Estructuras de datos básicas

Secuencias

Xavier Sáez Pous

PID_00146766

Material docente de la UOC

Universitat Oberta de Catalunya

www.uoc.edu

Primera edición: septiembre 2010
© Xavier Sáez Pous
Todos los derechos reservados
© de esta edición, FUOC, 2010
Av. Tibidabo, 39-43, 08035 Barcelona
Diseño: Manel Andreu
Realización editorial: Eureca Media, SL
ISBN: 978-84-693-4239-8
Depósito legal: B-33.175-2010

Ninguna parte de esta publicación, incluido el diseño general y de la cubierta, puede ser copiada, reproducida, almacenada o transmitido de ninguna manera ni por ningún medio, tanto eléctrico como químico, mecánico, óptico, de grabación, de fotocopia, o por otros métodos, sin la autorización previa por escrito de los titulares del copyright.de los titulares del copyright.

Índice

Introducción 5

Objetivos 7

1. Pilas 9
- 1.1. Representación 9
- 1.2. Operaciones 10
- 1.3. Implementación 12
- 1.4. Ejemplo 14

2. Colas 16
- 2.1. Representación 16
- 2.2. Operaciones 17
- 2.3. Implementación 18
- 2.4. Ejemplo 21

3. Listas 23
- 3.1. Representación 23
- 3.2. Operaciones 24
- 3.3. Implementación 25
  - 3.3.1. Implementación secuencial 25
  - 3.3.2. Implementación encadenada 28
- 3.4. Ejemplo 33

4. Punteros 35
- 4.1. Problemas de los vectores 35
- 4.2. La alternativa: punteros 35
- 4.3. Implementación 37
  - 4.3.1. Pila 37
  - 4.3.2. Cola 39
  - 4.3.3. Lista 41
- 4.4. Peligros de los punteros 43
  - 4.4.1. La memoria dinámica no es infinita 43
  - 4.4.2. El funcionamiento del TAD depende de la representación escogida 44
  - 4.4.3. Efectos laterales 48
  - 4.4.4. Referencias colgadas 49
  - 4.4.5. Retales 49
  - 4.4.6. Conclusión 51
- 4.5. Ejemplo de implementación definitiva: lista encadenada 52

4.6. Otras variantes 55
4.6.1. Listas circulares 55
4.6.2. Listas doblemente encadenadas 56
4.6.3. Listas ordenadas 56

Resumen 59

Actividades 60

Ejercicios de autoevaluación 62

Solucionario 63

Glosario 64

Bibliografía 66

Introducción

Una **secuencia** es un conjunto de elementos dispuestos en un orden específico. Fruto de esta ordenación, dado un elemento de la secuencia, hablaremos del *predecesor* (como el elemento anterior) y del *sucesor* (como el elemento siguiente).

> Dada la secuencia $<v_0\ v_1\ ...\ v_i\ v_{i+1}\ ...\ v_n>$, se dice que el elemento v_{i+1} es el **sucesor** del elemento v_i y que el elemento v_i es el **predecesor** del elemento v_{i+1}. De la misma manera, diremos que v_0 es el **primer** elemento de la secuencia y que v_n es el **último**.

La mayoría de algoritmos que desarrollaremos se centrarán en la repetición de un conjunto de acciones sobre una secuencia de datos. Ahí radica la importancia de saber trabajar con las secuencias. Sin ir más lejos, en asignaturas anteriores ya habéis estudiado los esquemas de recorrido y búsqueda en una secuencia.

Pero además, aparte de la importancia del tratamiento de las secuencias, en este módulo averiguaremos cómo se almacenan de manera que posteriormente podamos acceder a ellas secuencialmente.

> Un **tipo de datos** consta de un conjunto de valores y de una serie de operaciones que pueden aplicarse a esos valores. Las operaciones deben cumplir ciertas propiedades que determinarán su comportamiento.

Las operaciones necesarias para trabajar con una secuencia son:

- Crear la secuencia vacía.
- Insertar un elemento dentro de la secuencia.
- Borrar un elemento de la secuencia.
- Consultar un elemento de la secuencia.

El comportamiento que se establezca para cada una de las operaciones (¿dónde se inserta un elemento nuevo? ¿qué elemento se borra? ¿qué elemento se puede consultar?) definirá el **tipo de datos** que necesitaremos.

Un **tipo abstracto de datos** (abreviadamente **TAD**) es un **tipo de datos** al cual se ha añadido el concepto de **abstracción** para indicar que la implementación del tipo es invisible para los usuarios del tipo.

Un TAD cualquiera constará de dos partes:

- **Especificación.** En la que se definirá completamente y sin ambigüedades el comportamiento de las operaciones del tipo.

- **Implementación.** En la que se decidirá una representación para los valores del tipo y se codificarán las operaciones a partir de esta representación. Dado un tipo, podemos hallar varias implementaciones, cada una de las cuales deberá seguir la especificación del tipo.

De este modo, el único conocimiento necesario para usar un TAD es la *especificación*, ya que explica las propiedades o el comportamiento de las operaciones del tipo.

Las operaciones del TAD se dividirán en **constructoras** (devuelven un valor del mismo tipo) y **consultoras** (devuelven un valor de otro tipo). Dentro de las operaciones constructoras, se denominarán **generadoras** las que formen parte del conjunto mínimo de operaciones necesarias para generar cualquier valor del tipo, mientras que el resto se denominarán **modificadoras**.

En este módulo nos centraremos en los tres TAD más típicos para representar las secuencias:

- Pilas
- Colas
- Listas

La explicación de cada tipo seguirá la misma estructura. Primero, se presentará el tipo de una manera intuitiva para entender el concepto. A continuación, se describirá el comportamiento de las operaciones del tipo de una manera formal. Y, para acabar, se desarrollará una implementación del tipo acompañada de algún ejemplo para ver su uso.

Objetivos

Al finalizar este módulo habréis alcanzado los objetivos siguientes:

1. Conocer los TAD *pila*, *cola* y *lista*; y saber qué propiedades los diferencian.
2. Dado un problema, decidir cuál es el TAD más adecuado para almacenar los datos y saberlo usar.
3. Ser capaz de implementar cualquiera de los TAD presentados en el módulo, mediante el uso de vectores o punteros. Ser también capaz de implementar nuevas operaciones en estos TAD.
4. Entender los costes (temporales y espaciales) que se derivan de las diferentes implementaciones de un TAD, y valorar según estos criterios cuál es la mejor implementación en cada situación.
5. Codificar algoritmos mediante el uso de los TAD presentados.

1. Pilas

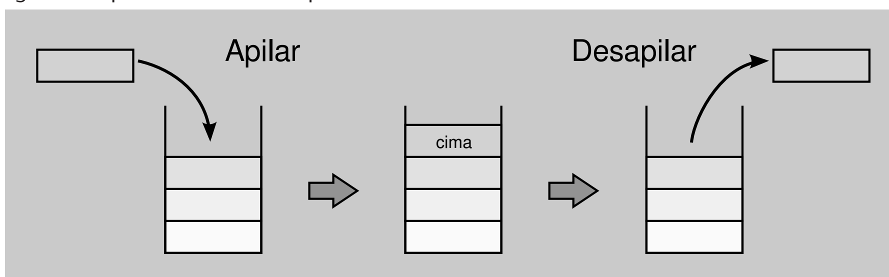
1.1. Representación

Una **pila** es un TAD que se caracteriza por el hecho que el último elemento en entrar es el primero en salir.

En inglés una pila se suele denominar con las siglas LIFO (*last in first out*).

La definición de pila nos indica que todas las operaciones trabajan sobre el mismo extremo de la secuencia. En otras palabras, los elementos se sacan de la estructura en el orden inverso al orden en que se han insertado, ya que el único elemento de la secuencia que se puede obtener es el último (figura 1).

Figura 1. Representación de una pila



Ejemplos de pilas

En nuestra vida diaria podemos ver este comportamiento muy a menudo. Por ejemplo, si apilamos los platos de una vajilla, únicamente podremos coger el último plato añadido a la pila, porque cualquier intento de coger un plato del medio de la pila (como el plato oscuro de la figura 2) acabará en un destrozo. Otro ejemplo lo tenemos en los juegos de cartas, en los que generalmente robamos las cartas (de una en una) de la parte superior del mazo (figura 2).

Figura 2. Pila de platos y de cartas

En el mundo informático también encontramos pilas, como por ejemplo, en los navegadores web. Cada vez que accedemos a una nueva página, el navegador la añade a una

pila de páginas visitadas, de manera que cuando seleccionamos la opción `Anterior`, el navegador coge la página que se encuentra en la cima de la pila, porque es justamente la última página visitada.

Otro ejemplo lo tenemos en los procesadores de textos, en los que los cambios introducidos en el texto también se almacenan en una pila. Cada vez que apretamos la combinación de teclas `Ctrl+Z` deshacemos el último cambio introducido, mientras que cada vez que apretamos la combinación `Ctrl+Y` volvemos a añadir a la pila el último cambio deshecho.

1.2. Operaciones

En la tabla 1 tenéis las operaciones básicas para trabajar con pilas.

Tabla 1

Nombre	Descripción
crear	Crea una pila vacía
apilar	Inserta un elemento en la pila
desapilar	Extrae el elemento situado en la cima de la pila
cima	Devuelve el elemento situado en la cima de la pila
vacia	Devuelve *cierto* si la pila está vacía y *falso* en caso contrario

Las operaciones del TAD *pila* se clasifican en:

- **Operaciones constructoras:** `crear`, `apilar` y `desapilar`.
- **Operaciones consultoras:** `cima` y `vacia`.

Recordad

Recordad que las operaciones constructoras son las operaciones que modifican el estado de la pila, mientras que las operaciones consultoras son las que consultan el estado de la pila sin modificarla.

El **estado de una pila** está definido por los elementos que contiene la pila y por el orden en que están almacenados. Todo estado es el resultado de una secuencia de llamadas a operaciones constructoras.

Igualmente, las operaciones constructoras se clasifican en:

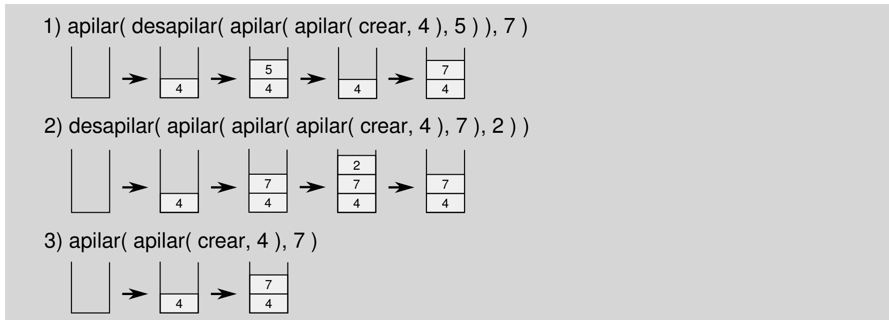
- **Operaciones generadoras:** `crear` y `apilar`, porque son imprescindibles para conseguir cualquier estado de la pila.

- **Operaciones modificadoras:** `desapilar`, porque modifica el estado de la pila extrayendo el elemento de la cima, pero no es una operación imprescindible para construir una pila.

Dado un estado de la pila, se puede llegar a él a través de diversas secuencias de llamadas a operaciones constructoras, pero de entre todas las secuencias **sólo habrá una que esté formada exclusivamente por operaciones generadoras** (figura 3).

Observad que la tercera secuencia de llamadas de la figura 3 es la mínima para llegar al estado deseado, ya que no podemos eliminar ninguna de las llama-

Figura 3. Ejemplos de secuencias de operaciones que producen una pila con el estado <4,7>.



das que la forman. Por lo tanto, tal como se ha mencionando anteriormente, apilar y crear son las operaciones generadoras.

> Una vez establecidas las operaciones del tipo, hay que especificar su comportamiento de una manera formal mediante la **signatura**.

A continuación introducimos la **signatura** del tipo *pila*, que establece, para cada operación, cuáles son los parámetros, el resultado y también las ecuaciones que reflejan su comportamiento y las condiciones bajo las cuales la operación puede provocar errores.

tipo pila (elem)
  operaciones
    crear: → pila
    apilar: pila elem → pila
    desapilar: pila → pila
    cima: pila → elem
    vacia: pila → booleano
  errores
    desapilar(crear)
    cima(crear)
  ecuaciones $\forall p \in pila;\ \forall e \in elem$
    desapilar(apilar(p,e)) = p
    cima(apilar(p,e)) = e
    vacia(crear) = cierto
    vacia(apilar(p,e)) = falso
ftipo

En la signatura del tipo,

- Hablamos de **pila(elem)** para denotar que la pila almacena elementos del tipo genérico *elem*.

- Las secciones **errores** y **ecuaciones** definen el comportamiento de las operaciones y cubren todos los casos posibles: ante una pila vacía (representada por `crear`) o ante una pila no vacía (representada por `apilar(p,e)`, ya que la pila resultante contiene al menos un elemento).

- La sección **ecuaciones** explica el comportamiento normal de las operaciones; por ejemplo: el resultado de desapilar un elemento de una pila es la pila que había antes de apilar el último elemento, la cima de una pila es el último elemento apilado y una pila solo está vacía si no hay ningún elemento apilado.

- La sección **errores** determina qué situaciones producen errores en las operaciones. En el caso de las pilas, extraer o consultar la cima de una pila vacía produce un error.

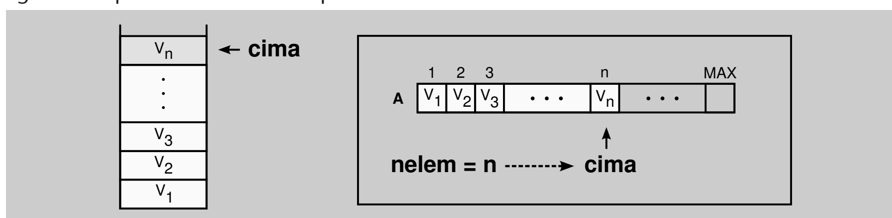
Las **funciones parciales** son las funciones que no están definidas para todo el dominio de los parámetros de entrada. En el caso de las pilas, las operaciones `desapilar` y `cima` son funciones parciales, porque hay situaciones en que producen errores y, por lo tanto, no están definidas.

1.3. Implementación

La implementación más sencilla de una pila es mediante el uso de un **vector** de elementos. Como los vectores no tienen una dimensión infinita, será necesario definir el número máximo de elementos (`MAX`) que podrá almacenar, y también será necesario añadir una operación (`llena`) para comprobar si la pila está llena antes de apilar un nuevo elemento. De este modo evitaremos el error de apilar un elemento cuando ya no quede espacio libre en el vector.

La figura 4 muestra la representación de una pila mediante un vector. El primer elemento de la pila se almacena en la primera posición del vector, el segundo elemento se almacena en la segunda posición, y así sucesivamente hasta llegar al último elemento.

Figura 4. Implementación de una pila con un vector



La implementación incluirá, aparte del vector donde se guardan los elementos (`A`), un atributo entero (`nelem`) que nos indicará el número de elementos que hay en la pila en todo momento e, implícitamente, la posición del vector

donde se halla la cima de la pila. Este apuntador también dividirá el vector en dos partes: parte ocupada y parte libre.

```
tipo
  pila = tupla
           A : tabla [MAX] de elem;
           nelem : entero;
  ftupla
ftipo
```

Dada la representación del tipo, el siguiente paso es implementar las operaciones del tipo pila:

```
funcion crear() : pila
  var p : pila fvar
  p.nelem := 0;
  devuelve p;
ffuncion

funcion apilar(p : pila; e : elem) : pila
  si p.nelem = MAX entonces
      error {pila llena};
  sino
       p.nelem := p.nelem + 1;
       p.A[p.nelem] := e;
  fsi
  devuelve p;
ffuncion

funcion desapilar(p : pila) : pila
  si p.nelem = 0 entonces
      error {pila vacia};
  sino
       p.nelem := p.nelem - 1;
  fsi
  devuelve p;
ffuncion

funcion cima(p : pila) : elem
  var e : elem fvar
  si p.nelem = 0 entonces
      error {pila vacia};
  sino
       e := p.A[p.nelem];
  fsi
  devuelve e;
ffuncion
```

<u>funcion</u> vacia(*p* : **pila**) : **booleano**
  <u>devuelve</u> *p.nelem* = 0;
<u>ffuncion</u>

<u>funcion</u> llena(*p* : **pila**) : **booleano**
  <u>devuelve</u> *p.nelem* = *MAX*;
<u>ffuncion</u>

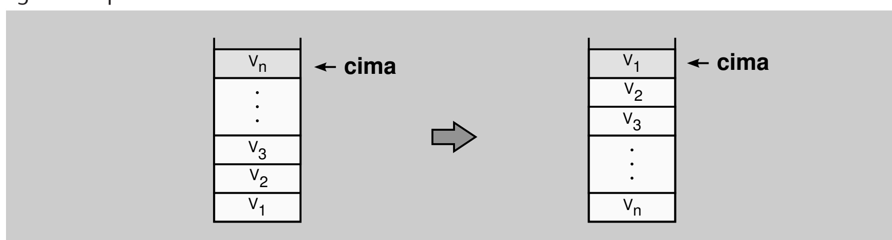
Queda por hacer un análisis aproximado de la eficiencia temporal y espacial de la implementación para valorar su validez. Para hacerlo, se calculará el coste asintótico, Θ, de cada operación.

El **coste temporal** de las operaciones es óptimo, Θ(1), ya que no hay ningún bucle en el cuerpo de las funciones. Por otro lado, el **coste espacial** es bastante pobre, Θ(*MAX*), ya que el vector reserva un espacio (concretamente, *MAX* posiciones) independientemente del número de elementos que almacene la pila en un momento dado.

1.4. Ejemplo

Se pide definir una nueva operación que amplíe el TAD pila. La operación se denominará ***invertir*** *y dará la vuelta al contenido de la pila; es decir, el primer elemento de la pila pasará a ser el último, el segundo pasará a ser el penúltimo, y así sucesivamente (figura 5).*

Figura 5. Operación `invertir`



Con el objetivo de implementar la nueva operación, aprovecharemos el comportamiento **LIFO** de las pilas* para leer los elementos de la pila en orden inverso de llegada. Así, bastará con desapilar cada elemento de la pila origen y, a continuación, apilarlo en la pila resultado. De este modo obtendremos fácilmente una pila en la que los elementos estarán en orden inverso al que tenían en la pila original.

***El último elemento que entra es el primero en salir.**

El algoritmo de la operación **invertir** constará de los siguientes pasos:

1) Crear la pila resultado en la que se apilaran los elementos.
2) Leer el elemento de la cima de la pila origen y apilarlo en la pila resultado.

3) Desapilar el elemento acabado de leer de la cima de la pila origen.
4) Volver al punto 2 mientras la pila origen no esté vacía.

Finalmente, implementamos la operación:

```
funcion invertir(p : pila) : pila
   var pinv : pila fvar
   pinv := crear();
   mientras ¬vacia(p) hacer
            pinv := apilar(pinv,cima(p));
            p := desapilar(p);
   fmientras
   devuelve pinv;
ffuncion
```

2. Colas

2.1. Representación

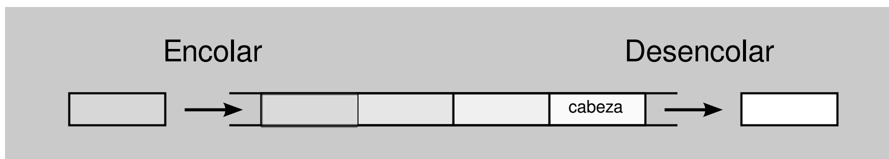
Las colas son el segundo TAD que estudiaremos para representar las secuencias. Básicamente, las colas se diferencian de las pilas en la extracción de los datos.

> Una **cola** es un TAD caracterizado por el hecho que el primer elemento en entrar es el primero en salir.

En inglés, una cola suele denominarse con las siglas FIFO (*first in first out*).

La definición de cola nos indica que las operaciones trabajan sobre ambos extremos de la secuencia: un extremo para añadir los elementos y el otro para consultarlos o extraerlos. En otras palabras, los elementos se extraen en el mismo orden en que se han insertado previamente, ya que se insertan por el final de la secuencia y se extraen por la cabecera (figura 6).

Figura 6. Representación de una cola



Ejemplos de colas

Las colas aparecen a menudo en nuestra vida diaria... Sin ir más lejos, podemos afirmar que pasamos una parte de nuestra vida haciendo colas: para comprar la entrada en un cine, para pagar en la caja de un supermercado, para visitarnos por el médico, etc. La idea siempre es la misma: se atiende la primera persona de la cola, que es la que hace más rato que espera, y una vez atentida sale de la cola y la persona siguiente pasa a ser la primera de la cola (figura 7).

Figura 7. Cola de mochilas para entrar en un albergue

Si en el mundo real es habitual ver colas, en el mundo informático todavía lo es más. Cuando el sistema operativo ha de **gestionar el acceso a un recurso compartido** (procesos que quieren ejecutarse en la CPU, trabajos que se envían a una impresora, descarga de ficheros, etc.), una de las estrategias más utilizadas es organizar las peticiones por medio de colas. Por ejemplo, la figura 8 nos muestra una captura de una cola de impresión en un instante dado. En este caso, la tarea 321 se está imprimiendo porque es la primera en la cola, mientras que la tarea 326 será la última en imprimirse porque ha sido la última en llegar.

Figura 8. Captura de una cola de impresión

Estado de la impresión de documentos

Tarea	Documento	Impresora	Tam	Hora de envío	Estado
326	resum.txt	HP-LaserJet-2420	24k	2 min	Pend
323	IMG_0054.JPG	HP-LaserJet-2420	4456k	5 min	Pend
322	codi.c	HP-LaserJet-2420	23k	6 min	Pend
321	seq.pdf	HP-LaserJet-2420	1637k	8 min	Proces

2.2. Operaciones

Dada la representación de una cola, en la tabla 2 definimos las operaciones para trabajar con ella.

Tabla 2

Nombre	Descripción
crear	Crea una cola vacía
encolar	Inserta un elemento en la cola
desencolar	Extrae el elemento situado al principio de la cola
cabeza	Devuelve el elemento situado al principio de la cola
vacia	Devuelve *cierto* si la cola está vacía y *falso* en caso contrario

Como en el TAD anterior, las operaciones del TAD *cola* se clasifican según su comportamiento en generadoras, modificadoras y consultoras:

- **Operaciones constructoras:**
  - **Operaciones generadoras:** `crear` y `encolar`.
  - **Operaciones modificadoras:** `desencolar`.
- **Operaciones consultoras:** `cabeza` y `vacia`.

La **signatura del TAD *cola*** especifica de una manera formal el comportamiento de las operaciones para todos los casos posibles:

tipo cola (elem)
  operaciones
    crear: → cola
    encolar: cola elem → cola
    desencolar: cola → cola
    cabeza: cola → elem
    vacia: cola → booleano

errores
desencolar(crear)
cabeza(crear)
ecuaciones $\forall c \in cola$; $\forall e \in elem$
desencolar(encolar(crear,e)) = crear
[¬vacia(c)] ⇒ desencolar(encolar(c,e)) = encolar(desencolar(c),e)
cabeza(encolar(crear,e)) = e
[¬vacia(c)] ⇒ cabeza(encolar(c,e)) = cabeza(c)
vacia(crear) = cierto
vacia(encolar(c,e)) = falso
ftipo

El comportamiento de las operaciones `desencolar` y `cabeza` se resume en tres casos:

1) sobre una cola vacía (`crear`),

2) sobre una cola con un único elemento (`encolar(crear,e)`) y

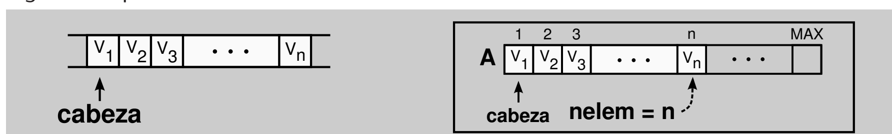
3) sobre una cola con más de un elemento (`encolar(c,e)`, donde `c` no es una cola vacía).

2.3. Implementación

Nuevamente, la opción más sencilla para implementar una cola es usar un **vector**. Por lo tanto, como volvemos a tener una implementación acotada, hay que definir el número máximo de elementos que puede almacenar (`MAX`) y una operación que nos indique cuándo está **llena**.

Al vector de elementos le añadimos un atributo entero (`nelem`) para saber el número de elementos que hay en la cola en todo momento. Tal como podemos ver en la figura 9, la primera posición del vector será siempre el inicio de la cola, mientras que el atributo `nelem` nos indicará cuál es el último elemento e, implícitamente, el inicio del espacio libre.

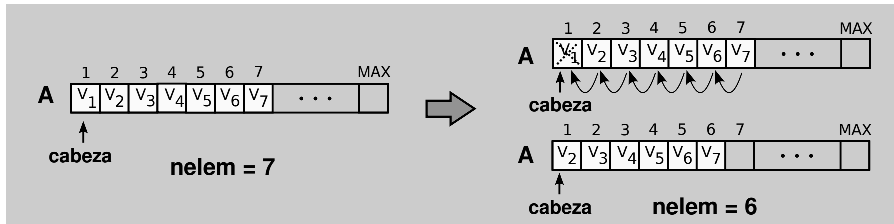
Figura 9. Implementación de una cola con un vector



Esta representación presenta un **grave problema de ineficiencia** en la operación **desencolar**. Cada vez que se elimina el primer elemento de la cola,

mantener la condición por la cual *"el inicio de la cola es siempre la primera posición del vector"* nos obliga a desplazar todos los elementos una posición (como se puede ver en la figura 10). Este desplazamiento comporta que la operación tenga un coste lineal Θ(*nelem*).

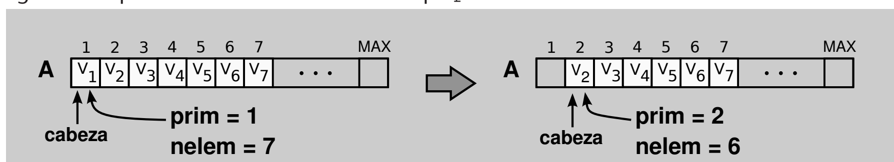
Figura 10. Operación **desencolar**



Para mejorar la implementación anterior añadimos un nuevo atributo entero (`prim`), que apuntará al primer elemento de la cola de modo que los dos extremos de la misma se desplazarán sobre el vector, mientras que los elementos almacenados no se moverán (figura 11).

De esta manera, la nueva representación soluciona el problema de la ineficiencia en la operación `desencolar`, porque siempre que se suprima un elemento de la cola bastará con incrementar el atributo `prim` para apuntar al elemento siguiente.

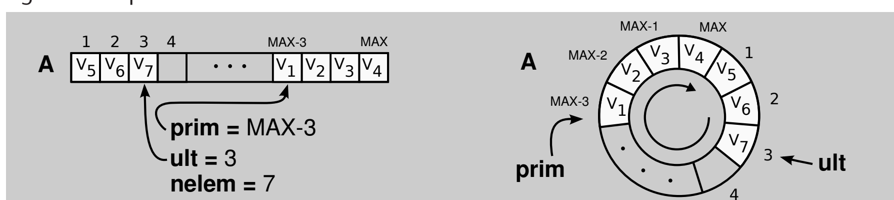
Figura 11. Operación **desencolar** con el campo `prim`



Es más, esta solución consigue que todas las operaciones tengan un coste óptimo Θ(1). Sin embargo, presenta otro problema grave: **desaprovecha mucho espacio**, ya que no se reutilizan las posiciones liberadas en la parte inicial del vector, hasta el punto de que si encolamos y desencolamos muchas veces (fruto de un uso normal de la cola) nos podemos quedar sin espacio, incluso aunque la cola tenga solo un elemento.

Para reaprovechar estas posiciones iniciales sin mover los elementos, consideraremos que el vector es una **estructura circular**, es decir, que después de la última posición del vector vendrá la primera posición. Estamos ante una estructura sin principio ni fin, tal como podemos ver en la figura 12.

Figura 12. Representación de una cola en un vector circular



Para **gestionar el vector circularmente** recurrimos a la operación **módulo**, que calcula el residuo de la división de dos números enteros. Dada la posición *p* de un vector de tamaño *m*, el cálculo para averiguar la siguiente posición del vector es:

$$(p \bmod m) + 1$$

Así pues, la representación escogida será la del **vector circular con el puntero** `prim`. El **coste temporal** de las operaciones será óptimo, $\Theta(1)$, mientras que el **coste espacial** será tan pobre como en las pilas $\Theta(MAX)$ (ya que el vector se crea inicialmente con un tamaño *MAX*). No obstante, la implementación circular consigue llenar todo el vector gracias a la reutilización de las posiciones liberadas para almacenar nuevos elementos.

```
tipo
  cola = tupla
           A : tabla [MAX] de elem;
           prim,ult,nelem : entero;
  ftupla
ftipo
```

Dada la representación del tipo, mostramos la implementación de las operaciones del tipo cola:

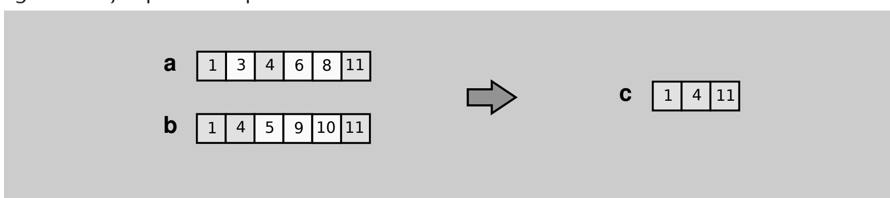
```
funcion crear() : cola
  var c : cola fvar
  c.prim := 1;
  c.ult := 0;
  c.nelem := 0;
  devuelve c;
ffuncion

funcion encolar(c : cola; e : elem) : cola
  si c.nelem = MAX entonces
      error {cola llena};
  sino
      c.ult := (c.ult mod MAX) + 1;
      c.A[c.ult] := e;
      c.nelem := c.nelem + 1;
  fsi
  devuelve c;
ffuncion
```

```
funcion desencolar(c : cola) : cola
  si c.nelem = 0 entonces
      error {cola vacia};
  sino
        c.prim := (c.prim mod MAX) + 1;
        c.nelem := c.nelem - 1;
  fsi
  devuelve c;
ffuncion

funcion cabeza(c : cola) : elem
  var e : elem fvar
  si c.nelem = 0 entonces
      error {cola vacia};
  sino
        e := c.A[c.prim];
  fsi
  devuelve e;
ffuncion

funcion vacia(c : cola) : booleano
  devuelve c.nelem = 0;
ffuncion

funcion llena(c : cola) : booleano
  devuelve c.nelem = MAX;
ffuncion
```

2.4. Ejemplo

Se pide definir una nueva operación, denominada ***interseccion****, que amplíe el TAD cola. Esta operación recibirá como entrada dos colas con los elementos ordenados de manera creciente y devolverá una cola ordenada de forma estrictamente creciente solo con los elementos que aparezcan en las dos colas de entrada (figura 13).*

Figura 13. Ejemplo de la operación `interseccion`



El hecho que la cola esté ordenada nos evita tener que recorrer toda la cola cada vez que queramos comprobar si contiene un elemento dado. En otras palabras, si el primer elemento de la cola *a* es más pequeño que el primer

elemento de la cola *b*, ya es seguro que no encontraremos un elemento igual en toda la cola *b*, porque todos los elementos de la cola *b* serán mayores que el primero de la cola *a*.

Por lo tanto, con esta información establecemos que la estrategia de la operación **interseccion** será la siguiente:

1) Crear la cola en la que se encolarán los elementos comunes a ambas colas de entrada.
2) Mientras las dos colas no estén vacías, leer el primer elemento de ambas.
3) Si los dos elementos son iguales, encolar el elemento en la cola resultado y desencolarlo de las dos colas de entrada. En caso contrario, si los dos elementos no son iguales, desencolar solo el elemento más pequeño.
4) Volver al punto 2 mientras no se haya vaciado ninguna de las dos colas.

Finalmente, implementamos la operación siguiendo el esquema anterior:

```
funcion interseccion(c1 : cola; c2 : cola) : cola
  var c3 : cola fvar
  c3 := crear();
  mientras ¬vacia(c1) ∧ ¬vacia(c2) hacer
            si cabeza(c1) = cabeza(c2) entonces
                c3 := encolar(c3,cabeza(c1));
                c1 := desencolar(c1);
                c2 := desencolar(c2);
           sino si cabeza(c1) < cabeza(c2) entonces
                    c1 := desencolar(c1);
               sino
                    c2 := desencolar(c2);
                fsi
            fsi
  fmientras
  devuelve c3;
ffuncion
```

3. Listas

3.1. Representación

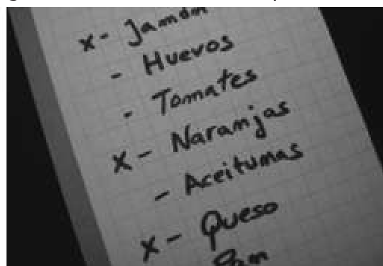
Las listas son el último TAD que estudiaremos para representar las secuencias. Mientras que en los TAD anteriores hemos visto cómo las operaciones trabajaban solo con los extremos de la secuencia, las listas nos ofrecerán la posibilidad de acceder a cualquier punto de esta.

Una **lista** es un TAD caracterizado por el hecho de que permite añadir, borrar o consultar cualquier elemento de la secuencia. Es la estructura lineal más flexible, hasta el punto de considerar los TAD **pila** y **cola** casos particulares del TAD **lista**.

Ejemplos de listas

También en este caso encontramos listas en nuestra vida cotidiana. Por ejemplo, la lista de la compra. Cuando estamos en el supermercado generalmente eliminamos los artículos a medida que los encontramos en el recorrido que seguimos con el carro, que no tiene porqué coincidir con el orden en que los hemos escrito en nuestra lista (figura 14).

Figura 14. Una lista de la compra



Desde el punto de vista informático también encontramos ejemplos, como los editores de textos. Cuando escribimos un código, en el fondo editamos una lista de palabras dentro de una lista de líneas. Hablamos de listas, porque en cualquier momento nos podemos desplazar sobre cualquier palabra del fichero para modificarla o para insertar nuevas palabras.

Hay diferentes modelos de listas, uno de los más habituales es el llamado **lista con punto de interés**. Esta lista contiene un **elemento distinguido** que sirve de referencia para aplicar las operaciones. El elemento distinguido es apuntado por el llamado **punto de interés**, que puede desplazarse.

El punto de interés divide una secuencia en dos fragmentos, que a su vez también son secuencias. Por lo tanto, dada una lista *l* cualquiera, se puede dividir en una secuencia situada a la izquierda del punto de interés (*s*) y otra secuencia que va del punto de interés (elemento distinguido, *e*) hacia la derecha (*et*). Podemos representar esta unión de dos secuencias para formar la lista *l* de la manera siguiente: $l = < s, et >$. La secuencia vacía (es decir, sin ningún elemento) se representará con la letra λ (lambda).

3.2. Operaciones

En la tabla 3 tenéis la operaciones para trabajar con las listas.

Tabla 3

Nombre	Descripción
crear	Crea una lista vacía
insertar	Inserta un elemento en la lista delante del punto de interés
borrar	Extrae el elemento distinguido y desplaza el punto de interés al elemento siguiente
actual	Devuelve el elemento distinguido
vacia	Devuelve *cierto* si la lista está vacía y *falso* en caso contrario
principio	Sitúa el punto de interés sobre el primer elemento de la lista
avanzar	Desplaza el punto de interés al elemento siguiente
fin	Devuelve *cierto* si el punto de interés está en el extremo derecho (final) de la lista y *falso* en caso contrario

Podemos ver que además de las operaciones típicas para trabajar con secuencias, esta vez se han añadido operaciones para cambiar el elemento distinguido y así poder desplazar el punto de interés: `principio` y `avanzar`.

El paso siguiente es determinar la **signatura del tipo lista**:

tipo lista (elem)
  operaciones
    crear: → lista
    insertar: lista elem → lista
    borrar: lista → lista
    actual: lista → elem
    vacia: lista → booleano
    principio: lista → lista
    avanzar: lista → lista
    fin: lista → booleano
  errores
    borrar(<s,λ>)
    avanzar(<s,λ>)
    actual(<s,λ>)

ecuaciones $\forall < s,t > \in$ *lista*; $\forall e \in$ *elem*
crear = <λ,λ>
insertar(<s, t>, e) = <se, t>
borrar(<s, et>) = <s, t>
principio(<s, t>) = <λ, st>
avanzar(<s, et>) = <se, t>
actual(<s, et>) = e
[t = λ] ⇒ fin(<s, t>) = cierto
[t ≠ λ] ⇒ fin(<s, t>) = falso
vacia(crear) = cierto
vacia(insertar(<s, t>,e)) = falso
ftipo

En este caso, el conjunto de **operaciones constructoras generadoras** está formado por `crear`, `insertar` y `principio`. Por tanto, la mínima secuencia de llamadas que necesitaremos para obtener la lista `<ho,la>` es una combinación de estas operaciones:

```
{insertar(insertar(principio(insertar(insertar(crear, l), a)), h), o)}
```

3.3. Implementación

La implementación de las listas con punto de interés se puede hacer de varias maneras:

- **Implementación secuencial.** Los elementos de la lista se almacenan en un vector respetando la norma que los elementos consecutivos ocupan posiciones consecutivas.

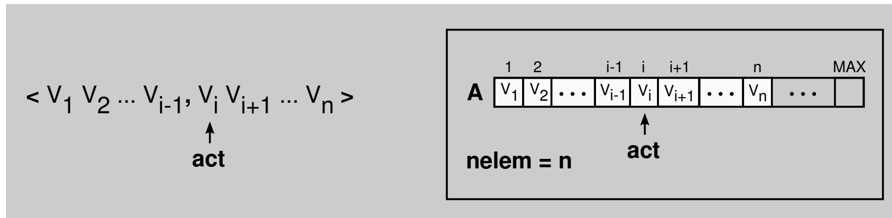
- **Implementación encadenada.** Los elementos se almacenan sin seguir la norma anterior, ya que cada elemento del vector guarda la posición en que se halla el siguiente elemento de la lista.

3.3.1. Implementación secuencial

La representación requiere tres elementos: un vector (`A`), un indicador del número de elementos almacenados (`nelem`) y un indicador del punto de interés o elemento distinguido (`act`).

El indicador `nelem` nos sirve para controlar que no sobrepasemos el número máximo de elementos (`MAX`) que puede almacenar el vector, mientras que el indicador `act` apunta al elemento distinguido de la lista (figura 15).

Figura 15. Implementación secuencial de una lista con un vector




```
tipo
  lista = tupla
              A : tabla [MAX] de elem;
              act,nelem : entero;
  ftupla
ftipo
```

Dada la representación del tipo, implementamos las operaciones del tipo lista:

```
funcion crear() : lista
  var l : lista fvar
  l.act := 1;
  l.nelem := 0;
  devuelve l;
ffuncion

funcion insertar(l : lista; e : elem) : lista
  var i : entero fvar
  si l.nelem = MAX entonces
      error {lista llena};
  sino
      para i := l.nelem hasta l.act paso -1 hacer
          l.A[i + 1] := l.A[i];
      fpara
      l.A[l.act] := e;
      l.nelem := l.nelem + 1;
      l.act := l.act + 1;
  fsi
  devuelve l;
ffuncion
```

```
funcion borrar(l : lista) : lista
  var i : entero fvar
  si l.act > l.nelem entonces
      error {fin de lista o lista vacia};
  sino
       para i := l.act hasta l.nelem - 1 hacer
            l.A[i] := l.A[i + 1];
       fpara
       l.nelem := l.nelem - 1;
  fsi
  devuelve l;
ffuncion

funcion principio(l : lista) : lista
  l.act := 1;
  devuelve l;
ffuncion

funcion avanzar(l : lista) : lista
  si l.act > l.nelem entonces
      error {fin de lista o lista vacia};
  sino
       l.act := l.act + 1;
  fsi
  devuelve l;
ffuncion

funcion actual(l : lista) : elem
  var e : elem fvar
  si l.act > l.nelem entonces
      error {fin de lista o lista vacia};
  sino
       e := l.A[l.act];
  fsi
  devuelve e;
ffuncion

funcion fin(l : lista) : booleano
  devuelve l.act > l.nelem;
ffuncion

funcion vacia(l : lista) : booleano
  devuelve l.nelem = 0;
ffuncion
```

funcion llena(*l* : **lista**) : **booleano**
  devuelve *l.nelem* = *MAX*;
ffuncion

Pero esta implementación del TAD *lista* presenta **varios problemas** cuando revisamos las operaciones `insertar` y `borrar`:

- Las operaciones presentan desplazamientos de elementos dentro del vector, concretamente en las posiciones a la derecha del punto de interés (en los bucles). Estos desplazamientos hacen que el **coste temporal** de ambas operaciones sea lineal y, por tanto, que su eficiencia sea muy mala teniendo en cuenta la frecuencia de uso (figura 16).

- Si los elementos de la lista son **grandes**, su desplazamiento dentro del vector (para no desaprovechar espacio) es muy costoso.

- La **integración de la lista en una estructura de datos** es compleja. Los elementos de la lista son apuntados desde diferentes componentes de la estructura, de modo que cada desplazamiento de un elemento del vector implica automáticamente la actualización de todos los apuntadores externos al vector que apuntan al elemento desplazado (figura 17). Como os podéis imaginar, esta tarea no es trivial.

Figura 16. Operación `borrar` en la implementación secuencial de una lista

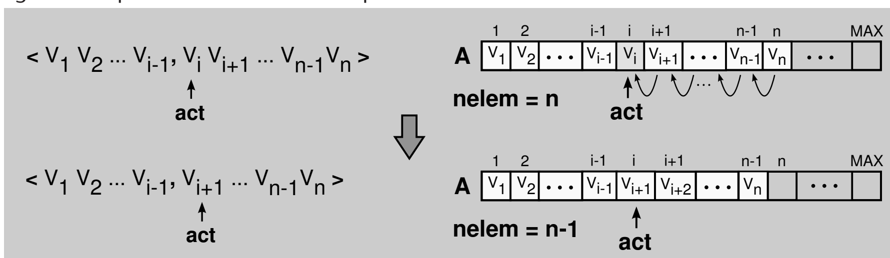
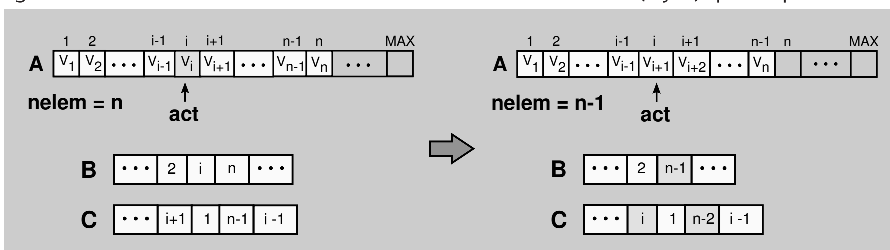


Figura 17. `Borrar` un elemento de la lista A afecta a las estructuras (B y C) que le apuntan

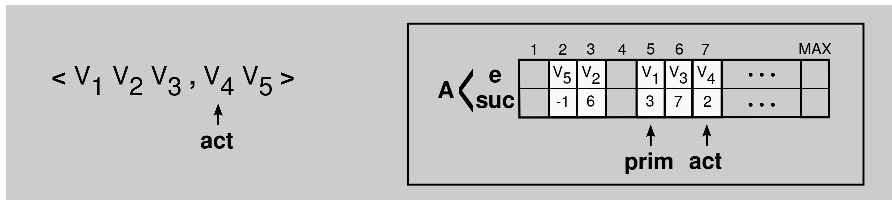


3.3.2. Implementación encadenada

Para romper el concepto de **secuencial**, en que el sucesor de un elemento es el que ocupa la siguiente posición del vector, introducimos a continuación el concepto de **encadenamiento**, en que cada elemento guarda la posición en que se halla su sucesor.

La representación, como se puede observar en la figura 18, almacena para cada posición del vector: el elemento (**e**) y un indicador de la posición en que se halla el siguiente elemento (**suc**). Este indicador se denominará, a partir de este momento, **encadenamiento**.

Figura 18. Implementación encadenada de una lista con un vector



La implementación encadenada resuelve los problemas que sufría la representación secuencial en las operaciones `insertar` y `borrar`, porque evita el desplazamiento de elementos en el vector, al no tener que mantenerlos en una ordenación secuencial.

Aunque el precio a pagar para ahorrarse estos desplazamientos es el almacenamiento de un campo extra por cada elemento, este coste parece asumible, especialmente en listas muy volátiles (con muchas operaciones de insertar y borrar elementos) o con elementos de grandes dimensiones (que hagan negligible el espacio ocupado por este campo extra).

Por tanto, en este caso, la representación de la lista constará de un indicador al primer elemento (`prim`), otro al elemento distinguido o actual (`act`) y un vector (`A`) con un indicador para cada elemento (**e**) de cuál es el siguiente (`suc`). En el caso del último elemento, el indicador tendrá un **valor nulo**, que por convenio suele ser -1 (ya que no existe una posición negativa en un vector).

```
tipo
  lista = tupla
            A : tabla [MAX] de tupla
                                  e : elem;
                                  suc : entero;
                                ftupla;
            act,prim : entero;
  ftupla
ftipo
```

A partir de esta representación del tipo, implementamos las operaciones que nos permitirán recorrer la lista y consultar/borrar/insertar elementos en ella.

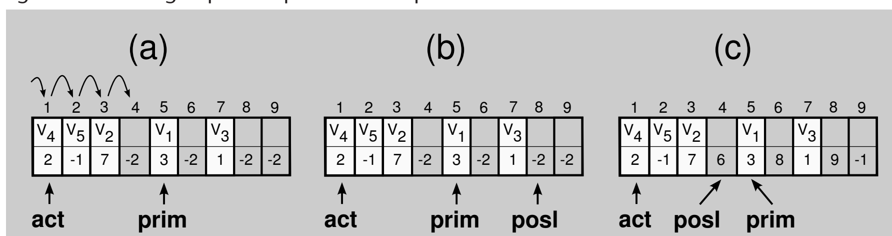
Pero antes hay que pensar cómo se pueden **reaprovechar las posiciones del vector que se han borrado** para almacenar nuevos elementos. A continuación analizamos algunas de las estrategias que se pueden aplicar para reaprovechar estas posiciones:

- Marcar todas las posiciones del vector como ocupadas o libres aprovechando el campo `suc`, siempre que no se disponga de ninguna estructura extra que nos indique qué posiciones están libres. Para diferenciar las posiciones libres escogeremos un valor no utilizado en el campo `suc`, como por ejemplo -2, ya que las posiciones del vector son enteros positivos y el valor nulo es -1. *Problema*: para cada inserción hay que hacer una búsqueda de una posición libre en el vector y eso puede ser muy costoso (figura 19a).

- Añadir un indicador (`posl`) que apunte a la primera posición libre del vector (figura 19b). Cuando se acaben las posiciones libres se tendrá que reorganizar el vector desplazando todos los elementos hacia la izquierda para dejar a la derecha todos los espacios libres liberados en el borrado de elementos. *Problema*: tal como se ha comentado en la implementación secuencial, la reorganización del vector es demasiado costosa.

- Usar una pila para gestionar las posiciones libres: **pila de sitios libres**. Esta pila se implementa sobre el mismo vector de la lista, aprovechando el campo `suc` para encadenar las posiciones libres y evitar utilizar más espacio (figura 19c). El funcionamiento es simple: cuando se inserta un elemento en la lista se extrae el primer sitio libre de la pila (`posl`), mientras que cuando se borra un elemento de la lista la posición liberada se añade a la pila.

Figura 19. Estrategias para reaprovechar las posiciones liberadas del vector



Solo queda resolver otro detalle para afrontar la implementación de las operaciones `insertar` y `borrar`: **cómo se puede actualizar el indicador** `suc` **del elemento anterior** para que apunte al nuevo sucesor.

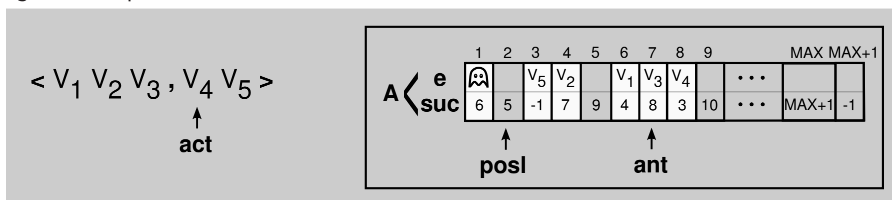
Puesto que la representación encadenada no asegura que los datos estén dispuestos secuencialmente en el vector, el acceso al elemento anterior al actual (o escogido) no es trivial y se debe buscar una solución:

- Si no se añade nada más a la representación, hay que recorrer la lista desde el principio, buscando el elemento que tenga en el campo `suc` la posición del elemento actual. Pero esto implica que cada llamada a las operaciones `insertar` y `borrar` tendrá un coste temporal lineal, $\Theta(n)$.

- Si cambiamos el indicador del elemento actual (`act`) por un indicador al elemento anterior al actual (`ant`), se podrá acceder al elemento anterior y al elemento actual (sucesor del elemento anterior) de manera trivial y con un coste temporal constante, $\Theta(1)$.

  Problema: ¿cuál es el elemento anterior al primero? Para resolver esto tenemos dos opciones: o bien almacenar un valor especial (-2) en el campo `ant` cuando el elemento actual sea el primero (opción que complica la codificación de las operaciones al añadir nuevos casos) o bien guardar siempre en la primera posición del vector un elemento ficticio, llamado **fantasma** o **centinela** (figura 20), que no se borrará nunca y que hará que el elemento actual siempre tenga uno anterior (aunque perdamos una posición del vector para almacenar datos).

Figura 20. Implementación encadenada de una lista con elemento fantasma



- Si usamos una **lista doblemente encadenada** cada posición del vector contendrá el elemento, un encadenamiento al elemento sucesor y otro encadenamiento al predecesor. De esta manera el acceso al elemento anterior para actualizar los indicadores también es trivial. Esta opción la veremos más adelante en el subapartado 4.6.2.

Finalmente, la representación escogida es la de una **lista encadenada con elemento fantasma y usando un indicador al elemento anterior al actual** (`ant`), porque nos permite codificar todas las operaciones de la signatura (excepto `crear`) con un coste constante, $\Theta(1)$.

Fijaros en que la lista podrá almacenar hasta `MAX` elementos, ya que se ha sumado una posición `MAX` + 1 para almacenar el fantasma:

```
tipo
  lista = tupla
            A : tabla [MAX + 1] de tupla
                                     e : elem;
                                     suc : entero;
                                   ftupla;
            ant,posl : entero;
  ftupla
ftipo
```

Como se ha mencionado anteriormente, esta representación nos permite implementar todas las operaciones con un **coste temporal** constante, excepto por lo que respecta a la operación `crear`, que tendrá un coste lineal al tener que crear la pila de espacios libres y llenarla con todas las posiciones del vector:

```
funcion crear() : lista
   var l : lista fvar
   l.ant := 1;
   l.A[l.ant].suc := -1;     {inserta elemento fantasma}
   para i := 2 hasta MAX hacer
          l.A[i].suc := i + 1;
   fpara
   l.A[MAX + 1].suc := -1;
   l.posl := 2;
   devuelve l;
ffuncion

funcion insertar(l : lista; e : elem) : lista
   var tmp : entero fvar
   si l.posl = -1 entonces
        error {lista llena};
   sino
          tmp := l.posl;
          l.posl := l.A[l.posl].suc;
          l.A[tmp].e := e;
          l.A[tmp].suc := l.A[l.ant].suc;
          l.A[l.ant].suc := tmp;
          l.ant := tmp;
   fsi
   devuelve l;
ffuncion

funcion borrar(l : lista) : lista
   var tmp : entero fvar
   si l.A[l.ant].suc = -1 entonces
        error {fin de lista o lista vacia};
   sino
          tmp := l.A[l.ant].suc;
          l.A[l.ant].suc := l.A[tmp].suc;
          l.A[tmp].suc := l.posl;
          l.posl := tmp;
   fsi
   devuelve l;
ffuncion
```

```
funcion principio(l : lista) : lista
   l.ant := 1;
   devuelve l;
ffuncion

funcion avanzar(l : lista) : lista
   si l.A[l.ant].suc = -1 entonces
         error {fin de lista o lista vacia};
   sino l.ant := l.A[l.ant].suc;
   fsi
   devuelve l;
ffuncion

funcion actual(l : lista) : elem
   var e : elem fvar
   si l.A[l.ant].suc = -1 entonces
         error {fin de lista o lista vacia};
   sino
          e := l.A[l.A[l.ant].suc].e;
   fsi
   devuelve e;
ffuncion

funcion fin(l : lista) : booleano
   devuelve l.A[l.ant].suc = -1;
ffuncion

funcion vacia(l : lista) : booleano
   devuelve l.A[1].suc = -1;
ffuncion

funcion llena(l : lista) : booleano
   devuelve l.posl = -1;
ffuncion
```

3.4. Ejemplo

Cuando hablamos de los esquemas de programación sobre secuencias, todos conocemos los esquemas de recorrido y búsqueda. A continuación, implementamos estos dos esquemas sobre una lista "l".

Para empezar, recordemos que:

- el esquema de recorrido consiste en aplicar un `tratamiento` a todos los elementos de la lista,

- el esquema de búsqueda consiste en situar el punto de interés sobre el primer elemento de la lista que cumple una `propiedad`.

Así, ya podemos implementar el esquema de recorrido:

```
l := principio(l);
mientras ¬fin(l) hacer
        tratamiento(actual(l));
        l := avanzar(l);
fmientras
```

Mientras que la implementación del esquema de búsqueda será:

```
l := principio(l);
ok := FALSO;
mientras ¬fin(l) ∧ ¬ok hacer
        si propiedad(actual(l)) entonces
            ok := CIERTO;
        sino
            l := avanzar(l);
        fsi
fmientras
```

4. Punteros

4.1. Problemas de los vectores

Todas las implementaciones hechas hasta ahora en el módulo se han basado en los vectores porque son fáciles de usar, pero estos también presentan algunos problemas que hay que tener en cuenta:

- Los vectores se crean con un tamaño que quedará fijo durante toda la ejecución; por lo tanto, **no se adaptan a la evolución dinámica del programa**. El vector ocupará el mismo espacio independientemente del número de elementos que almacene, de modo que se desperdician las posiciones que no guardan elementos. Este hecho puede ser importante cuando un programa trabaja con diversas estructuras de datos y llega un momento en que el programa no puede continuar porque una de las estructuras se ha llenado mientras que el resto aún tenía espacio libre.

- Se tiene que **determinar inicialmente la capacidad del vector** sin disponer de información suficiente. Generalmente, cuando se declaran las variables, no se tiene una idea clara de cuántos elementos almacenará cada vector.

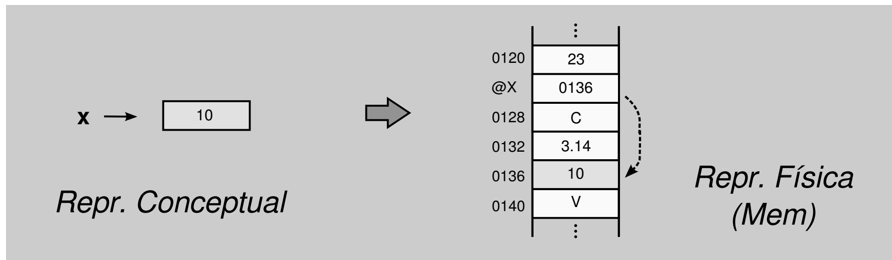
- Tal como hemos visto en las implementaciones hechas, **nuestro código ha tenido que gestionar el espacio libre** en el vector.

4.2. La alternativa: punteros

La alternativa al uso de los vectores es la **gestión de memoria dinámica**. Este mecanismo nos evita tener que estimar inicialmente el espacio que necesitaremos para almacenar los elementos, ya que pediremos el espacio a medida que lo necesitemos.

Para trabajar con la memoria dinámica usaremos un tipo denominado **puntero**, de manera que una variable de tipo `puntero a T` (en que *T* es un tipo de datos concreto) será un apuntador a un objeto de tipo *T*. El puntero ofrecerá un camino de acceso al objeto apuntado. Por ejemplo, si `x` es una variable de tipo `puntero a Entero`, entonces `x` apuntará a un entero (figura 21).

Figura 21. Ejemplo de un puntero a entero



Un **puntero** es una variable que guarda la dirección de memoria donde empieza a almacenarse el objeto al que apunta.

En este caso, el sistema operativo es el encargado de gestionar el espacio de memoria. Para trabajar con los punteros disponemos de las **primitivas** y los **operadores** siguientes:

- **puntero a**: sirve para declarar un tipo de apuntadores a objetos de otro tipo.

Ejemplo

Declaración de un tipo `tpAbc` de apuntadores a objetos de tipo `tAbc`.

```
tipo
   tAbc =  tupla
              e : entero;
              ...
   ftupla
   tpAbc = puntero a tAbc;
ftipo
```

- **obtenerEspacio**: función para solicitar el espacio necesario para almacenar un objeto del tipo apuntado. La función devuelve un puntero que apunta al espacio concedido para almacenar el objeto.

Ejemplo

El puntero `p` apunta a un objeto del tipo `tAbc`.

```
var p : tpAbc fvar
   p := obtenerEspacio();
```

- **operador** '^': operador que permite acceder al objeto apuntado por un puntero. Escribiremos el símbolo detrás del nombre del puntero y devolverá el objeto apuntado por el puntero.

Ejemplo

Se asigna 8 al campo `e` de la tupla apuntada por el puntero `p`.

```
p^.e := 8;
```

- **liberarEspacio**: acción que destruye el objeto apuntado por el puntero que recibe como parámetro. El objeto deja de ser accesible y se libera el espacio ocupado para poder ser reutilizado.

Ejemplo

Se libera el objeto de tipo `tAbc` apuntado por el puntero `p`.

liberarEspacio(*p*);

- **NULO**: es un valor especial que indica que un puntero no apunta a ninguna parte. La función `obtenerEspacio` devuelve este valor cuando no queda espacio disponible y, en consecuencia, no puede proporcionar el espacio solicitado. La acción `liberarEspacio` asigna el valor `NULO` al puntero una vez ha destruido el objeto apuntado. Por otro lado, obtendremos un error siempre que llamemos a la acción `liberarEspacio` o apliquemos el operador '`^`' sobre un puntero que contenga el valor `NULO`.

Ejemplo

Después de solicitar el espacio para almacenar un objeto, siempre debe comprobarse si el sistema nos lo ha concedido, verificando que el puntero `p` apunta a algún sitio.

```
var p : tpAbc fvar
  p := obtenerEspacio();
  si p = NULO entonces error {no hay espacio libre};
  sino
       p^.e := p^.e + 1;
  fsi
```

4.3. Implementación

Volvemos a implementar las estructuras básicas explicadas anteriormente, sin embargo, en lugar de utilizar vectores, esta vez utilizaremos punteros.

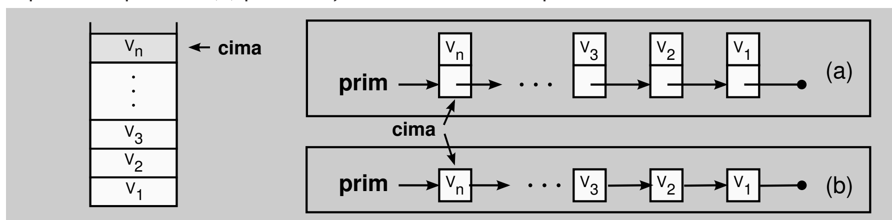
Fijaros que ya no se utilizará la constante que indicaba la capacidad máxima de la estructura, pero sí que se continuará controlando que haya el espacio libre suficiente en la inserción de nuevos elementos.

> El encadenamiento entre elementos (`suc`) se hará con punteros.

4.3.1. Pila

A continuación establecemos la representación y os mostramos la implementación de las operaciones del tipo pila. De la representación (figura 22), comentaremos que el puntero `prim` siempre apunta a la **cima** de la pila, excepto cuando está vacía.

Figura 22. Implementación de una pila con punteros. A partir de ahora utilizaremos el esquema simplificado (b) para dibujar las estructuras con punteros.



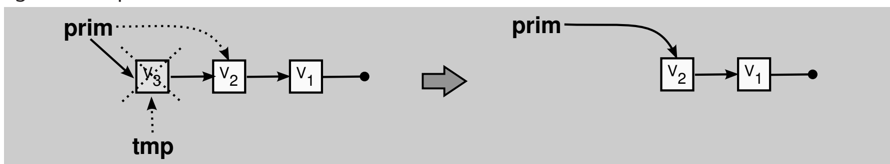

```
tipo
   nodo = tupla
            e : elem;
            suc : puntero a nodo;
   ftupla
   pila = tupla
            prim : puntero a nodo;
   ftupla
ftipo
```

Los **nodos** son los componentes usados en la construcción de las estructuras de datos cuando se trabaja con punteros. Cada **nodo** contiene el elemento a almacenar y uno o más encadenamientos a otros nodos para poder desplazarse entre ellos.

```
funcion crear() : pila
   var p : pila fvar
   p.prim := NULO;
   devuelve p;
ffuncion

funcion apilar(p : pila; e : elem) : pila
   var tmp : puntero a nodo fvar
   tmp := obtenerEspacio();
   si tmp = NULO entonces error {no hay espacio libre};
   sino
         tmp^.e := e;
         tmp^.suc := p.prim;
         p.prim := tmp;
   fsi
   devuelve p;
ffuncion
```

funcion desapilar(*p* : **pila**) : **pila**
  var *tmp* : **puntero a** *nodo* **fvar**
  si *p.prim* = NULO **entonces** *error* {*pila vacia*};
  sino
    tmp := *p.prim*;
    p.prim := *p.prim^.suc*;
    liberarEspacio(*tmp*);
  fsi
  devuelve *p*;
ffuncion

Figura 23. Operación `desapilar`



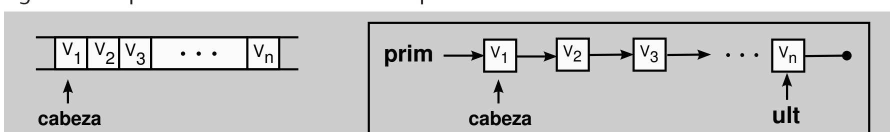
funcion cima(*p* : **pila**) : **elem**
  var *e* : **elem** **fvar**
  si *p.prim* = NULO **entonces**
    error {*pila vacia*};
  sino
    e := *p.prim^.e*;
  fsi
  devuelve *e*;
ffuncion

funcion vacia(*p* : **pila**) : **booleano**
  devuelve *p.prim* = NULO;
ffuncion

4.3.2. Cola

El **principio** de la cola (o extremo por donde se extraen los elementos) estará siempre apuntado por el puntero `prim`, mientras que el **final** de la cola (o extremo por donde se añaden los elementos) estará apuntado por el puntero `ult` (figura 24). Cuando la cola esté vacía, los punteros `prim` y `ult` no apuntarán a ningún elemento.

Figura 24. Implementación de una cola con punteros



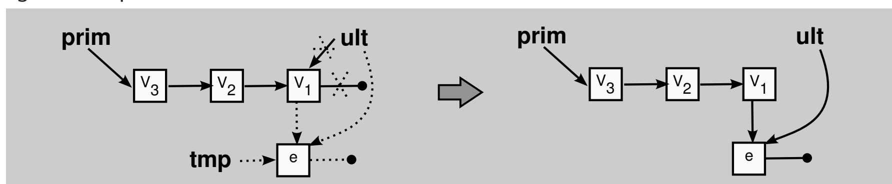
```
tipo
   nodo = tupla
              e : elem;
              suc : puntero a nodo;
   ftupla
   cola = tupla
             prim,ult : puntero a nodo;
   ftupla
ftipo
```

Dada la representación del tipo cola, mostramos la implementación de sus operaciones:

```
funcion crear() : cola
   var c : cola fvar
   c.prim := NULO;
   c.ult := NULO;
   devuelve c;
ffuncion
```

```
funcion encolar(c : cola; e : elem) : cola
   var tmp : puntero a nodo fvar
   tmp := obtenerEspacio();
   si tmp = NULO entonces error {no hay espacio libre};
   sino
         tmp^.e := e;
         tmp^.suc := NULO;
         c.ult^.suc := tmp;
         c.ult := tmp;
   fsi
   devuelve c;
ffuncion
```

Figura 25. Operación `encolar`

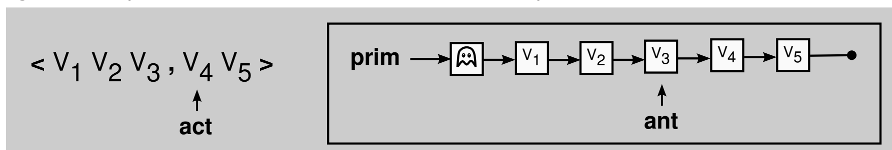


```
funcion desencolar(c : cola) : cola
   var tmp : puntero a nodo fvar
   si c.prim = NULO entonces error {cola vacia};
   sino
         tmp := c.prim;
         c.prim := c.prim^.suc;
         liberarEspacio(tmp);
   fsi
   devuelve c;
ffuncion

funcion cabeza(c : cola) : elem
   var e : elem fvar
   si c.prim = NULO entonces error {cola vacia};
   sino
         e := c.prim^.e;
   fsi
   devuelve e;
ffuncion

funcion vacia(c : cola) : booleano
   devuelve c.prim = NULO;
ffuncion
```

4.3.3. Lista

Volvemos a escoger la **lista encadenada con punto de interés y elemento fantasma**. El puntero `ant` apuntará al elemento anterior al punto de interés y no al elemento distinguido. De este modo, se podrá acceder fácilmente al predecesor del elemento distinguido (figura 26).

Figura 26. Implementación de una lista encadenada con punteros

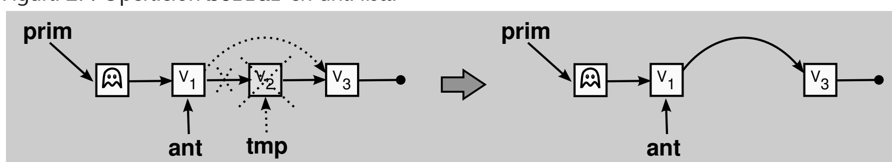



```
tipo
   nodo =  tupla
                e : elem;
                suc : puntero a nodo;
   ftupla
   lista =  tupla
               prim,ant : puntero a nodo;
   ftupla
ftipo
```

Dada la representación del tipo lista, mostramos la implementación de sus operaciones:

```
funcion crear() : lista
   var l : lista fvar
   l.prim := obtenerEspacio();     {para el elemento fantasma}
   si l.prim = NULO entonces
         error {no hay espacio libre};
   sino
         l.ant := l.prim;
         l.ant^.suc := NULO;
   fsi
   devuelve l;
ffuncion

funcion insertar(l : lista;  e : elem) : lista
   var tmp : puntero a nodo fvar
   tmp := obtenerEspacio();
   si tmp = NULO entonces
         error {no hay espacio libre};
   sino
         tmp^.e := e;
         tmp^.suc := l.ant^.suc;
         l.ant^.suc := tmp;
         l.ant := tmp;
   fsi
   devuelve l;
ffuncion

funcion borrar(l : lista) : lista
   var tmp : puntero a nodo fvar
   si l.ant^.suc = NULO entonces
         error {fin de lista o lista vacia};
   sino
         tmp := l.ant^.suc;
         l.ant^.suc := tmp^.suc;
         liberarEspacio(tmp);
   fsi
   devuelve l;
ffuncion
```

Figura 27. Operación `borrar` en una lista



```
funcion principio(l : lista) : lista
  l.ant := l.prim;
  devuelve l;
ffuncion

funcion avanzar(l : lista) : lista
  si l.ant^.suc = NULO entonces
      error {fin de lista o lista vacia};
  sino
      l.ant := l.ant^.suc;
  fsi
  devuelve l;
ffuncion

funcion actual(l : lista) : elem
  var e : elem fvar
  si l.ant^.suc = NULO entonces
      error {fin de lista o lista vacia};
  sino
      e := l.ant^.suc^.e;
  fsi
  devuelve e;
ffuncion

funcion fin(l : lista) : booleano
  devuelve l.ant^.suc = NULO;
ffuncion

funcion vacia(l : lista) : booleano
  devuelve l.prim^.suc = NULO;
ffuncion
```

4.4. Peligros de los punteros

Antes de continuar, hay que tener en cuenta que los punteros no son la panacea para implementar estructuras de datos, ya que también presentan problemas. En los siguientes subapartados veremos unos cuantos.

4.4.1. La memoria dinámica no es infinita

La memoria dinámica no puede garantizar la capacidad de una estructura (cuántos objetos cabrán), porque es un recurso compartido (como un saco) al que todos los programas solicitan el espacio que necesitan. Eso imposibilita que el gestor de memoria pueda predecir todas las peticiones que harán los programas y, por lo tanto, desconozca el espacio de que dispondrá en cada instante.

Por este motivo, siempre que se solicita espacio a la memoria dinámica, tenemos que comprobar que se nos ha concedido, porque en cualquier momento el espacio libre se puede agotar.

De todas maneras, si necesitáis asegurar una capacidad máxima en una estructura, probablemente tendréis que implementar la estructura con vectores, ya que el tamaño declarado inicialmente está garantizado para toda la ejecución.

4.4.2. El funcionamiento del TAD depende de la representación escogida

Este punto es inaceptable, ya que significa que el comportamiento del algoritmo dependerá de la implementación que se haya hecho del tipo. Y eso quiere decir que **se pierde la propiedad principal de los TAD, que es la abstracción**. En otras palabras, se pierde la transparencia al utilizar los tipos.

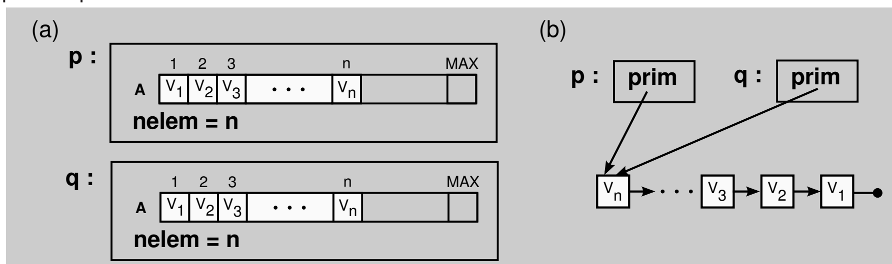
A continuación, os detallamos con ejemplos qué construcciones algorítmicas se comportan de manera diferente en función de si el tipo se ha implementado con vectores o con punteros, y también os detallamos como solucionarlo:

1) Asignación. Dadas dos variables `p` y `q` de tipo pila, el resultado de la asignación `p := q` depende de como se haya implementado el tipo (figura 28):

a) Si se han utilizado vectores, `p` y `q` son dos tuplas que contienen un vector y un entero. La asignación copiará el campo entero y el contenido del vector, dando como resultado **dos pilas iguales e independientes**.

b) Si se han utilizado punteros, `p` y `q` son dos tuplas que contienen un puntero (`prim`). La asignación simplemente copiará el campo de las tuplas; es decir, el puntero `prim` de las dos pilas apuntará a la misma dirección de memoria. Por tanto, el resultado será una **única pila** compartida por `p` y `q`.

Figura 28. Resultado de la operación `p := q` en función de la implementación escogida para las pilas

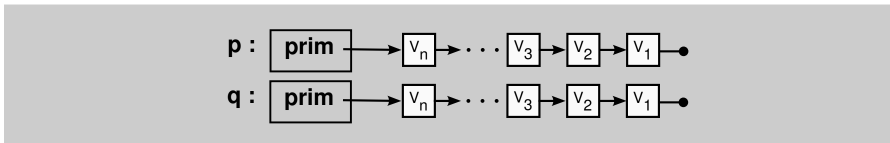


La solución consistirá en definir una nueva operación denominada **duplicar**, que hará la copia exacta de un objeto sobre otro objeto del mismo TAD.

Como ejemplo, os mostramos la implementación de la operación `duplicar` en el TAD *pila*, tanto para la representación con vectores como para la representación con punteros:

```
funcion duplicar(p : pila) : pila
   var q : pila; i : entero fvar
   q.nelem := p.nelem;
   para i := 1 hasta p.nelem hacer
         q.A[i] := duplicar(p.A[i]);
   fpara
   devuelve q;
ffuncion

funcion duplicar(p : pila) : pila
   var q : pila; n1,n2 : puntero a nodo fvar
   si p.prim = NULO entonces
         q.prim = NULO;
   sino
         n1 := p.prim;
         n2 := obtenerEspacio();
         si n2 = NULO entonces
               error {no hay espacio libre};
         sino
                q.prim := n2;
                n2^.e := duplicar(n1^.e);
         fsi
         mientras n1^.suc ≠ NULO hacer
                     n2^.suc := obtenerEspacio();
                     si n2^.suc = NULO entonces
                           error {no hay espacio libre};
                     sino
                           n1 := n1^.suc;
                           n2 := n2^.suc;
                           n2^.e := duplicar(n1^.e);
                     fsi
         fmientras
         n2^.suc := NULO;
   fsi
   devuelve q;
ffuncion
```

Figura 29. Resultado de la operación `duplicar(p,q)`, implementando las pilas con punteros



2) Comparación. Dadas dos variables `p` y `q` de tipo pila, el resultado de la comparación `p = q` también depende de cómo se haya implementado el tipo:

a) Si se han utilizado vectores, `p` y `q` son dos tuplas que contienen un vector y un entero. El operador de comparación comparará el valor de los campos y del contenido del vector y dará como resultado la comparación de igualdad correcta.

b) Si se han utilizado punteros, `p` y `q` son dos tuplas que contienen un puntero (`prim`). El operador de comparación comparará el valor de los campos, es decir, comparará si el puntero `prim` de ambas pilas apunta a la misma dirección de memoria. Por lo tanto, **el resultado no será una comparación del contenido de las pilas `p` y `q`**, sino una comparación de sus posiciones en la memoria.

Para resolver este problema aplicaremos la misma solución que en la operación de asignación. En lugar de usar el operador de igualdad, cada TAD definirá una operación denominada `igual`, que comprobará si dos objetos son iguales según la definición de igualdad en el tipo.

Por ejemplo, en el caso de las pilas diremos que dos pilas son iguales si contienen los mismos elementos en el mismo orden, mientras que el caso de las listas también pediremos que el punto de interés esté situado en el mismo lugar.

Como ejemplo de la operación `igual`, os adjuntamos su implementación en el TAD *pila*, tanto para la representación con vectores (primera función) como para la de punteros (segunda función):

```
funcion igual(p1,p2 : pila) : booleano
   var ig : booleano; i : entero fvar
   ig := (p1.nelem = p2.nelem);
   i := 1;
   mientras (i ≤ p.nelem) ∧ ig hacer
            ig := igual(p1.A[i],p2.A[i]);
            i := i + 1;
   fmientras
   devuelve ig;
ffuncion
```

```
funcion igual(p1,p2 : pila) : booleano
   var ig : booleano; n1,n2 : puntero a nodo fvar
   n1 := p1.prim;
   n2 := p2.prim;
   ig := CIERTO;
   mientras (n1 ≠ NULO) ∧ (n2 ≠ NULO) ∧ ig hacer
            ig := igual(n1^.e,n2^.e);
            si ig entonces
                n1 := n1^.suc;
                n2 := n2^.suc;
            fsi
   fmientras
   devuelve (n1 = NULO) ∧ (n2 = NULO);
ffuncion
```

3) Parámetros de entrada. Dada una función que declara un puntero como parámetro de entrada, tenemos que **solo está protegido el valor del puntero, pero en cambio no lo está el valor del objeto apuntado por el puntero**. Esto implica que si pasamos una pila `p` como parámetro de entrada de una función, la protección del contenido dependerá de cómo se haya implementado el tipo:

a) Si se han utilizado vectores, `p` contiene un vector y un entero. En este caso los campos se duplican y ninguno de los campos de la tupla `p` estará modificado al volver de la llamada a la función.

b) Si se han utilizado punteros, `p` contiene un puntero al primer nodo de la pila (`prim`). El puntero `prim` se duplicará y al volver de la llamada a la función no se habrá modificado, pero sí que pueden haberse modificado los objetos almacenados en la pila, ya que no estaban duplicados al no estar protegidos por la definición de parámetro de entrada.

Para solucionar este problema, simplemente debemos evitar modificar los parámetros de entrada. Si fuese necesario modificar algún parámetro de entrada en la función, usaríamos la operación `duplicar` del tipo para crear una copia del objeto y trabajar con ella.

Pero esta solución puede provocar un **problema de ineficiencia**, ya que la duplicación de objetos puede ser muy costosa (por ejemplo, en el caso de una pila llena de elementos). Para resolverlo, podemos convertir las funciones en acciones, de manera que el resultado de la operación quede en uno de los parámetros de entsal y nos ahorremos así el coste de la copia.

Como ejemplo de la problemática, os damos la implementación de una función para fusionar dos pilas sin mantener el orden de los elementos. Como la función es externa al tipo, no podrá acceder a la representación y solo podrá utilizar las operaciones proporcionadas por el TAD *pila*.

La primera versión es incorrecta si las pilas se han implementado con punteros, porque, al retornar de la llamada, la pila `p2` se ha vaciado y la pila `p1` contiene nuevos elementos:

```
funcion fusionar(p1,p2 : pila) : pila
   mientras ¬vacia(p2) hacer
         p1 := apilar(p1,cima(p2));
         p2 := desapilar(p2);
   fmientras
   devuelve p1;
ffuncion
```

La segunda implementación es correcta, ya que es independiente de la implementación porque crea unas copias (`pa` y `pb`) para proteger las pilas `p1` y `p2`:

```
funcion fusionar(p1,p2 : pila) : pila
   var pa,pb : pila fvar
   pa := duplicar(p1);
   pb := duplicar(p2);
   mientras ¬vacia(pb) hacer
         pa := apilar(pa,cima(pb));
         pb := desapilar(pb);
   fmientras
   devuelve pa;
ffuncion
```

Para finalizar, la tercera implementación es la más eficiente de las tres, puesto que evita la duplicación de la pila `p1` mediante su conversión en un parámetro de *entsal*:

```
accion fusionar(entsal p1 : pila; ent p2 : pila)
   var pb : pila fvar
   pb := duplicar(p2);
   mientras ¬vacia(pb) hacer
         p1 := apilar(p1,cima(pb));
         pb := desapilar(pb);
   fmientras
faccion
```

4.4.3. Efectos laterales

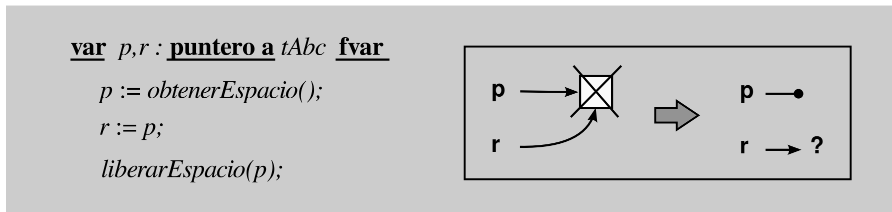
Dado un objeto apuntado por varios punteros, cualquier modificación del objeto mediante un puntero comporta un **efecto lateral**, y es que el resto de punteros verán el objeto modificado sin haberlo tocado.

Hay efectos laterales deseados o controlados, pero también los hay erróneos o no controlados.

4.4.4. Referencias colgadas

Hablamos de **referencia colgada** cuando un puntero apunta a un objeto que no existe. Si intentamos acceder a él (con el operador ^), el resultado es impredecible. Por ejemplo, en la figura 30, el puntero `r` acaba colgado.

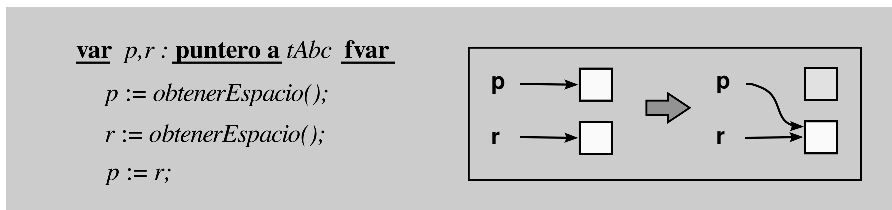
Figura 30. Ejemplo de una referencia colgada



4.4.5. Retales

Un **retal** es un objeto que ha obtenido el espacio a través de la función `obtenerEspacio`, pero que es inaccesible porque no está apuntado por ningún puntero. No hay forma de acceder a él. La figura 31 muestra cómo el objeto apuntado inicialmente por el puntero `p` se convierte en un retal.

Figura 31. Ejemplo de un retal



Los retales pueden provocar un error del sistema operativo o simplemente no afectar a la ejecución y dificultar la depuración del código. Hay sistemas operativos que se encargan de recuperar automáticamente estos retales u objetos inaccesibles, pero trataremos de evitarlos siempre que sea posible, ya que son peligrosos.

Algunos ejemplos de situaciones que pueden crear retales y que debemos vigilar son:

- La **declaración de variables auxiliares** del tipo. Si los objetos se han implementado con punteros, al salir de una función quedarán como retales, ya que se eliminarán los punteros que los hacían accesibles.

```
accion concatenar(entsal l1 : lista; ent l2 : lista)
   var lb : lista fvar
   ...
   lb := duplicar(l2);
   ...
   {vacia(lb) = FALSO}
faccion
```

- Otro ejemplo lo tenemos en lo que llamamos **reinicialización de variables**. En un punto del programa, el programador puede decidir reutilizar una variable (que ya no se usa) para no tener que declarar más variables. Si la inicializa llamando a la operación de creación (**crear**), generará un retal, porque el objeto anterior a la creación será inaccesible sin haber liberado su espacio.

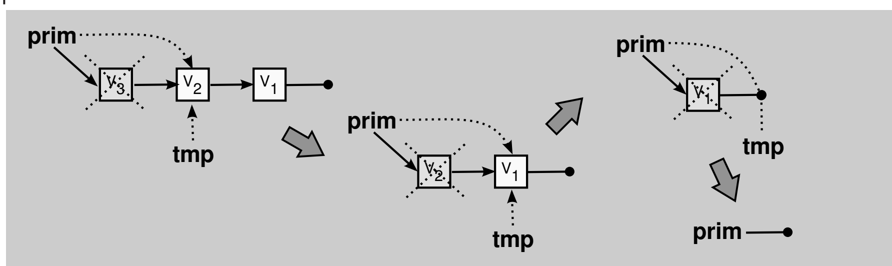
```
var p : pila fvar
p := crear();
...
p := apilar(p,e);
...
{vacia(p) = FALSO}
p := crear();
...
```

Para solucionar este problema, añadimos una operación denominada `destruir` al TAD, que se encargará de liberar el espacio ocupado por un objeto.

Como ejemplo os adjuntamos la implementación de la operación `destruir` en el TAD *pila*, tanto para la representación con vectores como para la de punteros:

```
accion destruir(entsal p : pila)
  var i : entero fvar
  para i := 1 hasta p.nelem hacer
       destruir(p.A[i]);
  fpara
faccion
```

Figura 32. Resultado de la operación `destruir(p)`, si la pila `p` se ha implementado con punteros.



```
accion destruir(entsal p : pila)
   var tmp : puntero a nodo fvar
   mientras p.prim ≠ NULO hacer
            tmp := p.prim^.suc;
            destruir(p.prim^.e);
            liberarEspacio(p.prim);
            p.prim := tmp;
   fmientras
faccion
```

Finalmente, como ejemplo, implementamos una acción que concatena dos listas y que incluye una llamada a la nueva operación `destruir` para liberar el espacio que ocupa la variable auxiliar `lb` y así evitar el problema de los retales:

```
accion concatenar(entsal l1 : lista; ent l2 : lista)
   var lb : lista fvar
   mientras ¬fin(l1) hacer
            l1 := avanzar(l1);
   fmientras
   lb := duplicar(l2);
   lb := principio(lb);
   mientras ¬fin(lb) hacer
            l1 := insertar(l1,actual(lb));
            lb := avanzar(lb);
   fmientras
   destruir(lb);
faccion
```

4.4.6. Conclusión

Los punteros ofrecen más posibilidades y flexibilidad a la hora de diseñar estructuras, pero también deben utilizarse con más cuidado, ya que son una fuente mayor de problemas.

Igualmente, siempre que implementemos un TAD incluiremos:

- Las operaciones indicadas en la signatura del tipo, porque son las que proporcionan la funcionalidad del mismo.

- Las operaciones `igual`, `duplicar` y `destruir`, porque hacen que el tipo sea transparente a la implementación.

En resumen, la implementación de un TAD debe incluir las operaciones de la signatura y las operaciones `igual`, `duplicar` y `destruir`.

4.5. Ejemplo de implementación definitiva: lista encadenada

```
tipo
  nodo = tupla
           e : elem;
           suc : puntero a nodo;
  ftupla
  lista = tupla
           prim,ant : puntero a nodo;
  ftupla
ftipo

accion crear(sal l : lista)
  si l.prim ≠ NULO entonces
     destruir(l);    {para evitar retales}
  fsi
  l.prim := obtenerEspacio();    {para el elemento fantasma}
  si l.prim = NULO entonces
     error {no hay espacio libre};
  sino
      l.ant := l.prim;
      l.ant^.suc := NULO;
  fsi
faccion

accion insertar(entsal l : lista; ent e : elem)
  var p : puntero a nodo fvar
  p := obtenerEspacio();
  si p = NULO entonces
     error {no hay espacio libre};
  sino
      duplicar(p^.e,e);
      p^.suc := l.ant^.suc;
      l.ant^.suc := p;
      l.ant := p;
  fsi
faccion
```

```
accion borrar(entsal l : lista)
  var p : puntero a nodo fvar
  si l.ant^.suc = NULO entonces
      error {fin de lista o lista vacia};
  sino
       p := l.ant^.suc;
       l.ant^.suc := p^.suc;
       destruir(p^.e);     {para evitar retales}
       liberarEspacio(p);
  fsi
faccion

accion principio(entsal l : lista)
  l.ant := l.prim;
faccion

accion avanzar(entsal l : lista)
  si l.ant^.suc = NULO entonces
      error {fin de lista o lista vacia};
  sino
       l.ant := l.ant^.suc;
  fsi
faccion

funcion actual(l : lista) : elem
  var e : elem fvar
  si l.ant^.suc = NULO entonces
      error {fin de lista o lista vacia};
  sino
       duplicar(e,l.ant^.suc^.e);
  fsi
  devuelve e;
ffuncion

funcion fin(l : lista) : booleano
  devuelve l.ant^.suc = NULO;
ffuncion

funcion vacia(l : lista) : booleano
  devuelve l.prim^.suc = NULO;
ffuncion
```

```
accion duplicar(sal l2 : lista; ent l1 : lista)
  var n1,n2 : puntero a nodo; e : elem fvar
  crear(l2);
  n1 := l1.prim^.suc;
  n2 := l2.prim;
  mientras n1 ≠ NULO hacer
          n2^.suc := obtenerEspacio();
          si n2^.suc = NULO entonces
             error {no hay espacio libre};
          sino
             n2 := n2^.suc;
             duplicar(n2^.e,n1^.e);
             si n1 = l1.ant entonces
                l2.ant := n2;
             fsi
             n1 := n1^.suc;
          fsi
  fmientras
  n2^.suc := NULO;
faccion

funcion igual(l1,l2 : lista) : booleano
  var ig : booleano; n1,n2 : puntero a nodo fvar
  n1 := l1.prim;
  n2 := l2.prim;
  ig := (igual(l1.ant,n1) ∧ igual(l2.ant,n2)) ∨ (¬igual(l1.ant,n1) ∧
     ¬igual(l2.ant,n2));    {No comparamos el contenido de los elementos fantasma}
  si ig entonces
     n1 := n1^.suc;
     n2 := n2^.suc;
  fsi
  mientras (n1 ≠ NULO) ∧ (n2 ≠ NULO) ∧ ig hacer
          ig := igual(n1^.e,n2^.e) ∧ ((igual(l1.ant,n1) ∧ igual(l2.ant,n2)) ∨
             (¬igual(l1.ant,n1) ∧ ¬igual(l2.ant,n2)));
          si ig entonces
             n1 := n1^.suc;
             n2 := n2^.suc;
          fsi
  fmientras
  devuelve (n1 = NULO) ∧ (n2 = NULO);
ffuncion
```

```
accion destruir(entsal l : lista)
   var tmp : puntero a nodo fvar
   mientras l.prim ≠ NULO hacer
            tmp := l.prim^.suc;
            destruir(l.prim^.e);
            liberarEspacio(l.prim);
            l.prim := tmp;
   fmientras
faccion
```

4.6. Otras variantes

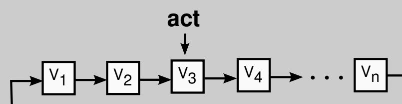
En el ámbito del diseño de estructuras de datos complejas, las listas encadenadas pueden presentar carencias e ineficiencias. Para afrontar esta situación, introducimos brevemente tres variantes de listas: las listas circulares, las listas doblemente encadenadas y las listas ordenadas.

4.6.1. Listas circulares

La **lista circular** recicla el encadenamiento del último elemento para que apunte al primer elemento de la lista. El resultado es una lista sin primero ni último.

La ventaja principal de este tipo de lista es que, dado un elemento, siempre se puede acceder a cualquier otro elemento de la lista, porque el último elemento de la lista siempre apunta al primero (figura 33).

Figura 33. Lista circular



La lista circular facilita la implementación de los algoritmos ya que, como todos los encadenamientos son del mismo tipo, nos evita tener que codificar un tratamiento especial para el último elemento.

Si añadimos un puntero a la lista circular para designar un elemento como el primero, entonces podremos variar el principio de la lista simplemente desplazando este puntero, mientras que en la lista encadenada era necesario desplazar elementos.

Para acabar, podemos implementar una cola con una lista circular añadiendo un puntero al último elemento, ya que el primer elemento siempre será el sucesor del último.

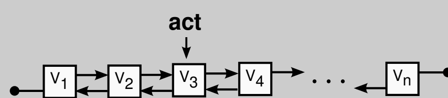
4.6.2. Listas doblemente encadenadas

La lista encadenada solo permite hacer recorridos en un sentido, ya que dado un elemento sólo conocemos su sucesor. Ello comporta que para conocer el elemento anterior se tenga que recorrer la lista desde el principio y que el coste de esta operación sea lineal. Por lo tanto, **si se quiere recorrer la lista en ambos sentidos (hacia adelante y hacia atrás) es necesario añadir encadenamientos en las dos direcciones**.

Esta es la idea de la **lista doblemente encadenada**, en la cual cada elemento tiene dos encadenamientos: uno al nodo anterior y otro al nodo siguiente (figura 34).

El coste de las operaciones será nuevamente constante, porque el acceso al elemento predecesor será inmediato, aun consumiendo algo más de espacio.

Figura 34. Lista doblemente encadenada

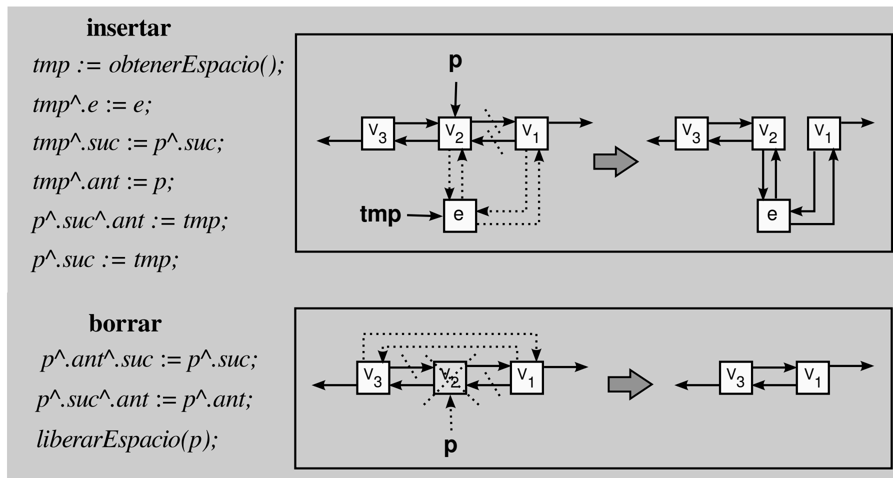


Pero el doble encadenamiento no solo es útil en los recorridos, también lo es **cuando la lista forma parte de una estructura de datos compleja y sus elementos son apuntados desde otros puntos de la estructura**. Fruto de esta relación, desde otros puntos de la estructura se pueden suprimir arbitrariamente elementos de la lista encadenada y entonces es cuando necesitamos acceder al predecesor del elemento suprimido para actualizar los encadenamientos de la lista (figura 35).

4.6.3. Listas ordenadas

Dada una lista cualquiera, a veces es necesario hacer un recorrido ordenado de los elementos por el valor de uno de sus campos. Este campo se llamará `clave`.

Figura 35. Esqueleto de las operaciones de inserción y borrado de un elemento de la lista



Para disponer de una **lista ordenada** no hay que modificar la representación, sino que basta con modificar la implementación de ciertas operaciones con la siguiente finalidad:

- mantener la lista desordenada en las actualizaciones y ordenarla antes de empezar un recorrido, o bien
- mantener la lista siempre ordenada.

Si escogemos la primera opción, entonces se tendrá que modificar la operación `principio`, que se convertirá en la operación más costosa, ya que tendrá que ordenar todos los elementos de la lista para hacer el recorrido. En cambio, si escogemos la segunda alternativa, la operación a modificar será `insertar`, ya que tiene que buscar secuencialmente el punto donde tiene que añadir el elemento para mantener la ordenación. En este caso, la operación más costosa será `insertar` pero, a cambio, la lista siempre estará ordenada.

Dependiendo de cómo se distribuyan las inserciones y los recorridos en una ejecución, nos interesará una alternativa u otra. Por ejemplo, si el algoritmo presenta dos fases diferenciadas, una primera en que se construye la lista y una segunda en que solo se consulta, entonces la primera alternativa parece la mejor, ya que las actualizaciones tendrán el coste mínimo y la lista se ordenará una sola vez (justo en el primer recorrido).

Resumen

En este módulo se han introducido los tipos básicos de datos para almacenar las secuencias: *pilas*, *colas* y *listas*. Los tres tipos ofrecen operaciones para acceder de una manera secuencial a los datos, pero se diferencian en la manera de hacerlo. Así pues, dependiendo de la situación en que nos encontremos, escogeremos el tipo que más se adecue al comportamiento requerido.

También hemos visto cómo se trabaja con un tipo de datos a partir de su signatura sin conocer la implementación. Esta es la característica principal de un *tipo abstracto de datos*. Asimismo, hemos comentado cómo se añaden *funciones externas* al tipo y como se integran diferentes tipos en una estructura de datos compleja.

Hemos aprendido a construir nuevos TAD para poder modelizar situaciones del mundo real. Primero, especificando la *signatura* del tipo con las operaciones que se podrán aplicar y luego abordando su *implementación*. La implementación consistirá en escoger una representación de los datos y codificar las operaciones evaluando su coste espacial y temporal.

Esta evaluación es importante porque, dada la especificación de un tipo, siempre habrá más de una implementación posible y necesitaremos un sistema para decidir cuál es la implementación más adecuada para nuestro problema.

Además, hemos estudiado cómo se almacenaban los datos aprovechando que la representación era *secuencial*, y hemos visto algunos de los problemas que presenta esta alternativa. Para superarlos, hemos introducido el concepto de representación *encadenada*, que consiste en especificar en la estructura de datos la relación que hay entre los elementos.

Otro tema en que se ha profundizado es en el uso de los *vectores* y de los *punteros*. Se han implementado los tipos básicos con ambos sistemas y se han comentado sus ventajas y desventajas. También se ha visto que es necesario añadir las operaciones igual, duplicar y destruir a la implementación de un tipo si se quiere garantizar un comportamiento idéntico independientemente de la implementación escogida.

Por otra parte, aprovechando el tema de los punteros, se ha explicado el sistema de *gestión de la memoria dinámica*, así como también los peligros que presenta (retales, referencias colgadas, etc.).

Para acabar, hemos visto brevemente tres variantes del tipo lista que aportan soluciones para situaciones especiales: *listas circulares*, *listas doblemente encadenadas* y *listas ordenadas*.

Actividades

1. Implementad dos pilas en un solo vector, de manera que el coste temporal sea Θ(1) y que el total de elementos almacenados no sobrepase la capacidad del vector.

2. El TAD *lista* es el más flexible de los explicados en el módulo, hasta el punto de considerar los TAD *cola* y *pila* como casos particulares de este. Para demostrarlo, implementad una pila y una cola usando una lista encadenada con punto de interés.

3. Ampliad el TAD *pila* (implementado con vectores) añadiendo las operaciones siguientes:

- `base(p)`, que devuelve el elemento que hace más tiempo que está apilado en la pila `p`;
- `ndesapilar(p,n)`, que desapila `n` elementos de la pila `p`. Si la pila contiene menos de `n` elementos, desapilará todos los que haya.

4. Ampliad el TAD *cola* implementado con punteros, añadiendo las operaciones siguientes:

- `num(c)`, que devuelve el número de elementos que hay en la cola `c`;
- `ndesencolar(c,n)`, que desencola `n` elementos de la cola `c`. Si no hay suficientes elementos en la cola para desencolar, se considerará una situación errónea.

5. Dado el TAD *lista* implementado en el módulo como una lista encadenada con punteros, añadid las operaciones siguientes:

- `leer(l,i)`, que devuelve el `i`-ésimo elemento situado a la derecha del punto de interés de la lista `l`;
- `esimo(l)`, que sitúa el punto de interés sobre el elemento `i`-ésimo respecto al principio de la lista `l`. Si la lista está vacía o no hay suficientes elementos, retorna el valor NULO.

6. En el módulo hemos visto implementadas las operaciones `destruir`, `duplicar` e `igual` para asegurar la transparencia de la implementación de los TAD *pila* y *lista*. Siguiendo el ejemplo de los TAD comentados, implementad estas operaciones en el TAD *cola*.

7. Diseñad una acción que muestre la representación binaria de un número leído de la secuencia de entrada. Para ello podéis usar una pila que almacene los resultados de las divisiones parciales por la base (que es 2). Por ejemplo, si la entrada es el número 77, la salida debería ser 1 0 0 1 1 0 1.

8. Construid un nuevo TAD denominado *pico* que mezcle las operaciones de las pilas y de las colas sobre una secuencia: `crear, apilar, encolar, desencolar, desapilar, cima, cola` y `vacia`.

9. Implementad una cola utilizando dos pilas. Codificad las operaciones básicas de la signatura del tipo cola.

10. Cread un TAD *conjunto* que represente un conjunto de elementos y que ofrezca como mínimo las operaciones siguientes:

- `añadir(s,e)`: añade el elemento `e` al conjunto `s`,
- `borrar(s,e)`: borra el elemento `e` del conjunto `s`,
- `union(s,t)`: añade los elementos del conjunto `t` al conjunto `s`,
- `interseccion(s,t)`: borra del conjunto `s` los elementos que no están en el conjunto `t`,
- `diferencia(s,t)`: borra del conjunto `s` los elementos que están en el conjunto `t`,
- `tamaño(s)`: devuelve el número de elementos del conjunto `s`.

11. Implementad un TAD para modelizar un editor de textos. Este TAD debería proporcionar las operaciones siguientes:

- `añadir(t,c)`: añade el carácter `c` en la posición del cursor y desplaza el cursor a la derecha,
- `borrar(t)`: borra el carácter donde está el cursor y desplaza el cursor al carácter de la izquierda,
- `izquierda(t)`: mueve el cursor un carácter a la izquierda,
- `derecha(t)`: mueve el cursor un carácter a la derecha,
- `inicio(t)`: mueve el cursor al inicio del texto o sobre el carácter situado más a la izquierda,
- `cursor(t)`: devuelve el carácter sobre el que está el cursor.

12. Escribid una acción `creciente` que, dada una lista de enteros, la devuelva ordenada de modo creciente. ¿Qué variación tendríais que hacer en el código para que la ordenación de la lista fuera estrictamente creciente?

13. Una emisora de radio en línea quiere automatizar la gestión de las listas de reproducción. Por este motivo decide organizar las canciones de que dispone en una estructura de datos. Los directivos de la emisora deciden que se necesitan las siguientes operaciones para programar la música a emitir:

- `crear: → radio`, la emisora tiene el archivo sonoro completamente vacío.
- `añadir: radio cancion → radio`, la emisora añade una canción nueva a su archivo sonoro.
- `sugerir: radio entero → cancion`, devuelve la canción que tiene la duración dada en segundos y que menos veces se ha emitido en la radio.
- `seleccionar: radio entero → cancion`, devuelve la canción que tiene la duración dada en segundos y que se ha emitido más veces.
- `emitir: radio cancion → radio`, se toma nota de que la canción se emitirá en la radio. Cada vez que se emite una canción se llama a esta operación.
- `borrar: radio entero → radio`, borra las *n* canciones que se han emitido menos veces. En caso de empate es indiferente la canción que se seleccione.

El número de canciones de que dispone la radio es conocido, pero también sabemos que cada semana la emisora adquiere nuevas canciones y las añade al archivo. Para cada canción se almacena un número que la identifica y su duración en segundos. La empresa os pide que defináis el TAD *radio* y que lo implementéis de manera que todas las operaciones tengan el mínimo coste temporal posible.

Ejercicios de autoevaluación

1. Dibujad el resultado de aplicar la secuencia de operaciones siguiente sobre una pila vacía `p`:
```
apilar(p,1); apilar(p,7); desapilar(p); apilar(p,2); apilar(p,3);
desapilar(p); apilar(p,5); apilar(p,9);
```

2. Dibujad el resultado de aplicar la secuencia de operaciones siguiente sobre una cola vacía `q`:
```
encolar(q,1); encolar(q,7); desencolar(q); encolar(q,2); encolar(q,3);
desencolar(q); encolar(q,5); encolar(q,9);
```

3. Escribid una función `balanceado` que lea una secuencia de entrada formada por paréntesis y corchetes y que compruebe que todos están correctamente balanceados utilizando una pila. Es decir, la función debe verificar que todos los paréntesis abiertos se cierran en el orden correspondiente. La secuencia debe acabar con un punto. Por ejemplo:

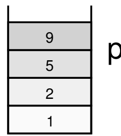
- Si la entrada es "([ ] (( ))) [ ] ." , retornará CIERTO.
- En cambio, si la entrada fuera "([( ))] ." , retornaría FALSO.

4. Escribid una acción `fibonacci` que calcule la sucesión de Fibonacci con ayuda de una cola. La sucesión de Fibonacci empieza así: 0 1 1 2 3 5 8 13 21 34 55 89 ...

Solucionario

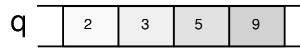
1. y **2.**

1. 2.



p

q



3. Cada vez que el algoritmo lee un corchete o paréntesis de apertura, tiene que apilar el símbolo en la pila, mientras que cada vez que encuentra un paréntesis o corchete de cierre en la secuencia de entrada, tiene que consultar la cima de la pila y comprobar que el símbolo de cierre leído se corresponde con el símbolo de apertura que hay en la cima. Si se corresponde, el algoritmo desapila la cima y prosigue la ejecución. En caso contrario, si no coincide o si la pila está vacía, la expresión no estará bien balanceada. La expresión estará bien balanceada si al leer el punto(.), que indica el final de la secuencia, la pila está vacía.

```
funcion balanceado() : booleano
  var ok : booleano; c : caracter; p : pila(caracter) fvar
  crear(p);
  ok := CIERTO;
  c := leerCaracter();
  mientras (c ≠ '.') ∧ ok hacer
          si (c = '(') ∨ (c = '[') entonces
             apilar(p,c);
             c := leerCaracter();
         sino
             si ¬vacia(p) entonces
                 si c = ')' entonces ok := cima(p) = '(';
                             sino ok := cima(p) = '[';
                 fsi
                 si ok entonces
                     desapilar(p);
                     c := leerCaracter();
                 fsi
            sino
                 ok := FALSO;
             fsi
          fsi
  fmientras
  devuelve ok ∧ vacia(p);
ffuncion
```

4.

```
accion fibonacci(ent n : entero)
  var q : cola(entero); a,b,i : entero; fvar
  crear(q);
  encolar(q,0);
  encolar(q,1);
  para i := 1 hasta n hacer
       a := cabeza(q);
       desencolar(q);
       b := cabeza(q);
       desencolar(q);
       encolar(q,b);
       encolar(q,a + b);
       escribirEntero(a);
  fpara
faccion
```

Glosario

abstracción *f* Simplificación de la realidad teniendo en cuenta solo una parte de esta.

actual *adj* Dicho del elemento distinguido de una lista, que sirve de referencia para aplicar las operaciones.

algoritmo *m* Conjunto de pasos para resolver un problema.

cabeza *m* Elemento situado al principio de una cola.

cima *f* Último elemento en llegar a una pila y que será el primero en salir de ella.

cola *f* Tipo abstracto de datos caracterizado por el hecho de que el primer elemento en entrar es el primero en salir.

coste constante $\Theta(1)$ *m* Medida que es fija y no depende del número de elementos.

coste espacial *m* Medida del espacio que ocupa una implementación concreta de un tipo de datos.

coste lineal $\Theta(n)$ *m* Medida que se relaciona linealmente con el número de elementos.

coste temporal *m* Medida del tiempo que requiere una operación en una implementación concreta de un tipo de datos.

encadenamiento *m* Indicador que identifica la posición que ocupa otro elemento.

estado *m* Descripción de una estructura en un instante determinado.

estructura circular *f* Estructura sin principio ni fin, en la cual tras la última posición viene la primera posición.

fantasma *m* Elemento especial que se guarda en la primera posición de una secuencia para que el elemento actual tenga siempre un predecesor.

FIFO *f First in first out*.

función externa *f* Función que no puede acceder a la representación del tipo y solo puede usar las operaciones facilitadas por la signatura del tipo.

función parcial *f* Función que no está definida para todo el dominio de los parámetros de entrada.

implementación *f* Codificación concreta de un tipo de datos.

LIFO *f Last in first out*.

lista *f* TAD caracterizado por el hecho de que permite añadir, borrar o consultar cualquier elemento de la secuencia.

lista con punto de interés *f* Lista que contiene un elemento distinguido apuntado por el punto de interés y que sirve de referencia para aplicar las operaciones.

lista circular *f* Lista en que se recicla el encadenamiento del último elemento para que apunte al primer elemento. El resultado es una lista sin primero ni último.

lista doblemente encadenada *f* Lista en que cada elemento tiene dos encadenamientos: uno al nodo anterior y otro al nodo siguiente.

lista ordenada *f* Lista en que los elementos están ordenados por el valor de uno de los campos. Este campo se denomina *clave*.

memoria dinámica *f* Mecanismo para solicitar memoria en tiempo de ejecución al sistema operativo.

módulo *m* Operación que calcula el resto de la división de dos números enteros.

NULO *m* Valor especial que indica que un puntero no apunta a ningún sitio.

operación constructora *f* Operación del tipo que devuelve un valor del mismo tipo.

operación consultora *f* Operación del tipo que devuelve un valor de otro tipo.

operación generadora *f* Operación que forma parte del conjunto mínimo de operaciones constructoras necesarias para generar cualquier valor del tipo.

operación modificadora *f* Operación constructora que no es generadora del tipo.

pila *f* Tipo abstracto de datos caracterizado por el hecho de que el último elemento en entrar es el primero en salir.

puntero *m* Variable que almacena la dirección de memoria en que empieza el objeto al que apunta.

referencia colgada *f* Puntero que apunta a un objeto que ya no existe. Si se intenta acceder a él el resultado es impredecible.

retal *m* Objeto que es inaccesible, porque no está apuntado por ningún puntero y no hay manera de acceder a él.

secuencia *f* Conjunto de elementos dispuestos en un orden específico. Fruto de esta ordenación, dado un elemento de la secuencia hablaremos del predecesor (como el elemento anterior) y del sucesor (como el elemento siguiente).

signatura *f* Especificación formal del comportamiento de las operaciones de un tipo. Para cada operación establece los parámetros, el resultado, las condiciones de error y las ecuaciones que reflejan su comportamiento.

tipo abstracto de datos *m* Tipo de datos al que se le ha añadido el concepto de abstracción para indicar que la implementación del tipo es invisible para los usuarios del tipo.
sigla **TAD**

tipo de datos *m* Conjunto de valores y de una serie de operaciones que se pueden aplicar sobre ellos. Las operaciones cumplirán ciertas propiedades que determinarán su comportamiento.

vacío -a *adj* Que, aplicado a una estructura, no contiene ningún elemento.

vector *m* Tabla unidimensional.

Bibliografía

Balcázar, J. L. (2001). *Programación metódica*. Madrid: McGraw-Hill.

Franch, X. (2001). *Estructures de dades. Especificació, disseny i implementació*. Barcelona: Edicions UPC.

Martí, N.; Ortega, Y.; Verdejo, J. A. (2001). *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*. Madrid: Pearson Educación.

Weiss, M. A (1995). *Estructuras de datos y algoritmos*. Madrid: Addison-Wesley.