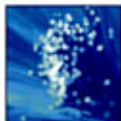


Segunda Edición

Libro de problemas

Fundamentos de programación I

Algoritmos, Estructuras de datos y Objetos I



**Mc
Graw
Hill**

Luis JOYANES AGUILAR

Las estructuras dinámicas de datos se pueden dividir en dos grandes grupos:

<i>lineales</i>	$\left\{ \begin{array}{l} \text{pilas} \\ \text{colas} \\ \text{listas enlazadas} \end{array} \right.$
<i>no lineales</i>	$\left\{ \begin{array}{l} \text{árboles} \\ \text{grafos} \end{array} \right.$

Las estructuras dinámicas de datos se utilizan para almacenamiento de datos del mundo real que están cambiando constantemente. Un ejemplo típico ya lo hemos visto como estructura estática de datos: la lista de pasajeros de una línea aérea. Si esta lista se mantuviera en orden alfabético en un array, sería necesario hacer espacio para insertar un nuevo pasajero por orden alfabético. Esto requiere utilizar un bucle para copiar los datos del registro de cada pasajero en el siguiente elemento del array. Si en su lugar se utilizara una estructura dinámica de datos, los nuevos datos del pasajero se pueden insertar simplemente entre dos registros existentes sin un mínimo esfuerzo.

Las estructuras dinámicas de datos son extremadamente flexibles. Como se ha descrito anteriormente, es relativamente fácil añadir nueva información creando un nuevo nodo e insertándolo entre nodos existentes. Se verá que es también relativamente fácil modificar estructuras dinámicas de datos, eliminando o borrando un nodo existente.

En este capítulo examinaremos las tres estructuras dinámicas lineales de datos: listas, colas y pilas, dejando para el próximo capítulo las estructuras no lineales de datos: árboles y grafos.

Una *estructura estática de datos* es aquella cuya estructura se especifica en el momento en que se escribe el programa y no puede ser modificada por el programa. Los valores de sus diferentes elementos pueden variar, pero no su estructura, ya que ésta es fija.

Una *estructura dinámica de datos* puede modificar su estructura mediante el programa. Puede ampliar o limitar su tamaño mientras se ejecuta el programa.

12.2. LISTAS

Una *lista lineal* es un conjunto de elementos de un tipo dado que pueden variar en número y donde cada elemento tiene un único predecesor y un único sucesor o siguiente, excepto el primero y último de la lista. Esta es una definición muy general que incluye los ficheros y vectores.

Los elementos de una lista lineal se almacenan normalmente contiguos —un elemento detrás de otro— en posiciones consecutivas de la memoria. Las sucesivas entradas en una guía o directorio telefónico, por ejemplo, están en líneas sucesivas, excepto en las partes superior e inferior de cada columna. Una lista lineal se almacena en la memoria principal de una computadora en posiciones sucesivas de memoria; cuando se almacenan en cinta magnética, los elementos sucesivos se presentan en sucesión en la cinta. Esta asignación de memoria se denomina *almacenamiento secuencial*. Posteriormente se verá que existe otro tipo de almacenamiento denominado *encadenado* o *enlazado*.

Las líneas así definidas se denominan *contiguas*. Las operaciones que se pueden realizar con listas lineales contiguas son:

1. Insertar, eliminar o localizar un elemento.
2. Determinar el tamaño —número de elementos— de la lista.
3. Recorrer la lista para localizar un determinado elemento.
4. Clasificar los elementos de la lista en orden ascendente o descendente.
5. Unir dos o más listas en una sola.

6. Dividir una lista en varias sublistas.
7. Copiar la lista.
8. Borrar la lista.

Una lista lineal contigua se almacena en la memoria de la computadora en posiciones sucesivas o adyacentes y se procesa como un array unidimensional. En este caso, el acceso a cualquier elemento de la lista y la adición de nuevos elementos es fácil; sin embargo, la inserción o borrado requiere un desplazamiento de lugar de los elementos que le siguen y, en consecuencia, el diseño de un algoritmo específico.

Para permitir operaciones con listas como arrays se deben dimensionar éstos con tamaño suficiente para que contengan todos los posibles elementos de la lista.

Ejemplo 12.1

Se desea leer el elemento j -ésimo de una lista P .

El algoritmo requiere conocer el número de elementos de la lista (su longitud, L). Los pasos a dar son:

1. **Conocer** longitud de la lista L .
2. **Si** $L = 0$ visualizar «error lista vacía».
Si_no comprobar si el elemento j -ésimo está dentro del rango permitido de elementos $1 \leq j \leq L$; en este caso, asignar el valor del elemento $P(j)$ a una variable B ; si el elemento j -ésimo no está dentro del rango, visualizar un mensaje de error «elemento solicitado no existe en la lista».
3. **Fin**.

El pseudocódigo correspondiente sería:

```

procedimiento acceso(E lista: P; S elementolista: B;
                    E entero: L, J)
inicio
    si  $L = 0$  entonces
        escribir('Lista vacía')
    si_no
        si  $(j \geq 1)$  y  $(j \leq L)$  entonces
             $B \leftarrow P[j]$ 
        si_no
            escribir('ERROR: elemento no existente')
        fin_si
    fin_si
fin
  
```

Ejemplo 12.2

Borrar un elemento j de la lista P .

Variables

L	longitud de la lista
J	posición del elemento a borrar
I	subíndice del array P
P	lista

Las operaciones necesarias son:

1. Comprobar si la lista es vacía.
2. Comprobar si el valor de J está en el rango del subíndice I de la lista $1 \leq J \leq L$.
3. En caso de J correcto, mover los elementos $j+1, j+2, \dots$, a las posiciones $j, j+1, \dots$, respectivamente, con lo que se habrá borrado el antiguo elemento j .
4. Decrementar en uno el valor de la variable L , ya que la lista contendrá ahora $L - 1$ elementos.

El algoritmo correspondiente será:

```

inicio
  si  $L = 0$  entonces
    escribir('lista vacia')
  si_no
    leer( $J$ )
    si  $(J \geq 1)$  y  $(J \leq L)$  entonces
      desde  $I \leftarrow J$  hasta  $L-1$  hacer
         $P[I] \leftarrow P[I+1]$ 
      fin_desde
       $L \leftarrow L-1$ 
    si_no
      escribir('Elemento no existe')
    fin_si
  fin_si
fin

```

listas

Una *lista contigua* es aquella en que los elementos son adyacentes en la memoria o soporte direccionable. Tiene unos límites izquierdo y derecho o inferior/superior que no pueden ser rebajados cuando se le añade un elemento.

La inserción o eliminación de un elemento, excepto en la cabecera o final de la lista, necesita una traslación de una parte de los elementos de la misma: la que precede o sigue a la posición del elemento modificado.

Las operaciones directas de añadir y eliminar se efectúan únicamente en los extremos de la lista. Esta limitación es una de las razones por las que esta estructura es poco utilizada.

Las *listas enlazadas* o de almacenamiento enlazado o encadenado son mucho más flexibles y potentes, y su uso es mucho más amplio que las listas contiguas.

12.3. LISTAS ENLAZADAS¹

Los inconvenientes de las listas contiguas se eliminan con las listas enlazadas. Se pueden almacenar los elementos de una lista lineal en posiciones de memoria que no sean contiguas o adyacentes.

Una *lista enlazada* o *encadenada* es un conjunto de elementos en los que cada elemento contiene la posición —o dirección— del siguiente elemento de la lista. Cada elemento de la lista enlazada debe

¹ Las listas enlazadas se conocen también en Latinoamérica con el término «ligadas» y «encadenadas». El término en inglés es *linked list*.

tener al menos dos campos: un campo que tiene el valor del elemento y un campo (enlace, *link*) que contiene la posición del siguiente elemento, es decir, su conexión, enlace o encadenamiento. Los elementos de una lista son enlazados por medio de los campos enlaces.

Las listas enlazadas tienen una terminología propia que se suele utilizar normalmente. Primero, los valores se almacenan en un *nodo* (Figura 12.1).

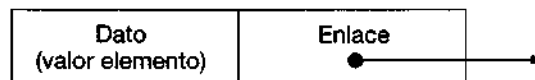
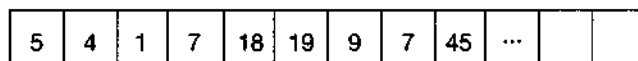
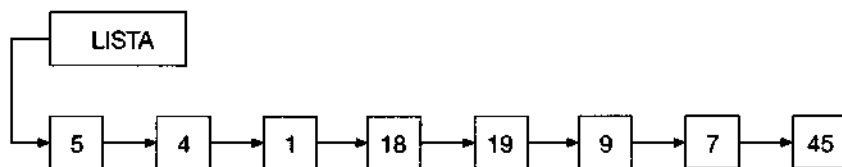


Figura 12.1. Nodo con dos campos.

Una lista enlazada se muestra en la Figura 12.2.



(a)



(b)

Figura 12.2. (a) array representado por una lista; (b) lista enlazada representada por una lista de enteros.

Los componentes de un nodo se llaman *campos*. Un nodo tiene al menos un campo *dato* o *valor* y un *enlace* (indicador o puntero) con el siguiente nodo. El campo enlace apunta (proporciona la dirección o referencia de) al siguiente nodo de la lista. El último nodo de la lista enlazada, por convenio, se suele representar por un enlace con la palabra reservada *nil* (*nulo*), una barra inclinada (/) y, en ocasiones, el símbolo eléctrico de tierra o masa (Figura 12.3).

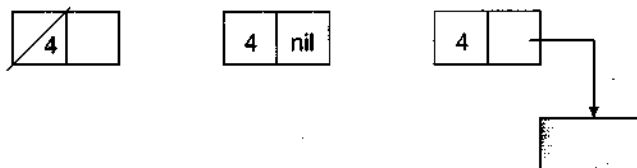


Figura 12.3. Representación del último nodo de una lista.

La implementación de una lista enlazada depende del lenguaje. C, C++, Pascal, PL/I, Ada y Modula-2 utilizan como enlace una *variable puntero*, o *puntero* (*apuntador*) simplemente. Java no dispone de punteros, por consiguiente, resuelve el problema de forma diferente y almacena en el enlace la referencia al siguiente objeto nodo. Los lenguajes como FORTRAN y COBOL no disponen de este tipo de datos y se debe simular con una variable entera que actúa como indicador o cursor. En nues-

tro libro utilizaremos a partir de ahora el término *puntero* (**apuntador**) para describir el enlace entre dos elementos o nodos de una lista enlazada.

Un *puntero* (**apuntador**) es una variable cuyo valor es la dirección o posición de otra variable.

En las listas enlazadas no es necesario que los elementos de la lista sean almacenados en posiciones físicas adyacentes, ya que el puntero indica dónde se encuentra el siguiente elemento de la lista, tal como se indica en la Figura 12.4.

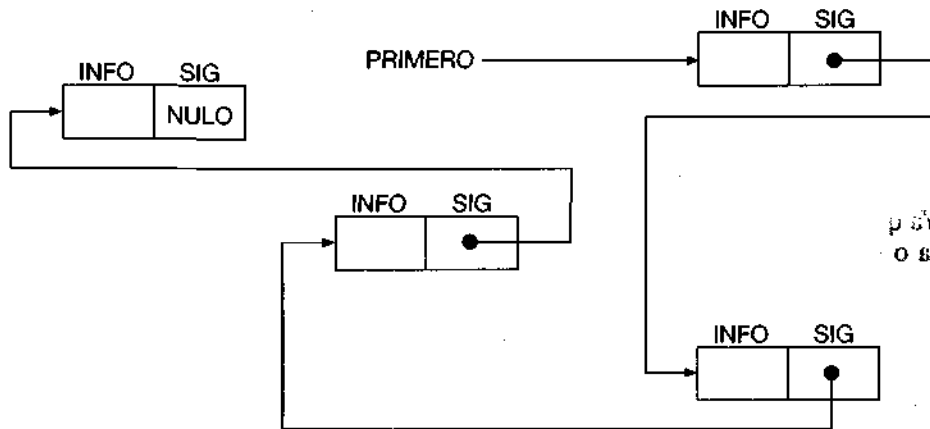


Figura 12.4. Elementos no adyacentes de una lista enlazada.

Por consiguiente, la inserción y borrado no exigen desplazamiento como en el caso de las listas contiguas.

Para eliminar el 45.º elemento ('INÉS') de una lista lineal con 2.500 elementos [Figura 12.5 (a)] sólo es necesario cambiar el puntero en el elemento anterior, 44.º, y que apunte ahora al elemento 46.º [Figura 12.5 (b)]. Para insertar un nuevo elemento ('HIGINIO') después del 43.º elemento, es necesario cambiar el puntero del elemento 43.º y hacer que el nuevo elemento apunte al elemento 44.º [Figura 12.5 (c)].

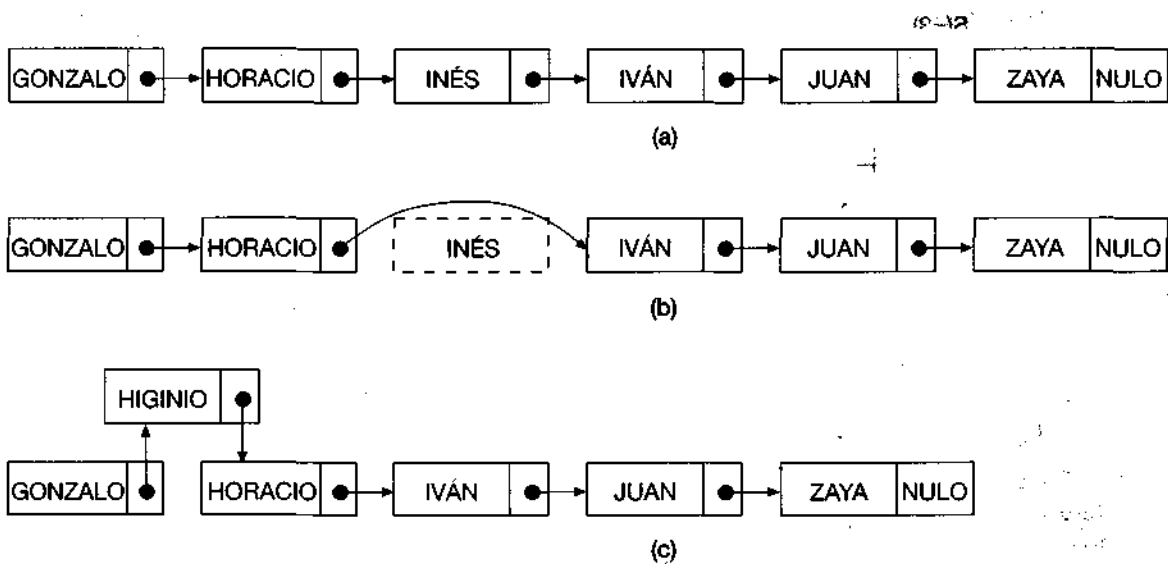


Figura 12.5. Inserción y borrado de elementos.

Una lista enlazada sin ningún elemento se llama *lista vacía*. Su puntero inicial o de cabecera tiene el valor nulo (*nil*).

Una lista enlazada se define por:

- El tipo de sus elementos: campo de información (datos) y campo enlace (*puntero*).
- Un puntero de cabecera que permite acceder al primer elemento de la lista.
- Un medio para detectar el último elemento de la lista: *puntero nulo* (*nil*).

Ejemplo 12.3

El director de un hotel desea registrar el nombre de cada cliente a medida de su llegada al hotel, junto con el número de habitación que ocupa —el antiguo libro de entradas—. También desea disponer en cualquier momento de una lista de sus clientes por orden alfabético.

Ya que no es posible registrar los clientes alfabéticamente y cronológicamente en la misma lista, se necesita o bien listas alfabéticas independientes o bien añadir punteros a la lista existente, con lo que sólo se utilizará una única lista. El método manual en el libro requería muchos cruces y reescrituras; sin embargo, una computadora mediante un algoritmo adecuado lo realizará fácilmente.

Por cada nodo de la lista el campo de información o datos tiene dos partes: nombre del cliente y número de habitación. Si x es un puntero a uno de estos nodos, $l[x].nombre$ y $l[x].habitación$ representarán las dos partes del campo información.

El listado alfabético se consigue siguiendo el orden de los punteros de la lista (campo puntero). Se utiliza una variable CABECERA (S) para apuntar al primer cliente.

CABECERA $\leftarrow 3$

Así, CABECERA (S) es 3, ya que el primer cliente, Antolín, ocupa el lugar 3. A su vez, el puntero asociado al nodo ocupado por Antolín contiene el valor 10, que es el segundo nombre de los clientes en orden alfabético y éste tiene como campo puntero el valor 7, y así sucesivamente. El campo puntero del último cliente, Tomás, contiene el puntero nulo indicado por un 0 o bien una Z .

$S(=3)$

Registro	Nombre	Habitación	Puntero
1	Tomás	324	z (final)
2	Cazorla	28	8
3	Antolín	95	10
4	Pérez	462	6
5	López	260	12
6	Sánchez	220	1
7	Bautista	115	2
8	García	105	9
9	Jiménez	173	5
10	Apolinar	341	7
11	Martín	205	4
12	Luzárraga	420	11
.	.	.	.
.	.	.	.

Figura 12.6. Lista enlazada de clientes de un hotel.

12.4. PROCESAMIENTO DE LISTAS ENLAZADAS

Para procesar una lista enlazada se necesitan las siguientes informaciones:

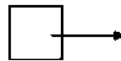
- Primer nodo (cabecera de la lista).
- El tipo de sus elementos.

Las operaciones que normalmente se ejecutan con listas incluyen:

1. Recuperar información de un nodo específico (acceso a un elemento).
2. Encontrar el nodo que contiene una información específica (localizar la posición de un elemento dado).
3. Insertar un nuevo nodo en un lugar específico de la lista.
4. Insertar un nuevo nodo en relación a una información particular.
5. Borrar (eliminar) un nodo existente que contiene información específica.

12.4.1. Implementación de listas enlazadas con punteros

Como ya hemos visto la representación gráfica de un puntero consiste en una flecha que sale del puntero y llega a la variable dinámica apuntada.



Para declarar una variable de tipo puntero:

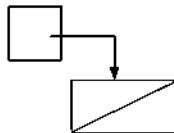
```
tipo
    puntero_a <tipo_dato>: punt
var
    punt : p, q
```

El <tipo_dato> podrá ser simple o estructurado.

Operaciones con punteros:

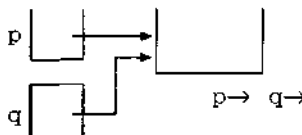
Inicialización

$p \leftarrow \text{nulo}$ A nulo para indicar que no apunta a ninguna variable.



Comparación

$p = q$ Con los operadores $=$ o $<>$.



Asignación

$p \leftarrow q$ Implica hacer que el puntero p apunte a donde apuntaba q .

Creación de variables dinámicas

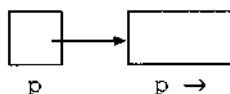
Reservar(p) Reservar espacio en memoria para la variable dinámica.

Eliminación de variables dinámicas

Liberar(p) Liberar el espacio en memoria ocupado por la variable dinámica.

Variables dinámicas

Variable simple o estructura de datos sin nombre y creada en tiempo de ejecución.



Para acceder a una variable dinámica apuntada, como no tiene nombre $p \rightarrow$

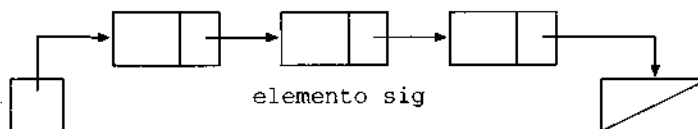
Las variables $p \rightarrow$ podrán intervenir en toda operación o expresión de las permitidas para una variable estática de su mismo tipo.

Nodo

Las estructuras dinámicas de datos están formadas por nodos.

Un nodo es una variable dinámica constituida por al menos dos campos:

- el campo dato o valor (elemento);
- el campo enlace, en este caso de tipo puntero (sig).



tipo

registro: nodo

//elemento es el campo que contiene la información

<tipo_elemento>: elemento

//punt apunta al siguiente elemento de la estructura

punt : sig

{según la estructura de que se trate podrá haber uno o varios campos de tipo punt}

... : ...

fin_registro

Creación de la lista

La creación de la lista conlleva la inicialización a nulo del puntero (inic), que apunta al primer elemento de la lista.

```

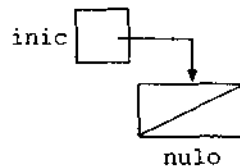
tipo
  puntero_a_nodo: punt
  registro: tipo_elemento
  ... : ...
  fin_registro
  registro: nodo
  tipo_elemento : elemento
  punt          : sig
  fin_registro

var  punt          : inic, posic, anterior
     tipo_elemento : elemento
     logico         : encontrado

inicio
  inicializar(inic)
  ...
fin

procedimiento inicializar(S punt: inic)
  inicio
    inic ← nulo
  fin_procedimiento

```



Inserción de un elemento

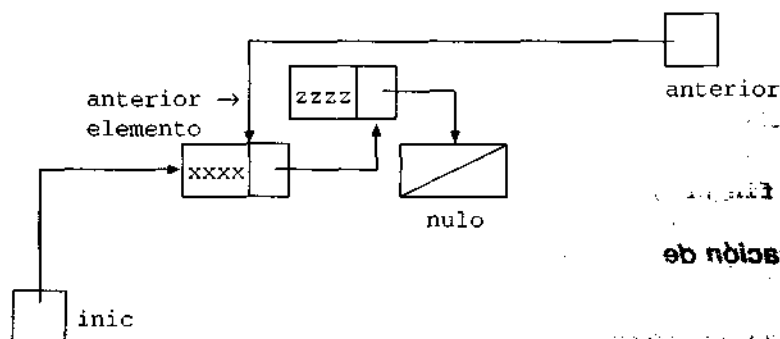
La inserción tiene dos casos particulares:

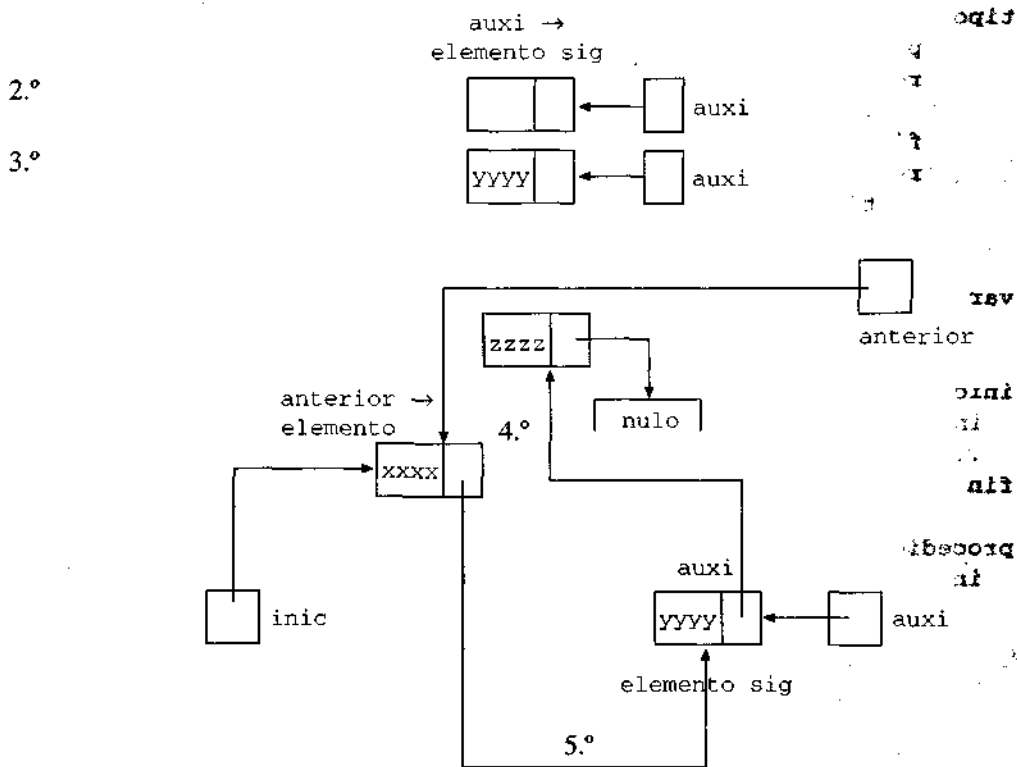
- Insertar el nuevo nodo en el frente, principio de la lista.
- Insertar el nuevo nodo en cualquier otro lugar de la lista.

El procedimiento insertar inserta un nuevo elemento a continuación de anterior, si anterior fuera nulo significa que ha de insertarse al comienzo de la lista.

insertar(inicio, anterior, elemento)

1.º





- 1.º Situación de partida.
- 2.º reservar(auxí).
- 3.º Introducir la nueva información en auxí → elemento.
- 4.º Hacer que auxí → sig apunte a donde lo hacía anterior → sig.
- 5.º Conseguir que anterior → sig apunte a donde lo hace auxí.

procedimiento insertar(E/S punt: inic,anterior;
E tipo_elemento: elemento)

```

var punt: auxi
inicio
  reservar(auxí)
  auxí→.elemento ← elemento
  si anterior = nulo entonces
    auxí →.sig ← inic
    inic ← auxí
  si_no
    auxí→.sig ← anterior→.sig
    anterior→.sig ← auxí
  fin_si
  anterior ← auxí // Opcional
fin_procedimiento
  
```

Eliminación de un elemento de una lista enlazada

Antes de proceder a la eliminación de un elemento de la lista, deberemos comprobar que no está vacía. Para lo que podremos recurrir a la función vacía.

```

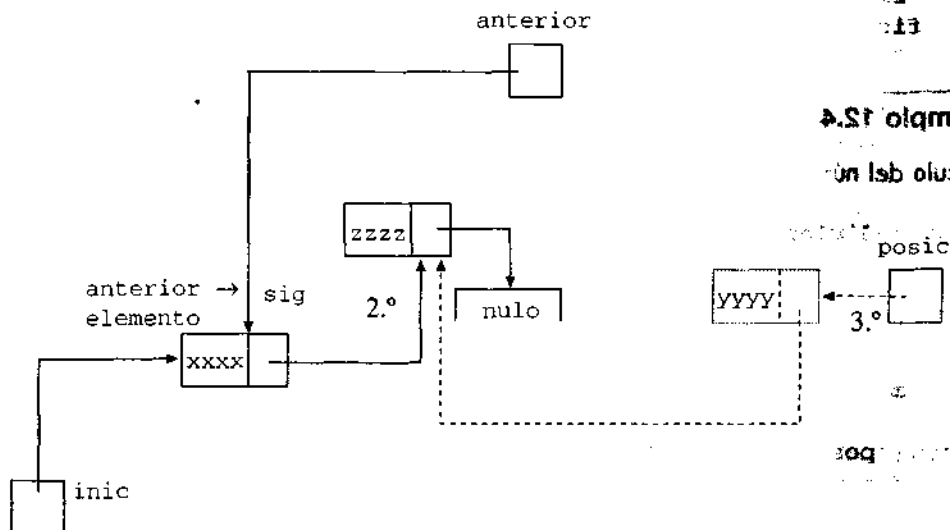
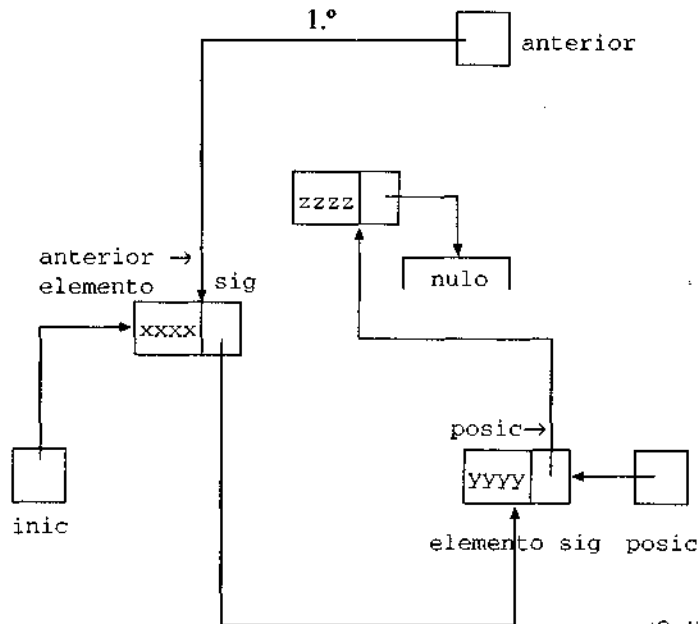
lógico función vacia(E punt: inic)
  inicio
    devolver(inic = nulo)
  fin_función

```

Al suprimir un elemento de una lista consideraremos dos casos particulares:

- El elemento a suprimir está al principio de la lista.
- El elemento se encuentra cualquier otro lugar de la lista.

Suprimir(inic, anterior, posic).



- 1.º Situación de partida
- 2.º anterior →.sig apunta a donde posic →.sig.
- 3.º liberar(posic).

```

procedimiento suprimir(E/S punt: inic, anterior, posic)
inicio
  si anterior = nulo entonces
    inic ← posic→.sig
  si_no
    anterior→.sig ← posic→.sig
  fin_si
  liberar(posic)
  anterior ← nulo // Opcional
  posic ← inic // Opcional
fin_procedimiento

```

Java y C# permiten la creación de listas enlazadas vinculando objetos nodo sin el empleo de punteros, en el enlace se almacena la referencia al siguiente nodo de la lista. En estos lenguajes, aunque la implementación de una lista enlazada resulta similar, hay que tener en cuenta que al crear un objeto nodo se reserva espacio en memoria para él y que este espacio se libera automáticamente, a través de un proceso denominado recolección automática de basura, cuando dicho objeto (nodo) deja de estar referenciado.

Recorrido de una lista enlazada

Para recorrer la lista utilizaremos una variable de tipo puntero auxiliar.

```

procedimiento recorrer(E punt: inic)
  var punt: posic
  inicio
    posic ← inic
    mientras posic <> nulo hacer
      proc_escribir(posic→.elemento)
      posic ← posic→.sig
    fin_mientras
  fin_procedimiento

```

Ejemplo 12.4

Cálculo del número de elementos de una lista enlazada.

```

procedimiento contar(E punt: primero; S entero: n)
  var punt: p
  inicio
    n ← 0 //contador de elementos
    p ← primero
    mientras p <> nulo hacer
      n ← n + 1
      posic ← posic →.sig
    fin_mientras
  fin_procedimiento

```

Acceso a un elemento de una lista enlazada

La búsqueda de una información en una lista simplemente enlazada sólo puede hacerse mediante un proceso secuencial o recorrido de la lista elemento a elemento, hasta encontrar la información buscada o detectar el final de la lista.

```

procedimiento consultar(E punt: inic; S punt: posic, anterior;
                        E tipo_elemento: elemento; S lógico: encontrado)
inicio
  anterior ← nulo
  posic ← inic
  mientras no igual(posic→.elemento, elemento) y (posic <> nulo) hacer
    { igual es una función que compara los elementos que le pasamos como
      parámetros, recurrimos a ella porque, si se tratara de registros,
      compararíamos únicamente la información almacenada en un
      determinado campo }
    anterior ← posic
    posic ← posic→.sig
  fin_mientras
  si igual(posic→.elemento, elemento) entonces
    encontrado ← verdad
  si_no
    encontrado ← falso
  fin_si
fin_procedimiento

```

Ejemplo 12.5

Encontrar el nodo de una lista que contiene la información de valor *t*, suponiendo que la lista almacena datos de tipo entero.

```

procedimiento encontrar(E punt: primero E entero: t)
  var punt : p
      entero: n
  inicio
    n ← 0
    p ← primero
    mientras (p→.info <> t) y (p <> nulo) hacer
      n ← n + 1
      posic ← posic→.sig
    fin_mientras
    si p→.info = t entonces
      escribir('Se encuentra en el nodo ', n, 'de la lista')
    si_no
      escribir('No encontrado')
    fin_si
  fin_procedimiento

```

Considere que la información se encuentra almacenada en la lista de forma ordenada, orden creciente, y mejore la eficacia del algoritmo anterior.

```

procedimiento encontrar(E punt: primero E entero: t)
var punt : p
    entero: n
inicio
    n ← 0
    p ← primero
    mientras (p →.info < t) y (p <> nulo) hacer
        n ← n + 1
        posic ← posic →.sig
    fin_mientras
    si p →.info = t entonces
        escribir('Se encuentra en el nodo ', n, ' de la lista')
    si_no
        escribir('No encontrado')
    fin_si
fin_procedimiento

```

12.4.2. Implementación de listas enlazadas con arrays

Las listas enlazadas deberán implementarse de forma dinámica, pero si el lenguaje no lo permite, lo realizaremos a través de arrays, con lo cual impondremos limitaciones en cuanto al número de elementos que podrá contener la lista y estableceremos una ocupación en memoria constante.

Los nodos podrán almacenarse en arrays paralelos o arrays de registros.

Cuando se empleen arrays de registros, el valor (dato o información) del nodo se almacenará en un campo y el enlace con el siguiente elemento se almacenará en otro.

Otra posible implementación, como ya se ha dicho antes, es con dos arrays: uno para los datos y otro para el enlace.

Un valor de puntero 0, o bien Z, indica el final de la lista.

ELEMENTO		SIG
XXXXXXXXXXXXXX	1	2
XXXXXXXXXXXXXX	2	4
	3	
XXXXXXXXXXXXXX	4	6
	5	
XXXXXXXXXXXXXX	6	0

	ELEMENTO	SIG
1	XXXXXXXXXXXXXX	2
2	XXXXXXXXXXXXXX	4
3		
4	XXXXXXXXXXXXXX	6
5		
6	XXXXXXXXXXXXXX	0

Para definir la lista se debe especificar la variable que apunta al primer nodo (cabecera), que en nuestro caso denominaremos inic.

inic ← 1

fin

Para insertar un nuevo elemento, que siga a m[1] y sea seguido por m[2], lo único que se hará es modificar los punteros.

M

1	XXXXXXXXXXXXXX	3
2	XXXXXXXXXXXXXX	4
3	XXXXXXXXXXXXXX	2
4	XXXXXXXXXXXXXX	6
5		
6	XXXXXXXXXXXXXX	0

Como el nuevo elemento se coloca en la primera posición libre deberemos tener un puntero vacío que apunte a dicha primera posición libre.

Es decir, utilizaremos el array para almacenar dos listas, la lista de elementos y la lista de vacíos.

Es, pues, necesario, al comenzar a trabajar, crear la lista de vacíos de la forma que a continuación se expone:

- Vacío apunta al primer registro libre.
- En el campo sig de cada registro se almacena información sobre el siguiente registro disponible.
- Cuando lleguemos al último registro libre, su campo sig recibirá el valor 0, para indicar que ya no quedan más registros disponibles.

vacio \leftarrow 1
inic \leftarrow 0

	ELEMENTO	SIG
1		2
2		3
3		4
4		5
5		6
6		0

Insertar el primer elemento

vacio \leftarrow 2
inic \leftarrow 1

1	XXXXXXXXXXXXXX	0
2		3
3		4
4		5
5		6
6		0

Al implementar una lista a través de arrays necesitaremos los procedimientos:

inicializar(...) iniciar(...) consultar(...) insertar(...) suprimir(...)
reservar(...) liberar(...)

y las funciones

vacía(...) llena(...)

El procedimiento `reservar(...)` nos proporcionará la primera posición vacía para almacenar un nuevo elemento y la eliminará de la lista de vacíos, pasando el puntero de vacíos (`vacio`) a la posición siguiente, `vacio` toma el valor del siguiente `vacio` de la lista.

`Liberar(...)` inserta un nuevo elemento en la lista de vacíos. Se podrían adoptar otras soluciones, pero nuestro procedimiento `liberar` insertará el nuevo elemento en la lista de vacíos por delante, sobrescribiendo el campo `m[posic].sig` para que apunte al que antes era el primer `vacio`. El puntero de inicio de los vacíos (`vacio`) lo cambiará al nuevo elemento.

Creación de la lista

Consideraremos el array como si fuera la memoria del ordenador y guardaremos en él dos listas: la lista de elementos y la de vacíos.

El primer elemento de la lista de elementos está apuntado por `inic` y, por `vacio`, el primero de la lista de vacíos:

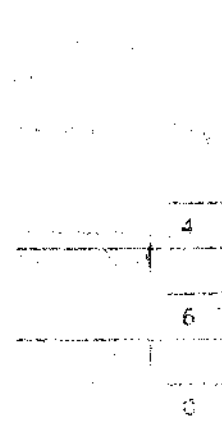
```
const
  max = <expresión>
tipo
  registro: tipo_elemento
  ... : ...
  ... : ...
fin_registro
registro: tipo_nodo
  tipo_elemento : elemento
  entero        : sig
  { actúa como puntero, almacenando la posición donde se
    encuentra el siguiente elemento de una lista }
fin_registro
array[1..Max] de tipo_nodo: arr
var
  entero      : inic,
               posic,
               anterior,
               vacio
  arr         : m
  //m representa la memoria de nuestro ordenador
  tipo_elemento : elemento
  logico       : encontrado
inicio
  iniciar(m, vacio)
  inicializar(inic)
  ...
fin
```

Al comenzar:

```
procedimiento inicializar(S entero: inic) //lista de elementos
inicio
  inic ← 0
fin_procedimiento
```

El primer elemento de la lista de elementos está apuntado por `inic` y, por `vacio`, el primero de la lista de vacíos:

El primer elemento de la lista de elementos está apuntado por `inic` y, por `vacio`, el primero de la lista de vacíos:



El primer elemento de la lista de elementos está apuntado por `inic` y, por `vacio`, el primero de la lista de vacíos:

El primer elemento de la lista de elementos está apuntado por `inic` y, por `vacio`, el primero de la lista de vacíos:

vacio $\leftarrow 1$ indica que el primer registro libre es m[1]

inic $\leftarrow 0$ inic señala que no hay elementos en la lista

	elemento	sig
1		2
2		3
3		4
4		5
5		6
6		0

```

procedimiento iniciar(S arr: m; S entero: vacio) //lista de vacíos
var
    entero: i
inicio
    vacio  $\leftarrow 1$ 
    desde i  $\leftarrow 1$  hasta Max-1 hacer
        m[i].sig  $\leftarrow i+1$ 
    fin_desde
    m[Max].sig  $\leftarrow 0$ 
    //Como ya no hay más posiciones libres a las que apuntar, recibe un 0
fin_procedimiento

```

Al trabajar de esta manera conseguiremos que la inserción o borrado de un determinado elemento, n -ésimo, de la lista no requiera el desplazamiento de otros.

Inserción de un elemento

Al actualizar una lista se pueden presentar dos casos particulares:

- Desbordamiento (*overflow*).
- Subdesbordamiento o desbordamiento negativo (*underflow*).

El desbordamiento se produce cuando la lista está llena y la lista de espacio disponible está vacía.

El subdesbordamiento se produce cuando se tiene una lista vacía y se desea borrar un elemento de la misma.

Luego, para poder insertar un nuevo elemento en una lista enlazada, es necesario comprobar que se dispone de espacio libre para ello. Al insertar un nuevo elemento en la lista deberemos recurrir al procedimiento `reservar(...)` que nos proporcionará, a través de `auxi`, la primera posición vacía para almacenar en ella el nuevo elemento, eliminando dicha posición de la lista de vacíos.

Por ejemplo, al insertar el primer elemento:

auxi $\leftarrow 1$
vacio $\leftarrow 2$

	elemento	sig
1	xxxxxxxxxxxxx	2 0
2		3
3		4
4		5
5		6
6		0

...), que se
no en la lista

ón es 3

- vacío señala que la primera posición libre es la 1 $\text{auxi} \leftarrow 1$
- El campo sig del registro $m[\text{vacío}]$ proporciona la siguiente posición vacía y reservar hará que vacío apunte a esta nueva posición $\text{vacío} \leftarrow 2$

Al insertar un segundo elemento:

- como vacío tiene el valor 2 $\text{auxi} \leftarrow 2$
- $\text{vacío} \leftarrow m[2].\text{sig}$, es decir $\text{vacío} \leftarrow 3$

$\text{auxi} \leftarrow 2$
 $\text{vacío} \leftarrow 3$

	elemento	sig
1	xxxxxxxxxxxxx	2 0
2	xxxxxxxxxxxxx	3 0
3		4
4		5
5		6
6		0

procedimiento reservar(**S** entero: auxi; **E** arr: m; **E/S** entero: vacío)

```

inicio
  si vacío = 0 entonces
    // Memoria agotada
    auxi ← 0
  si_no
    auxi ← vacío
    vacío ← m[vacío].sig
  fin_si
fin_procedimiento

```

El procedimiento insertar colocará un nuevo elemento a continuación de anterior, si anterior fuera 0 significa que ha de insertarse al comienzo de la lista.

procedimiento insertar(**E/S** entero: inic, anterior;
E tipo_elemento: elemento;
E/S arr: m ; **E/S** entero: vacío)

```

var
  entero: auxi
inicio
  reservar(auxi,m,vacío)
  m[auxi].elemento ← elemento
  si anterior = 0 entonces
    m[auxi].sig ← inic
    inic ← auxi
  si_no
    m[auxi].sig ← m[anterior].sig
    m[anterior].sig ← auxi
  fin_si
  anterior ← auxi // Opcional
  { Prepara anterior para que, si no especificamos otra cosa, la
    siguiente inserción se realice a continuación de la actual}
fin_procedimiento

```

Consideremos la siguiente situación y analicemos el comportamiento que en ella tendrían los procedimientos reservar e insertar: *Se desea insertar un nuevo elemento en la lista a continuación del primero y la situación actual, tras sucesivas inserciones y eliminaciones, es como se muestra a continuación:*

vacio \leftarrow 3
inic \leftarrow 1

1	xxxxxxxxxxxxxx	2
2	xxxxxxxxxxxxxx	4
3		5
4	xxxxxxxxxxxxxx	6
5		0
6	xxxxxxxxxxxxxx	0

el señalado

la lista va mos

El nuevo elemento se colocará en el array en la primera posición libre y lo único que se hará es modificar los punteros.

reservar(...) proporciona la primera posición libre

auxi \leftarrow 3

vacio \leftarrow 5

1	xxxxxxxxxxxxxx	3
2	xxxxxxxxxxxxxx	4
3	nuevo_elemento	2
4	xxxxxxxxxxxxxx	6
5		0
6	xxxxxxxxxxxxxx	0

m[3].elemento \leftarrow nuevo_elemento

como queremos insertar el nuevo elemento a continuación del primero de la lista, su anterior será el apuntado por inic

anterior \leftarrow 1
m[3].sig \leftarrow 2
m[1].sig \leftarrow 3

Eliminación de un elemento

Para eliminar un elemento de la lista deberemos recurrir al procedimiento suprimir(...), que, a su vez, llamará al procedimiento liberar(...) para que inserte el elemento eliminado en la lista de vacíos.

Supongamos que se trata de eliminar el elemento marcado con ***** cuya posición es 3

posic \leftarrow 3

el elemento anterior al 3 ocupa en el array la posición 2

anterior \leftarrow 2

y el primer vacío está en 5. Siendo el aspecto actual de la lista el siguiente:

```

inic ← 1
anterior ← 2
posic ← 3

vacio ← 5

```

	elemento	sig
1	XXXXXXXXXXXXXX	2
2	XXXXXXXXXXXXXX	3
3	XXXXXXXXXXXXXX	4
4	XXXXXXXXXXXXXX	0
5		6
6		0

Al suprimir el elemento 3 la lista quedaría:

```
m[2].sig ← 4
```

mediante el procedimiento liberar(...) incluimos el nuevo vacío en la lista de vacíos

```
m[3].sig ← 5      vacio ← 3
```

como el que se suprime no es el primer elemento de la lista, el valor de inic no varía

```
inic ← 1
```

	elemento	sig
1	XXXXXXXXXXXXXX	2
2	XXXXXXXXXXXXXX	4
3	XXXXXXXXXXXXXX	5
4	XXXXXXXXXXXXXX	0
5		6
6		0

```
procedimiento liberar(E entero: posic; E/S arr: m; E/S entero: vacio)
```

```
  inicio
```

```
    m[posic].sig ← vacio
```

```
    vacio ← posic
```

```
  fin_procedimiento
```

```
procedimiento suprimir(E/S entero: inic, anterior, posic; E/S arr: m;
                      E/S entero: vacio)
```

```
  inicio
```

```
    si anterior = 0 entonces
```

```
      inic ← m[posic].sig
```

```
    si_no
```

```
      m[anterior].sig ← m[posic].sig
```

```
  fin_si
```

```

liberar(posic, m, vacio)
anterior ← 0 // Opcional
posic ← inic // Opcional
{ Las dos últimas instrucciones preparan los punteros para que,
  si no se especifica otra cosa, la próxima eliminación se realice
  por el principio de la lista }
fin_procedimiento

```

Recorrido de una lista

El recorrido de la lista se realizará siguiendo los punteros a partir de su primer elemento, el señalado por inic.

El procedimiento recorrer (...) que se implementa a continuación, al recorrer la lista va mostrando por pantalla los diferentes elementos que la componen.

```

procedimiento recorrer(E entero: inic)
var entero: posic
inicio
  posic ← inic
  mientras posic <> 0 hacer
    { Recurrimos a un procedimiento, proc_escribir(...),
      para presentar por pantalla los campos del registro
      pasado como parámetro }
    proc_escribir(m[posic].elemento)
    posic ← m[posic].sig
  fin_mientras
fin_procedimiento

```

Búsqueda de un determinado elemento en una lista

El procedimiento consultar informará sobre si un determinado elemento se encuentra o no en la lista, la posición que ocupa dicho elemento en el array y la que ocupa el elemento anterior. Si la información se encontrara colocada en la lista de forma ordenada y creciente por el campo de búsqueda el procedimiento de consulta podría ser el siguiente:

```

procedimiento consultar(E entero: inic; S entero: posic, anterior;
                        E tipo_elemento: elemento; S lógico: encontrado;
                        E arr: m)
inicio
  anterior ← 0
  posic ← inic
  { Las funciones menor(...) e igual(...) comparan los registros por
    un determinado campo }
  mientras menor(m[posic].elemento, elemento) y (posic <> 0) hacer
    anterior ← posic
    posic ← m[posic].sig
  fin_mientras
  si igual(m[posic].elemento, elemento) entonces
    encontrado ← verdad
  si_no
    encontrado ← falso
  fin_si
fin_procedimiento

```

Funciones

Cuando implementamos una lista enlazada utilizando arrays, necesitamos las siguientes funciones:

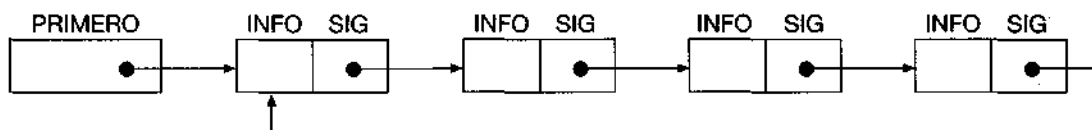
```

lógico función vacia(E entero: inic)
    inicio
        devolver(inic = 0)
    fin_función

lógico función llena(E entero: vacio)
    inicio
        devolver(vacio = 0)
    fin_función
  
```

12.5. LISTAS CIRCULARES

Las listas simplemente enlazadas no permiten a partir de un elemento acceder directamente a cualquiera de los elementos que le preceden. En lugar de almacenar un puntero NULO en el campo SIG del último elemento de la lista, se hace que el último elemento apunte al primero o principio de la lista. Este tipo de estructura se llama *lista enlazada circular* o simplemente *lista circular* (en algunos textos se les denomina listas en anillo).



Las listas circulares presentan las siguientes *ventajas* respecto de las listas enlazadas simples:

- Cada nodo de una lista circular es accesible desde cualquier otro nodo de ella. Es decir, dado un nodo se puede recorrer toda la lista completa. En una lista enlazada de forma simple sólo es posible recorrerla por completo si se parte de su primer nodo.
- Las operaciones de concatenación y división de listas son más eficaces con listas circulares.

Los *inconvenientes*, por el contrario, son:

- Se pueden producir lazos o bucles infinitos. Una forma de evitar estos bucles infinitos es disponer de un nodo especial que se encuentre permanentemente asociado a la existencia de la lista circular. Este nodo se denomina *cabecera* de la lista.

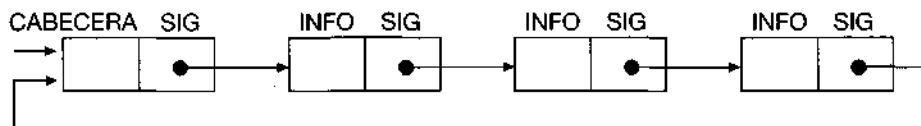


Figura 12.7. Nodo cabecera de la lista.

El nodo cabecera puede diferenciarse de los otros nodos en una de las dos formas siguientes:

- Puede tener un valor especial en su campo INFO que no es válido como datos de otros elementos.
- Puede tener un indicador o bandera (*flag*) que señale cuando es nodo cabecera.

El campo de la información del nodo cabecera no se utiliza, lo que se señala con el sombreado de dicho campo.

Una lista enlazada circularmente *vacía* se representa como se muestra en la Figura 12.8.

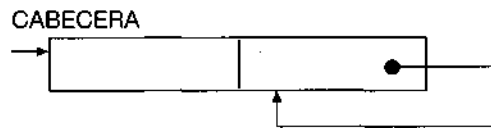


Figura 12.8. Lista circular vacía.

12.6. LISTAS DOBLEMENTE ENLAZADAS

En las listas lineales estudiadas anteriormente el recorrido de ellas sólo podía hacerse en un único sentido: *de izquierda a derecha* (principio a final). En numerosas ocasiones se necesita recorrer las listas en ambas direcciones.

Las listas que pueden recorrerse en ambas direcciones se denominan *listas doblemente enlazadas*. En estas listas cada nodo consta del campo INFO de datos y dos campos de enlace o punteros: ANTERIOR (ANT) y SIGUIENTE (SIG) que apuntan hacia adelante y hacia atrás (Fig. 12.9). Como cada elemento tiene dos punteros, una lista doblemente enlazada ocupa más espacio en memoria que una lista simplemente enlazada para una misma cantidad de información.

La lista necesita dos punteros CABECERA y FIN² que apuntan hacia el primero y último nodo.

La variable CABECERA y el puntero SIG permiten recorrer la lista en el sentido normal y la variable FIN y el puntero ANT permiten recorrerla en sentido inverso.

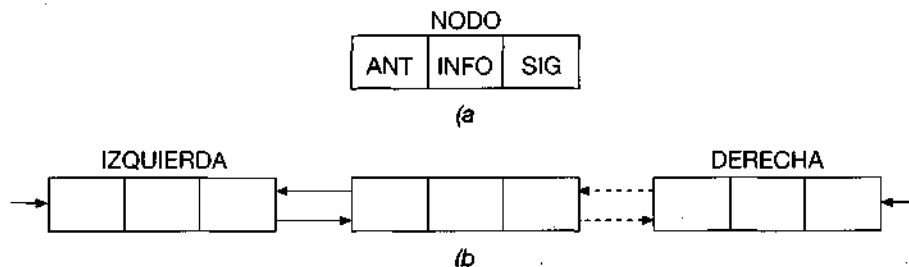


Figura 12.9. Lista doblemente enlazada.

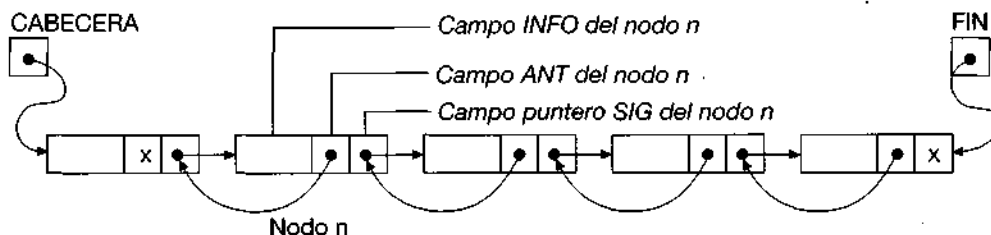


Figura 12.10. Lista doble.

Como se ve en la Figura 12.11, una propiedad fundamental de las listas doblemente enlazadas es que para cualquier puntero P de la lista:

² Se adoptan estos términos a efectos de normalización, pero el lector puede utilizar IZQUIERDA y DERECHA.


```
nodo [nodo[p]. sig].ant = p
nodo [nodo[p]. ant].sig = p
```

El campo de l

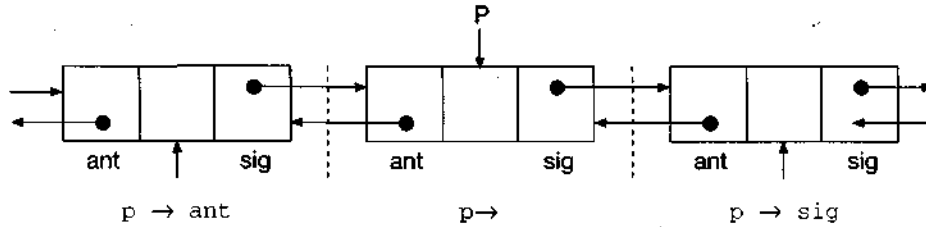


Figura 12.11.

LISTAS DOBLES

12.6.1. Inserción

La inserción de un nodo a la derecha de un nodo especificado, cuya dirección está dada por la variable M , puede presentar varios casos:

1. La lista está vacía; se indica mediante $M = \text{NULO}$ y CABECERA y FIN son también NULO. Una inserción indica que CABECERA se debe fijar con la dirección del nuevo nodo y los campos ANT y SIG también se establecen en NULO.
2. Insertar dentro de la lista: existe un elemento anterior y otro posterior de X .
3. Insertar a la derecha del nodo de la extrema derecha de la lista. Se requiere que el apuntador FIN sea modificado.

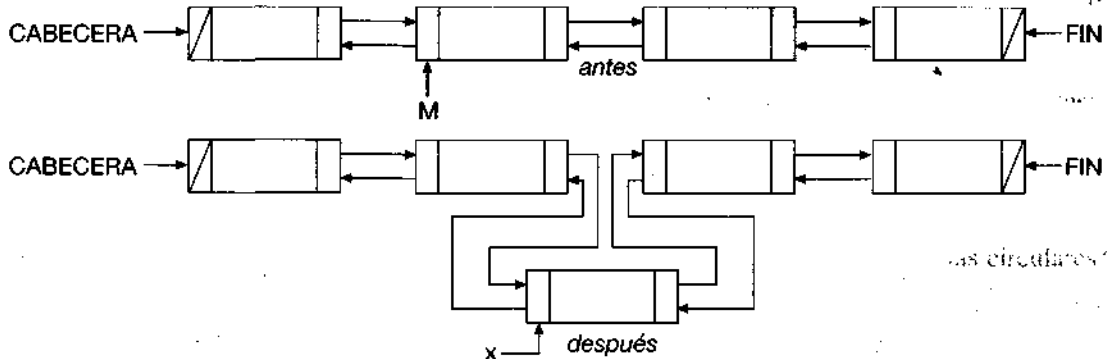


Figura 12.12. Inserción en una lista doblemente enlazada.

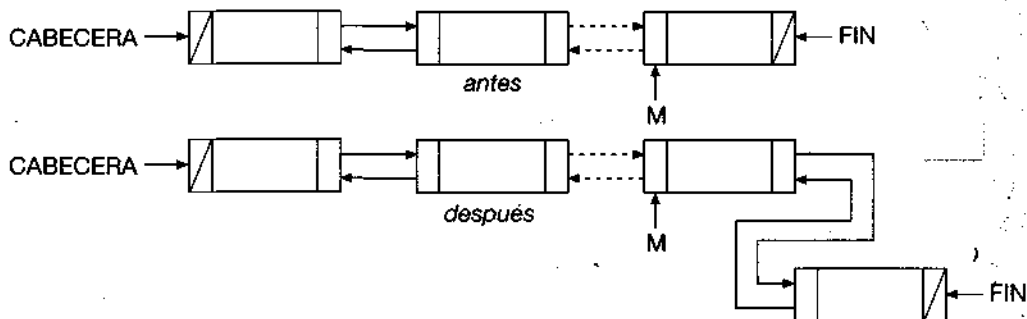


Figura 12.13. Inserción en el extremo derecho de una lista doblemente enlazada.

12.6.2. Eliminación

La operación de eliminación es directa.

Si la lista tiene un simple nodo, entonces los punteros de los extremos izquierdo y derecho asociados a la lista se deben fijar en NULO. Si el nodo del extremo derecha de la lista es el señalado para la eliminación, la variable FIN debe modificarse para señalar el predecesor del nodo que se va a borrar de la lista. Si el nodo del extremo izquierdo de la lista es el que se desea borrar, la variable CABECERA debe modificarse para señalar el elemento siguiente.

La eliminación se puede realizar dentro de la lista (Figura 12.14).

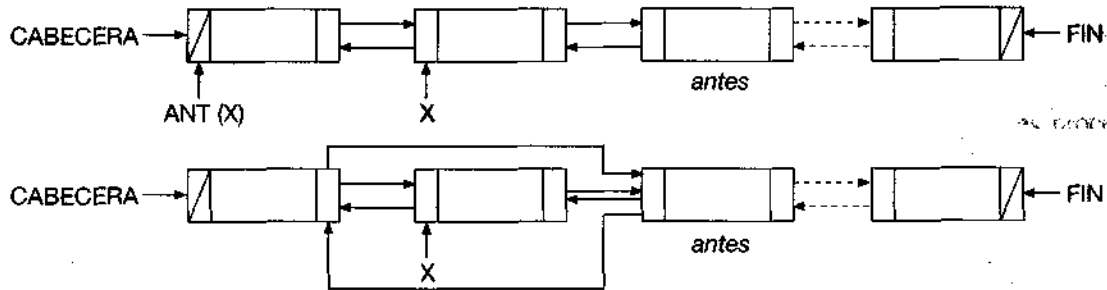


Figura 12.14. Eliminación de un nodo en una lista doblemente enlazada.

12.7. PILAS

Una pila (*stack*) es un tipo especial de lista lineal en la que la inserción y borrado de nuevos elementos se realiza sólo por un extremo que se denomina *cima* o *tope* (*top*).

La pila es una estructura con numerosas analogías en la vida real: una pila de platos, una pila de monedas, una pila de cajas de zapatos, una pila de camisas, una pila de bandejas, etc.

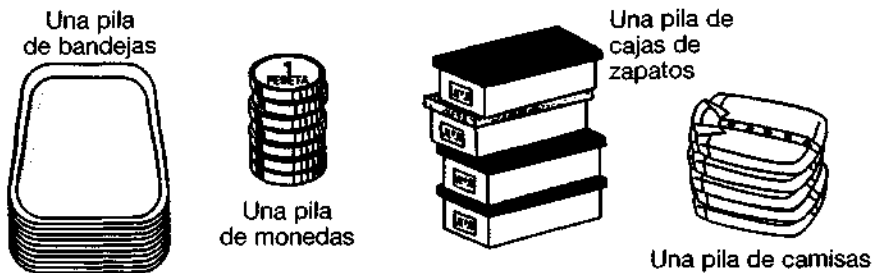


Figura 12.15. Ejemplos de tipos de pilas.

Dado que las operaciones de insertar y eliminar se realizan por un solo extremo (el superior), los elementos sólo pueden eliminarse en orden inverso al que se insertan en la pila. El último elemento que se pone en la pila es el primero que se puede sacar; por ello, a estas estructuras se les conoce por el nombre de **LIFO** (*last-in, first-out*, último en entrar, primero en salir).

Las operaciones más usuales asociadas a las pilas son:

- "push" Meter, poner o apilar: operación de insertar un elemento en la pila.
- "pop" Sacar, quitar o desapilar: operación de eliminar un elemento de la pila.

Las pilas se pueden representar en cualquiera de las tres formas de la Figura 12.16.

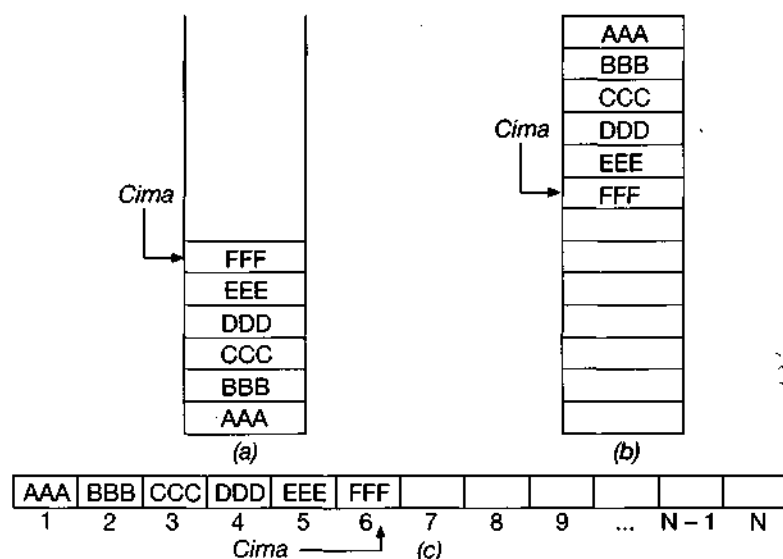
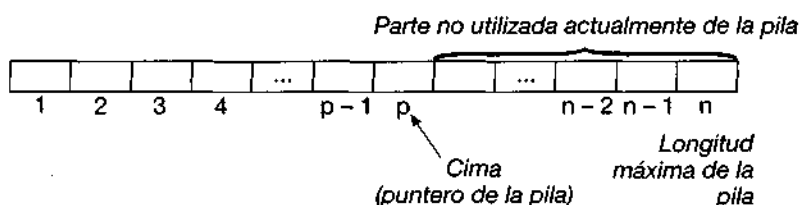


Figura 12.16. Representación de las pilas.

Para representar una pila *St*, se debe definir un vector con un determinado tamaño (longitud máxima):

```
var array [1..n] de <tipo_dato> : St
```

Se considerará un elemento entero *P* como el puntero de la pila (*stack pointer*). *P* es el subíndice del array correspondiente al elemento cima de la pila (esto es, el que ocupa la última posición). Si la pila está vacía, $P = 0$.



En principio, la pila está vacía y el puntero de la pila o CIMA está a cero. Al meter un elemento en la pila, se incrementa el puntero en una unidad. Al sacar un elemento de la pila se decrementa en una unidad el puntero.

Al manipular una pila se deben realizar algunas comprobaciones. En una pila vacía no se pueden sacar datos ($P = 0$). Si la pila se implementa con un array de tamaño fijo, se puede llenar cuando $P = n$ (n , longitud total de la pila) y el intento de introducir más elementos en la pila producirá un *desbordamiento de la pila*.

Idealmente una pila puede contener un número ilimitado de elementos y no producir nunca desbordamiento. En la práctica, sin embargo, el espacio de almacenamiento disponible es finito. La codificación de una pila requiere un cierto equilibrio, ya que si la longitud máxima de la pila es demasiado grande se gasta mucha memoria, mientras que un valor pequeño de la longitud máxima producirá desbordamientos frecuentes.

Para trabajar fácilmente con pilas es conveniente diseñar subprogramas de *poner (push)* y *quitar (pop)* elementos. También es necesario con frecuencia comprobar si la pila está vacía; esto puede con-

seguirse con una variable o función booleana VACIA, de modo que cuando su valor será *verdadero* la pila está vacía y *falso* en caso contrario.

P = CIMA *Puntero de la pila.*
 VACIA *Función booleana «pila vacía».*
 PUSH *Subprograma para añadir, poner o insertar elementos.*
 POP *Subprograma para eliminar o quitar elementos.*
 LONGMAX *Longitud máxima de la pila.*
 S(i) *Elemento i-ésimo de la pila S.*
 X *Elemento a añadir/quitar de la pila.*

Implementación con punteros

Si el lenguaje tiene punteros, deberemos implementar las pilas con punteros.

Para la manipulación de una pila mediante punteros deberemos diseñar los siguientes procedimientos y/o funciones: inicializar o crear, apilar o meter, desapilar o sacar, consultarCima y Vacía.

```

algoritmo pilas_con_punteros
tipo
  puntero_a nodo: punt
  registro : tipo_elemento
  .... : ....
  / .... : ....
fin_registro
  registro : nodo
  tipo_elemento : elemento
  punt : cima
fin_registro
var
  punt : cima
  elemento : tipo_elemento

inicio
  inicializar(cima)
  ...
fin

procedimiento inicializar(S punt: cima)
inicio
  cima ← nulo
fin_procedimiento

lógico función vacia(E punt: cima)
inicio
  devolver (cima = nulo)
fin_función

procedimiento consultarCima(E punt: cima;
                                S tipo_elemento: elemento)
inicio
  elemento ← cima→.elemento
fin_procedimiento
  
```

XXX

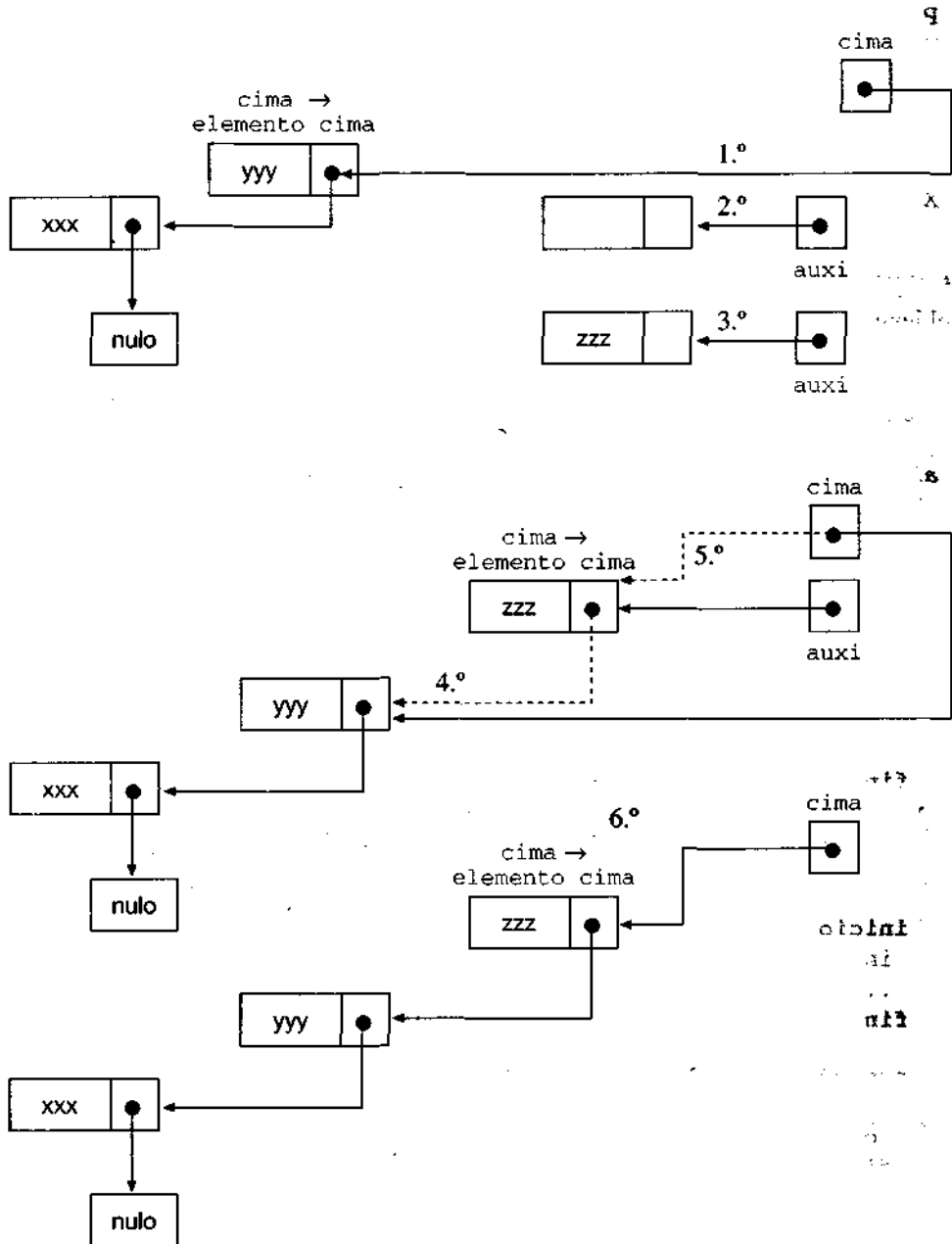
35MIO °.F

°C

°F

Los elementos se incorporan siempre por un extremo, cima.

Meter(cima, elemento)



- 1.º cima apunta al último elemento de la pila.
- 2.º reservar(aux).
- 3.º Introducimos la información en auxi→.elemento.
- 4.º Hacemos que auxi→.cima apunte a donde cima.
- 5.º Cambiamos cima para que apunte donde auxi.
- 6.º La pila tiene un elemento más.

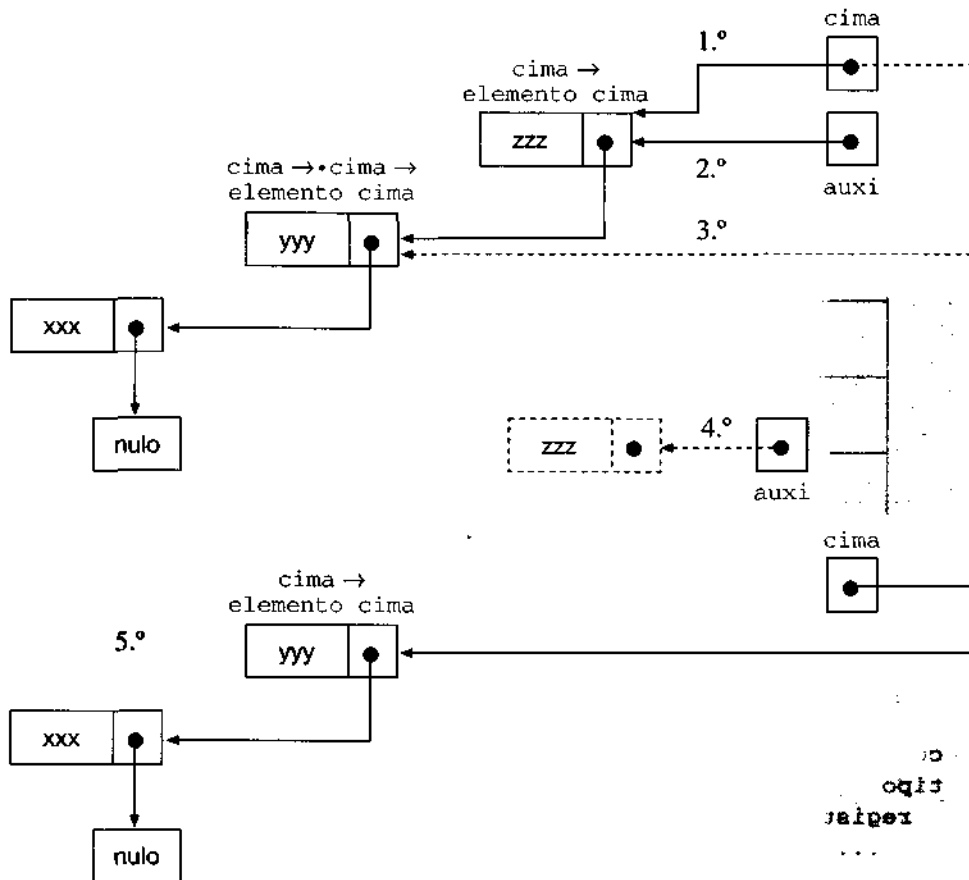
```

procedimiento meter(E/S punt: cima; E tipo_elemento: elemento)
var
    punt: auxi
inicio
    reservar(auxi)
    auxi→.elemento ← elemento
    auxi→.cima ← cima
    cima ← auxi
fin_procedimiento

```

Los elementos se recuperan en orden inverso a como fueron introducidos

Sacar(cima, elemento)



```

1.° cima apunta al último elemento de la pila
2.° Hacemos que auxi apunte a donde apuntaba cima
3.° Y que cima pase a apuntar a donde cima→.cima
4.° liberar(auxi)
5.° La pila tiene un elemento menos

```

```

procedimiento sacar(E/S punt:cima; S tipo_elemento: elemento)
var
    punt: auxi

```

```

inicio
  auxi ← cima
  elemento ← cima →.elemento
  cima ← cima →.cima
  liberar(auxi)
  {liberar es un procedimiento para la eliminación de
   variables dinámicas}
fin_procedimiento

```

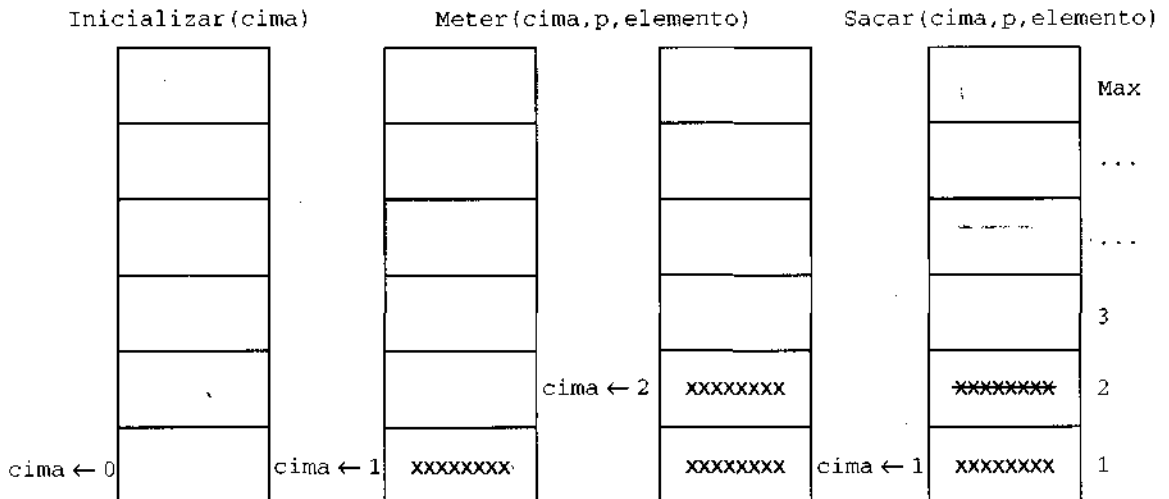
Implementación con arrays

Necesitaremos un array y una variable numérica cima que apunte al último elemento colocado en la pila.

La inserción o extracción de un elemento se realizará siempre por la parte superior.

Su implementación mediante arrays limita el máximo número de elementos que la pila puede contener y origina la necesidad de una función más.

Llena(...) de resultado lógico



```

const Max = <expresión>
tipo
  registro: tipo_elemento
  ... : ...
  ... : ...
fin_registro
  array[1..Max] de tipo_elemento: arr
var
  entero      : cima
  arr         : p
  tipo_elemento : elemento
inicio
  inicializar(cima)
  ...
fin

```

```

procedimiento inicializar(S entero: cima)
  inicio
    cima ← 0
  fin_procedimiento

lógico función vacia(E entero: cima)
  inicio
    si cima = 0 entonces
      devolver (verdad)
    si_no
      devolver (falso)
    fin_si
  fin_función

lógico función llena(E entero: cima)
  inicio
    si cima = Max entonces
      devolver (verdad)
    si_no
      devolver (falso)
    fin_si
  fin_función

procedimiento consultarCima(E entero: cima; E arr: p;
                           S tipo_elemento: elemento)
  inicio
    elemento ← p[cima]
  fin_procedimiento

procedimiento meter(E/S entero: cima; E/S arr: p;
                    E tipo_elemento: elemento)
  inicio
    cima ← cima + 1
    p[cima] ← elemento
  fin_procedimiento

procedimiento sacar(E/S entero: cima; E arr: p;
                    S tipo_elemento: elemento)
  inicio
    elemento ← p[cima]
    cima ← cima - 1
  fin_procedimiento

```

12.7.1. Aplicaciones de las pilas

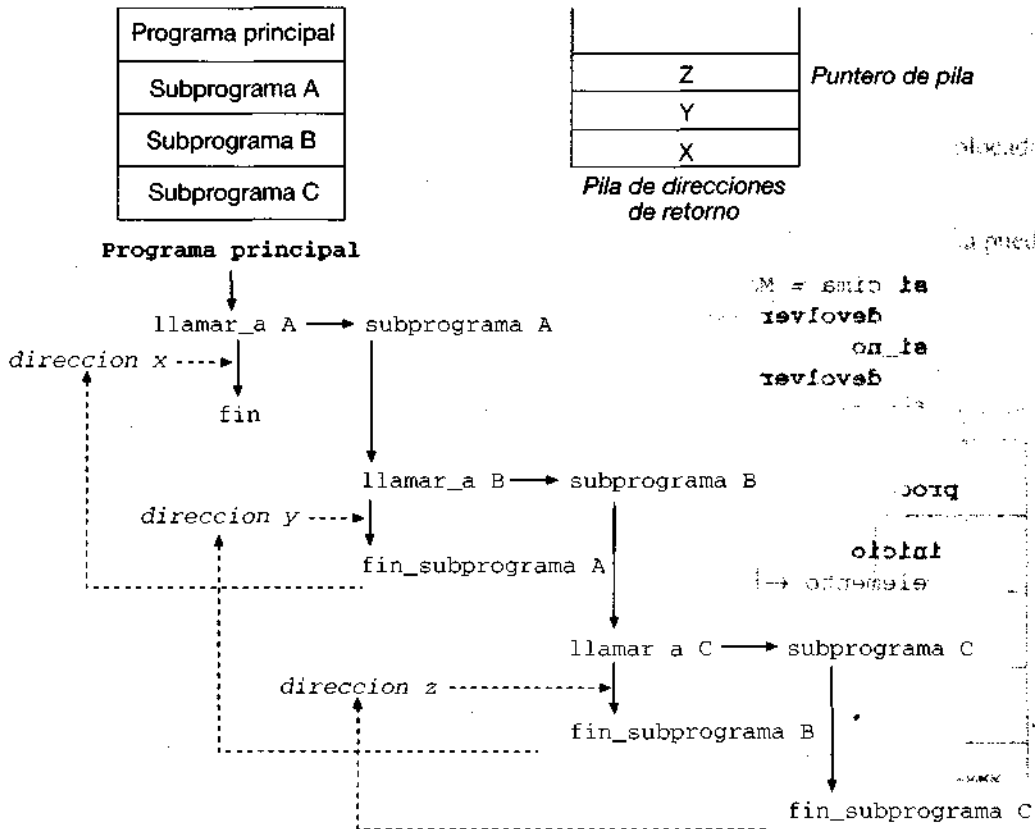
Las pilas son utilizadas ampliamente para solucionar una amplia variedad de problemas. Se utilizan en compiladores, sistemas operativos y en programas de aplicación. Veamos algunas de las aplicaciones más interesantes.

Llamadas a subprogramas

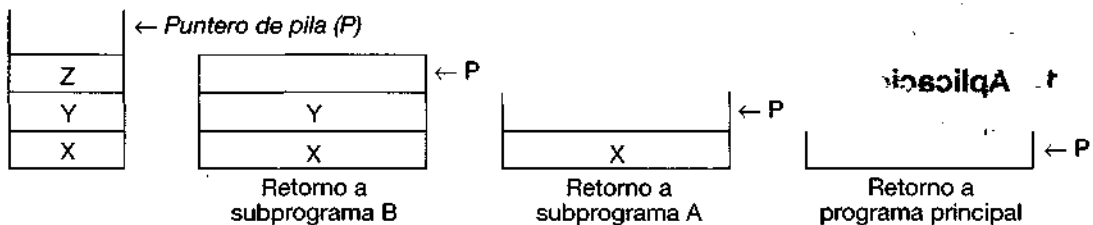
Cuando dentro de un programa se realizan llamadas a subprogramas, el programa principal debe recordar el lugar donde se hizo la llamada, de modo que pueda retornar allí cuando el subprograma se haya terminado de ejecutar.

Supongamos que tenemos tres subprogramas llamados A, B y C, y supongamos también que A invoca a B y B invoca a C. Entonces B no terminará su trabajo hasta que C haya terminado y devuelto su control a B. De modo similar, A es el primero que arranca su ejecución, pero es el último que la termina, tras la terminación y retorno de B.

Esta operación se consigue disponiendo las direcciones de retorno en una pila.



Cuando un subprograma termina, debe retornar a la dirección siguiente a la instrucción que le llamó (**llamar_a**). Cada vez que se invoca un subprograma, la dirección siguiente (x, y o z) se introduce en la pila. El vaciado de la pila se realizará por los sucesivos retornos, decrementándose el puntero de pila queda libre siempre apuntando a la siguiente dirección de retorno.



Ejemplo 12.6

Se desea leer un texto y separar los caracteres letras, dígitos y restantes caracteres para ser utilizados posteriormente.

Utilizaremos tres pilas (LETRAS, DIGITOS, OTROSCAR) para contener los diferentes tipos de caracteres. El proceso consiste en leer carácter a carácter, comprobar el tipo de carácter y según el resultado introducirlo en su pila respectiva.

```

algoritmo lecturacaracter
  const Max = <valor>
  tipo
    array [1..Max] de carácter:pila
  var
    entero      : cima1, cima2, cima3
    pila        : pilalettras, piladigitos, pilaotroscaracteres
    carácter    : elemento

inicio
  crear (cima1)
  crear (cima2)
  crear (cima3)
  elemento ← leercar
  mientras (codigo(elemento) <> 26) y no llena (cima1) y no
    llena(cima2) y no llena (cima3) hacer
    {saldremos del bucle en cuanto se llene alguna de las pilas o
    pulsemos ^Z}
    si (elemento >= 'A') y (elemento <= 'Z') o (elemento >= 'a') y
      (elemento >= 'z') entonces
      meter (cima1, pilalettras, elemento)
    si_no
      si (elemento >= '0') y (elemento <= '9') entonces
      meter (cima2, piladigitos, elemento)
      si_no
      meter (cima3, pilaotroscaracteres, elemento)
      fin_si
    fin_si
    elemento ← leercar
  fin_mientras
fin

procedimiento crear (S entero: cima)
  inicio
    cima ← 0
  fin_procedimiento

lógico función llena (E entero: cima)
  inicio
    devolver (cima = Max)
  fin_función

procedimiento meter(E/S entero: cima; E/S tipo_elemento: elemento)
  inicio
    cima ← cima+1
    p [cima] ← elemento
  fin_procedimiento

```

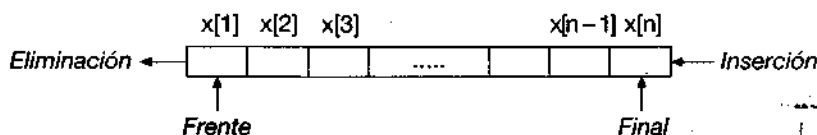
12.8. COLAS

Las colas son otro tipo de estructura lineal de datos similar a las pilas, diferenciándose de ellas en el modo de insertar/eliminar elementos.

Una *cola* (*queue*) es una estructura lineal de datos

```
var array [1..n] de <tipo_dato> : C
```

en la que las *eliminaciones* se realizan al principio de la lista, *frente* (*front*), y las *inserciones* se realizan en el otro extremo, *final* (*rear*). En las colas el elemento que entró el primero sale también el primero; por ello se conoce como listas **FIFO** (*first-in, first-out*, «primero en entrar, primero en salir»). Así, pues, la diferencia con las pilas reside en el modo de entrada/salida de datos; en las colas las inserciones se realizan al final de la lista, no al principio. Por ello las colas se usan para almacenar datos que necesitan ser procesados según el orden de llegada.



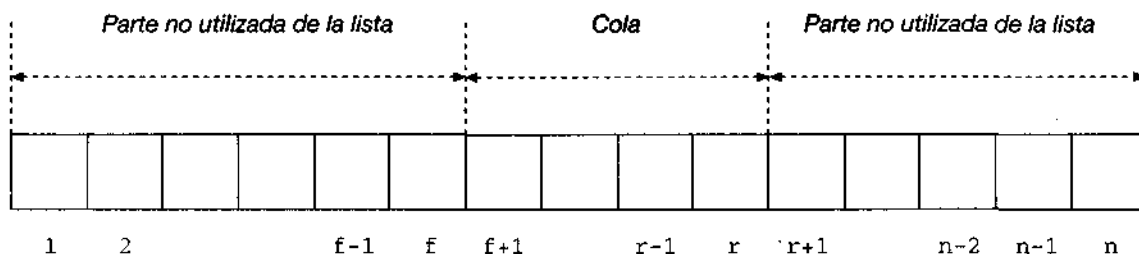
En la vida real se tienen ejemplos numerosos de colas: la cola de un autobús, la cola de un cine, una caravana de coches en una calle, etc. En todas ellas el primer elemento (pasajero, coche, etc.) que llega es el primero que sale.

En informática existen también numerosas aplicaciones de las colas. Por ejemplo, en un sistema de tiempo compartido suele haber un procesador central y una serie de periféricos compartidos: discos, impresoras, etc. Los recursos se comparten por los diferentes usuarios y se utiliza una cola para almacenar los programas o peticiones de los diferentes usuarios que esperan su turno de ejecución. El procesador central atiende —normalmente— por riguroso orden de llamada del usuario; por tanto, todas las llamadas se almacenan en una cola. Existe otra aplicación muy utilizada que se denomina *cola de prioridades*; en ella el procesador central no atiende por riguroso orden de llamada, aquí el procesador atiende por prioridades asignadas por el sistema o bien por el usuario, y sólo dentro de las peticiones de igual prioridad se producirá una cola.

12.8.1. Representación de las colas

Las colas se pueden representar por listas enlazadas o por arrays.

Se necesitan dos punteros: *frente*(*f*) y *final*(*r*), y la lista o array de *n* elementos (LONGMAX).



si la cola está vacía
eliminar elementos
añadir elementos

frente = nulo
frente \leftarrow frente + 1
final \leftarrow final + 1

o bien
o bien
o bien

f \leftarrow 0
f \leftarrow f - 1
r \leftarrow r + 1

q