Fundamentos de Programación II

Tema 4. Estructuras lineales de datos: listas, pilas, colas

Luis Rodríguez Baena (luis.rodriguez@upsam.es)

Universidad Pontificia de Salamanca
Escuela Superior de Ingeniería y Arquitectura

Tipos abstractos de datos

- Procedimientos y funciones *generalizan* el concepto de operador.
  - El programador puede definir sus propios operadores y aplicarlos sobre operandos de tipos no definidos en el lenguaje.
    - ✓ En un algoritmo en lugar de utilizar sólo los operadores que utiliza el lenguaje, mediante procedimientos y funciones se pueden aplicar sus propios operandos a tipos de datos no definidos por el lenguaje.
      - Por ejemplo, se puede ampliar el operador de multiplicación para multiplicar matrices.
  - Procedimientos y funciones encapsulan las operaciones, las aíslan del cuerpo del algoritmo.
- Tipo Abstracto de Datos (TAD).
  - Amplía el concepto de procedimiento a la definición de datos.
  - Modelo matemático del dato junto con las operaciones que se pueden definir sobre él.
  - Utilizan también generalización y encapsulamiento.
    - ✓ Son generalizaciones de los tipos de datos primitivos.
    - ✓ Facilitan la localización de la definición del tipo y de las operaciones para su manejo.

Tipos abstractos de datos (II)

- Por ejemplo, se puede definir el tipo de datos `Cuadrado` y las operaciones `Área(c)` y `Perímetro(c)`que devolverían el valor del área y del perímetro del cuadrado `c`.
- El cuadrado se puede implementar de distintas formas:
  - Con la posición de los cuatro vértices en las coordenadas de un plano.

```
registro=punto
   real : x,y
fin_registro
registro=cuadrado
   punto: infIzq, infDer, supIzq, supDer
fin_registro
```

  - Con la posición de un punto y el tamaño del lado.

```
registro=cuadrado
   punto:origen
   real: lado
fin_registro
```

Tipos abstractos de datos (III)

- El procedimiento área podría implementarse de distintas formas dependiendo de cómo se haya definido el cuadrado.

```
real función Área(valor cuadrado: c)
inicio
   devolver(c.lado * c.lado)
fin_función

real función Área(valor cuadrado: c)
var
   real : lado
inicio
   //lado es la distancia entre dos puntos
   lado ← raiz2((c.infIzq.x – infDer.x)**2+(c.infIzq.y – infDer.y)**2)
   devolver(lado * lado)
fin_función
```

  - Si en un programa utilizamos el tipo de datos `Cuadrado` y tenemos definidas en ese tipo de dato la función `Área`, la llamada a la función será `Area(c)`, independientemente de la implementación que hayamos hecho.

Tipos abstractos de datos (IV)

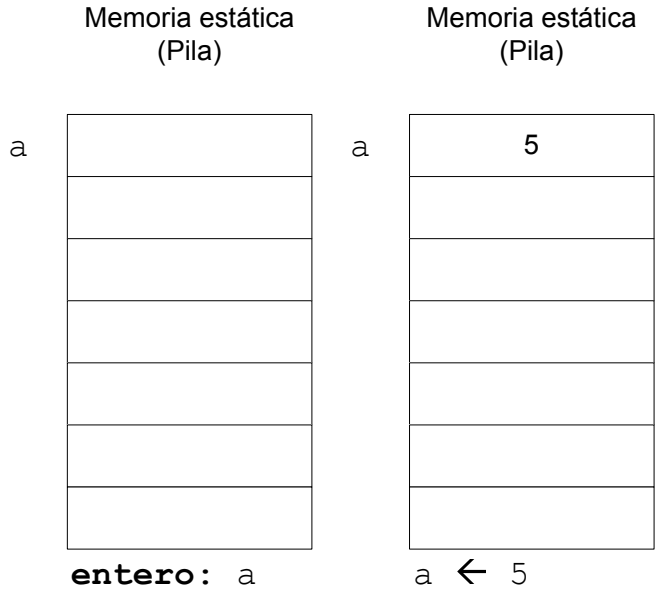
- Estructuras de datos "físicas" y "lógicas".
  - Podemos considerar que una estructura de datos física es la que implementa el lenguaje de programación que se está utilizando.
  - Una estructura de datos lógica sería una estructura de datos que no implementa el lenguaje y que creamos a partir de los tipos de datos y las estructuras de datos físicas que proporciona el lenguaje.
  - Este concepto puede variar de un lenguaje a otro:
    - ✓ Conjunto: presente en Pascal y no presente en C.
    - ✓ Cadena: presente en Java, pero no presente en Pascal estándar.
    - ✓ Archivos indexados: presente en Cobol y no presente en C.
- Las estructuras de datos que se han utilizado hasta ahora ya estaban definidas por el lenguaje de especificación de algoritmos utilizado (son estructuras de datos "físicas").
  - El lenguaje define el tipo array e incluye las herramientas necesarias para declarar un array, acceder a un elemento, etc.

Tipos abstractos de datos (V)

- Los lenguajes no soportan todas las estructuras de datos posibles.
  - En muchas ocasiones es necesario realizar la definición de la estructura de datos y declarar las operaciones primitivas que la manejan.
    - ✓ Haremos la definición del Tipo Abstracto de Datos.
      - Definir el tipo de dato que contendrán.
      - Establecer la organización de los datos utilizando los datos primitivos.
      - Establecer las **operaciones primitivas** para manejar el tipo de dato.
- Las estructuras de datos utilizadas en este tema (pilas, colas, listas enlazadas) no están implementadas en el lenguaje de programación.
  - Será necesario definirlas con las estructuras de datos disponibles.
  - Será necesario implementar las operaciones primitivas para manejarlas.

Datos estáticos y dinámicos

- Dato estático.
  - Dato que no puede variar su ubicación a lo largo de la ejecución del programa.
  - Se reserva espacio en tiempo de compilación.
    - ✓ Cuando el programa empieza su ejecución es necesario saber de antemano su tamaño y ubicación en la memoria.
  - Se almacena en una zona de memoria estática: pila.
  - El dato se identifica por una variable que no es más que una dirección de memoria dónde está almacenada la información.
  - Cuando se asigna un valor a ese dato, se almacena directamente en esa dirección de memoria.
  - Cuando se accede a ese dato, por ejemplo al ejecutar la instrucción **`escribir`**`(a)`, se accede de forma directa al contenido de esa dirección de memoria.

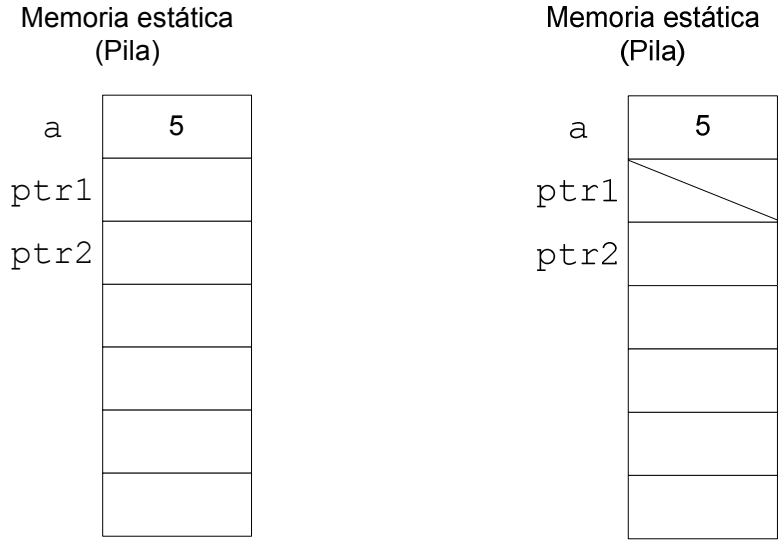


Datos estáticos y dinámicos (II)

- Dato dinámico.
  - Realiza una asignación dinámica de memoria.
    - ✓ Reserva espacio para almacenar la información en tiempo de ejecución.
  - Cuando el programa comienza su ejecución, sólo se reserva espacio para almacenar una referencia a la posición donde se almacenará el dato, no para almacenar el dato en sí.
    - ✓ La variable que guarda la dirección de memoria (la referencia) es el **puntero**.
    - ✓ El puntero se almacena en la pila.
  - Durante la ejecución del programa es posible reservar memoria para el dato al que apunta el puntero.
    - ✓ El dato en sí se almacena en una zona de memoria dinámica: el montículo.
  - En cualquier momento se puede reservar o liberar ese espacio.

Datos estáticos y dinámicos (III)

- Definición de un puntero.
  puntero_a `tipoDato : varPuntero`
  - `tipoDato`, cualquier tipo de dato estándar o definido por el usuario.
  - Al arrancar el programa la variable de tipo puntero no está inicializada (no tiene ningún valor).
  - Sólo es posible darle valor asignándola a otro puntero, reservando espacio en memoria o asignándola a **puntero nulo**.
- Puntero nulo.
  - Se corresponde a la constante `nulo`.
  - Se considera un puntero inicializado (cómo dar a una variable numérica un valor 0).
  - Apunta a una dirección de memoria nula.



puntero_a entero : ptr1
puntero_a entero : ptr2

ptr1 ← **nulo**

Datos estáticos y dinámicos (IV)

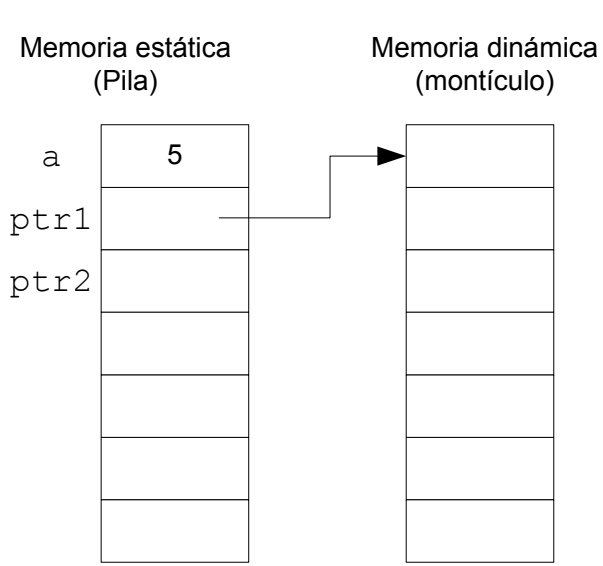
- Reservar espacio en el montículo.

  reservar(varPuntero)

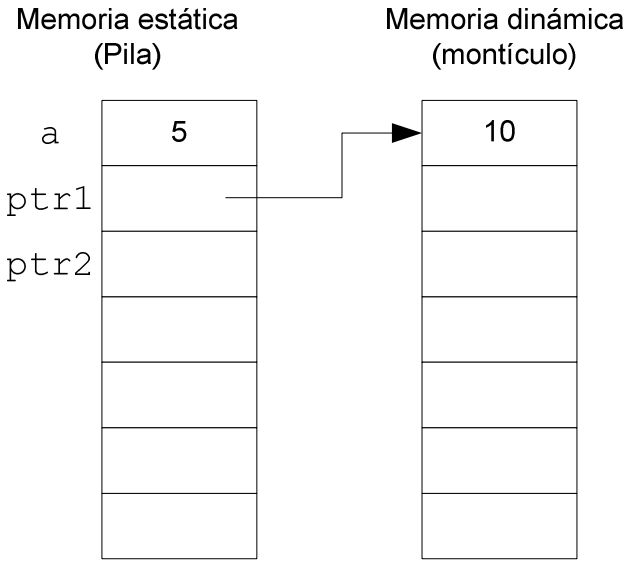
  - Busca espacio en memoria para almacenar una variable del tipo de varPuntero.
  - varPuntero se carga con la dirección de memoria que se ha encontrado.
- Referencia al dato apuntado por el puntero.
  - Se utiliza el operador ↑.

  varPuntero↑
  - Si varPuntero hace referencia a la zona de memoria a la que apunta, varPuntero↑ hace referencia al contenido de dicha zona de memoria.
  - Hay que tener en cuenta que:
    - ✓ varPuntero es una variable de tipo puntero, por lo que sólo es posible asignarla otro puntero (otra variable de tipo puntero, puntero nulo o reservar espacio).
      - Los operadores de asignación o las instrucciones de lectura o escritura no funcionan de la misma forma para variables de tipo puntero que para otro tipo de variables.
    - ✓ varPuntero↑ hace referencia al contenido de la memoria a la que apunta al puntero, por lo que es un dato del tipo base del puntero.
      - Sobre ella se podrán hacer todas las operaciones que se puedan hacer con el tipo base del puntero.

Datos estáticos y dinámicos (V)



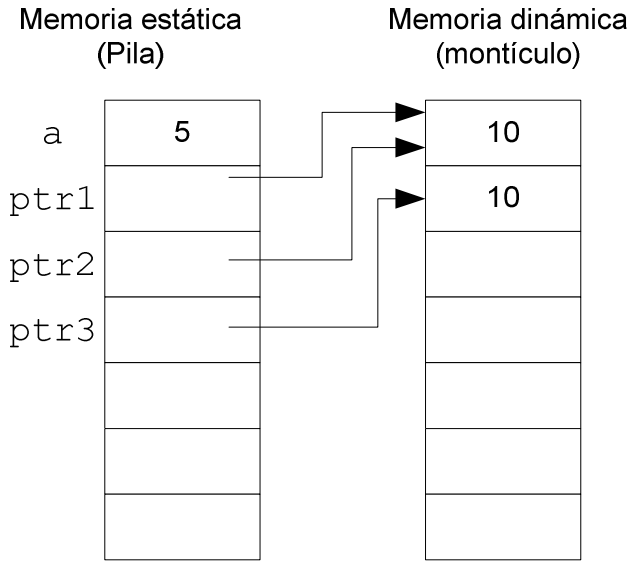

```
reservar(ptr1)
ptr1 ← 5 //Error
escribir(ptr1) //Error
leer(ptr1) //Error
```




```
ptr1↑← 10
escribir(ptr1↑)
leer(ptr1↑)
```

Datos estáticos y dinámicos (VI)

- Asignación de variables de tipo puntero.
  - Lo que asigna no es el contenido, sino la dirección de memoria, la referencia.
- Comparación de variables de tipo puntero.
  - Compara si dos punteros apuntan a la misma dirección de memoria.



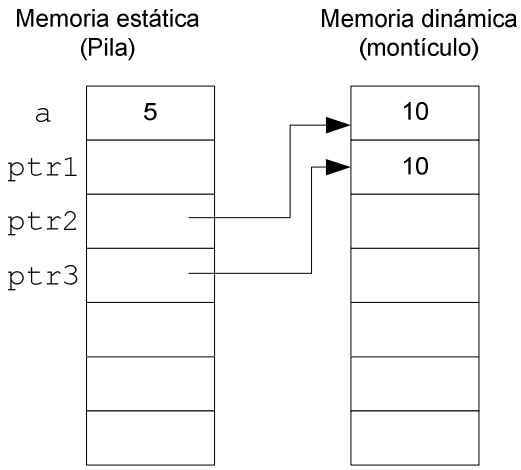

```
ptr2← ptr1
ptr3↑← 10
ptr1= ptr2  //Verdad
ptr1= ptr3  //Falso
```

Datos estáticos y dinámicos (VII)

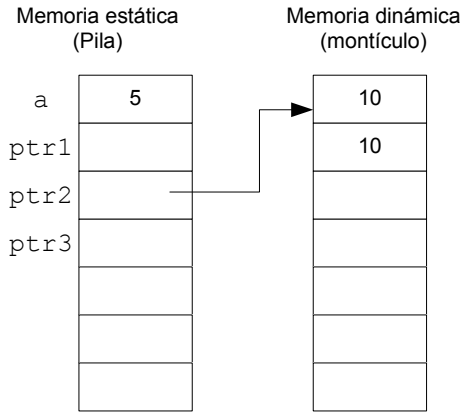
- Liberar espacio de almacenamiento.

  liberar(varPuntero)

  - Deja libre la posición de memoria, volviéndola a marcar como zona libre.
  - El puntero que apuntaba la posición queda indeterminado.
  - No es posible recuperar la información a la que apuntaba el puntero.
    - ✓ A no ser que exista otra referencia que también apuntaba al mismo dato (como `ptr2`).



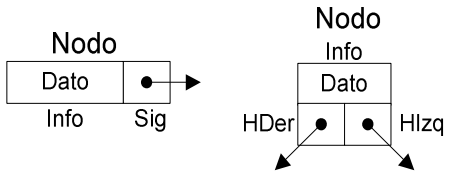
liberar(ptr1)



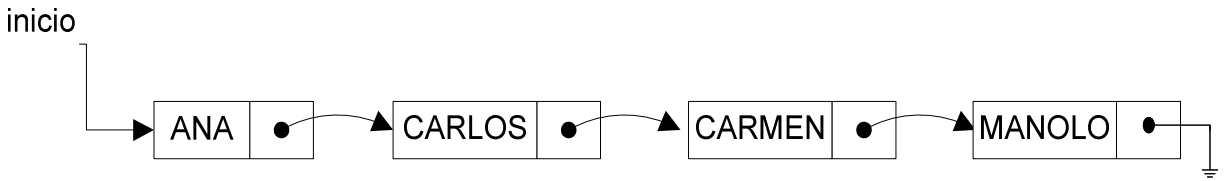
liberar(ptr3)

Estructuras de datos dinámicas

- Sus componentes están dispersos por la memoria.
  - Al no ocupar posiciones contiguas es necesario establecer un mecanismo para acceder al siguiente elemento de la estructura.
  - Es necesario saber cuál es el primer elemento de la estructura
- Cada elemento de la estructura es un **nodo**.
  - Cada nodo contiene la información y, por lo menos, un puntero indicando cual es el siguiente elemento de la estructura.

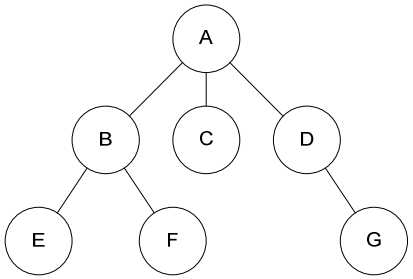


  - Existe una variable de tipo puntero que apunta al primer nodo de la estructura.

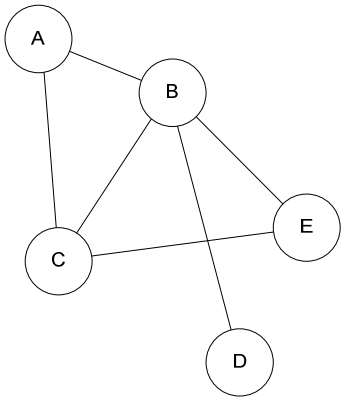


Estructuras de datos lineales y no lineales

- Estructuras de datos lineales.
  - Cada componente tiene un único sucesor y un único predecesor con excepción del último y el primero.
- Estructura de datos no lineal.
  - Cada componente puede tener varios sucesores y varios predecesores.



Árbol



Grafo

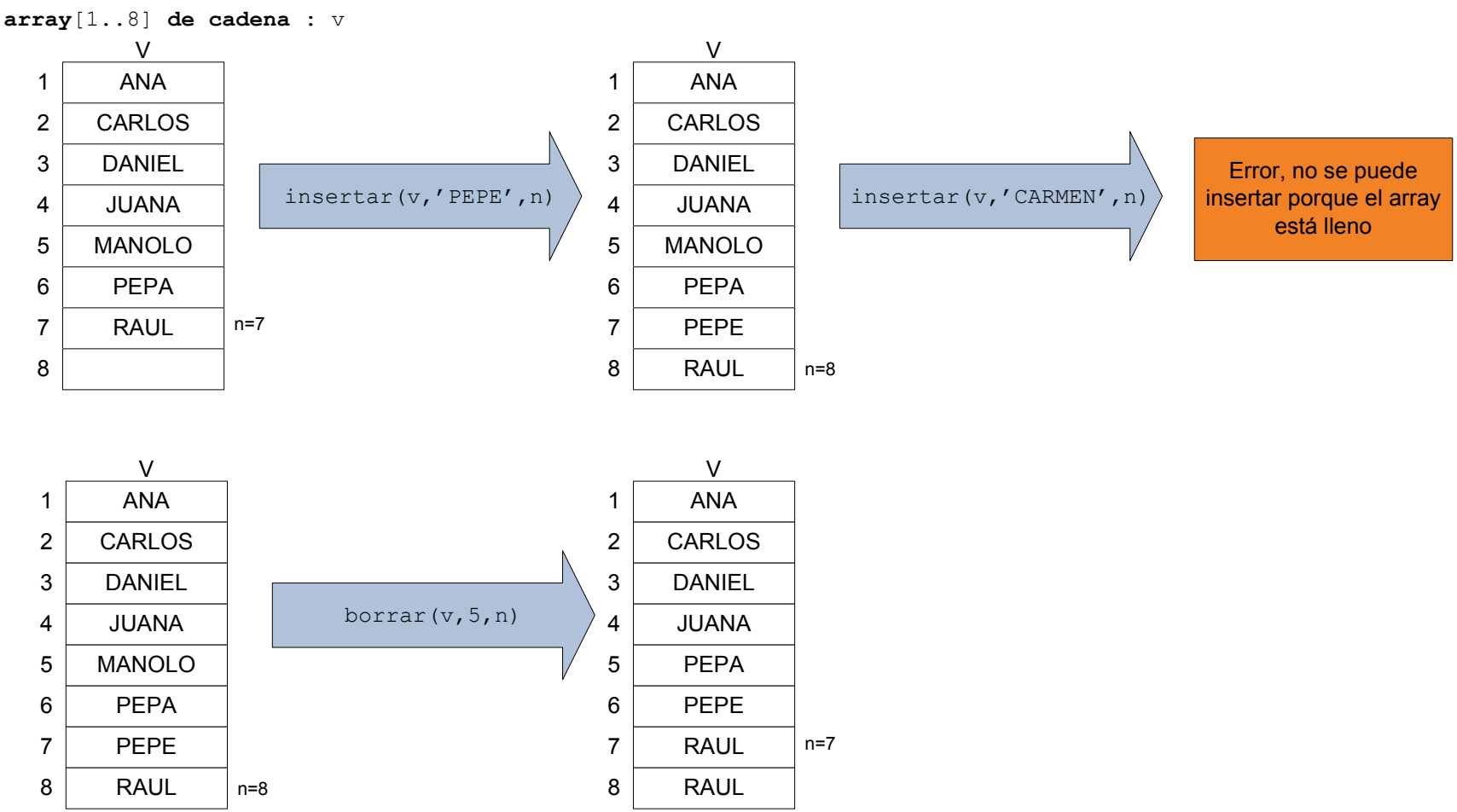
Estructuras de datos lineales: listas

- Estructura lineal compuesta por una secuencia de 0 o más elementos de algún tipo determinado y ordenados de alguna forma.
- Puede crecer o disminuir en el número de elementos y podrán insertarse o eliminarse elementos en cualquier posición sin alterar su orden lógico.

Estructuras de datos lineales: listas (II)

- Listas contiguas.
  - Los elementos ocupan posiciones contiguas de memoria.
  - Se pueden implementar con arrays.
    - ✓ Los elementos ocuparían posiciones correlativas del array.
    - ✓ Presentan dos problemas:
      - La inserción o el borrado de elementos implica mover las posiciones para mantener el orden original.
      - El número de elementos de la lista se puede modificar, pero no puede sobrepasar el tamaño máximo del array.

Estructuras de datos lineales: listas (III)

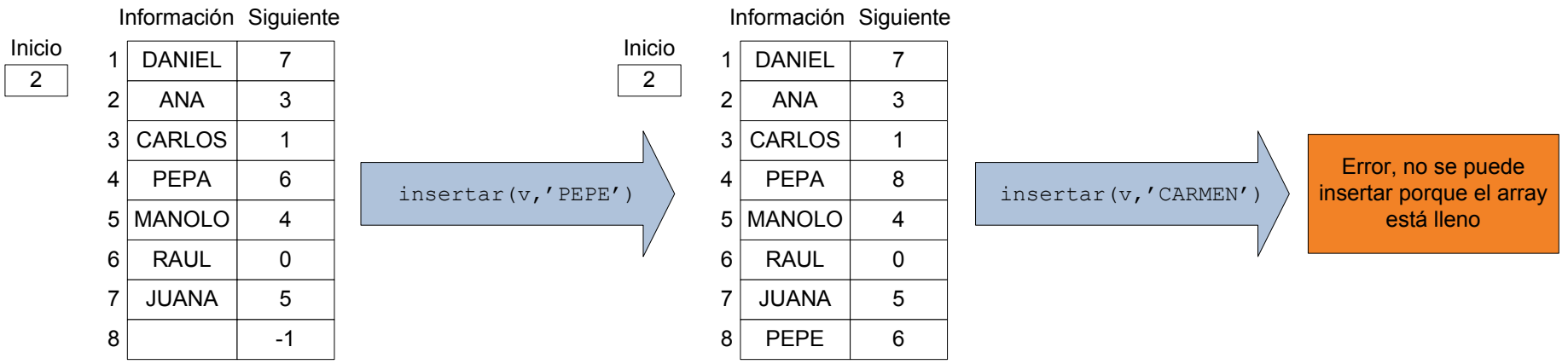


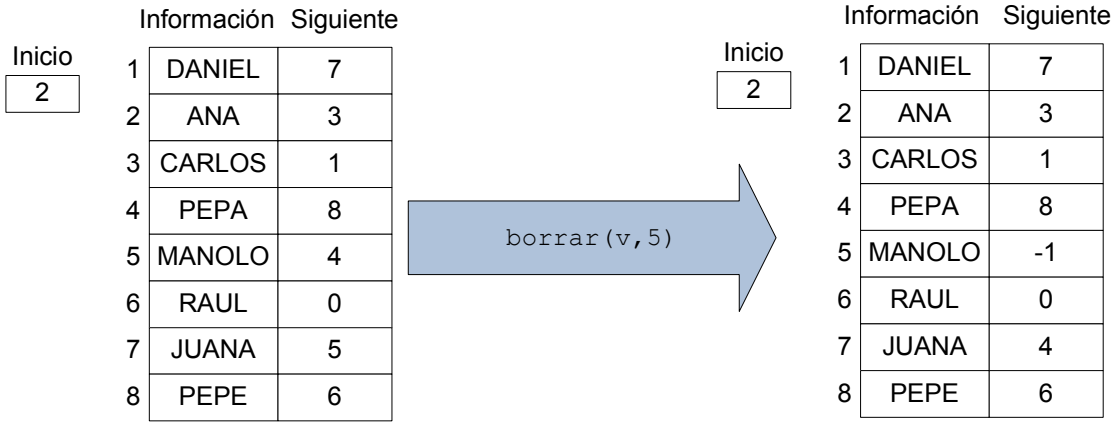
Estructuras de datos lineales: listas (IV)

- Listas enlazadas.
  - Los elementos no ocupan posiciones contiguas de memoria.
    - ✓ Aparecen dispersos por el almacenamiento.
  - Cada elemento contiene una referencia al siguiente elemento de la estructura.
  - El orden lógico lo darán los enlaces que hay entre elementos.
  - El primer elemento en el orden lógico no tiene por qué corresponderse con el primer elemento almacenado.
    - ✓ El necesario indicar cuál es el primer elemento en el orden lógico.
  - También es necesario indicar cual será el último elemento en el orden lógico de la estructura.
  - La inserción o eliminación de elementos no implica mover los elementos de sitio.
    - ✓ Sólo se modifican las referencias al siguiente elemento.
  - Si se utilizan estructuras de datos dinámicas el número de elementos será virtualmente ilimitado.

Estructuras de datos lineales: listas (V)

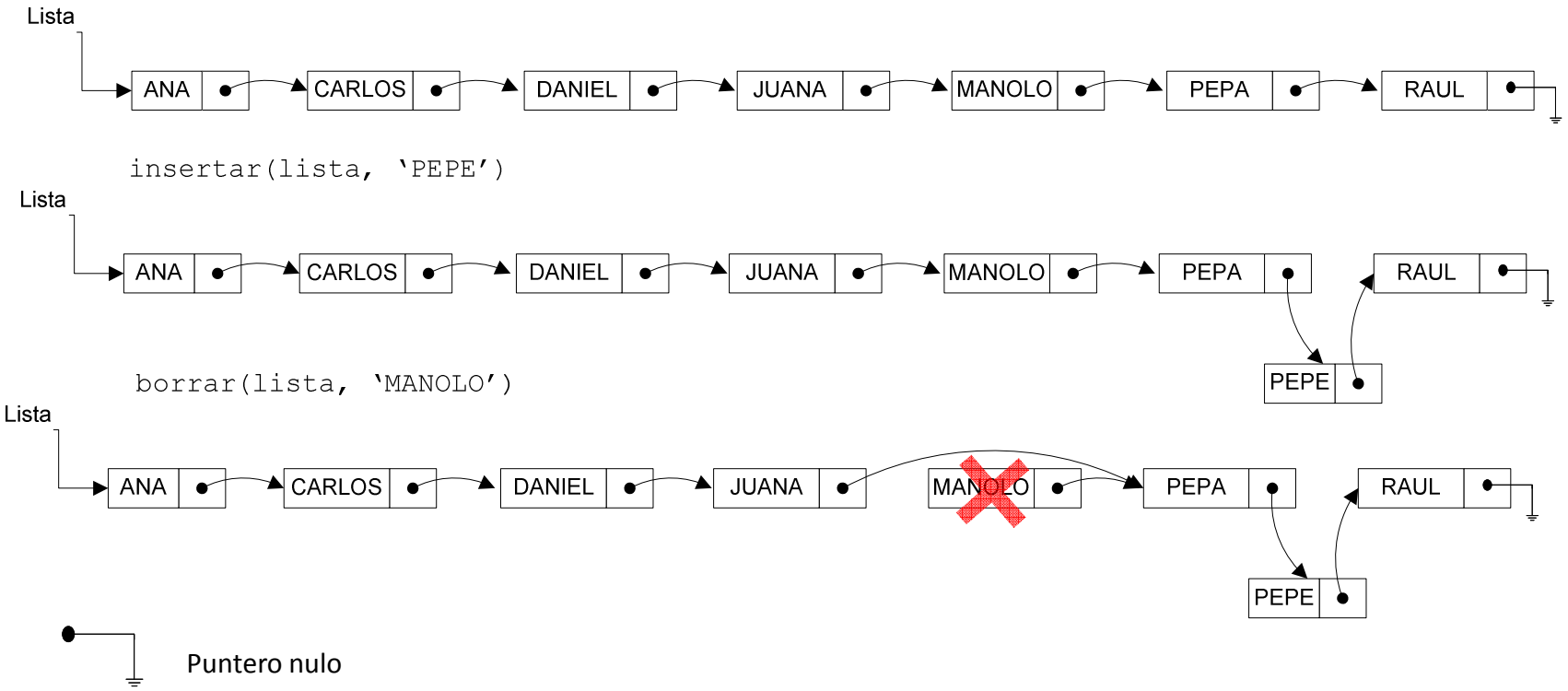
- Implementación mediante arrays de registros…





Estructuras de datos lineales: listas (VI)

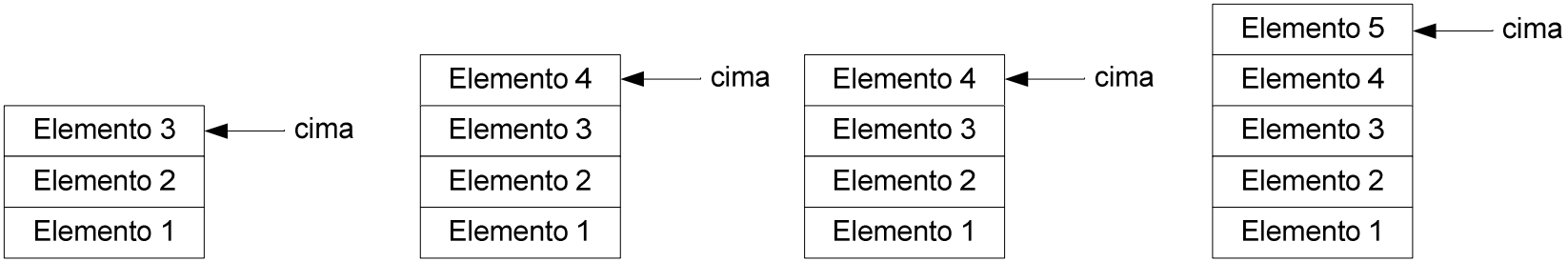
- Implementación mediante punteros…



Pilas

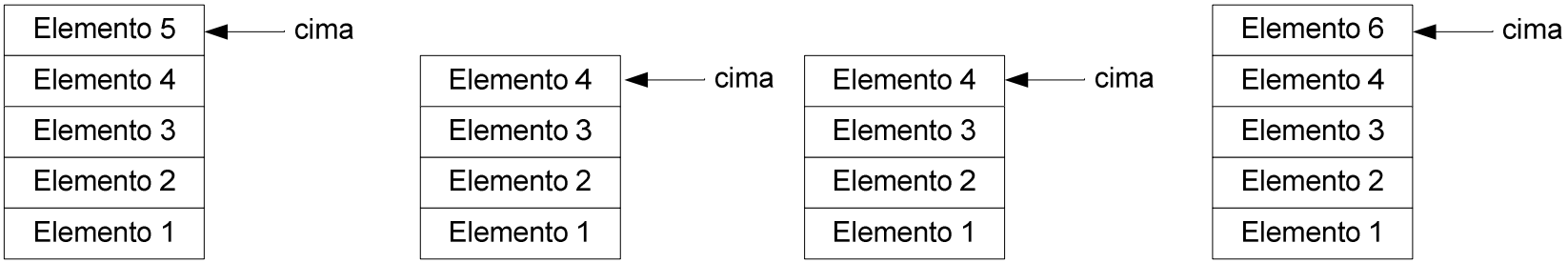
- Es un tipo especial de lista.
- Estructura lineal de datos compuesta por una secuencia de elementos en la que las operaciones sólo se pueden realizar por uno de sus extremos llamado **cima** (tope o *top*).
  - Estructuras de tipo LIFO (*Last In-First Out*).
- Se utiliza para poder recuperar elementos en orden inverso a como entran.
  - Ejemplos reales: montones de platos.
  - Aplicaciones en informática.
    - ✓ Evaluación de algunos tipos de expresiones
    - ✓ Cuadros de diálogos, pantallas y menús desplegables.
      - Cada cuadro se abre encima de otro; al cerrarse uno, el cuadro activo es el último que se ha abierto.
    - ✓ Simulación de la recursividad.
    - ✓ …
    - ✓ En general, cualquier aplicación en la que se necesite recuperar información en orden inverso a como ha entrado.

Pilas (II)



Insertar el elemento 4

Insertar el elemento 5



Sacar el elemento 5

Insertar el elemento 6

Pilas (III)

- La estructura de tipo Pila no se considera implementada en nuestro lenguaje de programación.
  - En otros lenguajes, como .NET, existe la clase Stack (pila en inglés) que implementa pilas e incluye todas las operaciones que se pueden hacer sobre las pilas.
- Será necesario crear el tipo de dato Pila.
  - Determinar las operaciones básicas que se pueden realizar sobre el tipo de dato Pila: las **operaciones primitivas**.
  - Definir el tipo de elementos que contendrá la pila.
  - Definir la organización de los datos utilizando los datos y estructuras de datos que ofrezca el lenguaje de programación.
  - Implementar las operaciones primitivas para la organización de los datos definida.
    - ✓ Dependiendo de la organización definida, la implementación de las operaciones primitivas variará.

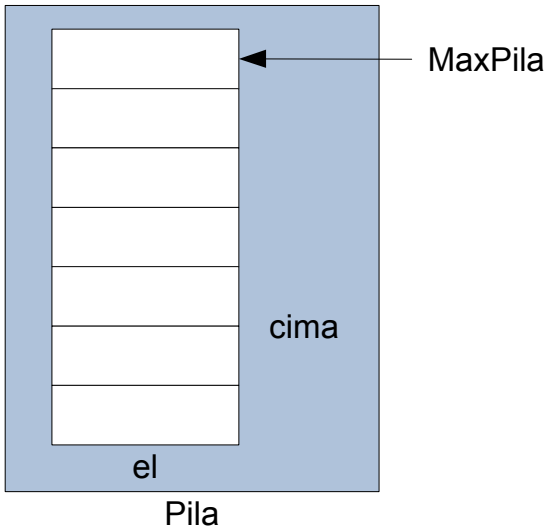
Pilas (IV)

- Operaciones primitivas.
  - Procedimiento `PilaNueva(`**`ref`** `pila: p)`
    - ✓ Crea un pila sin elementos.
  - Función lógica `EsPilaVacía(`**`valor`** `pila: p)`**.**
    - ✓ Devuelve `verdad` si la pila está vacía.
  - Procedimiento `PInsertar (`**`ref`** `pila:p;` **`valor`** `TipoElemento:e)` o `Push(`**`ref`** `pila : p;` **`valor`** `TipoElemento: e)`.
    - ✓ Inserta un elemento e en la pila y devuelve la pila resultante.
    - ✓ `TipoElemento` es un tipo de dato genérico que se corresponde al tipo de dato de los elementos que contendrá la pila.
  - Procedimiento `Cima(`**`valor`** `pila: p;` **`ref`** `TipoElemento : e)`**.**
    - ✓ Devuelve en el argumento e el elemento situado en la cima de la pila.
  - Procedimiento `PBorrar(`**`ref`** `pila : p)`.
    - ✓ Elimina el elemento cima de la pila y devuelve la pila resultante.
  - Procedimiento `Pop(`**`ref`** `pila : p;` **`ref`** `TipoElemento : e)` o `Sacar(`**`ref`** `pila : p;` **`ref`** `TipoElemento : e)`.
    - ✓ Elimina un elemento de la cima de la pila, devolviendo la pila resultante y el elemento extraído.

Realizaciones mediante arrays

- Las pilas se pueden implementar utilizando estructuras de datos dinámicas (listas enlazadas) o estáticas (arrays).
  - La pila está definida por la posición dónde está el último elemento (la `cima`).
    - ✓ Este dato es lo único que necesitamos para trabajar con pilas.
  - Si se implementa con un array, también es necesario determinar dónde se almacenará la información (el array `el`).
    - ✓ Como se trata de estructuras de datos estáticas, debemos definir también el tamaño máximo del array (`MaxPila`).
  - Además el tipo de dato `TipoElemento` definirá el tipo de elementos que almacenará la pila.
- Definición de las estructuras de datos.

```
const
 MaxPila = …
tipos
 … = TipoElemento
 registro = Pila
  entero : cima
  array[1..MaxPila] de TipoElemento : el
 fin_registro
```

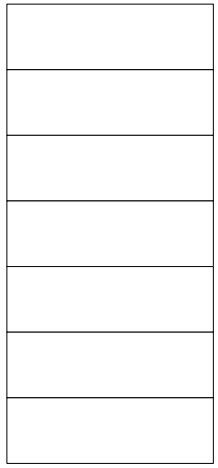


Realizaciones mediante arrays (II)

- Procedimiento `PilaNueva`.
  - Se limita a inicializar la cima de la pila a 0.
- Función `EsPilaVacía`,
  - Devuelve verdad si la pila está vacía.

```
procedimiento PilaNueva(ref pila : p)
inicio
   p.cima ← 0
fin_procedimiento

lógico: función EsPilaVacía(valor pila :p)
inicio
   devolver(p.cima = 0)
fin_función
```

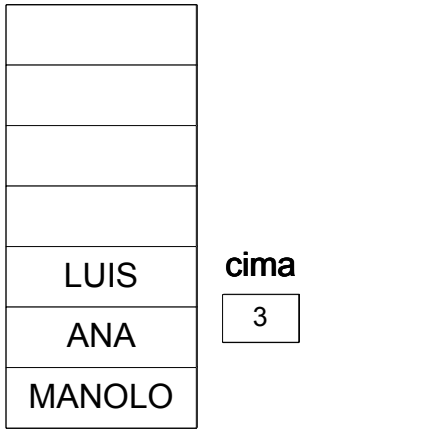



```
PilaNueva(p)
EsPilaVacía(p)   //Devuelve verdad
```

Realizaciones mediante arrays (III)

- Procedimiento `PInsertar`
  - Inserta en la posición cima un elemento de `TipoElemento`.
  - En la implementación con arrays, es necesario comprobar si la pila está llena (si `cima=MaxPila`).

```
procedimiento PInsertar(ref pila:p;
                        valor TipoElemento:e)
inicio
   si p.cima = MaxPila entonces
        // Error, la pila está llena
   si_no
        p.cima ← p.cima + 1
        p.el[p.cima] ← e
   fin_si
fin_procedimiento
```

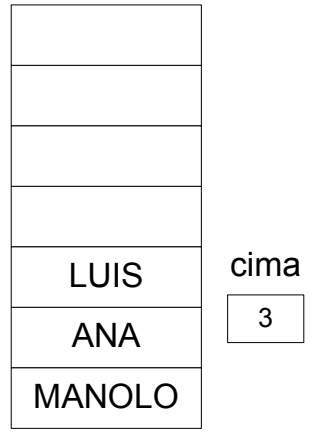



```
Pinsertar(p,'MANOLO')
Pinsertar(p,'ANA')
Pinsertar(p,'LUIS')
```

Realizaciones mediante arrays (IV)

- Procedimiento `Cima`
  - Devuelve el dato de `TipoElemento` situado en la posición `cima`.
  - Es necesario comprobar si la pila contiene elementos (si `cima <> 0`).

```
procedimiento Cima(valor pila : p ;
                   ref TipoElemento : e)
inicio
   si p.cima = 0 entonces
        // Error la pila está vacía
   si_no
        e ← p.el[p.cima]
   fin_si
fin_procedimiento
```

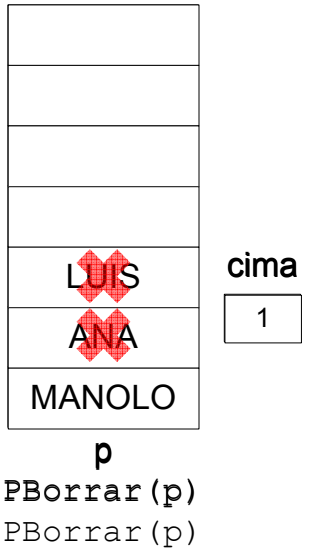



```
Cima(p,nombre)
escribir(nombre)
//Escribe 'LUIS'
```

Realizaciones mediante arrays (V)

- Procedimiento `PBorrar`
  - Elimina el elemento `cima` de la pila
  - Es necesario comprobar si la pila contiene elementos (si `cima <> 0`).

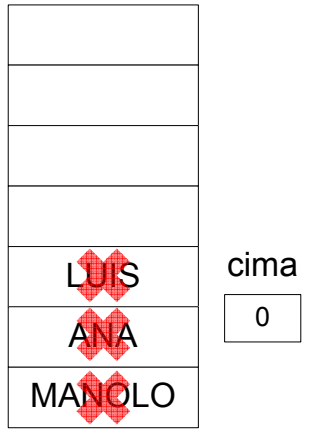
```
procedimiento PBorrar(ref pila : p)
inicio
   si EsPilaVacía(p) entonces
        // Error la pila está vacía
   si_no
        p.cima ← p.cima - 1
   fin_si
fin_procedimiento
```



Realizaciones mediante arrays (VI)

- Procedimiento `Pop (Sacar)`
  - Elimina y devuelve el elemento de `TipoElemento` situado en la `cima` de la pila.
  - Es necesario comprobar si la pila contiene elementos (si `cima <> 0`).

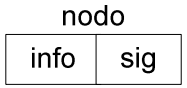
```
procedimiento Pop(ref pila:p; ref TipoElemento:e)
inicio
   si EsPilaVacía(p) entonces
         // Error la pila está vacía
   si_no
         e ← p.el[p.cima]
         p.cima ← p.cima - 1
   fin_si
fin_procedimiento
```




```
Pop(p,nombre)
escribir(nombre)
//escribe 'MANOLO'
```

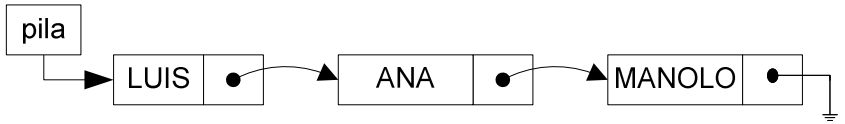
Realización mediante una lista enlazada

- La pila se considera un conjunto de nodos almacenados de forma dinámica.
- Cada elemento del nodo contiene la información (de tipo `TipoElemento`) y un puntero al siguiente elemento.



- La pila está definida por la posición del primer elemento que será por dónde hay que insertar y eliminar información.
  - No es necesario definir el almacenamiento como ocurría con los arrays:
    - ✓ Los nodos se almacenan dispersos por el montículo (memoria dinámica).
  - El dato que define la pila será ese puntero.

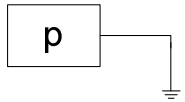
```
tipos
  … = TipoElemento
  puntero_a nodo = pila
  registro = nodo
     TipoElemento : info
     puntero_a nodo : sig
  fin_registro
```



Realización mediante una lista enlazada (II)

- Procedimiento `PilaNueva`

```
procedimiento PilaNueva(ref pila : p)
inicio
   p ← nulo
fin_procedimiento
```




```
PilaNueva(p)
EsPilaVacía(p)
//Devuelve Verdad
```

- Función `EsPilaVacía`

```
lógico: función EsPilaVacía(valor pila:p)
inicio
   devolver(p = nulo)
fin_función
```

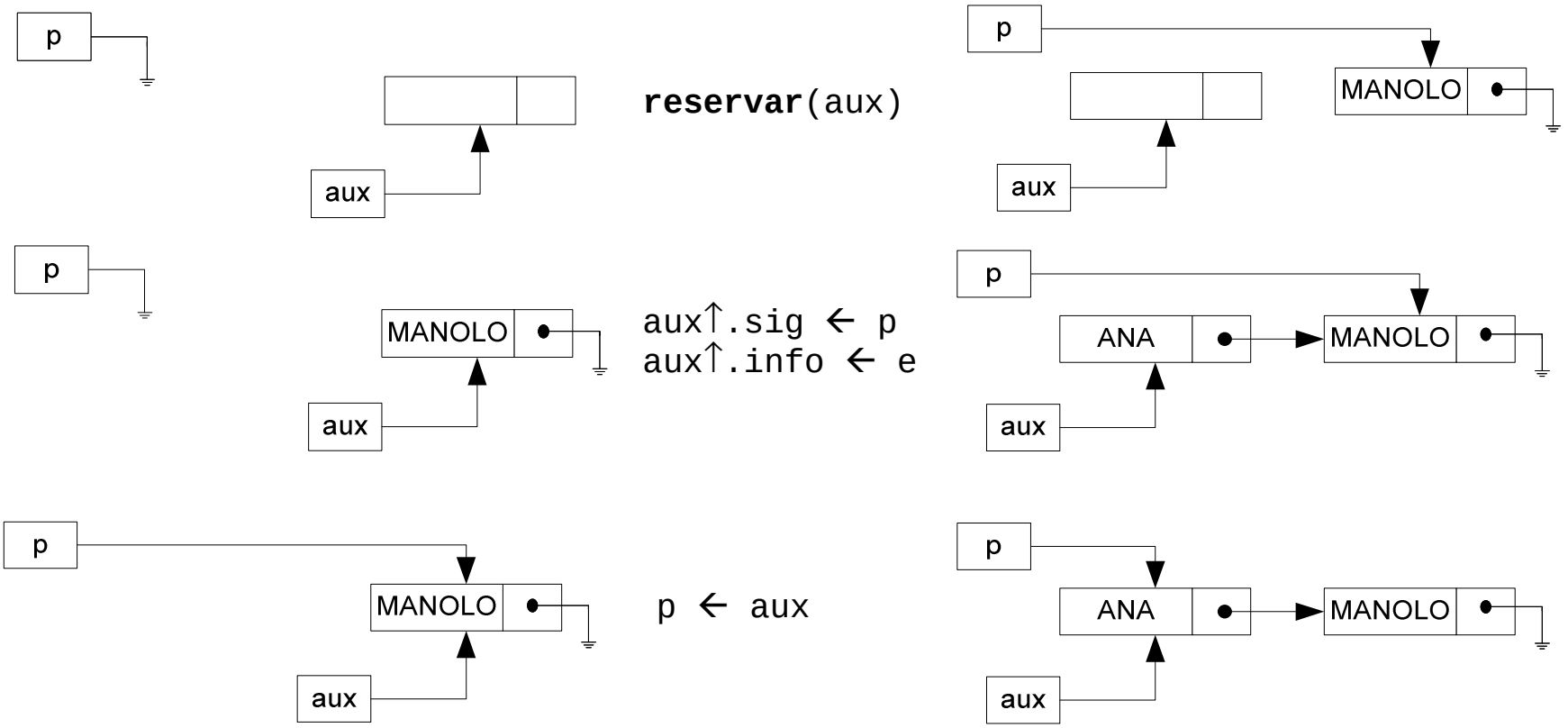
Realización mediante una lista enlazada (III)

- Procedimiento `PInsertar`
  - No es necesario comprobar si hay espacio suficiente para almacenar el elemento.
    - ✓ Se supone que en el montículo (memoria dinámica) siempre habrá espacio.

```
procedimiento PInsertar(ref pila : p  ; valor TipoElemento : e)
var
   pila : aux
inicio
   reservar(aux)
   aux↑.sig ← p
   aux↑.info ← e
   p ← aux
fin_procedimiento
```

Realización mediante una lista enlazada (IV)

```
PInsertar(p,'MANOLO')
PInsertar(p,'ANA')
```



Realización mediante una lista enlazada (V)

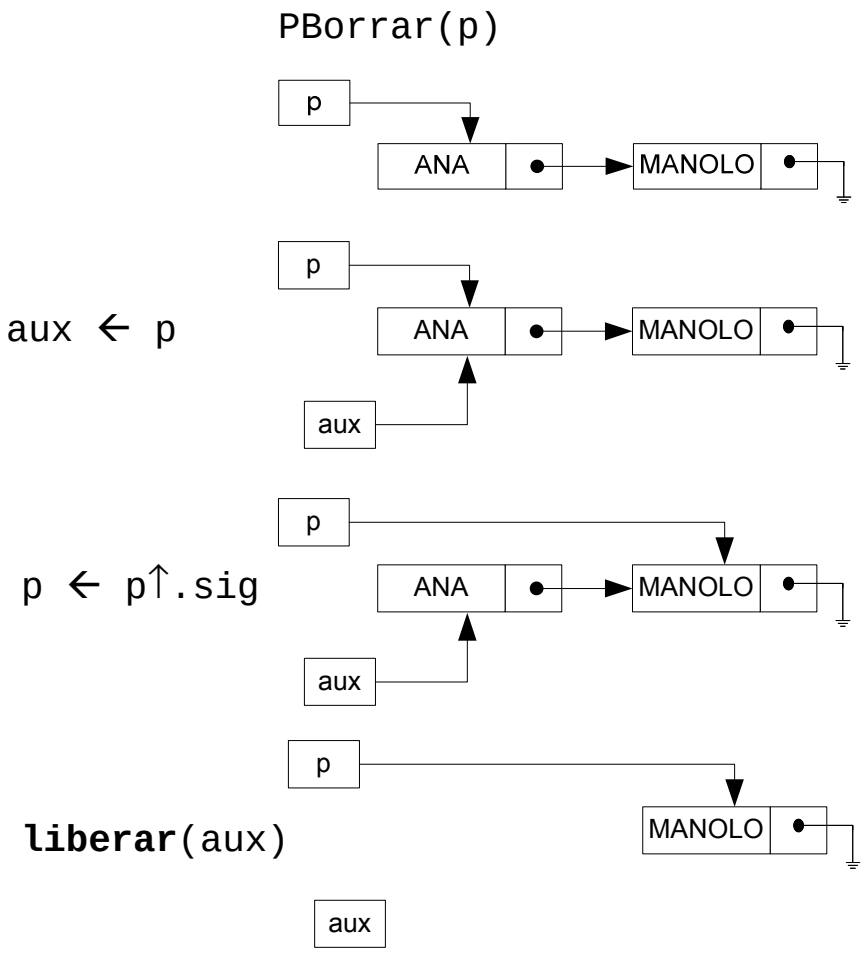
- Procedimiento `Cima`
  - Es necesario comprobar si la pila tiene elementos (p <> **nulo**)

```
procedimiento Cima(valor pila : p  ; ref TipoElemento : e)
inicio
   si p = nulo entonces
        // Error, la pila está vacía
   si_no
        e ← p↑.info
   fin_si
fin_procedimiento
```

Realización mediante una lista enlazada (VI)

- Procedimiento `PBorrar`
  - Es necesario comprobar si la pila tiene elementos (p <> **nulo**)

```
procedimiento PBorrar( ref pila : p)
var
   pila : aux
inicio
   si EsPilaVacía(p) entonces
        // error, la pila está vacía
   si_no
        aux ← p
        p ← p↑.sig
        liberar(aux)
   fin_si
fin_procedimiento
```



Realización mediante una lista enlazada (VII)

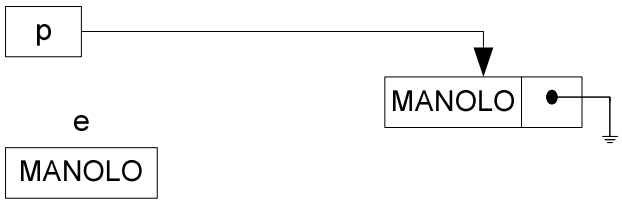
- Procedimiento Pop (Sacar).
  - Es necesario comprobar si la pila tiene elementos (p <> **nulo**)

```
procedimiento Pop( ref pila : p ; ref TipoElemento: e)
var
   pila : aux
inicio
   si EsPilaVacía(p) entonces
      // error, la pila está vacía
   si_no
      e ← p↑.info
      aux ← p
      p ← p↑.sig
      liberar(aux)
   fin_si
fin_procedimiento
```

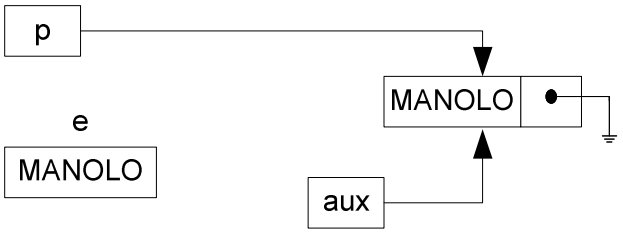
Realización mediante una lista enlazada (VIII)

Pop(p,e)

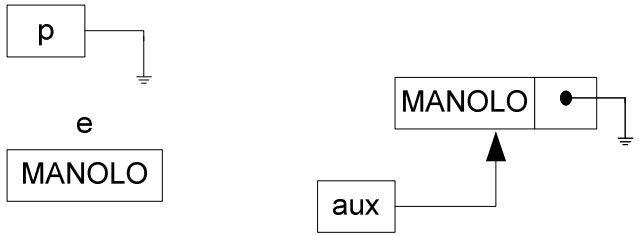
e ← p↑.info



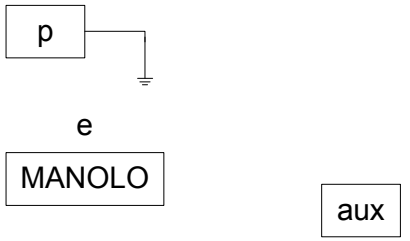
aux ← p



p ← p↑.sig

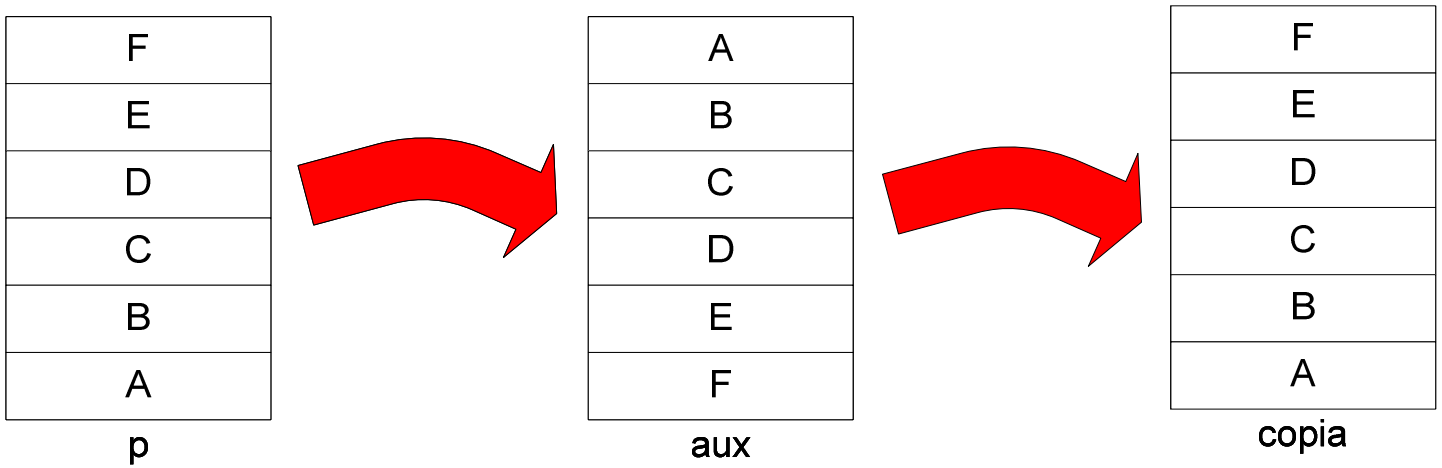


liberar(aux)



Ejemplo 3.1

- Utilizando las operaciones primitivas para trabajar con pilas, diseñe un procedimiento que devuelva una copia de una pila ya creada.
  - Es necesario utilizar una pila auxiliar para copiar los elementos en el mismo orden que la pila original.



Ejemplo 3.1 (II)

- Versión iterativa.

```
procedimiento CopiarPila(ref pila : p,copia)
var
   TipoElemento : e
   Pila : aux
inicio
   //Copia los elementos en orden inverso en una pila auxiliar
   PilaNueva(aux)
   mientras no EsPilaVacia(p) hacer
      Pop(p,e)
      PInsertar(aux,e)
   fin_mientras
   //Restaura los elementos en la pila copia
   PilaNueva(copia)
   mientras no EsPilaVacía(aux) hacer
      Pop(aux,e)
      PInsertar(copia,e)
      PInsertar(p,e)  //Hay que resturar la pila original porque se ha vaciado
   fin_mientras
fin_procedimiento
```

Ejemplo 3.1 (III)

- Versión recursiva.

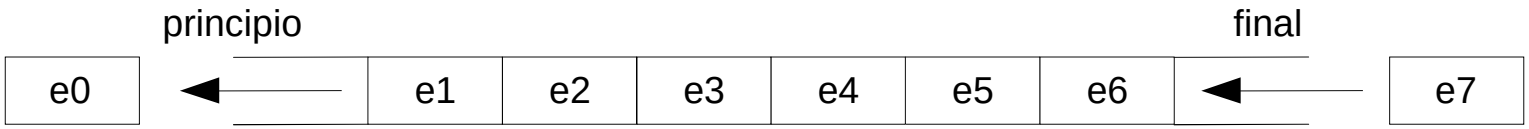
```
procedimiento CopiarPila(ref pila : p,copia)
var
   TipoElemento : e
inicio
   //Si p está vacía estaríamos en el caso trivial
   //Si p está vacía, la copia también es una pila vacía
   si EsPilaVacía(p) entonces
      PilaNueva(copia)
   si_no
      Pop(p,e)
      CopiarPila(p,copia)
      PInsertar(copia,e)
      Pinsertar(p,e)
   fin_si
fin_procedimiento
```

Ejercicios con pilas

1. Diseñe un procedimiento que permita eliminar el elemento n de una pila dejando los restantes elementos tal y como estaban.
2. Diseñe un procedimiento que permita buscar y eliminar un elemento de una pila dejando los restantes elementos tal y como estaban.
3. Realizar una función que devuelva el mayor de una pila de enteros.
4. Realizar un algoritmo que lea un archivo de texto y lo devuelva en otro archivo de texto con las palabras en orden inverso. Se supone que en el texto cada palabra está separada por un único espacio en blanco, con excepción de la última palabra.
5. Realizar un procedimiento QuickSort iterativo.

Colas

- Estructura lineal de datos compuesta por un conjunto de elementos en la que la adición de nuevos elementos se hará por un extremo de la cola, **final** (*rear*), y la salida de elementos por el contrario, **principio** (*front*).
- Estructura de datos de tipo FIFO (*first in-first out*), es decir el primer elemento en entrar es el primero en salir.
- En aplicaciones informáticas se utiliza para controlar procesos que tengan que realizarse en un cierto orden (colas de impresión, colas de prioridades, etc.)



Colas (II)

- La estructura de tipo Cola no se considera implementada en nuestro lenguaje de programación.
  - En otros lenguajes, como .NET, existe la clase Queue (cola en inglés) que implementa colas e incluye todas las operaciones que se pueden hacer sobre las colas.
- Será necesario crear el tipo de dato Cola.
  - Determinar las operaciones básicas que se pueden realizar sobre el tipo de dato Cola: las **operaciones primitivas**.
  - Definir el tipo de elementos que contendrá la cola.
  - Definir la organización de los datos utilizando los datos y estructuras de datos que ofrezca el lenguaje de programación.
  - Implementar las operaciones primitivas para la organización de los datos definida.
    - ✓ Dependiendo de la organización definida, la implementación de las operaciones primitivas variará.

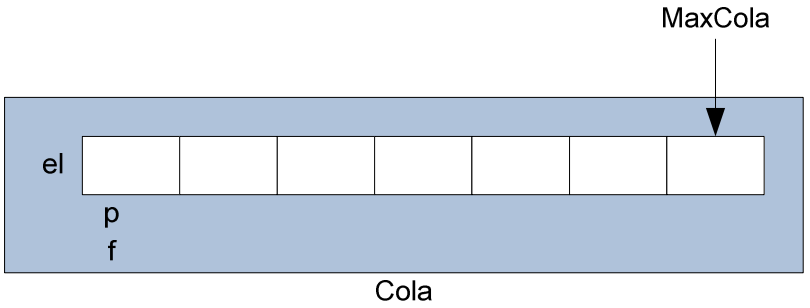
Colas (III)

- Operaciones primitivas.
  - `ColaNueva(`**`ref`** `cola: c)`, inicializa una nueva cola.
  - `EsColaVacía(`**`valor`** `cola :c)`, una función lógica que devuelve `verdad` si la cola está vacía.
  - `CInsertar(`**`ref`** `cola : c;` **`valor`** `tipoElemento: e)`, inserta un elemento de tipo `TipoElemento` en la cola en la posición `final`.
  - `CBorrar(`**`ref`** `cola : c)`, elimina el elemento `principio` de la cola.
  - `Primero(`**`valor`** `cola : c;` **`ref`** `tipoElemento : e)`, obtiene el valor del elemento situado en la posición `principio` de la cola.
  - `Sacar(`**`ref`** `cola : c;` **`ref`** `tipoElemento : e)`, elimina el primer elemento de la cola devolviendo su contenido en `e`.

Realizaciones mediante arrays

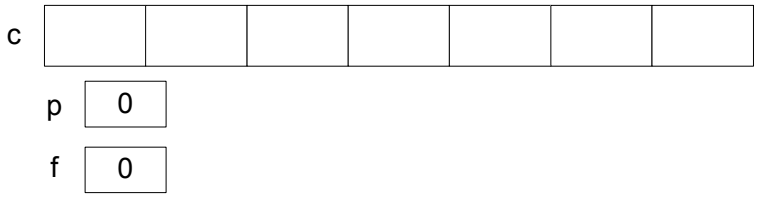
- Declaración de las estructuras de datos.
  - Para implementar una cola es necesario saber cuál es el primer elemento que se ha introducido y cuál es el último.
    - ✓ Se tratará de los enteros `p` y `f`.
  - Si se implementa como un array, es necesario también determinar dónde se almacenarán los elementos.
    - ✓ Se trata del array de elementos `el`.
      - El tipo base del array es el tipo de dato genérico `TipoElemento`.
      - El array tiene un tamaño de `MáxCola` elementos.

```
const
 MaxCola = …
tipos
 … = TipoElemento
 registro = cola
  entero : p,f
  array[1..MaxCola] de TipoElemento : el
 fin_registro
```



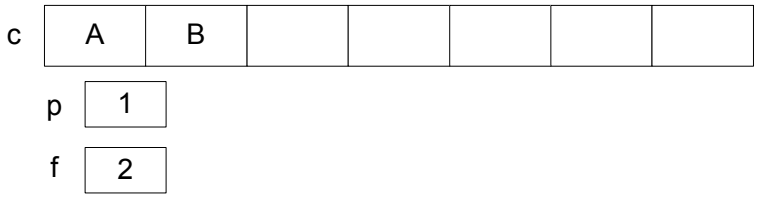
Realizaciones mediante arrays (II)

ColaNueva(c)

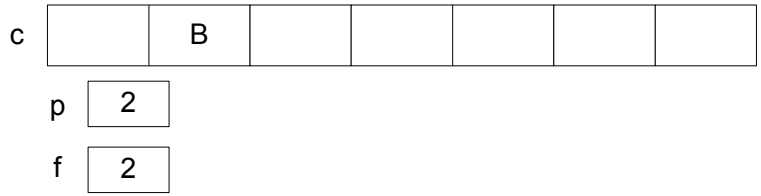


CInsertar(c,'A')

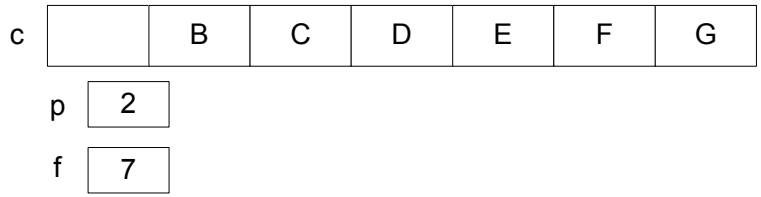
CInsertar(c,'B')



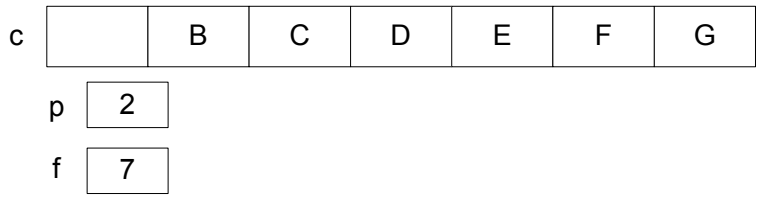
CBorrar(c)



CInsertar(c,'C')
CInsertar(c,'D')
...
CInsertar(c.'G')



CInsertar(c,'H')
//Error, el array se ha agotado



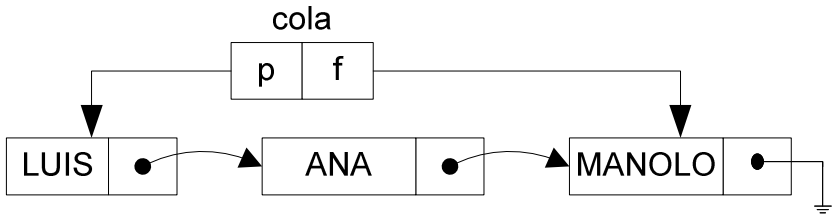
Realizaciones mediante arrays (III)

- Con esta implementación es posible que no se pueda insertar porque el puntero `f` ha llegado al último elemento del array y que, sin embargo, todavía quede sitio.
- Puede haber varias soluciones…
  - Mover todos los elementos hacia adelante cada vez que de borra un elemento.
    - ✓ Siempre se eliminaría el elemento 1 y el primer elemento siempre sería la posición 1.
  - Crear un array circular.
    - ✓ El siguiente elemento a la posición `MaxCola` es el elemento 1.
      - Si hay sitio, al llegar al final del array se seguiría insertando por el elemento 1.

Realización mediante una lista enlazada

- La cola se considera un conjunto de nodos almacenados de forma dinámica (una lista enlazada).
- Cada elemento del nodo contiene la información (de tipo `TipoElemento`) y un puntero al siguiente elemento.
- La cola estará compuesta por las direcciones del primer nodo de la estructura (principio) y el último nodo de la estructura (final).
  - El dato que define la cola es un registro formado por esos dos punteros.

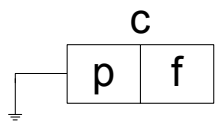
```
tipos
  … = TipoElemento
  registro : cola
     puntero_a nodo : p,f
  fin_registro
  registro = nodo
     TipoElemento : info
     puntero_a nodo : sig
  fin_registro
```



Realización mediante una lista enlazada (II)

- Procedimiento ColaNueva.

```
procedimiento ColaNueva(ref cola : c)
inicio
   c.p ← nulo
   //c.f ← nulo //No sería necesario
fin_procedimiento
```




```
ColaNueva(c)
EsColaVacía(c)
//Devuelve verdad
```

- Función EsColaVacía.

```
lógico: función EsColaVacía(valor cola : c)
inicio
   devolver(c.p = nulo) //o (c.f = nulo)
fin_función
```

Realización mediante una lista enlazada (III)

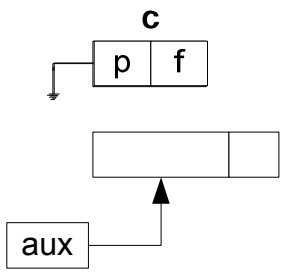
- Procedimiento CInsertar
  - No es necesario comprobar si hay espacio suficiente para almacenar el elemento.
    - ✓ Se supone que, con estructuras dinámicas, el espacio es ilimitado
  - Hay que comprobar si es el primer elemento de la cola.
    - ✓ Si es cierto, el frente de la cola también debe apuntar a ese elemento.
    - ✓ Si no, el último elemento deberá apuntar al nuevo elemento.

```
procedimiento CInsertar(ref cola : c  ; valor TipoElemento : e)
var
    puntero_a nodo : aux
inicio
    reservar(aux)
    aux↑.sig ← nulo
    aux↑.info ← e
    si c.p = nulo entonces
          c.p ← aux
    si_no
          c.f↑.sig ← aux
    fin_si
    c.f ← aux
fin_procedimiento
```

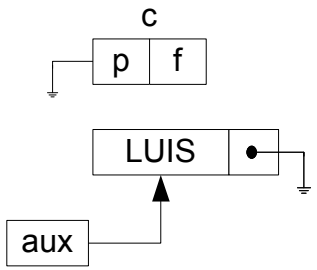
Realización mediante una lista enlazada (IV)

CInsertar(c,'LUIS')

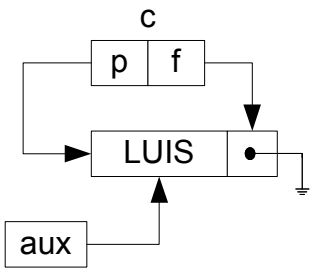
reservar(aux)



aux↑.info ← e
aux↑.sig ← **nulo**

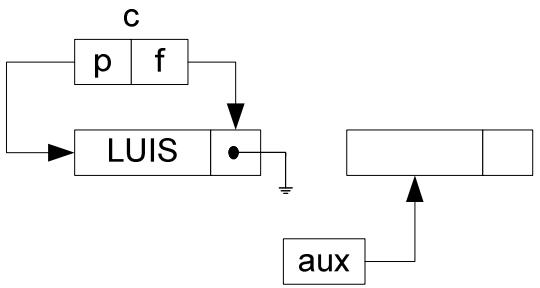


c.p ← aux
c.f ← aux

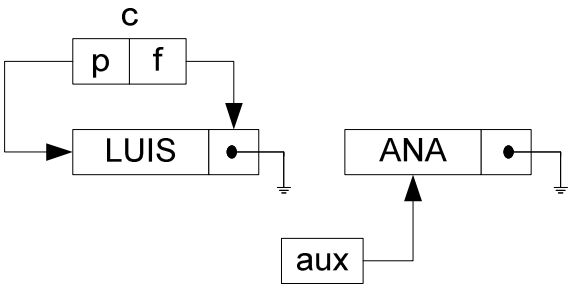


CInsertar(c,'ANA')

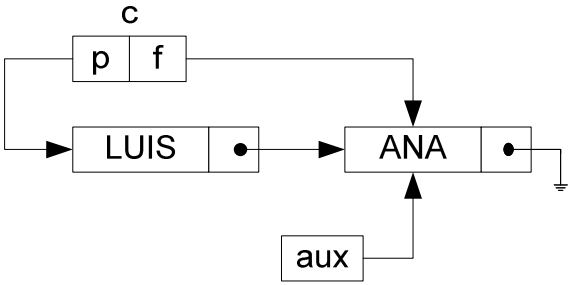
reservar(aux)



aux↑.info ← e
aux↑.sig ← **nulo**



c.f↑.sig ← aux
c.f ← aux



Realización mediante una lista enlazada (V)

- Procedimiento Primero
  - Es necesario comprobar si la pila tiene elementos (`c.p <> nulo`)

```
procedimiento Primero(valor cola : c; ref TipoElemento : e)
inicio
   si c.p = nulo entonces
        // Error, la cola está vacía
   si_no
        e ← c.p↑.info
   fin_si
fin_procedimiento
```

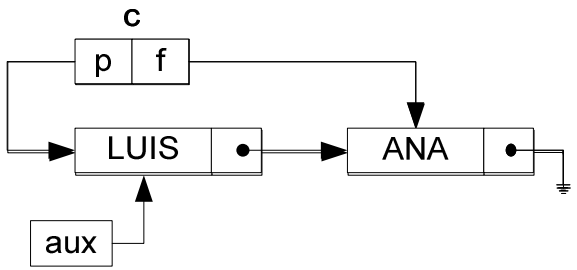
Realización mediante una lista enlazada (VI)

- Procedimiento CBorrar

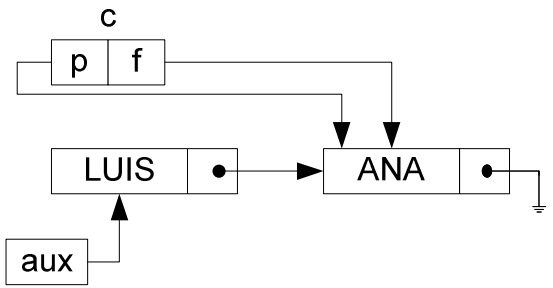
```
procedimiento CBorrar(ref cola : c)
var
   puntero_a nodo : aux
inicio
   si c.p = nulo entonces
        // Error, la cola está vacía
   si_no
        aux ← c.p
        c.p ← c.p↑.sig
        liberar(aux)
fin_procedimiento
```

CBorrar(c)

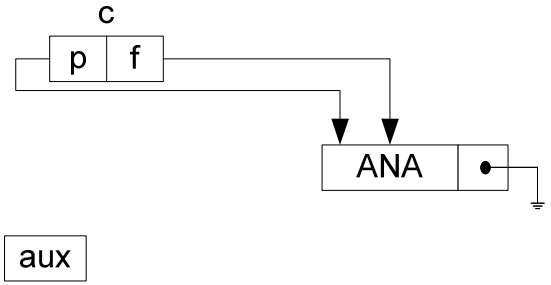
aux ← c.p



c.p← c.p↑.sig



liberar(aux)



Realización mediante una lista enlazada (VII)

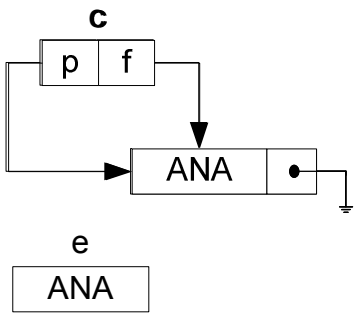
- Procedimiento Sacar.
  - Combina los procedimientos CBorrar y Primero.

```
procedimiento Sacar(ref cola : c ; ref TipoElemento : e)
var
   puntero_a nodo : aux
inicio
   si c.p = nulo entonces
      // Error, la cola está vacía
   si_no
      e ← c.p↑.info
      aux ← c.p
      c.p ← c.p↑.sig
      liberar(aux)
fin_procedimiento
```

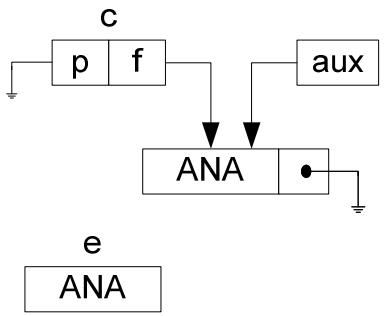
Realización mediante una lista enlazada (VIII)

```
Sacar(c,e)
```

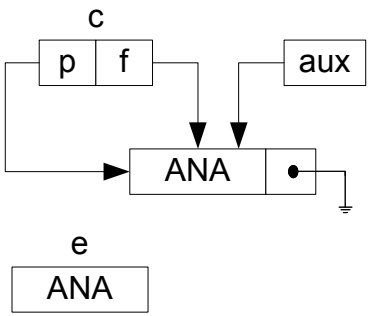
```
e ← c.p↑.info
```



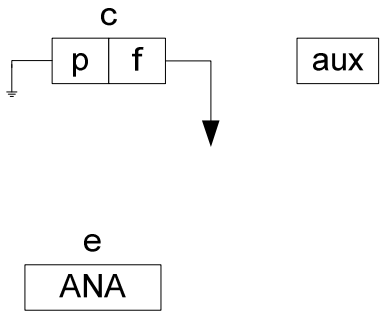

```
c.p ← c.p↑.sig
```




```
aux ← c.p
```



liberar(aux)



Ejemplo 3.2

- Utilizando una pila y una cola, realizar una función que reciba una cadena y devuelva el valor lógico **verdad**, si es un palíndromo.
  - Si se meten todos los caracteres de la cadena al mismo tiempo en una pila y en una cola, al sacarlos de las dos estructuras de datos saldrán en orden inverso: los caracteres de la cola saldrán del primero al último, los de la pila del último al primero. Si todos los caracteres son iguales, se tratará de un palíndromo.
    - ✓ La función **`longitud`**`(cadena)` devuelve el número de caracteres de una cadena.
      - Por ejemplo, **`longitud`**`('hola')` devuelve 4.
    - ✓ Para hacer referencia al carácter `n` de una cadena se utiliza el selector de array.
      - Si la cadena `c` es `'hola'`, `c[2]` sería el carácter `'o'`.

Ejemplo 3.2 (II)

```
lógico función EsPalíndromo(valor cadena : cad)
var
   Pila : p
   Cola : c
   TipoElemento : e1, e2
   entero : i
inicio
   PilaNueva(p)
   ColaNueva(c)
   //Se insertan todos los caracteres de la cadena en la
   //pila y en la cola al mismo tiempo
   desde i ← 1 hasta longitud(cad) hacer
      CInsertar(c,cad[i])
      PInsertar(p,cad[i])
   fin_desde
   //Se sacan los caracteres de la pila y la cola hasta que
   //se encuentra un carácter distinto o hasta que alguna de
   //las estructuras esté vacía
   repetir
      Pop(p,e1)
      Sacar(c,e2)
   hasta_que (e1 <> e2) o EsPilaVacía(p)
   //Los últimos caracteres sacados son iguales, es un palíndromo
   devolver(e1 = e2)
fin_función
```

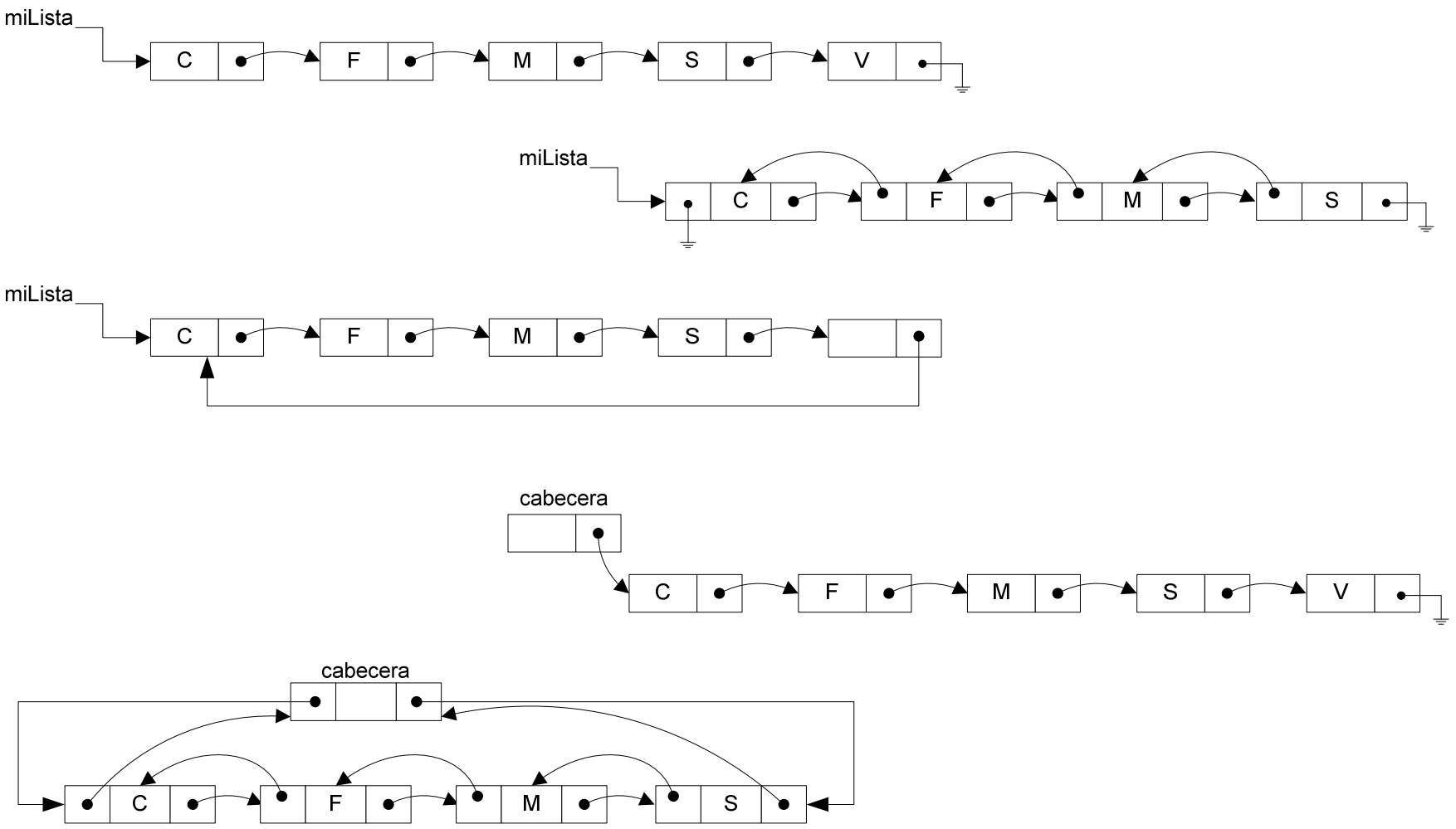
Ejercicios con colas

1. El problema de Josephus.
   - Cuenta una leyenda sobre el historiador judio Josephus Flavius que, durante las guerras judeo-romanas, él y otros 40 soldados judíos quedaron atrapados en una cueva rodeados por los romanos. Visto que tenían pocas posibilidades de salir con vida, decidieron suicidarse. Josephus y un amigo suyo no estaban muy felices con esa idea. Así pues, propusieron que si había que hacerlo, se hiciera con cierto orden: se colocarían en círculo y se irían suicidando por turno cada tres empezando a contar por uno determinado. Josephus y su amigo se colocaron de tal forma que fueron los dos últimos y decidieron seguir viviendo.
   - Realizar un algoritmo que devuelva el orden en que salieron los soldados. El número de soldados será m y el salto será n.
2. Se tiene una cola de procesos. Cada elemento contiene la prioridad (de 1 a 20) y un identificador de proceso único. Se desea hacer un listado con el orden de salida de los procesos.
3. Se tiene una cola de procesos. Cada elemento contiene la prioridad (de 1 a 20) y un identificador de proceso único. Se desea hacer un listado con el orden de salida de los procesos con prioridad p.
4. Se tiene una cola de procesos. Cada elemento contiene información sobre el identificador de proceso único y su prioridad (de 1 a 20). Los elementos de la cola están ordenados por prioridad. Diseñar un procedimiento que permita ordenar un nuevo proceso en la cola.

Listas enlazadas

- Estructura lineal de datos compuesta por un conjunto de nodos que ocupan posiciones no contiguas de memoria.
- Cada nodo contiene la información del componente y, al menos, un puntero que indica la posición del siguiente nodo.
  - En el último nodo, la posición del siguiente elemento será un puntero nulo.
  - Al no ocupar posiciones contiguas en memoria y ser variable la posición del primer nodo de la estructura, la lista tendrá otro puntero que indicará la dirección del primer nodo de la estructura.
- Pilas y colas son un tipo especial de listas enlazadas con la entrada y la salida limitadas.
  - En el caso de las listas enlazadas, las inserciones y eliminaciones se podrán hacer por cualquier punto de la estructura.
- Tipos de listas enlazadas.
  - Listas simplemente enlazadas o listas simples.
  - Listas doblemente enlazadas.
  - Listas circulares.
  - Listas con cabecera.
  - Combinaciones de las anteriores.

Listas enlazadas (II)



Listas enlazadas simples

- Sólo tienen un puntero indicando el siguiente elemento.
- La estructura de datos "Lista enlazada" no se considera implementada en nuestro lenguaje de programación".
  - Otros lenguajes, como .Net tienen la clase LinkedList que implementa listas enlazadas e incluye todas las operaciones que se pueden hacer sobre ellas.
- Será necesario crear el tipo de dato Lista.
  - Determinar las operaciones básicas que se pueden realizar sobre el tipo de dato Lista: las **operaciones primitivas**.
  - Definir el tipo de elementos que contendrá la lista.
  - Definir la organización de los datos utilizando los datos y estructuras de datos que ofrezca el lenguaje de programación.
  - Implementar las operaciones primitivas para la organización de los datos definida.
    - ✓ Dependiendo de la organización definida, la implementación de las operaciones primitivas variará.

Listas enlazadas simples (II)

- Operaciones primitivas.
  - `ListaNueva(`**`ref`** `lista:l)`, convierte la lista l en una lista vacía.
  - `EsListaVacía(`**`valor`** `lista:l)`, devuelve verdad si la lista l está vacía.
  - `LInsertar(`**`ref`** `lista:l;`**`valor`** `tipoElemento:e)`, inserta el elemento e en la lista l.
  - `LBorrar(`**`ref`** `lista:l)`, elimina un elemento de la lista l.
  - `LPrimero(`**`valor`** `lista:l; `**`ref`** `tipoElemento : e)`, devuelve la información del primer elemento de la lista en el argumento e.
  - `LSiguiente(`**`valor`** `lista:l; `**`ref`** `lista:sig)`, devuelve la dirección del siguiente nodo de la lista en el argumento sig.

Realizaciones mediante arrays

- Los datos se almacenarían en un array de nodos.
  - Cada nodo tendría la información y el índice del siguiente elemento de la lista.
  - Los elementos vacíos se marcan, por ejemplo, con -1 en el campo sig.
  - En la memoria pueden coexistir varias listas.
- La lista estaría definida por la dirección del primer nodo de la estructura.

Realizaciones mediante arrays (II)

- Insertar "ANA" en la lista de personas en la primera posición.

animales: 6

personas: 9

	info	sig
1	TORO	0
2	MANUEL	4
3		-1
4	PEPE	0
5	PERRO	1
6	DELFIN	10
7	JUANA	2
8		-1
9	ANA	7
10	GATO	5
11		-1

- Borrar el "GATO" de la lista de animales.

animales: 6

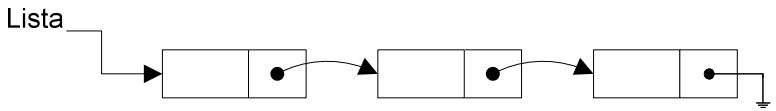
personas: 9

	info	sig
1	TORO	0
2	MANUEL	4
3		-1
4	PEPE	0
5	PERRO	1
6	DELFIN	5
7	JUANA	2
8		-1
9	ANA	7
10	GATO	-1
11		-1

Realizaciones mediante punteros

- Se considerará la lista enlazada como una estructura de datos recursiva.
  - Si la lista está vacía, ésta será un puntero nulo.
  - Si tiene elementos, será un puntero que apunta a un nodo con los campos `info`, con la información del elemento, y `sig` que será otra lista.
  - Cada lista será o bien un puntero nulo o un puntero al siguiente nodo.
- Una lista estará compuesta por un conjunto de listas, la última de la cuales estará vacía.
  - Serán listas, tanto el puntero que indica la dirección del primer elemento como cualquiera de los campo siguiente.

```
tipos
   … = TipoElemento
   puntero_a nodo : Lista
   registro = nodo
      TipoElemento : info
      Lista : sig
   fin_registro
```



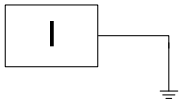
Realizaciones mediante punteros (II)

- Procedimiento `ListaNueva`

```
procedimiento ListaNueva(ref lista : l)
inicio
   l ← nulo
fin_procedimiento
```

- Función `EsListaVacía`

```
lógico función EsListaVacía(valor lista : l)
inicio
   devolver(l = nulo)
fin_función
```




```
ListaNueva(l)
EsListaVacía(l)
//Devuelve Verdad
```

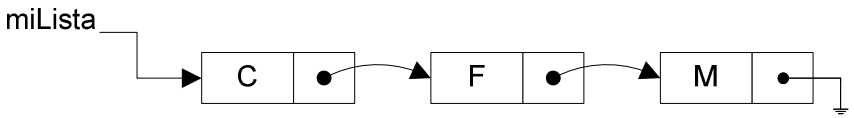
Realizaciones mediante punteros (III)

- Procedimiento `LInsertar`

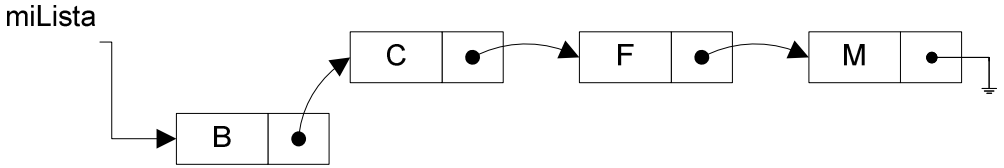
```
procedimiento LInsertar(ref lista : l; valor TipoElemento : e)
var
   lista : aux
inicio
   reservar(aux)
   aux↑.sig ← l
   aux↑.info ← e
   l ← aux
fin_procedimiento
```

- El argumento `l` puede ser tanto puntero de inicio de la lista como cualquier de los campos `sig` de cada nodo.

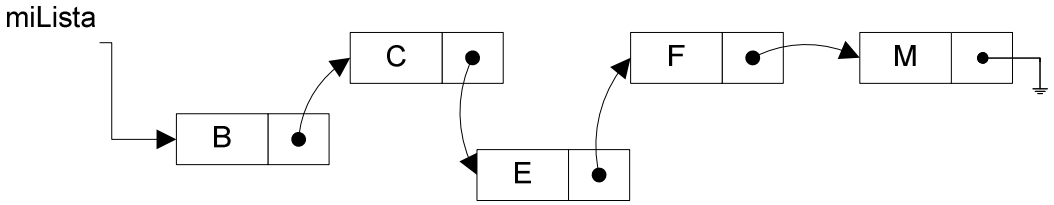
Realizaciones mediante punteros (IV)



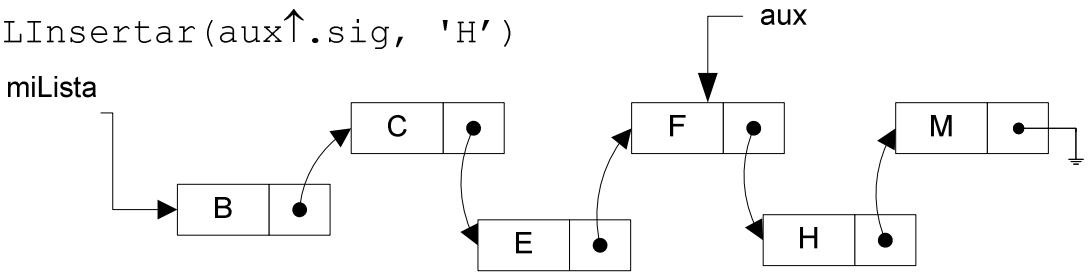

```
LInsertar(miLista, 'B')
```




```
LInsertar(miLista↑.sig↑.sig, 'E')
```



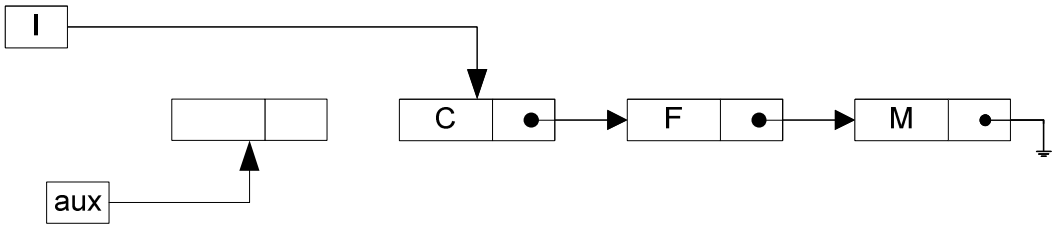

```
LInsertar(aux↑.sig, 'H')
```



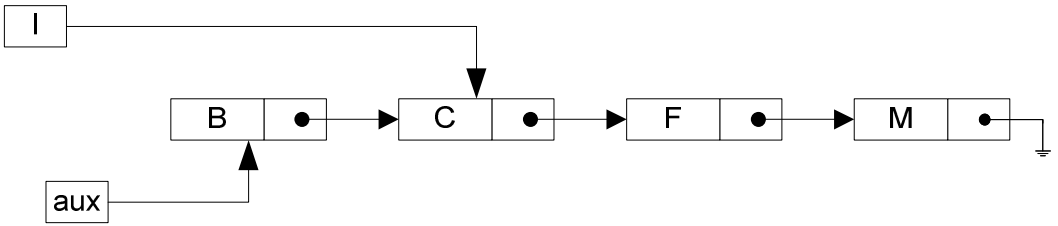
Realizaciones mediante punteros (V)

```
LInsertar(l,'B')
```

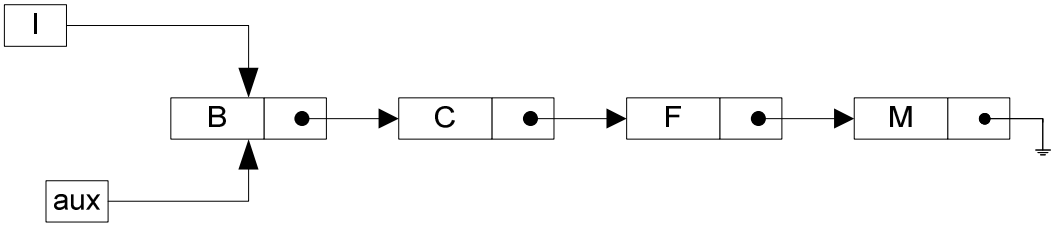
reservar(aux)



aux↑.sig ← l
aux↑.info ← e



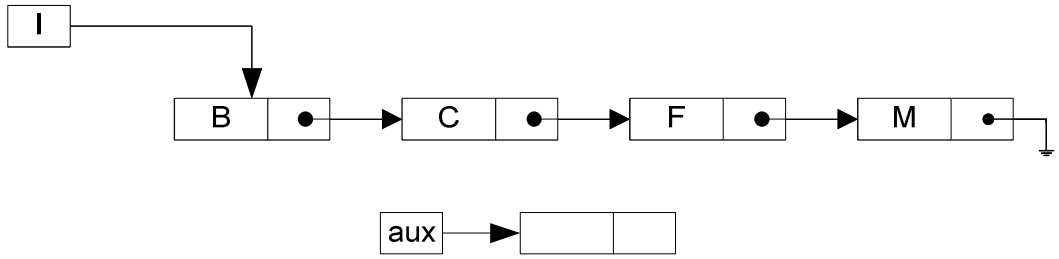
l ← aux



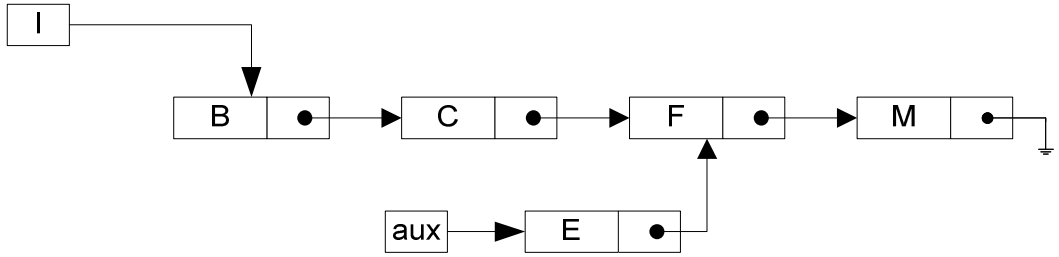
Realizaciones mediante punteros (VI)

```
LInsertar(l↑.sig↑.sig,'E')
```

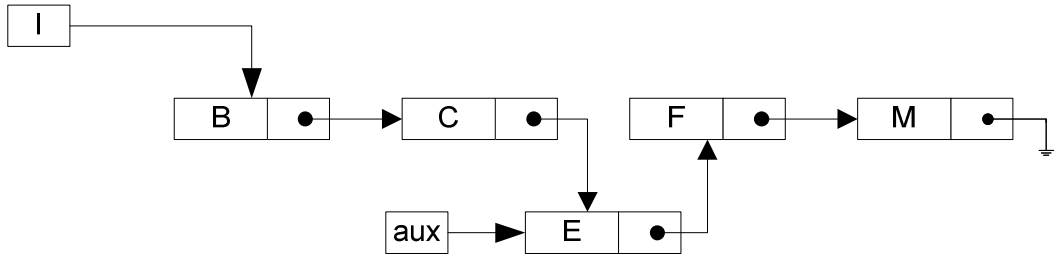
reservar(aux)




```
aux↑.sig ← l
aux↑.info ← e
```




```
l ← aux
```



Realizaciones mediante punteros (VII)

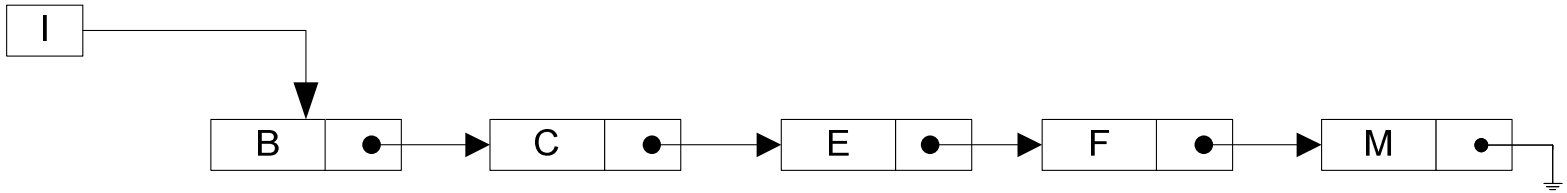
- Procedimiento `LPrimero`.
  - Devuelve la información del primer elemento de la lista.

```
procedimiento LPrimero(valor lista : l; ref TipoElemento : e)
inicio
   si l = nulo entonces
      // Error, la lista está vacía
   si_no
      e ← l↑.info
   fin_si
fin_procedimiento
```

  - Es necesario comprobar si la lista tiene elementos (`l <>` **`nulo`**).
  - El argumento `l`, puede ser tanto el puntero de inicio de la lista como cualquiera de los campos `sig` de cada nodo.

Realizaciones mediante punteros (VIII)

```
LPrimero(l,e)
escribir(e) //Escribe B
LPrimero(l↑.sig,e)
escribir(e) //Escribe C
LPrimero(l↑.sig↑.sig↑.sig,e)
escribir(e) //Escribe F
```



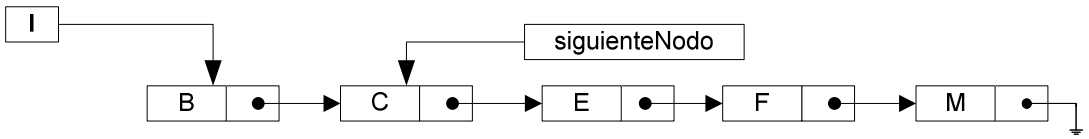
Realizaciones mediante punteros (IX)

- Procedimiento `LSiguiente.`
  - Devuelve la la dirección del siguiente nodo de la lista

```
procedimiento LSiguiente(valor lista : l; ref lista : siguiente)
inicio
   si l = nulo entonces
      // Error, la lista está vacía
   si_no
      siguiente ← l↑.sig
   fin_si
fin_procedimiento
```

  - Es necesario comprobar si la pila tiene elementos (p <> **nulo**).
  - El argumento l, puede ser tanto el puntero de inicio de la lista como cualquiera de los campos siguiente.

`LSiguiente(l↑.sig,siguienteNodo)`



Realizaciones mediante punteros (X)

- Recorrido de la lista.
  - Se utiliza una combinación de los procedimiento `LPrimero` y `LSiguiente`.

```
var
   lista : aux
   TipoElemento : e
...
   aux ← miLista
   mientras no EsListaVacía(aux) hacer
      LPrimero(aux,e)
      escribir(e)
      LSiguiente(aux,aux)
   fin_mientras
```

Realizaciones mediante punteros (XI)

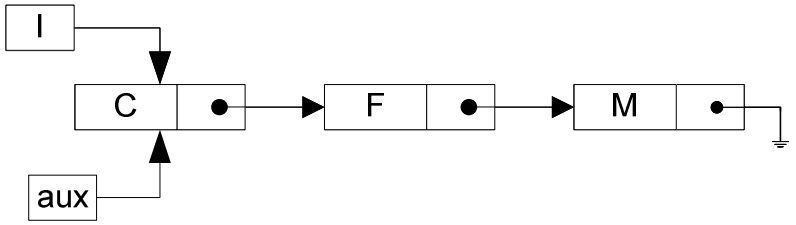
- Procedimiento `LBorrar`
  - Elimina el elemento al que apunta la lista pasada como argumento.

```
procedimiento LBorrar(ref lista : l)
var
   lista : aux
inicio
   si EsListaVacía(l) entonces
      // error, la lista está vacía
   si_no
      aux ← l
      l ← l↑.sig
      liberar(aux)
   fin_si
fin_procedimiento
```

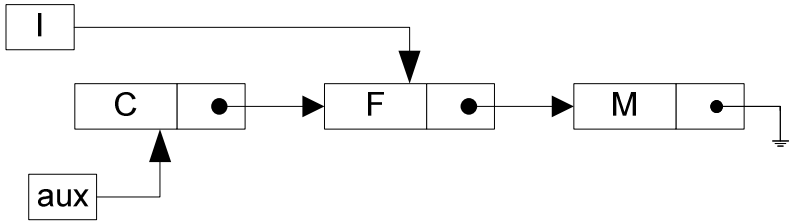
Realizaciones mediante punteros (XII)

```
LBorrar(l)
```

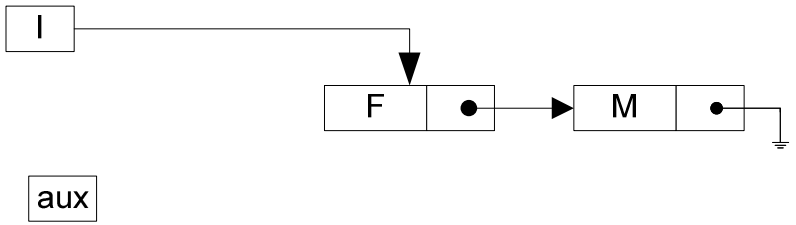
```
aux ← l
```




```
l ← l↑.sig
```



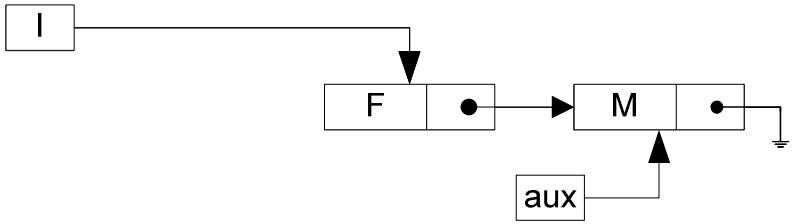

```
liberar(aux)
```



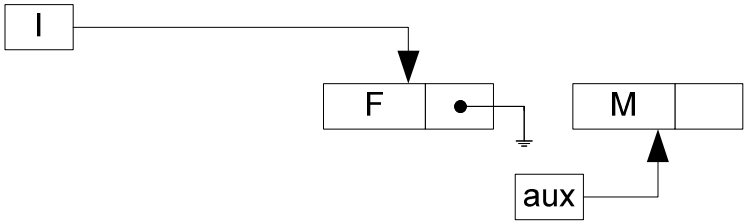
Realizaciones mediante punteros (XIII)

```
LBorrar(l↑.sig)
```

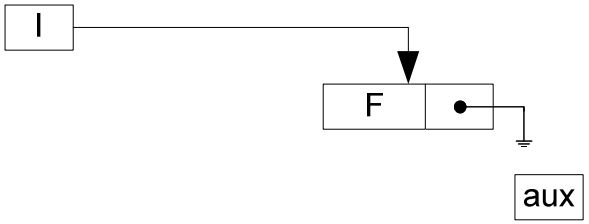
```
aux ← l
```




```
l ← l↑.sig
```



liberar(aux)



Ejemplo 3.3

- Utilizando las operaciones primitivas para trabajar con listas, diseñe las siguientes operaciones básicas:
  a. Un procedimiento que copie una lista.
  b. Una función que devuelva en número de elementos de una lista.
  c. Una función lógica que indique si un elemento pertenece a una lista.
  d. Una función qué busque un elemento en una lista. Si existe devolverá su dirección, en caso contrario devolverá un puntero nulo.
  e. Una función que devuelva la dirección del último nodo de la lista.
  f. Un procedimiento que invierta una lista.

Ejemplo 3.3. (II)

- La copia de una lista sería prácticamente igual que la copia de una pila.

```
procedimiento CopiarLista(valor lista: l ; ref lista:copia)
var
   TipoElemento : e
inicio
   //Si l está vacía estaríamos en el caso trivial
   //Si l está vacía, la copia también es una lista vacía
   si EsListaVacía(l) entonces
      ListaNueva(copia)
   si_no
      LPrimero(l,e)
      LSiguiente(l,l)
      CopiarLista(l,copia)
      LInsertar(copia,e)
   fin_si
fin_procedimiento
```

Ejemplo 3.3. (III)

- La función `Longitud` devuelve el número de elementos de una lista.

```
entero función Longitud(valor lista : l)
var
   entero : conta
inicio
   conta ← 0
   mientras no EsListaVacía(l) hacer
      conta ← conta + 1
      LSiguiente(l,l)
   fin_mientras
   devolver(conta)
fin_función
```

Ejemplo 3.3. (IV)

- La función `Pertenece` devuelve un valor lógico según el elemento pertenezca o no a la lista.

```
lógico función Pertenece(valor lista : l; valor TipoElemento : elem)
//Se supone que sobre el dato TipoElemento se admite
//el operador de igualdad
var
   TipoElemento : e
inicio
   LPrimero(l,e)
   mientras (e <> elem) y no EsListaVacía(l) hacer
      LPrimero(l,e)
      LSiguiente(l,l)
   fin_mientras
   devolver(e = elem)
fin_función
```

Ejemplo 3.3. (V)

- La función `Buscar` devuelve la dirección del nodo que contiene la información que pasamos como argumento, en caso contrario devuelve un valor nulo.

```
lista función Buscar(valor lista : l; valor TipoElemento : elem)
var
   TipoElemento : e
inicio
   LPrimero(l,e)
   mientras (e <> elem) y no EsListaVacía(l) hacer
      LPrimero(l,e)
      LSiguiente(l,l)
   fin_mientras
   si e = elem entonces
      devolver(l)
   si_no
      devolver(nulo)
   fin_si
fin_función
```

Ejemplo 3.3. (VI)

- La función `UltimoNodo` devuelve la dirección del último nodo de una lista o un puntero nulo si está vacía.

```
lista función UltimoNodo(valor lista : l)
var
   lista : ant
inicio
   ant ← nulo
   mientras no EsListaVacía(l) hacer
      ant ← l
      LSiguiente(l,l)
   fin_mientras
   devolver(ant)
fin_función
```

Ejemplo 3.3. (VII)

- La función `UltimoNodo` (versión recursiva, se supone que la lista está implementada con punteros).

```
lista función UltimoNodo(valor lista : l)
inicio
   si l = nulo entonces
      devolver(nulo)
   si_no
      si l↑.sig = nulo entonces
         devolver(l)
      si_no
         devolver(UltimoNodo(l↑.sig))
      fin_si
   fin_si
fin_función
```

Ejemplo 3.3. (VIII)

- El procedimiento `InvertirLista`, devuelve como argumento una copia de la lista en orden inverso.
  - Versión realizando una copia.

```
procedimiento InvertirLista(ref lista :l)
var
   lista : aux
   TipoElemento : e
inicio
   ListaVacía(aux)
   mientras no EsListaVacía(l) hacer
      LPrimero(l,e)
      LInsertar(aux,e)
      LBorrar(l)
   fin_mientras
   l ← aux
fin_procedimiento
```

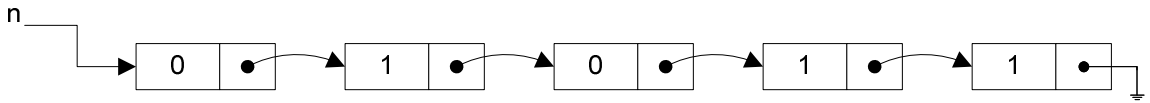
Ejemplo 3.3. (IX)

- Versión sin mover los nodos de su ubicación en memoria (suponiendo que está implementada con punteros).

```
procedimiento InvertirLista(ref lista : l)
var
   lista : act,sig,ant
inicio
   si l <> nulo entonces
      act ← l↑.sig
      ant ← l
      l↑.sig ← nulo
      mientras act <> nulo hacer
         sig ← act↑.sig
         act↑.sig ←ant
         ant ← act
         act ← sig
      fin_mientras
      l ← ant
   fin_si
fin_procdimiento
```

Ejercicios con listas

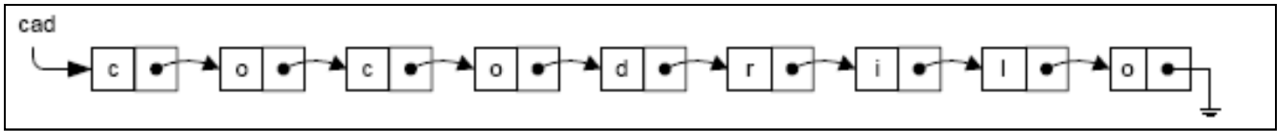
1. Se desea implementar números binarios utilizando listas enlazadas. Cada nodo de la lista almacenará el dígito binario (un 0 o un 1). El dígito menos significativo ocupará la primera posición de la lista. Por ejemplo, si el número n fuera 11010, se almacenaría como:



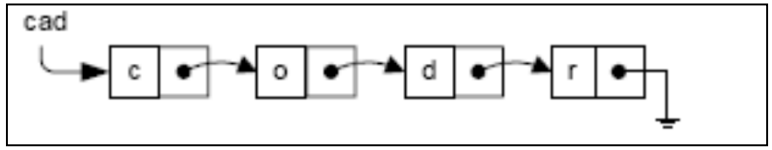
   A. Diseñe las estructuras de datos necesarias para realizar el problema.
   B. Escriba un procedimiento que permita leer un número binario desde teclado y almacenarlo en una lista enlazada de la forma indicada.
   C. Escriba un procedimiento que permita sacar el número por pantalla.
   D. Escriba un procedimiento que reciba dos números binarios almacenados en una lista enlazada y devuelva otra lista enlazada con la suma de ambos.

Ejercicios con listas (II)

2. Se desean implementar cadenas utilizando una lista enlazada. Cada elemento de la lista será un carácter de la cadena. Codifique los métodos adecuados (sin utilizar el tipo de dato estándar) para:
   A. Leer una cadena carácter a carácter hasta que el usuario pulse la tecla Intro.
   B. Escribir una cadena.
   C. Concatenar dos cadenas.
   D. Comparar dos cadenas. Devolverá 0 si son iguales, -1 si la primera cadena es menor que la segunda o 1 si la primera cadena es mayor que la segunda.
   E. Realizar un procedimiento subcadena que devuelva una subcadena de una cadena principal a partir de una posición el número de caracteres indicado. Por ejemplo, si la cadena cad es "cocodrilo"…



con la llamada subcadena(cad,3,4,scad), scad sería "codr" (una subcadena formada por los 4 caracteres a partir del carácter 3):



Listas ordenadas

- Si se requiere que los elementos de la lista estén ordenados por algún criterio, se puede incluir un procedimiento para insertar un elemento ordenado en la lista y mejorar el procedimiento de borrar.
- Insertar Ordenado.

```
procedimiento InsertarOrdenado(ref lista : l; valor TipoElemento : e)
var
   lista : act,ant
   lógico : encontrado
inicio
   encontrado← falso
   act ← l
```

Listas ordenadas (II)

```
    mientras no encontrado y (act <> nulo) hacer
      si e <= act↑.info entonces
         encontrado ← verdad
      si_no
            ant ← act
            act ← act↑.sig
      fin_si
   fin_mientras
   si act = l entonces
      LInsertar(l,e)
   si_no
      LInsertar(ant↑.sig,e)
   fin_si
fin_prodedimiento
```

Listas ordenadas (III)

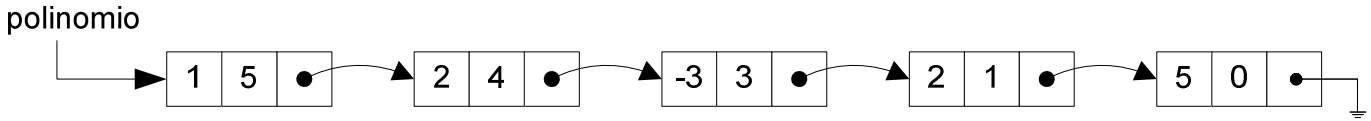
- Para borrar un elemento…

```
procedimiento BorrarElemento(ref lista : l; valor TipoElemento : e)
var
   lista : act,ant
   lógico : encontrado
inicio
   encontrado← falso
   act ← l
   mientras no encontrado y (act <> nulo) hacer
      si e <= act↑.info entonces
         encontrado ← verdad
      si_no
            ant ← act
            act ← act↑.sig
      fin_si
   fin_mientras
   si e = act↑.info entonces
      si act = l entonces
         LBorrar(l)
      si_no
         LBorrar(ant↑.sig)
      fin_si
   fin_si
fin_prodedimiento
```

Ejercicios con listas ordenadas

1. Se desean implementar conjuntos utilizando una lista enlazada. Los elementos del conjunto estarán ordenados de forma ascendente y no tendrán repeticiones. Implemente módulos que permitan incluir un elemento en el conjunto, averiguar si un elemento pertenece a un conjunto, obtener el conjunto unión, el conjunto intersección y el conjunto diferencia.
2. Se desea implementar polinomios mediante listas enlazadas. Cada elemento será uno monomio con el grado y el coeficiente. Los elementos se almacenarán en la lista ordenados de mayor a menor por grado de los monomios.

$$X^5+2x^4-3x^3+2x+5$$



Se desea diseñar un módulo que lea un polinomio, otro que lo escriba y otro que realice la suma de polinomios

Ejercicios con listas ordenadas (II)

3. Una empresa tiene almacenada en una lista enlazada información sobre los productos que guarda en el almacén. Por cada producto se almacena:
   - Código de producto (tipo cadena)
   - Stock (tipo entero)
   - Stock mínimo (tipo entero)
   - Código del proveedor que distribuye el producto (tipo cadena).

   La información de la lista ya está cargada y está ordenada por el código de producto.

   A. Declare las estructuras de datos necesarias para realizar todas las operaciones descritas a continuación:
   B. Desarrolle un módulo que copie en otra lista todos los artículos cuyo stock sea inferior al stock mínimo. La nueva lista estará ordenada por el código de producto.
   C. Desarrolle un módulo que elimine de la lista todos los elementos cuyo código de producto sea mayor que '100'.