extremo y a punto de interés.



La estructura en Modula-2 sería idéntica a la anterior:

```
TYPE
        LISTA=POINTER TO CABECERA;
        CABECERA=RECORD
                        PRIMERO,
                        ACTUAL: POINTER TO NODO;
                END;
        NODO=RECORD
                CONTENIDO:ELEMENTO;
```

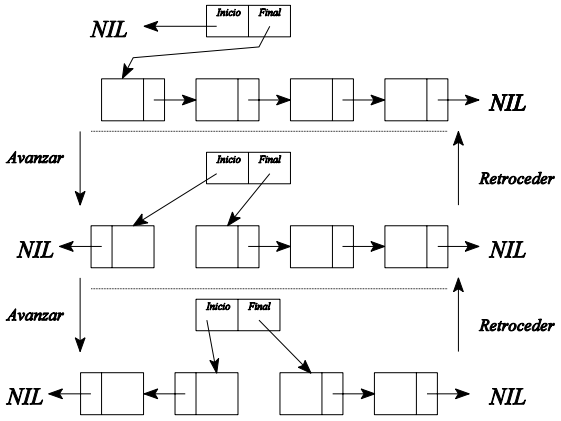
```
                SIGUIENTE: POINTER TO NODO;
        END;
```

Los punteros a un elemento intermedio tienen dos utilidades principales:
- Apuntar al último elemento accedido, empleando un criterio parecido al de las memoria caché, según el cual el último objeto accedido es el que tiene mayor probabilidad de volver a ser accedido en una operación posterior. Esto se emplea especialmente para implementar ficheros de acceso secuencial.
- Permite (si ACTUAL apunta al elemento situado en la posición Longitud()/2), dividir en dos el tiempo de acceso a cualquier elemento. Esto es especialmente útil en las listas; en tal caso, en la cabecera también habría que almacenar otro campo que nos indicase la posición del elemento al que apunta ACTUAL.

Por lo demás, se comporta como la lineal simplemente encadenada.

2.2.2.4 Estructura lineal simplemente encadenada con acceso a punto de interés.

Cuando sólo se desea acceder al elemento sobre el cual se halla el punto de interés (la ventana), y se quiere que la ventana se pueda desplazar a izquierda y derecha indistintamente, parece que la única solución posible es hacer un encadenamiento doble, si se quiere que los algoritmos de desplazamiento sean de orden O(1), o bien dejar la estructura de encadenamiento simple a costa de que una de las operaciones de movimiento sea de orden O(n). No obstante, hay otra solución, que se basa en hacer que los apuntadores puedan apuntar elementos tanto situados delante como detrás del nodo en el que se encuentran. Se adaptan muy bien a los ficheros secuenciales extendidos. Cualquier operación de colocar el punto de interés en algún extremo de la lista, será de O(n), ya que no sólo implica posicionar el puntero Final en dicho extremo, sino cambiar todos los punteros a su derecha o a su izquierda para que apunten en la misma dirección. Es importante observar cómo en este caso, la propia estructura secuencial almacena la información del punto de interés; en otras palabras, en las estructuras anteriores, es posible definir puntos de interés adicionales independientemente de la estructura lineal, cosa que no ocurre aquí, ya que los propios punteros almacenados en la estructura dependen de la posición del punto de interés, lo que imposibilita crear estructuras adicionales que sólo posean información sobre otros puntos de interés.

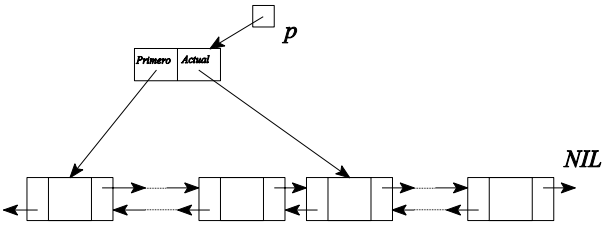


Nótese que con esta estructura, el puntero Final nunca puede apuntar a NIL, ya que final apunta a la cabeza del anillo.

De esta forma, podemos efectuar avances y retrocesos sin necesidad de complejas ni costosas búsquedas y recorridos de punteros. La estructura en Modula-2 es igual que en el caso anterior, sólo que el concepto de Primero y Actual se cambia por los de Inicio y final

respectivamente.

2.2.2.5 Estructura lineal doblemente encadenada con acceso a un extremo y a punto de interés.



Análoga a la doblemente encadenada: colas dobles y anillos con todas sus operaciones. La estructura en Modula-2 sería:

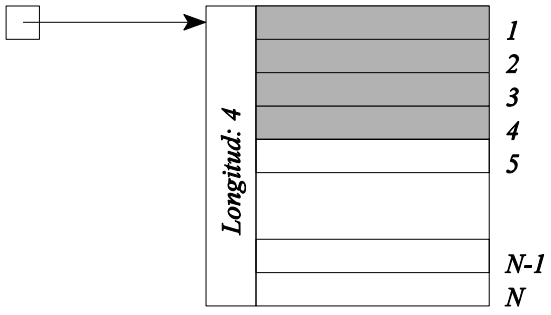
```
TYPE
        LISTA=POINTER TO CABECERA;
        CABECERA=RECORD
                        PRIMERO,
                        ACTUAL: POINTER TO NODO;
                END;
        NODO=RECORD
                CONTENIDO:ELEMENTO;
                SIGUIENTE,
                ANTERIOR: POINTER TO NODO;
            END;
```

2.3 ESTRUCTURAS ESTÁTICAS (ACOTADAS).

En estos casos, vamos a hacer uso de una tabla de longitud previamente definida, en la que vamos a ir almacenando los elementos. Asimismo, junto con esta tabla, guardaremos información adicional sobre el número de elementos que hay, la posición del primero, etc.. Podemos decir por tanto, que la cabecera se halla implícita.

2.3.1 De zona compacta.

2.3.1.1 De posición inicial fija.



En estos caso, los elementos se almacenarán siempre uno detrás de otro, desde el comienzo de la tabla; no habrá huecos entre elementos consecutivos. Cuando elimine un elemento del interior de la tabla, será necesario reorganizar los que le suceden para hacer de nuevo una estructura compacta.

Al añadir en las posiciones intermedias

de la tabla, será necesario desplazar los elementos que le sucedan.

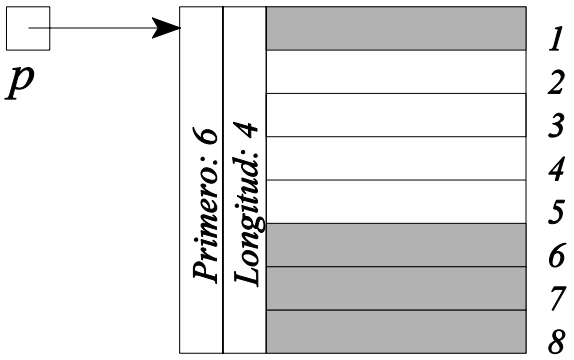
En cualquier caso, el primer elemento de la estructura siempre estará en la posición 1 de la tabla, siendo necesaria la existencia de una variable que indique o bien la cantidad de elementos almacenados, o bien la posición en que se encuentra el último.

Por motivos de eficiencia, sólo tiene sentido en la implementación de pilas; por motivos de ahorro de espacio, puede ser interesante usarlas para almacenar cadenas de caracteres.

En Modula-2 quedaría:

```
TYPE
        P=POINTER TO ESTRUCTURA;
        ESTRUCTURA=RECORD
                        ULTIMO : [0..N];
                        POSICION : ARRAY [1..N] OF ELEMENTO;
                END;
```

2.3.1.2 De posición inicial flotante.



Este método es especialmente útil para implementar colas:

- La operación insertar en la cola, metería el elemento en la posición Primero+Longitud, siempre que la tabla no esté llena, e incrementaría en 1 la longitud. La tabla se supone con estructura circular, o sea, la primera posición es la sucesora de la última.
- La operación Extraer, simplemente incrementaría el valor de Primero, y decrementaría la Longitud.

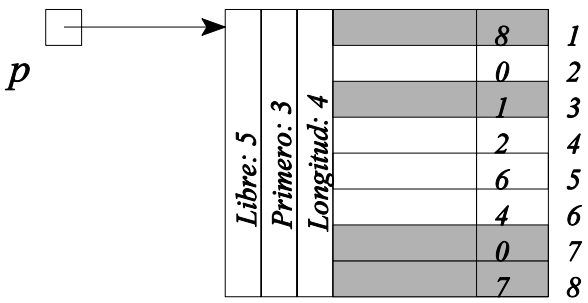
Su uso con las colas dobles también resulta evidente.

También puede servir para su uso con anillos: las operaciones de rotación, obligarían a eliminar un elemento extremo, y a insertarlo por el otro extremo.

En Modula-2 quedaría:

```
TYPE
        P=POINTER TO ESTRUCTURA;
        ESTRUCTURA=RECORD
                        PRIMERO : [1..N];
                        LONGITUD : [0..N];
                        POSICION : ARRAY [1..N] OF ELEMENTO;
                END;
```

2.3.2 Con cursores.



En esta implementación se intenta simular el concepto de puntero, pero a través de números naturales que representan posiciones en una tabla. Cada elemento de la tabla posee un campo (llamado cursor) que indica cual es la posición del elemento siguiente. Por supuesto, aparte de la tabla será necesario registrar dónde se encuentra el primer elemento de la lista (campo *primero*). Asimismo, el conjunto de elementos no usados en la tabla, también conformará otra lista, que llamaremos *lista libre*. El último elemento (aquél que no tiene sucesor), contendrá en su cursor una posición inexistente, que indicará que detrás de él ya no hay nadie.

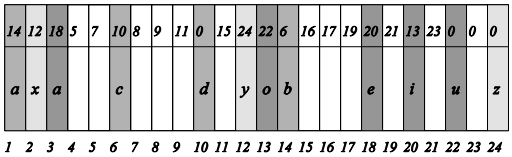
La figura adjunta representa cómo quedaría la estructura tras haber realizado numerosas inserciones y eliminaciones.

Dado que se simula el concepto de puntero, podemos establecer encadenamientos de la misma forma que hacíamos con las estructuras dinámicas: estructuras circulares, doblemente encadenadas, con punto de interés, etc..

Las inserciones y eliminaciones se hacen de igual forma que en el caso de listas dinámicas simplemente encadenadas.

La estructura de esto es:

```
TYPE
        P=POINTER TO ESTRUCTURA;
        ESTRUCTURA=RECORD
                        LIBRE : [0..N];
                        PRIMERO : [0..N];
                        POSICION : ARRAY [1..N] OF NODO;
                END;
        NODO=RECORD
                        CONTENIDO
: ELEMENTO;
                        SIGUIENTE :
[0..N];
                END;
```



En la *estructura de cabecera* se puede incluir toda la información adicional que se considere necesaria.

l1 = ['a', 'b', 'c', 'd'] *l1 := 1*
l2 = ['x', 'y', 'z'] *l2 := 2*
l3 = ['a', 'e', 'i', 'o', 'u'] *l3 := 3*
libre := 4

Poseen el inconveniente de ser

acotadas, y lo que es más, es posible que determinadas estructuras estén casi vacías, y otras tan llenas que tras varias inserciones den lugar a falta de memoria por ocupación completa. Este problema se debe a que la memoria asociada a una estructura es local, y no puede ser compartida por otra estructura. Por tanto, podemos crear una matriz global, que sea compartida por todas las estrucuras, con lo que se aprovecha mejor el espacio, a costa de reservar inicialmente una enorme cantidad de memoria que puede no ser aprovechada en su totalidad.

2.4 ESTUDIO COMPARATIVO.

Las diferencias existentes entre los dos métodos básicos de almacenamiento de estructuras que hemos visto (dinámico y estático), son muy numerosas, y podemos clasificarlas en dos aspectos fundamentales:

Uso de la memoria. Con la implementación estática, los requerimientos de memoria se hacen en tiempo de compilación; sin embargo esto implica una fuerte restricción con respecto a la extensibilidad dinámica de las estructuras, o sea, a su agrandamiento o empequeñecimiento, mientras dure su vida. P.ej., es posible que no se puedan introducir más elementos en una pila, por falta de previsión en el cálculo del tamaño máximo, a la vez que existen otras pilas en las que sobra mucho espacio porque tienen pocos elementos apilados. En definitiva, la implementación estática desperdicia espacio, impidiendo su total aprovechamiento. Además, un cambio de decisión en la longitud máxima de las estructuras, implica una recompilación del TAD.

Las estructuras dinámicas sin embargo, hacen un uso compartido de la memoria dinámica, de manera que si hay alguna falta de memoria, es porque en efecto, toda ha sido consumida. No obstante, deja en manos del usuario la tarea de recuperar la memoria cogida, una vez que se destruyan las estructuras, ya que de lo contrario queda basura almacenada que también consume memoria.

Por otro lado, la implementación estática da mayor control al programador sobre la memoria de almacenamiento. P.ej., olvidar liberar un puntero en la implementación dinámica, hace que ese trozo de memoria sea irrecuperable. En el caso dinámico (con cursores y matriz general), podemos suministrar una operación de limpieza general, que elimine todas las estructuras de la matriz. Llamando a esta operación en determinados puntos del programa usuario, se puede hacer que el sistema sea más tolerante a fallos, al asegurarse de que no habrá problemas de falta de memoria debidos a fallos en los programadores.

Si la implementación estática no utiliza cursores, entonces se aprovecha mejor la memoria, ya que no hay punteros que ocupen espacio; aunque con el enorme inconveniente de la ineficiencia en la inserción y eliminación de elementos en las posiciones intermedias de la matriz.

Eficiencia. El uso de punteros, implica un recargo de tiempo derivado de la necesidad de recoger memoria mediante la operación correspondiente (NEW, ALLOCATE, malloc, etc.). Para ello, es necesario ejecutar un gestor de memoria que se encarga de habilitar y recuperar trozos de memoria. Sin embargo, dado que se trabaja directamente con direcciones de memoria directamente entendibles por la máquina, este método es muy rápido. De hecho, es posible incluso que la traducción necesaria en tablas estáticas, de posición a dirección de memoria, requiera más

tiempo que la creación o eliminación de memoria.

Además, en la implementación estática, el tamaño de la estrucutra ha sido calculado por el programador en tiempo de compilación, con lo que la ejecución no se ve afectada. Además, aunque el tipo opaco sea obligatoriamente un puntero, sólo será necesario hacer NEW y DISPOSE una vez, mientras que en la impl. dinámica será una de las operaciones más comunes si la volatilidad es alta.

Por otro lado, con la implementación estática, cada vez que se pasa una lista como parámetro, es necesario copiar la estructura completa. Para evitar este recargo de tiempo, basta con hacer que el tipo lista sea opaco, o sea, un puntero a la estructura estática.

2.5 IMPLEMENTACIONES.

Es necesario reseñar que en las implementaciones que vamos a efectuar, no seguiremos las especificaciones funcionales al pie de la letra, de manera que al aplicar un constructor o un generador a un objeto del TAD, no se devolverá otro objeto TAD, sino que más bien se verá modificado el parámetro de entrada, que pasará a tomar el valor de la salida. P. ej., cuando incluímos un elemento **e** en una pila P_1, no devolvemos una pila P_2 igual a P_1 con el elemento **e** añadido, sino que modificamos la pila P_1 incluyéndole el elemento **e**.

Esto permite un mayor control de las operaciones por parte del usuario, y una gestión más eficiente de la memoria.

Por otro lado, el uso de punteros es peligroso, especialmente cuando el tipo base también es un puntero que apunta a una estructura. P.ej., que ocurriría en una operación Insertar(). En una operación tal pasamos como argumento un elemento del tipo base E, que en realidad es un puntero. Cuando en la estructura L en la que queremos insertarlo, habilitamos memoria para ello, estamos cogiendo un trozo de memoria del tamaño de un puntero, ¡no de la estructura a la que éste apunta!. Por tanto, si hacemos una mera asignación, lo que estamos haciendo es guardar No una copia del objeto E tal cual, sino guardar una copia del puntero a ese objeto, con lo cual tendremos un solo objeto E, apuntado por dos punteros:

- Un puntero, el que se pasó como parámetro.
- Otro, el que se ha almacenado en la estructura L.

Así, si decidimos modificar aquello a lo que apunta uno de estos dos punteros, también lo estamos variando desde el punto de vista del otro: se produce un efecto lateral; es lo que se denomina compartición estructural.

Para evitar este problema (hay casos en los que la compartición estructural puede ser un efecto deseado), se debería asociar a cada TAD la operación Copiar(), que devuelve un objeto igual a su parámetro, pero almacenado de forma independiente, o sea, sin compartición estructural. La operación Copiar() debería sustituir siempre a la asignación (:=).

Esta nueva operación sólo tiene sentido en las implementaciones, y no constará para nada en las especificaciones algebraicas, ya que carece de sentido (en todos los casos se definiría por la ecuación Copiar(l) == l).

2.5.1 Tratamiento de errores.

Se estudiarán tres maneras de controlar errores.

1.- La manera mas sencilla pero menos útil de controlar errores, consiste en emitir un mensaje por pantalla indicando el tipo de error detectado y salir del procedimiento mediante la función HALT.

Ejemplo.

```
PROCEDURE Asign (VAR r :RACIONAL;n,d :CARDINAL);
    ...
    ...
    ...
    IF d = O THEN
       WrStr('Error el denominador no puede ser 0'); HALT
     ...
     ...
END Asign;
```

2.- Tratamiento de errores a través de una variable que es introducida como parámetro de cada procedimiento, la cual toma un valor que puede ser analizado por el usuario y tomar las acciones que crea convenientes en cada circunstancia. Es un poco engorroso de manejar puesto que en cada llamada ha de mirar el valor de esta variable.

Ejemplo.

```
TYPE
        ERROR = (SinError, ObjetoNoCreado, DivisiónPorCero, NoHayMemoria);

PROCEDURE Asign(VAR r:RACIONAL; n,d :CARDINAL; VAR error:ERROR);
  ...
  ...
  Error := SinError;
  IF d = 0 THEN error := DivisiónPorCero
  ...
  ...
END Asign;
```

3.- El tercer control de errores es mediante una variable global definida en el módulo de definición, definiendo también en este punto los distintos errores a través de un tipo enumerado que definen los errores que pueden darse. Ejemplo en este TAD del número racional sería:

```
DEFINITION MODULE Racional;

TYPE
        ERROR = (SinError, ObjetoNoCreado, DivisiónPorCero, NoHayMemoria);
VAR
        error : ERROR;
```

En la implementación y al comienzo de cada procedimiento la variable se inicializa al valor SinError y se actualiza dependiendo de cada situación. Finalmente el tratamiento de errores se realizará en el driver a través de un procedimiento que testeando el valor de la variable **error** mediante un CASE, tome las medidas oportunas.

Podemos añadir un 4º método, que consiste en:

4.- El tipo que se está definiendo, se convierte, si no lo es ya, en registro, y se le añade un campo, llamado *error*, que será de tipo Lógico, y que contendrá Verdad, en caso de que ese elemento sea erróneo, y Falso en caso contrario. También podemos hacer que *error* sea de tipo enumerado y que contenga el código del error. Debe haber siempre un código que sea el del objeto no erróneo.

Una variante para el caso de los tipos opacos, que son punteros (y para preveer que no haya memoria), es considerar además, que todo objeto que apunta a NIL es erróneo.

Las características de este método son:

- En la especificación algebraica, permite que todas las operaciones sean totales.
- Es necesario especificar el comportamiento de las operaciones en caso de que alguno de sus parámetros sea un objeto erróneo. Hay que ampliar pues, el número de las ecuaciones en la especificación algebraica, como se vio en el tema 1º.
- El error es inherente al objeto, y no a la última operación que se haya hecho respecto a cualquier objeto del TAD.

En cualquier caso, conviene crear un tipo ERROR que sea un enumerado, y cuyos valores sean autodocumentados; p.ej:

```
TYPE
        ERROR = (No_Error, No_Memoria, Estructura_Vacía);
```

De esta forma se aumenta la legibilidad de los programas, evitando contínuas referencias a la página del código donde se definen y describen los valores. La única excepción a esto es cuando se desea diferenciar únicamente la existencia o inexistencia de errores, en cuyo caso el tipo ERROR equivaldrá al tipo BOOLEAN.

2.6 IMPLEMENTACIONES EN MODULA-2.

Primeramente se presenta una estructura acotada, en la que el tipo base son cadenas de caracteres. El método de almacenamiento empleado es de zona compacta con primera posición

fija.

Se hace n tratamiento de errores por mensaje y parada.

```
DEFINITION MODULE ListasEstaticas;

CONST
        N = 100;
        MAX = 100;

TYPE
        LISTA;
        ITEM = ARRAY[0..N] OF CHAR;

PROCEDURE Crear(): LISTA;
PROCEDURE Vacia(l: LISTA): BOOLEAN;
PROCEDURE Llena(l : LISTA): BOOLEAN;
PROCEDURE Longitud(l : LISTA): CARDINAL;
PROCEDURE Insertar(VAR l: LISTA; i: CARDINAL; x: ITEM);
PROCEDURE Cambiar(VAR l: LISTA; x: ITEM; i: CARDINAL);
PROCEDURE Eliminar(VAR l: LISTA; i: CARDINAL);
PROCEDURE Elemento(l: LISTA; i: CARDINAL): ITEM;
PROCEDURE Destruir(VAR l: LISTA);

END ListasEstaticas.
```

-----•O•-----

```
IMPLEMENTATION MODULE ListasEstaticas;
FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;

TYPE
        LISTA = POINTER TO ESTRUCTURA;
        ESTRUCTURA = RECORD
                longitud: CARDINAL;
                entrada: ARRAY [1..MAX] OF ITEM;
        END;

PROCEDURE Crear(): LISTA;
VAR
        l: LISTA;
BEGIN
        IF NOT Available(SIZE(ESTRUCTURA)) THEN
                WrLn;
                WrStr ('ERROR: No hay memoria suficiente.');
                HALT
        END
        ALLOCATE(l,SIZE(ESTRUCTURA));
        l^.longitud:=O;
        RETURN l
END Crear;

PROCEDURE Longitud(l: LISTA): CARDINAL;
BEGIN
                RETURN l^.longitud
```

```
END Longitud;

PROCEDURE Elemento(l: LISTA; i: CARDINAL): ITEM;
BEGIN
        WITH l^ DO
                IF (0<i) AND (i<=longitud) THEN
                        RETURN entrada[i]
                ELSE
                        WrLn;
                        WrStr ('ERROR: Lectura de Elemento Incorrecta');
                        HALT
                END
        END
END Elemento;

PROCEDURE Insertar(VAR l: LISTA; i: CARDINAL; x: ITEM);
VAR
        j:CARDINAL;
BEGIN
        WITH l^ DO
                IF (0<i) AND (i<=longitud+l) AND (longitud<MAX) THEN
                        FOR j := longitud TO i BY -1 DO
                                entrada[j+1] := entrada[j]
                        END;
                        INC(longitud);
                        entrada[i]:=x
                ELSE
                        WrLn;
                        WrStr ('ERROR: Inserción incorrecta.');
                        HALT
                END;
        END;
END Insertar;

PROCEDURE Eliminar(VAR l: LISTA; i: CARDINAL);
VAR
        j : CARDINAL;
BEGIN
        WITH l^ DO
                IF (0<i) AND (i<=longitud) THEN
                        DEC(longitud);
                        FOR j:=i TO longitud DO
                                entrada[j] := entrada[j+1];
                        END;
                ELSE
                        WrLn; WrStr('ERROR: Eliminación incorrecta.'); HALT
                END;
        END;
END Eliminar;

PROCEDURE Cambiar(VAR l: LISTA; x :ITEM; i: CARDINAL);
BEGIN
        WITH l^ DO
                IF (0 < i) AND (i <= longitud) THEN
                        entrada[i]:=x
                ELSIF (i = longitud + 1) AND (longitud < MAX) THEN
```

```
            (* Nótese que permitimos insertar en la última posición     *)
            (* a través de la operación Cambiar.                        *)
                INC(longitud);
                entrada[i] := x;
            ELSE
                WrStr ('ERROR: Cambio Incorrecto'); HALT
            END;
        END;
END Cambiar;

PROCEDURE Vacia(l: LISTA): BOOLEAN;
BEGIN
        RETURN (l^.longitud = 0)
END Vacia;

PROCEDURE Llena(l:LISTA): BOOLEAN;
BEGIN
        RETURN (l^.longitud = MAX)
END Llena;

PROCEDURE Destruir(VAR 1: LISTA);
BEGIN
        DEALLOCATE(l,SIZE(ESTRUCTURA))
END Destruir,

END ListasEstaticas.
```

-----•O•-----
-----•O•-----

Ahora presentamos una implementación dinámica en la que se ha optado por emplear una cabecera con puntero a extremo y a un punto intermedio. Dado que tratamos con listas, en la cabecera también es necesario almacenar, además del puntero al nodo actual, la posición que ocupa ese nodo actual. A pesar de ser dinámica, también es acotada, pues sólo permitimos un máximo de MAXLISTA elementos almacenados.

```
DEFINITION MODULE ListasDinamicas;

TYPE
        LISTA;
        ITEM = ARRAY [0..100] OF CHAR;

VAR
        ErrorDeLista: BOOLEAN;

PROCEDURE Crear(): LISTA;
PROCEDURE Vacia(l: LISTA): BOOLEAN;
PROCEDURE Llena(l: LISTA): BOOLEAN;
PROCEDURE Longitud(l: LISTA): CARDINAL;
PROCEDURE Insertar(VAR l: LISTA; i: CARDINAL; x: ITEM);
PROCEDURE Cambiar(VAR l: LISTA; x: ITEM; i: CARDINAL);
PROCEDURE Eliminar(VAR l: LISTA; i: CARDINAL);
```

```
PROCEDURE Elemento(l: LISTA; i: CARDINAL): ITEM;
PROCEDURE Destruir(VAR l: LISTA);

END ListasDinamicas.
```

-----•O•-----

```
IMPLEMENTATION MODULE ListasDinamicas;

FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;

CONST
        MaxLista = 10000;

TYPE
        LISTA = POINTER TO CabezaLista;
        PunteroLista = POINTER TO NodoLista;
        CabezaLista = RECORD
                longitud: CARDINAL;
                actual: CARDINAL;
                punteroActual: PunteroLista;
                primero: PunteroLista;
        END;
        NodoLista= RECORD
                contenido: ITEM;
                siguiente: PunteroLista
        END;

PROCEDURE Crear(): LISTA;
VAR
        l: LISTA;
BEGIN
        IF Available(SIZE(CabezaLista)) THEN
                ErrorDeLista := FALSE;
                NEW(l);
                WITH l^ DO
                        longitud := 0;
                        actual := 0;
                        punteroActual := NIL;
                        primero := NIL
                END;
                RETURN l
        ELSE
                ErrorDeLista := TRUE;
        END;
END Crear;


(* Procedimiento que devuelve un puntero al elemento i-ésimo de la lista      *)
(* aprovechando el puntero a punto de interés.                                *)
PROCEDURE EncuentraNodo(VAR l: LISTA; i: CARDINAL):PunteroLista;
VAR
        j: CARDINAL;
        aux : PunteroLista;
BEGIN
```

```
        WITH l^ DO
                IF i=actual THEN
                        RETURN punteroActual;
                ELSIF (i<1) OR (i>longitud) THEN
                        RETURN NIL;
                ELSE
                        IF i < actual THEN
                                j := 1;
                                aux := primero;
                        ELSE
                                j := actual + 1;
                                aux := punteroActual^.siguiente;
                        END;
                        WHILE j<>i DO
                                INC(j);
                                aux := aux^.siguiente;
                         END;
                        RETURN aux;
                END
        END
END EncuentraNodo;

PROCEDURE Cambiar(VAR l: LISTA; x: ITEM; i: CARDINAL);
VAR
        aux : PunteroLista;
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN; END;
        WITH l^ DO
                aux := EncuentraNodo(l,i);
                IF aux <> NIL THEN
                        ErrorDeLista := FALSE;
                        aux^.contenido := x;
                        punteroActual := aux;
                        actual := i
                ELSE
                        ErrorDeLista := TRUE
                END
        END
END Cambiar;

PROCEDURE Elemento(l: LISTA; i: CARDINAL): ITEM;
VAR
        aux: PunteroLista;
        basura : ITEM;
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN basura; END;
        WITH l^ DO
                aux := EncuentraNodo(l,i);
                IF aux <> NIL THEN
                        ErrorDeLista := FALSE;
                        punteroActual := aux;
                        actual := i;
                        RETURN punteroi^.contenido;
                ELSE
                        ErrorDeLista := TRUE;
                        RETURN basura;
```

```
            END;
        END;
END Elemento;

PROCEDURE Insertar(VAR l: LISTA; i: CARDINAL; x: ITEM);
VAR
        anterior, nuevo : PunteroLista;
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN; END;
        WITH l^ DO
            IF (i>=1) AND (i<=longitud+1) AND (longitud<MaxLista) AND (Available(SIZE(NodoLista)))
            THEN
                INC(longitud);
                NEW(nuevo);
                nuevo^.contenido := x;
                IF i=1 THEN
                    nuevo^.siguiente := primero;
                    primero := nuevo;
                ELSE
                    anterior := EncuentraNodo(l,i-1);
                    nuevo^.siguiente := anterior^.siguiente;
                    anterior^.siguiente := nuevo;
                END;
                actual := i;
                punteroActual := nuevo;
                ErrorDeLista := FALSE;
            ELSE
                ErrorDeLista := TRUE;
            END
        END
END Insertar;

PROCEDURE Vacia(l: LISTA): BOOLEAN;
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN TRUE; END;
        ErrorDeLista := FALSE;
        RETURN l^.longitud = 0;
END Vacia;

PROCEDURE Llena(l : LISTA): BOOLEAN;
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN TRUE; END;
        ErrorDeLista := FALSE;
        RETURN l^.longitud = MaxLista;
END Llena;

PROCEDURE Longitud(l: LISTA): CARDINAL;
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN 0; END;
        ErrorDeLista := FALSE;
        RETURN l^.longitud;
END Longitud;

PROCEDURE Eliminar(VAR l: LISTA; i: CARDINAL);
VAR
        aux : PunteroLista;
```

```
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN; END;
        ErrorDeLista := FALSE;
        WITH l^ DO
                IF (i<1) OR (i>longitud) THEN
                        ErrorDeLista := TRUE;
                ELSIF i=1 THEN
                        DEC(longitud);
                        aux := primero;
                        primero := primero^.siguiente;
                        actual := 1;
                        punteroActual := primero;
                        DISPOSE(aux);
                ELSE
                        punteroActual := EncuentraNodo(l,i-1);
                        actual := i-1;
                        aux := punteroActual^.siguiente;
                        punteroActual^.siguiente := aux^.siguiente;
                        DEC(longitud);
                        DISPOSE(aux);
                END;
        END
END Eliminar;

PROCEDURE Destruir(VAR l: LISTA);
VAR
        aux : PunteroLista;
BEGIN
        IF l = NIL THEN ErrorDeLista = TRUE; RETURN; END;
        ErrorDeLista := FALSE;
        WHILE l^.primero <> NIL DO
                aux := l^.primero;
                l^.primero := l^.primero^.siguiente;
                DISPOSE(aux)
        END;
        DISPOSE(l);
END Destruir,

BEGIN (* Del módulo de implementación.      *)
        ErrorDeLista := FALSE;
END ListasDinamicas.
```

-----•o•-----

-----•o•-----

Pasamos ahora a ver la implementación de una estructura (Lista) lineal doblemente encadenada con punteros a los extremos, a la que además se ha añadido un puntero a un punto intermedio. El tratamiento de errores se ha hecho mediante variable de estado interna, que sólo se puede consultar a través de la función Error().

```
DEFINITION MODULE ListaDobleEnlace;

        TYPE
                LISTA;
```

```
            ITEM = ARRAY [0..20] OF CHAR;
            ERRORLISTA = (correcto, listaVacia, fueraDeRango, error);

    PROCEDURE Crear(): LISTA;
    PROCEDURE Vacia(l: LISTA): BOOLEAN;
    PROCEDURE Longitud(l: LISTA): CARDINAL;
    PROCEDURE Insertar(VAR l: LISTA; i: CARDINAL; x: ITEM);
    PROCEDURE Cambiar(VAR l: LISTA; x:ITEM; i:CARDINAL);
    PROCEDURE Eliminar(VAR l: LISTA; i: CARDINAL);
    PROCEDURE Elemento(l: LISTA; i: CARDINAL): ITEM;
    PROCEDURE Error(l: LISTA): ERRORLISTA;
    PROCEDURE Destruir(VAR l:LISTA);

END ListaDobleEnlace.
```

-----•o•-----

```
IMPLEMENTATION MODULE ListaDobleEnlace;

FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;
FROM IO IMPORT WrStr,WrLn;

TYPE
    LISTA = POINTER TO CABECERA;
    ENLACE = POINTER TO NODO;
    CABECERA = RECORD
        longitud: CARDINAL;
        actual: CARDINAL;
        puntActual: ENLACE;
        primero, ultimo: ENLACE;
        estado:ERRORLISTA
    END;
    NODO = RECORD
        cont: ITEM;
        sig, ant: ENLACE
    END;

PROCEDURE Error(l: LISTA): ERRORLISTA;
BEGIN
    IF l = NIL THEN
        RETURN error;
    ELSE
        RETURN l^.estado;
    END;
END Error;

PROCEDURE Crear(): LISTA;
VAR
    l: LISTA;
BEGIN
    IF NOT Available(SIZE(CABECERA)) THEN RETURN NIL; END;
    NEW(l);
    WITH l^ DO
        estado := correcto;
        longitud := 0;
```

```
            actual := 0;
            puntActual := NIL;
            primero := NIL;
            ultimo := NIL;
        END;
        RETURN l;
END Crear;

PROCEDURE Longitud(l: LISTA): CARDINAL;
VAR
        basura : CARDINAL;
BEGIN
        IF l = NIL THEN RETURN basura; END;
        l^.estado := correcto;
        RETURN l^.longitud;
END Longitud;

(* Hace que el puntero puntActual apunte al elemento i-ésimo,    *)
(* o a NIL, si el elemento i-ésimo no existe.                    *)
(* Se actualiza asimismo el valor de actual.                     *)
PROCEDURE EncuentraNodo(VAR l : LISTA; i: CARDINAL);
BEGIN
        WITH l^ DO
                IF (i<1) OR (i>longitud) THEN
                        actual := 0;
                        puntActual := NIL
                ELSIF i>actual THEN
                        IF longitud-i>i-actual THEN
                                (* Buscamos desde actual en adelante.       *)
                                WHILE actual # i DO
                                        INC(actual);
                                        puntActual := puntActual^.sig
                                END
                        ELSE
                                (* Buscamos desde Ultimo hacia atrás.       *)
                                actual := longitud;
                                puntActual := ultimo;
                                WHILE actual # i DO
                                        DEC(actual);
                                        puntActual := puntActual^.ant
                                END
                        END
                ELSIF i<actual THEN
                        IF i-1>actual-i THEN
                                (* Buscamos desde actual hacia atrás.       *)
                                WHILE actual # i DO
                                        DEC(actual);
                                        puntActual := puntActual^.ant
                                END
                        ELSE
                                (* Buscamos desde primero hacia adelante.   *)
                                actual := 1;
                                puntActual := primero;
                                WHILE actual # i DO
                                        INC(actual);
```

```
                                puntActual := puntActual^.sig
                        END
                END
        END
    END
END EncuentraNodo;

PROCEDURE Vacia(l: LISTA): BOOLEAN;
VAR
    basura : BOOLEAN;
BEGIN
    IF l = NIL THEN RETURN basura; END;
    l^.estado := correcto;
    RETURN l^.longitud=0;
END Vacia;

PROCEDURE Cambiar(VAR l: LISTA; x: ITEM; i: CARDINAL);
BEGIN
    IF l = NIL THEN RETURN; END;
    WITH l^ DO
        IF (i>0) AND (i<=longitud+1) THEN
            IF (i=longitud+1) THEN
                Insertar(l,i,x)
            ELSE
                EncuentraNodo(l,i);
                puntActual^.cont := x;
                estado := correcto;
            END
        ELSE
            estado := fueraDeRango;
        END
    END
END Cambiar;

PROCEDURE Elemento(l: LISTA; i: CARDINAL): ITEM;
VAR
    basura : ITEM;
BEGIN
    IF l = NIL THEN RETURN basura; END;
    WITH l^ DO
        IF (i>0) AND (i<=longitud+1) THEN
            EncuentraNodo(l,i);
            estado := correcto;
            RETURN puntActual^.cont;
        ELSE
            estado := fueraDeRango;
            RETURN basura;
        END
    END
END Elemento;

PROCEDURE Insertar(VAR l: LISTA; i: CARDINAL; x: ITEM);
VAR
    nuevo : ENLACE;
BEGIN
    IF l = NIL THEN RETURN; END;
```

```
        WITH l^ DO
                IF (0<i) AND (i<=longitud+1) THEN
                        IF NOT Available(SIZE(NODO)) THEN estado := error; RETURN; END;
                        NEW(nuevo);
                        aux^.cont := x;
                        IF longitud=0 THEN
                                INC(longitud);
                                nuevo^.sig := NIL;
                                nuevo^.ant := NIL;
                                primero := nuevo;
                                ultimo := nuevo;
                        ELSIF i=1 THEN
                                INC(longitud);
                                nuevo^.sig := primero;
                                primero^.ant := nuevo;
                                nuevo^.ant := NIL;
                                primero := nuevo;
                        ELSIF i=longitud+1 THEN
                                INC(longitud);
                                nuevo^.sig := NIL;
                                nuevo^.ant := ultimo;
                                ultimo^.sig := nuevo;
                                ultimo := nuevo;
                        ELSE
                                INC(longitud);
                                EncuentraNodo(l,i-1);
                                nuevo^sig := puntActual^.sig;
                                nuevo^ant := puntActual;
                                puntActual^.sig^.ant := nuevo;
                                puntActual^.sig := nuevo;
                        END;
                        actual := i;
                        puntActual := nuevo;
                        estado := correcto;
                ELSE
                        estado := fueraDeRango;
                END
        END
END Insertar;

PROCEDURE Eliminar(VAR l: LISTA; i: CARDINAL);
VAR
        aux: ENLACE;
BEGIN
        IF l = NIL THEN RETURN; END;
        WITH l^ DO
                IF (i<1) OR (i>longitud) THEN
                        estado := fueraDeRango;
                ELSE
                        IF longitud=1 THEN
                                DEC(longitud);
                                DISPOSE(primero);
                                primero := NIL;
                                ultimo := NIL;
                        ELSIF i=1 THEN
                                DEC(longitud);
```

```
                        aux := primero;
                        primero := primero^.sig;
                        primero^.ant := NIL;
                        actual := 1;
                        puntActual := primero;
                ELSIF i=longitud THEN
                        DEC(longitud);
                        aux := ultimo;
                        ultimo := ultimo^.ant;
                        ultimo^.sig := NIL;
                        actual := longitud;
                        puntActual := ultimo;
                ELSE
                        EncuentraNodo(l,i-1);
                        DEC(longitud);
                        aux := puntActual^.sig;
                        puntActual^.sig := aux^.sig;
                        puntActual^.sig^.ant := puntActual;
                END;
                DISPOSE(aux);
                estado := correcto;
        END
    END
END Eliminar;

PROCEDURE Destruir(VAR l: LISTA);
BEGIN
    IF l = NIL THEN RETURN, END;
    WITH l^ DO
        WHILE primero<>NIL DO
            puntActual := primero;
            primero := primero^.sig;
            DISPOSE(puntActual)
        END;
    END;
    DISPOSE(l);
END Destruir;

END ListaDobleEnlace.
```

-----•o•-----

Ahora veremos el programa Driver encargado de comprobar las operaciones de las implementaciones anteriormente definidas: estáticas, dinámicas, y doblemente encadenadas. El control de errores que debe hacer el driver variará de una versión a otra.

```
MODULE DriLista;

FROM IO IMPORT WrLn, WrStr, WrCard, RdStr, RdKey, KeyPressed, RdCard, WrChar;
FROM Lista IMPORT LISTA, ITEM, Insertar, Eliminar, Elemento,  Longitud, Llena, Vacia, Crear, Cambiar;

VAR
    l: LISTA;
    cadena: ITEM;
```

```
        opcion: CHAR;
        pos: CARDINAL;

PROCEDURE Insercion(VAR p: LISTA);
BEGIN
        WrStr('INSERCION');WrLn;
        WrStr('Elemento a insertar (de tipo STRING) ');
        RdStr(cadena);
        LOOP
                WrStr('Posición');
                pos:=RdCard();
                IF (pos>0) AND (pos<=Longitud(p)+1) THEN EXIT END;
        END;
        Insertar(p,pos,cadena)
END Insercion;

PROCEDURE Cambio(l: LISTA);
BEGIN
        WrStr('CAMBIO');WrLn;
        IF NOT Vacia(l) THEN
                WrStr('Elemento a introducir (de tipo STRING) ');
                RdStr(cadena);
                LOOP
                        WrStr('Posición '); pos:=RdCard();
                        IF (pos>0) AND (pos<=Longitud(l)) THEN EXIT END
                END;
                Cambiar(l,cadena,pos)
        END
END Cambio;

PROCEDURE Eliminacion(l : LISTA);
BEGIN
        WrStr('ELIMINACION');WrLn;
        IF NOT Vacia(l) THEN
                LOOP
                        WrStr('Posición del elemento a eliminar ');
                        pos:=RdCard();
                        IF (pos>0) AND (pos<=Longitud(l)) THEN EXIT END
                END;
                Eliminar(l,pos)
        END
END Eliminacion;

PROCEDURE EscribirLista(p: LISTA);
VAR
        i,long:CARDINAL;
BEGIN
        WrStr('La lista en este momento es:');WrLn;
        long := Longitud(p);
        IF long=0 THEN WrStr('Lista Nula');WrLn END;
        FOR i :=1 TO long DO
                WrCard(i,2); WrStr(': '); WrStr(Elemento(p,i)); WrLn;
        END;
END EscribirLista;

BEGIN
```

```
        WrStr ("Driver para Listas:");Wrl;
        WrStr ("      n      Crear Lista Nula");Wrl,n;
        WrStr ("      i      Insertar Elemento");WrLn;
        WrStr ("      c      Cambiar Elemento");WrLn;
        WrStr ("      e      Eliminar Elemento");WrLn;
        WrStr ("      l      Consultar Longitud");WrLn;
        WrStr ("      f      Finalizar");WrLn;
        LOOP
                WrLn; WrStr('opcion ');
                REPEAT
                UNTIL KeyPressed();
                opcion := RdKey();
                WrChar(opcion);WrLn;
                CASE opcion OF
                |       'N','n': l:= Crear();
                |       'I','i':  Insercion(l);
                                  EscribirLista(l);
                |       'C','c': Cambio(l);
                                        EscribirLista(l);
                |       'E','e':        Eliminacion(l);
                                        EscribirLista(l);
                |       'L','l':  WrStr('Longitud de la lista es ');
                                        WrCard(Longitud(l), 1);
                                        WrLn;
                |       'F','f':  EXIT
                END
        END;
END DriLista.
```

-----•o•-----
-----•o•-----

Vamos a ver a continuación la implementación de los anillos usando la estructura lineal simplemente encadenada con acceso a punto de interés. Un detalle de implementación, que forma parte del invariante del bucle es que:

$$a\text{^}.Dcho = NIL \Leftrightarrow Es\_Vacio(a)$$

```
DEFINITION MODULE Anillo;

TYPE
   ANILLO;
   ITEM = CARDINAL;
   ERROR = (SinError, SinMemoria, AnilloVacio, AnilloNoCreado);
VAR
   error : ERROR;

PROCEDURE Crear() : ANILLO;
PROCEDURE Insertar(VAR a : ANILLO; e : ITEM);
PROCEDURE Eliminar(VAR a : ANILLO);
PROCEDURE Rotar_Dch(VAR a : ANILLO);
PROCEDURE Rotar_Izq(VAR a : ANILLO);
PROCEDURE Cabeza(a : ANILLO) : ITEM;
```

```
PROCEDURE Longitud(a : ANILLO) : CARDINAL;
PROCEDURE Es_Vacio(a : ANILLO) : BOOLEAN;

END Anillo.
```

-----•o•-----

Y el módulo de implementación.

```
IMPLEMENTATION MODULE Anillo;
FROM Storage IMPORT ALLOCATE, DEALLOCATE, Available;

TYPE
   ANILLO = POINTER TO CABECERA;
   PTR_NODO = POINTER TO NODO;
   CABECERA = RECORD
                 Hueco_Izq,
                 Hueco_Dch  : PTR_NODO;
              END;
   NODO = RECORD
             Dato : ITEM;
             Sig  : PTR_NODO;
          END;

PROCEDURE Crear() : ANILLO;
VAR
   a : ANILLO;
BEGIN
   error := SinError;
   IF NOT Available(SIZE(CABECERA)) THEN error := SinMemoria; RETURN NIL; END;
   NEW(a);
   a^.Hueco_Izq := NIL;
   a^.Hueco_Dch := NIL;
   RETURN a;
END Crear;

PROCEDURE Insertar(VAR a : ANILLO; e : ITEM);
VAR
   nuevo : PTR_NODO;
BEGIN
   IF a = NIL THEN error := AnilloNoCreado; RETURN; END;
   IF NOT Available(SIZE(NODO)) THEN error := SinMemoria; RETURN; END;
   error := SinError;
   NEW(nuevo);
   nuevo^.Dato := e;
   nuevo^.Sig := a^.Hueco_Dch;
   a^.Hueco_Dch := nuevo;
END Insertar;

PROCEDURE Eliminar(VAR a : ANILLO);
VAR
   aux, aux2 : PTR_NODO;
BEGIN
   IF a = NIL THEN error := AnilloNoCreado; RETURN; END;
```

```
   error := SinError;
   IF Es_Vacio(a) THEN RETURN; END;
   aux := a^.Hueco_Dch;
   IF a^.Hueco_Dch^.Sig = NIL THEN
        (* Estamos situados en el extremo derecho de la lista. *)
        IF a^.Hueco_Izq = NIL THEN
           (* La lista posee un solo elemento. *)
           a^.Hueco_Izq := NIL;
           a^.Hueco_Dch := NIL;
        ELSE
           (* La lista posee más de un elemento. *)
           (* Hay que hacer una rotación a la derecha. *)
           aux2 := a^.Hueco_Izq;
           Rotar_Dch(a);
           aux := aux2^.Sig;
           aux2^.Sig := NIL;
        END;
   ELSIF a^.Hueco_Izq = NIL THEN
        (* Estamos situados en el extremo izquierdo de la lista. *)
        (* La lista posee más de un elemento.                    *)
        a^.Hueco_Dch   := a^.Hueco_Dch^.Sig;
   ELSE
        (* Estamos enmedio de la lista.      *)
        (* La lista posee más de un elemento. *)
        a^.Hueco_Dch := a^.Hueco_Dch^.Sig;
   END;
   DISPOSE(aux);
END Eliminar;

PROCEDURE Rotar_Dch(VAR a : ANILLO);
VAR
   i : CARDINAL;
   aux : PTR_NODO;
BEGIN
   IF a = NIL THEN error := AnilloNoCreado; RETURN; END;
   error := SinError;
   IF Longitud(a) <= 1 THEN RETURN; END;
   IF a^.Hueco_Dch^.Sig = NIL THEN
        (* Estamos en el extremo derecho de la lista. *)
        FOR i := 1 TO Longitud(a) - 1 DO
           Rotar_Izq(a);
        END;
   ELSIF a^.Hueco_Izq = NIL THEN
        (* Estamos en el extremo izquierdo de la lista. *)
        a^.Hueco_Izq := a^.Hueco_Dch;
        a^.Hueco_Dch := a^.Hueco_Dch^.Sig;
        a^.Hueco_Izq^.Sig := NIL;
   ELSE
        (* Estamos enmedio de la lista. *)
        aux := a^.Hueco_Dch;
        a^.Hueco_Dch := a^.Hueco_Dch^.Sig;
        aux^.Sig := a^.Hueco_Izq;
        a^.Hueco_Izq := aux;
   END;
END Rotar_Dch;
```

```
PROCEDURE Rotar_Izq(VAR a : ANILLO);
VAR
   i : CARDINAL;
   aux : PTR_NODO;
BEGIN
   IF a = NIL THEN error := AnilloNoCreado; RETURN; END;
   error := SinError;
   IF Longitud(a) <= 1 THEN RETURN; END;
   IF a^.Hueco_Izq = NIL THEN
        (* Estamos en el extremo izquierdo de la lista. *)
        FOR i := 1 TO Longitud(a) - 1 DO
           Rotar_Dch(a);
        END;
   ELSE
        (* Estamos enmedio de la lista. *)
        aux := a^.Hueco_Izq;
        a^.Hueco_Izq := a^.Hueco_Izq^.Sig;
        aux^.Sig := a^.Hueco_Dch;
        a^.Hueco_Dch := aux;
   END;
END Rotar_Izq;

PROCEDURE Cabeza(a : ANILLO) : ITEM;
VAR
   basura : ITEM;
BEGIN
   IF a = NIL THEN error := AnilloNoCreado; RETURN basura; END;
   IF Es_Vacio(a) THEN error := AnilloVacio; RETURN basura; END;
   error := SinError;
   RETURN a^.Hueco_Dch^.Dato;
END Cabeza;

PROCEDURE Longitud(a: ANILLO) : CARDINAL;
VAR
   contador : CARDINAL;
   aux : PTR_NODO;
BEGIN
   IF a = NIL THEN error := AnilloNoCreado; RETURN 0; END;
   error := SinError;
   aux:= a^.Hueco_Izq;
   contador := 0;
   WHILE aux <> NIL DO
        INC(contador);
        aux := aux^.Sig;
   END;
   aux:= a^.Hueco_Dch;
   WHILE aux <> NIL DO
        INC(contador);
        aux := aux^.Sig;
   END;
   RETURN contador;
END Longitud;

PROCEDURE Es_Vacio(a : ANILLO) : BOOLEAN;
BEGIN
   IF a = NIL THEN error := AnilloNoCreado; RETURN FALSE; END;
```

```
   error := SinError;
   RETURN Longitud(a) = 0;
END Es_Vacio;

BEGIN
   error := SinError;
END Anillo.
```

-----•o•-----
-----•o•-----

Pasamos ahora a crear las colas con prioridad, suponiendo que el ITEM que usamos como tipo base, es un registro que contiene un campo llamado Prioridad, que poseerá una relación de orden total. Primero se insertan los elementos de menor prioridad.

```
DEFINITION MODULE ColaPrioridad;

IMPORT ListasDinamicas;

TYPE
        COLAPRIORIDAD;
        ITEM = ListasDinamicas.ITEM;
        ERRORPRIORIDAD = (noError,colaVacia,errorDesconocido);

VAR
        ErrorDeColaPrioridad: ERRORPRIORIDAD;

PROCEDURE Crear(): COLAPRIORIDAD;
PROCEDURE Vacia(q: COLAPRIORIDAD): BOOLEAN;
PROCEDURE Insertar(VAR q: COLAPRIORIDAD; x: ITEM);
PROCEDURE Extraer(VAR q: COLAPRIORIDAD);
PROCEDURE Frente(q: COLAPRIORIDAD): ITEM;

END ColaPrioridad.
```

Como puede observarse en este caso, vamos a hacer uso de las listas dinámicas para implementar las colas. Sin embargo, la línea

ITEM = ListasDinamicas.ITEM

situada en el módulo de definición de las colas con prioridad, da información al usuario de la cola de este hecho, y lo que es peor, le obliga a definir el tipo ITEM, no en la cola, sino en la lista dinámica (que no es el tipo que el usuario quiere utilizar). Para conseguir que el usuario se relacione sólo con el tipo cola, y a la vez conseguir que la implementación de las colas haga uso de la lista, bastará con cambiar la definición del tipo ITEM en las listas, y ponerla como:

ITEM = ColaPrioridad.ITEM

incluyendo en su módulo de definición, la línea:

IMPORT ColaPrioridad;

con lo que (hablando en terminología cliente/servidor), las listas se convierten en servidoras de las colas, y no al revés.

Con este hecho conseguimos que la definición de las listas haga uso del tipo ITEM que ponga el usuario en el módulo de definición de ColaPrioridad, y además, no habrá ningún

problema en que la implementación de las colas importe a su vez el tipo ListaDinamica. No existen ciclos de referencia puesto que las definiciones se referencian por un lado, y las implementaciones por otro.

-----•o•-----

He aquí el módulo de implementación correspondiente. La cola se implementa haciendo uso de las listas con punteros. Un tema fundamental a no olvidar cuando se implementa un TAD en base a otro, es el de la propagación de errores.

```
IMPLEMENTATION MODULE ColaPrioridad;

IMPORT ListasDinamicas;

TYPE
        COLAPRIORIDAD = ListasDinamicas.LISTA;

PROCEDURE BuscaPosicion(x: ITEM; q: COLAPRIORIDAD): CARDINAL;
VAR
        basura : CARDINAL;
        i,long:CARDINAL;
BEGIN
        long:=ListasDinamicas.Longitud(q);
        IF ListasDinamicas.ErrorDeLista THEN RETURN basura; END;
        i := 1;
        WHILE (i<=long) DO
                IF ListasDinamicas.Elemento(q,i).prioridad > x.prioridad THEN
                        RETURN i
                END;
                INC(i)
        END;
        RETURN long+1
END BuscaPosicion;

PROCEDURE Crear(): COLAPRIORIDAD;
VAR
        c : COLAPRIORIDAD;
BEGIN
        c := ListasDinamicas.Crear()
        IF ListasDinamicas.ErrorDeLista THEN
                ErrorDeColaPrioridad := errorDesconocido;
        ELSE
                ErrorDeColaPrioridad := noError;
        END;
        RETURN c;
END Crear;

PROCEDURE Vacia(q: COLAPRIORIDAD): BOOLEAN;
VAR
        retorno : BOOLEAN;
BEGIN
        retorno := ListasDinamicas.Vacia(q)
        IF ListasDinamicas.ErrorDeLista THEN
```

```
            ErrorDeColaPrioridad := errorDesconocido;
        ELSE
            ErrorDeColaPrioridad := noError;
        END;
        RETURN retorno;
END Vacia;

PROCEDURE Insertar(VAR q: COLAPRIORIDAD; x : ITEM);
BEGIN
        ListasDinamicas.Insertar(q,BuscaPosicion(x,q),x);
        IF ListasDinamicas.ErrorDeLista THEN
            ErrorDeColaPrioridad := errorDesconocido;
        ELSE
            ErrorDeColaPrioridad := noError;
        END;
END Insertar;

PROCEDURE Extraer(VAR q: COLAPRIORIDAD);
VAR
        resultado : BOOLEAN;
BEGIN
        resultado := Vacia(q);
        IF ListasDinamicas.ErrorDeLista THEN
            ErrorDeColaPrioridad := errorDesconocido;
        ELSIF NOT resultado THEN
            ListasDinamicas.Eliminar(q,1)
            ErrorDeColaPrioridad := noError;
        ELSE
            (* Puede ser error o no, dependiendo de lo que se haya dicho en la especificación.   *)
            ErrorDeColaPrioridad := colaVacia
        END
END Extraer;

PROCEDURE Frente(q: COLAPRIORIDAD): ITEM;
VAR
        x: ITEM;
BEGIN
        ErrorDeColaPrioridad := noError;
        x := ListasDinamicas.Elemento(q,1);
        IF ListasDinamicas.ErrorDeLista THEN
            (* Es posible también que el error se produzca por no haber sido creada.       *)
            ErrorDeColaPrioridad := colaVacia;
        END;
        RETURN x
END Frente;

BEGIN (* Del módulo de implementación.      *)
        ErrorDeColaPrioridad := noError;
END ColaPrioridad.
```

-----•o•-----

Este es el módulo driver que se encarga de verificar el funcionamiento de la cola con prioridad.

```
MODULE DriPrioridad;

FROM IO IMPORT WrLn,WrStr,WrCard,RdStr,RdChar,RdCard;
FROM ColaPrioridad IMPORT COLAPRIORIDAD, ITEM, ERRORPRIORIDAD,
Prioridad, ErrorDeColaPrioridad, Insertar, Extraer, Vacia, Crear,
Frente;

VAR
        c: COLAPRIORIDAD;
        elemento:ITEM;
        opcion:CARDINAL;

PROCEDURE Menu();
BEGIN
        WrStr ("Driver para Colas de Prioridad:");WrLn;
        WrStr (" Visualizar (M)enu");WrLn;
        WrStr (" (C)rear");WrLn;
        WrStr (" (I)nsertar ");WrLn;
        WrStr (" (E)xtraer");WrLn;
        WrStr (" (F)rente");WrLn;
        WrStr (" (V)acia");WrLn;
        WrStr (" Finalizar (Q)");WrLn;
END Menu;

BEGIN
        Menu();
        LOOP
                WrLn;WrStr('opción ');
                opcion := RdCard();
                CASE opcion OF
                |       'm','M': Menu();
                |       'c','C': c :=Crear();
                |       'i','I':   WrStr('INSERCION');WrLn;
                                  WrStr('Elemento a insertar (STRING): ');
                                  RdStr(elemento.contenido);
                                  WrStr('Prioridad: ');
                                  elemento.prioridad := RdCard();
                                  Insertar(c, elemento);
                |       'e','E': WrStr('EXTRACCION'); WrLn;
                                        Extraer(c);
                |       'f','F':  WrStr('FRENTE: ...');
                                        elemento := Frente(c);
                                        WrStr(elemento.contenido);
                                        WrStr(" con prioridad ");
                                        WrCard(elemento.prioridad,2); WrLn
                |       'v','V':  IF Vacia(c) THEN
                                                WrStr('Cola vacia'); WrLn
                                        ELSE
                                                WrStr('Cola no vacia'); WrLn
                                        END;
                |       'q','Q': EXIT;
                END;
                IF ErrorDeColaPrioridad = noError THEN
                        WrStr("No se han producido errores."); WrLn;
```

```
            ELSIF ErrorDeColaPrioridad = colaVacia THEN
                WrStr("La cola está vacía"); WrLn;
            ELSE
                WrStr("Se ha producido otro error."); WrLn
            END;
        END;
END DriPrioridad.
```

-----•o•-----

Vamos a inventarnos ahora nuestro propio TAD. Vamos acrear un nuevo tipo que será el tipo Polinomio. El ITEM heredado de las ListasDinámicas debe ser un registro (llamado monomio) que contenga al menos dos campos:

- Coeficiente.
- Exponente.

El polinomio se implementa como una lista dinámica de estos registros. Los monomios se almacenan en orden decreciente según el exponente.

```
DEFINITION MODULE Polinomio;

TYPE
    POLINOMIO;

PROCEDURE CrearPolinomio(): POLINOMIO;
PROCEDURE Grado(P: POLINOMIO): LONGREAL;
PROCEDURE Coeficiente(i: LONGREAL; P: POLINOMIO): LONGREAL;
PROCEDURE Suma(P,Q: POLINOMIO): POLINOMIO;
PROCEDURE Resta(P,Q: POLINOMIO): POLINOMIO;
PROCEDURE Producto(P,Q: POLINOMIO): POLINOMIO;
PROCEDURE EscribePolinomio(P: POLINOMIO);

ENDPolinomio.
```

-----•o•-----

Veamos a continuación la implementación del tipo Polinomio. No se ha hecho control de errores para facilitar la lectura. Nótese el mal uso de la palabra MONOMIO para denotar al ITEM. Debiera haberse empleado otro identificador, ya que cualquier lector tenderá a pensar que el tipo MONOMIO no es más que un polinomio con un solo término, lo cual no es cierto. ¡¡Cuidado con los nombres de identificador!!.

```
IMPLEMENTATION MODULE Polinomio;

IMPORT ListasDinamicas;
FROM IO IMPORT WrStr, WrLn, RdLngReal, WrLngReal;

TYPE
    POLINOMIO = ListasDinamicas.LISTA;
    MONOMIO = ListasDinamicas.ITEM;

PROCEDURE InsertarEnPolinomio(p: POLINOMIO; m: MONOMIO);
```

```
VAR
        i,long:CARDINAL;
        temp:MONOMIO;
BEGIN
        long:=ListasDinamicas.Longitud(p);
        i := 1;
        WHILE (i<=long) DO
                temp := ListasDinamicas.Elemento(p,i);
                IF temp.exponente = m.exponente THEN
                        temp.coeficiente := temp.coeficiente + m.coeficiente;
                        ListasDinamicas.Cambiar(p,temp,i);
                        RETURN
                ELSIF temp.exponente < m.exponente THEN
                        ListasDinamicas.Insertar(p, i, m);
                        RETURN
                END;
                INC(i)
        END;
        ListasDinamicas.Insertar(p,i,m);
END InsertarEnPolinomio;

PROCEDURE CrearPolinomio(): POLINOMIO;
VAR
        p: POLINOMIO;
        m: MONOMIO;
BEGIN
        p := ListasDinamicas.Crear();
        WrStr('Inserte 0 como coeficiente para terminar'); WrLn;
        LOOP
                WrStr('Coeficiente: ');
                m.coeficiente := RdLngReal();
                IF m.coeficiente = 0.0 THEN
                        RETURN p
                END;
                WrStr('Exponente: ');
                m.exponente := RdLngReal();
                InsertarEnPolinomio(p,m)
        END
END CrearPolinomio;

PROCEDURE Grado(P: POLINOMIO): LONGREAL;
BEGIN
        IF ListasDinamicas.Vacia(P) THEN
                RETURN 0.0
        ELSE
                RETURN ListasDinamicas.Elemento(P,1).exponente
        END
END Grado;

PROCEDURE Coeficiente(i: LONGREAL; P: POLINOMIO): LONGREAL;
VAR
        j,long:CARDINAL;
        temp:MONOMIO;
BEGIN
        long:=ListasDinamicas.Longitud(P);
        j:=1;
```

```
        WHILE (j<=long) AND (ListasDinamicas.Elemento(P, j).exponente # i) DO
                INC(j)
        END;
        IF j>long THEN
                WrS("Error: El polinomio no tiene término de ese grado");
                HALT
        ELSE
                RETURN ListasDinamicas.Elemento(P,j).coeficiente
        END
END Coeficiente;

PROCEDURE Suma(P,Q: POLINOMIO): POLINOMIO;
VAR
        R: POLINOMIO;
        i,long: CARDINAL;
BEGIN
        R := ListasDinamicas.Crear();
        long := ListasDinamicas.Longitud(P);
        FOR i :=1 TO long DO
                InsertarEnPolinomio(R,ListasDinamicas.Elemento(P,i))
        END;
        long := ListasDinamicas.Longitud(Q);
        FOR i := 1 TO long DO
                InsertarEnPolinomio(R,ListasDinamicas.Elemento(Q,i))
        END;
        RETURN R
END Suma;

PROCEDURE Resta(P,Q: POLINOMIO): POLINOMIO;
VAR
        R: POLINOMIO;
        aux: MONOMIO;
        i,long: CARDINAL;
BEGIN
        R := ListasDinamicas.Crear();
        long := ListasDinamicas.Longitud(P);
        FOR i := 1 TO long DO
                aux := ListasDinamicas.Elemento(P,i);
                aux.coeficiente := aux.coeficiente;
                InsertarEnPolinomio(R,aux)
        END;
        long := ListasDinamicas.Longitud(Q);
        FOR i := 1 TO long DO
                aux := ListasDinamicas.Elemento(Q,i);
                aux.coeficiente := aux.coeficiente;
                InsertarEnPolinomio(R,aux)
        END;
        RETURN R
END Resta;

PROCEDURE Producto(P,Q: POLINOMIO): POLINOMIO;
VAR
        R: POLINOMIO;
        aux, auxq, auxp: MONOMIO;
        i,j: CARDINAL;
BEGIN
```

```
        R := ListasDinamicas.Crear();
        FOR i :=1 TO ListasDinamicas.Longitud(P) DO
                FOR j := 1 TO ListasDinamicas.Longitud(Q) DO
                        auxp := ListasDinamicas.Elemento(P,i);
                        auxq := ListasDinamicas.Elemento(Q,j);
                        aux.exponente := auxp.exponente + auxq.exponente;
                        aux.coeficiente := auxp.coeficiente * auxq.coeficiente;
                        InsertarEnPolinomio(R,aux)
                END
        END;
        RETURN R
END Producto;

PROCEDURE EscribePolinomio(P: POLINOMIO);
VAR
        i : CARDINAL;
        aux : MONOMIO;
BEGIN
        FOR i := 1 TO ListasDinamicas.Longitud(P) DO
                aux := ListasDinamicas.Elemento(P,i);
                IF aux.coeficiente >= 0.0 THEN
                        WrStr(" + ")
                END;
                WrLngReal(aux.coeficiente,2,2);
                WrStr("x^");
                WrLngReal(aux.exponente,2,2)
        END;
        WrLn;
END EscribePolinomio;

END Polinomio.
```

2.7 EJERCICIOS.

1.- Comprobar algebraicamente mediante reducciones que la operación Rotar_Dcha() aplicada tres veces al anillo Insertar(a, Insertar(b, Insertar(c, Crear))), lo deja invariante.

2.- Construir las especificaciones formales de las siguientes operaciones:

Último: Lista $\not\longrightarrow$ Elemento

que devuelve el último elemento de la lista (el situado más a la derecha).

Está_en: Elemento × Lista $\longrightarrow$ Lógico

Devuelve Verdad si el Elemento está en la Lista, y Falso en otro caso.

Duplicar: Lista $\longrightarrow$ Lista

Devuelve una lista en la que cada elemento de la lista de entrada ha sido duplicado. Ej. Duplicar('H','O','L','A') devuelve ('H','H','O','O','L','L','A','A').

3.- Especificar las ecuaciones que describen el comportamiento de:

Concatenar(l_1,l_2)

PREcondición: $l_1 = (a_1, a_2, .., a_n)$

$l_2 = (b_1, b_2, .., b_m)$

POSTcondición: Concatenar(l_1, l_2)=(a_1,a_2,..,a_n,b_1,b_2,..,b_m)

Posición(i,l)

PREcondición: $l = (a_1, a_2, .., a_n)$

POSTcondición: $\text{Posición}(i,l) = \begin{cases} 0 & si\ \forall k,\ a_k \neq i \\ j & si\ \exists j,\ a_j = i \end{cases}$

4.-Construir las dos especificaciones algebraicas posibles de Cadena de caracteres usando los dos grupos de generadores que hemos visto.

5.- Especificar operaciones para eliminar, insertar y sustituir subcadenas en una cadena.

6.- Haciendo uso de las listas funcionales, especificar la operación

$$\text{Incluir: Lista} \times \mathbb{N} \times \text{Elemento} \nrightarrow \text{Lista}$$

de manera que Incluir(l, p, e), incluirá en la posición p de l, al elemento e, con la precondición $1 \leq p \leq \text{Longitud(l)} + 1$.

Para ello, se supone que la Cabeza(l) corresponde a la posición 1; Cabeza(Cola(l)) corresponde a la 2; Cabeza(Cola(Cola(l))) a la 3; y así sucesivamente.

Hacer lo mismo con la operación:

$$\text{Quitar: Lista} \times \mathbb{N} \longrightarrow \text{Lista}$$

7.- Escribir una especificación formal de una operación que retorne el último elemento de una lista funcional.

8.- Escribir una especificación formal de una operación que retorne el elemento i-ésimo de una lista funcional, supuesto que la cabeza está en la primera posición.

9.- Dado el TAD Lista posicional, escribir la especificación formal de las siguientes operaciones:

(Intercambia en una lista los dos elementos cuyas posiciones se dan. *)*

$\text{Cambia: Lista} \times \mathbb{N} \times \mathbb{N} \nrightarrow \text{Lista}$

(Indica si dos listas poseen los mismo elementos en las misma posiciones. *)*

$\text{Igual: Lista} \times \text{Lista} \longrightarrow \text{Lógico}$

(Ordena una lista. *)*

$\text{Ordenar: Lista} \longrightarrow \text{Lista}$

Para las dos últimas operaciones, suponer que el tipo base posee la operación <=. Indicar las precondiciones que se estimen oportunas.

10.- En el tipo Fichero secuencial, crear la operación:

$\text{Reescribir: Elemento} \times \text{Fichero} \nrightarrow \text{Fichero}$

que reescribe el registro sobre el que se halla situada la ventana. Indicar las precondiciones que se estimen oportunas.

11.- Escribir una especificación formal de Rotar_Izq, para el anillo restringido, o sea, el que consta sólo de las operaciones:

* Crear.

* Insertar en la cabeza.
* Eliminar la cabeza.
* Rotar a la derecha.
* Longitud.
* Es_Vacía.

12.- En las listas posicionales, dar una especificación formal del comportamiento de la operación

$$\text{Buscar: Lista} \times \text{Elemento} \longrightarrow \mathbb{N}$$

cuyo objetivo sea retornar la posición i-ésima en la que se almacena el valor dado como argumento. Ej.:

Buscar([A,B,C,G,H,S,C,A,G], H) = 5
Buscar([A,B,C,G,H,S,C,A,G], P) = 0
Buscar([A,B,C,G,H,S,C,A,G], G) = 4

donde [A,B,C,G,H,S,C,A,G] representa a una lista posicional, con objeto de clarificar este ejemplo.

13.- Con cadenas de caracteres, hacer la especificación de la operación subcadena, que devuelva la posición de comienzo de una subcadena en otra más grande.

14.- Evaluar

-Cima(Desapilar(Desapilar(Apilar(3, Desapilar(Apilar(4, Apilar(5, Apilar(6, Crear)))))))).
- Es_Vacío(Desapilar(Desapilar(Apilar(8, Crear)))).

hasta llegar a la forma canónica asociada, indicando en cada paso el axioma aplicado.

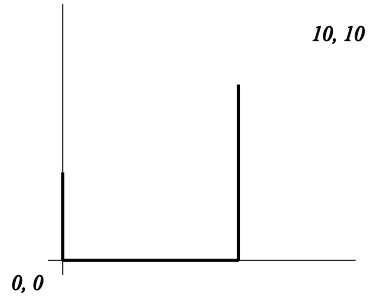
15.- Crear el tipo Punto_Cardinal que poseerá 4 constantes: Norte, Sur, Este y Oeste.

16.- Crear el tipo Línea_Cartesiana, que representará todas las posibles línea quebradas, en las que los segmentos estarán en dirección horizontal o vertical; la longitud de cada segmento vendrá dada por un natural. Habrá un generador inicial que posicionará el lápiz:

$$\text{Sitúa: } Z \times Z \longrightarrow \text{Línea\_Cartesiana}$$

y que dibujará un punto (segmento de longitud 0). El otro generador será

$$\text{Línea: Línea\_Cartesiana} \times \text{Punto\_Cardinal} \times \mathbb{N} \longrightarrow \text{Línea\_Cartesiana}$$



Se supone que cada línea se dibuja a partir de donde acabó el trazo de la línea anterior, y en el sentido especificado.

Nótese que varios segmentos consecutivos en el mismo sentido, equivalen a un único segmento en el mismo sentido, y de longitud suma de los anteriores. Asimismo, un segmento de longitud 0 puede eliminarse sin más.

Incluir además las operaciones

$$\text{Horizontal: Línea\_Cartesiana} \longrightarrow \text{Lógico}$$
$$\text{Vertical: Línea\_Cartesiana} \longrightarrow \text{Lógico}$$

que devuelven Verdad cuando la línea cartesiana de entrada corresponde únicamente al dibujo de

una línea horizontal o vertical respectívamente.

Ej.: Línea(Línea(Linea(Sitúa(10,10), Sur, 10), Oeste, 10), Norte, 5)
crearía el dibujo de la figura.

17.- Con el tipo anterior, crear las operaciones

$$\text{Cuadrado: } Z \times Z \times \mathbb{N} \longrightarrow \text{Línea\_Cartesiana}$$

donde los dos primeros parámetros representan la posición de la esquina inferior izquierda, y el tercero, representa la longitud del lado. El significado de esta operación será el evidente.

18.- Crear el operador

$$\text{Rectángulo: } Z \times Z \times \mathbb{N} \times \mathbb{N} \longrightarrow \text{Línea\_Cartesiana}$$

que será igual que el anterior, excepto por el hecho de que se le pasan 4 parámetros. Los dos últimos representa el ancho y el alto.

19.- ¿Las figuras geométricas que se pueden formar con este tipo, tienen una única representación canónica? Ayuda: ¿De cuántas formas podemos representar un cuadrado, de manera que el trazo acabe en la misma posición?

20.- Crear el predicado Es_cuadrado: Línea_Cartesiana $\longrightarrow$ Lógico, que devuelva V si su parámetro posee cuatro segmentos, y conforma un cuadrado.

21.- Crear el predicado

$$\text{Son\_cuadrados\_iguales: Línea\_Cartesiana} \times \text{Línea\_Cartesiana} \longrightarrow \text{Lógico}$$

que devuelva Verdadero si sus parámetros son ambos cuadrados, de igual tamaño, y con el final del trazo en el mismo punto.

22.- Una manera de representar las pilas de números de naturales del 1 al n, consiste en imaginar que la pila es, en realidad, un número natural, expresado en base n, que en cada momento tiene tantos dígitos como elementos hay en la pila, y que el dígito menos significativo es la cima de la pila. Implementar el tipo pila acotada basándose en esta estrategia.

Ayuda: Dado que en Modula-2, un entero sólo tiene 16 bits, suponed que en la pila sólo se almacenan naturales del 0 al 3, y que el tamaño máximo de la pila es de 7 elementos. Para poder representar una pila que contenga sólo ceros, indicaremos el final de la pila con un 1. Así: 10032(4, representará una pila en la que se han apilado (por orden): el el 0, el 0, el 3, y el 2.

23.- Hacer lo mismo con colas.

24.- Decimos que un palíndromo es una frase que se lee igual de izq. a dcha. que de dcha. a izq., descartando los espacios en blanco. P.ej.: "Dábale arroz a la zorra el abad". Especificar la función capicúa sobre una cola, (suponiendo que el tipo base de la cola son caracteres, que poseen la operación =: Carácter × Carácter $\longrightarrow$ Lógico).

25.- Un agente del CSID, ha inventado un método para codificar mensajes de alto secreto gubernamental. El método consiste en dos fases:

a) Sea X el mensaje original. Toda tira de caracteres no vocales se sustituye por su inversa. El resultado es X'.
b) Si X' es de longitud n, el mensaje codificado definitivo X'' viene dado por:
X''=X'(1).X'(n).X(2).X(n-1)..., donde X'(i) representa el carácter i-ésimo de X'.

Así p.ej., el mensaje X="Construcción de bombas", pasaría a X'="Cortsnucciód neb obmas", y por último a: X''="Csoarmtbson ubcecni ód".

Escribir los algoritmos de codificación y descodificación. Suponer que se posee el TAD Pila, y el TAD Cola doble, empleando como tipo base el carácter.

26.- Implementar una lista doblemente encadenada con cursores. Nótese que en condiciones normales se necesitarían dos campos para cursores: uno al siguiente elemento, y otro al anterior. Sin embargo, si la dimensión del vector es lo bastante pequeña, será posible codificar el valor de los dos cursores, como un único número entero que caiga dentro del rango de la máquina. Calcular además, la relación que debe haber entre la dimensión del vector N y el valor entero más grande de la máquina (MAX(INTEGER)).

27.- Un problema muy concreto de las estructuras lineales representadas con punteros, es la supresión de la estructura completa. Para recuperar el espacio que ocupa esta estructura es necesario recorrerla toda y liberar las celdas individualmente, lo cual representa un coste lineal. Otra opción consiste en mantener en el programa una lista de elementos que se han querido eliminar, lista que inicialmente estará vacía, y a la cual van a parar las celdas al borrar la estructura entera. Cuando alguna estructura necesita obtener una celda para guardar nuevos elementos, primero consulta si hay espacio en la lista libre: si lo hay lo usa, y si no, lo crea con NEW. Modificar la representación encadenada con punteros, para adaptarla a este esquema; no olvidar la operación *destruir* de la estructura entera. También es bueno que haya una operación que libere físicamente la lista de libres, por si se agota la memoria requerida por otros tipos.

28.- Considerar el TAD formado por matrices cuadradas de enteros, y las operaciones matriz_cero, definir el valor de una posición de la matriz, sumar, multiplicar, y calcular la traspuesta. Implementar el tipo en los dos casos siguientes:

a) Usando matrices bidimensionales de CARDINAL.

b) Usando listas de tuplas de la forma (columna, fila, valor), donde sólo se almacenan las posiciones no nulas. Discutir la conveniencia de ordenar la lista siguiendo algún criterio.

29.- Hacer la especificación de una operación que obtenga una lista con todas las permutaciones con repetición de los naturales del 1 al n, tomados de n en n. La sintaxis de la operación es:
permutaciones: $\mathbb{N} \longrightarrow$ ListaL
donde ListaL es un tipo Lista cuyo Elemento es del tipo ListaN, que, a su vez es un tipo Lista cuyo ELemento es del tipo Natural.

P.ej.: permutaciones(2) devolvería:
Construir(*Construir(2, Construir(2, Crear))*,
Construir(*Construir(2, Construir(1, Crear))*,
Construir(*Construir(1, Construir(2, Crear))*,
Construir(*Construir(1, Construir(1, Crear))*,

Crear))))

Los valores en itálica representan datos de tipo ListaN, y el resto es de tipo ListaL.

30.- Se llama lista generalizada a una lista que, o bien es vacía, o bien es de la forma Cons(h, t), donde t es una lista generalizada, y h es, o bien otra lista generalizada, o bien un átomo del tipo base.

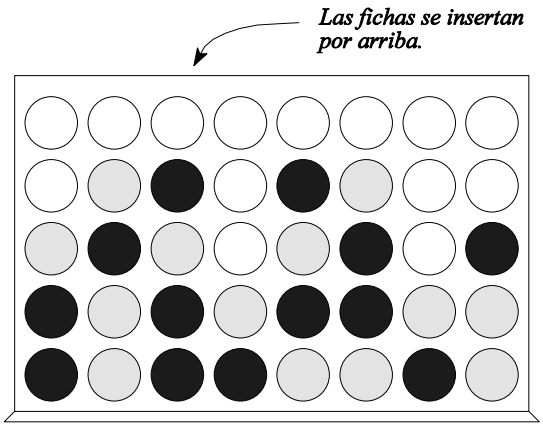
Estudiar qué operaciones generadoras serían necesarias para representar las formas canónicas de este tipo.

31.- Hacer una operación que extraiga todos los átomos de una lista generalizada, y los inserte ordenadamente en una lista normal.

32.- Se pretende representar el juego de las 4 en raya, a través de la especificación del TAD 4EnRaya correspondiente. Recordemos que en este juego existe un tablero vertical dividido en casillas, de manera que, alternativamente, cada uno de los dos jugadores va insertando una ficha de su color. Las fichas se insertan por la parte superior del tablero, en la columna que quiera, y van a parar sobre la última ficha insertada en esa columna, como si de una secuencia de pilas de fichas se tratase. Gana el jugador que consiga situar cuatro fichas del mismo color en línea, ya sea vertical, horizontal o diagonal.

Establecer las operaciones generadoras necesarias para representar el TAD. ¿Un tablero queda representado por un único término de la especificación creada? En otras palabras, ¿la familia de generadores es libre?

33.- Siguiendo con el ejercicio anterior, modificarlo para establecer una cota superior al número de fichas que se pueden insertar en cada columna. Establecer asimismo una cota al número de columnas del tablero. Modificar adecuadamente la especificación.



34.- Siguiendo con el ejercicio anterior, especificar una operación que devuelva el color de la ficha situada en una posición del tablero que vendrá determinada por sus coordenadas en el eje horizontal y en el vertical. Puede suponerse la existencia de un color nulo.

35.- Siguiendo con el ejercicio anterior, especificar una operación que diga si en una disposición del tablero hay o no cuatro fichas en línea, y en caso afirmativo, cual es el color de las fichas de tal línea.