Capítulo 7 Uso y diseño de tipos abstractos de datos

A lo largo de los diferentes capítulos de este libro se ha introducido el concepto de tipo abstracto de datos como eje central en el desarrollo de software a gran escala, y se han presentado varias estructuras de datos de interés surgidas del estudio de su implementación eficiente. Ahora bien, las aplicaciones de gran tamaño son algo más que la simple definición de uno o más tipos abstractos de datos: consisten en algoritmos, muchas veces complejos, que se basan, o bien en el uso de los tipos clásicos vistos en los capítulos 3 a 6, ya sea tal como se han definido o con modificaciones, o bien en la definición de nuevos tipos abstractos que respondan a una funcionalidad y a unos requerimientos de eficiencia determinados; en este último capítulo nos centramos precisamente en la problemática de cómo usar tipos ya existentes y cómo crear nuevos tipos de datos.

Cabe decir que, actualmente, no hay ninguna estrategia universal de desarrollo de programas a gran escala; dicho de otra manera, no existe una fórmula mágica que, dado un enunciado, permita identificar claramente los tipos abstractos que deben formar parte de la solución ni tampoco las implementaciones más eficientes. Por ello, este capítulo se ha planteado como una colección de problemas resueltos, cuya elección responde al intento de ilustrar algunas situaciones muy habituales y sus resoluciones más comunes, que pueden extrapolarse a otros contextos.

Como punto de partida es necesario recordar que, tal como se expuso en el apartado 2.3.5, la programación con tipos abstractos de datos obliga frecuentemente a elegir entre eficiencia y modularidad como criterio principal de diseño, tanto al usar tipos conocidos como al definir nuevos tipos, y éste será el punto central de discusión en los enunciados aquí resueltos. Es decir, una aplicación altamente modular construida como una combinación de módulos simples, ya existentes e implementados con sendas estructuras de datos, generalmente acaba siendo un programa no del todo eficiente, ya que, por un lado, puede repetirse información redundante en las diversas subestructuras y por tanto el espacio necesario será voluminoso; por otro, puede que la codificación de las operaciones de un tipo (evidentemente, hecha después de su especificación) no explote adecuadamente todas las características de la estructura subyacente (v. el ejemplo de los conjuntos del apartado 2.3.5). El usuario o diseñador ha de decidir cuál es el criterio más importante y desarrollar su

software en consecuencia. Debe considerarse, no obstante, que hay excepciones a la regla general en las que un programa altamente modular es también el más eficiente, en cuyo caso no hay duda posible.

Para la resolución de los problemas presentados en este capítulo, supondremos que los usuarios y los diseñadores tienen a su disposición una biblioteca de universos en la que residen, como mínimo, todos los tipos vistos a lo largo del libro junto con los enriquecimientos más interesantes que han aparecido (recorridos de árboles y grafos, algoritmos de caminos mínimos, etc.). A veces supondremos que estos tipos presentan algunas operaciones adicionales de interés general (por ejemplo, longitud o pertenencia en listas). La existencia de esta biblioteca de componentes reusables es fundamental en el desarrollo de programas a gran escala, sobre todo en aquellas instalaciones compartidas por un gran número de usuarios, porque permite reutilizar los módulos que en ella residen, normalmente a través de las instancias oportunas, y reducir así el tiempo de desarrollo de nuevas aplicaciones.

7.1 Uso de tipos abstractos de datos existentes

El primer punto clave en la construcción de nuevas aplicaciones consiste en la capacidad de reutilizar universos ya existentes. La reusabilidad es posible gracias al hecho de que los universos son, casi siempre, parametrizados, lo que favorece su integración en contextos diferentes[1]. Es necesario estudiar, por tanto, qué modalidades de uso de los tipos pueden identificarse y éste será el objetivo de la sección.

Los ejemplos se han elegido desde una doble perspectiva. Por un lado, se pretende ilustrar los diferentes grados posibles de integración de los tipos de datos en una aplicación; en concreto, se presenta un enunciado que reusa completamente dos tipos de datos de la biblioteca, otro que sólo puede reutilizar los razonamientos pero no el código, y un tercero que combina ambas situaciones. Por otro, los enunciados son ejemplos clásicos dentro del ámbito de las estructuras de datos, tal como se comenta en cada uno de ellos.

Notemos que no se presenta la especificación ecuacional de los algoritmos. La razón es, una vez más, la sobreespecificación debida a la semántica inicial, que conduce a una implementación por ecuaciones más que a una especificación abstracta. En realidad, para esta clase de operaciones parece más adecuado construir otro tipo de especificación o cambiar la semántica de trabajo (por ejemplo, la semántica de comportamiento donde la operación se especificaría estableciendo las propiedades que toda resolución del algoritmo debería cumplir).

[1] En aquellos lenguajes que no ofrecen un mecanismo de parametrización de módulos, la estrategia de desarrollo es la misma que se presenta aquí, pero exige la interacción del usuario para adaptar los tipos al nuevo contexto (por simple sustitución textual) y recompilarlos posteriormente.

7.1.1 Un evaluador de expresiones

En este apartado se desarrolla hasta el último detalle una aplicación que simplemente usa las definiciones de los tipos abstractos de datos de las pilas y las colas introducidas en el capítulo 3. Se trata, por tanto, de un ejemplo paradigmático de la integración directa, sin ningún tipo de modificación, de unos componentes genéricos (residentes en la biblioteca de universos) en un algoritmo.

Más concretamente, se quiere construir un programa para evaluar expresiones aritméticas, tales como 9 / 6 ^ (1 + 3) * 4 - 5 * 9 (que, para simplificar el problema, supondremos sintácticamente correctas), compuestas de operandos, operadores y paréntesis; tomamos como operandos el conjunto de los enteros y como operadores la exponenciación ('^'), la suma ('+'), la resta ('-'), el producto ('*') y la división ('/'). Este es un ejemplo clásico en el ámbito de las estructuras de datos que aparece en gran número de textos (destacamos, por ejemplo, el tratamiento detallado de [HoS94, pp. 114-121]). Además, según el contexto, se puede resolver usando estructuras diferentes; en esta sección utilizamos una pila para calcular el valor directamente a partir de la expresión, pero si el proceso de evaluación se llevara a cabo repetidas veces sería recomendable transformar la expresión a una representación arborescente o en forma de grafo, de manera que la evaluación consistiera en un simple recorrido de la estructura (v. ejercicios 5.4, 5.23 y 6.13).

El primer paso en la resolución del problema consiste en definir con exactitud cómo se evalúa una expresión (es decir, en qué orden se aplican los operadores sobre los operandos); para ello, debe determinarse:

- La prioridad de los operadores: tomamos como más prioritaria la exponenciación, después los operadores multiplicativos (producto y división) y finalmente los aditivos (suma y resta). Así, la expresión 2 + 3 * 5 es equivalente a 2 + (3 * 5).
- La asociatividad de los operadores: a idéntica prioridad, consideramos que todos los operadores asocian por la izquierda, excepto la exponenciación, que asocia por la derecha. Así, la expresión 2 ^ 3 ^ 5 es equivalente a 2 ^ (3 ^ 5), mientras que la expresión 2 + 3 - 5 es equivalente a (2 + 3) - 5.

El orden de aplicación dado por las prioridades y la asociatividad puede romperse con los paréntesis. Así, la expresión 9 / 6 ^ (1 + 3) * 4 - 5 * 9 equivale a ((9 / (6 ^ (1 + 3))) * 4) - (5 * 9). Esta parentización queda claramente expuesta en una representación jerárquica de la expresión similar a la elegida para los términos de una signatura (v. fig. 7.1).

Un primer inconveniente que surge en la resolución del problema es el tratamiento simultáneo de los paréntesis, de la asociatividad y de las prioridades, mezclado con la aplicación de los operadores sobre los operandos. Para solucionarlo, se transforma previamente la expresión original en otra equivalente, pero escrita en una notación diferente, denominada notación polaca (ing., polish notation) o postfix; en la notación polaca los

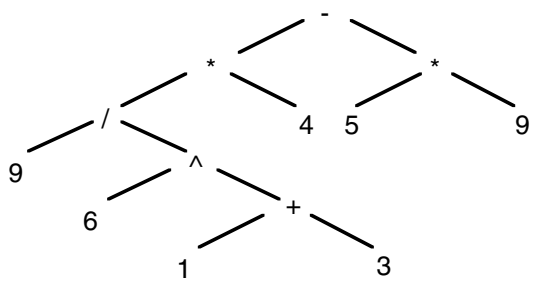


Fig. 7.1: representación jerárquica de la expresión 9/6 ^ (1+3) * 4 - 5*9.

operadores aparecen después de los operandos sobre los que se aplican, en contraposición con la notación infix habitual, donde los operadores aparecen entre los operandos. La propiedad que hace atractivas estas expresiones es que para evaluarlas basta con recorrerlas de izquierda a derecha, aplicando los operadores sobre los dos últimos operandos obtenidos; la regla de evaluación dada no depende de prioridades ni de asociatividades, y no puede alterarse mediante el uso de paréntesis.

2 + 3 * 5	2 3 5 * +
(2 + 3) * 5	2 3 + 5 *
9 / 6 ^ (1 + 3) * 4 - 5 * 9	9 6 1 3 + ^ / 4 * 5 9 * -

Fig. 7.2: unas cuantas expresiones en notación infix (a la izquierda) y postfix (a la derecha).

El algoritmo que evalúa una expresión en notación polaca (v. fig. 7.3) se basa en el uso de una pila: se leen de izquierda a derecha todos los símbolos que aparecen en la expresión; si el símbolo es un operando, se guarda dentro de la pila y, si es un operador, se desempilan tantos operandos como exige el operador, se evalúa esta subexpresión y el resultado se vuelve a empilar. Si la expresión era sintácticamente correcta, tal como se requiere en la precondición, al acabar el algoritmo la pila contendrá un único valor, que precisamente será el resultado de la evaluación.

Para codificar el algoritmo, introducimos diversos tipos auxiliares que detallamos a continuación:

- Tipo expresión: por el momento, con operaciones para obtener el primer símbolo de una expresión, eliminarlo y decidir si una expresión está o no está vacía.
- Tipo símbolo: un símbolo es un componente simple de la expresión; puede ser un operador o un operando (de momento, olvidemos los paréntesis, que no existen en las expresiones polacas). Hay una operación para decidir si un símbolo es operador o

operando; si es un operando, se le puede aplicar una operación que nos devuelve el valor que representa; sino, puede aplicarsele una operación que devuelve el operador que representa.

- Tipo pila_valores: será una instancia del universo parametrizado de las pilas.

Para que el evaluador sea lo más útil posible, se decide parametrizarlo mediante el tipo de los valores y los operadores permitidos; de esta manera, se puede depositar la aplicación en la biblioteca de componentes y reusarla fácilmente en diferentes contextos. Los parámetros se encapsularán en el universo de caracterización TIPO_EXPR que, de momento, define los géneros valor y operador y una operación aplica para aplicar un operador sobre dos operandos. Si se quisiera hacer el evaluador aún más general, TIPO_EXPR podría definir diversas operaciones de aplicación para operadores con diferente número y tipos de operandos. Al final del apartado se muestra la instancia necesaria para adaptar el evaluador al contexto del enunciado.

```
{Función evalúa_polaca(e): siendo e una expresión polaca, la evalúa con la ayuda
 de una pila. Precondición: e es una expresión sintácticamente correcta}
función evalúa_polaca (e es expresión) devuelve valor es
var p es pila_valores; v, w son valor; s es símbolo fvar
    p := PILA.crea
    mientras ¬vacía?(e) hacer
        {I ≅ la parte de expresión ya examinada de e ha sido evaluada; en la cima de
             la pila reside el resultado parcial acumulado y en el resto de la pila se
             almacenan todos aquellos valores con los que todavía no se ha operado}
        s := primer_símbolo(e); e := elimina_símbolo(e)
        si ¬ es_operador?(s) entonces {es un valor y se empila}
            p := empila(p, valor(s))
        si no {se obtienen y desempilan los operandos, se opera y se empila el resultado}
            v := cima(p); p := desempila(p)   {v es el segundo operando}
            w := cima(p); p := desempila(p)   {w es el primer operando}
            p := empila(p, aplica(w, operador(s), v))
        fsi
    fmientras
devuelve cima(p)   {cima(p) = resultado de la evaluación de e}
```

Fig. 7.3: algoritmo de evaluación de una expresión en notación polaca.

A continuación, diseñamos el algoritmo para transformar una expresión de notación infix en polaca; dado que el orden de los operandos es el mismo en ambas expresiones, la estrategia consiste en copiarlos directamente de la expresión infix a la postfix insertando adecuadamente los operadores.

Comenzamos por no considerar la existencia de paréntesis ni la asociatividad por la derecha de la exponenciación. En este caso simplificado, es suficiente recorrer la expresión de izquierda a derecha: si el símbolo que se lee es un operando, se escribe directamente en la expresión polaca; si es un operador, se escriben todos los operadores que hayan salido antes y que sean de prioridad mayor o igual a la suya. Esta escritura debe realizarse en orden inverso a la aparición, lo que indica la conveniencia de gestionar los operadores mediante una pila.

En la fig. 7.4 se muestra paso a paso la formación de la expresión polaca correspondiente a la expresión infix 3 + 2 * 5, usando una pila (el guión '-' significa la expresión vacía y λ significa la pila vacía). En (1), se empila '*' porque tiene mayor prioridad que '+'; en caso contrario, se hubiera desempilado '+' y se hubiera escrito en la expresión postfix. Cuando la expresión infix ha sido totalmente explorada, se copia lo que queda de la pila sobre la expresión postfix.

Infix	Pila	Postfix	
3 + 2 * 5	λ	-	
+ 2 * 5	λ	3	
2 * 5	+	3	
* 5	+	3 2	
5	+ *	3 2	(1)
-	+ *	3 2 5	
-	+	3 2 5 *	
-	λ	3 2 5 * +	

Fig. 7.4: formación de una expresión polaca.

Seguidamente se incorporan los paréntesis al algoritmo. La aproximación más simple consiste en tratarlos como unos operadores más con las siguientes prioridades: el paréntesis de abrir siempre se empila, y el de cerrar obliga a copiar todos los operadores que haya en la pila sobre la expresión polaca hasta encontrar el paréntesis de abrir correspondiente. En la fig. 7.5 se muestra la evolución del algoritmo para la expresión 2 * (3 + 2) * 5. En (2), se reducen operadores hasta llegar al primer paréntesis abierto de la pila.

Para implementar el algoritmo se asignan dos prioridades diferentes a todos los operadores: la que tiene como símbolo de la expresión infix, y la que tiene como símbolo de la pila; los operadores se desempilan siempre que su prioridad de pila sea mayor o igual que la prioridad de entrada del operador en tratamiento. La distinción de estas prioridades viene determinada por el tratamiento del paréntesis de abrir: cuando se encuentra en la expresión infix debe empilarse siempre, mientras que cuando se encuentra en la pila sólo se desempila al encontrarse el paréntesis de cerrar correspondiente, lo que da como resultado dos

prioridades diferentes. Para facilitar la codificación, determinaremos las prioridades mediante enteros. En la fig. 7.6 se muestra una posible asignación que cumple los requerimientos del problema. No es necesario que el paréntesis de cerrar tenga prioridad de pila, porque el algoritmo nunca lo empilará; su prioridad en la entrada debe ser tal que obligue a desempilar todos los operandos hasta encontrar el paréntesis de abrir correspondiente. El resto de operadores tienen idéntica prioridad en la entrada que en la pila, lo que significa que la asociatividad de los operadores es por la izquierda.

Infix	Pila	Postfix	
2 * (3 + 2) * 5	λ	-	
* (3 + 2) * 5	λ	2	
(3 + 2) * 5	*	2	
3 + 2) * 5	* (	2	
+ 2) * 5	* (	2 3	
2) * 5	* (+	2 3	
) * 5	* (+	2 3 2	
* 5	*	2 3 2 +	(2)
5	*	2 3 2 + *	
-	*	2 3 2 + * 5	
-	λ	2 3 2 + * 5 *	

Fig. 7.5: formación de una expresión polaca tratando paréntesis.

Operador	Prior. pila	Prior. entrada
)	-	0
^	3	3
*, /	2	2
+, -	1	1
(	-1	10

Fig. 7.6: posible asignación de prioridades a los operadores.

Precisamente este último punto es el que todavía queda por modificar, pues el enunciado del problema establece que la exponenciación asocia por la derecha. La solución es simple: si la prioridad de la exponenciación es mayor en la entrada que en la pila (por ejemplo, 4), se irán empilando diferentes operaciones consecutivas de exponenciación y acabarán aplicándose en el orden inverso al de su aparición, es decir, de derecha a izquierda.

El algoritmo se codifica en la fig. 7.7. Destaquemos que se requiere un nuevo operador,

nulo, para facilitar su escritura, que, al empezar, se coloca en la pila de operadores y al final de la expresión infix; su prioridad de pila garantiza que no saldrá nunca (es decir, la pila nunca estará vacía y el algoritmo no deberá de comprobar esta condición; para conseguir este objetivo, la prioridad debe ser lo más pequeña posible, por ejemplo -2) y tiene una prioridad de entrada que obligará a desempilar todos los símbolos que haya en la pila, excepto el mismo nulo inicial (es decir, no será necesario vaciar la pila explícitamente al final; esta prioridad debe ser lo más pequeña posible, pero mayor de -2 para no sacar el nulo inicial, por ejemplo, -1, valor que no entra en conflicto con los paréntesis de abrir, ya que no habrá ninguno en la pila al final del algoritmo).

Además, el algoritmo necesita otras operaciones y tipos auxiliares: sobre las expresiones, una operación para crear la expresión vacía y otra para añadir un símbolo al final de la expresión; sobre los operadores, dos operaciones para saber las prioridades y otra más para comparar; sobre los símbolos, una operación crear_op para crear un símbolo a partir de un operador (necesaria para no tener problemas de tipo); y una nueva instancia de las pilas para disponer de una pila de operadores.

```
{Función a_polaca(e) : siendo e una expresión infix, la transforma en polaca con la
  ayuda de una pila. Precondición: e es una expresión sintácticamente correcta}
función a_polaca (e es expresión) devuelve expresión es
var e2 es expresión; p es pila_ops; s es símbolo; op es operador fvar
    p := empila(PILA.crea, nulo)
    e := añade_símbolo(e, crea_op(nulo)); e2 := crea
    mientras ¬ vacía?(e) hacer
        {I ≅ los valores ya examinados de e están en e2 en el mismo orden, y los
            operadores ya examinados de e están en e2 o en p, según sus prioridades}
        s := primer_símbolo(e)
        si ¬ es_operador?(s) entonces {es un valor y se añade al resultado}
            e2 := añade_símbolo(e2, s)
        si no {operador que se empila o se añade al resultado según la prioridad}
            op := operador(s)
            mientras prio_pila(cima(p)) ≥ prio_ent(op) hacer
                e2 := añade_símbolo(e2, crea_op(cima(p)))
                p := desempila(p)
            fmientras
            si op = par_cerrar entonces p := desempila(p)   {paréntesis de abrir}
                                  si no p := empila(p, op)
            fsi
        fsi
        e := elimina_símbolo(e)
    fmientras
devuelve e2
```

Fig. 7.7: algoritmo para transformar una expresión infix en notación polaca.

En la fig. 7.8 se muestra el encapsulamiento en universos de la aplicación. El universo EVALUADOR, parametrizado por los valores y los operadores sobre los que se define el problema, introduce los géneros de los símbolos y de las expresiones y la función evalúa. Los símbolos se especifican como una unión disjunta de dos tipos, mientras que las expresiones, dadas las operaciones y el comportamiento requeridos, se definen como colas de símbolos; por los motivos ya citados, la función evalúa no se especifica. Por lo que respecta al universo IMPL_EVALUADOR, implementa el tipo de las expresiones eligiendo una de las implementaciones para colas existentes en la biblioteca (por punteros, para que así no se precise ninguna cota), el tipo de los símbolos como una tupla variante, y la función evalúa usando como operaciones auxiliares las ya codificadas a_polaca y evalúa_polaca y realizando las instancias necesarias sobre las pilas. Las pilas auxiliares también se implementan mediante punteros. Por último, notemos que los parámetros formales encapsulados en TIPO_EXPR presentan todos los requerimientos comentados durante la resolución del problema, sobre todo en lo que respecta a las prioridades de los operadores.

Los tres universos dados deberían residir en la biblioteca de componentes; según el sistema gestor de la biblioteca, podrían residir los tres en un único componente, o bien cada uno por separado, si se distingue en la biblioteca un apartado para universos de especificación, otro para universos de implementación y un tercero para universos de caracterización. Sea como fuere, es muy sencillo crear un evaluador concreto a partir de los requisitos de un problema; por ejemplo, en la fig. 7.9 se muestra el evaluador que responde a las características del enunciado. Primero se especifican y se implementan los géneros y las operaciones que actuarán de parámetros reales y, a continuación, se instancia el evaluador; notemos que se construye directamente una implementación, porque la especificación queda clara a partir de la relación existente entre el universo parametrizado y la instancia.

7.1.2 Un gestor de memoria dinámica

A continuación nos enfrentamos con una situación completamente inversa a la anterior: se dispone de una estructura de datos dada, que simula la memoria dinámica a la que acceden los programas en ejecución a través de punteros y que se quiere gestionar con las primitivas habituales obtener_espacio (reserva de espacio para guardar un objeto) y liberar_espacio (devolución del espacio cuando no se necesita más el objeto). En este contexto, no es recomendable usar un tipo de datos auxiliar, porque se malgastaría espacio, sino que se aprovecha la misma memoria para diseñar una estructura de bloques libres con los trozos de memoria no ocupada; esta estructura será lineal y para implementarla eficientemente usaremos diversas técnicas vistas en la sección 3.3.

También esta aplicación es un clásico en el mundo de las estructuras de datos y existen varias estrategias para implementarla (v. [HoS94, pp. 172-188], que es la base de la propuesta que aquí se da, y también [AHU83, cap. 12] o [Knu68, pp. 470-489]).

```
universo TIPO_EXPR caracteriza
    usa ENTERO, BOOL
    tipo valor, operador
    ops nulo, par_cerrar, par_abrir: → operador
        aplica: valor operador valor → valor
        _=_, _≠_: operador operador → bool
        prio_ent, prio_pila: operador → entero
    ecns prio_ent(nulo) > prio_pila(nulo) = cierto
         [x ≠ nulo ∧ x ≠ par_cerrar] ⇒ prio_ent(x) > prio_ent(par_cerrar) = cierto
         ... y el resto de propiedades sobre los paréntesis (similares a la anterior), _=_ y _≠_
funiverso

universo EVALUADOR (TIPO_EXPR) es
    tipo símbolo                    {primero, definición del TAD de los símbolos}
    ops crea_valor: valor → símbolo
        crea_op: operador → símbolo
        es_operador?: símbolo → bool
        valor: símbolo → valor
        operador: símbolo → operador
    errores [¬ es_operador?(s)] ⇒ operador(s); [es_operador?(s)] ⇒ valor(s)
    ecns es_operador?(crea_op(op)) = cierto; es_operador?(crea_valor(v)) = falso
         valor(crea_valor(v)) = v; operador(crea_op(op)) = op
    instancia COLA(ELEM) donde elem es símbolo  {a continuación el TAD de las expresiones}
    renombra cola por expresión, encola por añade_símbolo
             cabeza por primer_símbolo, desencola por elimina_símbolo
    ops evalúa: expresión → valor               {por último la función de interés}
funiverso

universo IMPL_EVALUADOR (TIPO_EXPR) implementa EVALUADOR(TIPO_EXPR)
    tipo símbolo es tupla
                    caso es_operador de tipo bool igual a
                        cierto entonces op es operador
                        falso entonces v es valor
                ftupla
    ftipo
    tipo expresión implementado con COLA_POR_PUNTEROS(ELEM) ftipo
    ...implementación de las operaciones sobre símbolos, trivial
    instancia PILA(ELEM) implementada con PILA_POR_PUNTEROS(ELEM)
        donde elem es operador renombra pila por pila_ops
    instancia PILA(ELEM) implementada con PILA_POR_PUNTEROS(ELEM)
        donde elem es valor renombra pila por pila_valores
    función evalúa (e es expresión) devuelve valor es devuelve evalúa_polaca(a_polaca(e))
    función privada evalúa_polaca (e es expresión) devuelve valor es … {v. fig. 7.3}
    función privada a_polaca (e es expresión) devuelve expresión es … {v. fig. 7.7}
funiverso
```

Fig. 7.8: encapsulamiento en universos del evaluador.

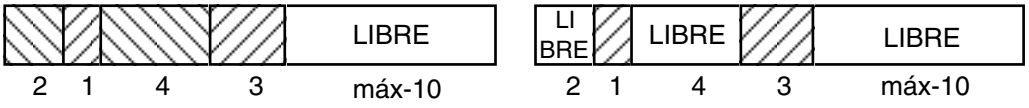
```
universo OPS_Y_VALORES_ENTEROS es
    usa ENTERO, BOOL
    tipo op_entero
    ops suma, resta, mult, div, exp, par_cerr, par_abrir, nulo: → op_entero
        aplica: entero op_entero entero → entero
        _=_, _≠_: op_entero op_entero → bool
        prio_ent, prio_pila: op_entero → entero
    ecns aplica(v, suma, w) = v + w; ...
         prio_ent(suma) = 1; prio_pila(suma) = 1; ...
funiverso

universo IMPL_OPS_Y_VALORES_ENTEROS implementa OPS_Y_VALORES_ENTEROS
    usa ENTERO, BOOL
    tipo op_entero = (suma, resta, mult, div, exp, par_cerr, par_abrir) ftipo
    ... funciones aplica, prio_ent y prio_pila: distinguen el tipo de operador con una
      alternativa múltiple y actúan en consecuencia; funciones _=_ y _≠_: simple
      comparación de valores
funiverso

universo EVALUADOR_ENTEROS es
    usa OPS_Y_VALORES_ENTEROS
        implementado con IMPL_OPS_Y_VALORES_ENTEROS
    instancia EVALUADOR(TIPO_EXPR)
        implementado con IMPL_EVALUADOR(TIPO_EXPR)
        donde valor es entero, operador es op_entero ...
funiverso
```

Fig. 7.9: posible instancia del evaluador.

Exactamente, la situación es la siguiente: se dispone de una memoria sobre la que los usuarios realizan peticiones de espacio de una dimensión determinada (generalmente, peticiones diferentes exigirán tamaño diferente). Para cada petición se busca en la memoria un espacio contiguo mayor o igual que el pedido; si no se encuentra, se da error, si se encuentra, se deja este espacio disponible para el usuario (por ejemplo, devolviendo un apuntador a su inicio) y, de alguna manera, se toma nota de que está ocupado; si el espacio encontrado es mayor que el pedido, el trozo sobrante debe continuar disponible para atender nuevas peticiones. Además, los usuarios irán liberando este espacio cuando ya no lo necesiten. Generalmente, el orden de reserva no coincidirá con el orden de liberación y se intercalarán reservas y liberaciones de manera aleatoria. Notemos que en la memoria pueden quedar agujeros (es decir, bloques libres), que han de tratarse en reservas posteriores.

Cuando un usuario pide un trozo de memoria de tamaño n, debe buscarse un bloque libre mayor o igual a n. Hay diversas políticas posibles:

- Buscar por la memoria hasta encontrar el primer bloque no ocupado de tamaño mayor o igual que n. Es la política del primer ajuste (ing., first fit).



(a) estado de la memoria después de reservar bloques de 2, 1, 4 y 3 Ks.

(b) estado de la memoria después de liberar el primer y el tercer bloque.

Fig. 7.10: reserva y liberación de bloques en la memoria.

- Buscar el bloque que más se ajuste a la dimensión n por toda la memoria. Es la política del mejor ajuste (ing., best fit).

El first fit es más sencillo y, además, se puede observar empíricamente que generalmente aprovecha la memoria igual de bien que el best fit, porque con esta última estrategia pueden quedar trozos de memoria muy pequeños y difícilmente utilizables. Por ello, en el resto de la sección se adopta la política first fit [2].

Estudiemos la implementación. Consideraremos la memoria como un vector de palabras y haremos que todas las estructuras de datos necesarias para implementar el enunciado residan en ella, de forma que la estructura resultante sea autocontenida. Asímismo, consideraremos que la unidad que se almacena en una palabra es el entero:

```
tipo palabra es entero ftipo
tipo memoria es vector [de 0 a máx-1] de palabra ftipo
```

En la memoria coexisten los bloques libres con los bloques ocupados; para localizar rápidamente los bloques libres, éstos se encadenan formando una lista, de modo que, cada vez que el usuario necesite un bloque de un tamaño determinado, se buscará en la lista y cuando el usuario libere un bloque, se insertará al inicio de la misma. Cada uno de estos bloques debe guardar su dimensión, para determinar si se puede asignar a una petición.

Así, cada una de las palabras de la memoria se encuentra en una de tres situaciones posibles:

- En un bloque ocupado, es decir, contiene datos significativos.
- Empleada para la gestión de la lista de sitios libres: supondremos que la primera palabra de un bloque libre contiene la dimensión del bloque, mientras que la segunda contiene el apuntador al siguiente de la lista.
- Cualquier otra palabra en un bloque libre, es decir, no contiene ningún dato significativo.

[2] Hay otra estrategia similar, la del peor ajuste (ing., worst fit), que aquí no consideramos.

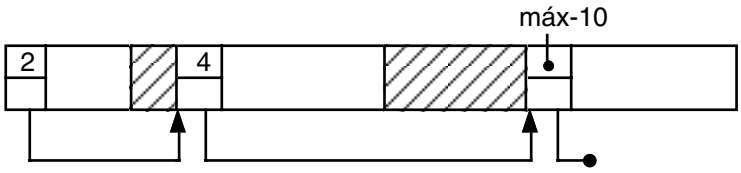


Fig. 7.11: implementación de la memoria de la figura 7.10(b).

En la fig. 7.12 aparece una primera versión del algoritmo de la operación de obtención de espacio, que busca el primer bloque de tamaño mayor o igual al pedido; cuando lo encuentra, devuelve la dirección de la parte del bloque de dimensión n que acaba al final del bloque. Si el bloque tiene exactamente[3] el espacio pedido, lo borra de la lista (con coste constante, porque siempre se guarda un apuntador al anterior). Si no lo encuentra, devuelve el valor -1 (que es el valor del encadenamiento del último bloque de la lista). Notemos que la búsqueda comienza en el segundo bloque de la lista. Supondremos que al crear la memoria se deposita al inicio de la lista un bloque fantasma de dimensión cero que simplifica los algoritmos; este bloque reside físicamente a partir de la posición 0 de la memoria. El invariante usa la función cadena adaptada al tipo de la memoria para establecer que ningún bloque examinado puede satisfacer la petición.

```
acción obtener_espacio[4] (ent/sal M es memoria; ent tam es nat; sal p es nat) es
var q es nat; encontrado es bool fvar
        {posición 1: encadenamiento del fantasma}
    p := M[M[1]]; q := 0; encontrado := falso
    mientras ¬encontrado ∧ (p ≠ -1) hacer
            {I ≅ ∀x: x∈(cadena(M, 0)-cadena(M, p)): M[x] < tam ∧
                encontrado ⇒ M[p] ≥ tam ∧ la memoria se actualiza convenientemente}
        si M[p] ≥ tam entonces {se selecciona el bloque}
            encontrado := cierto; M[p] := M[p] - tam
            si M[p] < 2 entonces M[q+1] := M[p+1]    {debe borrarse}
                        si no p := p + M[p]   {se devuelve el trozo final}
            fsi
        si no q := p; p := M[p+1]   {sigue la búsqueda}
        fsi
    fmientras
facción
```

Fig. 7.12: algoritmo de obtención de espacio libre.

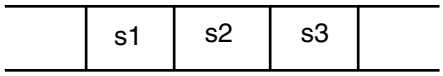
[3] En realidad, el algoritmo procura no dejar bloques libres menores de dos palabras para guardar el tamaño y el encadenamiento.

[4] La codificación de obtener_espacio que se presenta no se corresponde exactamente con la primitiva obtener_espacio que ofrece el lenguaje, porque aquí se declara como parámetro la dimensión del bloque; se puede suponer que esta llamada la genera el compilador del lenguaje (que conocerá el número de palabras exigido por una variable de un tipo dado) a partir de la instrucción correspondiente.

Esta implementación presenta un par de problemas de fácil solución:

- Es posible que dentro de la lista queden bloques de una dimensión tan pequeña que difícilmente puedan ser usados para satisfacer nuevas peticiones. Por ello, la lista va degenerando lentamente y, en consecuencia, se ralentiza su recorrido. La solución consistirá en definir un valor ε tal que, si el hipotético bloque libre sobrante de una petición tiene una dimensión inferior a ε, se ocupe el bloque entero sin fragmentar. El valor ε debe ser mayor o igual que el número de palabras necesario para la gestión de la lista de sitios libres.
- Como siempre se explora la lista desde su inicio, en este punto se concentran bloques de dimensión pequeña que sólo satisfacen peticiones de poco espacio, lo que alarga el tiempo de búsqueda para peticiones de mucho espacio, pues se efectúan consultas inútiles al principio. Por ello, la exploración de la lista no comenzará siempre por delante, sino a partir del punto en que terminó la última búsqueda; para facilitar esta política se implementará la lista circularmente.

Estudiemos a continuación el algoritmo de liberación de espacio. En un primer momento parece que su implementación es muy sencilla: cada vez que el usuario libera un bloque, éste debe insertarse en la lista de bloques libres; ahora bien, es necesario considerar el problema de la fragmentación de memoria; supongamos que en la situación siguiente:



se libera el bloque s2. ¿Qué ocurre si s1 y/o s3 también son bloques libres? ¡Pues que resultarán dos o tres bloques libres adyacentes! Es decir, se fragmenta innecesariamente la memoria; obviamente, es mucho mejor fusionar los bloques libres adyacentes para obtener otros más grandes y así poder satisfacer peticiones de gran tamaño.

Con la implementación actual, detectar la adyacencia de bloques no es sencillo. Por lo tanto, modificaremos la estructura de datos de la siguiente manera (v. fig. 7.13):

- Tanto la primera como la última palabra de todo bloque identificarán si es un bloque libre o un bloque ocupado. Así, para consultar desde s2 el estado de los dos bloques adyacentes, bastará con acceder a la última palabra de s1 y a la primera de s3, siendo ambos accesos inmediatos. Se sabrá que un bloque es libre porque esas dos palabras serán positivas, mientras que serán negativas o cero en caso contrario; si la primera palabra es negativa de valor -n, se interpreta como un bloque ocupado de tamaño n.
- La última palabra de cada bloque libre tendrá también un apuntador a la primera palabra del mismo bloque. Así, si s1 es un bloque libre, se accederá rápidamente a su inicio desde s2. En caso de estar ocupado, esta palabra valdrá -1 para cumplir la condición anterior.

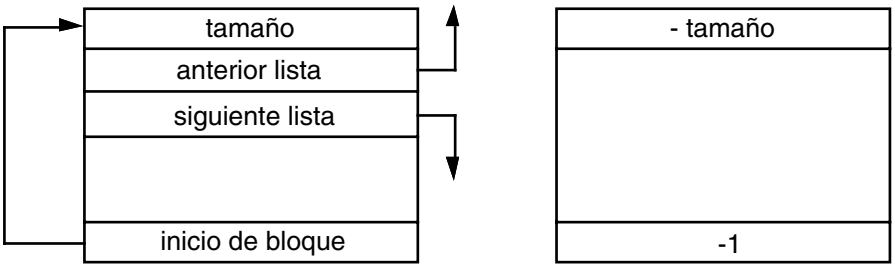


Fig. 7.13: información asociada a cada bloque de memoria (izq., libre; der., ocupado).

Además, para optimizar el coste temporal de los algoritmos, los bloques libres formarán una lista circular doblemente encadenada y se conservará el elemento fantasma al inicio de la lista (físicamente, en la posición más baja de la memoria). También se dejará una palabra fantasma de valor -1 a la derecha del todo de la memoria simulando un bloque ocupado, mientras que el bloque fantasma de la izquierda se considerará ocupado a efectos del estudio de la adyacencia, aunque esté en la lista de bloques libres, de manera que ninguno de los dos se fusionará nunca; con la existencia de los dos bloques fantasma, no habrá casos especiales en el estudio de la adyacencia y los algoritmos quedarán simplificados. En la fig. 7.14 se muestra el algoritmo de creación de memoria resultante de estas convenciones. Sería deseable recapitular todas estas condiciones en el invariante de la representación de la memoria, lo que queda como ejercicio para el lector.

```
acción crear_espacio (sal M es memoria; sal act[5] es entero) es
        {bloque fantasma al inicio de la lista de sitios libres}
    M[0] := 0; M[1] := 4; M[2] := 4; M[3] := -1
        {bloque fantasma a la derecha de la memoria}
    M[máx-1] := -1
        {bloque libre con el resto de la memoria}
    M[4] := máx-4; M[5] := 0; M[6] := 0; M[máx-2] := 4
        {actualización de apuntador al actual}
    act := 0
facción
```

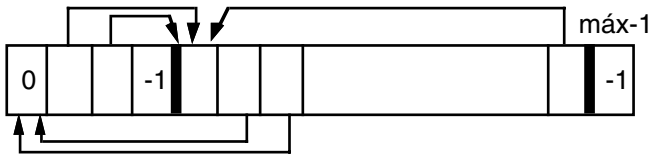


Fig. 7.14: creación de la memoria: algoritmo y esquema.

[5] De hecho, act también podría residir en alguna palabra de la memoria M.

El algoritmo de liberación de espacio de un bloque s2 (v. fig 7.15) ha de considerar las cuatro combinaciones posibles de los estados de los bloques adyacentes al liberado (s1 y s3).

- s1 y s3 están ocupados: se inserta s2 dentro de la lista de bloques libres y se actualiza su información de control.
- s1 está libre y s3 está ocupado: debe fusionarse el bloque liberado con s1, lo que implica modificar campos tanto de s2 como de s1.
- s1 está ocupado y s3 está libre: debe fusionarse el bloque liberado con s3, lo que implica modificar campos tanto de s2 como de s3.
- s1 y s3 están libres: deben fusionarse los tres bloques en uno, lo que implica modificar campos tanto de s1 como de s3 (pero no de s2).

En la figura se incluye también el algoritmo de obtención de espacio. En ambos algoritmos debe recordarse que, siendo p el comienzo de un bloque libre, se dispone de los siguientes datos en las siguientes posiciones: en M[p], el tamaño del bloque; en M[p+1] y M[p+2], los apuntadores a los elementos anterior y siguiente de la lista, respectivamente; y en M[p+M[p]-1], el apuntador al inicio del bloque (es decir, p mismo). En el caso de que p sea la dirección de inicio de un bloque ocupado, M[p] es el tamaño del bloque ocupado con el signo cambiado (es decir, negativo), mientras que M[p-M[p]-1] es igual a -1. Las pre y postcondiciones de los algoritmos quedan como ejercicio para el lector.

7.1.3 Un planificador de soluciones

En la situación propuesta en este tercer apartado intervienen varios TAD típicos, que son usados como estructuras auxiliares en un algoritmo de planificación. Debe destacarse que uno de ellos, un grafo dirigido, existe sólo a nivel conceptual para formular la estrategia de resolución del problema.

El planteamiento general del problema de planificación se ha expuesto en el apartado 6.3.2: dado un estado inicial y un conjunto de transiciones atómicas, se trata de determinar la menor secuencia de transiciones que lleva hasta un estado final determinado. En el ejercicio 7.1, entre otras cosas, se pide precisamente diseñar un algoritmo genérico aplicable a cualquier situación que responda a este planteamiento general. En esta sección, no obstante, se estudia un enunciado concreto: se dispone de tres recipientes de capacidades 8, 5 y 3 litros, de forma que inicialmente el primero se llena de vino y los otros dos están vacíos, y se quiere determinar cuál es la secuencia mínima de movimientos de líquido que debe hacerse para pasar de la situación inicial a la final, que consiste en que el recipiente más pequeño quede vacío y los otros dos con cuatro litros cada uno.

Es obvio que este problema puede plantearse a partir del esquema general enunciado antes. Para ello, basta con definir qué es un estado y cuáles son las posibles transiciones.

```
acción liberar_espacio (ent/sal M es memoria; ent/sal act es nat; ent p es entero) es
var q1, q2, n son enteros fvar
    n := -M[p]
    si (M[p-1] = -1) ∧ (M[p+n] < 0) entonces {no se fusiona}
            {conversión del bloque en libre}
        M[p] := n; M[p+n-1] := p; M[p+1] := act; M[p+2] := M[act+2]
            {inserción en la lista doblemente encadenada}
        M[M[p+2]+1] := p; M[act+2] := p
        act := p
    si no si (M[p-1] ≠ -1) ∧ (M[p+n] < 0) entonces {fusión del bloque liberado con el vecino izquierdo}
        q1 := M[p-1]; M[q1] := M[q1] + n; M[p+n-1] := q1
        act := q1
    si no si (M[p-1] = -1) ∧ (M[p+n] > 0) entonces {fusión del bloque liberado con el vecino derecho}
        M[p] := n+M[p+n]; M[p+M[p]-1] := p; M[M[p+n+1]+2] := p; M[M[p+n+2]+1] := p
        M[p+2] := M[p+n+2]; M[p+1] := M[p+n+1]
        act := p
    si no {fusión de los tres bloques}
        M[M[p+n+1]+2] := M[p+n+2]; M[M[p+n+2]+1] := M[p+n+1]
        q1 := M[p-1]; M[q1] := M[q1]+n+M[p+n]; M[q1+M[q1]-1] := q1
        act := q1
    fsi
facción

acción obtener_espacio (ent/sal M es memoria; ent tam es nat; ent/sal act es nat; sal p es entero) es
var q1, q2, n son enteros; encontrado es bool fvar
    p := M[act+2]; encontrado := falso        {bloque de partida}
    {act = A}
    repetir
        {I ≅ ∀x: x∈(cadena(M, A)-cadena(M, p)): M[x] < tam ∧
            (encontrado ⇒ M[p] ≥ tam ∧ act = p ∧ la memoria se ha actualizado convenientemente)}
        si M[p] ≥ tam entonces {el bloque satisface la petición}
            encontrado := cierto; n := M[p]
            si n-tam < ε entonces      {asignación del bloque entero}
                q1 := M[p+2]; q2 := M[p+1]; M[q2+2] := q1; M[q1+1] := q2    {se borra de la lista}
                M[p] := -n; M[p+n-1] := -1                                  {el bloque pasa a estar ocupado}
                act := q1                  {la próxima búsqueda comenzará por el siguiente bloque}
            si no      {se devuelven las últimas tam palabras del bloque}
                M[p] := n-tam; M[p+n-tam-1] := p                {formación del nuevo bloque libre}
                act := p                        {la próxima búsqueda comenzará por este bloque}
                p := p+n-tam                                    {comienzo del bloque asignado}
                M[p] := -tam; M[p+tam-1] := -1                  {el bloque pasa a estar ocupado}
            fsi
        si no p := M[p+2]   {avanza al siguiente bloque}
        fsi
    hasta que (p = M[act+2]) ∨ encontrado
    si ¬encontrado entonces p := -1 fsi
facción
```

Fig. 7.15: algoritmos de obtención y liberación de espacio.

- Un estado viene determinado por los contenidos de los recipientes; diferentes combinaciones son diferentes estados. Así, un estado se identifica mediante tres naturales, uno para cada recipiente, que representan la cantidad de vino.
- Una transición consiste en trasvasar líquido de un recipiente a otro hasta que se vacíe el primero o se llene el segundo. Así, todas las transiciones posibles se identifican según los recipientes fuente y destino del movimiento.

En la fig. 7.16 se muestra una especificación ESTADO para los estados y las transiciones, que cumple estas características; las seis transiciones posibles se representan con un natural: 1, movimiento del recipiente mayor al mediano; 2, movimiento del recipiente mayor al menor; 3, movimiento del recipiente mediano al mayor; 4, movimiento del recipiente mediano al menor; 5, movimiento del recipiente menor al mayor y 6, movimiento del recipiente menor al mediano. Se define el tipo de los estados como un producto cartesiano de tres enteros que cumplen las características dadas. Por lo que respecta a las transiciones, hay una operación para determinar si la transición es válida y otra para aplicarla. Evidentemente la especificación resultante no puede crearse como instancia de ningún universo de la biblioteca; además, no parece indicado incluirla en ella, porque es un TAD ad hoc para el problema resuelto. En cambio, en caso de generalizar la solución, ésta se podría depositar en la biblioteca el universo parametrizado para utilizarlo en cualquier problema de planificación, y también el universo de caracterización de los estados y las transiciones, del que ESTADO sería un parámetro asociado válido.

```
universo ESTADO es
    usa NAT, BOOL
    tipo estado
    ops <_, _, _>: nat nat nat → estado
        inicial, final: → estado
        _=_, _≠_: estado estado → bool
        posible?: estado nat → bool
        aplica: estado nat → estado
    errores ∀n,x,y,z∈nat; ∀e∈estado
        [(x > 8) ∨ (y > 5) ∨ (z > 3) ∨ (x+y+z ≠ 8)] ⇒ <x, y, z>
        [(n = 0) ∨ (n > 6)] ⇒ posible?(e, n), aplica(e, n)
        [¬ posible?(e, n)] ⇒ aplica(e, n)
    ecns ∀a,b,c,x,y,z∈nat
        inicial = <8, 0, 0>; final = <4, 4, 0>
        posible?(<x, y, z>, 1) = x > 0 ∧ y < 5; ...
        [x ≥ 5-y] ⇒ aplica(<x, y, z>, 1) = <x-5+y, 5, z>; ...
        [x < 5-y] ⇒ aplica(<x, y, z>, 1) = <0, x+y, z>; ...
        (<a, b, c> = <x, y, z>) = (a = x) ∧ (b = y) ∧ (c = z)
        (<a, b, c> ≠ <x, y, z>) = ¬ (<a, b, c> = <x, y, z>)
funiverso
```

Fig. 7.16: especificación de los estados.

La resolución del problema se puede plantear en términos de un grafo cuyos nodos sean estados y las aristas transiciones. El grafo presenta muchos caminos diferentes que llevan del estado inicial al final, y unos son más cortos que otros, porque los más largos pueden dar vueltas o contener ciclos. Para encontrar el camino más corto, basta con recorrer el grafo en amplitud a partir del estado inicial y hasta visitar el estado final; una primera opción, pues, es construir el grafo y, posteriormente, recorrerlo. Obviamente, el algoritmo resultante es muy ineficiente en cuanto al tiempo, porque la construcción del grafo obliga a generar todas las posibles transiciones entre estados, y también en cuanto al espacio, porque se generan nodos que, en realidad, es evidente que no formarán parte de la planificación mínima, ya que se obtienen después de aplicar más transiciones que las estrictamente necesarias. Debe considerarse, no obstante, que sólo hay 24 posibles combinaciones válidas de 8 litros de vino en tres recipientes de 8, 5, y 3 litros; por ello, en el contexto de este enunciado no puede descartarse la generación del grafo y queda como ejercicio para el lector.

Si no estamos seguros de que el proceso de generación del grafo vaya a ser costoso, por pocos estados que tenga o, simplemente, si queremos plantear el mismo problema pensando en la posibilidad de adaptarlo posteriormente a un caso más general, se puede simular el recorrido en amplitud usando una cola para ordenar los nodos (v. apartado 6.3.2). Cada elemento de la cola será, no sólo el vértice a visitar, sino también el camino seguido para llegar a él, de manera que cuando se genere el vértice correspondiente al estado final se pueda recuperar la secuencia de transiciones seguidas. Es decir, los elementos de la cola serán pares formados por el nodo a visitar y una lista con las transiciones aplicadas sobre el estado inicial para llegar al vértice actual. El resultado es una descomposición modular más eficiente que la anterior estrategia, aunque todavía se está desperdiciando espacio, porque las listas residentes en la cola se podrían fusionar (como haremos más adelante). La resolución se puede mejorar si se usa el típico conjunto para guardar los estados que ya han sido visitados en el recorrido, y evitar así su tratamiento reiterado.

La fig. 7.17 muestra la estructuración en universos de esta solución: el universo de especificación, PLANIFICADOR, sólo fija la signatura y el universo de implementación, PLANIFICADOR_MODULAR, la codifica. En la signatura se declara como tipo del resultado una lista de naturales, que representan las transiciones aplicadas sobre el estado inicial para llegar al final, en el orden que deben aplicarse. En la implementación se definen las instancias necesarias y se implementa la estrategia descrita. Para controlar la posibilidad de que el estado inicial sea inalcanzable (aunque no es el caso en este ejemplo), se comprueba que la cola no esté vacía en la condición de entrada del bucle, en cuyo caso se devuelve una lista vacía. Se ha elegido una implementación por punteros de la cola y de las listas para no preocuparnos de calcular el espacio (aunque sabemos que la cola, como ya se ha dicho anteriormente, no podrá tener más de 24 elementos) y para su adaptación futura al caso general, donde los tamaños de las variables pueden ser absolutamente desconocidos. No obstante, notemos que la función planifica no elige las implementaciones de los pares, ni de los estados ni de los conjuntos; este paso es necesario para finalizar la aplicación. Por lo que

respecta a los pares y a los estados, se pueden usar las tuplas de dos y tres campos, respectivamente. Por lo que respecta a los conjuntos, a priori, para determinar rápidamente la pertenencia de un elemento la representación óptima es la dispersión; ahora bien, como sólo puede haber 24 estados como mucho, en realidad una simple estructura encadenada es suficiente y así se podría hacer en el universo.

Esta solución es eficiente temporalmente, pero contiene información redundante, ya que en la cola habrá listas de transiciones que, en realidad, compartirán sus prefijos, porque representarán caminos dentro del grafo que tendrán las primeras aristas iguales. Si el tamaño del grafo fuera mayor, esta redundancia sería muy problemática e, incluso, prohibitiva. Por ello, estudiamos una variante que se puede aplicar en el caso general, que evita la redundancia de información sin perder rapidez, pero sacrificando la modularidad de la solución; según el criterio del programador y el contexto de uso, será preferible una estrategia u otra.

La variante propuesta modifica el planteamiento anterior de manera que la cola no contenga listas sino apuntadores a nodos del grafo, que se irán calculando a medida que se recorra en amplitud: cada vez que, desde un estado ya existente, se llega a un nuevo estado, se incorpora al grafo un vértice que lo representa y una arista que lo une con el de partida; el proceso se repite hasta llegar al estado final. En consecuencia, el grafo resultante es en realidad un árbol porque, por un lado, sólo aparecerán los vértices y las aristas visitados en el recorrido en amplitud desde el estado inicial hasta el final y, por el otro, un vértice sólo se visita una vez. Para poder recuperar la secuencia de movimientos que llevan del estado inicial al final, este árbol se implementa con apuntadores al padre (v. fig. 7.18).

```
universo PLANIFICADOR es
    usa NAT
    instancia LISTA_INTERÉS(ELEM) donde elem es nat renombra lista por lista_nat
    ops planifica: → lista_nat
funiverso

universo PLANIFICADOR_MODULAR implementa PLANIFICADOR
    usa ESTADO, NAT, BOOL
    instancia CJT_∈ (ELEM_=) donde elem es estado, = es ESTADO.= renombra cjt por cjt_estados
    instancia PAR (A, B son ELEM) donde A.elem es estado, B.elem es lista_nat
    instancia COLA (ELEM) implementada con COLA_POR_PUNTEROS(ELEM)
        donde elem es par
        renombra cola por cola_pares
    tipo lista_nat implementado con LISTA_INTERÉS_ENC_PUNTEROS(ELEM) ftipo

    función planifica devuelve lista_nat es
    var  S es cjt_estados; l es lista_nat; C es cola_pares
        fuente, destino son estado; i es nat; encontrado es bool
    fvar
          {inicialmente, se encola el estado inicial con una lista vacía de transiciones, se
           marca este estado inicial como tratado y se inicializa la variable de control}
        C := encola(COLA.crea, <ESTADO.inicial, LISTA_INTERÉS.crea>)
        S := Ø∪{ ESTADO.inicial}; encontrado := falso
        mientras ¬ encontrado ∧ ¬ COLA.vacía?(C) hacer {cola vacía ⇒ estado final inalcanzable}
                {se obtiene el siguiente en amplitud}
            <fuente, l> := cabeza(C); C := desencola(C)
                {a continuación se estudian las seis transiciones posibles}
            i := 1
            mientras (i ≤ 6) ∧ ¬ encontrado hacer
                si posible?(fuente, i) entonces  {si no, no es necesario estudiar la transición}
                    destino := aplica(fuente, i)
                    si destino = ESTADO.final entonces
                        encontrado := cierto; l := inserta(l, i)  {se guarda la última transición}
                    si no {si el estado no se había generado, se marca y se encola}
                        si ¬ (destíno ∈ S) entonces
                            S := S∪{destino}; C := encola(C, <destino, inserta(l, i)>)
                        fsi
                    fsi
                fsi
                i := i + 1
            fmientras
        fmientras
        si ¬ encontrado entonces l := LISTA_INTERÉS.crea fsi
    devuelve l
funiverso
```

Fig. 7.17: implementación modular del planificador.

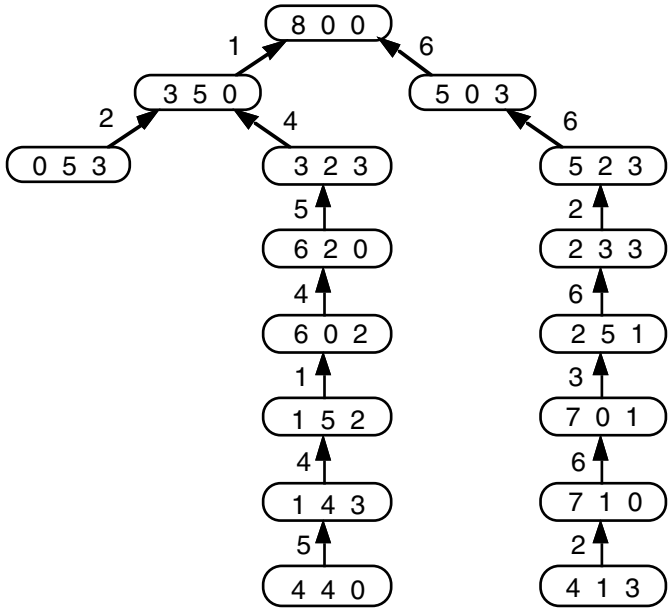


Fig. 7.18: árbol resultante de recorrer el grafo asociado al problema en amplitud.

La fig. 7.19 contiene la implementación de planifica en esta versión. Notemos que, si bien el conjunto de estados se mantiene igual, el árbol se implementa directamente en el algoritmo: no se define en un universo aparte, por ser un tipo con operaciones ciertamente peculiares y de poca utilidad en otros contextos; sobre todo lo caracteriza el hecho de que se accede a sus nodos directamente mediante un apuntador. El resto del algoritmo es similar y no merece ningún comentario.

```
universo PLANIFICADOR_EFICIENTE implementa PLANIFICADOR
    usa ESTADO, NAT, BOOL
    instancia CJT_∈ (ELEM_=) implementado con CJT_∈_POR_PUNTEROS(ELEM)
        donde elem es estado, = es ESTADO.=
        renombra cjt por cjt_estados
    tipo privado nodo es     {un nodo del árbol}
        tupla e es estado; transición es nat; padre es ^nodo ftupla
    ftipo
    instancia COLA(ELEM) implementado con COLA_POR_PUNTEROS(ELEM)
        donde elem es ^nodo
        renombra cola por cola_nodos
    tipo lista_nat implementado con LISTA_INTERÉS_ENC_PUNTEROS(ELEM) ftipo
```

Fig. 7.19: implementación eficiente del algoritmo de planificación.

```
función planifica devuelve lista_nat es
var S es cjt_estados; l es lista_estados; C es cola_nodos
    p, q son ^nodo; fuente, destino son estado; i es nat
fvar
     {inicialmente se crea un nodo para el estado inicial, se encola un apuntador a él, se marca
      el estado inicial como tratado y se inicializa la variable de control}
    p := obtener_espacio
    si p = NULO entonces error
    si no
        p^ := <ESTADO.inicial, -1, NULO>; C := encola(COLA.crea, p)
        S := Ø∪{ESTADO.inicial}; encontrado := falso
        mientras ¬ encontrado ∧ ¬ COLA.vacía?(C) hacer
                {se obtiene el siguiente en amplitud}
            p := cabeza(C); C := desencola(C)
                {a continuación se estudian las seis transiciones posibles}
            fuente := p^.e; i := 1
            mientras (i ≤ 6) ∧ ¬ encontrado hacer
                si posible?(fuente, i) entonces  {si no, no es necesario estudiar la transición}
                    destino := aplica(fuente, i)     {se aplica la transición}
                    si destino = ESTADO.final entonces  {se enraiza el nodo final}
                        encontrado := cierto; q := obtener_espacio
                        si q = NULO entonces error si no q^ := <destino, i, p> fsi
                    si no {si el estado no se había generado, se enraiza, se encola y se marca}
                        si ¬ (destíno ∈ S) entonces
                            q := obtener_espacio
                            si q = NULO entonces error
                            si no q^ := <destino, i, p>; S := S∪{destino}; C := encola(C, q)
                            fsi
                        fsi
                    fsi
                fsi
                i := i + 1
            fmientras
        fmientras
    fsi
    si encontrado entonces l := recupera_camino(q) si no l := LISTA_INTERÉS.crea fsi
devuelve l

{Función auxiliar recupera_camino: dado un nodo que representa una hoja dentro de un árbol
 implementado con apuntadores al padre, sube hasta la raíz y guarda las transiciones en la lista
 resultado, en el orden adecuado}
función privada recupera_camino (p es ^nodo) devuelve lista_nat es
var l es lista_nat fvar
    l := LISTA_INTERÉS.crea
    mientras p^.padre ≠ NULO hacer l := inserta(principio(l), p^.transición); p := p^.padre fmientras
devuelve l
funiverso
```

Fig. 7.19: implementación eficiente del algoritmo de planificación (cont.).

7.2 Diseño de nuevos tipos abstractos de datos

Durante el desarrollo de una aplicación es habitual que surja la necesidad de introducir tipos abstractos de datos que no respondan al comportamiento de ninguna de las familias vistas en este texto. Puede ser que el tipo no sea más que un componente auxiliar en el programa, o que sea el tipo más importante alrededor del cual gira todo el diseño modular de la aplicación. En cualquier caso, la aparición de un nuevo tipo de datos obliga primero a escribir su especificación, es decir, a determinar las operaciones que forman la signatura (lo que no siempre es sencillo) y, a continuación, escribir sus ecuaciones. Al contrario que en la situación estudiada en la sección anterior, la escritura de la especificación en este contexto es factible y generalmente no demasiado costosa, por lo que es recomendable construirla. Así, se estudia el comportamiento del tipo y, si es necesario, puede experimentarse su validez prototipando la especificación con técnicas de reescritura que, por un lado, pueden mostrar posibles errores y, por el otro, pueden llevar el diseñador a replantearse la funcionalidad del tipo. Además, la especificación actúa como medio de comunicación entre diferentes diseñadores y también entre el diseñador y el implementador.

Una vez especificado el tipo, la implementación se puede construir según los dos enfoques tradicionales: o bien combinando diversos TAD residentes en la biblioteca de componentes para obtener la funcionalidad requerida, o bien representando directamente el tipo para responder a las posibles restricciones de eficiencia de la aplicación. En esta sección estudiamos tres ejemplos que se mueven en este espectro: el primero, una tabla de símbolos para un lenguaje de bloques implementada modularmente; el segundo, una cola compartida implementada directamente; el tercero, un almacén de información voluminosa con varias operaciones de modificación y consulta, resuelto con los dos enfoques y en el que se propone un método que intenta conjugar la modularidad y la eficiencia, sacrificando cierto rigor matemático.

7.2.1 Una tabla de símbolos

Se quiere construir una implementación para las tablas de símbolos usadas por los compiladores de lenguajes de bloques y estudiadas en el apartado 1.5.2. Recordemos que estas tablas se caracterizan por que, cuando el compilador entra en un nuevo bloque B, todos los objetos que se declaran en él esconden las apariciones de objetos con el mismo nombre que aparecen en los bloques que envuelvan a B, y cuando el compilador sale del bloque, todos los objetos que estaban escondidos vuelven a ser visibles. Por otro lado, un bloque tiene acceso exclusivamente a los objetos declarados en los bloques que lo envuelven (incluyendo él mismo) que no estén escondidos.

La especificación del tipo aparece en el apartado 1.5.2, y en la fig. 7.20 se resume la signatura. Por motivos de reusabilidad, se parametriza por las características de los objetos

almacenados en la tabla, encapsuladas en un universo ELEM_ESP_=, y se renombran algunos símbolos para una mejor comprensibilidad del código.

```
universo TABLA_SÍMBOLOS(ELEM_ESP_=) es
    usa CADENA, BOOL
    renombra elem por caracts, esp por indef
    tipo ts
    ops crea: → ts
        entra, sal: ts → ts
        declara: ts cadena caracts → ts
        consulta: ts cadena → caracts
        declarado?: ts cadena → bool
funiverso
```

Fig. 7.20: signatura de la tabla de símbolos.

Estudiemos la implementación del tipo. Básicamente, hay dos hechos fundamentales: por un lado, las dos operaciones consultoras necesitan acceder rápidamente a la tabla a través de un identificador; por el otro, los bloques son gestionados por el compilador con política LIFO, es decir, cuando se sale de un bloque, siempre se sale del último al que se entró. A partir de aquí, se puede considerar la tabla de símbolos como una pila de pequeñas tablas, una por bloque; cada una de estas subtablas contiene los identificadores que se han declarado en el bloque respectivo. Evidentemente, en cada momento sólo existen las subtablas correspondientes a los bloques que se están examinando. Las operaciones son inmediatas y el único punto a discutir es cómo se implementan las subtablas.

Está claro que las subtablas han de ser precisamente objetos de tipo tabla, dadas las operaciones definidas. Ahora bien, ¿se implementa la tabla por dispersión? Si consideramos el entorno de uso de la tabla de símbolos, parece difícil que en un bloque se declaren más de 10 ó 20 objetos[6]. En ese caso, no tiene sentido implementar la tabla por dispersión, porque la ganancía sería mínima y, en cambio, deberíamos escoger una función de dispersión que, además, podría comportarse mal y empeorar el tiempo de acceso (todo esto, sin mencionar el espacio adicional requerido por las organizaciones de dispersión). Por ello, elegimos implementar la tabla usando una lista representada mediante punteros.

El resultado se presenta en la fig. 7.21, donde se puede comprobar la fácil codificación de las diferentes operaciones del tipo, gracias a la modularidad de la solución. El invariante de la representación es cierto, porque no existe ninguna restricción sobre la representación. Se

[6] Recordemos que el concepto de bloque que presentan estos lenguajes se refiere a subprograma o trozo de código; si un módulo también pudiera desempeñar el papel de bloque, los objetos declarados en él podrían ser más y debería reconsiderarse las decisiones que se toman aquí.

supone que existe un universo en la biblioteca de componentes que implementa las listas ordenadas por punteros. El coste resultante de las diversas operaciones del tipo es constante, excepto consulta que, en el caso peor, ha de examinar toda la pila. Ahora bien, lo normal es que haya pocos bloques anidados en un programa y por ello la ineficiencia no es importante.

```
universo TABLA_SÍMBOLOS_POR_PILAS(ELEM_ESP_=)
    implementa TABLA_SÍMBOLOS(ELEM_ESP_=)
    renombra elem por caracts, esp por indef
    usa CADENA, BOOL
    instancia TABLA(ELEM_=, ELEM_ESP)
        implementada con LISTA_INTERÉS_POR_PUNTEROS(A es ELEM_=, B es ELEM_ESP)
        donde A.elem es cadena, A.= es CADENA.=,
              B.elem es caracts, B.esp es indef
    instancia PILA(ELEM) implementada con PILA_POR_PUNTEROS(ELEM) donde elem es tabla
    tipo ts es pila ftipo
    invariante (t es ts): cierto
    función crea devuelve ts es
    devuelve PILA.empila(PILA.crea, TABLA.crea)   {abre el bloque principal}
    función entra (t es ts) devuelve ts es
    devuelve PILA.empila(t, TABLA.crea)   {abre un nuevo bloque}
    función sal (t es ts) devuelve ts es
    devuelve PILA.desempila(PILA.crea)   {cierra el bloque actual}
    función declara (t es ts; id es cadena; c es caracts) devuelve ts es
        si PILA.consulta(PILA.cima(t), id) ≠ indef
            entonces error {el identificador ya está declarado}
            si no t := PILA.empila(PILA.desempila(t), TABLA.asigna(PILA.cima(t), id, c))
        fsi
    devuelve t
    función declarado? (t es ts; id es cadena) devuelve bool es
    devuelve TABLA.consulta(PILA.cima(t), id) ≠ indef
    función consulta (t es ts; id es cadena) devuelve caracts es
    var v es caracts fsi
        v := indef
        mientras ¬ PILA.vacía?(t) ∧ v = indef hacer
            v := TABLA.consulta(PILA.cima(t), id); t := PILA.desempila(t)
        fmientras
    devuelve v
funiverso
```

Fig. 7.21: implementación del tipo de la tabla de símbolos.

7.2.2 Una cola compartida

Se dispone de una cola de elementos que debe ser compartida por n procesos en un entorno concurrente. Un proceso se representa mediante un natural de 1 a n. La característica principal de esta cola es que cada proceso puede tener acceso a un elemento diferente, según la secuencia de operaciones que ha efectuado sobre ella. Las operaciones son:

crea: $\rightarrow$ cola, cola sin elementos; los n procesos no tienen acceso a ningún elemento
inserta: cola elem $\rightarrow$ cola; un proceso cualquiera (no importa cuál) deposita un valor al final de la cola. Este elemento pasa a ser el elemento accesible para todos aquellos procesos que no tenían acceso a ninguno
primero: cola nat $\rightarrow$ elem, devuelve el valor de la cola accesible por un proceso dado; da error si el proceso no tiene acceso a ningún elemento
avanza: cola nat $\rightarrow$ cola; el valor accesible para un proceso determinado es ahora el siguiente de la cola; da error si el proceso no tiene acceso a ningún elemento
borra: cola nat $\rightarrow$ cola, elimina, para todos los procesos, el valor accesible por un proceso determinado; todos aquellos procesos que tienen este elemento como actual pasan a tener acceso al siguiente dentro de la cola, si es que hay alguno; da error si el proceso no tiene acceso a ningún elemento

Así pues, los diferentes procesos pueden acceder a diferentes valores, según lo que haya avanzado cada uno. Por ejemplo, si se forma una cola q con inserta(inserta(crea, a), b), todo proceso tiene acceso inicialmente al valor a. Si un proceso p1 ejecuta la operación de avanzar sobre q, el resultado será una cola donde el proceso p1 tenga acceso al elemento b y cualquier otro proceso tenga acceso al elemento a. Si ese mismo proceso p1 ejecuta la operación de borrar, se obtiene una cola con un único valor a, donde p1 no tiene acceso a ningún elemento y el resto de procesos lo tienen al elemento a. Si a continuación se inserta un elemento c, el proceso p1 pasa a tener acceso a él.

En la fig. 7.22 se presenta una especificación para el tipo que merece ser comentada. En vez de aplicar directamente el método general, que da como resultado un conjunto de constructoras generadoras con muchas impurezas, se consideran las colas como funciones que asocian una lista de elementos a los naturales de 1 a n; las n listas asignadas tienen los mismos elementos, pero el punto de interés puede estar en sitios diferentes. Las instancias siguen esta estrategia. Primero se definen las listas de elementos a partir de la especificación residente en un universo ampliado, que ofrece algunas operaciones aparte de las típicas: la longitud, la inserción por el final, la supresión del elemento i-ésimo y la parte izquierda de una lista. Este universo puede o no residir en la biblioteca de componentes; incluso el mismo universo básico de definición de las listas puede incluir éstas y otras operaciones, si los bibliotecarios lo consideran oportuno y, si no existe, es necesario especificarlo e implementarlo. Después se definen las funciones, siendo la lista vacía el valor indefinido, y

se consignan como errores los accesos con índice no válido. Por lo que respecta a las ecuaciones, son simples en este modelo. Se introducen dos operaciones auxiliares para insertar y borrar en todas las listas. La operación de creación queda implícitamente definida como la creación de la tabla. No es necesario explicitar las condiciones de error de la cola compartida, porque pueden deducirse a partir de aquellas introducidas al instanciar la tabla y aquellas definidas en los mismos universos de las listas y las tablas. Por último, las operaciones sobre tablas que no han de ser visibles a los usuarios del tipo se esconden (las listas se instancian directamente como privadas).

```
universo COLA_COMPARTIDA(A es ELEM, V es VAL_NAT) es
    usa NAT, BOOL
    renombra V.val por n
    instancia privada LISTA_INTERÉS_ENRIQUECIDA(B es ELEM) donde B.elem es A.elem
    renombra lista por lista_elem
    instancia TABLA(B es ELEM_=, C es ELEM_ESP)
        donde B.elem es nat, B.= es NAT.=, C.elem es lista_elem, C.esp es crea
    errores [k = 0 ∨ k > n] ⇒ asigna(t, k, v), consulta(t, k, v), borra(t, k, v)
    renombra tabla por cola_compartida
    ops
        inserta: cola_compartida elem → cola_compartida
        primero: cola_compartida nat → elem
        avanza, borra: cola_compartida nat → cola_compartida
        privada inserta_todos: cola_compartida elem nat → cola_compartida
        privada borra_todos: cola_compartida nat nat → cola_compartida
    ecns
        primero(c, k) = actual(consulta(c, k))
        avanza(c, k) = asigna(c, k, avanza(consulta(c, k)))
        inserta(c, v) = inserta_todos(c, v, 1)
        borra(c, k) = borra_todos(c, long(parte_izquierda(l)), 1)
        inserta_todos(c, v, n) = asigna(c, n, inserta_por el_final(consulta(c, n), v))
        [k < n] ⇒ inserta_todos(c, v, k) =
                    inserta_todos(asigna(c, k, inserta_por el_final(consulta(c, k), v)), k+1)
        borra_todos(c, v, n) = asigna(c, n, borra_i_ésimo(consulta(c, n), v))
        [k < n] ⇒ borra_todos(c, v, k) =
                    borra_todos(asigna(c, k, borra_i_ésimo(consulta(c, k), v)), k+1)
    esconde TABLA.asigna, TABLA.consulta, TABLA.borra
funiverso
```

Fig. 7.22: especificación de la cola compartida.

Estudiemos la implementación del tipo. Una primera aproximación, funcionalmente válida, consiste en traducir directamente la especificación anterior a una implementación escrita en el estilo imperativo del lenguaje; de hecho, el universo de la fig. 7.22 se puede considerar más una implementación por ecuaciones, en la línea del apartado 2.2, que una especificación propiamente dicha porque, en realidad, se están simulando los valores del tipo de las colas compartidas con los valores del tipo de las tablas.

Como es habitual, esta implementación modular es muy ineficiente. Si se considera prioritario el criterio de la eficiencia, está claro que todos los elementos de la cola han de residir en una única estructura, tanto por cuestiones de espacio (para no duplicarlos, sobre todo si son voluminosos) y de tiempo (con una lista por proceso, es necesario insertar y borrar elementos en n listas); la estructura más indicada para organizar los elementos es una lista con punto de interés, dado que en cualquier momento debe ser posible acceder al medio. El único problema es que cada proceso ha de tener un punto de interés propio y éste no es el modelo conocido de las listas. Para reutilizar el modelo sin ningún cambio, es necesario definir un vector indexado por naturales de uno a n y, en cada posición, guardar el ordinal del elemento actual del proceso que representa (que será igual a la longitud más uno, si el proceso no tiene elemento actual).

En la fig. 7.23 se muestra la representación y la codificación de las diferentes operaciones con esta estrategia, suponiendo que las listas tienen una operación de longitud e implementándolas por punteros. La solución es todavía clara y modular; notemos, no obstante, que la mayoría de operaciones son de coste lineal sobre la lista y el vector.

El diseño de una estructura de datos realmente eficiente pasa ineludiblemente por tener acceso directo a los elementos de la lista, usando apuntadores que residan en una tabla de procesos, de manera que efectivamente haya n puntos de interés diferentes en la lista, todos ellos externos. Por tanto, para la gestión de la lista sólo existe un apuntador al primer elemento y un apuntador al último. Este esquema simplifica el coste de las operaciones primero y avanza; si, además, encadenamos todas las posiciones del vector que no tienen elemento actual, también inserta quedará constante.

```
universo COLA_COMPARTIDA_MODULAR (A es ELEM, V es VAL_NAT)
    implementa COLA_COMPARTIDA(A es ELEM, V es VAL_NAT)
    renombra V.val por n
    instancia LISTA_INTERÉS(B es ELEM)
        implementada con LISTA_INTERÉS_ENC_PUNT(ELEM)
        donde B.elem es A.elem
    tipo cola_compartida es tupla
                                procesos es vector [de 1 a n] de nat
                                cola es lista
                            ftupla
    ftipo
    invariante (c es cola_compartida): ∀k: 1 ≤ k ≤ n: 0 < procesos[k] ≤ long(cola)+1
    función crea devuelve cola_compartida es
    var k es nat; c es cola_compartida fvar
        para todo k desde 1 hasta n hacer c.procesos[k] := 1 fpara todo
        c.cola := LISTA_INTERÉS.crea
    devuelve c

    función inserta (c es cola_compartida; v es elem) devuelve cola_compartida es
    var k es nat fvar
        mientras ¬ final?(c.cola) hacer c.cola := avanzar(c.cola) fmientras
        c.cola := LISTA_INTERÉS.inserta(c.cola, v)
    devuelve c

    función primero (c es cola_compartida; k es nat) devuelve elem es
        c.cola := principio(c.cola)
        hacer c.procesos[k]-1 veces c.cola := avanzar(c.cola) fhacer
    devuelve actual(c.cola)  {provoca error si c.procesos[k] = long(cola)+1}

    función borra (c es cola_compartida; k es nat) devuelve cola_compartida es
    var i es nat fvar
        c.cola := principio(c.cola)
        hacer c.procesos[k]-1 veces c.cola := avanzar(c.cola) fhacer
        c.cola := borra(c.cola) {provoca error si c.procesos[k] = long(cola)+1}
            {actualización de los puntos de interés afectados por la supresión}
        para todo i desde 1 hasta n hacer
            si c.procesos[i] > c.procesos[k] entonces
                c.procesos[i] := c.procesos[i] - 1
            fsi
        fpara todo
    devuelve c

    función avanza (c es cola_compartida; k es nat) devuelve cola_compartida es
        si c.procesos[k] = long(cola)+1 entonces error {no hay elemento actual}
        si no c.procesos[k] := c.procesos[k] + 1
        fsi
    devuelve c
funiverso
```

Fig. 7.23: implementación modular de la cola compartida.

Sin embargo, la supresión es todavía ineficiente. Por un lado, si los procesos apuntan directamente al elemento actual, su supresión es lenta, porque para acceder al anterior es necesario recorrer la lista desde el inicio, o bien usar dobles encadenamientos; por ello, los procesos apuntarán al anterior al actual y se añadirá un elemento fantasma al inicio de la lista. Por otro lado, al borrar el elemento apuntado por un proceso es necesario actualizar todos los otros procesos que también apuntaban a él; por este motivo, todos los procesos que tienen el mismo elemento actual estarán encadenados. Es necesario, además, que este encadenamiento sea doble, porque cuando un proceso avance ha de borrarse de la lista que ocupa e insertarse en la lista del siguiente, que se convierte en el nuevo anterior al actual. Precisamente, para que esta segunda inserción también sea rápida, los elementos de la cola han de apuntar a un proceso cualquiera de la lista de los que lo tienen como anterior al actual.

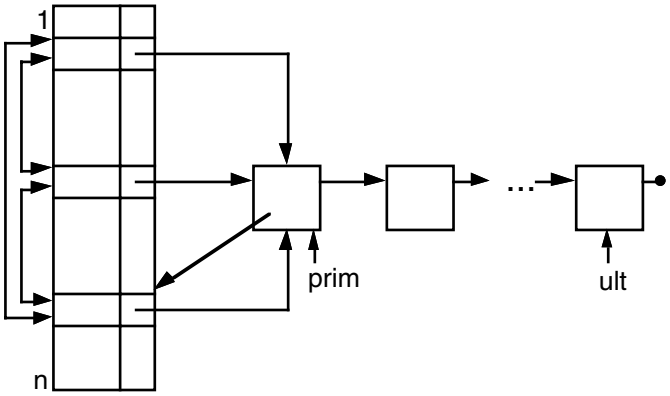


Fig. 7.24: esquema de una estructura de datos eficiente para la cola compartida.

En la fig. 7.25 se presenta la implementación del tipo, más complicada que la versión anterior (basta con ver el invariante, en el que aparecen diversas modalidades de la operación cadena según el encadenamiento que se siga, y que refleja la gran cantidad de relaciones que hay entre las diferentes partes de la estructura), porque la gestión de los diversos encadenamientos complica el código; como ya se ha dicho, éste es el precio a pagar por la eficiencia. Notemos que, efectivamente, la implementación es más eficiente que la anterior en cuanto al tiempo, porque sólo tiene una operación de coste no constante, la supresión (sin considerar la creación de la estructura, como es habitual). Incluso ésta, no obstante, es más eficiente que antes, porque sólo se accede a los procesos que realmente deben modificarse. Sin embargo, por lo que respecta al espacio, esta solución usa varios campos adicionales y resulta más ineficiente; si los elementos son numerosos y voluminosos, este espacio de más se puede considerar no significativo.

```
universo COLA_COMPARTIDA_EFICIENTE(A es ELEM, V es VAL_NAT)
    implementa COLA_COMPARTIDA(A es ELEM, V es VAL_NAT)
    renombra V.val por n
    tipo cola_compartida es
        tupla
            procesos es vector [de 1 a n] de tupla
                                                ant, seg son nat
                                                pelem son ^nodo
                                            ftupla
            cola es tupla prim, ult son ^nodo ftupla
        ftupla
    ftipo
    tipo privado nodo es
        tupla
            v es elem
            enc es ^nodo
            pproc es nat  {si vale -1, el elemento no está apuntado por ningún proceso}
        ftupla
    ftipo
    invariante (c es cola_compartida):
      c.cola.ult∈cadena(c.cola.prim) ∧ c.cola.ult^.enc = NULO  ∧
      ∀p: p∈cadena(c.cola.prim)-{NULO} ∧ p^.proc ≠ -1: (sea i = p^.proc)
                1 ≤ i ≤ n ∧ cadena_ant(c.procesos, i) = cadena_seg(c.procesos, i) ∧
                ∀k: k∈cadena_ant(c.procesos, i): c.procesos[k].pelem = p  ∧
     (∪p: p∈cadena(c.cola.prim)-{NULO} ∧ p^.proc ≠ -1: cadena_ant(c.procesos, p^.proc)) = [1, n]

    función crea devuelve cola_compartida es
    var c es cola_compartida fvar
        c.cola.prim := obtener_espacio {para el fantasma}
        si c.cola.prim = NULO entonces error
        si no     {se encadenan todos los procesos, que apuntan al fantasma}
            para todo k desde 1 hasta n hacer c.procesos[k] := <k+1, k-1, c.cola.prim> fpara todo
            c.procesos[1].ant := n; c.procesos[n].seg := 1
                {a continuación, se actualiza la cola}
            c.cola.prim^.enc = NULO; c.cola.prim^.pproc := 1; c.cola.ult := c.cola.prim
        fsi
    devuelve c

    función inserta (c es cola_compartida; v es elem) devuelve cola_compartida es
    var p es ^nodo fvar
        p := obtener_espacio {para el nuevo elemento}
        si p = NULO entonces error
        si no     {se inserta el nuevo elemento al final de la cola; implícitamente, todos los
                   procesos que no tenían actual pasan a tenerlo como actual}
            p^ := <v, NULO, -1>; c.cola.ult^.enc := p; c.cola.ult := p
        fsi
    devuelve c
```

Fig. 7.25: implementación eficiente de la cola compartida.

```
    función primero (c es cola_compartida; k es nat) devuelve elem es
    var v es elem fvar
        si c.procesos[k].pelem = c.cola.ult entonces error {no hay elemento actual}
        si no v := c.procesos[k].pelem^.enc^.v
        fsi
    devuelve v

    función avanza (c es cola_compartida; k es nat) devuelve cola_compartida es
    var ant, act son ^nodo; i es nat fvar
        ant := c.procesos[k].pelem
        si ant = c.cola.ult entonces error {no hay elemento actual}
        si no     {supresión en la lista doblemente encadenada en la que reside actualmente}
            act := ant^.enc
            si c.procesos[k].seg = k entonces {único proceso en la lista}
                ant^.pproc := -1
                si ant = c.cola.prim entonces {si era el primero, queda inaccesible y se borra}
                    liberar_espacio(ant); c.cola.prim := act {nuevo fantasma}
                fsi
            si no si ant^.pproc = k entonces ant^.proc := c.procesos[k].seg fsi
                  c.procesos[c.procesos[k].ant].seg := c.procesos[k].seg
                  c.procesos[c.procesos[k].seg].ant := c.procesos[k].ant
            fsi
                {inserción en la lista doblemente encadenada que le toca}
            i := act^.pproc
            si i = -1 entonces {se forma una nueva lista de un único elemento}
                  act^.pproc := k; c.procesos[k] := <i, i, act>
            si no c.procesos[k] := <c.procesos[i].ant, i, act>
                  c.procesos[c.procesos[i].ant].seg := k; c.procesos[i].ant := k
            fsi
        fsi
    devuelve c

    función borra (c es cola_compartida; k es nat) devuelve cola_compartida es
    var aux, i son nat; ant, act son ^nodo fvar
        ant := c.procesos[k].pelem    {anterior al elemento que se ha de borrar}
        si ant = c.cola.ult entonces error {no hay elemento actual}
        si no  i := ant^.enc^.pproc
               si i ≠ -1 entonces {hay elementos que han de pasar a apuntar a ant }
                     {se actualizan los apuntadores de los procesos afectados}
                  repetir
                     c.procesos[i].pelem := ant; i := c.procesos[i].seg
                  hasta que c.procesos[i].pelem = ant
                     {se concatenan las dos listas doblemente encadenadas}
                  aux := c.procesos[i].seg; c.procesos[c.procesos[k].ant].seg := aux
                  c.procesos[i].seg := k
                  c.procesos[aux].ant := c.procesos[k].ant; c.procesos[k].ant := i
               fsi
               act := ant^.enc; ant^.enc := act^.enc; liberar_espacio(act) {se borra de la lista}
        fsi
    devuelve c
funiverso
```

Fig. 7.25: implementación eficiente de la cola compartida (cont.).

7.2.3 Una emisora de televisión

Se quiere diseñar una estructura de datos para gestionar la compra y el uso de videoclips, películas, documentales, etc., que denominamos genéricamente ítems, en tu pantalla amiga, Tele Brinco. Las operaciones que se consideran son:

crea: $\rightarrow$ tele, emisora sin ítems
añade: tele ítems $\rightarrow$ tele, registra que la emisora compra un nuevo ítem; da error si ya existía algun ítem con el mismo nombre
cuál_toca: tele nat $\rightarrow$ ítem, proporciona el último ítem añadido a la estructura, que todavía no haya sido emitido y que tenga la duración dada (que supondremos expresada en minutos); si todos los ítems de esta duración han sido ya emitidos, selecciona el que hace más tiempo que se emitió; da error si no hay ningún ítem de la duración dada
emite: tele nat $\rightarrow$ tele, toma nota de que se ha emitido el ítem correspondiente de la duración dada; da error si no hay ningún ítem de la duración dada
da_todos: tele cadena $\rightarrow$ lista_ítem_y_nat, proporciona todos los ítems publicados por una compañía cinematográfica, musical, etc., por orden alfabético de su nombre y, además, da el número de veces que se ha emitido cada uno

El tipo ítem, que supondremos definido dentro del universo ÍTEM, presenta operaciones para preguntar el nombre, la compañía y la duración ; además, existe un elemento especial esp, que representa el ítem indefinido, de futura utilidad. La duración máxima de cualquier ítem se da como parámetro formal del universo.

En la fig. 7.26 se presenta la especificación del tipo, que introduce una constructora generadora auxiliar, emite_ítem, la cual simplifica el universo, pues es más sencillo especificar el resto de operaciones respecto a la emisión de un ítem dado, que a la emisión del ítem que toca por la duración dada. Las relaciones entre las tres constructoras generadoras consisten, como es habitual, en diversos errores y conmutatividades (es imposible que se dé la relación de error entre añade y emite_ítem, y se incluye sólo por aplicación estricta del método general). La política de emisión de ítems puede expresarse aplicando el método general y aprovechando la estructura misma de los términos; se introduce una operación privada para borrar un ítem dado (eliminando tanto sus emisiones como su añadido). Por lo que respecta al listado ordenado, se apoya íntegramente en tres operaciones: la obtención de la lista desordenada de los ítems de la compañía dada, la ordenación de una lista de cadenas (operación que suponemos existente en la biblioteca de componentes) y el cálculo del número de emisiones de los ítems. Es necesario instanciar los pares y las listas para declarar la signatura de la función da_todos; además, a causa de la existencia de la operación auxiliar que devuelve una lista de ítems, también se declara una instancia para este tipo.

universo TELE_BRINCO (MÁX es VAL_NAT) es
usa ÍTEM, CADENA, NAT
instancia PAR (A, B son ELEM) donde A.elem es ítem, B.elem es nat
renombra par por ítem_y_nat, _.c1 por _.v, _.c2 por _.n
instancia LISTA_INTERÉS(ELEM) donde elem es ítem_y_nat renombra lista por lista_ítem_y_nat
instancia LISTA_INTERÉS(ELEM) donde elem es ítem renombra lista por lista_ítem
tipo tele
ops crea: → tele
añade, privada emite_ítem: tele ítem → tele
cuál_toca: tele nat → ítem
emite: tele nat → tele
da_todos: tele cadena → lista_ítem_y_nat
privada está: tele ítem → bool {comprueba si el ítem existe}
privada ninguno_emitido: tele nat → bool {comprueba si se ha emitido algun ítem de la duración}
privada ninguno_sin_emitir: tele nat → bool {comprueba si hay ítems sin emitir de la duración}
privada borra_ítem: tele ítem → tele {elimina un ítem de la emisora}
privada lista_ítems: tele cadena → lista_ítem {da todos los ítems de una compañía}
privada n_emisiones: tele ítem → nat {cuenta el número de emisiones de un ítem}
privada cnt_emisiones: tele lista_ítem → lista_ítem_y_nat {añade n_emisiones a cada ítem}
errores [está(t, v)] ⇒ añade(t, v); [¬ está(t, v)] ⇒ emite_ítem(t, v); cuál_toca(crea, k)
ecns
[duración(v) ≠ duración(w)] ⇒ añade(añade(t, v), w) = añade(añade(t, w), v)
[duración(v) ≠ duración(w)] ⇒ añade(emite_ítem(t, v), w) = emite_ítem(añade(t, w), v)
[duración(v) ≠ duración(w)] ⇒ emite_ítem(emite_ítem(t, v), w) = emite_ítem(emite_ítem(t, w), v)
[duración(v) = k] ⇒ cuál_toca(añade(t, v), k) = v
[duración(v) ≠ k] ⇒ cuál_toca(añade(t, v), k) = cuál_toca(t, k)
[duración(v) ≠ k ∨ n_emisiones(t, v) > 0] ⇒ cuál_toca(emite_ítem(t, v), k) = cuál_toca(t, k)
[duración(v) = k ∧ n_emisiones(t, v) = 0 ∧ ¬(ninguno_sin_emitir(borra_ítem(t, v), k) ∧
ninguno_emitido(t, k))] ⇒ cuál_toca(emite_ítem(t, v), k) = cuál_toca(borra_ítem(t, v), k)
[duración(v) = k ∧ n_emisiones(t, v) = 0 ∧ ninguno_sin_emitir(borra_ítem(t, v), k) ∧
ninguno_emitido(t, k)] ⇒ cuál_toca(emite_ítem(t, v), k) = v
emite(t, k) = emite_ítem(t, cuál_toca(t, k))
{Especificación de las operaciones privadas para emitir}
está(crea, v) = falso; está(emite_ítem(t, v), w) = está(t, w)
está(añade(t, v), w) = está(t, w) ∨ (nombre(v) = nombre(w))
ninguno_emitido(crea, k) = cierto; ninguno_emitido(añade(t, v), k) = ninguno_emitido(t, k)
ninguno_emitido(emite_ítem(t, v), k) = ninguno_emitido(t, k) ∧ (k ≠ duración(v))
ninguno_sin_emitir(crea, k) = cierto
ninguno_sin_emitir(añade(t, v), k) = ninguno_sin_emitir(t, k) ∧ (k ≠ duración(v))
ninguno_sin_emitir(emite_ítem(t, v), k) = ninguno_sin_emitir(borra_ítem(t, v), k)
borra_ítem(crea, v) = crea
[nombre(v) = nombre(w)] ⇒ borra_ítem(añade(t, v), w) = t
[nombre(v) ≠ nombre(w)] ⇒ borra_ítem(añade(t, v), w) = añade(borra_ítem(t, w), v)
[nombre(v) = nombre(w)] ⇒ borra_ítem(emite_ítem(t, v), w) = borra_ítem(t, w)
[nombre(v) ≠ nombre(w)] ⇒ borra_ítem(emite_ítem(t, v), w) = emite_ítem(borra_ítem(t, w), v)

Fig. 7.26: especificación del tipo tele.

```
        {Especificación de da_todos y sus operaciones auxiliares}
    da_todos(t, c) = cnt_emisiones(t, ordena(lista_ítems(t, c)))
    lista_ítems(crea) = LISTA_INTERÉS.crea; lista_ítems(emite(t, k)) = lista_ítems(t)
    [c = compañía(v)] ⇒ lista_ítems(añade(t, v), c) = inserta(lista_ítems(t, c), v)
    [c ≠ compañía(v)] ⇒ lista_ítems(añade(t, v), c) = lista_ítems(t, c)
    cnt_emisiones(t, LISTA_INTERÉS.crea) = LISTA_INTERÉS.crea
    cnt_emisiones(t, inserta(l, v)) = inserta(cnt_emisiones(t, l), <v, n_emisiones(t, v)>)
    n_emisiones(crea, v) = 0; n_emisiones(añade(t, v), w) = n_emisiones(t, w)
    [nombre(v) = cual_toca(t, k)] ⇒ n_emisiones(emite(t, k), v) = n_emisiones(t, v) + 1
    [nombre(v) ≠ cual_toca(t, k)] ⇒ n_emisiones(emite(t, k), v) = n_emisiones(t, v)
funiverso
```

Fig. 7.26: especificación del tipo tele (cont.).

A continuación estudiamos el diseño y la implementación de la estructura de datos correspondiente. Supondremos que el contexto de uso de la aplicación determina, por un lado, que los ítems son objetos voluminosos y, por el otro, que da_todos puede ser de coste lineal respecto al número de ítems de una compañía, pero que el resto de operaciones, excepto la creación, deben ser de orden constante o, como mucho, logarítmico. Además, sabemos que el número de ítems y de compañías, si bien desconocidos, serán el primero de ellos muy elevado y el segundo muy pequeño.

Hay varios puntos importantes en el diseño de la estructura. Para empezar, las operaciones emite y cuál_toca, que no pueden ser lentas, piden acceso por duración y, por ello, se dispone un vector, indexado por naturales entre 0 y MÁX.val, donde se organizan los ítems de cada duración. En concreto, en cada posición hay dos estructuras, una pila para organizar los ítems todavía no emitidos y una cola para los ítems emitidos; así, es inmediato saber el ítem que toca emitir de una duración dada: si la pila tiene algún elemento, es la cima de la pila y, si no, la cabeza de la cola.

Por otro lado, la operación da_todos pide acceso por compañía; en consecuencia, se define una tabla que tiene como clave el nombre de la compañía y como información todos los ítems de esta compañía. Como la operación debe ser lineal, los ítems han de estar ya ordenados. Así pues, se definen como una tabla ordenada en la que la clave es el ítem y la información asociada es el número de veces que se ha emitido. Para cumplir los requisitos de eficiencia exigidos, la tabla correspondiente a cada compañía se implementa con un árbol de búsqueda (concretamente, un árbol AVL para evitar degeneraciones en la estructura) y, así, las inserciones son logarítmicas; además, no es necesario fijar un número máximo de ítems por compañía. En cambio, la tabla de todas las compañías se puede implementar con una simple lista ordenada, dado que el enunciado dice explícitamente que hay pocas.

Con estas decisiones, la eficiencia temporal de las operaciones es óptima: la inserción de un

ítem es logarítmica (inserción en una pila, una cola y un árbol AVL), la consulta de cuál toca es constante (accesos a la cima de una pila y a la cabeza de una cola), la emisión de un ítem también es logarítmica (desempilar, desencolar y encolar un elemento, y consultar y modificar un árbol AVL, debido al cambio del número de emisiones) y el listado ordenado es lineal (recorrido de un árbol AVL). Ahora bien, el coste espacial no es suficientemente bueno porque los ítems, que son voluminosos, se almacenan dos veces, una en la estructura lineal y otra en el árbol, buscando acceso directo a la información, lo cual es necesario para no penalizar ninguna operación. Para mejorar este esquema, introducimos un conjunto donde depositaremos todos los ítems, y en las estructuras lineales y los árboles guardaremos los nombres de los ítems y no los ítems mismos; para no perder eficiencia en las operaciones, es necesario implementar el conjunto por dispersión. Con esta decisión, parece lógico incluir el número de emisiones dentro del conjunto de ítems, y convertir las tablas ordenadas correspondientes a las compañías en conjuntos ordenados.

En la fig. 7.27 se muestra el universo resultante de aplicar las decisiones de diseño dadas. El número aproximado de ítems se define como parámetro formal para poder dimensionar la estructura de dispersión. El universo ELEM_CJT_DISP añade a ELEM_CJT la función de dispersión y la dimensión de la tabla. En la primera parte de la figura se efectúan todas las instancias necesarias para implementar el esquema. Destaca la instancia de la tabla de dispersión para los ítems, que precisa dos pasos: el primero para definir la función (se elige una suma ponderada con división) y el segundo para definir la tabla (en este caso indirecta), que tiene la función recién definida como parámetro. Todas las estructuras se implementan por punteros, a causa del desconocimiento de las dimensiones concretas de las estructuras. El universo UTILIDADES_DISPERSIÓN es un módulo que introduce diversas operaciones de interés general para las diferentes funciones y organizaciones de dispersión presentadas en el texto; así, la función natural_más_próximo devuelve el número más próximo a un natural determinado que cumpla las propiedades necesarias para ser un buen divisor en la función de dispersión de la división.

Notemos que la resolución no fija las implementaciones de los tipos de los pares ni de las listas. Para los pares, se puede hacer el mismo razonamiento que en el caso del planificador; por lo que respecta a las listas, se deja la responsabilidad de decidir al usuario del tipo, quien tendrá la información necesaria para elegir la representación más adecuada.

Es preciso hacer algunos comentarios sobre el universo resultante. Por un lado, su dimensión es intrínseca a la complejidad del problema y no se puede reducir. Como contrapartida, la resolución es simple y altamente fiable; la mejor prueba es la claridad del invariante, debido a que la complejidad de la implementación queda diluida en los diferentes componentes, que cumplirán sus propios invariantes. También queda muy facilitada la implementación de las diversas operaciones, que se concreta en la combinación de las operaciones adecuadas sobre los tipos componentes.

```
universo TELE_BRINCO_MODULAR (MÁX, NÍTEMS son VAL_NAT)
    implementa TELE_BRINCO(MÁX es VAL_NAT)
    usa ÍTEM, UTILITADES_DISPERSIÓN, CADENA, NAT, BOOL
            {definición de la estructura indexada por la duración}
    instancia PILA(ELEM) implementada con PILA_PUNTEROS(ELEM) donde elem es cadena
      renombra pila por inéditos
    instancia COLA(ELEM) implementada con COLA_PUNTEROS(ELEM) donde elem es cadena
      renombra cola por emitidos
    tipo privado duraciones es
              vector [de 0 a MÁX.val] de tupla E es emitidos; I es inéditos ftupla
    ftipo
            {definición de la estructura indexada por la compañía}
    instancia CONJUNTO_ORDENADO(ELEM_<_=)
      implementado con CONJUNTO_POR_ÁRBOL_AVL_POR_PUNTEROS(ELEM_<_=)
      donde elem es cadena, < es CADENA.<, = es CADENA.=
      renombra tabla por compañía
    instancia TABLA(ELEM_=, ELEM_ESP)
      implementada con LISTA_ORD_POR_PUNTEROS(B es ELEM_<_=, C es ELEM_ESP)
      donde B.elem es cadena, B.< es CADENA.<, B.= es CADENA.=
            C.elem es compañía, C.esp es CONJUNTO_ORDENADO.crea
      renombra tabla por compañías
            {definición del almacen de ítems}
    instancia SUMA_POND_Y_DIV (B es ELEM_DISP_CONV, C es VAL_NAT)
      donde B.elem es cadena, B.ítem es carácter, B.= es CADENA.=
            B.base es 26, B.i_ésimo es CADENA.i_ésimo, B.nítems es CADENA.long
            C.val es natural_más_próximo(NÍTEMS.val)
    instancia CONJUNTO(ELEM_CJT)
      implementado con TABLA_INDIRECTA(ELEM_CJT_DISP)
      donde clave es cadena, elem es tupla v es ítem_tele; nemisiones es nat ftupla
             id es nombre(_.v), = es CADENA.=, esp es <ÍTEM.esp, 0>
             val es natural_más_próximo(NÍTEMS.val), h es suma_pond_y_div
      renombra tabla por hemeroteca
            {definición del tipo}
      tipo tele es
          tupla H es hemeroteca; C es compañías; D es duraciones ftupla
      ftipo
      invariante (t es tele):
          ∀v: v ∈ítem: está?(t.H, nombre(v)) ⇔
                        v∈t.D[duración(v)].E ∨ v∈t.D[duración(v)].I ⇔
                        está?(consulta(t.C, compañía(v)), nombre(v)) ∧
          ∀k: 0 ≤ k ≤ MÁX.val: ∀x: x∈t.D[k].I: consulta(t.H, x).nemisiones = 0 ∧
                                ∀x: x∈t.D[k].E: consulta(t.H, x).nemisiones > 0 ∧
          ∀c: c∈cadena ∧ está?(t.C, c): ∀x: x∈cadena ∧ está?(consulta(t.C, c), x):
                                          compañía(consulta(t.H, x).v) = c
```

Fig. 7.27: implementación modular de la estructura de ítems.

```
    función crea devuelve tele es
    var t es tele; k es nat fvar
        t.C := TABLA.crea; t.H := CONJUNTO.crea
        para todo k desde 0 hasta MAX.val hacer t.D[k] := <COLA.crea, PILA.crea> fpara todo
    devuelve t

    función añade (t es tele; v es ítem_tele) devuelve tele es
        si está?(t.H, nombre(v)) entonces error {el ítem ya existe}
        si no     {primero se añade el ítem a la hemeroteca}
            t.H := añade(t.H, <v, 0>)
                {a continuación se empila como inédito en los ítems de su duración}
            t.D[duraciones(v)].inéditos := empila(t.D[duraciones(v)].inéditos, nombre(v))
                {por último, se añade a los ítems de la compañía}
            t.C := asigna(t.C, compañía(v), añade(consulta(t.C, compañía(v)), nombre(v)))
        fsi
    devuelve t

    función cuál_toca (t es tele; k es nat) devuelve ítem es
    var nombre es cadena fvar
        si PILA.vacía?(t.D[k].inéditos) entonces
            si COLA.vacía?(t.D[k].emitidos) entonces error {no hay ítems de la duración}
            si no nombre := cabeza(t.D[k].emitidos)   {el emitido hace más tiempo}
            fsi
        si no nombre := cima(t.D[k].inéditos)   {el último añadido}
        fsi
    devuelve consulta(t.H, nombre).v

    función emite (t es tele; k es nat) devuelve tele es
    var nombre es cadena; v es ítem; n es nat  fvar
            {primero se actualiza la estructura por duraciones}
        si PILA.vacía?(t.D[k].inéditos) entonces
            si COLA.vacía?(t.D[k].emitidos) entonces error {no hay ítems de la duración}
            si no nombre := cabeza(t.D[k].emitidos); t.D[k].emitidos := desencola(t.D[k].emitidos)
            fsi
        si no nombre := cima(t.D[k].inéditos); t.D[k].inéditos := desempila(t.D[k].inéditos)
        fsi
        t.D[k].emitidos := encola(t.D[k].emitidos, nombre)
            {a continuación se registra la emisión del ítem}
        <v, n> := consulta(t.H, nombre); t.H := asigna(t.H, <v, n+1>)
    devuelve t

    función da_todos (t es tele; nombre_comp es cadena) devuelve lista_ítems_y_nat es
          {se recorre la tabla ordenadamente y accediendo a la hemeroteca por el nombre del ítem}
        l := LISTA_INTERÉS.crea
        para todo nombre_ítem dentro de todos_ordenados(consulta(t.C, nombre_comp)) hacer
            l := inserta(l, consulta(t.H, nombre_ítem))
        fpara todo
    devuelve l
funiverso
```

Fig. 7.27: implementación modular de la estructura de ítems (cont.).

Esta solución puede ser lo bastante satisfactoria como para considerarla definitiva. Ahora bien, presenta dos inconvenientes que nos pueden conducir a la búsqueda de posibles mejoras. Primero, el nombre de un mismo ítem aparece en tres estructuras diferentes; si todos los ítems fueran tan cortos como "Hey!", este hecho no sería demasiado importante, pero la emisora tendrá seguramente ítems con títulos largos, como "The rise and fall of Ziggy Stardust and The Spiders from Mars". Segundo, la obtención del ítem a partir de su nombre es por dispersión, y la tabla puede degenerar si la previsión de ocupación falla o bien si la distribución de las claves almacenadas conlleva un mal resultado para la función de dispersión elegida, por lo que deben limitarse los accesos. Estas dos desventajas pueden solucionarse si se elimina el conjunto de ítems, almacenándose éstos en los árboles de las compañías y accediendo a ellos desde las pilas y las colas mediante apuntadores, en tiempo constante.

La estrategia citada presenta un problema evidente: los diversos tipos de datos utilizados no tienen operaciones que permitan acceder a sus elementos mediante un apuntador. Esto se debe al hecho de que el concepto de apuntador es propio del nivel de implementación, mientras que las operaciones se definen en la especificación algebraica, previa a la implementación. Si se quieren usar apuntadores es necesario, pues, representar todos los tipos auxiliares directamente en el universo que implementa el tipo de interés tele, tal como se ha hecho en el ejemplo de la cola compartida en el apartado anterior. La construcción de este universo queda como ejercicio para el lector. Es evidente que el resultado será mucho más voluminoso, complicado y propenso a errores que la propuesta anterior, por mucho que se puedan hacer algunas simplificaciones, entre las que destaca la posibilidad de fusionar todas las estructuras que almacenan los ítems en una sola, que tenga suficientes campos como para implementar las estructuras lineales y el árbol AVL. La organización por dispersión desaparece porque ya no es necesario acceder rápidamente a los ítems dado su nombre. Precisamente, esta fusión que optimiza al máximo la eficiencia de la implementación aún complica más el código, porque las operaciones de las estructuras lineales y del árbol AVL se entremezclan; el invariante mostrará claramente la dificultad de su gestión.

A continuación se formula una alternativa de diseño que intenta combinar la modularidad y la eficiencia de las dos opciones vistas. Se trata de suponer que toda implementación de un tipo abstracto de datos, además de las operaciones de la especificación y de todas aquellas introducidas en el apartado 3.3.4, presenta:

- Un tipo apuntador, que representa un medio de acceso a los elementos almacenados en la estructura.
- Una operación acceso que, dada la estructura E y un apuntador p, devuelve el elemento dentro de E apuntado por p.
- Diversas operaciones (las que convengan en cada caso) que devuelvan el apuntador al último elemento afectado por una operación.

En la fig. 7.28 se presenta el universo que implementa esta opción. Notemos la similitud con la solución de la fig. 7.27 por lo que respecta a la estructuración del tipo y la simplicidad de las operaciones. A causa de la desaparición de la tabla de dispersión, la codificación es incluso más corta, ya que los ítems se almacenan en los árboles AVL asociados a las compañías junto con el número de emisiones, y en las estructuras lineales residen apuntadores a los elementos de los árboles. Como ventaja adicional, la inexistencia de estructuras de dispersión comporta la desaparición del parámetro para el número esperado de ítems. El invariante se mantiene claro, lo que indica la simplicidad de la solución. De acuerdo con la estrategia mencionada, suponemos que los árboles de búsqueda implementados en el universo CONJUNTO_POR_ÁRBOL_AVL_POR_PUNTEROS presentan una operación nueva último_insertado, que devuelve un apuntador al último elemento insertado.

Notemos que no todas las implementaciones de un TAD encajan en este esquema, que exige que un elemento ocupe la misma posición mientras pertenezca a la estructura. Este problema, no obstante, se puede resolder en estructuras implementadas con vectores (por ejemplo, al implementar las tablas de dispersión de direccionamiento abierto y redispersión lineal con movimiento de elementos en la supresión), con el uso de tablas intermedias para traducir la posición obtenida por el usuario del tipo a la posición real que ocupa.

Pese al buen comportamiento de esta aproximación al diseño de las estructuras de datos, debe destacarse que las operaciones añadidas son una violación flagrante del principio del diseño de programas con TAD, dado que se crea una visión paralela del tipo que permite acceso a sus elementos sin respetar la política definida por las ecuaciones. Notemos que las nuevas operaciones no tienen nada que ver con las operaciones definidas en el apartado 3.3.4, que precisamente se introducían para abundar en la transparencia de la información; en este sentido, se puede decir que los universos pasan a actuar como simples plantillas de código desprovistas de auténtico significado matemático, con los problemas de razonamiento que ello conlleva[7].

Una solución de compromiso consiste en distinguir, para toda estrategia de implementación de un TAD, dos universos diferentes: la implementación habitual con claro significado matemático y la implementación que permite acceso mediante apuntadores. El diseñador de un nuevo tipo de datos usará los primeros siempre que pueda; cuando, por razones de eficiencia, necesite acceder a la estructura mediante apuntadores, usará los universos de la segunda clase. Para subrayar que este segundo tipo de uso de los universos de implementación no sigue la especificación dada, la instancia se realizará directamente de la implementación y no de la especificación (como ocurre con los conjuntos en fig. 7.28), dejando clara constancia que es una mera encapsulación de código y no de un tipo de datos.

[7] De hecho, se podrían construir especificaciones algebraicas para los TAD accesibles mediante apuntadores, siempre y cuando el concepto mismo de apuntador forme parte.

```
universo TELE_BRINCO_MODULAR_Y_EFICIENTE (MÁX es VAL_NAT)
    implementa TELE_BRINCO(MÁX es VAL_NAT)
    usa ÍTEM, CADENA, NAT, BOOL
    instancia CONJUNTO_POR_ÁRBOL_AVL_POR_PUNTEROS(ELEM_CJT_<)
        donde clave es cadena, elem es tupla v es item_tele; nemisiones es nat ftupla
                id es nombre(_.v), < es CADENA.<, = es CADENA.=, esp es <ÍTEM.esp, 0>
        renombra tabla por compañía, apuntador por @_ítem
    instancia TABLA(ELEM_=, ELEM_ESP)
        implementada con LISTA_ORD_POR_PUNTEROS(B es ELEM_<_=, C es ELEM_ESP)
        donde B.elem es cadena, < es CADENA.<, = es CADENA.=
                C.elem es compañía, C.esp es CONJUNTO_ORDENADO.crea
        renombra tabla por compañías
    tipo privado duraciones es
            vector [de 0 a MÁX.val] de tupla E es emitidos; I es inéditos ftupla
    ftipo
    instancia PILA(ELEM) implementada con PILA_PUNTEROS(ELEM) donde elem es @_ítem
        renombra pila por inéditos
    instancia COLA(ELEM) implementada con COLA_PUNTEROS(ELEM) donde elem es @_ítem
        renombra cola por emitidos
    tipo tele es tupla C es compañías; D es duraciones ftupla ftipo
    invariante (t es tele):
        ∀v: v∈ítem: p∈t.D[duración(v)].E ∨ p∈t.D[duración(v)].I ⇔
                    está?(consulta(t.C, compañía(acceso(p)), nombre(acceso(p))) ∧
        ∀k: 0 ≤ k ≤ MÁX.val: ∀p: p∈t.D[k].I: acceso(p).nemisiones = 0 ∧
                                ∀p: p∈t.D[k].E: acceso(p).nemisiones > 0 ∧
        ∀c: c∈cadena ∧ está?(t.C, c):
                ∀x: x∈cadena ∧ está?(consulta(t.C, c), x):
                        compañía(consulta(consulta(t.C, c), x).v) = c
    función crea devuelve tele es
    var t es tele; k es nat fvar
        t.C := TABLA.crea
        para todo k desde 0 hasta MÁX.val hacer t.D[k] := <COLA.crea, PILA.crea> fpara todo
    devuelve t
    función añade (t es tele; v es ítem_tele) devuelve tele es
    var p es @_ítem fvar
        si está?(t.H, nombre(v)) entonces error {el ítem ya existe}
        si no     {se añade a los ítems de la compañía con cero emisiones}
            t.C := asigna(t.C, compañía(v), añade(consulta(t.C, compañía(v)), <v, 0>))
                {a continuación se empila el apuntador como inédito en los ítems de su duración}
            p := último_añadido(consulta(t.C, compañía(v)))
            t.D[duraciones(v)].inéditos := empila(t.D[duraciones(v)].inéditos, p)
        fsi
    devuelve t
```

Fig. 7.28: implementación modular y eficiente de la estructura de ítems.

```
    función cuál_toca (t es tele; k es nat) devuelve ítem es
    var p es @_nodo fvar
        si PILA.vacía?(t.D[k].inéditos) entonces
            si COLA.vacía?(t.D[k].emitidos) entonces error {no hay ítems de la duración}
            si no p := cabeza(t.D[k].emitidos)   {el emitido hace más tiempo}
            fsi
        si no p := cima(t.D[k].inéditos)   {el último añadido}
        fsi
    devuelve acceso(p).v

    función emite (t es tele; k es nat) devuelve tele es
    var p es @_nodo; v es ítem; n es nat  fvar
            {primero, se actualiza la estructura por duraciones}
        si PILA.vacía?(t.D[k].inéditos) entonces
            si COLA.vacía?(t.D[k].emitidos) entonces error {no hay ítems de la duración}
            si no p := cabeza(t.D[k].emitidos); t.D[k].emitidos := desencola(t.D[k].emitidos)
            fsi
        si no p := cima(t.D[k].inéditos); t.D[k].inéditos := desempila(t.D[k].inéditos)
        fsi
        t.D[k].emitidos := encola(t.D[k].emitidos, p)
            {a continuación, se registra la emisión del ítem}
        <v, n> := acceso(p)
        t.C := asigna(t.C, compañía(v), añade(consulta(t.C, compañía(v)), <v, n+1>))
    devuelve t

    función da_todos (t es tele; nombre_comp es cadena) devuelve lista_ítems_y_nat es
            {como las emisiones residen en el mismo conjunto ordenado de la compañía, se puede
             devolver la lista que ha resultado al recorrerla ordenadamente}
    devuelve todos_ordenados(consulta(t.C, nombre_comp))
funiverso
```

Fig. 7.28: implementación modular y eficiente de la estructura de ítems (cont.).

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, estudiar la eficiencia espacial de la representación y la temporal de las operaciones, mencionando también el posible coste espacial auxiliar que puedan necesitar las operaciones, en caso de que no sea constante.

7.1 Sea el típico juego consistente en un rectángulo de 7x4 con 27 piezas cuadradas y, por tanto, con una posición vacía. Cada pieza tiene una letra diferente del alfabeto y, en caso de ser adyacente a la posición vacía, puede moverse horizontalmente o verticalmente, de tal manera que pase a ocupar la posición anteriormente vacía y deje como vacía la posición que

la pieza ocupaba antes del movimiento. Determinar los TAD adecuados que permitan diseñar un algoritmo que, dada una configuración inicial del tablero, construya la sucesión mínima de movimientos que se han de realizar para que el rectángulo quede ordenado alfabéticamente con la posición vacía en el extremo inferior derecho. Formular la propuesta de manera que sea fácilmente adaptable a cualquier problema similar. Una vez escrito el algoritmo, decir si los TAD resultantes se corresponden a tipos ya conocidos; en caso afirmativo, decir cuáles y cómo se implementarían y, en caso contrario, implementarlos.

7.2 Se quiere implementar una estructura de datos que sirva de base para un juego solitario de cartas de la baraja francesa. El objetivo de este juego consiste en distribuir todas las cartas en cuatro montones correspondientes a los cuatros palos, de manera que todas las cartas de un palo estén ordenadas en orden ascendente, a partir del as (que queda debajo) hasta el rey (que queda arriba); a estos montones los llamaremos destinatarios. Para alcanzar el objetivo, se dispone de seis montones auxiliares que contendrán cartas; a estos montones los llamaremos temporales y los identificaremos por un natural del uno al seis.

Inicialmente todos los destinatarios están vacíos y en el temporal i-ésimo depositamos i cartas, una sobre otra, todas boca abajo excepto la superior, que queda visible. El resto de cartas quedan en el mazo, todas boca abajo. A partir de este estado inicial, la evolución sigue las reglas siguientes:

- Una carta que se vuelve visible en el temporal, queda visible durante el resto del juego siempre que siga en algún temporal o sea la carta superior de un destinatario; en los destinatarios sólo queda visible la carta superior (que ya da suficiente información).
- Cuando un destinatario está vacío, la única carta que se puede mover sobre él es un as. A partir de este momento, las cartas entran en el destinatario correspondiente de una en una y boca arriba, manteniendo el palo y el orden ascendente consecutivo en la numeración. La carta superior siempre puede volver a un temporal si la estrategia del jugador lo requiere, y entonces queda visible la que había debajo.
- Las cartas visibles de un temporal se pueden mover según los criterios siguientes:
  - ◊ Siempre se puede mover la carta superior a un destinatario según la regla anterior.
  - ◊ La carta visible de un temporal a se puede mover encima de otro temporal b, siempre que sea de un palo de color diferente y que su numeración sea inmediatamente inferior a la de la carta visible del temporal b.
  - ◊ Se pueden mover de golpe todas las cartes visibles de un temporal sobre otro, siempre que la carta inferior de las movidas cumpla las mismas relaciones de color y numeración que en el punto anterior.

  Si aplicando cualquiera de los movimientos citados, un temporal queda sin cartas visibles, se puede destapar la carta superior (si hay alguna carta).
- En todo momento se puede pedir una carta del mazo. Esta carta se puede colocar, o bien en un destinatario, o bien en un temporal, siguiendo las relaciones de color y numeración de las reglas anteriores; además, si la carta es un rey se puede mover a un temporal vacío. Si no se ubica en ninguno de ellos, se pone boca arriba encima de otro

mazo auxiliar, que contiene todas las cartas que se han ido destapando del mazo y no se han ubicado. Notemos, pues, que en este auxiliar todas las cartas quedan boca arriba y sólo la última es accesible. En cualquier momento, si la estrategia del jugador lo requiere, se puede ubicar la carta visible del auxiliar sobre un destinatario o temporal según las reglas anteriores, y entonces quedaría visible la carta que había entrado en el auxiliar antes que ella.

- Si se acaban las cartas del mazo y el juego no ha terminado, se empieza otra ronda, que consiste en volcar el auxiliar, que asume el papel de mazo principal, y continuar jugando.
- Si durante una ronda entera no se ha ubicado ninguna carta del auxiliar a un destinatario o un auxiliar, el juego acaba con fracaso. El éxito se produce cuando los cuatro destinatarios contienen todas las cartas.

Implementar un algoritmo que sirva de basa para este juego, es decir, debe ser capaz de ejecutar las diversas órdenes que le dé el jugador, controlar que se respeten las reglas dadas, y avisar de la finalización del juego, con éxito o fracaso.

7.3 Una empresa diseñadora de muebles de cocina construye un programa que necesita un tipo pared para representar el conjunto de muebles situados en una pared de la cocina. Una pared se crea con una longitud positiva; un mueble se identifica por una anchura y una posición, que indica la distancia de su lateral izquierdo al extremo izquierdo de la pared (que puede considerarse el origen). La profundidad no se considera. La signatura del tipo es:

crea: nat $\rightarrow$ pared, crea una pared de una determinada longitud
añade: pared nat nat $\rightarrow$ pared, añade un mueble de una anchura determinada en una posición dada; da error si el mueble se sale de los límites
cabe?: pared nat nat $\rightarrow$ bool, dice si un mueble de una anchura dada puede colocarse en una posición dada
nre: pared $\rightarrow$ nat, da el número de muebles en la pared
elem: pared nat $\rightarrow$ mueble, dice cuál es el mueble i-ésimo de la pared, comenzando por la izquierda; da error si la pared no tiene i muebles
borra: pared nat $\rightarrow$ pared, elimina el mueble i-ésimo encontrado con la operación elem para la misma i; da error si la pared no tiene i muebles
mueve: pared nat $\rightarrow$ pared, desplaza el mueble i-ésimo (encontrado con la operación elem para la misma i) hacia la izquierda, hasta chocar con el mueble (i-1)-ésimo o el origen de la pared; da error si la pared no tiene i muebles

El tipo mueble se tratará como una instancia de los pares con dos naturales (posición y anchura). Especificar, diseñar e implementar el tipo pared, controlando todos los posibles errores.

7.4 Un estudiante aplicado de informática decide implementar una estructura para mantener sus innumerables programas, de manera que pueda hacer consultas eficientes. En concreto, la signatura del tipo progs es:

crea: → progs, estructura vacía
nuevo_programa: progs programa → prog, añade un nuevo programa; si ya hay uno con el mismo nombre, da error
borra: progs cadena → progs, borra un programa dado su nombre; si no lo encuentra, da error
purga: progs fecha → progs, borra todos los programas anteriores a una fecha dada
todos: progs → lista_programas, da la lista de todos los programas en orden alfabético
obsoletos: progs fecha → lista_programas, da la lista de programas anteriores a una fecha dada en cualquier orden
tema: progs cadena → lista_programas, da la lista de programas de un tema en cualquier orden; da error si el tema no existe

Suponer que el tipo fecha está disponible con las operaciones necesarias y que el tipo programa dispone de operaciones para obtener su nombre, su tema y su fecha; además, puede haber más información. Suponer que el repertorio de temas no es conocido a priori. Especificar el tipo y, a continuación, diseñarlo e implementarlo para que favorezca la ejecución de las operaciones consultoras y no ralentice purga innecesariamente (nuevo_programa y borra pueden ser tan lentas como se quiera).

7.5 En una galaxia muy, muy lejana, la cofradía de planetólogos quiere informatizar la gestión de la producción agrícola en su mundo. Hay diversas zonas climáticas que cubren las diversas regiones del planeta; en cada una de estas regiones el clima favorece el cultivo de diferentes productos. Por tanto, la cofradía necesita saber las relaciones existentes entre climas y regiones para poder planificar los diferentes cultivos buscando el máximo beneficio. Hay que tener en cuenta que el catálogo de climas es muy pequeño y que, si bien un clima se puede dar en diversas regiones, cada región sólo tiene asociado un tipo de clima. Un planeta puede tener un número muy elevado de regiones. Concretamente, las operaciones son:

crea: → meteorología, crea la estructura sin regiones
añadir: meteorología región clima → meteorología, añade una región a su clima; da error si el clima no existe; si la región ya tenía clima, lo sustituye (se considera un cambio climático)
planeta: meteorologia → lista_regiones, da la lista de todas las regiones del planeta en orden alfabético creciente
igual_clima: meteorologia clima → lista_regiones, da la lista de todas las regiones del planeta que tienen el clima dado en orden alfabético creciente, o error si el clima no existe

Especificar, diseñar e implementar el tipo meteorología. Es especialmente importante que la operación añadir sea eficiente, porque la cofradía quiere vender la aplicación a todos sus colegas de la galaxia, que tendrán que crear sus propios sistemas de zonas; además, las consultoras tampoco han de ser lentas, porque una vez configurado el sistema serán las más

usadas. Los tipos región y clima tienen mucha información y ofrecen operaciones para obtener su nombre, que será una cadena.

7.6 Se quiere informatizar el centro de control de una compañía ferroviaria que gestiona una gran cantidad de estaciones, vías y trenes y controla el movimiento de trenes de una estación a otra mediante unas determinadas órdenes de avance. Cuando el centro de control decide que un tren ha de viajar de una estación A de origen a una estación B de destino realiza la secuencia de operaciones siguiente:

plan_viaje: calcula la ruta más corta de A a B

avanzar: ordena al maquinista avanzar un tramo más en el plan de viaje.

ha_llegado: se ejecuta cuando el centro de control recibe confirmación de un jefe de estación que le comunica que un tren ha llegado.

El centro de control repite las dos últimas operaciones hasta que el tren llega a su estación de destino. Hay que tener en cuenta que las capacidades de las vías son limitadas, es decir, para cada tramo de vía que une dos estaciones sólo podrán circular, por razones de seguridad, un número máximo de trenes en cada sentido. Cuando un tren ha recibido la orden de avanzar por un tramo que está saturado en el sentido de la marcha, esperará en la estación donde se encuentre hasta que tenga vía libre. Se supone que en las estaciones pueden esperar un número ilimitado de trenes. Cada vez que un tren llega a una estación, si el tramo por el cual circulaba estaba saturado, el tren que haga más tiempo que esperaba para circular por ese tramo podrá ocuparlo. En concreto, la signatura del tipo es:

inic: $\rightarrow$ central, estructura sin estaciones, tramos ni trenes

añ_estación: central estación $\rightarrow$ central, añade una estación; da error si la estación ya existía

añ_vía: central estación estación nat nat nat $\rightarrow$ central, añade un tramo de vía entre dos estaciones con una capacidad máxima, que puede ser diferente en cada sentido, y una longitud; da error si alguna de las estaciones no existía o si el tramo ya existía

plan_viaje: central estación estación tren $\rightarrow$ central, planifica un viaje entre dos estaciones por el camino más corto con un tren determinado; da error si alguna estación no existe o si el tren está ya asignado

ha_llegado: central tren $\rightarrow$ central, registra que un tren que avanza por un tramo ha llegado a la siguiente estación dentro de su plan de viaje; da error si el tren no está a medio viaje

ocupaciones: central $\rightarrow$ lista_pares_estación_y_natural, devuelve una lista con todas las estaciones en orden decreciente según su ocupación máxima; la ocupación máxima de una estación se define como el número máximo de trenes que han tenido que esperar en algun momento en esta estación, por estar saturados los tramos

Las operaciones que lo necesiten tienen la posibilidad de llamar a la operación avanzar, que es una orden que la central da a un maquinista para que avance en su recorrido. Se suponen construidos los módulos correspondientes a los tipos estación y tren con las operaciones

necesarias, sabiendo que los primeros se identifican con una cadena de caracteres y los segundos con un natural. Notar que no hay operaciones explícitas de creación de trenes, sino que un tren es conocido la primera vez que interviene en un viaje y desaparece de la estructura cuando llega a su estación destino. Diseñar e implementar el tipo para que favorezca la ejecución de la operación ocupaciones sin que las otras sean innecesariamente lentas.

7.7 a) Se quiere informatizar la agencia de turismo Bon Voyage que organiza viajes a ciudades a un precio determinado, que depende de la compañía de transportes elegida. Las compañías se identifican por su nombre. Concretamente, las operaciones son:

crea: $\rightarrow$ agencia, forma una estructura sin compañías ni viajes ni ciudades
ofrece: agencia cadena compañía nat $\rightarrow$ agencia, registra que la agencia ofrece viajes a una ciudad a un precio dado y trabajando con la compañía dada; si ya existía alguna compañía con el mismo nombre, sus características no cambian; si ya existía algún viaje a la misma ciudad con la misma compañía, se olvida el más caro
ofertas_ciudad: agencia cadena $\rightarrow$ lista_pares_compañía_y_precio, devuelve todos los pares <compañía, precio> que representan viajes que la agencia organiza a la ciudad dada, en orden ascendente de precio; da error si la ciudad no existe
más_baratos: agencia precio $\rightarrow$ lista_pares_ciudad_y_compañía, devuelve todos los pares <ciudad, compañía> que representan viajes más baratos que el precio dado, en orden ascendente de precio

Suponer que el tipo compañía contiene mucha información. Se sabe, además, que habrá alrededor de 10.000 ciudades y pocas compañías de transporte. Especificar el tipo. Diseñarlo e implementarlo, de manera que se favoreza la ejecución de las consultoras, que las otras no sean más lentas de lo necesario y que no se malgaste espacio inútilmente.

b) La agencia ofrece ahora paquetes de viajes, compuestos de varias ciudades (no más de diez), que tienen una capacidad y un precio determinados, de manera que los clientes pueden reservar plazas. Concretamente, las nuevas operaciones son:

nuevo_paquete: agencia paquete $\rightarrow$ agencia, añade un nuevo paquete; da error si ya había algun paquete con el mismo nombre
reserva: agencia cadena nat $\rightarrow$ agencia; la agencia hace una reserva de las plazas especificadas en el paquete dado; da error si no hay ningun paquete con este nombre o no hay suficientes sitios libres
no_llenos: agencia $\rightarrow$ lista_paquetes, devuelve todos los paquetes que no se han llenado aún en orden ascendente de precio para hacer ofertas de final de temporada

Los paquetes se pueden considerar como 4-tuplas de <nombre, precio, capacidad, lista de ciudades>; toda esta información se puede obtener usando las operaciones nombre, precio, capacidad y ciudades. Se sabe que hay del orden de 200 paquetes. Especificar las nuevas operaciones. Ampliar el diseño anterior de manera que las operaciones reserva y

no_llenos sean las más eficientes y que nuevo_paquete no sea más lenta de lo necesario. No malgastar espacio inútilmente. Implementar el tipo resultante.

7.8 a) Se quiere informatizar las datos que hacen referencia a todos los jugadores de ajedrez federados del mundo (que son muchos, del orden de 1.000.000). De esta manera, se podrán organizar los torneos con más facilidad y también será más fácil actualizar los "elo" de los jugadores. El "elo" es una puntuación (entero positivo) que tiene todo jugador de ajedrez y que establece su posición respecto a los otros jugadores. Cada jugador tiene, además de su "elo", un nombre, una nacionalidad y un club. Los clubes son muy numerosos (el último censo registró 20.000), cada uno tiene un nombre y puede tener una o más nacionalidades. Los nombres y las nacionalidades son cadenas de caracteres sobre las que se tienen definidas las operaciones de comparación habituales.

```
universo JUG es
    usa CLUB, ...
    tipo jug
    ops
        jj: cadena nat cadena club → jug
        nombre?: jug → cadena
        elo?: jug → nat
        nacionalidad?: jug → cadena
        club?: jug → club
    ecns
        nombre?(jj(n, e, nac, c)) = n
        ... etc. ...
funiverso
```

```
universo CLUB es
    usa COLA_NACS, ...
    tipo club
    ops
        cc: cadena cola_nacs → club
        nombre?: club → cadena
        nacs?: club: → cola_nacs
    ecns
        nombre?(cc(n, c)) = n
        nacs?(cc(n, c)) = c
funiverso
{La parte cola_nacs del club es una cola
que contiene las nacionalidades del club}
```

El tipo de datos ajedrez tiene la signatura siguiente:

inic: → ajedrez, sin ningun jugador

añ_jugador: ajedrez jug → ajedrez, añade un jugador a la estructura; da error si ya había un jugador con el mismo nombre. Si ya existe un club con el mismo nombre que el club del jugador, no es necesario comprobar ni cambiar las nacionalidades

cambio_elo: ajedrez cadena nat → ajedrez, modifica el "elo" de un jugador; da error si el jugador no está

cambio_club: ajedrez cadena club → ajedrez, modifica el club del jugador; da error si el jugador no está. Si ya existe un club con el mismo nombre que el club dado, no es necesario comprobar ni cambiar las nacionalidades

elo?: ajedrez cadena → nat, da el "elo" de un jugador o error si el jugador no está

club?: ajedrez cadena → club, da el club de un jugador o error si el jugador no está

jug_club?: ajedrez cadena → cola_jugs, da los jugadores de un club ordenados por "elo" o error si el club no existe

clasificación_mundial?: ajedrez → cola_jugs, da los jugadores del mundo ordenados por "elo"
mejor_club?: ajedrez → club, da el club con mejor proporción: suma de "elo" de sus jugadores / número de jugadores

Nótese que no hay operaciones explícitas de alta de club, sino que un club se conoce cuando se usa por primera vez desde añ_jugador o cambio_club. Las colas se comportan de la manera esperada y ofrecen las operaciones normales; ahora bien, suponer que su implementación viene dada y es desconocida. Especificar el tipo. Diseñarlo e implementarlo para optimizar la ejecución de las consultoras y no malgastar espacio.

b) Modificar el diseño anterior de manera que se pueda obtener información por nacionalidades, que son pocas y casi fijas (muy raramente se añaden o eliminan nacionalidades). Concretamente, el nuevo diseño ha de posibilitar la implementación eficiente de las operaciones de consulta anteriores y, además, las siguientes:

clubs_nac: ajedrez cadena → cola_clubs, da todos los clubs que incluyen una nacionalidad en su cola de nacionalidades
jug_nac: ajedrez cadena → cola_jugs, da todos los jugadores de una nacionalidad ordenados por "elo"

7.9 El juego de la pirámide es iniciado por un grupo de diez jugadores privilegiados, que previamente acuerdan repartirse a partes iguales los más que probables beneficios. Para empezar, los jugadores elaboran una lista con sus nombres (y sus datos bancarios) y la venden al módico precio de 5.000 PTA. El último de esta lista busca tres (ingenuos) compradores a los que les vende la lista y les explica las reglas de funcionamiento:

"Has de enviar 5.000 PTA a la primera persona de la lista. A continuación, borra su nombre y pon el tuyo propio al final de la lista. Has de vender como mucho tres listas, y así recuperas tu inversión (5.000 PTA de compra más 5.000 PTA de envío) y, además, ganas 5.000 PTA. Es más, pasado un tiempo empezarás a recibir dinero que te enviarán los compradores de futuras listas y así ganarás millones de pesetas sin arriesgar nada."

Al contrario que otros juegos, el participante de la pirámide parece que gana siempre, sin que intervenga el azar. El juego, no obstante, acaba en el momento en que no se encuentran compradores para las listas; un artículo reciente del eminente investigador Dr. Keops ha establecido que el número esperado de jugadores de la pirámide se mueve en el intervalo [5.000, 9.000]. Cuando se llega a esta situación, los participantes en el juego se dividen en dos grandes categorías: los ganadores y los perdedores (la mayoría).

Consideramos el tipo pirámide con operaciones:

crear: lista_jugadores → pirámide, inicia el juego con los 10 primeros jugadores
vender: pirámide cadena cadena → pirámide, registra que el primer jugador vende la lista

a una persona que todavía no la había comprado. Los jugadores se identifican por su nombre, que es una cadena. La venta sólo es posible si el vendedor ha pagado al primero de la lista. En la venta se realiza la transacción monetaria correspondiente del comprador al vendedor

pagar: pirámide jugador → pirámide, registra que un jugador paga al primero de la lista; da error si el jugador no había comprado previamente la lista

ganadores: pirámide → lista_jugadores, da la lista de ganadores ordenada por ganancias

n_perdedores: pirámide → nat, da el número de personas que no han recuperado el dinero invertido

balance: pirámide cadena → entero, da la cantidad perdida o ganada por un participante en el juego o error si la persona no ha jugado

Especificar el tipo. A continuación diseñarlo e implementarlo de manera que se favorezcan las operaciones consultoras.

7.10 Los bancos más ricos de Europa deciden formar una asociación denominada ABBA (Asociación de Bancos Básicamente Alemanes) para competir con los japoneses y emires árabes. La ABBA es muy selectiva y sólo admite 5.000 asociados como mucho; una vez se ha llegado a esta cifra, si un banco pide el ingreso con un capital que supere el de alguno de los bancos de la asociación, entra en ella y se elimina el banco de capital más reducido de los que había. Para evitar esta situación, un banco siempre puede absorber otro (de menor capital) mediante una OPA hostil y así aumentar su capital; una absorción provoca la desaparición del banco absorbido.

El conde Mario, presidente de la ABBA, decide informatizar la asociación con las siguientes operaciones:

crea: → abba, asociación sin bancos

pide: abba cadena nat → abba, añade un banco con un capital dado en la asociación, si hay menos de 5000 bancos; en caso contrario, si el capital dado supera el capital de algún banco, también lo incorpora y borra de la asociación el banco de capital mínimo; si hay varios bancos igual de pobres, se elimina uno cualquiera de ellos; da error si el banco que se ha de añadir ya era socio de la ABBA

absorbe: abba cadena cadena → abba, registra que el primer banco absorbe al segundo y, en consecuencia, incrementa su capital; da error si alguno de los bancos no era socio de la ABBA

todos: abba → lista_cadenas, devuelve un listado de todos los bancos de la asociación en orden alfabético

más_rico: abba → cadena, da el banco de la asociación con más capital; si hay varios en esta situación, devuelve cualquiera de ellos

Suponer que las operaciones para comparar cadenas en orden lexicográfico son muy rápidas. Especificar el tipo. A continuación, diseñarlo e implementalo, favoreciendo las operaciones consultoras y evitando que las constructoras sean lineales.

7 . 11 La emisora Canal-Mus decide imitar a Tele-Brinco y quiere organizar sus vídeos en una estructura de datos. Sin embargo, los socios capitalistas del grupo BRISA deciden una política de emisión de vídeos algo diferente, que genera la siguiente signatura:

crea: $\rightarrow$ tele, emisora sin vídeos
añadir: tele vídeo $\rightarrow$ tele, registra que la emisora compra un nuevo vídeo; no es necesario comprobar repeticiones
cual_toca?: tele nat $\rightarrow$ vídeo; si hay algún vídeo con la duración dada que todavía no haya sido emitido, ha de devolver el más antiguo de ellos; si todos los vídeos de esta duración han sido emitidos, ha de devolver el que haga más tiempo que se emitió
emitir: tele nat $\rightarrow$ tele, toma nota de que el vídeo seleccionado por la operación anterior con la duración dada ya ha sido emitido
borra_malos: tele $\rightarrow$ tele, elimina los 10 vídeos que se han emitido menos veces
emitir_top_ten: tele nat $\rightarrow$ <tele, vídeo>, devuelve el vídeo que figura en la posición n-ésima en la clasificación de emisiones, n $\leq$ 10, y lo emite

Considerar que los vídeos sólo contienen información sobre su nombre y su duración, que viene dada en minutos (y es inferior a 4 horas). No se tiene ninguna idea sobre el número de vídeos que habrá en la estructura. Especificar el tipo. Diseñarlo e implementarlo para que todas las operaciones sean lo más rápidas posible, excepto emitir.

7.12 La emisora de televisión Antonia Tres ha decidido automatizar toda la gestión de la contratación y emisión de anuncios. Se sabe que, a lo largo de la semana, hay n períodos de emisión de anuncios (n < 1000). Cada período, identificado con un natural de 1 a n, tiene asignado un coste por minuto de anuncio y una duración (tiempo en minutos disponible para emitir anuncios). Se quieren realizar las siguientes operaciones, que permiten programar la distribución de anuncios a lo largo de la semana:

crear: $\rightarrow$ tele, inicializa la estructura, determinando cuáles son los períodos, de qué duración y su coste
comprar: tele nat anuncio $\rightarrow$ tele, registra la emisión de un anuncio dentro de un período; da error si el período no es válido o si no hay suficiente tiempo disponible en él
consultar: tele nat nat $\rightarrow$ lista_nat, devuelve una lista de todos los períodos que tienen un coste por minuto menor o igual al coste dado (segundo parámetro), y una duración disponible superior o igual a la duración dada (tercer parámetro), ordenada por el coste
períodos_de_emisión: tele anuncio $\rightarrow$ lista_nat, da la lista de períodos que se han comprado para ese anuncio
emitir: tele nat $\rightarrow$ lista_anuncios, da la lista de todos los anuncios que han comprado ese período o error si el natural no identifica un período válido

Los anuncios se identifican por su nombre; como no se pueden borrar de la estructura, habrá muchos anuncios a lo largo del tiempo. Especificar el tipo. Diseñarlo e implementarlo para que se favorezcan las operaciones consultoras.

7.13 En el año 2001, Europa se organiza federalmente respetando, sin embargo, las diversas comunidades históricas que en ella se ubican. Hay pocas federaciones (del orden de 20 ó 30: Atlántico Meridional, Iberia, las Islas Británicas, etc.) pero un número considerable de comunidades históricas (sobre unas 5.000, de diferente tamaño: Catalunya, Val d'Aran, Occitània, la Comunidad Celta, Lofoten, etc.); una comunidad histórica puede estar en varias (pero no muchas) federaciones a la vez (por ejemplo, la Comunidad Celta está a caballo entre Iberia, el Atlántico Meridional y las Islas Británicas), y ambas clases de objetos ofrecen una operación nombre que devuelve la cadena que los identifica. Cada comunidad histórica consta de un número indeterminado pero grande de municipios, cada uno de los cuales está gobernado por un único alcalde; considerar que un municipio no puede estar a caballo entre dos comunidades históricas. Los tipos federación, comunidad y alcalde ofrecen una operación nombre que devuelve la cadena que los identifica. El presidente de Europa, Narcís Guerra, decide informatizar la gestión administrativa para facilitar la tarea de los euro-funcionarios. Las operaciones que considera son:

crea: → europa; inicialmente, no hay ninguna federación ni comunidad
nueva_fede: europa federación lista_comunidades → europa, añade una nueva federación dentro de la cual coexisten diversas comunidades históricas; da error si ya existía alguna federación con el mismo nombre
elige: europa comunidad alcalde → europa, registra la elección de un nuevo alcalde en la comunidad
dimite: europa cadena → europa; un alcade identificado por su nombre presenta la dimisión por razones personales o políticas
colegas?: europa cadena cadena → bool, dice si dos alcaldes identificados por su nombre son cabezas de municipio en la misma comunidad
todos_fede: europa cadena → lista_alcaldes, devuelve la lista de alcaldes que gobiernan en municipios situados en las comunidades que forman la federación dada; da error si no hay ninguna federación con este nombre

Especificar el tipo. Diseñarlo e implementarlo para favorecer la ejecución de las operaciones consultoras y que, una vez alcanzado este objetivo prioritario, también favorezca dimite (se prevé que pueda haber muchas irregularidades financieras).

7.14 Sajerof es un informático reputado que ha hecho de la buena cocina su afición predilecta. Decidido a sacar provecho de su oficio, decide implementar un recetario de cocina que debe durarle toda la vida y que organiza las recetas de Arguiñano según su ingrediente principal y su precio. Las operaciones son:

crea: → recetario; sin recetas
añadir: recetario receta → recetario, añade una nueva receta; si ya hay alguna con el mismo nombre, da error
borrar: recetario cadena → recetario, borra la información asociada a una receta; si no hay ninguna con este nombre, da error

cocina: recetario cadena fecha → recetario, registra que la receta de un determinado nombre ha sido cocinada el día dado; da error si la receta no existe o si la fecha es anterior a la última en que se había cocinado la receta
más_barata: recetario nat fecha → receta, da una receta de precio menor o igual que el precio dado y que no se haya usado desde el día dado; da error si no hay ninguna que responda a esta descripción
capricho: recetario cadena → receta, da una receta cualquiera que tenga el ingrediente dado como principal; da error si no hay ninguna

El tipo receta, que es muy voluminoso, ofrece operaciones para obtener su nombre, su precio y su ingrediente principal. El número de recetas será de 2.000, aproximadamente. Especificar el tipo. Diseñarlo e implementarlo para que favorezca la ejecución de las operaciones más usades, capricho y más_barata (de momento, el estado económico de Sajerof es bastante delicado) sin que importe que las otras sean lentas; no obstante, cuidar el espacio.

7.15 Con el paso del tiempo, Sajerof empieza a perder la memoria y, ante los problemas existentes para recordar qué vino combina con qué plato, decide montar una estructura de vinos y platos que combinen entre sí. Las operaciones que necesita son las habituales:

crea: → platos_y_vinos; estructura vacía
combina: platos_y_vinos receta vino → platos_y_vinos, anota que un vino determinado acompaña bien a un plato dado
descombina: platos_y_vinos cadena cadena → platos_y_vinos, anota que un plato y un vino, identificados por su nombre, ya no combinan bien; esta operación es necesaria porque, como consecuencia de las modas, las relaciones cambian. Da error si el vino o el plato no existen o ya estaban relacionados
platos?: platos_y_vinos cadena → lista_recetas, da todas las recetas que combinan con un vino identificado por su nombre o da error si el vino no existe
vinos?: platos_y_vinos cadena → lista_vinos, da todos los vinos que combinan con una receta identificada por su nombre o da error si la receta no existe

El tipo receta es el mismo que en el ejercicio anterior; el tipo vino presenta una operación para obtener el nombre. En combina, suponer que siempre que se use un plato o un vino que ya está en la estructura, sus características no cambian. Especificar el tipo. Diseñarlo e implementarlo favoreciendo las operaciones consultoras, sabiendo que un vino puede combinar con muchos, muchos platos, pero que un plato combina con pocos vinos. El número de platos es conocido, 5.000, pero no así el de vinos.

7.16 Sajerof afronta un nuevo reto: sus numerosos convites le hacen temer la posibilidad de confeccionar un mismo plato más de una vez a algún amigo lo que, sin duda, provocaría comentarios del estilo ¿Qué pasa? ¿Es que sólo sabes cocinar este plato?, que herirían sus sentimientos. Por ello, quiere construir un nuevo tipo recetario de operaciones:

crea: → recetario: estructura vacía
añadir: recetario recepta → recetario, añade una nueva receta; si ya hay alguna con el mismo nombre, da error
comer: recetario cadena cadena fecha → recetario, anota que un amigo ha comido un plato (identificado por su nombre) un día dado; el amigo es simplemente una cadena. Da error si el plato no existía
qué_ha_comido?: recetario cadena → lista_recetas, devuelve todos los platos que ya ha comido un amigo dado
ha_comido?: recetario cadena cadena → <bool, fecha>, dice si un amigo dado ha comido un plato determinado y, en caso afirmativo, dice cuándo fue la última vez
lista: recetario → lista_recetas_y_amigos, lista en orden alfabético todas las recetas indicando para cada una qué amigos la han comido
qué_no_ha_cocinado?: recetario cadena cadena → lista_recetas, devuelve todas las recetas que tienen un ingrediente principal determinado y que un amigo dado no ha comido; da error si no hay ningún plato con ese ingrediente principal

Especificar el tipo. Diseñarlo e implementarlo para optimizar las operaciones consultoras, sabiendo que Sajerof tiene muchos amigos, del orden de 2.000 (es muy popular) y que recetas hay aproximadamente 20.000.

7.17 Sajerof ha contactado con una quesería y, muy hábilmente, les ha endosado la estructura platos_y_vinos. A causa del éxito fulminante del producto, que les ha hecho incrementar notablemente la clientela, la quesería decide ofrecer en su carta la posibilidad de que los clientes confeccionen tablas de quesos a su gusto, usando un terminal que tendrá disponible cada mesa del local. Un queso tendrá como información (consultable mediante operaciones sobre el tipo queso) su nombre, el origen, el tipo y el grado de sabor; el nombre es el identificador y, al igual que el origen y el tipo, es una cadena; la fortaleza es un natural. A tal efecto, Sajerof confecciona un programa que, aparte de las típicas operaciones de crear la estructura vacía y añadir y borrar quesos, ofrece tres operaciones consultoras: carta, que devuelve la lista de todos los quesos por orden alfabético de origen y, dentro de cada origen, por orden de tipo; lista1, que devuelve la lista de todos los quesos de un origen y tipo dados en orden decreciente de fortaleza; y lista2, que devuelve la lista de todos los quesos de un tipo dado y de un grado de fortaleza superior a un grado dado. Especificar, diseñar e implementar el tipo, optimizando las operaciones consultoras para que los clientes no esperen inútilmente; las operaciones constructoras pueden ser tan lentas como se quiera (dado que, gracias a la venta del programa anterior, Sajerof es rico y puede pagar una persona que entre los datos).

7.18 Como culminación de sus sueños gastronómicos, nuestro buen amigo Sajerof inaugura un restaurante selecto que ofrece una carta de delicatessen; una delicatessen se define como un par plato-vino y tiene un precio determinado (valor de tipo natural). Los platos son objetos voluminosos identificados con una cadena que se puede obtener con la

operación ident. Los vinos son tuplas de tres campos: una cadena que los identifica, una cadena que representa una región de procedencia y un natural que representa su calidad; toda esta información se puede obtener mediante tres operacions, ident, región y calidad. Las operaciones son:

crea: → carta, estructura vacía

ofrece: carta plato vino nat → carta, añade a la carta una delicatessen formada por un plato y un vino con un precio. Si la delicatessen ya existía, se actualiza el precio. Si no había ningún plato con el mismo nombre que el plato dado, se da de alta; lo mismo ocurre con el vino

retira: carta cadena → carta, elimina de la carta un vino de nombre dado; a los platos de todas las delicatessen que tenían este vino, se les asigna automáticamente el vino de calidad inmediatamente más baja que se tenga dentro de la misma región, pero sin cambiar su precio; en caso de que se intente repetir una delicatessen ya existente, no se forma la nueva. Da error si no hay ningún vino con este nombre o si es el más barato de su región

vinos: carta cadena → lista_vinos, devuelve la lista de vinos que se ofrecen en las delicatessen junto con el plato de nombre dado; da error si no hay ningún plato con este nombre

más_selecta: carta → vino, devuelve el vino que se usa en la delicatessen más cara del restaurante; si hay más de una, devuelve cualquier vino de los que intervienen. Da error si no hay ninguna delicatessen

Es necesario considerar varios hechos al diseñar la estructura, dadas las especialidades que Sajerof tiene previsto ofrecer: un plato interviene en pocas delicatessen (no más de seis) pero, en cambio, un vino puede intervenir en muchas, muchas delicatessen; por otro lado, se sabe que, después de una fase inicial totalmente despreciable, el número de platos será aproximadamente máx_platos y en ninguna caso habrá más de máx_platos, mientras que el número de delicatessen fluctuará mínimamente alrededor de máx_carta (valor que depende de la superficie y del número de hojas de la carta que ofrecerá el restaurante y que no puede ser superado), y el número de vinos no se sabe exactamente, pero habrá entre 2.000 y 10.000; por último, considerar que hay muy pocos vinos de una misma región (no más de seis) y que de regiones hay un máximo de máx_reg. Especificar el tipo. Diseñarlo e implementarlo, favoreciendo la ejecución de las operaciones retira, vinos y más_selecto sin que importe que ofrecer sea lenta.

7.19 Un buen colega de Sajerof, apodado "Andalejos" por su asistencia reiterada a congresos cuanto más lejanos mejor, entusiamado por las prestaciones de los programas gastronómicos del primero, decide informatizar su vasta colección de música barroca. A efectos prácticos, una obra de música barroca puede considerarse como una tupla <identificador, título, compositor, año de finalización de la composición>; el identificador es una cadena, mientras que el año será un entero perteneciente al intervalo [1.550, 1.850]. Dadas sus necesidades, las operaciones que Andalejos define son:

crea: → barroco, estructura vacía
compra: barroco obra → barroco, añade una nueva obra a la colección; si la obra ya existe, da error
pincha: barroco cadena → barroco, registra la audición de una obra determinada, identificada por la cadena correspondiente; da error si la obra no existe
por_autor: barroco cadena → lista_obras, devuelve la lista de todas las obras disponibles de un compositor determinado
menos_oídas: barroco → lista_obras, devuelve la lista de las diez obras menos pinchadas de la colección, para su eventual eliminación con
elimina: barroco cadena → barroco, elimina de la colección una obra determinada (por poco apetecible, porque se raya, etc.); da error si la obra no existe
más_apetecibles: barroco → lista_obras, devuelve la lista de las diez obras más pinchadas de la colección, con los que Andalejos se asegura buenos ratos en las largas noches de hotel durante los congresos, en ausencia de otros placeres más mundanos
por_época: barroco nat nat → lista_obras, devuelve la lista de todas las obras compuestas entre dos años determinados; da error si alguno de los años no pertenece al intervalo [1.550, 1.850]

Debe saberse que, gracias a su buen sueldo de catedrático, Andalejos posee una casa-chalé de grandes dimensiones en la sierra, donde almacena miles y miles de obras barrocas. El número de autores de música barroca es no tiene un máximo conocido ya que, al entender de Andalejos, la clasificación de una obra como "barroca" es hasta cierto punto arbitraria. Especificar el tipo. Diseñarlo e implementarlo, de manera que ninguna operación (excepto probablemente la creación) sea lineal sobre el número total de obras o de autores.