11. LISTAS

Conceptos: *Lista, lista dinámica simple, lista circular, lista doblemente enlazada, pila, cola*

Resumen: Este capítulo se centra en la estructura lineal de tipo dinámico más general, estudiándose tanto su descripción (que se basará en el concepto de puntero) como las principales a realizar sobre ella. Se define una lista como una colección de elementos del mismo tipo entre los que existe una relación de orden determinada por la posición de dicho elemento. Cada uno de estos elementos consta de un campo que almacena la información contenida en el nodo. Dependiendo del tipo de lista que se trate, cada elemento tendrá, además uno o dos campos de tipo puntero que apuntarán a otros elementos de la lista. Así, en el caso de las listas dinámicas simplemente enlazadas existe un único puntero que apunta al siguiente elemento de la lista. En las listas doblemente enlazadas existe otro puntero adicional que apunta al elemento anterior de la lista. Las listas circulares tienen la propiedad de que su último enlaza con el primero de forma que se facilita el acceso a todos los elementos de la lista desde cualquiera de ellos. Los algoritmos considerados para cada una de estas estructuras de datos son los de creación de una lista vacía, recorrido, búsqueda de información e inserción y borrado de elementos, que se apoyarán en los recursos básicos de reserva y liberación dinámica de memoria.

Objetivos específicos. Al finalizar el tema, el alumno deberá ser capaz de:

a) Definir el concepto de lista (*Conocimiento*)
b) Escribir la declaración de tipos puntero y de variables de variables necesarios para poder trabajar una estructura de tipo lista dinámica (*Comprensión*)
c) Interpretar el código fuente de un programa que trabaje con una estructura de datos de tipo lista (*Comprensión*)
d) Codificar una tarea sencilla convenientemente especificada, utilizando una estructura de datos de tipo lista (*Aplicación*)

11.1. INTRODUCCIÓN

Matemáticamente, una lista es una **sucesión finita** de cero, uno o más elementos del mismo tipo. Desde el punto de vista de las estructuras de datos, una lista es un conjunto finito de elementos, en el que para acceder a uno de ellos hay que pasar por todos los situados antes que él. Una lista es, por lo tanto, una estructura de datos secuencial. Ejemplos de listas utilizadas cotidianamente pueden ser: la lista de personas en la cola para visitar a un médico, la secuencia de pasos de una receta de cocina, la lista de jugadores de un equipo de fútbol,...

Para Aho, las listas son estructuras de datos particularmente **flexibles** ya que pueden ir creciendo o decreciendo según se necesite y pueden accederse, insertarse o eliminarse elementos en cualquier posición dentro de la lista.

En un programa se puede trabajar con una lista de datos a través de una variable de tipo `array`, es decir, una estructura de datos estática, pero es preferible implementar la lista mediante una estructura de datos dinámica para aprovechar y emplear de una forma más flexible y óptima la memoria.

Como ya se adelantó en el capítulo anterior, para llevar esto a cabo hay que utilizar variables de tipo apuntador o puntero. En este capítulo se tratará el tema de las listas no a nivel teórico, sino fundamentalmente, desde el punto de vista de su implementación como una estructura de datos dinámica dentro de un programa.

11.2. GENERACIÓN DE UNA LISTA DINÁMICA

Para construir una lista dinámica de datos se han de definir los elementos de la lista como datos de tipo `record` con dos clases de campos: una serie de campos que almacenan información y otra serie de campos de tipo apuntador o puntero. Al menos se necesita uno de estos campos para que almacene la posición de memoria del siguiente elemento de la lista. Por lo tanto, los campos puntero de los elementos de las listas utilizados en general en los programas almacenarán direcciones de memoria de ese mismo tipo de datos (apuntarán a otros datos que son elementos de la propia lista). El acceso a un elemento de una lista dinámica sería comparable a la búsqueda de un tesoro escondido en la que, en cada pista que se halla, se dice donde encontrar la siguiente. Un ejemplo de definición de un elemento de una lista *simple* sería el siguiente:

```
type  puntero = ^elemento;
      elemento = record
            dato : integer;
            sig  : puntero
            end;
var   p : puntero;
```

Se denomina **lista dinámica simple** porque sus elementos sólo necesitan un campo de tipo puntero. En este caso, los elementos de la lista simple son de tipo `record` con dos campos: en uno de ellos se almacena un número entero y en el otro la dirección de memoria del siguiente elemento en la lista. Hay que hacer constar que estas declaraciones no dan lugar a un error de compilación al utilizar el identificador `elemento` en la definición de `puntero` sin haberlo previamente declarado: es una excepción a la regla. Ya que, en todo caso, ¿qué habría que definir primero: `elemento` ó `puntero`? El compilador decide, en este caso, que se debe declarar primero el tipo de dato `puntero`. El siguiente programa utiliza la declaración de tipos anterior para generar una lista y posteriormente visualizar los valores que se acaban de introducir.

```
program lista_1;
{ Crea una lista en el orden contrario a su introduccion }
```

```
{ Primero introduce los datos y luego los visualiza }
type  puntero  = ^elemento;
      elemento = record
        dato : integer;
        sig : puntero
        end;
var p, q : puntero;
    a : integer;
begin
p:=nil;            { Almacena en p la constante nil }
readln(a);         { Asigna un valor a la variable entera a }
while a<>0 do
  begin
  new(q);      { Reserva espacio para una variable dinamica q^ }
                    { y almacena en q su direccion de memoria }
  q^.dato:=a;       { Almacena el valor de a en el campo dato }
  q^.sig:=p;     { Almacena en sig la misma direccion que en p }
  p:=q;             { Y ahora en p la misma direccion que en q }
  readln(a)         { Asigna otro valor a la variable entera a }
  end;
  { Cuando se cumple que a=0 el bucle dejar de ejecutarse }
writeln('Valores de la lista : ');
  { El siguiente bucle visualiza por pantalla el valor
    almacenado en el campo dato de todos los elementos
    de la lista. Acaba cuando la variable q almacena
    la constante nil: esta indica el final de la lista }
while q<>nil do
  begin
  writeln(q^.dato);
  q:=q^.sig
  end
end.
```

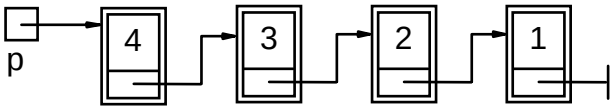

Figura 70. Lista generada con el programa `lista_1` al introducir los valores 1, 2, 3, 4 y 0.

11.3. CREACIÓN DE UNA LISTA SIMPLE

El siguiente programa (que posteriormente se convertirá en un procedimiento) permite la creación de una lista que almacena una secuencia de números enteros con el mismo orden que el de su introducción:

```
program crearlista;
type  puntero = ^elemento;
      elemento = record
        dato : integer;
        sig  : puntero
        end;
var prim,p,q : puntero; aux : integer;
begin
new(prim);
write('valor : ');
readln(prim^.dato);
p:=prim;
write('Continuar (0 para terminar) ');
readln(aux);
while aux<>0 do
      begin
      new(q);
      write('valor : ');
      readln(q^.dato);
      p^.sig:=q;
      p:=q;
      write('Continuar (0 para terminar) ');
      readln(aux)
```

```
      end;
p^.sig:=nil
end.
```

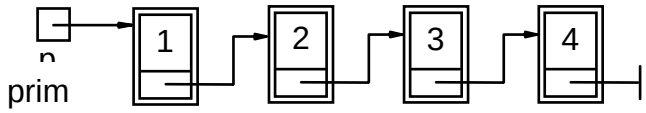

Figura 71. Lista creada durante la ejecución del programa `crearlista` al introducir los valores `1`, `2`, `3`, `4` y `0`.

Para convertir este programa en un procedimiento, que pueda ir incluido de forma independiente en cualquier programa, hay que sustituir la cabecera del programa por la cabecera del procedimiento:

```
procedure crearlista(var prim:puntero);
```

A continuación, eliminar la declaración de tipos (ha de estar en la del programa principal) y la variable `prim` de la declaración de variables y sustituir el punto final del programa por un punto y seguido. Es importante remarcar el detalle de que, en el procedimiento `crearlista`, `prim` debe ir como parámetro variable, para que la variable de tipo puntero correspondiente en la llamada, almacene la dirección de memoria del primer elemento de la lista una vez terminado de ejecutar el procedimiento.

```
procedure crearlista (var prim:puntero);
            var p,q : puntero;
                aux : integer;
            begin
            new(prim);
            write('valor : ');
            readln(prim^.dato);
            p:=prim;
            write('Continuar (0 para terminar) ');
            readln(aux);
            while aux<>0 do
                  begin
                  new(q);
                  write('valor : ');
                  readln(q^.dato);
                  p^.sig:=q;
                  p:=q;
                  write('Continuar (0 para terminar) ');
                  readln(aux)
                  end;
            p^.sig:=nil
            end;
```

Pero, ¿para qué se puede emplear una lista de datos dinámicos en un programa?

11.4. EJEMPLO DE APLICACIÓN DE UNA LISTA SIMPLE

El siguiente programa calcula el producto escalar de dos vectores de números enteros cuyas componentes están almacenadas en sendas listas. Se va a aprovechar el procedimiento anterior `crearlista` que permite almacenar los componentes de cada vector en una lista.

```
program producto_escalar;
type  puntero = ^elemento;
      elemento = record
        dato : integer;
        sig  : puntero
        end;
var   p1,p2 : puntero; s : integer;
procedure crealista(var prim:puntero);
      begin
      ...
      end;
{ ****** Programa principal ****** }
```

```
begin
writeln('Introducir 1° vector :');
crealista(p1);
writeln('Introducir 2° vector :');
crealista(p2);
s:=0;
while (p1<>nil) and (p2<>nil) do    { mientras las dos listas }
      begin                       { tengan al menos un elemento }
      s:=s+p1^.dato*p2^.dato;
      p1:=p1^.sig;
      p2:=p2^.sig
      end;
writeln('Producto escalar = ',s)
end.
```

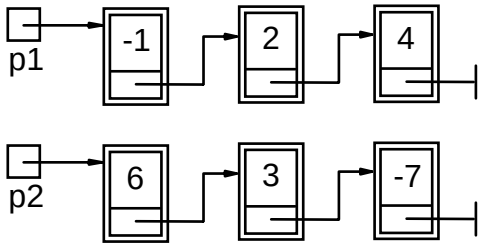

Figura 72. Ejemplo de listas dinámicas generadas con el programa anterior

Como consecuencia de este ejemplo puede plantearse la siguiente pregunta: ¿es más cómodo utilizar punteros para construir estructuras de datos dinámicas que emplear variables de tipo `array` u otras estructuras estáticas en un programa? En el caso anterior es posible que no sea más cómodo (pruebe el lector a construir el mismo programa con variables de tipo `array`).

Pero, en otros casos, es importante tener en cuenta que las estructuras de datos estáticas no pueden ser creadas durante la ejecución del programa, ni cambiar de tamaño, ni modificar su estructura, mientras que las estructuras de datos dinámicas sí lo pueden hacer durante en tiempo de ejecución.

11.5. MANIPULACIÓN Y MODIFICACIÓN DE UNA LISTA

Una vez creada y almacenada la lista dinámica en la memoria del ordenador puede manipularse y modificarse cuantas veces sea necesario durante la ejecución del programa: las listas pueden crecer, decrecer,… A continuación, se mostrarán distintas operaciones básicas que pueden realizarse con una lista dinámica simple. Las declaraciones de tipos de dato utilizadas son las siguientes:

```
type  puntero = ^elemento;
            elemento = record
                  dato : integer;
                  sig  : puntero
                  end;
```

11.5.1. Recorrido de una lista

Una secuencia de pasos o algoritmo para realizar un recorrido por todos los elementos de una lista dinámica es el siguiente:

1. Empezar por el primer elemento –principio- de la lista
2. Mientras que no se llegue al final de la lista:
   2.1. *Operación* con el elemento correspondiente
   2.2. Acceder al siguiente elemento de la lista
3. Fin del algoritmo

Por ejemplo, el siguiente procedimiento sigue el algoritmo anterior para visualizar por pantalla el campo `dato` de todos los elementos de una lista dinámica a cuyo primer elemento apunta el parámetro formal por valor `p` de tipo `puntero`:

```
{ El procedimiento listado visualiza por pantalla el valor del
  campo dato de todos los elementos de una lista dinamica a
  cuyo primer elemento apunta el parametro por valor p }
procedure listado(p:puntero);          {1}
      begin
      while p<>nil do                  {2}
            begin
            writeln(p^.dato);          {2.1}
            p:=p^.sig                  {2.2}
            end
      end;                             {3}
```

Al realizar en el programa la llamada al procedimiento `listado(primero);` durante la ejecución del procedimiento, el parámetro formal `p` de tipo `puntero` va recorriendo cada uno de los elementos de la lista, desde el primero hasta el último, a la vez que se va visualizando por pantalla el campo `dato` del elemento al que apunta.

Como una lista puede considerarse una estructura recursiva que, o bien está vacía o está formada por un elemento seguido por otra lista compuesta por el resto de los elementos de la lista original, algunas operaciones que se realizan con este tipo de estructuras pueden definirse de forma recursiva. Por ejemplo, el procedimiento `listadoR` realiza la misma tarea que el procedimiento `listado` mostrado anteriormente, pero de manera recursiva:

```
{ El procedimiento listadoR visualiza por pantalla el valor del
  campo dato de todos los elementos de una lista dinamica a
  cuyo primer elemento apunta el parametro por valor p }
procedure listadoR(p:puntero);
            begin
            if p<>nil then
                  begin
                  writeln(p^.dato);
                  listadoR(p^.sig)
                  end
            end;
```

11.5.2. Búsqueda de un elemento de una lista

Un algoritmo para realizar una búsqueda de un elemento que cumpla una determinada condición dentro de una lista dinámica es el que se indica a continuación:

1. Empezar por el primer elemento de la lista
2. Mientras que no se llegue al final de la lista y el elemento no cumpla la condición de la búsqueda
    2.1. Acceder al siguiente elemento de la lista
3. Si se ha llegado al final de la lista
    3.1 entonces, elemento no encontrado
    3.2 en caso contrario, elemento encontrado
4. Fin del algoritmo

El siguiente procedimiento realiza una búsqueda de un elemento cuyo campo `dato` almacene el valor `v`, en una lista a cuyo primer elemento apunta el parámetro `primero`. En caso de hallarse el dato en la lista, el parámetro por variable `p` devuelve la posición de memoria del elemento buscado y el otro parámetro por variable, `eureka`, devuelve el valor `true`. En caso de no hallarse, el parámetro por variable `p` devuelve la constante `nil` y el otro parámetro por variable `eureka` devuelve el valor `false`.

```
procedure buscar(primero:puntero; v:integer;
                 var p:puntero; var eureka:boolean);
```

```
      begin
p:=primero;
while (p<>nil) and (p^.dato<>v) do
            p:=p^.sig;
if p=nil
            then  begin
                  writeln('Elemento no encontrado');
                  eureka:=false
                  end
            else  begin
            writeln('Elemento encontrado');
                  eureka:=true
                  end
end;
```

En lugar de haber construido este procedimiento, podría haberse escrito una función que devolviera el valor booleano `true`, en caso de encontrar cierto valor en la lista y `false` en caso contrario. En cualquier caso, este procedimiento va a ser de utilidad más adelante. También puede implementarse una versión recursiva de este procedimiento, por ejemplo:

```
procedure buscarR(primero:puntero; v:integer;
                  var p:puntero; var eureka:boolean);
      begin
p:=primero;
if p=nil
            then  begin
                  writeln('Elemento no encontrado');
                  eureka:=false
                  end
            else  if p^.dato=v
                        then  begin
                              writeln('Elemento encontrado');
                              eureka:=true
                              end
                        else buscarR(prim^.sig,v,p,eureka)
end;
```

11.5.3. Inserción de un nuevo elemento en una lista

Dada una lista simple dinámica se puede insertar un nuevo elemento al que apunta una variable puntero `q` después del elemento de la lista al que apunta la variable puntero `p`.

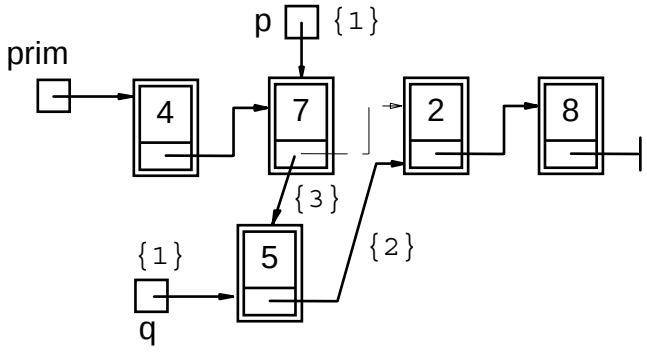

Figura 73. Representación gráfica del algoritmo de inserción de un elemento en una lista simple

La secuencia de pasos para resolver este problema ó algoritmo va a ser la siguiente. En primer lugar, es necesario almacenar en `q` la dirección de memoria del nuevo elemento a insertar y en `p` la posición del elemento detrás del cual se realiza la inserción `{1}`. En segundo lugar se enlaza el nuevo elemento con el siguiente en la lista `{2}` y en tercer lugar se reengancha el que le precede con el nuevo `{3}`. El algoritmo ya traducido al lenguaje de TurboPascal y convertido en un procedimiento es el mostrado a continuación:

```
procedure insertar(p,q:puntero);            {1}
```

```
      begin
      q^.sig:=p^.sig;                    {2}
      p^.sig:=q                          {3}
      end;
```

Existe un pequeño inconveniente en el anterior procedimiento `insertar`: tal y como está diseñado, no permite insertar un nuevo elemento como primero de una lista o en una lista vacía. Para conseguir esto, es necesario modificarlo convenientemente. Se deja como ejercicio propuesto de programación.

11.5.4. Eliminación de un elemento de una lista

Existen también diversas formas de eliminar un elemento de una lista dinámica simple de datos. El algoritmo para borrar un elemento de una lista simple puede ser el siguiente:

1. Determinar el elemento a borrar y la lista en la que se realiza esta modificación. Para ello se almacena en `p` la dirección de memoria del elemento que va a eliminarse y en `prim` la dirección del primer elemento de la lista.
2. Almacenar en una variable puntero auxiliar `q` la dirección de memoria del elemento que precede en la lista al que se va a eliminar.
3. Modificar en este elemento el valor del campo `puntero` dándole la dirección de memoria del elemento que sucede al que se va a borrar.
4. Liberar a la memoria del espacio ocupado por este elemento.

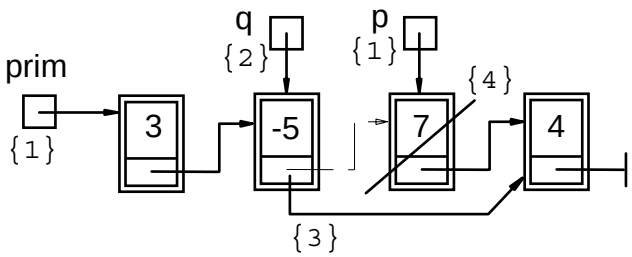

Figura 74. Representación gráfica del algoritmo de borrado de un elemento de una lista

Para llevar a cabo esto en un programa de TurboPascal puede emplearse el siguiente procedimiento que utiliza el algoritmo anteriormente explicado:

```
procedure borrar(var prim:puntero; p:puntero);    {1}
  var q:puntero;
  begin
  if p=prim then prim:=prim^.sig
     else begin
          q:=prim;
          while q^.sig<>p do q:=q^.sig;          {2}
          q^.sig:=p^.sig                         {3}
          end;
  dispose(p)                                     {4}
  end;
```

11.5.5. Integración de todos los procedimientos en un programa

Este programa permite gestionar y actualizar el trabajo con estructuras de datos de tipo lista simple dinámica integrando los procedimientos implementados anteriormente.

```
program gestion_listas;
type  puntero  = ^elemento;
      elemento = record
        dato : integer;
        sig  : puntero
        end;
var   primero, p1, p2 : puntero;
```

```
      opcion, valor   : integer;
      exito           : boolean;
procedure crealista(var prim:puntero);
  var p,q : puntero;
      aux : integer;
  begin
  new(prim);
  write('valor : ');
  readln(prim^.dato);
  p:=prim;
  write('Continuar (0 para terminar) ');
  readln(aux);
  while aux<>0 do
      begin
      new(q);
      write('valor : ');
      readln(q^.dato);
      p^.sig:=q;
      p:=q;
    write('Continuar (0 para terminar) ');
                  readln(aux)
                  end;
            p^.sig:=nil
            end;
procedure buscar(prim:puntero;valor:integer;
                 var p:puntero;var b:boolean);
  begin
  ...
  end;
procedure borrar(var prim:puntero; p:puntero);
  begin
  ...
  end;
procedure visualiza(p:puntero);
  begin
  writeln('Elementos de la lista : ');
  while p<>nil do
      begin
      writeln(p^.dato);
      p:=p^.sig
      end
  end;
procedure insertar(p,q:puntero);
  begin
  ...
  end;
{ *** PROGRAMA PRINCIPAL *** }
begin
repeat
writeln('   0: Fin;
            1: Crear lista;
            2: Añadir elemento;
            3: Consultar;
            4: Borrar;
            5: Listar.');
readln(opcion);
case opcion of
      1: crealista(primero);
      2: begin
         new(p2);
         writeln('Valor a añadir : ');
         readln(p2^.dato);
         writeln('¿Detras de que dato se va a insertar?');
         readln(valor);
         buscar(p1,primero,valor,exito);
         if exito then insertar(p1,p2);
         end;
      3: begin
         writeln('Valor del dato a encontrar : ');
         readln(valor);
```

```
            buscar(p1,primero,valor,exito);
            if exito then writeln(p1^.dato)
            end;
         4: begin
            writeln('Dato que se desea borrar : ');
            readln(valor);
            buscar(p1,primero,valor,exito);
            if exito then borrar(primero,p1)
            end;
         5: visualiza(primero)
         end;
until opcion=0;
writeln('Fin del programa.');
end.
```

11.6. LISTAS SIMPLES CIRCULARES

Una lista circular es una clase de lista en la que el campo de tipo puntero del último elemento apunta al primer elemento de la lista.

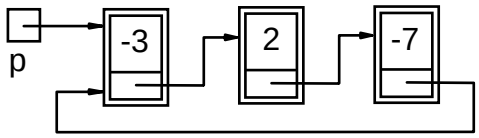

Figura 75. Ejemplo gráfico de una lista simple circular

Este tipo particular de lista simple puede facilitar en algunos casos el acceso a sus elementos. Hay que hacer notar que no existe ningún campo `puntero` que almacene la constante `nil`, con lo cual no puede utilizarse esto para saber cuál es el último elemento de la lista. Por esta razón, para manipular una lista simple circular en un programa sería necesario modificar algunos de los procedimientos utilizados en las listas simples para no caer en un bucle infinito. Estas modificaciones de los procedimientos anteriormente vistos para que puedan ser empleados en listas simples circulares se deja como ejercicio de programación.

11.7. LISTAS DOBLEMENTE ENLAZADAS

En las listas simples es fácil acceder desde un elemento determinado a otro que esté situado posteriormente, pero no lo es tanto si este acceso se quiere realizar en sentido contrario, es decir, desde un elemento a otro que esté situado antes. En los elementos de las listas doblemente enlazadas o listas dobles se incluye al menos un campo apuntador que almacena la dirección de memoria del siguiente elemento en la lista y otro que apunta al anterior. Esto facilita el acceso a los elementos que se encuentran en cualquier posición relativa en la lista. Un ejemplo de declaración de un tipo de dato que componga una lista dinámica doblemente enlazada es el siguiente:

```
type puntero = ^elemento
      elemento = record
       dato : integer;
       sig : puntero;
       prec : puntero
       end;
```

Es habitual utilizar dos variables puntero por cada estructura de tipo lista doble que quiera construirse. Declarando `var primero,ultimo:puntero;` podría construirse la estructura de datos de la figura 76.

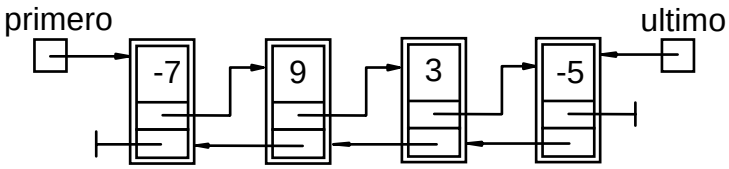

Figura 76. Ejemplo gráfico de una lista doblemente enlazada

11.8. RUTINAS PARA MANIPULAR LISTAS DOBLES

De forma parecida a lo realizado con las listas simples pueden escribirse procedimientos y funciones que permitan crear, manipular y modificar una lista doble durante la ejecución de un programa.

11.8.1. Creación de listas dobles

El siguiente procedimiento crea una lista doblemente enlazada de datos de tipo entero introducidos por el teclado. Con el valor dado a la variable entera `aux` puede elegirse si continuar o no la introducción de elementos en la lista.

```
procedure crealistadoble(var primero,ultimo:puntero);
  var p, q : puntero;
      aux : integer;
  begin
  new(primero);
  write('valor : ');
  readln(primero^.dato);
  p:=prim;
  p^.prec:=nil;
  write('Continuar (0 para terminar) ');
  readln(aux);
  while aux<>0 do
      begin
      new(q);
      write('valor : ');
      readln(q^.dato);
      q^.prec:=p;
      p^.sig:=q;
      p:=q;
      write('Continuar (0 para terminar) ');
      readln(aux)
      end;
p^.sig:=nil;
ultimo:=p
end;
```

Al realizar la llamada al procedimiento `crealistadoble(a,b);` una vez finalizada su ejecución, la variable puntero `a` se deja apuntando al primer elemento de la lista y la variable puntero `b` al elemento final.

11.8.2. Recorrido o listado de elementos de una lista doble

El procedimiento para realizar un listado en una lista doble es prácticamente el mismo que el construido para una lista simple:

```
procedure listado(p:puntero);
  begin
  writeln('Elementos de la lista : ');
  while p<>nil do
      begin
      writeln(p^.dato);
      p:=p^.sig
     end
```

```
  end;
```

Al realizar en el programa la llamada al procedimiento `listado(primero);` durante la ejecución del procedimiento el parámetro `p` de tipo `puntero` va recorriendo cada uno de los elementos de la lista, desde el primero hasta el último, a la vez que se va visualizando por pantalla el campo `dato`.

11.8.3. Listado inverso de los elementos de una lista doble

El procedimiento para realizar un listado en una lista doble es muy parecido al anterior:

```
procedure listadoinverso(p:puntero);
  begin
  writeln('Elementos de la lista : ');
  while p<>nil do
            begin
            writeln(p^.dato);
            p:=p^.prec
            end
  end;
```

En este caso al realizar en el programa la llamada al procedimiento `listadoinverso(ultimo);` el parámetro `p` de tipo `puntero` recorre, durante la ejecución de la rutina, la lista desde el último elemento al primero de la lista.

11.8.4. Inserción de elementos en una lista doble

El siguiente procedimiento permite la inserción de un nuevo elemento en una lista doblemente enlazada siguiendo el algoritmo mostrado gráficamente en la figura 8.

```
procedure insercion(p,q:puntero);      {1}
  begin
  q^.sig:=p^.sig;                      {2}
  q^.prec:=p;                          {3}
  p^.sig^.prec:=q;                     {4}
  p^.sig:=q                            {5}
  end;
```

Este procedimiento permite realizar la inserción de un elemento al que apunta la variable puntero `q` después del elemento de la lista al que apunta la variable puntero `p`. Este procedimiento tiene un inconveniente: no se puede introducir ningún elemento al principio o al final de la lista. Se deja como ejercicio de programación el construir un procedimiento que sí permita en estos casos la inserción.

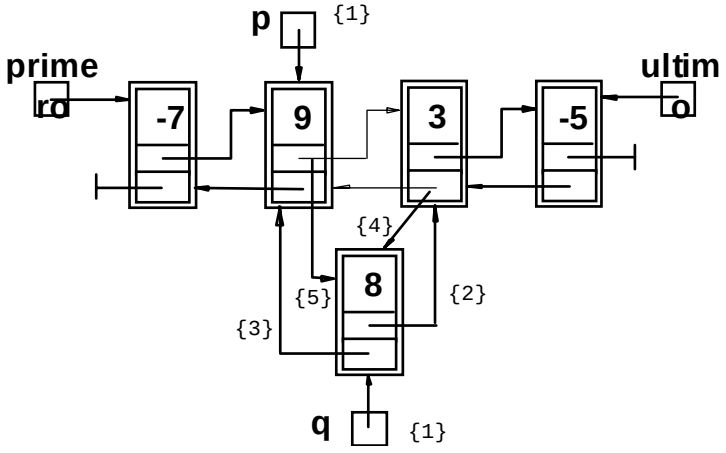

Figura 77. Inserción de un elemento en una lista doblemente enlazada

11.8.5. Borrado de elementos de una lista doble

El siguiente procedimiento permite la eliminación de un elemento en una lista doblemente enlazada siguiendo el algoritmo mostrado gráficamente en la figura 9.

```
procedure borrado(p:puntero);                    {1}
  begin
  p^.prec^.sig:=p^.sig;                          {2}
  p^.sig^.prec:=p^.prec;                         {3}
  dispose(p)                                     {4}
  end;
```

Este procedimiento permite eliminar un elemento al que apunta la variable puntero de una lista doble. Pero, ¿qué ocurre cuando se intenta borrar el primero ó el último elemento de la lista con este procedimiento? En estos casos, los pasos `{2}` y `{3}` del procedimiento anterior no son correctos, por lo que la rutina debe modificarse conveniente para considerarlos de forma adecuada.

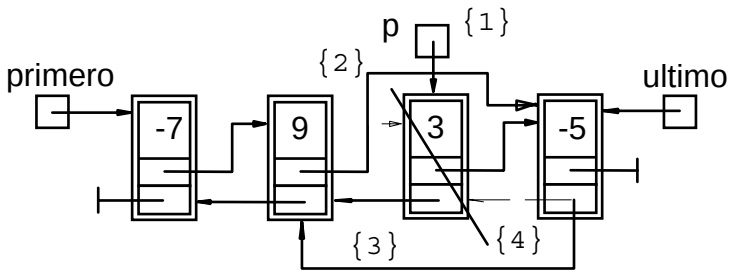

Figura 78. Borrado de un elemento de una lista doblemente enlazada

Como ejercicio se sugiere la integración de todos los procedimientos de listas doblemente enlazadas en un programa.

11.9. LISTAS CIRCULARES DOBLEMENTE ENLAZADAS

En las listas circulares doblemente enlazadas el primero y el último de los elementos están enlazados mediante los respectivos campos puntero siguiendo el esquema de la figura 79. En este caso, no hay ningún campo que almacene la constante `nil`.

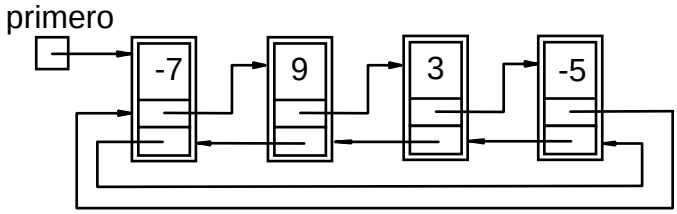

Figura 79. Lista circular doblemente enlazada

11.10. CASOS PARTICULARES DE LISTAS: PILAS Y COLAS

En programación pueden emplearse casos particulares de listas. A continuación se mostrarán dos ejemplos: la pila y la cola.

11.10.1. Pilas

Una pila (*stack* en la terminología inglesa) es un caso particular de lista de datos en el que todas las inserciones y lecturas (éstas conllevan la eliminación del dato de la pila) se hacen por un mismo extremo, que se denomina cima (*top*). En la bibliografía informática una pila se

clasifica como una estructura de tipo lista LIFO: *Last-In-First-Out* (Ultimo en entrar, primero en salir).

En la vida normal se emplea muchas veces el concepto de pila. Por ejemplo: una pila de platos en un armario, una pila de libros sobre una mesa o la ejecución de las llamadas a procedimientos anidados en un programa como el que se muestra a continuación:

```
program ejemplo;
procedure A;
    begin
    ...
    end;
procedure B;
    begin
    A;
    end;
procedure C;
    begin
    B;
    end;
begin
C;
end.
```

Es interesante observar como van realizándose las llamadas a cada uno de los procedimientos y que el último procedimiento en llamarse es el primero que termina de ejecutarse.

11.10.2. Operaciones que pueden realizarse con pilas

Las operaciones básicas que pueden realizar con una estructura tipo pila son las siguientes:

a) Crear-inicializar una pila.
b) Insertar un elemento.
c) Extraer-eliminar un elemento.
d) Comprobar si la pila está vacia.
e) Número de elementos de la pila.
f) Determinar el tamaño máximo de la pila (en caso de que lo tenga).
g) Comprobar si la pila está llena.
h) Ver el último elemento.
i) Visualizar todos los elementos.
j) Inicializar la pila eliminando todas las variables dinámicas de la memoria...

Pueden implementarse pilas de datos de distintas maneras. Por ejemplo, empleando estructuras de datos estáticas como `array`s, o bien estructuras de datos dinámicas y punteros.

11.10.3. Implementación de una pila con una variable array estática

A continuación se muestra un ejemplo de código con el que se construye una pila con una variable estática de tipo `array`.

```
const  max_pila = 10; { ¡determinacion y
                        limitacion del tamaño! }
type   elemento = integer;
       pila = array[1..max_pila] of elemento;
var    p     : pila;
       cima  : 0..max_pila;
       valor : elemento;
```

Sentencias para insertar en la pila el entero almacenado en `valor`:

```
cima := cima+1;
p[cima] := valor;
```

Para extraer un elemento:

```
valor:=p[cima];
cima:=cima-1;
```

11.10.4. Implementación con punteros y variables dinámicas

En este caso, se utilizan variables dinámicas de tipo `record` de forma equivalente a como se hacía con las listas dinámicas (al fin y al cabo, una pila es un caso particular de lista):

```
type  puntero  = ^elemento;
      elemento = record
            dato : integer;
            ant  : puntero
            end;
var cima, p: puntero;
```

Las operaciones que pueden realizarse con estos elementos y las sentencias correspondientes son:

i) Sentencia para crear-inicializar una pila:

```
cima:=nil;
```

ii) Para insertar un elemento:

```
new(p);
p^.dato:=a;
p^.ant:=cima;
cima:=p;
```

iii) Para extraer-eliminar un elemento:

```
p:=cima;
cima:=p^.ant;
dispose(p);
```

iv) Para saber si la pila está vacía:

```
pilavacia:=(cima=nil);
```

v) ¿Pila llena?

```
pilallena:=(longitud(...)=max_pila);
```

Se deja como ejercicio escribir las funciones y procedimientos necesarios que permiten manipular una pila de datos.

11.10.5. Colas

Una cola es un caso particular de estructura tipo lista en la que todas las inserciones se hacen por un extremo y todas las extracciones-eliminaciones se efectúan en el otro extremo. En la bibliografía informática, una cola se califica como una estructura de tipo FIFO: *First-In-First-Out* (Primero en entrar, primero en salir). Por ejemplo: la cola de personas que sube a un autobús, la cola de documentos a imprimir en una impresora...

Se deja como ejercicio escribir las funciones y procedimientos necesarios que permitan manipular una cola de datos.

11.11. APLICACIONES DE LAS ESTRUCTURAS DE DATOS DINÁMICAS: MANIPULACIÓN DE MATRICES

Para manipular una matriz en un programa será necesario almacenarla de alguna manera en memoria. Existen diferentes modos de almacenar en memoria los elementos de una matriz bidimensional de m filas y n columnas:

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

Figura 80. Matriz bidimensional de m filas y n columnas de valores reales.

1º Método. Mediante una variable estática de tipo `array` bidimensional. Ejemplo:

```
type  matriz = array[1..3,1..3] of real;
var   a    : matriz;
      i,j  : integer;
begin
writeln('INTRODUCIR VALORES PARA LOS ELEMENTOS DE A:');
for i:=1 to 3 do
      for j:=1 to 3 do
            begin
            write('Valor de a[',i,',',j,'] = ');
            readln(a[i,j])
            end;
...
```

Ventajas:
1. Sencillez de manipulación.

Inconvenientes:
1. Estructura rígida.
2. Si se emplea una variable global se reserva espacio en memoria durante toda la ejecución del programa. Si se emplea una variable local se reserva espacio durante la ejecución de la rutina.
3. Tamaño del segmento de datos limitado a algo menos de 64kB (para variables globales de programas y unidades). Tamaño del segmento de pila limitado a 64kB como máximo (para variables locales de funciones y procedimientos).
4. En cualquier caso el compilador no permite definir una estructura de datos de este tipo de más de 64kB.

2º Método. Mediante una variable de tipo apuntador que apunte a una variable dinámica de tipo `array` bidimensional (con dos subíndices). Ejemplo:

```
const m = 5;
      n = 8;
type  elemento = integer;
      matriz = array[1..m,1..n] of elemento;
      puntero = ^matriz;
var   a : puntero;
      i,j : integer;
begin
new(a);
writeln('INTRODUCIR VALORES PARA LOS ELEMENTOS DE A:');
for i:=1 to m do
      for j:=1 to n do
            begin
            write('Valor de A[',i,',',j,'] = ');
            readln(a^[i,j])
```

```
                end;
      ...
      dispose(a);
      ...
      end.
```

Ventajas:
1. Sencillez de manipulación.
2. Se reserva espacio en memoria sólo cuando se necesita durante la ejecución del programa y en el segmento montículo (*heap*) o de almacenamiento dinámico (menor limitación de espacio).
3. Tamaño del segmento de montículo limitado al espacio libre en memoria.

Inconvenientes:
1. Estructura rígida: nº de elementos fijo a priori.
2. En cualquier caso no puede definirse una estructura de datos de este tipo de más de 64kB.

3º Método. Mediante una lista simple en la que se almacenan los elementos de la matriz ordenados por filas.

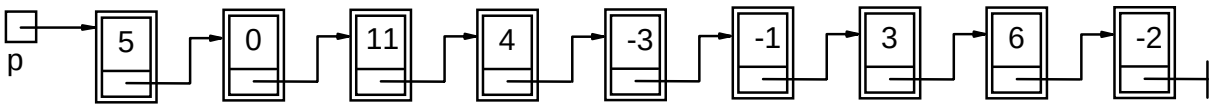

Figura 81. Estructura de tipo lista simple

Ventajas:
1. Flexibilidad. El número máximo de elementos no se fija en la compilación. La única limitación es el espacio libre en memoria.
2. Uso racional de la memoria disponible: se reserva espacio sólo cuando se necesita.

Inconvenientes:
1. El acceso a los elementos de la matriz no es el más sencillo: es complicado determinar a qué fila o columna pertenece un elemento. (Trátese de multiplicar dos matrices almacenadas de esta forma.)

4º Método. Mediante estructuras de datos de tipo listas cruzadas ó entrelazadas. Cada elemento de la matriz pertenece a dos listas que almacenan, respectivamente, todos los elementos de la fila y columna correspondientes. Las variables dinámicas empleadas son de tipo `record` con, al menos, dos campos de tipo puntero: en uno se almacena la posición en memoria del siguiente elemento en la lista, y en el otro, la del siguiente elemento en la columna.

```
type puntero  = ^elemento;
       elemento = record
         dato : real;
          fil : puntero;
          col : puntero;
         end;
       vector   = array[1..10] of puntero;
var dimension : integer;
      f1, c1    : vector;
procedure crea_matriz(var f,c:vector; n:integer);
      var   p,q : puntero;
            i,j : integer;
      begin
      writeln('INTRODUCIR ELEMENTOS DE LA MATRIZ POR FILAS.');
      for i:=1 to n do
            begin
            new(f[i]);
            p:=f[i];
            write('Elemento : ');
            readln(p^.dato);
```

```
            for j:=2 to n do
                  begin
                  new(p^.fil);
                  p:=p^.fil;
                  write('Elemento : ');
                  readln(p^.dato)
                  end;
            p^.fil:=nil
            end;
      p:=f[1];
      for i:=1 to n do
            begin
            c[i]:=p;
            p:=p^.fil
            end;
      for i:=2 to n do
            begin
            p:=f[i-1];
            q:=f[i];
            for j:=1 to n do
                  begin
                  p^.col:=q;
                  q^.col:=nil;
                  p:=p^.fil;
                  q:=q^.fil
                  end
            end;
      if n=1 then c[1]^.col:=nil
      end;

  (* *******  Programa principal  ******* *)
  begin
  write('Dimension (Maxima = 10): ');
  readln(dimension);
  crea_matriz(f1,c1,dimension);
  { ... }
  end.
```

Ventajas: 1. Flexibilidad de la estructura.
2. Se optimiza el uso del espacio de memoria,
3. Fácil acceso a cualquier elemento.

Inconvenientes: 1. ¿Diseño y algoritmos de manipulación complicados?

En función de las necesidades de programación, el programador deberá elegir alguno de los métodos anteriores, o bien buscar otra solución, para un problema dado.

Bibliografía básica

- **García-Beltrán, A., Martínez, R.** y **Jaén, J.A.** *Métodos Informáticos en TurboPascal*, Ed. Bellisco, 2ª edición, Madrid, 2002
- **Aho, A.H., Hopcroft, J.E. y Ullman, J.D.** *Estructuras de Datos y Algoritmos*, Addison-Wesley Iberoamericana, 1988
- **Collins, W.J.** *Data Structures: an Object-Oriented Approach*, Addison-Wesley Publishing Company Inc. 1992
- **Hale, G.J. y Easton, R.J.** *Applied Data Structures Using Pascal*, D.C. Heath and Company, 1987
- **Kruse, R.** *Estructuras de Datos y Diseño de Programas*, Prentice-Hall, 1988
- **Weiss, M.A.** *Data Structures and Algorithm Analysis*, The Benjaming-Cummings Publishing Company Inc, 1992
- **Wirth, N.** *Algoritmos + Estructuras de Datos = Programas*, Ediciones del Castillo, Madrid, 1986