Capítulo 3 Secuencias

Dado un alfabeto V, se define el conjunto de secuencias o cadenas de elementos de V (ing., sequence o string), denotado por V*, como:

- $\lambda \in V^*$.
- $\forall s \in V^*: \forall v \in V: v.s, s.v \in V^*$.

λ representa la secuencia vacía, mientras que el resto de secuencias se pueden definir como el añadido de un elemento (por la derecha o por la izquierda) a una secuencia ya existente. Así, la secuencia $s \in V^*$ se puede considerar como $s = v_0. \ldots .v_n . \lambda$, con $v_i \in V$, o, abreviadamente, $v_0 \ldots v_n$; diremos que v_0 es el primer elemento de la secuencia y v_n el último. También podemos decir que v_0 es el elemento de más a la izquierda de la secuencia y v_n el de más a la derecha, así como que v_{i+1} es el sucesor o siguiente de v_i y que v_i es el predecesor o anterior de v_{i+1} o igualmente, que v_{i+1} está a la derecha de v_i y que v_i está a la izquierda de v_{i+1}.

Las operaciones básicas que se definen sobre las secuencias son: crear la secuencia vacía, insertar un elemento del alfabeto dentro de una secuencia, y borrar y obtener un elemento de una secuencia. Para definir claramente el comportamiento de una secuencia es necesario determinar, pues, en qué posición se inserta un elemento nuevo y qué elemento de la secuencia se borra o se obtiene. En función de la respuesta, obtenemos diversos tipos abstractos diferentes, de los que destacan tres: pilas, colas y listas; su implementación se denomina tradicionalmente estructuras de datos lineales (ing., linear data structures) o, simplemente, estructuras lineales, siendo "lineal" el calificativo que indica la disposición de los elementos en una única dimensión.

3.1 Pilas

El TAD de las pilas (ing., stack) define las secuencias LIFO (abreviatura del inglés last-in, first-out, o "el último que entra es el primero que sale"): los elementos se insertan de uno en uno y se sacan, también de uno en uno, en el orden inverso al cual se han insertado; el único elemento que se puede obtener de dentro de la secuencia es, pues, el último insertado.

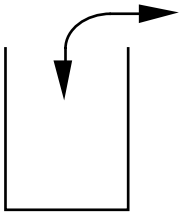


Fig. 3.1: representación gráfica de las pilas, dada su gestión.

Este comportamiento es habitual en la vida cotidiana; por ejemplo, si apilamos todos los platos de un banquete uno encima de otro, cualquier intento de obtener un plato del medio de la pila acaba irremediablemente con la ruptura de la vajilla. En la programación, las pilas no sólo se utilizan en el diseño de estructuras de datos, sino también como estructura auxiliar en diversos algoritmos y esquemas de programación; por ejemplo, en la transformación de algunas clases de funciones recursivas a iterativas (v. recorridos de árboles y grafos en los capítulos 5 y 6), o bien en la evaluación de expresiones (v. sección 7.1). Otra situación típica es la gestión de las direcciones de retorno que hace el sistema operativo en las llamadas a procedimientos de un programa imperativo (v. [HoS94, pp. 97-99]). Sea un programa P que contiene tres acciones A1, A2 y A3, que se llaman tal como se muestra en la fig. 3.2 (donde r, s y t representan la dirección de retorno de las llamadas a los procedimientos); durante la ejecución de A3 se disponen los puntos de retorno de los procedimientos dentro de una pila, tal como se muestra en la misma figura (m representa la dirección a la cual el programa ha de retornar el control al finalizar su ejecución), de manera que el sistema operativo siempre sabe dónde retornar el control del programa al acabar la ejecución del procedimiento en curso, obteniendo el elemento superior de la pila y eliminándolo seguidamente.

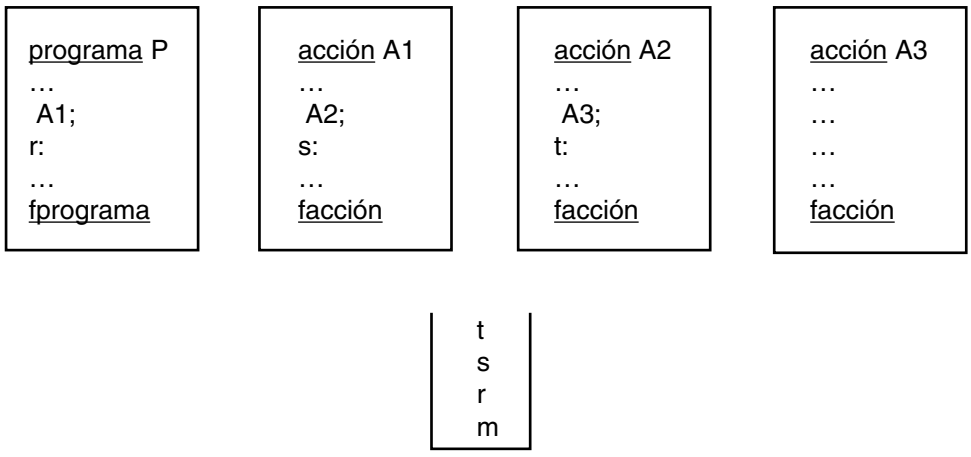


Fig. 3.2: un programa con tres acciones (arriba)
y las direcciones de retorno dentro de una pila (abajo).

3.1.1 Especificación

Dada una pila correspondiente a la secuencia $p \in V^*$, $p=v_0v_1\ldots v_n$, con $\forall i: 0 \leq i \leq n: v_i \in V$, y dado un elemento $v \in V$, se pueden definir diversas operaciones de interés:

- Crear la pila vacía: crea (ing., create), devuelve la pila λ.
- Insertar un elemento: empila(p, v) (ing., push), devuelve $v_0v_1\ldots v_nv$.
- Sacar un elemento: desempila(p) (ing., pop), devuelve $v_0v_1\ldots v_{n-1}$.
- Consultar un elemento: cima(p) (ing., top), devuelve v_n.
- Decidir si la pila está vacía: vacía?(p) (ing., empty?), devuelve cierto si $p = \lambda$ y falso en caso contrario.

En la fig. 3.3 se muestra una especificación para el tipo de las pilas infinitas de elementos cualesquiera con estas operaciones. La signatura establece claramente los parámetros y los resultados de las operaciones, y las ecuaciones reflejan el comportamiento de las diferentes operaciones sobre todas las pilas, dado que una pila es, o bien la pila vacía representada por crea, o bien una pila no vacía representada por empila(p, v), donde v es el último elemento añadido a la pila (como mínimo, hay uno) y p es la pila sin ese último elemento (dicho en otras palabras, el conjunto de constructoras generadoras del tipo es {crea, empila}). Destacamos la importancia de identificar las condiciones de error resultantes de sacar o consultar un elemento de la pila vacía. El identificador elem denota el tipo de los elementos, especificado en el universo de caracterización ELEM (v. fig. 1.27).

```
universo PILA(ELEM) define
    usa BOOL
    tipo pila
    ops
        crea: → pila
        empila: pila elem → pila
        desempila: pila → pila
        cima: pila → elem
        vacía?: pila → bool
    errores desempila(crea); cima(crea)
    ecns ∀p∈pila; ∀v∈elem
        desempila(empila(p, v)) = p; cima(empila(p, v)) = v
        vacía?(crea) = cierto; vacía?(empila(p, v)) = falso
funiverso
```

Fig. 3.3: especificación del TAD de las pilas.

3.1.2 Implementación

La representación más sencilla de las pilas sigue un esquema similar a la implementación de los conjuntos presentada en el apartado 2.1.4: los elementos se almacenan dentro de un vector tal que dos elementos consecutivos de la pila ocupan dos posiciones consecutivas en él, y se usa un apuntador de sitio libre que identifica rápidamente la posición del vector afectada por las operaciones de empilar, desempilar y consultar (v. fig. 3.4); es la denominada representación secuencial de las pilas. Con este apuntador ya no es necesario nada más, ya que el primer elemento de la pila ocupa la primera posición del vector, el segundo, la segunda, etc., hasta llegar al apuntador de sitio libre. El comportamiento de las operaciones de inserción y supresión de las pilas sobre esta implementación se muestra esquemáticamente en la fig. 3.5.

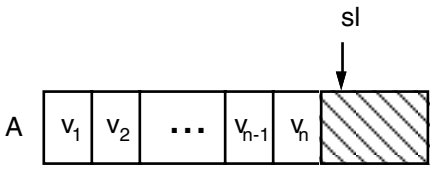


Fig. 3.4: representación secuencial de la pila $v_1v_2... v_n$.

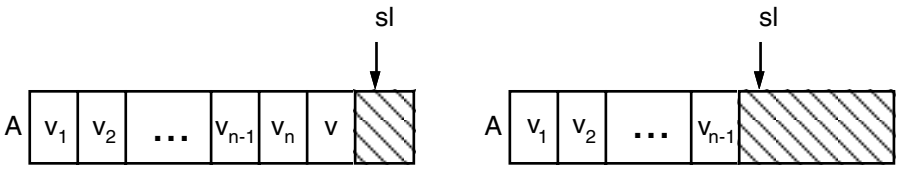


Fig. 3.5: apilamiento de un elemento (izq.) y desapilamiento (der.) en la pila de la fig. 3.4.

En la fig. 3.6 aparece el universo de implementación del tipo. Como ya se ha dicho en la sección 1.6, la implementación por vectores obliga a fijar el número de elementos que caben en el objeto, de manera que la pila ya no es de dimensión infinita y por ello la especificación correspondiente no es la de la fig. 3.3, sino que sería necesario modificarla aplicando los razonamientos vistos en el caso de los conjuntos. Entre otras cosas, el TAD ofrece una operación para comprobar si la pila está llena? para que el usuario pueda detectar esta situación antes de provocar el correspondiente error.

```
universo PILA_SEC(ELEM, VAL_NAT) es
    implementa PILA(ELEM, VAL_NAT)
    usa NAT, BOOL
    tipo pila es tupla
                    A es vector [de 0 a val-1] de elem
                    sl es nat
                ftupla
    ftipo
    invariante (p es pila): p.sl ≤ val
    función crea devuelve pila es
    var p es pila fvar
        p.sl := 0
    devuelve p
    función empila (p es pila; v es elem) devuelve pila es
        si p.sl = val entonces error   {pila llena}
        si no p.A[p.sl] := v; p.sl := p.sl+1
        fsi
    devuelve p
    función desempila (p es pila) devuelve pila es
        si p.sl = 0 entonces error   {pila vacía}
        si no p.sl := p.sl-1
        fsi
    devuelve p
    función cima (p es pila) devuelve elem es
    var v es elem fvar
        si p.sl = 0 entonces error   {pila vacía}
        si no v := p.A[p.sl-1]
        fsi
    devuelve v
    función vacía? (p es pila) devuelve bool es
    devuelve p.sl = 0
    función llena? (p es pila) devuelve bool es
    devuelve p.sl = val
funiverso
```

Fig. 3.6: implementación secuencial de las pilas.

El coste temporal de las operaciones es óptimo, $\Theta(1)$, siempre y cuando el coste de duplicar elementos sea negligible[1]; en cambio, el coste espacial es pobre, porque una pila tiene un espacio reservado $\Theta(val)$, independientemente del número de elementos que la formen en un momento dado[2], y considerando negligible el espacio ocupado por los elementos.

Siendo ésta la primera implementación vista, es interesante plantearnos cómo podríamos demostrar su corrección, y lo haremos siguiendo el método axiomático. Nos restringiremos a la demostración de la corrección de una operación, empila, siendo similar la estrategia para el resto de operaciones de la signatura. La especificación axiomática para empila es:

$$\{ sl_1 \leq val \}$$
$$<A_2, sl_2> := empila(<A_1, sl_1>, v)$$
$$\{ sl_2 \leq val \wedge abs_{pila}(<A_2, sl_2>) = empila(abs_{pila}(<A_1, sl_1>), v) \}$$

Como la demostración involucrará operaciones de vectores, definimos en la fig. 3.7 una especificación para los vectores indexados con índices naturales de cero a val-1 (la extensión a vectores más generales queda como ejercicio para el lector). La invocación as(A, i, v) representa la asignación A[i] := v mientras que cons(A, i) representa la indexación A[i] en el contexto de una expresión.

universo VECTOR(ELEM, VAL_NAT) és
  usa NAT, BOOL
  tipo vector
  ops crea: → vector
    as: vector nat elem → vector
    cons: vector nat → elem
  errores ∀A∈vector; ∀i∈nat; ∀v∈elem
    [i ≥ val] ⇒ as(A, i, v), cons(A, i); cons(crea, i)
  ecns ∀A∈vector; ∀i,j∈nat; ∀v,w∈elem
    (E1) as(as(A, i, v), i, w) = as(A, i, w)
    (E2) [¬NAT.ig(i, j)] ⇒ as(as(A, i, v), j, w) = as(as(A, j, w), i, v)
    (E3) cons(as(A, i, v), i) = v
    (E4) [¬NAT.ig(i, j)] ⇒ cons(as(A, i, v), j) = cons(A, j)
funiverso

Fig. 3.7: especificación de un TAD para los vectores.

[1] Para simplificar el estudio de la eficiencia, en el resto del libro nos centraremos en el coste intrínseco de las operaciones del TAD, sin considerar el coste de las duplicaciones, comparaciones y otras operaciones elementales de los parámetros formales de los universos que los definen; si dichos costes no fueran negligibles, deberían considerarse para obtener la eficiencia auténtica del TAD.

[2] Este hecho no es especialmente importante si la pila ha de crecer en algun momento hasta val. De todas formas, el problema se agrava cuando en vez de una única pila existen varias (v. ejercicio 3.20).

Para empezar, debe determinarse la función de abstracción y la relación de igualdad, amén del invariante de la representación, ya establecido. La función de abstracción es similar al caso de los conjuntos, lo cual es lógico ya que la estrategia de representación es idéntica:

(A1) $abs_{pila}(<A, 0>) = crea$
(A2) $abs_{pila}(<A, x+1>) = empila(abs_{pila}(<A, x>), A[x])$

Por lo que respecta a la relación de igualdad, es incluso más simple ya que, dadas las representaciones de dos pilas, los elementos en la parte ocupada de los correspondientes vectores deben ser los mismos y en el mismo orden:

(R) $R_{pila}(<A_1, sl_1>, <A_2, sl_2>) = (sl_1 = sl_2) \wedge (\forall k: 0 \leq k \leq sl_1\text{-}1: A_1[k] = A_2[k])$

Supongamos que el código de empila que se propone inicialmente para cumplir esta especificación consiste simplemente en la asignación $<A_2, sl_2> := <as(A_1, sl_1, v), sl_1+1>$. Es decir, la demostración que debe realizarse es:

$$sl_1 \leq val \Rightarrow (sl_1 + 1 \leq val \wedge abs_{pila}(<as(A_1, sl_1, v), sl_1+1>) = empila(abs_{pila}(<A_1, sl_1>), v))$$

Estudiamos primero la satisfacción de la segunda fórmula de la conclusión. La igualdad se puede demostrar de la manera siguiente:

$abs_{pila}(<as(A_1, sl_1, v), sl_1+1>) =$ { aplicando (A2) }
$empila(abs_{pila}(<as(A_1, sl_1, v), sl_1>), cons(as(A_1, sl_1, v), sl_1)) =$ { aplicando (E3)}
$empila(abs_{pila}(<as(A_1, sl_1, v), sl_1>), v) =$ { conjetura }
$empila(abs_{pila}(<A_1, sl_1>), v)$

La conjetura que aparece, que se puede formular como lema, establece que una asignación en una posición libre del vector no afecta el resultado de la función de abstracción. La demostración es obvia aplicando (R) ya que el predicado $R_{pila}(<as(A_1, sl_1, v), sl_1>, <A_1, sl_1>)$ efectivamente evalúa a cierto.

Notemos, no obstante, que la invocación $as(A_1, sl_1, v)$ provoca un error si $sl_1 = val$, el valor máximo del vector, que es un valor posible según la precondición; por esto, el código de empila debe proteger la asignación con una condición, tal como aparece en la fig. 3.6. De paso, esta protección asegura que también se cumple la primera conjunción de la postcondición, es decir, el invariante de la representación de la pila resultado, con lo que queda definitivamente verificada la implementación de empila. Otra alternativa válida consiste en modificar la precondición de empila para evitar este valor del apuntador de sitio libre.

3.2 Colas

En esta sección se presenta el segundo modelo clásico de secuencia: el tipo de las colas (ing., queue). La diferencia entre las pilas y las colas estriba en que en estas últimas los elementos se insertan por un extremo de la secuencia y se extraen por el otro (política FIFO: first-in, first-out, o "el primero que entra es el primero que sale"); el elemento que se puede consultar en todo momento es el primero insertado.

Las colas aparecen en diversos ámbitos de la actividad humana; la gente hace cola en los cines, en las panaderías, en las librerías para comprar este libro, etc. También tienen aplicación en la informática; una situación bien conocida es la asignación de una CPU a procesos de usuarios por el sistema operativo con una política round-robin sin prioridades, donde los procesos que piden el procesador se encolan de manera que, cuando el que se está ejecutando actualmente acaba, pasa a ejecutarse el que lleva más tiempo esperando.

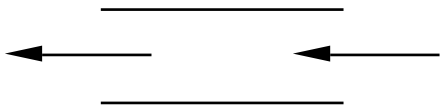


Fig. 3.8: representación gráfica de las colas, dada su gestión.

3.2.1 Especificación

Dada una cola correspondiente a la secuencia $c \in V^*$, $c=v_0v_1\ldots v_n$, con $\forall i : 0 \leq i \leq n : v_i \in V$, y dado un elemento $v \in V$, definimos las operaciones siguientes:

- Crear la cola vacía: crea, devuelve la cola λ.
- Insertar un elemento: encola(c, v) (ing., put o enqueue), devuelve $v_0v_1\ldots v_nv$.
- Sacar un elemento: desencola(c) (ing., tail o dequeue), devuelve $v_1\ldots v_n$.
- Consultar un elemento: cabeza(c) (ing., head o first), devuelve v_0.
- Decidir si la cola está vacía: vacía?(c), devuelve cierto si $c = \lambda$ y falso en caso contrario.

Como es habitual, el tipo cola se encapsula en un universo parametrizado por el tipo de los elementos, tal como se muestra en la fig. 3.9. En la especificación de desencola y cabeza se distingue el comportamiento de las operaciones sobre la cola vacía, la cola con un único elemento y la cola con más de un elemento, y así se cubren todos los casos. Es un error intentar desencolar o consultar la cabeza de la cola vacía, desencolar un elemento de una cola con un único elemento da como resultado la cola vacía, la cabeza de una cola con un único elemento es ese elemento, y desencolar o consultar la cabeza de una cola con más de un elemento se define recursivamente sobre la cola que tiene un elemento menos.

universo COLA (ELEM) define
  usa BOOL
  tipo cola
  ops
    crea: → cola
    encola: cola elem → cola
    desencola: cola → cola
    cabeza: cola → elem
    vacía?: cola → bool
  errores desencola(crea); cabeza(crea)
  ecns ∀c∈ cua; ∀v∈ elem
    desencola(encola(crea, v)) = crea
    [¬ vacía?(c)] ⇒ desencola(encola(c, v)) =
                    encola(desencola(c), v)
    cabeza(encola(crea, v)) = v
    [¬ vacía?(c)] ⇒ cabeza(encola(c, v)) = cabeza(c)
    vacía?(crea) = cierto
    vacía?(encola(c, v)) = falso
funiverso

Fig. 3.9: especificación del TAD de las colas.

3.2.2 Implementación

Como en el caso de las pilas, se distribuyen los elementos secuencialmente dentro de un vector y, además, se necesitan:

- Apuntador de sitio libre, sl, a la posición donde insertar el siguiente elemento.
- Apuntador al primer elemento de la cola, prim: dado que la operación destructora de las colas elimina el primer elemento, si éste residiese siempre en la primera posición sería necesario mover todos los elementos una posición a la izquierda en cada supresión, solución extremadamente ineficiente, o bien marcar los elementos borrados, con lo que se malgastaría espacio y sería necesario regenerar periódicamente el vector. La solución a estos problemas consiste en tener un apuntador al primer elemento que se mueva cada vez que se desencola un elemento. Una consecuencia de esta estrategia es que uno de los apuntadores (o ambos) puede llegar al final del vector sin que todas las posiciones del vector estén ocupadas (es decir, sin tener una cola totalmente llena); para reaprovechar las posiciones iniciales sin mover los elementos del vector, se gestiona éste como una estructura circular (sin principio ni final), en la que el primer elemento suceda al último (v. fig. 3.10).

- Indicador de si está llena o vacía: la política de implementación circular no da suficiente información para decidir si una cola está vacía o no, porque la condición de cola vacía es la misma que la de cola llena, prim = sl ; por eso se puede añadir un booleano a la representación o bien un contador de elementos; la segunda opción es especialmente útil si se quiere implementar dentro del universo de definición de las colas una operación que dé su tamaño.

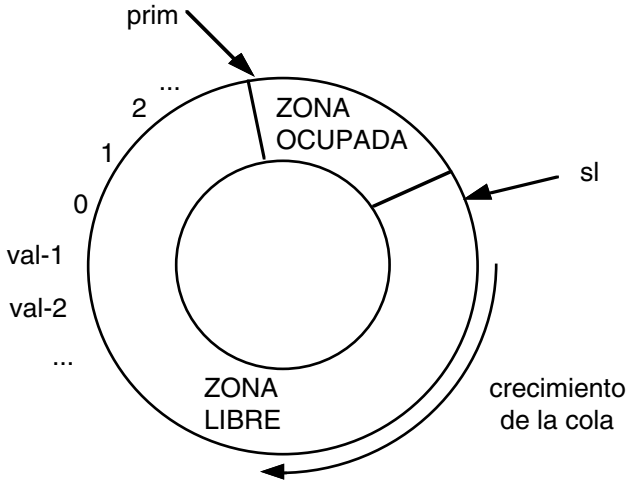


Fig. 3.10: implementación de una cola dentro de un vector circular de val posiciones.

En la fig. 3.11 se presenta una implementación en Merlí de las colas acotadas que sigue todas estas consideraciones, estableciendo el invariante de la representación que acota los valores válidos de los apuntadores y del contador y, además, los relaciona. Notemos el uso de mod (la operación que devuelve el resto de la división de dos números naturales) para tratar el vector circularmente. El coste temporal de las operaciones es $\Theta(1)$ y el coste espacial de la estructura queda $\Theta(val)$, como cabía esperar en ambos casos.

En la literatura sobre el tema se encuentran algunas mínimas variantes de este esquema, de las que destacamos dos:

- Se puede evitar el booleano o contador adicional desaprovechando una posición del vector, de manera que la condición de cola llena pase a ser que sl apunte a la posición anterior (módulo val) a la que apunta prim, mientras que la condición de cola vacía continua siendo la misma.
- Se puede evitar el apuntador sl sustituyendo cualquier referencia al mismo por la expresión (c.prim + c.cnt) mod val, tal como determina el invariante.

```
universo COLA_CIRCULAR (ELEM, VAL_NAT) es
    implementa COLA (ELEM, VAL_NAT)
    usa NAT, BOOL
    tipo cola es
            tupla
                A es vector [de 0 a val-1] de elem
                prim, sl, cnt son nat
            ftupla
    ftipo
    invariante (c es cola):
            c.prim < val  ∧  c.cnt ≤ val  ∧  c.sl = (c.prim+c.cnt) mod val
    función crea devuelve cola es
    var c es cola fvar
        c.prim := 0; c.sl := 0; c.cnt := 0
    devuelve c
    función encola (c es cola; v es elem) devuelve cola es
        si c.cnt = val entonces error  {cola llena}
        si no c.A[c.sl] := v; c.sl := (c.sl+1) mod val; c.cnt := c.cnt+1
        fsi
    devuelve c
    función desencola (c es cola) devuelve cola es
        si c.cnt = 0 entonces error          {cola vacía}
        si no c.prim := (c.prim+1) mod val; c.cnt := c.cnt-1
        fsi
    devuelve c
    función cabeza (c es cola) devuelve elem es
    var v es elem fvar
        si c.cnt = 0 entonces error          {cola vacía}
        si no v := c.A[c.prim]
        fsi
    devuelve v
    función vacía? (c es cola) devuelve bool es
    devuelve c.cnt = 0
    función llena? (c es cola) devuelve bool es
    devuelve c.cnt = val
funiverso
```

Fig. 3.11: implementación de las colas con un vector circular.

3.3 Listas

Las listas (ing., list) son la generalización de los dos tipos abstractos anteriores: mientras que en una pila y en una cola las operaciones sólo afectan a un extremo de la secuencia, en una lista se puede insertar un elemento en cualquier posición, y borrar y consultar cualquier elemento.

El uso de listas en la informática es tan antiguo como la existencia misma de los ordenadores y ha dado lugar a diversos modelos del tipo. Uno de los más habituales, que estudiaremos aquí, son las llamadas listas con punto de interés[3]. Asimismo, veremos algunas variantes en su uso e implementación. Hay que destacar, no obstante, que frecuentemente se pueden combinar características de éste y de otros modelos para formar otros nuevos, siempre que el resultado se adapte mejor a un contexto dado.

3.3.1 Especificación de las listas con punto de interés

En las listas con punto de interés se define la existencia de un elemento distinguido dentro de la secuencia, que es el que sirve de referencia para las operaciones, y este rol de distinción puede cambiarse de elemento a elemento con la aplicación de otras funciones del tipo. Decimos que el punto de interés de la lista está sobre el elemento distinguido. También puede ocurrir, no obstante, que esté más allá del último elemento de la lista, en cuyo caso decimos que el punto de interés está a la derecha de todo; en cambio, el punto de interés no puede estar nunca a la izquierda del primer elemento.

Un ejemplo inmediato de este modelo es la línea actual de texto en un terminal, donde el cursor representa el punto de interés y sirve de referencia para las operaciones de insertar y borrar caracteres, mientras que las flechas y, usualmente, algunas teclas o combinaciones de teclas sirven para mover el cursor por la línea de manera que el punto de interés puede llegar a situarse sobre cualquier carácter.

La existencia del punto de interés conduce a un modelo de secuencia diferente a los vistos hasta ahora. En éste se distingue el trozo de secuencia a la izquierda del punto de interés y el trozo que va del elemento distinguido al de la derecha del todo, y por eso el modelo es realmente $V^* \times V^*$. Notemos que cualquiera de las dos partes puede ser λ según la situación del punto de interés. En consecuencia, dada una lista l, $l = \langle s, t \rangle \in V^* \times V^*$, $s = s_0 s_1 \ldots s_n$, $t = t_0 t_1 \ldots t_m$, con $\forall i, j : 0 \leq i \leq n \wedge 0 \leq j \leq m : s_i, t_j \in V$, y dado un elemento $v \in V$, se definen las operaciones:

- Crear la lista vacía: crea, devuelve la lista $\langle \lambda, \lambda \rangle$; es decir, devuelve una lista sin elementos y el punto de interés queda a la derecha del todo.

[3] El nombre no es estándar; de hecho, no existe una nomenclatura unificada para los diferentes modelos de lista.

- Insertar un elemento: inserta(l, v) (ing., insert), devuelve $<s_0s_1\ldots s_nv, t_0t_1\ldots t_m>$; es decir, inserta delante del punto de interés, que no cambia.
- Sacar un elemento: borra(l) (ing., delete o remove), devuelve $<s_0s_1\ldots s_n, t_1\ldots t_m>$; es decir, borra el elemento distinguido (en caso de que el punto de interés esté a la derecha del todo, da error) y el punto de interés queda sobre el siguiente elemento (o a la derecha del todo, si el borrado es el último).
- Consultar un elemento: actual(l) (ing., current o get), devuelve t_0; es decir, devuelve el elemento distinguido (o da error si el punto de interés está a la derecha del todo).
- Decidir si la lista está vacía o no: vacía?(l), devuelve cierto si $l = <\lambda, \lambda>$ y falso en caso contrario.
- Poner al inicio el punto de interés: principio(l) (ing., reset), devuelve $<\lambda, s \,.\, t>$; es decir, el punto de interés queda sobre el primer elemento, si hay, o bien a la derecha del todo, si no hay ninguno.
- Avanzar el punto de interés: avanza(l) (ing., next), devuelve $<s_0s_1\ldots s_nt_0, t_1\ldots t_m>$; es decir, el elemento distingido pasa a ser el siguiente al actual (o da error si el punto de interés está a la derecha del todo).
- Mirar si el punto de interés está a la derecha del todo: final?(l) (ing., end? o eol?), devuelve cierto si $t = \lambda$ y falso en caso contrario.

Esta signatura permite implementar de manera sencilla los dos esquemas de programación más usuales sobre secuencias: el esquema de recorrido y el esquema de búsqueda (v. fig. 3.12). En el esquema de recorrido se aplica un tratamiento T sobre todos los elementos de la lista, mientras que en el esquema de búsqueda se explora la lista hasta encontrar un elemento que cumpla una propiedad A o bien hasta llegar al final; en caso de encontrarse dicho elemento, el punto de interés queda situado sobre él. Por lo que respecta a la búsqueda, la situación más habitual consiste en comprobar si un elemento particular está o no dentro de la lista En las pre y postcondiciones y también en los invariantes de estas y otras funciones, usaremos una operación $\_\in\_$ tal que, dado un elemento v y una lista l, $v \in l$ comprueba si v está dentro de l y también dos funciones pi y pd que, dada una lista $l = <s, t>$, devuelven s y t, respectivamente.

Precisamente el esquema de recorrido será uno de los más usados en los algoritmos sobre listas; es por este motivo por el que en el resto del texto seguiremos una notación abreviada que clarificará su escritura: siendo l una lista con punto de interés de elementos de tipo V y siendo v una variable de tipo V, el bucle:

```
para todo v dentro de l hacer
        tratar v
fpara todo
```

```
{l = <v0v1…vk-1, vk…vn}                {l = <v0v1…vk-1, vk…vn}
l := principio(l)                      l := principio(l); está := falso
mientras ¬ final?(l) hacer             mientras ¬ final?(l) ∧ ¬ está hacer
    { I ≅ ∀i: 0 ≤ i ≤ ||pi(l)||: T(vi) }       { I ≅ (∀i: 0 ≤ i ≤ ||pi(l)||: ¬ A(vi)) ∧ está ⇒ A(actual(l)) }
    T(actual(l))                       si A(actual(l)) entonces está := cierto
    l := avanza(l)                                 si no l := avanza(l)
fmientras                              fsi
{∀i: 0 ≤ i ≤ n: T(vi)}                 fmientras
                                       {está ≡ ∃i: 0 ≤ i ≤ n: A(vi) ∧ está ⇒ A(actual(l))}
```

Fig. 3.12: esquemas de secuencias sobre listas con punto de interés (izquierda, recorrido; derecha, búsqueda).

es equivalente a la secuencia de instrucciones:

```
l := principio(l)
mientras ¬ final?(l) hacer
        tratar actual(l); l := avanza(l)
fmientras
```

La especificación algebraica de este tipo de listas se basa precisamente en el modelo que se acaba de introducir[4], donde se considera que una lista es un par de secuencias, concretamente pilas dadas las operaciones que se necesitan. Estas dos pilas están confrontadas, de manera que la pila de la izquierda tiene la cima justo antes del punto de interés y la pila de la derecha tiene como cima el punto de interés, y sus sentidos de crecimiento son opuestos. Sobre estas pilas se necesita un enriquecimiento, concatena, que pasa todos los elementos de la pila de la izquierda a la pila de la derecha; la operación se especifica convenientemente en el mismo universo de las listas. El universo genérico (v. fig. 3.13) queda finalmente definido sobre los pares de dos pilas (la secuencia de la izquierda y la secuencia de la derecha) con una operación privada que es la constructora generadora del género. Observemos que las operaciones del TAD lista han sido especificadas respecto a las constructoras generadoras del TAD pila donde ha sido necesario. En concreto, se ha distinguido la pila vacía (representada por crea) de la pila no vacía (representada por empila(p, v)) allí donde ha convenido. El resultado es correcto porque toda operación ha quedado definida para todas las listas posibles. Destaquemos también que la instancia de las pilas ha sido efectuada con el parámetro formal de las listas, y como los dos parámetros se definen en el mismo universo de caracterización, es necesario bautizarlos en la cabecera de los universos genéricos para poder distinguir el tipo elem en ambos contextos. La instancia es privada porque el usuario de las listas no ha de tener acceso.

[4] También se podría considerar el conjunto de constructoras generadoras {crea, inserta, principio} y especificar por el método tradicional, pero el resultado sería más complicado.

Una modificación interesante del tipo consiste en hacer que el punto de interés sólo sea significativo en las operaciones de lectura y que no afecte ni a la inserción ni a la supresión de elementos, borrando siempre el primer elemento (o el último) e insertando siempre por el inicio (o por el final). Incluso podríamos tener diversos tipos de operaciones de inserción y supresión dentro del mismo universo. También se podría añadir una operación de modificar el elemento distinguido sin cambiar el punto de interés. Estas variantes y otras aparecen en diversos textos y su especificación y posterior implementación quedan como ejercicio para el lector.

universo LISTA_INTERÉS (A es ELEM) es
  usa BOOL
  instancia privada PILA (B es ELEM) donde B.elem es A.elem
  ops privada concat: pila pila → pila
  ecns ∀p,p_1,p_2∈ pila; ∀v∈ elem
    concat(crea, p) = p
    concat(empila(p_1, v), p_2) = concat(p_1, empila(p_2, v))
  tipo lista
  ops
    privada <_, _>: pila pila → lista
    crea: → lista
    inserta: lista elem → lista
    borra, principio, avanza: lista → lista
    actual: lista → elem
    final?, vacía?: lista → bool
  errores ∀p∈ pila
    borra(<p, PILA.crea>); avanza(<p, PILA.crea>)
    actual(<p, PILA.crea>)
  ecns ∀p,p_1,p_2∈ pila; ∀v∈ elem
    crea = <PILA.crea, PILA.crea>
    inserta(<p_1, p_2>, v) = <PILA.empila(p_1, v), p_2>
    borra(<p_1, PILA.empila(p_2, v)>) = <p_1, p_2>
    principio(<p_1, p_2>) = <PILA.crea, concat(p_1, p_2)>
    avanza(<p_1, PILA.empila(p_2, v)>) = <PILA.empila(p_1, v), p_2>
    actual(<p_1, PILA.empila(p_2, v)>) = v
    final?(<p_1, p_2>) = PILA.vacía?(p_2)
    vacía?(<p_1, p_2>) = PILA.vacía?(p_1) ∧ PILA.vacía?(p_2)
funiverso

Fig. 3.13: especificación del TAD lista con punto de interés.

3.3.2 Implementación de las listas con punto de interés

Para implementar las listas con punto de interés se puede optar entre dos estrategias:

- Representación secuencial. Los elementos se almacenan dentro de un vector y se cumple que elementos consecutivos en la lista ocupan posiciones consecutivas en el vector.
- Representación encadenada. Se rompe la propiedad de la representación secuencial y se introduce el concepto de encadenamiento: todo elemento del vector identifica explícitamente la posición que ocupa su sucesor en la lista.

a) Representación secuencial

Mostrada en las figs. 3.14 y 3.15, su filosofía es idéntica a la representación de las pilas y las colas; sólo es necesario un apuntador adicional al elemento distinguido para poder aplicar las operaciones propias del modelo de lista. El problema principal de esta implementación es evidente: una inserción o supresión en cualquier posición del vector obliga a desplazar algunos elementos para no tener posiciones desaprovechadas, lo que resulta en un coste lineal, inadmisible cuando estas operaciones son frecuentes o la dimensión de los elementos es muy grande; además, los movimientos de elementos propios de una representación secuencial son especialmente inconvenientes en el contexto de una estructura de datos compleja que tiene apuntadores desde diferentes componentes hacia el vector, porque el movimiento de un elemento de una posición a otra exige actualizar el valor de estos apuntadores, tarea ésta que puede ser laboriosa. La resolución de estos problemas lleva a la representación encadenada.

```
universo LISTA_INTERÉS_SEC(ELEM, VAL_NAT) es
    implementa LISTA_INTERÉS(ELEM, VAL_NAT)
    usa NAT, BOOL
    tipo lista es
        tupla
            A es vector [de 0 a val-1] de elem
            act, sl son nat
        ftupla
    ftipo
    invariante (l es lista): l.act ≤ l.sl ≤ val
    función crea devuelve lista es
    var l es lista fvar
        l.act := 0; l.sl := 0
    devuelve l
```

Fig. 3.14: implementación secuencial de las listas.

```
función inserta (l es lista; v es elem) devuelve lista es
var i es nat fvar
    si l.sl = val entonces error  {lista llena}
    si no {desplazamiento de los elementos a la derecha del punto de interés una
           posición a la derecha, para hacer sitio}
        para todo i desde l.sl bajando hasta l.act+1 hacer l.A[i] := l.A[i-1] fpara todo
        l.sl := l.sl+1; l.A[l.act] := v; l.act := l.act+1
    fsi
devuelve l

función borra (l es lista) devuelve lista es
var i es nat fvar
    si l.act = l.sl  entonces error  {final de lista o lista vacía}
    si no {desplazamiento de los elementos a la derecha del punto de interés una
           posición a la izquierda, para sobreescribir}
        para todo i desde l.act hasta l.sl-2 hacer l.A[i] := l.A[i+1] fpara todo
        l.sl := l.sl-1
    fsi
devuelve l

función principio (l es lista) devuelve lista es
    l.act := 0
devuelve l

función actual (l es lista) devuelve elem es
var v es elem fvar
    si l.act = l.sl entonces error   {final de lista}
    si no v := l.A[l.act]
    fsi
devuelve v

función avanza (l es lista) devuelve lista es
    si l.act = l.sl entonces error si no l.act := l.act + 1 fsi
devuelve l

función final? (l es lista) devuelve bool es
devuelve l.act = l.sl

función vacía? (l es lista) devuelve bool es
devuelve l.sl = 0

función llena? (l es lista) devuelve bool es
devuelve l.sl = val

funiverso
```

Fig. 3.14: implementación secuencial de las listas (cont.).

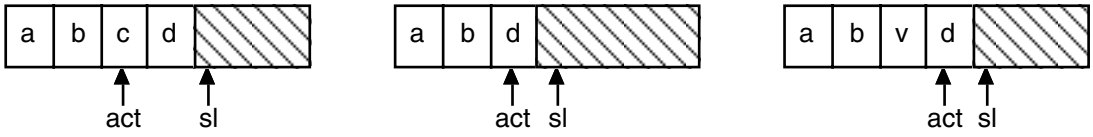


Fig. 3.15: operaciones sobre listas con punto de interés: izquierda, la lista <ab, cd>; medio, supresión del elemento distingido; derecha, inserción del elemento v en la lista resultante.

b) Representación encadenada

El objetivo de la estrategia encadenada (ing., linked; también llamada enlazada) es evitar los movimientos de elementos del vector cada vez que se borra o inserta. Por lo tanto, los elementos consecutivos de la lista ya no ocuparán posiciones consecutivas dentro del vector; es más, la posición que ocupa un elemento dentro del vector deja de tener significado. Para registrar qué elemento va a continuación de uno dado se necesita introducir el concepto de encadenamiento (ing., link): una posición del vector que contiene el elemento v_k de la lista incluye un campo adicional, que registra la posición que ocupa el elemento v_{k+1}. En la fig. 3.16 se muestra una primera versión de la representación del tipo, donde queda claro que el precio a pagar por ahorrar movimientos de elementos es añadir un campo entero en cada posición del vector. Parece una opción recomendable, sobre todo con listas muy inestables, o cuando los elementos de la lista son de dimensión lo bastante grande como para que el espacio de los encadenamientos sea desdeñable (o bien cuando el contexto de uso no imponga requerimientos espaciales de eficiencia). Notemos que la disposición de los elementos dentro del vector ha dejado de ser importante y depende exclusivamente de la historia de la estructura y de la implementación concreta de los algoritmos; es más, para una misma lista hay muchísimas configuraciones del vector que la implementan y esta propiedad debería quedar establecida en la relación de igualdad del tipo. El concepto de representación encadenada es extensible a todas las estructuras lineales estudiadas hasta ahora.

```
tipo lista es
    tupla
        A es vector [de 0 a val-1] de
            tupla
                v es elem; enc es entero
            ftupla
        act, prim son entero
    ftupla
ftipo
```

Fig. 3.16: primera versión de la representación encadenada de las listas con punto de interés.

Así, pues, los elementos de la lista tienen un encadenamiento que los une, y con un apuntador al primero de la lista, otro al elemento actual y una indicación de qué elemento es el último (por ejemplo, dando a su encadenamiento un valor nulo, habitualmente -1) la podemos recorrer, consultar, e insertar y borrar elementos. En la fig. 3.17 se muestran dos configuraciones posibles de las muchas que existen para una lista dada. En el apartado (a), cada posición del vector incluye el valor de su encadenamiento, mientras que en el apartado (b) se muestran estos encadenamientos gráficamente apuntando cada elemento a su sucesor, excepto el último, que tiene un encadenamiento especial; esta notación es más clara y por esto la preferimos a la primera.

Fig. 3.17(a): dos posibles representaciones encadenadas de la lista <ab, cd>.

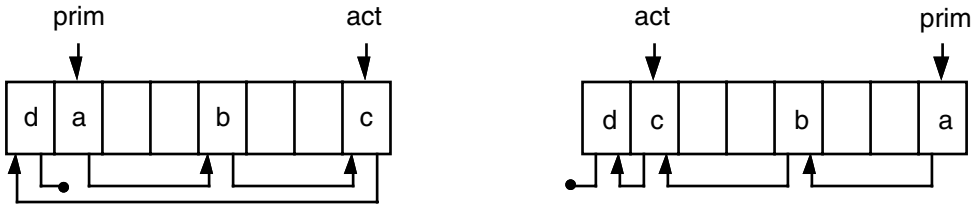


Fig. 3.17(b): ídem, pero mostrando gráficamente los encadenamientos.

Todavía queda por resolver la gestión del espacio libre del vector, ya que cada vez que se inserta un elemento es necesario obtener una posición del vector donde almacenarlo y, al borrarlo, es necesario recuperar esta posición para reutilizarla en el futuro. Una primera solución consiste en marcar las posiciones del vector como ocupadas o libres, ocupar espacio del final del vector hasta que se agote, y reorganizar entonces el vector, pero, precisamente, el problema de este esquema son las reorganizaciones. Como alternativa, los sitios libres se pueden organizar también como una estructura lineal con operaciones para obtener un elemento, borrar e insertar. En este caso, y dado que no importa la posición elegida, el espacio libre se organiza de la forma más sencilla posible, es decir, como una pila, llamada pila de sitios libres, que compartirá el vector con la lista, aprovechando el mismo campo de encadenamiento para enlazar los elementos y así no desperdiciar espacio (v. fig. 3.18).

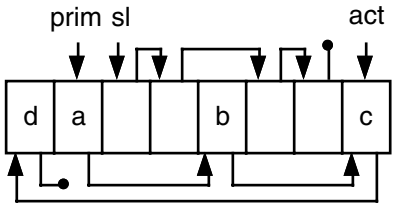


Fig. 3.18: ídem fig. 3.17(b), izquierda, con gestión del espacio libre.

La representación resultante presenta todavía un problema más: al insertar y borrar elementos es necesario modificar el encadenamiento del elemento anterior al distinguido. Actualmente, la única manera de encontrar el elemento anterior al actual es recorrer la lista desde el inicio hasta llegar al elemento en cuestión, por lo que la inserción y la supresión tendrían un coste temporal lineal. La solución obvia consiste en no tener un apuntador al elemento actual, sino al anterior al actual y, así, tanto la inserción como la supresión tendrán un coste constante. De todos modos, esta modificación introduce un nuevo problema: ¿cuál es el elemento anterior al primero? Este caso especial obliga a escribir unos algoritmos más complicados, que actúan dependiendo de si la lista está o no vacía, de si el elemento a suprimir es o no es el primero, etc. Para evitar estos problemas, existe una técnica sencilla pero de gran utilidad, que consiste en la inclusión dentro de la lista de un elemento fantasma o centinela (ing., sentinel), es decir, un elemento ficticio que siempre está a la izquierda de la lista, desde que se crea, de manera que el elemento actual siempre tiene un predecesor[5]. Dentro del vector, este elemento ocupará la posición 0, y como los algoritmos no lo moverán nunca de sitio, no es necesario un apuntador al primer elemento de la lista.

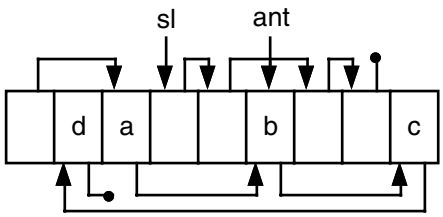


Fig. 3.19: ídem fig. 3.18, con un elemento fantasma y apuntador al anterior.

En la fig. 3.20 se da un universo de implementación de las listas encadenadas con elemento fantasma y apuntador al anterior del actual, que deja todas las operaciones constantes excepto crea. En la fig. 3.21 se muestra la mecánica de inserción y supresión de elementos en listas encadenadas (para que los esquemas sean más claros, no se dibujan los vectores ni el elemento fantasma, sino directamente la secuencia de elementos y la pila de sitios libres).

[5] Es importante distinguir el modelo y la implementación de las listas, ya que la técnica del fantasma es totalmente irrelevante para la especificación del tipo y, en consecuencia, para sus usuarios.

Notemos que el invariante de la representación es más complejo que los que hemos visto hasta ahora, porque es necesario establecer ciertas propiedades que cumplan los encadenamientos. En concreto, en la primera línea se dan los valores permitidos para ant y sl, y en la segunda y tercera líneas se establece que toda posición del vector forma parte, o bien sólo de la lista de elementos, o bien sólo de la pila de sitios libres. Para escribir cómodamente este predicado se utiliza una función auxiliar definida recursivamente, cadena, que devuelve el conjunto de posiciones que cuelga de una posición determinada del vector siguiendo los encadenamientos; así, cadena(l, 0) devuelve el conjunto de posiciones ocupadas dentro de l, y cadena(l, l.sl) el conjunto de posiciones que forman la pila de sitios libres dentro de l. La mayoría de estas propiedades del invariante se repetirán en todas las representaciones encadenadas que estudiaremos al lo largo del texto, como también la función cadena, en esta forma o con pequeñas variaciones.

3.3.3 Implementación de estructuras de datos con punteros

Las diferentes implementaciones vistas hasta ahora se basan en el uso de vectores para almacenar los elementos de las secuencias. Esta política presenta algunos problemas:

- En un momento dado, se desaprovechan todas las posiciones del vector que no contienen ningún elemento. Este hecho es especialmente grave cuando hay varias estructuras lineales y alguna de ellas se llena mientras que las otras todavía tienen espacio disponible, inalcanzables para la estructura llena (v. ejercicio 3.20).
- Hay que predeterminar la dimensión máxima de la estructura de datos, aunque no se disponga de suficiente información para escoger una cota fiable, o se trate de un valor muy fluctuante durante la existencia de la estructura. Es más, a diferencia de la notación empleada en este texto, muchos de los lenguajes de programación imperativos existentes en el mercado no permiten parametrizar la dimensión del vector, lo cual agrava considerablemente el problema y puede obligar a definir diversos módulos para estructuras idénticas que sólo difieran en esta característica.
- Los algoritmos han de ocuparse de la gestión del espacio libre del vector.

Los lenguajes imperativos comerciales presentan un mecanismo incorporado de gestión de espacio para liberar al programador de estas limitaciones, que se concreta en la existencia de un nuevo tipo de datos (en realidad, un constructor de tipo) llamado **puntero** (ing., pointer): una variable de tipo puntero a T, donde T es un tipo de datos cualquiera, representa un apuntador a un objeto de tipo T. En términos de implementación, el espacio libre es gestionado por el sistema operativo, que responde a las peticiones del programa, y un puntero no es más que una dirección de memoria que representa la posición de inicio del objeto de tipo T, con el añadido de un mecanismo de control de tipo para impedir el uso indebido de los objetos. El espacio así gestionado se denomina **memoria dinámica** (ing., dynamic storage) del programa.

```
universo LISTA_INTERÉS_ENC(ELEM, VAL_NAT) es
    implementa LISTA_INTERÉS(ELEM, VAL_NAT)
    tipo lista es tupla
                    A es vector [de 0 a val] de
                            tupla v es elem; enc es entero ftupla
                    ant, sl son entero
                ftupla
    ftipo
    invariante (l es lista): (1 ≤ l.sl ≤ val  ∨  l.sl = -1)  ∧  l.ant∈ cadena(l, 0) - {-1}  ∧
                             cadena(l, 0)  ∩  cadena(l, l.sl) = {-1} ∧
                             cadena(l, 0)  ∪  cadena(l, l.sl) = [-1, val]
        donde se define cadena: lista nat → P(nat) como:
            cadena(l, -1) = {-1}
            n ≠ -1 ⇒ cadena(l, n) = {n} ∪ cadena(l, l.A[n].enc)

        {En todas las operaciones, recordar que el punto de interés
         es l.A[l.ant].enc  y que el fantasma reside en la posición 0}

    función crea devuelve lista es
    var l es lista; i es entero fvar
        l.ant := 0; l.A[0].enc := -1  {se inserta el elemento fantasma}
            {se forma la pila de sitios libres}
        para todo i desde 1 hasta val-1 hacer l.A[i].enc := i+1 fpara todo
        l.A[val].enc := -1; l.sl := 1
    devuelve l

    función inserta (l es lista; v es elem) devuelve lista es
    var temp es entero fvar
        si l.sl = -1 entonces error  {lista llena}
        si no  {v. fig. 3.21, abajo, derecha}
            temp := l.A[l.sl].enc; l.A[l.sl] := <v, l.A[l.ant].enc>
            l.A[l.ant].enc := l.sl; l.ant := l.sl; l.sl := temp
        fsi
    devuelve l

    función borra (l es lista) devuelve lista es
        si l.A[l.ant].enc = -1 entonces error {lista vacía o final de lista}
        si no  {v. fig. 3.21, abajo, izquierda}
            temp := l.A[l.ant].enc; l.A[l.ant].enc := l.A[temp].enc
            l.A[temp].enc := l.sl; l.sl := temp
        fsi
    devuelve l
```

Fig. 3.20: implementación encadenada de las listas.

```
función principio (l es lista) devuelve lista es
    l.ant := 0
devuelve l

función actual (l es lista) devuelve elem es
var v es elem fvar
    si l.A[l.ant].enc = -1 entonces error
    si no v := l.A[l.A[l.ant].enc].v
    fsi
devuelve v

función vacía? (l es lista) devuelve bool es
devuelve l.A[0].enc = -1

función avanza (l es lista) devuelve lista es
    si l.A[l.ant].enc = -1 entonces error
    si no l.ant := l.A[l.ant].enc
    fsi
devuelve l

función final? (l es lista) devuelve bool es
devuelve l.A[l.ant].enc = -1

función llena? (l es lista) devuelve bool es
devuelve l.sl = -1

funiverso
```

Fig. 3.20: implementación encadenada de las listas (cont.).

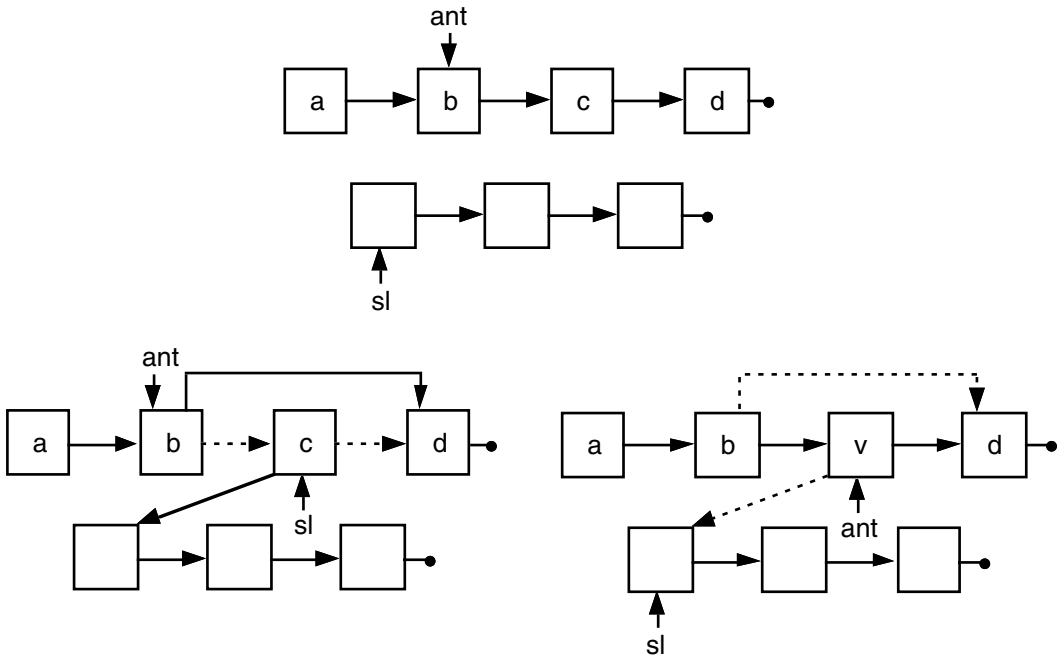


Fig. 3.21: representación encadenada de la lista <ab, cd> (arriba), supresión del elemento distingido (abajo, a la izquierda) e inserción del elemento v (abajo, a la derecha).

Usando punteros, se puede obtener espacio para guardar objetos de un tipo determinado y posteriormente devolver este espacio cuando el programa ya no necesita más el objeto; estas operaciones se traducen en la existencia de dos primitivas sobre el tipo:

- obtener_espacio: función tal que, aplicada sobre una variable de tipo puntero a T, obtiene el espacio necesario para guardar un objeto de tipo T, y devuelve como resultado un apuntador que es el camino de acceso a ese objeto.
- liberar_espacio: acción tal que, aplicada sobre una variable de tipo puntero a T, "destruye" el objeto apuntado por esta variable, de manera que se vuelve inaccesible desde el programa. La invocación de esta acción es el único mecanismo existente para destruir un objeto creado mediante obtener_espacio.

Desde el momento en que se ha obtenido espacio para un objeto hasta que se libera, este objeto se refiere mediante su puntero asociado, escribiendo el símbolo '^' como sufijo del nombre del puntero. Cuando una variable de tipo puntero no apunta a ningún objeto (antes de asociarle un objeto, después de desasociarle uno, o cuando lo forzamos explícitamente mediante una asignación), tiene un valor especial que llamamos NULO[6]. Este valor también es devuelto por obtener_espacio cuando no queda memoria dinámica disponible[7]; aplicar liberar_espacio sobre un puntero igual a NULO provoca un error. También es conveniente destacar que los objetos creados con obtener_espacio no se destruyen automáticamente al finalizar la ejecución del procedimiento donde se han creado (con la única excepción del bloque correspondiente al programa principal); ésta es una diferencia fundamental con los objetos "estáticos" habituales. En la fig. 3.22 se ofrece un ejemplo de uso de los punteros.

En la fig. 3.23 se desarrolla una implementación para las listas mediante punteros. De acuerdo con el esquema dado, el tipo auxiliar de los elementos de la cadena se define recursivamente. Fijémonos que el cambio de vector por punteros no afecta a los algoritmos de las funciones sino únicamente a la notación. El invariante es similar al caso de los vectores, adaptando la notación y sabiendo que el sitio libre no es referible desde la representación. Por último, notemos que no hay ningún parámetro formal que defina la dimensión máxima de la estructura; ahora bien, en la inserción es necesario controlar que no se agote el espacio de la memoria. Precisamente este hecho provoca que las especificaciones de TAD estudiadas hasta ahora, infinitas o con cota conocida, no sean válidas para la idea de implementaciones por punteros; es necesaria una especificación que, de alguna manera, incorpore la noción de memoria dinámica, que se puede agotar en cualquier momento, lo cual no es trivial y no se estudia aquí.

Por lo que se refiere al cálculo de la eficiencia temporal, consideramos que un apuntador ocupa un espacio $\Theta(1)$. Es necesario tener siempre presente, no obstante, que si el apuntador tiene un valor no nulo, habrá un objeto apuntado, que tendrá su propia dimensión. En caso de calcular el espacio real de una estructura de datos, la dimensión exacta de un apuntador será un dato necesario. Por otro lado, la indirección propia del operador '^' se considera también constante en cuanto a la eficiencia temporal.

[6] Algunos lenguajes no asignan forzosamente NULO como valor inicial de una variable puntero, lo que puede dar lugar a alguna dificultad adicional en la gestión del tipo.

[7] Igualmente, no todos los lenguajes siguen esta norma.

```
tipo T es
    tupla
        c es nat
        ...
    ftupla
ftipo
```

```
tipo pT es ^T ftipo        {declaración de un tipo de apuntadores a objetos de tipo T }
var p, q son pT fvar       {declaración de variables para apuntar a objetos de tipo T;
                            inicialmente, p = q = NULO}
p := obtener_espacio       {p apunta a un objeto de tipo T (o vale NULO, si no había
                            suficiente memoria)}
p^.c := 5                  {asignación de un valor a un componente de la tupla apuntada
                            por p}
q := obtener_espacio
si p = q entonces ...      {comparación de punteros (no de los objetos asociados), da
                            cierto si p y q apuntan al mismo objeto, lo que será falso
                            excepto por una asignación explícita p := q; también puede
                            comprobarse p ≠ q, pero lo que no tiene ningún sentido es
                            hacer una comparación como p < q}
q^.c := p^.c               {notemos que las referencias a objetos pueden aparecer tanto
                            en la parte derecha como en la parte izquierda de una
                            asignación}
liberar_espacio(p)         {el objeto p^ se vuelve inaccesible; a partir de aquí y hasta que no
                            se vuelva a obtener espacio usando este puntero, p = NULO y
                            cualquier referencia a p^ provoca error}
```

Fig. 3.22: ejemplo de uso del mecanismo de punteros.

La codificación con punteros del tipo lista muestra las diversas ventajas de la representación por punteros sobre los vectores: no es necesario predeterminar un número máximo de elementos en la estructura, no es necesario gestionar el espacio libre y, en cada momento, el espacio ocupado por la estructura es estrictamente el necesario exceptuando el espacio requerido por los encadenamientos[8]. De todas formas, el uso de punteros también presenta algunas desventajas:

- No es verdad que la memoria sea infinita: en cualquier momento durante la ejecución del programa se puede agotar el espacio, con el agravante de no conocer a priori la capacidad máxima de la estructura.

[8] Posiblemente un campo de encadenamiento con una implementación mediante punteros ocupe más bits que un campo de encadenamiento con una implementación mediante vectores, pero la diferencia acostumbra a ser irrelevante respecto a la dimensión total de los elementos.

```
universo LISTA_INTERÉS_ENC_PUNT(ELEM) implementa LISTA_INTERÉS(ELEM)
    tipo lista es tupla prim, ant son ^nodo ftupla ftipo
    tipo privado nodo es tupla v es elem; enc es ^nodo ftupla ftipo
    invariante (l es lista): l.prim ≠ NULO ∧ NULO∈ cadena(l.prim) ∧ l.ant∈ cadena(l.prim)-{NULO}
        donde se define cadena: ^nodo → P(^nodo) como:
            cadena(NULO) = {NULO}
            p ≠ NULO ⇒ cadena(p) = {p} ∪ cadena(p^.enc)
    función crea devuelve lista es
    var l es lista fvar
        l.prim := obtener_espacio  {para el elemento fantasma}
        si l.prim = NULO entonces error  {no hay espacio libre}
        si no l.ant := l.prim; l.ant^.enc := NULO
        fsi
    devuelve l
    función inserta (l es lista; v es elem) devuelve lista es
    var p es ^nodo fvar
        p := obtener_espacio
        si p = NULO entonces error  {no hay espacio libre}
        si no p^.v := v; p^.enc := l.ant^.enc; l.ant^.enc := p; l.ant := p
        fsi
    devuelve l
    función borra (l es lista) devuelve lista es
    var temp es ^nodo fvar
        si l.ant^.enc = NULO entonces error {lista vacía o final de lista}
        si no temp := l.ant^.enc; l.ant^.enc := temp^.enc; liberar_espacio(temp)
        fsi
    devuelve l
    función principio (l es lista) devuelve lista es
    devuelve <l.prim, l.prim>
    función actual (l es lista) dev elem es
        si l.ant^.enc = NULO entonces error
        si no v := l.ant^.enc^.v
        fsi
    devuelve v
    función avanza (l es lista) devuelve lista es
        si l.ant^.enc = NULO entonces error
        si no l.ant := l.ant^.enc
        fsi
    devuelve l
    función final? (l es lista) devuelve bool es
    devuelve l.ant^.enc = NULO
    función vacía? (l es lista) devuelve bool es
    devuelve l.prim^.enc = NULO
funiverso
```

Fig. 3.23: implementación encadenada por punteros de las listas.

- Los punteros son referencias directas a memoria, por lo que es posible que se modifiquen insospechadamente datos (e incluso código, si el sistema operativo no lo controla) en otros puntos del programa a causa de errores algorítmicos.
- La depuración del programa es más difícil, porque es necesario estudiar la ocupación de la memoria.
- Que el sistema operativo gestione los sitios libres es cómodo, pero a veces ineficiente (v. ejercicio 3.17).
- Algunos esquemas típicos de los lenguajes imperativos (por ejemplo, el mecanismo de parámetros, la asignación y la entrada/salida) no funcionan de la manera esperada: dados los punteros p y q a objetos de tipo T, dado el fichero f de registros de tipo T y dada la función llamada con un parámetro de entrada de tipo T, destacamos:
  - ◊ escribe(f, p) no escribe el objeto designado por p, sino el valor del puntero, que es una dirección de memoria, y es necesario invocar escribe(f, p^). Ahora bien, esta llamada también grabará en el fichero aquellos campos de T que sean punteros (si los hay), que serán direcciones de memoria particulares de la ejecución actual del programa. Si en otra ejecución posterior se leen los datos del fichero, los punteros obtenidos no tendrán ningún valor y se habrá perdido la estructuración lógica de los datos.
  - ◊ p := q no asigna el valor del objeto apuntado por q al objeto apuntado por p, sino que asigna la dirección q a p.
  - ◊ llamada(p) protege el valor del puntero p, pero no el valor del objeto apuntado por p, que puede modificarse por mucho que el parámetro se haya declarado sólo de entrada (por valor, en la terminología habitual de estos lenguajes).
- Diversos punteros pueden designar un mismo objeto, en cuyo caso, la modificación del objeto usando un puntero determinado tiene como efecto lateral la modificación del objeto apuntado por el resto de punteros.
- Una particularización del problema anterior son las llamadas referencias colgadas (ing., dangling reference). Por ejemplo, dado el trozo de código:

```
var p, q son ^nodo fvar
p := obtener_espacio; q := p
liberar_espacio(p)
```

  se dice que q queda "colgado", en el sentido de que su valor es diferente de NULO, pero no apunta a ningún objeto válido. Una referencia a q^ tiene efectos impredecibles.
- El problema simétrico al anterior es la creación de basura (ing., garbage); dado:

```
var p, q son ^nodo fvar
p := obtener_espacio; q := obtener_espacio; p := q
```

  observemos que el objeto inicialmente asociado a p queda inaccesible después de la segunda asignación, a pesar de que existe porque no ha sido explícitamente

destruido. Eventualmente, esta basura puede provocar errores de ejecución o, como mínimo, dificultar la depuración del programa. Muchos sistemas operativos tienen incorporado un proceso de recuperación de estos objetos inaccesibles llamado recogida de basura (ing., garbage collection), cuya programación ha dado lugar a diversos algoritmos clásicos dentro del ámbito de las estructuras de datos (v., por ejemplo, [AHU83, cap. 12]).

Algunos de estos problemas no son exclusivos de este mecanismo sino del mal uso de las estructuras encadenadas, pero es al trabajar con punteros cuando surgen con frecuencia.

3.3.4 Transparencia de la representación usando punteros

El uso indiscriminado del mecanismo de punteros puede presentar los problemas que se acaban de comentar, los cuales tienen algunas consecuencias perniciosas sobre el método de desarrollo de programas que se sigue en este texto, sobre todo la pérdida de la transparencia de la representación en diversos contextos. En este apartado se detalla cuáles son exactamente los problemas y cómo se pueden solucionar, y se dan algunas convenciones para no complicar con detalles irrelevantes la codificación de los tipos y de los algoritmos en el resto del libro.

a) El problema de la asignación

Consiste en el comportamiento diferente de la asignación entre estructuras según se usen vectores o punteros para implementarlas. Así, dadas dos variables p y q, declaradas de tipo lista, la asignación p := q da como resultado: a) dos objetos diferentes que tienen el mismo valor, uno identificado por p y el otro por q, en caso de que las listas estén implementadas con vectores; b) un único objeto identificado al mismo tiempo por p y q, en caso de que las listas estén implementadas con punteros. Es decir, el funcionamiento de un algoritmo donde aparezca esta asignación depende de la implementación concreta del TAD lista, lo cual es inaceptable.

Para solucionar este problema, todo TAD T ha de ofrecer una operación duplica que, dado un objeto de tipo T, simplemente hace una réplica exacta del mismo. Entonces, un usuario del TAD empleará duplica para copiar un objeto en lugar de la asignación estándar. Eso sí, dentro del TAD que implementa T se puede usar la asignación normal con su comportamiento tradicional, pues la representación es accesible.

b) El problema de la comparación

De igual manera que en el caso anterior, la comparación p = q entre dos objetos de tipo T se comporta de forma diferente según se haya implementado el tipo; es más, en este caso y sea cual sea la representación, el operador = seguramente no implementará la igualdad auténtica del tipo, que será la relación de igualdad vista en la sección 2.2.

Así, todo TAD T ha de ofrecer una operación ig que, dados dos objetos de tipo T, los compare (es decir, compruebe si están en la misma clase de equivalencia según la relación de igualdad del tipo). Igualmente, dentro del TAD que implementa T sí se puede usar la comparación normal con su comportamiento tradicional.

c) El problema de los parámetros de entrada

Dado el TAD T y dada una función que declare un objeto t de tipo T como parámetro de entrada, ya se ha comentado que, si T está implementado con punteros, cualquier cambio en la estructura se refleja en la salida de la función. Ello es debido a que el parámetro de entrada es el puntero mismo, pero no el objeto, que no se puede proteger de ninguna manera. Por ejemplo, en la fig. 3.24 se muestra una función para concatenar dos listas de elementos (dejando el punto de interés a la derecha del todo; la especificación queda como ejercicio para el lector) que se implementa fuera del universo de definición de las listas, por lo que no puede acceder a la representación; pues bien, si las listas están implementadas por punteros, el parámetro real asociado a l_1 se modifica, pese a ser un parámetro de entrada.

```
función concatena (l1, l2 son lista) devuelve lista es
    mientras ¬ final?(l1) hacer l1 := avanza(l1) fmientras
    l2 := principio(l2)
    mientras ¬ final?(l2) hacer
        l1 := inserta(l1, actual(l2)); l2 := avanza(l2)
    fmientras
devuelve l1
```

Fig. 3.24: una función que concatena dos listas.

Una vez más es necesario desligar el funcionamento de un algoritmo de la representación concreta de los tipos de trabajo y, por ese motivo, simplemente se prohíbe la modificación de los parámetros de entrada. En caso de que, por cualquier razón, sea necesario modificar un parámetro de entrada, es obligado hacer previamente una copia con duplica y trabajar sobre la copia. Opcionalmente y por motivos de eficiencia, las operaciones que hasta ahora se están implementando mediante funciones se pueden pasar a codificar mediante acciones; de esta manera, se pueden escoger uno o más objetos como parámetros de entrada y de salida de forma que el resultado de la operación quede almacenado en ellos, y ahorrar así las copias antes comentadas. La toma de una decisión en este sentido debe documentarse exhaustivamente en el código de la función. Todos estos hechos relativos a la eficiencia deben tratarse cuidadosamente sobre todo al considerar la traducción del código escrito en Merlí a un lenguaje imperativo concreto.

En la fig. 3.25 se presenta la adaptación de la función concatena a estas normas. Notemos que la función pasa a ser una acción donde se declaran dos parámetros: el primero, de

entrada y de salida, almacena el resultado, y el segundo, sólo de entrada, que se duplica al empezar la acción porque se modifica (aunque en este ejemplo concreto no era necesario, dado que los elementos de l_2 no cambian, excepto el punto de interés, que sí habrá sido duplicado por el mecanismo de llamada a procedimientos propio de los lenguajes de programación imperativos). Opcionalmente, se podría haber respetado la estructura de función, y habría sido necesario declarar una lista auxiliar para generar el resultado.

```
acción concatena (ent/sal l1 es lista; ent l2 es lista) es
var laux es lista fvar
    duplica(l2, laux); laux := principio(laux)
    mientras ¬ final?(l1) hacer l1 := avanza(l1) fmientras
    mientras ¬ final?(laux) hacer
        l1 := inserta(l1, actual(laux)); laux := avanza(laux)
    fmientras
facción
```

Fig. 3.25: la función de la fig. 3.24 convertida en acción.

d) El problema de las variables auxiliares

Sea una variable v de tipo T declarada dentro de una función f que usa T; si al final de la ejecución de f la variable v contiene información y T está implementado por punteros, esta información queda como basura. Por ejemplo, en la fig. 3.25 la lista auxiliar laux contiene todos los elementos de l_2 al acabar la ejecución de concatena. La existencia de esta basura es, como mínimo, estilísticamente inaceptable y puede causar problemas de ejecución.

Para evitar este mal funcionamiento, el TAD T ha de ofrecer una función destruye que libere el espacio ocupado por una variable de tipo T, y es necesario invocar esta operación antes de salir de la función sobre todos los objetos que contengan basura (v. fig. 3.26).

```
acción concatena (ent/sal l1, l2 son lista) es
var laux es lista fvar
    duplica(l2, laux); laux := principio(laux)
    mientras ¬ final?(l1) hacer l1 := avanza(l1) fmientras
    mientras ¬ final?(laux) hacer
        l1 := inserta(l1, actual(laux)); laux := avanza(laux)
    fmientras
    destruye(laux)
facción
```

Fig. 3.26: la acción de la fig. 3.25 liberando espacio.

e) El problema de la reinicialización

Al empezar a trabajar con un objeto v de tipo T, si el TAD está implementado con punteros se irá reservando y liberando espacio según convenga. Eventualmente, el valor actual de v puede quedar obsoleto y puede ser necesario reinicializar la estructura, situación en la que, si se aplica descuidadamente la típica operación de creación, el espacio ocupado por v previamente a la nueva creación pasa a ser inaccesible; es necesario, pues, invocar antes destruye sobre v. En general, es una buena práctica llamar a destruye en el mismo momento en que el valor de una variable deje de ser significativo.

Un caso particular de esta situación son los parámetros sólo de salida o de entrada y de salida. En el caso de parámetros de entrada y de salida es básico que el programador de la acción sea consciente de que el parámetro real puede tener memoria ocupada al comenzar la ejecución del procedimiento mientras que, en el caso de parámetros sólo de salida, el posible espacio ocupado por el parámetro real tiene que ser liberado antes de invocar la acción para no perder acceso. En los lenguajes que no distinguen entre parámetros de entrada y de salida y sólo de salida, se puede dar a los segundos el mismo tratamiento que a los primeros; ahora bien, si el lenguaje no inicializa siempre los punteros a NULO puede haber ambigüedades, porque no se sabe si un puntero no nulo está inicializado o no, en cuyo caso se pueden tratar todos como si fueran parámetros sólo de salida.

Resumiendo, en el resto del texto convendremos que todo tipo abstracto presenta, no sólo las operaciones que específicamente aparecen en la signatura, sino también las introducidas en este apartado (orientadas a la transparencia total de la información), y que las funciones y los algoritmos que utilizan los tipos siguen las normas aquí expuestas para asegurar el funcionamento correcto en todos los contextos conflictivos citados en este apartado. Como ejemplo ilustrativo, en la figura 3.27 se presentan las dos representaciones encadenadas de las listas con punto de interés, una con vectores y la otra con punteros, que siguen todas las convenciones dadas. Las operaciones de inserción y borrado se ven modificadas por este esquema; en la inserción, debe duplicarse el elemento almacenado en el vector mediante duplica, mientras que en la segunda debe destruirse el espacio ocupado usando destruye; al haber acordado que todo TAD presenta estas operaciones, no es necesario requerirlas como parámetros formales. El algoritmo de duplica asegura no sólo que los elementos son los mismos sino además que su posición se conserva, así como la situación del punto de interés. La operación de comparación también usa la igualdad de elementos, definida en cada una de las implementaciones del tipo; notemos que los elementos fantasma de las listas involucradas no son comparados y que la posición ocupada por el punto de interés es significativa de cara al resultado. La destrucción de la lista obliga a destruir el espacio individual que ocupa cada elemento y, además, en el caso de implementación por punteros, el espacio de cada celda de la lista. Finalmente, destacamos que presentamos dos posibles opciones para crear las listas: con vectores, el usuario es el responsable de destruir la estructura antes de recrearla, si lo considera necesario; con punteros, la destrucción se realiza al comenzar la creación, incondicionalmente, de forma transparente al usuario del tipo.

```
universo LISTA_INTERÉS_ENC(ELEM, VAL_NAT) es
    implementa LISTA_INTERÉS(ELEM, VAL_NAT)
    ... la representación del tipo no cambia

    acción crea (sal l es lista) es
    var i es entero fvar
        l.ant := 0; l.A[0].enc := -1
        para todo i desde 1 hasta val-1 hacer l.A[i].enc := i+1 fpara todo
        l.A[val].enc := -1; l.sl := 1
    facción

    acción inserta (ent/sal l es lista; ent v es elem) es
    var temp es entero fvar
        si l.sl = -1 entonces error
        si no temp := l.A[l.sl].enc; ELEM.duplica(v, l.A[l.sl].v); l.A[l.sl].enc := l.A[l.ant].enc
         l.A[l.ant].enc := l.sl; l.ant := l.sl; l.sl := temp
        fsi
    facción

    acciones borra, principio y avanza: siguen los algoritmos de la fig. 3.20, pero
      adaptando las funciones a acciones de manera similar a inserta

    funciones actual y final? : idénticas a la fig. 3.20

    función ig (l1, l2 son lista) devuelve bool es
    var i1, i2 son enteros; iguales es booleano fvar
        i1 := l1.A[0].enc; i2 := l2.A[0].enc; iguales := cierto
        mientras (i1 ≠ -1) ∧ (i2 ≠ -1) ∧ iguales hacer
          si ELEM.ig(l1.A[i1].v, l2.A[i2].v) ∧ ((i1 = l1.ant ∧ i2 = l2.ant) ∨ (i1 ≠ l1.ant ∧ i2 ≠ l2.ant))
            entonces i1 := l1.A[i1].enc; i2 := l2.A[i2].enc
            si no iguales := falso
          fsi
        fmientras
    devuelve (i1 = -1) ∧ (i2 = -1)

    acción duplica (ent l1 es lista; sal l2 es lista) es
        crea(l2)
            { se copian los elementos a partir del punto de interés }
        i := l1.A[l1.ant].enc
        mientras i ≠ -1 hacer inserta(l2, l1.A[i].v); i := l1.A[i].enc fmientras
            { a continuación, se copian los elementos anteriores al punto de interés }
        principio(l2); i := l1.A[0].enc
        mientras i ≠ l1.A[l1.ant].enc hacer inserta(l2, l1.A[i].v); i := l1.A[i].enc fmientras
    facción
```

Fig. 3.27(a): implementación encadenada con vectores de las listas con punto de interés.

```
    acción destruye (ent/sal l es lista) es
            i := l.A[0].enc
            mientras i ≠ -1 hacer
                    ELEM.destruye(l.A[i].v); i := l.A[i].enc
            fmientras
    facción
funiverso
```

Fig. 3.27(a): implementación encadenada con vectores de las listas con punto de interés (cont.).

```
universo LISTA_INTERÉS_ENC_PUNT(ELEM) implementa LISTA_INTERÉS(ELEM)
    ... la representación del tipo no cambia
    acción crea (sal l es lista) es
        {obtiene espacio para el fantasma, liberando previamente si es necesario, en
         cuyo caso se liberan todas las celdas menos una, precisamente para el fantasma}
        si l.prim ≠ NULO entonces libera(l.prim^.enc) si no l.prim := obtener_espacio fsi
        si l.prim = NULO entonces error si no l.prim^.enc := NULO; l.ant := l.prim fsi
    devuelve l
    Acciones inserta, borra, principio y avanza: siguen los algoritmos de la fig. 3.23,
    pero adaptando las funciones a acciones de manera similar a la función inserta de
    la fig. 3.27(a). Las funciones actual y final? son idénticas a la fig. 3.23
    Función ig y acción duplica: idénticas a la operacions ig y duplica de la fig. 3.27(a),
    adaptando la notación de vectores a punteros
    acción destruye (ent/sal l es lista) es libera(l.prim) facción

    {Acción libera(p): devuelve al sistema las celdas que cuelgan a partir del puntero p
    siguiendo el campo de encadenamiento (v. ejercicio 3.17 para una variante)}
    acción privada libera (ent/sal p es ^nodo) es
    var q es ^nodo fvar
        mientras p ≠ NULO hacer
                ELEM.destruye(p^.v)     { se libera el espacio del elemento }
                q := p; p := p^.enc
                liberar_espacio(q)      { se libera el espacio de la celda de la lista }
        fmientras
    facción
funiverso
```

Fig. 3.27(b): implementación con punteros de las listas con punto de interés (cont.).

Para evitar una sobrecarga del código de los TAD con estas tareas mecánicas, en el resto del texto se toman una serie de convenciones que las llevan a término implícitamente[9]. En concreto, dado un tipo abstracto T [10] se cumple:

- Cualquier implementación de T presenta las operaciones duplica, ig y destruye, aunque no aparezcan explícitamente. Como su esquema es siempre muy similar al que aparece en la fig. 3.27, en este texto no se insistirá más, y supondremos que los programadores del TAD las escribirán correctamente en el momento de programar el tipo en un lenguaje de programación concreto. En lo que se refiere a ig, destacamos que ha de ser la implementación de la relación de igualdad del tipo.
- Siendo u y v de tipo T, toda asignación u := v hecha fuera de un universo de implementación de T se interpreta como una llamada a duplica(v, u). Por otro lado, toda comparación u = v hecha fuera de un universo de implementación de T se interpreta como una llamada a ig(u, v).
- Siendo u un parámetro de entrada de tipo T de una función o acción f cualquiera, su valor después de ejecutar f no varía, aunque cambie dentro de la función. Se puede suponer que f está manipulando una copia de u obtenida mediante duplica.
- Antes de salir de una función o acción, se llamará implícitamente al procedimiento destruye sobre todos aquellos objetos declarados como variables locales.
- Tota reinicialización de un objeto v de tipo T va implícitamente precedida de una llamada a destruye(v).

Con estas convenciones, los algoritmos y las implementaciones de tipo de datos vistos hasta ahora responden al principio de la transparencia de la representación, como es deseable.

Evidentemente, al calcular la eficiencia de los programas es necesario tener presentes estas convenciones; por ejemplo, dado un parámetro de entrada es necesario comprobar si se modifica o no su valor, y conocer también su contexto de uso, para determinar el coste de la invocación a la función correspondiente; también deben ser consideradas las destrucciones de las estructuras al salir de las funciones.

3.3.5 Algunas variantes en la implementación de listas

Para algunos requerimientos particulares, las implementaciones vistas hasta ahora pueden presentar ineficiencias, sobre todo con respecto a su coste temporal. Introducimos tres variantes que permiten resolver algunas situaciones habituales, usadas frecuentemente en el diseño de estructuras de datos complejas que están formadas por diversas subestructuras fuertemente interrelacionadas, generalmente a través de apuntadores (v. cap. 7).

[9] Obviamente, sería mejor disponer de un lenguaje que solucionara estos problemas en su definición, pero como éste no es el caso habitual, se ha optado por dar aquí unas convenciones fácilmente aplicables a los lenguajes imperativos más comunes.

[10] Exceptuando los tipos predefinidos del lenguaje (booleanos, naturales, etc.).

a) Listas circulares

Ocasionalmente puede que no interese la noción clásica de lista, en la que existen dos elementos claramente distinguidos, el primero y el último, sino que simplemente se quieran encadenar los elementos, o incluso que el papel de primer elemento vaya cambiando dinámicamente. Para implementar esta idea, basta con encadenar el último elemento de la lista con el primero; este tipo de lista se denomina lista circular (ing., circular list).

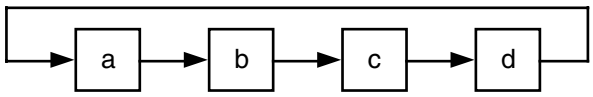


Fig. 3.28: implementación circular de una lista.

El concepto de circularidad también es útil en otras situaciones:

- Algunos algoritmos pueden quedar simplificados si no se distingue el último elemento.
- Desde todo elemento se puede acceder a cualquier otro, lo cual es especialmente útil si hay un elemento de la cadena con información diferenciada.
- Permite que la representación de colas con encadenamientos sólo necesite un apuntador al último elemento, porque el primero siempre será el apuntado por el último.

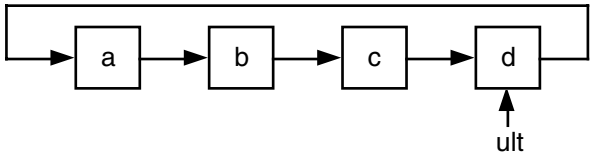


Fig. 3.29: representación encadenada circular de la cola abcd, donde a es la cabeza.

b) Listas doblemente encadenadas

Las listas encadenadas en un único sentido presentan un inconveniente ya conocido: dado un elemento, la única manera de saber cuál es el anterior consiste en recorrer la lista desde el inicio y el coste resultante será lineal. Hay dos situaciones donde es imprescindible saber qué elemento es el anterior a uno dado:

- Cuando la lista ha de ofrecer recorridos tanto hacia adelante como hacia atrás. Notemos que este modelo exige tener más operaciones en la signatura simétricas a las de recorrido vistas hasta ahora: una para situarse al final de la lista, otra para retroceder un elemento y una última para saber si ya estamos al principio de la lista.
- Cuando la lista forma parte de una estructura de datos compleja y sus elementos se

pueden suprimir mediante apuntadores desde otras partes de la estructura, sin conocer la posición que ocupa su predecesor. Este caso es especialmente importante porque aparece con relativa frecuencia.

Si el coste lineal de estas operaciones es inaceptable, es necesario modificar la representación para obtener listas doblemente encadenadas (ing., doubly-linked list): listas tales que todo elemento tiene dos encadenamientos, uno al nodo anterior y otro al siguiente. Así, las operaciones vuelven a ser constantes, a costa de utilizar más espacio.

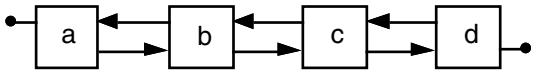
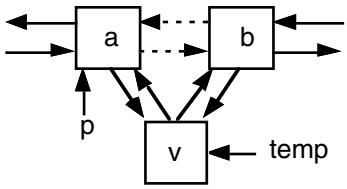
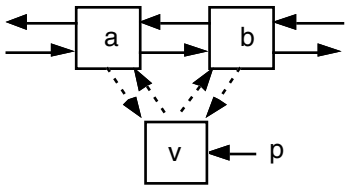


Fig. 3.30: implementación de una lista usando dobles encadenamientos.

Una vez más, para simplificar los algoritmos de inserción y de supresión (v. fig. 3.31) se añade al principio un elemento fantasma, de modo que la lista no esté nunca vacía; además, es aconsejable implementar la lista circularmente. La codificación del universo queda como ejercicio para el lector.




```
temp := obtener_espacio
temp^.enc_dr := p^.enc_dr; temp^.enc_izq := p
p^.enc_dr^.enc_izq := temp; p^.enc_dr := temp
temp^.v := v
```




```
p^.enc_izq^.enc_dr := p^.enc_dr
p^.enc_dr^.enc_izq := p^.enc_izq
liberar_espacio(p)
```

Fig. 3.31: inserción (izquierda) y supresión (derecha) en una lista doblemente encadenada.

c) Listas ordenadas

Un requerimiento que surge a menudo en el procesamiento de una lista es la ordenación de sus elementos según el valor de uno de sus campos, llamado clave (ing., key). Además, esta clave es el medio de acceso individual a los elementos a suprimir y consultar, en vez del punto de interés. Es evidente que, para que el recorrido ordenado no sea excesivamente lento, es necesario que los elementos de la lista estén previamente ordenados según su clave. Hay dos posibilidades:

- Se puede tener la lista desordenada mientras se insertan y se borran elementos y ordenarla justo antes de empezar el recorrido. La ventaja de este esquema es que las inserciones quedan $\Theta(1)$ (se inserta en cualquier lugar -siempre que se permitan repetidos-, porque después ya se ordenará), las supresiones de orden lineal y la única operación de recorrido que tiene una eficiencia baja es principio, que ha de ordenar los elementos de la lista (con un buen algoritmo de ordenación, quedaría $\Theta(n\log n)$, donde n es el número de elementos de la estructura).
- Una alternativa sería mantener la lista siempre ordenada; esto implica que, para cada actualización, es necesario buscar secuencialmente la posición correspondiente al elemento y entonces resulta un orden lineal, pero todas las operaciones de recorrido quedan de orden constante.

La elección de una alternativa concreta depende de la relación esperada entre modificaciones y recorridos ordenados de la lista. Por ejemplo, para poder aplicar el primer esquema es aconsejable que la gestión de la lista presente dos fases diferenciadas, una primera de actualizaciones y una segunda de consultas. También es importante saber que, cuando la lista está ordenada y su representación es secuencial, se puede buscar un elemento usando un algoritmo llamado "búsqueda dicotómica", que tiene un coste logarítmico sobre el número de elementos y que se presenta en este libro en el capítulo siguiente.

El concepto de lista ordenada por una clave puede aplicarse en el caso de tener listas ordenadas por diversas claves; es decir, los elementos que forman una lista pueden tener más de un campo de ordenación. Hay dos situaciones comunes:

- Las diversas claves han de permitir recorridos ordenados independientes. En este caso, habrá que modificar la signatura del tipo para saber siempre bajo qué campo de ordenación se quieren aplicar las operaciones de recorrido.
- Las diversas claves tienen un orden de importancia, de manera que los elementos de la lista se ordenan por la clave más importante y aquellos elementos que tienen el mismo valor de esta clave se ordenan respecto al valor de la segunda clave, etc.

En cuanto a la implementación de estas estructuras, en el primer caso es necesario incorporar a los elementos tantos encadenamientos como campos de ordenación haya; además, se necesitan tantos apuntadores a primeros elementos como campos de encadenamiento, porque lo que en realidad tenemos son diversas listas que comparten los elementos. En el segundo caso, la implementación habitual no cambia.

Ejercicios

Nota preliminar: En los ejercicios de este capítulo, por "implementar un tipo" se entiende: dotar al tipo de una representación, escribir su invariante, codificar las operaciones de la signatura basándose en la representación y estudiar la eficiencia espacial de la representación y temporal de las operaciones, mencionando también el posible coste espacial auxiliar que puedan necesitar las operaciones en caso que no sea constante.

3.1 Escribir un algoritmo iterativo que genere las permutaciones con repeticiones de los naturales de 1 a n tomados de n en n, $n \geq 1$. Así, para n = 2 debe generarse 11, 12, 21, 22.

3.2 Una manera original de representar pilas de naturales de 1 a n (propuesta en [Mar86]) consiste en imaginar que la pila es, en realidad, un número (natural) en base n, que en cada momento tiene tantos dígitos como elementos hay en la pila, y que el dígito menos significativo es la cima de la pila y tal que las cifras no van de 0 a n-1 sino de 1 a n (para evitar algunas ambigüedades que podrían aparecer). Implementar el tipo basándose en esta estrategia (incluir la función de abstracción). Repetir la misma idea para implementar las colas.

3.3 a) Especificar en un universo PILA_2(ELEM) un enriquecimiento de las pilas consistente en las operaciones concatenar dos pilas (el elemento del fondo de la segunda pila queda por encima de la cima de la primera pila y el resto de elementos conserva su posición relativa a estos dos), girar los elementos dentro de la pila y obtener el elemento del fondo de la pila. Escribir un universo IMPL_PILA_2(ELEM) que lo implemente usando funciones no recursivas.

b) Dado el universo PILA(ELEM) que encapsula la especificación habitual de las pilas, modificarlo para que incluya todas las operaciones citadas en el apartado a) (no es necesario volverlas a especificar). A continuación, modificar el universo que implementa PILA(ELEM) para que también se implementen estas nuevas operaciones. Comparar este método de trabajo con el usado en el apartado anterior en cuanto a los criterios de corrección, eficiencia, modificabilidad y reusabilidad de los programas.

3.4 Decimos que una frase es capicúa si se lee igual de izquierda a derecha que de derecha a izquierda desdeñando los espacios en blanco que haya. Por ejemplo, la frase: "dábale arroz a la zorra el abad" (de dudosas consecuencias) es capicúa. Especificar la función capicúa? e implementarla usando los tipos pila y cola, determinando además su eficiencia. Suponer que la frase se representa mediante una cola de caracteres.

3.5 Una doble cola (ing., abreviadamente, dequeue) es una cola en la cual los elementos se pueden insertar, eliminar y consultar en ambos extremos. Determinar la signatura de este tipo de datos y especificarlo dentro de un universo parametrizado. Implementar el tipo según una estrategia secuencial con gestión circular del vector.

3.6 El agente 0069 ha inventado un método de codificación de mensajes secretos. Este método se define mediante dos reglas básicas que se aplican una detrás de otra:

- Toda tira de caracteres no vocales se substituye por su imagen especular.
- Dada la transformación Y del mensaje original X de longitud n, obtenida a partir de la aplicación del paso anterior, el mensaje codificado Z se define como sigue:

  Z = Y[1].Y[n].Y[2].Y[n-1]..., donde Y[i] representa el carácter i-ésimo de Y.

Así, por ejemplo, el mensaje "Bond, James Bond" se transforma en "BoJ ,dnameB sodn" después del primer paso y en "BnodJo s, dBneam" después del segundo. Escribir los algoritmos de codificación y descodificación del mensaje usando la especificación de tipos conocidos y determinar su coste. Suponer que los mensajes se representan mediante colas de caracteres.

3.7 Modificar la representación encadenada sobre el vector de las listas con punto de interés para que la operación de crear la lista tenga coste constante sin cambiar el coste de las demás.

3.8 Pensar una estrategia de implementación de las listas con punto de interés que permita moverlo, no sólo hacia delante (con la operación avanza), sino también hacia atrás (con la operación retrocede), sin añadir a los elementos de la lista ningún campo adicional de encadenamiento y manteniendo todas las operaciones en orden constante excepto principio. Codificar las operaciones adaptando la idea tanto a una representación encadenada como a otra no encadenada.

3.9 Implementar los TAD pila, cola y doble cola usando el mecanismo de punteros.

3.10 Implementar los polinomios especificados en el apartado 1.5.1 siguiendo las políticas secuencial y encadenada y sin usar ningún universo de definición de listas (es decir, directamente sobre los constructores de tipo de Merlí).

3.11 Puede que una estructura doblemente encadenada implementada sobre un vector no necesite más que un campo adicional, si la dimensión del vector es lo bastante pequeña como para codificar el valor de los dos encadenamientos con un único número entero que caiga dentro del rango de la máquina. Calcular la relación que ha de existir entre la dimensión del vector, N, y el valor entero más grande de la máquina, maxent, representable con k bits, y formular las funciones de consulta y de modificación de los dos encadenamientos codificados en este único campo.

3.12 a) Escribir en un universo parametrizado LISTA_2 un enriquecimiento de las listas con punto de interés consistente en las operaciones: contar el número de elementos de una lista, eliminar todas las apariciones de un elemento dado, invertir una lista, intercambiar el elemento de interés con su sucesor y concatenar dos listas. En la operación invertir, el nuevo punto de interés es el que antes era el anterior (si era el primero, el punto de interés queda ahora a la derecha de todo). En la operación eliminar, si se elimina el elemento

distinguido, el punto de interés se desplaza hasta la primera posición a la derecha del punto de interés que no se borra (o hasta la derecha de todo si es el caso). En la operación intercambiar, el elemento distinguido no varía. En la operación concatenar, el elemento distinguido pasa a ser el primero. Determinar los parámetros formales del universo. A continuación, escribir un universo IMPL_LISTA_2 que implemente LISTA_2.

b) Dado el universo LISTA_INTERÉS(ELEM) que encapsula la especificación habitual de las listas con punto de interés, modificarlo para que incluya todas las operaciones citadas en el apartado a) (no es necesario volverlas a especificar). A continuación, modificar el universo de implementación siguiendo una estrategia encadenada y con punteros para que también aparezcan estas operaciones. Comparar este método de trabajo con el usado en el apartado a) en cuanto a los criterios de corrección, eficiencia, modificabilidad y reusabilidad de los programas.

3.13 Otro modelo habitual de lista es el llamado listas ordenadas por posición (ing., ordered list, v. [HoS94, pp. 58-59]). En este tipo no hay punto de interés, sino que las operaciones hacen referencia al ordinal de la posición afectada y los elementos se numeran consecutivamente en orden creciente a partir del uno. En concreto, se definen las operaciones siguientes:

crea: $\rightarrow$ lista, devuelve la lista vacía
inserta: lista nat elem $\rightarrow$ lista, inserta delante del elemento i-ésimo o bien a la derecha del todo si se especifica como parámetro la longitud de la lista más uno; la numeración de los elementos cambia convenientemente
borra: lista nat $\rightarrow$ lista, borra el elemento i-ésimo; la numeración de los elementos cambia convenientemente
modifica: lista nat elem $\rightarrow$ lista, sustituye el elemento i-ésimo por el valor dado
consulta: lista nat $\rightarrow$ lista, consulta el elemento i-ésimo
longitud: lista $\rightarrow$ nat, devuelve el número de elementos de la lista

Especificar este TAD controlando las posibles referencias a posiciones inexistentes. A continuación determinar su implementación.

3.14 Dada una especificación parametrizada para los conjuntos con las típicas operaciones de Ø, añadir, $\_\in\_$, $\_\cap\_$, $\_\cup\_$ y $\_\supset\_$ (v. ejercicio 1.9), implementar el tipo según dos estrategias diferentes: a partir de la especificación para algún tipo de lista y escribiendo directamente la representación en términos de los constructores de tipo de Merlí.

3.15 Especificar el TAD colapr de las colas con prioridad (ing., priority queue; se estudian detalladamente en la sección 5.5) en las que el primer elemento es siempre el que tiene una prioridad más alta independientemente de cuándo se insertó; dos elementos con la misma prioridad conservan el orden de inserción dentro de la cola. Como operaciones se pueden considerar la cola vacía, insertar un elemento en la cola, obtener el menor elemento (que es el que tiene la prioridad más alta), desencolar el menor elemento y averiguar si la cola está vacía?. Determinar las posibles implementaciones para el tipo.

3.16 Queremos implementar una estructura de datos llamada montón que almacene cadenas de caracteres de longitud variable. No obstante, se quiere ahorrar espacio haciendo que estas cadenas estén dispuestas secuencialmente sobre un único vector de caracteres dimensionado de uno a máx_letras. Hará falta, además, una estructura adicional que indique en qué posición del vector empieza cada cadena y qué longitud tiene; esta estructura será un vector de uno a máx_par, donde máx_par represente el número máximo de palabras diferentes que se permiten en la estructura. Las operaciones son:

vacío: $\rightarrow$ montón, crea el montón sin palabras
inserta: montón cadena $\rightarrow$ <montón, nat>, añade una palabra nueva a la estructura y devuelve, además, el identificador con el cual se puede referir la palabra
borra: montón nat $\rightarrow$ montón, borra la cadena identificada por el natural; esta función no ha de intentar comprimir el espacio del montón
consulta: montón nat $\rightarrow$ cadena, obtiene la cadena identificada por el natural
listar: montón $\rightarrow$ lista_cadenas, lista todas las palabras del montón en orden alfabético

(Suponer que las listas y las cadenas ofrecen las operaciones habituales.)
Con estas operaciones, al insertar una palabra puede ocurrir que no haya suficiente espacio al final del vector de letras, pero que hayan quedado agujeros provocados por supresiones. En este caso, es necesario regenerar el montón de manera que contenga las mismas cadenas con los mismos identificadores, pero dejando todas las posiciones ocupadas del vector a la izquierda y las libres a la derecha. Implementar el tipo de manera que todas las operaciones sean lo más rápidas posible excepto la inserción, que de todas maneras tampoco ha de ser innecesariamente lenta.

3.17 Un problema muy concreto de las representaciones encadenadas de estructuras lineales definidas sobre elementos de tipo T e implementadas con punteros es la supresión de la estructura entera. Para recuperar el espacio que ocupa esta estructura es necesario recorrerla toda y liberar las celdas individualmente, lo cual representa un coste lineal. Otra opción consiste en mantener en el programa un pool de celdas de tipo T, que inicialmente está vacío y al cual van a parar las celdas al borrar la estructura entera. Cuando alguna estructura con elementos de tipo T necesita obtener una celda para guardar nuevos elementos primero se consulta si hay espacio en el pool y, si lo hay, se utiliza. Modificar la representación encadenada con punteros de las listas con punto de interés para adaptarla a este esquema, sin olvidar la operación de destruir la estructura entera.

3.18 a) Sea un TAD para las relaciones de equivalencia sobre elementos cualesquiera (que se estudia detalladamente en la sección 5.4) con operaciones: relación vacía, añadir una nueva clase con un único elemento, fusionar dos clases identificadas a partir de dos elementos que pertenecen a ellas, averiguar cuántas clases hay en la relación y si dos elementos dados son congruentes? (es decir, si están dentro de la misma clase). Determinar claramente la signatura del tipo, especificarlo e implementarlo según dos estrategias: usando la especificación de alguna estructura lineal y representando el tipo directamente con los constructores del lenguaje.

b) En algunos algoritmos puede ser útil que las clases diferentes se identifiquen mediante un natural, que puede ser asignado directamente por el usuario o bien por la estructura. Modificar el apartado anterior para adaptarlo a este nuevo requerimiento. Concretamente, la operación añadir requiere el número de la clase en el primer caso y la operación fusionar los nombres que identifican las dos clases, mientras que la operación congruentes? desaparece y da lugar a la operación clase, que devuelve el número de la clase donde reside un elemento dado. En las fusiones, considerar que la clase resultante se identifica mediante el número de la clase más grande de las dos que intervienen, mientras que la clase más pequeña desaparece. Implementar el tipo según las dos estrategias del apartado anterior.

3.19 Considerar un TAD para las matrices de enteros cuadradas dispersas (es decir, con la mayoría de elementos a cero) y con operaciones: matriz cero, definir el valor en una posición dada, obtener el valor de una posición dada, sumar y multiplicar dos matrices y calcular la matriz traspuesta. Implementar el tipo para todas las estrategias propuestas a continuación: **a)** Implementación con un vector bidimensional. **b)** Implementación con una lista de 3-tuplas <fila, columna, valor> donde sólo aparecen las posiciones no nulas. Discutir la conveniencia de ordenar la lista siguiendo algún criterio. **c)** Considerar el caso de matrices triangulares inferiores (todos los elementos por encima de la diagonal principal son nulos). En este caso, el número de elementos del triángulo es fácilmente calculable. Implementar el tipo sin operación traspuesta usando un vector unidimensional. **d)** Repetir el apartado b) considerando matrices tridiagonales (todos los elementos son nulos excepto los de la diagonal principal y sus dos diagonales adyacentes), pero sin multiplicar y con traspuesta.

3.20 a) Nos enfrentamos con el problema de implementar con encadenamientos n listas con punto de interés sobre elementos de un mismo tipo. En este caso, no podemos simplemente asignar un vector diferente a cada lista, porque si se llena uno quedando espacio en otros el programa abortaría incorrectamente. Por ello, lo que se hace es disponer de un único vector donde se almacenan todos los elementos de la lista que se encadenan adecuadamente. Además, es necesario añadir algunos vectores más para tener acceso al primer elemento, al último y al elemento distinguido de cada lista. Implementar el tipo.

b) Repetir el apartado anterior considerando las estructuras implementadas secuencialmente. A tal efecto, dividir inicialmente el espacio del vector en n trozos de la misma dimensión. Eventualmente alguno de estos trozos puede llegar a llenarse mientras que otros pueden estar poco ocupados, y en este caso es necesario redistribuir el espacio libre del vector para que las n estructuras vuelvan a tener espacio libre. Escribir el universo de implementación según dos estrategias de redistribución del espacio: **i)** Dando a cada una de las n estructuras el mismo espacio libre. **ii)** Dando a cada una de las n estructuras un espacio libre proporcional a su dimensión actual, considerando que las estructuras que han crecido más hasta ahora también tenderán a crecer más en el futuro. (Se puede consultar [Knu68, pp. 262-268].)

c) Implementar el caso particular de representación secuencial de dos pilas.