Departamento de Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

Apuntes para la asignatura

Informática

Facultad de Ciencias (Matemáticas)

http://www.lcc.uma.es/personal/pepeg/mates

Tema 12. Estructuras de datos dinámicas

Bibliografía

- *Programación 1*. José A. Cerrada y Manuel Collado. Universidad Nacional de Educación a Distancia.
- *Programming with TopSpeed Modula-2*. Barry Cornelius. Addison-Wesley.
- *Fundamentos de programación. Algoritmos y Estructuras de datos*. L. Joyanes Aguilar. McGraw Hill.
- *Pascal y estructuras de datos.* Nell Dale y Susan C. Lilly. McGraw-Hill

Introducción

Las estructuras de datos dinámicas permiten trabajar con variables cuyo tamaño no es conocido en tiempo de compilación. Estas estructuras pueden crecer o decrecer durante la ejecución del programa, según sea necesario. En *Modula-2*, estas estructuras son implementadas utilizando el tipo puntero (POINTER).

En este tema, introducimos el tipo puntero y sus operaciones asociadas. Una de las estructuras dinámicas más simples que se pueden implementar con punteros es la lista enlazada. Implementaremos un módulo de biblioteca para el manejo de listas enlazadas.

12.1 Gestión dinámica de la memoria

Hasta ahora, todos los tipos de datos estudiados, ya sean simples o estructurados, tienen una propiedad común: son *estáticos*. Esto significa que las variables de estos tipos mantienen la misma estructura y tamaño durante toda la ejecución del programa. El tamaño de estas variables se debe anticipar cuando se escribe el programa y queda fijado durante la compilación (*tiempo de compilación*). Es imposible ampliar el tamaño de la estructura durante la ejecución del programa (*tiempo de ejecución*).

Así, por ejemplo:

- Si declaramos un Array de cinco elementos de tipo INTEGER, el contenido de cada una de las casillas podrá cambiar, pero el tamaño máximo de este será siempre fijo (cinco elementos enteros).
- Si declaramos una variable REAL puede cambiar su contenido, pero no el tamaño asignado por el compilador para ella (el necesario para un único valor de tipo REAL).

Sin embargo, hay muchas situaciones en las que no sólo debe cambiar el contenido o valor de una variable, sino también su tamaño. La estructura debe ir creciendo y decreciendo mientras se ejecuta el programa. En esos casos la cantidad de memoria necesaria para almacenar cierta información no se puede determinar en tiempo de compilación, sino que hay que esperar a que el programa se esté ejecutando (*tiempo de ejecución*).

La técnica usada para manejar estas situaciones es la *asignación dinámica de memoria*. Con esta técnica tendremos *variables dinámicas*, que se caracterizan porque pueden crearse o destruirse en tiempo de ejecución. Veremos que estas estructuras dinámicas son extremadamente flexibles, aunque es muy fácil cometer errores cuando son programadas.

El siguiente ejemplo muestra la necesidad de esta técnica: supongamos que se desea escribir un programa que lea una cantidad de números reales variable desde teclado, calcule la media de todos ellos y, por último, muestre por pantalla aquellos números leídos cuyos valores sean mayores a la media. La cantidad total de números se leerá al principio del programa.

Si nos dicen que el número máximo de valores que se puede leer es MAX, podemos definir un array de reales con este tamaño. Sin embargo, esta aproximación tiene un problema. Si MAX es grande, estaremos desperdiciando bastante memoria si el total introducido luego es bastante menor.

Otro problema más difícil de resolver es que no se sepa de antemano el número máximo de números a leer. En este caso la solución del Array no es válida, ya que, sea cual sea el valor que demos a MAX, siempre podemos quedarnos cortos. Una gran cantidad de problemas reales para los que se utilizan los ordenadores tiene esta característica: no se conoce la cantidad máxima de datos a tratar.

Para resolver el problema anterior, necesitamos que el lenguaje de programación permita:

1) Ir creando nuevas variables cada vez que las necesitemos (cada vez que vayamos a leer un número en nuestro ejemplo).
2) Acceder a las variables creadas.
3) Destruir variables cuando ya no sean necesarias.

Conseguiremos las dos primeras características mediante instrucciones de *asignación* y *liberación* dinámica de memoria. La tercera se consigue mediante las variables de *tipo puntero*.

Mediante la asignación dinámica de memoria conseguiremos estructuras de datos cuyo tamaño será variable. Es importante observar que el tamaño de estos datos solo estará acotado por el tamaño máximo de la memoria del ordenador en la cual se esté ejecutando el programa, pero no por un valor máximo determinado a priori.

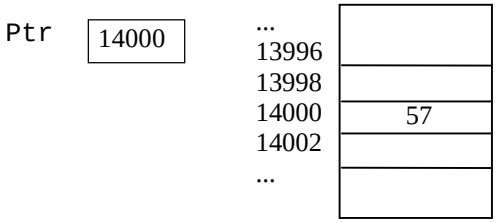
12.2 El tipo POINTER

Una *variable de tipo puntero* no contiene un dato, como los tipos estudiados hasta ahora, sino que contiene la dirección (o posición) que ocupa en la memoria del ordenador otra variable. Se dice que la variable puntero apunta a esta última variable. A la variable puntero se la llama también *variable referencia*. A la variable apuntada se la llama *variable referenciada* o *anónima*.

Cuando se declara una variable puntero es necesario indicar el tipo de la variable apuntada. El siguiente ejemplo declara una variable, cuyo nombre es `Ptr`, como una variable de tipo puntero a un valor entero:

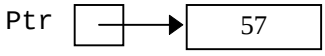
```
VAR
  Ptr : POINTER TO INTEGER;
```

Esto significa que el valor de la variable `Ptr` es una posición en la memoria del ordenador en la cual hay situada un dato de tipo INTEGER:



En el ejemplo, la variable `Ptr` contiene como valor la posición de memoria 14000. En esta dirección de memoria está situado un valor de tipo INTEGER (57).

Normalmente, las direcciones de memoria concretas almacenadas en las variables puntero no son importantes, por lo que la situación anterior la representamos como:

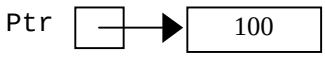


Es decir, `Ptr` apunta a una posición de memoria en la cual se almacena una variable de tipo INTEGER cuyo valor es 57. En este ejemplo la variable referencia es `Ptr`, mientras que la referenciada es la casilla en la que se encuentra el valor 57. Esta última se llama también anónima porque no tiene nombre.

El único modo de acceder a la variable anónima es mediante el puntero que la referencia. Si queremos asignar el valor 100 a la variable anónima, la instrucción correspondiente en *Modula-2* es:

```
Ptr^ := 100;
```

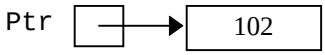
Esta instrucción se lee: asignar a la variable apuntada por `Ptr` el valor 100. Para indicar que lo que queremos modificar es la variable anónima hay que escribir un signo ^ tras el nombre de la variable referencia. (Obsérvese que `Ptr^` se asemeja con partir de la variable `Ptr` en el dibujo y seguir la flecha). Tras esta instrucción la situación es:



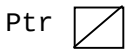
Si queremos acceder a la variable anónima para leer su valor también hemos de utilizar el nombre de la variable referencia y un signo ^. Así, para incrementar en dos el valor de la variable anónima hemos de escribir:

```
Ptr^ := 2 + Ptr^;
```

con lo obtenemos:



Una variable puntero puede almacenar también un valor especial: `NIL`. Este valor indica que el puntero no apunta a nada (no contiene una dirección de memoria válida). Representaremos gráficamente con una línea diagonal un puntero que contiene el valor `NIL`:



Normalmente el valor `NIL` se utiliza para inicializar las variables de tipo puntero. Esta inicialización no es automática por lo que debe ser realizada explícitamente por el programador (las variables punteros no son una excepción a la regla en *Modula-2*, por lo que su valor predeterminado es indeterminado).

12.3 Procedimientos NEW y DISPOSE

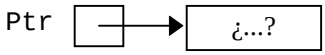
Cuando declaramos una variable de tipo puntero como por ejemplo

```
VAR Ptr : POINTER TO INTEGER;
```

la variable referencia `Ptr` se crea estáticamente, en tiempo de compilación, pero la variable referenciada o anónima debe crearse posteriormente, en tiempo de ejecución. La forma de crear una variable anónima es mediante el procedimiento de asignación dinámica de memoria **`NEW`**. Este procedimiento toma como argumento una variable de tipo puntero, crea una nueva variable del tipo al cual apunta el argumento y hace que el puntero argumento apunte a la variable dinámica creada. El valor previo que tuviese la variable puntero se pierde. Tras ejecutar la instrucción

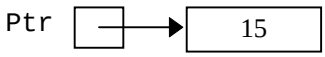
```
NEW(Ptr);
```

la situación es la siguiente:



Obsérvese que se ha creado una nueva variable anónima y que `Ptr` apunta a ella, pero que la variable anónima no toma ningún valor por defecto. El programador debe darle un valor explícitamente:

```
Ptr^ := 15;
```

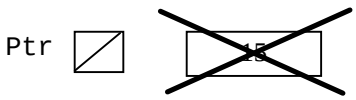


Con este procedimiento podemos crear tantas variables (dinámicamente) como necesitemos durante la ejecución del programa, y podemos referenciarlas mediante punteros, con lo que solucionamos el problema de partida.

Al crear una variable dinámica la cantidad de memoria libre en el ordenador disminuye. Las variables dinámicas, una vez creadas, siguen ocupando espacio de memoria hasta que se destruyan explícitamente. Cuando se destruye una variable dinámica el espacio que ocupaba vuelve a quedar disponible. Para destruir una variable dinámica se utiliza el procedimiento **`DISPOSE`**. Este procedimiento toma como argumento un puntero a una variable dinámica, libera la memoria que la variable dinámica ocupa y pone el puntero a valor `NIL` para indicar que ya no apunta a nada. Tras ejecutar

```
DISPOSE(Ptr);
```

la situación es la siguiente:



Una vez destruida, la variable dinámica no existe, por lo que es ilegal intentar acceder a ella. La memoria libre del sistema aumenta al destruir una variable dinámica.

En realidad, **`NEW`** y **`DISPOSE`** son solo dos modos más sencillos de escribir llamadas a dos subprogramas más complicados `ALLOCATE` y `DEALLOCATE` que se encuentran en la librería `Storage`. Por esto, cuando un programa va a utilizar **`NEW`** y **`DISPOSE,`** debe importar ALLOCATE y DEALLOCATE:

```
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
```

Nótese que, curiosamente, no es válido importar **`NEW`** y **`DISPOSE`** directamente.

12.4 Operaciones con Punteros

Las operaciones que se pueden realizar con variables de tipo puntero son básicamente tres:

1) *Derreferenciación*.
2) *Asignación*
3) *Comparación*

La *derreferenciación* es la operación mediante la cual accedemos a la variable anónima apuntada por el puntero, añadiendo para ello el operador ^ tras el nombre de la variable puntero.

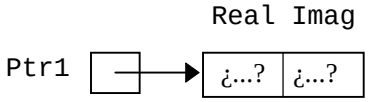
A modo de ejemplo, supongamos las siguientes declaraciones de tipos y variables:

```
TYPE
```

```
  COMPLEJO  =   RECORD
                  Real, Imag : REAL
                END;
  PCOMPLEJO  =  POINTER TO COMPLEJO;

VAR
  Ptr1, Ptr2 : PCOMPLEJO;
```

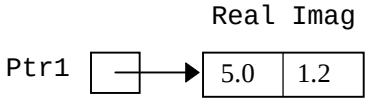
Para comenzar a trabajar con estas variables, crearemos una variable dinámica y la referenciaremos mediante la variable `Ptr1`.

```
NEW(Ptr1);
```



La nueva variable dinámica es un registro que cuenta con dos campos (`Real` e `Imag`) ambos de tipo REAL. Sin embargo, los valores de estos campos están indefinidos. Para asignar valores a estos campos es necesario derreferenciar la variable puntero:

```
Ptr1^.Real  := 5.0;
Ptr1^.Imag  := 1.2;
```



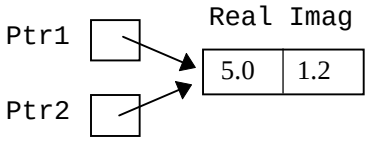
El mismo resultado puede ser obtenido usando la sentencia WITH:

```
WITH Ptr1^ DO
  Real   := 5.0;
  Imag := 1.2
END;
```

La *asignación* es la operación que permite asignar a una variable puntero otra variable del mismo tipo. El efecto conseguido al asignar dos variables puntero es que ambas apuntan a la misma variable dinámica. Si en la situación actual ejecutamos la sentencia

```
Ptr2 := Ptr1;
```

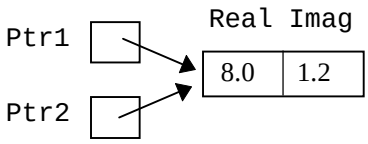
las variables quedan:



Es decir, podemos acceder a la variable anónima por dos caminos: `Ptr1^` o `Ptr2^`. En este caso decimos que la variable anónima está múltiplemente referenciada. Como no hay dos copias de la variable anónima, si se modifica su contenido mediante un camino, al acceder mediante el otro veremos el cambio. Por ejemplo, el siguiente código produce como salida por pantalla 8.0:

```
Ptr1^.Real := 8.0;
WrReal(Ptr2^.Real, 1, 0);
```

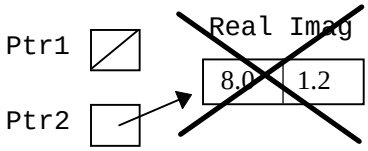
ya que el estado de la memoria es:



Un fallo muy común a la hora de trabajar con referencias múltiples es que se libere la variable dinámica mediante una de las referencias pero no se ponga a `NIL` la otra:

```
DISPOSE(Ptr1);
WrReal(Ptr2^.Real, 1, 0); (* ERROR !!! *)
```

produciría un error ya que la variable anónima ha sido destruida, por lo que no se debe acceder a ella a través de `Ptr2`:



Por otro lado, a una variable anónima siempre se le puede asignar el valor `NIL`, sea cual sea el tipo de la variable anónima. Éste es el segundo tipo de asignación disponible para variables punteros

```
Ptr2 := NIL;
```

Es un error intentar derreferenciar un puntero cuyo valor sea `NIL`. En este caso, el programa finaliza inmediatamente con el consiguiente error. Este es uno de los errores más comunes que se comenten al programar con punteros. Es MUY IMPORTANTE asegurarnos de que un puntero no vale `NIL` antes de derreferenciarlo.

Por último, las variables de tipo puntero pueden compararse entre sí si tienen el mismo tipo o con `NIL` (siempre). El resultado de la expresión Ptr1 = Ptr2 es TRUE si las dos variables apuntan a la misma variable anónima (referencias múltiples). Sin embargo, si las dos variables punteros apuntan a dos variables anónimas distintas el resultado será FALSE, aunque el contenido de las dos variables anónimas sea idéntico. Así, el siguiente ejemplo

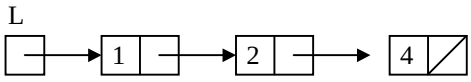
```
NEW(Ptr1);
NEW(Ptr2);
WITH Ptr1^ DO
  Real := 15.0;
  Imag := 20.0
END;
WITH Ptr2^ DO
  Real := 15.0;
  Imag := 20.0
END;
IF Ptr1 = Ptr2 THEN
  WrStr("Iguales")
ELSE
  WrStr("Distintas")
END
```

produce por pantalla la salida `"Distintas"`. Si lo que se desea es comprobar que los contenidos de las dos variables son iguales, se debe usar la derreferenciación `Ptr1^ = Ptr2^`, que será TRUE si el contenido de la variable anónima apuntada por `Ptr1` es igual al de la variable anónima apuntada por `Ptr2`, aunque dichas variables no sean las mismas.

12.5 Listas enlazadas

Lo que realmente hace de los punteros una herramienta potente es que pueden apuntar a variables que a su vez contienen punteros (típicamente registros en los cuales al menos uno de los campos es un puntero). Gracias a esta característica, podemos crear estructuras de datos dinámicas (el tamaño crece y decrece).

Una de estas estructuras es la *lista enlazada*. Se trata de una estructura que almacena una cantidad variable de datos, pero con la restricción de que todos tienen el mismo tipo (*tipo base*). La lista se representa mediante un puntero que apunta a un registro. Este registro contiene dos campos: el primero almacena el primer elemento de la lista(un dato del tipo base) y el segundo un puntero a otro registro (que corresponde al siguiente dato). De este modo, cada dato contiene la dirección del siguiente elemento. Cada uno de los registros que forman parte de la lista se llaman *nodos*. El campo puntero del último nodo de la lista enlazada contiene el valor `NIL`. El siguiente ejemplo muestra una lista de tres datos:



Cuando la lista está vacía (no contiene ningún dato), el valor del puntero inicial debe ser `NIL`:

El acceso a los elementos de una lista no es directo (como ocurría en los arrays). Para acceder a un elemento partimos del puntero inicial y hemos de ir pasando por todos los nodos anteriores a él. Además, solo es posible pasar de un nodo a su siguiente, pero no al previo.

Las declaraciones de tipo para la estructura lista (con tipo base INTEGER) son:

```
TYPE
  ELEMENTO   = INTEGER;
  PNODO      = POINTER TO NODO;
  NODO       = RECORD
                 Elem : ELEMENTO;
                 Sig  : PNODO
               END;
  LISTA      = PNODO;
```

Obsérvese que el tipo `NODO` aparece en la definición de `PNODO` antes de estar declarado. En *Modula-2*, solo se puede utilizar un tipo aún no definido en la definición de otro, cuando se está definiendo un tipo puntero. Por supuesto, el tipo no definido debe definirse posteriormente (como ocurre en el ejemplo).

La estructura lista es una estructura recursiva, ya que es un puntero a un registro de dos componentes, donde la segunda es a su vez una lista.

Junto a la definición del tipo para las listas, es necesario definir una serie de operaciones para su manejo. Podemos hacer esto mediante un módulo de biblioteca. El módulo de definición correspondiente puede ser el siguiente:

```
DEFINITION MODULE Lista;
TYPE
  ELEMENTO   = INTEGER;
  PNODO      = POINTER TO NODO;
  NODO       = RECORD
                 Elem : ELEMENTO;
                 Sig  : PNODO
               END;
  LISTA      = PNODO;
```

```
PROCEDURE Crear               () : LISTA;
PROCEDURE InsertarInicio      (VAR L : LISTA; Elem : ELEMENTO);
PROCEDURE InsertarFinal       (VAR L : LISTA; Elem : ELEMENTO);
PROCEDURE InsertarEnPosicion  (VAR L : LISTA; Elem : ELEMENTO;
                               Pos : CARDINAL);
PROCEDURE Eliminar            (VAR L : LISTA; Elem : ELEMENTO);
PROCEDURE EliminarEnPosicion  (VAR L : LISTA; Pos : CARDINAL);
PROCEDURE ElementoEnPosicion  (L : LISTA; Pos :CARDINAL) : ELEMENTO;
PROCEDURE Longitud            (L : LISTA) :CARDINAL;
PROCEDURE WrLista             (L : LISTA);
PROCEDURE Destruir            (VAR L : LISTA);

END Lista.
```

El significado de cada una de las operaciones es:

- `Crear`: devuelve una lista vacía (sin ningún elemento). Se usa para inicializar variables del tipo LISTA.
- `InsertarInicio`: inserta un nuevo elemento en la primera posición de una lista.
- `InsertarFinal`: inserta un nuevo elemento en la última posición de una lista.
- `InsertarEnPosición`: inserta un nuevo elemento dentro de la lista en la posición indicada por el tercer parámetro. Los nodos de la lista comienzan a numerarse por uno.
- `Eliminar`: elimina la primera aparición del elemento `Elem` de la lista. Si el elemento está repetido varias veces dentro de la lista solo se elimina la primera (más a la izquierda) ocurrencia.
- `EliminarEnPosicion`: elimina de una lista el elemento que ocupa la posición dada por el segundo argumento.
- `ElementoEnPosicion`: devuelve el valor del elemento que ocupa la posición dada por el segundo argumento, pero no lo elimina de la lista.
- `Longitud`: devuelve el número de elementos en una lista.
- `WrLista`: escribe el contenido de una lista por pantalla.
- `Destruir`: elimina todos los elementos de una lista, liberando la memoria que ocupaban.

Obsérvese que en las operaciones donde la lista se ve modificada es necesario pasar el correspondiente argumento por referencia (**VAR**).

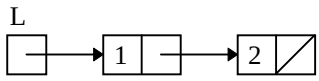
La implementación de cada operación aparecerá en el correspondiente módulo de implementación. Es necesario importar `ALLOCATE` y `DEALLOCATE` ya que usaremos **`NEW`** y **`DISPOSE`**. También definimos un par de constantes que representan la lista vacía y el valor del puntero final de una lista (ambos son `NIL`):

```
IMPLEMENTATION MODULE Lista;

FROM Storage   IMPORT ALLOCATE, DEALLOCATE;
FROM IO        IMPORT WrStr, WrInt, WrChar;

CONST
  LISTAVACIA= NIL;
  FINLISTA  = NIL;

...
END Lista.
```

La operación `Crear` es inmediata. Basta con devolver `NIL` ya que por convenio las listas vacías se representan con este valor.

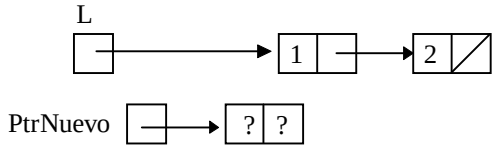
```
PROCEDURE Crear () : LISTA;
BEGIN
  RETURN LISTAVACIA
END Crear;
```

Supongamos que queremos insertar el elemento 3 al principio de la lista



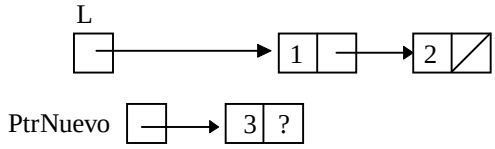
Para ello hemos de seguir los siguientes pasos:

1) Creamos un nuevo nodo (lo referenciaremos mediante la variable puntero `PtrNuevo`):



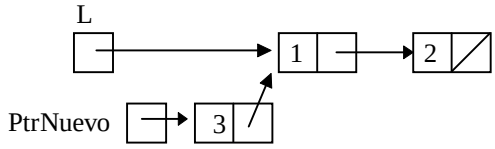

```
NEW(PtrNuevo);
```

2) Copiamos el valor a insertar en el campo `Elem`



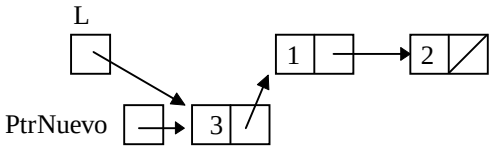

```
PtrNuevo^.Elem := 3;
```

3) Enlazamos el nuevo nodo con el principio de la lista




```
PtrNuevo^.Sig := L;
```

4) Por último, la lista comienza ahora por el nuevo nodo:



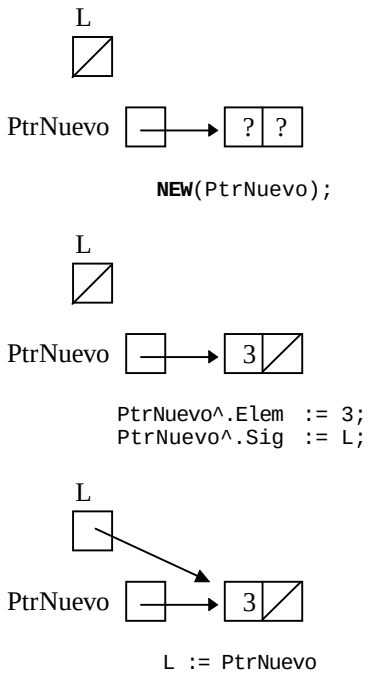

```
L := PtrNuevo;
```

Podemos unir todo lo anterior en el siguiente subprograma:

```
PROCEDURE InsertarInicio (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrNuevo : PNODO;
BEGIN
  NEW(PtrNuevo);
  PtrNuevo^.Elem := Elem;
  PtrNuevo^.Sig  := L;
  L              := PtrNuevo
END InsertarInicio;
```

La variable `PtrNuevo` será destruida al salir del subprograma, ya que es una variable local a éste.

Para que el subprograma anterior pueda ser considerado correcto es necesario comprobar que también funciona en el caso de que la lista inicial sea vacía:



El orden en que se realizan las operaciones debe ser exactamente el indicado en el subprograma, ya que si invirtiésemos el orden de las dos últimas instrucciones, perderíamos la referencia al antiguo primer nodo de la lista. A la hora de trabajar con listas enlazadas es muy importante no dejar nunca ningún nodo inaccesible. Para evitar esto se usarán variables punteros auxiliares siempre que sea necesario.

También es importante observar que el parámetro `L` pasa por referencia ya que su valor se ve modificado dentro del subprograma.

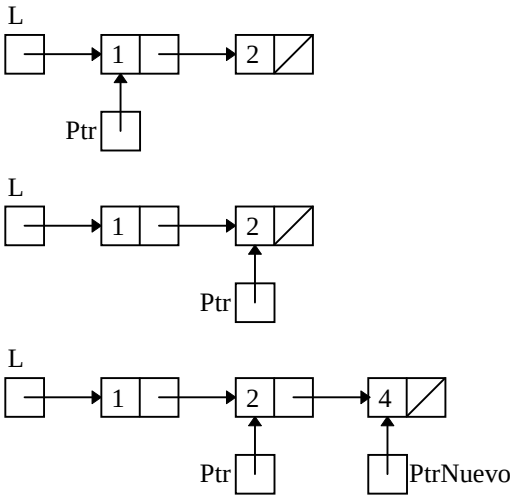
Para insertar un elemento nuevo al final de una lista creamos un nuevo nodo, copiamos el valor del elemento y ponemos el puntero del campo `Sig` a `FINLISTA` (ya que este nodo ocupará la última posición de la lista). A continuación consideramos dos casos:

- Si la lista está vacía enlazamos `L` con este nodo.
- En otro caso hay que recorrer la lista hasta encontrar el último nodo. Esto se suele hacer utilizando una variable de tipo puntero (`Ptr` en nuestro caso) que se desplaza hasta alcanzar un nodo cuyo campo `Sig` sea `FINLISTA`. Una vez localizado el último nodo lo enlazamos con el nuevo.

```
PROCEDURE InsertarFinal (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrNuevo, Ptr : PNODO;
BEGIN
  NEW(PtrNuevo);
  PtrNuevo^.Elem := Elem;
  PtrNuevo^.Sig  := FINLISTA;
  IF L = LISTAVACIA THEN
     L := PtrNuevo
  ELSE
     Ptr := L;
     WHILE Ptr^.Sig <> FINLISTA DO
       Ptr := Ptr^.Sig
     END;
```

```
    Ptr^.Sig := PtrNuevo
  END
END InsertarFinal;
```

El siguiente ejemplo muestras los pasos seguidos para insertar un elemento de valor 4 en una lista con dos elementos (1 y 2):

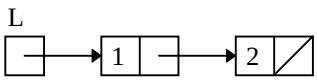


Este algoritmo también se puede describir más elegantemente de modo recursivo. Para insertar un elemento en una lista vacía basta con crear un nodo y enlazarlo como en `InsertarInicio`. En otro caso hacemos una llamada recursiva, pero con el segmento de lista que comienza por el segundo elemento:

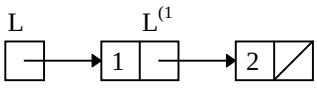
```
PROCEDURE InsertarFinal (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrNuevo : PNODO;
BEGIN
  IF L = LISTAVACIA THEN
     NEW(PtrNuevo);
     PtrNuevo^.Elem := Elem;
     PtrNuevo^.Sig  := FINLISTA;
     L              := PtrNuevo
  ELSE
     InsertarFinal(L^.Sig, Elem)
  END
END InsertarFinal;
```

Por ejemplo, si queremos insertar un 4 en la lista mostrada se producen dos llamadas recursivas. El valor de L en la primera llamada recursiva es $L^{(1}$ y en la segunda $L^{(2}$. En la última llamada, como $L^{(2}$ es `LISTAVACIA`, se crea el nuevo nodo y se enlaza.

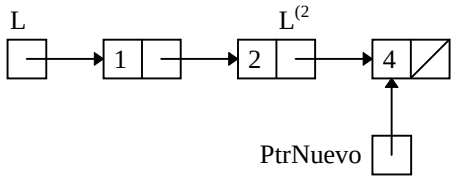
1) Llamada inicial: `InsertarFinal (L, 4)`



2) Primera Llamada recursiva: `InsertarFinal (L(1, 4)`



3) Segunda Llamada recursiva: `InsertarFinal (L(2, 4)`



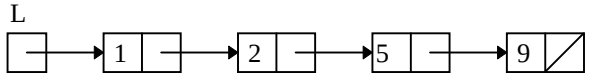
Las operaciones relativas a listas enlazadas suelen ser fáciles de implementar de modo recursivo ya que la estructura es recursiva. En las operaciones recursivas suele ser necesario considerar siempre dos casos: la lista está vacía (caso base) o no. La llamada recursiva suele aproximarse al caso base tomando la lista que se obtiene saltando el nodo inicial (`L^.Sig`).

Para insertar un nuevo elemento en una posición dada de una lista se contemplan tres casos:

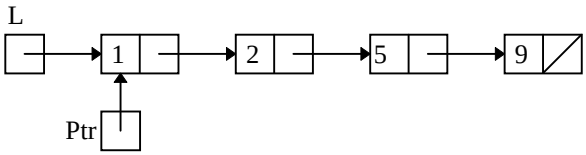
- Si la posición es cero, se produce un error ya que la primera posición válida es uno que indica insertar al comienzo de la lista.
- Si la posición es uno llamamos al subprograma `InsertarInicio`.
- En otro caso utilizamos `Ptr` para localizar el nodo previo a donde se ha de insertar. Para ello se parte del comienzo de la lista y se avanza `Pos`-2 veces (cuidando no alcanzar el final de la lista). Si al intentar localizar el nodo previo se alcanzó el final de la lista es que la posición indicada para insertar no es válida (es mayor al número de elementos en la lista). En otro caso, creamos un nuevo nodo y lo enlazamos tras el nodo localizado.

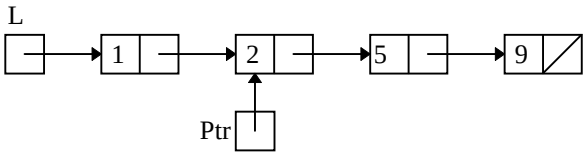
```
PROCEDURE InsertarEnPosicion (VAR L : LISTA; Elem : ELEMENTO;
                                  Pos : CARDINAL);
VAR PtrNuevo, Ptr : PNODO;
    Avanzados     : CARDINAL;
BEGIN
  IF Pos = 0 THEN
     WrStr("Error: InsertarEnPosicion fuera de rango");
     HALT
  ELSIF Pos = 1 THEN
     InsertarInicio(L, Elem)
  ELSE  (* Pos >= 2 *)
     Avanzados := 0;
     Ptr       := L;
     WHILE (Ptr<>FINLISTA) AND (Avanzados < Pos-2) DO
        Ptr := Ptr^.Sig;
        INC(Avanzados)
     END;

     IF Ptr = FINLISTA THEN
       WrStr("Error: InsertarEnPosicion fuera de rango");
       HALT
     ELSE
       NEW(PtrNuevo);
       PtrNuevo^.Elem  := Elem;
       PtrNuevo^.Sig   := Ptr^.Sig;
       Ptr^.Sig        := PtrNuevo
     END
  END
END InsertarEnPosicion;
```

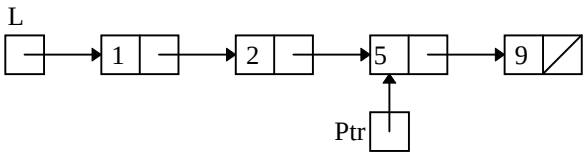
Veamos el algoritmo en funcionamiento si queremos insertar un elemento de valor 7 en la posición cuarta de la siguiente lista:



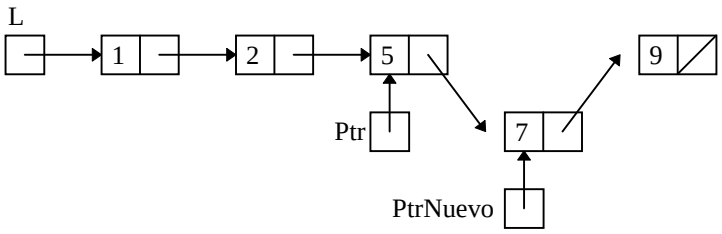
1) Avanzar dos nodos (4-2 = 2):







2) Crear y enlazar el nuevo nodo:



Obsérvese el orden seguido al enlazar el nuevo nodo: primero se enlaza el nuevo nodo con el siguiente nodo al que apunta `Ptr` y por último se enlaza el nodo apuntado por `Ptr` con el nuevo. Este orden es importante, ya que si lo hiciésemos en orden inverso perderíamos la referencia al último nodo.

El algoritmo `InsertarEnPosicion` escrito de modo recursivo es:

```
PROCEDURE InsertarEnPosicion (VAR L : LISTA; Elem : ELEMENTO;
                              Pos : CARDINAL);
  PROCEDURE InsertarAux (VAR L : LISTA; Elem : ELEMENTO;
                         Pos : CARDINAL);
  VAR PtrNuevo : PNODO;
  BEGIN
     IF Pos = 1 THEN
       NEW(PtrNuevo);   (* Se puede usar InsertarInicio(L,Elem) *)
       PtrNuevo^.Elem := Elem;
       PtrNuevo^.Sig  := L;
       L              := PtrNuevo
     ELSIF L = LISTAVACIA THEN
       WrStr("Error: InsertarEnPosicion fuera de rango");
       HALT
     ELSE
       InsertarAux(L^.Sig, Elem, Pos-1)
     END
  END InsertarAux;
BEGIN
  IF Pos = 0 THEN
    WrStr("Error: InsertarEnPosicion fuera de rango");
    HALT
  ELSE
    InsertarAux (L, Elem, Pos)
  END
END InsertarEnPosicion;
```

Describimos ahora un algoritmo para eliminar la primera ocurrencia de un elemento dado de una lista enlazada:

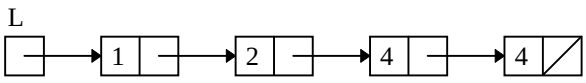
```
PROCEDURE Eliminar (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrBorrar, PtrPrev : PNODO;
BEGIN
  PtrBorrar := L;
  PtrPrev   := NIL;
  WHILE (PtrBorrar<>FINLISTA) AND (PtrBorrar^.Elem<>Elem) DO
       PtrPrev   := PtrBorrar;
       PtrBorrar := PtrBorrar^.Sig
  END;
  IF PtrBorrar = FINLISTA THEN
     WrStr("Error: Eliminar, elemento no encontrado");
     HALT
  ELSE
     IF  PtrPrev = NIL THEN
       L := L^.Sig
     ELSE
       PtrPrev^.Sig := PtrBorrar^.Sig;
     END;
   DISPOSE(PtrBorrar)
  END
END Eliminar;
```

Para ello localizaremos con `PtrBorrar` el primer nodo cuyo elemento coincida con el valor que se quiere eliminar. Como *Modula-2* utiliza *evaluación en cortocircuito* de expresiones booleanas, si `PtrBorrar` alcanza en algún momento el final de la lista (porque el elemento buscado no se encuentre en ella) la segunda parte de la expresión AND no llega a evaluarse, y gracias a esto, en ese caso no se llega a derreferenciar `PtrBorrar` (lo cual sería un error ya que el valor de esta variable en ese punto sería `NIL`).

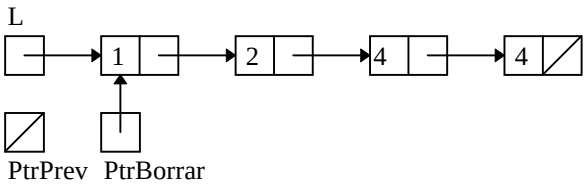
Se utiliza la técnica del puntero retrasado. `PtrPrev` siempre apuntará al nodo previo a `PtrBorrar` o será `NIL` si `PtrBorrar` está situado sobre el primer nodo de la lista. Es necesario comprobar en todo momento que no hemos alcanzado el final de la lista. Tras el bucle, si hemos localizado el nodo a eliminar (`PtrBorrar` <> `FINLISTA`) contemplamos dos casos:

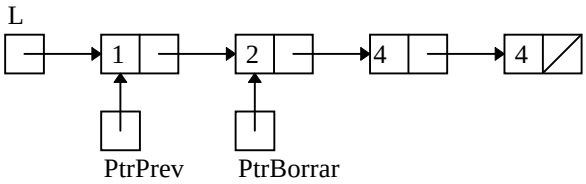
1) El nodo a eliminar es el primero de la lista. En este caso `PtrPrev` valdrá `NIL`. Para eliminar el nodo avanzamos `L` y como `PtrBorrar` apunta al primer nodo llamamos a **`DISPOSE`**`(PtrBorrar)`para liberar la memoria que ocupaba.
2) El nodo a eliminar no es el primero de la lista. Enlazamos el nodo previo al eliminado con el siguiente al eliminado. De nuevo `PtrBorrar` apunta al nodo a eliminar y devolvemos su memoria.

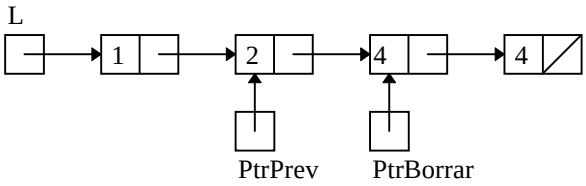
Veamos como funciona el algoritmo si queremos eliminar la primera ocurrencia del valor 4 de la siguiente lista:



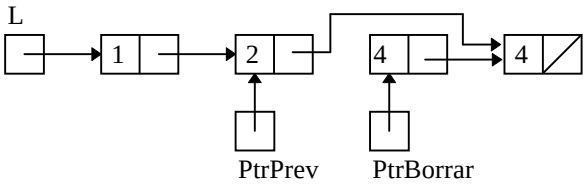
1) Localizar el nodo a borrar



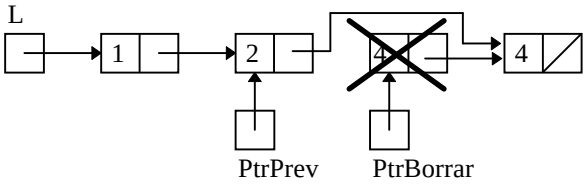




2) Desenlazarlo de la lista



3) Liberar memoria



El alumno debería verificar que el algoritmo descrito también funciona si el nodo que se elimina es el primero de la lista o el último.

De modo recursivo tenemos:

```
PROCEDURE Eliminar (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrBorrar : PNODO;
BEGIN
  IF L = LISTAVACIA THEN
    WrStr("Error: Eliminar, elemento no encontrado");
    HALT
  ELSIF L^.Elem = Elem THEN
    PtrBorrar   := L;
    L             := L^.Sig;
    DISPOSE(PtrBorrar);
  ELSE
    Eliminar (L^.Sig, Elem)
  END
END Eliminar;
```

El siguiente procedimiento elimina el nodo que ocupa la posición indicada de una lista. El algoritmo es muy similar al anterior salvo que en vez de avanzar `PtrBorrar` hasta encontrar el elemento se avanza `Pos-1` veces:

```
PROCEDURE EliminarEnPosicion (VAR L : LISTA; Pos : CARDINAL);
VAR  PtrBorrar, PtrPrev  : PNODO;
     Avanzados           : CARDINAL;
BEGIN
  IF Pos = 0 THEN
    WrStr("Error: EliminarEnPosicion fuera de rango");
    HALT
```

```
  ELSE
     PtrBorrar := L;
     PtrPrev   := NIL;
     Avanzados := 0;
     WHILE (Avanzados<Pos-1) AND (PtrBorrar<>FINLISTA) DO
      PtrPrev    := PtrBorrar;
      PtrBorrar  := PtrBorrar^.Sig;
      INC(Avanzados)
     END;

     IF PtrBorrar = FINLISTA THEN
       WrStr("Error: EliminarEnPosicion fuera de rango");
       HALT
     ELSE
       IF PtrPrev = NIL THEN
         L := L^.Sig
       ELSE
         PtrPrev^.Sig := PtrBorrar^.Sig;
       END;
       DISPOSE(PtrBorrar)
     END
  END
END EliminarEnPosicion;
```

El algoritmo recursivo es:

```
PROCEDURE EliminarEnPosicion (VAR L : LISTA; Pos : CARDINAL);
  PROCEDURE EliminarAux(VAR L : LISTA; Pos : CARDINAL);
  VAR PtrBorrar : PNODO;
  BEGIN
    IF L = LISTAVACIA THEN
      WrStr("Error: EliminarEnPosicion fuera de rango");
      HALT
    ELSIF Pos = 1 THEN
      PtrBorrar  := L;
      L             := L^.Sig;
      DISPOSE(PtrBorrar);
    ELSE
      EliminarAux(L^.Sig, Pos-1)
    END
  END EliminarAux;
BEGIN
  IF Pos = 0 THEN
    WrStr("Error: EliminarEnPosicion fuera de rango");
    HALT
  ELSE
    EliminarAux(L, Pos)
  END
END EliminarEnPosicion;
```

Para devolver el valor del elemento que se encuentra en una posición dada dentro de la lista, localizamos el nodo correspondiente avanzando `Ptr` Pos-1 posiciones en la lista y devolvemos el valor almacenado en el nodo apuntado por `Ptr`. Es importante comprobar si hemos alcanzado el final de la lista:

```
PROCEDURE ElementoEnPosicion (L:LISTA; Pos:CARDINAL) : ELEMENTO;
VAR  Ptr       : PNODO;
     Avanzados : CARDINAL;
     Resultado : ELEMENTO;
BEGIN
  IF Pos = 0 THEN
    WrStr("Error: ElementoEnPosicion fuera de rango");
    HALT
  ELSE
```

```
    Ptr         := L;
    Avanzados   := 0;
    WHILE (Ptr <> FINLISTA) AND (Avanzados < Pos-1) DO
       Ptr := Ptr^.Sig;
       INC (Avanzados)
    END;

    IF Ptr = FINLISTA THEN
      WrStr("Error: ElementoEnPosicion fuera de rango");
      HALT
    ELSE
      Resultado := Ptr^.Elem
    END;

    RETURN Resultado
  END
END ElementoEnPosicion;
```

Recursivamente:

```
PROCEDURE ElementoEnPosicion (L:LISTA; Pos:CARDINAL) : ELEMENTO;
  PROCEDURE ElementoAux(VAR L:LISTA; Pos:CARDINAL) : ELEMENTO;
  BEGIN
    IF L = LISTAVACIA THEN
      WrStr("Error: ElementoEnPosicion fuera de rango");
      HALT
    ELSIF Pos = 1 THEN
      RETURN L^.Elem
    ELSE
      RETURN ElementoAux(L^.Sig, Pos-1)
    END
  END ElementoAux;
BEGIN
  IF Pos = 0 THEN
    WrStr("Error: ElementoEnPosicion fuera de rango");
    HALT
  ELSE
    RETURN ElementoAux(L, Pos)
  END
END ElementoEnPosicion;
```

Para calcular la longitud de una lista recorremos todos sus nodos con la variable `Ptr` hasta encontrar el fin de la lista. Utilizamos un contador que inicializamos a cero y que incrementamos cada vez que avanzamos un nodo:

```
PROCEDURE Longitud (L : LISTA) : CARDINAL;
VAR  Ptr          : PNODO;
     LaLongitud   : CARDINAL;
BEGIN
  LaLongitud   := 0;
  Ptr          := L;
  WHILE Ptr <> FINLISTA DO
     Ptr := Ptr^.Sig;
     INC(LaLongitud)
  END;
  RETURN LaLongitud
END Longitud;
```

De modo recursivo, basta observar que la longitud de la lista vacía es cero y en otro caso devolvemos uno más la longitud de la lista que empieza en el nodo siguiente:

```
PROCEDURE Longitud (L : LISTA) : CARDINAL;
BEGIN
  IF L = LISTAVACIA THEN
```

```
    RETURN 0
  ELSE
    RETURN 1 + Longitud(L^.Sig)
  END
END Longitud;
```

Escribiremos una lista entre corchetes y separando los elementos por comas. Es necesario comprobar para cada elemento si el último de la lista, ya que en dicho caso no se escribe la coma correspondiente:

```
PROCEDURE WrLista (L : LISTA);
VAR Ptr : PNODO;
BEGIN
  WrChar('[');
  Ptr := L;
  WHILE Ptr <> FINLISTA DO
    WrInt(Ptr^.Elem, 0);
    IF Ptr^.Sig <> FINLISTA THEN
      WrChar(',')
    END;
    Ptr := Ptr^.Sig
  END;
  WrChar(']')
END WrLista;
```

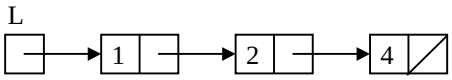
La solución recursiva también es fácil, aunque necesitamos un subprograma auxiliar:

```
PROCEDURE WrLista (L : LISTA);
  PROCEDURE WrAux(L : LISTA);
  BEGIN
    IF L <> LISTAVACIA THEN
      WrInt(L^.Elem, 0);
      IF L^.Sig <> FINLISTA THEN
        WrChar(',')
      END;
      WrAux(L^.Sig)
    END
  END WrAux;
BEGIN
  WrChar('[');
  WrAux(L);
  WrChar(']')
END WrLista;
```

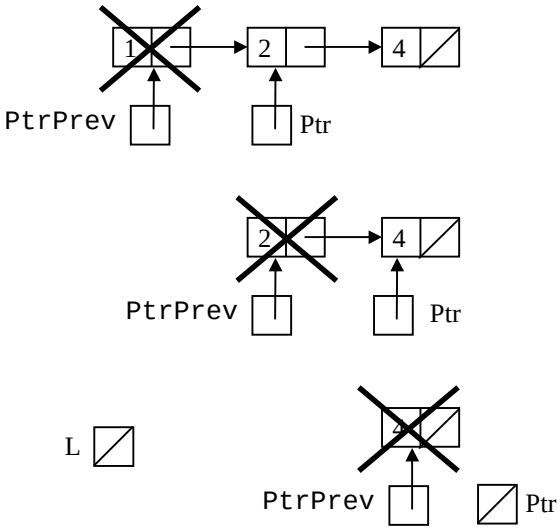
Por último necesitamos una operación que deje la lista vacía liberando toda la memoria ocupada por los nodos. En este caso necesitamos recorrer la lista utilizando la técnica del puntero retrasado. Los elementos apuntados por el puntero retrasado son los que se destruyen:

```
PROCEDURE Destruir (VAR L : LISTA);
VAR PtrPrev, Ptr : PNODO;
BEGIN
   Ptr := L;
  PtrPrev := NIL;
  WHILE Ptr <> FINLISTA DO
     PtrPrev   := Ptr;
     Ptr       := Ptr^.Sig;
     DISPOSE(PtrPrev)
  END;
   L := LISTAVACIA
END Destruir;
```

Gráficamente, si destruimos la lista



los pasos seguidos son

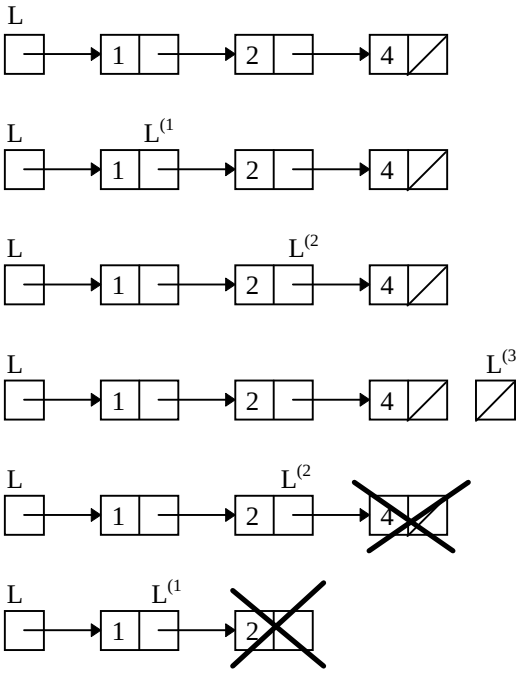


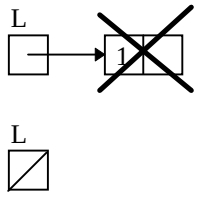
Nótese que `L` queda a `NIL` finalmente (lo cual es adecuado, ya que la lista está vacía).

El algoritmo recursivo correspondiente es más breve:

```
PROCEDURE Destruir (VAR L : LISTA);
BEGIN
  IF L <> LISTAVACIA THEN
    Destruir(L^.Sig);
    DISPOSE(L)
  END
END Destruir;
```

Los pasos seguidos por el algoritmo recursivo son los siguientes:





En cada momento, el número en el superíndice de `L` indica el anidamiento de la llamada recursiva que se está ejecutando.

Un programa que usa la biblioteca definida para leer una cantidad de números no conocida a priori y mostrarlos es:

```
MODULE Demo;
FROM Lista  IMPORT Crear, InsertarFinal, WrLista, Destruir,
                   LISTA, ELEMENTO;
FROM IO     IMPORT RdInt, RdCard, RdLn, WrStr;

VAR
  Lista     : LISTA;
  Numero    : ELEMENTO;
  Total, i  : CARDINAL;

BEGIN
  WrStr("Dame el total de números a introducir: ");
  Total := RdCard(); RdLn;

  Lista := Crear();

  FOR i := 1 TO Total DO
    WrStr("Dame un número: ");
    Numero := RdInt(); RdLn;
    InsertarFinal(Lista, Numero)
  END;

  WrStr("Los números introducidos fueron: ");
  WrLista(Lista);

  Destruir(Lista)

END Demo.
```

Es importante liberar la memoria que ocupa la lista (en este caso mediante la operación `Destruir`) cuando ésta ya no es necesaria.

12.5.1 Listas ordenadas

Una lista ordenada (en orden ascendente, aunque se pueden definir de modo similar en orden descendente) es una lista en la cual el valor de un nodo es siempre menor o igual al nodo que le sigue.

Las operaciones de búsqueda de elementos dentro de la lista pueden ser implementadas de un modo más eficiente que si se usan listas no ordenadas. A la hora de localizar un elemento no hay que recorrer toda la lista para asegurarnos que no existe. Basta con llegar a un nodo cuyo valor sea mayor al buscado. Esta mejora puede ser bastante considerable, especialmente si la lista tratada tiene una longitud grande.

Por otro lado, una lista ordenada también se usa si se desea obtener un listado ordenado de los datos almacenados.

En el caso de las listas ordenadas, a la hora de insertar un elemento, no se especifica la posición que éste debe ocupar. Solo existe una operación `InsertarOrdenada` que toma un nuevo dato y lo coloca en su posición adecuada dentro de la lista, de modo que la lista resultante sigue estando ordenada. Se mantiene la operación para eliminar por posición dentro de la lista y la de eliminación de la primera ocurrencia de un dato dado.

El siguiente módulo de definición corresponde a una biblioteca de listas ordenadas. De las operaciones que aparecen en el módulo, solo `InsertarOrdenado` y `Eliminar` son distintas a las presentadas en el apartado anterior.

```
DEFINITION MODULE ListaOrd;
TYPE
  ELEMENTO   =  INTEGER;
  PNODO      =  POINTER TO NODO;
  NODO       =  RECORD
                  Elem : ELEMENTO;
                  Sig  : PNODO
                END;
  LISTA      =  PNODO;

PROCEDURE Crear               () : LISTA;
PROCEDURE InsertarOrdenada    (VAR L : LISTA; Elem : ELEMENTO);
PROCEDURE Eliminar            (VAR L : LISTA; Elem : ELEMENTO);
PROCEDURE EliminarEnPosicion  (VAR L : LISTA; Pos : CARDINAL);
PROCEDURE ElementoEnPosicion  (L : LISTA; Pos :CARDINAL) : ELEMENTO;
PROCEDURE Longitud            (L : LISTA) :CARDINAL;
PROCEDURE WrLista             (L : LISTA);
PROCEDURE Destruir            (VAR L : LISTA);

END ListaOrd.
```

Para insertar un nuevo dato hemos de buscar su posición adecuada dentro de la lista. Para ello buscamos con `PtrSig` el primer nodo cuyo contenido sea mayor o igual al que estamos insertando. Habrá que insertar el nuevo nodo justo delante del nodo al cual apunte `PtrSig`. Para ello utilizamos la técnica del puntero retrasado, de modo que cuando `PtrSig` encuentra el nodo siguiente, `PtrPrev` apunta al nodo anterior y solo hay que enlazar el nuevo nodo entre estos. Previamente es necesario comprobar si `PtrSig` no llegó a avanzar ni un solo nodo (en ese caso `PtrPrev` = `NIL`) ya que en ese caso el nuevo nodo iría colocado al principio de la lista:

```
PROCEDURE InsertarOrdenada (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrNuevo, PtrSig, PtrPrev : PNODO;
BEGIN
  NEW(PtrNuevo);
  PtrNuevo^.Elem := Elem;

  PtrSig    := L;
  PtrPrev   := NIL;
  WHILE (PtrSig<>FINLISTA) AND (PtrSig^.Elem<Elem) DO
       PtrPrev   := PtrSig;
       PtrSig    := PtrSig^.Sig
  END;

  PtrNuevo^.Sig := PtrSig;
  IF PtrPrev = NIL THEN
    L := PtrNuevo
  ELSE
    PtrPrev^.Sig := PtrNuevo;
  END
END InsertarOrdenada;
```

El alumno debería comprobar que el algoritmo también funciona si el nuevo nodo va a parar justo al final de la lista (o sea, es mayor que cualquiera de los nodos en la lista).

La función que elimina un dato es muy similar a la de las listas enlazadas no ordenadas, pero en este caso basta con alcanzar un nodo con valor mayor al buscado para concluir que el nodo no está en la lista:

```
PROCEDURE Eliminar (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrBorrar, PtrPrev : PNODO;
BEGIN
  PtrBorrar := L;
  PtrPrev   := NIL;
  WHILE (PtrBorrar<>FINLISTA) AND (PtrBorrar^.Elem<Elem) DO
     PtrPrev     := PtrBorrar;
     PtrBorrar := PtrBorrar^.Sig
  END;

  IF (PtrBorrar=FINLISTA) OR (PtrBorrar^.Elem<>Elem) THEN
    WrStr("Error: Eliminar elemento no encontrado");
    HALT
  ELSE
     IF  PtrPrev = NIL THEN
       L := L^.Sig
     ELSE
       PtrPrev^.Sig := PtrBorrar^.Sig;
     END;
     DISPOSE(PtrBorrar)
  END
END Eliminar;
```

De modo recursivo:

```
PROCEDURE InsertarOrdenada (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrNuevo : PNODO;
BEGIN
  IF (L = LISTAVACIA) OR (Elem <= L^.Elem) THEN
    (* Crear el nodo nuevo *)
    NEW(PtrNuevo);
    PtrNuevo^.Elem := Elem;

    (* Enlazarlo *)
    PtrNuevo^.Sig := L;
    L             := PtrNuevo
  ELSE
    InsertarOrdenada(L^.Sig, Elem)
  END
END InsertarOrdenada;


(* Elimina el primer elemento en la lista que coincide con Elem *)

PROCEDURE Eliminar (VAR L : LISTA; Elem : ELEMENTO);
VAR PtrBorrar : PNODO;
BEGIN
  IF (L = LISTAVACIA) OR (L^.Elem > Elem) THEN
    WrStr("Error: Eliminar, elemento no encontrado");
    HALT
  ELSIF L^.Elem = Elem THEN
    (* Enlazar con el siguiente y liberar memoria *)
    PtrBorrar   := L;
    L         := L^.Sig;
    DISPOSE(PtrBorrar);
  ELSE
    Eliminar (L^.Sig, Elem)
  END
END Eliminar;
```

Relación de Problemas (Tema 12)

1. Responder verdadero o falso teniendo en cuenta las siguientes declaraciones:

```
TYPE
  PUNTERO   = POINTER TO NODO;
  NODO      = RECORD
                 Info      : INTEGER;
                 Siguiente : PUNTERO
              END;
VAR
  Ptr    : PUNTERO;
  UnNodo : NODO;
```

a) El espacio que ocupa la variable `Ptr` se asigna dinámicamente en tiempo de ejecución

b) El espacio que ocupa `Ptr^` se asigna dinámicamente en tiempo de ejecución.

c) `Ptr^` está inicialmente indefinido.

d) Después de la sentencia **`NEW`**`(Ptr)`, `Ptr^.Siguiente` vale **`NIL`**.

e) `Ptr` ocupa la misma cantidad de espacio en memoria que un registro del tipo `NODO`.

f) La declaración de `UnNodo` es sintácticamente incorrecta porque los registros tipo `NODO` sólo pueden ser asignados dinámicamente.

2. Dadas las declaraciones

```
TYPE
  CADENA =  ARRAY[1..20] OF CHAR;
  ELEMENTO  =  RECORD
                  Nombre : CADENA;
                  ID     : INTEGER
               END;
  PUNTERO   =  POINTER TO NODO;
  NODO      =  RECORD
                  Datos              : ELEMENTO;
                  Anterior,Siguiente : PUNTERO
               END;
VAR
  Lista, Ptr : PUNTERO;
```

identificar el tipo de cada una de las siguientes expresiones como un puntero, un registro, un carácter, una cadena de caracteres o un entero.

a) `Lista^.Siguiente`

b) `Lista^.Siguiente^`

c) `Lista^.Datos.Nombre`

d) `Ptr^.Anterior^.Datos.ID`

e) `Ptr^.Anterior`

f) `Ptr^.Datos.Nombre[1]`

3. Dadas dos listas enlazadas ordenadas escribir un procedimiento o función que genere una nueva lista mezcla ordenada de ambas. Las dos listas originales deben quedar intactas (los nodos que constituyen la lista mezcla deben crearse). Escribir una versión iterativa y otra recursiva del subprograma.

4. Diseña una biblioteca (módulo de definición e implementación) donde se defina un nuevo tipo `CONJUNTO`. Las variables de este tipo representarán conjuntos de números reales. Para ello cada conjunto debe representarse como una lista enlazada. Incorpora al módulo las operaciones normales predefinidas sobre conjuntos (`Crear`, `Incluir`, `Excluir`, `Pertenece`, `EsVacio`, `Cardinal`, `Union`, `Interseccion` y `Diferencia`).

NOTA: En la lista que represente un conjunto no deben aparecer elementos repetidos.

5. Dadas dos listas enlazadas que almacenan números enteros escribir un programa que genere todas las combinaciones posibles de parejas de números, uno de cada lista, cuya suma es menor o igual que un número `N` introducido por teclado.

6. Un polinomio puede ser representado por una lista enlazada en la que cada nodo contiene un coeficiente (de tipo `REAL`), el valor del exponente (un valor `CARDINAL`) y un puntero hacia el siguiente elemento. Escribir un módulo de biblioteca en el que se defina el tipo correspondiente y subprogramas que permitan:

a) Crear polinomios.

b) Añadir términos al polinomio.

c) Leer polinomios de teclado.

d) Escribir polinomios en pantalla.

e) Sumar dos polinomios.

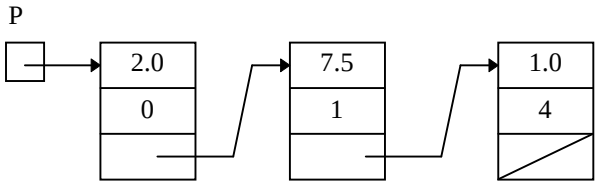
f) Restar dos polinomios.

g) Calcular el producto de dos polinomios.

h) Calcular la derivada de un polinomio dado.

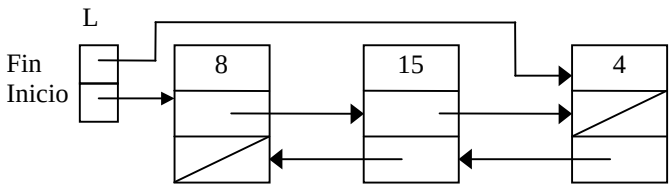
i) Evaluar el polinomio para un valor de la variable independiente también de tipo REAL

NOTA: Las listas enlazadas que representan los polinomios estarán ordenadas en forma ascendente por el grado de los nodos. Por ejemplo, el polinomio $x^4 + 7.5x + 2$ estaría representado por la lista:



Las operaciones de suma, resta, producto y derivada deben generar listas nuevas (los nodos que las forman deben crearse)

7. Las listas doblemente enlazadas son listas en las cuales cada nodo consta de un campo para almacenar un dato y dos campos de enlace o punteros que apuntan respectivamente al nodo anterior y posterior al actual. Estas listas pueden recorrerse en dos sentidos: hacia delante y hacia atrás. Las listas se representan mediante un registro de dos punteros (`Inicio` y `Fin`) que apunten hacia el primer y el último nodo respectivamente.



Escribir un módulo de biblioteca para implementar esta estructura y operaciones para su manejo.

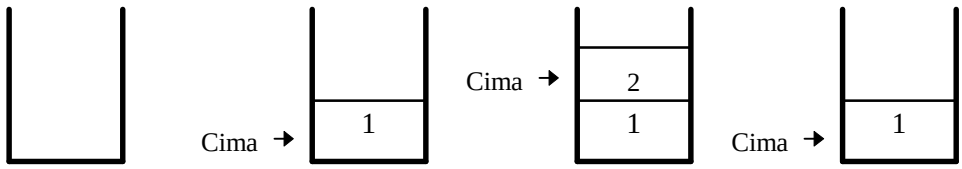
8. Una pila es un tipo especial de lista en la cual la inserción y eliminación de elementos se realiza sólo por un extremo que se denomina *cima* o *tope*. Los elementos sólo pueden eliminarse en orden inverso al que se insertan. El último elemento que se insertó en la pila es el primero que se puede sacar. A este tipo de estructuras se las llama *LIFO* (siglas de la expresión inglesa *Last-In, First-Out*).

Las operaciones asociadas a una pila son

1) `Crear`: devuelve una pila vacía.
2) `Insertar`: añade un elemento a la pila
3) `Sacar`: extrae el último elemento insertado en la pila. Es un error extraer de una pila vacía.
4) `Cima`: devuelve el valor del último elemento insertado pero no lo extrae. Es un error obtener la cima de una pila vacía.
5) `EsVacia`: devuelve `TRUE` si la pila está vacía (no contiene ningún elemento).

Un ejemplo de uso de una pila de enteros es el siguiente:

```
P := Crear();      Insertar(P,1);      Insertar(P,2);      Extraer(P);
```



Escribir un módulo de biblioteca donde se defina el tipo pila de valores enteros y las operaciones asociadas.

NOTA: Una pila se representará mediante una lista enlazada. La operación de inserción en la pila consiste en insertar al inicio de la lista, mientras que la de extracción en eliminar el primer elemento de la lista.

9. Escribe un subprograma que tome como argumento una lista simplemente enlazada y escriba su contenido por pantalla en orden inverso.

10. Escribe un subprograma `Filtro` que tome como argumento una lista de valores enteros y una función que toma un entero y devuelve un valor booleano. Esta función debe devolver una nueva lista consistente en los argumentos de la lista original para los que la función se evalúa a `TRUE`. Los nodos de la lista resultante deben crearse. Escribe un programa que, utilizando esta función, lea una serie de valores enteros, los almacene en una lista, calcule una nueva lista donde queden solo los pares y por último escriba esta lista por pantalla.

11. Dadas las definiciones de datos vistas en el tema para una lista enlazada de valores enteros, escribe procedimientos o funciones para:

1) Duplicar una lista enlazada (crear otra lista enlazada con los mismos elementos

que el argumento).

2) Borrar el nodo que contiene el máximo valor de una lista enlazada.

3) Intercambiar el valor *n*-ésimo con el *m*-ésimo de la lista.

4) Dada la siguiente cabecera:

```
PROCEDURE Busca (L:LISTA; e:ELEMENTO): PNODO;
```

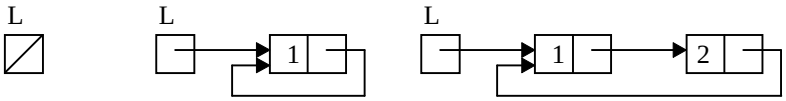
devuelva el puntero al primer nodo de `L` cuyo valor es `e` o `NIL` si no existe ningún nodo con dicho valor.

5) Dada la siguiente cabecera:

```
PROCEDURE InsBusca(VAR L:LISTA; e:ELEMENTO):PNODO;
```

añada `e` a la lista `L`, si no está en ella, y siempre devuelva un puntero al nodo que contiene a `e`

12. Una lista enlazada circular es una lista enlazada donde el puntero del último nodo no es NIL, sino que apunta al primer nodo:



Listas circulares de cero, uno y dos elementos.

Escribe un módulo de biblioteca donde definas el tipo para listas circulares y operaciones para insertar al principio, insertar al final y escribir una lista por pantalla, entre otras.

Relación Complementaria (Tema 12)

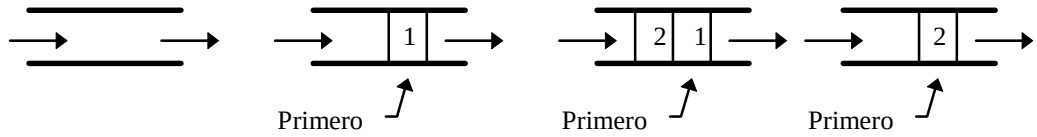
13. Una cola es un tipo especial de lista en la cual las eliminaciones de datos se realizan al principio de la lista (frente) y las inserciones se realizan en el otro extremo (final) (es como la cola del autobús). Las colas se utilizan para almacenar datos que deben ser procesados en el orden de llegada. El primer elemento que entra es el primero que sale. A este tipo de estructuras se las llama *FIFO* (*First-In, First-Out*).

Las operaciones asociadas a una cola son

1) `Crear`: devuelve una cola vacía.
2) `Insertar`: añade un elemento al final de la cola
3) `Sacar`: extrae el primer elemento en la cola. Es un error extraer de una cola vacía.
4) `Primero`: devuelve el valor del primer elemento en la cola pero no lo extrae. Es un error obtener el primer elemento de una cola vacía.
5) `EsVacia`: devuelve `TRUE` si la cola está vacía (no contiene ningún elemento).

Un ejemplo de uso de una cola de enteros es el siguiente:

```
C := Crear();      Insertar(C,1);     Insertar(C,2);     Extraer(C);
```



Escribir un módulo de biblioteca donde se defina el tipo cola de valores enteros y las operaciones asociadas.

14. Escribe un subprograma que tome como argumento dos listas simplemente enlazadas y construya una nueva lista que contiene tanto los nodos de la primera como los de la segunda. Los nodos de la lista resultante deben crearse.

15. Escribe un subprograma `Map` que tome como argumento una lista de valores enteros y una función que toma un entero y devuelve otro entero. Esta función debe devolver una nueva lista cuyos valores son los obtenidos al aplicar la función argumento a cada uno de los valores en la lista original. Escribe un programa que usando esta función lea una serie de valores en una lista, calcule una nueva lista con sus cuadrados y la escriba por pantalla.

16. Diseña un procedimiento `Purga` que elimine los elementos duplicados de una lista enlazada que tome como parámetro.

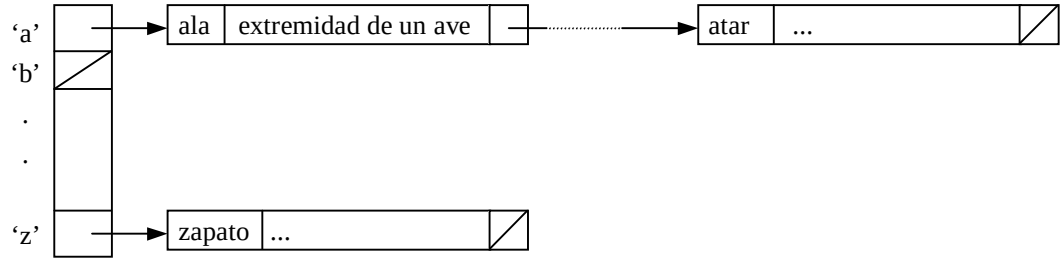
17. Escribe procedimientos recursivos e iterativos que a partir de una lista enlazada de enteros

1) Calcule el mayor elemento de la lista.
2) Calcule la suma de los elementos de la lista.
3) Calcule la longitud de la lista.

18. Una implementación más eficiente de una cola se obtiene usando una lista enlazada circular. Si el puntero exterior del anillo apunta siempre al último elemento de la cola, podemos siempre acceder al primero en un solo paso. Implementa una cola mediante una lista enlazada circular utilizando esta idea.

19. Un modo de almacenar un diccionario de palabras es mediante una *tabla hash*. Esta

estructura consiste en un array cuyo índice es `['a'..'z']` y donde cada posición del array es un puntero a una lista enlazada. Cada nodo de las listas enlazadas poseerá el nombre de una palabra y su definición (cadenas de caracteres)



A la hora de almacenar una palabra se crea un nuevo nodo con el nombre y la definición y se enlaza en la lista correspondiente a la primera letra de la palabra. La estructura es bastante eficiente, ya que para localizar una palabra basta con mirar en la lista correspondiente a su primera letra.

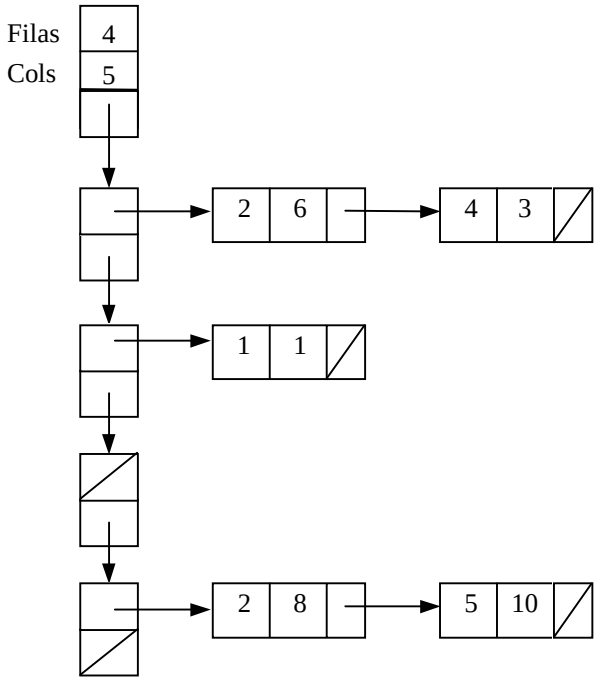
Escribe un módulo de biblioteca donde se defina el tipo TABLAHASH y operaciones para

- Insertar una palabra y su correspondiente definición en una tabla.
- Dada una palabra, buscar su definición en una tabla.
- Dada una palabra, eliminarla de una tabla.
- Mostrar todas las palabras y las definiciones asociadas en una tabla.

20. Una matriz dispersa es una matriz donde muchos de sus elementos tienen valor cero. En predicción meteorológica, por ejemplo, es necesario manejar matrices de este tipo con grandes dimensiones. Si utilizamos un array bidimensional estamos desperdiciando una gran cantidad de memoria. Una solución a este problema es utilizar listas enlazadas. Una matriz se representa por un registro de tres campos. El primero es el número total de filas de la matriz, el segundo el de columnas y el tercero una lista enlazada vertical con tantos elementos como filas tenga la matriz. Cada nodo de la lista vertical es un puntero a una lista enlazada con los nodos no nulos de la fila. Estos nodos tienen dos campos además del puntero: la columna a la cual corresponde el dato y el dato. Las listas horizontales están ordenadas de menor a mayor según el contenido del primer campo (la columna a la cual corresponde el dato). Por ejemplo la siguiente matriz

$$\begin{bmatrix} 0 & 6 & 0 & 3 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 & 10 \end{bmatrix}$$

es representada como:



Escribe un módulo de biblioteca donde se defina el tipo `MATRIZ` y las siguientes operaciones:

- `Crear`: toma como argumentos el número total de filas y columnas de una matriz y devuelve una matriz vacía.
- `Asignar`: toma como argumentos una matriz, una fila, una columna y un valor. Asigna este valor a la correspondiente posición de la matriz. Para ello hay que buscar si existe un nodo en la matriz correspondiente a la posición. Si es así, se asigna el nuevo valor. En otro caso se crea un nuevo nodo con los valores adecuados y se enlaza en su posición correspondiente.
- `Elemento`: toma una matriz, una fila y una columna y devuelve el valor del elemento que ocupa dicha posición en la matriz. Si no existe el correspondiente nodo en la matriz, se devolverá cero.
- `WrMatriz`: escribe una matriz que toma como argumento por pantalla.
- `RdMatriz`: permite crear una matriz y leer sus componentes desde teclado.

Como aplicación de la biblioteca, escribe un programa que lea dos matrices del teclado, calcule su producto y lo escriba por pantalla.